软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007425] [CSTR: 32375.14.jos.007425]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

CoDefense: 面向对抗性攻击的多粒度代码归一化防御方法*

田 朝, 邝仕琦, 闫 明, 王海弛, 陈俊洁

(天津大学 智能与计算学部, 天津 300350)

通信作者: 陈俊洁, E-mail: junjiechen@tju.edu.cn

摘 要: 近年来, 以代码为输入的预训练模型在许多基于代码的关键任务中取得了显著的性能优势, 但这类模型可能易受到通过保留语义的代码转换实现的对抗性攻击, 这种攻击会显著降低模型鲁棒性并可能进一步引发严重的安全问题. 尽管已有对抗性训练方法通过生成对抗性样本作为增强数据来提升模型鲁棒性, 但其有效性和效率在面对不同粒度和策略的未知对抗性攻击时仍显不足. 为了克服这一局限性, 提出一种基于代码归一化的预训练代码模型对抗性防御方法, 命名为 CoDefense. 该方法的核心思想是作为代码模型的一个前置数据处理模块, 通过多粒度代码归一化技术, 对训练阶段的原始训练集和推理阶段的代码输入进行归一化预处理, 以避免潜在对抗性样本对代码模型的影响. 这种策略能够高效地防御不同粒度和策略的对抗性攻击. 为验证 CoDefense 的有效性和效率, 针对 3 种先进的对抗性攻击方法、3 种流行的预训练代码模型以及 3 个基于代码的分类和生成任务, 共设计了 27 个实验场景进行全面的实证研究. 实验结果表明, CoDefense 相较于最先进的对抗性训练方法, 在防御对抗性攻击方面显著提升了有效性和效率. 具体而言, CoDefense 平均成功防御了 95.33% 的对抗性攻击. 同时, 在时间效率上, CoDefense 相对于对抗性训练方法平均提升了 85.86%.

关键词: 对抗性防御; 预训练代码模型; 深度学习

中图法分类号: TP311

中文引用格式: 田朝, 邝仕琦, 闫明, 王海弛, 陈俊洁. CoDefense: 面向对抗性攻击的多粒度代码归一化防御方法. 软件学报. http://www.jos.org.cn/1000-9825/7425.htm

英文引用格式: Tian Z, Kuang SQ, Yan M, Wang HC, Chen JJ. CoDefense: Defending Method Against Adversarial Attacks with Multi-granularity Code Normalization. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7425.htm

CoDefense: Defending Method Against Adversarial Attacks with Multi-granularity Code Normalization

TIAN Zhao, KUANG Shi-Qi, YAN Ming, WANG Hai-Chi, CHEN Jun-Jie

(College of Intelligence and Computing, Tianjin University, Tianjin 300350, China)

Abstract: In recent years, pre-trained models that take code as input have achieved significant performance gains in various critical code-based tasks. However, these models remain susceptible to adversarial attacks implemented through semantic-preserving code transformations, which can severely compromise model robustness and pose serious security issues. Although adversarial training, leveraging adversarial examples as augmented data, has been employed to enhance robustness, its effectiveness and efficiency often fall short when facing unseen attacks with varying granularities and strategies. To address these limitations, a novel adversarial defense technique based on code normalization, named CoDefense, is proposed. This method integrates a multi-granularity code normalization approach as a preprocessing module, which normalizes both the original training data during training and the inputcode during inference. By doing so, the proposed method mitigates the impact of potential adversarial examples and effectively defends against attacks of diverse types and granularities. To evaluate the effectiveness and efficiency of CoDefense, a comprehensive experimental study is constructed, encompassing 27 scenarios across three representative adversarial attack methods, three widely-used pre-trained code models, and three

* 基金项目: 国家自然科学基金 (62322208, 62232001, 12411530122)
收稿时间: 2024-09-05; 修改时间: 2024-10-17; 采用时间: 2025-02-19; jos 在线出版时间: 2025-08-20

code-based classification and generation tasks. Experimental results demonstrate that CoDefense significantly outperforms state-of-the-art adversarial training methods in both robustness and efficiency. Specifically, it achieves an average defense success rate of 95.33% against adversarial attacks and improves time efficiency by an average of 85.86%.

Key words: adversarial defense; pre-trained model of code; deep learning

1 引 言

随着深度学习技术的迅速发展, 许多研究人员已将其应用于解决各种基于代码的软件工程任务, 并取得了显著成果, 极大地推动了软件开发和维护的发展[1−4]. 最近提出的基于自注意力机制的 Transformer 架构[5]在处理长距离依赖问题时表现出色, 衍生出了一系列最先进的深度学习模型, 包括 PLBART[6]和 CodeGPT[7]等预训练代码模型. 此类预训练代码模型利用预训练技术从海量的代码语料库中学习和获取代码知识, 并被进一步应用于诸如漏洞检测[8]、代码克隆检测[9]和代码摘要生成[10]等下游任务.

