DIP: Dead code Insertion based Black-box Attack for Programming Language Model

CheolWon Na, YunSeok Choi, Jee-Hyong Lee[†]
College of Computing and Informatics
Sungkyunkwan University
Suwon, South Korea
`{ncw0034, ys.choi, john}@skku.edu`

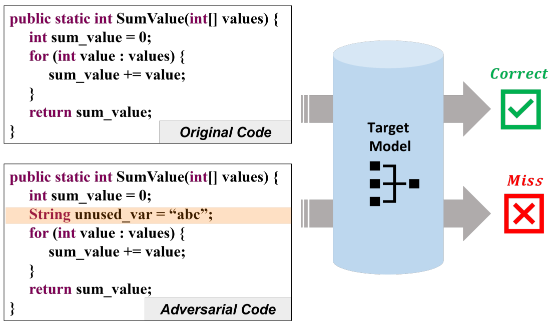
Abstract

Automatic processing of source code, such as code clone detection and software vulnerability detection, is very helpful to software engineers. Large pre-trained Programming Language (PL) models (such as CodeBERT, GraphCodeBERT, CodeT5, *etc.*), show very powerful performance on these tasks. However, these PL models are vulnerable to adversarial examples that are generated with slight perturbation. Unlike natural language, an adversarial example of code must be semantic-preserving and compilable. Due to the requirements, it is hard to directly apply the existing attack methods for natural language models. In this paper, we propose **DIP** (***D**ead code **I**nsertion based Black-box Attack for **P**rogramming Language Model*), a high-performance and efficient black-box attack method to generate adversarial examples using dead code insertion. We evaluate our proposed method on 9 victim downstream-task large code models. Our method outperforms the state-of-the-art black-box attack in both attack efficiency and attack quality, while generated adversarial examples are compiled preserving semantic functionality.

1 Introduction

Automatic processing of source code (such as clone detection, code completion, defect detection, *etc.*) is an important task that increases the productivity of software engineers (Laguë et al., 1997; Li et al., 2006; Mockus, 2007; Kapser and Godfrey, 2008; Islam et al., 2016; Choi et al., 2021, 2023). For these tasks, deep learning-based models, such as code2seq (Alon et al., 2019a) and code2vec (Alon et al., 2019b), were developed. Recently, transformer-based large pre-trained programming language (PL) models (Feng et al., 2020; Guo et al., 2021b; Wang et al., 2021; Ahmad et al., 2021; Ding et al., 2021; Guo et al., 2022), showed powerful performance. However, these PL models are also vulnerable to adversarial examples as other natural language (NL) models (Choi et al., 2022; Zhang et al., 2020; Yang et al., 2022; Jha and Reddy, 2022; Jin et al., 2020; Li et al., 2020, 2021).

[†]Corresponding author.



```
public static int SumValue(int[] values) {
    int sum_value = 0;
    for (int value : values) {
        sum_value += value;
    }
    return sum_value;
}
```

```
public static int SumValue(int[] values) {
    int sum_value = 0;
    String unused_var = “abc”;
    for (int value : values) {
        sum_value += value;
    }
    return sum_value;
}
```

Figure 1: Dead code which is non-affecting to functionality is inserted into the original code in order to mislead the target model.

Source code is textual data just like natural language, but the requirements of attack methods are different from NL model attacks. Attacks against NL models should consider semantic and fluency of original sentences, but attacks against PL models should preserve functionality and guarantee compilability. To consider for these issues, recent studies (Yefet et al., 2020; Yang et al., 2022; Srikant et al., 2021; Rabin et al., 2021; Zhang et al., 2020; Ramakrishnan et al., 2020) use *semantic-preserving code transformation* which is commonly used in software engineering. The semantic-preserving code transformation includes *Variable Renaming*, *Dead Code Insertion*, *Statement Permutation*, *Loop Exchange*, *Switch-to-If*, and *Boolean Exchange*. Most of the prior works used *Variable Renaming* or *Dead Code Insertion* to attack PL models. Other transformations *Statement-Permutation*, *Loop-Exchange*, *Switch-to-If*,*Boolean-Exchange*, *etc*, were less used for attacks because they are applicable only to corresponding operations. Instead, they were usually used for data augmentation (Jain et al., 2020; Rabin et al., 2021). *Variable Renaming* and *Dead Code Insertion* can be much stronger attacks because attackers can flexibly choose various variable names or statements considering the contextual information of code.

In the white-box setting, Yefet et al. (2020) proposed attack methods on code2vec, using *Variable Renaming* and *Dead Code Insertion*, which inserted perturbations sampled from the output distribution into input one-hot vectors representing variables. Ramakrishnan et al. (2020) and Srikant et al. (2021) proposed white-box attacks, optimizing position and perturbation of adversarial attacks, for basic PL models such as seq2seq, code2seq, and code2vec. However, those gradient-based white-box attack methods are inefficient, time-consuming, and computationally expensive because large pre-trained PL models, such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021b) and CodeT5 (Wang et al., 2021), have a very large number of parameters. Also, it is hard to apply white-box attacks on large pre-trained models in the real world because the models are not fully opened and are just accessible through API. Therefore, the black-box attack is more practical than the white-box for attacking large pre-trained models.

