2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-5278-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/SANER56733.2023.00076

How Robust Is a Large Pre-trained Language Model for Code Generation? A Case on Attacking GPT2

Rui Zhu Cunming Zhang
Yunnan University
Kunming, YN China
rzhu@ynu.edu.cn, azio1246@mail.ynu.edu.cn

***Abstract*—Large pre-trained language models have shown strong capabilities in the field of natural language processing, and in recent years research demonstrated that they can also produce surprising results from natural language descriptions in automatic code-generation applications. Although such models have performed well in a variety of domains, there is evidence that they can be affected by adversarial attacks, Consequently, it has become important to measure the robustness of models by producing adversarial examples. In this study, we proposed an attack method called the Modifier for Code Generation Attack (M-CGA), which is the first time a white box adversarial attack has been applied to the field of code generation. The M-CGA method measures the robustness of a model by producing adversarial examples that can cause the model to produce code that is incorrect or does not meet the criteria for use. Preliminary experimental results showed the M-CGA method to be an effective attack method, providing a new research direction in automatic code synthesis.**

***Keywords*—Code Generation, Adversarial Attack, Large Pre-Trained Language Models, Robustness**

I. Introduction

In modern society, computer programming is a requirement in almost any industry, whether it the industrial, medical, or the educational sectors, among others. There is an increasing demand for high quality code, for experienced programmers to code a variety of tasks to achieve the application needs of various industries. In recent years, large pre-trained language models have shown excellent performance or potential promise in a variety of tasks, while code in large datasets [1][2] and the programming power of language models trained on these datasets [3] have driven the progress of automatic program synthesis. Early work on the generative pre-trained transformer 3 model (GPT-3) found that it could synthesize simple Python programs [4], and Codex [5] was proposed in subsequent work by OpenAI, which provided model support for the later code autosynthesis tool Copilot. Other code-generation tools—such as aiXcoder—are better able to implement method-level code, while CodeBERT [6] and PyMT5 [7] have been used with large language models for code autosynthesis.

Some of the above research such as Copilot has been applied in business or to help programmers generate code, it have major social and economic benefits, however, few code generation tasks considered their robustness. it has been shown that some natural language processing tasks are susceptible to adversarial attacks even when using large pre-trained language models [8][9], where minor perturbations can be applied to the original input of the target model to generate adversarial examples, spoofing the target model and rendering it ineffective. Moreover, if the code-generation model suffers from similar problems—either by incorrect inputs from the user or by a lack of clarity in the input formulation—it could reveal blind spots in the model, causing it to malfunction.

Original example
Problem: Generate bubbleSort for an array arr of length n

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Adversarial example
Problem: Generate a bubbleSort for an array arr, length n

```
def bubbleSort(arr):
    n = len(a)
    for i in range(n):
        for j in range(0, n-a):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Fig. 1. An example of an adversarial attack in code generation, where the code generated by the adversarial example does not perform the task of bubble sorting.

To explore the robustness of existing code generation methods and to investigate whether it can be affected by adversarial attacks, we attacked the code-generation pre-training model. We introducing the Modifier for Code-Generation Attack (M-CGA)—a method for generating adversarial examples by adversarial attacks in the code-generation process—which could find perturbations by segmenting the model code and applying optimization in the reverse direction to produce adversarial examples.

