Towards Robustness of Deep Program Processing Models—Detection, Estimation, and Enhancement

HUANGZHAO ZHANG, ZHIYI FU, and GE LI, Peking University, China
LEI MA, University of Alberta, Canada
ZHEHAO ZHAO, HUA'AN YANG, and YIZHE SUN, Peking University, China
YANG LIU, Nanyang Technological University, Singapore
ZHI JIN, Peking University, China

Deep learning (DL) has recently been widely applied to diverse source code processing tasks in the software engineering (SE) community, which achieves competitive performance (e.g., accuracy). However, the robustness, which requires the model to produce consistent decisions given minorly perturbed code inputs, still lacks systematic investigation as an important quality indicator. This article initiates an early step and proposes a framework CARROT for robustness detection, measurement, and enhancement of DL models for source code processing. We first propose an optimization-based attack technique CARROT_A to generate valid adversarial source code examples effectively and efficiently. Based on this, we define the robustness metrics and propose robustness measurement toolkit CARROT_M, which employs the worst-case performance approximation under the allowable perturbations. We further propose to improve the robustness of the DL models by adversarial training (CARROT_T) with our proposed attack techniques. Our in-depth evaluations on three source code processing tasks (i.e., functionality classification, code clone detection, defect prediction) containing more than 3 million lines of code and the classic or SOTA DL models, including GRU, LSTM, ASTNN, LSCNN, TBCNN, CodeBERT, and CDLH, demonstrate the usefulness of our techniques for ❶ effective and efficient adversarial example detection, ❷ tight robustness estimation, and ❸ effective robustness enhancement.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Artificial intelligence;**

Additional Key Words and Phrases: Source code processing, big code, adversarial attack, robustness enhancement

This research is supported by the National Key R&D Program of China under Grant No. 2020AAA0109400, and the National Natural Science Foundation of China under Grant Nos. 62072007, 61832009, 61620106007. Lei Ma is supported by Canada CIFAR AI Program, NSERC Discovery Grant of Natural Sciences and Engineering Research Council of Canada, as well as JSPS KAKENHI Grant No. 20H04168 and JST-Mirai Program Grant No. JPMJMI20B8, Japan.
Authors' addresses: H. Zhang, Z. Fu, G. Li (corresponding author), Z. Zhao, H. Yang, Y. Sun, and Z. Jin (corresponding author), Peking University, China; emails: {zhang_hz, ypfzy, lige, zhaozhehao, chrisyoung, yizhe, zhijin}@pku.edu.cn; L. Ma, University of Alberta, Canada; email: ma.lei@acm.org; Y. Liu, Nanyang Technological University, Singapore; email: yangliu@ntu.edu.sg.

1049-331X/2022/04-ART50
https://doi.org/10.1145/3511887

ACM Reference format:
Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards Robustness of Deep Program Processing Models—Detection, Estimation, and Enhancement. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 50 (April 2022), 40 pages.
https://doi.org/10.1145/3511887

1 INTRODUCTION

Over the past decade, **deep learning (DL)** has made lots of progress and has achieved **state-of-the-art (SOTA)** performance for many applications across domains, e.g., image processing, speech recognition, **natural language processing (NLP)**, medical diagnosis. In the **software engineering (SE)** community, we have also witnessed a recent increasing trend of adopting DL for various source code processing tasks. Up to the present, DL could achieve SOTA performance in terms of accuracy or F1-score for many tasks of source code processing such as functionality classification [95], clone detection [85], method naming [2], code completion [53], and code comment generation [40]. Some of these techniques have further been developed as industrial solutions to accelerate software development productivity such as the code completion toolkits TabNine and aiXcoder.[1]

While achieving competitive performance in terms of accuracy or F1-score in the field of source code processing, an important quality indicator, the *robustness* of DL model, still lacks systematic investigations so far. A robust model is supposed to make consistent predictions when the input code snippet is slightly perturbed, and robustness is the metric to measure how robust a source code processing model is. Ideally, analysis at the program semantic level would be preferable, since most of the source code processing tasks tend to approximate the semantic meanings and relations of the code snippets, e.g., code snippet functionality classification and code snippet clone detection. However, existing DL techniques mostly work on code snippets that often have high flexibility and variety, making the model robustness even more important. For instance, in a malware detection system, such naturally occurring or intentional variety may deceive and bypass the DL detector, sending flawed code to downstream processing modules. Such non-robustness could be fatal to the whole system.

In practice, source code corpus naturally exhibits diversity during the software development process. For example, developers with different coding styles often implement the same task in different ways. Even for the same developer, she/he could program the same task differently under different situations (e.g., at the time in the middle of a week or on Friday shortly before the weekend). Similarly, the same algorithm or functionality might be implemented based on different libraries, leading to many variants of the code snippets. To adapt to source code diversity, ideally, a DL model, such as a malware detector, should behave consistently (i.e., robustly) upon the code variants with the same semantics. Otherwise, if a variant even with some minor perturbations (e.g., with some identifiers changed) causes the inconsistent prediction, then the DL model should not be deployed, potentially hindering its wide adoption in the real-world software development scenarios. Specifically, from the perspective of security, the adversarial attack makes intentional transformations upon the code to create the aforementioned variants (also known as adversarial examples) to bypass the DL model, even leading to possible failure of the entire system. In security-relevant tasks, such as malware detection or defect detection, adversary allows the attacker to directly bypass the DL detector by exploiting the carefully designed code variants, sending flawed code to downstream processing modules. It would cause fatal error to the whole system.

[1]See https://www.tabnine.com/ and https://www.aixcoder.com/en/#/ for details of TabNine and aiXcoder.

Robustness has been extensively studied over the past several years in classic DL application domains, e.g., image processing [34, 49, 67, 76], speech recognition [18, 70, 74], and **natural language processing (NLP)** [5, 26, 27, 44, 94]. As more and more DL solutions are proposed for tasks in the SE community, they also raise concerns that, besides high accuracy, to what extent the robustness of the current SOTA source code processing DL models are. In the previous work, **Metropolis-Hastings Modifier (MHM)** [93] made a preliminary study and spotted the non-robust issues in the DL source code classification models. The DL-based code functionality classifiers and code clone detectors can be misled to produce erroneous outputs given perturbed source code inputs, where the identifiers are iteratively randomly renamed in the manner of Metropolis-Hastings sampling. MHM takes the very first step to detect the non-robust issue, but to the best of our knowledge, it still lacks a general framework for systematic robustness analysis and enhancement of DL models in the context of source code processing.

In practice, adversarial attack is commonly used to estimate the robustness of a DL model under the worst-case perturbations, where a stronger adversarial attack technique gives more accurate approximation on the *true robustness*. In the source code context, the adversarial attack and robustness analysis become more challenging due to the multiple unique constraints and requirements. Existing robustness analysis on image or speech voice cases are mostly based on distances in Euclidean space. The robustness can be defined as the largest norm ball (i.e., L_p norm) such that all inputs after perturbations still reside inside the norm ball without changing the prediction result of a DL model. On contrary, the source code is discrete and strictly follows the formal grammar (e.g., context-free grammar), which makes the robustness measurement difficult to be defined by distances in the Euclidean space. Only code snippets that follow the grammar are valid and can be used for robustness analysis and estimation.

To bridge this gap, in this article, we propose a general framework **CARROT** (**C**ode **A**dve**R**sarial-attack-based **RO**bustness measurement and **T**raining) for DL model robustness detection, measurement, and enhancement in the context of source code processing, which are the essentials of many software engineering tasks in the era of big code [1]. We first propose an effective and efficient adversarial attack technique CARROT_A specially designed for DL of source code processing with gradient guidance. Based on this, we propose robustness metrics and the measurement toolkit CARROT_M for robustness approximation. Furthermore, we propose the robustness enhancement technique CARROT_T based on adversarial training.

To approximate the robustness and address the unique characteristics of source code processing in DL, the design of our adversarial attack CARROT_A takes the following properties into consideration: ❶ ***Grammar validity***. The generated code (adversarial example) must be compilable (i.e., grammar-valid) and preserves the semantics of the original code snippet. ❷ ***Generation Effectiveness***. The attack should be strong to enable the robustness issue detection in the huge space under complicated constraints. ❸ ***Generation efficiency***. The generation process should be efficient to be feasible in practice. ❹ ***Diversity***. The attack algorithm should identify diverse potential code segment modifications and be able to perform perturbations at multiple levels.

To be specific, CARROT_A does not constrain the perturbations with L_p norms as used in image processing.[2] Instead, it applies rule-based constraints embedded within the attacking techniques, where the robustness is assessed by the performance of the model under a set of perturbation (i.e., code transformation) patterns. In such a case, the more effective attacker helps to obtain the more accurate robustness estimation. Although the previous proposed MHM [93] is capable to produce

[2] Adversarial attack is widely used for robustness estimation. However, unlike continuous image space, where L_p norms can be adopted to constrain the perturbations, the source code space is discrete and non-Euclidean, which makes the Euclidean L_p norms unsuitable.

grammar valid adversarial examples, it lacks diversity and only performs identifier-level perturbations. In addition, it also lacks efficiency, the renaming process of which is iteratively uniformly sampled, with a large computational overhead, where the gradient information is not fully leveraged either. To further address these challenges, we design CARROT$_A$ in a way that any grammar valid synonymous transformation is feasible, and the gradient information is incorporated to guide the searching process for adversarial examples. With the proposed adversarial attack method, it enables to make accurate quantitative estimation and effective enhancement of the DL model robustness for source code. Based on this, we propose robustness metrics CARROT$_M$ to estimate the robustness bound, and toolkit CARROT$_T$ for robustness enhancement.

To demonstrate the usefulness of our proposed framework, we perform in-depth evaluations on three typical source code processing tasks (i.e., functionality classification, clone detection, defect prediction) containing a total of more than 3 million lines of code, and seven classic or SOTA DL models, including GRU, LSTM, ASTNN [95], LSCNN [42], TBCNN [62], CodeBERT [28], and CDLH [85] for each task. The results demonstrate that: ❶ the DL models with high accuracy or F1-score might not be robust, as the simple random baseline approach (RW) can easily reduce the performance of the evaluated models by more than 40%. ❷ CARROT$_A$ generates adversarial examples effectively, which on average reduces the performance of the evaluated models by 87.2% on token-level, outperforming the SOTA MHM method (75.5%). In particular, for the defect prediction task, CARROT$_A$ even reduces the performance (i.e., F1-score) of GRU, LSTM, ASTNN, and LSCNN to almost zero. ❸ CARROT$_A$ generates adversarial examples efficiently, with an average time of 2.4 seconds and invocation number of 192 to successfully generate an adversarial example, outperforming the MHM that takes 5.8 seconds and 457 invocations. ❹ CARROT$_M$ also enables to estimate the robustness more accurately with a tighter bound. The robustness bounds achieved by CARROT$_M$ are much tighter than RW and MHM baselines. In particular, for the defect prediction task, CARROT$_M$ confines the robustness bound of GRU, LSTM, ASTNN, and LSCNN to nearly zero. ❺ CARROT$_T$ is useful for robustness enhancement of DL models, as the robustness increases to 5.3 times on average compared with the original models, which is 1.7 times higher than the MHM methods. Specifically, in functionality classification, the robustness of LSTM after CARROT$_T$ is even improved to 13.8 times.

Overall, the main contributions of this article are summarized as follows:

- We propose an adversarial attack technique CARROT$_A$ for the DL models in the context of source code processing. *To be efficient and effective*, the attack technique adopts the optimization-based technique that uses gradient information for source code perturbation. *To be diverse*, the attack is designed to work at different perturbation levels and locations.
- Based on our attack method, we further propose robustness metrics and method CARROT$_M$, which enables more accurate quantitative approximation on the robustness of DL models.
- After detecting adversarial examples and measuring the robustness, we further propose the adversarial retraining method CARROT$_T$ to enhance the robustness of the DL models.
- These proposed techniques are integrated as a general framework CARROT for systematic robustness detection, measurement, and enhancement of DL models for source code processing. Our in-depth evaluations on three typical source code processing tasks and multiple classic or SOTA DL models demonstrate its effectiveness and usefulness.

As more and more DL models are used for source code domains in the SE community, the robustness of the DL models can be a big concern, which directly impacts its generality and effectiveness under diverse scenarios. Until this far, however, the robustness issue for source code has not received enough attention with limited progress made by existing work. The results of our article find that the current state-of-the-art DL models in the context of source code may not perform so

well, as it seems in terms of robustness, although achieving rather high accuracy, F1 score, and so on. As an important quality indicator, we call for the attention of SE researchers and practitioners to take the robustness into consideration during designing and evaluating new DL techniques for source code. Our proposed framework CARROT is open-sourced and publicly available[3] to provide the support for further systematic robustness research of DL models for SE researchers in the era of big code [1].

2 RELATED WORK

In this section, we will discuss the most relevant works to this article, including DL for source code processing tasks, general purpose adversarial learning, formal verification of the DL models, DL model testing, traditional fuzzing testing, traditional mutation testing, and program obfuscation.

2.1 Source Code Processing by DL

Quite a lot of progress has been recently made on automated source code processing by DL techniques, which could be largely categorized as classification tasks and generation tasks. In this section, we only discuss the most relevant DL-based works to this article. More comprehensive progress about the field can be referred to the recent survey on big code processing [1].

The DL model in a classification task makes classification based on a transformed parameterized representation of the source code. Mou et al. [62] propose the tree-based TBCNN for functionality classification. Huo and Li [42] propose the structural LSCNN for defect prediction. Wei and Li [85] propose CDLH for code clone detection. Gupta et al. [36] propose NeuralBugLocator for bug localization. Wang et al. [82] propose MatchGNet for malware detection. These approaches are classic solutions to DL for source code processing, and they are the backbones and the basis of the current SOTA models. More recently, Alon et al. [4] propose AST-path-based Code2Vec for program encoding. Zhang et al. [95] propose the AST-based ASTNN for source code representation and verify its capacity in multiple downstream classification tasks. Wang et al. [83] employ FA-AST and **graph neural networks (GNN)** for code clone detection. Feng et al. [28] employ the transformer-based architecture and propose the pre-trained CodeBERT for code representation in general purpose. The listed models are among the SOTA solutions to DL for classification tasks in SE.

A DL model for generation task takes code snippets or natural language descriptions as input, but outputs a sequence of information, e.g., code snippets, comment texts, method names, and so on. Gu et al. [35] propose DeepAPI for API sequence generation. Allamanis et al. [2] propose convolutional attention networks for method naming. Li et al. [53] propose pointer mixture networks for code completion. Hellendoorn et al. [39] propose DeepTyper for type inference. Hu et al. [40] propose DeepCom for code comment generation. Alon et al. [4] and Alon et al. [3] propose code2vec and code2seq for code encoding and generation. Generation tasks are often based on code classification, as the generation process can be seen as a sequence of classifications. For example, at each timestep, a typical generation model makes a classification as the current output, based on the input (e.g., token sequence, abstract syntax tree) and the previous outputs.

In this article, as an early step to construct the automated framework and study robustness in source code processing, we mostly focus on DL models for source code classification tasks, i.e., functionality classification, code clone detection, and code defect prediction. Also, we select our subject DL models based on generality and performance. Therefore, we choose seven classic or SOTA models, including the classic sequential GRU and LSTM, the SOTA AST-based ASTNN, the classic structural LSCNN, the classic tree-based TBCNN, the transformer-based pre-trained CodeBERT, and the classic task-oriented CDLH (specially designed for code clone detection). The

[3]We have open sourced this project at the GitHub repository https://github.com/SEKE-Adversary/CARROT.

required data format covers token sequence (GRU, LSTM, and CodeBERT), statement structure, (LSCNN) and AST (ASTNN, TBCNN, and CDLH). The architecture covers recurrent (GRU, LSTM, and ASTNN), recursive (CDLH), convolutional (LSCNN and TBCNN), and transformer (CodeBERT) neural networks. The design of the models covers task-oriention (CDLH), program representation (GRU, LSTM, ASTNN, LSCNN, TBCNN), and pre-training for general purpose (CodeBERT).

2.2 Adversarial Learning

Adversarial learning, including adversarial example, adversarial attack, and adversarial training, draws much more attention regarding the DL quality (e.g., reliability, security, and safety). The adversarial vulnerability of machine learning models has been revealed years ago. For instance, against spam mail filtration models, spam messages bypass the model by misspelling the bad words or inserting the good words [9, 13]. Adversarial machine learning involves mainly three strategies: evasion [8, 10, 63], which is similar to the concept of adversarial attack; poisoning [8, 10, 11, 47], which injects malicious examples into the training data to disrupt model (re)training; and model stealing [16, 84], which reconstructs a black-box model and steals the training data. In this article, we focus on the evasion attack. More comprehensive progress about adversarial machine learning can be referred to the survey [12].

