Transfer Attacks and Defenses for Large Language Models on Coding Tasks

CHI ZHANG, Carnegie Mellon University, USA
ZIFAN WANG, Center for AI Safety, USA
RAVI MANGAL, Carnegie Mellon University, USA
MATT FREDRIKSON, Carnegie Mellon University, USA
LIMIN JIA, Carnegie Mellon University, USA
CORINA PĂSĂREANU, Carnegie Mellon University, USA

Modern large language models (LLMs), such as ChatGPT, have demonstrated impressive capabilities for coding tasks including writing and reasoning about code. They improve upon previous neural network models of code, such as code2seq or seq2seq, that already demonstrated competitive results when performing tasks such as code summarization and identifying code vulnerabilities. However, these previous code models were shown vulnerable to adversarial examples, i.e. small syntactic perturbations that do not change the program's semantics, such as the inclusion of "dead code" through false conditions or the addition of inconsequential print statements, designed to "fool" the models. LLMs can also be vulnerable to the same adversarial perturbations but a detailed study on this concern has been lacking so far. In this paper we aim to investigate the effect of adversarial perturbations on coding tasks with LLMs. In particular, we study the transferability of adversarial examples, generated through white-box attacks on smaller code models, to LLMs. Furthermore, to make the LLMs more robust against such adversaries without incurring the cost of retraining, we propose prompt-based defenses that involve modifying the prompt to include additional information such as examples of adversarially perturbed code and explicit instructions for reversing adversarial perturbations. Our experiments show that adversarial examples obtained with a smaller code model are indeed transferable, weakening the LLMs' performance. The proposed defenses show promise in improving the model's resilience, paving the way to more robust defensive solutions for LLMs in code-related applications.

Additional Key Words and Phrases: Large Language Models (LLMs), Code Models, Adversarial Attacks, Robustness

1 INTRODUCTION

Modern large language models (LLMs), such as ChatGPT[1], have fundamentally changed the landscape of computational tasks. These LLMs, characterized by their ability to understand and generate human-like text across a wide range of topics, have demonstrated remarkable capabilities in coding tasks. They perform roles like writing code [Austin et al. 2021; Chen et al. 2021; Li et al. 2022b; Liu et al. 2023] and reasoning about its functionality [Pei et al. 2023; Xia et al. 2023], and in doing so, they go beyond the capabilities of existing neural network models that have already shown promise in tasks such as code summarization [Allamanis et al. 2016; Alon et al. 2019a,b] and identifying code vulnerabilities [Allamanis et al. 2018; Dinella et al. 2020; Pradel and Sen 2018; Vasic et al. 2019].

While these LLMs offer clear advancements, they also inherit vulnerabilities from their predecessors. One notable weakness is susceptibility to adversarial examples, a form of machine learning attack that has been well-documented in previous research [Goodfellow et al. 2015; Szegedy et al. 2014; Zou et al. 2023]. Adversarial examples consist of subtly altered inputs designed to mislead machine learning models without changing the underlying meaning of the data. In code-related

[1] https://chat.openai.com

Authors' addresses: Chi Zhang, chiz5@andrew.cmu.edu, Carnegie Mellon University, USA; Zifan Wang, zifan@safe.ai, Center for AI Safety, USA; Ravi Mangal, rmangal@andrew.cmu.edu, Carnegie Mellon University, USA; Matt Fredrikson, mfredrik@cmu.edu, Carnegie Mellon University, USA; Limin Jia, liminjia@andrew.cmu.edu, Carnegie Mellon University, USA; Corina Păsăreanu, pcorina@andrew.cmu.edu, Carnegie Mellon University, USA.

tasks, these could involve the insertion of *dead code* via false conditions or the addition of inconsequential print statements, both of which are aimed at leading the model astray [Allamanis et al. 2018; Gao et al. 2023; Srikant et al. 2021; Yefet et al. 2020].

Despite the increased utility and application of LLMs in coding tasks, research that systematically evaluates their vulnerability to these types of adversarial attacks is lacking, prompting the need for our study. We aim to evaluate LLMs' resilience, or lack thereof, to adversarial manipulations—particularly those attacks that have proven effective against smaller, specialized code models, and are therefore cheap to compute.

Through experiments, we show that adversarial examples obtained with a state-of-the-art attack technique for a smaller code model (seq2seq [Srikant et al. 2021; Sutskever et al. 2014]) are indeed transferable to multiple LLMs (GPT-3.5 and GPT-4 from OpenAI [OpenAI 2023], Claude-Instant-1 and Claude-2 from Anthropic [Anthropic 2023]), weakening the analyzed LLMs' performance.

To defend against such attacks and improve the resistance of the models against adversarial examples, we discuss post-hoc approaches, i.e., ones that do require retraining or fine-tuning the model. This is necessitated by the black-box nature of commercial LLMs. In particular, we propose novel *self-defense* strategies that leverage LLMs's own advanced capabilities of performing in-context learning [Brown et al. 2020] and understanding human instructions as well as code. The self-defense strategies involve modifying a manually crafted prompt to include additional information such as examples of adversarially perturbed code and explicit instructions for reversing adversarial perturbations. Our experimental evaluation of the proposed defenses indicates promise in improving the model's resilience, paving the way to more robust defensive solutions for LLMs in code-related applications.

We also present *meta-prompting* for leveraging the LLMs themselves to generate the self-defense prompts. The performance of LLMs can be very sensitive to the prompts used but, at the moment, prompt design is an empirical process lacking sound principles. We meta-prompt an LLM with examples of perturbed and corresponding unperturbed snippets, and ask the model to synthesize a prompt that can be used to unperturb the code using an LLM. Our experiments show that self-defense prompts generated via meta-prompting can be much more effective than the prompts that are manually crafted.

We summarize our contributions as follows: (1) We study the transferability of code attacks obtained based on a small code model to five commercial and open-source state-of-the-art LLMs. Through experimental results we find that all studied LLMs are susceptible to such attacks, with an attack success rate (ASR) of at least 21% observed on GPT-4. (2) We propose self-defense techniques against transfer attacks that leverage LLMs' own capabilities of performing in-context learning and understanding human instructions. This defense reduces the effects of the attack, achieving e.g., an ASR of 14% for GPT-4 and reduced rates for other models. (3) We also propose a meta-prompting technique that uses an LLM to generate its own defense prompt, further reducing the attack effectiveness, e.g., obtaining an ASR of 4% for GPT-4. (4) A secondary contribution of this work is a set of techniques aimed to customize LLMs for code summarization tasks.

2 BACKGROUND

2.1 Code Models: Modern LLMs vs. Previous Code Models

In this paper we make the distinction between modern, state-of-the-art LLMs, and previous, smaller specialized code models (which we denote as pre-LLM), for which known, practical attack methods exist. We distinguish them in terms of capabilities and sheer size. Modern, state-of-the-art LLMs, such as the proprietary GPT-4 [OpenAI 2023] and open-source Llama-2 [Touvron et al. 2023], generally have a transformer [Vaswani et al. 2017] architecture, capable of following

instructions, and are trained on general-purpose data, not only code. Smaller specialized models, such as code2vec [Alon et al. 2019b], code2seq [Alon et al. 2019a], or seq2seq [Alon et al. 2019a; Srikant et al. 2021], have various types of architectures and have been trained specifically for coding tasks. These models are much smaller than the modern LLMs (e.g., millions vs. billions of parameters), and are, therefore, easier to analyze. While most of existing LLMs are decoder-only, there are also encoder-based powerful models, such as BERT [Devlin et al. 2019] and code-specific CodeBERT [Feng et al. 2020], that, however, are still much smaller than the general purpose LLMs, and as such we would include them in the second category.

We aim to evaluate whether the attacks computed for the much smaller models, which are relatively cheap to compute, transfer to the much larger, general-purpose modern LLMs. Of course, an interesting question is how to compute attacks that are optimized directly on the LLMs. However, such attacks are very expensive in practice but we hope to explore this direction in future work.

