BLACK-BOX ADVERSARIAL ATTACK GUIDED BY MODEL BEHAVIOR FOR PROGRAMMING PRE-TRAINED LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Pre-trained models for programming languages are widely used to solve code tasks in Software Engineering (SE) community, such as code clone detection and bug identification. Reliability is the primary concern of these machine learning applications in SE because software failure can lead to intolerable loss. However, deep neural networks are known to suffer from adversarial attacks. In this paper, we propose a novel black-box adversarial attack based on model behaviors for pre-trained programming language models, named Representation Nearest Neighbor Search(RNNS). The proposed approach can efficiently identify adversarial examples via variable replacement in an ample search space of real variable names under similarity constraints. We evaluate RNNS on 6 code tasks (e.g., clone detection), 3 programming languages (Java, Python, and C), and 3 pre-trained code models: CodeBERT, GraphCodeBERT, and CodeT5. The results demonstrate that RNNS outperforms the state-of-the-art black-box attacking method (MHM) in terms of both attack success rate and quality of generated adversarial examples.

1 INTRODUCTION

A great deal has been accomplished with deep learning in many fields, such as image recognition (Li, 2022) and natural language processing (Otter et al., 2020). However, it is well known that deep learning systems are vulnerable to adversarial attacks (Szegedy et al., 2013) which are able to trick machine learning algorithms by perturbing inputs slightly without changing semantic context. Normally, adversarial attacks can be divided into three categories depending on the extent to which attackers can access victim deep learning models (e.g., model gradient and model architecture). 1) White-box attack, which can fully access the internal information of victim models, e.g., FGSM (Goodfellow et al., 2014). 2) Black-box attack, which attacks the victim models only using the final output information, such as Label-Only Attack (Ilyas et al., 2018). 3) Grey-box attack, which partially accesses victim models, e.g., additionally training a victim model (Xu et al., 2021). To defend against adversarial attacks, multiple approaches have been proposed (Qiu et al., 2019). In which, adversarial training (AT) (Bai et al., 2021) is the most effective defense practice to enhance the robustness of deep learning systems. Specifically, AT maximizes the permutation within the limitation while minimizing training loss, by including adversarial examples for training.

Recently, Hindle et al. (2016) and Allamanis et al. (2018) demonstrate that code data has properties in common with natural language which leads a hot direction in the SE community – machine learning for source code. The pre-trained deep learning models for programming languages have been proposed to solve code tasks where the transformer architecture (e.g., CodeBERT (Feng et al., 2020)) is most popular due to its promising performance. However, the same as deep learning models in other fields, code models also suffer from adversarial attacks (Yefet et al., 2020). The reliability of code models and their related applications is under exploration.

Different from adversarial attacks on image and natural language data, adversarial attacks on source code have more constraints in syntax correction and semantic reservation, e.g., all the identifier names in a valid code should follow the syntax of the programming language. Besides, black-box adversarial attacks are more practical for code models since they are usually embedded in IDEs and the internal information is not available, for example, Tabnine in PyCharm. Recently, some

black-box attacks on code models have been proposed which generate adversarial examples by variable name replacement based on generation models or rule-based modifications (Yang et al., 2022; Srikant et al., 2021). The major limitations of existing black-box attack methods are: 1) the search space size is limited, 2) the generated alternatives have a data shift from the real data from the developers, and 3) they do not fully utilize model uncertainty and failed attacks, which could also be helpful to teach the subsequent attacking trial

In this paper, we present a black-box attack on pre-trained models for programming languages based on model uncertainty, named **Representaion Nearest Neighbor Search**(**RNNS**). Specifically, first, RNNS utilizes real code data to construct the search space based on semantic similarity. Then, it uses an efficient search way guided by the domain probability change to find the adversarial examples in the real name space. Importantly, failure attacking trials are used to teach the next round of attacking by the memory mechanism in RNNS. To show the effectiveness and efficiency of RNNS, we investigate three (3) big code pre-trained models CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020) and CodeT5 (Wang et al., 2021), and evaluate our approach on six (6) code tasks in multiple programming languages. The results on 18 victim models demonstrate that compared to the SOTA approach MHM, RNNS achieves a higher attack access rate(ASR) with a maximum of about 100% improvement and 18/18 times as the winner. Furthermore, we analyze the adversarial examples statistically and find that RNNS introduces smaller perturbation than MHM. In the end, we apply the RNNS to attack three (3) defended models. In summary, our main contributions are:

- We propose a novel black-box adversarial attack RNNS for code models guided by model behavior change and searching in the real identifier name space from actual code data.
- We demonstrate that RNNS outperforms SOTA methods, and introduce perturbation into inputs slightly in terms of number of modified variables and code length.

2 Related Work

Adversarial Attacks in NLP. Zhang et al. (2020b) categorize adversarial methods into three (3) groups, character-level, word-level and sentence-level attacks based on attacking granularity. A character-level adversarial attack modifies the characters in a word. DeepWordBug (Gao et al., 2018) applies simple character-level transformations to the text inputs. VIPER (Eger et al., 2019) replaces characters with their nearest neighbors in a visual embedding space. A word-level adversarial attack first finds the risky word, and then replace it with the alternative word. TextBugger (Li et al., 2019) fools the models by modifying the vulnerable word. BERT-Attack (Li et al., 2020) uses BERT to generate the adversarial substitues to replace the vulnerable word while preserving semantics. BAE (Garg & Ramakrishnan, 2020) also utilizes the BERT capability of predicting masking tokens to generate adversarial examples. A sentence-level adversarial attack generates fake inputs directly or manipulates multiple words while keeping inputs semantically equivalent. Zhao et al. (2018) uses Generative Adversarial Networks (GAN) to output adversarial samples. SEA (Ribeiro et al., 2018) constructs a set of universal replacement rules to generate semantically equivalent adversarial examples. Except that TextBugger (Li et al., 2019) can be configured under both white-box and black-box setting, all aforementioned adversarial attacks belong to the black-box group. Overall, with the exception of generating fake inputs directly, all adversarial attacks must first locate where should change and then decide what should be filled.

Adversarial Attack in Code Models. We introduce the state-of-the-art(SOTA) adversarial attacks for code models based on the concealment levels of model information, i.e., black-box and white-box attacks. A black-box attack for code models queries the model outputs and selects the substitutes using a score function. NaturalAttack (Yang et al., 2022) finds the adversarial examples using variable-name substitutes generated by pre-trained masked models for programming language. MHM (Zhang et al., 2020a) uses Metropolis–Hastings to sample the replacement of code identifiers. STRATA (Springer et al., 2020) generates adversarial examples by replacing the code tokens based on the token distribution. Chen et al. (2022) apply pre-defined semantics-preserving code transformations to attack code models. CodeAttack (Jha & Reddy, 2022) uses code structure to generate adversarial data. White-box attacks require the code model gradient to modify inputs for adversarial example generation. CARROT (Zhang et al., 2022) selects code mutated variants based on the model gradient. Ramakrishnan et al. (2020) attack code models by gradient-based optimization of the abstract syntax tree transformation. Srikant et al. (2021) uses optimized program obfuscations

to modify the code. DAMP (Yefet et al., 2020) derives the desired wrong prediction by changing inputs guided by the model gradient.