In the black-box setting, Zhang et al. (2020) proposed MHM, a sampling-based black-box attack method for programming language model using *Variable Renaming*. Yang et al. (2022) proposed ALERT, a genetic algorithm based black-box attack method using *Variable Renaming*. However, these attack methods using *Variable Renaming* are inefficient in attack success rate and in query count, because attack targets are limited to variables, and the search space for candidate tokens is very large. Additionally, these methods have a critical problem that compilability is not guaranteed. In programming language, unlike natural language, compilability is more prior than naturalness, and must be guaranteed. There are some variables that should not be renamed. For example, global variables or class static variables in Java, such as `System.out` for standard input/output, should not be changed. However, *Variable Renaming* can replace such variables, and could raise a critical problem such as compile errors.

We propose **DIP** (***D**ead code **I**nsertion based Black-box Attack for **P**rogramming Language Model*), an efficient and the state-of-the-art adversarial attack method using *Dead Code Insertion* with an `unused variable` statement. Under the black-box setting, we do not need any information of a large pre-trained model, no additional computation for training is required for attack.

For efficiency, we reduce the candidate search space by inserting a statement from other code, not tokens. Randomly selected dissimilar codes are used to obtain statements to be inserted. We extract statements from code using the attention score of a pre-trained model, which is based on transformer architectures. The snippet extracted by our method is powerful and transferable. In addition, our proposed method guarantee to preserve functionality and compilability in any case because the attack is based on dead statement insertion.

We show that our proposed method outperforms the existing methods to attack pre-trained programming language models in Section 4. We choose 3 pre-trained PL models, CodeBERT, GraphCodeBERT, and CodeT5, and 3 tasks, code clone detection, defect detection, and authorship attribution task. We build nine victim models in total. We also conduct various experiments to explore our method. The ablation study (§4.4) examines the effectiveness of each component in DIP. In §4.5, we test how strong DIP is on adversarially trained models. We also test whether adversarial samples used to attack a victim model can also fool another in §4.6.

2 Task Definition

We evaluate DIP on code clone detection, defect detection, and authorship attribution tasks. For these tasks, we build the model by adding MLP classifier to the state-of-the-art pre-trained PL models (such as CodeBERT, GraphCodeBERT, and CodeT5). Then, we fine-tune the nine victim models.

Black-box setting. Since our proposed method is a black-box attack, we do not use any information from the victim models. To obtain the code representation and attention score in DIP, we need a pre-trained model based on Transformer architectures (Vaswani et al., 2017). In this paper, we use CodeBERT, a general pre-trained PL model which is not fine-tuned, to obtain code embeddings without any information from victim models.

Non-targeted Adversarial attack. Adversarial attack can be usually categorized as either non-targeted attack or targeted attack. Non-targeted adversarial attack is to slightly modify the source

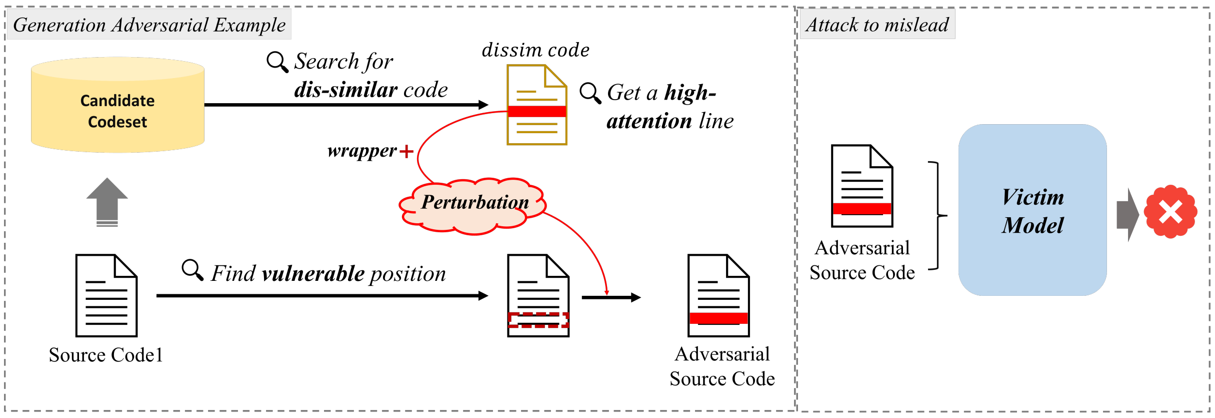


Figure 2: Overview of DIP.

input in a way that the input will be misclassified far from the ground truth. Targeted adversarial attack is to slightly modify the source input in a way that the input will be misclassified as a specified target class. Since our method is a non-targeted attack, the goal is defined as follows.

Goal. Given an input sample (c_i, y), find c_i^{adv}, by adding perturbation to c_i, that preserves functionality and guarantees compilability, and that misleads the victim model. The adversarial code, c_i^{adv}, needs to satisfy the followings:

$$M(c_i^{adv}) \neq y, \tag{1}$$

where $c_i^{adv} = c_i + perturbation$, M is the victim model, and y is the ground truth.

3 DIP

We propose **DIP**, using *dead code insertion* to guarantee compilability while preserving the semantics of source codes. *Dead code insertion* is very suitable for source code attack because it does not affect functionality and compilability of code.

Our method consists of three main steps as shown in Figure 2: (1) We find vulnerable positions to insert dead code in the original source code. Dead code will be inserted between statements in source code as a statement to guarantee compilability. We evaluate the vulnerability of each position in source code by a pre-trained code model which is not fine-tuned, and choose some of them as candidate vulnerable positions. (2) We obtain dissimilar codes of which snippets will be used as dead code. Dissimilarity of source code is defined based on the cosine similarity to the original source code using [CLS] token representations. (3) To minimize perturbation, we extract snippets from dissimilar source codes. Snippets are extracted by the attention score in dissimilar source codes. We wrap the obtained snippet with an `unused variable` statement and insert it into the source code as dead code. We summarize our proposed method, DIP, in Algorithm 1.