2.2 Attacks on Pre-LLM Code Models

An adversarial example is a seemingly benign input with well-crafted tiny perturbations that maliciously cause a machine learning (ML) model to make a mistake, e.g. a wrong prediction. Adversarial perturbations can be very small (imperceptible to the human eye) in the case of perception models [Goodfellow et al. 2015; Szegedy et al. 2014]. Code models are susceptible to adversarial perturbations as well. Such attacks can have adverse consequences in settings such as security and compliance automation [Srikant et al. 2021]. An adversary can perturb malign programs in such a way that a classifier predicts them as `benign` (while they are actually malicious) or can make changes to pass off open-source code in a proprietary code base.

In this section, we review the attack proposed by Srikant et al. [2021] and reuse it in our studies. The attack leverages program obfuscations, which have traditionally been used to avoid attempts at reverse engineering programs, as adversarial perturbations. These perturbations modify programs in ways that do not alter their functionality (i.e., they are semantics preserving) yet they deceive an ML model when making a decision.

Specifically, Srikant et al. [2021] describes a way to generate adversarial programs for a pre-LLM code model, which is trained to solve the code summarization task, i.e., given the code corresponding to the body of a function, the model predicts the function name from a pre-defined set of names. These adversarial examples are generated by solving a constrained combinatorial optimization problem. Their method employs two kinds of semantics-preserving changes, i.e. *replace* and *insert*, to the original benign program in order to transform it into an adversarial example. Namely, *replace* changes include actions such as `renaming local variables`, `renaming function parameters`, `renaming object fields`, and `replacing boolean literals`. The *insert* changes involve steps such as `inserting print statements` and `adding dead code`. The attacker is assumed to have white-box access to the code model for the purpose of generating the adversarial inputs.

The authors address two key problems:

- *Site-Selection Problem*: This problem posits the question: given n sites in a program, if we are permitted to select at most k sites, which subset of k sites would exert the most significant impact on the performance of the downstream model?
- *Site-Perturbation Problem*: Once the k sites have been selected, the question arises as to which tokens should be inserted or replaced at these chosen locations.

```
( self application name = python gntp notifications = [ ] default notifications = none
    application icon = none hostname = localhost password = none port = 23053 ) : self .
    application name = application name self . notifications = list ( notifications ) if
    default notifications : self . default notifications = list ( default notifications )
    else : self . default notifications = self . notifications self . application icon =
    application icon self . password = password self . hostname = hostname self . port = int
    ( port )
```

Listing 1. Original Python code example. The snippet is missing indentations since the baseline seq2seq model consumes code in this form.

```
( self application name = python gntp notifications = [ ] default notifications = none
    application icon = none hostname = localhost password = none port = 23053 ) :    self .
    application name = application name    self . notifications = list ( notifications )
    if default notifications : if false : traverse = 1   self . default notifications = list
    ( default notifications )    else : self . default notifications = self . notifications
       self . application icon = application icon  print ( pspace )  self . password =
    password    self . hostname = hostname if false : validity = 1   self . port = int ( port
     ) if false : sl = 1  if false : evoked = 1
```

Listing 2. Adversarial Python code example. The sections highlighted in gray are the adversarial perturbations added to the code by the attack.

Generating an adversarial program is formulated as an optimization problem:

$$\begin{aligned} \underset{\mathbf{z},\mathbf{u}}{\text{minimize}} \quad & l_{\text{attack}}((\mathbf{1}-\mathbf{z})\cdot P + \mathbf{z}\cdot\mathbf{u}; P, \boldsymbol{\theta}) \\ \text{subject to} \quad & \mathbf{1}^T\cdot\mathbf{z} \leq k, \\ & \mathbf{z}\in\{0,1\}^n, \\ & \mathbf{1}^T\cdot\mathbf{u}_i = 1,\ \mathbf{u}_i \in \{0,1\}^{|\Omega|},\ \forall i \end{aligned} \tag{1}$$

Here $l_{\text{attack}}()$ represents the loss of the attack algorithm; the authors use cross-entropy loss in an untargeted setting. P represents a benign program, composed of a sequence of n tokens $\{P_i\}_{i=1}^n$; each $\{P_i\} \in \{0,1\}^{|\Omega|}$ is considered a one-hot vector of length $|\Omega|$, where Ω is a vocabulary of tokens. $\mathbf{z}$ is a vector of boolean variables indicating whether a site is selected for perturbation or not. $\mathbf{u}$ is a one-hot vector encoding the selection of a token from Ω, designated to be the insert/replace token for a selected transformation at a chosen site. $\boldsymbol{\theta}$ denotes the parameters of the given code model.

Due to the discrete nature of the problem, the loss landscape of the optimization problem in Equation 1 is not smooth. Srikant et al. [2021] further propose to use randomized smoothing [Duchi et al. 2012], that convolves the original loss landscape with a smooth function, to smoothen the overall loss landscape and ease the optimization problem, which improved their attack success rate. Through these methodologies, adversarial examples were generated for two code models based on the seq2seq [Sutskever et al. 2014] architecture. The underlying code models were trained on a dataset with 150K Python samples [Raychev et al. 2016a] and one with 700K Java samples [Alon et al. 2019a] respectively. In the rest of this paper, we only consider the model trained on Python samples. The authors generate perturbations only for the samples that the model had originally correctly predicted and evaluate the attack success rate. We utilize these adversarial examples in our experiments to evaluate the transferability of the attack from a pre-LLM to LLM code model; furthermore, we use these examples to evaluate the effectiveness of our prompt-based defenses.

Listing 1 shows an original Python code snippet from the clean dataset, while Listing 2 shows the corresponding adversarial example generated using the attack algorithm of Srikant et al. [2021]. This code snippet corresponds to the `__init__` function of a class used for sending messages

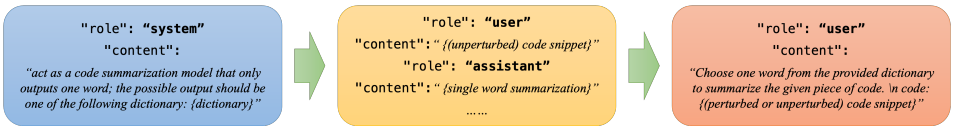

Fig. 1. LLM prompt for the code summarization task. The sections of the prompt represented by {·} are substituted with the corresponding values in the actual prompt. Prompts are structured as a conversation comprising a sequence of messages. Each message consists of a `role` identifying the speaker and `content` of the speaker's words. In this case, the prompt starts (leftmost box) with a **system** message that helps the model produce outputs in the desired format. Next (middle box), a series of messages between **user** and **assistant** with unperturbed code snippets and corresponding function names serve as few-shot examples of the desired behavior. Note that **assistant** refers to the model. Finally, the prompt ends (rightmost box) with the actual code snippet to be summarized.

using a transport protocol. In this case, we see that the adversarially perturbed version has a print statement and multiple branches with false condition inserted.

3 USING LARGE LANGUAGE MODELS FOR CODE SUMMARIZATION

This section introduces our use of modern LLMs to do code summarization before we delve into the implementation of the transfer attack to the task in the following sections.

The LLMs of interest in this work are *generative* models, taking a string of text, which is often referred to as a *prompt*, and generating a completion to the input prompt. Prompts often include instructions, e.g. *Outline a plan to organize a party at my home*, and LLMs are tuned to follow the instruction in the prompts to generate corresponding completions [Ouyang et al. 2022]. To use generative LLMs for code summarization, we prompt the LLMs to choose a single class from a list of potential classes that is most likely to describe the functionality of the program in the prompt. Figure 1 showcases our prompt templates for querying GPT-3.5 and GPT-4 models.

3.1 Preliminary Approach

As is described in Section 2, we take clean and adversarial data from Srikant et al. [2021] to evaluate the robustness of other LLMs. As our first step towards ensuring that the model's outputs are aligned with the desired output format (i.e., a single word function name from a pre-defined set of function names), we experiment with integrating the *dictionary*[2] of potential function names into the prompt. Our initial prompt takes the following form where {*dictionary*} is substituted by all possible function names and {*code*} by the function body under consideration (for instance, either Listing 1 or 2):

> *Use one word from the set {dictionary} to describe the following piece of code: {code}. Don't provide any other description. The reply form should be "word".*

Initially, we attempt to use a comprehensive dictionary containing all possible target function names (approximately 14,000 unique names for the dataset under consideration). However, this approach is problematic as the total length of the prompts exceeds the available context length of the model (4,096 tokens for GPT-3.5 which translates roughly to 3000 words), necessitating a modification in our strategy. To overcome the constraints, we revise our strategy by tailoring the dictionary to only contain 500 function names including the correct function name for the code being summarized. All the names, apart from the correct one, are randomly chosen from the set of all function names. We acknowledge that utilizing a smaller and more focused dictionary inherently

[2]We refer to a set of potential function names that the code summarization classifier can choose from as the *dictionary*.

simplifies the code summarization task. However, the goal of our experiments is to evaluate the susceptibility of LLMs to adversarial attacks rather than the inherent accuracy of the LLMs for code summarization. Any change that aids model accuracy, has the effect of reducing the model susceptibility to adversarial attacks. Therefore, the attack success rates we report in our evaluations should be considered a lower bound, with the actual situation likely to be worse.

