



Discrete Adversarial Attack to Models of Code

FENGJUAN GAO*, Nanjing University of Science and Technology, China

YU WANG*, Nanjing University, China

KE WANG, Visa Research, USA

The pervasive brittleness of deep neural networks has attracted significant attention in recent years. A particularly interesting finding is the existence of adversarial examples, imperceptibly perturbed natural inputs that induce erroneous predictions in state-of-the-art neural models. In this paper, we study a different type of adversarial examples specific to code models, called *discrete adversarial examples*, which are created through program transformations that preserve the semantics of original inputs. In particular, we propose a novel, general method that is highly effective in attacking a broad range of code models. From the defense perspective, our primary contribution is a theoretical foundation for the application of adversarial training – the most successful algorithm for training robust classifiers – to defending code models against discrete adversarial attack. Motivated by the theoretical results, we present a simple realization of adversarial training that substantially improves the robustness of code models against adversarial attacks in practice.

We extensively evaluate both our attack and defense methods. Results show that our discrete attack is significantly more effective than state-of-the-art whether or not defense mechanisms are in place to aid models in resisting attacks. In addition, our realization of adversarial training improves the robustness of all evaluated models by the widest margin against state-of-the-art adversarial attacks as well as our own.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Computing methodologies** → **Neural networks**.

Additional Key Words and Phrases: Discrete Adversarial Attack, Adversarial Training, Models of code

ACM Reference Format:

Fengjuan Gao, Yu Wang, and Ke Wang. 2023. Discrete Adversarial Attack to Models of Code. *Proc. ACM Program. Lang.* 7, PLDI, Article 113 (June 2023), 24 pages. <https://doi.org/10.1145/3591227>

1 INTRODUCTION

While deep neural networks have achieved state-of-the-art performance in a variety of application domains such as image classification [He et al. 2016; Krizhevsky et al. 2012] and natural language processing [Devlin et al. 2019; Sutskever et al. 2014; Vaswani et al. 2017], they are found to be vulnerable to adversarial attacks: tiny changes that are typically imperceptible to humans can flummox the best neural networks around. Szegedy et al. [2014] are the first to discover the existence of adversarial examples in image classification field, in particular, they show that applying an imperceptible, systematic perturbation to a test image can arbitrarily change the network's prediction. Later, more powerful attack methods have been proposed [Carlini and Wagner 2017;

*Both authors contributed equally to this work.

Authors' addresses: Fengjuan Gao, School of Computer Science and Engineering, Nanjing University of Science and Technology, China, fjgao@njust.edu.cn; Yu Wang, State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, China, yuwang_cs@nju.edu.cn; Ke Wang, Visa Research, USA, kewang@visa.com.



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

© 2023 Copyright held by the owner/author(s).

2475-1421/2023/6-ART113

<https://doi.org/10.1145/3591227>

[Goodfellow et al. 2015; Moosavi-Dezfooli et al. 2016], which also expand the scope of attack to other domains than image classification (e.g., natural language processing [Alzantot et al. 2018]).

In the same spirit of aforementioned works, this paper studies a different type of adversarial attack specific to code models, called *discrete adversarial attack*, where adversarial examples are created through program transformations that preserve the semantics of original inputs. By code models (or models of code), we mean machine learning models learned from code written in high-level programming languages like Java, C#. Moreover, this paper only considers code models that predict properties that strictly adhere to the semantics of input programs. In general, a property is semantically-adhering if it evaluates the same way for a program p and all other programs that are semantically equivalent to p (Section 2 presents the formalization). While the vast majority of code models fit in this category (e.g., bug detection [Allamanis et al. 2018; Wang et al. 2020], semantic labeling [Alon et al. 2019a,b; Feng et al. 2020]), there exist models that do not satisfy this constraint, for example, in authorship attribution [Li et al. 2022; Quiring et al. 2019], models are learned to predict the author of a code snippet, corresponding to a stylistic property that can vary from one program p_1 to another p_2 even if p_1 and p_2 are semantically equivalent. Attacking models that predict non-semantically-adhering properties can create a tricky issue: how to determine the ground-truth label of adversarial examples. We illustrate this point using an attack scenario in above-mentioned authorship attribution. After Alice has written a program p , Bob injects his own stylometric features into p hoping to trick authorship attribution models to predict that he instead is the author of p , such that he can (wrongfully) claim the credit of Alice's work. Now a question naturally arises: what is the ground-truth label of the resultant program \hat{p} of Bob's changes. One may argue the author could still be Alice if Bob only made tiny, trivial modifications to p , or Bob if he changed p so much that \hat{p} now resembles his own stylistic patterns, or neither if Bob changed one part of p and left the rest intact, as a result, \hat{p} accidentally renders the coding style of a third person. All in all, the ground-truth label of \hat{p} can not be determined, thus, in this case, the correctness of model behavior can not be defined, ultimately, even a successful attack does not suffice to prove the vulnerability of code models. Recall the example above, even if Bob managed to make models predict himself to be the author of \hat{p} , his successful attack does not entail the vulnerability of models because, as we explained, there lacks a systematic procedure specifying what models should predict for \hat{p} . On the contrary, this paper is only concerned with adversarial attacks that can validate the vulnerability of code models. To this end, we attack exclusively models that predict semantically-adhering properties, in particular, our attack adopts a simple methodology: find an adversarial example \hat{p} that is semantically equivalent to the original input p such that models predict a different label for \hat{p} than p . Under this attack framework, every successful adversarial example is a sufficient proof of the vulnerability of code models against adversarial attacks. This is because the ground-truth label of adversarial examples is well-defined (which is the same as the original input since it is a semantically-adhering property that does not vary with semantically-preserving transformations), therefore, if predicted labels for adversarial examples and original inputs differ, then models are indeed the party at fault.

Conceptually, discrete adversarial attack addresses a fundamental limitation of classical adversarial attack, which assumes the continuous, end-to-end, differentiability of neural networks such that the gradient of the loss function can be directly used to manipulate original inputs for creating adversarial examples. It is clear that the assumption does not hold for code models because their loss is not continuously differentiable *w.r.t.* programs (i.e., discrete tokens) the way the loss of image classification models is *w.r.t.* pixels (i.e., integer values). To address this issue, DAMP [Yefet et al. 2020] leverages the partial differentiability existed in code models (e.g., from the loss function to the initial layer that embeds a program into a one-hot vector) in an attempt to create an adversarial version of input programs in the embedding space. Despite a notable step forward, projecting an

adversarial embedding back to a concrete adversarial program remains a tremendous challenge. As a result, DAMP only considers transformations pertaining to variable names to ease the projection process (e.g., changing existing variables into adversarial names or adding new variables with adversarial names). Apart from the differentiability issue of the classical adversarial attack, discrete adversarial attack enjoys another crucial advantage: it does not require the access to model parameters, therefore, by nature it is a black-box approach that is more generally applicable.

A Method of Discrete Adversarial Attack. In this paper, we propose a novel, general method of Discrete adversarial attack, namely DaK that is highly effective in attacking a wide range of code models. It is worth mentioning we focus exclusively on the targeted attack where adversaries attempt to fool models to predict a particular class (also called target label) other than the ground-truth. This is a challenging problem due to the vast space of programs that are semantically equivalent to the original input and the stringent requirement that semantically equivalent programs must make model predict a particular label of adversary's choice. Therefore, a naive approach of enumerating model predictions for each program in this enormous space would be infeasible. On the contrary, untargeted attack — where any prediction other than the ground-truth counts as a successful attack — is a relatively easy problem to which prior works [Yefet et al. 2020; Zhang et al. 2022; Zhou et al. 2022] already offer decent solutions. Our attack consists of three major components: a *Destroyer*, a *Finder* and a *Merger*. The *Destroyer* is responsible for erasing, in the program under attack, significant features that models could use for prediction; next, the *Finder* aims to discover critical features of another program for which models predict the target label; finally, the *Merger* inserts the discovered critical features into the destroyed program to generate discrete adversarial examples. Our intuition is that after being dealt with by the *Destroyer*, features of the program under attack are substantially weakened so that they are no longer capable of emitting powerful signals for models to make confident predictions; in the meanwhile, critical features from the other program is arguably the strongest, most concentrated set of features to make models predict the target label. Therefore, the combination of the critical features and the destroyed program should make a convincing discrete adversarial example for models to predict the target label. In the following, we demonstrate the usefulness of our attack method, in particular, we discuss its implications in the real-world.

Implications of Discrete Adversarial Attack. Considering that learning-based bug detectors have been increasingly integrated into the development pipeline, aiding programmers to write high-quality code [Raychev et al. 2015], the correctness and reliability of those bug detectors, especially against adversarial attacks will be a key factor to their success. In fact, we show that our discrete attack causes a Gated Graph Neural Network [Li et al. 2016] (GGNN) model to miss bugs. Figure 1a shows a GGNN model trained on VARMISUSE task [Allamanis et al. 2018] spotted `presetRead` at line 90 as the wrong variable to use, and suggested correctly `alreadyRead` as the fix. However, by transforming the program to Figure 1b in a strictly semantically-preserving manner, the same model now considers `presetRead` the correct variable to use. Considering that our transformations do not in any way affect the validity of the code (i.e., the new program still compiles), our attack leaves a real bug in function `ProcessRecord`. What's more, this error may allow the ensuing `read` operation (line 93 in Figure 1b) to trigger a buffer overflow (when `presetRead ≤ total/2` and `total % presetRead ≠ 0`), the kind of bugs frequently cause software security vulnerabilities, especially for languages without built-in safety mechanisms (e.g., automatic bounds checking). In general, the susceptibility of bug detection models to adversarial attack invites a variety of malicious behaviors. For example, malicious developers inside an organization or an open-source project can hide bugs in their codebase by writing the buggy code in a specific manner that bypasses bug detection tools. Furthermore, adversarial attacks to other kinds of code models may lead to even more serious

<pre> 1 protected override void ProcessRecord(long presetRead) { ... 26 foreach (ContentHolder holder in contentStreams) { 27 long total = holder.Reader.Count; 28 long alreadyRead = 0; // alreadyRead records the number of items that have been retrieved so far ... 87 IList results = null; 88 do { /* actualRead/presetRead is the number of content items to be retrieved in each iteration */ 89 long actualRead = presetRead; /* (total - actualRead < #) is intended for safeguarding the ensuing read access at line 92 against a potential buffer overflow, thus the correct variable to use at # is alreadyRead because (total - actualRead < alreadyRead) specifies the condition under which the buffer overflow occurs. */ 90 if ((total > 0) && (total - actualRead < presetRead)) 91 { actualRead = total - alreadyRead; } // reading actualRead objects from holder.reader with an offset alreadyRead 92 try { results = holder.Reader.Read(alreadyRead, actualRead); } catch (Exception e) { ... } ... 130 if (results != null && results.Count > 0) 131 alreadyRead += results.Count; //add up the total number of content items that have been retrieved 132 } while (results != null && results.Count > 0 && (alreadyRead < total)); } ... 158 } </pre>	Prediction: alreadyRead(98.4%)
---	---------------------------------------

(a) The original input for which GGNN correctly predicts presetRead (highlighted in code) to be the misused variable. Instead, alreadyRead (i.e., GGNN's prediction in the top-right corner) should have been used.