与计算机视觉[11,12]和自然语言处理[13,14]领域的深度学习模型一样, 预训练代码模型也存在模型鲁棒性问题[15−17]. 现有研究表明[18,19], 通过对预训练代码模型的原始代码输入执行特定的保留语义的代码转换 (例如变量名替换), 能够生成与原始代码语义等价的对抗性样本, 进而误导预训练代码模型生成完全不同的预测结果. 考虑到代码模型已部署在漏洞检测等关键任务, 此类对抗性攻击可能会带来巨大的危害[16]. 例如, 攻击者可以利用对抗性攻击方法生成含有漏洞的对抗性代码, 欺骗代码模型将其标记为“无漏洞代码”.

近年来, 已有一系列针对以代码为输入的预训练模型的对抗性攻击方法[15,16,18,19]被提出, 例如 ALERT[19]和 StyleTransfer[16]. 一般来说, 这类方法通常包含两个主要步骤: (1) 设计保留语义的代码转换规则; (2) 基于某种搜索策略在规则定义的空间中找到攻击成功的对抗性样本. 例如, ALERT 通过变量名替换规则, 结合代码自然性和遗传算法来引导攻击过程[19]. StyleTransfer 设计了 8 种保留语义的结构转换规则 (例如 while 和 for 循环语句的转换), 并利用模型预测变化引导攻击过程[16]. 目前, 针对预训练代码模型的对抗性攻击方法主要围绕攻击粒度和攻击策略两个方面展开研究, 旨在高效地生成对抗性样本. 在攻击粒度方面, 现有的对抗性攻击方法设计了多个级别的保留语义的代码转换规则, 例如变量名级别的变量名替换[18], 语句级别的死代码插入[20]和代码块级别的代码结构转换[16]. 在攻击策略方面, 已有对抗性攻击方法通过随机选择[21]或预定义的启发式方法[18], 来确定特定粒度下的代码转换规则的攻击位置与攻击内容 (例如变量名粒度攻击的替换位置和候选变量名) 以生成对抗性样本. 此外, 不同粒度和不同策略的对抗性攻击方法产生的对抗性样本也不尽相同, 这给设计有效的对抗性防御方法带来了巨大的挑战.

现阶段, 对抗性训练 (adversarial training) 是一种最广泛使用的缓解此类对抗性攻击威胁的防御方法, 并已被应用在诸多研究工作中[15,16,19,20]. 一般来说, 对抗性训练策略主要包括两个步骤: 首先, 通过特定的对抗性攻击方法生成对抗性样本, 以揭示代码模型中的鲁棒性缺陷; 然后, 利用这些对抗性样本增强训练数据, 对受到攻击的代码模型进行对抗性训练, 进而增强其鲁棒性. 尽管对抗性训练策略已被表明是一种相对有效的对抗性防御方法, 但其仍然存在很大的局限性.

(1) 无法有效地防御不同粒度的对抗性攻击. 现有的对抗性攻击方法涉及多种不同的攻击粒度 (例如, 变量名替换[18]、死代码插入[20]或代码结构转换[16]), 而不同粒度的对抗性攻击方法生成的对抗性样本是截然不同的. 因此, 基于特定粒度的对抗性攻击方法生成的对抗性样本进行对抗性训练, 无法有效地防御不同粒度的对抗性攻击技术[15,20].

(2) 无法有效地防御不同策略的对抗性攻击. 基于代码转换规则生成对抗性样本时, 攻击空间非常庞大. 以基于变量名替换的对抗性攻击为例, 所有有效的变量名都可以作为候选变量名, 导致潜在的对抗性样本数量极其庞大. 这意味着即使是涉及相同粒度的对抗性攻击方法, 其不同的攻击策略也会产生不同的对抗性样本. 这种对抗性攻击的多样性使得对抗性训练无法全面涵盖所有潜在的攻击. 在实际应用中, 对抗性训练后的模型面对未知或未覆盖的同粒度对抗性样本时仍然脆弱[15].

(3) 对抗性训练的效率较低. 现有的对抗性攻击方法通常需要基于模型预测来引导攻击过程, 因此频繁地调用代码模型导致对抗性样本生成过程效率低下[16]. 由于对抗性训练依赖于此类对抗性样本, 因此其效率也会受到限制. 此外, 在面对不同粒度和策略的对抗性攻击时, 对抗性训练可能需要多次执行, 这进一步降低了其效率. 这表明对抗性训练在实际应用中的时间成本非常高, 因此提升对抗性防御的效率仍是一个亟待解决的问题.