As DL has begun to dominate the field of artificial intelligence, the vulnerability and robustness of DL models become research hotspots. The vulnerability of DL models was first discovered in image classification tasks [76]. Later, the gradient-based and optimization-based perturbations were intensively studied in the field of image processing, such as FGSM [34], BIM [49], and JSMA [67]. However, these techniques are not suitable for DL models of source code processing due to two major challenges. ❶ There are no direct norm ball constraints in code space, as the L_p norm between two separate tokens can hardly be defined in discrete code space. Although, an alternative way is to compute the L_p norm in the embedding space, where the discrete tokens are mapped to continuous vectors. This leads to the other challenge. ❷ Jumping along the projection of gradient from one token is very likely to result in an invalid token. In the embedding space, which is often high-dimensional, a small jump from a token vector may lead to a vector that does not correspond to any valid token. Therefore, techniques such as FGSM are not quite suitable for DL in source code processing. It is not until recently, the adversarial examples are discovered in sequence signal processing tasks, such as speech recognition [17, 70, 74] and NLP [5, 26, 27, 44, 94]. Still, the existing SOTA attack techniques are not suitable for source code either, because the strict constraints formulated by lexical, syntactical, and grammatical rules cannot be easily ensured. Although the recent proposed MHM [93] is capable to discover valid identifier-level perturbations for source code processing, which iteratively performs random renaming of identifiers and leverages reject sampling in the manner of **Metropolis-Hastings (M-H)** approach [22, 38, 61], as discussed earlier, MHM cannot produce diverse and higher-level perturbations, such as statement perturbations, and lack of efficiency, due to the computational overhead of M-H and the neglection of gradient guidance.

Different from existing attack techniques, the proposed CARROT_A applies semantic equivalent transformation, which is grammar-valid, semantic-preserving, and extensible. Furthermore, we incorporate gradient information into CARROT_A to guide the effective adversarial attack process. CARROT_A is not only able to generate adversarial perturbations with higher quality, but also provides more accurate robustness estimation (CARROT_M) and more effective robustness enhancement (CARROT_T).

2.3 Verification and Testing of DL Models

Formal verification of DL models, achieved by solving a minimax problem (defined later by Equation (6)), provides a formal guarantee on the robustness of DL models. **Mixed-integer**

linear programming (MILP) solves the minimax problem directly with high computational cost [15, 21, 56]. Other researchers transform it into an SAT problem and solve it utilizing **Satisfiability Modulo Theory (SMT)** solvers [45]. However, these verification techniques are at an experimental phase, and they can hardly scale to deep and large sequential models in the real world, i.e., they are not quite practical. Incomplete methods are also proposed by solving a convex relaxation of the verification problem [86, 88] or computing an approximate upper bound using bound propagation [41]. However, bound propagation accumulates error as the model grows deep and at last produces meaningless loose boundary. In particular, the recently proposed POPQORN [48] is designed for RNN models, but it works only on small models in short-sequence dataset, such as serialized MNIST. The scalability is still not satisfying yet.

Testing techniques are also proposed to detect the defects (incorrect behaviors) of DL models in the context of image processing. Most approaches are based on fuzzing testing, which automatically generates inputs to try to trigger, monitor, and detect exceptions of the target application [54]. We introduce and discuss the traditional fuzzing techniques later in Section 2.4. In the field of image processing, DeepXplore [68] performs differential testing to detect the inconsistencies of DL models for the same task. TensorFuzz [65] develops coverage-guided fuzzing to detect errors in the model. DeepTest [78] and DeepHunter [91] generate new tests based on basic image transformation with coverage feedback. In the safety-critical scenario of automated driving, DeepRoad [97] generates tests by using GAN that performs more advanced scene transformation. There is a major difference between fuzzing and our proposed CARROT in this article—fuzzing focuses on error detection within the target, from the perspective of testing, while CARROT detects, estimates, and enhances the robustness as an intact framework from the perspective of adversary. Still, it is quite a reasonable research direction of employing fuzzing to test DL models. However, as an early step to analysis robustness of DL for SE, it is not our major goal of this article, and we leave it for future exploration. Meanwhile, test criteria are also proposed to measure the testing sufficiency, such as neuron coverage in DeepXplore [68], major functional coverage and corner case coverage in DeepGauge [60], MC/DC coverage in DeepConcolic [75], and surprise adequacy [46]. DeepReduce [98] selects representative examples from the whole test set to efficiently test the DL model. More comprehensive discussion on DL verification and testing could be referred to the recent surveys [90, 96]. In our proposed CARROT_M, we adopt the testing-like approach to estimate the robustness of DL models for source code process. To be more specific, CARROT_M tests against the DL model with multiple adversarial attack approaches to verify whether the decisions from the DL model are consistent.

Existing works of DL verification and testing are mostly carried out in the classic DL application scenarios (e.g., image, speech). This article focuses on source code processing, which is different and more challenging due to the discrete input space and rigid constraints caused by compilation rules of programming languages. As there is an important potential direction that leverages DL for diverse SE tasks, we keep this article relevant to the SE community. We also make a very early step to systematically study the robustness of DL models in typical SE tasks to understand the current SOTA models and try to find a possible direction for future improvement.

2.4 Fuzzing, Mutation Testing, and Program Obfuscation

Fuzzing is an automated software testing technique to discover vulnerabilities in the software [54]. It generates massive normal and abnormal test cases against the target software and tries to trigger, monitor, and detect the exceptions such as crash or memory leak. Generation-based fuzzing generates test cases directly according a pre-defined configuration file, where the format of the test cases is provided. There are mature open-sourced toolkits and frameworks for generation-based fuzzing, including Spike [87], Sulley [30], and its successor BooFuzz [29]. Another mutation-based fuzzing

generates test cases with some mutation rules upon a set of valid seed inputs [66]. Although the working procedure of mutation-based fuzzing seems similar with the code transformations in this article, as they both manipulate the inputs (test cases for fuzzing and code snippets in this article) to probe the target (software applications for fuzzing and DL models in this article), the purposes are quite different. Traditional fuzzing employs heuristic or random transformations to produce massive test cases with high ratio of valid formats [33, 71] to detect plausible exceptions in the target software; while in this article, CARROT aims to not only detect the non-robust issue, but also estimate the worst-case robustness and further enhance the adversarial resilience. Still, adopting the idea of fuzzing testing in DL is a reasonable research direction. There are many testing approaches for DL employing fuzzing testing proposed in recent years, as we have discussed in Section 2.2. More comprehensive progress about fuzzing can be referred to the survey cited in Reference [54].

Mutation testing is a straightforward but powerful technique to evaluate the quality of software tests [25, 37]. It generates mutants from the original code by modifying the code in small ways, and the quality of software tests is measured by the percentage of mutants that they kill (detect) [31, 32]. Although mutators for mutation testing seems similar with code transformation in this article, as they all perform slight modifications upon the code, the semantic equivalence property is totally different. The transformation operators in this article are supposed to modify the code without changing the compilation or execution results; while mutators for mutation testing alters the semantic meaning, such as replacement of Boolean subexpressions with true or false, replacement of arithmetic operations with others (e.g., "+"⇔"-"), replacement of Boolean relations with others (e.g., "<"⇔"<= "), removing method body [64], and implemention in PITest [80]. Therefore, traditional mutation testing is quite different from our proposed method, but for simplicity, we borrow the term "mutation" in this article. More comprehensive progress about mutation testing can be referred to the survey cited in Reference [20].

However, **equivalence modulo input (EMI)** fits the definition of semantic equivalent code modification, which generates semantic equivalent code snippets to test compilers [50, 51, 55, 73]. The idea of EMI is to directly delete or insert dead code to create semantic-preserving mutations. We absorb this idea into CARROT_A, forming a dead code statement insertion/deletion attack, namely, S-CARROT_A.

Another relevant technique to this article is program obfuscation, which is a pivotal technique to protect software intellectual property [92]. General obfuscation buries the useful code (or bytecode) in the redundant logic or transforms it into a less comprehensible version to avoid code theft and to protect the intellectual property of the developers. The transformation operators for obfuscation are semantically equivalent, but they usually change the code greatly from the perspective of programmers. Classic work of obfuscation includes identifier scrambling [19], program item reordering [58, 89], bogus control flow injection [6, 23, 72, 99], bytecode anti-disassembication [24, 57, 69], and so on. Program obfuscation has also been deployed in some software development platforms, such as ProGuard and DexGuard for Android.[4] More advanced and comprehensive progress about program obfuscation can be found in the survey cited in Reference [92].

There are two major differences between program obfuscation and our work. ❶ Obfuscation transforms source code, bytecode, or intermediate code according to the concrete scenario, while our proposed CARROT focuses on source code processing and manipulates source code only. ❷ Obfuscation aims to transform the program greatly to make it hard to understand, while CARROT is expected to minorly perturb the code to mislead the DL model. Nevertheless, from the perspective of program obfuscation, we draw the classic ideas of identifier scrambling and bogus injection, creating the identifier renaming I-CARROT and the dead code inserting S-CARROT.

[4] https://www.guardsquare.com.

Adversary of DL for source code processing may also leverage other existing transformations from obfuscation, yet in this article, as an early step, we mainly study the aforementioned transformations, leaving others for future explorations. In addition, we must emphasize that CARROT treats the transformations as internal modules and is likely to be compatible with the existing obfuscation work and tools. Similarly, we leave these analyses and evaluations for our future work.

3 BACKGROUND AND OVERVIEW

In this section, we first provide the formulation for the general definition of source code classification tasks. Then, we define the notations used in this article and briefly introduce the general concepts of adversarial learning and robustness.

3.1 Source Code Classification

Source code classification. As the basis of DL for source code processing, we provide the general formulation of source code classification. A classification model is supposed to take source code snippets as inputs and predicts the classification labels. A typical well-labeled dataset $\mathcal{D} = (\mathcal{X}, \mathcal{Y})$ consists of a set of code snippets ($\mathcal{X}$), where each $x_i \in \mathcal{X}$ is a code snippet in the form of character sequences, token sequences, **abstract parsing trees (ASTs), control flow graphs (CFGs), data flow graphs (DFGs)**, and so on, according to the model requirements for input format, and a set of ground-truth labels ($\mathcal{Y}$), in which each $y_i \in \mathcal{Y}$ is a label corresponding to x_i. In the rest of this article, we denote the code-label pair with (x, y).

Classification. The classifier C encodes the input code snippet $x \in \mathcal{X}$, extracting the feature vector $\mathcal{F}(x) = \{f_1(x), f_2(x), \ldots, f_{n_f}(x)\}$ utilizing neural networks. (Some neural networks may produce feature matrices or tensors, which can be extended from $\mathcal{F}$ defined in this article.) Then $\mathcal{F}(x)$ is fed into the classification layer to obtain the predicted probability of each class. The classification layer consists of a linear transformation layer ($W_S\mathcal{F}(x) + b_S$), which maps the dimension of $\mathcal{F}(x)$ (n_f) to the number of classes (n_c), and a softmax activation, which produces the normalized predicted probabilities. At last, the model selects the class with the highest probability as the final prediction, using an argmax function. The overall classification process can be deducted as:

$$C(x) = \arg\max_i \frac{e^{(W_S\mathcal{F}(x)+b_S)_i}}{\sum_{j=1}^{n_c} e^{(W_S\mathcal{F}(x)+b_S)_j}}, \tag{1}$$

where e is the natural constant with the value of about 2.718 employed in the softmax activation, n_c is the number of classes, $W_S \in \mathcal{R}^{n_c \times n_f}$ and $b_S \in \mathcal{R}^{n_c}$ are parameters of the classification layer, and $(\cdot)_i$ means to take the ith element from the vector. Note that the DL models are designed to leverage its network layers for feature extraction $\mathcal{F}(x)$.

3.2 Symbol Notations and Important Definitions

Table 1 summarizes the notations and symbols used in this article, besides which we further highlight several important definitions below. These definitions will appear frequently in the rest of this article, therefore, we present these important definitions here to impress the readers.

Training objective. The parameters of the model C, denoted as Θ_C, are obtained by optimizing the instance-level loss function $L(y, C(x))$ over the entire training set $\mathcal{D}^{(t)}$, i.e., $\min_{\Theta_C} \sum_{x,y \in \mathcal{D}^{(t)}} L(y, C(x))$.

Table 1. Summary of Notations and Symbols in This Article

Scope	Notation	Definition
DL model	(x, y)	A pair of input-output examples, i.e., the code and its label.
	$\mathcal{D}^{(t)}$, $\mathcal{D}^{(v)}$, $\mathcal{D}^{(e)}$	The datasets for training, validation, and testing.
	$\mathcal{F}(x)$	The encoding function producing the feature vector of x.
	C	The DL model for source code processing.
	Θ_C	The trainable parameters in C.
	$L(y, C(x))$	The instance-level loss function.
Code validity	$\mathcal{E}$	The full set of compilable code snippets.
	$E(x, i)$	The execution results of snippet x given a valid input i.
	$\mathcal{A}(x, y; C)$	The set of adversarial examples of (x, y) against C.
	$\mathcal{X}(x, x')$	The indicator function of semantic equivalency of snippets x and x'.
	$T(x)$	The set of mutations of x.
Adversary	A	The adversarial attack toolkit.
	$R(C)$	The true robustness of C.
	$\tilde{R}_A(C)$	The estimated robustness of C by adversarial toolkit A.

Validity of source code. Code snippet x is valid if and only if it satisfies all lexical, syntactical and grammatical rules, i.e., $x \in \mathcal{E}$, where $\mathcal{E}$ is the full set of all compilable snippets. To be specific, a valid code is error-free during compilation and can be further executed.

Semantic equivalency of source code. In general, the definition of semantic equivalency is often very complicated. The semantic equivalence analysis of code snippets is even undecidable. To facilitate the readers' understanding, we adopt an incomplete approximation by checking the consistency of the execution results. More specifically, given a pair of valid code snippets x_1, x_2, we analyze whether they reach consistent execution results under the same input test case set with a sufficient capacity. Another approximation to determine the semantic equivalency is to analyze whether one code snippet can be transformed to the other under a set of well-designed equivalent code transformation rules. In this article, we adopt the second approximation with equivalent code transformations.

3.3 Adversary

Model robustness on a single example. The outputs of a robust classification model should remain the same after some minor perturbations upon the input. We employ the binary-valued function $R(C|x, y)$ to indicate whether the model C is robust on example pair (x, y). $R(C|x, y) = 1$ when C is guaranteed to be robust on x, if and only if the outputs are consistent, as:

$$C(x') = C(x), \text{ s.t. } \|x' - x\|_p \leq \delta, \tag{2}$$

otherwise, the outputs are inconsistent, and $R(C|x, y) = 0$. This definition forces the model to produce consistent outputs under perturbations when we aim for $R(C|x, y) = 1$.

Adversarial examples. In general, adversarial examples are generated from an input that can be correctly handled by a DL model (i.e., $C(x) = y$) with some minor perturbations. An adversarial example is very similar to its original counterpart from the human's perspective but is incorrectly handled to the inconsistent results with its original counterpart by the model. The set of adversarial examples $\mathcal{A}$ can be defined as [14, 17]:

$$\mathcal{A}(x, y; C) = \{\hat{x} | C(\hat{x}) \neq C(x) = y \wedge \|\hat{x} - x\|_p \leq \delta\}, \tag{3}$$

where x and $\hat{x}$ are the original example and the adversarial example, respectively. Equation (3) defines the untargeted adversarial examples, which mislead the victim DL model to an erroneous prediction $\hat{y}$ other than the ground-truth y ($\hat{y} \neq y$).[5] Note that $C(x) = y$ in the first constraint is not

[5]As for targeted adversarial examples, the DL model is erroneously guided to a pre-determined $\hat{y}$. The definition becomes $\mathcal{A}(x, y, \hat{y}; C) = \{\hat{x} | C(\hat{y}) = \hat{y} \neq C(x) = y \wedge \|\hat{x} - x\|_p \leq \delta\}$.

necessary in terms of adversary. However, we prefer to study those correctly handled examples—if an example x is wrongly predicted by the model originally, analyzing perturbations and robustness upon such examples is not quite meaningful. Therefore, we add this constraint in this article. The first constraint ($C(\hat{x}) \neq C(x) = y$) indicates that $\hat{x}$ should mislead C to an erroneous output, and the second constraint ($\|\hat{x} - x\|_p \leq \delta$) limits the perturbation within the allowable L_p distance δ.

Adversarial attack. Adversarial attack is the process to generate adversarial examples from the original inputs. A commonly adopted idea is to transform adversarial attack into an optimization problem such as CW method [17]. FGSM [34] and BIM [49] further assume the allowable perturbations are within a linear L_∞ norm ball and jump along the direction of the gradient projection. In this article, we focus on untargeted attack, where any $\hat{x}$ causing $C(\hat{x}) \neq y$ is feasible, and the optimization objective can be defined as:

$$\max_{\hat{x}} L(y, C(\hat{x})), \text{ s.t. } \|\hat{x} - x\|_p \leq \delta. \tag{4}$$

$L(y, C(\hat{x}))$ shows the impact of the perturbation. By maximizing $L(y, C(\hat{x}))$, we are able to eventually mislead C if $R(C|x, y) = 0$. In practice, the optimization process terminates when an adversarial example is found, without having to exactly reach the global optima. Therefore, the generated adversarial attack often finds a positive lower bound of the objective. Different attack methods may find different bounds, and the less-effective attack may miss the adversarial examples.

Robustness analysis. We employ the proportion of examples in the dataset, on which the model C is robust, as the metrics of robustness. The robustness of C on a set of data $\mathcal{D}$ is defined as:

$$R(C|\mathcal{D}) = \frac{1}{\|\mathcal{D}\|} \sum_{(x,y)\in\mathcal{D}} R(C|x, y). \tag{5}$$

$R(C|x, y) = 1$ suggests that Equation (4) is negative and $R(C|x, y) = 0$ suggests that Equation (4) is positive. Therefore, we can determine the robustness of a model given a pair of example by computing the sign of Equation (4).