Although the use of the dictionary in the prompt is helpful, the LLM demonstrated variability in adhering to the desired format, sporadically generating more extensive sentences, or indulging in a detailed analysis of each line of code, thus deviating from the expected succinct response format.

3.2 Conforming with Output Format via `System` Role

Faced with the challenge of the models not adhering to the required output format (i.e., single-word function names from a pre-determined set of names), we turned to the functionality of *roles* provided by LLMs. Roles are special words or tokens in the prompt that LLMs are trained to recognize as indicators of who is speaking, and they can have a large effect on how the model interprets the prompt. LLMs typically recognize at least three roles, namely, `System`, `User`, or `Assistant`, with each having a unique impact on shaping the conversation.

- `System`: This role, while optional, is crucial for setting up the model's behavior by providing high-level instructions or context. Instructions provided under this role tend to be treated as a higher privilege and hence, ones that need to be strongly followed.
- `User`: This role represents input prompts from the end-user or an application. This is the typical role used when providing prompts to the model in an interactive session.
- `Assistant`: This role signifies responses from the model. When engaged in a dialogue with the model, the entire history of the conversation is included in each prompt. In this history, the previous model responses are designated as being generated under this role, and therefore, it is essential for maintaining the continuity of dialogue.

To address the challenge of LLMs outputs not conforming to the desired output format, we strategically leveraged the `System` role to influence the model's behavior. By setting the following explicit instruction through this role, *act as a code summarization model that only outputs one word*, we achieved greater conformance with the desired output format compared with the use of the default `User` role.

We also explored the effect of providing the dictionary of function names to the model through the `System` role. Our experiments revealed that incorporating the dictionary through the `System` role yielded more structurally well-formed results than doing so through the `User` role. Based on this observation, we established the final prompt for the `System` role as (leftmost box in Figure 1):

act as a code summarization model that only outputs one word; the possible output should be one of the following dictionary: {dictionary}.

3.3 Improving Accuracy via Few-shot Prompting

Having designed a prompt that helps the model produce outputs in the format needed for the code summarization task, our next step was to figure out prompting strategies to improve the model's accuracy. For this, we employed prompt-based few-shot learning. Few-shot learning [Wang et al. 2020] refers to the notion of modifying the behavior of a machine learning model using very few examples, and this is typically achieved by fine-tuning the model parameters using a gradient descent based algorithm. However, a striking observation about the behavior of LLMs has been that they are able to function as few-shot learners without any updates to their parameters [Brown et al. 2020]. They can learn to perform new tasks by simply being exposed to a limited number of examples in the model prompt, negating the need for large datasets or extensive fine-tuning.

In our experiment, we utilized few-shot learning by introducing a series of messages between the `User` and the `Assistant` as examples in the prompt (see middle box of Figure 1). Each of these `User` messages included unperturbed code corresponding to the body of a function (for instance, Listing 1) while the `Assistant` messages included the associated function names. These examples were designed to prime the model for the task of code summarization. Employing this few-shot learning approach improved adaptability and accuracy in performing the code summarization task, enhancing the model's ability to generate accurate code summaries.

3.4 Assessing Model Uncertainty via Self-reported Probabilities

We experimented with the idea of asking the model to output its confidence along with the predicted function name. The confidence is a measure of the model's subjective probability of its prediction being correct. If confidence predictions of the model are well-calibrated, i.e., probability associated with the predicted function name reflects its ground truth correctness likelihood, then the predicted confidence can be used to decide if the predicted function name is likely to be a misclassification or not. This can be particularly useful for detecting adversarial perturbations. We also hoped that having the model output its confidence would aid in improving its accuracy. Recent empirical results suggesting that LLMs are indeed able to output well-calibrated confidence scores for various tasks [Kadavath et al. 2022; Tian et al. 2023] served as initial evidence for the validity of this approach. Accordingly, to evaluate this strategy, we constructed the following prompt:

> *Use one word from the set {dictionary} to describe the following piece of code: {code}. Also, output your confidence in your choice as a probability between 0 and 1. Don't provide any other description. The reply form should be "word (confidence)."*

This prompting strategy turns out to be unsuccessful. In practice, the confidences predicted by the model were not trustworthy. To evaluate the reliability of the model's confidence metrics, we executed the same prompt repeatedly using an exceptionally low temperature. In the context of LLMs, the temperature is a scaling factor applied to the token logits before converting them to token probabilities. An LLM produces the next output token by sampling from this distribution of next tokens. A lower temperature value makes the model's outputs more deterministic, emphasizing the most probable outcomes, whereas a higher value increases randomness, resulting in diverse outputs. By employing a low temperature, our objective was to curb the model's inherent randomness, leading to more deterministic outputs.

We initially hypothesized that, when the model is repeatedly invoked on the same input, if the predicted confidences are well-calibrated, outputs with higher predicted confidence values would manifest as consistent model responses while those with lower confidence would display higher variability. Contrary to our expectations, the model displayed surprising behavior. For a given fixed code snippet A, the model consistently produced the same function name across multiple runs, regardless of fluctuating predicted confidence levels. However, for another snippet B, the model's outputs varied significantly across trials, even though the expressed confidence levels remained comparably high. Such incongruities led us to infer that the model's confidence metrics were not dependable indicators of the likelihood of the output being correct and therefore, this prompting strategy was not viable.

3.5 Adding Abstain Option to Model Output

Although the model fails to predict its confidence score accurately, it might still be the case that the model is able to distinguish between situations where it is confident and where it is very uncertain. If this were true, it could suggest that in situations where the model is uncertain, the model might essentially be resorting to an arbitrary choice since it is compelled to provide an outcome. To give

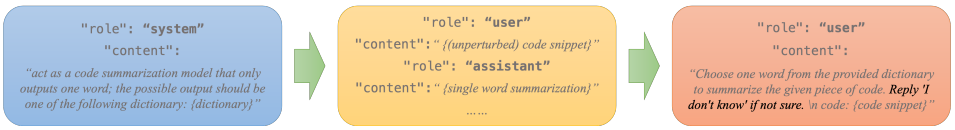

Fig. 2. LLM prompt to allow the model to abstain. The **bold** text in the rightmost box shows the text that was added to the prompt from Figure 1 to allow the model to abstain from making predictions.

the model an escape hatch in such cases of high uncertainty, we experimented with updating the prompt so that the model is allowed to abstain from predictions during moments of indecision, as shown in Figure 2. To this end, we empower the model to output *"I don't know"* whenever it encounters scenarios evoking doubt in its predictions. When exposed to adversarial examples, ideally, a model should respond *"I don't know"* rather than mis-classify them. This strategy turns out to be only partially successful, as we discuss in Section 6.

4 TRANSFER ATTACKS AND DEFENSES

4.1 Transferability of Adversarial Examples

The goal of this work is to assess the robustness of LLMs in performing the code summarization task when faced with malicious inputs designed to induce misclassification. To this end, we investigate the transferability of adversarial examples as crafted by a state-of-the-art white-box attack on the more specialized, smaller code model seq2seq [Alon et al. 2019a; Srikant et al. 2021]. The attack is described in Section 2.

For this assessment, we consider LLMs as classifiers assigned to the task of code summarization, as described in Section 3 and we expose them to the adversarial examples generated from the smaller model. As we will demonstrate in Section 6, our findings indicate that a significant proportion of adversarial examples, initially generated based on seq2seq models, are transferable to LLMs, thereby highlighting potential vulnerabilities in those large language models. In the rest of this section we discuss defenses against these transfer attacks.