基于此, 本文提出了一种对抗性防御方法 CoDefense, 其能够更有效地防御多粒度、不同策略的对抗性攻击, 并显著提升时间效率, 同时不损害原始模型的性能和代码嵌入质量. 不同于已有的对抗性训练策略, CoDefense 的核心思想是通过多粒度代码归一化策略将 (训练阶段的) 原始训练集和 (推理阶段的) 原始代码输入进行预处理, 以避免潜在的对抗性样本输入到代码模型, 因此能够有效防御不同粒度和策略的对抗性攻击. 基于这一核心思想, CoDefense 的多粒度代码归一化策略包括 3 种类型: 变量名归一化、死代码消除和代码结构归一化. 具体来说, CoDefense 的部署包括两个阶段: (1) 在训练阶段, CoDefense 对原始训练集进行代码归一化, 并使用处理后的代码数据训练防御增强的代码模型; (2) 在推理阶段, CoDefense 对输入的原始代码 (可能包括对抗性样本) 进行与训练阶段相同的代码归一化处理, 然后再输入到增强后的代码模型中, 从而避免可能存在对抗性扰动的代码直接输入到代码模型. 此外, CoDefense 的创新之处还在于其作为代码模型的一个前置的数据处理模块, 与代码模型集成, 共同构成整体预测系统, 而非依赖于基于对抗性样本的对抗性训练. 与最先进的对抗性训练方法相比, CoDefense 能够更有效地防御不同粒度和策略的对抗性攻击. 由于 CoDefense 不依赖于生成的攻击成功的对抗性样本, 因此相比于对抗性训练方法, 其显著提升了时间效率.

为了探索 CoDefense 与对比方法在防御对抗性攻击方面的有效性和效率, 本文在 27 个实验场景 (3 个对抗性攻击方法×3 个代码模型×3 个数据集) 下进行了大规模的实证研究, 涵盖了 3 种最先进的对抗性攻击方法 (即 WIR-Random[21]、ALERT[19]和 StyleTransfer[16])、3 种广泛使用的预训练代码模型 (即 CodeBERT[1]、CodeGPT[7]和 PLBART[6]) 以及 3 个关键的基于代码的分类和生成任务 (即漏洞检测[8]、代码克隆检测[9]和代码摘要生成[10]). 实验结果表明, CoDefense 在防御有效性和效率方面均优于最先进的对抗性训练方法. 具体来说, 相比于对抗性训练方法, CoDefense 平均提升了 10.75% 的防御成功率, 并平均降低了 65.14% 的错误防御率. 同时, CoDefense 完成全部实验场景的防御过程的总运行时间为 59.82 h, 而对抗性训练方法则需 130.56–997.81 h. 另外, 我们研究了 CoDefense 和对抗性训练对于原始模型性能和嵌入质量的影响. 结果表明, CoDefense 不会损害原始模型的性能和嵌入质量, 而对抗性训练方法在部分实验场景会损害原始模型的性能, 进一步表明了 CoDefense 在提升模型鲁棒性和保持模型性能方面的优势. 综上所述, 本文的主要贡献总结如下.

(1) 本文提出了一种基于代码归一化的预训练代码模型对抗性防御方法 CoDefense, 该方法可以作为代码模型的前置处理模块, 通过多粒度代码归一化策略将 (训练阶段的) 原始训练集和 (推理阶段的) 原始代码输入进行预处理, 以避免对抗性样本影响代码模型, 进而能够有效地防御不同粒度和策略的对抗性攻击.

(2) 本文在 3 种先进的对抗性攻击方法、3 种流行的预训练代码模型和 3 个基于代码的分类和生成任务共 27 个实验场景下进行了广泛的实证研究. 实验结果表明, 相比于对抗性训练方法, CoDefense 平均提升了 10.75% 的防御成功率和 85.86% 的时间效率, 并降低了 65.14% 的错误防御率. 这展现了 CoDefense 相对于最先进的对抗性训练策略的有效性和高效性.

(3) 本文实现并开源了 CoDefense, 将相关代码开源在工具主页 https://github.com/tianzhaotju/CoDefense, 以便其他研究者开展未来研究和实际使用.

2 背景知识

2.1 预训练代码模型

近年来, 基于深度学习的预训练模型在基于代码的软件工程任务上取得了显著进展, 这些模型在智能代码场景中展现出广泛的应用前景, 我们称之为预训练代码模型[16,22,23]. 这种预训练-微调范式通常包括两个阶段: 预训练阶段和微调阶段. 在预训练阶段, 代码模型在大型未标记语料库上基于自监督的预训练任务来学习通用代码知识,

如通过掩码词元预测 (masked token prediction)[5,24]来训练代码模型对代码片段中缺失部分的预测能力. 进入微调阶段后, 经过预训练的代码模型可以在特定的下游任务上进行进一步的微调. 这一步骤使得代码模型能够适应不同的任务需求, 从而提升其在特定任务上的性能. 通过这种分阶段的学习方式, 预训练代码模型能够在保持通用性的同时, 实现对特定任务的优化.

