Natural Attack for Pre-trained Models of Code

Zhou Yang, Jieke Shi, Junda He and David Lo
School of Computing and Information Systems
Singapore Management University
{zyang,jiekeshi,jundahe,davidlo}@smu.edu.sg

ABSTRACT

Pre-trained models of code have achieved success in many important software engineering tasks. However, these powerful models are vulnerable to adversarial attacks that slightly perturb model inputs to make a victim model produce wrong outputs. Current works mainly attack models of code with examples that preserve *operational program semantics* but ignore a fundamental requirement for adversarial example generation: perturbations should be natural to *human judges*, which we refer to as *naturalness* requirement.

In this paper, we propose *ALERT* (**Na**tura**l**n**e**ss Awa**r**e A**t**tack), a black-box attack that adversarially transforms inputs to make victim models produce wrong outputs. Different from prior works, this paper considers the *natural* semantic of generated examples at the same time as preserving the *operational* semantic of original inputs. Our user study demonstrates that human developers consistently consider that adversarial examples generated by *ALERT* are more natural than those generated by the state-of-the-art work by Zhang et al. that ignores the naturalness requirement. On attacking CodeBERT, our approach can achieve attack success rates of 53.62%, 27.79%, and 35.78% across three downstream tasks: vulnerability prediction, clone detection and code authorship attribution. On GraphCodeBERT, our approach can achieve average success rates of 76.95%, 7.96% and 61.47% on the three tasks. The above outperforms the baseline by 14.07% and 18.56% on the two pre-trained models on average. Finally, we investigated the value of the generated adversarial examples to harden victim models through an adversarial fine-tuning procedure and demonstrated the accuracy of CodeBERT and GraphCodeBERT against *ALERT*-generated adversarial examples increased by 87.59% and 92.32%, respectively.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Search-based software engineering**; • **Computing methodologies → Neural networks**.

KEYWORDS

Genetic Algorithm, Adversarial Attack, Pre-Trained Models

ACM Reference Format:
Zhou Yang, Jieke Shi, Junda He and David Lo. 2022. Natural Attack for Pre-trained Models of Code. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510146

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510146

1 INTRODUCTION

Recently, researchers [35, 51, 52] have shown that models of code like *code2vec* [3] and *code2seq* [2], can output different results for the two code snippets sharing the same operational semantics, one of which is generated by renaming some variables in the other. The modified code snippets are called *adversarial examples*, and the models under attack are called *victim models*.

Naturalness is a fundamental requirement in adversarial example generation. For example, perturbations to images are constrained with the infinity norm to ensure naturalness [19, 31]. Attack for NLP models also requires adversarial examples to be fluent and natural [24]. We propose that the naturalness requirement is also essential for attacking models of code. Casalnuove et al. [11] provide a dual-channel view of source code: machines that compile and execute code mainly focus on the operational semantics, while developers often care about natural semantics of code (e.g., names of variables) that can assist human comprehension. Although many automated tools have been included into the software development process, there is no doubt that software development is still a process led by humans. Code that violates coding convention or has poor variable names, may be acceptable for machines but rejected by humans. For example, Tao et al. [44] report that 21.7% of patches in Eclipse and Mozilla projects were rejected because the patches used bad identifier names or violated coding conventions. As a result, unnatural adversarial examples may not even pass code reviews and not to mention being merged into codebases.

Existing works on attacking models of code are effective [35, 51, 52], but they focus on preserving operational semantics and barely pay attention to whether adversarial examples are natural to human judges. For instance, the state-of-the-art black-box method, MHM [52], randomly selects replacements from a fixed set of variable names without considering semantic relationships between original variables and their substitutes. Figure 1(b) shows an adversarial example generated by replacing the variable name `buffer` in Figure 1(a) to `qmp_async_cmd_handler`. Even though the new program preserves the operational semantics, `qmp_async_cmd_handler` is not a natural replacement of `buffer` to human judges considering the context (surrounding code). The natural semantics (i.e., human understanding) of `buffer` clearly do not overlap with `qmp_async_cmd_handler`. In this paper, we argue that adversarial examples for models of code should consider preserving the semantics at two levels: operational semantics (catering for machines as audience) and natural semantics (catering for humans as audience).

The neglect of naturalness requirements in current attack methods motivates us to propose *ALERT* (**Na**tura**l**n**e**ss Awa**r**e A**t**tack),

```
static int buffer_empty(Buffer *buffer)
{
    return buffer->offset == 0;
}
```

(a) An original code snippet that can be correctly classified by a model fine-tuned on CodeBERT.

```
static int buffer_empty(Buffer *qmp_async_cmd_handler)
{
    return qmp_async_cmd_handler->offset == 0;
}
```

(b) MHM generates an adversarial example by replacing the variable `buffer` to `qmp_async_cmd_handler`.

```
static int buffer_empty(Buffer *queue)
{
    return queue->offset == 0;
}
```

(c) *ALERT* generates an adversarial example by replacing the variable `buffer` to `queue`.

Figure 1: The original example in (a) is from the dataset used in Zhou et al.'s study [55]. Both MHM and *ALERT* can generate successful adversarial examples by substituting a variable name. But HMH uses an unnatural replacement while *ALERT* uses a more natural replacement that can better fit into the context and more closely relates to the original variable name.

a black-box attack that is aware of natural semantics when generating adversarial examples of code. Similar to MHM [52], *ALERT* renames variables to generate adversarial examples. Our approach has three main parts: a natural perturbation generator, a heuristic method that tries to generate adversarial examples as fast as possible, and a genetic algorithm-based method to search for adversarial examples more comprehensively in case the heuristic method fails.

This paper investigates the victim models that are fine-tuned on the state-of-the-art pre-trained models, CodeBERT [14] and GraphCodeBERT [20]. *ALERT* first uses the *masked language prediction* function in pre-trained models to generate natural substitutes. Given a code snippet with several masked tokens, this function can utilize the context information to predict the potential values of masked tokens. We leverage such a function in CodeBERT and GraphCodeBERT to generate candidate substitutes for each variable. Then, to pick the substitutes that are semantically closer, we use pre-trained models to compute contextualized embeddings of these new tokens and calculate its Cosine similarity (for measuring the semantic distances) [32] with embeddings of the original tokens. We rank these candidates according to Cosine similarities and only select the top-k candidates as natural substitution candidates.

ALERT has two steps to search adversarial examples using natural substitution candidates. It first uses a greedy algorithm (Greedy-Attack) and then applies a genetic algorithm (GA-Attack) if the former fails. The Greedy-Attack defines a metric to measure the *importance* of variable names in a code snippet and starts to substitute variables with the highest importance. An algorithm guided by the importance can find successful adversarial examples faster than the random sample strategy used in MHM [52]. When substituting a variable, Greedy-Attack greedily selects the replacement (out of all *natural substitutes*), from which the generated adversarial example makes the victim model produce lower confidence on the ground truth label. If it fails to change the prediction results, Greedy-Attack continues to replace the next variable until all the variables are considered or an adversarial example is obtained. But generating adversarial examples for code is essentially a combinatorial problem, and the greedy algorithm may generate sub-optimal results. If the greedy algorithm fails, we use the GA-Attack to perform a more comprehensive search.