Algorithm 1: DIP Pseudo-code

```
Input  : Source code $c_i$, true label $y$, target model $M$
Output : Adversarial Example $c_i^{adv}$
Compute position importance $V_p \forall\ p \in c_i$
Generate $k$ candidate dead code
dead code list $\mathbb{D} = [d_1, \ldots, d_k]$ ordered by the dissimilarity
for $p$ in ascending order of $V_p$ do
    $c_i^{adv} \leftarrow c_i$
    for $d \in \mathbb{D}$ do
        $c_i^{adv} \leftarrow$ insert $d$ into $c_i$ at $p$
        if $M(c_i^{adv}) \neq y$ then
            # Success Attack
            Return: $c^{adv}$
        end
    end
end
# Fail Attack
Return: $c^{adv} \leftarrow None$
```

3.1 Vulnerable position selection

First, we choose vulnerable positions to insert dead code in the original source code for efficient attack. Dead code will be generated as a statement. Since dead code can be inserted after any statement in the original source code, the position is defined as line (or statement) numbers in the source code.

Algorithm 2: Dead Code Generation

```
Input  : Source code $c_i$, true label $y$,
         attention layer of CodeBERT $T$
Output : dead code list $\mathbb{D}$
$\mathbb{C} \leftarrow \{c_n | c_n \notin train/test\ data, n \neq i\}$
Sample $c_1, \dots, c_k$ from $\mathbb{C}$
$\mathbb{C} \leftarrow \{c_1, \dots, c_k\}$
# sort $\mathbb{C}$ using ScoreD in Section3.2
for $c \in \mathbb{C}$ do
    # tokenize and get attention score of c
    $\mathbb{S} \leftarrow \{att_T(tok_1^c), \dots, att_T(tok_n^c)\}$
    # get line index(start,end of line) list $\mathbb{P}$
    $\mathbb{P} \leftarrow [(1, a), (a+1, b), \dots, (n+1, m)]$
    $\alpha \leftarrow 0$
    for $(s, e) \in \mathbb{P}$ do
        if $\alpha < \max\{\mathbb{S}[s:e]\}$ then
            $\alpha \leftarrow \max\{\mathbb{S}[s:e]\}$
            $best_{idx} \leftarrow (s:e)$
        end
        $snippet_c \leftarrow c[best_{idx}]$
    end
    $d \leftarrow string\ var =$ "$snippet_c$";
    # append d to $\mathbb{D}$
end
$\mathbb{D} \leftarrow \{d_1, \dots, d_k\}$
Return: $\mathbb{D}$
```

To find out vulnerable positions and sort them by vulnerability, we use [CLS] representations which is obtained from a pre-trained PL model which is not fine-tuned (such as CodeBERT). We insert a UNK sequence, u = [UNK, UNK, ..., UNK], into a position of the original source code to be attacked. Since we attack to insert the code token sequence as a statement in the original source code to be modified, we could find a vulnerable position by inserting u. The original source code will be denoted by c, and the modified code by inserting u into c at position p will be denoted by c'. We put c into a pre-trained PL model to get the [CLS] representation of c, r_c. Similarly, we put c' to the pre-trained PL model to obtain $r_{c'}$. The position score V_p of position p is defined as follows:

$$V_p = \frac{r_c \cdot r_{c'}}{||r_c|| \cdot ||r_{c'}||} \quad (2)$$

We evaluate the position score of each position, and rank all the positions by position scores. We may choose top-K vulnerable positions, or use all the positions. The lower the position score, the more vulnerable the position.

3.2 Dissimilar code selection

We obtain dissimilar source codes from an open set of source code. To evaluate dissimilarity scores of source code, we also use the cosine similarity of [CLS] tokens in a similar way to Equation 2. Let c and c_i denote the source code to be modified and another source code not in train/test dataset, respectively. We obtain r_c and r_{c_i}, [CLS] representations of c and c_i, from a pre-trained PL model, respectively. Then, we use the cosine similarity to calculate dissimilarity score, D, of c_i against c. In order to reduce the search space and prevent biased collection of dissimilar source code by some factor, such as functionality of source code, we first randomly select K source codes and sort them by dissimilarity score D.

3.3 Snippet extraction

We extract a statement from each dissimilar code, and insert it to the source code to minimize perturbation instead of inserting the whole of dissimilar code. In the dissimilar code, the statement with the highest attention score is extracted. We obtain the attention score of each token in a statement from the second-to-last attention layer of a pre-trained PL model. To obtain a general representation that is less biased in object functions of a pre-trained PL model, we use the representation of second-to-last attention layer. We use CodeBERT, which is not fine-tuned, as a pre-trained PL model. The higher attention score is the more important in the source code. Since the pre-trained PL model is based on an attention mechanism, the higher attention score indicates that the corresponding token is more focused on the model. Then, we use the max value, which is the highest attention score in token sequence of statement, as score α in Algorithm 2 (line 13). This process is described in Algorithm 2 (lines 6-16).

