On Extracting Specialized Code Abilities from Large Language Models: A Feasibility Study

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

Chaozheng Wang
The Chinese University of Hong Kong
Hong Kong SAR, China
czwang23@cse.cuhk.edu.hk

Pingchuan Ma
The Hong Kong University of Science and Technology
Hong Kong SAR, China
pmaab@cse.ust.hk

Chaowei Liu
National University of Singapore
Singapore, Singapore
e1011116@u.nus.edu

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

Daoyuan Wu*
Nanyang Technological University
Singapore, Singapore
daoyuan.wu@ntu.edu.sg

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

Yang Liu
Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

ABSTRACT

Recent advances in large language models (LLMs) significantly boost their usage in software engineering. However, training a well-performing LLM demands a substantial workforce for data collection and annotation. Moreover, training datasets may be proprietary or partially open, and the process often requires a costly GPU cluster. The intellectual property value of commercial LLMs makes them attractive targets for imitation attacks, but creating an imitation model with comparable parameters still incurs high costs. This motivates us to explore a practical and novel direction: *slicing commercial black-box LLMs using medium-sized backbone models.*

In this paper, we explore the feasibility of launching imitation attacks on LLMs to extract their *specialized code abilities*, such as "code synthesis" and "code translation." We systematically investigate the effectiveness of launching code ability extraction attacks under different code-related tasks with multiple query schemes, including zero-shot, in-context, and Chain-of-Thought. We also design response checks to refine the outputs, leading to an effective imitation training process. Our results show promising outcomes, demonstrating that with a reasonable number of queries, attackers can train a medium-sized backbone model to replicate specialized code behaviors similar to the target LLMs. We summarize our findings and insights to help researchers better understand the threats posed by imitation attacks, including revealing a practical attack surface for generating adversarial code examples against LLMs.

CCS CONCEPTS

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

KEYWORDS

Large Language Models, Imitation Attacks

ACM Reference Format:
Zongjie Li, Chaozheng Wang, Pingchuan Ma, Chaowei Liu, Shuai Wang, Daoyuan Wu, Cuiyun Gao, and Yang Liu. 2024. On Extracting Specialized Code Abilities from Large Language Models: A Feasibility Study. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639091

1 INTRODUCTION

Recent advancements in the development of large language models (LLMs) have led to a significant increase in their usage in software engineering [79]. Major enterprises, such as OpenAI [12], have already deployed their LLM APIs to assist humans in writing code and documents more accurately and efficiently [90]. A large-scale code corpus with related natural language comments is used to build the models that improve the productivity of computer programming. The advanced LLMs are advocated to play a role as an "AI programming assist" that can handle various code-related tasks. These tasks include both interactive tasks, such as helping developers write automated scripts [19] or providing reviews [18, 55, 65], and complex tasks that require reasoning skills, such as finding vulnerabilities [88] and clone detection [47]. Moreover, some LLMs like ChatGPT [5] are designed with more interactive ability, which means they can learn from their interactions with humans and improve their performance over conversational turns, making them

*Corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE 2024, April 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/3597503.3639091

more effective at finding and fixing bugs that are difficult to reach with traditional tools [15, 80].

Despite the growing popularity of LLM APIs, it is widely acknowledged that training a well-performing LLM requires a plethora of workforce for data collection and annotation, as well as massive GPU resources [21, 26]. As a result, although some model architectures are publicly available, the model weights and training data are viewed as intellectual property (IP) that model owners own and have rights over. The IP behind a model is quite valuable, as model providers offer their services to a large number of users. Recently, it has been validated that computer vision (CV) and natural language processing (NLP) models are vulnerable to *imitation attacks* (or model extraction attacks) [35, 36, 67, 81, 82], in which adversaries send carefully-designed queries to the victim model and collect its responses to train a local model (often referred to as an *imitation model*) that mostly "imitate" the target remote model's behavior.

At first glance, conducting an imitation attack on commercial LLMs may seem impractical, as adversaries would need to prepare an imitation model with comparable parameters, resulting in high costs. However, in contrast to the broad capabilities of LLMs, developers typically require only specialized subsets of these abilities, as they concentrate on particular tasks such as code translation and summarization. This inspires us to explore a practical and novel direction: slicing commercial, black-box LLMs using medium-sized backbone models. In other words, we aim to demonstrate the high possibility of extracting *specialized code abilities* of LLMs using *medium-sized backbone models*. For instance, attackers may be particularly interested in extracting the ability of "code translation" from an LLM, which is a specialized code ability that allows the LLM to translate source code from one programming language to another. This ability is highly valuable in the software development industry and has been widely used in commercial products [62, 87]. The imitation attack also has the benefit of allowing users to avoid sharing their code snippets with third-party providers. This is possible by locally deploying the specialized imitation model extracted from LLMs via medium-sized backbone models.

Extracting specialized code abilities poses unique technical challenges. Depending on the specific code tasks, LLMs may process natural language (NL) or programming language (PL) inputs and emit NL/PL outputs accordingly. Moreover, modern LLMs can often be elicited by various in-context [77] or Chain-of-Thought prompts [78], making it difficult to determine the attack surface of LLMs for code abilities. Furthermore, it is unclear how to use the target LLM outputs to improve the performance of an imitation model, given the complexity of code-related tasks and potential ambiguity in the outputs.

Overall, this work is the first to conduct a systematic and practical study on the effectiveness of extracting specialized code abilities from LLMs using common medium-sized models. To address the above technical challenges, we design and conduct comprehensive imitation attacks on LLMs, including all three code-related tasks and different query schemes (Sec. 4.1). Additionally, we develop several methods to refine the received outputs of LLMs (Sec. 4.2), which make the polished outputs more effective in training the imitation model using two popular backbone models, CodeT5 [74] and CodeBERT [29] (Sec. 4.3). Finally, we demonstrate that the imitation model can boost critical downstream tasks, e.g., adversarial example generation (Sec. 4.4).

Based on the experimental results, we show that imitation attacks are effective for LLMs on code-related tasks, and the performance of the imitation models surpasses that of the target LLMs (with an average improvement of 10.33%). Moreover, we find substantial variance (from 4.94% to 62.83%) in the impact of different query schemes across tasks. Queries providing adequate context generally improve performance on all tasks, while the Chain-of-Thought scheme only benefits code synthesis noticeably. These findings underscore the importance of selecting an appropriate query scheme.

We also explore the influence of different hyperparameters. We find that while *temperature* and top_p have an observable (yet not significant) impact, the number of queries and in-context examples significantly affect the performance (from 41.97% to 115.43%). We recommend using three in-context examples for code-related tasks to balance the effectiveness and cost. Additionally, we demonstrate that the imitation model can assist in generating adversarial examples, discovering up to 9.5% more adversarial examples than the existing state-of-the-art models [40, 48, 70]. Finally, we show the generalizability of our method on different LLM APIs, with the imitation models trained on gpt-3.5-turbo achieving approximately 92.84% of the performance of those trained on text-davinci-003.

In summary, our contributions are as follows:

- We have conducted the first systematic study on the effectiveness of imitation attacks on a code knowledge slice of LLMs.
- Our study includes three representative query schemes and different code-related tasks. We have also designed several methods to refine the LLM outputs and enhance the training effectiveness of imitation models.
- We have formulated five research questions (RQs) to comprehensively evaluate the effectiveness of imitation attacks on LLMs for code abilities and their potential downstream applications, such as adversarial example generation. We have aggregated the results and observations to deduce empirical findings.

2 BACKGROUND AND RELATED WORK

2.1 Large Language Models for Code

With a substantial amount of training resources (e.g., webtext corpora [31]) and model parameters (up to hundreds of billions [21]), LLMs have manifested highly impressive performance on various tasks. Typically, input training data is segmented into sentences and then into tokens, where each token consists of a sequence of characters before being fed to LLMs. Previous works follow the classic "pre-train and fine-tune" paradigm [51], where a large number of datasets are used to train a general model as the public backbone, and users can fine-tune it with their private dataset and specialized task definition. Following this paradigm, CodeBERT [29] and CodeT5 [74] are two representative frameworks for code-related tasks.

Moreover, for models scaled to 7B+ parameters[1], users can often guide them to generate appropriate answers with texts referred to as *prompts*. Thus, a new paradigm called "pre-train and prompt" [50]

[1]We refer to the models with 7B+ parameters as large-sized models, 0.1B to 7B as medium-sized models, and the rest are small-sized models.

