How Important are Good Method Names in Neural Code Generation? A Model Robustness Perspective

GUANG YANG, YU ZHOU*, and WENHUA YANG, Nanjing University of Aeronautics and Astronautics, China
TAO YUE, Simula Research Laboratory, Norway
XIANG CHEN, Nantong University, China
TAOLUE CHEN†, Birkbeck, University of London, UK

Pre-trained code generation models (PCGMs) have been widely applied in neural code generation which can generate executable code from functional descriptions in natural languages, possibly together with signatures. Despite substantial performance improvement of PCGMs, the role of method names in neural code generation has not been thoroughly investigated. In this paper, we study and demonstrate the potential of benefiting from method names to enhance the performance of PCGMs, from a model robustness perspective. Specifically, we propose a novel approach, named RADAR (neuRAl coDe generAtor Robustifier). RADAR consists of two components: RADAR-Attack and RADAR-Defense. The former attacks a PCGM by generating adversarial method names as part of the input, which are semantic and visual similar to the original input, but may trick the PCGM to generate completely unrelated code snippets. As a countermeasure to such attacks, RADAR-Defense synthesizes a new method name from the functional description and supplies it to the PCGM. Evaluation results show that RADAR-Attack can reduce the CodeBLEU of generated code by 19.72% to 38.74% in three state-of-the-art PCGMs (i.e., CodeGPT, PLBART, and CodeT5) in the fine-tuning code generation task, and reduce the Pass@1 of generated code by 32.28% to 44.42% in three state-of-the-art PCGMs (i.e., Replit, CodeGen, and CodeT5+) in the zero-shot code generation task. Moreover, RADAR-Defense is able to reinstate the performance of PCGMs with synthesized method names. These results highlight the importance of good method names in neural code generation and implicate the benefits of studying model robustness in software engineering.

CCS Concepts: • **Software and its engineering**; • **Computing methodologies → Artificial intelligence**;

Additional Key Words and Phrases: Code generation, Adversarial examples, Robustness, Passive defense, Pre-trained model

ACM Reference Format:
Guang Yang, Yu Zhou, Wenhua Yang, Tao Yue, Xiang Chen, and Taolue Chen. . How Important are Good Method Names in Neural Code Generation? A Model Robustness Perspective. 0, 0, Article 0 ( ), 35 pages.

*Corresponding author.
†Corresponding author.

Authors' addresses: Guang Yang, novelyg@outlook.com; Yu Zhou, zhouyu@nuaa.edu.cn; Wenhua Yang, ywh@nuaa.edu.cn, Nanjing University of Aeronautics and Astronautics, Nanjing, China; Tao Yue, tao@simula.no, Simula Research Laboratory, Norway; Xiang Chen, xchencs@ntu.edu.cn, Nantong University, Nantong, China; Taolue Chen, t.chen@bbk.ac.uk, Birkbeck, University of London, London, UK.

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 ACM 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.
© Association for Computing Machinery.
XXXX-XXXX//0-ART0 $15.00
https://doi.org/

1 INTRODUCTION

Context. Neural code generation generally refers to the task of generating executable code from functional descriptions in natural language using neural networks and it has the potential to reduce the development pressure on programmers. While early studies on automatic code generation mainly focus on domain-specific programming languages (e.g., card game code [53], Bash [52], and regular expressions [56]), recent neural code generation for common programming languages takes the inspiration from the impressive achievements of pre-trained deep learning models in natural language processing, and has attracted a lot of attention recently [2, 14, 16, 19, 57, 67, 80, 88].

In literature, neural code generation typically focuses on method-level code generation, i.e., generating a method body by taking mainly two types of input: (1) functional description of the intended code only [2, 57, 67, 88], henceforth denoted by FD; or (2) both the functional description and the method signature (i.e., the combination of the method name and the parameter list [15, 18, 19, 34]), henceforth denoted by $\mathrm{FD}^{\mathrm{Sig}}$. Furthermore, we categorize the existing benchmarks into two groups based on their data size and the availability of test cases, i.e., fine-tuning code generation tasks and zero-shot code generation tasks. For example, we classify Human-Eval [16] as a zero-shot code generation task due to its limited dataset size (164 data items), which includes test cases. This dataset is insufficient to adequately fine-tune the model. In contrast, CONCODE [42] is classified as a fine-tuning code generation task. It consists of numerous data items without accompanying test cases, thereby providing an extensive dataset for fine-tuning the model.

Motivation. Evidence from the literature has shown that taking signature information as input can largely boost the performance of neural code generation, i.e., generating more syntactically and semantically correct code [49, 50]. For example, the BLEU score of the PyMT5 model was nearly doubled by taking signature information as input [19]. Our experiment results (Section 4.2) also confirm this observation. However, a natural, scientifically intriguing question of engineering importance is: what contribution does the additional signature information make so the $\mathrm{FD}^{\mathrm{Sig}}$ approaches become more effective? Clearly, a thorough investigation of this question would be very useful in further improving the performance of neural code generation. Considering that not every code method contains the parameter list, we prioritize our research on the method names in the signature. In this paper, we study the impact of method names through the lens of *robustness* of the pre-trained deep learning models.

Robustness refers to the ability of a model to cope with erroneous inputs and errors that occurred during its execution [20]. In particular, in deep learning, by adding minor perturbations to the benign inputs of a neural network model, one can generate adversarial examples, which may spoof the model, thereby causing significant derivations in the model output. A vast amount of attention has been paid to studying the robustness of deep learning models, typically in domains such as image classifications, computer vision, and natural language processing [13, 29, 83], where adversarial examples often pose both safety and security concerns. We remark that, though, in the context of neural code generation applications, adversarial examples of deep learning models may not bring severe safety concerns, which is in stark contrast to other application domains such as autonomous driving [22], studying and improving the robustness of pre-trained code generation models (PCGMs), indeed, bring (previously-unexpected) benefits, as we will demonstrate in this paper.

However, state-of-the-art PCGMs may *not* be robust. Fig. 1(a) presents an example (with the code collected from the PyPi project.[1]) to illustrate the robustness challenge faced by the three representative PCGMs (i.e., CodeGPT, PLBART, and CodeT5) in the fine-tuning code generation task. After fine-tuning, we use the functional description and the signature as the input to each

[1]https://pypi.org/project/fomoro-pyoneer/

Reference

```
def range_moments(minval, maxval):
  """
  Compute elementwise mean and varian
ce from min and max values.
  Args:
    minval:A tensor of minimum values.
    maxval:A tensor of maximum values.
  Returns:
      Tuple of (mean, variance).
  """
  mean = (maxval + minval) / 2
  variance = tf.square((maxval -
minval) / 2)
  return mean, variance
```

CodeGPT under Foo-Attack

```
def foo(minval, maxval):
  """
  Compute elementwise mean and varian
ce from min and max values.
  Args:
    minval:A tensor of minimum values.
    maxval:A tensor of maximum values.
  Returns:
      Tuple of (mean, variance).
  """
  mean = tf.gradients(foo,
slice=foo.slices)
  var = tf.reduce_sum(tf.abs(tf.stop_
gradient(mean))) –
tf.suared_error(mean)
  return minval, maxval
```

PLBART under Foo-Attack

```
def foo(minval, maxval):
  """
  Compute elementwise mean and varian
ce from min and max values.
  Args:
    minval:A tensor of minimum values.
    maxval:A tensor of maximum values.
  Returns:
      Tuple of (mean, variance).
  """
  p = [minval, maxval]
  if p:
    mean = p[0]
    variance = p[1]
  else:
    mean = maxval - minval
    variance = minval - maxval
  return mean, variance
```

CodeT5 under Foo-Attack

```
def foo(minval, maxval):
  """
  Compute elementwise mean and varian
ce from min and max values.
  Args:
    minval:A tensor of minimum values.
    maxval:A tensor of maximum values.
  Returns:
      Tuple of (mean, variance).
  """
  mn = foo(minval)
  mx = foo(maxval)
  return mn, mx
```

(a) Evaluation on the Python dataset

Reference

```
def greatest_common_divisor(a: int, b
: int) -> int:
  """ Return a greatest common
divisor of two integers a and b
  """
  while b:
    a, b = b, a % b
  return a
```

Replit under Foo-Attack

```
def foo(a: int, b: int) -> int:
  """ Return a greatest common
divisor of two integers a and b
  """
  if a == 0:
    return b
  elif b == 0:
    return a
  elif a < b:
    return foo(b, a%b)
  else:
    return foo(a, b%a)
```

CodeGen under Foo-Attack

```
def foo(a: int, b: int) -> int:
  """ Return a greatest common
divisor of two integers a and b
  """
  return gcd(a, b)
```

CodeT5+ under Foo-Attack

```
def foo(a: int, b: int) -> int:
  """ Return a greatest common
divisor of two integers a and b
  """
  return a if a < b else b
```

(b) Evaluation on the Human-Eval dataset

Fig. 1. The motivating examples illustrating the non-robustness challenge faced by popular PCGMs

model (code highlighted in light grey in Fig. 1). The generated code snippets are exactly the same as the reference (the leftmost). However, if we simply replace the method name `range_moments` with `foo` and keep the functional description untouched, all three models generate totally incorrect code (highlighted in the dark grey). Fig. 1(b) presents an illustrative example, utilizing code collected from Human-Eval [16], to highlight the challenge of robustness encountered by three representative PCGMs (i.e., Replit [75], CodeGen [63] and CodeT5+ [87]) in the zero-shot code generation task. For each model, we input the functional description and the signature, resulting in generated code snippets that successfully pass the test cases, akin to the reference code shown on the leftmost side. However, when a simple substitution is made by replacing the method name `greatest_common_divisor` with `foo` while retaining the functional description, all three models produce completely incorrect code that fails to pass the test cases (highlighted in the dark grey). Note that `foo` is the most commonly used variable name in computer tutorial textbooks. This clearly shows that these models are not robust for the current input. Indeed, as shown in Section 4.2, poor robustness of PCGMs is commonly seen and greatly impacts their performance. For instance, our attack method can generate meaningful (adversarial) and natural method names that could reduce the CodeBLEU score of the generated code by 19.72%–38.74% in CodeGPT [57], PLBART [2] and CodeT5 [88] in the fine-tuning code generation task. In the zero-shot code generation task, our attack can reduce the Pass@1 score of the generated code by 32.28%–44.42% in Replit [75], CodeGen [63], and CodeT5+ [87]. Hence, we conclude that FD^{Sig} approaches, albeit demonstrating a better performance, are fragile (hence less robust) as they heavily rely on the selection of the input method name. This is a serious matter, since developers (i.e., users of PCGMs) may select a low-quality name in coding practice (due to inexperience, carelessness, bad habits, or otherwise just a typo), an ill-formed method name might largely degrade the performance of PCGMs, which thus generate unwanted code.

```
Commit b31e5592bb65f3d91323f6fd2106026b154a91ca
- public static ButtonType guiConformationAlert(String aTitle, String aHeaderText, String aContentText){
+public static ButtonType guiConfirmationAlert(String aTitle, String aHeaderText, String aContentText){
```

```
Commit 0bdb66924dd9f076bd225f2930e2075d3a15974d
- find . -mindepth 1 -type d | wc -1
+find . -mindepth 1 -type d | wc -l
```

Fig. 2. Two typo fixes for code refactoring in Github

In a real-world software development context, it is often the case that developers refactor their code simply due to typos. The study conducted by Liu et al. [54] shows that an important code refactoring operation is due to simple typos (cf. Fig. 2. For instance, developers spelled 'Confirmation' as 'Conformation' in a method name or spelled 'l' as '1' in bash code). Meanwhile, a study conducted by Murphy-Hill et al. [61] on activity from over 13,000 Java developers finds that renaming methods was the most commonly used refactoring operation, accounting for 74.8% of all refactoring operations. This indicates that existing naming guidelines make it difficult for developers, especially novices, to come up with meaningful, concise, and compact method names [25]. Moreover, developers might have different naming styles [12, 39]. It is also likely that a code generation system fails due to different styles in method names. Previous works [27, 69, 85, 90] focus on studying the impact of the method name quality on the readability and maintainability of source code. However, the role of the method name quality for code automation tasks has not been thoroughly investigated.

A possible approach to address the robustness challenge is to synthesize proper method names to replace those provided by developers, by which the performance of FD^{Sig} approaches can hopefully be reinstated. Generating high-quality method names is an interesting task in its own right.

Proposed solution. In this paper, we propose a novel method, along with a tool suite, named RADAR (neuRAl coDe generAtor Robustifier), of two major components: RADAR-Attack and RADAR-Defense. Specifically, RADAR-Attack imitates the undesirable behavior (just like typos) of developers mentioned above and then generates natural, visually, and semantically similar method names. They serve as adversarial examples to reveal the robustness problem of PCGMs, but can also be considered as a tool to assess the robustness of PCGMs. RADAR-Defense, on the other hand, aims to reinstate the performance of PCGMs. One way is via adversarial training whereby we adapt the ACCENT approach [104], leveraging the generated adversarial examples to retrain a model. The other is to sanitize the input whereby we propose a passive and lightweight defense method, which synthesizes meaningful and concise method names based on the given functional descriptions. These method names are inputted into the PCGMs together with the functional descriptions and other signature information (e.g., parameter lists).