3.4 Adversarial code generation

We wrap the snippet obtained in §3.3 using a wrapper, an unused variable statement, to make it dead code. To consider the naturalness of the adversarial examples, we define an unused variable name as *var*_2 using the most used variable name `var` in the code. For example, if the most used variable name is `str`, then the unused variable name will be `str_2`. If there is a snippet extracted from dissimilar code, the dead statement generated with a wrapper of unused variable is as follows:

```
String var_2 ="snippet";
```

We extract one snippet from each of K dissimilar codes ordered by the dissimilarity score D. We repeat inserting each dead statement into all positions p sorted by V_p, in the original source code until the attack succeeds. Our method can generate powerful and effective adversarial examples. Algorithm 2 describes the process of adversarial example generation by inserting dead code.

4 Experiment

4.1 Experimental setup

Datasets and Tasks We evaluate DIP on code clone detection, defect detection, and authorship attribution tasks with 1000/1000/132 test samples, respectively. Datasets of clone detection and defect detection are included as a part of the CodeXGLUE benchmarks (Lu et al., 2021). The dissimilar code set is obtained from the validation dataset in our experiments. For a fair comparison with the baselines, we use the same training dataset as the baselines to fine-tune victim models (Yang et al., 2022).

- *Clone Detection* is to determine whether the functionalities of two given source codes are the same or not. We use the BigCloneBench (Svajlenko and Roy, 2015) dataset, which is a widely used benchmark in the clone detection task. This dataset contains true clone pairs and false clone pairs from Java projects.
- *Defect Detection* aims to find whether a given source code is insecure or not. We use a dataset served by (Zhou et al., 2019). This dataset is extracted from large-scale open-source C projects. This dataset includes 27,318 functions.
- *Authorship Attribution* task is to identify the author of a given source code. We use the Google Code Jam (GCJ) dataset, which is collected by Alsulami et al. (2017). This dataset contains 660 python files (66 authors and 10 files per author).

Target Models We analyze the vulnerability of three popular and powerful pre-trained PL models: *CodeBERT*, *GraphCodeBERT*, and *CodeT5*. *CodeBERT* (Feng et al., 2020)'s objectives are masked language modeling on NL-PL pairs and replaced token detection. *GraphCodeBERT* (Guo et al., 2021b) is pre-trained with code structure information as data flow. *CodeT5* (Wang et al., 2021) is an encoder-decoder PL model, which is pre-trained by an identifier-aware objective. We build nine victim models by fine-tuning these models in three tasks. More details of victim models are listed in Appendix A.

Baselines We compare with the state-of-the-art black-box attack methods on PL models, MHM (Zhang et al., 2020) and ALERT (Yang et al., 2022). They used *Variable Renaming* transformations to attack PL models. MHM was based on the metropolis-hastings sampling method. ALERT used genetic algorithm with a masked language model. We conduct the experiments in the same environment as the baselines. CodeBERT and GraphCodeBERT results of baselines are referred from served by Yang et al. (2022) To evaluate on CodeT5, we implement MHM and ALERT.

Hyperparameter Our method has two hyperparameters: M and K. M is the number of [UNK] tokens to evaluate vulnerable position scores, and K is the number of randomly chosen dissimilar source codes. Since the average length of statements in source code of the test dataset is 9.7, the number of [UNK] tokens to be inserted is set to M=10, which is slightly longer than the average to make enough difference in CLS vectors. We set K=30. If K is small, the attack success rate will be low, and if it is large, the attack will be inefficient. We choose a number small but large enough to include various source code. For comparison with the baselines, we use the same maximal input length to 512.

4.2 Metrics

We use four metrics to evaluate our method. To measure the *efficiency* of the generated adversarial code, we use ASR and $Query$. We also use $Pert$ and $CodeBLEU$ to measure *quality* of the generated examples. We define the following metrics.

Attack Success Rate (ASR) ASR is the success ratio of attacks. The higher ASR, the better performance of an attack method. Let C^{adv} denote a generated adversarial example from the original input C, M is the target model, and y is the true label. Then ASR isndefined as follows:

$$ASR = \frac{|\{C|M(C^{adv}) \neq y \wedge M(C) = y\}|}{|\{C\}|} \quad (3)$$

Task	Victim	Attack	Attack efficiency		Attack quality	
(Language)	Model	Method	ASR	Query	Pert	CodeBLEU
Clone Detection (Java)	CodeBERT	MHM	20.2	667.7	0.32	0.56
		ALERT	28.6	529.4	**0.13**	0.73
		DIP (ours)	**46.7**	**19.9**	**0.13**	**0.92**
	GraphCodeBERT	MHM	4.2	1025.9	0.36	0.32
		ALERT	9.2	448.6	0.13	0.72
		DIP (ours)	**36.6**	**78.2**	**0.14**	**0.85**
	CodeT5	MHM	4.6	104.5	0.26	0.42
		ALERT	22.0	762.2	0.14	0.73
		DIP (ours)	**31.8**	**38.2**	**0.11**	**0.93**
Defect Detection (C/C++)	CodeBERT	MHM	27.4	451.9	0.33	0.32
		ALERT	31.4	277.6	**0.11**	0.76
		DIP*	**44.6**	**47.6**	0.19	**0.91**
	GraphCodeBERT	MHM	41.3	316.7	0.33	0.31
		ALERT	46.7	263.6	**0.10**	0.76
		DIP (ours)	**49.7**	**71.0**	0.13	**0.79**
	CodeT5	MHM	49.3	333.5	0.10	0.78
		ALERT	46.9	187.4	**0.08**	0.80
		DIP (ours)	**49.7**	**61.0**	0.16	**0.92**
Authorship Attribution (Python)	CodeBERT	MHM	15.9	444.0	0.13	0.78
		ALERT	29.6	545.4	0.13	0.79
		DIP (ours)	**31.1**	**300.15**	**0.09**	**0.85**
	GraphCodeBERT	MHM	26.5	774.9	0.30	0.49
		ALERT	50.8	573.2	0.15	0.75
		DIP (ours)	**61.4**	**292.6**	**0.08**	**0.81**
	CodeT5	MHM	36.4	684.6	0.16	0.78
		ALERT	41.7	373.4	0.10	0.83
		DIP (ours)	**43.9**	**47.2**	**0.07**	**0.92**
All	Average	MHM	25.1	537.7	0.25	0.53
		ALERT	34.1	440.1	**0.12**	0.76
		DIP (ours)	**43.9**	**106.2**	**0.12**	**0.88**