<pre> 1 protected override void ProcessRecord(long presetRead) { ... 26 foreach (ContentHolder holder in contentStreams) { 27 long total = holder.Reader.Count; long alreadyRead = 0; ... 87 IList results = null; 88 do { 89 long actualReadinit = presetRead; 90 + if (presetRead * init < 0) {presetRead = alreadyRead;} // this line is deadcode /* when presetRead is no greater than half of total, line 92 will never get executed, as a result, the read access at line 93 will trigger a buffer overflow, when total % presetRead != 0 */ 91 if ((total > 0) && (total - actualReadinit < presetRead)) 92 { actualReadinit = total - alreadyRead; } 93 try {results = holder.Reader.Read(alreadyRead, actualReadinit);} catch (Exception e) { ... } ... 131 if (results != null && results.Count > 0) alreadyRead += results.Count; 132 } while (results != null && results.Count > 0 && (alreadyRead <= total total > alreadyRead)); } ... } </pre>	Prediction: presetRead(84.4%)
--	--------------------------------------

(b) The adversarial example for which GGNN no longer makes the right prediction, in particular, it now considers presetRead the correct variable to use. All transformations preserve the semantics of the original input (e.g., line 90 is deadcode). + code (resp., eode) signals added (resp., removed) code.

Fig. 1. A successful attack to GGNN in VARMisUSE task.

consequences. As an example, a malware author can modify an existing piece of malware with semantically-preserving transformations in an attempt to fool malware detection models [Kim et al. 2018; Vinayakumar et al. 2019; Yuan et al. 2014] to classify the modified malware as benign. Overall, we believe discrete adversarial attack, which has rather negative implications in the real world, deserves serious attention from the research community.

Defending against Discrete Adversarial Attack. Adversarial training [Madry et al. 2018] is by far the most successful algorithm for training robust neural networks against classical adversarial attack (e.g., FGSM [Goodfellow et al. 2015], PGD [Bubeck 2015], etc.). The key novelty is a natural saddle point (min-max) formulation (Figure 2) that captures the notion of security against adversarial attack. Technically, the inner maximization problem aims to find an adversarial example of a data point x that achieves the maximum loss on the model. On the other hand, the goal of the outer minimization problem is to find model parameters so that the adversarial loss $\rho(\theta)$ given by the inner attack problem is minimized. Training robust models under this saddle point formulation must overcome a crucial challenge. That is, how to compute gradients $\nabla_{\theta} \rho(\theta)$ for the outer minimization problem? Unlike a standard machine learning task, the adversarial loss $\rho(\theta)$ now corresponds to a maximization problem, thus, we cannot simply apply the standard backpropagation algorithm [Rumelhart et al. 1986]. A natural alternative is to compute the gradient at the maximizer of the inner maximization problem, meaning, finding the value of δ , denoted by $\delta^{\#}$, that induces the strongest possible adversarial examples (formally, $\delta^{\#} = \arg \max_{\delta} L(\theta, x + \delta, y)$), and

$$\min_{\theta} \rho(\theta), \text{ where } \rho(\theta) = \mathbb{E}_{(x,y) \sim \mathcal{D}} \left[\max_{\delta \in \mathcal{S}} L(\theta, x + \delta, y) \right]$$

where $\rho(\theta)$ is the adversarial loss of a model with parameters θ . \mathbb{E} , the expectation of $\rho(\theta)$ over data point x (with label y) drawn from distribution \mathcal{D} , defines the risk of the hypothesis learned by the model. \mathcal{S} is the set of allowed perturbations, in this case, represented by a ℓ_{∞} -ball around x [Goodfellow et al. 2015], and $L(\theta, x, y)$ denotes the loss function.

Fig. 2. The saddle point (min-max) formulation.

then using those examples to compute gradients $\nabla_{\theta} L(\theta, x + \delta^{\#}, y)$ of the loss function $L(\theta, x + \delta^{\#}, y)$. Indeed, Madry et al. [2018] prove that this approach is valid, however, Danskin's theorem, at the core of their proof, assumes the continuous differentiability of the loss function *w.r.t.* network parameters θ as well as adversarial perturbations δ . Clearly, this assumption does not hold in discrete adversarial attack because the adversarial perturbation, now specified by program transformations, is discrete in nature. Precisely for this reason, prior work [Li et al. 2022], which takes for granted the applicability of adversarial training in discrete adversarial attack, is flawed. To address this issue, we provide a much needed theoretical foundation for the application of adversarial training to discrete adversarial attack, in particular, we prove the approach of computing the gradient at the maximizer of the inner maximization problem is valid regardless of the nature of adversarial perturbations (e.g., continuous or discrete). The key insight of our proof is to show the directional derivative $\nabla_{\nabla_{\theta} L(\theta, \phi^{\#}(x), y)} \rho(\theta)$ of the adversarial loss $\rho(\theta)$ along $\nabla_{\theta} L(\theta, \phi^{\#}(x), y)$ at any point θ is greater than zero. Here, $\phi(x)$ is a generalized notion of adversarial perturbations which encompasses discrete perturbations like program transformations or continuous perturbations like ℓ_{∞} -ball; and $\phi^{\#}(x) = \arg \max_{\phi} L(\theta, \phi(x), y)$. Because a consistently positive $\nabla_{\nabla_{\theta} L(\theta, \phi^{\#}(x), y)} \rho(\theta)$ means moving θ along the opposite direction of $\nabla_{\theta} L(\theta, \phi^{\#}(x), y)$ is guaranteed to reduce the adversarial loss $\rho(\theta)$, which is precisely the goal we aim to accomplish for training robust classifiers using the min-max formulation. Specifically, we break down our proof into two sub-tasks: (1) finding the lower bound of $\nabla_{\nabla_{\theta} L(\theta, \phi^{\#}(x), y)} \rho(\theta)$; (2) proving the lower bound of $\nabla_{\nabla_{\theta} L(\theta, \phi^{\#}(x), y)} \rho(\theta)$ is positive. Motivated by these theoretical results, our primary contribution from the defense side, we take a simple measure to realize adversarial training, which results in a highly effective defense method against discrete attacks in practice.

We conduct a comprehensive evaluation on both our attack and defense methods for discrete adversarial attack. Regarding the effectiveness of DaK, we set out to compare DaK against existing attacks in the exact setting an existing attack considers when it's originally proposed. This means we deploy DaK to attack the same model using the same programs and aiming for the same target labels as an existing attack in its original setting. It turns out that DAMP [Yefet et al. 2020] is the only eligible baseline in the literature because not only does DAMP target models (e.g., code2vec [Alon et al. 2019b], GGNN) that predict semantically-adhering properties but it also has the capability of performing targeted attack. To demonstrate the generality of our attack, we also deploy DaK to attack CodeBERT [Feng et al. 2020], an influential model that predicts natural language descriptions of the functionality of source code (which is also a semantically-adhering property). In particular, CodeBERT is built upon an entirely different neural architecture from code2vec and GGNN, furthermore, it tackles a separate downstream task to code2vec and GGNN. In addition to DaK, we choose DAMP and Imitator [Quiring et al. 2019] (thanks to their generality) to attack CodeBERT. From the defense perspective, we consider as baselines two of the most noteworthy defense methods — adversarial training [Madry et al. 2018] and outlier detection [Yefet et al. 2020]. Specifically, for each baseline defense, we set our method, called enhanced adversarial training, to defend against the same adversarial attacks that the baseline method is proposed to defend against. Additionally, we have all defenses protect against DaK such that the strength of

DaK in attacking defended models can be on full display. Finally, we evaluate the robustness of hardened CodeBERT *w.r.t.* outlier detection, adversarial training, RoPGen [Li et al. 2022] (all of which are transplanted from their original setting to defend CodeBERT), and enhanced adversarial training against DAMP, Imitator and DaK. Results show that DaK is significantly more effective than all baseline attacks whether or not defense mechanisms are in place to aid models in resisting attacks. Take DaK's results in attacking code2vec as an example, DaK is on average almost ten times more effective than DAMP when code2vec is not defended by any measure, and more than ten times (*resp.*, almost three times) as effective as DAMP when code2vec is hardened by outlier detection (*resp.*, adversarial training). In addition, we find that enhanced adversarial training is the most effective against every attack among all defenses.

Main Contributions. This paper makes the following contributions:

- A method of discrete adversarial attack to models of code.
- A theoretical foundation for the application of adversarial training to defending against discrete adversarial attack.
- An evaluation of both our attack and defense methods in discrete adversarial attack. Results show that (1) DaK significantly outperforms state-of-the-art attacks regardless of the presence of existing defense techniques; (2) enhanced adversarial training is the most effective in defending against state-of-the-art attacks as well as DaK.

2 FORMALIZATION

Let M be a code model which takes a program p as input and returns a prediction c as output, *i.e.*, $M(p) = c$. For presentation simplicity and w.l.o.g., we assume M deals with one programming language at a time (*i.e.*, making predictions for programs written in a different language requires retraining M). We denote the language that M currently accepts by \mathcal{L} with formal semantics $\llbracket \cdot \rrbracket$. First, we define semantically-adhering properties which helps narrow down the code models we consider in this paper.

Definition 2.1. (*Semantically-Adhering Property*) A property τ of a program p adheres to the semantics of p **iff** $\forall p' \in \mathcal{L} \llbracket p' \rrbracket = \llbracket p \rrbracket \implies p'.\tau = p.\tau$ where $p.\tau/p'.\tau$ of program p/p' .

In this paper, we consider exclusively code models that predict semantically-adhering properties. To ensure that the ground-truth label of adversarial examples is well-defined, we enforce another constraint: an adversarial example \hat{p} must be semantically equivalent to the original input p (*i.e.*, $\llbracket p \rrbracket = \llbracket \hat{p} \rrbracket$). Because applying random changes to p would make the ground-truth label of the resultant program undefined. Below, we give a formal definition of discrete adversarial examples.

Definition 2.2. (*Discrete Adversarial Examples*) Let $p \in \mathcal{L}$ be an input program for which a code model M makes a prediction c . M is a black-box whose internal information (*e.g.*, weights, biases) is not accessible. Let label \hat{c} be the choice of the adversary. Let $\mathcal{T} = \{t_1, t_2 \dots t_n\}$ be the complete set of program transformations in language \mathcal{L} where each $t_i : \mathcal{L} \rightarrow \mathcal{L}$ is a function that maps one program to another. A program $\hat{p} \in \mathcal{L}$ is a discrete adversarial example of p **iff** (1) $\hat{p} = T(p)$ where $T \in \mathcal{T}^*$; (2) $\llbracket p \rrbracket = \llbracket \hat{p} \rrbracket$; (3) $M(\hat{p}) = \hat{c}$.

To define program transformations T , we take into account the following criteria:

- **Simplicity.** We define each t_i to be a simple, semantically-preserving transformation by itself such that we do not require a deeper analysis to obtain the guarantee of the semantic equivalence between p and \hat{p} .
- **Generality.** We define each t_i to be general and widely applicable such that they together can induce a considerable number of semantically equivalent programs to any input.

Table 1. Every semantically-preserving transformation used in this paper.