To evaluate the effectiveness of RADAR, we consider six state-of-the-art, large-scale PCGMs (i.e., CodeGPT, PLBART, and CodeT5 in the fine-tuning code generation task and CodeGen, CodeT5+, and Replit in the zero-shot code generation task). Experiment results show that RADAR-Attack is effective in attacking these PCGMs, and RADAR-Defense can improve their robustness and thus reinstate their performance by generating higher-quality method names. For instance, the CodeT5 model has a CodeBLEU value of 46.09 when not being attacked on the Java dataset, which drops to 31.58 under RADAR-Attack. Using the method names synthesized by RADAR-Defense, the CodeBLEU value is back to 46.11.

Contributions.

- We devise RADAR-Attack to attack PCGMs based on functional descriptions and signatures, showing that their performance is susceptible to provided method names.
- We propose a defense method RADAR-Defense to recover the performance of the attacked PCGMs.
- As a byproduct, we provide novel approaches to automatically synthesize method names, which are meaningful in various contexts such as software refactoring.

Key findings. Based on our empirical study, we conclude that good names play a crucial role in neural code generation, and they can be synthesized from the functional description with a well-designed approach. In other words, functional description + parameter list + RADAR-Defense would provide a strong performance boost for state-of-the-art PCGMs. To the best of our knowledge, this represents one of the first works on studying the robustness of neural code generation models via adversarial examples. More importantly, at the methodological level, this paper promotes, with solid evidence, the importance of studying the robustness of deep learning models in neural code generation and even software engineering in general, where they are playing an increasingly important role.

To facilitate reproducibility and further research, source code, benchmarks, and experimental data are released at https://github.com/NTDXYG/RADAR.

Structure. The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes the framework and key approaches in RADAR. Section 4 provides the experiment results and their analysis. Section 5 discusses the quantitative study of the effectiveness of RADAR and the potential threats to the validity of our empirical study. Section 6 concludes this paper and discusses future work.

2 RELATED WORK

2.1 Neural Code Generation

Previous studies on code generation mainly focus on domain-specific languages [52, 53, 56]. Studies on code generation for general programming languages [60, 77] use sequence-to-sequence models, and they formalize code generation as text sequence generation based on the hypothesis of code naturalness [4, 38]. Some studies [79, 96] use tree-based models, by capturing the grammar of the natural language as a priori-knowledge to generate complex programs. Other studies [35, 36] use retrieval-enhanced models, i.e., benefiting from information retrieval to compensate for the lack of ability of neural networks to memorize large and complex structures.

In recent years, researchers have gradually utilized pre-trained models for neural code generation tasks, which can be classified into two types based on benchmark requirements: fine-tuning code generation tasks and zero-shot code generation tasks. Fine-tuning code generation tasks are typically applied to benchmarks that lack test cases, such as CONCODE [42] and CoNaLa [95]. These benchmarks are divided into training, validation and test sets, with pre-trained models (often with parameter numbers less than a billion) fine-tuned on the training set to be adapted to the specific task. For example, models like CodeGPT [57], PLBART [2], and CodeT5 citewang2021codet5 leverage the GPT, BART, and T5 architectures of language models pre-trained on code corpora. Extensive evaluations on the CONCODE benchmark have demonstrated their robust code generation capabilities. Moreover, models such as PyMT5 [19], CoTexT [67], and NatGen [14] have also exhibited promising performance on code generation tasks, depending on the specific pre-training tasks. However, these models are more suitable for fine-tuning code generation tasks, as their parameter numbers are not large enough to demonstrate emergent capabilities in zero-shot scenarios.

With the development of neural networks, Hestness et al. [37] point out that the performance of Transformer-based models improved in a predictable way as the amount of computation or the size of the network increased, and is called “scaling laws” [43]. When the model scales to a certain level, the phenomenon of “emergent capacity" [89] can occur. Building upon this understanding, researchers have increasingly employed large language models with over a billion parameters for zero-shot code generation tasks. These models have demonstrated substantial enhancements in the performance of code generation benchmarks, aligning with the aforementioned theory.

The zero-shot code generation task is typically applied to benchmarks that include test cases but often have limited data size due to the costly manual construction of test cases. In this context, Chen et al [16] first introduced and evaluated the capabilities of Codex, which is pre-trained on GitHub code with 12 billion model parameter. Subsequently, Li et al. [48] proposed AlphaCode with 1.1 billion parameters, and Chowdhery et al. [17] introduced PaLM-Coder, with 540 billion parameters. These models were evaluated for their performance on HumanEval. However, all of these models are of closed-source. For the open-source models, Fried et al. [24] proposed InCoder, which is trained for program synthesis (via left-to-right generation) and editing (via masking and infilling). Nijkamp et al. [62, 63] proposed CodeGen and CodeGen2, which are large language models for code with multi-turn program synthesis. Zheng et al. [103] proposed CodeGeeX, a multilingual model with 13 billion parameters for code generation. CodeGeeX is pre-trained on 850 billion tokens of 23 programming languages. Li et al. [47] proposed StarCoder, a 15.5 billion parameter model with an 8K context length, infilling capabilities, and fast large-batch inference enabled by multi-query attention. In addition, the Replit company proposed replit-code-v1-3b model [75], which is trained on a subset of the Stack Dedup v1.2 dataset, and the training mixture includes 20 different languages. Differing from the aforementioned decoder-only model, Wang et al. [87] introduced CodeT5+, a family of encoder-decoder LLMs for code-related tasks.

In contrast to the previous studies, our primary objective is to evaluate the influence of method names on neural code generation from the perspective of model robustness. We have observed a significant improvement in the performance of neural code generation when incorporating signature information as input. This observation has motivated us to further investigate the impact of method names, an essential component of signatures, on the code generation process. By examining the relationship between method names and code generation, we gain insights into the overall robustness and effectiveness of neural models in generating high-quality code. To achieve this objective, we have conducted empirical investigations on both fine-tuning code generation tasks and zero-shot code generation tasks.

2.2 Adversarial Attack on Code-related Models

Adversarial attacks on code can be divided into two categories: white-box adversarial attacks and black-box adversarial attacks. These attack methods differ primarily in their underlying assumptions. In the case of white-box attacks, the attacker assumes access to the internal structure of the victim models and their training parameters. For instance, Yefet et al. [94] proposed the white-box attack method DAMP, which leverages gradient information from the victim model to manipulate variables in the code. However, white-box attacks are often less practical in real-world scenarios. This is because victim models are typically deployed remotely, and obtaining model’s internal details can be challenging or even impossible.

In contrast to white-box attacks, black-box attacks assume that the attacker has no knowledge of the internal details of the victim models and can only interact with the model through its output. For instance, Applis et al. [6] proposed LAMPION, a method that evaluates the robustness of the CodeBERT model by generating new code snippets that are equivalent to the original test set. Zhang et al. [100] proposed MHM, which utilizes Metropolis-Hastings sampling-based identifier renaming to perform code obfuscation. Tian et al. [81] proposed QMDP, a Q-learning-based Markov decision process, which enables semantically equivalent transformations on the structure of source code.

Rabin et al. [70] employed variable renaming to evaluate the generalizability of neural program analyzers for the task of method name prediction. Liguori et al. [49] explored the use of unseen synonyms and missing information to evaluate line-based code generation tasks. Zeng et al. [98] employed a wide range of NLP-based adversarial attack methods to evaluate pre-trained models and discovered that random attack methods can outperform carefully designed adversarial attack methods in most cases.

In recent research, there has been a growing focus on addressing the naturalness aspect of adversarial examples. Yang et al. [93] proposed a naturalness-aware attack called ALERT, which takes into account the natural semantics of generated examples. ALERT generates multiple natural candidates using the GraphCodeBERT model and the mask language model task in the CodeBERT model. It then calculates the cosine similarity to filter out natural and similar adversarial samples. Zhou et al. [104] proposed ACCENT, an identifier substitution approach for crafting adversarial code snippets in source code summarization. ACCENT aims to generate code snippets that are syntactically correct and semantically similar to the original code snippet. Zhang et al. [99] introduced CARROT, an optimization-based attack technique that assesses and improves the robustness of deep program processing models. Wang et al. [84] presented ReCode, a tool that provides over 30 transformations specifically designed for code generation. These transformations cover various aspects such as document strings, function and variable names, code syntax, and code formatting. Notably, six of these transformations are dedicated to modifying function names.

Moreover, due to the extensive search space of adversarial examples, numerous attack methods utilize optimization algorithms to enhance the efficiency of searching and thus improve the attack performance. In the field of natural language processing, commonly employed optimization algorithms include greedy algorithms [92], genetic algorithms [5], and particle swarm optimization algorithms [97]. These optimization algorithms are also widely applied in adversarial attack methods for code-related tasks.

In this paper, we present a novel black-box attack approach targeting code generation. Different from the previous studies, our focus is on real-world scenarios where neither users nor attackers have access to the internal structure of PCGMs. Our approach not only generates semantically equivalent adversarial examples but also considers typos and visual similarity, thereby expands the range of adversarial examples explored. To improve the efficiency of attacking PCGMs, we leverage genetic algorithms, which optimize the search process and enhance the effectiveness of our attacks.

2.3 Adversarial Defense on Code-related Models

Current studies on adversarial defense for code-related tasks mainly focus on active defense. Bielik et al. [9] attempted adversarial defense with the assistance of gradient-based adversarial training method [28]. They observed that relying solely on gradient-based adversarial training can provide insights into the model's robustness but may also lead to a decline in performance on the original task. Zhang et al. [100] and Yang et al. [93] proposed the adversarial training method, which uses adversarial examples for data augmentation to re-train the model. However, this approach is highly dependent on the quality of adversarial examples. Zhou et al. [104] and Zhang et al. [102] proposed a lightweight adversarial training method named mask training algorithm, which reduces the model's dependence on the non-robust features since any perturbations on these features may cause a large-scale change in the output.

In contrast to the previous studies, our defense method presents a novel passive approach to effectively restore the performance of PCGMs. This defense method is particularly advantageous in scenarios where PCGMs cannot undergo fine-tuning, such as zero-shot code generation tasks. By implementing this passive defense method, our goal is to improve the robustness of PCGMs, ensuring their effectiveness even in challenging zero-shot code generation scenarios.

3 APPROACH

We show an overview of RADAR in Fig. 3 and RADAR includes two main parts: RADAR-Attack and RADAR-Defense. In particular, RADAR-Attack proposes a black-box, gradient-free optimization attack algorithm and RADAR-Defense proposes a passive defense method based on retrieval-enhanced prompt learning for passive defense.

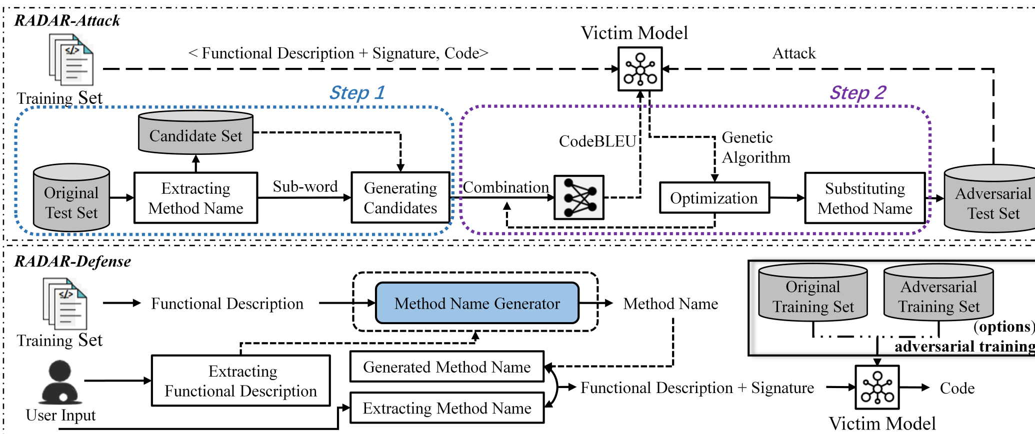


Fig. 3. The Framework of RADAR

3.1 RADAR-Attack

In the fine-tuning code generation task, we commence by fine-tuning pre-trained code generation models using a provided dataset. This process yields a model $\mathcal{F}$, which maps each pair $\mathbf{x}$ consisting of functional description and signature to code $\boldsymbol{y} = \mathcal{F}(\mathbf{x})$. In the zero-shot code generation task, we directly load the weights of the pre-trained model, resulting in the model $\mathcal{F}$. For attacking model $\mathcal{F}$, our goal is to generate an adversarial example $\boldsymbol{x_{adv}}$ for a given $\boldsymbol{x}$, which is visually and semantically similar to $\boldsymbol{x}$, but minimizes the CodeBLEU score between $\boldsymbol{y}$ and $\mathcal{F}(\boldsymbol{x_{adv}})$. Recall that CodeBLEU is a widely recognized automatic evaluation metric of code generation, which subsumes BLEU in the n-gram match and injects code syntax via abstract syntax trees (AST) and code semantics via data flow analysis. In the absence of test cases, CodeBLEU offers a sensible surrogate for automated evaluation. Given the expense of manual test case construction and the absence of corresponding test cases in most datasets, we have utilized CodeBLEU as the optimization objective function for both fine-tuning code generation tasks and zero-shot code generation tasks. Meanwhile, there is a correlation between the metrics, as seen in Table 4, Table 3 and Table 6, when the CodeBLEU value increases the BLEU metric also increases, so to some extent neither the choice of CodeBLEU nor BLEU has much influence on the selection of the adversarial example. Formally, we aim to solve the following problem:

$$\boldsymbol{x_{adv}} = \arg\min_{\boldsymbol{x'}} \boldsymbol{CodeBLEU}\left(\boldsymbol{y}, \mathcal{F}(\boldsymbol{x'})\right) \tag{1}$$

Note that we only consider part of the input $\boldsymbol{x}$ when generating adversarial examples; we only modify the method name in $\boldsymbol{x}$ (i.e., part of the signature), as parameters are optional for the signature. We assume that the attacker is unaware of the model architecture, parameters, and training data, and can only interact with the model through its output. Therefore, instead of utilizing the gradient-based optimization, we adopt a gradient-free optimization attacking approach, based on a genetic algorithm (GA) as shown in Algorithm 1.

