Structural Adversarial Attack for Code Representation Models

Yuxin Zhang, Ruoting Wu, Jie Liao, and Liang Chen(✉)

School of Computer Science, Sun Yat-sen University, Guangzhou, China
{zhangyx355,wurt8,liaoj27}@mail2.sysu.edu.cn, chenliang6@mail.sysu.edu.cn

Abstract. As code intelligence and collaborative computing advances, code representation models (CRMs) have demonstrated exceptional performance in tasks such as code prediction and collaborative code development by leveraging distributed computing resources and shared datasets. Nonetheless, CRMs are often considered unreliable due to their vulnerability to adversarial attacks, failing to make correct predictions when faced with inputs containing perturbations. Several adversarial attack methods have been proposed to evaluate the robustness of CRMs and ensure their reliable in application. However, these methods rely primarily on code's textual features, without fully exploiting its crucial structural features. To address this limitation, we propose STRUCK, a novel adversarial attack method that thoroughly exploits code's structural features. The key idea of STRUCK lies in integrating multiple global and local perturbation methods and effectively selecting them by leveraging the structural features of the input code during the generation of adversarial examples for CRMs. We conduct comprehensive evaluations of seven basic or advanced CRMs using two prevalent code classification tasks, demonstrating STRUCK's effectiveness, efficiency, and imperceptibility. Finally, we show that STRUCK enables a more precise assessment of CRMs' robustness and increases their resistance to structural attacks through adversarial training.

Keywords: Code Intelligence · Model Robustness · Code Representation Model · Adversarial Attack

1 Introduction

Recently, the integration of collaborative computing and artificial intelligence has facilitated the development of distributed deep learning (DL) training methods [5]. These methods leverage the power of multiple computing devices to train DL models, enabling the continuous expansion of model sizes and the enhancement of computational capabilities. Consequently, these models have demonstrated powerful data processing, analysis, and learning capabilities across various fields, including computer vision, natural language processing, and graphical data analysis [10,21,41]. Code intelligence field is also growing quickly as more

H. Gao et al. (Eds.): CollaborateCom 2023, LNICST 562, pp. 392–413, 2024.
https://doi.org/10.1007/978-3-031-54528-3_22

and more DL models benefiting from collaborative computing have been applied to various tasks such as source code processing and collaborative code development [14]. To further enhance the productivity of programmers in collaborative software development on platforms such as GitHub, numerous tools utilizing DL models have emerged, with typical examples including Copilot[1] and ChatGPT[2]. However, the widespread application of DL models in the code intelligence field faces a potential crisis due to their lack of robustness. Different programming styles and intentional modifications can result in the generation of adversarial example (x_{adv}) with the same semantics as the original code (x). Even the state-of-the-art (SOTA) DL models may produce completely inconsistent results for x and x_{adv} [24]. This lack of robustness, especially in security-sensitive tasks like malicious code classification [28], can lead to severe consequences, including system crashes. Furthermore, this issue hinders the development of deep learning models integrated with collaborative computing in the field of code intelligence.

The issue of non-robustness in DL models has received significant attention across various domains [6,19,20,32]. Researchers within the code intelligence field also recognize the importance of evaluating and improving models' robustness. To assess the robustness of DL models in code intelligence field, several adversarial attack methods against CRMs have been proposed [34–37]. However, most of them draw inspiration from the attack methods in NLP filed by renaming variable names in the code to obtain adversarial examples. For example, MHM [37] generates x_{adv} by performing iterative variable name renaming based on Metropolis-Hastings sampling. We argue that these methods focus solely on textual features and overlook the crucial structural features of the code.

As a structured language, code's structural features contain a wealth of semantic information, rendering them an essential component for learning and comprehending code. These features can be obtained from code's structured representation, such as abstract syntax trees (AST) and data flow graphs (DFG). An increasing number of CRMs use the structural features of code to learn its semantics and thus improve the performance in downstream tasks [31]. It is necessary to consider the structural features of code when designing the attack methods for CRMs. While methods such as S-CARROT$_{\text{A}}$ [36] and CLONEGEN [40] attempt to generate adversarial examples using code's structural features, they have not yet fully exploited these structural characteristics. This results in limitations in accurately assessing model robustness and generating naturally adversarial examples. The substantial potential to utilize code structure for devising potent adversarial attacks against CRMs merits further academic exploration.

To compensate for the lack of prior attack methods that insufficiently exploit structural features of code and to further explore the potential of these features in developing attack methods, we present STRUCK (**Stru**ctural adversarial atta**ck**). STRUCK is an attack method that uniquely capitalizes on the structural features of code. As a non-targeted, black-box attack method, it solely utilizes the original inputs and outputs of target models during the attack process.

[1] https://github.com/features/copilot.
[2] https://openai.com/blog/chatgpt.

By thoroughly leveraging the structural features of code, STRUCK facilitates a more accurate evaluation of the robustness of CRMs compared to previous non-targeted, black-box attack methods.

Inspired by research in graph attack [9] and considering the hierarchical structural features of code, STRUCK incorporates two perturbation levels: global (STRUCK_G) and local (STRUCK_L). Each perturbation level contains multiple perturbation methods. Similar to graph attack methods that modify connected edges in graph data, the perturbation methods in STRUCK_G perturb the code's structural features at the global level by refactoring the code. Furthermore, similar to the graph attack methods which add dummy nodes and corresponding connected edges to the graph data, the perturbation methods in STRUCK_L perturb the structural features of the code at the local level by adding code statements of different sizes to insertable positions.

To demonstrate the performance of STRUCK, we conduct a comprehensive evaluation of two representative source code classification tasks: functionality classification and defect prediction. Three categories of CRMs are assessed in the evaluation process: traditional sequential models (LSTM, GRU), classical structural models (GGNN [4], GCN [18]), and high-performing pre-trained models (CodeBERT [11], GraphCodeBERT [15], UniXcoder [14]). The experimental results demonstrate that STRUCK can effectively generate adversarial examples. Specifically, it induces an average relative performance degradation of 82.9% and 99.92% in functionality classification and defect prediction tasks respectively for target models, outperforming advanced adversarial attack methods such as MHM [37], S-CARROT_A [36] and CLONEGEN [40]. Meanwhile, the average number of model invocations by STRUCK is only half of MHM's, indicating a more efficient generation of adversarial examples. Moreover, STRUCK demonstrates a lower average perturbation size (52) compared to S-CARROT_A (55) and CLONEGEN (140) which also leverage code structural features. Our user study confirms that from programmers' perspective, STRUCK exhibits superior imperceptibility, rendering the generated adversarial examples more natural compared to those generated by S-CARROT_A and CLONEGEN. Furthermore, we investigate the value of STRUCK for CRMs' robustness assessment and enhancement. The results reveal that STRUCK accurately evaluates the model's robustness and effectively improves its robustness against structural attacks through adversarial training. The contributions of this paper are summarized as follows:

1. We highlight the idea of fully utilizing the structural features of code to perform attacks on CRMs.
2. We propose an adversarial attack method (STRUCK) for CRMs. It comprehensively explores and exploits the structural features of code, integrating multiple perturbation methods to generate adversarial examples.
3. Extensive experiments demonstrate that STRUCK is superior in terms of its efficacy, efficiency, and imperceptibility when attacking CRMs.
4. We have demonstrated that STRUCK can accurately evaluate the robustness of CRMs and enhance their robustness to structural attacks via adversarial training, thereby improving their reliability.