Table 1: Comparison of our proposed method with the baseline methods on nine victim models. We set all of the hyper-parameters (α, β, γ, and δ) in CodeBLEU to 0.25, respectively. The best performance is in **boldface**, and the next is underlined.

Number of Queries (Query) $Query$ is the average query number of successful attacks. Our method is a black-box approach, so queries are the only accessible way to the target model. The number of queries is one of important metrics to evaluate efficiency of attack methods. Let q_i denote the number of queries for i-th succeed attack, $Query$ is defined as follows:

$$Query = \frac{\sum q_i}{\sum f(i)} \tag{4}$$

where $i \in \{j | f(j) = 1\}$, and

$$f(j) = \begin{cases} 1, & \text{if } M(C_j^{adv}) \neq y_j \wedge M(C_j) = y_j \\ 0, & \text{otherwise} \end{cases}$$

Ratio of Perturbation (Pert) The ratio of perturbation indicates how many perturbations are injected into the original source code. A lower $Pert$ indicates that examples are generated with less perturbation. Let C_i^{adv} is an adversarial example of C_i of which the truth label is y_i, and $t(\cdot)$ is the number of tokens. $Pert$ is defined as follows:

$$Pert = \frac{\sum t(C_i^{adv}) - t(C_i^{adv} \cap C_i)}{\sum t(C_i)} \tag{5}$$

where i are defined same as in Eq. 4.

CodeBLEU Ren et al. (2020) proposed $CodeBLEU$ to measure generated code by machine learning models. $CodeBLEU$ considers functional and structural information of code such as AST match and Data-flow match. $CodeBLEU$ is a more efficient metric than $BLEU$ to measure the consistency of generated code. If $CodeBLEU$ is close to 1, the code preserves the semantic meaning of the original code.

$CodeBLEU$ is defined as follows:

$$\begin{aligned} CodeBLEU = \alpha \cdot BLEU + \beta \cdot BLEU_{weight} \\ + \gamma \cdot Match_{ast} + \delta \cdot Match_{df} \end{aligned} \tag{6}$$

Method	ASR	Pert	Query
DIP	**46.7**	**0.13**	**19.9**
w/o Dissim	45.2±0.9	0.14±0.01	26.0±5.1
w/o Position	46.7±0.0	0.14±0.00	29.6±5.0
w/o Att-line	47.0±0.5	0.12±0.00	110.3±6.1

Table 2: Ablation study on DIP. "*w/o Dissim*" denotes random selection of dissimilar codes without dissimilar scores in Eq. 2, "*w/o Position*" denotes random position insertion without vulnerable position score, and "*w/o Att-line*" denotes random extraction of statements as snippets from dissimilar code without attention score. The mean and variance of the five runs are presented. In certain cases, identical results are observed across all five runs, resulting in a standard variance of zero.

where $BLEU_{weight}$ is the weighted keyword n-gram match. $Match_{ast}$ and $Match_{df}$ are scores considering tree structures of AST and data-flow of code, respectively. α, β, γ, and δ are hyper-parameters.

4.3 Overall Results

We perform experiments with 9 victim models for 3 tasks fine-tuned from 3 pre-trained PL models. Baselines are MHM (Zhang et al., 2020) and ALERT (Yang et al., 2022) which are the state-of-the-art baselines in black-box attacks on the PL model. Experiments demonstrate that DIP outperforms the baselines in all metrics (ASR, $Query$, $Pert$, and $CodeBLEU$) as shown in Table 1. Our proposed method, DIP, shows the best ASR and $Query$ on all victim models, which means that DIP more effectively attacks the victim models than the baselines. Compared to the baseline MHM, DIP improves the average ASR by 74.9%, and the average $Query$ by 431.5% which is 5 times more efficient. Compared with the strong baseline ALERT, DIP also improves the average ASR by 28.7%, and the average $Query$ by 333.9% which is 4 times more efficient. $Pert$ and $CodeBLEU$ measure the quality of generated code. DIP shows the best $CodeBLEU$ on all the victim models, and the best $Pert$ on most victim models, which indicates more semantic-preserving (as functionality). Since variable names play important roles to represent code semantics (Wang et al., 2021), code semantics may be easily broken by variable renaming-based methods.