Algorithm 1: Adversarial Example Generation Algorithm

Input: Pre-trained Code Generation Model $\mathcal{F}$;
Code Generation DataSet D, where $(\boldsymbol{x}, \boldsymbol{y}) \in D$
Output: Adversarial DataSet D_{adv};

Initialize: Candidate Method Name Set $V \leftarrow \emptyset$, Adversarial DataSet $D_{adv} \leftarrow \emptyset$;
for *each* $(\boldsymbol{x}, \boldsymbol{y}) \in D$ **do**
  Extract the method name in $\boldsymbol{x}$;
  $V \leftarrow V \cup \{M \mid M =< m_1, \ldots, m_n >$ to represent the sequence of sub-words from the method name$\}$;
Training Method Name Embedding *Embed* via V;
for *each* $(\boldsymbol{x}, \boldsymbol{y}) \in D$ **do**
  Extract the method name set M in $\boldsymbol{x}$;
  Adversarial method name set $M' \leftarrow \emptyset$;
  for *each* $\boldsymbol{m} \in M$ **do**
    $M' \leftarrow L_m$ based on semantic and visual similarity via *Embed* in V;
  $t \leftarrow 0$;
  Initial population generation $\mathcal{P}^t$;
  while $t \leq max\_iterations$ **do**
    Calculate fitness value;
    Selection;
    Crossover;
    if $mutation\_prob \geq random\_prob$ **then**
      Mutation;
    $M' \leftarrow$ minimize evaluate fitness of $\mathcal{P}^t$;
    if *Minimum fitness value is not updated in n iters* **then**
      Early stop;
    $\mathcal{P}^{t+1} \leftarrow buildNewGeneration(\mathcal{P}^t)$;
    $t \leftarrow t + 1$;
  $D_{adv} \leftarrow D_{adv} \cup \{(\boldsymbol{x}.replace(M, M'), \boldsymbol{y})\}$;
return D_{adv};

In Algorithm 1, RADAR-Attack first extracts method names from all the signatures in the dataset and then tokenize each method name according to the method naming convention (e.g., the camel case or the snake case) to build a set of sub-words. RADAR-Attack then creates a candidate set for each sub-word. The candidates are selected based on their **visual similarity** (to model typos) and **semantics similarity** (to model programmers' preferences of the use of English words). Finally, RADAR-Attack generates adversarial examples for method names by considering various combinations. It uses GA to generate the best replacement for the original method name by minimizing the CodeBLEU value [74]. We now elucidate these two main steps, i.e., *Step 1* candidates generation (the blue box in Fig. 3) and *Step 2* optimization with GA (the purple box in Fig. 3).

3.1.1 Step 1. Candidates Generation. The first step aims to generate high-quality candidate adversarial examples that have high visual and semantic similarity with the original words. According to previous studies [46, 73], the text semantic is likely to be retained or deduced after the user changes a few characters. Therefore, we make small-scale changes to the original words for human comprehension, which can help to generate visual similar candidates. Moreover, as method names often contain a variety of domain-specific acronyms, jargon, and their combinations, they are frequently outside the vocabulary of the word embedding model in the general domain. In this study, based on our previous work [104], we first train a general word2vec [59] model based on the

Wiki dataset and then continue to train it for a corpus of method names (Line 2-5 in Algorithm 1). Finally, we select the *top* 5 nearest candidate sub-words for each sub-word in the method name based on the cosine similarity.

Based on these observations, we propose four operators to generate candidate samples (Line 9-10 in Algorithm 1):

- **Delete Operator:** Randomly delete a character of the sub-word.
- **Swap Operator:** Randomly swap two adjacent letters in the word.
- **Replace-vis Operator:** Replace characters with visually similar characters (e.g., replacing "l" with "1", "O" with "0") or special coding styles words (e.g., replacing "2" with "to", "4" with "for").
- **Replace-sem Operator:** Replace a sub-word in the method name with its most semantic similar Top5 candidate sub-words in a high-dimensional vector space.

Notice the first two operators are designed to model that developers type carelessly. The **Replace-vis** operator is designed to model the novice behaviors (e.g., copy the code from course materials to their program tasks). An example in Fig. 4 illustrates the four operators. Method name `decode_dict_to_str` can be divided into four sub-words (i.e., decode, dict, to, and str). Each operator generates multiple candidate sub-words, which form the discrete search space of the original sub-words.

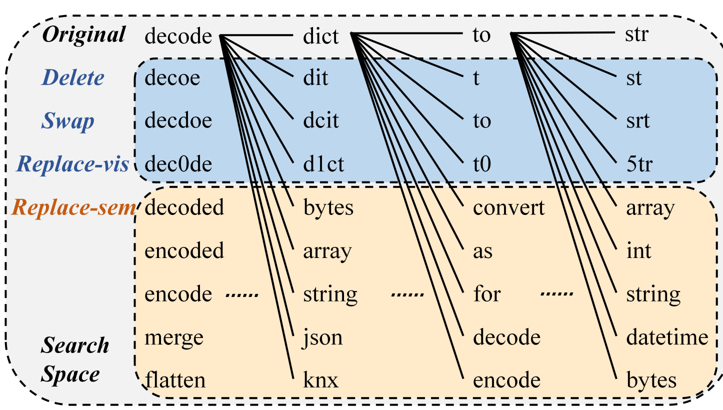


Fig. 4. An Example of permutations of candidate sub-words

3.1.2 Step 2. Optimization with GA. This step aims to find the most effective adversarial examples in the discrete search space that can successfully fool the victim model, with GA. Let $\boldsymbol{M} = \langle m_1, \ldots, m_n \rangle$ be the sequence of sub-words from the method name. The discrete search space can be represented as $\boldsymbol{M^k} = \{\langle m_1^k, \ldots, m_n^k \rangle \mid m_i^k \in \mathbb{V}(m_i)\}$, where k denotes the number of the generated candidate sub-words, $\mathbb{V}(m_i)$ is the set of candidates of m_i.

By Equation 1, the fitness function of RADAR-Attack can be formalized as

$$\boldsymbol{y_{goal}} = \arg\min_{\boldsymbol{M'}} \boldsymbol{CodeBLEU}\left(\boldsymbol{y}, \mathcal{F}\left(\boldsymbol{x}.replace(\boldsymbol{M}, \boldsymbol{M'})\right)\right)$$

where $\boldsymbol{M'}$ represents the set of solutions with n variables (i.e., the number of sub-words). Values of each variable are in the range $[0, k]$, where k denotes the number of candidates.

We denote the initial population as the initial generation $\mathcal{P}^0$ (Line 12 in Algorithm 1). The size of the population is denoted as *size_population*. To get a new generation (i.e., transiting from $\mathcal{P}^t$ to $\mathcal{P}^{t+1}$), the operations of selection, crossover (with *crossover_prob*), and mutation (with *mutation_prob*) are performed (Line 14-18 in Algorithm 1). The termination condition is the maximum number of generations, which is denoted as *max_iterations*. To improve the computational efficiency of GA, we refer to the early-stop strategy used by Garcia et al. [26]. The evolution ends when the average fitness of the population does not improve above a certain threshold in the last n generations (Line 20-21 in Algorithm 1). To avoid experimental bias due to the randomness of GA, we repeat the run 30 times, taking the average values as the final result.

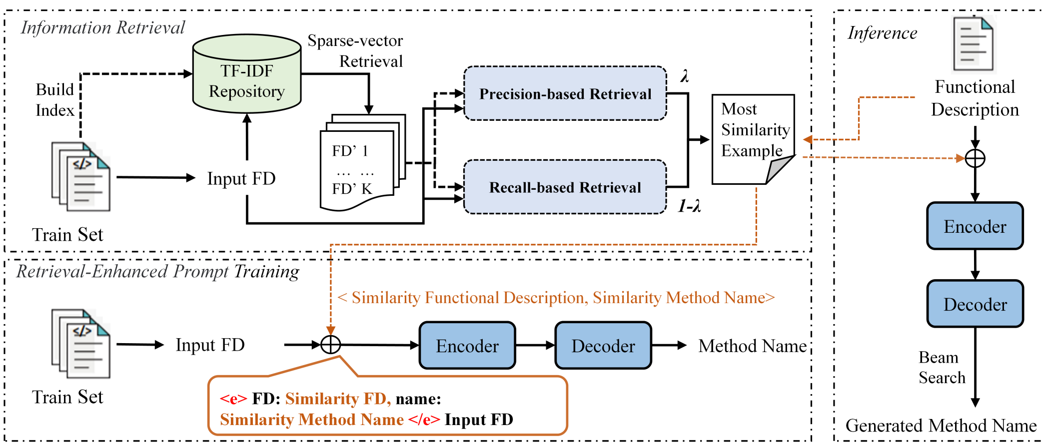


Fig. 5. Overview of RADAR-Defense

3.2 RADAR-Defense

RADAR-Defense can adapt the adversarial training approach which leverages generated adversarial examples to retrain a model in the fine-tuning code generation task, but this is optional, as we mentioned in RADAR-Attack the model black box assumption, we expect RADAR-Defense is able to reinstate its performance without retraining PCGMs. Thus RADAR-Defense's main purpose is to synthesize a new method name for a given functional description to replace the original method name in the signature. As shown in Fig. 5, RADAR-Defense mainly consists of two steps: 1) generating the most similar example via information retrieval, and 2) training the model with the augmented function description via prompt training.

In general, we treat the training set as a corpus, from which a list of key-value pairs ($\mathcal{T} = \{(c_i, m_i)\}$) can be constructed, with c_i and m_i denoting the functional description and the method name, respectively. Given a functional description c, the retrieval model aims to retrieve the most relevant example $z = (c_r, m_r)$ from the corpus. To achieve this, we first retrieve top-K similar functional descriptions from the corpus based on the standard TF-IDF due to low computational cost, out of which we further retrieve the most similar functional description based on lexical similarity.

First, we adopt standard TF-IDF[3] and cosine distance; each functional description c is associated with the semantic sparse-vector TF-IDF$(c) \in \mathbb{R}^D$, where D denotes the total number of words in the corpus, and the similarity is defined as the cosine distance:

$$semantic(a, b) = \frac{\text{TF-IDF}(a) \cdot \text{TF-IDF}(b)}{\| \text{TF-IDF}(a) \| \cdot \| \text{TF-IDF}(b) \|}$$

Second, for lexical similarity, we utilize precision-based and recall-based retrieval methods. In our study, we use two evaluation metrics, (i.e., BLEU [65] and ROUGE [51]), which measure the similarity based on precision and recall, respectively.

For the given functional descriptions a and b, lexical similarity can be computed as:

$$lexical(a,b) = \lambda \, \text{BLEU}(a,b) + (1-\lambda) \, \text{ROUGE}(a,b),$$

where λ is a hyper-parameter for allowing the flexible control of precision and recall in information fusion.

In the next step, we resort to a retrieval-enhanced prompt training approach. This approach is based on the observations [11, 30, 45, 66] that by granting a model access to external memory via information retrieval techniques, more information can be obtained in the model generation process and thus the uncertainty can be reduced. With retrieval-based models, knowledge can be explicitly introduced through plug-and-play mechanisms, making them more scalable. Additionally, compared to generating text from scratch, retrieval-enhanced approaches leverages reference information obtained through retrieval, which can alleviate the difficulty of text generation to some extent. This approach is similar to contextual learning of Large Language Models.

Recall that for the given functional description c, we obtain the most relevant sample $z = (c_r, m_r)$ in the first step. We augment c to form a retrieval-enhanced functional description c'.

$$c' = \langle \mathbf{e} \rangle \mathbf{FD:} c_r, \mathbf{name:} m_r \langle / \mathbf{e} \rangle \oplus c$$

where, z is tagged and concatenated with c, such that the model can learn the most similar functional description and method name information.

Our model is based on UniXcoder [32], a unified cross-modal pre-trained model which can support both code-related understanding and generation tasks based on *Transformer* [82], and utilizes mask attention matrices with prefix adapters to control the access to context for each token.