Directly solving Equation (4) with linear programming tools (e.g., MILP [15, 21, 56]) or SAT solvers [45] can be unscalable and impractical for large-scale models, due to expensive computational cost. Bound propagation [41], which computes the bounds layer-by-layer to estimate Equation (4), however, accumulates error during propagation and the estimated bounds become rather loose, as the models are deep. At last, adversarial attack estimates the lower bound of Equation (4) and is more scalable. Adversarial attack gives the guaranteed non-robust proportion, while the rest are not guaranteed to be robust.

Adversarial training. Adversarial training is an approach to improve the robustness of DL models by optimizing the model parameters under the worst-case perturbations, which can be formulated as a minimax problem defined as:

$$\min_{\Theta_C} \sum_{x,y\in\mathcal{D}^{(t)}} \max_{\hat{x}} L(y, C(\hat{x})), \text{ s.t. } \|\hat{x} - x\|_p \leq \delta. \tag{6}$$

Intuitively, if C performs well under the worst-case perturbations found by the inner maximization, then it is able to perform well under other perturbations, and therefore C becomes more robust.

3.4 Overview of CARROT Framework

Figure 1 shows the overview of our proposed CARROT framework and its three major components. The adversarial attack component CARROT$_\text{A}$ (the upper part of Figure 1) forms the basis

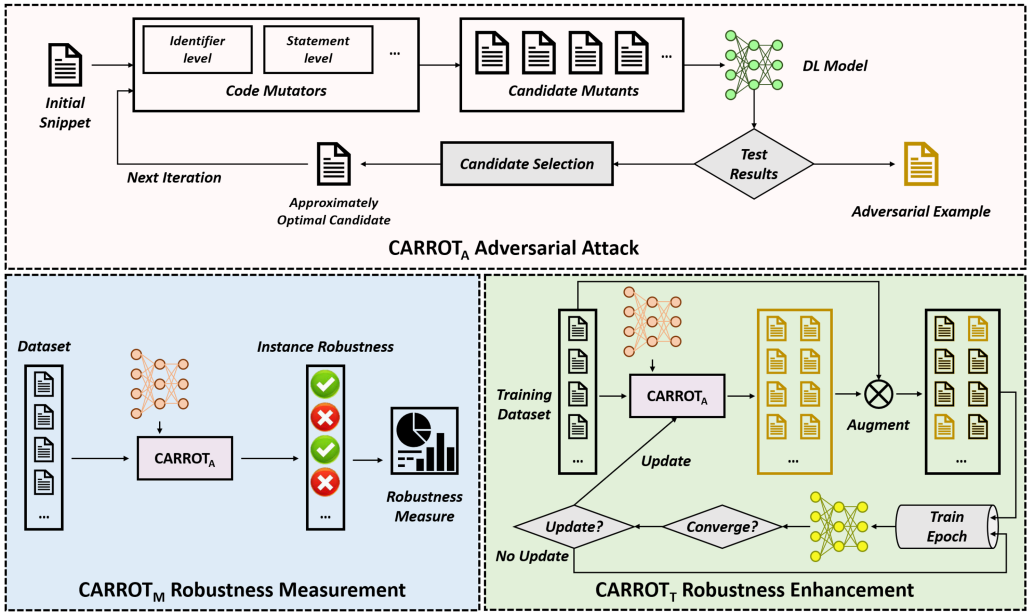


Fig. 1. Overview of CARROT framework.

of other components for further analysis. Given a DL model and a code snippet (that the model classifies correctly), CARROT$_A$ iteratively attacks the code snippet through semantically equivalent code mutators (see Table 2 for examples) until an adversarial example (i.e., valid code snippet) is obtained. In practice, the possible transformation searching space can be huge. Therefore, to be effective, CARROT$_A$ mutates the code at multiple levels and generates possible candidates under different candidate program locations; to be efficient, CARROT$_A$ incorporates gradient information to guide the searching process where possible. Then, we design a robustness measurement component CARROT$_M$ (the lower left part of Figure 1) based on our proposed robustness metrics and pluggable adversarial attack toolkit. Besides our proposed attack techniques, the inclusion of multiple adversarial attack algorithms enables more diverse adversarial attack searching towards obtaining more accurate robustness estimation, i.e., with a tighter upper bound $\tilde{R}$ of the true robustness R. To further improve the robustness, the third component CARROT$_T$ (the lower right part of Figure 1) is proposed to perform adversarial training. During training, we periodically augment the training set with adversarial examples obtained by adversarial attacks. This process is an approximation to solve the minimax problem in Equation (6). Similar to CARROT$_M$, CARROT$_T$ is designed to include an adversarial attack toolkit as well to generate diverse perturbations. Sections 5.2, and 5.3 elaborate these components in details, respectively.

4 CARROT ADVERSARIAL ATTACK

4.1 Adversarial Attack in Source Code Context

Perturbation constraint. Different from the classic DL application domains (e.g., image, speech processing), adversary in source code processing cannot be constrained by L_p norm distances, due to the discrete nature of source code space. Although **edit distance (ED)** is often used to limit the perturbations in NLP, a small perturbation without altering the semantics for source code can cause big changes in the source code sequence (e.g., variable renaming). Therefore, minor perturbations upon the source code may lead to large EDs, making ED not suitable in the context of source

Table 2. Mutation Operations Applied in CARROT$_A$

Type	Operation	Definition	Example
Token	Var renaming	Renaming user-defined global or local variables.	"int a;" ⇔ "int i;"
	Data type renaming	Renaming user-defined types.	"struct a {}" ⇔ "struct s {}"
	Func renaming	Renaming user-defined functions or methods.	"void func() ;" ⇔ "void foo() ;"
Stmt	Empty stmt ins/del	Inserting or deleting empty statements.	"a = b + 1;" ⇔ "a = b + 1; ;"
	Branch ins/del	Inserting or deleting branches never being true.	"a++; c++;" ⇔ "a++; if (false); c++;"
	Loop ins/del	Inserting or deleting loops with false conditions.	"i += j; j++;" ⇔ "i += j; while(0); j++"

code. Note that "small" and "minor" here do not refer to the distance in the feature space (e.g., embedding), instead, they refer to semantic equivalence of the code snippets. In the image field, where adversary is originally proposed and studied [34, 76], it is natural to constrain the distance (e.g., Euclidean distance) of the images to keep the semantics. We inherit "small" and "minor" to describe the perturbations in this article, referring to semantic equivalence of the perturbed code snippet. In this article, we adopt the rule-based constraints to define the valid adversarial perturbation space following the previous work [93]:

$$\mathcal{X}(x', x) = \begin{cases} 1, & \text{If } x' \in \mathcal{E} \wedge \forall i \in \{\text{valid inputs}\}.\mathrm{E}(\mathrm{x}', \mathrm{i}) = \mathrm{E}(\mathrm{x}, \mathrm{i}) \\ 0, & \text{Otherwise.} \end{cases} \tag{7}$$

Intuitively, $\mathcal{X}(x', x) = 1$ requires the generated examples to be valid and semantically equivalent as its original counterpart. Unlike perturbation constraint in image processing or NLP, which comes from the perspective of human beings, in source code processing, Equation (7) requires the perturbation to be imperceptible and indistinguishable for compilers and executors.

Adversarial example. With the rule-based perturbation constraints (Equation (7)), we define the adversarial example for source code as:

$$\mathcal{A}(x, y; C) = \{\hat{x} | C(\hat{x}) \neq C(x) = y \wedge \mathcal{X}(\hat{x}, x) = 1\}. \tag{8}$$

An important constraint in Equations (7) and (8) is that the adversarial example (code snippet) preserves the semantics. However, proving the semantic equivalence of two code snippets is theoretically undecidable. It is often not easy either to check their concrete execution results on all the possible inputs, which is often quite huge or even infinite to be practical. Therefore, we approximate $\mathcal{X}$ with a set of mutation operators T (i.e., code transformation rules; see Table 2), where $x' = T(x)$ is guaranteed to satisfy $\mathcal{X}(x', x) = 1$. Multiple transformations of x for k times, denoted as $T^k(x)$, still produce valid and equivalent snippets. Then, the code transformation-based definition of adversarial examples is:

$$\mathcal{A}(x, y; C) = \{\hat{x} | C(\hat{x}) \neq C(x) = y \wedge \hat{x} \in T^k(x)\}. \tag{9}$$

Adversarial attack. After introducing the rule-based constraints $\mathcal{X}$ and the mutation approximation T into Equation (4), the objective of adversarial attack in source code processing can be defined as:

$$\max_{\hat{x}} L(y, C(\hat{x})), \text{ s.t. } \hat{x} \in T^k(x). \tag{10}$$

4.2 Previous MHM-based Method

Metropolis-Hastings Modifier (MHM) [93] regards the problem as a sampling problem, the stationary (target) distribution π is defined as:

$$\pi(x') \propto (1 - \mathrm{Prob}(x', y; C)) \cdot \mathcal{I}\{x' \in \mathcal{E}\}, \tag{11}$$

where $\mathrm{Prob}(\hat{x}, y; C)$ is the probability of y predicted by C, and $\mathcal{I}\{x \in \mathcal{E}\}$ ensures the validity of the sampled code. The sampling formulation is similar to the optimization problem (Equation (10)),

because sampling from Equation (11) tends to sample examples with low $\text{Prob}(x', y; C)$, which finally leads to a large loss $L(y, C(x'))$.

MHM is the current SOTA adversarial attack algorithm that attacks a DL classification model for source code iteratively, which consists in two phases at each iteration. The first phase generates a proposal to rename an identifier s, which is randomly sampled from all identifiers defined or declared within the snippet, in the code x to a new identifier t, which is randomly sampled from a candidate set, leading to a new snippet x'. Also, the candidate set is randomly generated. The second phase accepts or rejects the proposal according to the acceptance rate α, which can be approximately computed by:

$$\alpha = \min\{1, \alpha^*\} \approx \min\left\{1, \frac{1 - \text{Prob}(x', y; C)}{1 - \text{Prob}(x, y; C)}\right\}. \tag{12}$$

Although MHM is capable to produce valid adversarial examples for source code processing models, it is still limited with several drawbacks: ❶ MHM adopts a simple random-sampling-based method, which takes much effort to probe on π. Gradient guidance should be considered to boost the efficiency. ❷ MHM can only produce identifier-level perturbations, i.e., identifier renaming, which lacks diversity, and a multi-level attacking algorithm is needed. ❸ MHM requires to access the model for multiple times to generate the proposal and to compute the acceptance rate for M-H, therefore, an easy-to-compute and easy-to-understand algorithm is needed. In this article, we propose CARROT_A to address these limitations and aim to propose a more effective and efficient attacking technique with the guidance of gradient information.

4.3 CARROT Adversarial Attacker

The goal of adversarial attack is to find the code snippet that solves the optimization problem in Equation (10). We propose CARROT_A to generate valid adversarial examples by sharing the similar spirit of hill climbing, which is an iterative strategy seeking for the potential optima. Compared with other simulation approaches such as **Metropolis-Hastings (M-H)** sampling, the overhead of hill climbing is often small, making it feasible in practice.

Objective transformation. Performing hill climbing to search for optimal $\hat{x}$ of Equation (10) is equivalent to minimizing the predicted probability of y by C, as follows:

$$\min_{\hat{x}} \text{Prob}(\hat{x}, y; C), \text{ s.t. } \hat{x} \in T^k(x) \tag{13}$$

Equation (10) finds $\hat{x}$ that causes y and $C(\hat{x})$ to be the most different. Maximizing J, while minimizing $\text{Prob}(y; C)$ achieves the same goal. In other words, Equations (10) and (13) are equivalent. One benefit and purpose that we eventually choose to work on optimizing Equation (13) is that it reduces the computational cost, since we no longer need to compute the loss $L(y, C(\hat{x}))$.

CARROT_A algorithm. CARROT_A takes the DL model C and code snippet as inputs and outputs an adversarial example at its best effort. In Algorithm 1 (also see Figure 1), at ith iteration, the mutators (see Table 2) are used to randomly generate a set of equivalent code variants from the current code snippet x_{i-1} as candidates, denoted as $\mathcal{T} = \{x_1^*, \ldots, x_n^*\} \subset T(x_{i-1})$, among which all variants are able to pass the compilation and are equivalent to x_{i-1} (Line 3). Then, CARROT_A tests all candidates against C to determine whether an adversarial example is found (Lines 4–8). If an adversarial example is not found, then the candidate with the lowest probability on y is selected, denoted as x_{idx}^*. When the probability decreases, x_{idx}^* is passed to the next iteration as x_i; otherwise, x_i remains the same as x_{i-1} (Lines 9–14). The generation process continues until the allocated budget (i.e., iteration size) exhausts or an adversarial example is found.

ALGORITHM 1: CARROT Adversarial Attacker Algorithm.

Inputs:
Source code classification model C, Data pair $(x, y) \in \mathcal{D}$, s.t. $y = C(x)$;
Max iteration m, candidate size n.
Outputs:
Adversarial example $\hat{x} \in \mathcal{A}(x, y; C)$, or None (fail).

```
1: Initialize x_0 ← x, prob ← Prob(x, y; C)
2: for i in {1, 2, ..., m} do
3:     {x_1^*, x_2^*, ..., x_n^*} ← Mutator(x_{i-1}, n)
4:     for j in {1, 2, ..., n} do
5:         if C(x_j^*) ≠ C(x_{i-1}) then
6:             return x_j^*
7:         end if
8:     end for
9:     idx ← arg min_j Prob(x_j^*, y; C)
10:    if Prob(x_idx^*, y; C) < prob then
11:        x_i ← x_idx^*, prob ← Prob(x_idx^*, y; C)
12:    else
13:        x_i ← x_{i-1}
14:    end if
15: end for
16: return None
```

CARROT$_{\text{A}}$ takes both effectiveness and efficiency into consideration during the design. Hill climbing is similar to gradient descent, which is widely adopted in DL, as it is guaranteed to find the local optima for non-convex objectives and global optima for convex objectives. The computational efforts of CARROT$_{\text{A}}$ mainly consist of two parts—mutation operations and DL model probing. We consider the invocation of DL models to estimate the computational cost, including forward prediction and backward gradient propagation, because other arithmetic or logic operations in Algorithm 1 are much less time-consuming than one DL model invocation. Mutators in CARROT$_{\text{A}}$ at most require one invocation of the model to obtain the gradient information, and in other situations where the gradient is not employed, this invocation is not even required. Therefore, The computational cost of mutators is $O(1)$. However, at Line 9 in Algorithm 1, the probability evaluation requires to invoke the DL model for n times (candidate size), and the computational cost of this part is $O(n)$. Therefore, the computational cost of CARROT$_{\text{A}}$ is $O(n)$, and the overhead of the mutators can be neglected. CARROT$_{\text{A}}$ seeks every opportunity to make the objective decrease, leading to fewer invocations of C with higher efficiency. We further incorporate gradient information into the mutation operation whenever available, guiding the searching process with even higher efficiency. Although retrieving the gradients may cost much more time during the mutation operations, it can effectively reduce the searching iterations.

4.4 Mutators for Candidate Code Generation

Mutators play an important role in CARROT$_{\text{A}}$, which could potentially influence the diversity and effectiveness of the attack. We design mutators to follow rule-based transformations that satisfy the constraints in Equation (7). We also take the mutator set extensibility into consideration, where different mutators could be easily integrated. As an early step in adversarial attack for source code, in this article, we cover basic while efficient mutators at different levels for attacking integration (please refer to "I-Mutator with gradient guidance" and "S-Mutator" in this section for detailed discussion about the efficiency). In particular, we include six common mutators at two levels, i.e., token-level (I-Mutator) and statement-level (S-Mutator) (see Table 2).

The token-level mutators are inherited from the previous work for adversarial attack against DL for source code processing [93]. They manipulate tokens in the tokenized source code sequences, such as variable (global/local) renaming, data type renaming, function renaming (please refer to "Token" in Table 2). The statement-level mutation performs higher-level transformations, which involves dead statement insertion and deletion, including empty statement (e.g., ";" in C/C++)

insertion/deletion, no-entry branch (e.g., "if(false);") insertion/deletion and no-entry loop (e.g., "while(false);") insertion/deletion (please refer to "Stmt" in Table 2). The statement-level operations are inspired by EMI [50], which generates semantically equivalent code snippets for compiler testing by inserting or deleting dead code. In particular, no-entry branch and loop insertion and deletion in Table 2 are absorbed from Hermes [73], which is an extension of EMI.

I-Mutator. I-CARROT$_{\mathrm{A}}$ is CARROT$_{\mathrm{A}}$ equipped with the I-Mutator, which performs multiple renaming operations. I-Mutator requires to gather all renamable candidate identifiers (e.g., variables, structures, unions, enumerates, functions) that form the set $\mathcal{S}$. $\mathcal{S}$ can be obtained by a fast scan during pre-processing, and it is updated during the attack iterations. The target identifiers are drawn from a set $\mathcal{T}(x, s) = \mathrm{ID}(\mathcal{V}_C) - \mathcal{S}$, where x is the code snippet, $s \in \mathcal{S}$ is the identifier to be renamed, and $\mathrm{ID}(\mathcal{V}_C)$ are legal identifiers in the vocabulary of C. This definition ensures the validity of the renaming operation and avoids duplicated names.