Name	Description	Example
Renaming	We replace names of variables with common variable names randomly selected from other programs.	<pre>//before: f(int length){size=length+1;} //after: f(int funcName){size=funcName+1;}</pre>
Statements permutation	We permute two statements without data and control dependency.	<pre>//before: //after: int n=1; int m=length; int m=length; int n=1;</pre>
Operands permutation	We permute two operands of binary operations satisfying the commutative law.	<pre>//before: //after: r=a+b; r=b+a;</pre>
Operators toggling	We toggle arithmetical operators associated with variables.	<pre>//before: //after: d=a-b+c; d=a-(b-c);</pre>
Boolean negation	We negate the value of a boolean variable, and propagates this change in the program to ensure a semantic equivalence of the transformed program.	<pre>//before: boolean flag=true; if (flag){...} //after: boolean flag=false; if (!flag){...}</pre>
Branch rewriting	We replace a switch statement with if statements or vice versa. If the branch is a if-then-else statement, we replace it with a ternary conditional operator or vice versa.	<pre>//before: switch(n){ case 1: r=a; break; case 2: r=b; break; ... //after: if(n==1) r=a; else if(n==2) r=b; ...</pre>
Loop rewriting	We replace for loops with while loops or vice versa.	<pre>//before: for(int i=0;i<n;i++){...} //after: int i=0; while(i<n){...;i++}</pre>
Expression unfolding	We replace expression with a single variable, with common variable name, holding the computed value.	<pre>//before: Math.min(values.length, length); //after: int var=values.length; Math.min(var, length);</pre>
Deadcode insertion	We insert code into branches or loops whose condition are always evaluated to false (the details are presented in Section 3.1.3).	<pre>//before: int i = Math.random(); //after: int i = Math.random(), j = i * i; if(j < 0) { /* deadcode */ }</pre>

- **Diversity.** We define transformations to produce different types of changes (e.g., insertion and substitution) on different parts of an input program (e.g., variables, operators, and statements) such that generated programs are manifested in diverse forms.

Table 1 provides the details for every transformation used in this paper.

Order of Transformations. Our goal is to apply the maximum number of transformations to an input program such that we can generate a sufficient number of equivalent programs for our attack method to consider. The transformations in Table 1 can be divided into two categories: substitution (first seven) and insertion (last two). For transformations in the same category, we randomly apply them without considering the order. This is because all substitutions are independent of each other, namely applying any substitution transformation does not change the applicability of any other substitution transformation. The same goes for insertions. However, since the insertion may provide new opportunities for the substitution to apply, we always perform insertion first then substitutions later. This way every transformation only needs to be considered once, which is a simple, clean approach to applying transformations. Note that different orders of the application of transformations can result in different programs. However, this is not an issue because we only require resultant programs to be semantically equivalent to original inputs regardless of their syntactic properties.

3 METHODOLOGY

In this section, we first give a detailed presentation on our discrete adversarial attack. Next, we prove the applicability of adversarial training to defending against discrete adversarial attack. Motivated by the theoretical results, we then introduce our realization of adversarial training.

3.1 Discrete Adversarial Attack

Attack Workflow. The workflow of our attack consists of three key components:

- **Destroyer** (Section 3.1.1). For starters, given an input program, we tamper with its features to weaken the signal that models rely on to make predictions. We choose to tamper with the entire program so that models can not confidently predict any label. As previously explained, we apply exclusively semantically-preserving transformations when tampering with input programs.
- **Finder** (Section 3.1.2). For a target label we aim to fool models to predict, we first find a set of programs for which models predict the target label (hereafter referred to as *suppliers*). Then, we compute the critical features of each *supplier*, which can be deemed as the strongest, most concentrated features for making models predict the target label.
- **Merger** (Section 3.1.3). With the critical features of each *supplier* at our disposal, we simply inject them into the tampered program to generate the discrete adversarial example. Because *suppliers*' critical features are strong, concentrated features that are likely to overpower the remaining features in the tampered program, models should predict the target label for the discrete adversarial example. Because the semantics of input programs must be preserved throughout the workflow of the discrete adversarial attack, we insert the critical features of a *supplier* as dead code into the tampered program.

In the rest of Section 3.1, we explain in detail how we generate the discrete adversarial example for the program in Figure 1a (hereafter referred to as the *attack program*) that successfully evades the bug finder powered by GGNN.

3.1.1 Destroyer. When tampering with input programs, a natural idea is to dismantle exclusively their critical features such that models no longer predict the label that they predict for original inputs. However, the issue with this approach is that dismantling exclusively the critical features, albeit would significantly decrease the probability that models predict the label before, may considerably increase the probability models predict a different label. The cause of this phenomenon, which frequently occurs in a preliminary study we conducted, is that under the suppression of the critical features, other features cannot exert their influence as effectively, but once the critical features were disrupted, some becomes the new critical features which is certainly capable of making models predict another label with high probability.

Considering the method in Figure 3a, which is generated through transformations of only the critical features of the *attack program* (we explain what we mean by critical features and how to find them in Section 3.1.2). GGNN now predicts a new label `total` with 66.6% probability, a value that is still considerably higher than the probabilities that GGNN predicts other labels. This means strong features still exist which can be difficult for our attack method to overcome. To address this

Algorithm 1 Destroyer (Tampering with Input programs).

```

1: procedure DESTROYER( $p, M, TList$ )
    $\triangleright$   $p$  is an input program.  $M$  represents the model.  $TList$  stands
   for transformation list, which contain insertion transformations
   first in random order, then substitutions (also in random order)
2: foreach  $t \in TList$  do
    $\triangleright$  find all the elements in  $p$  to which  $t$  is applicable.
3:    $TransformableElements \leftarrow TransformableElements(p, t)$ 
4:   foreach  $elem \in TransformableElements$  do
5:      $p' \leftarrow Apply(p, elem, t)$   $\triangleright$  perform  $t$  on  $elem$  in  $p$ 
      $\triangleright$   $p'$  is taken into account only when it produces a lower
     standard deviation (SD) among prediction probabilities of
     all labels from  $M$  than  $p$ 
6:      $p \leftarrow ReturnTheProgramWithLowerSD(p, p', M)$ 
7:   end foreach
8: end foreach
9: return  $p$ 
10: end procedure

```

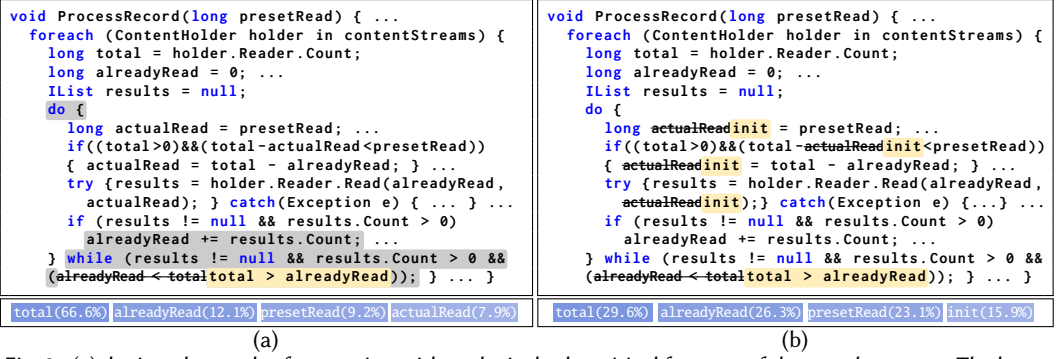


Fig. 3. (a) depicts the result of tampering with exclusively the critical features of the *attack program*. The box at the bottom shows the predictions made by GGN. Clearly, the predicted probability of the top-1 label *total* is still significantly higher than others. (b) depicts the result of tampering with the whole program. Apparently, GGN now predicts uniformly low probabilities for all labels. The transformations applied are ① **Operands Permutation**: (e.g., `alreadyRead<total` \rightarrow `total>alreadyRead`) and ② **Renaming**: (`actualRead` \rightarrow `init`). New variable names are randomly selected from common identifiers which we crawled from GGN’s training set. Because they appear in a vast amount of code, they are not particularly indicative of any label GGN may predict. `+ code` (resp., `code`) signals added (resp., removed) code. `code` are critical features.

issue, we opt to tamper with the entire input program such that ideally all labels will be predicted with low probabilities. In theory, finding the global optimal requires a brute-force search through all combinations of transformations, which yields an exponential time algorithm that is likely to be infeasible in practice. For efficiency purposes, we propose an optimization instead. The key idea is to apply a transformation only if the standard deviation of model’s prediction probabilities for all labels on the resultant program decreases. This is because a decreased standard deviation means the transformation is effective in evening out discrepancies among model’s prediction probabilities, which also indicates we are towards wiping out all significant features from an input program.

Algorithm 1 gives the details. Given a program p and a set of transformations TL , a transformation t is applied to p only if the resultant program p' has a lower standard deviation among models’ prediction probabilities for all labels than p . If t is applicable to multiple places in p , t ’s applicability at each place is examined individually against the same criterion (lines 3 to 7). As we explained previously, we consider insertions first followed by substitutions (all in random order) when applying transformations. Since deadcode insertion does not help destroy the significant features of input programs, we exclude it at the current stage of our attack. Since the number of our transformations is constant, assuming each transformation can be applied everywhere on an input program with size n (i.e., the number of tokens), the time complexity of the algorithm is $O(n)$. As a greedy approach, *Destroyer* may only find a local optimal, however, we find substantial evidence that original programs are sufficiently tampered (Table 9), paving the way for the creation of highly effective adversarial examples. Following Algorithm 1, we tamper with the *attack program* in its entirety. Figure 3b shows the resultant program in which `alreadyRead<total` is replaced with `total>alreadyRead`, and `actualRead` is renamed into `init`. GGN predicts all labels with uniformly low probabilities, a sign that we erased all significant features from the original program.

3.1.2 Finder. As we explained in the attack workflow, the goal of *Finder* is to find the critical features of *suppliers*. Finding *suppliers* themselves is in general an easy task because all we require is that the model under attack predicts the target label for *suppliers*; in addition, *suppliers* can be easily acquired from public datasets or open-source projects. In VARMISUSE task, finding *suppliers* is slightly more complicated in engineering terms. The reason is the prediction labels are not a fixed

set of words or phrases but rather the variables that are in scope *w.r.t.* the variable being predicted for one particular program. Clearly, those in-scope variables vary from program to program, which makes it harder to find the program with the target label. Figure 4 shows the *supplier* that we find for the *attack program*. We defer to Section 4.1.2 for a detailed discussion on how it is found.

After obtaining a *supplier*, now we can focus on finding its critical features. Obviously, the first challenge we need to resolve is to define what we mean by critical features. In this regard, we refer to Wang et al. [2021], which define two concrete criteria that statements/expressions must comply with in order to be critical features: (1) models make the same predictions for the critical features as they do for original programs, (2) models make different predictions for the remaining code after critical features are removed from the original program. We illustrate the two criteria with code2vec [Alon et al. 2019b], a pioneering model for method name prediction. Given the method in Figure 5a, Figure 5b shows that with the first statement — `int n = Math.min(values.length, data.length-start);` — alone in the method body, code2vec still predicts `load`, the label it predicts for the original method. Moreover, removing this statement makes code2vec alter its prediction to `set` (Figure 5c). Therefore, Wang et al. [2021] regard `int n = Math.min(values.length, data.length-start);` to be the critical features that code2vec uses to predict the name of the method in Figure 5a. We note Wang et al. [2021]’s definition supersedes the prior work [Rabin et al. 2021], which only considers the first criterion. We illustrate the flaw of Rabin et al. [2021] with an example in Figure 6. Figure 6a shows another statement in the same method, `data[start+i] = (short)(values[i] & mask);`, complies with the first criterion, thus would have been considered as the critical features by Rabin et al. [2021]. However, as Figure 6b demonstrates, removing `data[start+i] = (short)(values[i] & mask);` from the method body does not alter code2vec’s prediction, indicating that code2vec does not even need `data[start+i] = (short)(values[i] & mask);` let alone uses it as the critical features for prediction. Below, we define *critical features* based on Wang et al. [2021]’s definition¹.