Table 1: Benchmarking prior knowledge for imitation attacks. The symbols ✓, ◇, ✗ denote require, partially require, and not require the corresponding knowledge when launching the model extraction, respectively.

	Data Distribution	Model Architecture	Output Probability	Object	Task	Victim model	Size
Chandrasekaran [23]	✓	✓	✓	-	classification	-	-
Jagielski [39]	◇	✓	✓	image	classification	academic	25.6M
Orekondy [58]	◇	✓	✓	image	classification	academic	21.8M
Yu et al. [85]	◇	✗	✗	image	classification	commercial	200M
He et al. [34]	✗	✗	◇	text	generation	academic	340M
Wallace [67]	✗	✗	✗	text	generation	commercial	-
Ours	✗	✗	✗	code	generation	commercial	175B

has emerged. This paradigm has been a huge success due to its powerful performance and high flexibility. For example, OpenAI's ChatGPT [5] allows people to ask it to complete different works with impressive accuracy and coherence given appropriate prompts. Due to the drastic improvement caused by prompts, an increasing number of commercial tools (e.g., Copilot [7]) are setting their backbones to LLMs that follow the "pre-train and prompt" paradigm.

Note that not all LLMs possess capabilities for various code-related tasks, despite having substantial model parameters. The effectiveness of a model is related to the datasets and methods used during training. For example, the compile rate [28], a popular metric assessing the compilation correctness of LLCG outputs, for LaMDA's (137B) on DeepFIX is only 4.3%, whereas Codex (12B) gains 81.1%. Therefore, this work only considers LLMs that have demonstrated reasonable success on code-related tasks.

With the surge of Machine-Learning-as-a-Service (MLaaS), model owners often provide their services via API or mature interface, and billing for queries can be broadly categorized as either monthly or "pay-as-you-go." For example, GitHub Copilot charges 10 USD per month [3]. 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 API. For instance, j1-jumbo [2] model from AI21 charges 0.03 USD per 1K tokens and 0.0003 USD per query. All these models only share generated content with their users, and thus the owners can maintain the confidentiality of the underlying model architecture and training data.

2.2 In-context Learning and Chain-of-Thought

Mainstream LLMs significantly benefit from context prompts. It is shown that simply prompted with task definitions [51] (also known as zero-shot learning) can often achieve satisfying results, and the performance can be further improved if more concrete in-context examples exist [77]. Since the LLMs do not seem to share the same understanding of prompts with humans [53, 75], a series of works in prompt engineering have been proposed [63, 64, 72, 78].

Among them, one particular prompt strategy called Chain-of-Thought (CoT) has been shown to elicit strong reasoning abilities in LLMs by asking the model to incorporate intermediate reasoning steps (rationales) while solving a problem [44, 46, 71, 78]. Wang et al. [72] sample from the reasoning path and vote for the majority result. Furthermore, Wang et al. [71] identify rationale sampling in the output space as the key component to robustly improve performance, and thus extending CoT to more tasks. Such reasoning ability is not a feature only found in LLMs, and several works have explored incorporating it in small models. Li et al. [44] use CoT-like reasoning from LLMs to train smaller models on a joint task of generating the solution and explaining the solution generated. With multi-step reasoning, Fu et al. [30] concentrate small models on a specific task and sacrifice its generalizability in exchange for high performance. To unleash the full potential of model extraction attacks, this work explores multiple query schemes (including CoT) and quantifies their influence on performance.

2.3 Imitation Attack

Grey-Box Imitation Attack. The imitation attack, also known as model extraction attack, aims to emulate the behavior of the victim model. Successfully extracting a model, especially commercial APIs, is quite challenging, and previous works [23, 39, 85] tend to simulate attacks in a grey-box setting, where different kinds of prior knowledge are required. As shown in Table 1, this prior knowledge includes the distribution of data, the model architecture, and the output with its probability. For example, jagielski et al. [39] require both prior knowledge like model architecture and posterior knowledge such as the probability of output tokens to aid in extraction. Moreover, since they mainly focus on the image classification task, domain knowledge like class types can be used to boost model extraction. For example, Yu et al. [85] ask for prior knowledge of class types to establish a query set. He et al. [34] demonstrate the vulnerability of powerful APIs built on fine-tuned BERT models to model extraction attacks, which exploit the generated tokens and corresponding posterior probabilities.

Black-Box Imitation Attack. It is evident that prior knowledge of the target model is not always available in practice. LLM architecture (e.g., GPT-4 [10]) and training datasets are typically kept hidden by their owners. With this regard, black-box imitation attack is more practical as it does not require the adversaries to have any prior knowledge about the model internals and its training data. To launch black-box imitation attack, attackers often first prepare a proxy dataset, which is further derived into a query set Q based on the API documentation of the target LLM. Then, each query $q \in Q$ is sent to the remote LLM API to obtain the corresponding output o. An imitation model can be trained with the collected dataset $\{q_i, o_i | q_i \in Q, o_i \in O\}$.

2.4 Adversarial Examples (AEs)

Adversarial attacks are known for their ability to introduce imperceptible changes to input data, resulting in incorrect output generated by a model. The inputs that induce failures are referred

to as AEs [32]. While it is relatively easy to generate AEs for common deep neural networks, LLMs are believed to be more resilient toward AEs [22, 70] due to the following reasons: (1) LLM often exposes only APIs in a "black-box" setting, and (2) the model vendors usually put a considerable amount of efforts into fine-tuning the model. For instance, OpenAI has disclosed that it spent over six months making GPT-4 safer and more aligned, assembling a team of over a hundred domain experts specializing in model alignment and adversarial testing before its public release [57].

Several works have been proposed to test and exploit the potential of adversarial examples in models designed for code-related tasks. CodeAttack [40] collected logit information from CodeT5 and CodeBert, makes small adjustments to code tokens, and generates AEs accordingly. Yang et al. [83] proposed a search-based framework named Radar to generate function names that cause AEs. CCTest [48] mutated Python code using several semantics-preserving transformations and detected inconsistent outputs in Copilot over those mutated inputs.

3 ETHICS AND RESPONSIBLE DISCLOSURE

The aim of our work is to enhance the resilience of LLM APIs, and we strongly believe that providing democratized access to sophisticated LLM APIs benefits all of humanity. To accomplish this goal, we conduct our experiments in a responsible manner and endeavor to minimize any potential real-world harm.

Minimal Real-world Harm. Since we discovered adversarial examples using our imitation model, we minimized harm by (1) avoiding any damage to real users and (2) reporting potential adversarial examples to OpenAI, as well as the detailed attacking process and methods used to find them.

Responsible Data Disclosure. As in previous work [17, 27, 66], we collected the dataset from OpenAI and used it to train the imitation model, which provides competitive services and aids in adversarial attacks. Therefore, to prevent potential misuse, we only make the scripts used for collecting the dataset publicly available and reveal the AEs that have already been fixed. Furthermore, we adhere strictly to the licenses of the backbone models and proxy datasets, and no profit has been gained.

Overall, we believe that the security of LLM ecosystem is best advanced by responsible researchers and model owners surfacing these problems.

4 TECHNICAL PIPELINE

In this work, we aim to launch imitation attacks to extract a "slicing" of code knowledge from LLMs using medium-sized backbone models. This task is challenging primarily due to LLMs can be queried in various ways, making it necessary to explore various query schemes. Their strong in-context understanding capabilities and flexibility across different tasks add to the complexity of benchmarking their attack surfaces. Specifically, LLMs have demonstrated strong in-context understanding capabilities [51]. With fewer examples provided, LLMs can achieve a better understanding of downstream tasks with higher performance [77]. Also, Chain-of-Thought reasoning [72, 78] elicits the complex reasoning ability in LLMs. These novel tasks and schemes make LLMs quite flexible and versatile under various tasks, thus making it highly challenging and costly to systematically benchmark the attack surfaces of LLMs against model extraction.

Fig. 1 presents an overview of our imitation attack, which consists of four phases: (1) query generation, (2) response checking, (3) imitation model training, and (4) downstream (adversarial) applications. Given one or more proxy datasets, our attack framework first generates LLM queries according to different code tasks and query schemes. We then employ a rule-based filter to check the correctness and quality of the responses provided by LLMs. Responses that pass the filter are considered of high-quality and used to train the imitation model. Next, we train the imitation model by fine-tuning medium-sized backbone models with the filtered responses. Finally, we use the imitation model for various downstream (malicious) applications, such as providing competitive service and boosting the generation of AEs. We now describe each phase in detail.