2 Preliminary

2.1 Definitions for Codes

Semantic Equivalence of Codes. x and x' are considered semantically equivalent if x can be transformed into x' through a series of equivalence transformations. We use $F(x,k)$ to represent the result of applying k equivalent transformations to x and if $x' = F(x,k)$, x and x' are considered semantically equivalent.

$$F(x,i) = T(F(x,i-1),t_i), i = \{1,2,\ldots,k\} \tag{1}$$

with the initial condition $F(x,0) = x$ and the constraint that $t_i \in \mathcal{T}(x_{i-1})$. Here, $\mathcal{T}(x)$ is a subset of all equivalence transformation types $\mathcal{T}$, denoting the set of valid equivalence transformation types that can be applied to x. $T(x,t)$ represent the result of transforming x using $t \in \mathcal{T}(x)$.

Code Distance. We represent the differences between two semantically equivalent codes using the code distance $D(x,x')$, which measures the number of distinct tokens between them. For a transformation $t \in \mathcal{T}(x)$, we denote its effect on x as $d(x,t)$.The value of $d(t,x)$ is jointly determined by t and x, representing the number of tokens changed, added, or deleted in x. Letting $x_0 = x$, we can obtain the following relationship:

$$D(x,x') = D(x,F(x,k)) \leq \sum_{i=1}^{k} d(x_{i-1},t_i). \tag{2}$$

$D(x,x')$ is bounded by the accumulated sum of the effects of k equivalent transformations, as some transformations may counteract the changes introduced by previous ones. We can assume that the smaller the effects of the used equivalence transformations, the smaller the value of $D(x,x')$.

2.2 Code Representation

Code Graph is a significant structured representation of code based on AST and DFG, comprising essential structural information such as function calls and semantic flow. The left of Fig. 1 shows an example of converting a code snippet from text to graph representation, following the approach proposed by Allamanis et al. [4]. In the code graph, nodes serve as fundamental units that connect through various types of edges, such as Child Edges derived from the AST of code, to form subgraphs. Each subgraph corresponds to a unique statement or functional module in the code snippet. The combination of all these subgraphs results in the formation of the complete code graph.

Code representation model (CRM), denoted as R, can map code snippets into low-dimensional dense real-valued vectors, which can be understood by other models. Learning code semantics through R is essential for performing various downstream tasks. Different R has distinct input format requirements and

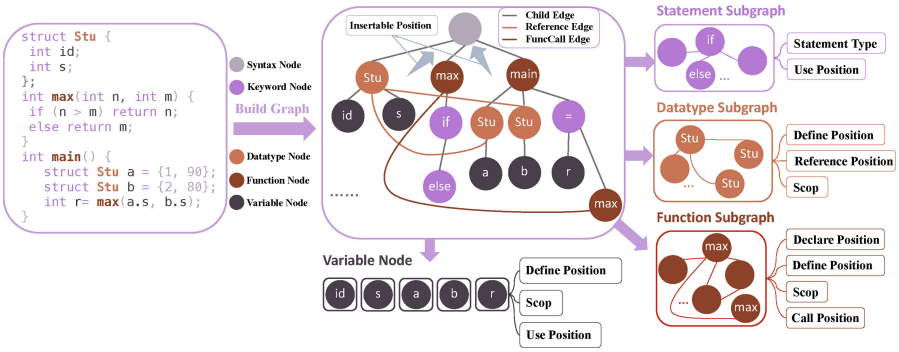


Fig. 1. Left: Conversion of C code to a graph representation. Right: Extraction of code graph structural information using *Extract*.

uses different code features [31]. Sequential models [2] leverage textual features to comprehend the semantics of code sequences, while structural models [4] learn from structured representations of code, such as AST, DFG, and code graphs.

Code classification task we focused on is a fundamental task that aims to predict the type of a given code. This task relies on a labeled code dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$. We use $\mathcal{D}_{\text{train}}/\mathcal{D}_{\text{dev}}/\mathcal{D}_{\text{test}}$ to respectively denote the train/valid/test set used, and (x, y) represents a code-label pair. Code classification model $(M = R \mid C)$ is obtained by combining R with a classifier (C). The C takes the code representation vector $(r = R(x))$ as input and outputs the prediction type of x.

2.3 Adversarial Learning

Adversarial attack aims to mislead DL models by generating adversarial examples through minute input perturbations, leading to unexpected outcomes. A trained model (M) that excels at the downstream classification task can correctly identify x as y $(y = M(x))$. A successful attack obtains x_{adv} by perturbing x so that $y =\neq M(x_{adv})$, thereby misleading M. We call M the target model of attack, and name x and x_{adv} the original and adversarial example, respectively.

Given the significant impact of adversarial attacks on the deployment of DL models, a variety of attack methods have been proposed [3,39]. Attack methods can be classified into targeted and non-targeted types depending on if they have a specific target or not. We focus on the non-targeted attack, which we consider as the foundation for targeted attacks. Based on the accessibility of information related to the target model, attack methods also can be classified into white-box, gray-box, and black-box [33]. In white-box configurations, attackers have access to all target model parameters. Black-box settings limit attackers to model outputs. Gray-box settings are between the two. DL models are frequently remotely deployed, attackers can only access models through API to call them and receive the outputs. So we focus on the more practical black-box type.

We use $\mathcal{A}((x,y)|M,\delta)$ [7] to denote the set of all non-targeted adversarial examples of the (x,y) that mislead M under the restriction of perturbation δ.

$$\mathcal{A}((x,y)|M,\delta) = \{x_{adv} \mid M(x_{adv}) \neq M(x) \wedge D(x,x_{adv}) < \delta\}. \tag{3}$$

We focus on (x,y) satisfies $M(x) = y$. Simultaneously, we employ the distance between x and x_{adv} ($D(x,x_{adv})$) to measure the perturbation size **(Psize)** generated by attackers during the generation of x_{adv}, considering only the adversarial examples produced under the perturbation size constraint δ as valid.

Model Robustness. A robust DL model's outputs should remain unchanged after input perturbations. $Rob.(M|(x,y))$ [36] indicates if M is robust to (x,y).

$$Rob.(M \mid (x,y)) = \begin{cases} 1, & \text{If } \mathcal{A}((x,y)|M,\delta) = \varnothing, \\ 0, & \text{Otherwise.} \end{cases} \tag{4}$$

The model is robust to (x,y) when $Rob.(M \mid (x,y)) = 1$, indicating that attackers cannot use (x,y) to generate x_{adv} that mislead M. Further, we denote by $Rob._{adv}(M|(x,y)) = 1$ that the attacker adv cannot generate x_{adv} for M using (x,y). We count the examples in $\mathcal{D}$ for which the model is robust and use the proportion of robust examples to measure the model's robustness.

$$Rob.(M \mid \mathcal{D}) = \frac{\sum_{(x,y)\in\mathcal{D}_M} Rob.(M \mid (x,y))}{|\mathcal{D}_M|}, \tag{5}$$

where $\mathcal{D}_M = \{(x,y)|M(x) = y\}$. We estimate DL models' robustness using different attack methods. $Rob._{adv}(M \mid \mathcal{D})$ gives the proportion of examples with $Rob._{adv}(M|(x,y)) = 1$ in $\mathcal{D}_M$. And M's robustness cannot exceed this proportion, i.e. $Rob.(M \mid \mathcal{D}) \leq Rob._{adv}(M \mid \mathcal{D})$. The precision of assessing M's robustness correlates positively with the effectiveness of the attack methods used.