We illustrate the two criteria with code2vec [Alon et al. 2019b], a pioneering model for method name prediction. Given the method in Figure 5a, Figure 5b shows that with the first statement — `int n = Math.min(values.length, data.length-start);` — alone in the method body, code2vec still predicts `load`, the label it predicts for the original method. Moreover, removing this statement makes code2vec alter its prediction to `set` (Figure 5c). Therefore, Wang et al. [2021] regard `int n = Math.min(values.length, data.length-start);` to be the critical features that code2vec uses to predict the name of the method in Figure 5a. We note Wang et al. [2021]’s definition supersedes the prior work [Rabin et al. 2021], which only considers the first criterion. We illustrate the flaw of Rabin et al. [2021] with an example in Figure 6. Figure 6a shows another statement in the same method, `data[start+i] = (short)(values[i] & mask);`, complies with the first criterion, thus would have been considered as the critical features by Rabin et al. [2021]. However, as Figure 6b demonstrates, removing `data[start+i] = (short)(values[i] & mask);` from the method body does not alter code2vec’s prediction, indicating that code2vec does not even need `data[start+i] = (short)(values[i] & mask);` let alone uses it as the critical features for prediction. Below, we define *critical features* based on Wang et al. [2021]’s definition¹.

Definition 3.1. (*critical features*) The *critical features* that M uses for predicting the label of p is a set of statements \tilde{p} such that (1) \tilde{p} is a *constituent* of p : \tilde{p} ’s token sequence, denoted by $(t_n^{\tilde{p}})_{n \in \mathbb{N}}$, is a *subsequence* of p ’s denoted by $(t_m^p)_{m \in \mathbb{N}}$. Formally, $(t_n^{\tilde{p}})_{n \in \mathbb{N}} = (t_{m_k}^p)_{k \in \mathbb{N}}$ where $(m_k)_{k \in \mathbb{N}}$ is a strictly increasing sequence of positive integers. (2) \tilde{p} is *sufficient*: \tilde{p} is predicted by M to be in the same class as p (i.e., $M(p) = M(\tilde{p}) = c$); (3) \tilde{p} is *necessary*: $p \setminus \tilde{p}$ is predicted by M to be in a different

<pre>void load(int start, int[] values, int mask) { int n = Math.min(values.length, data.length - start); for (int i = 0; i < n; i++) { System.out.println("index: "+start+" "+i); data[start+i] = (short)(values[i] & mask); } }</pre>	<pre>void load(int start, int[] values, int mask) { int n = Math.min(values.length, data.length - start); for (int i = 0; i < n; i++) { System.out.println("index: "+start+" "+i); data[start+i] = (short)(values[i] & mask); } }</pre>	<pre>void Set(int start, int[] values, int mask) { int n = Math.min(values.length, data.length - start); for (int i = 0; i < n; i++) { System.out.println("index: "+start+" "+i); data[start+i] = (short)(values[i] & mask); } }</pre>
(a)	(b)	(c)

Fig. 5. (a) shows an example method whose name is correctly predicted by code2vec [Alon et al. 2019b]. (b) shows the first statement alone (i.e., `int n = Math.min(values.length, data.length-start);`) makes code2vec predict the same label `load`. (c) shows removing the statement alters the prediction of code2vec to `set`. `code` signals removed code. `code` are method names predicted by code2vec.

¹We note that Wang et al. [2021] name critical features “wheat” and the rest “chaff” for an input program.

<pre>void load(int start, int[] values, int mask) { int n = Math.min(values.length, data.length - start); for (int i = 0; i < n; i++) { System.out.println("index: " + start + " " + i); data[start + i] = (short) (values[i] & mask); } }</pre>	<pre>void load(int start, int[] values, int mask) { int n = Math.min(values.length, data.length - start); for (int i = 0; i < n; i++) { System.out.println("index: " + start + " " + i); data[start + i] = (short) (values[i] & mask); } }</pre>
(a)	(b)

Fig. 6. Demonstrating the flaw of Rabin et al. [2021]. (a) shows the statement `data[start+i]=(short)(values[i] & mask);` alone keeps the prediction that code2vec makes for the method in Figure 5a, thus would have been considered critical features by Rabin et al. [2021]. However, (b) shows removing the statement does not alter code2vec's prediction, which strongly indicates `data[start+i]=(short)(values[i] & mask);` are not the critical features. `code`, `code` denote the same meaning as in Figure 5.

class from p (i.e., $M(p \setminus \tilde{p}) \neq M(p)$) where $p \setminus \tilde{p}$ denotes the operation that subtracts program \tilde{p} from p (Remark 1). Finally (4) \tilde{p} is *1-minimal*: removing any token from \tilde{p} causes \tilde{p} to violate either *sufficient* or *necessary* requirement, or both.

Remark 1. (*Subtraction*) Given a program p and a set of statements \tilde{p} , subtraction $p \setminus \tilde{p}$ means for each statement s in \tilde{p} , first locate the subtree, which is equivalent to the Abstract Syntax Tree (AST) of s , within the AST of p ; then remove the located subtree from the AST of p . Finally, serialize the resultant AST of p 's back to source code.

Unlike Wang et al. [2021]'s definition, we do not require *critical features* to be the global minimum code because 1-minimality already ensures the precision of *critical features*, in turn, the strength of resultant adversarial examples as our extensive evaluation demonstrates.

Identifying *critical features* from an input program is a challenging problem. We apply a two-step, coarse-to-fine method, called *Reduce* and *Mutate* proposed by Wang et al. [2021]. Conceptually, the method first finds the minimum fragment of code that contains *critical features*, then it further mutates the minimum fragment to pinpoint the fine-grained *critical features*. Technically, [Wang et al. 2021] queries models with fragments sorted by size in ascending order (i.e., starting from one statement, then two, three statements until reaching the entire program). Under this approach, *Reduce* returns the first and smallest code fragment (i.e., with the least number of statements) that satisfies both *sufficient* and *necessary* requirement. For the *supplier* in Figure 4, statement `presetRead = presetRead - consume;` is the fragment produced by *Reduce*. Next, *Mutate* aims to remove the irrelevant code within the fragment discovered in *Reduce* step. The remaining code, which keeps both *sufficient* and *necessary* requirement satisfied, will be regarded as the *critical features* (e.g., the statement `presetRead = presetRead - consume;` is further mutated into `presetRead = consume;` as the *critical features*). Because the code that *Reduce* and *Mutate* produce satisfies all three requirements in Definition 3.1, it is indeed the *critical features* of an input program. Interested readers may refer to [Wang et al. 2021] for the details of *Reduce* and *Mutate*.

Collecting a Multitude of Suppliers. To increase the success rate of our discrete attack, we collect a set of *suppliers* for an input program because if any of them helps create a successful adversarial example, we have achieved our goal.

3.1.3 Merger. Given the tampered input program (i.e., the output of *Destroyer*) and the *critical features* of a *supplier* (i.e., the output of *Finder*), we are ready to create discrete adversarial examples. Specifically, we insert *critical features* of a *supplier* into the tampered input programs as dead code such that the semantics of the original input is preserved. Considering *critical features* are likely to overpower any feature left in the tampered input, we randomly select a position in the input program to insert *critical features*.

Ranking Suppliers. We rank the set of *suppliers* according to the strength of the signal their *critical features* emit. We approximate the signal strength for *critical features* from the following two sources: (1) the probability that models predict the correct label (denoted by \bar{c}) in order to have

critical features satisfy the *sufficient* requirement; and (2) the difference between the probabilities that models predict \bar{c} for the original input and the resultant code of the subtraction operation (Remark 1) when *critical features* satisfy the *necessary* requirement. In both cases, the higher the value, the more powerful the *critical features* are deemed to be. We adopt the addition of the two values which is shown to work better than either value alone in practice. Figure 1 in the supplemental material illustrates the strength of the *critical features* of the *supplier* in Figure 4. Following the ranking of *suppliers*, we use each to create an adversarial example, in particular, we take two measures:

- (1) We inject *critical features* within a branch (e.g., if, switch, etc.) whose condition is always evaluated to false. We do not place *critical features* into unreachable code blocks (e.g., after return), otherwise, they will be easily removed by compilers, which can be deemed as the first line of defense for code models.
- (2) In light of deadcode elimination, we construct common (i.e., will not make adversarial examples clear outliers) but non-obvious (i.e., complicated enough to bypass compilers) branch or loop conditions to guard injected *critical features*. In particular, we define a set of rules for creating expressions guaranteed to be evaluated to false, such as the result of multiplication of an integer with itself is less than zero (as exemplified in Figure 7); the size of empty strings is greater than zero; the return value of Min() (resp., Max()) function is greater (resp., less) than any of its parameters; etc. Although state-of-the-art SMT solvers may recognize the disguised conditions, we can still confuse them with more complicated opaque predicates [Pierazzi et al. 2020].

When attack GGNN in VARMISUSE task, creating adversarial example requires an extra step. That is, renaming variables in the *critical features* of a *supplier* to those in the original input so that the *critical features* no longer contain foreign variables, thus, the adversarial example to be created still compiles. On the other hand, variable renaming must be done in a manner that does not affect the validity of the *supplier*, meaning, GGNN must predict the target label also for the renamed *supplier*. Technically, we adopt a brute-force approach to enumerate all mappings between variables in the *critical features* and input program. Considering the small number of variables in both *critical features* and the input program, an exhaustive approach is still efficient. If none of the mappings serves our purpose, we will discard the *supplier* and move on to the next in the ranking. Given the *supplier* shown in Figure 4 and the tampered *attack program* in Figure 3b, Figure 7 depicts the complete adversarial example. Specifically, we first rename consume to alreadyRead and inject the renamed *critical features*, `presetRead = alreadyRead`, within an if branch, whose condition `presetRead * init < 0` will always be evaluated to false.

<pre> void ProcessRecord(long pre- setRead) { ... foreach (ContentHolder holder in contentStreams){ long total = holder.Reader.Count; long alreadyRead = 0; ... IList results = null; do { long init = presetRead; ... + if (presetRead*init<0){presetRead = alreadyRead;} if ((total > 0) && (total - init < presetRead)) { init = total - alreadyRead; } ... try {results = holder.Reader.Read(alreadyRead, init);} catch(Exception e) { ... } ... if (results != null && results.Count > 0) alreadyRead += results.Count; } while (results != null && results.Count > 0 && (total > alreadyRead)); } ... } </pre>	Prediction: presetRead(84.4%)
--	--------------------------------------

Fig. 7. A complete adversarial example for which GGNN no longer predicts presetRead (highlighted in `code`) as misused. `+ code` signals added code.