4.2 Prompt-based Defenses for LLMs

Defenses against adversarial examples typically involve re-training the model [Madry et al. 2019] in an adversary-aware, which would be prohibitively expensive for LLMs. In fact, due to the black-box nature of proprietary LLMs, we do not have even access to the model weights to be able to re-train the model. We instead propose cost-effective defenses that are prompt-based, i.e., defenses that modify the prompt to defend against adversarial attacks. The motivation for such defenses is that LLMs have shown remarkable capability to understand natural language instructions and to perform in-context learning [Brown et al. 2020]. These capabilities, combined with their ability to understand code, can be leveraged to design prompts that instruct the model to reverse the semantics-preserving code perturbations that an adversary might apply before summarizing the code. For our prompt-based defenses, we assume that the capabilities of the adversary (i.e., the kinds of semantics-preserving perturbations they can apply) are known, and also that we have access to some adversarially perturbed code samples along with their correct function names. These are standard assumptions made by the machine learning community when designing defenses against adversarial examples.

We experiment with two different approaches: defense via few-shot examples (`FSD`) and defense via inverse transformation (`InvD`). In both cases, we design a prompt with defensive capabilities. Later, in Section 5, we present an approach that we refer to as *meta-prompting* where we ask the

Fig. 3. Prompt-based defense via few-shot examples (FSD). The **bold** text in the middle box shows the text that was added to the prompt from Figure 1 to help the model defend against adversarial attacks.

Fig. 4. Prompt-based defense via inverse transformation (InvD). The **bold** text in the rightmost box shows the text that was added to the prompt from Figure 1 to help the model defend against adversarial attacks.

LLM itself to generate a prompt that can help defend adversarial examples. Quite remarkably, it turns out that the LLM-generated defensive prompt can outperform the manually-generated prompts. However, in the rest of this section, we describe the two types of defensive prompts that we manually designed to improve the model's robustness to adversarial code perturbations.

*4.2.1 Defense via Few-Shot Examples (*FSD*).* Building upon the strategies discussed in Section 3.3, where we incorporate few-shot learning in LLMs for the code summarization task, in this subsection, we explore a similar approach as a defense mechanism. Specifically, in addition to few-shot examples that represent unperturbed code snippets and the corresponding function names, we also incorporate typical adversarial examples (i.e., perturbed code snippets) with the correct function names as few-shot examples in the model prompt (shown in Figure 3).

This approach is grounded in the hypothesis that exposing the model to adversarial examples during the few-shot learning phase can enhance its resilience against such inputs, allowing it to better generalize and correctly classify perturbed data. By providing the model with a more diverse set of examples, including both genuine and adversarial inputs, we aim to equip the LLMs with the necessary knowledge to recognize and counteract adversarial manipulations, thereby improving their robustness and reliability in the face of malicious attacks.

*4.2.2 Defense via Inverse Transformation (*InvD*).* In this subsection, we introduce a novel self-defense technique via prompting that is both cost-effective and broadly applicable. This strategy utilizes the inherent capabilities of LLMs to understand instructions. The method involves giving prompt instructions to the LLM to guide its behavior in a way that neutralizes the effects of the adversarial attack. Specifically, given the nature of code perturbations that the adversary can perform, the prompt includes instructions for undoing or inverting such perturbations. For example, prompts can be crafted to instruct LLMs to disregard or eliminate *dead code* and unnecessary print statements. This can also extend to standardizing variable names to a canonical form as a

countermeasure against variable renaming attacks. Figure 4 shows the prompt we designed for this purpose.

Two-step Defense. Inspired by the effectiveness of Chain-of-Thought (CoT) prompting [Wei et al. 2022], we initially explored a two-step approach for defense via inverse transformation. CoT prompting is a method for improving the capability of LLMs to perform complex reasoning tasks by breaking them down into simpler sub-problems. The method involves asking the LLM to generate intermediate steps of reasoning and feeding back these LLM-generated intermediate steps into the prompt for additional context. In our case, we broke down the code summarization task into two steps. The first step entailed asking the model to invert the perturbed code to its original, unaltered state. In the second step, the model was asked to summarize this reconstituted code to produce the final classification result. Each step was guided by specialized prompts designed to direct the LLM in accomplishing these tasks. During the first step of unperturbing the code, we carefully evaluated the capability of our prompts in guiding the LLM to accurately restore the original code from the adversarially perturbed version. If the prompt was found to be ineffective, it was fine-tuned iteratively to achieve better performance.

We observed that for instances of adversarially perturbed data, where the unperturbed version is correctly labeled by the LLM, the effect of the two-step inverse transformation can be categorized into three cases: **Full Success Case**, **Partial Success Case**, and **Failure Case**.

- **Full Success Case:** In this case, the prompt defense based on inverse transformation proves effective both in correctly unperturbing the code and in ensuring that the model assigns the correct function name to the code snippet. This outcome represents the ideal scenario where the prompt-based defense robustly counters adversarial attacks.
- **Partial Success Case:** In this case, while the first step of the defense effectively undoes adversarial perturbations and retrieves the original source code, the model is still unable to correctly classify the code snippet, i.e., the two-step defense is not effective in ensuring correct classification. This points to a potential need for further optimization of the prompt used for the second step to harmoniously integrate the code transformation and classification steps.
- **Failure Case:** In this category, the prompt-based defense struggles to even revert the perturbed code to the original unperturbed version, and, as a consequence, the code snippet is not labeled accurately by the classifier.

The first scenario represents the ideal case, demonstrating that prompt-based defenses can indeed safeguard against adversarial examples. The second scenario highlights the need for refining our approach to identify a more synergistic prompt that can merge the two steps effectively. However, the third step also demonstrates the perils of this approach—the LLM cannot always successfully transform the code back to the original version even if the nature of code perturbations that the adversary can potentially perform is included as a part of the prompt.

Listings 3 and 4 show the responses of GPT-3.5 and GPT-4, respectively, after the first step of this two-step defense. We see that GPT-4 is able to remove all the perturbations added by the adversary whereas GPT-3.5 fails to remove one of the code modifications (highlighted in gray). To invert the perturbed code, the models were given the following prompt with *{code snippet}* replaced by the perturbed code in Listing 2:

Remove the if false statement and the print statement in the code before the summarization.\n code: {code snippet}

Single-step Defense. Following the initial exploration with the two-step defense, we later refined our strategy by consolidating it into a single comprehensive prompt. A single-step approach is

```
code: ( self application name = python gntp notifications = [ ] default notifications = none
    application icon = none hostname = localhost password = none port = 23053 ) :    self .
    application name = application name    self . notifications = list ( notifications )
    if default notifications : traverse = 1   self . default notifications = list ( default
    notifications )    else : self . default notifications = self . notifications    self .
    application icon = application icon    self . password = password    self . hostname =
    hostname    self . port = int ( port )
```

Listing 3. GPT-3.5 generated response based on crafted prompt for InvD defense.

```
code: ( self application name = python gntp notifications = [ ] default notifications = none
    application icon = none hostname = localhost password = none port = 23053 ) :    self .
    application name = application name    self . notifications = list ( notifications )
    if default notifications : self . default notifications = list ( default notifications )
       else : self . default notifications = self . notifications    self . application icon
    = application icon    self . password = password    self . hostname = hostname    self .
    port = int ( port )
```

Listing 4. GPT-4 generated response based on crafted prompt for InvD defense.

more cost-effective since it requires fewer queries to the LLM. The unified prompt directs the LLM to, both, invert the adversarial code transformations with the hope of recovering the original code and producing the final classification result in a single pass. In other words, LLM is now guided first to recover the original code and then directly output only the final classification result, all under the guidance of this integrated prompt. The model never explicitly outputs the inverted code. Figure 4 shows the prompt we used for this single-step approach. As stated earlier, this consolidation aims to improve cost effectiveness while maintaining the robustness of our defensive strategy. In Section 6, we present results demonstrating the effectiveness of the single-step inverse transformation defense.

5 META-PROMPTING: LLM-GENERATED DEFENSIVE PROMPTS

The effectiveness of our prompt-based defenses hinges on the the quality of the prompts used to instruct the LLMs to neutralize known adversarial attacks. Leveraging the generative capabilities of LLMs we experimented with a technique that we call *meta-prompting*, i.e., we instructed an LLM, namely GPT-4, to generate the prompt defense by itself. As it turns out, the LLM generated prompts seem much more effective than the manually engineered prompts in defending against adversarial attacks.