II. Related work

Based on the continued development of artificial intelligence technology, research in the automatic synthesis of such programs continues to advance. Related benchmarks and datasets were generated while conducting research on automatic code-generation methods. Search-based pseudocode to code (SPoC) is a method for converting pseudocode to code using the seq2seq machine translation model and an additional search step [10]. To train the SPoC model, Amazon Mechanical Turk was used to collect line-by-line descriptions of C++ programs. More recently, the CodeXGLUE [11] benchmark was proposed, which

2640-7574/23/$31.00 ©2023 IEEE
DOI 10.1109/SANER56733.2023.00076

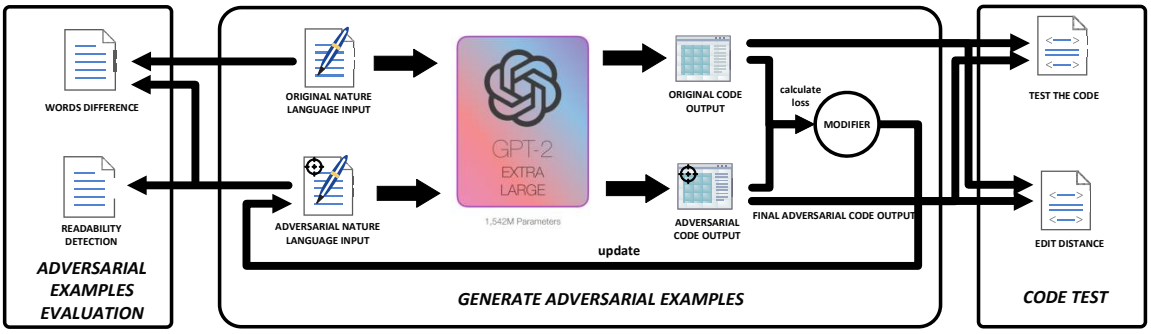

Fig. 2. The overall process of adversarial attack

aggregates various previous benchmarks, using CodeBLEU [12] and CONCODE [13] to examine the generation of Java code from document strings and BLEU to evaluate performance. To benchmark Codex, humanEval was introduced to create a dataset of 164 original programming problems with unit tests. aiXcoder introduced aiXbench [14] for method-level test data from natural language to code-generation models, primarily to evaluate the ability of code-generation models. The Automated Programming Progress Standard (APPS) [15] was used to evaluate models by examining the code generated in test cases.

III. Research Questions

To verify that large code-generated pre-trained language models can be affected by adversarial attacks, we aim to progressively explore the following research questions in this study.

RQ1: Are code-generated pre-trained models affected by adversarial attacks to the extent that their effectiveness is reduced?
We aim to see if the model is affected by adding large perturbations to the adversarial example.

RQ2: Can we find an example where a slight change in input can cause a large deviation in the output of the code-generated model?
We aim to find an example like the one in Fig 1 to test our conjecture.

RQ3: Is the phenomenon in RQ2 widespread?
We randomly selected questions from the APPS dataset for a large-scale test.

These three problems, if present, present a challenge to the stability and security of automatic code-generation methods—that is, if changing the input statements slightly can affect the model results, then the model will be less useful and less secure.

IV. Methodology

This section describes the methods used in the M-CGA and the steps taken to implement them. The overall process of adversarial attack is shown in Fig 2.

A. Problem Definition

The task of generating an adversarial example for automatic code synthesis can be described as follows: given a fragment of natural language description with a word count of n, $X = \{x_1, x_2, \dots, x_n\}$. And defining the code-generation model to be $F: X \to Y$, where the Y model generates code with a word count of m, $Y = \{y_1, y_2, \dots, y_m\}$.

We need to generate adversarial examples that do not differ much from the original natural language description $X' = \{x'_1, x'_2, \dots, x'_{n'}\}$ such that the mapping F does not generate Y, where $x' = x + \delta$, δ being a perturbation. To find an adversarial example that can be formulated as an optimization task [16], as follows:

$$minL(X') + cD(\delta) \quad (1)$$

Where L denotes the loss function that controls the success of the attack, D denotes the penalty term of the perturbation that controls how much the generated adversarial example deviates from the original input, and c denotes the regularization parameter that balances the bias and the success of the attack; a smaller c making the attack more likely to succeed, but at the cost of a larger bias.

B. Methodology Design

The implementation steps are shown in Fig 2, steps 1, 2 for "generate adversarial examples", step 3 for "code test", and step 4 for "adversarial examples evaluation".

Step 1: We need to obtain the original output of the model, while we use the already trained model (GPT-2) as the attack target and use the original input as the model input, the original output of the obtained model being used as the label of the attack and to perform the loss operation using the adversarial output generated later.