4.1 Imitation Query Generation

According to our preliminary tests, query schemes and prompt quality have a significant impact on eliciting LLM outputs. Therefore, to fully unleash the potential of this attack, we benchmark three query schemes as detailed below. Before introducing them, we first clarify how a query is decomposed into two parts: the question head Q_{head} and the question body Q_{body}. Note that the former varies from task to task, and the latter can be collected from proxy datasets. For example, for the code summarization task, we set the sentence "Summarize the following code in one sentence" as Q_{head}. This is beneficial because a clear and concise question can help LLMs understand their specific role within a task, which in turn enhances their abilities to effectively complete that task.

Zero-Shot Query (ZSQ). This scheme specifies that attackers will iteratively use each question $q_i \in Q$ to query the target LLM without preparing any context. Accordingly, the attackers will gather the responses to train the imitation model. ZSQ is universal and adopted by nearly all prior model extraction works [35, 36, 58, 85] in both classification and generation tasks. Since the tasks we evaluated (see Table 2) are all generation tasks, we follow [36, 85] to prepare the queries.

In-context Query (ICQ). Several studies [21, 42, 53, 78] have shown that providing in-context information can significantly improve LLM performance. Therefore, we also consider this query scheme. Specifically, for each question q_i, the query would provide several examples along with the question head Q_{head}. The choice of examples used as contexts plays a crucial role in enabling the model to understand the task. For instance, the number of examples can not be too few or too many because a short in-context may not provide enough information to the victim model, while a long one may cause the query to exceed the length limit. We study its influence in Sec. 5.3.

Zero-Shot CoT (ZS-COT). This scheme was first proposed in [41]. Unlike [78], which heavily relies on manually crafted prompts, the core idea behind ZS-CoT is quite simple: adding a prompt sentence like "*Let's think it step by step.*" to extract the reasoning process hidden in the model. We follow the definition proposed by [41] and treat ZS-CoT as a two-step prompt process. Specifically, given a standard query sample $s_i \in S$, which contains a question $q_i \in Q$, ZS-CoT first constructs a prompt that requires explanation (or

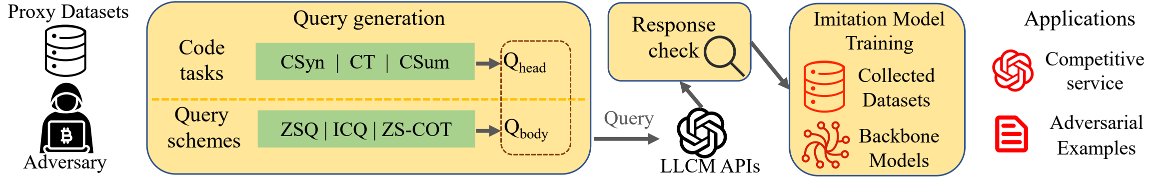

Figure 1: An overview of our imitation attack framework, including query generation, response check, imitation training, and downstream applications.

rationale) to the victim model and collects the response as r_i. In the second stage, it uses both q_i and r_i to ask for the final answer of victim models. Usually, the two-stage query sequence follows this form: "Q_{head} *Q:* $\langle q_i \rangle$. *A: Let's think it step by step.* $\langle r_i^1 \rangle$. *Therefore, the answer is* $\langle a_i^2 \rangle$."[2] In the experiment, we slightly changed the answer trigger pattern to adapt to the answer format. For example, we used "*Therefore, the translated C# code is*" for code translation task and "*Therefore, the summarization is*" for code summarization task. It is worth noting that even with a carefully designed filtering system provided by [37], it is hard to guarantee the correctness of the rationale. This is because the code-related tasks considered in this paper are open-ended, which poses a greater challenge than the multiple-choice questions in NLP reasoning tasks. As a result, we assess the generated rationales for each task (see Sec. 5.2 for details). It is worth noting that we do not consider in-context CoT (IC-CoT) in this paper. The main reason for this exclusion is the difficulty of constructing appropriate Chain-of-Thought reasoning processes [78] for code-related tasks, as opposed to arithmetic or symbolic reasoning tasks.

4.2 Response Check Scheme

As mentioned earlier, the collected responses should be refined to achieve high effectiveness in training an imitation model. That said, when attackers receive the response from LLMs, it is desirable to first perform a check before adding it to the training data. This can improve the overall quality of the dataset by helping attackers identify and rule out poor quality content, thereby increasing the average quality of responses. Additionally, trimming the training data can reduce the overall training cost of imitation models. Specifically, given a list of LLM outputs O^L, we check each output $o_i^L \in O^L$ based on several metrics and keep the high quality answers.

To handle LLMs that may produce both NL and PL outputs, we have devised distinct filtering rules for each. For NL outputs, we count the length and discard any text below or above pre-defined thresholds. Following the setting in CodexGLUE [52], the upper bound is set to 256, and the lower bound is set to 3.

For PL outputs, we only keep those that pass a grammar check by a parser. For this purpose, we use treesitter [14], a parser generator tool that can parse incomplete code fragments. Unlike language-specific compilers or interpreters (e.g., CPython for Python), treesitter supports most mainstream languages[3] with a unified interface, which makes it convenient to adapt to various PLs. Moreover, treesitter provides error messages for incorrect code syntax, which enables us to count the number of failures and further investigate the reasons behind them (see Sec. 5.2).

4.3 Imitation Model Training

Similar to previous imitation attacks [34, 67], answers that pass the response selection module are considered as high-quality answers and used to train the imitation model. Specifically, the collected response dataset $\{q_i, o_i | q_i \in Q, o_i \in O\}$ is used to fine-tune the public backbone model. In this subsection, we first provide details on the target LLMs and their code-related tasks that are the focus of our imitation attacks. We then explain the evaluation metrics used and the process for training the imitation models.

Target LLMs. Unless otherwise specified, we use OpenAI's text-davinci-003 [8] as the victim LLM for all experiments, since it has been widely used in prior research [20, 68], which supports its efficacy and reliability. In our evaluation (Sec. 5.5), we further demonstrate the generalizability of our attack by evaluating it with another LLM API called gpt-3.5-turbo.

Table 2: The evaluated tasks and datasets. CSyn, CT, and CSum denote code synthesis, code translation, and code summarization, respectively. CSN represents the CodeSearchNet dataset. D_{proxy} and D_{ref} are the proxy and reference datasets.

Category	D_{proxy}	D_{ref}	# Queries	Stat. of D_{ref}
CSyn	XLCOST [89]	CONALA [84]	2k	2k/-/500
CT	XLCOST [89]	CodeXGLUE [52]	10k	10k/500/1k
CSum	DualCODE [76]	CSN [38]	8k	25k/14k/15k

Target LLM Tasks. Before providing details on the filtering rules, we introduce the target LLM tasks in this research. Drawing from the variation in input and output types, we select three representative tasks: code synthesis, code translation, and code summarization. Importantly, our imitation attacks are not limited to these tasks. Two datasets, D_{proxy} and D_{ref}, are used here to emulate the extracting process. Adversaries are only allowed to use the train split of the proxy dataset D_{proxy} to construct queries. The reference dataset D_{ref}, on the other hand, is assumed to be inaccessible to adversaries. To establish baselines for comparison, backbone models will be trained on the two datasets to form M_{proxy} and M_{ref}, with details provided in Sec. 5.1.

Code Synthesis (CSyn). CSyn in this study refers to an "NL-PL" task that aims to generate specific programs based on NL descriptions. As shown in Table 2, we use CONALA as the proxy dataset for this task, which contains 2,879 annotations with their corresponding Python3 solutions manually collected from StackOverflow. We did not use an online-judgment dataset such as CodeContests [45] due

[2] The superscript for r_i and a_i indicates the stage at which they were collected. Readers can refer to [1] for details.
[3] Treesitter version 0.20.7 now supports 113 different programming languages.

to the input token length limitations. Further discussion on this is provided in Sec. 6.

Code Translation (CT). As a "PL-PL" task, CT involves migrating legacy software from one language to another. As shown in Table 2, we use XLCOST as the proxy dataset and CodeXGLUE as the reference one, where code snippets written in Java and C# that share the same functionality are paired together. We select Java as the source language and C# as the target language.

Code Summarization (CSum). As a "PL-NL" task, CSum generates an NL comment that summarizes the functionality of a given PL snippet. As shown in Table 2, we reuse the DualCODE dataset as the proxy dataset. We use D_{proxy} and D_{ref} to represent the proxy and reference dataset, respectively.