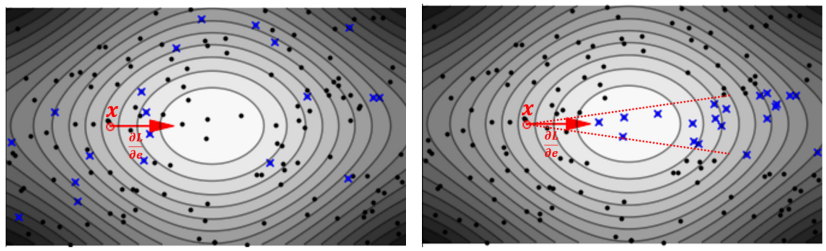
There may be some concerns that identifier renaming can be too naive and trivial. Ideally, the DL models for SE should be capable to resist against semantic equivalent transformations, such as identifier renaming, for better generalization ability. However, as an early stage to study the robustness of DL for source code processing, this article takes one step further towards the ultimate goal. In addition, our experimental results reveal that the classic or even the current SOTA DL models for source code processing cannot handle the renaming perturbations very well (please refer to Section 6.2). The subject models are obtained following instructions from the original papers, and they produce comparable performance upon training, validation, and test sets for each subject task. In short, our findings reveal that the DL models for source code processing contain severe potential risks of non-robustness, even against simple perturbations such as identifier renaming. This article aims to better understand how well the current DL models for SE perform against adversary, and we also would like to learn the challenges and the opportunities along this research direction.

Simple random I-Mutator. A simple implementation of I-Mutator is to randomly draw $t \in \mathcal{T}(x, s)$, generating one single candidate by renaming s in x to t. CARROT$_{\mathrm{A}}$ equipped with random I-Mutator, denoted as I-RW,[6] can be viewed as a degeneration of the previously proposed MHM [93]. I-RW first generates a random renaming proposal and then accepts it if the probability on the ground-truth class decreases, otherwise, I-RW rejects it. I-RW also serves as the random baseline in this article.

I-Mutator with gradient guidance. The random sampling approach can be useful but inefficient, since the attack space is quite large and simple random walking without gradient guidance would miss the optimal candidate in many cases, as illustrated in Figure 2(a). Inspired by gradient-based attack algorithms (e.g., FGSM [34] and BIM [49]) that optimize Equation (4) by jumping according to the gradient direction, we incorporate gradient information, which can be produced by most DL models for source code processing, into I-Mutator. In particular, I-Mutator generates $s \in \mathcal{S}$, which change towards similar directions to the gradient in the embedding space, as candidates. The similarity is measured by a cosine-similarity-like scoring function, and I-Mutator chooses the top-n identifiers. The candidate size is a hyper-parameter of I-Mutator. I-Mutator renames code snippet x on identifier s based on the scoring function defined as:

$$S(t, s|x) = \frac{e(t) - e(s)}{\|e(t) - e(s)\|_2} \frac{\partial L(y, C(x))}{\partial e(s)}, \tag{14}$$

[6] "RW" here refers to random walking, because the random renaming process is quite similar to the process of random walking.



(a) MHM (b) I-CARROT$_A$

Fig. 2. An illustrative example to illustrate the effectiveness gaining from gradient guidance (best viewed in color). The scatters are source code pieces in the embedding space, and the altitude refers to the loss L, expressed by contour lines. The circle ($\circ$) represents the code currently to be renamed (x), the crosses ($\times$) refer to code selected as candidates, and the dots ($\cdot$) are the rest of the code snippets. The arrow ($\rightarrow$) refers to the gradient vector ($\frac{\partial L(y,C(x))}{\partial e(x)}$), which points out the direction where L rises fastest. (a) MHM uniformally randomly samples code snippets among all legal perturbations, regardless of the gradient guidance. Therefore, MHM generates many random "bad" candidates, which cannot increase L or even miss the desirable perturbations. (b) However, with the guidance of gradient, I-CARROT$_A$ picks the code snippets in the cone with the gradient as the axis, represented with dash lines ($\cdots$), which produces better candidates than random sampling.

where $t \in \mathcal{T}(x, s)$ is the possible target identifier, $e(s)$ and $e(t)$ are embedding vectors of s and t, and $\frac{\partial L(y,C(x))}{\partial e(s)}$ is the embedding-level gradient vector of s. Equation (14), which is proportional to the cosine similarity, measures the similarity between the changing direction of renaming s to t and the gradient direction. By selecting the top-n identifiers that cause the similar changes as the gradient, we introduce the gradient into I-CARROT$_A$. This technique picks the code snippets lying within a thin cone with the gradient as its axis (Figure 2(b)), and therefore includes the desirable candidate identifiers, which leads to great increasing in $L(y, C(x))$. We select $t_1, \ldots, t_n = \arg n \max S(t, s|x)$ and renames in x to $t_1, \ldots, t_n$ to generate the equivalent candidates $x_1^*, x_2^*, \ldots, x_n^*$ (see Line 3 in Algorithm 1).

Unlike the completely random strategy in MHM, the gradient information brings two impacts towards I-CARROT$_A$: ❶ The computational cost does not increase due to the gradient operation. I-Mutator is relevant to the DL model C for sure, since it needs to invoke C to obtain $\frac{\partial L(y,C(x))}{\partial e(s)}$. However, compared to the testing step in CARROT$_A$ (Lines 4–8 in Algorithm 1 invoke C for n times), the overhead of this single invocation can be ignored. ❷ The efficiency of the iterative process can be improved greatly with the gradient guidance. MHM randomly samples identifiers, forming the candidate set, as shown in Figure 2(a). Such strategy does not consider the desirable changing direction, instead, it mainly picks the "bad" candidates, which cannot increase the loss. However, candidates from I-CARROT$_A$ (Figure 2(b)) are purposeful. These candidates are in a thin cone with the gradient as its axis, and they are guided to perturb the code towards the direction of increasing the loss. As I-CARROT$_A$ generates more effective substitution candidates than MHM, I-CARROT$_A$ would overall show better effectiveness and efficiency.

Validity of I-Mutator. We have taken compilation and semantically equavalent issue during transformation into consideration during design. By design, I-Mutator makes no influence to the compilation of the perturbed code, because we rename only the user-defined identifiers, which have no dependencies with the external environment. Meanwhile, identifier substitutions by I-Mutator do not alter the execution as well, because the renaming towards a certain identifier results in identical executable file after compilation as the original. Therefore, $\mathcal{X}(x', x) = 1$ is guaranteed

in I-Mutator. To be more specific, I-Mutator collects all renamable identifiers, forming $\mathcal{S}$, from the given compilable and executable code. For instance, given a single source code file, which includes the whole program, $\mathcal{S}$ is collected by traversal searching for definition or declaration nodes in the syntax tree. In a more complex scenario, where the project consists of multiple files with dependency, I-Mutator compiles the whole project to obtain the syntax tree and still traverses the tree to collect $\mathcal{S}$. I-Mutator does not substitute those identifiers that may have dependencies from other files that are not provided. Identifiers in $\mathcal{S}$ obtained in such approach exists in and only within the given code itself. During perturbation, I-Mutator renames identifiers in the given code files and all downstream files to ensure validity, i.e., $\mathcal{X}(x', x) = 1$.

S-Mutator. S-CARROT$_{\text{A}}$ represents CARROT$_{\text{A}}$ equipped with the S-Mutator. S-Mutator inserts (or deletes) three kinds of statements into (or from) the snippets, including ❶ empty statements, e.g., ";" and "printf("");" in C/C++, ❷ no-entry branches, e.g., "if(false);", and ❸ no-entry loops, e.g., "while(false);" and "for(;false;);". For S-Mutator, we define an insertable statement set (denoted as $\mathcal{S}$) and collect all positions (denoted as $\mathcal{P}$) in the code snippet, where statement insertion is possible, by one pass scanning during pre-processing. Also, a set $\mathcal{I}$ is needed to record all deletable statements.

At each iteration, S-Mutator first randomly decides whether to insert or delete a statement. For insertion, S-Mutator randomly samples $s \in \mathcal{S}$ and $p \in \mathcal{P}$, and inserts s into x at position p, to generate a candidate. Otherwise, S-Mutator samples $d \in \mathcal{I}$ and deletes d in x. At the end of an iteration, S-Mutator updates $\mathcal{P}$ and $\mathcal{I}$ according to the decision of S-CARROT$_{\text{A}}$. In addition, the probability of insertion is also adjusted at each iteration to avoid too many insertions. The probability of insertion is defined as $r = 1 - \frac{n_{ins}}{n_{max}}$, where n_{ins} is the number of currently inserted statements and n_{max} is the max insertion threshold.

In S-Mutator, which is in fact black-box, gradient information is not incorporated due to two major challenges. ❶ Most DL models for source code cannot produce statement-level gradient. Although it can be tackled by aggregating (e.g., averaging) gradient vectors of all tokens in the perturbed statement, such technique still cannot deal with the second challenge. ❷ S-Mutator performs statement insertion and deletion, which are discrete and underivable operations. Gradient for operations such as insertion or deletion is undefined and nonexistent. Therefore, we adopt random searching in S-Mutator.

Still, S-CARROT$_{\text{A}}$ meets the same convergence with or without the gradient guidance. Hill climbing in S-CARROT$_{\text{A}}$ can be viewed as a special case of **Metropolis-Hastings (M-H)** sampling [22, 38, 61]. As long as the transition proposal in M-H is aperiodic and ergodic, given enough iterations, the algorithm converges to the same stationary distribution. In S-CARROT$_{\text{A}}$, the transition proposal includes insertion and deletion of three types of statements that are aperiodic and ergodic, hence, S-CARROT$_{\text{A}}$ converges to the same stationary distribution.

Simple random S-Mutator. Similar to I-RW, we also implement a simple random baseline for S-CARROT$_{\text{A}}$, denoted as S-RW. S-RW generates one single candidate by inserting a random statement $s \in \mathcal{S}$ into a random position $p \in \mathcal{P}$ or deleting a random statement $d \in \mathcal{I}$ from the snippet. To be clear, the difference between S-Mutator and S-RW is the candidate size. S-Mutator generates multiple (n) candidates of insertion or deletion, while S-RW produces only one candidate. Therefore, S-RW is based on randomness, which creates a scenario similar to random walking, as the name suggests.

Validity of S-Mutator. By design, S-Mutator does not alter the compilation and execution validity of the code after perturbation as well, because the inserted or deleted statements are dead code. Such perturbations do not change the execution trace in data flow or control flow. The insertable

position $\mathcal{P}$ is collected from the complete syntax tree, where the statements can be easily split. The inserted statements $\mathcal{S}$ are all in fact dead code, which does not change the execution results. However, from the perspective of compilation process, the inserted dead code may be eliminated due to some optimizations. For instance, GCC eliminates the dead code when the "-fdce" optimization option is on. Some pre-processing with dead code elimination is very possible to invalidate S-CARROT$_{\mathrm{A}}$. Nevertheless, as an early step, we notice that most DL models for SE do not contain such pre-processing, making S-CARROT$_{\mathrm{A}}$ still a potential threat to them.

5 CARROT ROBUSTNESS ESTIMATION & ENHANCEMENT

5.1 Robustness on a Single Example

As discussed in Section 3.3, the proportion of examples in the test set, on which the model C is robust, can be used as the metrics of model robustness. To achieve this, we need to first determine whether C is robust on a single given example pair (x, y).

As Equation (2) indicates, model C is robust on x when the output remains accurate under any perturbations that satisfy the constraints. We define robustness on a single example $R(C|x, y)$ in source code processing as:

$$R(C|x,y) = \begin{cases} 0, & \text{If } \exists \hat{x} \in T^k(x), \text{ s.t. } C(\hat{x}) \neq C(x); \\ 1, & \text{Otherwise.} \end{cases} \tag{15}$$

If one can find an adversarial example for x towards C iteratively utilizing the transofrmations (T), then C is not robust on x ($R(C|x, y) = 0$). As we need to adopt the rule-based transformations in Equation (10) instead of the L_p norm constraint, robustness analysis becomes much more challenging. Directly solving Equation (10) utilizing MILP is impractical due to the discrete and complex constraint rules. The transformation rules also make the allowable perturbations discontinuous (while the L_p norm constrains a regular norm ball), making the continuous approaches (e.g., bound propagation) infeasible. Therefore, adversarial attack remains to be the most promising and practical solution to estimate (non-)robustness of DL models in source code processing.

During robustness estimation given mutator set T, the operations in T in Equation (10) are allowed. We employ a set of attack algorithms as a toolkit A, which covers all allowable mutation operations defined in T. The toolkit searches within the allowable perturbations. The estimated upper bound of $R(C|x, y)$ with toolkit A is formally defined as:

$$R(C|x,y) \leq \tilde{R}_A(C|x,y) = \begin{cases} 0, & \text{If find } \hat{x} = A(x,y;C); \\ 1, & \text{Otherwise.} \end{cases} \tag{16}$$

Where $\hat{x} = A(x, y; C)$ means that A is capable to find an adversarial example, otherwise all algorithms in A cannot generate any adversarial examples. This estimation of $\tilde{R}_A(C|x, y)$ is sound but incomplete, because $R(C|x, y)$ is guaranteed to be 0 when an adversarial is found and no conclusion could be guaranteed otherwise.

5.2 Robustness on a Dataset

The robustness $R(C|\mathcal{D})$ of C on a set of examples $\mathcal{D}$ can be calculated as Equation (5). $R(C|\mathcal{D})$ shows the proportion of robust examples among all examples of a dataset. A higher $R(C|\mathcal{D})$ indicates that C is more likely to be robust on a new example.

Similarly to the scenario of a single example, we leverage a set of adversarial attack algorithms, denoted as toolkit A, to estimate $R(C|\mathcal{D})$ within the allowable perturbations T. The estimation

process is shown in Figure 1, and the estimation is formulated as:

$$R(C|\mathcal{D}) \leq \tilde{R}_A(C|\mathcal{D}) = \frac{1}{\|\mathcal{D}\|} \sum_{(x,y)\in\mathcal{D}} \tilde{R}_A(C|x,y). \tag{17}$$

$\tilde{R}_A(C|\mathcal{D})$ is also a sound but incomplete estimation to $R(C|\mathcal{D})$, which provides an upper bound of $R(C|\mathcal{D})$. The diverse mutators T and powerful attacks will often obtain a more accurate estimation $\tilde{R}_A(C|\mathcal{D})$ of $R(C|\mathcal{D})$.

$\tilde{R}$ as likelihood. From the perspective of probability statistics, $\tilde{R}_A(C|\mathcal{D})$ is a likelihood of how robust the model C is against attack toolkit A, retrieved on set $\mathcal{D}$. To be more specific, $\tilde{R}_A(C|\mathcal{D})$ provides *a priori* of $R(C|\mathcal{D})$, where the attack toolkit A and the dataset $\mathcal{D}$ are the condition. Therefore, as we incorporate more types of attack approaches into A, we have a higher chance to hit the non-robust issue with adversarial examples generated by A, and finally this may lead to a more accurate likelihood. Hence, $\tilde{R} \approx 0$ suggests that the subject model is very likely to be non-robust; however, $\tilde{R} \approx 1$ indicates the model to be very likely robust.

Diversity of the mutator. Besides the size of $\mathcal{D}$, the diversity of the mutators is also related to how accurate $\tilde{R}$ estimates R. As aforementioned, we approximate semantic equivalency (Equation (8)) with the semantic equivalent transformation (Equation (9)). It is incomplete, since there exists many or even infinite transformation operators, while we may only employ a small part of the them due to the algorithm capacity. By diversifying the transformation operators, we can cover more types of semantically equivalent code snippets, reveal more possible risks (adversarial examples) in the DL model, and finally produce a more accurate estimation of the model robustness R. Therefore, the diversity of the mutator is related to the effectiveness of the robustness estimation in CARROT_M.

CARROT_M robustness estimation. CARROT_Mutilizes CARROT_A (I-CARROT_A and S-CARROT_A) as the adversarial attack toolkit to estimate the robustness of DL model for source code processing. As we illustrated, CARROT_A is effective, efficient, and multi-granular, which is suitable for robustness measurement. CARROT_M is also designed to be extensible, where new attack methods and robustness metrics could be easily incorporated.

5.3 Robustness Enhancement

Adversarial training. Similar to adversarial examples and adversarial attack, adversarial training for source code processing is defined by altering the constraints in Equation (6) as:

$$\min_{\Theta_C} \sum_{x,y\in\mathcal{D}^{(t)}} \max_{\hat{x}} L(y, C(\hat{x})), \text{ s.t. } \hat{x} \in T^k(x). \tag{18}$$

Previous adversarial training approaches largely fall into the following three categories: ❶ solving the minimax problem directly utilizing integer programming [56], ❷ minimizing the upper bound of the inner maximization through bound propagation [41], and ❸ training the DL model with adversarially perturbed examples [81]. The former two approaches may not be feasible for source code processing tasks, due to the extremely high computational cost.

Adversarial training through data augmentation. We adopt the third approach and improve the robustness of the DL models by training with adversarially perturbed examples, formulated as:

$$\min_{\Theta_C} \sum_{x,y\in\mathcal{D}^{(t)}} L(y, C(x)) + \lambda \sum_{x,y\in\mathcal{D}_A^{(t)}} L(y, C(x)), \tag{19}$$

ALGORITHM 2: Data Augmentation Algorithm in CARROT Adversarial Training

Inputs:
DL model C, Dataset $\mathcal{D}$, Adversarial attack toolkit A, Generation size n
Outputs:
Adversarially augmented dataset $\mathcal{D}_A$
1: Initialize $\mathcal{D}_A \leftarrow \mathcal{D}$
2: **for** i in $\{1, 2, \ldots, n\}$ **do**
3: Sample x, y from $\mathcal{D}$ without replacement
4: $\mathcal{D}_A \leftarrow \mathcal{D}_A \cup \{A(x, y; C_A)\}$
5: **end for**
6: **return** $\mathcal{D}_A$

ALGORITHM 3: CARROT Adversarial Training Algorithm.