3.2 Defenses against Discrete Adversarial Attack

To apply adversarial training to defend against discrete adversarial attacks, the first problem we must solve is to specify a new threat model for the inner maximization problem of the saddle point formulation (Figure 2) because continuous perturbation like ℓ_∞ -ball is no longer applicable. Equation 1 below first defines the power of discrete adversaries, in particular, they create adversarial examples by transforming original inputs in a semantically-preserving manner; furthermore, it gives

the overall new saddle point formulation for discrete adversarial attacks. Now, we face the same challenge: how do we compute gradients $\nabla_{\theta} \rho_d(\theta)$ for the outer minimization problem if we were to apply stochastic gradient descent to find optimal model parameters θ . If we borrow Madry et al. [2018]'s approach of computing gradients at the maximizer of the inner maximization problem, we can compute the gradients $\nabla_{\theta} L(\theta, R^{\#}(x), y)$ of the loss function on the strongest discrete adversarial example induced by the transformations $R^{\#}$ s.t. $R^{\#} = \arg \max_R L(\theta, R(x), y)$. However, this approach lacks a scientific foundation because the theory behind Madry et al. [2018]'s approach makes the assumption on the continuous differentiability of the adversarial loss w.r.t. network parameters as well as adversarial perturbations, which clearly does not hold for discrete adversaries.

$$\min_{\theta} \rho_d(\theta), \text{ where } \rho_d(\theta) = \mathbb{E}_{(x,y) \sim \mathcal{D}} \left[\max_{R \in \mathcal{T}'} L(\theta, R(x), y) \right] \quad (1)$$

where $\mathcal{T}' = \{T \mid T \in \mathcal{T}^* \wedge \llbracket T(x) \rrbracket = \llbracket x \rrbracket\}$ and \mathcal{T} is defined in Definition 2.2.

Proving the validity of Madry et al. [2018]'s approach. To validate Madry et al. [2018]'s approach in the setting of discrete adversarial attack, a natural idea is to prove the directional derivative $\nabla_{\nabla_{\theta} L(\theta, R^{\#}(x), y)} \rho_d(\theta)$ of $\rho_d(\theta)$ along $\nabla_{\theta} L(\theta, R^{\#}(x), y)$ at any point θ is greater than zero. Because if $\nabla_{\nabla_{\theta} L(\theta, R^{\#}(x), y)} \rho_d(\theta)$ can be proven to be a positive value at all times, then moving θ along the opposite direction of $\nabla_{\theta} L(\theta, R^{\#}(x), y)$ is guaranteed to reduce $\rho_d(\theta)$, which is precisely the goal we intend to accomplish in training robust models using the saddle point formulation. Conceptually, our proof for the positivity of $\nabla_{\nabla_{\theta} L(\theta, R^{\#}(x), y)} \rho_d(\theta)$ consists of two major components: first, finding the lower bound of $\nabla_{\nabla_{\theta} L(\theta, R^{\#}(x), y)} \rho_d(\theta)$; and second, proving the lower bound of $\nabla_{\nabla_{\theta} L(\theta, R^{\#}(x), y)} \rho_d(\theta)$ is greater than zero.

Theorem 3.1. Let $R^{\#}$ be the maximizer for $\max_{R \in \mathcal{T}'} L(\theta, R(x), y)$ in formulation 1, formally, $R^{\#} = \arg \max_R L(\theta, R(x), y)$. Then, the directional derivative of $\max_{R \in \mathcal{T}'} L(\theta, R(x), y)$ along $\nabla_{\theta} L(\theta, R^{\#}(x), y)$ is positive at any point θ .

PROOF. Assume θ_{pre} is an arbitrary set of model parameters, moving θ_{pre} in the direction of $\nabla_{\theta} L(\theta, R^{\#}(x), y)$ with an infinitely small step size $\gamma \in \mathbb{R}_+$ arrives at θ_{post} . Thus,

$$\theta_{post} - \theta_{pre} = \gamma \nabla_{\theta} L(\theta_{pre}, R^{\#}(x), y) \quad (2)$$

We now turn to the difference quotient to compute the directional derivative of $\max_{R \in \mathcal{T}'} L(\theta, R(x), y)$ along $\nabla_{\theta} L(\theta, R^{\#}(x), y)$ at θ_{pre} . $\theta_{post} \rightarrow \theta_{pre}$ denotes that θ_{post} is infinitely close to θ_{pre} .

$$\lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{\max_{R \in \mathcal{T}'} L(\theta_{post}, R(x), y) - \max_{R \in \mathcal{T}'} L(\theta_{pre}, R(x), y)}{|\theta_{post} - \theta_{pre}|} \quad (3)$$

Part I: find the lower bound of Equation 3.

First, we define $R_{\theta_{post}}^{\#}, R_{\theta_{pre}}^{\#}$ to be:

$$R_{\theta_{post}}^{\#} = \arg \max_{R \in \mathcal{T}'} L(\theta_{post}, R(x), y) \quad R_{\theta_{pre}}^{\#} = \arg \max_{R \in \mathcal{T}'} L(\theta_{pre}, R(x), y)$$

Thus, Equation 3 can be rewritten into:

$$\begin{aligned} \lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{\max_{R \in \mathcal{T}'} L(\theta_{post}, R(x), y) - \max_{R \in \mathcal{T}'} L(\theta_{pre}, R(x), y)}{|\theta_{post} - \theta_{pre}|} &= \lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{L(\theta_{post}, R_{\theta_{post}}^{\#}(x), y) - L(\theta_{pre}, R_{\theta_{pre}}^{\#}(x), y)}{|\theta_{post} - \theta_{pre}|} = \\ \lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{L(\theta_{post}, R_{\theta_{post}}^{\#}(x), y) - L(\theta_{post}, R_{\theta_{pre}}^{\#}(x), y) + L(\theta_{post}, R_{\theta_{pre}}^{\#}(x), y) - L(\theta_{pre}, R_{\theta_{pre}}^{\#}(x), y)}{|\theta_{post} - \theta_{pre}|} &= \\ \lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{L(\theta_{post}, R_{\theta_{post}}^{\#}(x), y) - L(\theta_{post}, R_{\theta_{pre}}^{\#}(x), y)}{|\theta_{post} - \theta_{pre}|} + \lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{L(\theta_{post}, R_{\theta_{pre}}^{\#}(x), y) - L(\theta_{pre}, R_{\theta_{pre}}^{\#}(x), y)}{|\theta_{post} - \theta_{pre}|} & \quad (4) \end{aligned}$$

Because $R_{\theta_{post}}^\# = \operatorname{argmax}_{R \in \mathcal{T}'} L(\theta_{post}, R(x), y)$, the numerator of the first term on the third line of Equation 4 $-L(\theta_{post}, R_{\theta_{post}}^\#(x), y) - L(\theta_{post}, R_{\theta_{pre}}^\#(x), y)$ is no less than zero, thus,

$$\lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{\max_{R \in \mathcal{T}'} L(\theta_{post}, R(x), y) - \max_{R \in \mathcal{T}'} L(\theta_{pre}, R(x), y)}{|\theta_{post} - \theta_{pre}|} \geq \lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{L(\theta_{post}, R_{\theta_{pre}}^\#(x), y) - L(\theta_{pre}, R_{\theta_{pre}}^\#(x), y)}{|\theta_{post} - \theta_{pre}|} \quad (5)$$

According to Equation 5, we find the lower bound of Equation 3 to be $\lim_{\theta_{post} \rightarrow \theta_{pre}} (L(\theta_{post}, R_{\theta_{pre}}^\#(x), y) - L(\theta_{pre}, R_{\theta_{pre}}^\#(x), y)) / |\theta_{post} - \theta_{pre}|$.

Part II: prove $\lim_{\theta_{post} \rightarrow \theta_{pre}} (L(\theta_{post}, R_{\theta_{pre}}^\#(x), y) - L(\theta_{pre}, R_{\theta_{pre}}^\#(x), y)) / |\theta_{post} - \theta_{pre}|$, the lower bound of Equation 3, to be positive.

Because $L(\theta, R_{\theta_{pre}}^\#(x), y)$ is a continuous function over the complete set of values of θ^2 , we apply mean value theorem in several variables to $(L(\theta_{post}, R_{\theta_{pre}}^\#(x), y) - L(\theta_{pre}, R_{\theta_{pre}}^\#(x), y)) / |\theta_{post} - \theta_{pre}|$:

$$\lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{L(\theta_{post}, R_{\theta_{pre}}^\#(x), y) - L(\theta_{pre}, R_{\theta_{pre}}^\#(x), y)}{|\theta_{post} - \theta_{pre}|} = \lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{(\theta_{post} - \theta_{pre}) \cdot \nabla_\theta L(\theta', R_{\theta_{pre}}^\#(x), y)}{|\theta_{post} - \theta_{pre}|} \quad (6)$$

where $\theta' = (1 - \alpha)\theta_{pre} + \alpha\theta_{post}$ for some α within $(0, 1)$. Specifically, θ' is some point on the segment joining θ_{pre} to θ_{post} . Because θ_{post} is infinitely close to θ_{pre} , θ' will also be infinitely close to θ_{pre} . Thus, taking the limit of $\nabla_\theta L(\theta', R_{\theta_{pre}}^\#(x), y)$ as θ_{post} approaches to θ_{pre} leads to $\nabla_\theta L(\theta_{pre}, R_{\theta_{pre}}^\#(x), y)$. Then, the right side of Equation 6 equals to:

$$\frac{(\theta_{post} - \theta_{pre}) \cdot \nabla_\theta L(\theta_{pre}, R_{\theta_{pre}}^\#(x), y)}{|\theta_{post} - \theta_{pre}|} \quad (7)$$

Because of Equation 2, Equation 7 can be rewritten into:

$$\frac{(\theta_{post} - \theta_{pre}) \cdot (\theta_{post} - \theta_{pre}) / \gamma}{|\theta_{post} - \theta_{pre}|} \quad (8)$$

Because $\theta_{pre} \neq \theta_{post}$ and $\gamma \in \mathbb{R}_+$ is a positive number, Equation 8 is positive, thus we prove $\lim_{\theta_{post} \rightarrow \theta_{pre}} (L(\theta_{post}, R_{\theta_{pre}}^\#(x), y) - L(\theta_{pre}, R_{\theta_{pre}}^\#(x), y)) / |\theta_{post} - \theta_{pre}|$, the lower bound of Equation 3, is positive. Substituting the right side of Equation 6 with Equation 8 gives rise to

$$\lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{L(\theta_{post}, R_{\theta_{pre}}^\#(x), y) - L(\theta_{pre}, R_{\theta_{pre}}^\#(x), y)}{|\theta_{post} - \theta_{pre}|} = \frac{(\theta_{post} - \theta_{pre}) \cdot (\theta_{post} - \theta_{pre}) / \gamma}{|\theta_{post} - \theta_{pre}|} > 0 \quad (9)$$

Finally, putting Equation 5 and 9 together established the following:

$$\lim_{\theta_{post} \rightarrow \theta_{pre}} \frac{\max_{R \in \mathcal{T}'} L(\theta_{post}, R(x), y) - \max_{R \in \mathcal{T}'} L(\theta_{pre}, R(x), y)}{|\theta_{post} - \theta_{pre}|} > 0$$

Because θ_{pre} is an arbitrary set of model parameters, we prove the directional derivative of $\max_{R \in \mathcal{T}'} L(\theta, R(x), y)$ along $\nabla_\theta L(\theta, R^\#(x), y)$ is positive at any point θ . \square

In fact, our proof is general since it does not depend on any specific implementations of $R(\cdot)$, therefore, it applies to both discrete adversaries (e.g., $R(\cdot)$ specifies program transformations)

²The regular loss function is indeed continuously differentiable w.r.t. network parameters when adversarial examples are created through a fixed set of transformations of original inputs.

and continuous adversaries (e.g., $R(\cdot)$ represents ℓ_∞ -ball). Guided by Theorem 3.1, our primary contribution on the defense side, we present our realization of adversarial training.