一般来说, 预训练代码模型根据模型架构可分为 3 类: Encoder-only 模型[1,2]、Decoder-only 模型[7,25]和 Encoder-Decoder 模型[3,6]. Encoder-only 预训练代码模型主要使用双向 Transformer 编码器来学习代码的词元表示. 通过让每个词元相互关注, 增强了模型的代码理解能力. 相比之下, Decoder-only 预训练代码模型通常采用 Left-to-Right Transformer 架构, 让当前的词元更关注前一个词元, 以更好地捕获词元之间的关联. 被广泛使用的 CodeGPT[7]是一个代表性的基于 Transformer 的 Decoder-only 预训练代码模型. 此外, 最近的研究[26,27]探索了 Encoder-Decoder 预训练代码模型, PLBART[6]和 CodeT5[3]是此类模型的典型代表. 在我们的研究中, 我们采用了最新的对抗性攻击的实证研究[16]中使用的所有 3 个预训练代码模型 (即 CodeBERT[1]、CodeGPT[7]和 PLBART[6]), 分别对应了 3 种模型架构, 实验模型的具体细节将在第 3.2 节介绍. 通过在相应任务的数据集上对预训练代码模型进行微调, 预训练代码模型在许多基于代码的任务中取得了突破性的进展, 自动化的代码智能可以协助软件开发并显著提高开发者的效率[1−3].

需要着重指出的是, 当前针对预训练代码模型所设计的对抗性攻击技术[15,16,19], 其核心关注点普遍聚焦于以代码作为直接输入的任务范畴内, 诸如代码分类任务与代码摘要生成等任务. 因此, 这些对抗性攻击技术尚无法直接迁移至以自然语言为输入的任务 (例如, 代码生成[28]). 为了保持与现有对抗性攻击技术的一致性, 本研究的焦点同样严格限定于处理代码输入的预训练模型范畴, 并遵循既有文献中的术语体系, 继续沿用“预训练代码模型”这一称谓, 以确保研究的连贯性与学术探讨的精准性.

2.2 对抗性攻击

通过利用上述预训练-微调范式, 预训练代码模型能够从大规模公开代码库中学习和获取领域知识, 这些先验知识可以进一步用于下游任务, 如漏洞检测[8]、代码克隆检测[9]和代码摘要生成[10]. 尽管预训练代码模型在许多代码相关任务中取得了优异的表现, 但它们依然存在安全风险和鲁棒性问题[15,16]. 最近的研究表明[15–17], 类似于计算机视觉和自然语言处理领域的深度学习模型, 预训练代码模型在处理两个语义等价的代码片段时可能会产生完全不同的预测结果. 其中产生错误输出的代码片段 (又称对抗性样本) 是对抗性攻击方法通过对原始代码执行特定的保留语义的代码转换而生成的. 接下来, 我们分别对基于代码的分类任务和生成任务给出了两个对抗性攻击的形式化定义.

对于分类任务, 给定一个代码片段 $x \in X$, 将该代码片段 x 处理成预训练代码模型 $M: X \to Y$ 所需的格式 (例如词元序列、抽象语法树、控制流图或数据流图), M 可以预测 x 的概率向量, 其中每个元素代表将 x 归类到相应类别的概率. 概率最大的类别即为 M 对 x 的最终预测结果. 如果 M 的预测结果 $M(x)$ 与 x 的真实标签 (表示为 $y \in Y$) 不同, 则表示 M 对 x 的预测是错误的; 否则, M 做出了正确的预测.

现有的针对预训练代码模型的对抗性攻击方法通常通过在目标输入执行特定的保留语义的代码转换, 从而生成对抗性示例[15,16,18,19]. 为便于理解, 我们将问题定义为: 给定目标代码片段 x, 对抗性攻击方法为目标代码模型 M 找到攻击成功的对抗性样本 x_{adv}. 具体来说, x_{adv} 需要满足 3 个条件 $(x_{\text{adv}} \triangleq x) \wedge (y = M(x)) \wedge (M(x) \neq M(x_{\text{adv}}))$, 下面给出具体的描述.

(1) $x_{\text{adv}} \triangleq x$ 表示满足语法约束 (例如 Java 语言的变量名只能包含字母、数字和下划线) 并保留原始代码 x 的代码语义 (即具有完全相同的功能并在给定任意相同输入的情况下得到相同的输出).