Step 2: We obtain an adversarial example by adding a perturbation to the original input. In the initial iteration, we input the original input into the model to obtain the output with labels to calculate the loss, followed by back-propagation to update the perturbations, adding the newly generated perturbation values to the embeddings so that the samples generated can be used as new inputs. We stop updating the perturbations after a certain number of iterations, using the input samples with the greatest impact on the model-generated code as the adversarial examples.

Step 3: We evaluate the code generated by the model by first testing the original generated code and measuring the proportion of it that can pass the test case, and then measuring the attacked code. The tree edit distance (TED) is measured for both codes to gauge the change.

Step 4: We measure the gap between the adversarial examples and the original input and measure it by the BLEU score and the number of words changed by the adversarial examples.

C. Loss Function

The design of the loss function is the main factor in the success of the optimization attack. Moreover, we need a loss function that can adjust the relationship between loss and perturbation. Python code is semi-structured, and the division between code semantics can be done across lines, where each line of code forms a small but functionally complete unit. Our approach compares the divided lines of code with the original output to find the next section that describes the design of the code-generation task.

The original output code segment of the model is defined as $P_o = \{s_{o1}, s_{o2}, \dots, s_{on}\}$, $s_{oi} = \{t_{oi1}, t_{oi2}, \dots, t_{oim}\}$ being a line of code containing m words, where t_{oij} denotes the word in the j^{th} position in the i^{th} line of code. The code segment generated by the model under the attack being P_a, which has the same form as P_o, the generated words forming a word list v of length d, while the output of the model under the attack is the output of the logit layer of the model $Y = \{l_1, l_2, \dots, l_n\}, l = \{z_1, z_2, \dots, z_m\}$. We let the attack generate a gap between the generated code and the original generated code—that is, $P_o \neq P_a$—which can be interpreted as the existence of a situation where the output value of the current logit layer is greater than the output value of the original output logit layer, expressed as follows:

$$z_{ij}(t_{oij}) < max_{y \in v, y \neq t_{oij}} z_{ij}(y), \quad \exists i \in 1, \dots, n, \exists j \in 1, \dots, m \tag{2}$$

Thus, the loss function we define needs to calculate the difference between the label and the logit layer of the output for each word in each line of generated code.

$$L = \sum_{i=1}^{n} \sum_{j=1}^{m} max\{-\epsilon, z_{ij}(t_{oij}) - max_{y \neq t_{oij}} z_{ij}(y)\} \tag{3}$$

Moreover, its needs a threshold parameter (ϵ) to control the confidence difference between the original generated code words and the attack generated code words. When ϵ is larger, the attacked model needs to have a higher confidence level for words other than the original generated code, generally having a higher attack success rate, although changes to the original input may also be more pronounced.

As the adversarial example is changed, inconsistencies in the number of lines of code output by the model and inconsistencies in the length of the individual lines with the previous original output occur. When the number of lines or words in the model output is less than the original output—assuming the original output to be correct—we can assume that the model output code is not functional enough to support the correct operation of the code (that is, there is a high probability that key variables are missing). However, when the model outputs more lines or words than the original output, some loss adjustment is required, as it has been established experimentally that the model output generates a redundant and irrelevant line of code, and we can assume the attack to be unsuccessful and that further adjustment is required. The loss can be expanded as follows:

$$L = \sum_{i=1}^{q} \left\{ \begin{array}{c} \sum_{j=1}^{p} max\left\{-\epsilon, z_{ij}(t_{oij}) - max_{y \neq t_{oij}}\{z_{ij}(y)\}\right\} \\ + c_1 \sum_{j=p+1}^{m} max\{z_{ij}(y)\} \end{array} \right\} + c_2 \sum_{i=q+1}^{n} \sum_{j=1}^{m} max\{z_{ij}(y)\} \tag{4}$$

Where n is the number of lines of model output and m is the number of tokens of a certain line of code. When the number of lines of original output q is less than n, or the number of tokens of a line of code in the original output p is less than m, the confidence of the logit layer of the model output redundant tokens is added as a loss. The hyperparameters c_1 and c_2 control the degree of influence of the redundant code on the result, the smaller the value means that the model output redundant code is influenced to a higher degree, and the larger the value means that the model output redundant code is irrelevant and more iterations may be needed to continue the search for adversarial examples.