4.4 Ablation Study

We perform an ablation study with each component of DIP to verify the effectiveness of dissimilar code selection, vulnerable position selection, and snippet extraction. We evaluate with the CodeBERT fine-tuned for the clone detection. Table 2 shows the results of the ablation study on DIP. We replace the code selection by dissimilarity score (Eq. 2) with random selection in DIP *w/o Dissim*, the position selection by vulnerability score with random selection in DIP *w/o Position*, and the snippet extraction by attention score with random extraction in DIP *w/o Att-line*. As shown in Table 2, all the ablated DIPs show lower ASR, and higher $Pert$ and $Query$ than the original DIP. We verify that all the components are very effective for PL model attack. DIP *w/o Dissim* randomly selects the code which a snippet is extracted from, resulting in decrease of ASR and increase of $Pert$. It shows that our dissimilar code selection effects much on the attack quality. DIP *w/o Position* and DIP *w/o Att-line* show similar attack qualities (ASR and $Pert$), but lower attack efficiencies ($Query$). DIP *w/o Att-line* shows $Query$ about 6 times higher. Our position selection and snippet extraction methods play important roles to increase the attack efficiency. More various snippet extractions are evaluated in Appendix C.

Method	MHM	$ALERT$	DIP	All
MHM	2.5	1.5	9.2	1.0
ALERT	5.5	4.5	**28.1**	4.5
DIP (ours)	**32.0**	**41.5**	15.0	**8.0**

Table 3: ASR on adversarially trained models. MHM, $ALERT$, and DIP are the models adversarially trained by MHM, ALERT, and DIP, respectively. All is adversarially trained with all the three attack methods.

4.5 Attack on Adversarially Trained Models

We test the attack performance of MHM, ALERT, and DIP against adversarially trained models. We conduct experiments with the CodeBERT fine-tuned for clone detection. In Table 3, MHM, $ALERT$, and DIP denote the models trained with adversarial examples generated by MHM, ALERT, and DIP, respectively. All denotes an adversarially trained model by all three attack methods. Zhang et al. (2020) and Yang et al. (2022) reported their adversarial examples improving the robustness of target models. However, as shown in Table 3, DIP successfully attacks the adversarially trained models by MHM and ALERT. Adversarial training should improve the robustness of the model, but MHM and ALERT are not enough to increase

Adversarial Examples	Code-BERT	G. Code-BERT	Code-T5
against CodeBERT	-	31.3	22.0
against G.CodeBERT	12.6	-	17.8
against CodeT5	24.8	29.3	-

Table 4: Transferability of our proposed method. We use ASR as evaluate metric. Columns are the tested victim models, and rows are adversarial examples against each model.

robustness. If we combine all the three methods, the model becomes hard to attack. DIP shows the highest ASR, but it is relatively low. Adversarial training with combination of variable renaming and dead code insertion may improve the robustness of models.

4.6 Transferability

In this section, we test the transferability of our adversarial examples. We obtain adversarial examples of successful attacks on victim models to attack other models. As shown in Table 4, adversarial examples by DIP are transferable in the clone detection task. This experiment showed the potential of DIP in the other tasks.

5 Discussion

We discuss the compliability and detectactibility for the attack on PL models.

5.1 Complilability

When we modify the source code to attack PL models, we should consider compilability. If a source code is modified, it must be guaranteed to be compiled. DIP is guaranteed to compile in any case unlike MHM and ALERT, which are variable renaming methods. As mentioned in §1, we can easily find uncompilable cases modified by ALERT. In Appendix E, we present an example.

5.2 Dead Code Detection

Since we use an unused string variable, it could be filtered out by a dead code detector. However, it would be very hard to detect dead code by the static analysis because it is theoretically equivalent to the halting problem. We may simply eliminate code which is not locally accessed in a function. However, the dead code may include global variables. Simple elimination of statements including global variables may cause critical problems. If we add unused variables as if they are global, it would be very hard to detect them with a high certainty. For this reason, there are very few tools for dead code elimination. Most existing dead code eliminator tools, such as UCDetector and J2ObjC, they can detect only unused classes or method units, but not detect unused variables inside methods.

6 Related Work

The adversarial attack for language data (such as text, source code) is significantly difficult due to the discrete property of token embedding space. Both text and source code have a discrete property, but there is a difference in the attack methods. In the adversarial attack for natural language, the text should be considered for fluency and semantic consistency. The previous works (Maheshwary et al., 2020; Jin et al., 2020; Li et al., 2020; Garg and Ramakrishnan, 2020; Li et al., 2021; Wang et al., 2020; Guo et al., 2021a) proposed adversarial attack methods, which considered semantic and fluency of sentence. However, the source code is more important in compilability than fluency. In the adversarial attack for programming language, the source code should be considered for semantic (functionality) preserving and must be compiled. The previous works, Rabin et al. (2021) evaluated the generalizability of code2vec, code2seq, GGNN by using the various transformation (*Permute Statement, Boolean Exchange and Loop Exchange, etc.*). Zhang et al. (2020) proposed MHM, in black-box setting, which is the *Variable Renaming* based on metropolis-hastings sampling method. Yang et al. (2022) proposed ALERT, to improve MHM, which use the *Variable Renaming* based on genetic algorithm. Ramakrishnan et al. (2020); Yefet et al. (2020) proposed white-box attack method using the AST transformation and the output distribution into one-hot vector, which is represented variable name. Srikant et al. (2021) proposed white-box attack method to optimize the attack position and perturbation at the same time.

7 Conclusion

In this paper, we presented **DIP** (***D****ead code* ***I****nsertion based Black-box Attack for* ***P****rogramming Language Model*), the state-of-the-art black-box attack method. We proposed an attack method consist of vulnerable position selection, dissimilar code selection and snippet extraction. Experiment results demonstrate the high-performance and effectiveness of DIP, also we evaluated the transferability to apply other models. Under the black-box setting, our proposed method guaranteed compilability in any case and preserved functionality.