For the input c', our model first tokenizes it to obtain an input sequence $\{\mathbf{c'}_i\}_{i=1}^{|c'|}$. We utilize UniXcoder to encode the c' and decode it to synthesize the method name. Note that the parameters of the encoder and decoder in UniXcoder are shared. The final decoder's output of the UniXcoder $\mathbf{H}^t$ is sent to a fully connected neural network. This network can pass a softmax layer to predict the probability of the next token, which can be defined as follows.

$$p\left(m_{t+1} \mid m_1, \cdots, m_t\right) = \text{softmax}\left(\mathbf{H}^t \mathbf{W} + \mathbf{b}\right)$$

In model training, we use the Incomplete-Trust (In-trust) [40] loss function, viz.,

$$\mathcal{L}_{\text{In-trust}}(\theta) = \alpha \mathcal{L}_{\text{CE}}(\theta) + \beta \mathcal{L}_{\text{DCE}}(\theta)$$

where $\mathcal{L}_{\text{CE}}(\theta) = -\sum_{i=1}^{|m|} q \log p$ and $\mathcal{L}_{\text{DCE}}(\theta) = -\sum_{i=1}^{|m|} p \log(\delta p + (1-\delta)\, q)$. Here $\mathcal{L}_{\text{CE}}$ represents the Cross-Entropy function which is not noise-tolerant but benefits the convergence of the model, $\mathcal{L}_{\text{DCE}}$ represents the robust Distrust-Cross-Entropy and can effectively prevent the model from overfitting noisy samples; p denotes the model's prediction distribution and q denotes the trust label distribution.

4 EVALUATION

We aim to evaluate the effectiveness of our approach by answering the following three research questions (RQs).

RQ1 How effective is RADAR-Attack in degrading the performance of FD^{Sig} by attacking method names?

RQ2 How effective is RADAR-Defense in reinstating the performance of FD^{Sig}?

RQ3 How effective is RADAR-Defense in terms of the method name generation?

4.1 Experiment Design

4.1.1 Dataset. In the fine-tuning code generation task, widely used open-source datasets include CONCODE [42] for the Java language, and Django [64], CoNaLa [95], and Juice [1] for the Python language.

Issues	*NL*	*Code*	*Repository*

Incomplete Function Description	{@inheritDoc}	`public synchronized Map<String, ByteString> getTags() {`
		`    if (tags != null)`
		`        return Maps.newHashMap(tags);`
		`    else`
		`        return Maps.newHashMap();`
		`}`
			Git: https://github.com/1000000000001/bitcoinj
			Path: core/src/main/java/org/bitcoinj/utils/BaseTaggableObject.java

Irregularity Method Name	Send a {@link #DEBUG_LEVEL} log message.	`public static void d(Object obj) {`
		`    if (Log.DEBUG > DEBUG_LEVEL) {`
		`        String tag = getClassName();`
		`        String msg = obj != null ? obj.toString() : "obj == null";`
		`        Log.d(tag, msg);`
		`    }`
		`}`
			Git: https://github.com/pranavlathigara/android-utils-1
			Path: src/com/ihongqiqu/util/LogUtils.java

URL Leakage	convert from from_currency to to_currency by requesting API	`def convert_using_api(from_currency, to_currency):`
		`    convert_str = from_currency + '_' + to_currency`
		`    options = {'compact': 'ultra', 'q': convert_str}`
		`    api_url = 'https://free.currencyconverterapi.com/api/v5/convert'`
		`    result = requests.get(api_url, params=options).json()`
		`    return result[convert_str]`
			Git: https://github.com/NearHuscarl/py-currency
			Path: currency/currency.py

Fig. 6. Irregularity issues in the common fine-tuning code generation dataset

In our research, we have uncovered irregularity issues within specific datasets that can impact the quality and reliability of the data. These issues are illustrated in Fig. 6, and we provide a detailed description of each problem. For example, in the original CONCODE dataset, we have observed instances of incomplete function descriptions and irregular method names. These inconsistencies pose challenges and hinder the advancement of code generation tasks. To support our findings, we present specific examples and indicate their sources within the dataset. Similarly, in the CodeSearchNet dataset [41], we have identified instances of URL leakage issues. These issues contribute to the presence of low-quality data, further limiting the progress in code generation tasks. To illustrate these concerns, we provide concrete examples along with relevant references. The presence of irregularity issues and low-quality data in these datasets emphasizes the significance of addressing data quality concerns in code generation research.

To evaluate our approaches in the fine-tuning code generation task, we need to construct new high-quality datasets to avoid these issues and biases, which include functional descriptions, signatures, and their corresponding code. To ensure the quality of our newly constructed datasets, we designed six heuristic rules to filter out noisy data items by following previous study [42].

H1 The code needs to be parsed through the AST tool to ensure that the syntax is correct.
H2 The number of sub-words of the method name is no less than 2, and the length of each sub-word is no more than 16.
H3 The length of the functional description should be no more than 50 and no less than 4.
H4 The length of the code should be no more than 256.
H5 Remove annotation information, exception code, and URL information from the code.
H6 Unify method names in Java data to hump naming rules and unify method names in Python data to snake naming rules.

Our Java dataset is collected from the raw CONCODE [42] dataset, which is from Java projects on GitHub, and our python dataset is collected from the raw PyTorrent [8] dataset, which is from Python package libraries on PyPI and Anaconda.

In the context of the zero-shot code generation task, several popular open-source datasets are available. For the Java language, the Aix-bench dataset [34] is commonly utilized. For the Python language, widely evaluated datasets include Human-Eval [16], MBPP [7], and GSM8K-Python [17]. Among these datasets, Human-Eval is particularly prominent. However, we have observed that the functional descriptions in the Human-Eval dataset contain test case prompts that include method names. To mitigate the potential impact of these method names on the code generated by the model, we adopt an approach of removing the test case prompts from the functional descriptions. By eliminating the prompts related to the test cases, our aim is to minimize potential bias or influence that the method names in the prompts may have on the code generation process.

Descriptive statistics of our datasets, including their length distributions of functional description (FD), signature (Sig), method name (MD), and Code, are provided in Table 1. Following the previous work [42], we randomly select 100,000 examples for training, 2,000 examples for validation, and 2,000 examples for testing in the fine-tuning code generation task. For the zero-shot code generation task, the Human-Eval dataset consists of a total of 164 test data samples.

Table 1. Descriptive statistics of the datasets when tokenized by BPE algorithm

FD	Avg	Mode	Median	< 16	< 32	< 64
Java	14.25	8	11	69.52%	93.52%	99.99%
Python	17.88	8	13	58.45%	82.86%	99.85%
Sig	Avg	Mode	Median	< 8	< 16	< 32
Java	8.49	7	7	58.44%	93.94%	99.85%
Python	7.78	6	6	55.48%	96.92%	99.98%
MD	Avg	Mode	Median	< 4	< 8	< 16
Java	2.85	2	3	79.36%	99.58%	99.99%
Python	2.74	2	3	83.58%	99.92%	100%
Code	Avg	Mode	Median	< 64	< 128	< 256
Java	40.46	28	38	88.86%	99.99%	100%
Python	69.44	42	63	50.38%	92.54%	100%

4.1.2 Victim Models. The victim models (i.e., the target models under adversarial attacks) are based on large-scale pre-trained language models for source code, which can represent state-of-the-art research for the code generation task.

In the context of the fine-tuning code generation task, we selected CodeGPT, PLBART, and CodeT5 as our models. These models have parameter sizes ranging from 100 million to 300 million.

- **CodeGPT** [57] is a Transformer-based decoder-only model inspired by GPT [71], following similar pre-training tasks of GPT including the causal language model.
- **PLBART** [2] is a Transformer-based encoder-decoder model inspired by BART [44], following similar pre-training tasks of BART, including token masking, token deletion, and token infilling.
- **CodeT5** [88] is a Transformer-based encoder-decoder model inspired by T5[72]. It proposes a novel identifier-aware pre-training task to leverage code-specific structural information.

In the context of the zero-shot code generation task, we selected Replit, CodeGen, and CodeT5+ with the best performance within 3 billion parameters, based on the evaluation results of Gunasekar et al. [31] and Wang et al. [87].

- **Replit** [57] is a Transformer-based decoder-only model [71], which uses Flash Attention [21] for efficient training and inference, and incorporates AliBi positional embeddings [68] to handle variable context length during inference.

- **CodeGen** [2] is a Transformer-based decoder-only model, which adopts rotary position embedding for improving the ability to handle long documents, and uses JAX [10] for training the model.
- **CodeT5+** [88] is a Transformer-based encoder-decoder model, which employs a "shallow encoder and deep decoder" architecture [48], both encoder and decoder are initialized from pretrained checkpoints and connected by cross-attention layers.

4.1.3 Baselines. As for baselines, we select six attack methods to generate adversarial examples, one defense method to improve the robustness of PCGMs, as well as eight method name generation methods, which are described below.

Baselines for adversarial attack and defense. In terms of the baselines for the adversarial attack, we select Foo-Attack, Random-Attack, ALERT-Attack, Genetic-Attack, ReCODE-Attack, and ACCENT-Attack.

- **Foo-Attack** is the attack method we introduced in the motivation, involving the replacement of all method names with the term "foo".
- **Random-Attack** is a method proposed by Zeng et al. [98] that involves randomly substituting method names. In their empirical study, Random-Attack demonstrates improved attack effectiveness compared to existing NLP-based adversarial attack algorithms.
- **ALERT-Attack** is a method proposed by Yang et al. [93]. It utilizes CodeBERT and GraphCodeBERT to generate natural candidates and employs a combination of greedy search and genetic algorithm for optimization.
- **Genetic-Attack** is a method proposed by Alzantot et al. [5]. It utilizes Glove and GoogleLM to generate candidates and employs a genetic algorithm for optimization.
- **ReCODE-Attack** is a method proposed by Wang et al. [84]. It utilizes rule-based transformations to generate candidates and employs a greedy search for optimization.
- **ACCENT-Attack** is a method proposed by Zhou et al. [104]. It first selects several of the most important tokens and then employs word2vec to generate candidates.

When addressing adversarial defense, several common defense methods can be employed, such as gradient-based adversarial training, data augmentation, and mask training (proposed by ACCENT-Defense). It is important to note that gradient-based adversarial training may lead to a decline in model performance, while the effectiveness of data augmentation relies on the quality of the adversarial samples. Among these defense methods, ACCENT-Defense stands out as a lightweight mask learning approach based on active defense. Its objective is to enhance both the robustness and performance of the model. Given its effectiveness and relevance to our research, we consider ACCENT-Defense as the primary baseline for our study.

Baselines for method name generation. We consider eight name generation methods, which are classified into three groups: information-retrieval (including BM25[76], NNGen [55], and CCGIR [91]), deep-learning (including RNN-Att-Copy [25], CodeBERT [23], and UniXcoder [32]), and retrieval-enhanced methods (including Rencos [101] and REINA [86]).

These methods are widely used in method name generation, text summarization, and code summarization. In this study, we train them with functional descriptions as the input and method names as the output, as per the individual model.

4.1.4 Evaluation Metrics and Hyper-parameters. To assess the effectiveness of adversarial attacks in the fine-tuning code generation task, we consider three evaluation metrics: BLEU [65], CodeBLEU [74], and Attack Success Rate (ASR [104]). Here, ASR is defined as the percentage of generated adversarial examples that successfully decrease the CodeBLEU score of the generated code. For the zero-shot code generation task, we consider four evaluation metrics: BLEU, CodeBLEU, Pass@1 [16],

Table 2. Hyper-parameters settings of RADAR

Category	Hyper-parameter Name	Hyper-parameter Value
RADAR-Attack	size_population	20
	max_iterations	50
	crossover_prob	0.9
	mutation_prob	0.001
	early_stop	3
RADAR-Defense	top-K in Java	9
	λ in Java	0.6
	top-K in Python	3
	λ in Python	0.1
	max_source_length	128
	max_target_length	24
	batch_size	64
	max_epoch	50
	learning_rate	4e-5
	early_stop	3

and Attack Success Rate (ASR). Here, ASR is defined as the percentage of generated adversarial examples that successfully reduce the Pass@1 score of the generated code. For method name generation, we use three evaluation metrics, i.e., Exact Match (EM) [25], BLEU and Edit Distance (ED) [25]. These performance measures have been widely used in previous studies for neural code generation and automatic method name generation [23, 25, 32, 33, 57, 88, 104]. Note that the scores of *BLEU*, *CodeBLEU*, *Pass@1*, *Exact Match*, and *Success rate* are in the range of [0,1]; the higher, the better. *Edit Distance* is measured in actual values; the smaller, the better.

The hyper-parameters are optimized according to actual performance and the values are summarized in Table 2. The first four rows mean the parameters of GA in RADAR-Attack and the following rows mean the parameters of model training and inference in RADAR-Defense. For the implementation of GA, we utilize the scikit-opt[2] library. For the implementation of RADAR-Defense, we utilize the Pytorch[3] and Transformers[4] libraries.

4.1.5 Experiment Platform. All the experiments were run on Intel(R) Xeon(R) Silver 4210 CPU and GeForce RTX3090 GPU with 24 GB memory. The operating system is Linux Debian.

4.2 Experimental Results

RQ1: How effective is RADAR-Attack in degrading the performance of FD$^{\text{Sig}}$ by attacking method names?