Adversarial training is an efficient way to boost DL models' robustness. It first update $\mathcal{D}_{\text{train}}$ with adversarial examples generated by adv for M to obtain $\mathcal{D}_{\text{adv}}$, then train M from scratch with $\mathcal{D}_{\text{adv}}$. In general, M achieves somewhat improved robustness against adv after adversarial training [37].

3 Present Work: STRUCK

In this section, we first give an overview of STRUCK Subsect. (3.1), followed by the introduction of the extracted **structural information** (*codeinfo*) from codes by STRUCK Subsect. (3.2). Subsequently, we provide a detailed exposition of the equivalent perturbation methods at the global Subsect. (3.3) and local Subsect. (3.4) levels. Finally, we will go over the *controller* used by STRUCK during the attack in detail Subsect. (3.5).

3.1 Overview

STRUCK is a non-targeted, black-box attack method that generates adversarial examples by fully utilizing the structural features of original examples. Algorithm 1 gives the specific operation for M using (x, y), while Fig. 2 illustrates a single iteration of the process, which can aid in understanding the algorithm.

Before starting an iterative attack, preparations are needed (Line 1–4). STRUCK extracts the $codeinfo$ from x and obtains the Psize limitation δ. The δ is based on the number of x's tokens and α (perturbation limitations factor). STRUCK then initializes the *controller* to select the perturbation methods.

Algorithm 1: STRUCK Algorithm.

Input:
Target Model M, Code-label pair (x, y), s.t. $y = M(x)$;
Max attack iteration m, Max candidate number n, Perturbation limitations factor α.
Output:
Adversarial example $x_{adv} \in \mathcal{A}((x, y) \mid M, \delta)$, or None.

1 $x_0 = x$, $prob = \text{Prob}(x, y; M)$;
2 $codeinfo = Extract(x)$;
3 $\delta = codeinfo.nums \times \alpha$;
4 $controller = Controller(codeinfo, m)$;
5 **for** i *in* $\{0, 1, \cdots, m-1\}$ **do**
6 $\quad t_{adv} = controller.choose(codeinfo, i, \delta)$;
7 $\quad$ **if** t_{adv} *is* $\varnothing$ **then**
8 $\quad\quad$ return None;// Fail
9 $\quad$ **end**
10 $\quad \{x'_{i,1}, \cdots, x'_{i,n}\} = t_{adv}(x_i, codeinfo, n, \delta)$;
11 $\quad$ **for** j *in* $\{1, 2, \cdots, n\}$ **do**
12 $\quad\quad$ **if** $x'_{i,j}$ *in* $\mathcal{A}((x, y) \mid M, \delta)$ **then**
13 $\quad\quad\quad$ return $x'_{i,j}$;// Success
14 $\quad\quad$ **end**
15 $\quad$ **end**
16 $\quad$ ind $= min$ Prob$(\{x'_{i,1}, \cdots, x'_{i-1,n}\}, y; M)$;
17 $\quad prob_{ind} =$Prob$(x'_{i,ind}, y; M)$;
18 $\quad$ **if** $prob_{ind} < prob$ **then**
19 $\quad\quad x_{i+1} = x'_{i,ind}$;
20 $\quad\quad prob = prob_{ind}$;
21 $\quad\quad codeinfo = Extract(x'_{i,ind})$;
22 $\quad\quad \delta = \delta - D(x_i, x'_{i,ind})$;
23 $\quad$ **else**
24 $\quad\quad x_{i+1} = x_i$;
25 $\quad$ **end**
26 $\quad controller.update(t_{adv})$;
27 **end**
28 return None;// Fail

In Algorithm 1 (also see Fig. 2), at each iteration, STRUCK first utilizes the *controller* to choose the equivalent perturbation method t_{adv}. Then the selected t_{adv} uses $codeinfo$ to generate n candidate examples ($\mathcal{S} = \{x'_{i,1}, \cdots, x'_{i,n}\}$) for the current code snippet x_i, and $\forall x' \in \mathcal{S}$ satisfies $D(x_i, x') < \delta$. Next, STRUCK tests $\mathcal{S}$, if $\mathcal{R} = \mathcal{S} \cap \mathcal{A}((x, y) \mid M, \delta) \neq \varnothing$, the attack process will be completed, and $\forall x' \in \mathcal{R}$ is an adversarial example for M. Otherwise, STRUCK tests the candidate example $x'_{i,ind}$ with the lowest predicted probability value of the original label y from $\mathcal{S}$. If the predicted probability value Prob$(x'_{i,ind}, y; M)$ is less

than the saved predicted probability value *prob*, it means t_{adv} is valid since t_{adv} misleads M, so update the relevant information (line 19–22). Whether the perturbation is valid or not, the *controller* records t_{adv} (line 26), preventing the same perturbation method from being performed indefinitely. STRUCK assures the diversity of perturbations through *controller* recording. The attack process terminates when: I. an adversarial example is generated (line 13); II. no perturbation method is available (line 8); III the attack reaches m iterations (line 28).

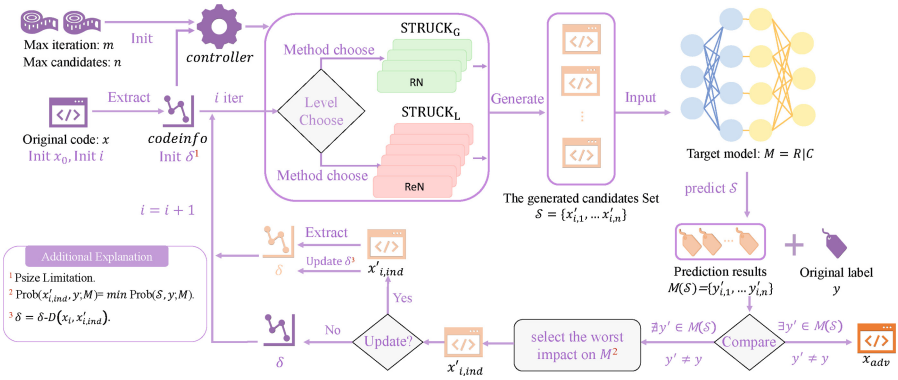


Fig. 2. Workflow of STRUCK (start at the top left).

3.2 Code Information

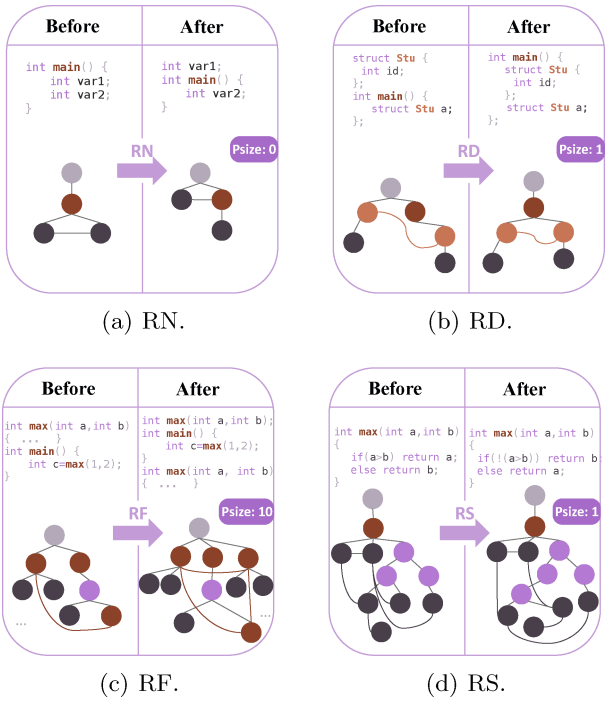
To more effectively exploit the structural features of code when attacking target models, STRUCK employs *Extract* to comprehensively extract structural information of varying granularity from the input code prior to attacking. $codeinfo = Extract(x)$ represents the extracted structural information. In particular, *Extract* analyzes the structure of the code and extracts its essential information, which is then organized within $codeinfo$. As illustrated on the right side of Fig. 1, *Extract* can draw out different substructures from the code graph of the example, including Variable Nodes, Statement Subgraphs, Datatype subgraphs, and Function Subgraphs. It also captures key information within these substructures, such as the definition position and scope (valid range) of variable nodes.