In black-box attacks, model inside information is not accessed. White-box attacks are quite effective but are limited by the availability of model information. Our approach considers the identifier replacement like MHM (Zhang et al., 2020a), ensuring that the adversarial example keeps the same semantic (code behavior) as the original one. In our approach, we locate vulnerable variables based on the victim model uncertainty, and search in the substitute embedding space guided via model behavior signals.

3 METHODOLOGY

3.1 PRELIMINARY OF PROCESSING TEXTUAL CODE

The nature of code data (in text format with discrete input space) makes that it is impossible to feed one code input x directly into deep learning models. Thus, transferring code data to learnable continuous vectors is the first step in source code learning. **Dense encoding** is one common method used to vectorize textual code data. First, we need to learn a tokenizer that splits the code text into a token sequence called **Tokenization**. It can alleviate the Out-of-Vocabulary(OOV) problem and reduce vocabulary size by reusing the elemental tokens. After tokenization, code x is represented by a sequence of tokens, namely, $x = (s_0, ..., s_j, .., s_l)$ where s_i is one token. The code vocabulary dictionary consists of s_i, denoted $\mathbb{V}$. Every word(token) in $\mathbb{V}$ is embedded by a learned vector $\boldsymbol{v}_i$ with dimension d. We use $\boldsymbol{E}^{|\mathbb{V}|\times d}$ to represent the embedding matrix for $\mathbb{V}$. Then, x can be converted into a embedding matrix $\boldsymbol{R}^{l\times d} = (\boldsymbol{v}_0, ..., \boldsymbol{v}_j, .., \boldsymbol{v}_l)$. Pre-trained code models based on the transformer take the matrix $\boldsymbol{R}^{l\times d}$ as inputs, and learn the contextual representation of x for downstream tasks via Masked Language Modeling (MLM) and Causal Language Modeling (CLM).

Figure 1 shows the critical steps of the code models for the downstream classification tasks. First, we tokenize the textual code x into a token sequence that is represented in a discrete integer space. Then, we map the discrete sequence ids into the token vector space $R^{l\times d}$. Next, we feed the token vectors into the task model $f(\theta)$. $f(\theta)$ is built on the top pre-trained models. Finally, we can predict the domain probabilities after fine-tuning.

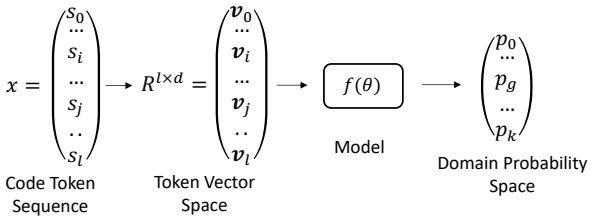

Figure 1: One code model demo on the downstream task

3.2 PROBLEM STATEMENT

Considering a code classification task, we use $f(x;\theta) \rightarrow y : \boldsymbol{R}^{l\times d} \rightarrow \mathbb{C} = \{i|0 \leq i \leq k\}$ to denote the victim model that maps a code token sequence x to a label y from a label set $\mathbb{C}$ with size k, where l is the sequence length and d is the token vector dimension, and i is one integer. By querying dictionary dense embedding $\boldsymbol{E}^{|\mathbb{V}\times d|}$, a code token sequence $x = (s_0, ..., s_j, .., s_l)$, is vectorized into $\boldsymbol{R}^{l\times d}$. **Adversarial attacks** for code models create an adversarial example x' by modifying some vulnerable tokens of x limited in a **maximum perturbation** ϵ to change the correct label y to a wrong label y'. Simply, we get a perturbed x' by modifying some tokens in $(s_0, ..., s_j, .., s_l)$ such that $f(x';\theta) \neq f(x;\theta)$ where $x' = x + \sigma$, $+$ represent perturbation execution, σ is the perturbation code transformation for $(s_0, ..., s_j, .., s_l)$, and $\sigma \leq \epsilon$. For black-box attacks, the model parameters are not available.

There are two main dynamic limitations of adversarial code attacks. First, the adversarial example can be compiled. Second, it should not change code semantics, meaning the adversarial example should behave exactly the same as the original one. These limitations differ from adversarial attacks for image and natural language data. Code transformations are designed to attack code models but maintain the code semantics during an attack, e.g., variable replacement and rewriting condition statements in another format.

3.3 PROPOSED METHOD: REPRESENTAION NEAREST NEIGHBOR SEARCH(RNNS)

3.3.1 MOTIVATION

A hidden correlation exists between the token vector space and the domain probability space in Figure 1, which $f(\theta)$ can learn. As a result, changes in the token vector space can be reflected in the domain probability space and vice versa. These motivate us to use the change trending of the domain probability space to find adversarial identifiers in the token vector space. Our approach is one black-box adversarial attack, which means the token vector space in Figure 1 is not accessible because it is part of downstream task models after fine-tuning. Instead of the fine-tuned token embedding encoder, we use pre-trained models trained by MLM or CLM as the surrogate encoder to infer variable generalization embedding. We can pre-train such a surrogate encoder from scratch by ourselves, but it makes no difference that we use public pre-trained models.

Our approach considers the replacement perturbation of variable names for code attacking. Variable Replacement can guarantee the code is still syntax correct and semantically equivalent to the original code after modification. Token Replacement is also shown as the most significant transformation (Li et al., 2022). The variable-replacement adversarial attack has two steps. First, we should locate **where we should attack** and then decide **what adversarial tokens** we should use. Except for renaming variables, it is unclear whether structural changes are equivalent to the original code, known as the equivalent mutant detection (Papadakis et al., 2019) in Software Engineering. It implies that the generated example can have different functionality/semantics from the original code.

3.3.2 METHODOLOGY

Figure 2 demonstrates the overview of our approach, RNNS. Generally, it has two steps where we should attack (①, ②) and what adversarial tokens we should use (③-⑦). Initially, we uses Variable Extractor to obtain a list of the variables and their corresponding positions in the code input (step ①). Later, at the step ②,we rank the list based on uncertainty of the variables via Uncertainty Ranking Estimator. Next, we pop the first variable and get its vector representation by Variable Name Encoder (step ③). We query the real variable dataset and randomly select K replacements for this variable (step ④). Then, we evaluate these K replacements and choose the best one by Replacement Evaluator (step ⑤). After replacement, we query the victim model to see if the label is changed. If it is changed, we stop attacking. Otherwise, we will check if the confidence of the victim model for ground truth label decreases. If the confidence increases, we move to next variable in the ranked list and go back to the step ③. If we find the victim model reduces its confidence for the ground truth label, we update our search seed by Search Seed Generator (step ⑥) and query the Top-K replacement set again (step ⑦). We continue the loop ⑤ - ⑥ - ⑦ - ⑤ until we find one adversarial example, or we reach the maximum loop iterations and move to next variable in the ranked list (step ③).