We investigate whether the existing FD$^{\text{Sig}}$ PCGMs are vulnerable to method name attacks, and in case they are, whether our defense method can reinstate their performance. As discussed in Section 4.1.2, we include three PCGMs, namely CodeGPT, PLBART, and CodeT5, for the fine-tuning code generation task. For the zero-shot code generation task, we consider three PCGMs, namely Replit, CodeGen, and CodeT5+. Here we consider four performance measures (i.e., BLEU, CodeBLEU,

[2]https://github.com/guofei9987/scikit-opt
[3]https://pytorch.org/
[4]https://github.com/huggingface/transformers

Table 3. Evaluation results of comparing RADAR and the baselines in terms of adversarial attack in the Java dataset

Model	Method	BLEU	CodeBELU	ASR
CodeGPT	FD	11.56	14.78	–
	FD^{Sig}	23.18	26.33	–
	Foo-Attack	16.95 (↓ 26.88%)	20.09 (↓ 23.70%)	55.40%
	Random-Attack	15.52 (↓ 33.05%)	19.82 (↓ 24.72%)	58.25%
	ALERT-Attack	13.85 (↓ 40.25%)	17.24 (↓ 34.52%)	65.52%
	Genetic-Attack	14.25 (↓ 38.52%)	17.88 (↓ 32.09%)	60.48%
	ReCODE-Attack	15.11 (↓ 34.81%)	18.48 (↓ 29.81%)	59.58%
	ACCENT-Attack	14.31 (↓ 38.27%)	17.60 (↓ 33.16%)	61.05%
	RADAR-Attack	**13.02 (↓ 43.83%)**	**16.13 (↓ 38.74%)**	**67.25%**
PLBART	FD	20.84	29.38	–
	FD^{Sig}	35.19	43.71	–
	Foo-Attack	27.47 (↓ 21.94%)	36.32 (↓ 16.91%)	56.15%
	Random-Attack	25.22 (↓ 28.33%)	33.67 (↓ 22.97%)	58.85%
	ALERT-Attack	23.52 (↓ 33.16%)	32.62 (↓ 25.37%)	63.58%
	Genetic-Attack	22.85 (↓ 35.07%)	31.52 (↓ 27.89%)	67.20%
	ReCODE-Attack	24.59 (↓ 30.12%)	32.98 (↓ 24.55%)	62.48%
	ACCENT-Attack	23.34 (↓ 33.67%)	32.53 (↓ 25.58%)	64.40%
	RADAR-Attack	**22.61 (↓ 35.75%)**	**31.31 (↓ 28.37%)**	**67.60%**
CodeT5	FD	20.53	30.43	–
	FD^{Sig}	38.45	46.09	–
	Foo-Attack	31.21 (↓ 18.83%)	37.83 (↓ 17.92%)	54.15%
	Random-Attack	28.74 (↓ 25.25%)	36.39 (↓ 21.05%)	59.10%
	ALERT-Attack	26.40 (↓ 31.34%)	34.16 (↓ 25.88%)	64.88%
	Genetic-Attack	25.45 (↓ 33.81%)	33.66 (↓ 26.97%)	67.52%
	ReCODE-Attack	25.87 (↓ 32.72%)	33.95 (↓ 26.34%)	66.21%
	ACCENT-Attack	25.81 (↓ 32.87%)	33.38 (↓ 27.58%)	66.25%
	RADAR-Attack	**24.48 (↓ 36.33%)**	**31.58 (↓ 31.48%)**	**74.65%**

Pass@1, Attack Success rate), which have been widely used in previous studies of neural code generation [2, 14, 16, 19, 57, 67, 80, 88] and adversarial example generation [6, 49, 81, 93, 94, 98, 104].

Table 3 and Table 4 show the evaluation results of these three victim models before and after the attacks for fine-tuning code generation tasks, respectively. The second column gives the used method. Columns 3–5 in Table 3 show the performance metrics for the Java dataset while columns 3–5 in Table 4 show the counterparts for the Python dataset. The rows marked by FD and FD^{Sig} show the performance of each PCGM when the signature is either excluded or included in the input. The following three rows show how the model performs under different adversarial attacks (i.e., with modified method names).

From this table, we can first observe that the performance of the code generation with FD^{Sig} is consistently better than that with FD, in terms of all the metrics. For instance, for the CodeT5 model, on the Java dataset, in terms of both BLEU and CodeBLEU, the code generation with FD^{Sig} performs nearly 1.5 times better than with FD. On the Python dataset, the code generation with FD^{Sig} performs nearly four times better than with FD in BLEU performance and nearly twice as

Table 4. Evaluation results of comparing RADAR and the baselines in terms of adversarial attack in the Python dataset

Model	Method	BLEU	CodeBELU	ASR
CodeGPT	FD	5.06	18.77	–
	FD^{Sig}	11.94	24.27	–
	Foo-Attack	9.02 (↓ 24.46%)	22.10 (↓8.94%)	56.05%
	Random-Attack	8.11 (↓ 32.08%)	20.88 (↓ 13.97%)	56.55%
	ALERT-Attack	7.94 (↓ 33.50%)	18.47 (↓ 23.90%)	61.20%
	Genetic-Attack	7.48 (↓ 37.35%)	18.32 (↓ 24.52%)	60.50%
	ReCODE-Attack	7.92 (↓ 33.67%)	19.12 (↓ 21.22%)	59.28%
	ACCENT-Attack	7.65 (↓ 35.93%)	18.58 (↓ 23.44%)	60.00%
	RADAR-Attack	**7.09 (↓ 40.62%)**	**17.86 (↓ 26.41%)**	**63.20%**
PLBART	FD	7.85	20.60	–
	FD^{Sig}	19.99	30.12	–
	Foo-Attack	16.93 (↓ 15.31%)	26.13 (↓ 13.25%)	56.15%
	Random-Attack	14.39 (↓ 28.01%)	25.89 (↓ 14.04%)	57.95%
	ALERT-Attack	14.21 (↓ 28.91%)	25.24 (↓ 16.20%)	60.55%
	Genetic-Attack	13.68 (↓ 31.57%)	24.98 (↓ 17.07%)	63.85%
	ReCODE-Attack	14.63 (↓ 26.81%)	25.85 (↓ 14.18%)	57.80%
	ACCENT-Attack	**13.00 (↓ 34.97%)**	24.61 (↓ 18.29%)	62.35%
	RADAR-Attack	13.31 (↓ 33.42%)	**24.18 (↓ 19.72%)**	**65.80%**
CodeT5	FD	5.35	19.11	–
	FD^{Sig}	21.69	33.26	–
	Foo-Attack	19.37 (↓ 10.70%)	29.23 (↓ 12.12%)	53.50%
	Random-Attack	15.11 (↓ 30.34%)	27.59 (↓ 17.05%)	58.95%
	ALERT-Attack	14.59 (↓ 32.73%)	26.53 (↓ 20.23%)	64.75%
	Genetic-Attack	13.84 (↓ 36.19%)	25.68 (↓ 22.79%)	69.50%
	ReCODE-Attack	14.21 (↓ 34.49%)	25.94 (↓ 22.01%)	68.50%
	ACCENT-Attack	13.57 (↓ 37.44%)	25.04 (↓ 24.71%)	71.00%
	RADAR-Attack	**13.23 (↓ 39.00%)**	**24.52 (↓ 26.28%)**	**72.80%**

well as in CodeBLEU performance. In short, the code generation with FD^{Sig} performs nearly twice as well as with FD in most cases.

Furthermore, we observe that all the PCGMs are vulnerable to adversarial attacks in the fine-tuning code generation task, as their performance decreases largely when the method names are modified. However, the impact of adversarial attacks varies across these models. Among them, the simplest foo-Attack can cause 9%-27% performance degradation in code generation on the test set for all three models. In addition, well-designed attacks (such as ACCENT-Attack and RADAR-Attack) can have a more severe impact on the model performance.

Take the CodeT5 model as an example, RADAR-Attack degrades its BLEU and CodeBLEU performance on the Java dataset by 36.33% and 31.58% respectively, and can successfully attack 74.65% of the test set samples. On the Python dataset, the CodeT5's BLEU and CodeBLEU performance is degraded by 39.00% and 26.28% respectively, and RADAR-Attack can successfully attack 72.80% of the test set samples.

Table 5. Evaluation results of comparing RADAR and the baselines in terms of adversarial attack in the Human-Eval dataset

Model	Method	BLEU	CodeBELU	Pass@1	ASR
Replit	FD	–	–	–	–
	FD^{Sig}	28.56	29.98	18.90	–
	Foo-Attack	**25.48 (↓ 10.78%)**	**27.73 (↓ 7.51%)**	15.85 (↓ 16.14%)	29.03%
	Random-Attack	26.26 (↓ 8.05%)	28.99 (↓ 3.30%)	16.46 (↓ 12.91%)	25.81%
	ALERT-Attack	26.24 (↓ 8.12%)	29.21 (↓ 2.57%)	14.02 (↓ 25.82%)	32.26%
	Genetic-Attack	26.50 (↓ 7.21%)	29.14 (↓ 2.80%)	15.24 (↓ 19.37%)	29.03%
	ReCODE-Attack	26.40 (↓ 7.56%)	28.62 (↓ 4.54%)	15.85 (↓ 16.14%)	25.81%
	ACCENT-Attack	25.90 (↓ 9.31%)	28.36 (↓ 5.40%)	13.41 (↓ 29.05%)	35.48%
	RADAR-Attack	25.87 (↓ 9.42%)	28.27 (↓ 5.70%)	**12.80 (↓ 32.28%)**	**45.16%**
CodeGen	FD	–	–	–	–
	FD^{Sig}	30.18	33.01	21.34	–
	Foo-Attack	30.71 (↑ 1.76%)	32.48 (↓ 1.61%)	17.68 (↓ 17.15%)	25.71%
	Random-Attack	28.12 (↓ 6.83%)	31.80 (↓ 3.67%)	15.24 (↓ 28.58%)	42.86%
	ALERT-Attack	26.71 (↓ 11.50%)	29.75 (↓ 9.88%)	14.02 (↓ 34.30%)	45.71%
	Genetic-Attack	28.76 (↓ 4.71%)	30.89 (↓ 6.42%)	13.41 (↓ 37.16%)	37.14%
	ReCODE-Attack	28.90 (↓ 4.24%)	30.96 (↓ 6.21%)	18.90 (↓ 11.43%)	20.00%
	ACCENT-Attack	27.70 (↓ 8.22%)	30.19 (↓ 8.54%)	14.02 (↓ 34.30%)	42.86%
	RADAR-Attack	**26.51 (↓ 12.16%)**	**28.44 (↓ 13.84%)**	**12.20 (↓ 42.83%)**	**51.43%**
CodeT5+	FD	–	–	–	–
	FD^{Sig}	27.21	30.92	21.95	–
	Foo-Attack	25.75 (↓ 5.37%)	29.10 (↓ 5.89%)	20.73 (↓ 5.56%)	25.00%
	Random-Attack	25.63 (↓ 5.81%)	29.31 (↓ 5.21%)	16.46 (↓ 25.01%)	36.11%
	ALERT-Attack	24.18 (↓ 11.14%)	26.88 (↓ 13.07%)	13.41 (↓ 38.91%)	44.44%
	Genetic-Attack	**23.35 (↓ 14.19%)**	**26.04 (↓ 15.78%)**	13.41 (↓ 38.91%)	44.44%
	ReCODE-Attack	24.89 (↓ 8.53%)	27.58 (↓ 10.80%)	18.29 (↓ 16.67%)	25.00%
	ACCENT-Attack	24.13 (↓ 11.32%)	26.63 (↓ 13.87%)	14.63 (↓ 33.35%)	47.22%
	RADAR-Attack	26.51 (↓ 2.57%)	28.48 (↓ 7.89%)	**12.20 (↓ 44.42%)**	**50.00%**

Table 5 presents the evaluation results of three victim models (Replit, CodeGen, and CodeT5+) before and after the attacks in the zero-shot code generation task. Similar to the findings in the fine-tuning code generation task, it is evident that all PCGMs are susceptible to adversarial attacks, resulting in significant performance degradation when method names are modified. In our experiments, we observed that in certain cases, the model generated incorrect code based on the original prompt but made correct predictions when presented with perturbed prompts, which aligns with the findings of Wang et al. [84]. To accurately evaluate the ASR, we computed the ratio of samples where the model correctly generated code based on the original prompt but made incorrect predictions on perturbed prompts, to the total number of samples where the model correctly generated code based on the original prompt. Using the CodeGen model as an example, the RADAR-Attack method leads to a reduction in BLEU and CodeBLEU performance by 12.16% and 13.84%, respectively. Moreover, it successfully attacks 51.43% of the samples in the test set. These results highlight the vulnerability of PCGMs to adversarial attacks, emphasizing the importance of robust defense mechanisms in code generation tasks.