Inputs:
Traditionally trained DL model C, Original training set $\mathcal{D}^{(t)}$, Adversarial attack toolkit A, Generation size n_g, Generation period N_g, Max epoch N_e
Outputs:
Adversarially trained model C_A
1: Initialize $n_{es} \leftarrow 0$, $\mathcal{D} \leftarrow$ DataAugment(C, $\mathcal{D}^{(t)}$, A, n_g), $C_0 \leftarrow$ RandomInit(C), $C_A \leftarrow C_0$
2: **for** e in $\{1, 2, \ldots, N_e\}$ **do**
3: **if** e mod $N_g = 0$ **then**
4: $\mathcal{D} \leftarrow$ DataAugment(C_A, $\mathcal{D}^{(t)}$, A, n_g)
5: **end if**
6: $C_e \leftarrow$ TrainEpoch(C_{e-1}, $\mathcal{D}$);
7: **if** Performance($\mathcal{D}^{(v)}$; C_e) > Performance($\mathcal{D}^{(v)}$; C_A) **then**
8: $C_A \leftarrow C_e$, $n_{es} \leftarrow 0$
9: **else**
10: $n_{es} \leftarrow n_{es} + 1$.
11: **end if**
12: **if** EarlyStopping(n_{es}) **then**
13: **return** C_A.
14: **end if**
15: **end for**
16: **return** C_A.

where $\mathcal{D}_A^{(t)}$ is a static set of examples perturbed by toolkit A from a subset of $\mathcal{D}^{(t)}$ against an identically well-trained model C_0. λ and the size of $\mathcal{D}_A^{(t)}$ are hyper-parameters, which regulate the proportion of adversary during training.

CARROT robustness enhancement. CARROT$_\mathrm{T}$ augments the training set with adversarial examples periodically. Algorithm 2 presents the data augmentation via adversarial examples. The algorithm samples some (n to be more specific) of the examples in dataset $\mathcal{D}$ and generates adversarial examples using toolkit A to augment the original $\mathcal{D}$, forming the augmented dataset $\mathcal{D}_A$. Algorithm 3 gives the details of adversarial training, and Algorithm 2 is invoked as a function named "DataAugment." Especially, the augmented training set is updated every n_g epochs, with new adversarial examples (Lines 3–5). At the eth epoch, the model C_e is optimized upon the augmented training set based on C_{e-1} of the last epoch (Lines 6–14). Note that n_{es} records how many rounds of the model without performance improvement, according to which the algorithm decides to early stop (Line 7). This periodic update helps to renew the perturbations against the non-robustness of the current best model and improve the robustness of the model.

There are two major reasons for generating adversarial examples for only a part of the training set. ❶ Training with all perturbed examples may lose too much performance (e.g., accuracy or F1-score). During training, when the proportion of adversarial example increases, the accuracy would first increase and then decrease after reaching a threshold, which is shown in the previous work [93]. ❷ Generating adversarial examples from the whole training set is too time-consuming. For instance, in our experiment, in OJClone, ASTNN attacked by I-CARROTA, the average adversarial example generation time is 10.4 s (see Table 11) and the training set size is 41,581 (see Table 3). Therefore, the process of the whole training set augmentation would cost about 120 hours (5 days).

Table 3. Statistics of the Subject Datasets

Dataset	Train #	Test #	Class #	Vocab #	LOC
OJ	41,581	10,395	104	10,283	~1,868K
OJClone	40,000	10,000	2	2,827	~278K
CodeChef	27,058	6,764	4	3,755	~1,168K

[†] Counted in the whole dataset (training + testing).
[⋄] Counted only in the training set.

In addition, in these 5 days, we can only generate adversarial examples from the whole training set, without actually training the DL model. To facilitate the feasibility and practicalness of adversarial training, we apply uniform sampling to randomly select which example in the training set is to be perturbed.

6 EVALUATION

We implemented CARROT as an extensible framework in Python based on the DL framework PyTorch (ver.1.6.0). With CARROT and its three major components, we perform a large-scale study to investigate the following research questions:

RQ1 (Non-robust Issue): Are the classic or SOTA DL models of source code processing robust against simple random perturbations?
RQ2 (Adversarial Attack): Are the proposed I-CARROT$_A$ and S-CARROT$_A$ able to attack the DL models effectively and efficiently?
RQ3 (Human Evaluation): Are the adversarial examples produced by CARROT indistinguishable from the original examples to human beings?
RQ4 (Robustness Measurement): Whether or how robust are the DL models under our proposed robustness metrics? Is CARROT$_M$ able to obtain a tighter estimation of the true robustness?
RQ5 (Robustness Enhancement): Is CARROT$_T$ useful for enhancing the robustness of DL models for source code processing?

6.1 Experiment Settings

Subject tasks and datasets. As a very early attempt to study the robustness of DL models for source code processing, we select three representative source code classification tasks and datasets (i.e., functionality classification OJ [62, 95], code clone detection OJClone [85, 95], code defect prediction CodeChef), which are the basis for more complex generation tasks. Table 3 summarizes the statistics of the subject datasets, which contains code snippets with more than 3.3 million LOC. In particular, OJ and OJClone are based on the Open Judge benchmark dataset proposed by Mou et al. 2016 [62] and are included in the recently proposed CodeXGLUE benchmark [59]. We follow the previous work to pre-process OJ and OJClone [95] for further analysis. CodeChef is a code defect prediction dataset originally created in this work. Compilable C/C++ code snippets are retrieved from the CodeChef platform[7] and labeled by the execution results from the platform, i.e., "OK" (no defect), "WA" (defect-1), "TLE" (defect-2), and "RE" (defect-3). The detailed definitions of each class in CodeChef are listed in Table 4.

Subject models. For each dataset, we mainly investigate CARROT against three subject models (i.e., GRU, LSTM, and ASTNN) used in previous work [53, 95]. To further verify the versatility of CARROT, we also evaluate it against the classic LSCNN [42] and TBCNN [62] models, the recently proposed transformer-based pre-trained CodeBERT [28] model, and the classic task-oriented

[7] https://www.codechef.com/.

Table 4. Definitions of Different Classes in CodeChef

Lacel	Class	Definition	Examples
0	OK	No defect. Pass all test cases.	–
1	WA	Defect-1. Inconsistent outputs.	Wrong answer
2	TLE	Defect-2. Timeout.	Infinite loop, high computational cost, etc.
3	RE	Defect-3. Runtime error.	Memory leak, divided by zero, etc.

CDLH [85] model, which is designed for code clone detection. We select these seven models from the perspective of the data format, the model architecture, and the purpose of the design, which have been explained in Section 2.1.

GRU and **LSTM** are the most classic sequential models adopted in the SE community. Although current SOTA models do not directly apply LSTM or GRU architectures, these sequential architectures are often utilized as backbones [3, 35, 42, 52, 53, 95]. E.g., DeepAPI [35] utilizes GRU backbone and Seq2seq [7] architecture to generate API sequences from natural language descriptions, DeepCommenter [52] adopts GRU to generate code comments, and Code2Seq [3] employs LSTM to process the node paths in the AST. Therefore, GRU and LSTM are two of the most representative sequential architectures, and plenty of SOTA models for different tasks are derived from them. In our experiments, we implement bidirectional GRU and LSTM with attention mechanism [7]. These two techniques (bidirection and attention) are often used together with GRU and LSTM in the previous work [53]. Concretely, in our implementation, each GRU or LSTM layer consists of one forward sub-layer and one backward sub-layer, and we utilize attention to compute the probabilistic weights for weighted averaging of the hidden state sequence.

ASTNN [95] is designed for code representation and the downstream classification tasks. It is one of the SOTA models in functionality classification and clone detection. ASTNN splits the AST into a sequence of ST-trees (i.e., statements). It then utilizes recursive neural network to encode the ST-trees and GRU to encode the ST-tree sequence. We reuse the project open-sourced by the authors.[8]

LSCNN [42] is a classic architecture originally proposed to detect defects according to the source code and the natural language description. Since our subject datasets do not contain the natural language description, we employ only the LSCNN architecture to process the source code snippets, removing the LSTM module for natural language processing. The LSCNN model handles the hierarchy of the code and extracts the vectorized code representation. LSCNN employs one-dimensional convolution upon the statements of the code snippets and leverages LSTM to process the statement sequences. At last, the model performs max-pooling upon the hidden-states, obtaining the representations of the code snippets. Since the original paper does not provide the open-sourced project, we implement the model by ourselves using PyTorch.

TBCNN [62] is another classic model to process the AST structures of the code snippets. TBCNN utilizes tree-convolution to compute representations for each node in the AST and obtain the overall representation by max-pooling. Although the original paper provides an available project, it is implemented with C++. Therefore, to facilitate our experiments, we implement TBCNN with the Python packages of PyTorch and DGL (ver.0.6.1).[9] After consulting the original authors, we make a modification towards the original model for easier implementation—we employ a similar node embedding approach as ASTNN [95] by considering the values of the terminal nodes (corresponding to identifiers, names, etc.) and the types of the non-terminal nodes.

CodeBERT [28] is a recently proposed pre-trained model for programming and natural languages. It is based on the transformer [79] architecture. Instead of trained for specific tasks in an

[8]https://github.com/zhangj111/astnn.
[9]https://github.com/dmlc/dgl.

Table 5. Configurations of the Subject Models

Hyper-parameters	GRU	LSTM	ASTNN	LSCNN	TBCNN	CodeBERT	CDLH
Train # : dev #	4:1	4:1	4:1	4:1	4:1	4:1	4:1
Vocab # in OJ	5,000	5,000	10,283	5,000	5,000	Fixed Pre-trained Config	–
Vocab # in OJClone	2,000	2,000	2,827	2,000	2,000		2,000
Vocab # in CodeChef	3,000	3,000	3,755	3,000	3,000		–
Embedding size	512	512	128	512	256		128
Hidden size	600	600	100	400	256		128
Layers	2	2	1	1 Conv + 1 LSTM	1		1
Dropout	0.5	0.5	0.2	0.5	0.1	0.1	0.1
Batch size	32	32	64	32	8	10	32
Max epoch	15	15	5(OJClone), 15(Others)	15	15	15	15
Optimizer	Adam	Adam	AdaMax	Adam	Adam	Adam	Adam
Learning rate init value	0.003	0.003	0.002	0.001	0.001	0.00003	0.001
Learning rate decay	Exp	Exp	No decay	Exp	Exp	Exp	Exp
Early stopping	✓	✓	✓	✓	✓	✓	✓

Table 6. Performance of the Subject Models

Model	OJ	OJClone			CodeChef		
	Acc (%)	Prec (%)	Recall (%)	F1 (%)	Prec (%)	Recall (%)	F1 (%)
GRU	93.8	93.5	71.4	81.0	73.0	72.7	72.6
LSTM	95.6	95.1	86.7	90.7	74.1	73.6	73.9
ASTNN	98.0	98.0	91.5	94.6	74.8	75.8	75.0
LSCNN	94.9	69.5	46.3	55.5	77.3	76.8	77.0
TBCNN	90.6	88.5	67.8	76.8	66.2	65.9	65.9
CodeBERT	96.9	0.0	0.0	0.0	76.6	75.9	76.2
CDLH	–	85.1	68.3	75.8	–	–	–

end-to-end style, CodeBERT follows another paradigm named "pre-training and finetuning." CodeBERT, consisting of 12 layers of transformers, is pre-trained upon CodeSearchNet [43] dataset, which consists of functions in six programming languages (Python, Java, JavaScript, Php, Ruby, and Go) along with the natural language documentations. For a certain downstream task, we can easily fine-tune the pre-trained model upon the specific dataset and obtain the fine-tuned CodeBERT with good performance. In the recent proposed CodeXGLUE [59] benchmark, CodeBERT achieves the SOTA performance on many different tasks. We use the open-sourced CodeBERT-base model,[10] supported by the Python package of transformers[11] (ver.3.3.0, with PyTorch backend support).

CDLH [85] is a classic model specially designed for code clone detection. It is based on N-ary Tree-LSTM [77], processing the binarized AST recursively in a bottom-up manner to extract code representations. At last, the model compares the representations of the code pair to determine whether they are clones. Because the code is not open-sourced in the original paper, we implement the model with PyTorch and DGL. For easier implementation, we make a modification towards the model—instead of N-ary Tree-LSTM, we adopt another Child-sum Tree-LSTM [77], which does not limit the number of child nodes in the tree, and therefore, we do not have to binarize the AST.

The detailed hyper-parameter configurations of these victim models are listed in Table 5. The configuration of CodeBERT is fixed, because it is a pre-trained model, so there is a gray-shade box in Table 5. Most of the subject models achieve competitive evaluation performance (e.g., accuracy, F1 score) upon the studied tasks on the test set (see Table 6). Note that the F1 score of CodeBERT for OJClone is 0.0, indicating it is not a satisfactory fine-tuned model. Our further analysis finds that the accuracy (considering both the clone and the non-clone examples in the test set) is about 93%, which is consistent with ratio of Clone and Non-Clone examples in OJClone (Clone : Non-clone

[10] https://huggingface.co/microsoft/codebert-base.
[11] https://github.com/huggingface/transformers.

Table 7. Configurations of the Attack Algorithms

Attack algorithm	GRU, LSTM, ASTNN		LSCNN, TBCNN, CodeBERT, CDLH	
	Candidate size	Max iteration	Candidate size	Max iteration
I-RW	1	100	1	40
S-RW	1	20	1	40
MHM	40	50	10	20
I-CARROT$_A$	40	50	5	40
S-CARROT$_A$	40	20	5	40

[†] Candidate sizes of I-RW and S-RW are always just 1.

$\approx 1:14$). This issue caused by unbalanced dataset is also confirmed by the authors of CodeXGLUE [59]. Because CodeBERT for OJClone is not satisfactory, we decide not to evaluate against it in our experiment.[12] CDLH for OJ and CodeChef in Table 6 is also empty, because it is specially designed for code clone detection only.

Performance indicators. In OJ, we employ accuracy (Acc) as the indicator, because accuracy has been adopted since TBCNN [62], which proposes this benchmark. In OJClone, previous work such as ASTNN [95] and CDLH [85] adopts precision (Prec), recall, and F1-score (F1) as the indicators. Especially, F1-score, which balances precision and recall, is widely adopted in the negative-positive pair classification tasks. Therefore, we employ F1-score as the major indicator for OJClone. As for CodeChef, which is a defect prediction dataset, there are four classes ("OK," "WA," "TLE," and "RE" in Table 4) and three negative-positive pairs, i.e., "OK"-"WA," "OK"-"TLE," and "OK"-"RE." Therefore, we list macro precision, recall, and F1-score in Table 6 and employ the macro F1-score as the major indicator during our evaluation.

Baseline algorithms. To better understand the advantage of our technique, we select RW (I-RW and S-RW) and the previous SOTA method **Metropolis-Hastings Modifier (MHM)** [93] as the baseline attack algorithms for comparison. **RW**, as a previously introduced random baseline of CARROT$_A$, is a simple method that perturbs code snippets in the similar manner of random walking, which carries out token-level (I-RW) and statement-level (S-RW) perturbations. **MHM** is a recently proposed SOTA adversarial attack approach, which performs iterative identifier replacement based on M-H sampling [22, 38, 61]. The hyper-parameters of the attack approaches are listed in Table 7.

Experimental configurations. We leverage CARROT to perform large-scale comparative experiments to answer the RQs. For **RQ1**, we attack the subject models in OJ, OJClone, and CodeChef (besides CodeBERT for OJ, CDLH for OJ and CodeChef) with the simple RWs (i.e., I-RW and S-RW) to demonstrate the non-robust issue of the DL models. For **RQ2**, to demonstrate the effectiveness and efficiency of CARROT$_A$, we attack the subject models in OJ, OJClone, and CodeChef (besides CodeBERT for OJ, CDLH for OJ and CodeChef) with I-CARROT$_A$ and S-CARROT$_A$, along with the baselines. The hyper-parameters of adversarial attack approaches are listed in Table 7. We uniformally sample a 20% subset from the test set and carry out adversarial attack upon the subset. In each attack, we repeat this process for five times to counteract the randomness. For **RQ3**, we invite nine independent volunteers (eight graduate students and one senior undergraduate student,

[12] In the original work of CodeXGLUE [59], CodeBERT for OJClone is obtained in two steps: ❶ fine-tune the model on the training set of OJClone, and ❷ search an activation threshold of the output probability p (e.g., "Clone" if $p >$ threshold; "Non-Clone" otherwise) upon the validation set. In our setting, because OJClone is highly unbalanced (Clone : Non-Clone$\approx 1:14$), the fine-tuned CodeBERT produces very low p given almost any input. The threshold is 0.01, with the searching granularity of 0.01, suggesting that the model predicts almost all examples as non-clone. We have consulted the authors of CodeXGLUE, and they also have confirmed this phenomenon. This result is not satisfactory for us, so we decide not to evaluate against CodeBERT for OJClone in our experiment.