(2) $y = M(x)$ 表示我们只将 M 做出正确预测的测试输入视为目标输入, 这是因为根据现有工作[15,19], 分析对此类输入的鲁棒性分析更有意义.

(3) $M(x) \neq M(x_{\text{adv}})$ 表示 x_{adv} 是从 x 生成的攻击成功的对抗性样本, 成功误导代码模型 M 的预测分类结果.

对于生成任务来说, 鉴于代码摘要生成模型的输出包含了多种可能, 直接套用代码分类任务中用以判断对抗

性攻击成功与否的标准是不切实际的. 因此, 为合理地评估此类攻击, 先前的研究广泛采纳了基于 BLEU 的度量方法, 该方法通过量化代码摘要间的文本相似度来反映对抗性攻击对模型输出的影响程度. 本文遵循这一评估策略, 给出了 3 个对应条件: 首先, 生成任务中的对抗性样本也需同时遵循分类任务中既定的要求 (1); 其次, 原始预测摘要与参考摘要之间的 BLEU 为非零, 这确保了在没有攻击干扰的情况下, 模型能够生成与真实摘要互相匹配的内容 (对应分类任务的要求 (2)); 最后, 对抗性摘要与参考摘要之间的 BLEU 为零, 这表明对抗性摘要与参考摘要之间几乎不存在任何匹配关系 (对应分类任务的要求 (3)).

2.3 对抗性防御

为了防御对抗性攻击, 目前最常见且有效的方法是对抗性训练[15,16,19,20]. 对抗性训练通过将生成的对抗性样本作为增强数据, 与已有的原始训练集结合进行训练. 该策略尽可能地在不影响代码模型原始性能的前提下, 提高模型防御对抗性攻击的能力. 对应第 2.2 节的描述, 我们分别给出了基于代码的分类任务和生成任务的对抗性防御的形式化定义.

对于分类任务来说, 给定一个代码片段 $x \in X$, 其真实类别标签为 $y \in Y$. 针对代码模型 $M: X \to Y$, 对抗性攻击方法生成了攻击成功的对抗性样本 x_{adv}. 我们通过对抗性防御方法得到了增强后的代码模型 $M_{\text{defense}}: X \to Y$. 具体来说, 成功的对抗性防御需要满足 3 个条件 $(x_{\text{adv}} \triangleq x) \wedge (y = M_{\text{defense}}(x)) \wedge (M_{\text{defense}}(x) = M_{\text{defense}}(x_{\text{adv}}))$, 下面给出具体的描述.

(1) $x_{\text{adv}} \triangleq x$ 表示满足语法约束并保留原始代码 x 的代码语义.

(2) $y = M_{\text{defense}}(x)$ 表示 M_{defense} 能够对给定输入 x 得到正确预测类别.

(3) $M_{\text{defense}}(x) = M_{\text{defense}}(x_{\text{adv}})$ 表示 M_{defense} 没有被对抗性样本 x_{adv} 误导为错误的分类结果, 成功地防御了对抗性样本 x_{adv} 的攻击.

类似地, 对于生成任务来说, 同样需要满足 3 个条件: 首先, 生成任务中的对抗性样本也需遵循分类任务中既定的要求 (1); 其次, 原始预测摘要与参考摘要之间的 BLEU 为非零, 这确保了在没有攻击干扰的情况下, 模型能够生成与真实摘要互相匹配的内容 (对应分类任务的要求 (2)); 最后, 对抗性摘要与参考摘要之间的 BLEU 也为非零, 这表明对抗性摘要无法误导模型的预测结果 (对应分类任务的要求 (3)).

3 基于代码归一化的对抗性防御方法

在本节中, 我们首先通过一个简化的动机示例来直观地激发我们的关键思想, 接着介绍了 CoDefense 的概述设计, 然后介绍 CoDefense 的多粒度代码归一化策略以及实际场景下的部署和应用过程.

3.1 动机示例

我们使用一个简化的示例来阐明多粒度对抗性防御方法的动机与必要性. 给定一个原始的代码片段, 对抗性攻击技术可以通过执行特定的等价代码转换来生成一系列对抗性样本. 总结了最近 5 年的主要会议和期刊上发表的最先进的针对代码模型的对抗性攻击方法 (具体细节将在第 4.3 节介绍), 并将它们的对抗性攻击策略分为 3 类: 变量名替换、死代码插入和代码结构转换.

基于变量名替换的对抗性攻击通常利用随机策略或启发式搜索策略, 引导对抗样性本的生成过程[18]. 该策略在满足代码语法和语义约束的前提下, 不产生语法错误且不改变代码语义, 通过逐步改变原始代码的预测结果, 最终生成攻击成功的对抗性样本. 以图 1 中的 AdvCode-1 为例, 对抗性攻击方法通过把原始代码中的变量名 *n* 和 *k* 分别替换为新的变量名 *nan* 和 *kick* 生成了攻击成功的对抗性样本, 该对抗性样本成功地误导了代码模型的预测结果.