When RNNS searches the adversarial examples, it has **tow constraints**:

1) similarity distance,

$$1 - similarity(v_{adv}, v_{org}) < \epsilon \tag{1}$$

2) variable length change

$$|length(v_{adv}) - length(v_{org})| < \delta \tag{2}$$

,where v_{adv} is the adversarial variable and v_{org} is the original variable. The successful adversarial examples have to satisfy both at the same time.

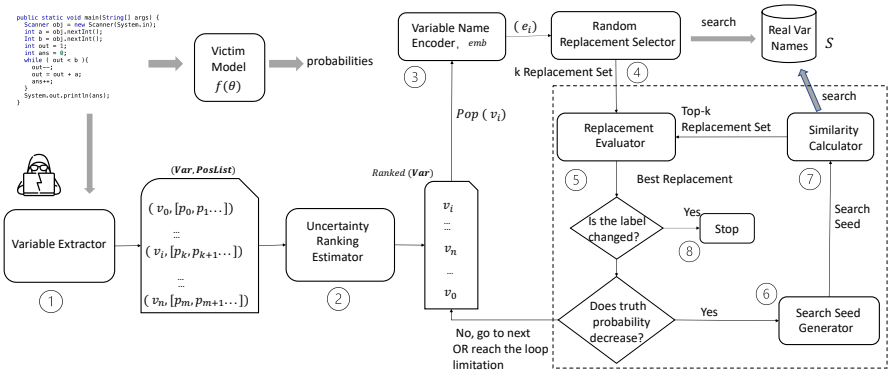

Figure 2: Overview of RNNS

Depending on if using the change direction of the search seed or if using the change direction historical information at the step (6), our approach RNNS has three variants, RNNS-Smooth, RNNS-Direct and RNNS-Raw. RNNS-Smooth combines the current change direction of the search seed and its change direction history. RNNS-Direct only considers the current change direction. RNNS-Raw does not use either of them.

Algorithm 1 shows the details of RNNS-Smooth. As an overview, line 1 is **where** we should attack (line 1) and lines 2-23 are **what adversarial tokens** we should use. In the first stage, we use model uncertainty to assign different priorities to variable positions. In the second stage, we start attacking the model to find adversarial examples.

RNNS has some hyperparameters, as shown at the beginning of Algorithm 1. max_itr defines the maximum attacking iterations. K is the number of adversarial candidates we consider at each trial. α is used to smooth the moving/changing direction. ϵ and δ are used to limit the maximum perturbation. The maximum token similarity distance should be within ϵ, and the maximum string length difference should be less than δ. $dist$ is the distance metric, and the default is cosine distance. RNNS is independent of victim downstream-task models. We need one encoder (denoted as emb) that can map one variable var into an independent token vector space, denoted as $\boldsymbol{e}_{var}$. We use the pre-trained CodeBERT model as the surrogate encoder without domain knowledge about downstream tasks. If want, it is possible to pre-train the surrogate encoder from scratch, but it makes no difference under the same pre-training settings. Search Space $\mathbb{S}$ consists of variable names from the actual code.

Before we go through Algorithm 1, we introduce some important symbols about Search Seed Generator ((6)). $\boldsymbol{e}_{var}$ is the model-independent vector representation of var from **Variable Name Encoder** emb ((3)) . $\Delta\boldsymbol{e}$ is the difference between the current adversarial variable state $\boldsymbol{e}_{cur}$ and the last adversarial variable state $\boldsymbol{e}_{pre}$, defined by $\Delta\boldsymbol{e} \leftarrow \boldsymbol{e}_{cur} - \boldsymbol{e}_{pre}$ at line 12. $\Delta\boldsymbol{e}_{smo}$ is the smooth history memory for $\Delta\boldsymbol{e}$, and it is initialized at the first attacking iteration(line 14), and updated by $(1-\alpha)\Delta\boldsymbol{e}_{smo} + \alpha\Delta\boldsymbol{e}$(line 15), where α is the smooth rate between 0 and 1.

First, we extract all variables $Vars$ and their positions from x ((1)). Then, we need to rank the pairs $(Vars, PosList)$ according to the descending order of model uncertainty at line 1 ((2)). $\mathbf{Rank\_Uncertainty}$ replaces each variable with a group of pre-defined random variables with different lengths and then measures model output uncertainty(variance).

Next, we start attacking the ranked pairs (var, pos). Line 3 initializes the embedding vectors used later ((3)). In line 4, we randomly select $topK$ replacement candidates within the distance to $\boldsymbol{e}_{var}$, that is less than ϵ ((4)). Then, we iteratively search adversarial examples. Given a list of variables $topK$, $\mathbf{Get\_Cur\_Best}$ returns the variable var^{cur} with the lowest probability p_g^{cur} of the ground truth label g for x' ((5)). Boolean value is_suc marks if var^{cur} can make $f(\theta)$ mis-classify. If is_suc is false and the current sate probability p_g^{cur} decreases compared with last time p_g^{pre}, we

update $\Delta \boldsymbol{e}_{smo}$ using $\Delta \boldsymbol{e}$ and then make the current state $\boldsymbol{e}_{cur}$ move in this direction(lines 15-16, (6)), denoted $\boldsymbol{e}_{tmp}$. Next, $\mathbf{Get\_TopK}$ computes the top similar variables with $\boldsymbol{e}_{tmp}$ in $\mathbb{S}$ under the two limitations ((7)). $\mathbf{Get\_TopK}$ contains two constraints of RNNS as described by Equation 1 and Equation 2. First, the similarity limitation is less than ϵ between candidates and $\boldsymbol{e}_{tmp}$. Second, the length difference limitation is less than δ between candidates and $\boldsymbol{e}_{var}$. Our approach continues searching the adversarial example until it reaches the quit conditions: 1) finding one adversarial example, 2) reaching the maximum iteration and moving to next variable trial ((3)).