Table 8. Numbers of Adversarial Examples to Augment the Training Set During Adversarial Training

Toolkit	OJ		OJClone	
	LSTM	ASTNN	LSTM	ASTNN
MHM	2,000	2,000	2,000	500
I-CARROT$_A$	2,000	2,000	2,000	1,000
S-CARROT$_A$	2,000	2,000	2,000	500
CARROT$_A$	4,000	4,000	4,000	1,500

all of whom major in computer science and have at least three years C/C++ programming and software development experience) to rate the generated adversarial examples from CARROT, RW, or MHM. We collect the original and the corresponding adversarial examples with the length constraint of 100 and filter those originally correctly predictred by the subject LSTM model. Among the selected examples, we further choose those that have misled the LSTM model after perturbation by all evaluated approaches. Finally, we retrieved 100 identifier-level perturbed examples (25 {I-CARROT$_A$,I-RW,MHM}+25 original). We randomly distribute the retrieved examples to the volunteers and each example is evaluated by at least two volunteers. For **RQ4**, we utilize the results of adversarial attack produced in RQ2 to estimate the robustness of all the subject models in the three subject tasks. CARROT$_A$ (I-CARROT$_A$ and S-CARROT$_A$), RW (I-RW and S-RW), and MHM (MHM only) are employed as toolkits in CARROT$_M$ estimation. Note that this article takes an early step in the field of robust DL for source code processing; there is a lack of existing baseline measurements. Therefore, we adopt the random RW and the previous MHM into the CARROT$_M$ framework for comparison, which is a second best choice we can make. For **RQ5**, we perform adversarial training on LSTM and ASTNN in OJ and OJClone, employing I-CARROT$_A$, S-CARROT$_A$, CARROT$_A$ (I-CARROT$_A$ and S-CARROT$_A$ both), and MHM baseline as toolkits and attack the adversarially trained models with all attack algorithms adopted in RQ2 to test their robustness. The generation period (N_g in Algorithm 3) in CARROT$_T$ is 14. The sizes of D_A (n_g) are listed in Table 8. The settings of adversarial attacks are the same as RQ2.

In summary, our evaluation consists of about 200 experimental configurations (about 100 in adversarial attack, and 100 in adversarial training). All the experiments were run on a server of Ubuntu 16.04 system with 32-core 2.10 GHz Xeon CPU, 125 GB RAM and 10 NVIDIA TITAN Xp 12 G GPUs. Overall, our evaluation takes more than 1,500 GPU hours to complete.

6.2 RQ1: Non-robust Issue

Table 9 includes the performance of the GRU, LSTM, and ASTNN subject models before and after random perturbation by I-RW and S-RW. Δ indicates the relative performance decrease after the random perturbation. A larger Δ indicates bigger performance drop, which suggests that the model is less robust against the code perturbations. We can see that different DL models exhibit different robustness performance drop under random perturbations, which can be dependent on both models and subject tasks. However, the obvious performance drops can be observed in most cases even by simple random perturbations. On average, token-level random perturbations (I-RW) reduce the performance of GRU, LSTM, ASTNN by 66.1%, 60.0%, 45.7% across all tasks, respectively. In particular, on CodeChef, the performance of all models decreases by more than 70%, suggesting they are not robust under token-level perturbations. Similarly, the statement-level random perturbations (S-RW) reduce the performance of GRU, LSTM, ASTNN by 40.7%, 34.0%, 29.9%, on average.

To demonstrate that the non-robust issue in general lies within many DL models for SE, we further present the performance of LSCNN, TBCNN, CodeBERT, and CDLH before and after perturbation by I-RW and S-RW in Table 10. I-RW reduces the performance of the classic LSCNN and TBCNN models by 51.9% and 65.2% across all three tasks on average, respectively, and the

Table 9. Performance of GRU, LSTM, and ASTNN Before and After Adversarial Perturbations

Model	Adversarial Attack	OJ		OJClone		CodeChef	
		Acc(%)	Δ(%)	F1(%)	Δ(%)	F1(%)	Δ(%)
GRU	None	93.8	–	81.0	–	72.6	–
	I-RW	62.6	33.3	19.7	75.7	7.8	89.3
	S-RW	68.3	27.2	56.3	30.5	25.8	64.5
	MHM	26.4	71.6	4.8	94.1	0.2	99.7
	I-CARROT$_A$	6.4	**93.2**	0.7	**99.1**	0.0	**100**
	S-CARROT$_A$	18.9	79.9	8.9	89.0	2.2	97.0
LSTM	None	95.6	–	90.7	–	73.9	–
	I-RW	76.4	20.1	23.0	74.6	10.9	85.3
	S-RW	74.4	22.2	66.3	26.9	34.7	53.0
	MHM	40.8	57.3	5.0	94.5	0.4	99.5
	I-CARROT$_A$	7.9	**91.7**	1.3	**98.6**	0.0	**100**
	S-CARROT$_A$	20.1	79.0	14.9	83.6	3.2	95.7
ASTNN	None	98.0	–	94.6	–	75.0	–
	I-RW	62.1	36.6	70.5	25.5	18.8	74.9
	S-RW	95.8	2.2	65.4	30.9	32.6	56.5
	MHM	8.0	91.8	8.4	**91.1**	8.5	88.7
	I-CARROT$_A$	0.8	**99.2**	40.7	57.0	0.1	**99.9**
	S-CARROT$_A$	75.4	23.1	34.4	63.6	1.8	97.6

[†] In OJ, $\Delta = 1 - \frac{\text{Acc}}{\text{Acc}_{\text{None}}}$, while in OJClone & CodeChef, $\Delta = 1 - \frac{\text{F1}}{\text{F1}_{\text{None}}}$; Δ suggests the effectiveness of the corresponding attack algorithm.

Table 10. Performance of LSCNN, TBCNN, CodeBERT, and CDLH Before and After Adversarial Perturbations

Model	Adversarial Attack	OJ		OJClone		CodeChef	
		Acc(%)	Δ(%)	F1(%)	Δ(%)	F1(%)	Δ(%)
LSCNN	None	94.9	–	55.5	–	77.0	–
	I-RW	43.1	54.6	41.0	26.1	19.3	74.9
	S-RW	63.0	33.6	17.7	68.1	51.1	33.6
	MHM	51.3	45.9	28.8	58.1	26.1	66.1
	I-CARROT$_A$	6.8	**92.8**	32.6	41.3	0.1	**99.9**
	S-CARROT$_A$	48.4	48.6	7.5	**86.5**	33.6	56.4
TBCNN	None	90.6	–	76.8	–	65.9	–
	I-RW	20.9	76.9	61.1	20.4	1.1	**98.3**
	S-RW	85.5	5.6	66.8	13.0	53.5	18.8
	MHM	30.1	66.8	12.7	**84.5**	17.5	73.4
	I-CARROT$_A$	15.3	**83.1**	49.6	35.4	1.2	98.2
	S-CARROT$_A$	79.2	12.6	49.1	36.1	40.0	39.3
CodeBERT	None	96.9	–	Not Evaluated		76.2	–
	I-RW	52.3	46.0			17.4	77.2
	S-RW	89.3	7.8			43.9	42.4
	MHM	84.8	12.5			19.6	74.3
	I-CARROT$_A$	27.7	**71.4**			5.4	**92.9**
	S-CARROT$_A$	79.6	17.9			24.4	68.0
CDLH	None	Not Evaluated		75.8	–	Not Evaluated	
	I-RW			27.0	64.4		
	S-RW			55.0	27.4		
	MHM			7.9	**89.6**		
	I-CARROT$_A$			13.4	82.3		
	S-CARROT$_A$			33.6	55.7		

[†] In OJ, $\Delta = 1 - \frac{\text{Acc}}{\text{Acc}_{\text{None}}}$, while in OJClone & CodeChef, $\Delta = 1 - \frac{\text{F1}}{\text{F1}_{\text{None}}}$; Δ suggests the effectiveness of the corresponding attack algorithm.

statement-level S-RW reduces the performance of LSCNN and TBCNN by 45.1% and 12.5% separately. When it comes to the pre-trained and transformer-based CodeBERT, I-RW and S-RW reduce the performance by 61.6% and 25.1%, respectively, on average in OJ and CodeChef. As for the task-oriented architecture, I-RW and S-RW reduce the performance of CDLH designed for clone detection by 64.4% and 27.4%, respectively, in OJClone.

In this section, to answer RQ1, we do not seek comparisons between I-RW and S-RW, instead, we focus on comparing the performance difference before and after the model is attacked by I-RW or S-RW. As demonstrated in the last two paragraphs, we can find that for most subject models, I-RW reduces the average performance across the three datasets by at least 40%, and S-RW reduces the average performance by at least 25%. These results reveal the non-robust issue of DL for source code processing, since even the simple random baselines (i.e., I-RW and S-RW) are capable

Table 11. Average Time Cost and Invocations to Generate an Adversarial Example

Model	Adv. Attack	OJ		OJClone		CodeChef	
		T (s)	Inv #	T (s)	Inv #	T (s)	Inv #
GRU	MHM	1.3	439	4.0	517	0.3	103
	I-CARROT$_A$	0.9	**152**	3.4	**255**	0.5	**67**
	S-CARROT$_A$	0.6	204	1.9	280	0.4	87
LSTM	MHM	1.8	482	3.0	587	0.3	112
	I-CARROT$_A$	1.1	**218**	2.1	**274**	0.5	**65**
	S-CARROT$_A$	0.4	225	1.0	301	0.5	103
ASTNN	MHM	4.5	1,269	28.2	426	9.2	177
	I-CARROT$_A$	3.2	**123**	10.4	319	1.2	**92**
	S-CARROT$_A$	8.6	381	3.9	**198**	1.4	113

[†] "T" = average time cost, "Inv #" = average invocation number.

to reduce the performance of the classic or current SOTA models for SE greatly. The results imply the non-robust limitations, challenges, and potential risks within DL for source code processing. Therefore, we evaluate our proposed CARROT for robustness detection, estimation, and enhancement in the following sections. In addition, we call on the SE community to pay attention to the challenging but important non-robust issue of DL for source code processing.

Answer to RQ1: Although achieving competitive performance, the classic sequential GRU and LSTM, the structural LSCNN, the tree-based TBCNN and ASTNN, the pre-trained CodeBERT, and the task-oriented CDLH, all face the non-robust issues in OJ, OJClone, and CodeChef against both token-level or statement-level random perturbations to a certain extent, indicating that the robustness issue could be a general concern to DL for source code processing and should be investigated as an important indicator besides accuracy and F1-score, and so on.

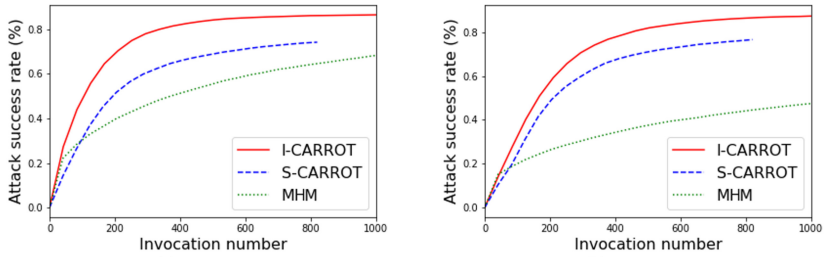
6.3 RQ2: Adversarial Attack

Effectiveness. Following RQ 1, Table 9 also shows the performance of GRU, LSTM, and ASTNN models before and after perturbations by CARROT$_A$ along with all the baselines. On average across all models and datasets, I-CARROT$_A$ and S-CARROT$_A$ reduce the performance of the DL models by 93.2% and 78.8%, respectively, outperforming the other baseline configurations in most cases. In particular, CARROT$_A$ reduces the F1-score of GRU, LSTM, and ASTNN in CodeChef by even more than 95%.

To demonstrate the versatility of CARROT$_A$, Table 10 lists the performance of LSCNN, TBCNN, CodeBERT, and CDLH before and after perturbations by the attack algorithms. In these experimental configurations, I-CARROT$_A$ and S-CARROT$_A$ reduce the performance of the subject models on average by 77.4% and 46.8%, respectively, also outperforming the other baselines in most cases. Specifically, I-CARROT$_A$ reduces the F1-score of LSCNN, TBCNN, and CodeBERT by more than 90%.

There are also some exceptions in the results, where in ASTNN and TBCNN for OJClone, the MHM baseline outperforms CARROT$_A$ greatly. There are two plausible explanations. ❶ Our in-depth investigation on the intermediate log reveals that CARROT$_A$, especially I-CARROT$_A$, may fall into the local optima, which may happen a lot during I-CARROT$_A$ attacking the subject models in OJClone. ❷ Note that both of the exceptions are related to OJClone. This indicates that in code clone, the token-level adversarial attack algorithm may benefit from randomness a lot, as the random sampling-based MHM outperforms the gradient guided I-CARROT$_A$ in several cases.

Efficiency. Table 11 shows the efficiency of CARROT$_A$ against GRU, LSTM, and ASTNN in line with SOTA MHM algorithm. Due to the low effectiveness, we do not include the random method



(a) OJ, GRU (b) OJ, LSTM

Fig. 3. Efficiency curves of the attack algorithms in OJ against GRU and LSTM.

for efficiency comparison. The results show that CARROT$_A$ generates adversarial examples with higher efficiency in all cases. On average, adversarial attack by I-CARROT$_A$ and S-CARROT$_A$ takes 2.6 and 2.1 seconds, 174 and 210 invocations, respectively, outperforming the SOTA MHM algorithm that takes 5.8 seconds and 457 invocations. In particular, I-CARROT$_A$ generates an adversarial example for ASTNN in OJ by only 123 invocations, which is 10 times more efficient than MHM. Furthermore, we also plot the success rate at different invocation numbers (see Figure 3). We can see that I-CARROT generates most of the examples in the early steps and quickly converges in success rate. This not only demonstrates the efficiency of CARROT$_A$, but also confirms the validity of our m selection.

Answer to RQ2: In general, CARROT$_A$ is capable to carry out adversarial attack against all subject DL models in OJ, OJClone, and CodeChef. In particular, I-CARROT$_A$ demonstrates its advantages in terms of both effectiveness and efficiency than MHM and random perturbations in OJ and CodeChef, where the gradient information is utilized. The effectiveness, efficiency, and naturally occurring diversity make CARROT$_A$ a feasible attacking algorithm.

6.4 RQ3: Human Evaluation

The general definition of adversarial examples requires the perturbations to be imperceptible to human beings [76]. Although we constrain the source code adversarial examples to be equivalent from the perspective of compilation and execution in Equation (8), we still would like to examine whether the perturbations of the generated adversarial examples are perceptible to programmers. Therefore, we carry out human evaluation by asking the programmers to distinguish whether the code is original or perturbed.

Questionnaire. We design the questionnaire to investigate whether or not the generated perturbations are perceptible to programmers from the perspective of human beings. The questionnaire for each participant consists of 25 sets of questions, each of which follows a similar template like the one shown in Table 12. All the adversarial examples (i.e., code snippets) come from the aforementioned 100 retrieved adversarial examples (see Section 6.1).

The questionnaire provides a code snippet (original or perturbed) along with the problem description of the ground-truth class and consists of two rating questions and one counting question to be filled by the volunteers. For the rating questions, the score ranges from −2 (bad) to 2 (good), and 0 indicates neutral or undecidable. The volunteers are first asked to rate whether the code matches the problem description, from −2 (cannot match) to 2 (match very well). Then, the volunteers are asked to rate if the code is understandable, from −2 (cannot understand at all) to 2 (easy to understand). At last, we ask how many unusual parts (e.g., unnatural variables and function names) in the code the volunteers can find in a relatively short time (20–40 sec). The "unusual

Table 12. Questionnaire for Human Evaluation

ID	Code	Problem description

1	`void main () {`
	`  int worker[<int>][<int>], i = <int>, j;`
	`  while(scanf(<str>, worker[i]) != EOF) {`
	`  worker [i]; i++;`
	`  }`
	`  for(j = i-<int>; j >= <int>; j--) {`
	`  if(j != <int>)`
	`  printf(<str>, worker[j]);`
	`  else`
	`  printf(<str>, worker[j]);`
	`  }`
	`}`
		Given a sentence, output all words in inverted order, and separate them with spaces.

Question	Score
Match	2 (The code matches the desc. well.)
Understandable	-1 (I can hardly understand the code.)
Unusual #	1 (The name "worker" is unusual.)

[†] In the code, "<int>" and "<str>" refer to integer and string constants, respectively.
[⋄] Items to be filled by the volunteers. In this table, we fill these items as an illustrative example.

Table 13. Human Evaluation Results Upon LSTM in OJ

Adv. Attack	Match		Understandability		Unusual #	
	Rate	Δ(%)	Rate	Δ(%)	#	Δ(%)
None	1.576	–	1.441	–	0.424	–
I-RW	1.222	22.5	1.167	19.0	1.019	140.3
MHM	1.474	**6.5**	1.333	**7.5**	0.965	127.6
I-CARROT$_A$	1.446	8.2	1.250	13.3	0.839	**97.9**

[†] For match and understandability, $\Delta = \frac{\text{Rate}_{\text{None}} - \text{Rate}}{\text{Rate}_{\text{None}}}$, while for unusual count, $\Delta = \frac{\#-\#_{\text{None}}}{\#_{\text{None}}}$. Lower Δ suggests better human evaluation result.

part" refers to those that do not conform to usual programming habits or other unwritten specifications or rules. Programmers are able to notice such unusual parts easily. For instance, the identifier "worker" in Table 12 may be unusual, because the codes are written to invert the input sentences (see the "Problem description"), where the functionality has no connection with the identifier "worker." The volunteers are asked to find these unusual parts through their intuition.