Although the code graph includes various types of nodes, STRUCK focuses on variable nodes (such as "Stu" nodes) because they are nodes defined by programmers and possess more uncertainty than keyword nodes (such as "if" nodes) and other types of nodes. STRUCK also pays attention to conditional/loop statement subgraphs which often play key roles in the code. The user-defined datatype subgraphs and function subgraphs extracted by STRUCK are very helpful to the analysis of the code graph. STRUCK also uses the isolation of subgraphs in the code graph to collect information on insertable positions, denoted as Pos. The isolation of subgraphs implies that there is no node intersection between two

adjacent subgraphs. Inserting customized nodes or subgraphs between these two subgraphs will not affect their semantics. For example, in Fig. 1, there are no node intersections between the "Stu datatype subgraph" and the "Max Function Subgraph", indicating that the position marked by the arrow is an insertable location for perturbations and belongs to *Pos*.

3.3 STRUCK$_{\mathbf{G}}$

Due to the code's specificity, its semantics can remain constant even if statements, functions, and other elements modify their positions and expressions. STRUCK$_{\mathrm{G}}$ (STRUCK on the global level) is a proposed perturbation set on the global level of code based on code refactoring. Figure 3 shows the perturbation methods in STRUCK$_{\mathrm{G}}$ by recombining nodes and subgraphs in code graph. These perturbation methods simulate programming development situations.



(a) RN. (b) RD.

(c) RF. (d) RS.

Fig. 3. Perturbation methods in STRUCK$_{\mathrm{G}}$ performing on example codes.

Reconstruct-Node (RN), **Reconstruct-Datatype (RD)**, and **Reconstruct-Function (RF)** take advantage of uncertainty in the location of variables, user-defined datatypes, and user-defined functions, respectively, to perform perturbations. **Reconstruct-Statement (RS)** utilizes the diversity of

loops and conditional statements to accomplish code refactoring, as previously implemented in CLONEGEN [40]. And these perturbation methods make use of $codeinfo$ to complete code's refactoring, ensuring $\forall t \in \text{STRUCK}_\text{G}$ satisfies $t \in \mathcal{T}(x)$. The perturbable objects for RN, RD, RF and RS are variables, user-defined datatypes, user-defined functions and conditional/loop statements, respectively. The idea behind STRUCK_G is to refactor code without introducing new information, which leads to perturbation methods with negligible Psize. Specifically, the RN, RD and RF achieve a minimum perturbation size d_{min} of 0 on the code, denoted as $\forall t \in \{RN, RD, RF\},\ d_{min}(t) \approx 0$. For RS, some auxiliary identifiers will be added, such as Fig. 3(d) by adding "!" to reconstruct the conditional statement, so $d_{min}(t_{RS}) = 1$. Since the lower minimum Psize, the modification to the input code caused by the perturbation method in STRUCK_G has high imperceptibility.

3.4 STRUCK_L

Redundant code refers to unnecessary segments of code that do not influence the semantics of it. STRUCK_L (STRUCK on the local level) is a perturbation set based on the redundant code mistakes made by programmers. The perturbations in STRUCK_L perturb the local level by inserting redundant subgraphs at any $p \in Pos$, as shown in Fig. 4. It is noteworthy that $\forall t \in \text{STRUCK}_\text{L},\ d_{min}(t) \geq 3$.

Repeat-Node (ReN) is inspired by the Copy-Move attack in the image processing field [8]. As shown in Fig. 4(a), ReN first copies "var1" and then pastes it twice at the insertable position near "var1" to introduce perturbation. **Change-Node-Decl (CND)** generates a new declaration statement subgraph for a variable node in code graph and inserts the new subgraph outside the scope of the variable node. Figure 4(b) illustrates the process of CND using "var1" for perturbation. **Change-Recover-Node (CRN)** takes advantage of the property that the value of a variable node in an inactive state does not change the code semantics and achieves perturbation by changing and recovering the value of variables in pairs. Figure 4(c) illustrates the process of CRN changing and recovering the value of "var1". **Redundant-Node (RedN)** uses common idempotent operations (self-assignment, self-division, bitwise operations, etc.) to generate a new assignment statement subgraph for a variable node in code graph without changing its value, and then inserts it at a insertable position. Figure 4(d) illustrates the process of RedN for "var". **Generate-Node (GN)** introduces a declaration statement subgraph of a new variable at any insertable location. Figure 4(f) shows the process of GN inserting a new variable "a" declaration subgraph into the code graph. **Generate-Statement (GS)** uses redundant conditional expressions to perturb the original examples. It generates redundant expressions and the corresponding loop/conditional statement subgraphs with imperceptibility according to the characteristics of each code at different locations. Figure 4(f) shows that GS first generates a redundant conditional expression "var != 1" using the variable node "var". Then it generates a redundant loop statement subgraph in combination with the existing declaration statement subgraph ("var = 1").

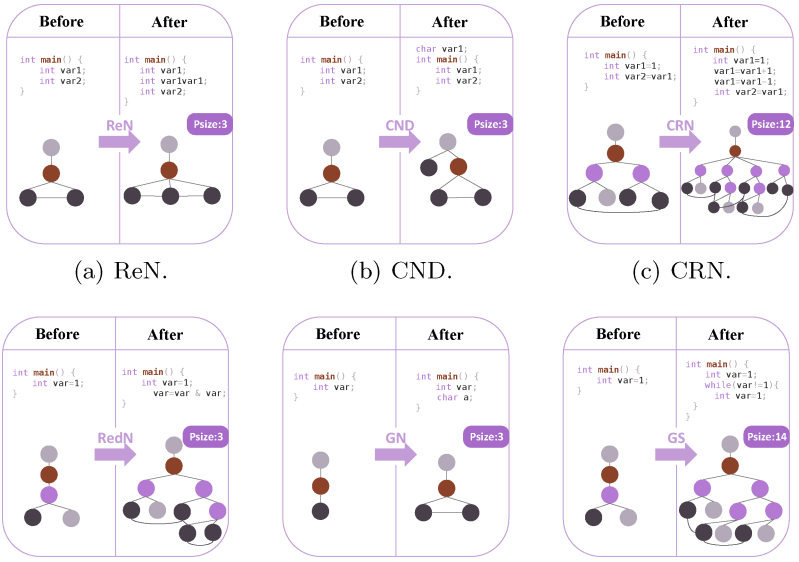
The perturbable objects of ReN, CND, CRN, and RedN are user-defined variables. GN and GS generate redundant subgraphs based on *Pos*, so their perturbable objects are *Pos*.