类似地, 基于死代码插入的对抗性攻击通常利用随机策略或启发式搜索策略, 迭代地寻找死代码插入的位置和内容[20]. 死代码插入规则不会产生语法错误也不会改变代码语义, 通过迭代攻击过程最终生成攻击成功的对抗性样本. 以图 1 中的 AdvCode-2 为例, 对抗性攻击方法通过在原始代码的第 4 行插入死代码 while (i!=i){int b=10; i+=b;}以及在第 11 行插入死代码 if (false) {a+=n;}生成了对抗性样本, 该对抗性样本成功地误导代码模型的预测

结果.

对于基于代码结构转换的对抗性攻击方法, 它们首先设计一系列等价的代码结构转换规则, 然后通过搜索策略确定代码转换位置和转换规则[16]. 等价代码结构转换不会产生语法错误, 也不会影响代码语义, 通过迭代攻击过程最终生成攻击成功的对抗性样本. 以图 1 中的 AdvCode-3 为例, 对抗性攻击方法通过把原始代码的第 3 行和第 4 行的 for 循环语句转换为等价的 while 循环语句, 并将第 3 行的 i+=1 和第 4 行的 j+=1 分别转换为 i=i+1 和 j=j+1 得到了对抗性样本, 其成功地误导了代码模型的预测结果.

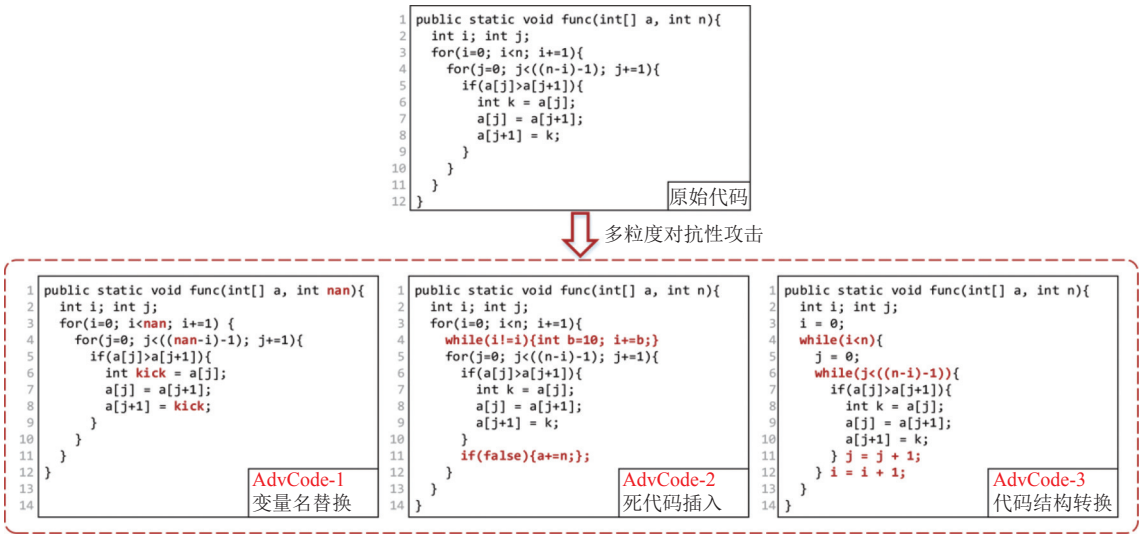


图 1 一个简化的动机示例

通过上述例子, 可以发现对抗性样本涉及了 3 种粒度的对抗性攻击方法 (即变量名替换[18]、死代码插入[20]和代码结构转换[16]), 不同粒度的对抗性样本有着显著的区别. 目前, 最常见的对抗性训练方法通常只针对特定粒度生成对抗性样本[19,21], 然后通过数据增强的方式将其添加到原始训练集中, 对模型进行对抗性训练. 然而, 这种方法无法覆盖到所有粒度的对抗性样本, 因此不能有效地防御不同粒度的对抗性攻击. 这促使我们设计一种能够针对 3 种粒度的对抗性攻击进行有效防御的方法. 此外, 基于代码转换规则生成对抗性样本时, 攻击空间非常巨大. 以基于变量名替换的对抗性攻击为例, 所有有效的变量名都可以作为候选变量名, 导致潜在的对抗性样本数量极其庞大. 这意味着即使是具有相同粒度的对抗性攻击方法, 不同的攻击策略也会产生不同的对抗性样本. 这种多样性使得对抗性训练无法全面涵盖所有潜在的攻击, 面对未知或未覆盖的同粒度对抗性样本时仍然脆弱[19]. 这促使我们设计一种对抗性防御方法, 能够有效防御不同策略的所有对抗性攻击.