Evaluation Metrics. We explore two common similarity metrics for measuring the quality of generated content. We categorize them as follows:

NL Content. For CSum, whose generated content is NL text, we follow previous works [29, 48, 74] and use the smoothed BLEU-4 score (referred to as BLEU in the rest of this paper) to evaluate the generated NL summarization. Given the generated text and its ground truth, BLEU examines the number of matched subsequences, and a higher BLEU score suggests greater similarity at the token level.

PL Content. The outputs of the CSyn and CT tasks are PL code snippets, which cannot be directly evaluated using NL metrics. Therefore, similar to previous works [16, 45, 52, 74], we use CodeBLEU [61], a metric that takes into account token-level, structural-level, and semantic-level information. Overall, CodeBLEU consists of four components: n-gram matching score $BLEU$, weighted n-gram matching score $weighted_BLEU$, syntactic AST matching score AST_Score, and semantic data flow matching score DF_Score. Specifically,

$$\begin{aligned} CodeBLEU &= \alpha * BLEU + \beta * weighted_BLEU \\ &+ \gamma * AST_Score + \delta * DF_Score \end{aligned} \tag{1}$$

where $\alpha, \beta, \gamma, \delta$ are the weights for each component. As suggested in [52, 74], they are all set to 0.25. Note that both BLEU and CodeBLEU scores range from 0 to 100, with higher scores indicating a greater level of similarity.

Imitation Models. As noted in Sec. 2, adversaries can follow the classic "pre-train and fine-tune" paradigm, training their imitation models via fine-tuning a well-trained backbone model. In this work, we choose two representative models for code-related tasks: CodeBERT [29] and CodeT5 [74]. CodeT5, a variant of the text-to-text Transformer [60] model, treats all text tasks as a sequence-to-sequence paradigm with different task-specific prefixes, making it suitable for both code understanding and code generation tasks. On the other hand, CodeBERT is built on a multi-layer bidirectional Transformer encoder and pre-trained on large-scale text-code pairs using two tasks: masked language modeling (MLM) and replaced token detection (RTD). In the MLM task, the model predicts the original token in the masked positions, while in the RTD task, a discriminator is used to distinguish the replaced tokens from the normal ones. These two models are selected for this study due to their widespread adoption and strong performance on various tasks, as evidenced by prior work [56, 69]. Specifically, CodeBERT-base employs a 125 million parameter encoder-only architecture, whereas CodeT5-base utilizes a 220 million parameter encoder-decoder design tailored for sequence-to-sequence tasks.

Training Setup. To ensure the reproducibility of our results, we train the imitation model three times for each experiment and report the median of the achieved results. In addition, we analyze the impact of different hyperparameters in Sec. 5.4. All experiments are performed on a machine with an Intel Xeon Patinum 8276 CPU, 256 GB of main memory, and 4 NVIDIA A100 GPUs. By using the early stopping strategy, the training process takes an average of 5 hours and 23 minutes to complete.

4.4 AE Generation

As discussed in Sec. 2.4, AE generation has become a common technique to improve model robustness. However, discovering AEs effectively remains challenging for advanced models. To address this, we demonstrate the feasibility of boosting AE generation for code-related tasks using an imitation model. In contrast to the deficiency of prior methods against modern LLMs (detailed in Table 6), our approach using the imitation model allows for generating AEs in an approximately white-box setting. Specifically, given the M_{imi} on hand, we rely on the attention scores to locate sensitive tokens in an input prompt, and then apply several semantically-equivalent transformations (derived and extended from the CCTest's codebase [4]) to iteratively mutate those sensitive tokens until generating AEs on M_{imi}. These generated AEs are then fed to the remote victim LLM to test whether they can induce failures.

To demonstrate, we apply our approach to the code summarization (CSum) task, where a successful AE means a subtle mutation in the code input leads to a dramatically changed, incorrect summary. We generate potential AEs in two steps: (i) identifying the most sensitive tokens and (ii) applying semantically-equivalent transformation passes on those tokens. To do so, we start with an input sequence $\mathcal{X} = [x_1, .., x_i, ..., x_m]$ and generate the output sequence (code summary) with its original attention score Att_{ori}. Next, we iteratively replace each token x_i with $[MASK]$ and re-generate the summary with its corresponding attention score Att_i. We then quantify the gap score by computing $Gap_i = Att_i - Att_{ori}$ for each token. Finally, we rank all tokens according to their gap scores in descending order to find the most sensitive tokens. We then check each of the top-k sensitive tokens and decide if it satisfies any of the transformation passes offered by CCTest. If it does, we apply the transformation to generate the potential AE.[4]

5 RESEARCH QUESTIONS AND RESULTS

In this section, we aim to experimentally investigate the effectiveness of extracting specialized abilities from LLMs by answering the following research questions (RQs).

- RQ1: How effective is the imitation attack in code-related tasks?
- RQ2: How do different query schemes impact the performance of the imitation attack?
- RQ3: How do hyperparameters affect the performance and cost of imitation attacks?

[4]Due to page limit, we provide the full documentation and tutorial on our website [1] for this step.

- RQ4: To what extent can the use of imitation models help generate adversarial examples against LLM APIs?
- RQ5: How generalizable is the imitation attack across different LLM APIs?

5.1 RQ1: Effectiveness of the Imitation Attack

To answer RQ1, we study the effectiveness of our launched imitation attack on three code-related tasks: CSyn, CT, and CSum. As noted in Sec. 4.3 and Table 2, we use the proxy dataset to launch queries towards the target LLM and validate the performance of the imitation model on the reference dataset.

Table 3: The main results of our imitation attack. "I/O" stands for "Input/Output." All results are presented as BLEU scores or CodeBLEU scores on the test split of reference datasets, where M_{proxy} and M_{ref} represent the backbone models trained on the proxy and reference datasets, respectively. M_{imi} is the imitation model trained on the collected dataset and "API" stands for the best original LLM result under all three query settings. M_{pure} is the backbone model without fine-tuning.

	I/O Type	Model	API	M_{imi}	M_{proxy}	M_{ref}	M_{pure}
CSyn	NL/PL	CodeT5	27.51	24.84	11.53	24.21	1.40
		CodeBERT		18.61	9.41	17.09	N/A
CT	PL/PL	CodeT5	69.15	72.19	27.21	84.30	4.38
		CodeBERT		68.58	24.82	79.05	N/A
CSum	PL/NL	CodeT5	12.90	17.72	17.25	18.95	3.84
		CodeBERT		14.09	12.20	14.87	N/A

Table 3 presents the main results of our model extraction attack. The M_{proxy} and M_{ref} columns show two baseline settings in which the backbone models are trained directly on the proxy dataset and the reference dataset, respectively. The M_{imi} column reports the attack accuracy, while the API column shows the LLM performance. It is essential to note that M_{proxy}, M_{ref}, and M_{imi} have the same model architecture (either CodeT5-base or CodeBERT-base), and their training datasets are of equal size. We tested all three models on the same test dataset and chose the best performance from all three query schemes. The values in each cell represent the BLEU score for natural language texts and the CodeBLEU score for programming language contents, as mentioned earlier in Sec. 4.3. Furthermore, since all three models (and the LLM APIs) are assessed using the test split of the reference dataset, it is reasonable to observe that M_{ref} (which is trained using the train split of the same dataset) consistently achieves the best performance across all three tasks.

From Table 3, it is evident that the imitation attacks are highly effective. First, the performance of CSyn and CT tasks is quite promising, as the imitation models M_{imi} outperform M_{proxy} by an average of 57.59% and 56.62% on CodeT5 and CodeBert, respectively. Note that the proxy dataset comprises a set of input-output tuples, whereas M_{imi} is trained using the same inputs and their corresponding outputs from the LLM API. Therefore, we attribute the superiority of M_{imi} over M_{proxy} to the fact that LLM APIs provide high-quality code snippets as outputs, which further enhance the performance of the imitation model. Moreover, M_{proxy} consistently performs the worst on all three tasks, indicating the high value of the outputs provided by the LLMs.

Code	`def resource_patch(context, data_dict):`
	`    _check_access('resource_patch', context, data_dict)`
	`    show_context = {'model': context['model'], 'session': context['session']}`
	`    resource_dict = _get_action('resource_show')(show_context)`
	`    patched = dict(resource_dict)`
	`    patched.update(data_dict)`
	`    return _update.resource_update(context, patched)`

Sum	Ground truth: Patch a resource.
	LLM: Update a resource by checking access, showing the context and patching the resource with updated data.