Algorithm 1: RNNS-Smooth

Hyperarameter: maximum attacking iteration max_itr, maximum candidates K, direction smooth rate α, maximum distance ϵ, maximum length difference δ, distance metrics $dist$
Input: code x with ground label g, search space $\mathbb{S}$, victim model $f(\theta)$, independent token encoder emb
Output: adversarial example x', attacking success is_suc
Initialization:
$x' \leftarrow x$, $p_g^{pre} = f_g(x)$, $(Vars, PosList) \leftarrow VarExtractor(x)$, $i \leftarrow 1$
1 $(Vars, PosList) \leftarrow Rank\_Uncertainty(Vars, PosList, x, f(\theta))$
2 **for** $(var, pos) \in (Vars, PosList)$ **do**
3 $\boldsymbol{e}_{var} \leftarrow emb(var)$, $\Delta \boldsymbol{e}_{smo} \leftarrow 0$, $\boldsymbol{e}_{pre} \leftarrow \boldsymbol{e}_{var}$,
4 $topK \leftarrow random\_topK(\mathbb{S}, K, e_{var}, \epsilon, \delta, dist)$
5 **while** $i \leq max\_itr$ **do**
6 $is\_suc, var^{cur}, p_g^{cur} = Get\_Cur\_Best(f(\theta), x', g, pos, K))$
7 **if** is_suc **then**
8 $x' \leftarrow Replace(x', var, var^{cur})$
9 return x',is_suc
10 **else if** $p_g^{cur} - p_g^{pre} < 0$ **then**
11 $\boldsymbol{e}_{cur} \leftarrow emb(var^{cur})$
12 $\Delta \boldsymbol{e} \leftarrow \boldsymbol{e}_{cur} - \boldsymbol{e}_{pre}$
13 **if** $i == 1$ **then**
14 $\Delta \boldsymbol{e}_{smo} \leftarrow \Delta \boldsymbol{e}$
15 $\Delta \boldsymbol{e}_{smo} \leftarrow (1 - \alpha)\Delta \boldsymbol{e}_{smo} + \alpha \Delta \boldsymbol{e}$
16 $\boldsymbol{e}_{tmp} \leftarrow \boldsymbol{e}_{cur} + \Delta \boldsymbol{e}_{smo}$
17 $topK \leftarrow Get\_topK(\boldsymbol{e}_{tmp}, topK, \mathbb{S}, \boldsymbol{e}_{var}, \epsilon, \delta, dist)$
18 $p_g^{pre} \leftarrow p_g^{cur}$
19 $\boldsymbol{e}_{pre} \leftarrow \boldsymbol{e}_{cur}$
20 $var^{pre} \leftarrow var^{cur}$
21 $i \leftarrow i + 1$
22 **else**
23 $x' \leftarrow Replace(x', var, var^{pre})$
24 break
25 return x', False

By modifying Algorithm 1 (RNNS-Smooth), we can get the other two variants, RNNS-Direct and RNNS-Raw. First, RNNS-Direct does not use $\Delta \boldsymbol{e}$ history information $\Delta \boldsymbol{e}_{smo}$ and directly use $\Delta \boldsymbol{e}$ to update $\boldsymbol{e}_{tmp} \leftarrow \boldsymbol{e}_{cur} + \Delta \boldsymbol{e}$ (line 15). RNNS-Raw drops $\Delta \boldsymbol{e}$, $\boldsymbol{e}_{tmp} \leftarrow \boldsymbol{e}_{cur}$ (line 15). Figure 3 demonstrates their difference. Both of RNNS-Smooth and RNNS-Direct use $\Delta \boldsymbol{e}$ while RNNS-Smooth introduces the history of $\Delta \boldsymbol{e}$ for smoothing.

4 Experiments Setup

We first compare RNNS with SOTA black-box adversarial attacks (MHM and NaturalAttack) to evaluate its efficiency. Then, we study the quality of adversarial examples and see how many changes we introduce during the attack. Next, we remove the constraints of RNNS to see how these constraints affect our approach. Ultimately, we apply our approach to attack the defended models via adversarial training by NaturalAttack.

Dataset and Model. We evaluate our approach on six (6) datasets. BigCloneBench (Wang et al., 2020) is one code clone detection dataset. Devign (Zhou et al., 2019) is used for vulnerability

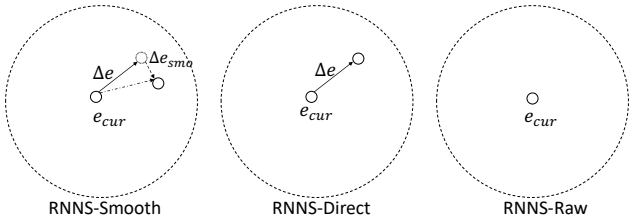

Figure 3: Difference among RNNS-Smooth, RNNS-Direct, RNNS-Raw

detection in C. Authorship (Alsulami et al., 2017) is for code owner classification. We also employ Java250, Python800 and C1000 from ProjectCodeNet (Puri et al., 2021), and they are about problem-solving classification tasks. We investigate the pre-trained models for programming languages CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020) and CodeT5 (Wang et al., 2021) on all six (6) datasets, and fine-tune 18 victim models. Table 1 summarizes the datasets and fine-tuned models in the experiments. We implement our approach in PyTorch and run all experiments on 32G-v100 GPUs.

Baseline. We compare our approach with MHM (Zhang et al., 2020a) and NaturalAttack (Yang et al., 2022). MHM is a sampling search-based black-box attack and generates the substitutes from the vocabulary based on lexical rules for identifiers. MHM is our primary baseline and we use it for all six (6) tasks. We compare RNNS with NaturalAttack on three (3) tasks. Then, we employ RNNS and MHM to attack three (3) defended models by adversarial training from NaturalAttack (Yang et al., 2022).

Evaluation Metric. To evaluate the effectiveness of adversarial attack methods, we employ the commonly used attack success rate (ASR) as the measurement. A higher ASR means that the attack method can generate more adversarial examples. ASR is defined as:

$$ASR = \frac{\text{Number of Successful Attacks}}{\text{Total Number of Attack}}$$

.

To evaluate the efficiency of the attack methods, we use query times (QT) to check the average number of querying the victim model for one attack. A smaller QT means the attack method can find the adversarial examples by visiting victim models in a less query times number. We do not use the running time as the measurement metric because it can be affected by many factors, e.g., the hardware, the running environment and the implementation. Finally, we use the change of replaced-variable length and the number of replaced variables to study the quality/perturbation of adversarial examples. A smaller score means the attack method can generate adversarial examples with less perturbation injection.

Parameter Setting. We set the maximum iteration $max\_itr = 6$, the number of candidates $K = 60$, the smooth rate $\alpha = 0.2$ the maximum length change $\delta = 4$ and the maximum cosine distance $\epsilon = 0.15$. The search space $\mathbb{S}$ consists of the variable names from the actual code. All variants of RNNS uses the same settings.