Meta-prompting is achieved by feeding the model higher-level instructions or templates, along with examples of both original (unperturbed) and perturbed code. The LLM, utilizing these inputs, generates effective prompts tailored to the specific nature of the perturbations and the desired outcome. We experimented with two techniques, as illustrated in Figures 5 and 6.

(1) **Example-Based Meta-Prompting (EBMP):** This technique involves providing the LLM with a couple of examples of original and corresponding perturbed code, without any explicit insight into the nature of the perturbations. The LLM is instructed to analyze these examples and generate prompts that could effectively guide the inverse transformation process. This method relies on the LLM's own capabilities to deduce the necessary transformations. The generated prompt is as follows:

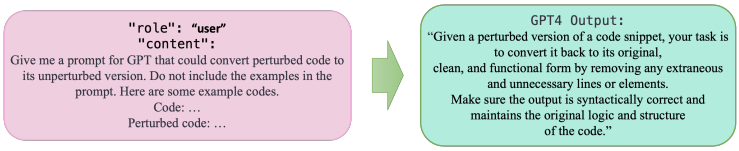

Fig. 5. Example-based meta-prompting

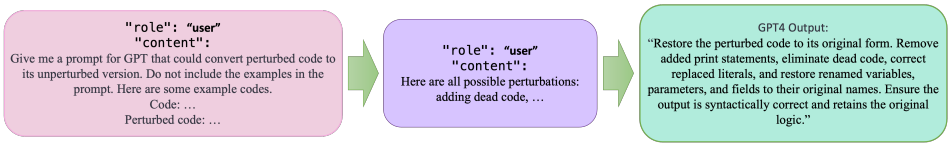

Fig. 6. Perturbation-aware meta-prompting

> *"Given a perturbed version of a code snippet, your task is to convert it back to its original, clean, and functional form by removing any extraneous and unnecessary lines or elements. Make sure the output is syntactically correct and maintains the original logic and structure of the code."*

(2) **Perturbation-Aware Meta-Prompting (PAMP):** The second technique assumes some knowledge about the possible perturbations. This allows the LLM to create a specialized prompt that is more aligned with the specific code transformations employed by an adversary. This method is anticipated to be more effective but requires prior knowledge or assumptions about the attacker's perturbations. The generated prompt is as follows:

> *"Restore the perturbed code to its original form. Remove added print statements, eliminate dead code, correct replaced literals, and restore renamed variables, parameters, and fields to their original names. Ensure the output is syntactically correct and retains the original logic."*

These LLM-generated prompts are incorporated into our prompt-based defense via inverse transformation (Section 4.2.2 and Figure 4). In particular, the following instruction in Figure 4

> *"Remove the if false statement and the print statement in the code before the summarization."*

is replaced by the instruction *"Before summarization, {LLM-generated prompt}."* where *{LLM-generated prompt}* is substituted by the prompt generated via meta-prompting. The other parts of the prompt from Figure 4 remain unchanged.

Note the LLM-generated prompts include additional instructions that were not included in the manually crafted prompt either by design or due to inadvertent omission. In particular, the last sentences in both the prompts instructing that the output should be syntactically correct and retain the original logic of the code are, in hindsight, useful instructions that should have been included in the manually crafted prompt as well. We also note that the perturbation-aware nature of PAMP leads it to generate a prompt that includes even more suggestions for reverting the code to its original form than in our hand-crafted prompt. In this case, however, the omissions are by design—a manually crafted prompt that instructed the model to rename variables and parameters was less effective than our final defensive prompt presented in Figure 4.

5.1 Example Comparing GPT-3.5 vs. GPT-4 using the Generated Prompts

We discuss here the effect on issuing the meta-generated prompts to the GPT-3.5 and GPT-4 models on the perturbed example shown in Listings 2 that was generated by adverarially perturbing

```
self.application_name = application_name
self.notifications = list(notifications)
if default_notifications:
    traverse = 1
    self.default_notifications = list(
        default_notifications)
else:
    self.default_notifications = self.
        notifications
self.application_icon = application_icon
print(pspace)
self.password = password
self.hostname = hostname
validity = 1
self.port = int(port)
sl = 1
evoked = 1
```

Listing 5. GPT-3.5 generated response via EBMP.

```
class Application:
    def __init__(self, application_name,
         notifications,
         default_notifications,
         application_icon, hostname,
         password, port):
        self.application_name =
             application_name
        self.notifications = list(
             notificatiions)
        if default_notifications:
            self.default_notifications =
                 list(
                 default_notifications)
        else:
            self.default_notifications =
                 self.notifications
        self.application_icon =
             application_icon
        self.password = password
        self.hostname = hostname
        self.port = int(port)
```

Listing 6. GPT-3.5 generated response via PAMP.

````
The cleaned code should look like this:

```python
class MyClass:
    def __init__(self, application_name,
         notifications=[],
         default_notifications=None,
         application_icon=None, hostname='
         localhost', password=None, port
         =23053):
        self.application_name =
             application_name
        self.notifications = list(
             notifications)
        if default_notifications:
            self.default_notifications =
                 list(
                 default_notifications)
        else:
            self.default_notifications =
                 self.notifications
        self.application_icon =
             application_icon
        self.password = password
        self.hostname = hostname
        self.port = int(port)
```

I assumed that this code is a part of a
     class, because it seems to be a
     constructor method (it initializes
     instance variables). I also removed
     all the unnecessary lines and
     elements.
````

Listing 7. GPT-4 generated response via EBMP.

````
The restored code should look like this:

```python
def __init__(self, application_name='
     python_gntp', notifications=[],
     default_notifications=None,
     application_icon=None, hostname='
     localhost', password=None, port
     =23053):
    self.application_name =
         application_name
    self.notifications = list(
         notifications)
    if default_notifications:
        self.default_notifications = list(
             default_notifications)
    else:
        self.default_notifications = self.
             notifications
    self.application_icon =
         application_icon
    self.password = password
    self.hostname = hostname
    self.port = int(port)
```

In the restored code, I have removed all
     the unnecessary print statements and
     dead code (code that can never be
     executed). I have also corrected the
     function definition line to make it a
      valid Python function. The
     parameters have been renamed to their
      original names and the function now
     has a name (`__init__`). The literals
      have been corrected to their
     original values.
````

Listing 8. GPT-4 generated response via PAMP.

a randomly chosen sample (shown in Listing 1) from our dataset. The effect of applying the prompts generated via Example-Based Meta-Prompting and the Perturbation-Aware Meta-Prompting to the

adversarial example is illustrated in Listings 5, 6 (for GPT-3.5) and Listings 7, 8 (for GPT-4). Note that these results are generated by directly issuing the prompts shown in the rightmost boxes of Figures 5 and 6.

We make a number of observations. In all the cases here, GPT-3.5 and GPT-4 are able to infer the correct indentation for the code snippet even though the meta-generated prompts do not include any such instruction. Moreover, except for the GPT-3.5 and EBMP combination, the models are even able to infer a correct name for the function (`__init__`). In some cases (Listings 6 and 7), the model also invents class names. We see that the combination of GPT-3.5 and EBMP fails to remove some of the adversarial perturbations in the code. GPT-3.5 also fails to pass correct default values to the function parameters whereas GPT-4 is better at this, particularly when using the prompt generated via PAMP. GPT-4 also tends to produce explanations for its actions. Overall, this simple exercise suggests that the combination of GPT-4 and the PAMP generated prompt may lead to the best results, as one would expect, and this is indeed what we observe in our experiments. However, we note that although all these responses are generated with the temperature parameter set to 0, the generation process still exhibits some randomness, and also the behavior of the models on a different example might be different.

6 EXPERIMENTS

We report on our experiments that aim to answer the following research questions:

- RQ1: Do code attacks transfer from a small model to LLMs?
- RQ2: Do the manually-engineered prompt defenses succeed against attacks?
- RQ3: Does meta-prompting help?
- RQ4: How does the abstain option influence model accuracy, transfer of attacks, and defense?