V. Experiments

A. Datasets and Models

The data we used came from the APPS [15], a dataset containing 10,000 programming problems, with 131,777 test cases for checking solutions, the average length of a problem being 293.2 words. The programming difficulty data were divided into Introductory, Interview, and Competition levels, the specific components of which are shown in Table I. In the preliminary experiment, we selected 100 typical problems from each difficulty level of the APPS test set as the attack object and used GPT-2 to test the robustness model, provided by the APPS.

TABLE I. Data Composition of The APPS Dataset

Problem Difficulty	Total Number of Problems	Number of Test Set Problems
Introductory Level	3,639 problems	1,000 in test set
Interview Level	5,000 problems	3,000 in test set
Competition Level	1,361 problems	1,000 in test set

B. Steps of Attack

Our experiment was divided into two steps:

1) We first attacked GPT-2 by using the M-CGA, in which we had a large perturbation amplitude, despite the readability of the adversarial examples. At the same time, these adversarial examples allowed the model to generate code with large gaps from the original output code.

2) During the second stage, we controlled the perturbation magnitude of the attack, as we needed adversarial examples that did not differ much from the original input. We performed the attacks on problems selected from the APPS dataset, iterating the attacks 10 times for each problem and selecting samples from which the output code changed more as the result.

C. Attack Results

We measured the effectiveness of the attack by testing the code generated by the original input and the adversarial example, and calculating the proportion of test cases that passed in the presence of the two generated code samples. We use the TED to measure the gap between the abstract syntax tree structure of the model-generated code, which was a tree-like representation of the abstract syntax structure of the source code. The TED let us measure the gap between the generated code from a compilation perspective, with a larger TED indicating a larger gap between the two pieces of code. The attack success rate was a test of the robustness of the model. We defined the calculation of the attack success rate to include only the problem of the model generating code

correctly based on the original input and the problem of the generated code passing at least one of the test cases; if the model could not generate correct code in the first place, this was a problem of the model's performance, not its robustness. The attack success rate (ASR) can be defined as follows:

$$ASR = 1 - \frac{CodeA_{pass}}{Code_{all} + Code_{pass}} \quad (5)$$

where $Code_{all}$ and $Code_{pass}$ denote the number of problems where the model can generate code correctly based on the original input and the number of problems where the generated code passes some of the test cases, respectively. $CodeA_{pass}$ denotes the number of problems that can still pass at least one test case after being attacked.

The experimental results are shown in Table II. If the model output code can pass at least one test case, we denote the code as "TRUE"—that is, the generated code can solve the problem. *Original Output* denotes the proportion of the original output code recorded as "TRUE" and *Adversarial Output* denotes the proportion of output code recorded as "TRUE" after the attack, as shown in Fig 3. For both the *Introductory* and *Interview* levels, the percentage of code generated after the attack that passes the test cases drops considerably, and we can assume that the effectiveness of the model is reduced for these two levels of problems. The model generates fewer code for the Competition level problems can pass the test case, which does not reflect the effect of the attack. For *Introductory* level problems, the success rate of the attack is not as high as that for *Interview* level problems.

TABLE II. ATTACK RESULTS

	Original Output (%)	Adversarial Output (%)	TED	ASR (%)
Introductory	46	24	22.19	47.83
Interview	62	19	51.83	69.35
Competition	9	2	28.60	77.78
Average	39	15	34.20	64.99

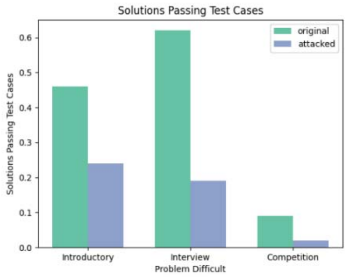

Fig. 3. Number of test cases that can be passed by the generated code before and after the attack.