Table 1: Datasets and Victim Model Performance (Accuracy, %)

Task	**Train / Val / Test**	**CodeBERT**	**GraphCodeBERT**	**CodeT5**
Defect	21,854 / 2,732 / 2,732	63.76	63.65	74.27
Clone	90,102 / 4,000 / 4,000	96.97	97.36	97.84
Authorship	528 / – / 132	82.57	77.27	88.63
C1000	320,000 / 80,000 / 100,000	82.53	83.79	84.46
Python800	153,600 / 38,400 / 48,000	96.39	96.29	96.79
Java250	48,000 / 11,909 / 15,000	96.91	97.27	97.72

5 EVALUATION

5.1 COMPARISON RESULTS

We compare three variants of our approach with the state-of-the-art black box attacks MHM (Zhang et al., 2020a) and NatualAttack (Yang et al., 2022). Table 2 shows the comparison result with MHM. In conclusion, RNNS outperforms MHM. MHM fails to attack GraphCodeBERT on BigClone dataset and only has 9.58% ASR, while RNNS has more than 40% ASR. RNNS has almost two times larger ASR than MHM on Java250+CodeT5 and Python800+CodeT5. RNNS with the direction $\Delta \boldsymbol{e}$ can improve the attacking success rate (ASR) and reduce the query times, achieving 13 highest ASR and 11 lowest QT among 18 attacking tasks. RNNS-Smooth has almost the same performance as RNNS-Direct, and they are complementary to each other, 6/18 and 7/18, for different tasks, respectively. We also find that the victim models using CodeT5 are more robust to RNNS, which has lower ASRs than CodeBERT and GraphCodeBERT on all tasks except the defection detection. We also compare RNNS with NatualAttack (Yang et al., 2022) (named ALERT), as shown in Table 2. We can see that RNNS is much better than NatualAttack.

Table 2: Comparing Results with MHM, ASR %

Task+Model	**ALERT**		**MHM**		**RNNS-Smooth**		**RNNS-Direct**		**RNNS-Raw**	
	ASR	QT	ASR	QT	ASR	QT	ASR	QT	ASR	QT
Clone+CodeBert	28.67	2155.39	39.66	972.15	46.50	**666.48**	47.07	967.21	**47.79**	964.25
Clone+GrpahCodeBert	10.4	1466.68	9.58	**490.99**	41.28	1122.01	42.04	1121.81	**42.24**	1118.28
Clone+CodeT5	29.2	2359.70	38.79	1069.06	**39.61**	**895.79**	38.59	988.52	38.99	1001.93
Defect+CodeBert	52.29	1079.68	50.51	862.18	**69.18**	**588.35**	*69.18*	595.80	67.37	602.46
Defect+GrpahCodeBert	74.29	621.77	75.19	539.93	81.63	**404.73**	**83.67**	414.58	82.73	410.84
Defect+CodeT5	76.66	721.02	86.51	**344.08**	**89.45**	344.29	89.21	890.80	88.69	891.81
Authorship+CodeBert	34.98	**682.57**	64.70	775.11	73.39	1029.59	75.22	1002.26	**76.14**	972.82
Authorship+GrpahCodeBert	58.82	1227.36	75.49	**632.10**	**80.39**	696.64	78.43	709.90	77.45	715.87
Authorship+CodeT5	64.95	1078.40	66.97	**715.89**	**71.79**	970.44	68.37	981.98	70.94	981.05
Java250+CodeBert	50.5	958.96	74.03	961.60	75.12	815.91	**75.85**	**798.02**	73.58	825.17
Java250+GrpahCodeBert	46.74	1026.15	46.05	946.52	72.30	853.74	**73.23**	**851.30**	72.51	863.49
Java250+CodeT5	52.04	1189.42	30.59	1107.95	63.80	1049.46	**64.85**	**1034.10**	62.34	1130.23
Python800+CodeBert	58.3	513.63	56.67	919.37	77.88	514.19	**79.75**	**503.82**	78.40	516.34
Python800+GrpahCodeBert	51.87	577.70	54.15	917.92	71.42	730.14	**71.94**	**723.12**	69.04	760.71
Python800+CodeT5	52.84	777.20	36.95	1127.44	69.07	662.28	**69.70**	**658.50**	67.73	666.61
C1000+CodeBert	53.5	525.43	59.75	**340.88**	**72.96**	537.76	72.84	538.59	71.75	547.03
C1000+GrpahCodeBert	52.68	**566.18**	45.93	837.09	72.23	634.27	68.77	634.11	**72.47**	636.37
C1000+CodeT5	47.86	843.33	36.45	**668.15**	59.00	697.06	59.12	689.56	**59.24**	707.90
Count	0/18	2/18	0/18	6/18	6/18	4/18	7/18	6/18	5/18	0/18

5.2 ADVERSARIAL EXAMPLE STUDY

We count the original and adversarial variable length (Table 3). The second and fifth columns are the average length for original variables (Var Len) that are replaced. The third and sixth columns are the average length for adversarial variables(named Adv Var Len). The fourth and seventh columns are the average and variance ($mean \pm variance$) of the absolute length difference between original variables and adversarial variables (named Difference). We observe that MHM prefers to replace the long-length variables while RNNS likes replacing short-length variables if we compare the second and fifth columns. Meanwhile, the change of variable length is changed to be less than MHM by RNNS. MHM introduces the average length difference of 3.39-6.82 while RNNS only has 2.02-2.54. MHM has much higher variances than RNNS in the length change.

Table 4 statistically shows the number of replaced variables for our approach RNNS and MHM ($mean \pm variance$). It can be seen that RNNS replaces around an average of 3.6 variables with a smaller variance of around (3.4-4.6) while MHM needs to modify about an average of 5.4 variables with a larger variance ($\geq$ 11.14). We can summarize that RNNS introduces smaller perturbations into inputs when generating adversarial examples. From Table 4 and Table 3, we can see that RNNS is also competitive with ALERT, replacing less number of variables.

Figure 4 shows one example that both RNNS and MHM attack successfully from Java250 dataset. The left is the original code. The center is the adversarial example generated by RNNS. The right is the adversarial example from MHM. The changes are highlighted by shadow markers. RNNS only renames one variable **b** to **h**, while MHM almost renames all variables and also prefers longer names. Appendix A.1 shows more examples.