We first conduct a user study to examine whether searching from substitutes generated by *ALERT* can produce adversarial examples that are natural to human judges. Participants give a *naturalness score* (1 for very unnatural and 5 for very natural) to each adversarial example. Results show that participants consistently provide a higher score to *ALERT*-generated examples (on average 3.95) than examples generated by the original MHM [52] (on average 2.18) that selects substitutes randomly over all variables.

Then, we evaluate MHM and *ALERT* on the six victim models (2 pre-trained models × 3 tasks). We consider three relevant tasks that may be adversely affected by such an attack: vulnerability prediction, clone detection and code authorship attribution.[1] Since we argue that adversarial examples should look natural to human developers, we make MHM search on the same set of natural substitutes generated by *ALERT*. On CodeBERT, *ALERT* can achieve attack success rates of 53.62%, 27.79%, and 35.78% across three downstream tasks. MHM only reaches 35.66%, 20.05% and 19.27%, respectively, which means that *ALERT* can improve attack success rates over MHM by 17.96%, 7.74% and 16.51%. On GraphCodeBERT, our approach achieves success rates of 76.95%, 7.96% and 61.47% on the same three tasks, outperforming MHM by 21.78%, 4.54%, and 29.36%. Finally, we investigate the value of generating adversarial examples by using them to harden the models through an adversarial fine-tuning strategy. We demonstrate that the robustness of CodeBERT and GraphCodeBERT increased by 87.59% and 92.32% after adversarial fine-tuning with examples generated by *ALERT*. The contributions of this paper include:

- We are the first to highlight the naturalness requirement in generating adversarial examples for models of code. We also propose *ALERT* that is aware of natural semantics when generating adversarial variable substitutes. A user study confirms that using these substitutes can generate adversarial examples that look natural to human judges. *ALERT* can also achieve higher attack success rates than a previous method.
- We are the first to develop adversarial attacks on CodeBERT and GraphCodeBERT, and show that models fine-tuned on state-of-the-art pre-trained models are vulnerable to such attacks.
- We show the value of *ALERT*-generated examples: adversarially fine-tuning victim models with these adversarial examples can improve the robustness of CodeBERT and GraphCodeBERT against *ALERT* by 87.59% and 92.32%, respectively.

The rest of this paper is organized as follows. Section 2 briefly describes preliminary materials. In Section 3, we elaborate on the design of the proposed approach *ALERT*. We describe the settings of the experiment in Section 4, and present the results of our experiments that compare the performance of *ALERT* and some baselines in Section 5. After summarising the threats to validity in Section 6, Section 7 discusses some related works. Finally, we conclude the paper and present future work in Section 8.

[1]For example, malicious users may write vulnerable code snippets and do not want them to be identified.

2 PRELIMINARIES

This section briefly introduces some preliminary information of this study, including pre-trained models of code, adversarial example generation for DNN models, and the Metropolis-Hastings Modifier (MHM) method that we use as the baseline.

2.1 Pre-trained Models of Code

Pre-trained models of natural language like BERT [12] have brought breakthrough changes to many natural language processing (NLP) tasks, e.g., sentiment analysis [53]. Recently, researchers have created pre-trained models of code [14, 20] that can boost the performance on programming language processing tasks.

Feng et al. propose CodeBERT [14] that shares the same model architecture as RoBERTa [29]. CodeBERT is trained on a bimodal dataset (CodeSearchNet [23]), a corpus consisting of natural language queries and programming language outputs. CodeBERT has two training objectives. One objective is *masked language modeling* (MLM), which aims to predict the original tokens that are masked out in an input. The other objective is *replaced token detection* (RTD), in which the model needs to detect which tokens in a given input are replaced. Experiment results have shown that in downstream tasks like code classification or code search, which requires understanding the code, CodeBERT could yield superior performance, although it is less effective in code-generation tasks. GraphCodeBERT [20] also uses the same architecture as CodeBERT, but the former additionally considers the inherent structure of code, i.e., data flow graph (DFG). GraphCodeBERT keeps the MLM training objective and discards the RTD objective. It designs two DFG-related tasks: data flow edge prediction and node alignment. GraphCodeBERT outperforms CodeBERT on four downstream tasks.

There are some other pre-trained models of code. CuBERT [26] is trained on Python source code and C-BERT [9] is a model trained on the top-100 starred GitHub C language repositories. CodeGPT [30] is a Transformer-based language model pre-trained on programming languages for code generation tasks. In this paper, we focus on analyzing CodeBERT and GraphCodeBERT, as they can work on multiple programming languages. Besides, recent studies [30, 49, 54] have empirically shown that CodeBERT and GraphCodeBERT demonstrate state-of-the-art performance across multiple code processing tasks.

2.2 Adversarial Example Generation

Although Deep Neural Network (DNN) models have achieved great success on many tasks, many research works [17, 50] have shown that state-of-the-art models are vulnerable to adversarial attacks. Adversarial attacks aim to fool DNN models by slightly perturbing the original inputs to generate adversarial examples that are natural to human judges. Many techniques have been proposed to show that adversarial examples can be found for models in different domain, including, image classification [19, 31], reinforcement learning [18, 21], sentiment analysis [6], speech recognition [10], machine translation [13], etc.

According to the information of victim models that an attacker can access, adversarial attacks can be divided into two types: *white-box* and *black-box*. In white-box settings, attackers can access all the information of the victim models, e.g., using model parameters to compute gradients. But white-box attacks often lack practicality since the victim models are usually deployed remotely (e.g., on cloud services), and typically attackers can only access the APIs to query models as well as corresponding outputs. Black-box attacks mean that an attacker only knows the inputs and outputs of victim models (e.g., predicted labels and corresponding confidence). This paper proposes a novel black-box attack to mislead models that have the state-of-the-art performance.

Adversarial attacks can also be categorized into *non-targeted* attack and *targeted* attack. Non-targeted attacks only aim to make a victim model produces wrong predictions, while targeted attacks force a victim model to make specific predictions. For example, a targeted attack may require a classifier to predict all the deer images as a horse while a non-target requires a classifier to predict an image incorrectly. The attack proposed in this paper is non-targeted.

2.3 Metropolis-Hastings Modifier (MHM)

Considering the fact that models can be remotely deployed so that model parameters are inaccessible, we focus on black-box attacks for models of code. This section introduces the baseline used in this paper. Zhang et al. [52] formalizes the process of adversarial example generation as a sampling problem. The problem can be decomposed into an iterative process consisting of three stages: (1) selecting the variable to be renamed (2) selecting the substitutions and (3) deciding whether to accept to replace the variable with selected substitution.

Zhang et al. proposed Metropolis-Hastings Modifier (MHM) [52], a Metropolis-Hastings sampling-based [33] identifier renaming technique to solve this problem and generate adversarial examples for models of code. This method is a black-box attack that randomly selects replacements for local variables and then strategically determines to accept or reject replacements. It uses both predicted labels and corresponding confidence of the victim model to select adversarial examples more effectively. MHM pre-defines a large collection of variable names, from which the replacements are selected. However, neither the creation of this collection nor selecting replacements considers the natural semantics. As a result, MHM produces examples that are not natural to human judgments. For example, suppose we change a variable name to an extremely long string that is not semantically close to the original variable. In that case, it may change the result of CodeBERT and GraphCodeBERT since the long name will be tokenized into multiple sub-tokens, impacting the output significantly. However, developers certainly will not accept this code.