We speculate that the code generated by the *Introductory* level problems is relatively simple and it is difficult to change the output space much by replacing a few words in the input problem in the pre-trained language model.

At the same time, the TED between the original output code and the code generated after the attack is large, with an average change of 34.2. Only a few samples in the experiments do not differ from the original output code due to changes in the input, most samples being successfully changed. Moreover, the code generated by the adversarial examples changes more from a compilation viewpoint, so it

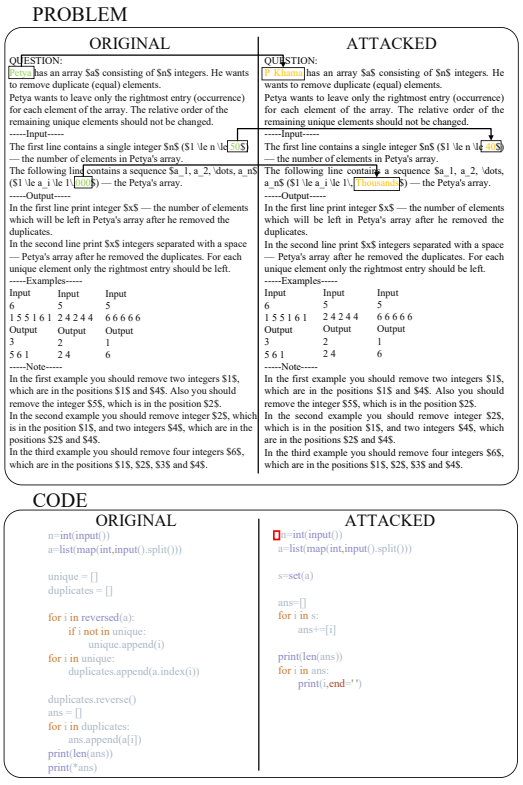

Fig. 4. An example of an adversarial attack.

can be assumed that the adversarial attack has a major impact on the code-generation model.

D. Readability of Generating Adversarial Examples

We measured the deviation between the original input and the adversarial example by the number of changes in the BLEU score and the number of words in the adversarial example. If the deviation value was too large, it could indicate that the readability of the adversarial example had deteriorated, so we wanted to determine the ideal adversarial example not too far from the original input. The higher the BLEU score, the smaller the difference between the original input and the adversarial example, and the higher the quality of the adversarial example.

TABLE III. BLEU SCORES AND WORD CHANGE COUNTS FOR THE ORIGINAL AND ADVERSARIAL EXAMPLES

	BLEU	Changed
Introductory	97.75	1.5
Interview	97.80	2.7
Competition	98.16	1.2
Average	97.90	1.8

The results are shown in Table III. The mean BLEU score between the adversarial example generated by the M-CGA and the original input is 97.9, the adversarial example changing little from the original input, with "changed" being the number of words changed by the adversarial example (the average number of words changed being 1.8). This is because the number of iterations of the attack or the perturbation magnitude are not sufficiently large. However, if the perturbation magnitude is increased, the difference between

the adversarial example and the original input may be too large, and the readability may be lost. Consequently, we consider the case of no change in the adversarial example to be an attack failure.

E. Example

The example we provided is shown in Fig 4. The top section is the text describing the problem as input to the model, the bottom section being the code for the model output, with the original input and the original output on the left, and the adversarial example and output after being attacked on the right. The wireframe and connecting lines indicate that the words in the original input are changed to words in the adversarial example by the adversarial attack.

The adversarial example changes the name of a character in the problem by entering the maximum value of the array length and replacing the "000" in "1,000" with "Thousands". From a human perspective, the adversarial example does not change the original input strongly enough for the changed factor to have a major impact on the human completing the programming question.The original output code is cumbersome and correct, but the code generated from the adversarial example not only fails to complete the task described in the problem, but also has indentation formatting errors (red boxes indicate a space). Experimental results similar to the above example were widespread.

VI. Conclusions and Future Work

A. Answer to RQs

For the **RQ1**, we attacked the model by applying the M-CGA and got the model to generate incorrect code, so demonstrating that the large code-generation pre-trained model could be affected by adversarial attacks.