Table 3: Replaced-Variable Length Comparison ($mean \pm variance$)

Task+Model	RNNS-Smooth			MHM			ALERT		
	Var Len	Adv Var Len	Difference	Var Len	Adv Var Len	Difference	Var Len	Adv Var Len	Difference
Clone+CodeBert	6.12	6.79	2.35 ± 4.5	6.47	10.6	6.34 ± 10.98	5.91	6.21	1.32 ± 2.02
Clone+GraphCodeBert	6.32	6.97	2.54 ± 6.43	6.58	10.41	6.82 ± 21.67	5.5	5.93	1.45 ± 2.49
Clone+CodeT5	6.45	6.69	2.51 ± 8.3	6.46	10.46	6.17 ± 25.78	6.25	6.61	1.32 ± 2.72
Defect+CodeBert	4.64	5.44	2.08 ± 2.49	4.44	9.59	6.57 ± 28.78	4.85	5.06	1.36 ± 1.93
Defect+GraphCodeBert	4.08	5.34	2.13 ± 1.83	4.37	9.73	6.48 ± 26.51	4.47	5.22	1.33 ± 1.83
Defect+CodeT5	3.95	5.17	2.03 ± 1.93	4.33	9.81	6.59 ± 29.98	4.36	5.01	1.27 ± 1.57
Authorship+CodeBert	3.81	5.18	2.28 ± 1.56	3.97	7.94	5.45 ± 16.72	4.42	5.35	1.4 ± 2.25
Authorship+GraphCodeBert	3.69	5.23	2.36 ± 1.71	4.39	7.64	5.24 ± 15.38	3.74	4.46	1.22 ± 1.82
Authorship+CodeT5	3.95	5.18	2.03 ± 2.66	3.95	7.98	5.59 ± 20.94	3.81	4.5	1.22 ± 1.62
Java250+CodeBert	2.35	4.22	2.11 ± 1.02	3.21	6.5	4.34 ± 15.2	3.22	3.65	0.937 ± 1.63
Java250+GraphCodeBert	2.48	4.31	2.13 ± 1.07	3.13	6.59	4.42 ± 14.84	3.05	3.5	0.979 ± 1.54
Java250+CodeT5	2.76	4.47	2.1 ± 1.17	3.2	6.54	4.33 ± 14.6	3.16	7.31	4.41 ± 18.73
Python800+CodeBert	1.5	3.54	2.21 ± 1.02	1.97	5.11	3.64 ± 9.06	1.78	2.27	0.64 ± 1.34
Python800+GraphCodeBert	1.88	3.9	2.18 ± 0.78	1.99	6.01	4.46 ± 16.52	1.8	2.33	0.76 ± 1.3
Python800+CodeT5	1.65	3.59	2.13 ± 0.95	1.97	4.95	3.49 ± 8.18	1.88	5.84	4.1 ± 12.64
C1000+CodeBert	1.58	3.44	2.08 ± 0.88	2.41	5.05	3.65 ± 12.02	2.13	2.52	0.67 ± 1.17
C1000+GraphCodeBert	1.6	3.59	2.1 ± 0.85	2.39	5.35	3.9 ± 12.98	2.18	2.67	0.66 ± 1.23
C1000+CodeBert	1.38	3.33	2.02 ± 0.85	2.36	4.82	3.39 ± 10.98	2.1	6.56	4.74 ± 13.24

Table 4: Replaced-Variable Number Comparison, $mean \pm variance$

Task	CodeBERT			GraphCodeBERT			CodeT5		
	RNNS-Smooth	MHM	ALERT	RNNS-Smooth	MHM	ALERT	RNNS-Smooth	MHM	ALERT
Clone	3.55 ± 4.6	6.72 ± 16.57	6.86 ± 18.85	4.12 ± 4.94	6.21 ± 15.13	6.95 ± 18.99	3.43 ± 5	5.68 ± 14.01	7.65 ± 25.57
Defect	3.39 ± 4.96	2.78 ± 7.89	3.49 ± 3.99	2.67 ± 1.75	2.84 ± 9.5	4.1 ± 11.05	2.51 ± 1.45	2.16 ± 3.58	3.49 ± 3.99
Authorship	4.24 ± 7.47	7.52 ± 25.82	6.6 ± 22.96	3.65 ± 3.32	6.67 ± 22.29	7.75 ± 33.12	4.39 ± 9	5.72 ± 13.02	6.06 ± 18.74
Java250	3.87 ± 4.7	7.11 ± 21.18	7.82 ± 28.96	3.87 ± 4.25	6.41 ± 16.24	7.83 ± 25.06	4.71 ± 6.87	7.04 ± 15.29	8.92 ± 25.97
Python800	3.06 ± 1.87	5.21 ± 12.28	4.96 ± 8.47	4.12 ± 3.68	5 ± 10.83	4.63 ± 6.76	3.57 ± 3.04	5.29 ± 13.51	6.18 ± 11.45
C1000	3 ± 1.86	4.42 ± 7.49	4.13 ± 5.59	3.37 ± 2.38	5.14 ± 7.3	4.88 ± 6.24	3.39 ± 2.48	5.2 ± 7.43	5.43 ± 6.99
mean	$\mathbf{3.52 \pm 4.24}$	5.63 ± 15.21	5.65 ± 14.80	$\mathbf{3.63 \pm 3.39}$	5.38 ± 13.55	6.02 ± 16.87	$\mathbf{3.67 \pm 4.64}$	5.18 ± 11.14	6.29 ± 15.45

5.3 Ablation Study

In comparing, we remove the constraints of RNNS, named RNNS-Unlimited. Table 5 demonstrates the result. RNNS-Unlimited requires less query times (15/18) and achieves higher ASRs (18/18) in general than RNNS with constraints. Without any constraint, RNNS can find adversarial examples more easily.

Table 5: Removing Constraints

Task	CodeBERT				GraphCodeBERT				CodeT5			
	RNNS-Unlimited		RNNS-Smooth		RNNS-Unlimited		RNNS-Smooth		RNNS-Unlimited		RNNS-Smooth	
	ASR	QT	ASR	QT	ASR	QT	ASR	QT	ASR	QT	ASR	QT
Defect	**72.29**	361087	69.18	**359483**	**87.77**	**243604**	81.63	258216	**91.64**	**218614**	89.45	222412
Clone	**50.66**	930154	46.5	**648486**	**48.16**	**1083006**	41.28	1099569	**41.38**	899474	39.61	**875189**
Authorship	**91.74**	**48797**	73.39	112225	**91.17**	**44746**	80.39	71057	**88.88**	**72605**	71.79	113542
C1000	**74.7**	**414165**	72.96	443650	**76.82**	**417365**	72.23	530884	**61.96**	**594974**	59.00	588320
Python800	**83.9**	**443864**	77.88	495162	**79**	**477440**	71.42	702390	**72.69**	**625251**	69.07	640426
Java250	**79.7**	**737379**	75.12	790616	**81.94**	**721485**	72.3	827275	**75.52**	**890933**	63.8	1026367
Count	6/6	4/6	0/6	2/6	6/6	6/6	0/6	0/6	6/6	5/6	0/6	1/6

5.4 Attack Defended Model

We use RNNS and MHM to attack three (3) defended models from NaturalAttack (Yang et al., 2022)[1]. There defended models are against adversarial examples by adversarial training. The adversarial variables generated by NaturalAttack are close to the actual data because it uses the masked mechanism of pre-trained code models to generate the variable substitutes. Pre-trained code models can learn the data distribution by pre-training methods, i.e., Masked Language Modeling (MLM) and Causal Language Modeling (CLM). RNNS uses the actual variable names as the search space. This means that attacking such defended models can benefit MHM in some way. Table 6 illustrates the result. RNNS-Unlimited is one RNNS variant in that we remove all constraints. We can see that RNNS outperforms MHM in two (2) tasks (defect detection, authorship), and MHM is better in one (1) task (code clone).