The evaluation results are listed in Table 13. We can first see that all attack approaches (I-RW, MHM, and I-CARROT$_A$) decrease the match and the understandability score from the original examples (see the row of "None"). I-CARROT$_A$ outperforms I-RW baseline on all three metrics and produces comparable match and understandability score to MHM but better unusual count. In particular, the renaming operation from I-CARROT$_A$ is less noticeable to human beings, as the average unusual count is about 0.839, less than 1.019 of random I-RW and 0.965 of random-sampling-based MHM. These results suggest that incorporating gradient not only improves the effectiveness and efficiency of the attack algorithm (Section 6.3), but also makes the perturbed examples less distinguishable. A possible explanation is that with higher efficiency brought by the gradient guidance, the attack algorithm makes fewer manipulations upon the code, resulting in smaller perturbations.

Although the generated adversarial examples by I-CARROT$_A$ are less perceptible in terms of unusual counts than other approaches, they are still distinguishable for human beings, since in Table 13, the unusual counts of I-CARROT$_A$ are about twice that of "None" (the original examples). In fact, generating imperceptible adversarial examples from the perspectives of both compilers/executors (semantic equivalency) and human beings (unusual counts) is very challenging, because the algorithm must balance between the semantics and the literal meanings during transformation. As an early stage of studying the robustness of DL for source code processing, our work makes one step further to produce less distinguishable examples, and these results are the best we can do currently. We hope that this article may draw more efforts from the SE community and achieve the imperceptibility of adversarial examples for code from the perspective of human beings.

Table 14. Robustness Estimation of GRU, LSTM, and ASTNN

Model	Measurement	OJ		OJClone		CodeChef	
		$\tilde{R}$ (%)	Δ	$\tilde{R}$ (%)	Δ	$\tilde{R}$ (%)	Δ
GRU	RW	50.76	–	83.01	–	3.90	–
	MHM	26.41	1.9	73.32	1.1	0.24	16.3
	CARROT_M	1.36	**37.3**	46.19	**1.8**	0.00	**>500**
LSTM	RW	62.77	–	85.26	–	8.32	–
	MHM	40.81	1.5	77.01	1.1	0.31	26.8
	CARROT_M	1.40	**44.8**	47.58	**1.8**	0.00	**>500**
ASTNN	RW	61.53	–	95.31	–	13.00	–
	MHM	7.98	7.7	86.90	1.10	9.08	1.4
	CARROT_M	0.79	**77.9**	84.70	**1.13**	0.03	**433.3**

[†] $\Delta = \frac{\tilde{R}_{\text{RW}}}{\tilde{R}}$ gives robustness estimation improvement compared with random method.

Table 15. Robustness Estimation of LSCNN, TBCNN, CodeBERT, and CDLH

Model	Measurement	OJ		OJClone		CodeChef	
		$\tilde{R}$ (%)	Δ	$\tilde{R}$ (%)	Δ	$\tilde{R}$ (%)	Δ
LSCNN	RW	40.41	–	88.52	–	23.94	–
	MHM	51.03	0.8	77.57	**1.1**	31.34	0.8
	CARROT_M	6.15	**6.6**	84.17	1.0	0.07	**342.0**
TBCNN	RW	20.70	–	94.41	–	1.26	–
	MHM	30.81	0.7	54.94	**1.7**	17.26	0.1
	CARROT_M	14.64	**1.4**	90.84	1.0	1.26	**1.0**
CodeBERT	RW	51.13	–	Not Evaluated		21.84	–
	MHM	84.75	0.6			20.58	1.1
	CARROT_M	26.13	**2.0**			5.47	**4.0**
CDLH	RW	Not Evaluated		83.53	–	Not Evaluated	
	MHM			86.13	1.0		
	CARROT_M			64.22	**1.3**		

[†] $\Delta = \frac{\tilde{R}_{\text{RW}}}{\tilde{R}}$ gives robustness estimation improvement compared with random method.

Answer to RQ3: CARROT_A may be able to perform similar adversarial perturbations against LSTM on OJ, as the generated code matches the description well and is easy to understand. Furthermore, the renaming operations in I-CARROT_A are less distinguishable than other baseline approaches. However, the current I-CARROT_A is still perceptible compared to the natural original examples from the perspective of human beings, suggesting more efforts to be made in future work.

6.5 RQ4: Robustness Measurement

Robustness estimation by various techniques. Table 14 shows the robustness estimation of GRU, LSTM, and ASTNN by different methods. We can see that CARROT_M gives much tighter robustness upper bound estimation than the other methods across all settings. It indicates that the effectiveness of the attack method is quite important for accurate robustness estimation. In OJ, CARROT_M achieves more than 50 and 19 times estimation improvement on average compared with RW and MHM. Similar improvements could also be observed in OJClone and CodeChef. In particular, in OJ and CodeChef, CARROT_M estimates the robustness of GRU, LSTM, and ASTNN to be less than 2%, which are very tight bounds.

Table 15 shows the robustness estimation of LSCNN, TBCNN, CodeBERT, and CDLH by CARROT_M along with the RW and MHM baselines. In general, the bounds estimated by CARROT_M are on average 40 and 51 times tighter than RW and MHM. In particular, in CodeChef, CARROT_M estimates the robustness of LSCNN, TBCNN, and CodeBERT to be less than even 6%. It is peculiar that in OJ and CodeChef, the random-based RW is in general better than MHM. One possible reason is the small configuration of the candidate set (10), which makes MHM degenerate

Table 16. Robustness Resilience Gaining of LSTM and ASTNN Against Adversarial Perturbations by Adversarial Training

Task	Model	Adversarial Train Toolkit	Original Acc (%)	I-RW		S-RW		MHM		I-CARROT$_A$		S-CARROT$_A$	
				Acc (%)	Δ	Acc (%)	Δ	Acc (%)	Δ	Acc (%)	Δ	Acc (%)	Δ
OJ	LSTM	None	95.6	76.4	–	74.4	–	40.8	–	7.9	–	20.1	–
		MHM	95.8	85.1	1.11	74.7	1.00	56.8	1.39	19.1	2.42	19.5	0.97
		I-CARROT$_T$	95.7	85.9	**1.12**	77.7	1.04	64.4	**1.58**	29.4	3.72	24.3	1.21
		S-CARROT$_T$	95.4	78.7	1.03	86.9	1.17	43.7	1.07	9.3	1.18	53.8	**2.68**
		CARROT$_T$	95.5	85.6	**1.12**	87.7	**1.18**	61.5	1.51	30.1	**3.81**	52.7	2.62
	ASTNN	None	98.0	62.1	–	95.8	–	8.0	–	0.8	–	75.4	–
		MHM	98.2	86.5	**1.39**	94.6	0.99	76.3	**9.54**	6.3	**7.88**	75.4	1.00
		I-CARROT$_T$	98.1	86.2	**1.39**	95.9	1.00	71.8	8.98	5.6	7.00	82.5	1.09
		S-CARROT$_T$	98.2	64.1	1.03	97.2	**1.01**	31.2	3.90	0.9	1.13	93.5	**1.24**
		CARROT$_T$	98.3	84.5	1.36	97.2	**1.01**	65.0	8.13	4.2	5.25	93.8	**1.24**
			Original F1 (%)	F1 (%)	Δ	F1 (%)	Δ	F1 (%)	Δ	F1 (%)	Δ	F1 (%)	Δ
OJClone	LSTM	None	90.7	23.0	–	66.3	–	5.0	–	1.3	–	14.9	–
		MHM	92.2	43.9	1.91	71.1	1.07	10.3	2.06	1.6	1.23	12.9	0.87
		I-CARROT$_T$	92.7	58.1	**2.53**	66.6	1.00	17.2	**3.44**	5.8	**4.46**	13.0	0.87
		S-CARROT$_T$	84.1	13.9	0.60	69.7	1.05	2.6	0.52	0.3	0.23	21.5	1.44
		CARROT$_T$	92.1	37.4	1.63	79.3	**1.20**	9.7	1.94	2.6	2.00	34.3	**2.30**
	ASTNN	None	94.6	70.5	–	65.4	–	8.4	–	40.7	–	34.4	–
		MHM	86.0	75.3	1.07	78.2	1.20	33.6	**4.00**	37.6	0.92	46.5	1.35
		I-CARROT$_T$	89.3	67.9	0.96	68.0	1.04	13.3	1.58	50.1	**1.23**	46.7	1.36
		S-CARROT$_T$	94.5	78.5	**1.11**	92.2	**1.41**	7.1	0.85	38.5	0.95	79.9	**2.32**
		CARROT$_T$	93.8	78.2	**1.11**	87.4	1.34	19.5	2.32	49.3	1.21	76.2	2.22

[$\diamond$] $\Delta = \frac{\text{Acc}}{\text{Acc}_{\text{None}}}$ or $\Delta = \frac{\text{F1}}{\text{F1}_{\text{None}}}$, indicating the resilience of the model against the attack algorithm.

to total randomness. Another exception lies in OJClone, as MHM performs better than CARROT$_M$ in LSCNN and TBCNN. This issue is related to the discussions in RQ2.

Robustness of the DL models. We compare the robustness of different models estimated by CARROT$_M$. The results indicate that the robustness of different DL models using different architecture might render differences in their robustness. For example, ASTNN might be more robust than GRU and LSTM, as ASTNN gives similar $\tilde{R}$ to GRU and LSTM in OJ and CodeChef, but gives much higher robustness estimation in OJClone. It is reasonable, because ASTNN takes ASTs as inputs, which may contain more structural information about the snippet. Also, CodeBERT may be more robust than LSCNN and TBCNN, as $\tilde{R}$ of CodeBERT is much higher than LSCNN and TBCNN. It is sensible, as CodeBERT has more parameters and is pre-trained upon larger corpus. Models in OJ and CodeChef may not be robust, as the true robustness is less than 10% in most cases, while models in OJClone may be much more robust, as $\tilde{R}$ by CARROT$_M$ is above 40%. This finding suggests that besides DL architectures, non-robust issues may also be highly related to the datasets. Similar findings are also discussed in the context of image processing by Goodfellow et al. [34].

Answer to RQ4: We demonstrate that CARROT$_M$ is capable to estimate a tight bound of the true robustness of the DL models. In addition, the robustness issue might also be relevant to the task dataset and the DL architecture, which should be paid attention to by the DL researchers and practitioners for source code processing.

6.6 RQ5: Robustness Ehancement

Resilience against adversarial attacks. Table 16 shows the robust resilience (obtained by different attacks) of the DL models under different adversarial training toolkit. We adopt MHM, I-CARROT$_T$, S-CARROT$_T$, and CARROT$_T$ (see Column 3) to perform adversarial training on LSTM and ASTNN in OJ and OJClone. Then, the robustness of the corresponding obtained models are estimated by different adversarial attacks (see Columns 5–14). Overall, adversarial training often

Table 17. Robustness Improvement of the Subject Models After Adversarial Training

Task	Model	Adversarial Train Toolkit	$\tilde{R}_{\text{CARROT}_\text{M}}$ (%)	Improve
OJ	LSTM	None	1.4	–
		MHM	3.8	×2.72
		I-CARROT$_\text{T}$	7.2	×5.13
		S-CARROT$_\text{T}$	6.1	×4.37
		CARROT$_\text{T}$	19.4	**×13.77**
	ASTNN	None	0.8	–
		MHM	6.0	**×7.61**
		I-CARROT$_\text{T}$	5.3	×6.73
		S-CARROT$_\text{T}$	0.9	×1.17
		CARROT$_\text{T}$	4.2	×5.32
OJClone	LSTM	None	47.6	–
		MHM	55.6	×1.17
		I-CARROT$_\text{T}$	62.1	**×1.31**
		S-CARROT$_\text{T}$	44.4	×0.93
		CARROT$_\text{T}$	55.9	×1.18
	ASTNN	None	84.7	–
		MHM	83.8	×0.99
		I-CARROT$_\text{T}$	91.2	**×1.08**
		S-CARROT$_\text{T}$	84.9	×1.00
		CARROT$_\text{T}$	90.1	×1.06

enables the improvement of the DL model resilience, to a different extent. For example, CARROT$_\text{T}$ improves the model robust resilience against all attack algorithms, i.e., OJ LSTM to 2.0 times, OJ ASTNN to 3.4 times, OJClone LSTM to 1.8 times, and OJClone ASTNN to 1.6 times, averaged across all attacks. However, other methods might not be able to improve the robustness across all settings. In particular, adversarial training with only token-level perturbations (i.e., MHM and I-CARROT$_\text{T}$) might not be resilience against statement-level perturbations (i.e., S-RW and S-CARROT$_\text{A}$), and vice versa. For example, in the case of OJClone LSTM, the performance even decreases to 87% after adversarial training (MHM) attacked by S-CARROT$_\text{A}$. Therefore, to be robust against diverse potential perturbations in practice, the inclusion of diverse perturbations is important. CARROT$_\text{T}$ is designed to be extensible, where new perturbation mutators could be easily integrated. We leave the inclusion of more advanced mutators in future work.

Robustness improvement. Table 17 shows the robustness improvement estimated by CARROT$_\text{M}$. Sharing the same configuration as Table 16, we estimate the robustness by CARROT$_\text{M}$ (equipped with I-CARROT$_\text{A}$ and S-CARROT$_\text{A}$) of DL models before and after adversarial training. We could observe that, on average, the robustness of the DL models after adversarial training by CARROT$_\text{T}$ increases to 5.3 times, higher than MHM in most cases, which has 3.1 times improvement on average. For example, the robustness of LSTM in OJ after CARROT$_\text{T}$ even increases from 1.4% to 19.4% (13.8 times).

Answer to RQ5: CARROT$_\text{T}$ framework, especially equipped with the CARROT$_\text{A}$ toolbox, is helpful for improving the robustness of DL models for source code. The results also indicate that the inclusion of diverse perturbation mutators could be helpful for improving robustness (by adversarial training) to be resilient against various perturbations in practice.

6.7 Threat to Validity

The subject dataset selection could be a threat to validity. We counteract it by selecting tasks and datasets included in the CodeXGLUE benchmark [59], which are important and widely utilized. Another threat could be the victim model selection. We counteract it by considering the representativeness and the performance. We mainly study the most widely adopted and representative GRU and LSTM (equipped with attention), which is the backbone of various of DL models for SE, and we also select ASTNN, which is one of the SOTA models for functionality classification and clone detection. Besides, we also evaluate adversarial attack approaches against many other models, including the classic LSCNN and TBCNN models, the pre-trained CodeBERT model, and CDLH specially designed for clone detection. The subject models cover sequential models (GRU and LSTM), hierarchical models (LSCNN), tree-based models (ASTNN and TBCNN), transformer-base

pre-trained models (CodeBERT), and architectures specially designed for certain tasks (CDLH). We also counteract it by following the guidance of the authors and the instructions in the original papers [42, 62, 85]. We have tried our best to build and train the parameters in our experiments, and the performance of our model is comparable with the original papers during our early-stage experiments upon their datasets. Furthermore, the code transformation operation selection could also be a threat to validity. We design the mutators by employing transformations in different levels (token and statement) from the previous work [50, 93]. In human evaluation, the composition of the volunteers could be a threat, too. We counteract it by choosing graduate and undergraduate students majoring in computer science, with at least three years C/C++ programming and software development experience. The results that I-CARROT$_A$ is still distinguishable in human evaluation could also be a threat. We counteract this by comparing our method with other baselines. Generating imperceptible code adversarial examples for human beings with semantic equivalent constraints is still quite challenging at this moment and could be an important research direction in the future. We have also tried our best to design our method, which obtains better results and outperforms other existing approaches. We hope to propose novel techniques to continuously address this challenge in our future work. At last, the randomness could also be a threat. We counteract it by repeating the adversarial attack on a 20% uniformally sampled subset of the test set five times and report the average results.

7 CONCLUSION & DISCUSSION

In this article, we propose CARROT, a general framework for robustness detection (CARROT$_A$), measurement (CARROT$_M$) and enhancement (CARROT$_T$) for DL models in the context of source code processing. We demonstrate the potential risk within the DL models for source code processing. Specifically, we find that all the subject models for evaluation cannot handle the semantic equivalent perturbations very well. Our in-depth evaluations confirm the effectiveness and efficiency of CARROT$_A$ in adversarial attack and the usefulness of CARROT$_T$ in adversarial training. The results also indicate that our proposed robustness metrics $\tilde{R}$ serve as a suitable robustness estimation.

At last, we provide several suggestions for developers to hopefully improve the robustness of source code processing systems: ❶ Do a rough human review before feeding the code into the DL model. Programmers are still capable to distinguish the adversarial examples in a short time, as the human evaluation results suggest. ❷ Normalize the source code appropriately during preprocessing. For instance, a DL model trained upon a normalized dataset that renames the first identifier as "var1," the second as "var2," and so on, may resist I-CARROT$_A$ for sure, since the renaming perturbations by I-CARROT$_A$ are removed during the normalization (pre-processing). In this concrete suggestion, normalization refers to removing the known certain kinds of noises and perturbations. To gain robustness against certain known kinds of adversarial attack, one of the easiest ways is to normalize all noises and perturbations on this level directly, as the above example illustrates. However, it is risk-taking, because such normalization may also remove those important features and may decrease the performance of the DL model greatly. It would be a tradeoff between robustness against specific perturbations and the evaluation performance. ❸ Adversarially train model with diverse kinds of perturbations. The adversarial training results have demonstrated the resilience gaining.