6.1 Dataset

In Section 2.2, we described the methodology for generating the adversarial dataset according to Srikant et al. [2021]. We started with a dataset for the code summarization task such that given a function body as input, the output should be a single-word label representing the function name. The programming language used in this dataset is Python. We generated adversarial examples with respect to a small code model (based on the seq2seq [Sutskever et al. 2014] architecture) trained on the code summarization task by applying semantics-preserving perturbations to the input code snippets as proposed in the white-box attack of Srikant et al. [2021]. As highlighted in Section 2.2, the attack algorithm solves a joint optimization problem to address the site selection and site perturbation problems. We used the Randomized Smoothing [Duchi et al. 2012] option to ease the optimization problem and improve the generation of adversarial programs, as suggested by Srikant et al. [2021]. One parameter that needs to be set for the adversarial generation process is the perturbation strength, denoted as k. In our experiments, we set $k = 5$, implying that an attacker can perturb a maximum of five sites within a given program. This constraint ensures that the adversarial programs remain close to the original while still maintaining their adversarial intent. Though the original attack work by Srikant et al. [2021] reported results on 2800 samples (out of a Python dataset of 150K programs [Raychev et al. 2016b]) whose unperturbed versions are correctly labeled by the small model (seq2seq),we randomly sampled 1000 examples from this set for our experiments; this is due to the significant computational resources required to run experiments with state-of-the-art LLMs. We use S to denote this dataset of 1000 unperturbed samples.

6.2 Models

As mentioned, we use a small code model (with 48M parameters) trained on the code summarization task that has a seq2seq [Sutskever et al. 2014] architecture to generate the attacks. This **seq2seq** model consists of two main components: an encoder that processes the input sequence and compresses it into a fixed-length vector (context), and a decoder that generates an output sequence based on this context. We evaluate the attack generated from the seq2seq model on the proprietary and open-source LLMs that are described below. We set temperature to 0 for all our experiments to control the randomness of the models.

Proprietary Models. We evaluate state-of-the-art proprietary LLMs, including **GPT-3.5** (`gpt-3.5-turbo-0613`) and **GPT-4** (`gpt-4-0613`) from OpenAI [OpenAI 2023], and **Claude-Instant-1** (`claude-instant-1`) and **Claude-2** from Anthropic [Anthropic 2023], on the generated attacks. These models are amongst the top-5 entries on a public leaderboard[3] that tracks the performance of LLMs on multiple tasks. GPT-3.5 and GPT-4 have been shown to possess remarkable abilities on various human tasks, including code understanding and code co-piloting. On the other hand, Claude models are able to process long sequences (up to 100k tokens) and have been shown as the most robust models under a state-of-the-art attack [Zou et al. 2023] that aims to elicit harmful knowledge from the models. In our experiments, we focus on their ability to perform code understanding and assess their response in the presence of adversarial code examples.

Open-Source Models. We also evaluate **CodeLlama** [Rozière et al. 2023], a state-of-the-art open-source LLM trained to follow instructions related to coding problems. CodeLlama is fine-tuned from the weights of Llama-2 [Touvron et al. 2023], an LLM released by Meta in early July 2023. While Llama-2 is developed to follow general instructions, CodeLlama is further fine-tuned on code datasets so is more capable of understanding code compared to the original Llama-2. There are three different sizes of CodeLlama (i.e. with 7B, 13B and 34B parameters) and two tuning options (i.e. with or without instruction-following tuning to understand natural language inputs). In our experiment, we use `CodeLlama-7B-Instruct`, which has 7B parameters and is tuned to answer coding questions written in natural language, e.g. summarizing the purpose of a given function.

6.3 Notations

We use S to denote the set of 1000 clean inputs that are correctly classified by the baseline seq2seq model. We use S_M to denote the subset of S that each model M correctly classifies. For instance, when M =GPT-3.5, S_M denotes the subset of S that GPT-3.5 classifies correctly. When M =seq2seq, $S_M = S$. We compute S_M in this way to allow for a fair comparison among the models, as Srikant et al. [2021] apply adversarial code perturbations only to the data that is classified correctly by the model. We use S_M^{adv} to denote the set of adversarial inputs obtained by applying the perturbations on inputs in S_M; note that the sizes of S_M and S_M^{adv} are the same, i.e., $|S_M| = |S_M^{adv}|$. For some dataset X, we use notations $Correct(X)$, $Wrong(X)$, $Abstain(X)$ to denote the number of instances in X for which the output of a model is correctly classified, mis-classified, or *I don't know*, respectively, where the model is implicit from the context.

6.4 RQ1: Do code attacks transfer from a small model to LLMs?

To investigate the transferability of adversarial attacks from our baseline model, seq2seq, we ran the various LLMs on the adversarial dataset, as well as on the corresponding clean, unperturbed dataset, for comparison; the results are reported in Table 1. The table contrasts the accuracy of

[3] https://chat.lmsys.org (Accessed in November, 2023)

Table 1. Transferability of the attack across different models.

Model (M)	**Acc on S** ($Correct(S)/|S|$)	**ASR** ($Wrong(S_M^{adv})/|S_M|$)
seq2seq	100	44.76
GPT-3.5	60.7	29.49
GPT-4	67	21.04
Claude-Instant-1	47.4	25.11
Claude-2	59.6	22.99
CodeLlama	9	53.33

each model on the clean dataset, S, with the Attack Success Rate (ASR) [Srikant et al. 2021], which are defined below.

- Given a model M and a dataset S, the accuracy **Acc on** S is defined as $Correct(S)/|S|$.
- Given a model M and dataset S_M of clean inputs that are correctly classfied by M, the **Attack Success Rate (ASR)** is defined as $Wrong(S_M^{adv})/|S_M|$.

S, as discussed, is a Python dataset with 1000 clean inputs (sampled from a dataset with a 150K programs) that are correctly classified by the seq2seq model and on which the attack was applied. Therefore, the accuracy of the seq2seq model on S is 100%. However, the overall accuracy of this seq2seq model on the entire dataset of Python programs is quite low, around 25%.

From our data, the seq2seq model has an ASR of 44.76%. The attack was able to deceive the seq2seq model for nearly half of the adversarial samples effectively. The results indicate that the attacks from the baseline model, seq2seq, exhibit varying degrees of transferability across different LLMs. For instance, GPT-3.5, which has an accuracy of 60.7% on S, presents an ASR of 29.49%, suggesting a notably large transferability on this attack. GPT-4 has the highest accuracy of 67% and the lowest ASR at 21.04%, which is still significant. The Claude models, Claude-Instant-1 and Claude-2, with the accuracy of 47.4% and 59.6%, respectively, display ASRs of 25.11% and 22.99%, indicating that they are also quite vulnerable to attacks.

Different from other LLMs, CodeLlama displays the lowest accuracy of 9% and shows the highest ASR of 53.33%, indicating a pronounced susceptibility to attacks from the seq2seq model. Generally, CodeLlama underperforms compared to other models. We believe this is primarily due to its tendency to generate lengthy explanations of the input code snippet rather than succinctly summarizing it with a single keyword. Although we assess CodeLlama's output based on the inclusion of target words, its performance still falls short of other LLMs in terms of output quality and precision. Therefore, the results that we report on this model throughout the paper need to be taken with a grain of salt, as the model was not well-suited for the code summarization task.

In summary, the attacks transfer from smaller models such as seq2seq to all the larger models with significant attack success rates. These results indicate that one can effectively attack these black-box, very large, performant LLMs by simply generating adversarial examples based on much smaller models. This makes it significantly more feasible for attackers to succeed.

6.5 RQ2: Do the manually-engineered prompt defenses succeed against attacks?

To evaluate the proposed defenses, `FSD` (Defense via Few-Shot Examples, Section 4.2.1) and `InvD` (Defense via Inverse Transformation, Section 4.2.2), we first compute the accuracy of the model with the defense on clean, unperturbed inputs. Specifically, we measure the percentage of clean inputs (out of S_M for each model M) that remain correctly labeled after the application of the defense (denoted as accuracies `FSD` **Acc on** S_M and `InvD` **Acc on** S_M in the table). This is to determine if

Table 2. Effectiveness of the defenses across different models.