[1]https://github.com/soarsmu/attack-pretrain-models-of-code

```java
public static void main(String[] args) {
   Scanner obj = new Scanner(System.in);
   int a = obj.nextInt();
   int b = obj.nextInt();
   int out = 1;
   int ans = 0;
   while ( out < b ){
      out--;
      out = out + a;
      ans++;
   }
   System.out.println(ans);
}
```

Original Code

```java
public static void main(String[] args) {
   Scanner obj = new Scanner(System.in);
   int a = obj.nextInt();
   int h = obj.nextInt();
   int out = 1;
   int ans = 0;
   while ( out < h ){
      out--;
      out = out + a;
      ans++;
   }
   System.out.println(ans);
}
```

Adversarial Code from RNNS

```java
public static void main(String[] args) {
   Scanner FastScanner = new Scanner(System.in);
   int tdigit = FastScanner.nextInt();
   int colArr = FastScanner.nextInt();
   int tempOp = 1;
   int number_array = 0;
   while (tempOp < colArr ){
      tempOp--;
      tempOp = tempOp + tdigit;
      number_array++;
   }
   System.out.println(number_array);
}
```

Adversarial Code from MHM

Figure 4: Original vs. RNNS vs. MHM

Table 6: Attack Defended Models, ASR %

Defended Model	RNNS-Smooth	RNNS-Unlimited	MHM
Clone+CodeBert	12.90	23.47	**28.17**
Defect+CodeBert	**96.36**	95.37	92.23
Authorship+CodeBert	51.88	**71.69**	39.62

6 Conclusion

We propose a novel black-box adversarial attack based on variable replacement, RNNS. Variable replacement is a safe way to transform code without changing its dynamic behavior (semantics). RNNS can generate practical adversarial examples with as little perturbation as possible. It uses failed attacks to teach the next attack and searches for adversarial alternatives to variables in the real variable names from developers based on model output behaviors. RNNS has two stages: 1) locating the vulnerable-variable position based on model uncertainty, and 2) finding adversarial alternatives based on model output-behavior change. We evaluate our approach on 18 victim models for six (6) SE tasks and three (3) widely used pre-trained code models (CodeBERT, GraphCodeBERT, and CodeT5). RNNS achieves higher ASRs and less QTs compared with SOTA MHM and NaturalAttack. RNNS requires fewer replaced variables and increases code length slightly. When RNNS attacks defended models, it can achieve acceptable performance. In summary, RNNS is a useful black-box adversarial attack on code models. We public our prototypical implementation of RNNS[2].

References

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

Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. Source code authorship attribution using long short-term memory based networks. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (eds.), *Computer Security – ESORICS 2017*, pp. 65–82, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66402-6.

Tao Bai, Jinqi Luo, Jun Zhao, Bihan Wen, and Qian Wang. Recent advances in adversarial training for adversarial robustness. *arXiv preprint arXiv:2102.01356*, 2021.

Penglong Chen, Zhen Li, Yu Wen, and Lili Liu. Generating adversarial source programs using important tokens-based structural transformations. In *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 173–182, 2022. doi: 10.1109/ICECCS54210.2022.00029.

Steffen Eger, Gözde Gül Şahin, Andreas Rücklé, Ji-Ung Lee, Claudia Schulz, Mohsen Mesgar, Krishnkant Swarnkar, Edwin Simpson, and Iryna Gurevych. Text processing like humans do:

[2]https://anonymous.4open.science/r/RNNS-for-code-attack-B888

Visually attacking and shielding NLP systems. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 1634–1647, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1165. URL https://aclanthology.org/N19-1165.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. Black-box generation of adversarial text sequences to evade deep learning classifiers. In *2018 IEEE Security and Privacy Workshops (SPW)*, pp. 50–56, 2018. doi: 10.1109/SPW.2018.00016.

Siddhant Garg and Goutham Ramakrishnan. BAE: BERT-based adversarial examples for text classification. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6174–6181, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.498. URL https://aclanthology.org/2020.emnlp-main.498.

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

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

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

Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In *International Conference on Machine Learning*, pp. 2137–2146. PMLR, 2018.

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

J Li, S Ji, T Du, B Li, and T Wang. Textbugger: Generating adversarial text against real-world applications. In *26th Annual Network and Distributed System Security Symposium*, 2019.

Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. Bert-attack: Adversarial attack against bert using bert. *arXiv preprint arXiv:2004.09984*, 2020.

Yaoxian Li, Shiyi Qi, Cuiyun Gao, Yun Peng, David Lo, Zenglin Xu, and Michael R Lyu. A closer look into transformer-based code intelligence through code transformation: Challenges and opportunities. *arXiv preprint arXiv:2207.04285*, 2022.

Yinglong Li. Research and application of deep learning in image recognition. In *2022 IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA)*, pp. 994–999. IEEE, 2022.

Daniel W Otter, Julian R Medina, and Jugal K Kalita. A survey of the usages of deep learning for natural language processing. *IEEE transactions on neural networks and learning systems*, 32(2): 604–624, 2020.

Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pp. 275–378. Elsevier, 2019.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021.

Shilin Qiu, Qihe Liu, Shijie Zhou, and Chunjiang Wu. Review of artificial intelligence adversarial attack and defense technologies. *Applied Sciences*, 9(5):909, 2019.

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

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Semantically equivalent adversarial rules for debugging NLP models. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 856–865, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1079. URL https://aclanthology.org/P18-1079.

Jacob M Springer, Bryn Marie Reinstadler, and Una-May O'Reilly. Strata: Simple, gradient-free attacks for models of code. *arXiv preprint arXiv:2009.13562*, 2020.

Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. Generating adversarial computer programs using optimized obfuscations. *arXiv preprint arXiv:2103.11882*, 2021.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 261–271. IEEE, 2020.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Ying Xu, Xu Zhong, Antonio Jimeno Yepes, and Jey Han Lau. Grey-box adversarial attack and defence for sentiment classification. *arXiv preprint arXiv:2103.11576*, 2021.

Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pp. 1482–1493, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510146. URL https://doi.org/10.1145/3510003.3510146.

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

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

Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. Towards robustness of deep program processing models—detection, estimation, and enhancement. *ACM Trans. Softw. Eng. Methodol.*, 31(3), apr 2022. ISSN 1049-331X. doi: 10.1145/3511887. URL https://doi.org/10.1145/3511887.