In this paper, similar to MHM, we use variable renaming as the adversarial example generation technique and explore how to produce adversarial examples that are natural. We choose MHM [52] as our baseline as it does not require gradient information and also uses fine-grained model outputs (i.e., predicted results and corresponding confidence) to perform renaming and achieves good attack success rates (of one degree of magnitude higher than other black-box approaches [35]). For example, Pour et al.'s approach [35] only causes an absolute decrease of 2.05% to *code2vec*'s performance on the method name prediction task. To ensure that renamed variables make no changes in operational semantics, similar to MHM, we only rename local variables in code snippets.

3 METHODOLOGY

This paper proposes *ALERT* (**Na**tura**l**n**e**ss Awa**r**e A**t**tack), a black-box attack that leverages the pre-trained models that victim models are fine-tuned on. It generates substitutes that are aware of natural semantics, which are called naturalness-aware substitutions in this paper. *ALERT* takes two steps to search for adversarial examples that are likely to be natural to human judges. The first step (Greedy-Attack) is optimized to find adversarial examples fast, and the second step (GA-Attack) is applied to do a more comprehensive search if the former fails.

3.1 Naturalness-Aware Substitution

ALERT leverages the two functions of pre-trained models to generate and select naturalness-aware substitutes for variables: masked language prediction and contextualized embedding. To generate natural substitutes for one single variable (e.g., `index2dict`), it operates in three steps:

Step 1. We convert code snippets into a format that CodeBERT or GraphCodeBERT can take as inputs. Source code often contains many domain-specific abbreviations, jargon and their combinations, which are usually not included in the vocabulary set and cause the out-of-vocabulary problem [27, 40]. Both CodeBERT and GraphCodeBERT use Byte-Pair-Encoding (BPE) [16, 39] to deal with such out-of-vocabulary problems by tokenizing a word into a list of sub-tokens. For example, a variable `index2dict` can be converted into three sub-words (`index`, `2`, `dict`) and then fed into the model.

Step 2. Then, we generate potential substitutes for each sub-token. For the sake of simplicity but without any loss of generality, let us imagine a case where there is only one variable (e.g., `index2dict`) that only appears once in an input. We use $T = \langle t_1, t_2, \cdots, t_m \rangle$ to represent the sequence of sub-tokens that BPE produces from the variable name. For each sub-token in the sequence, we use the masked language prediction function of CodeBERT or GraphCodeBERT to produce a ranked list of potential substitute sub-tokens. Instead of just picking a single output, we select the top-j substitutes. Intuitively, these substitutes are what pre-trained models think can fit the context better (compared to other sub-tokens). Still, not all of them are semantically similar to the original sub-tokens.

Step 3. We assume that $\langle t_i, t_{i+1}, t_{i+2} \rangle$ is a sequence of sub-tokens of one variable name (e.g., corresponding to `index`, `2` and `dict`). We replace the sub-tokens in the original sequence T with candidate sub-tokens (e.g., t'_i, t'_{i+1}, t'_{i+2}) generated in Step 2 to get T'. After that, the pre-trained model computes the contextualized embeddings of each sub-token in T', and we fetch the embeddings for t'_i, t'_{i+1} and t'_{i+2}. We concatenate these new embeddings and compute its Cosine similarity with concatenated embeddings of t_i, t_{i+1} and t_{i+2} in T. The cosine similarity is used as a metric to measure to what extent a sequence of candidate sub-tokens is similar to the original variable's sequence of sub-tokens. We rank the substitutes in descending order by the value of Cosine similarity. In the end, we select top-k sequences of substitute sub-tokens with higher similarity values and revert them into concrete variable names.

One code snippet often contains multiple variables that appear in various positions. Algorithm 1 displays how we apply the above process to each variable extracted from the source code. First, we use a parser to extract variable names (`vars`) from the input (`extract()` at Line 2) and then enumerate all the variables and their occurrences in the code (Line 3-4). The process discussed above is then applied to each variable occurrence to generate potential substitutes (`perturb()` at Line 6). We take the union of the substitutes sets for all occurrences of a variable (Line 7). We then remove duplicated and invalid words, e.g., those that do not comply with the variable naming rules or those that are keywords in programming languages (`filter()` at Line 9), after which we return filtered substitutes (Line 11). We refer to these filtered substitutes as the *naturalness-aware* substitutes.

Algorithm 1: Naturalness Aware Substitutes Generation

```
   Input: c: input source code, M: pre-trained model
   Output: subs: substitutes for variables
1  subs = ∅;
2  vars = extract(c);
3  for var in vars do
4      for occ in var.occurrences do
5          # var.occurrences returns all occurrences of var ;
6          tmp_subs = perturb(occ, c, M);
7          subs[var] = subs[var] ⋃ tmp_subs;
8      end
9      subs[var] = filter(subs[var]);
10 end
11 return subs
```

3.2 Greedy-Attack

3.2.1 Overall Importance Score. To perform semantic-preserving transformation by renaming variables, an attacker first needs to decide which tokens in a code snippet should be changed. Inspired by adversarial replacements for NLP tasks [28] that prioritizes more important tokens in a sentence, for each variable in a code snippet, we first measure its contribution to helping the model make a correct prediction. We introduce a metric called the importance score to quantify such contribution. Formally speaking, the importance score of the i^{th} token in a code snippet c is defined as follow:

$$IS_i = M(c)[y] - M(c^*_{-i})[y] \tag{1}$$

In the above formula, y is the ground truth label for c and $M(c)[y]$ represents the confidence of M's output corresponding to the label y. A new code snippet generated by substituting variable names is called a *variant*. A variant c^*_{-i} and is created by replacing the i^{th} token (which must be a variable name) in c with $\langle unk \rangle$, which means that the literal value at this position is unknown. Intuitively, the importance score approximates how knowing the value of the i^{th} token affects the model's prediction on c. If $IS_i > 0$, it means that the token t_i can help model make correct prediction on c. As stated in Section 3.1, one code snippet often contains multiple variables that appear in multiple positions. All the occurrences of a variable should be updated accordingly when performing adversarial attacks, so we extend the definition of importance score for a single token to the overall importance score (OIS) for a variable.

OIS is computed as follow:

$$OIS_{var} = \sum_{i \in var[pos]} IS_i \quad (2)$$

where var is a variable in c, and $var[pos]$ means all occurrences of var in c. It is noticed that the definition of OIS can better reflect the unique property of attacking models of programming languages as compared to models of natural languages. Even though a variable at one position is trivial, appearing more often can make it an important variable (i.e., a vulnerable word) in adversarial attacks. The overall importance score can be viewed as an analogy to the gradient information in white-box attacks. For example, if the gradients are larger at some positions of inputs (e.g., certain pixels), then it is easier to change the model outputs if we perturb those positions.

Based on tree-sitter[2], a multi-language parser generator tool, we implement a name extractor that can retrieve all the variable names from syntactically valid code snippets written in C, Python or Java. More specifically, to avoid altering the operational semantics, we only extract the local variables that are defined and initialized within the scope of the code snippet and swap them with valid variable names that have never occurred in the code. To improve accuracy, variable names that collide with a field name are also excluded. After extraction, we compute the OIS for each variable and proceed to the next step.