Figure 2: An example to demonstrate why LLM APIs tend to have low scores on the code summarization task, which is because the ground truth for this task uses short summaries.

When models are trained using publicly available resources, they may perform poorly when tested on different datasets, as evidenced by the results in the M_{proxy} column of Table 3. This is one reason why adversaries may resort to extracting LLMs. Encouragingly, we find that the imitation model M_{imi}, when being enhanced with knowledge extracted from the LLM APIs, can largely outperform the baseline model M_{proxy} in both the CSyn and CT tasks, achieving an average improvement of 140.3% and 170.8% on CodeT5 and CodeBert, respectively. Moreover, we observe highly promising results that M_{imi} can even outperform the LLM APIs on the CT and CSum tasks, demonstrating the general effectiveness of our imitation model.

Comparing M_{imi} and M_{proxy}, we find that the improvement of the imitation model M_{imi} on the CSum task is less significant than the improvement on the other tasks. However, we also notice a perplexing observation that the LLM APIs perform even worse than M_{proxy} on the CSum task. Upon manual inspection, we discover that the APIs tend to return verbose contents with abundant information if no additional context is provided, leading to a decrease in performance. For instance, in Fig. 2, the ground truth answer for the PL input is "*patch a resource,*" which is concise and straightforward. However, the answer provided by LLMs is "*Update a resource by checking access, showing the context, and patching the resource with updated data,*" which is more lengthy. This phenomenon has been previously mentioned in [22], where GPT-4 was utilized to address the issues with the similarity metrics. However, we find this solution impractical because employing powerful LLM APIs such as GPT-4 as an automatic judge would still yield biased judgment results [86]. LLM judges tend to assign higher scores to lengthy responses [73], even when conciseness suffices.

Impact on Pre-Trained Models. As noted in Sec. 4.3, both CodeT5 and CodeBERT were pre-trained on large corpora that may overlap with our chosen test sets. To mitigate this threat to validity, we report the baseline results of the unmodified backbone models on the test splits in Table 3, in the M_{pure} column. For CodeT5, performance is substantially lower across all three tasks compared to the other settings, indicating that the main capability of the imitation model M_{imi} is not mainly inherent from pre-training. Note that CodeBERT is an encoder-only model: benchmarking its decoding capability requires training a task-specific decoder, and its baseline performance is thus marked as "N/A". Overall, the largely

improved fine-tuning performance M_{imi} indicates that pre-training alone cannot account for the models' proficiency on three code-related tasks.

Finding 1: Extracting specialized code abilities of LLMs through medium-sized backbone models is effective for representative code-related tasks. The trained imitation models achieve comparable, if not better performance than the original LLMs in those specialized code abilities.

Table 4: Attack effectiveness using different query schemes. CBLEU denotes the CodeBLEU metric.

Task	Model	Metric	ZSQ	ICQ	ZS-COT
CSyn	CodeT5	CBLEU	23.39	23.67	24.84
	CodeBERT	CBLEU	15.99	16.59	18.61
		r_f%	2.48	2.40	4.62
CT	CodeT5	CBLEU	36.31	72.19	34.64
	CodeBERT	CBLEU	37.13	68.58	34.68
		r_f%	28.61	20.72	27.02
CSum	CodeT5	BLEU	10.95	17.72	12.25
	CodeBERT	BLEU	9.80	14.09	11.51
		r_f%	0.00	0.15	0.06

5.2 RQ2: Influence of Different Query Schemes

To answer RQ2, we aim to explore the impact of different query schemes on the data quality and the performance of imitation attacks. Recall in Sec. 2, we have introduced three query schemes. Here, we present the comparison results on three tasks in Table 4. For each setting, we represent the performance scores and the failure rates.

Imitation Performance. From the result listed, we can find that the ICQ achieves better performance than the other schemes on both CSum and CT tasks. In particular, the BLEU score on CSum for ICQ achieves 62.83% and 36.91% improvement, compared with ZSQ and ZS-COT, respectively. These findings suggest that incorporating context information can greatly enhance the quality of imitation training by providing more accurate answers. We note that this finding is consistent with previous research [21, 42] that highlights the importance of context in natural language processing tasks.

Moreover, ZS-COT achieves a CodeBLEU score of 24.84 on the CSyn task for CodeT5, which outperforms ICQ by 4.94%. We further compare four subscores in CodeBLEU (see Equ. 1) and find that the semantic data flow matching score *DF_Score* of ZS-COT is 40.65, 24.12% higher than that of the ICQ scheme. To explore the root cause of our findings, two authors of this paper manually checked 150 ZS-COT responses for each task. Surprisingly, only 7 (4.6%) responses provide a meaningful rationale on CSum.[5] For the remaining responses, the rationale r_i was the same as the final answer a_i, indicating that ZS-COT spends a large amount of resources but gains a limited amount of useful information.

On the other hand, 13 (8.6%) rationales are meaningful on CSyn, which is nearly twice as many as those in other schemes. We attribute this to the fact that the PL inputs in CSum and CT lack clear multi-step thinking in their questions, unlike the NL inputs in reasoning tasks. We find that reasoning problems present their thought processes more explicitly, while implicit thoughts in code tasks are harder for ZS-COT to capture. Therefore, it is understandable why ZS-COT performs better on CSyn, as its context is more closely aligned with natural language expressions found in reasoning tasks.

[5] The Cohen's Kappa is 0.96, indicating that the inspection results among two authors are highly consistent.

Failure Rates. To give a comprehensive understanding of the influence of query schemes, we also explore the difference between generated contents. As we have described in Sec. 4.2, we count the number of failures among the texts and code snippets and return the failure rate r_f%. For the CSum task, all schemes share a low failure rate (less than 0.2%), with CoT having a slightly higher failure rate as it may contain the extra reasoning steps in the final content that exceeds the upper bound; it may also generate a too concise summary that is below the lower bound. Additionally, in comparison with ZSQ and ICQ, ZS-COT on CSyn has a notably high failure rate of 4.62%. This implies that the Chain-of-Thought may be less appropriate for tasks that aim to generate code snippets. Note that all three schemes manifest high failure rates (25%) on CT, which is due to the nature of the CT task. As described in Sec. 4, we convert the Java programs into C# programs at the function-level, which diminishes some key information. To empirically confirm this, we repeat the same grammar check experiment on the test split of the reference datasets, which turns out to have a comparably high failure rate of 17%.

Influence of Response Check. As described in Sec. 4, we design a response check algorithm to remove low-quality responses. At this step, we explore its influence on the CSum task. We report that when enabling this algorithm, our approach can lower the training time by approximately 34.81% and 14.98% for CodeT5 and CodeBERT, respectively. Additionally, the imitation models' performance is decreased by a maximum of only 4%.

Finding 2: Query schemes have a major impact on the performance of imitation attacks. It is vital to construct query schemes with a suitable template design. Queries with sufficient context generally enhance the attack toward all tasks, while the CoT only shows a noticeable boost in code synthesis.

5.3 RQ3: Impact of Hyperparameters

To answer RQ3, we need to study the impact of hyperparameters. To do so, we explore the following three aspects: 1) victim model parameters, 2) the number of in-context examples, and 3) the number of issued queries.

Victim Model Parameters. According to OpenAI [8], text-davinci-003 mainly provides two parameters that control its generated outputs: *temperature*, which represents the propensity for choosing the unlikely tokens; and top_p, which controls the sampling for the set of possible tokens at each step of generation. To determine the appropriate value for these parameters, we prompt text-davinci-003 with all 25 combinations (5 $temperature \times 5\ top_p$) at step increments of 0.25. We use the CSum task at this step and query those 25 sets of combinations, and the responses are then compared with the ground truth answers to compute a BLEU score.

Fig. 3 (a) reports the evaluation results. For each setting, we issue 50 queries, collect the answers yielded by the LLM, and compare them with the ground truth (for CSum, ground truth is natural language snippets for code summarization). An answer deems passing

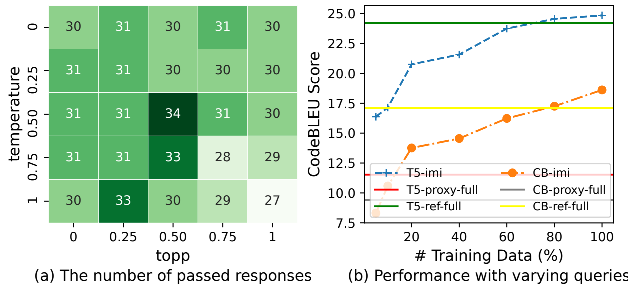

Figure 3: The impact of hyperparameters (RQ3).

the check if its similarity score is higher than the average value and otherwise as a failure. We count the number of passing answers and present them in each cell. The performance clearly degrades with edge values for both hyperparameters. Across the experiments, the best parameters are seen as $top_p = 0.5$ and $temperature = 0.5$.