3.5 Controller

Controller Initialization. The *controller* initializes with the extracted *codeinfo* from x and the maximum attack iteration m (Line 4 in Algorithm 1). Based on the types of perturbable objects for different perturbation methods described in Sect. 3.3 and Sect. 3.4, the *controller* first initializes the specific perturbable objects for perturbation methods in STRUCK_G and STRUCK_L. We use $objects(t, x)$ to denote the set of specific perturbable objects of perturbation method t on x. In Fig. 4(b), $objects(t_{\text{CND}}, x) = \{var1, var2\}$ means CND can perturb x by generating and inserting a new declaration statement subgraph for $var1$ or $var2$. If $objects(t, x) = \varnothing$, the *controller* removes t from its corresponding perturbation level. Otherwise, the *controller* initializes the number of times that t can be performed using $objects(t, x)$ and the hyperparameter attack amplification factor *scale*.

$$times_t = \mid objects(t, x) \mid \times scale. \tag{6}$$

Since there are always *Pos* in code, t_{GN} and t_{GS} can always be executed, allowing for the application of STRUCK_L. The *controller* maintains the perturbable levels of x, denoted as L_c. When $L_c = \{\text{STRUCK}_\text{L}\}$, $m_L = m$ is STRUCK_L's



(a) ReN. (b) CND. (c) CRN.

(d) RedN. (e) GN. (f) GS.

Fig. 4. Perturbation methods in STRUCK_L performing on example codes.

maximum perturbation iterations. When $L_c = \{\text{STRUCK}_\text{G}, \text{STRUCK}_\text{L}\}$, the STRUCK_G's maximum iterations of perturbations is $m_G = m \times \theta$ by using the hyperparameter $\theta \in (0, 1)$. Correspondingly, $m_L = m - m_G$.

Controller Choose. In each iteration of STRUCK, the *controller* is used to choose the perturbation method t_{adv}. Only if t_{adv} is selected successfully can STRUCK continue to perturb x. Otherwise, STRUCK attack x for M fails. In this part, we only discuss the case of $L_c = \{\text{STRUCK}_\text{G}, \text{STRUCK}_\text{L}\}$. As shown in Fig. 2, the *controller* first selects the perturbation level T_L from L_c. The perturbation methods in STRUCK_G exhibit higher imperceptibility due to their low d_{min}, so the *controller* gives priority to STRUCK_G when choosing. Only when there is no perturbation method available in STRUCK_G, or the current number of perturbation iteration i exceeds m_G, the *controller* will choose STRUCK_L. After the *controller* selects T_L, it will further choose t_{adv} from T_L.

We consider that the more perturbable objects are, the more candidate examples can be generated, and the higher the likelihood of successfully generating adversarial examples is. Therefore, when the *controller* uses the sampling method to select a perturbation method from T_L, the probability of each perturbation method being selected is as follows:

$$chance(t) = \frac{|objects(t,x)|(1 + \frac{|\delta - d_{min}(t)|}{\delta - d_{min}(t) + \text{1e-8}})}{2\sum_{i=1}^{|T_L|} |objects(t_i,x)|(1 + \frac{|\delta - d_{min}(t_i)|}{\delta - d_{min}(t_i) + \text{1e-8}})}. \tag{7}$$

$\forall t \in T_L$, when $d_{min}(t) < \delta$, the larger $objects(t, x)$ is, the greater the chance that *controller* selects t is. For example, in the code graph of Fig. 1, there are four variables and one user-defined datatype. Therefore $chance(t_{\text{RN}}) > chance(t_{\text{RD}})$.

Controller Update. If *controller* successfully selects t_{adv} in the iteration, it records t_{adv} regardless of whether t_{adv} is valid or not. When the number of perturbations of t_{adv} exceeds $times_{t_{adv}}$ (calculated using Eq. (6)), *controller* removes t_{adv} from the corresponding perturbation level. Also, if all perturbation methods within a perturbation level in L_c are removed, the *controller* will remove that perturbation level from L_c. This updating process enables the *controller* to choose the appropriate perturbation method in the next iteration.

4 Experiment Setup

4.1 Dataset

We choose the source code classification task as the downstream task of the code representation, which serves as the foundation for more complex tasks. Specifically, we have chosen the Open Judge function classification dataset (OJ) [38] and the code defect prediction dataset (CC) [1] as our research objects. The OJ consists of 52,000 executable C/C++ code snippets categorized into 104 classes

with 500 snippets per class. The CC includes 34,174 executable C/C++ code snippets classified into four categories based on their execution results on the CodeChef platform. Prior to in-depth analysis, we preprocess the C/C++ code snippets in both datasets based on [37]. Following TBCNN [22], we divide each dataset into $\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{dev}}$, $\mathcal{D}_{\text{test}}$ in the ratio of 3.2:0.8:1. On the OJ, following [22], we employ Accuracy (Acc) as the performance metric to evaluate different models. And on the CC, which is a 4-class classification problem, we employ Precision (Prec), Recall, and F1 score (F1), with F1 as the primary performance metric.

4.2 Target Models

When selecting target models, we consider input format, generalizability, and performance. Our selection includes: fundamental sequential models such as LSTM and GRU; classical structural models like GGNN [4] and GCN [18]; popular pre-trained models including CodeBERT [11], GraphCodeBERT [15] and UniXcoder [14]. **LSTM** and **GRU** are routinely employed as the backbones of well-performing models to tackle various downstream tasks [17,30]. **GGNN** is the first graph representation model adopted in the code intelligence field and widely utilized as a fundamental component in complex models [12,16]. **GCN** is a conventional graph model included in our study to explore the robustness of graph models in learning code representations. **CodeBERT (CB)** is the first bimodal pre-trained model for programming language and natural language. **GraphCodeBERT (GCB)**, based on CB, leverages a semantic-level structure of code, i.e., DFG, in the pre-training stage. **UniXcoder (UX)** is a unified cross-modal pre-trained model for programming language that incorporates semantic and syntax information from code comment and AST.

Such comprehensive selection of target models not only can reveals the robustness levels of current advanced models, but also provides essential references for assessing and enhancing the robustness of future models with collaborative computing. We have implemented these models according to their original papers, and their performance on the OJ and CC datasets is presented in Table 1.

Table 1. Performance of the Target Models(%).

Model	OJ	CodeChef		
	Acc	Prec	Recall	F1
LSTM	96.88	77.06	75.46	76.19
GRU	96.54	76.08	75.40	75.68
GGNN	95.68	66.59	64.33	65.05
GCN	94.54	58.55	55.54	55.30
CB	98.30	**85.40**	82.61	**83.80**
GCB	97.70	82.92	80.69	81.69
UX	**98.47**	83.99	**82.65**	83.26

4.3 Baselines

To demonstrate the superiority of STRUCK, we compare it with popular non-targeted, black-box attack methods that emphasize either textual features (MHM, I-CARROT$_A$-S and ALERT [34]) or structural features (S-CARROT$_A$ [36] and CLONEGEN [40]). We implement these baseline based on official descriptions.

MHM is a SOTA black-box attack method for CRMs that generates candidate examples by iteratively renaming variable nodes and accepting suggestions based on an acceptance probability. I-CARROT$_A$-S is a simplified version of the SOTA white-box attack method I-CARROT$_A$ [36], which uses random sampling to obtain candidate replacement tokens without exploiting gradient information. **ALERT** is a SOTA black-box attack method for pre-trained models that uses another pre-trained model to generate natural-meaning substitutions when renaming user-defined variables in target code. However, ALERT is limited to attacking pre-trained models because it requires both the target and pre-trained models to use the same vocabulary to maintain the validity of replacement nodes. S-CARROT$_A$ focuses on exploiting code's structural features by inserting and deleting fixed redundant code snippets, with a Psize limitation for fair comparison with STRUCK. **CLONEGEN** evaluates the robustness of clone detection models by generating adversarial examples using 15 simple transformation operators and a deep reinforcement learning-based strategy. We have made appropriate modifications to ensure a fair comparison with STRUCK.