基于上述发现, 设计了一种多粒度代码归一化策略, 能够将训练阶段的原始训练集和推理过程的原始代码输入进行归一化预处理, 以避免潜在的对抗性样本直接输入到代码模型, 进而从根本上提高代码模型的对抗性防御能力. 接下来的部分, 本文将详细介绍该方法.

3.2 概 述

在本文中, 我们提出了一种对抗性防御方法 (称为 CoDefense), 其能够更有效地防御多粒度、不同策略的对抗性攻击, 并显著提升时间效率. 图 2 展示了 CoDefense 的方法流程图. 代码归一化策略由 3 个主要部分组成: 变量名归一化、死代码消除和代码结构归一化. 不同于已有的对抗性训练策略, CoDefense 的核心思想是通过多粒度代码归一化将 (训练阶段的) 原始训练集和 (推理阶段的) 原始代码输入进行预处理, 以避免潜在的对抗性样本直接输入到代码模型, 因此能够有效防御不同粒度和策略的对抗性攻击. 具体来说, CoDefense 的部署包括两个阶段: (1) 在训练阶段, CoDefense 对原始训练集进行代码归一化处理, 并使用处理后的数据训练代码模型; (2) 在推理阶

段, CoDefense 对输入的原始代码 (可能是对抗性样本) 进行与训练阶段相同的代码归一化处理, 然后再输入到代码模型中, 从而避免可能存在对抗性扰动的代码直接输入到代码模型. 此外, CoDefense 的创新之处在于, 其作为代码模型的一个前置的数据处理模块与代码模型集成, 共同构成整体预测系统, 而非依赖于基于对抗性样本的对抗性训练. 与最先进的对抗性训练方法相比, CoDefense 能够更有效地防御不同粒度和策略的对抗性攻击, 且对原始模型的性能和嵌入质量影响更小. 由于 CoDefense 不依赖于生成的对抗性样本, 因此相比于对抗性训练方法, 其显著提升了时间效率. 需要强调的是, 本文提出的 CoDefense 方法在所有的编程语言上都是适用的. 由于计算资源限制, 在最流行的 Java 上进行了实现和评估, 工具的实现方式可以在本项目的 GitHub 主页获取. 后续将会在第 3.3 节介绍 CoDefense 的多粒度代码归一化策略, 然后在第 3.4 节介绍 CoDefense 在实际场景中的部署流程.

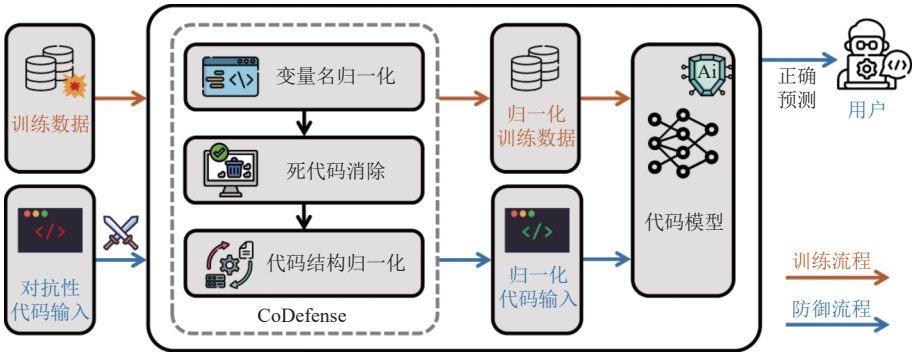


图 2 CoDefense 方法流程图

3.3 多粒度代码归一化

根据第 3.1 节和第 3.2 节的介绍, 针对预训练代码模型的对抗性攻击策略可分为 3 类: 变量名替换[18]、死代码插入[20]和代码结构转换[16]. 针对这 3 种粒度的对抗性攻击策略, 分别设计了 3 种粒度的代码归一化方式: 变量名归一化、死代码消除和代码结构归一化.

3.3.1 变量名归一化

先前的研究表明[29], 软件开发人员享有高度的自由度来定义变量名, 这些变量名的复杂性与多样性几乎不受限制. 这一基本特性对构建于大型代码语料库之上的预训练代码模型带来了巨大挑战, 因为这些模型不得不面对一个极为庞大且高度稀疏的变量名空间. 尤为棘手的是, 这种变量名空间的广阔性加剧了模型在面对测试集中未曾预见变量名时的脆弱性, 尤其是在面对精心设计的对抗性攻击技术时, 这些攻击可以利用模型对低频或未知变量名的敏感反应, 从而有效降低代码模型的预测性能和鲁棒性. 然而现有的对抗性攻击方法[15,18–20], 大部分都涉及变量名替换的攻击策略. 为了有效防御这种粒度的对抗性攻击, 我们首先考虑变量名归一化的防御策略.