In-context Examples Number. As noted in Sec. 2.2, the number of in-context examples E plays an important role in understanding the task, and previous works [25, 50] have used a range of E varying from two to dozens. However, this is not practical in code-related tasks, since the average token length of code snippets is much longer than the natural language texts. Moreover, the victim LLM used in this paper has a maximum token length limit (4,097) for each query, including both inputs and outputs. Therefore, following [42], we only use example numbers ranging from 1 to 5 on the CSum task to avoid exceeding the request limitation.

Table 5: Attack performance under different number of in-context examples. T5 and CB stand for CodeT5 and CodeBERT, respectively.

# In-context examples		1	2	3	4	5
Model	T5	14.20	16.51	17.05	17.72	16.96
	CB	10.87	12.90	13.95	14.09	13.88
Cost		11.24	20.53	30.97	42.85	53.29

The outcomes are shown in Table 5, indicating that the performance level escalates with an increase in #in-context examples and reaches its peak at $E = 4$. However, there exists a discrepancy between the improvement in performance and the accompanying cost. Although performance gradually enhances at a dropping rate over time, sustaining optimal performance incurs linearly growing costs. This is because every new example added demands extra tokens associated with that example for all queries, even those that have already accomplished satisfactory performance. Considering this trade-off, adversaries must carefully assess their options. For imitation, we suggest fixing the example number to three. We regard this as a reasonable trade-off that adversaries are willing to embrace during implementation.

Number of Queries. Considering that the performance of imitation attacks heavily relies on the number of collected responses [58, 67], an evident trade-off exists between the number of queries and the performance of the imitation model. Too few queries can lead to poor performance while too many queries may cause an unaffordable cost, and neither scenario is desirable for adversaries. We now study the impact of #queries on imitation attack performance.

The evaluation results are shown in Fig. 3(b), where T5 and CB represent CodeT5 and CodeBERT, respectively. Due to the limited space, we report the evaluation results on the CSyn task. Other tasks share similar findings; see [1] for the full results. In short, when the adversary-collected dataset is equivalent in size to that of the model owners' training dataset, the imitation model surpasses the model trained on the proxy dataset ("T5/CB-proxy-full") and attains comparable performance to the model trained on the reference dataset ("T5/CB-ref-full"). This clearly shows the value of conducting imitation attacks, as the adversaries in real-world usually face the problem of lacking appropriate training data. We note that for the CSyn task, the increasing number of queries could help address the insufficient data problem and boost performance. With only 5% queries, our imitation model M_{imi} trained on the collected data can achieve a notable improvement of 41.97% on CodeT5 as compared to M_{proxy}, and the performance keeps increasing and reaches a peak with a 115.43% improvement.

Upon analyzing the collected dataset, we view that the internal diversity and high quality of gathered data were the reasons behind the improvement. For example, when given the same NL instruction "*decode a hex string '4a4b4c' to 'UTF-8'*", the model trained on the proxy dataset returns the "*hex = hex_decode(hex) return hex*" that is full of repeated tokens "*hex*". In contrast, M_{imi} generates "*bytes.fromhex('4a4b4c').decode('utf-8')*", a compilable code snippet that uses the built-in APIs correctly. One remaining question is whether such improvement can keep increasing to a considerably large extent. Unfortunately, we notice that the boost from the increasing number of queries gradually diminishes with its growing size, which reflects the margin effect of #queries.

> **Finding 3:** The output/sampling hyperparameters have an observable (yet not significant) impact on the attack performance. In contrast, #queries and #in-context examples notably affect the attack performance.

5.4 RQ4: Boosting AE Generation

After attackers obtain a well-performing imitation model M_{imi}, this RQ investigates the feasibility of generating AEs to further exploit the target LLM and manipulate its outputs.

In this RQ, we study the code summarization task (CSum), where a successful AE for this task constitutes a subtle perturbation to the input code which results in a significantly altered, incorrect summary from the model. We note that the same method can be easily extended to other tasks. As discussed in Sec. 5.1, the natural language descriptions in the CodeSearchNet dataset are highly ambiguous, which increases the difficulty in distinguishing the real AEs. Therefore, we use the Leetcode [11] dataset as the test dataset, as it provides clearer explanations for the given code snippets.

As introduced in Sec. 4.4, there are two primary steps in generating the potential adversarial examples: (i) Finding the most vulnerable tokens. (ii) Applying the semantically-equivalent transformation passes on selected tokens. We illustrate the process in Fig. 4, where the original input code aims to test whether a given number is a palindrome. The selected tokens are "test, x, False, ba" (in red text box). Since variable "x" satisfies the mathematical constant transformation in CCTest, we replace it with "x*x/x",

and it turns out to be an AE for the text-davinci-003 API. It is important to note that AE generation for NLP tasks typically takes hours to produce a single instance [91]. In contrast, our proposed method can generate adversarial examples in just seconds to minutes, depending on the input length and the choice of the imitation backbone model. This efficiency arises from the use of an imitation model to guide the adversarial example generation. As the imitation model is much smaller than the target language models, inference is considerably faster.

Code:

```
def test(x):
    def mydiv(x,ba):
        return x//ba,x%ba
    back,ba = 0,10
    b = back + x
    if x < 0:
        return False
    while x>0:
        x,div_tmp = mydiv(x,ba*ba/ba)
        back = back * ba
        back += div_tmp
    return back == b
```

Adversarial Example:

```
def test(x):
    def mydiv(x,ba):
        return x//ba,x%ba
    back,ba = 0,10
    b = back + x*x/x
    if x < 0:
        return False
    while x>0:
        x,div_tmp = mydiv(x,ba*ba/ba)
        back = back *ba
        back += div_tmp
    return back == b
```

Sum:

Before: This code tests if a given number is a palindrome.

After: This code tests if a given number is equal to the sum of its digits squared.

Figure 4: Adversarial examples generation.

Besides, we also compare our approach with several previous works introduced in Sec. 2.4 to demonstrate its effectiveness. Overall, these works are considered state-of-the-art (SOTA) AE generation methods for code models under white-box or black-box settings, and we follow their original settings to conduct the experiments below.

Table 6: Comparison for different adversarial attacking methods. Sem EQ, SAE rate, and UAE rate stand for semantically equal, stable AE rate and unstable AE rate.

Method	Type	Sem EQ?	SAE rate	UAE rate
CodeAttack	Whitebox	False	1.11 %	4.44 %
Radar	Blackbox	True	0 %	1.47 %
CCTest	Blackbox	True	0 %	1.13 %
Ours	M_{imi}-enabled Whitebox	True	9.5 %	4.78%

We generate up to 90 potential AEs for each of the prior works and our method, and use them to query text-davinci-003. Then, we manually inspect the responses and report the results in Table 6. Note that OpenAI models are non-deterministic [8], meaning that identical inputs can yield different outputs. Setting $top_p = 1$ and $temperature = 0$ will make the outputs mostly deterministic, but a small amount of variability may remain. Therefore, we repeat querying each AE three times and divide them into two categories: stable AE (SAE) and unstable AE (UAE). The former demonstrates that this example can be triggered every time we query (more desirable), while the latter indicates that the attack succeeds for at least one time. Both types of AEs have practical implications, as real-world users may query a model multiple times to gain confidence in the result, or just once without repeating. It is thus evident that the former users are presumably vulnerable under UAE and constantly vulnerable under SAE, whereas the latter is constantly vulnerable under SAE. It is worth noting that the example shown in Fig. 4 represents a stable AE, meaning it can be triggered each time we query the text-davinci-003 API.

Table 6 reports highly encouraging results, indicating that our method is effective in finding potential AEs in such a challenging setting. Among methods that preserve input semantics (a generally more desirable property), CCTest fails to trigger any incorrect content, while Radar only generates one AE which cannot be triggered stably. In contrast, our method can find both stable and unstable AEs. We deem that the imitation model M_{imi} provides valuable guidance to the process of AE generation.