Model (M)	**FSD Acc on S_M** $(Correct(S_M)/\|S_M\|)$	**FSD ASR** $(Wrong(S_M^{adv})/\|S_M\|)$	**InvD Acc on S_M** $(Correct(S_M)/\|S_M\|)$	**InvD ASR** $(Wrong(S_M^{adv})/\|S_M\|)$
GPT-3.5	93.57	30.15	92.59	24.71
GPT-4	91.94	14.63	91.79	16.57
Claude-Instant-1	90.51	9.70	79.75	19.20
Claude-2	86.58	10.91	82.38	20.97
CodeLlama	74.44	66.67	50	74.44

Table 3. Evaluating InvD defense with LLM-generated prompts

Model(M)	**EBMP + InvD**		**PAMP + InvD**	
	Acc on S_M $(Correct(S_M)/\|S_M\|)$	**ASR** $(Wrong(S_M^{adv})/\|S_M\|)$	**Acc on S_M** $(Correct(S_M)/\|S_M\|)$	**ASR** $(Wrong(S_M^{adv})/\|S_M\|)$
GPT-3.5	89.71	10.48	94.3	6.07
GPT-4	95.4	4.3	94.96	3.86

the defense does not unintentionally fool the model into misclassifying clean inputs that would otherwise be correctly classified by the model without any defense.

To answer the main question of defense effectiveness against attacks, we also compare the Attack Success Rate (ASR) for the models without defenses against their ASR when these defenses are applied. The results are presented in Table 2.

We first note that the accuracy (Acc on S_M) remains high when the defenses are applied, indicating that defenses do not accidentally lead to too many misclassifications of clean inputs; the outlier is again CodeLlama.

In terms of defense effectiveness, we note that for the GPT-3.5 model, the ASR reduces from an original 29.49% (computed on the model without defense; see Table 1) to 24.71% using the InvD defense, indicating its effectiveness. However, the FSD defense seems not to be effective, as the ASR is approximately unchanged (in fact, slightly larger). In the context of the GPT-4 model, both defenses curtail the ASR from the baseline 21.04% (model without defense; see Table 1), with FSD exhibiting superior performance (ASR 14.63%). The Claude-Instant-1 and Claude-2 models both benefit from the defenses, with ASRs decreasing from their initial values of 25.11% and 22.99%, respectively on the models without defense, to FSD ASR of 9.7% and 10.91%; again, the FSD defense is more effective. For the CodeLlama model, while the ASR diminishes from an initial 53.33% with the FSD defense, the InvD defense markedly amplifies it.

In summary, the FSD defense appears to generally improve the models' resilience against adversarial attacks. However, the effectiveness of the InvD defense is less clear and appears to be model-dependent.

6.6 RQ3: Does meta-prompting help?

We evaluate the integration of the two novel prompts, generated via Example-Based Meta-Prompting (EBMP) and Perturbation-Aware Meta-Prompting (PAMP), with the InvD method. Table 3 shows our results when using GPT-3.5 and GPT-4. We report the same metrics as in Section 6.5 where we evaluated the effectiveness of the manually constructed prompt-based defenses.

We see that the effect of these defenses on the clean code samples (reported in columns **Acc on S_M**) is similar to the results observed for the InvD defense with manually generated prompts.

Table 4. Effect of abstain option on transfer of attack.

Model (M)	Experiment 1: Clean Samples		Experiment 2: Adversarial Samples	
	Abs on S ($Abstain(S)/\|S\|$)	**Acc on** S ($Correct(S)/\|S\|$)	**Abs on** S_M^{adv} ($Abstain(S_M^{adv})/\|S_M\|$)	**ASR** ($Wrong(S_M^{adv})/\|S_M\|$)
GPT-3.5	27.8	40.6	83.20	6.43
GPT-4	0	64.5	0	22.09
Claude-Instant-1	1	44.1	6.33	21.10
Claude-2	2.3	58.1	8.05	19.13
CodeLlama	59.9	6.5	91.11	1.11

For instance, GPT-3.5 with a PAMP-generated prompt has an accuracy of 94.3% on S_M compared to 92.59% with the manually generated prompt. Similarly, GPT-4 with PAMP-generated prompt has an accuracy of 94.96% on S_M compared to 91.79% with the manually generated prompt. These results suggest that EBMP and PAMP generated prompts maintain high accuracy in labeling non-adversarial examples while implementing defensive measures.

More interestingly, there is a significant reduction in the Attack Success Rate (ASR) when employing these techniques, indicating that the LLM generated prompts can provide a strong defense against adversarial attacks. Notably, using the EBMP generated prompt, all of the samples whose clean code versions are correctly labeled continue to be correctly labeled for their corresponding adversarially perturbed code versions, i.e., the attack fails to have any effect. This implies a near-elimination of the effects of adversarial perturbations which we find particularly striking.

PAMP-generated prompts lead to the lowest ASR—6.07% for GPT-3.5 and 3.86% for GPT-4, compared to 24.71% and 16.57%, respectively, when using the manually-crafted prompt. This improved efficacy can be attributed to PAMP's design, which incorporates information about potential perturbations an attacker might employ. The observation that the combination of PAMP and GPT-4 leads to the best results is in sync with the effects on the example code snippet that we discussed in Section 5.1.

Discussion. We find it intriguing that LLM-generated prompts are more effective than the manually crafted prompts at defending the model from adversarial perturbations. In terms of the actual content of the prompt, the LLM-generated prompts are almost identical to the manual prompts except for some additional instructions, in particular, the instruction to ensure that the output is syntactically correct and retains the original logic that appears both in the EBMP and PAMP generated prompts. We hypothesize that the effectiveness of meta-prompting is an instance of a generally observed surprising capability of LLMs to generate prompts that are more effective that ones crafted by humans [Pryzant et al. 2023; Zhou et al. 2023].

Note that due to resource constraints, we were only able to evaluate the GPT-3.5 and GPT-4 models. Further evaluation with the other LLMs is left for future work.

6.7 RQ4: How does the abstain option influence model accuracy, transfer of attacks, and defense?

To answer this research question, we designed two experiments on the LLMs, allowing the model to abstain from giving a summarization word when the model is not sure about its answer as described in Section 3.5.

- **(Experiment 1:)** In the first experiment, we evaluate each LLM with the abstain option on the clean dataset S to determine how often the model abstains on clean inputs.

Table 5. Effect of abstain option for the defenses.

Model	FSD Abs on S_M^{adv} $(Abstain(S_M^{adv})/\|S_M\|)$	FSD ASR $(Wrong(S_M^{adv})/\|S_M\|)$	InvD Abs on S_M^{adv} $(Abstain(S_M^{adv})/\|S_M\|)$	InvD ASR $(Wrong(S_M^{adv})/\|S_M\|)$
GPT-3.5	59.14	14.00	73.97	6.59
GPT-4	0	15.52	0.29	17.16
Claude-Instant-1	0.42	11.81	0	23
Claude-2	1.01	18.12	2.01	23.49
CodeLlama	97.78	1.11	81.11	13.33

- **(Experiment 2:)** In the second experiment, we evaluate each LLM with the abstain option on the perturbed inputs from dataset S_M. Here, S_M is the same as before, i.e., the subset of S that the LLM M (without abstain option) correctly labeled. This allows us to evaluate the effectiveness of the attack (measured by ASR) in the presence of the abstain option and compare it with the ASR computed for the models without the abstain option (Table 1).

Table 4 displays the results of these experiments. We use **Abs** on various datasets to denote **abstain rates** which are meant to measure how often the models abstain when classifying inputs from the datasets. **Acc** is accuracy and **ASR** is attack success rate.

The results indicate that, as desired, ASR decreases for all models (when compared to ASR for the models without abstain option; see Table 1); one exception is GPT-4. Furthermore, for GPT-3.5 and CodeLlama the abstain rate is high on adversarial inputs, indicating that the approach could be potentially useful for detecting adversarial inputs on those models. However, the drawback is that the accuracy decreases for all the models except for GPT-4, and the abstain rate may be too high on clean inputs (as in the case of GPT-3.5 and CodeLlama).

The abstain mechanism offers some defense by itself (as it can flag adversarial inputs as *I don't know*), which is orthogonal to the other defenses (`FSD` and `InvD`). We also experimented with combining the abstain option with `FSD` and `InvD` defenses. The results are displayed in Table 5.