4.4 Metrics

Effectiveness. We use Relative performance degradation (RPD) to measure the effectiveness of attack methods, which is defined as follows:

$$\text{RPD} = \frac{P_{\text{before}} - P_{\text{after}}}{P_{\text{before}}}. \tag{8}$$

Table 2. Performance of Attackers on Target models (%).

Attacker	LSTM		GRU		GGNN		GCN		CB		GCB		UX	
	OJ	CC	OJ	CC	OJ	CC	OJ	CC	OJ	CC	OJ	CC	OJ	CC
MHM	55.20	99.92	69.54	99.91	62.94	97.70	75.95	90.54	19.41	87.15	34.39	94.35	22.30	89.84
I-CARROT$_A$-S	52.03	99.73	67.16	99.66	61.47	97.00	69.56	90.15	18.97	85.79	18.97	92.91	21.96	87.82
ALERT	–	–	–	–	–	–	–	–	24.04	80.23	19.24	84.43	19.93	91.77
S-CARROT$_A$	57.04	96.25	44.66	98.03	26.09	75.54	76.85	76.56	6.85	93.33	6.85	86.98	8.45	83.18
CLONEGEN	27.08	67.75	16.30	53.35	51.72	47.94	24.42	52.07	65.96	70.40	71.00	77.64	59.18	84.00
STRUCK$_G$	13.10	37.79	8.45	39.72	8.43	34.99	29.36	29.49	1.98	20.19	3.27	29.33	1.69	24.49
STRUCK$_L$	77.05	**100.0**	81.57	99.83	81.25	97.66	89.11	99.41	85.12	99.01	77.29	98.69	68.24	99.32
STRUCK	**80.06**	**100.0**	**83.28**	**100.0**	**88.95**	**98.48**	**92.15**	**99.52**	**86.85**	**99.37**	**79.67**	**99.60**	**69.59**	**99.76**

Table 3. Average number of invocations (Invos) of attackers in OJ.

Invos#↓	LSTM	GRU	GGNN	GCN	CB	GCB	UX
MHM	507	455	407	300	530	544	583
I-CARROT$_{A}$-S	372	342	300	303	426	409	445
ALERT	–	–	–	–	439	520	596
S-CARROT$_{A}$	148	156	154	112	138	170	120
CLONEGEN	96	104	97	99	100	96	96
STRUCK$_{G}$	**60**	**62**	**53**	**44**	**39**	**74**	**68**
STRUCK$_{L}$	103	98	89	85	109	96	105
STRUCK	212	217	209	175	252	234	265

For OJ, the P refers to Acc, while for the CC, it represents the F1. The relative performance change of the target model before and after the adversarial attack can visually demonstrate the effectiveness of the attacker. A larger RPD indicates a higher effectiveness of the attacker.

Efficiency. We evaluate the efficiency of *adv* based on the average number of model invocations (Invos) during the attack. Given that M is typically deployed remotely, multiple invocations can be time-consuming and expensive. Thus, a more efficient attack method requires fewer invocations of M.

Imperceptibility. Adversarial examples should have hard-to-detect perturbations [29]. We evaluate the imperceptibility of attack methods from two angles: code imperceptibility (Code-I) and programmer imperceptibility (Programmer-I). For Code-I, we calculate the average Psize (APsize) of generated adversarial examples; a smaller APsize indicates higher Code-I, offering a simple and clear evaluation. To assess Programmer-I for attack methods using code structural features, we conduct a questionnaire survey asking programmers to identify perturbations in code snippets. By analyzing the proportion of detected perturbations, we evaluate the Programmer-I of attack methods.

5 Experiment Results and Analysis

This section will answer the following questions.

RQ1 **Attack Performance**: How successful are the STRUCK attacks on CRMs?
RQ2 **Imperceptibility Evaluation**: Are the adversarial examples generated by STRUCK imperceptible?
RQ3 **Ablation Study**: Is it necessary for STRUCK to employ different perturbation levels and methods?
RQ4 **Robustness Assistance**: Does STRUCK effectively assess and improve the robustness of models?

5.1 RQ1: Attack Performance

Effectiveness. Table 2 presents the RPD of all baselines, STRUCK_G, STRUCK_L, and STRUCK. STRUCK outperforms all other attackers in all datasets and target models, demonstrating its generalizability. In the OJ, STRUCK achieves an average RPD of 82.93% on the target models, which is a significant improvement over MHM (48.53%) and S-CARROT (36.05%), with improvements of 70.89% and 130.30%, respectively. Despite the high susceptibility of target models in the CC dataset to adversarial attacks, STRUCK still achieves the best average RPD of 99.53%. Compared to MHM (94.20%) and S-CARROT (87.12%), STRUCK shows improvements of 5.66% and 14.24%, respectively.

Efficiency. Table 3 shows the Invos needed for various attackers to generate successful adversarial examples on the OJ. Notably, STRUCK_G has the lowest Invos (57). In earlier attack methods, higher attack performance often required more Invos. However, STRUCK's Invos for all target models are lower than MHM, with an average of 223, a 54.35% decrease compared to MHM (489).

Answers to RQ1: STRUCK's success in attacking CRMs shows that it is both effective and efficient, making it a practical and feasible method.

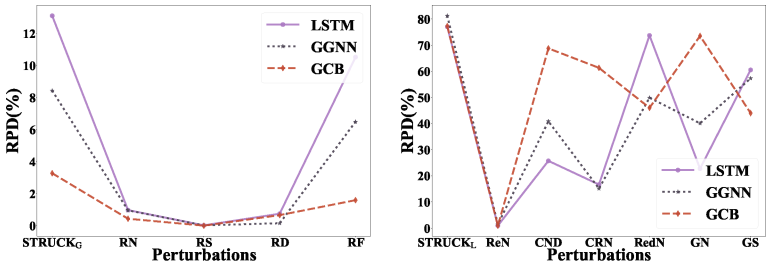
Table 4. Structural attackers' Imperceptibility evaluation in OJ.

Attacker	APsize↓	Unnatural		Avg. unnatural blocks	
		Prop.(%)	△(%)↓	#	△(%) ↓
None	0	40.74	–	1.52	–
S-CARROT_A	55	89.81	1.20	1.89	0.24
CLONEGEN	140	95.37	1.34	4.62	2.04
STRUCK_L	66	78.70	0.93	1.75	0.15
STRUCK	**52**	75.93	**0.86**	1.56	**0.03**

For Unnatural, $\triangle = \frac{\text{Prop.} - \text{Prop.}_\text{None}}{\text{Prop.}_\text{None}}$, while for Avg. unnatural blocks, $\triangle = \frac{\# - \#_\text{None}}{\#}$. Lower △ is better programmer-I.

5.2 RQ2: Imperceptibility Evaluation

Since STRUCK mainly uses code's structural features for adversarial attacks, directly comparing it with methods like ALERT, which focus on textual features, would be unfair. Structural perturbations are usually more noticeable than textual ones. In order to demonstrate the superior imperceptibility of STRUCK at the structural level, we compare it to S-CARROT and CLONEGEN.