Additionally, we notice that CodeAttack, as a SOTA *white-box* attack method, can also trigger some misleading content. Specifically, it discovers stable and unstable adversarial examples at rates of 1% and 4%, respectively. However, we find that it does not preserve the semantic equivalence of the input, as it would aggressively replace or delete sensitive tokens. This renders nearly one-third of the code snippets unable to be compiled without any error [40]. Furthermore, it is evident that a white-box attacking method is not applicable to real-world LLMs.

Finding 4: The imitation model provides useful information, e.g., attention score, to facilitate the generation of adversarial examples against LLMs. With this information, we successfully discover multiple AEs that can be stably triggered on LLM APIs.

5.5 RQ5: Generalizability

In this RQ, we investigate the generalizability of our imitation attack. To do so, we follow an identical process to query other LLM APIs and train our imitation model on the collected dataset. As we introduced in Sec. 4, we conducted all experiments on text-davinci-003 due to its great capacity. OpenAI also provides gpt-3.5-turbo, which is highly optimized for chat. This model is also popular, due to its relatively lower cost (1/10th of the cost) in comparison to the davinci series of models. Notably, we exclude GPT-4 here since its API remains publicly unavailable, and its web service has a strict rate limit. While it is unclear about the internals of gpt-3.5-turbo, we believe evaluating two highly popular and representative LLMs (text-davinci-003 and gpt-3.5-turbo) is sufficient to demonstrate the generalizability of our approach.

Table 7: Comparison for additional LLM APIs. TD-003 and GPT-35 stand for “text-davinci-003” and “gpt-3.5-turbo”.

	API		IMI	
	TD-003	GPT-35	TD-003	GPT-35
CSyn	27.51	24.11	24.84	22.85
CT	69.15	65.33	72.19	67.40
CSum	12.90	12.2	17.72	16.51

Table 7 presents consistently promising results when attacking gpt-3.5-turbo. Similar to RQ1, we report the best result among all query strategies (M_{imi} performance is in the “IMI” column). The values in each cell represent the BLEU score for NL texts and the CodeBLEU score for PL contents. Notably, gpt-3.5-turbo achieves competitive performance on all three tasks compared to text-davinci-003 (in the “API” column), with an average decrease of 7.78%. Overall, we find that the imitation models trained on data collected from gpt-3.5-turbo achieve approximately 92.84% of the performance of those trained on text-davinci-003, demonstrating the high generalizability of our imitation attack method. As an implication, attackers in reality may prefer to launch imitation attacks toward gpt-3.5-turbo, which is more cost-effective and offers comparable imitation model performance.

Finding 5: Our imitation attack method exhibits encouraging generalizability and can adapt to different LLM APIs without extra adaptiveness. This illustrates potentially more severe threats in reality, as attackers in reality may smoothly transfer the attack to other LLM APIs with lower cost.

6 DISCUSSION

Threats to Validity. The findings of this work face certain threats to validity that merit acknowledgment. First, the generalizability of the results may be limited given the specific LLMs and backbone models evaluated. While text-davinci-003 and gpt-3.5-turbo are selected as representative LLMs, more advanced models such as GPT-4 [10] and Claude2 [6] are not assessed. Additionally, we consider only two backbone models (CodeT5 and CodeBERT); recently proposed alternatives such as Starcoder [43] and WizardCoder [54] are not tested. It remains unclear how effective the proposed imitation attacks would be against these alternative LLMs and backbones. Second, the performance measurements obtained may depend heavily on the specific tasks selected for analysis. While we choose three representative code tasks, outcomes could vary substantially on different tasks such as code completion. Overall, more analysis on diverse models and tasks would strengthen conclusions.

Dataset Choice. As mentioned in Sec. 4, we do not choose those common online-judgement datasets such as CodeContests [45] for imitation attacks. This is because the average length of its NL description is over hundreds of tokens (not including the corresponding explanation). Our preliminary study shows that this is often too lengthy for medium-sized models to learn. Additionally, recall that we explored the in-context query scheme for each task, which requires the length to be several times larger than the original question. Hence, lengthy queries would exceed the max token length limit frequently, thereby failing the query process.

Semantically Equal Measurement. In this research, we use the CodeBLEU score to assess the caliber of the produced code snippets. However, these methods are imperfect since they are built on the token level and the AST level rather than the semantic level, and this leads to the scenario that two semantically equal code snippets may have a relatively low CodeBLEU score. Some methods like dynamic testing [33] or symbolic execution [24, 59] have been proposed for semantics-level comparison; however, such methods are either too heavy to be practical or even impractical for code snippet which does not have a clear input and output. In recent studies, pass@k has been proposed as a metric for evaluating the quality of generated code [26]. This metric measures the percentage of generated code snippets that pass all given test cases. However, the strict requirements of the test cases pose a challenge to its application in our research context. In sum, we clarify that it is common to use CodeBLEU scores as a metric for assessing the quality of generated code and to demonstrate that the model is capable of producing better code [16, 45, 52, 74], and we leave exploring other metrics for future work.

Mitigation with Watermarking. Watermarking is a common technique for protecting intellectual property (IP). In the context of neural networks, watermarking methods may involve subtly modifying the text/code (e.g., distributions of certain synonyms) to embed IP information [35, 36, 49]. Model owners can then verify their IP using statistical tests, such as Student's t-test. LLM vendors may have embedded working prototypes of watermarks in their APIs [13], though the details are unclear. However, we envision that attackers may consider its potential enforcement, and pursuing attacks imprudently would be improper, especially when imitation models are later used for commercial purposes. For instance, Google has been accused of training its AI chatbot Bard on data from OpenAI's ChatGPT without authorization [9]. Therefore, it is possible that extracting LLM's specialized code abilities may not be as concerning as our paper suggests. Looking ahead, we advocate better use of watermark and other relevant techniques to mitigate model extraction attacks.

Reflection of Our Findings. The present study demonstrates the feasibility of extracting specialized code abilities of LLMs through imitation attacks. By employing various query schemes, we show that the imitation models can achieve comparable performance to the original LLMs in three code tasks and can also enhance downstream applications such as adversarial example generation. Our findings hold significant implications for the research community. First, the ability to extract LLMs' specialized code abilities poses a serious threat to the LLM vendors who must safeguard their intellectual property. Second, the effectiveness of the imitation attacks is heavily influenced by the query schemes employed, highlighting the need for a balanced approach that optimizes both cost and performance. Third, the imitation models can positively impact downstream applications such as adversarial example generation, which can further enhance the robustness of LLMs. Given the significance of these findings, further research is warranted in this area, and we hope that our study will inspire future investigations.

7 CONCLUSION

In this paper, we experimentally investigated the effectiveness of extracting specialized code abilities from LLMs using common medium-sized models. To do that, we designed an imitation attack framework that comprises query generation, response check, imitation training, and downstream malicious applications. Our evaluation showed that the generated imitation models can achieve comparable performance to or even outperform the target LLM APIs, as well as provide useful information to facilitate the generation of adversarial examples against LLMs. We summarized our findings and insights to help researchers better understand the threats posed by imitation attacks.

ACKNOWLEDGEMENT

We thank anonymous reviewers for their valuable feedback. The HKUST authors were supported in part by an RGC GRF grant under the contract 16214723, RMGS24EG03, and RMGS24CR01. The HITSZ authors were supported in part by Natural Science Foundation of Guangdong Province under grant No. 2023A1515011959, and Shenzhen Basic Research under grant No. JCYJ20220531095214031. The NTU authors were supported in part by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] [n. d.]. Artifacts. https://sites.google.com/view/saellcm/main.

[2] [n. d.]. billing for AI21 LAB. https://studio.ai21.com/pricing.

[3] [n. d.]. billing for GitHub Copilot. https://docs.github.com/en/billing/managing-billing-for-github-copilot/about-billing-for-github-copilot.

[4] [n. d.]. CCTest's codebase. https://sites.google.com/view/cctest-info/main-page.

[5] [n. d.]. chatgpt. https://chat.openai.com/chat.

[6] [n. d.]. claude2. https://www.anthropic.com/index/claude-2.

[7] [n. d.]. Copilot. https://github.com/features/copilot.

[8] [n. d.]. davinci. https://platform.openai.com/docs/models/gpt-3-5.

[9] [n. d.]. Google Bard steal. https://www.theinformation.com/articles/alphabets-google-and-deepmind-pause-grudges-join-forces-to-chase-openai.

[10] [n. d.]. gpt4. https://cdn.openai.com/papers/gpt-4-system-card.pdf.

[11] [n. d.]. leetcode. https://github.com/qiyuangong/leetcode.

[12] [n. d.]. Openai. https://openai.com/product.