3.2.2 Word Replacement. We design an OIS-based greedy algorithm to search substitutes that can generate adversarial examples. Algorithm 2 illustrates the process of this Greedy-Attack. First, we rank extracted variables from the original code snippet in descending order according to their OIS (Line 2 to 3). We select the first variable from them and find all its candidate substitutes generated following the process described in Section 3.1 (Line 4 to 6). We replace the variable in the original input with these substitutes to create a list of variants, after which these variants are sent to query the victim model. We collect returned results and see if at least one variant makes the victim model make wrong predictions (Line 9 to 12). If there is such a variant, the Greedy-Attack returns it as a successful adversarial example. Otherwise, we replace the original input with the variant that can mostly reduce the victim model's confidence on the results and select the next variable to repeat the above processes (Line 15). Greedy-Attack terminates either when a successful adversarial example is found (Line 11) or when all the extracted variables are enumerated (Line 17).

Considering OIS information is beneficial to the Greedy-Attack in two aspects. First, as discussed in Section 3.2.1, if a variable has a higher OIS, it indicates significant impacts of modifying this variable in the code snippet. Giving higher priorities to variables with larger OIS can help find successful adversarial examples faster, which means that fewer queries to the victim model are required. It increases the usability of our attack in practice since remotely deployed black-box models often constrain the query frequency. Secondly, finding successful adversarial examples early also means fewer variables are modified in an original code snippet, making the generated adversarial examples more natural to human judges.

[2]https://tree-sitter.github.io/tree-sitter/

Algorithm 2: Greedy-Attack Workflow

Input: c: input source code, $subs$: substitutes for variables in c
Output: c': adversarial example

```
1  c' = c;
2  vars = extract(c) # extract vars from c;
3  vars = sort(vars) # sort vars according to OIS;
4  for var in vars do
5      list_c = ∅;
6      for sub in subs[var] do
7          # iterate all the substitutes for var;
8          tmp_c = replcae(c', var, sub);
9          if M(tmp_c) ≠ M(c) then
10             c' = tmp_c;
11             return c';
12         end
13         list_c = list_c ⋃ tmp_c;
14     end
15     c' = select(list_c) # select the adversarial example with lowest
         model's confidence on the ground truth label;
16 end
17 return c'
```

3.3 GA-Attack

Finding appropriate substitutes to generate adversarial examples is essentially a combinatorial optimization problem, whose objective is to find the optimal combination of variables and corresponding substitutes that minimizes the victim model's confidence on the ground truth label. Greedy-Attack can run faster but may be stuck in a single local optimal, leading to low attack success rates. We also design an attack based on genetic algorithms (GA), called GA-Attack. If the Greedy-Attack fails to find a successful adversarial example, we apply GA-Attack to search more comprehensively. Algorithm 3 shows the overview of how GA-Attack works. It first initializes the population (Line 1, more detailed are given in Section 3.3.2), and then performs genetic operators to generate new solutions (Line 2 to 11). GA-Attack computes the fitness function (Section 3.3.4) and keep solutions with larger fitness values (Line 13). In the end, the algorithm returns the solution with the highest fitness value (Line 15 to 16).

3.3.1 Chromosome Representation. In GA, the chromosome represents the solution to a target problem, and a chromosome consists of a set of genes. In this paper, each gene is a pair of an original variable and its substitution. GA-Attack represents chromosomes as a list of such pairs. For example, assuming that only two variables (a and b) can be replaced in an input program, the chromosome $\langle \mathtt{a} : \mathtt{x}, \mathtt{b} : \mathtt{y} \rangle$ means replacing a to x and b to y.

3.3.2 Population Initialization. In the running of GA, a population (a set of chromosomes) evolves to solve the target problem. GA-Attack maintains a population whose size is the number of extracted variables that can be substituted. Since GA-Attack will be triggered only after Greedy-Attack fails, it can leverage the information discovered in the previous step. For each extracted variable, Greedy-Attack finds its substitution that can decrease the victim model's confidence on the ground truth label most. Given one variable and

Algorithm 3: GA-Attack Workflow

Input: c: input source code, max_iter: max iteration, r: crossover rate, $child\_size$: number of generated children in each iteration
Output: c': adversarial example

```
1  population = greedy_initialization(c);
2  while not exceed max_iter do
3      child_list = [] ;
4      while len(child_lis) < child_size do
5          p = ∪(0, 1) ;
6          if p < r then
7              child = crossover(population) ;
8          else
9              child = mutation(population);
10         end
11         child_list.append(child);
12     end
13     population = selection(population ∪ child_list);
14 end
15 c' = argmax(population); # select the one with highest fitness value
16 return c'
```

the substitution found by Greedy-Attack, GA-Attack creates a chromosome that only changes this variable to the substitution and keeps other variables unchanged. The process is repeated for each variable in a code snippet to obtain a population. For example, assuming that three variables (a, b and c) are extracted from an input program, and Greedy-Attack suggests $\langle a : x, b : y, c : z\rangle$, GA-Attack initializes a population of three chromosomes: $\langle a : x, b : b, c : c\rangle$, $\langle a : a, b : y, c : c\rangle$. and $\langle a : a, b : b, c : z\rangle$.

3.3.3 Operators. Greedy-Attack runs in multiple iterations. In each iteration, two genetic operators (mutation and crossover) are used to produce new chromosomes (i.e., children). We apply crossover with a probability of r and mutation with a probability of $1 - r$ (Line 8). The mutation operation (Line 9) on two chromosome (c_1 and c_2) works as follows: we first randomly select a cut-off position h, and replace c_1's genes after the position h with c_2's genes at the corresponding positions. As an example, for two chromosomes ($c_1 = \langle a : x, b : y, c : c\rangle$ and $c_2 = \langle a : x, b : b, c : z\rangle$) and a cut-off position $h = 2$, the child generated by crossover is $\langle a : x, b : y, c : z\rangle$. Given a chromosome in the population, the mutation operator randomly selects a gene and then replaces it with a randomly selected substitute. For instance, a in $\langle a : x, b : b\rangle$ is selected and a : x becomes a : aa.

3.3.4 Fitness Function. GA uses a fitness function to measure and compare the quality of chromosomes in a population. A higher fitness value indicates that the chromosome (variable substitutions) is closer to the target of this problem. We compute the victim model's confidence values with respect to the ground truth label on the original input and the variant. The difference between confidence values is used as the fitness value. Assuming T is the original input and T' is a variant corresponding to a chromosome, the fitness value of this chromosome is computed by:

$$fitness = M(T)[y] - M(T')[y] \quad (3)$$

After generating children in one iteration, we merge them to the current population and perform a selection operator (Line 14). GA-Attack always maintains a population of the same size (i.e., numbers of extract variables). It discards the chromosomes that have lower fitness values.

4 EXPERIMENT SETUP

4.1 Datasets and Tasks

We introduce the three downstream tasks and their corresponding datasets used in our experiments. The statistics of datasets are presented in Table 1.

4.1.1 Vulnerability Prediction. This task aims to predict whether a given code snippet contains vulnerabilities. We use the dataset that was prepared by Zhou et al. [55]. The dataset is extracted from two popular open-sourced C projects: FFmpeg[3] and Qemu[4]. In Zhou et al.'s dataset, 27,318 functions are labeled as either containing vulnerabilities or clean. This dataset is included as part of the CodeXGLUE benchmark [30] that has been used to investigate the effectiveness of CodeBERT for vulnerability prediction. CodeXGLUE divides the dataset into training, development and test set that we reuse in this study.