As an early step to study the robustness of the DL models for source code, our work confirms that the SOTA DL models for source code processing may not be robust-resilient under simple random perturbations. Considering the code variants often exist in practice software development, besides high accuracy, we call for software engineering researchers and practitioners' attention to

consider robustness when designing new DL solutions for source code processing. We have made our framework publicly available and hope it would benefit the SE community towards building more robust DL solutions for various SE tasks in the era of big code.

8 ACKNOWLEDGEMENTS

We would like to thank Wenhan Wang from Nanyang Technological University for his help in the implementation of the subject DL models. We also would like to thank all the reviewers and the editor for their long-term support to our work and constructive suggestions to this article.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.* 51, 4 (2018), 81:1–81:37. DOI: https://doi.org/10.1145/3212695

[2] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33rd International Conference on Machine Learning*. JMLR.org, 2091–2100. Retrieved from http://proceedings.mlr.press/v48/allamanis16.html.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net. Retrieved from https://openreview.net/forum?id=H1gKYo09tX.

[4] 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. DOI: https://doi.org/10.1145/3290353

[5] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani B. Srivastava, and Kai-Wei Chang. 2018. Generating natural language adversarial examples. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2890–2896. DOI: https://doi.org/10.18653/v1/d18-1316

[6] Genevieve Arboit. 2002. A method for watermarking Java programs via opaque predicates. In *Proceedings of the 5th International Conference on Electronic Commerce Research (ICECR'02)*. Citeseer, 102–110.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations*. Retrieved from http://arxiv.org/abs/1409.0473.

[8] Battista Biggio, Igino Corona, Blaine Nelson, Benjamin I. P. Rubinstein, Davide Maiorca, Giorgio Fumera, Giorgio Giacinto, and Fabio Roli. 2014. Security evaluation of support vector machines in adversarial environments. *CoRR* abs/1401.7727 (2014).

[9] Battista Biggio, Giorgio Fumera, and Fabio Roli. 2010. Multiple classifier systems for robust classifier design in adversarial environments. *Int. J. Mach. Learn. Cybern.* 1, 1-4 (2010), 27–41. DOI: https://doi.org/10.1007/s13042-010-0007-7

[10] Battista Biggio, Giorgio Fumera, and Fabio Roli. 2017. Security evaluation of pattern classifiers under attack. *CoRR* abs/1709.00609 (2017).

[11] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2011. Support vector machines under adversarial label noise. In *Proceedings of the 3rd Asian Conference on Machine Learning*. JMLR.org, 97–112. Retrieved from http://proceedings.mlr.press/v20/biggio11/biggio11.pdf.

[12] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recog.* 84 (2018), 317–331. DOI: https://doi.org/10.1016/j.patcog.2018.07.023

[13] Michael Brückner, Christian Kanzow, and Tobias Scheffer. 2012. Static prediction games for adversarial learning problems. *J. Mach. Learn. Res.* 13 (2012), 2617–2654. Retrieved from http://dl.acm.org/citation.cfm?id=2503326.

[14] Sébastien Bubeck, Yin Tat Lee, Eric Price, and Ilya P. Razenshteyn. 2019. Adversarial examples from computational constraints. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 831–840. Retrieved from http://proceedings.mlr.press/v97/bubeck19a.html.

[15] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. 2017. Piecewise linear neural network verification: A comparative study. *CoRR* abs/1711.00455 (2017).

[16] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B. Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2020. Extracting training data from large language models. *CoRR* abs/2012.07805 (2020).

[17] Nicholas Carlini and David A. Wagner. 2017. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 39–57. DOI: https://doi.org/10.1109/SP.2017.49

[18] Nicholas Carlini and David A. Wagner. 2018. Audio adversarial examples: Targeted attacks on speech-to-text. In *Proceedings of the IEEE Security and Privacy Workshops*. IEEE Computer Society, 1–7. DOI: https://doi.org/10.1109/SPW.2018.00009

[19] Jien-Tsai Chan and Wuu Yang. 2004. Advanced obfuscation techniques for java bytecode. *J. Syst. Softw.* 71, 1–2 (2004), 1–10. DOI: https://doi.org/10.1016/S0164-1212(02)00066-3

[20] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Comput. Surv.* 53, 1 (2020), 4:1–4:36. DOI: https://doi.org/10.1145/3363562

[21] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. 2017. Maximum resilience of artificial neural networks. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (Lecture Notes in Computer Science)*, Deepak D'Souza and K. Narayan Kumar (Eds.), Vol. 10482. Springer, 251–268. DOI: https://doi.org/10.1007/978-3-319-68167-2_18

[22] Siddhartha Chib and Edward Greenberg. 1995. Understanding the Metropolis-Hastings algorithm. *Amer. Statist.* 49, 4 (1995), 327–335.

[23] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A Taxonomy of Obfuscating Transformations*. Technical Report. Citeseer.

[24] Saad M. Darwish, Shawkat K. Guirguis, and Mohamed S. Zalat. 2010. Stealthy code obfuscation technique for software security. In *Proceedings of the International Conference on Computer Engineering & Systems*. IEEE, 93–99.

[25] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41. DOI: https://doi.org/10.1109/C-M.1978.218136

[26] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. DeepStellar: Model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 477–487. DOI: https://doi.org/10.1145/3338906.3338954

[27] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2018. HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 31–36. DOI: https://doi.org/10.18653/v1/P18-2006

[28] 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 *Proceedings of the Conference on Empirical Methods in Natural Language Processing: Findings*, Trevor Cohn, Yulan He, and Yang Liu (Eds.), Vol. EMNLP 2020. Association for Computational Linguistics, 1536–1547. DOI: https://doi.org/10.18653/v1/2020.findings-emnlp.139

[29] BooFuzz Framework. 2021. Retrieved from https://github.com/jtpereyda/boofuzz.

[30] Sulley Fuzzing Framework. 2017. Retrieved from https://github.com/OpenRCE/sulley.

[31] Gordon Fraser and Andrea Arcuri. 2015. Achieving scalable mutation-based generation of whole test suites. *Empir. Softw. Eng.* 20, 3 (2015), 783–812. DOI: https://doi.org/10.1007/s10664-013-9299-z

[32] Milos Gligoric, Lingming Zhang, Cristiano Pereira, and Gilles Pokam. 2013. Selective mutation testing for concurrent code. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 224–234. DOI: https://doi.org/10.1145/2483760.2483773.

[33] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 50–59. DOI: https://doi.org/10.1109/ASE.2017.8115618

[34] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *Proceedings of the 3rd International Conference on Learning Representations*. Retrieved from http://arxiv.org/abs/1412.6572.

[35] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642. DOI: https://doi.org/10.1145/2950290.2950334

[36] Rahul Gupta, Aditya Kanade, and Shirish K. Shevade. 2019. Neural attribution for semantic bug-localization in student programs. In *Proceedings of the Conference on Neural Information Processing Systems*. 11861–11871. Retrieved from http://papers.nips.cc/paper/9358-neural-attribution-for-semantic-bug-localization-in-student-programs.

[37] Richard G. Hamlet. 1977. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.* 3, 4 (1977), 279–290. DOI: https://doi.org/10.1109/TSE.1977.231145

[38] W. K. Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57, 1 (1970), 97–109.

[39] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 152–162. DOI: https://doi.org/10.1145/3236024.3236051

[40] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 200–210. DOI: https://doi.org/10.1145/3196321.3196334

[41] Po-Sen Huang, Robert Stanforth, Johannes Welbl, Chris Dyer, Dani Yogatama, Sven Gowal, Krishnamurthy Dvijotham, and Pushmeet Kohli. 2019. Achieving verified robustness to symbol substitutions via interval bound propagation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, 4081–4091. DOI: https://doi.org/10.18653/v1/D19-1419

[42] Xuan Huo and Ming Li. 2017. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. ijcai.org, 1909–1915. DOI: https://doi.org/10.24963/ijcai.2017/265

[43] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *CoRR* abs/1909.09436 (2019).

[44] Robin Jia and Percy Liang. 2017. Adversarial examples for evaluating reading comprehension systems. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2021–2031. DOI: https://doi.org/10.18653/v1/d17-1215

[45] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proceedings of the 29th International Conference on Computer-aided Verification (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10426. Springer, 97–117. DOI: https://doi.org/10.1007/978-3-319-63387-9_5

[46] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. 1039–1049.

[47] Marius Kloft and Pavel Laskov. 2012. Security analysis of online centroid anomaly detection. *J. Mach. Learn. Res.* 13 (2012), 3681–3724. Retrieved from http://dl.acm.org/citation.cfm?id=2503359.

[48] Ching-Yun Ko, Zhaoyang Lyu, Lily Weng, Luca Daniel, Ngai Wong, and Dahua Lin. 2019. POPQORN: Quantifying robustness of recurrent neural networks. In *Proceedings of the 36th International Conference on Machine Learning. (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 3468–3477. Retrieved from http://proceedings.mlr.press/v97/ko19a.html.

[49] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *Proceedings of the 5th International Conference on Learning Representations*. OpenReview.net. Retrieved from https://openreview.net/forum?id=HJGU3Rodl.

[50] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 216–226. DOI: https://doi.org/10.1145/2594291.2594334

[51] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 386–399. DOI: https://doi.org/10.1145/2814270.2814319

[52] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. 2020. DeepCommenter: A deep code comment generation tool with hybrid lexical and syntactical information. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1571–1575. DOI: https://doi.org/10.1145/3368089.3417926

[53] Jian Li, Yue Wang, Irwin King, and Michael R. Lyu. 2017. Code completion with neural attention and pointer networks. *CoRR* abs/1711.09573 (2017).

[54] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: A survey. *Cybersecurity* 1, 1 (2018). DOI: https://doi.org/10.1186/s42400-018-0002-y

[55] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 65–76. DOI: https://doi.org/10.1145/2737924.2737986

[56] Wang Lin, Zhengfeng Yang, Xin Chen, Qingye Zhao, Xiangkun Li, Zhiming Liu, and Jifeng He. 2019. Robustness verification of classification deep neural networks via linear programming. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. Computer Vision Foundation/IEEE, 11418–11427. DOI: https://doi.org/10.1109/CVPR.2019.01168

[57] Cullen Linn and Saumya K. Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*. ACM, 290–299. DOI: https://doi.org/10.1145/948109.948149

[58] Douglas Low. 1998. Protecting Java code via code obfuscation. *XRDS* 4, 3 (1998), 21–23. DOI: https://doi.org/10.1145/332084.332092

[59] 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).

[60] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 120–131. DOI: https://doi.org/10.1145/3238147.3238202

[61] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *J. Chem. Phys.* 21, 6 (1953), 1087–1092.

[62] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. AAAI Press, 1287–1293. Retrieved from http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775.

[63] Blaine Nelson, Benjamin I. P. Rubinstein, Ling Huang, Anthony D. Joseph, Steven J. Lee, Satish Rao, and J. D. Tygar. 2012. Query strategies for evading convex-inducing classifiers. *J. Mach. Learn. Res.* 13 (2012), 1293–1332. Retrieved from http://dl.acm.org/citation.cfm?id=2343688.

[64] Rainer Niedermayr, Elmar Jürgens, and Stefan Wagner. 2016. Will my tests tell me if I break this code? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*. ACM, 23–29. DOI: https://doi.org/10.1145/2896941.2896944

[65] Augustus Odena, Catherine Olsson, David G. Andersen, and Ian J. Goodfellow. 2019. TensorFuzz: Debugging neural networks with coverage-guided fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 4901–4911. Retrieved from http://proceedings.mlr.press/v97/odena19a.html.

[66] Jeff Offutt and Wuzhi Xu. 2004. Generating test cases for web services using data perturbation. *ACM SIGSOFT Softw. Eng. Notes* 29, 5 (2004), 1–10. DOI: https://doi.org/10.1145/1022494.1022529

[67] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *Proceedings of the IEEE European Symposium on Security and Privacy*. IEEE, 372–387. DOI: https://doi.org/10.1109/EuroSP.2016.36

[68] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18. DOI: https://doi.org/10.1145/3132747.3132785

[69] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. 2007. Binary obfuscation using signals. In *Proceedings of the 16th USENIX Security Symposium*. USENIX Association. Retrieved from https://www.usenix.org/conference/16th-usenix-security-symposium/binary-obfuscation-using-signals.

[70] Yao Qin, Nicholas Carlini, Garrison W. Cottrell, Ian J. Goodfellow, and Colin Raffel. 2019. Imperceptible, robust, and targeted adversarial examples for automatic speech recognition. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 5231–5240. Retrieved from http://proceedings.mlr.press/v97/qin19a.html.

[71] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR* abs/1711.04596 (2017).

[72] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2008. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society. Retrieved from https://www.ndss-symposium.org/ndss2008/impeding-malware-analysis-using-conditional-code-obfuscation/.

[73] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 849–863. DOI: https://doi.org/10.1145/2983990.2984038

[74] Sining Sun, Ching-Feng Yeh, Mari Ostendorf, Mei-Yuh Hwang, and Lei Xie. 2018. Training augmentation with adversarial examples for robust speech recognition. In *Proceedings of the 19th Annual Conference of the International Speech Communication Association*. ISCA, 2404–2408. DOI: https://doi.org/10.21437/Interspeech.2018-1247

[75] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. 109–119.

[76] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *Proceedings of the 2nd International Conference on Learning Representations*. Retrieved from http://arxiv.org/abs/1312.6199.

[77] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*. The Association for Computer Linguistics, 1556–1566. DOI: https://doi.org/10.3115/v1/p15-1150

[78] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 303–314. DOI: https://doi.org/10.1145/3180155.3180220

[79] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Annual Conference on Neural Information Processing Systems*. 5998–6008. Retrieved from https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[80] Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry. 2018. Descartes: A PITest engine to detect pseudo-tested methods: Tool demonstration. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 908–911. DOI: https://doi.org/10.1145/3238147.3240474

[81] Dilin Wang, ChengYue Gong, and Qiang Liu. 2019. Improving neural language modeling via adversarial training. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 6555–6565. Retrieved from http://proceedings.mlr.press/v97/wang19f.html.

[82] Shen Wang, Zhengzhang Chen, Xiao Yu, Ding Li, Jingchao Ni, Lu-An Tang, Jiaping Gui, Zhichun Li, Haifeng Chen, and Philip S. Yu. 2019. Heterogeneous graph matching networks for unknown malware detection. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. ijcai.org, 3762–3770. DOI: https://doi.org/10.24963/ijcai.2019/522

[83] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 261–271. DOI: https://doi.org/10.1109/SANER48275.2020.9054857

[84] Xinran Wang, Yu Xiang, Jun Gao, and Jie Ding. 2020. Information laundering for model privacy. *CoRR* abs/2009.06112 (2020).

[85] 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 26th International Joint Conference on Artificial Intelligence*. ijcai.org, 3034–3040. DOI: https://doi.org/10.24963/ijcai.2017/423

[86] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane S. Boning, and Inderjit S. Dhillon. 2018. Towards fast computation of certified robustness for ReLU networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer G. Dy and Andreas Krause (Eds.), Vol. 80. PMLR, 5273–5282. Retrieved from http://proceedings.mlr.press/v80/weng18a.html.

[87] Fuzzing with Spike. 2017. Retrieved from https://samsclass.info/127/proj/p18-spike.htm.

[88] Eric Wong and J. Zico Kolter. 2018. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer G. Dy and Andreas Krause (Eds.), Vol. 80. PMLR, 5283–5292. Retrieved from http://proceedings.mlr.press/v80/wong18a.html.

[89] Gregory Wroblewski. 2002. General method of program code obfuscation. (2002).

[90] Weiming Xiang, Patrick Musau, Ayana A. Wild, Diego Manzanas Lopez, Nathaniel Hamilton, Xiaodong Yang, Joel A. Rosenfeld, and Taylor T. Johnson. 2018. Verification for machine learning, autonomy, and neural networks survey. *CoRR* abs/1810.01989 (2018).

[91] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: A coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 146–157. DOI: https://doi.org/10.1145/3293882.3330579

[92] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu. 2017. On secure and usable program obfuscation: A survey. *CoRR* abs/1710.01139 (2017).

[93] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*.

[94] Huangzhao Zhang, Hao Zhou, Ning Miao, and Lei Li. 2019. Generating fluent adversarial examples for natural languages. In *Proceedings of the 57th Conference of the Association for Computational Linguistics*. Association for Computational Linguistics, 5564–5569. DOI: https://doi.org/10.18653/v1/p19-1559

[95] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE/ACM, 783–794. DOI: https://doi.org/10.1109/ICSE.2019.00086

[96] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Trans. Softw. Eng.* (2020). DOI: https://doi.org/10.1109/TSE.2019.2962027

[97] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 132–142. DOI: https://doi.org/10.1145/3238147.3238187

[98] Jianyi Zhou, Feng Li, Jinhao Dong, Hongyu Zhang, and Dan Hao. 2020. Cost-effective testing of a deep learning model through input reduction. In *Proceedings of the 31st IEEE International Symposium on Software Reliability Engineering*. IEEE, 289–300. DOI: https://doi.org/10.1109/ISSRE5003.2020.00035

[99] William Zhu and Clark Thomborson. 2005. A provable scheme for homomorphic obfuscation in software security. In *Proceedings of the IASTED International Conference on Communication, Network and Information Security*. 208–212.

Received October 2020; revised November 2021; accepted November 2021