[13] [n. d.]. OpenAI Watermarking prototype. https://scottaaronson.blog/?p=6823.

[14] [n. d.]. treesitter. https://tree-sitter.github.io/tree-sitter/.

[15] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2023. Fixing Hardware Security Bugs with Large Language Models. *arXiv preprint arXiv:2302.01215* (2023).

[16] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tür, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics.

[17] Yuvanesh Anand, Zach Nussbaum, Brandon Duderstadt, Benjamin Schmidt, and Andriy Mulyar. 2023. GPT4All: Training an Assistant-style Chatbot with Large Scale Data Distillation from GPT-3.5-Turbo. https://github.com/nomic-ai/gpt4all.

[18] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*.

[19] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[20] Michael Bommarito II and Daniel Martin Katz. 2022. GPT Takes the Bar Exam. *arXiv preprint arXiv:2212.14402* (2022).

[21] 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.

[22] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).

[23] Varun Chandrasekaran, Kamalika Chaudhuri, Irene Giacomelli, Somesh Jha, and Songbai Yan. 2020. Exploring connections between active learning and model extraction. In *Proceedings of the 29th USENIX Conference on Security Symposium.*

[24] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. 2022. {SYMSAN}: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2531–2548.

[25] Mingda Chen, Jingfei Du, Ramakanth Pasunuru, Todor Mihaylov, Srini Iyer, Veselin Stoyanov, and Zornitsa Kozareva. 2022. Improving In-Context Few-Shot Learning via Self-Supervised Training. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022.*

[26] 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).

[27] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. https://vicuna.lmsys.org

[28] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[29] 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 *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics.

[30] Yao Fu, Hao Peng, Litu Ou, Ashish Sabharwal, and Tushar Khot. 2023. Specializing Smaller Language Models towards Multi-Step Reasoning. *arXiv preprint arXiv:2301.12726* (2023).

[31] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint arXiv: Arxiv-2101.00027* (2020).

[32] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[33] Elena L Grigorenko and Robert J Sternberg. 1998. Dynamic testing. *Psychological bulletin* 124, 1 (1998), 75.

[34] Xuanli He, Lingjuan Lyu, Lichao Sun, and Qiongkai Xu. 2021. Model Extraction and Adversarial Transferability, Your BERT is Vulnerable!. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.*

[35] Xuanli He, Qiongkai Xu, Lingjuan Lyu, Fangzhao Wu, and Chenguang Wang. 2022. Protecting Intellectual Property of Language Generation APIs with Lexical Watermark. *Proceedings of the AAAI Conference on Artificial Intelligence* 10 (2022). https://doi.org/10.1609/aaai.v36i10.21321

[36] 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).

[37] Namgyu Ho, Laura Schmid, and Se-Young Yun. 2022. Large Language Models Are Reasoning Teachers. *arXiv preprint arXiv:2212.10071* (2022).

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

[39] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. 2020. High accuracy and high fidelity extraction of neural networks. In *Proceedings of the 29th USENIX Conference on Security Symposium.*

[40] Akshita Jha and Chandan K Reddy. 2022. Codeattack: Code-based adversarial attacks for pre-trained programming language models. *arXiv preprint arXiv:2206.00052* (2022).

[41] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916* (2022).

[42] Andrew K. Lampinen, Ishita Dasgupta, Stephanie C. Y. Chan, Kory W. Mathewson, Mh Tessler, Antonia Creswell, James L. McClelland, Jane Wang, and Felix Hill. 2022. Can language models learn from explanations in context?. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics.

[43] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[44] Shiyang Li, Jianshu Chen, Yelong Shen, Zhiyu Chen, Xinlu Zhang, Zekun Li, Hong Wang, Jing Qian, Baolin Peng, Yi Mao, et al. 2022. Explanations from large language models make small reasoners better. *arXiv preprint arXiv:2210.06726* (2022).

[45] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022).

[46] Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. 2022. On the advance of making language models better reasoners. *arXiv preprint arXiv:2206.02336* (2022).

[47] 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.

[48] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. CCTEST: Testing and Repairing Code Completion Systems. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1238–1250.

[49] Zongjie Li, Chaozheng Wang, Shuai Wang, and Gao Cuiyun. 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 2023, Copenhagen, Denmark, November 26-30, 2023.*

[50] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What Makes Good In-Context Examples for GPT-3?. In *Proceedings of Deep Learning Inside Out: The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures, DeeLIO@ACL 2022, Dublin, Ireland and Online, May 27, 2022*, Eneko Agirre, Marianna Apidianaki, and Ivan Vulic

(Eds.). Association for Computational Linguistics.
[51] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* (2023).
[52] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *NeurIPS Datasets and Benchmarks 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.).
[53] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics.
[54] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *CoRR* abs/2306.08568 (2023).
[55] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2014. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects *(MSR 2014)*.
[56] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. 2136–2148.
[57] OpenAI. 2023. *Our approach to AI safety.* https://openai.com/blog/our-approach-to-ai-safety
[58] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2019. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition.*
[59] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *Proceedings of the 29th USENIX Conference on Security Symposium.* 181–198.
[60] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020).
[61] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
[62] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems* 33 (2020).
[63] Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. 2022. Multitask Prompted Training Enables Zero-Shot Task Generalization. (2022).
[64] Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *EMNLP 2020, Online, November 16-20, 2020*. Association for Computational Linguistics.
[65] Davide Spadini, Gül Çalikli, and Alberto Bacchelli. 2020. Primers or Reminders? The Effects of Existing Review Comments on Code Review *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA.
[66] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.
[67] Eric Wallace, Mitchell Stern, and Dawn Song. 2020. Imitation Attacks and Defenses for Black-box Machine Translation Systems. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics.
[68] Boshi Wang, Sewon Min, Xiang Deng, Jiaming Shen, You Wu, Luke Zettlemoyer, and Huan Sun. 2022. Towards Understanding Chain-of-Thought Prompting: An Empirical Study of What Matters. *arXiv preprint arXiv:2212.10001* (2022).
[69] 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.
[70] Jindong Wang, Xixu Hu, Wenxin Hou, Hao Chen, Runkai Zheng, Yidong Wang, Linyi Yang, Haojun Huang, Wei Ye, Xiubo Geng, et al. 2023. On the Robustness of ChatGPT: An Adversarial and Out-of-distribution Perspective. *arXiv preprint arXiv:2302.12095* (2023).
[71] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. 2022. Rationale-augmented ensembles in language models. *arXiv preprint arXiv:2207.00747* (2022).
[72] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023.*
[73] Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A Smith, Iz Beltagy, et al. 2023. How Far Can Camels Go? Exploring the State of Instruction Tuning on Open Resources. *arXiv preprint arXiv:2306.04751* (2023).
[74] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics.
[75] Albert Webson and Ellie Pavlick. 2022. Do Prompt-Based Models Really Understand the Meaning of Their Prompts?. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*. Association for Computational Linguistics.
[76] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems* 32 (2019).
[77] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).
[78] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).
[79] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839* (2023).
[80] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).
[81] 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).
[82] Qiongkai Xu, Xuanli He, Lingjuan Lyu, Lizhen Qu, and Gholamreza Haffari. 2021. Student Surpasses Teacher: Imitation Attack for Black-Box NLP APIs. *arXiv preprint arXiv:2108.13873* (2021).
[83] Guang Yang, Yu Zhou, Wenhua Yang, Tao Yue, Xiang Chen, and Taolue Chen. 2022. How Important are Good Method Names in Neural Code Generation? A Model Robustness Perspective. *arXiv preprint arXiv:2211.15844* (2022).
[84] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *International Conference on Mining Software Repositories (MSR)*. ACM. https://doi.org/10.1145/3196398.3196408
[85] Honggang Yu, Kaichen Yang, Teng Zhang, Yun-Yun Tsai, Tsung-Yi Ho, and Yier Jin. 2020. CloudLeak: Large-Scale Deep Learning Models Stealing Through Adversarial Examples.. In *NDSS.*
[86] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *arXiv preprint arXiv:2306.05685* (2023).
[87] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. *arXiv preprint arXiv:2303.17568* (2023).
[88] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.).
[89] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence. *arXiv preprint arXiv:2206.08474* (2022).
[90] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022*, Swarat Chaudhuri and Charles Sutton (Eds.).
[91] Andy Zou, Zifan Wang, J. Zico Kolter, and Matt Fredrikson. 2023. Universal and Transferable Adversarial Attacks on Aligned Language Models. arXiv:2307.15043 [cs.CL]