4.1.2 Clone Detection. The clone detection task aims to check whether two given code snippets are clones, i.e., equivalent in operational semantics. BigCloneBench [42] is a broadly recognized benchmark for clone detection, containing more than six million actual clone pairs and 260,000 false clone pairs from various Java projects. Each data point is a Java method. In total, the dataset has covered ten frequently-used functionalities. Following the settings of prior works [46, 48], we filtered the data which do not have a label and then balanced the dataset to make the ratio of true and false pairs to 1:1. To keep the experiment at a computationally friendly scale, we randomly select 90,102 examples for training and 4,000 for validation and testing.

4.1.3 Authorship Attribution. The authorship attribution task is to identify the author of a given code snippet. We did our experiments with the Google Code Jam (GCJ) dataset, which is originated from *Google Code Jam* challenge, a global coding competition that Google annually hosts. Alsulami et al. [4] collected the GCJ dataset and made it publicly available. The GCJ dataset contains 700 Python files (70 authors and ten files for each author), but we notice that some Python files are C++ code. After discarding these C++ source code files, we get 660 Python files in total. 20% of files are used for testing, and 80% of files are for training.

4.2 Target Models

This paper investigates the robustness of the state-of-the-art pre-trained models, CodeBERT [14] and GraphCodeBert [20]. To obtain the victim models, we fine-tune CodeBERT and GraphCodeBERT on the three tasks mentioned in Section 4.1.

[3]https://www.ffmpeg.org/
[4]https://sites.google.com/view/devign
[4]https://codingcompetitions.withgoogle.com/codejam

Table 1: Statistics of Datasets and of Victim Models.

Tasks	Train/Dev/Test	Model	Acc
Vulnerability Prediction [55]	21,854/2,732/2,732	CodeBERT	63.76%
		GraphCodeBERT	63.65%
Clone Detection [42]	90,102/4,000/4,000	CodeBERT	96.97%
		GraphCodeBERT	97.36%
Authorship Attribution [4]	528/–/132	CodeBERT	90.35%
		GraphCodeBERT	89.48%

4.2.1 CodeBERT. CodeBERT [14] is a pre-trained model that is capable of learning from bimodal data in the form of both programming languages and natural languages. When fine-tuning CodeBERT on vulnerability prediction and clone detection task, we use the same parameter settings adopted in the CodeXGLUE [30] except that we increase the maximal input length to 512 and achieve a slightly higher performance than results reported in the CodeXGLUE paper. Since there is no instruction on the hyper-parameter setting for fine-tuning on authorship attribution task, we use the same settings, and the obtained model can achieve 90.35% accuracy, slightly higher than the accuracy of the LSTM model reported in [4].

4.2.2 GraphCodeBERT. GraphCodeBERT [20] considers the inherent structure of the program and takes advantage of the data-flow representation. We set the maximal input length of GraphCodeBERT to 512 and follow the same setting for other hyper-parameters in the GraphCodeBERT paper [20] to fine-tune it on the three downstream tasks. On the clone detection task, the model can achieve an accuracy of 97.36%, almost the same with the performance of 97.3% reported in [20]. On the vulnerability prediction and authorship attribution task, GraphCodeBERT also achieves the performance that is comparable with the results of CodeBERT.

The performance of these models is displayed in Table 1. The results we obtain are closed to results reported in their original papers and another recent paper [14, 20, 30], highlighting that the victim models used in our experiments are adequately fine-tuned.

4.3 Settings of Attacks

ALERT has a number of hyper-parameters to be set, including the number of natural substitutions generated for each variable and parameters for GA-Attack in Algorithm 3. Our experiment setting allows *ALERT* to generate 60 candidate substitutions for each variable occurrence, and it selects the top 30 substitutions ranked by the cosine similarity with original embedding. For GA-Attack, we set *child_size* as 64 and set a dynamic value for the maximal iterations (*max_iter*): the larger one of 5 times the number of extracted variables or 10. The crossover rate r is set as 0.7.

We consider MHM [52] as our baseline, which has two hyper-parameters: the maximum number of iterations and the number of variables sampled in each iteration. The MHM paper [52] suggests setting the latter as 30 but does not provide a standard setting for the maximal iterations. In each iteration, MHM needs to query the victim model many times, which is time-consuming. We sampled 5% testing data from the vulnerability prediction task and found that over 95% successful adversarial examples are found before 100 iterations. To make the MHM experiment within a computational friendly scale, we set the maximum number of iterations of MHM to 100. The original MHM can only perturb C programs, so we extend it to perturb Python and Java code.

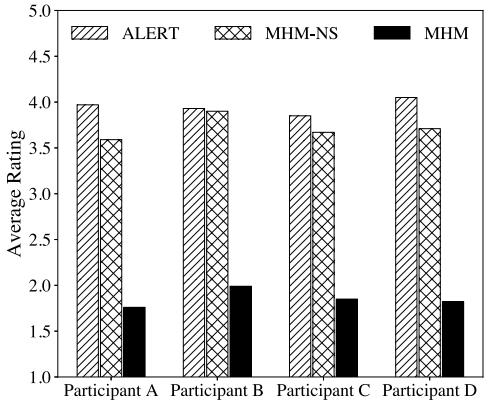


Figure 2: Results of the user study to evaluate naturalness of adversarial examples. The y-axis corresponds to the average ratings (5 means very natural; 1 means very unnatural). The x-axis represents distinguished independent participants.

5 EXPERIMENT RESULTS AND ANALYSIS

In this section, we perform experiments to answer research questions related to the performance of adversarial attacks. We care about naturalness, attack success rates and scalability as well as the value of using adversarial examples to improve model robustness via adversarial fine-tuning, which are discussed by answering three research questions, respectively.

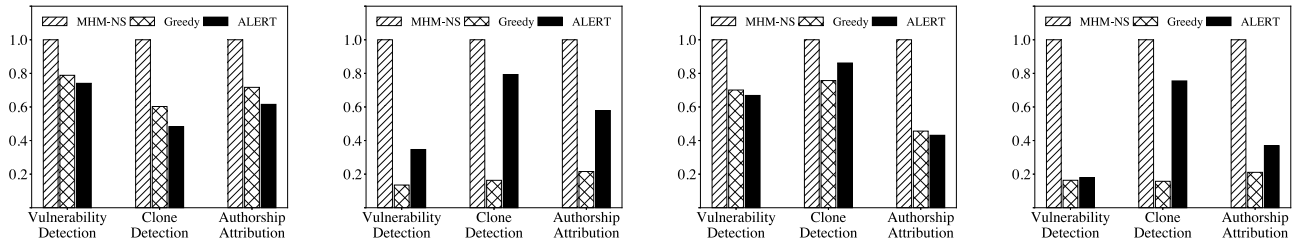
RQ1. How natural are the adversarial examples generated by *ALERT*?

When generating substitutions for variables in code, *ALERT* takes the natural semantics of adversarial examples into consideration. This research question explores whether these naturalness-aware substitutions can help produce adversarial examples that are more natural to human judges. To answer this question, we conduct a user study to analyze the naturalness of examples generated by MHM, MHM-NS and the proposed *ALERT* method. Unlike the original MHM that ignores the naturalness, MHM-NS selects a replacement from the same pool of naturalness-aware substitutions as *ALERT*.