Enhanced Adversarial Training. A key challenge in realizing adversarial training is solving the inner maximization problem in the saddle point formulation. For efficiency reasons, prior works [Li et al. 2022; Yefet et al. 2020] opt for successful adversarial examples found by an attack as solutions to the inner maximization problem. Clearly, adversarial examples that succeeded in fooling a model to make wrong predictions may not be those that maximize the model’s adversarial loss, a condition that is required for applying Madry et al. [2018]’s approach to training robust models (as Theorem 3.1 shows). To tackle this issue, we propose Enhanced Adversarial Training (Everl) which is generally applicable to defending against any adversarial attack in principle. The idea is to train a model only on the strongest adversarial example possible. Specifically, to defend against an attack, we aim to exhaustively enumerate all adversarial examples found by the attack for a given input to find the strongest. For the case specific to DaK, we first compute the transitive closure of the set of transformations presented in Table 1 on an input program. For each program within the transitive closure, we combine it with the *critical features* of each *supplier* to find the overall strongest adversarial example that achieves the maximum loss of the model. As for other adversarial attacks, specifically for continuous/classical adversaries like DAMP, we increase the adversarial step [Madry et al. 2018] and introduce random initialization for adversarial perturbation [Wong et al. 2020] to find the strongest adversarial example; for discrete adversaries like Imitator [Quiring et al. 2019], we increase the width and the depth of the search tree for program transformations. Although in theory Everl is not guaranteed to find the strongest among all adversarial examples an attack generates, the adversarial examples Everl uses to train a model are more likely to be the strongest than existing realizations of adversarial training use. Thus, Everl is more effective than existing adversarial training against DaK and many other attacks according to our evaluation.

4 EVALUATION

Our evaluation aims to answer two main questions: (1) how effective is DaK in attacking code models whether or not existing defenses mechanisms are in place to aid models in resisting attacks; (2) how effective is enhanced adversarial training against state-of-the-art attacks as well as DaK.

4.1 Effectiveness of DaK

4.1.1 Attack Setups. To evaluate the effectiveness of DaK, we set out to compare DaK against existing attacks in the exact setting each attack method considers when it is originally proposed. This means we deploy DaK to attack the same model (i.e., same neural architecture trained on the same dataset) using the same programs and aiming for the same target labels as the existing attack in its original setting. Among all existing attacks to code models, we find DAMP [Yefet et al. 2020] to be the only eligible baseline while others either target models that predict non-semantically-adhering properties [Li et al. 2022; Quiring et al. 2019], or are incapable of performing targeted attack [Zhang et al. 2022; Zhou et al. 2022]. Yefet et al. [2020] consider three models: code2vec [Alon et al. 2019b], GGNN [Allamanis et al. 2018] and GNN-FILM [Brockschmidt 2020]. code2vec predicts a descriptor (in natural language) about the functionality of a method which is obviously a semantically-adhering property. Similarly, GGNN/GNN-FILM, which predicts variable misuses, is also valid for discrete adversarial attack. This is because whether or not variables are misused does not vary with semantically-preserving transformations. Yefet et al. [2020] train code2vec on Java-large [Alon et al. 2019a], and GGNN/GNN-FILM on a C# dataset [Allamanis et al. 2018] respectively. Considering GNN-FILM and GGNN are built upon similar neural architectures for the same downstream tasks, and DAMP achieved significantly better results with GGNN, we do not include GNN-FILM in our evaluation. Regarding target labels for attacks to code2vec, Yefet

et al. [2020] randomly sampled labels that occurred at least 10k times in Java-large. As for attacks to GGNN, since all examples in the C# dataset have at least one type-correct replacement variable other than the correct variable. Thus, they pick each replacement variable in turn as the target label, and average the results over all target labels. In both attacks to code2vec and GGNN, they only focus on programs that models predicted correctly out of the test set.

To demonstrate the generality of DaK, we also consider CodeBERT [Feng et al. 2020], a model built upon a different neural architecture from code2vec and GGNN. It achieves state-of-the-art performance on both natural language code search and code documentation generation, the latter of which is used in this evaluation because the former takes natural language as inputs which is inapplicable to our setting. Since code documentation is generated according to the semantics of a code snippet, therefore, CodeBERT also predicts semantically-adhering properties. Unfortunately, there does not exist any work capable of performing targeted attack to CodeBERT. Therefore, we attempt to transplant methods proposed to attack other models to attack CodeBERT. DAMP, Imitator [Quiring et al. 2019], and ACSIA [Li et al. 2022] are the most noteworthy methods of targeted attack. Because ACSIA relies on pre-defined rules specific to coding styles (e.g., camelCase vs. snake_case) which do not apply to code documentation generation, we do not consider ACSIA for attacking CodeBERT. In contrast, DAMP and Imitator are general, thus, we adapt them to attack CodeBERT. We train CodeBERT using CodeSearchNet [Husain et al. 2019], the dataset on which CodeBERT is evaluated when it is first proposed. To keep our engineering effort manageable, we only use Java programs in CodeSearchNet (496,688 in total), which should be sufficient for our evaluation. We follow DAMP's approach in selecting target labels to attack CodeBERT. Specifically, we randomly sampled labels that occurred at least 1k times in the training set of CodeSearchNet, which should mount a good challenge given that the model should have learned the pattern for those labels sufficiently well. To select programs to attack CodeBERT, like DAMP, we also focus on programs which CodeBERT correctly predicts out of the test set.

4.1.2 Finding Suppliers. *Suppliers* are randomly selected out of the test set of the model under attack. A program qualifies to be a *supplier* if it is predicted by the model under attack to have the target label. We use 50 *suppliers* for an input program to attack models. Here we describe a simple yet effective technique for finding *suppliers* in VARMISUSE task:

- (1) First, we seek a program \bar{p} that has a statement/expression \bar{s} such that \bar{s} is syntactically identical/similar to the statement/expression s (in the original input p) in which GGNN predicts whether a variable v is misused. Figure 8a gives an example of \bar{p} .
- (2) Then, we feed \bar{p} to GGNN to get a prediction \bar{c} (which is a variable in \bar{p}) for \bar{v} , a variable in \bar{s} that corresponds to v in s . Figure 8a depicts GGNN's prediction \bar{c} (i.e., rest) for \bar{v} (i.e., rest also). Note that the value of \bar{c} is irrelevant; no matter what it is, the variable will be renamed to the target label as described in the next step.
- (3) Next, we rename the variable identified by \bar{c} to the target label we aim to fool GGNN to predict for v , and check if GGNN's prediction for \bar{v} switches from \bar{c} to the target label. If it does, we consider \bar{p} (after renaming \bar{c}) a *supplier*, otherwise, we go back to (1) to find a different program. Figure 8b shows GGNN indeed predicts presetRead after rest is renamed into presetRead.

4.1.3 Metric. For all experiments throughout the evaluation (including both attack and defense), we measure the strength of attacks/defenses with the robustness score of the model under attack, which is originally proposed by Yefet et al. [2020]. Specifically, robustness score is defined as the percentage of input programs for which the correctly predicted label was not changed to the adversary's desired label. If the predicted label was changed to a label that is not the adversarial label, we consider the model to be robust. The metric implies that the lower (resp., higher) model robustness, the higher effectiveness of the attack (resp., defense).

<pre> public void onResume() { ... long rest = all; ... long now = SystemClock.elapsedRealtime(); int hash = mActivityAcd.getCause(); long consume = now - start; rest = rest - consume; if (hash != 0 && (threshold - consume < <u>rest</u>)) { for (Widget widget : mWidgetsMap.values()) {...} } mNotificationHashTime = now; ... } </pre>	Prediction: rest (94.5%)	<pre> public void onResume() { ... long restpresetRead = all; ... long now = SystemClock.elapsedRealtime(); int hash = mActivityAcd.getCause(); long consume = now - start; restpresetRead = restpresetRead - consume; if (hash != 0 && (threshold - consume < restpresetRead)) { for (Widget widget : mWidgetsMap.values()) {...} } mNotificationHashTime = now; ... } </pre>	Prediction: presetRead (95.2%)
(a)		(b)	

Fig. 8. Finding a *supplier* for the *attack program*. The reason that (a) is selected to be a *supplier* is that it has an expression (underlined) that is syntactic similar to $(total > 0) \ \&\& \ (total - actualRead < presetRead)$, the expression in the *attack program* for which GGNN predicts whether `presetRead` is misused. GGNN’s prediction in the top-right corner is made for the variable highlighted in **code**. (b) shows that after renaming variable `rest` (i.e., GGNN’s prediction) in (a) to the target label `presetRead`, GGNN changes its prediction to `presetRead`. + code (resp., ~~code~~) signals added (resp., removed) code.

4.1.4 Hardware. All experiments in our evaluation are conducted on a Red Hat Linux server that has 64 Intel(R) Xeon(R) 2.10GHz CPU, 755GB RAM and 4 Tesla V100 GPU (32GB GPU memory).

4.1.5 Attack Results. Table 2 and 3 show the results of DaK in attacking code2vec and GGNN in DAMP’s setting. Table 4 shows the effectiveness of DAMP, Imitator and DaK in attacking CodeBERT. It is rather clear that DaK is by far the most effective attack across the board, in particular, using models’ average robustness score across all labels, DaK is almost ten times more effective than DAMP in attacking code2vec (6.21 vs. 61.46), and more than three times as effective as DAMP in attacking GGNN (Table 3). Although DaK’s advantage over DAMP and Imitator in attacking CodeBERT decreases, DaK still outperforms both baselines by a comfortable margin. A likely cause of DaK’s lesser advantage in this case is the type of predictions that CodeBERT makes for code snippets, which are sentences. Thus, even if DaK manages to fool CodeBERT to predict most of the words in a target label, it does not get partial credit for missing the few remaining ones. The reason that DaK easily beats out the baselines in all our experiments is that (1) DAMP only explores a very limited solution space, thus, it is less capable of finding strong adversarial examples; (2) Imitator is in essence heuristics that are significantly less intelligent than DaK. Regarding the efficiency of DaK, Table 5-7 show the time that DaK spent on average (in seconds) to create a successful adversarial example to attack code2vec, GGNN and CodeBERT. Despite being slightly outperformed by DAMP, DaK is still highly efficient.

4.1.6 Can Linters/Compilers Detect Deadcode in Adversarial Examples. In this experiment, we investigate if compilers and linters can detect deadcode that we insert into input programs. The first column of Table 8 shows the exhaustive list of patterns that we use to insert deadcode. Considering

Table 2. Comparing DaK with DAMP in attacking code2vec. Results of the strongest attack (i.e., the lowest robustness scores) are marked in bold.

Labels Attacks	init	mergeFrom	size	isEmpty	clear	remove	value	load	add	run
DAMP	48.44	10.39	78.27	79.04	82.80	63.15	76.75	55.65	68.60	51.52
DaK	0.00	8.11	21.29	0.00	0.00	4.27	28.40	0.00	0.00	0.00

Table 3. Comparing DaK with DAMP in attacking GGNN.

Attacks	Results
DAMP	57.99
DaK	18.20

Table 4. Comparing DaK with DAMP, Imitator in attacking CodeBERT.

Labels Attacks	private methods	create an instance	set up fields	output class	sets the image
DAMP	70.22	89.99	84.28	88.21	78.20
Imitator	81.20	92.28	96.24	100.0	91.90
DaK	43.20	53.28	56.24	62.64	50.12

Table 5. The time (all in seconds) that DAMP and DaK take on average to attack code2vec.