All the existing attack methods, including our proposed RADAR-Attack, have a detrimental impact on the performance of Replit, CodeGen, and CodeT5+ PCGMs, particularly in terms of the Pass@1 metric. However, in contrast to the PCGMs used in the fine-tuning code generation task,

Table 6. Evaluation results of comparing RADAR and the baselines in terms of attack and defense

Model	Method	Java		Python	
		BLEU	CodeBLEU	BLEU	CodeBLEU
CodeGPT	FD^{Sig}	**23.18**	**26.33**	11.94	24.27
	RADAR-Attack	13.02	16.13	7.09	17.86
	ACCENT-Defense	17.95	20.90	9.20	21.61
	RADAR-Defense	22.15	25.45	**12.54**	**24.44**
PLBART	FD^{Sig}	35.19	**43.71**	**19.99**	30.12
	RADAR-Attack	22.61	31.31	13.31	24.18
	ACCENT-Defense	27.57	36.24	14.49	26.52
	RADAR-Defense	**35.84**	43.61	19.64	**30.88**
CodeT5	FD^{Sig}	38.45	46.09	**21.69**	**33.26**
	RADAR-Attack	24.28	31.58	13.23	24.52
	ACCENT-Defense	30.31	37.43	16.01	27.22
	RADAR-Defense	**39.29**	**46.11**	21.31	32.90

these models (Replit, CodeGen, and CodeT5+) do not exhibit significant differences in token-level similarity metrics such as BLEU and CodeBLEU. The lack of substantial differentiation in token-based similarity metrics can be attributed to the gap between these metrics and execution-based metrics. As a result, the impact of RADAR-Attack on the CodeT5+ model, for example, only leads to a modest degradation of 2.57% in BLEU and 7.89% in CodeBLEU. Nonetheless, RADAR-Attack successfully attacks 50.00% of the samples in the test set. These findings highlight the limitations of token-level similarity metrics when assessing the robustness of PCGMs and emphasize the need to consider execution-based metrics for a comprehensive evaluation.

In general, we have observed that the ASR performance of RADAR-Attack is optimal across all datasets and victim models. Specifically, on the Java dataset, the ASR performance of RADAR-Attack is, on average, 4.40% higher than the second best baseline method. On the Python dataset, the ASR performance of RADAR-Attack is, on average, 2.96% higher than the second best baseline method. On the Human-Eval dataset, the ASR performance of RADAR-Attack is, on average, 17.73% higher than the second best baseline method. It is worth mentioning that since the Java dataset and the Python dataset do not support the calculation of the Pass@1 metric, we calculated the ASRs on these two datasets by reducing the CodeBLEU value. However, this method may not be as accurate as the Human-Eval dataset in terms of semantic consistency. Considering the significant improvement in performance on the Human-Eval dataset, it can be concluded that RADAR-Attack has a substantial impact on the ASR performance.

Summary for RQ1

Existing PCGMs are generally vulnerable to adversarial attacks on method names both in fine-tuning and zero-shot code generation tasks, which shows that the quality of the method names in the signature is crucial for PCGMs. In general, RADAR-Attack is the most effective method in attacking the models.

RQ2: How effective is RADAR-Defense in reinstating the performance of FD^{Sig}?

Table 7. Evaluation results of comparing RADAR and the baselines in terms of attack and defense

Model	Method	BLEU	CodeBLEU	Pass@1
Replit	FD^{Sig}	**28.56**	29.98	**18.90**
	RADAR-Attack	25.87	28.27	12.80
	ACCENT-Defense	–	–	–
	RADAR-Defense	28.51	**30.21**	18.29
CodeGen	FD^{Sig}	**30.18**	**33.01**	21.34
	RADAR-Attack	26.51	28.44	12.20
	ACCENT-Defense	–	–	–
	RADAR-Defense	29.95	32.99	**21.95**
CodeT5+	FD^{Sig}	**27.21**	**30.92**	**21.95**
	RADAR-Attack	26.51	28.48	12.20
	ACCENT-Defense	–	–	–
	RADAR-Defense	26.94	30.04	20.12

Table 6 summarizes evaluation results on the three victim models of the two defense strategies for fine-tuning code generation task. Rows of FD^{Sig} and RADAR-Attack recapitulate the performance of PCGMs when the method name is unattacked or attacked respectively, followed by two rows showing how the model performs under the two different defense strategies.

In terms of defense, we find that the mask training employed in ACCENT-Defense can indeed resist some attack examples, mainly because the mask training masks the attacked method name and lets the model learn the corresponding code generation after the mask. Compared to ACCENT-Defense, RADAR-Defense is a passive defense method to sanitize the input, and the performance of the defended model is almost the same as that of the original environment (e.g., CodeT5 has a BLEU metric of 21.69 on the Python dataset, and the metric drops to 13.23 after being attacked by RADAR-Attack, but after RADAR-Defense the metric reinstates to 21.31) Moreover, we are surprised to observe that some models can slightly improve their code generation performance after defending the method names in the signature. For instance, CodeT5's performance measured in BLEU and CodeBLEU is improved by 61.82% and 46.01% respectively, by RADAR-Defense on the Java dataset, when compared with that of the attacked model. ACCENT-Defense, on the other hand, only improved 24.84% of the BLEU performance and 18.52% of the CodeBLEU performance. These results show that the defense of RADAR-Defense is superior. Indeed, RADAR-Defense even exceeds the performance of the original methods on some combinations (e.g., CodeBLEU in Python using CodeGPT, BLEU in Java and CodeBLEU in Python using PLBART, and both BLEU and CodeBLEU in Java using CodeT5). It also indicates that the quality of method names in the signature is crucial for the model to generate code.

In the zero-shot code generation task, since the PCGMs are not fine-tuned on the HumanEval dataset, an approach based on active defense is not suitable for this scenario. Table 7 provides a summary of the evaluation results for the three victim models under our defense method in the zero-shot code generation task. Consistent with the findings from the fine-tuning code generation task, the defended models exhibit performance that is nearly equivalent to the original environment. Furthermore, we observe that some models can experience slight improvements in their code generation performance after defending the method names in the signatures. For example, CodeGen's Pass@1 metric increases from 21.34 in the original environment to 21.95 in the RADAR-Defense.

These results highlight the significance and advantages of employing well-chosen method names in neural code generation, both in the fine-tuning and zero-shot code generation tasks.

In general, we observe that our proposed RADAR-Defense method is a passive defense approach that ensures both clean performance and robustness of the model without the need for retraining. Therefore, our RADAR-Defense method provides a viable way that enhances model robustness without sacrificing clean performance. This passive defense approach has certain advantages over active defense methods, especially in scenarios with high costs and limitations in zero-shot scenarios.

Summary for RQ2

RADAR-Defense, as a passive defense method, shows better defense performance and is capable of bringing the performance of FD^{Sig} back. As well, it also shows that the quality of the method names in the signature is crucial for PCGMs.

RQ3: How effective is our proposed RADAR-Defense in terms of method name generation?

Results of RQ1 and RQ2 demonstrate the importance of method names in neural code generation. In RQ3, we investigate whether our method can synthesize high-quality method names for programmers. Note that for our zero-shot evaluation in the Human-Eval task, we utilize the model trained by RADAR-Defense on the Python dataset that we collected in Section 4.1.1.

For the baselines with shared code (e.g., NNGen, CCGIR, CodeBERT, UniXcoder, Rencos, and REINA), we directly used their implementation to obtain the optimal values of parameters and trained the models. Otherwise (e.g., BM25 and RNN-Att-Copy), we replicated them according to the description of the original studies.

Table 8. Evaluation results of comparing RADAR-Defense with the baselines for the Java dataset

Type	Method	EM	BLEU	ED
Information Retrieval	BM25	22.00	42.24	9.39
	NNGen	23.65	45.93	8.93
	CCGIR	23.50	46.97	8.71
	RADAR-IR	24.10	46.66	8.70
Deep Learning	RNN-Att-Copy	22.20	47.99	8.37
	CodeBERT	40.95	63.76	6.13
	UniXcoder	43.35	65.66	5.99
IR-Enhanced	Rencos	27.75	53.53	7.39
	REINA	41.00	63.51	6.39
	RADAR-Defense	**47.60**	**68.86**	**5.28**

Table 8, Table 9, and Table 10 show the results of RADAR-Defense and the baselines for the Java, Python, and Human-Eval datasets respectively. The second column of the tables shows the considered baselines. Columns 3–5 show the results of the performance metrics.

First, when comparing RADAR-Defense with the information retrieval baselines, we observe that, since CCGIR uses dense vectors for retrieval while both BM25 and NNGen use sparse vectors for retrieval, CCGIR performs slightly better than BM25 and NNGen on both datasets. Then CodeBERT used by CCGIR for semantic vectorization representation will take more time, and our

Table 9. Evaluation results of comparing RADAR-Defense with the baselines for the Python dataset

Type	Method	EM	BLEU	ED
Information Retrieval	BM25	14.50	31.39	10.68
	NNGen	14.75	32.00	10.42
	CCGIR	15.20	32.62	10.34
	RADAR-IR	15.10	34.58	9.98
Deep Learning	RNN-Att-Copy	11.60	37.66	9.29
	CodeBERT	25.35	50.18	7.58
	UniXcoder	27.40	52.46	7.67
IR-Enhanced	Rencos	17.55	39.63	9.12
	REINA	25.35	49.98	7.93
	RADAR-Defense	**32.60**	**57.56**	**6.65**

Table 10. Evaluation results of comparing RADAR-Defense with the baselines for the Human-Eval dataset

Type	Method	EM	BLEU	ED
Information Retrieval	BM25	0.61	7.90	13.42
	NNGen	0.61	4.98	12.95
	CCGIR	0.00	4.66	12.84
	RADAR-IR	1.22	10.05	12.43
Deep Learning	RNN-Att-Copy	1.22	9.71	11.07
	CodeBERT	14.63	32.33	8.22
	UniXcoder	29.88	46.62	7.24
IR-Enhanced	Rencos	7.58	18.45	10.14
	REINA	22.81	42.60	8.19
	RADAR-Defense	**32.93**	**49.62**	**6.09**

proposed information retrieval method can achieve better performance in less time, showing that our proposed method's information retrieval part is effective.

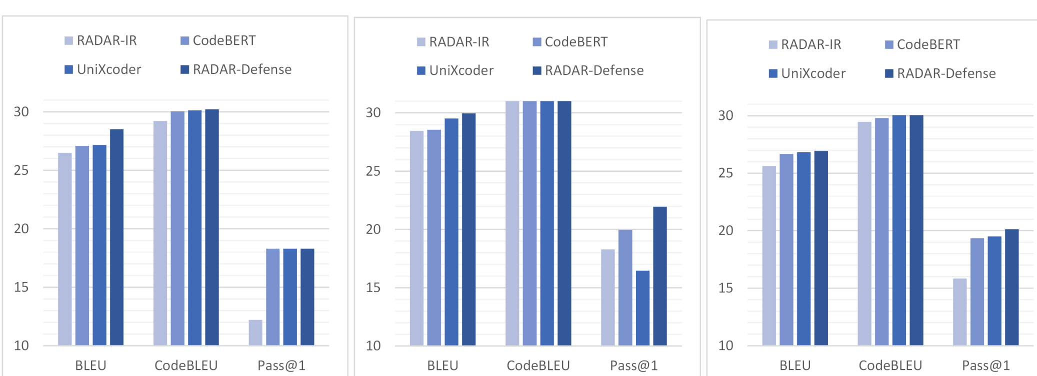
Second, when comparing RADAR-Defense with the deep learning baselines, we find that among all the deep learning baselines, RADAR-Defense has the best performance.

Last, results of comparing the hybrid baselines with our method show that RADAR-Defense can largely improve the performance of the methods. More specifically, compared to the best-performing baseline UniXcoder, on the Java dataset, RADAR-Defense improves the EM, BLEU, and ED performances by 9.80%, 4.87%, and 11.85% respectively; on the Python dataset, RADAR-Defense improves the EM, BLEU, and ED performances by 18.98%, 9.72%, and 12.27%, respectively; on the Human-Eval dataset, RADAR-Defense improves the EM, BLEU, and ED performances by 9.26%, 6.44%, and 15.88%, respectively.

To further investigate the component setting rationality of our proposed method RADAR-Defense, we carry out an ablation study. We have considered five variants through permutations between components. The experimental results are given in Table 11 and show that the inclusion of each component is reasonable. The most significant impact on model performance among these three components is our proposed prompt method. With the same settings for the remaining two components, adding the prompt will give RADAR-Defense a more substantial performance boost.

Table 11. Ablation experiments between three components

Dataset	IR	Prompt	In_trust Loss	EM	BLEU	ED
Java	–	–	–	43.35	65.66	5.99
	–	–	✓	43.75	66.07	5.90
	✓	–	–	43.45	66.04	5.83
	✓	–	✓	43.55	66.27	5.83
	✓	✓	–	47.10	67.70	5.34
	✓	✓	✓	**47.60**	**68.86**	**5.28**
Python	–	–	–	27.40	52.46	7.67
	–	–	✓	28.30	52.77	7.52
	✓	–	–	27.60	53.05	7.23
	✓	–	✓	28.40	53.69	7.33
	✓	✓	–	**32.60**	56.74	6.76
	✓	✓	✓	**32.60**	**57.56**	**6.65**
Human-Eval	–	–	–	29.88	46.62	7.24
	–	–	✓	29.88	46.23	6.95
	✓	–	–	30.58	47.85	6.88
	✓	–	✓	31.05	48.11	6.56
	✓	✓	–	32.76	49.11	6.27
	✓	✓	✓	**32.93**	**49.62**	**6.09**



(a) Evaluation on the Replit (b) Evaluation on the CodeGen (c) Evaluation on the CodeT5+

Fig. 7. The impact of the quality of generated method names on the robustness improvement of PCGMs

Furthermore, we conduct an investigation into the impact of data quality on the improvement of robustness. In the zero-shot code generation task, we generate method names using RADAR-IR, CodeBERT, UniXcoder, and RADAR-Defense. These methods for generating method names demonstrate increasing performance in the method name generation task. As depicted in Fig. 7, we observe a correlation between the quality of the generated data and the improvement in robustness. Across all three models, we notice that the BLEU and CodeBLEU metrics improve as the quality of the generated data increases. Moreover, in most cases, the Pass@1 metric also shows improvement as the quality of the generated data increases. These experimental findings further highlight the importance of utilizing high-quality method names in neural code generation tasks.