As the original MHM only works for code snippets written in C, we randomly sample some code snippets that can be successfully attacked by *ALERT*, MHM, and MHM-NS from the dataset of the vulnerability detection task (Section 4.1). We have introduced a few more constraints when sampling the code snippets: (1) To save participants from reading long code snippets, we intentionally sample succinct and short code segments by limiting the code snippet length to 200 tokens; and (2) The attack methods may choose to replace different variables in the same code snippet; we only select the examples for which at least one variable is modified by all the three methods to make the comparison fair. There are 196 C code

Table 2: Comparison results of Attack Success Rates (ASR) on attacking CodeBERT and GraphCodeBERT across three tasks. The numbers in the parentheses correspond to the absolute improvement with respect to the attack success rates of MHM-NS.

Task	CodeBERT			GraphCodeBERT		
	MHM-NS	Greedy-Attack	*ALERT*	MHM-NS	Greedy-Attack	*ALERT*
Vulnerability Detection	35.66%	49.42% (+13.76%)	**53.62% (+17.96%)**	55.17%	71.98% (+16.81%)	**76.95% (+21.78%)**
Clone Detection	20.05%	23.20% (+3.15%)	**27.79% (+7.74%)**	3.42%	6.75% (+3.33%)	**7.96% (+4.54%)**
Authorship Attribution	19.27%	30.28% (+11.01%)	**35.78% (+16.51%)**	32.11%	46.79% (+14.68%)	**61.47% (+29.36%)**
Average	24.99%	34.30% (+9.31%)	**39.06% (+14.07%)**	30.23%	42.17% (+11.94%)	**48.79% (+18.56%)**



(a) VCR on attacking CodeBERT (b) NoQ on attacking CodeBERT (c) VCR on attacking GraphCodeBERT (d) NoQ on attacking GraphCodeBERT

Figure 3: Comparison results of Variable Change Rate (VCR) and Number Of Queries (NoQ) on attacking CodeBERT and GraphCodeBERT. The y-axis corresponds to the normalized values of VCR and NoQ. The x-axis represents downstream tasks.

snippets that satisfy the aforementioned constraints. We compute a statistically representative sample size using a popular sample size calculator[5] with a confidence level of 99% and a confidence interval of 10. We sample 100 code snippets to conduct the user study, which is statistically representative.

For each selected code snippet, we can construct 3 pairs. Each pair contains the original code snippet and an adversarial example generated by either *ALERT*, MHM, or MHM-NS. We highlight the changed variables in each pair and present them to users. Users are asked to evaluate to what extent the substitutions are naturally fitting into the source code contexts. Given the statement: "The new variable name looks natural and preserves the original meaning", participants need to give scores on a 5-point Likert scale [25], where 1 means strongly disagree and 5 means strongly agree, following the same settings used by Jin et al. [24]. Participants do not know which attack method produces which adversarial example in a pair.

The user study involves four non-author participants who have a Bachelor/Master degree in Computer Science with at least four years of experience in programming. Each participant evaluates the 100 pairs individually. We calculate the average ratings given to adversarial examples generated by each attack method per participant, and present the results in Figure 2. The x-axis distinguishes each participant, and the y-axis shows the average ratings. The results show that the usage of *ALERT*-generated substitutions can help generate much more natural adversarial examples. The four participants give average scores of close to 4 to adversarial examples generated by *ALERT* and slightly lower average scores to adversarial examples generated by MHM-NS; these indicate that participants perceive that the substitutions generated by these two methods are natural. Participants consistently give lower scores (1.86 on average) to examples generated by MHM, showing that they think the variable substitutions are unnatural.

> **Answers to RQ1**: Participants consistently find that adversarial examples generated by ALERT (a naturalness-aware method) are natural while those generated by MHM (a naturalness-agnostic method) are unnatural.

RQ2. How *successful* and *minimal* are the generated adversarial examples? How *scalable* is the generation process?

To answer this question, we evaluate the effectiveness of *ALERT* and MHM-NS on attacking CodeBERT and GraphCodeBERT considering three dimensions. Specifically, we use three metrics, each capturing one quality dimension, to measure the performance of an adversarial example generation method. Each metric is defined based on a dataset X, where each element $x \in X$ is a code snippet that has at least one local variable and a victim model M that can predict all examples in X correctly. The three metrics are defined as follows.

- **Attack Success Rate** (ASR): The ASR of an adversarial example generation method is defined as $\frac{|\{x|x\in X \wedge M(x') \neq M(x)\}|}{|X|}$, where x' is a generated example. A higher ASR indicates that an attack method has better performance.
- **Variable Change Rate** (VCR): Assuming that an input code snippet x_i has m_i local variables, and an attacker renames n_i variables in x_i, we define variable change rate (VCR) of the attack over X as $\frac{\sum_i n_i}{\sum_i m_i}$. A lower VCR is preferable since it means that fewer edits are made to find successful adversarial examples.

[5]https://www.surveysystem.com/sscalc.htm. Accessed: 2021-08-19

- **Number of Queries** (NoQ): In adversarial attacks, especially the black-box ones, the number of queries to the victim model needs to be kept as low as possible. In practice, victims models are usually remotely deployed, and it is expensive (and maybe also suspicious) to query models too many times. We count the number of queries (NoQ) to the victim models when each attack generates adversarial examples on dataset X. Attacks that have lower NoQ are more scalable as well.

Table 2 displays the comparison results between MHM-NS and *ALERT* on the six victim models (2 models × 3 tasks as described in Section 4.1). We also report the results of solely using Greedy-Attack to emphasize the improvements brought by GA-Attack. Results show that Greedy-Attack has 49.42%, 23.20% and 30.28% attack success rate on CodeBERT across three downstream tasks, which corresponds to an improvement of 13.76%, 15.71% and 11.01% over MHM-NS, respectively. By employing GA-Attack in *ALERT*, we can boost the performance even further: MHM-NS results are improved by 17.96%, 7.74% and 16.51% in terms of ASR. On GraphCodeBERT, Greedy-Attack can outperform MHM-NS by 16.81%, 3.33% and 14.68% for the three tasks; the numbers are boosted to 21.78%, 4.54% and 29.36% when GA-Attack is employed.

Moreover, *ALERT* makes fewer edits to the original examples and is more scalable than the baseline. Figure 3 compares results in terms of VCR and NoQ. The x-axis corresponds to each downstream task, and the y-axis represents normalized values of the two evaluation metrics. On all victim models, *ALERT* modifies fewer variables to generate adversarial examples. It indicates that *ALERT* can make minimal changes to input code snippets and produce more natural and imperceptible adversarial examples. Besides, *ALERT* queries victim models less than MHM-NS does. The NoQ of solely using Greedy-Attack is 82.57% less than MHM-NS. When GA-Attack is employed, the NoQ increases but is still 49.62% less than MHM-NS, which shows that *ALERT* is more practical since victim models are usually remotely deployed and may be costly to query and may prevent frequent queries. Querying victim models is the most time-consuming part of experiments, so fewer NoQ also shows that *ALERT* has lower runtime.