The experiments are inconclusive in the sense that it is not clear that the combined defenses lead to lower ASR (as compared to results in Table 2 obtained on defended models without abstain option). One interesting observation is that the defenses tend to reduce the confusion of the models (evidenced by lower abstain rates). This is as expected, as the two defenses `FSD` and `InvD` aim to revert obfuscating transformations.

While it is unclear that the abstain option is useful for detecting and defending against adversarial attacks, our results indicate that it can potentially be used to detect the confusion of the model. This, in turn, may be useful in other applications, such as the detection of out-of-distribution inputs, that we plan to explore.

7 RELATED WORK

Deep neural networks (DNNs) have been shown to obtain state-of-the art performance for a variety of tasks, including image classification [Krizhevsky et al. 2012] and natural language processing [Devlin et al. 2019]. DNNs have also found success in solving programming language tasks, such as code summarization [Allamanis et al. 2016; Alon et al. 2019a,b; LeClair et al. 2020], bug prediction [Allamanis et al. 2018; Pradel and Sen 2018; Vasic et al. 2019], or program repair [Dinella et al. 2020; Xia et al. 2023]. Popular pre-LLM models include code2vec [Alon et al. 2019b], code2seq [Alon et al. 2019a], and CodeBERT [Feng et al. 2020].

However it is known that even highly performant neural networks are vulnerable to adversarial attacks [Goodfellow et al. 2015; Szegedy et al. 2014], i.e., small perturbations to an input designed

to change correct predictions of state-of-the-art DNNs. In natural language processing, several techniques have been proposed to generate adversarial examples in a black-box [Alzantot et al. 2018] or white-box [Ebrahimi et al. 2018] manner.

For code models, there are a few works that study the vulnerability of code models to adversarial examples, such as the inclusion of "dead code" through false conditions or the addition of inconsequential print statements, designed to "fool" the models [Allamanis et al. 2018; Gao et al. 2023; Srikant et al. 2021; Yefet et al. 2020]. However none of the previous works study the vulnerability of the much larger LLMs used in coding tasks to the same adversarial perturbations.

Recent work [Zou et al. 2023] describes a gradient-based attack to LLMs (but not specifically for coding tasks). The goal of the attack is to find a suffix to potentially harmful user prompts, e.g., "How to make a pipeline bomb", so the combined prompt would break typical LLM alignment filters. The work shows the adversarial suffixes found by the attack on open-source LLMs transfer well to commercial models like ChatGPT and Claude. In contrast we study LLMs specifically for coding tasks and show that attacks obtained on non-LLM models (such as seq2seq or code2seq) also transfer to LLMs. These attacks are much cheaper to compute, since the corresponding code models are much smaller than the LLMs.

Classical defense approaches involve adversarial training, i.e., training using perturbed inputs [Bielik and Vechev 2020; Li et al. 2022a; Madry et al. 2019; Ramakrishnan et al. 2020]. Such defenses remain challenging, as they often reduce model accuracy. To perform adversarial training for LLMs one can explore fine-tuning the pre-trained large models, as allowed by current APIs. However adversarial training for LLMs is likely very expensive and of limited effect. Recently, defenses based on either filtering the inputs or outputs of LLMs have been proposed but these approaches are geared towards defending the models against "jailbreaking" attacks, i.e., attacks that cause LLMs to produce toxic or harmful text [Jain et al. 2023; Kumar et al. 2023; Robey et al. 2023]. We instead propose cost-effective self-defense prompting techniques to counteract alterations caused by *known* adversarial code perturbations by leveraging an LLM's own capabilities, through prompt instructions.

8 CONCLUSION

In this paper we have shown that adversarial examples obtained with a smaller code model are indeed transferable to modern LLMs, weakening LLMs' performance on coding tasks. We further proposed and evaluated novel prompt-based self-defense strategies that do not require retraining the LLMs but instead utilize LLMs' powerful capabilities to perform in-context learning and understand human instructions as well as code. We also presented meta-prompting, a technique that leverages LLMs themselves to synthesize the self-defense prompts. Our experiments show that prompt-based self-defense is effective, and meta-prompting can lead to even more effective prompts than the ones that are manually crafted. We believe that leveraging LLMs for self-defense and for synthesizing prompts is a generally applicable strategy that could also be useful for other reasoning tasks. As future work, we also plan to investigate techniques that frame the search for finding the optimal self-defense prompts as an optimization problem solved either via gradient-based search over LLM parameters or via techniques suitable for black-box LLMs but requiring multiple calls to the LLM.

REFERENCES

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJOFETxR-

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. PMLR, 2091–2100.

Uri Alon, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1gKYo09tX

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. 2018. Generating Natural Language Adversarial Examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 2890–2896. https://doi.org/10.18653/v1/D18-1316

Anthropic. 2023. Model Card and Evaluations for Claude Models. https://efficient-manatee.files.svdcdn.com/production/images/Model-Card-Claude-2.pdf?dm=1689034733

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

Pavol Bielik and Martin Vechev. 2020. Adversarial Robustness for Code. arXiv:2002.04694 [cs.LG]

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.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=SJeqs6EFvB

John C Duchi, Peter L Bartlett, and Martin J Wainwright. 2012. Randomized smoothing for stochastic optimization. *SIAM Journal on Optimization* 22, 2 (2012), 674–701.

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2018. HotFlip: White-Box Adversarial Examples for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Melbourne, Australia, 31–36. https://doi.org/10.18653/v1/P18-2006

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

Fengjuan Gao, Yu Wang, and Ke Wang. 2023. Discrete Adversarial Attack to Models of Code. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 172–195.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6572

Neel Jain, Avi Schwarzschild, Yuxin Wen, Gowthami Somepalli, John Kirchenbauer, Ping-yeh Chiang, Micah Goldblum, Aniruddha Saha, Jonas Geiping, and Tom Goldstein. 2023. Baseline defenses for adversarial attacks against aligned language models. *arXiv preprint arXiv:2309.00614* (2023).

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, et al. 2022. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221* (2022).

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

Aounon Kumar, Chirag Agarwal, Suraj Srinivas, Soheil Feizi, and Hima Lakkaraju. 2023. Certifying llm safety against adversarial prompting. *arXiv preprint arXiv:2309.02705* (2023).

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. arXiv:2004.02843 [cs.SE]

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. 2022b. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158 arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158

Zhen Li, Guenevere (Qian) Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022a. RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1906–1918. https://doi.org/10.1145/3510003.3510181

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=1qvx610Cu7

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2019. Towards Deep Learning Models Resistant to Adversarial Attacks. arXiv:1706.06083 [stat.ML]

OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL]

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 27496–27520. https://proceedings.mlr.press/v202/pei23a.html

Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (oct 2018), 25 pages. https://doi.org/10.1145/3276517

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with" gradient descent" and beam search. *arXiv preprint arXiv:2305.03495* (2023).

Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2020. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043* (2020).

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016a. Probabilistic Model for Code with Decision Trees. *SIGPLAN Not.* 51, 10 (oct 2016), 731–747. https://doi.org/10.1145/3022671.2984041

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016b. Probabilistic model for code with decision trees. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 731–747.

Alexander Robey, Eric Wong, Hamed Hassani, and George J Pappas. 2023. SmoothLLM: Defending Large Language Models Against Jailbreaking Attacks. *arXiv preprint arXiv:2310.03684* (2023).

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2021. Generating Adversarial Computer Programs using Optimized Obfuscations. In *International Conference on Learning Representations*. https://openreview.net/forum?id=PH5PH9ZO_4

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. arXiv:1312.6199 [cs.CV]

Katherine Tian, Eric Mitchell, Allan Zhou, Archit Sharma, Rafael Rafailov, Huaxiu Yao, Chelsea Finn, and Christopher D Manning. 2023. Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback. *arXiv preprint arXiv:2305.14975* (2023).

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan,

Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ByloJ20qtm

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)* 53, 3 (2020), 1–34.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery.*

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

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large Language Models are Human-Level Prompt Engineers. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=92gvk82DE-

Andy Zou, Zifan Wang, J. Zico Kolter, and Matt Fredrikson. 2023. Universal and Transferable Adversarial Attacks on Aligned Language Models. *CoRR* abs/2307.15043 (2023). https://doi.org/10.48550/arXiv.2307.15043 arXiv:2307.15043