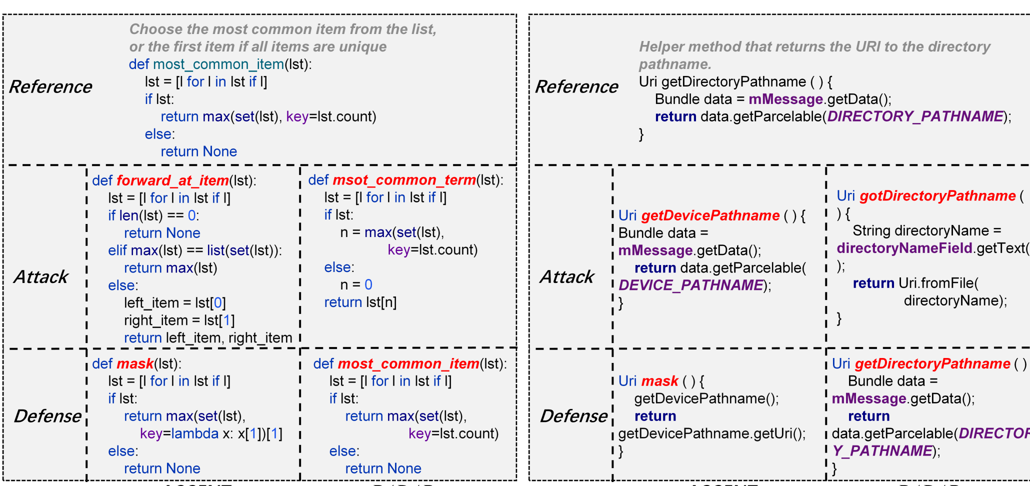
In general, we observe that our proposed RADAR-Defense method is ability to generate method names that are closer to the golden truth and the method names generated by RADAR-Defense can improve the accuracy of code generation by PCGMs. The success of RADAR-Defense can be attributed to the following factors: (1) the choice of the base model: UniXcoder. UniXcoder has demonstrated the best performance among existing baselines, making it a strong foundation for RADAR-Defense; (2) the retrieval-enhanced prompt learning method and the application of the In_trust loss, which are reflected in the ablation experiments presented in Table 11.

Summary for RQ3

RADAR-Defense can achieve better performance than eight state-of-the-art baselines of three different types. In our ablation study, the prompt component demonstrates the most influence on the performance of the method. More importantly, the quality of the method names also impacts the robustness improvement.

5 DISCUSSION

5.1 Qualitative Analysis



(a) An example in the Python Dataset

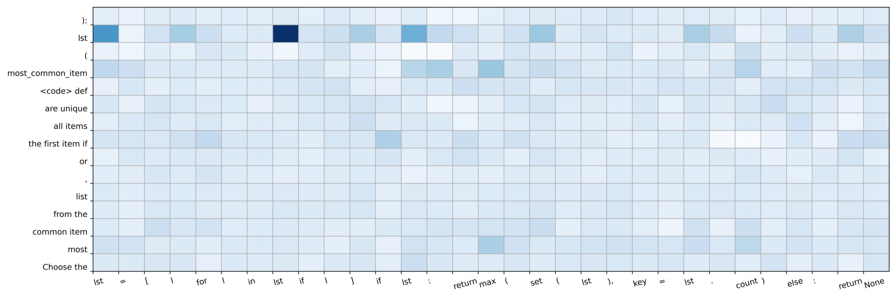
(b) An example in the Java Dataset

Fig. 8. Two examples of generated code by CodeT5 when attacked and defended by RADAR and ACCENT

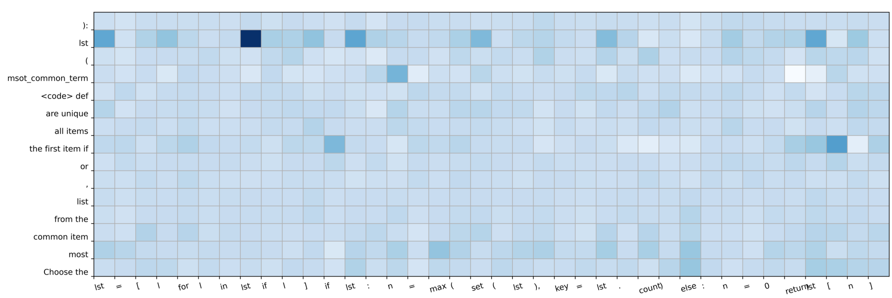
In Section 4.2, we design three RQs to provide a quantitative study of the effectiveness of conducted performance comparisons between RADAR and baselines automatically in terms of performance measures. However, these performance measures may not truly reflect the semantic similarity [78]. To further demonstrate the effectiveness of RADAR, we conduct qualitative analysis.

Examples in Robustness of Pre-trained Code Generation. For the fine-tuning code generation task, we give a Python example based on a real-world project[5] and a Java example based on a

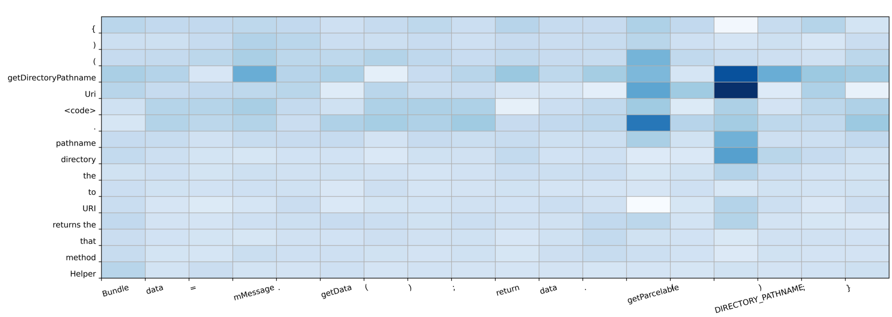
[5]https://pypi.org/project/spirit/2.1.1/



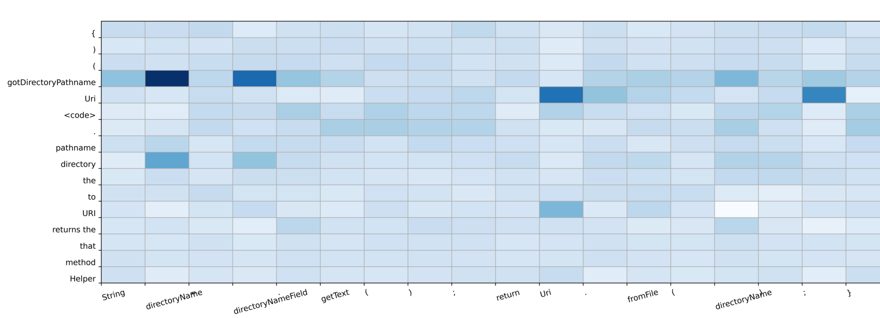
(a) Heat map before being attacked in Python example code



(b) Heat map after being attacked in Python example code



(c) Heat map before being attacked in Java example code



(d) Heat map after being attacked in Java example code

Fig. 9. Explore the effect of method names on the Python example code generated by CodeT5 before and after being attacked

real-world project[6] using the CodeT5 model. Fig. 8 shows these two examples of generated code by CodeT5 when attacked and defended by RADAR and ACCENT. The first row gives the functional description, signature, and reference code, where the generated code by CodeT5 is the same as the reference code. The second row shows adversarial examples generated by RADAR-Attack and ACCENT-Attack while the third row shows the effectiveness of two defensive methods.

From Fig. 8 (a), we can see that the original method name is `most_common_item`. The adversarial example `forward_at_item` generated by ACCENT-Attack is based on semantic similarity, which is not as natural as `msot_common_term` generated by RADAR-Attack, in which "msot" is generated by the **Swap** operator and "term" is generated by the **Replace-sem** operator.

From the Fig. 8(b), we can see that the original method name is `getDirectoryPathname`. ACCENT-Attack generates `getDevicePathname` as the adversarial method name based on semantic similarity, which is arguably not as natural as `gotDirectoryPathname` generated by RADAR-Attack, in which "got" is generated by the **Replace-sem** operator.

The code generated by RADAR-Attack in the above two examples can cause functional errors that can lead to the failure of PCGMs. This demonstrates the effectiveness of our RADAR-Attack, and that the robustness issue in PCGMs needs to be addressed properly.

We also explore the effectiveness of two defensive methods. ACCENT-Defense replaces the method name with $\langle mask \rangle$ and then feeds it into the mask learned model and generates the corresponding code. In contrast, RADAR-Defense synthesizes method names based on functional descriptions, replaces them in the adversarial examples, and then generates the corresponding code by the model. Two examples in Fig. 8 show that RADAR-Defense is capable of generating the correct method names, and the code generated by CodeT5 after being defended by RADAR-Defense can be reinstated to what it was before being attacked.

Moreover, in order to explore the effect of method names on the code generated by CodeT5 before and after being attacked, we visualize and analyze them with the SHAP tool.[7] In contrast to the work on model interpretation based on attention weight visualization, SHAP is based on game theory, which defines the additive feature attribution method and guarantees a unique solution. Research [58] shows that SHAP is similar to human intuition measurement and more effective.

Fig. 9 visualize the Python code and Java code in Fig. 8, as a way to analyze the effect of method names on the code generated by CodeT5 before and after being attacked. In Fig. 9(a), before being attacked, the normal method name "most_common_item" can lead CodeT5 to generate the correct code; from this heat map one can see that it has a greater impact on the tokens "max" and "count" of the code snippet. However, in Fig. 9(b), under the attack of "msot_common_term", CodeT5 generates semantically incorrect code, and the heat map shows that this method name only has a large effect on token "n" in the code snippet. Likewise, in Fig. 9(c), before being attacked, the normal method name "getDirectoryPathname" can lead CodeT5 to generate the correct code and it has a greater impact on tokens "getParcelable" and "DIRECTORY_PATHNAME" of the code snippet. However, in Fig. 9(d), under the attack of "gotDirectoryPathname", CodeT5 outputs incorrect code.

In the zero-shot code generation task, we provide an example from the Human-Eval dataset, specifically using the CodeGen model. In Fig. 10, we showcase the generated code snippets under various attacks as well as the RADAR-Defense approach. Additionally, we provide information on whether the generated code can be successfully compiled given the provided test cases. Initially, the original method name is `median`, and CodeGen can generate the correct code when this method name is utilized. However, when the method name is changed to `foo` or an adversarial method name generated by different attack methods, the code generated by CodeGen either fails the

[6]https://github.com/douglascraigschmidt/POSA-15
[7]https://github.com/slundberg/shap

Raw

```
def median(l: list):
  """Return median of elements in the
list l.  """
```

```
  l.sort()
  n = len(l)
  if n % 2 == 1:
    return l[n//2]
  else:
    return (l[n//2-1] + l[n//2])/2
```

passed

Foo-Attack

```
def foo(l: list):
  """Return median of elements in the
list l.  """
```

```
  if len(l) % 2 == 0:
    return (l[len(l) // 2 - 1]
 + l[len(l) // 2]) / 2
  else:
    return l[len(l) // 2]
```

failed

Random-Attack

```
def decimal_to_binary(l: list):
  """Return median of elements in the
list l.  """
```

```
  return median_of_three(l, 0,
len(l) - 1)
```

name 'median_of_three' is not defined

ALERT-Attack

```
def len(l: list):
  """Return median of elements in the
list l.  """
```

```
return median(l)
```

failed: name 'median' is not defined

Genetic-Attack

```
def incomes (l: list):
  """Return median of elements in the
list l.  """
```

```
return median(l)
```

name 'median' is not defined

ReCODE-Attack

```
def medlan(l: list):
  """Return median of elements in the
list l.  """
```

```
 return sum(l) / len(l)
```

failed

ACCENT-Attack

```
def variance(l: list):
  """Return median of elements in the
list l.  """
```

```
return median(l)
```

name 'median' is not defined

RADAR-Attack

```
def media(l: list):
  """Return median of elements in the
list l.  """
```

```
return median(l)
```

name 'median' is not defined

RADAR-Defense

```
def calculate_median(l: list):
  """Return median of elements in the
list l.  """
```

```
  l.sort()
  if len(l) % 2 == 0:
    return (l[len(l) // 2] + l[len(l) /
/ 2 - 1]) / 2
  else:
    return l[len(l) // 2]
```

passed

Fig. 10. One example of generated code by CodeGen under various attacks as well as the RADAR-Defense in Human-Eval dataset

test cases or contains syntax errors. In contrast, RADAR-Defense synthesizes the method name `calculate_median` based on functional descriptions, replaces it in the adversarial examples, and subsequently, CodeGen is able to generate the corresponding code that aligns with the desired functionality.

Examples in Method Name Generation. To further explore the quality of the method names synthesized by RADAR-Defense, we select three examples from the Java dataset, the Python dataset, and the Human-Eval dataset respectively for analysis in Table 12. In these samples, we find RADAR-Defense can synthesize more-accurate method names than baselines when compared with human-written method names.