> **Answers to RQ2**: In terms of the attack success rate, *ALERT* can outperform the MHM by 17.96%, 7.74% and 16.51% on CodeBERT, as well as 21.78%, 4.54% and 29.36% on GraphCodeBERT across three downstream tasks. In addition to achieving a superior attack success rate, our method also makes fewer changes and is more scalable.

RQ3. Can we use adversarial examples to harden the victim models?

In this research question, we explore the effectiveness of using adversarial fine-tuning [22] as a defense against attacks. We leverage *ALERT* to generate adversarial examples for each victim model on their corresponding training sets. If a victim model predicts wrongly on an original input or no local variable name can be extracted from it, we skip this example. For other inputs in the training sets, we select at most one adversarial example for each of them. If *ALERT* attacks successfully, we choose the firstly-found adversarial example. If *ALERT* fails to attack, we select the example that can minimize the victim model's confidence on the ground truth label. These generated adversarial examples are then augmented into the original training set and form the *adversarial training set*. We then fine-tune the victim model on the adversarial training set.

After adversarial fine-tuning, we obtain two models: CodeBERT-Adv and GraphCodeBERT-Adv and evaluate them on the adversarial examples generated in RQ2. Table 3 shows the new models' prediction accuracy on previously generated adversarial examples. It is noted that the original victim models (that are not hardened by adversarial retraining) predict all these examples wrongly (i.e., they have an accuracy of 0%). From Table 3, we can observe that all the adversarially fine-tuned models perform much better than the original ones. The average improvement on examples generated by solely using Greedy-Attack and employing GA-Attack is close. CodeBERT-Adv improves accuracy against Greedy-Attack and *ALERT* by 87.76% and 87.59%, respectively. GraphCodeBERT-Adv improves accuracy against Greedy-Attack and *ALERT* by 92.16% and 93.31%. Accuracy improvement on adversarial examples generated by MHM-NS is relatively more minor (67.89% and 75.93% on CodeBERT and GraphCodeBERT).

> **Answers to RQ3**: The adversarial examples generated by *ALERT* are valuable in improving the robustness of victim models. Adversarially fine-tuning victim models with *ALERT*-generated adversarial examples can improve the accuracy of CodeBERT and GraphCodeBERT by 87.59% and 92.32%, respectively.

6 THREATS TO VALIDITY

Internal validity: The results obtained in our experiment can vary under different hyper-parameters settings, e.g., input length, numbers of training epochs, etc. To mitigate the threats, we set the input length to CodeBERT and GraphCodeBERT as 512 (the maximal value) to ensure that they see the same numbers of tokens for the same code snippet. For the remaining hyper-parameters, we keep them the same as described in [14, 20]. We compare the performance of models obtained in this paper with results reported in the literature [14, 20, 30] to show that our models are properly trained.

External validity: In our experiments, we investigate two popular pre-trained models of code on three downstream tasks. However, our results may not generalize to other pre-trained models and downstream tasks. We use a generic parser to extract variable names from code snippets written in C, Python or Java, but it cannot work in other programming languages like Ruby.

7 RELATED WORK

This section describes the works that are related to this paper, including the pre-trained models of code and adversarial attacks on models of code.

Table 3: Robustness analysis on adversarially fine-tuned victim models. The numbers are the prediction accuraies of adversarially fine-tuned models (CodeBERT-Adv and GraphCodeBERT-Adv) on the adversarial examples generated in RQ2.

Tasks	CodeBERT-Adv			GraphCodeBERT-Adv		
	MHM-NS	Greedy	*ALERT*	MHM-NS	Greedy	*ALERT*
Vulnerability Detection	80.46%	87.93%	88.11%	80.81%	88.84%	89.04%
Clone Detection	59.33%	91.38%	87.31%	48.28%	91.23%	91.70%
Authorship Attribution	63.89%	83.97%	87.36%	98.72%	96.40%	96.21%
Overall	67.89%	87.76%	87.59%	75.93%	92.16%	92.32%

7.1 Pre-trained Models of Code

Code representation models like code2vec [3] and code2seq [2] that use syntactic and structural information have shown good performance on a range of downstream tasks. However, some pre-trained models for Natural Languages (NL) like BERT [12] and GPT-3 [8] have recently demonstrated excellent transferability to Programming Languages (PL) and stronger capabilities of capturing semantics information than code2vec or code2seq. Inspired by the success of these language models, pre-trained models of code have recently become more and more popular in the field of code intelligence and benefited a broad range of tasks [9, 14, 20, 26, 43, 47]. These current pre-trained models of code can be divided into two types: embedding models and generative models.

The two models (CodeBERT [14] and GraphCodeBERT [20]) investigated in our experiments are representatives of embedding models. We have described CodeBERT and GraphCodeBERT in Section 2.1. Here, we briefly describe other embedding models. Kanade et al. [26] used the same model architecture and training objectives as BERT but trained it on Python source code to produce CuBERT. Buratti et al. [9] introduced C-BERT, a transformer-based language model trained on 100 popular C language repositories on Github. Both CuBERT and C-BERT were trained on a single programming language, which limits their usage scenarios. CuBERT and C-BERT outperform generic baselines like LSTM models but do not show superior performance than CodeBERT and GraphCodeBERT, so we investigate the latter two in this work.

The other branch of pre-trained models is generative models, which are designed for generative tasks like code completion. Svyatkovskiy et al. [43] introduce GPT-C, a variant of GPT-2 [37] trained on a large corpus containing multiple programming languages and achieved impressive performance in code generation tasks. Lu et al. [30] provides CodeGPT, which has the same model architecture and training objective of GPT-2. CodeGPT was trained on Python and Java corpora from the CodeSearchNet dataset [23]. Despite their success on generation tasks, these models are unable to get complete contextual information as they are unidirectional decoder-only models which only rely on previous tokens and ignore the following ones [47], so we discard generative models in the investigation list of this work.

7.2 Adversarial Attack on Models of Code

Yefet et al. [51] proposed DAMP, a white-box attack technique that adversarially changes variables in code using gradient information of the victim model. Although their method shows effectiveness in attacking three models *code2vec* [3], *GGNN* [1], and *GNN-FiLM* [7], it requires victim models to process code snippets using one-hot encoding, which is not applicable to CodeBERT [14] and GraphCodeBERT [20] investigated in our paper as they use BPE [16, 29] to process tokens. Srikant et al. [41] apply PGD [31] to generate adversarial examples of code. Besides, these white-box approaches generate substitutes by changing a one-hot encoding to another and mapping it back to a token, which cannot guarantee to satisfy naturalness requirements. Such a white-box attack is less practical since victim models are usually deployed remotely, making parameter information hard to be accessed.

There are several black-box methods for evaluating the robustness of models of code. One that has been shown to be much more effective than the others is MHM [52], which we use as our baseline. We have presented the details of MHM in Section 2.3. Here we present the other related studies. Wang et al. [45] provide a benchmark consisting of refactored programs and evaluate the performance of neural embedding programs on it. Rabin et al. [36] also uses variable renaming to evaluate the generalizability of neural program analyzers, and show that GGNN [15] changes its prediction on 33.39% of transformed code. Pour et al. [35] proposed a testing framework for DNN of source code embedding, which can decrease the performance of code2vec [3] on method name prediction task by 2.05%. Applis et al. [5] use metamorphic program transformations to assess the robustness of ML-based program analysis tools in a black-box manner.