Labels Attacks	init	mergeFrom	size	isEmpty	clear	remove	value	load	add	run
DAMP	0.82	1.13	0.72	1.11	0.92	1.23	1.10	0.78	0.91	1.03
DaK	2.92	3.97	2.25	3.93	2.51	3.07	3.02	2.99	3.30	3.25

Table 6. Average time of DAMP and DaK for attacking GGNN. Table 7. Average time of DAMP, Imitator and DaK for attacking CodeBERT.

Attacks	Results
DAMP	2.87
DaK	5.52

Labels Attacks	private methods	create an instance	set up fields	output class	sets the image
DAMP	1.42	1.71	1.19	1.11	1.32
Imitator	5.06	5.34	4.97	4.86	4.25
DaK	4.10	4.63	4.21	4.72	4.15

that our adversarial examples are written in C# and Java, we choose compatible compilers and linters to perform this evaluation. Specifically, for compilers, we choose javac and JIT for Java; and

Table 8. Code highlighted in grey denotes original statements in input programs. Right after them is the deadcode that we insert. ✓(resp., ✗) means compilers or linters catch (resp., miss) the deadcode.

How dead code is inserted	Compilers		Linters	
Java: <code>int a=...; int b=a;</code> <code>if (a*b<0){deadcode}</code>	OpenJDK: javac	✗	CheckStyle	✗
	OpenJDK: java JIT	✓	findbugs	✗
C#: <code>int a=...; int b=a;</code> <code>if (a*b<0){deadcode}</code>	Mono: mcs	✗	uncrustify	✗
	.Net compiler: csc	✗	Roslyn	✗
Java: <code>int a=...;</code> <code>if (Math.pow(a,2)<0){deadcode}</code>	OpenJDK: javac	✗	CheckStyle	✗
	OpenJDK: java JIT	✗	findbugs	✗
C#: <code>int a=...;</code> <code>if (Math.Pow(a,2)<0){deadcode}</code>	Mono: mcs	✗	uncrustify	✗
	.Net Compiler: csc	✗	Roslyn	✗
Java: <code>int x=Math.max(a,b); or int x=Math.min(a,b);</code> <code>if (x<a or x<b){deadcode} or if (x>a or x>b){deadcode}</code>	OpenJDK: javac	✗	CheckStyle	✗
	OpenJDK: java JIT	✗	findbugs	✗
C#: <code>int x=Math.Max(a,b); or int x=Math.Min(a,b);</code> <code>if (x<a or x<b){deadcode} or if (x>a or x>b){deadcode}</code>	Mono: mcs	✗	uncrustify	✗
	.Net Compiler: csc	✗	Roslyn	✗
Java: <code>... double a=Math.random();</code> <code>if (a<0){deadcode}</code>	OpenJDK: javac	✗	CheckStyle	✗
	OpenJDK: java JIT	✗	findbugs	✗
C#: <code>... Random rand=new Random(); int a=rand.Next(...);</code> <code>if (a<0){deadcode}</code>	Mono: mcs	✗	uncrustify	✗
	.Net Compiler: csc	✗	Roslyn	✗
Java: <code>String s="" or "str";</code> <code>if (s.length()>0 or <0){deadcode}</code>	OpenJDK: javac	✗	CheckStyle	✗
	OpenJDK: java JIT	✗	findbugs	✗
C#: <code>string s="" or "str";</code> <code>if (s.Length>0 or <0){deadcode}</code>	Mono: mcs	✗	uncrustify	✗
	.Net Compiler: csc	✗	Roslyn	✗
Java: <code>String s=str.substring(n,m);</code> <code>if (!str.contains(s) or str.indexOf(s)==-1){deadcode}</code>	OpenJDK: javac	✗	CheckStyle	✗
	OpenJDK: java JIT	✗	findbugs	✗
C#: <code>string s=str.Substring(n,m);</code> <code>if (!str.Contains(s) or str.IndexOf(s)==-1){deadcode}</code>	Mono: mcs	✗	uncrustify	✗
	.Net Compiler: csc	✗	Roslyn	✗
Java: <code>String s=str.substring(n,m);</code> <code>if (s.length()>str.length()){deadcode}</code>	OpenJDK: javac	✗	CheckStyle	✗
	OpenJDK: java JIT	✗	findbugs	✗
C#: <code>string s=str.Substring(n,m);</code> <code>if (s.Length>str.Length){deadcode}</code>	Mono: mcs	✗	uncrustify	✗
	.Net Compiler: csc	✗	Roslyn	✗
Java: <code>String s=str1+...;</code> <code>if (s.length()<str1.length()){deadcode}</code>	OpenJDK: javac	✗	CheckStyle	✗
	OpenJDK: java JIT	✗	findbugs	✗
C#: <code>string s=str1+...;</code> <code>if (s.Length<str1.Length){deadcode}</code>	Mono: mcs	✗	uncrustify	✗
	.Net Compiler: csc	✗	Roslyn	✗

Table 9. Evaluating key components of DaK in attacking code2vec. “Original” denotes the robustness score of code2vec attacked by DaK in its original form.

Labels Methods	init	mergeFrom	size	isEmpty	clear	remove	value	load	add	run
Original	0.00	8.11	21.29	0.00	0.00	4.27	28.40	0.00	0.00	0.00
w/o <i>Destroyer</i>	10.02	37.20	28.31	7.89	8.90	19.10	35.12	14.09	17.83	20.21
w/o <i>Finder</i>	22.90	30.80	39.02	25.68	20.82	37.21	60.75	12.47	18.75	34.31
w/o <i>necessary</i> requirement	69.73	55.21	74.48	48.35	32.21	58.47	87.28	41.82	70.09	61.12

csc and mcs for C#. javac is the standard compiler for Java programming language that attempts to remove deadcode in compile time. JIT is the Java just-in-time compiler that can remove deadcode in execution time. csc is the C# compiler in the .NET Framework, and mcs is a C# compiler on Mono, an open-source implementation of .NET Framework. Regarding linters, we select CheckStyle and findbugs for Java, uncrustify and Roslyn for C#. We examine every single adversarial example created by DaK for attacking code2vec, GGNN, or CodeBERT using the corresponding compilers and linters. Results show that in almost all cases the compilers and linters can not discover deadcode in adversarial examples. The only exception goes to JIT which recognizes the opaque predicate shown in the first row of Table 8. Overall, we conclude that common compilers and linters can not detect deadcode that we insert into input programs to create adversarial examples.

4.1.7 Alternative Configurations of DaK. We examine the contribution of two key functions of DaK—destroying significant features of inputs (designated as “Without Destroyer”) and finding *critical features* of *suppliers* (designated as “Without Finder”)—considering the rest of DaK is obviously necessary. In addition, we also validate Wang et al. [2021]’s definition of *critical features* by examining the alternative proposed by Rabin et al. [2021]. For brevity, we evaluate alternative configurations of DaK only in attacking code2vec, on which DaK achieved the best results.

Without Destroyer. To show the importance of destroying significant features of input programs, we create adversarial examples by injecting *critical features* of a *supplier* directly into the original version of input programs. The second row in Table 9 shows that without *Destroyer*, the increase of the robustness score of code2vec is significant. This study demonstrates that even though the greedy approach of *Destroyer* may only find local optimal, input programs are severely weakened, together with the *critical features* of the *supplier*, resulting in highly effective adversarial examples.

Without Finder. In this study, we show the importance of *critical features* by inserting the whole *supplier* into the tampered inputs, without finding their *critical features* beforehand. The third row in Table 9 shows that inserting the whole *supplier* significantly hinders DaK’s effectiveness; the increase of the robustness score in most cases is even higher than “Without Destroyer”.

Without Necessary Requirement As we explained previously, Rabin et al. [2021] do not consider the *necessary* requirement of Definition 3.1 for *critical features*. To demonstrate the importance of the *necessary* requirement, we remove it from *Reduce* and *Mutate*, the underlying algorithm of *Finder*, such that the new *critical features* to be found do not need to satisfy the *necessary* requirement. The last row in Table 9 shows the results. For all target labels, code2vec’s robustness scores increase the most by far, which indicates Rabin et al. [2021]’s definition of *critical features* is rather inaccurate.

To sum up, this study demonstrates the necessity of *Destroyer* and *Finder*, in addition, it also validates our definition of *critical features*.

4.2 Effectiveness of Everl against Existing Attacks and DaK

4.2.1 Defense Setups. First, we pick outlier detection [Yefet et al. 2020] and adversarial training, two strongest defenses reported in Yefet et al. [2020], as baselines for this experiment. Here (and

Table 10. Comparing Everl against outlier detection and adversarial training in defending attacks to code2vec. Results of the strongest defense (*i.e.*, the highest robustness scores) *w.r.t.* each attack are marked in bold.

Attack Methods	Defense Methods	init	mergeFrom	size	isEmpty	clear	remove	value	load	add	run
DAMP	Outlier Detection	74.32	99.98	99.58	99.22	98.77	94.50	90.87	60.27	88.97	96.49
	Adversarial Training	78.17	98.91	99.39	98.91	85.09	89.29	99.47	88.29	95.90	94.48
	Everl	85.42	100.0	100.0	100.0	98.23	99.03	100.0	100.0	100.0	100.0
DaK	Outlier Detection	0.00	19.18	22.30	0.90	1.09	4.27	28.40	0.00	0.00	0.00
	Adversarial Training	0.00	53.22	58.72	62.07	36.16	19.24	59.05	0.00	16.10	19.30
	Everl	70.20	85.28	83.19	86.76	75.98	74.83	83.18	74.12	64.28	75.89

Table 11. Comparing Everl with outlier detection and adversarial training in defending against adversarial attacks to GGNN.

Attack Methods	Defense Methods	Results
DAMP	Outlier Detection	85.92
	Adversarial Training	89.98
	Everl	91.35
DaK	Outlier Detection	19.58
	Adversarial Training	29.98
	Everl	67.35

Table 12. Comparing Everl with outlier detection, adversarial training, and RoPGen in defending against adversarial attacks to CodeBERT.

Attack Methods	Defense Methods	private methods	create an instance	set up fields	output class	sets the image
DAMP	Outlier Detection	83.28	93.21	89.16	90.38	86.85
	Adversarial Training	85.28	95.16	92.23	90.88	88.82
	RoPGen	87.92	97.58	94.58	92.84	91.55
	Everl	91.33	98.91	99.21	97.10	97.31
Imitator	Outlier Detection	89.35	96.52	100.0	100.0	99.51
	Adversarial Training	93.21	99.56	100.0	100.0	100.0
	RoPGen	93.86	100.0	100.0	100.0	100.0
	Everl	95.21	100.0	100.0	100.0	100.0
DaK	Outlier Detection	43.20	54.29	57.29	64.18	52.18
	Adversarial Training	56.92	69.94	78.92	80.83	84.21
	RoPGen	60.10	71.85	81.46	80.93	87.63
	Everl	68.65	80.28	89.83	91.83	91.86

throughout Section 4.2), adversarial training refers to Yefet et al. [2020]’s realization which treats successful adversarial examples (*i.e.*, any example that succeeds in fooling models to predict the target label) found by their own attack as the solution to the inner maximization problem. Both baselines are proposed to defend against DAMP, in particular, they consider the original setting of DAMP presented in Section 4.1.1. For a fair comparison between baselines and Everl, we set Everl to defend against DAMP in its original setting too. Next, we replace DAMP with DaK as the attack method for all defenses (including Everl) to deal with so that we can evaluate DaK in attacking hardened models *w.r.t.* various defense mechanisms. Same as above, DaK also considers DAMP’s original setting; thus, their effectiveness in attacking defended models can be fairly compared. It is worth mentioning that RoPGen [Li et al. 2022], another notable defense method in the literature, is proposed to defend against attacks to authorship attribution models. As we previously explained, the author of a code snippet is not a semantically-adhering property. Thus, the outcome of a defense can not be defined in the same way the outcome of an attack can not be defined. For this reason, we do not directly compare Everl with RoPGen in its original setting. Instead, we transplant RoPGen to defend CodeBERT in code documentation task. Similarly, we also transplant outlier detection and adversarial training to defend CodeBERT. In this way, we can evaluate the robustness of hardened CodeBERT *w.r.t.* each defense against Imitator, DAMP, and DaK.