Limitation

We discuss some limitations of our study. Our proposed method needs a pre-trained model to obtain the attention score mentioned in Section 3.3. While our method achieved successful attacks with 4~5 times fewer queries compared to the baselines, the time spent on the attack was approximately half of the baselines. This is because preprocessing is necessary to calculate the score V in Section 3.1 and D in Section 3.2.

Acknowledgements

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-00421, AI Graduate School Support Program (Sungkyunkwan University), and No.2021-0-02068, Artificial Intelligence Innovation Hub). This work was also supported by Healthcare AI Convergence Research & Development Program through the National IT Industry Promotion Agency of Korea (NIPA) funded by the Ministry of Science and ICT (No. S0254-22-1006).

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *ArXiv*, abs/2103.06333.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29.

Bander Alsulami, Edwin Dauber, Richard E. Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source code authorship attribution using long short-term memory based networks. In *European Symposium on Research in Computer Security*.

YunSeok Choi, JinYeong Bak, CheolWon Na, and Jee-Hyong Lee. 2021. Learning sequential and structural information for source code summarization. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 2842–2851.

YunSeok Choi, Hyojun Kim, and Jee-Hyong Lee. 2022. TABS: Efficient textual adversarial attack for pre-trained NL code model using semantic beam search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5490–5498, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

YunSeok Choi, CheolWon Na, Hyojun Kim, and Jee-Hyong Lee. 2023. Readsum: Retrieval-augmented adaptive transformer for source code summarization. *IEEE Access*.

Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2021. Towards learning (dis)-similarity of source code from program contrasts. In *Annual Meeting of the Association for Computational Linguistics*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Siddhant Garg and Goutham Ramakrishnan. 2020. BAE: bert-based adversarial examples for text classification. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 6174–6181. Association for Computational Linguistics.

Chuan Guo, Alexandre Sablayrolles, Hervé Jégou, and Douwe Kiela. 2021a. Gradient-based adversarial attacks against text transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 5747–5757. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Annual Meeting of the Association for Computational Linguistics*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021b. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Judith F Islam, Manishankar Mondal, and Chanchal K Roy. 2016. Bug replication in code clones: An empirical study. In *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 68–78. IEEE.

Paras Jain, Ajay Jain, Tianjun Zhang, P. Abbeel, Joseph Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. In *Conference on Empirical Methods in Natural Language Processing*.

Akshita Jha and Chandan K. Reddy. 2022. Codeattack: Code-based adversarial attacks for pre-trained programming language models. *ArXiv*, abs/2206.00052.

Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. 2020. Is BERT really robust? A strong baseline for natural language attack on text classification and entailment. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 8018–8025. AAAI Press.

Cory J Kapser and Michael W Godfrey. 2008. “cloning considered harmful” considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692.

Bruno Laguë, Daniel Proulx, Jean Mayrand, Ettore M Merlo, and John Hudepohl. 1997. Assessing the benefits of incorporating function clone detection in a development process. In *1997 Proceedings International Conference on Software Maintenance*, pages 314–321. IEEE.

Dianqi Li, Yizhe Zhang, Hao Peng, Liqun Chen, Chris Brockett, Ming-Ting Sun, and Bill Dolan. 2021. Contextualized perturbation for textual adversarial attack. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 5053–5069. Association for Computational Linguistics.

Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. 2020. BERT-ATTACK: adversarial attack against BERT using BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 6193–6202. Association for Computational Linguistics.

Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192.

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. *ArXiv*, abs/2102.04664.

Rishabh Maheshwary, Saket Maheshwary, and Vikram Pudi. 2020. Generating natural language attacks in a hard label black box setting.

Audris Mockus. 2007. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*, pages 7–7. IEEE.

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*, 135:106552.

Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas W. Reps. 2020. Semantic robustness of models of source code. *CoRR*, abs/2002.03043.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv*, abs/2009.10297.

Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2021. Generating adversarial computer programs using optimized obfuscations. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Jeffrey Svajlenko and Chanchal K. Roy. 2015. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 131–140. IEEE Computer Society.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Tianlu Wang, Xuezhi Wang, Yao Qin, Ben Packer, Kang Li, Jilin Chen, Alex Beutel, and Ed H. Chi. 2020. Cat-gen: Improving robustness in NLP models via controlled adversarial text generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 5141–5146. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *ArXiv*, abs/2109.00859.

Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. *CoRR*, abs/2201.08698.

Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.*, 4(OOPSLA).

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*, volume 34, pages 1169–1176.

Yaqin Zhou, Shangqing Liu, J. Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *ArXiv*, abs/1909.03496.

A Statistics of Victim Models

Task	Model	Acc.
Clone Detection (# of Classes: 2)	CodeBERT	97.3%
	GraphCodeBERT	97.8%
	CodeT5	97.1%
Defect Detection (# of Classes: 2)	CodeBERT	63.3%
	GraphCodeBERT	64.2%
	CodeT5	63.7%
Authorship Attribution (# of Classes: 66)	CodeBERT	82.6%
	GraphCodeBERT	81.1%
	CodeT5	85.6%

Table 5: Statistics of Victim Models

B Length of dissimilar code

The K is the key hyperparameter as the number of candidate dissimilar codes. Figure 3 shows performance by K on the DIP. As seen in Figure 3 , the ASR and $Query$ increase with K increasing. We confirmed, increasing K from 1 to 30, the ASR increases significantly but hardly increases after that. When K is increasing, ASR increases because the search space is larger for attacking to victim model. However, when K is over 30, the ASR per $Query$ gets lower. It is inefficient in terms of $Query$.