Wei Emma Zhang, Quan Z. Sheng, Ahoud Alhazmi, and Chenliang Li. Adversarial attacks on deep-learning models in natural language processing: A survey. *ACM Trans. Intell. Syst. Technol.*, 11(3), apr 2020b. ISSN 2157-6904. doi: 10.1145/3374217. URL https://doi.org/10.1145/3374217.

Zhengli Zhao, Dheeru Dua, and Sameer Singh. Generating natural adversarial examples. In *International Conference on Learning Representations*, 2018.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

A APPENDIX

A.1 ADVERSARIAL EXAMPLES

Figure 5, Figure 6 Figure 7 and Figure 8 show four examples that are attacked successfully by RNNS and MHM on Defect, Authorship, Pythoon800 and C1000 respectively.

```
static int scsi_device_init(SCSIDevice *s)
{
    SCSIDeviceClass *sc = SCSI_DEVICE_GET_CLASS(s);
    if (sc->init) {
        return sc->init(s);
    }
    return 0;
}

static int scsi_device_init(SCSIDevice *X)
{
    SCSIDeviceClass *sc = SCSI_DEVICE_GET_CLASS(X);
    if (sc->init) {
        return sc->init(X);
    }
    return 0;
}


static int scsi_device_init(SCSIDevice *MigrationState)
{
    SCSIDeviceClass *sc = SCSI_DEVICE_GET_CLASS(MigrationState);
    if (sc->init) {
        return sc->init(MigrationState);
    }
    return 0;
}
```

Figure 5: Original (above) vs. RNNS (middle) vs. MHM (bottom) on Defect

```
filename = "A-small-attempt0.in"
inp = open(filename, "rU")
n = int(inp.readline().strip())
for case in range(1, n + 1):
    gr = lambda x: [list(map(int, inp.readline().strip().split(" "))) for p in range(4)][x - 1]
    ans1 = int(inp.readline().strip())
    row1 = set(gr(ans1))
    ans2 = int(inp.readline().strip())
    row2 = set(gr(ans2))
    sect = row1 & row2
    if len(sect) <= 0:
        print("Case #{}: Volunteer cheated!".format(case))
    elif len(sect) == 1:
        print("Case #{}: {}".format(case, sect.pop()))
    elif len(sect) > 1:
        print("Case #{}: Bad magician!".format(case))
    else:
        print("ERROR")


inputFile = "A-small-attempt0.in"
inp = open(inputFile, "rU")
n = int(inp.readline().strip())
for case in range(1, n + 1):
    gr = lambda x: [list(map(int, inp.readline().strip().split(" "))) for p in range(4)][x - 1]
    ans1 = int(inp.readline().strip())
    row1 = set(gr(ans1))
    ans2 = int(inp.readline().strip())
    row2 = set(gr(ans2))
    sect = row1 & row2
    if len(sect) <= 0:
        print("Case #{}: Volunteer cheated!".format(case))
    elif len(sect) == 1:
        print("Case #{}: {}".format(case, sect.pop()))
    elif len(sect) > 1:
        print("Case #{}: Bad magician!".format(case))
    else:
        print("ERROR")

right_edge = "A-small-attempt0.in"
diag = open(right_edge, "rU")
MyException = int(diag.readline().strip())
for MyException in range(1, MyException + 1):
    fout = lambda x: [list(map(int, diag.readline().strip().split(" "))) for row_index2 in range(4)][x - 1]
    DrawMines = int(diag.readline().strip())
    print_matrix = set(fout(DrawMines))
    fill_num = int(diag.readline().strip())
    IMPOSSIBLE = set(fout(fill_num))
    read_integer = print_matrix & IMPOSSIBLE
    if len(read_integer) <= 0:
        print("Case #{}: Volunteer cheated!".format(MyException))
    elif len(read_integer) == 1:
        print("Case #{}: {}".format(MyException, read_integer.pop()))
    elif len(read_integer) > 1:
        print("Case #{}: Bad magician!".format(MyException))
    else:
        print("ERROR")
```

Figure 6: Original (above) vs. RNNS (middle) vs. MHM (bottom) on Authorship

```
N,M=map(int,input().split())
A=[input() for i in range(N)]
B=[input() for j in range(M)]
for i in range(N-M+1):
    for j in range(N-M+1):
        if A[i][j:j+M]==B[0]:
            if all(A[i+k][j:j+M]==B[k] for k in range(1,M)):
                print("Yes")
                exit()
print("No")
```

```
N,T_r=map(int,input().split())
A=[input() for i in range(N)]
r_T=[input() for j in range(T_r)]
for i in range(N-T_r+1):
    for j in range(N-T_r+1):
        if A[i][j:j+T_r]==r_T[0]:
            if all(A[i+k][j:j+T_r]==r_T[k] for k in range(1,T_r)):
                print("Yes")
                exit()
print("No")
```

```
dp_t,l_tuple=map(int,input().split())
nCr=[input() for al_abs in range(dp_t)]
r4=[input() for passed in range(l_tuple)]
for al_abs in range(dp_t-l_tuple+1):
    for passed in range(dp_t-l_tuple+1):
        if nCr[al_abs][passed:passed+l_tuple]==r4[0]:
            if all(nCr[al_abs+repeat][passed:passed+l_tuple]==r4[repeat] for
                print("Yes")
                exit()
print("No")
```

Figure 7: Original (above) vs. RNNS (middle) vs. MHM (bottom) on Python800

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <cmath>
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
const int mod = 1000000007;
int main() {
    ios::sync_with_stdio(false);
    int n, z = 0;
    ll d, x, y;
    cin >> n >> d;
    while (n--) {
        cin >> x >> y;
        if (x * x + y * y <= d * d) z++;
    }
    cout << z;
}
```

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <cmath>
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
const int mod = 1000000007;
int main() {
    ios::sync_with_stdio(false);
    int n, z = 0;
    ll W, x, w_h;
    cin >> n >> W;
    while (n--) {
        cin >> x >> w_h;
        if (x * x + w_h * w_h <= W * W) z++;
    }
    cout << z;
}
```

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
#include<string>
#include<cmath>
using namespace std;
typedef long long ll;
typedef pair<int,int> pii;
const int mod=1000000007;
int main() {
    ios::sync_with_stdio(false);
    int pal,pqmin=0;
    ll al, u24, YesorNo;
    cin >> pal >> al;
    while (pal--) {
        cin >> u24 >> YesorNo;
        if(u24 * u24 + YesorNo * YesorNo <= al * al) pqmin++;
    }
    cout<<pqmin;
}
```


Figure 8: Original (above) vs. RNNS (middle) vs. MHM (bottom) on C1000