5.2 Threats to Validity

Internal threats. Internal threats refer to the potential defects in implementing our proposed approach and baselines. To alleviate this, we double-checked and peer-reviewed our code to ensure the fairness of the results. For all PCGMs, we used their publicly available models. For the attack baselines and method name generation baselines, we ran their open-source code directly or re-implemented them according to the original studies.

External threats. External threats refer to the choice of corpora and PCGMs. To alleviate this, we collected two datasets based on well-maintained open-source projects with high reputations according to the relevant heuristic rules for fine-tuning code generation tasks. For the zero-shot code generation task, we select the Human-Eval dataset. To ensure a fair comparison, we follow the settings from a previous study [42] when dividing the dataset. In terms of the choice of PCGMs,

Table 12. Examples of synthesized method name by RADAR-Defense and baselines in both Java and Python dataset

Case	Example
Java	Parse the string as a websocket request and return the value from WebSocket-Protocol header (See RFC 6455). Return empty string if not found.
	BM25: getClientWebSocketOrigin **NNGen:** getClientWebSocketOrigin **CCGIR:** getClientWebSocketOrigin **RNN-Att-Copy:** parseValue **CodeBert:** getWebsocketRequest **UniXcoder:** getWebsocketHeader **Rencos:** getClientWebSocketOrigin **REINA:** getProtocol **RADAR-Defense:** getClientWebSocketProtocol **Human Written:** getClientWebSocketProtocol
Python	Returns an * RGBA * tuple of 4 ints from 0 - 255
	BM25: to_rgb_255 **NNGen:** to_rgb_255 **CCGIR:** to_rgb_255 **RNN-Att-Copy:** format_rgba **CodeBert:** to_rgb_255 **UniXcoder:** to_rgb_255 **Rencos:** to_rgb_255 **REINA:** rgba4 **RADAR-Defense:** to_rgba_255 **Human Written:** to_rgba_255
Human-Eval	Check if in given list of numbers, are any two numbers closer to each other than given threshold.
	BM25: are_rooms_adjacent **NNGen:** connected_pair **CCGIR:** connected_pair **RNN-Att-Copy:** format_rgba **CodeBert:** is_numbers **UniXcoder:** are_adjacent **Rencos:** are_rooms_adjacent **REINA:** are_adjacent **RADAR-Defense:** is_closer **Human Written:** has_close_elements

we select three state-of-the-art models (CodeGPT, PLBART, and CodeT5) for the fine-tuning code generation task, and three state-of-the-art models (Replit, CodeGen, and CodeT5+) for the zero-shot code generation task. For other models, such as CodePilot, they have not made models or API interfaces publicly available, and can only be accessed through plugins, which is not suitable for large-scale empirical research. While ChatGPT does offer an API interface, its output is not deterministic, resulting in low reproducibility. As a result, these models were not included in our selection.

Construct threats. Construct threats concern the performance metrics used to evaluate RADAR and baselines. We use a set of metrics, which are also commonly used in similar studies. Due to the difference between natural languages and programming languages, we evaluated the quality primarily through CodeBLEU for fine-tuning code generation tasks. CodeBLEU has been widely used in the previous studies of code generation, which can not only consider the surface match similar to the original BLEU but also the grammatical correctness and the logic correctness, leveraging the abstract syntax tree and the data flow structure. For the zero-shot code generation task, we choose Pass@1 as the main metric.

6 CONCLUSION

We studied the role of method names in neural code generation from a robustness perspective. We showed that most PCGMs using both the functional description and method signature as input, albeit demonstrating impressive performance, are fragile with respect to the input method names, meaning that an ill-formed name may degrade their performance largely. We proposed approaches to synthesize method names from the functional description which can be utilized to reinstate the performance of PCGMs.

For future work, we plan to investigate the robustness of (now widely-adopted) deep learning models in software engineering systemically. This would shed light on, for instance, the performance and interpretability of these models in solving challenging SE tasks. We also plan to investigate the influence of natural language descriptions and parameter lists on the performance of PCGMs, and identify suitable defense mechanisms to enhance their robustness.

ACKNOWLEDGMENTS

The authors are grateful for the valuable feedback from domain experts. This work was partially supported by the National Natural Science Foundation of China (NSFC, No. 61972197 and No. 62372232), the Natural Science Foundation of Jiangsu Province (No. BK20201292), the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (No. KYCX23_0396). T. Chen is partially supported by an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03) and Birkbeck BEI School Project (EFFECT).

REFERENCES

[1] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. JuICe: A Large Scale Distantly Supervised Dataset for Open Domain Context-based Code Generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP).* 5436–5446.

[2] Wasi 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.* 2655–2668.

[3] Akiko Aizawa. 2003. An information-theoretic perspective of tf–idf measures. *Information Processing & Management* 39, 1 (2003), 45–65.

[4] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[5] 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.* 2890–2896.

[6] Leonhard Applis, Annibale Panichella, and Arie van Deursen. 2021. Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations. In *2021 36th IEEE/ACM International Conference on*

Automated Software Engineering (ASE). IEEE, 1377–1381.

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[8] Mehdi Bahrami, NC Shrikanth, Shade Ruangwan, Lei Liu, Yuji Mizobuchi, Masahiro Fukuyori, Wei-Peng Chen, Kazuki Munakata, and Tim Menzies. 2021. Pytorrent: A python library corpus for large-scale language models. *arXiv preprint arXiv:2110.01710* (2021).

[9] Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. In *International Conference on Machine Learning*. PMLR, 896–907.

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. 2018. JAX: Composable Transformations of Python+ NumPy Programs (v0. 2.5). *Software available from https://github. com/google/jax* (2018).

[11] Deng Cai, Yan Wang, Wei Bi, Zhaopeng Tu, Xiaojiang Liu, and Shuming Shi. 2019. Retrieval-guided dialogue response generation via a matching-to-generation framework. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 1866–1875.

[12] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *24th USENIX security symposium (USENIX Security 15)*. 255–270.

[13] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*. Ieee, 39–57.

[14] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 18–30. https://doi.org/10.1145/3540250.3549162

[15] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests. *arXiv preprint arXiv:2207.10397* (2022).

[16] 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).

[17] 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).

[18] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. 2022. PanGu-Coder: Program Synthesis with Function-Level Language Modeling. *arXiv preprint arXiv:2207.11280* (2022).

[19] Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 9052–9065.

[20] ISC Committee et al. 1990. IEEE standard glossary of software engineering terminology (IEEE Std 610.12-1990). Los Alamitos. *CA IEEE Comput. Soc* (1990).

[21] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[22] Yao Deng, Xi Zheng, Tianyi Zhang, Chen Chen, Guannan Lou, and Miryung Kim. 2020. An analysis of adversarial attacks and defenses on autonomous driving models. In *2020 IEEE international conference on pervasive computing and communications (PerCom)*. IEEE, 1–10.

[23] 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.

[24] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[25] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. 2019. A neural model for method name generation from functional description. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 414–421.

[26] Nicolás García-Pedrajas, Domingo Ortiz-Boyer, and César Hervás-Martínez. 2006. An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization. *Neural Networks* 19, 4 (2006), 514–528.

[27] Fan Ge and Li Kuang. 2021. Keywords guided method name generation. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 196–206.

[28] 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

[29] Gaurav Goswami, Nalini Ratha, Akshay Agarwal, Richa Singh, and Mayank Vatsa. 2018. Unravelling robustness of deep learning based face recognition against adversarial attacks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[30] Jiatao Gu, Yong Wang, Kyunghyun Cho, and Victor OK Li. 2018. Search engine guided neural machine translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[31] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks Are All You Need. *arXiv preprint arXiv:2306.11644* (2023).

[32] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7212–7225.

[33] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*.

[34] Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. AixBench: A Code Generation Benchmark Dataset. arXiv:arXiv:2206.13179

[35] Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy S Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. *Advances in Neural Information Processing Systems* 31 (2018).

[36] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-Based Neural Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 925–930.

[37] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. 2017. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409* (2017).

[38] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[39] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. 2017. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 217–227.

[40] Xiusheng Huang, Yubo Chen, Shun Wu, Jun Zhao, Yuantao Xie, and Weijian Sun. 2021. Named Entity Recognition via Noise Aware Training Mechanism with Data Filter. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. 4791–4803.

[41] 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).

[42] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 1643–1652.

[43] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[44] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871–7880.

[45] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. 2022. A Survey on Retrieval-Augmented Text Generation. *arXiv preprint arXiv:2202.01110* (2022).

[46] J Li, S Ji, T Du, B Li, and T Wang. 2019. TextBugger: Generating Adversarial Text Against Real-world Applications. In *26th Annual Network and Distributed System Security Symposium*.

[47] 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).

[48] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[49] Pietro Liguori, Erfan Al-Hossami, Domenico Cotroneo, Roberto Natella, Bojan Cukic, and Samira Shaikh. 2022. Can we generate shellcodes via natural language? An empirical study. *Automated Software Engineering* 29, 1 (2022), 1–34.

[50] Pietro Liguori, Erfan Al-Hossami, Vittorio Orbinato, Roberto Natella, Samira Shaikh, Domenico Cotroneo, and Bojan Cukic. 2021. EVIL: exploiting software via natural language. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 321–332.

[51] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[52] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*.

[53] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 599–609.

[54] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1–12.

[55] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 373–384.

[56] Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 1918–1923.

[57] 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 *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[58] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. 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/2017/file/8a20a8621978632d76c43dfd28b67767-Paper.pdf

[59] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[60] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. 2015. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211* (2015).

[61] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2011. How we refactor, and how we know it. *IEEE Transactions on Software Engineering* 38, 1 (2011), 5–18.

[62] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309* (2023).

[63] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[64] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[65] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[66] Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2719–2734.

[67] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. 40–47.
[68] Ofir Press, Noah Smith, and Mike Lewis. 2021. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. In *International Conference on Learning Representations*.
[69] Zhiheng Qu, Yi Hu, Jianhui Zeng, Bo Cai, and Shun Yang. 2022. Method Name Generation Based on Code Structure Guidance. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1101–1110.
[70] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.
[71] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[72] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* 21, 140 (2020), 1–67.
[73] Graham Rawlinson. 2007. The significance of letter position in word recognition. *IEEE Aerospace and Electronic Systems Magazine* 22, 1 (2007), 26–27.
[74] 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).
[75] replit. 2023. replit-code-v1-3b. https://huggingface.co/replit/replit-code-v1-3b
[76] Stephen Robertson and Hugo Zaragoza. 2009. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc.
[77] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1073–1083.
[78] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*. 2–13.
[79] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.
[80] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.
[81] Junfeng Tian, Chenxin Wang, Zhen Li, and Yu Wen. 2021. Generating Adversarial Examples of Source Code Classification Models via Q-Learning-Based Markov Decision Process. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 807–818.
[82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[83] Jingyi Wang, Jialuo Chen, Youcheng Sun, Xingjun Ma, Dongxia Wang, Jun Sun, and Peng Cheng. 2021. Robot: Robustness-oriented testing for deep learning systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 300–311.
[84] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2022. ReCode: Robustness Evaluation of Code Generation Models. *arXiv preprint arXiv:2212.10264* (2022).
[85] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. Lightweight global and local contexts guided method name recommendation with prior knowledge. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 741–753.
[86] Shuohang Wang, Yichong Xu, Yuwei Fang, Yang Liu, Siqi Sun, Ruochen Xu, Chenguang Zhu, and Michael Zeng. 2022. Training Data is More Valuable than You Think: A Simple and Effective Method by Retrieving from Training Data. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 3170–3179.
[87] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).
[88] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods*

in Natural Language Processing. 8696–8708.

[89] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. [n.d.]. Emergent Abilities of Large Language Models. *Transactions on Machine Learning Research* ([n. d.]).

[90] Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. 2019. Method name suggestion with hierarchical attention networks. In *Proceedings of the 2019 ACM SIGPLAN workshop on partial evaluation and program manipulation*. 10–21.

[91] Guang Yang, Ke Liu, Xiang Chen, Yanlin Zhou, Chi Yu, and Hao Lin. 2022. CCGIR: Information retrieval-based code comment generation method for smart contracts. *Knowledge-Based Systems* 237 (2022), 107858.

[92] Puyudi Yang, Jianbo Chen, Cho-Jui Hsieh, Jane-Ling Wang, and Michael I Jordan. 2020. Greedy attack and gumbel attack: Generating adversarial examples for discrete data. *The Journal of Machine Learning Research* 21, 1 (2020), 1613–1648.

[93] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-trained Models of Code. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1482–1493. https://doi.org/10.1145/3510003.3510146

[94] 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.

[95] 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 *Proceedings of the 15th International Conference on Mining Software Repositories*. 476–486.

[96] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 440–450.

[97] Yuan Zang, Fanchao Qi, Chenghao Yang, Zhiyuan Liu, Meng Zhang, Qun Liu, and Maosong Sun. 2020. Word-level Textual Adversarial Attacking as Combinatorial Optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 6066–6080.

[98] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 39–51.

[99] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards robustness of deep program processing models—detection, estimation, and enhancement. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–40.

[100] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1169–1176.

[101] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1385–1397.

[102] Xiaoqing Zhang, Yu Zhou, Tingting Han, and Taolue Chen. 2020. Training deep code comment generation models via data augmentation. In *12th Asia-Pacific Symposium on Internetware*. 185–188.

[103] 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).

[104] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald Gall. 2022. Adversarial robustness of deep code comment generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–30.