(a) STRUCK_G (b) STRUCK_L

Fig. 5. RPD of different perturbation methods in STRUCK_G and STRUCK_L.

Code-I. As shown in Table 4, STRUCK achieves an APsize of 52 on the target models, resulting in a reduction of 4.68% and 62.67% compared to S-CARROT (55) and CLONEGEN (140), respectively.

Programmer-I. To evaluate Programmer-I for attackers using code structural features, we conduct a user study. We sample code snippets from the OJ that can be successfully attacked by all selected attackers and generate adversarial examples for the questionnaire. We create 18 questionnaires with 30 test code snippets each, including original code and adversarial examples. Each test code is divided into five blocks for readability. We invite 18 programmers with at least four years of experience to participate and identify unnatural code blocks. Table 4 shows that even the original 108 codes (None) have a 40.74% Unnatural Proportion (Prob.) and 1.52 average unnatural blocks. This is due to different programming habits. We use the performance of the adversarial example sets obtained from the selected attackers relative to None ($\triangle$) to more accurately measure attackers' Programmer-I. STRUCK has the lowest $\triangle$ among the comparative methods. Compared to S-CARROT_A, the Unnatural Prob. and Avg. unnatural blocks of STRUCK are reduced by 28% and 89% respectively.

Answers to RQ2: Compared to other attackers that leverage code structural features, STRUCK demonstrates the highest level of Code-I and Programmer-I, providing strong evidence of its high imperceptibility.

5.3 RQ3: Ablation Study

STRUCK includes two perturbation levels, STRUCK_G and STRUCK_L, which draw inspiration from successful attack strategies in the graph data analysis field [9] while incorporating code-specific graph features. Table 4 shows that STRUCK has superior imperceptibility compared to STRUCK_L in both the Code-I and Programmer-I, highlighting the importance of designing perturbation methods at both global and local levels. Moreover, the multiple perturbation levels in

STRUCK ensure scalability and allow for the incorporation of new structural perturbation methods in future iterations.

To compare the performance of perturbation methods used in STRUCK_G and STRUCK_L, we conduct an ablation experiment. The experiment uses a single perturbation method in STRUCK_G or STRUCK_L to attack on representative models in OJ. Figure 5 shows the RPD of different perturbation methods. It can be seen that the RPD of any single perturbation method is lower than that of its corresponding level. STRUCK designs multiple perturbation methods for each level to enhance the effectiveness and diversity of perturbations.

Answers to RQ3: The design of two levels and multiple perturbation methods at each level ensure the effectiveness, imperceptibility, diversity and scalability of STRUCK.

Table 5. Model's Robustness on OJ Assessed by Attackers (%).

Robust	LSTM	GRU	GGNN	GCN	CB	GCB	UX
MHM	44.80	30.46	37.07	24.05	80.59	65.60	77.70
I-CARROT_A-S	47.97	32.84	38.53	30.44	81.03	68.78	78.04
ALERT	–	–	–	–	75.96	80.76	80.07
S-CARROT_A	42.96	55.34	48.28	23.15	93.15	69.34	91.55
CLONEGEN	72.92	83.70	73.91	75.58	34.04	29.00	40.82
STRUCK_G	86.90	91.55	91.57	70.64	98.03	96.73	98.65
STRUCK_L	22.95	18.43	18.75	10.89	14.88	22.71	31.76
STRUCK	**19.94**	**16.72**	**11.05**	**7.85**	**13.15**	**20.33**	**30.41**

Table 6. Results of Structural Adversarial Training in OJ (%).

Model	Acc.	S-CARROT_A		CLONEGEN		STRUCK	
		Rob.↑	Improv.↑	Rob.↑	Improv.↑	Rob.↑	Improv.↑
LSTM	96.88	42.96	–	72.91	–	19.94	–
+S-CARROT_A	96.56	77.33	80.00	76.04	4.29	20.07	15.72
+CLONEGEN	96.73	51.72	20.38	91.67	25.71	18.84	–0.05
+STRUCK	97.00	63.53	47.89	81.25	11.43	71.83	260.2
GGNN	96.54	48.28	–	73.91	–	11.05	–
+S-CARROT_A	95.85	78.72	63.03	73.91	0	22.62	104.6
+CLONEGEN	96.25	44.56	–0.08	90.21	22.06	19.04	72.23
+STRUCK	95.90	62.87	30.22	72.22	–0.02	78.26	608.0
GCB	97.90	69.34	–	29.00	–	20.32	–
+S-CARROT_A	97.69	91.55	32.03	36.00	24.13	8.84	–56.49
+CLONEGEN	97.86	63.27	–0.09	60.20	107.6	22.11	8.87
+STRUCK	97.91	83.51	20.43	31.00	6.90	86.87	336.3

5.4 RQ4: Robustness Assistance

Table 5 shows the robustness assessment of models on $\mathcal{D}_{\text{test}}$ using different attackers, with lower values indicating greater accuracy. The results demonstrate that STRUCK is more precise.

To explore whether adversarial training with STRUCK helps to improve the robustness of target models, we conduct experiments using LSTM, GGNN, and GCB as representative models. Our focus is on attackers that exploit code structural features. The adversarial training results are shown in Table 6. We can see that adversarial training with STRUCK effectively improves the robustness of target models against both STRUCK and S-CARROT$_{\text{A}}$. There is also some improvement in the robustness of LSTM and GCB against CLONEGEN. However, adversarial training with S-CARROT$_{\text{A}}$ or CLONEGEN has tiny effect on STRUCK. This shows that STRUCK is a more comprehensive method to attack using code structural features, and adversarial training with it can improve the robustness of models to structural attacks effectively.

Answers to RQ4: The target models' robustness can be assessed more accurately using STRUCK. And STRUCK adversarial training can be used to strengthen the robustness of models against structural attacks.

6 Related Work

DL models in code intelligence, particularly in the context of collaborative computing where multiple models or systems may interact, face robustness issues, prompting research to assess their robustness. We focus on the CRM target model. While works [26,36] show excellent performance in target models, such white-box attackers are less practical and not universal, as target models are typically unknown to attackers. We are more interested in black-box attack methods. MHM [37] and ALERT [34] are known SOTA black-box attack methods. They mainly modify user-defined variable names, with perturbations at the token level, ignoring structural features. They cannot generate different, higher-level perturbations and require multiple invocations to target models, lacking efficiency. S-CARROT$_{\text{A}}$ [36] and CLONEGEN [40] attempt adversarial attacks on CRMs using code structural features but have low RPD and imperceptibility.

There are a few more relevant studies for the robustness of CRMs available here. Rabin et al. [24] evaluate the generalizability of CRMs. Gao et al. [13] propose a framework based on counterfactual inference to mitigate naming bias in CRMs. Pour et al. [23] propose a testing framework that embeds source code into DL models. However, it has minimal impact on the performance of the target model. Works such as [25,27] perform poisoning attack on CRMs during the training phase by making changes to the training dataset.