For the **RQ2**, we detected the gap between the input samples in our experiments by comparing the BLEU values of the generated adversarial examples to the original input and the gap between the output codes, by comparing the TEDs of the generated attacked codes to the original codes. Samples in the experiments similar to those shown in Fig 3 show that using our M-CGA method could slightly alter the inputs to generate incorrect code.

For the **RQ3**, we conducted extensive experiments on the APPS dataset. From the results we could conclude that adversarial examples were widely available, and that the input samples could successfully attack the model with very few word changes if the attack was deliberate.

B. Summary

In this paper, we presented the M-CGA method, an optimization-based method for generating adversarial examples to a code-generating model that resulted in the model generating incorrect code with small perturbations. Moreover, we conducted various experiments to demonstrate that automatic code-generation tasks could indeed be affected by adversarial attacks.

C. Future Work

Since GPT-3 is not directly attackable by white-box attack methods, in future work, we will use GPT-Neo as a surrogate model for attacks, and use the adversarial examples generated by GPT-Neo to measure the resistance of GPT-3 to interference. We will also use the generated adversarial examples to test more models for migration experiments. Later we will set up defensive work such as adversarial training based on the generated adversarial examples to test whether our defensive approach can be effective on large pre-trained models.

Acknowledgment

This work was supported by National Natural Science Foundation of China under Grant 62002310; Yunnan Provincial Natural Science Foundation Fundamental Research Project under Grant 202101AT070004; Science Foundation of Yunnan Jinzhi Expert Workstation under Grant 202205AF150006; Science and Technology Innovation Project of Yunnan Provincial Transportation Investment and Construction Group Company under Grant YCIC-YF-2021- 07; Yunnan Xing Dian Talents Support Plan.

References

[1] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," arXiv preprint arXiv:1909.09436, 2019.

[2] L. Gao et al., "The pile: An 800gb dataset of diverse text for language modeling," arXiv preprint arXiv:2101.00027, 2020.

[3] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model," ed: May, 2021.

[4] T. Brown et al., "Language models are few-shot learners," Advances in neural information processing systems, vol. 33, pp. 1877-1901, 2020.

[5] M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.

[6] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," Online, November 2020: Association for Computational Linguistics, in Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 1536-1547.

[7] C. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "PyMT5: multi-mode translation of natural language and Python code with transformers," Online, November 2020: Association for Computational Linguistics, in Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 9052-9065.

[8] D. Jin, Z. Jin, J. T. Zhou, and P. Szolovits, "Is bert really robust? a strong baseline for natural language attack on text classification and entailment," in Proceedings of the AAAI conference on artificial intelligence, 2020, vol. 34, no. 05, pp. 8018-8025.

[9] L. Li, R. Ma, Q. Guo, X. Xue, and X. Qiu, "BERT-ATTACK: Adversarial Attack Against BERT Using BERT," Online, November 2020: Association for Computational Linguistics, in Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 6193-6202.

[10] A. Wang et al., "Superglue: A stickier benchmark for general-purpose language understanding systems," Advances in neural information processing systems, vol. 32, 2019.

[11] S. Lu et al., "Codexglue: A machine learning benchmark dataset for code understanding and generation," arXiv preprint arXiv:2102.04664, 2021.

[12] S. Ren et al., "Codebleu: a method for automatic evaluation of code synthesis," arXiv preprint arXiv:2009.10297, 2020.

[13] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping Language to Code in Programmatic Context," Brussels, Belgium, oct nov 2018: Association for Computational Linguistics, in Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pp. 1643-1652.

[14] Y. Hao et al., "AixBench: A Code Generation Benchmark Dataset," arXiv preprint arXiv:2206.13179, 2022.

[15] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, and A. Arora, "Measuring Coding Challenge Competence With APPS."

[16] N. Carlini and D. Wagner, "Towards Evaluating the Robustness of Neural Networks," 2017 IEEE Symposium on Security and Privacy (SP), 2017, pp. 39-57, doi: 10.1109/SP.2017.49.