Adversarial attack on models of code can be conducted beyond generating adversarial examples for a well-trained model. Schuster et al. [38] show that code completion models are vulnerable to poisoning attacks that add some carefully-designed files to the training data of a model. Nguyen et al. [34] show that the state-of-the-art API recommender systems can be attacked by injecting malicious data into their training corpus.

8 CONCLUSION AND FUTURE WORK

In this paper, we highlight the naturalness requirement in generating adversarial examples for models of code. We propose *ALERT* (**N**atura**l**n**e**ss Awa**r**e A**t**tack), a black-box attack that adversarially transforms inputs (code snippets) to force pre-trained models to produce wrong outputs. *ALERT* can generate naturalness-aware substitutes. A user study confirms that these substitutes can help generate adversarial examples that look natural to human judges. In contrast, users consistently think examples generated by a prior method that employs random selection to be unnatural. Apart from being aware of naturalness, *ALERT* is also effective in finding adversarial examples. We apply *ALERT* to victim models fine-tuned on state-of-the-art pre-trained models (CodeBERT and

GraphCodeBERT). The results show that on attacking CodeBERT, *ALERT* can achieve average success rates of 53.62%, 27.79%, and 35.78% across three downstream tasks: vulnerability prediction, clone detection and code authorship attribution. It outperforms the baseline by 17.96%, 7.74% and 16.51%. On GraphCodeBERT, our approach can achieve average success rates of 76.95%, 7.96% and 61.47% on the three tasks, respectively, outperforming the baseline by 21.78%, 4.54%, and 29.36%. We also explore the value of adversarial examples to harden CodeBERT and GraphCodeBERT through an adversarial fine-tuning procedure and demonstrated the robustness of CodeBERT and GraphCodeBERT against *ALERT* increased by 87.59% and 92.32%, respectively. We open-source *ALERT* at https://github.com/soarsmu/attack-pretrain-models-of-code.

In the future, we plan to consider more victim models and more downstream tasks. We also plan to boost the effectiveness of *ALERT* and improve the robustness of victim models further.

ACKNOWLEDGMENTS

This research was supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant.

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. 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*.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[4] 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 *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10492)*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer, 65–82.

[5] Leonhard Applis, Annibale Panichella, and Arie van Deursen. 2021. Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1377–1381.

[6] Muhammad Hilmi Asyrofi, Zhou Yang, Imam Nur Bani Yusuf, Hong Jin Kang, Ferdian Thung, and David Lo. 2021. BiasFinder: Metamorphic Test Generation to Uncover Bias for Sentiment Analysis Systems. *IEEE Transactions on Software Engineering* (2021).

[7] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. (2020).

[9] Luca Buratti, Saurabh Pujar, Mihaela A. Bornea, J. Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring Software Naturalness through Neural Language Models. *CoRR* abs/2006.12641 (2020). arXiv:2006.12641

[10] Nicholas Carlini and David A. Wagner. 2018. Audio Adversarial Examples: Targeted Attacks on Speech-to-Text. In *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*. IEEE Computer Society, 1–7.

[11] Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. 2020. A Theory of Dual Channel Constraints. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (Seoul, South Korea) *(ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 25–28.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 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)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186.

[13] Javid Ebrahimi, Daniel Lowd, and Dejing Dou. 2018. On Adversarial Examples for Character-Level Neural Machine Translation. In *Proceedings of the 27th International Conference on Computational Linguistics*. Association for Computational Linguistics, Santa Fe, New Mexico, USA, 653–663.

[14] 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*. Association for Computational Linguistics, Online, 1536–1547.

[15] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

[16] Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal archive* 12 (1994), 23–38.

[17] Xiang Gao, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. 2020. Fuzz Testing Based Data Augmentation to Improve Robustness of Deep Neural Networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1147–1158.

[18] Adam Gleave, Michael Dennis, Cody Wild, et al. 2020. Adversarial Policies: Attacking Deep Reinforcement Learning. In *ICLR*.

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

[20] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu andz Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

[21] Wenbo Guo, Xian Wu, Sui Huang, and Xinyu Xing. 2021. Adversarial Policy Learning in Two-player Competitive Games. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 3910–3919.

[22] Hossein Hosseini, Baicen Xiao, Mayoore Jaiswal, and Radha Poovendran. 2017. On the limitation of convolutional neural networks in recognizing negative images. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 352–358.

[23] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436

[24] 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*. AAAI Press, 8018–8025.

[25] Ankur Joshi, Saket Kale, Satish Chandel, and D Kumar Pal. 2015. Likert scale: Explored and explained. *British Journal of Applied Science & Technology* 7, 4 (2015), 396.

[26] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 5110–5121.

[27] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1073–1085.

[28] 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)*. Association for Computational Linguistics, Online, 6193–6202.

[29] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A

Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692

[30] 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. *CoRR* abs/2102.04664 (2021). arXiv:2102.04664

[31] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[32] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, USA.

[33] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics* 21, 6 (1953), 1087–1092.

[34] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Massimiliano Di Penta, and Davide Di Ruscio. 2021. Adversarial Attacks to API Recommender Systems: Time to Wake Up and Smell the Coffee?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 253–265.

[35] Maryam Vahdat Pour, Zhuo Li, Lei Ma, and Hadi Hemmati. 2021. A Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 36–46.

[36] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.

[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[38] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1559–1575.

[39] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. arXiv:1508.07909 [cs.CL]

[40] Jieke Shi, Zhou Yang, Junda He, Bowen Xu, and David Lo. 2022. Can Identifier Splitting Improve Open-Vocabulary Language Model of Code?. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society.

[41] 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*.

[42] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 476–480.

[43] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation Using Transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1433–1443.

[44] Yida Tao, DongGyun Han, and Sunghun Kim. 2014. Writing Acceptable Patches: An Empirical Study of Open Source Project Patches. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 271–280.

[45] Ke Wang and Mihai Christodorescu. 2019. COSET: A Benchmark for Evaluating Neural Program Embeddings. *CoRR* abs/1905.11445 (2019). arXiv:1905.11445

[46] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Networkand Flow-Augmented Abstract Syntax Tree. arXiv:2002.08653 [cs.SE]

[47] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 [cs.CL]

[48] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). 3034–3040.

[49] Chengran Yang, Bowen Xu, Junaed Younus Khan, Gias Uddin, Donggyun Han, Zhou Yang, and David Lo. 2022. Aspect-Based API Review Classification: How Far Can Pre-Trained Transformer Model Go?. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society.

[50] Zhou Yang, Jieke Shi, Muhammad Hilmi Asyrofi, and David Lo. 2022. Revisiting Neuron Coverage Metrics and Quality of Deep Neural Networks. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society.

[51] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 162:1–162:30.

[52] 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. (2020), 1169–1176.

[53] Ting Zhang, Bowen Xu, Ferdian Thung, Stefanus Agus Haryono, David Lo, and Lingxiao Jiang. 2020. Sentiment Analysis for Software Engineering: How Far Can Pre-trained Transformer Models Go?. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 70–80.

[54] Xin Zhou, DongGyun Han, and David Lo. 2021. Assessing Generalizability of CodeBERT. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*. IEEE, 425–436.

[55] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 10197–10207.