7 Conclusion and Future Work

In this paper, we propose STRUCK, a novel method for attacking CRMs using code structural features. The objective of STRUCK is to advance the development of reliable DL models in the code intelligence field, and effectively evaluate the robustness of these DL models when applied to collaborative code development, automated code analysis, and other collaborative computing scenarios. To our best knowledge, STRUCK is the first work to attack using code structural features systematically and comprehensively. STRUCK utilizes global-level and local-level perturbations to generate adversarial examples, highlighting the idea of using code structural features to perform attacks. The experimental results show that STRUCK has higher effectiveness, efficiency and imperceptibility than the existing black-box attack methods in terms of sequential, structural, and pre-trained models. We also explore the value of STRUCK on the robustness of target models. STRUCK can more accurately assess the robustness of CRMs, and adversarial training using STRUCK can effectively improve the robustness of models against structural attacks. We have open sourced STRCUK at the GitHub repository https://github.com/zhanghaha1707/STRUCK to provide the support for further systematic robustness research of DL models for the field of code intelligence.

In the future, we will continue to improve STRUCK's imperceptibility and incorporate more perturbation methods using code structural features. And we aim to test the performance of STRUCK in more complex tasks and more datasets, expanding its applicability to a wider range of scenarios.

Acknowledgment. The research is supported by the National Key R&D Program of China under grant No. 2022YFF0902500, the Guangdong Basic and Applied Basic Research Foundation, China (No. 2023A1515011050). Liang Chen is the corresponding author.The research is supported by the National Key R&D Program of China under grant No. 2022YFF0902500, the Guangdong Basic and Applied Basic Research Foundation, China (No. 2023A1515011050). Liang Chen is the corresponding author.

References

1. Codechef (2022). https://codechef.com/
2. Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.W.: A transformer-based approach for source code summarization. arXiv preprint arXiv:2005.00653 (2020)
3. Akhtar, N., Mian, A., Kardan, N., Shah, M.: Advances in adversarial attacks and defenses in computer vision: a survey. IEEE Access **9**, 155161–155196 (2021)
4. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: ICLR 2018 - Conference Track Proceedings (2018)
5. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. arXiv: Learning (2018)
6. Cao, H., et al.: Prevention of gan-based privacy inferring attacks towards federated learning. In: Collaborative Computing: Networking, Applications and Worksharing (2022)

7. Carlini, N., Wagner, D.: Towards evaluating the robustness of neural networks. In: IEEE Symposium on Security and Privacy (2017)
8. Christlein, V., Riess, C., Jordan, J., Riess, C., Angelopoulou, E.: An evaluation of popular copy-move forgery detection approaches. IEEE Trans. Inf. Forensics Secur. **7**(6), 1841–1854 (2012)
9. Dai, H., Li, H., Tian, T., Huang, X., Wang, L., Zhu, J., Song, L.: Adversarial attack on graph structured data, pp. 1115–1124. PMLR (2018)
10. Dong, S., Wang, P., Abbas, K.: A survey on deep learning and its applications. Comput. Sci. Rev. **40**, 100379 (2021)
11. Feng, Z., et al.: Codebert: a pre-trained model for programming and natural languages. In: Empirical Methods in Natural Language Processing (2020)
12. Fernandes, P., Allamanis, M., Brockschmidt, M.: Structured neural summarization. In: ICLR (2019)
13. Gao, S., Gao, C., Wang, C., Sun, J., Lo, D.: Carbon: a counterfactual reasoning based framework for neural code comprehension debiasing (2022)
14. Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J.: Unixcoder: unified cross-modal pre-training for code representation. In: ACL 2022, Dublin, Ireland (2022)
15. Guo, D., et al.: Graphcodebert: pre-training code representations with data flow. In: Learning (2020)
16. Hellendoorn, V.J., Sutton, C., Singh, R., Maniatis, P., Bieber, D.: Global relational models of source code. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, 26–30 April 2020 (2020)
17. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, pp. 2073–2083 (2016)
18. Kipf, T., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv: Learning (2016)
19. Li, J., Peng, J., Chen, L., Zheng, Z., Liang, T., Ling, Q.: Spectral adversarial training for robust graph neural network. IEEE Trans. Knowl. Data Eng. **35**, 9240–9253 (2022)
20. Li, J., Xie, T., Chen, L., Xie, F., He, X., Zheng, Z.: Adversarial attack on large scale graph. IEEE Trans. Knowl. Data Eng. **35**(1), 82–95 (2021)
21. Li, S., Zheng, X., Zhang, X., Chen, X., Li, W.: Facial expression recognition based on deep spatio-temporal attention network. In: Collaborative Computing: Networking, Applications and Worksharing (2022)
22. Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z.: Convolutional neural networks over tree structures for programming language processing. In: National Conference on Artificial Intelligence (2014)
23. Pour, M., Li, Z., Ma, L., Hemmati, H.: A search-based testing framework for deep neural networks of source code embedding. In: ICST (2021)
24. Rabin, M.R.I., Bui, N.D.Q., Wang, K., Yu, Y., Jiang, L., Alipour, M.A.: On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. Inf. Softw. Technol. **135**, 106552 (2021)
25. Ramakrishnan, G., Albarghouthi, A.: Backdoors in neural models of source code. arXiv: Learning (2020)
26. Ramakrishnan, G., Henkel, J., Wang, Z., Albarghouthi, A., Jha, S., Reps, T.: Semantic robustness of models of source code. arXiv: Learning (2020)
27. Schuster, R., Song, C., Tromer, E., Shmatikov, V.: You autocomplete me: poisoning vulnerabilities in neural code completion. In: Usenix Security Symposium (2021)
28. Sun, X., Tong, M.: Hindom: a robust malicious domain detection system based on heterogeneous information network with transductive classification. ArXiv (2019)

29. Szegedy, C., et al.: Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013)
30. Vasic, M., Kanade, A., Maniatis, P., Bieber, D., Singh, R.: Neural program repair by jointly learning to localize and repair. arXiv: Learning (2019)
31. Wu, R., Zhang, Y., Peng, Q., Chen, L., Zheng, Z.: A survey of deep learning models for structural code understanding. arXiv preprint arXiv:2205.01293 (2022)
32. Wu, X.: Blackbox adversarial attacks and explanations for automatic speech recognition. In: ESEC/FSE 2022 (2022)
33. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. **32**(1), 4–24 (2020)
34. Yang, Z., Shi, J., He, J., Lo, D.: Natural attack for pre-trained models of code. In: ICSE 2022, New York, NY, USA (2022)
35. Yefet, N., Alon, U., Yahav, E.: Adversarial examples for models of code. In: Proceedings of the ACM on Programming Languages, vol. 4, no. OOPSLA, pp. 1–30 (2020)
36. Zhang, H., et al.: Towards robustness of deep program processing models-detection, estimation, and enhancement. TOSEM **31**(3), 1–40 (2022)
37. Zhang, H., Li, Z., Li, G., Ma, L., Liu, Y., Jin, Z.: Generating adversarial examples for holding robustness of source code processing models. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 1169–1176 (2020)
38. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X.: A novel neural source code representation based on abstract syntax tree. In: ICSE. IEEE (2019)
39. Zhang, W.E., Sheng, Q.Z., Alhazmi, A., Li, C.: Adversarial attacks on deep-learning models in natural language processing: a survey. ACM Trans. Intell. Syst. Technol. (TIST) **11**(3), 1–41 (2020)
40. Zhang, W., Guo, S., Zhang, H., Sui, Y., Xue, Y., Xu, Y.: Challenging machine learning-based clone detectors via semantic-preserving code transformations. IEEE Trans. Softw. Eng. **49**(5), 3052–3070 (2023)
41. Zhou, Y., Shi, D., Yang, H., Hu, H., Yang, S., Zhang, Y.: Deep reinforcement learning for multi-UAV exploration under energy constraints. In: Collaborative Computing: Networking, Applications and Worksharing (2022)