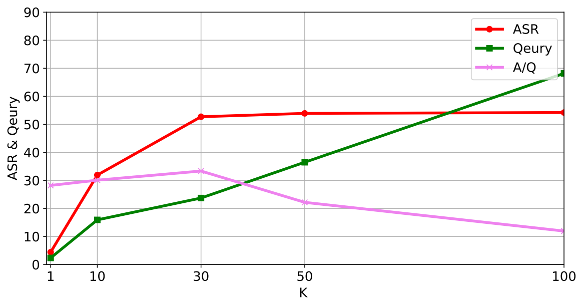


Figure 3: Using different parameter K from 1 to 100 in our attack method. The ASR and A/Q increase until K to 30

C Snippet Extraction

We compare various snippet extraction methods to generate dead statements as shown in Table 6. We analyze with DIP to observe the effects of snippet extraction method. The types of additional snippet are 4, which include the following:

- `[RandSeq-N]` is to randomly extract a sequence of N tokens from dissimilar code. The sequence can start at any token of dissimilar code.
- `[Rand-N]` is to randomly extract N tokens (not sequentially) from dissimilar code. Each token is extracted independently.

Snippet	ASR%	Pert	Query
Method: DIP			
RandSeq-10	42.7	0.09	131.6
Rand-10	21.1	0.09	154.4
RandSeq-5	23.4	0.06	180.2
Rand-5	11.9	0.06	238.1
RandSeq-5%	38.8	0.08	104.4
Natural language	0.1	0.40	1.00

Table 6: Performance by various snippet extractions.

- `[RandSeq-N%]` is same as `RandSeq-N` except that the length of the extracted sequence is proportional to the length of dissimilar code. Its length is N% of dissimilar code length.
- `[Natural language]` is to insert text in natural language not in programming language. We insert "*I want to fool this model. Hello, World! Python is a very simple and powerful language, and has a very straightforward syntax*".

`RandSeq-N` performs better than `Rand-N`. Since the model learns sequential information of source code during pre-training, consecutively extracted tokens are more effective then independently extracted ones. If we compare `RandSeq-N%` to `RandSeq-N`, it is better than `RandSeq-N`. We may conclude that it is effective to extract one whole statement as a snippet. It is interesting that a statement in natural language does not have any effect on adversarial attack. Since statements in natural language is very common in string variables, models may know how to handle it.

D Runtime Comparison

Our proposed method is very efficient, and less time-consuming. We reduce the search space by inserting a dead statement, not tokens, and obtain dissimilar codes after random sampling from code sets. The baseline methods try to rename variables with a large number of token combinations. Thus their methods are inefficient and high time-consuming. When attacking the PL models, DIP is about 3 times faster than MHM, and 2 times faster than ALERT.

E Qualitative Example

Original Code	private static void readAndRewrite(File inFile, File outFile) throws IOException {
	  // Some code . . .
	  ImageOutputStream out = ImageIO.create. . . Stream(new Buffere. . . Stream(new File..Stream(outFile)));
	  ds.writeDataset(out, dcmEncParam);
	  ds.writeHeader(out, dcmEncParam, Tags.PixelData, dcmParser.get. . . VR(), dcmParser.get. . . Length());
	  System.out.println("writing " + outFile + ". . . ");
	  // Some code . . .

ALERT	private static void readAndRewrite(File inFile, File outFile) throws IOException {
	  // Some code . . .
	  ImageOutputStream url = ImageIO.create. . . Stream(new Buffere. . . Stream(new File..Stream(outFile)));
	  ds.writeDataset(url, dcmEncParam);
	  ds.writeHeader(url, dcmEncParam, Tags.PixelData, dcmParser.get..VR(), dcmParser.get. . . Length());
	  System.url.println("writing " + outFile + ". . . ");
	  // Some code . . .

DIP(our)	private static void readAndRewrite(File inFile, File outFile) throws IOException {
	  // Some code . . .
	  ImageOutputStream out = ImageIO.create. . . Stream(new Buffere. . . Stream(new File..Stream(outFile)));
	  String out_2 ="r = new BufferedReader(new InputStreamReader(url.openStream()))";
	  ds.writeDataset(out, dcmEncParam);
	  ds.writeHeader(out, dcmEncParam, Tags.PixelData, dcmParser.get..VR(), dcmParser.get. . . Length());
	  System.out.println("writing " + outFile + ". . . ");
	  // Some code . . .

Table 7: The example for the qualitative comparison. The first example is Original Code. The second/third are generated adversarial examples by ALERT, DIP, respectively. The example of ALERT is not compiled because the variable *out*, which should not be changed, has been changed to *url*. In the example of DIP, red color is wrapper(`unused string variable`) and orange color is high attention snippet. The example of DIP is compiled.

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
"Limitation" section after the 7. Conclusions section

☑ A2. Did you discuss any potential risks of your work?
5.2

☑ A3. Do the abstract and introduction summarize the paper's main claims?
1

☒ A4. Have you used AI writing assistants when working on this paper?
Left blank.

B ☒ Did you use or create scientific artifacts?

Left blank.

☐ B1. Did you cite the creators of artifacts you used?
No response.

☐ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
No response.

☐ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
No response.

☐ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
No response.

☐ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
No response.

☐ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.
No response.

C ☒ Did you run computational experiments?

Left blank.

☐ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
No response.

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

☐ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
No response.

☐ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
No response.

☐ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
No response.

D ☒ Did you use human annotators (e.g., crowdworkers) or research with human participants?

Left blank.

☐ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
No response.

☐ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
No response.

☐ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
No response.

☐ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
No response.

☐ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
No response.