4.2.2 Defense Results. Table 10-12 depicts the effectiveness of every defense method in each aforementioned setting. Results show that Everl is the most effective defense in all cases. The reason that outlier detection is ineffective is (1) multiple variable names are renamed, so it is hard to determine which one is really the outlier; (2) outlier detection cannot handle other transformations than

Table 13. Time (in hours) taken to train code2vec with Everl for defending against DAMP or DaK.

Attack Methods	Defense Methods	Training Time
DAMP	Adversarial Training	29.4
	Everl	155.1
DaK	Adversarial Training	41.4
	Everl	274.3

Table 14. Time (in hours) taken to train GGNN with Everl for defending against DAMP or DaK.

Attack Methods	Defense Methods	Training Time
DAMP	Adversarial Training	70.5
	Everl	344.7
DaK	Adversarial Training	108.0
	Everl	528.1

variable renaming (e.g., operands permutation) for creating adversarial examples. Although adversarial training outperforms outlier detection, it is not as effective as Everl because the adversarial examples they use are very likely not as strong as those used in Everl. Despite taking extra measures (e.g., augmenting training sets with adversarial examples, improving robustness of sub-networks), RoPGen lacks the theoretical support that Everl enjoys, hence it is less effective than Everl in practice. From the attack perspective, we find that DaK is the strongest among all attack methods regardless of the model being attacked and the defense method being applied. For example, when attacking code2vec in the presence of existing defenses (using models' average robustness scores again as the metric), DaK is more than ten times as effective as DAMP against outlier detection (i.e., 7.61 vs. 90.3), and almost three times as effective as DAMP against adversarial training (i.e., 32.4 vs. 92.8). GGNN's robustness scores display a similar margin by which DaK beats DAMP against outlier detection or adversarial training. Together with the results reported in Section 4.1, we conclude DaK significantly outperforms state-of-the-art in attacking either undefended or hardened models regardless of the defense mechanism.

4.2.3 Efficiency of Everl. Table 13-15 depict the time (in hours) taken to train code2vec, GGNN and CodeBERT with adversarial training and Everl for defending against DAMP, Imitator and DaK. We can see that Everl is considerably less efficient than adversarial training due to Everl's process of finding the strongest adversarial examples. Despite the overhead, models will only need to be trained once to achieve the highest robustness against potentially an arbitrary number of attacks in test-time. Thus, we believe the overhead Everl incurs is justified.

Table 15. Time (in hours) taken to train CodeBERT with Everl for defending against DAMP, Imitator, or DaK.

Attack Methods	Defense Methods	Training Time
DAMP	Adversarial Training	11.6
	Everl	64.5
Imitator	Adversarial Training	19.5
	Everl	186.6
DaK	Adversarial Training	15.5
	Everl	148.3

5 RELATED WORK

Neural Models of Code. Deep neural networks have displayed cutting-edge performance to a variety of programming language tasks, such as method name prediction [Alon et al. 2019a,b; Fernandes et al. 2018; Wang and Su 2020], bug prediction [Allamanis et al. 2018; Pradel and Sen 2018; Wang et al. 2020], and program repair [Bader et al. 2019; Chen et al. 2019; Dinella et al. 2020]. As a few notable examples, code2vec [Alon et al. 2019b] represents a snippet of code as a continuously distributed vector and predicts a method's name from the vector. Gated Graph Neural Networks (GGNN) [Li et al. 2016] learns from graph structured data to find misused variables [Allamanis et al. 2018]. Wang et al. [2018] present three neural architectures for learning semantic embeddings from dynamic executions. CodeBERT [Feng et al. 2020] learns general-purpose representations for downstream software engineering tasks like natural language code search and code documentation generation.

Adversarial Attacks and Defenses for Machine Learning Models. Szegedy et al. [2014] first discovered that tiny changes to an input image can fool state-of-the-art deep neural networks. Later, Goodfellow et al. [2015] present a fast gradient sign method, which can generate stronger

adversarial examples in image classification. DeepFool [Moosavi-Dezfooli et al. 2016] computes adversarial examples based on an iterative linearization of the image classifier. CW [Carlini and Wagner 2017], an attack based on a different formulation of the attack objective, is shown to be general and highly effective against image classification models. In natural language processing, HotFlip [Ebrahimi et al. 2018] generates white-box adversarial examples by mutating characters of input for text classification. GeneticAttack [Alzantot et al. 2018] uses a black-box population-based optimization algorithm to generate semantically and syntactically similar adversarial examples. For models of code, Imitator [Quiring et al. 2019] adopts a Monte-Carlo tree search to find adversarial examples via a series of semantically-preserving transformations. Yefet et al. [2020] propose a gradient-based white-box approach capable of performing both targeted and untargeted attack. Srikant et al. [2020] use program obfuscations to generate perturbed adversarial programs for untargeted attack. ALERT [Yang et al. 2022] proposes a black-box method to generate naturalness aware adversarial programs for untargeted attack. Zhang et al. [2022] iteratively attack code snippets with semantically equivalent code mutators for untargeted attack. Zhou et al. [2022] propose an untargeted attack based on identifier substitution which can mislead DNNs to produce irrelevant code comments. All the above-mentioned attacks perform untargeted attack. The only exception is DAMP which adopts a white-box, gradient-based method whereas DaK performs black-box, discrete attack. From the defense perspective, adversarial training [Madry et al. 2018] is by far the most successful algorithm for training robust classifiers. To defend code models, Yefet et al. [2020] empirically examine two classes of defense (with and without re-training) and find adversarial training and outlier detection achieve the highest model robustness *w.r.t.* each category. Bielik and Vechev [2020] find that adversarial training alone is insufficient, and further refine program representations by focusing on parts most relevant to a given prediction. RoPGen [Li et al. 2022] combines data and gradient augmentation to improve the effectiveness of adversarial training. [Wang et al. 2022] propose a learning framework that utilize the normal form of data points to achieve guaranteed robustness, however, their approach is limited in practice because many program transformations can not be normalized (*e.g.*, deadcode injection in *Merger*). Compared to prior works, we propose a method that significantly outperforms the state-of-the-art in targeted attack. Defensively, our main contribution is a theoretical foundation for the application of adversarial training to defending against discrete adversarial attacks.

6 CONCLUSION

In this paper, we study discrete adversarial attack to code models, in which attackers create adversarial examples by transforming the original programs in a semantically-preserving manner. Technically, we present a novel and general discrete attack method with three key components: *Destroyer*, *Finder* and *Merger*. Our primary contribution from the defense side is a proof of the applicability of adversarial training to both discrete and continuous adversarial attacks. Empirically, we show our attack method, DaK significantly outperforms the state-of-the-art in attacking either undefended or hardened models; our defense method, Everl, is more effective than state-of-the-art in defending against adversarial attacks.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant No. 62232001, No. 62032010 and No. 62202220, Jiangsu Funding Program for Excellent Postdoctoral Talent, and CCF-Huawei Populus Grove Fund.

REFERENCES

- Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.
- Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=H1gKY09tX>
- Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
- Moustafa Alzantot, Yash Sharma Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. 2018. Generating Natural Language Adversarial Examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.
- Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.
- Pavol Bielek and Martin Vechev. 2020. Adversarial robustness for code. In *International Conference on Machine Learning*. PMLR, 896–907.
- Marc Brockschmidt. 2020. GNN-FiLM: Graph Neural Networks with Feature-wise Linear Modulation. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 1144–1152. <https://proceedings.mlr.press/v119/brockschmidt20a.html>
- Sébastien Bubeck. 2015. Convex Optimization: Algorithms and Complexity. *Found. Trends Mach. Learn.* 8, 3–4 (nov 2015), 231–357. <https://doi.org/10.1561/22000000050>
- Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 39–57.
- Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. <https://doi.org/10.18653/v1/N19-1423>
- Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*.
- 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 (Volume 2: Short Papers)*. 31–36.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.
- Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured Neural Summarization. In *International Conference on Learning Representations*.
- Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. *stat* 1050 (2015), 20.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
- Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).
- TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. 2018. A multimodal deep learning method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018), 773–788.
- Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).
- Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. 2016. Gated Graph Sequence Neural Networks. In *Proceedings of ICLR’16*.
- Zhen Li, Guenevere Qian Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022. RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1906–1918. <https://doi.org/10.1145/3510003.3510181>
- Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=rJzIBfZab>

- Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1332–1349.
- Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.
- Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading Authorship Attribution of Source Code using Adversarial Learning. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 479–496. <https://www.usenix.org/conference/usenixsecurity19/presentation/quiring>
- Md Rafiqul Islam Rabin, Vincent J Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 441–452.
- Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Mumbai, India) (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 111–124. <https://doi.org/10.1145/2676726.2677009>
- David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (Oct. 1986), 533–536.
- Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2020. Generating Adversarial Computer Programs using Optimized Obfuscations. In *International Conference on Learning Representations*.
- Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).
- Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014*.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
- R Vinayakumar, Mamoun Alazab, KP Soman, Prabakaran Poornachandran, and Sitalakshmi Venkatraman. 2019. Robust intelligent malware detection using deep learning. *IEEE Access* 7 (2019), 46717–46738.
- Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embeddings for Program Repair. In *International Conference on Learning Representations*.
- Ke Wang and Zhendong Su. 2020. Blended, Precise Semantic Program Embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (London, UK) (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 121–134. <https://doi.org/10.1145/3385412.3385999>
- Yizhen Wang, Mohammad Alhanahnah, Xiaozhu Meng, Ke Wang, Mihai Christodorescu, and Somesh Jha. 2022. Robust Learning against Relational Adversaries. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). <https://openreview.net/forum?id=WBp4dli3No6>
- Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 137 (Nov. 2020), 27 pages.
- Yu Wang, Ke Wang, and Linzhang Wang. 2021. WheaCha: A Method for Explaining the Predictions of Code Summarization Models. <https://doi.org/10.48550/ARXIV.2102.04625>
- Eric Wong, Leslie Rice, and J. Zico Kolter. 2020. Fast is better than free: Revisiting adversarial training. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=BJx040EFvH>
- Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-trained Models of Code. *arXiv preprint arXiv:2201.08698* (2022).
- Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
- Zhenlong Yuan, Yongqiang Lu, Zhaoquo Wang, and Yibo Xue. 2014. Droid-sec: deep learning in android malware detection. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 371–372.
- 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 Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–40.
- Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald Gall. 2022. Adversarial robustness of deep code comment generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–30.

Received 2022-11-10; accepted 2023-03-31