由于每个软件开发人员对于变量名的命名方式展现出的显著差异, 这为代码模型的训练增设了重重障碍[30]. 最新研究成果[31]进一步揭示, 代码输入中夹杂的噪声变量名会对代码模型的性能造成显著损害. 具体而言, 基于变量名替换的对抗性攻击策略, 在保持代码语义不变的前提下, 会对代码模型的训练过程施加干扰, 导致模型难以捕捉并学习代码的内在语义特征, 最终引发代码模型鲁棒性和泛化能力的双重退化.

为解决上述问题, 促使代码模型聚焦于代码的深层次语义而非表面的变量名特征, 我们引入了变量名归一化技术. 具体而言, CoDefense 将代码中的所有变量名依次替换为 $(Var1, Var2, Var3, \ldots)$. 这种归一化方式能够避免基于变量名替换的对抗性攻击干扰模型的预测过程, 同时归一化后的代码保证了语法合法性且保留了原代码的语义信息. 实验结果表明, 这种变量名归一化策略能够 100% 防御不同策略的基于变量名替换的对抗性攻击方法 (详细实验结果参见第 5 节). 更进一步, 变量名归一化可以促使代码模型学习更深层次的语义信息, 降低了模型对于变量名形式的过度依赖. 第 5 节及第 6.1 节的实验结果充分表明这种变量名归一化策略在增强代码模型鲁棒性的同时, 并未对模型的原始性能及代码嵌入分布造成太大负面影响. 在此步骤中, CoDefense 通过对给定代码实施变

量名归一化处理, 为后续的代码归一化处理奠定了基础.

3.3.2 死代码消除

根据现有研究, Brown 等人[32]将死代码称为“Lava Flow”, 解释为在动态变化的代码中未使用的代码片段. Mantyla 等人[33]将死代码定义为曾经使用过但目前不再使用的代码片段. Wake[34]将死代码定义为未使用的变量、参数、方法或类等. Martin[35]将死代码定义为从未执行的代码片段, 例如含有不可能发生条件的 if 语句的主体或从未调用的方法等. 尽管先前的研究提供的定义略有不同, 但死代码是不必要的, 消除冗余的死代码不会影响代码的运行结果. 本文中, 我们引用软件工程对“死代码”的定义, 指代未使用或无法访问的代码片段[36].

如表 1 所示, 我们总结了最近几年主要会议和期刊的对抗性攻击方法中常见的死代码插入规则[20,30,37–46], 并据此设计了 6 类死代码消除规则 (共计 19 条规则). 具体来说, 我们系统地考虑了所有常见的代码结构, 即循环结构 (规则 1)、分支结构 (规则 2) 和顺序结构 (规则 3–6). 表 1 中详细解释了这 6 类死代码消除规则, 并附有示例说明. 对于每类规则, 它可能包括若干条具体规则, 例如 $Loop_{d1}$ 包含 while 和 for 语句的多种情况. 总的来说, CoDefense 针对 6 类变换制定了 19 条具体规则. 请注意, 并非所有死代码消除规则都适用于所有编程语言, 例如, $Statement_{d3}$ 中的空语句“;;;;”不被 Python 支持. 限于篇幅, 我们将所有具体规则的详细信息放在本文的项目主页上. 基于这些死代码消除规则, CoDefense 利用正则匹配消除给定代码中的死代码片段, 并将其作为后续代码归一化过程的输入.

表 1 CoDefense 的死代码消除规则的描述

ID	死代码消除规则	描述	例子
1	$Loop_{d1}$	消除无法访问或主体为空的循环语句	while (false) {Body2;} while (condition){}
2	$Branch_{d2}$	消除无法访问或主体为空的分支语句	if (0) {Body2;} if (condition) {} else{}
3	$Statement_{d3}$	消除空语句	;;;; {{}{}}{};
4	$Assert_{d4}$	消除恒真断言	assertTrue(true); assertTrue(1>0);
5	$Print_{d5}$	消除无意义输出或打印语句	System.out.print(""); System.out.println("0000");
6	$Unused_{d6}$	未使用的变量或类	int a; boolean b; 注: a和b在之后的代码中未被使用

3.3.3 代码结构归一化

CoDefense 通过对代码输入应用保留语义的代码结构转换, 进一步进行代码结构归一化处理. 受到基于代码结构转换的对抗性攻击研究和代码重构研究启发[15,28,38,46–57], 我们在 CoDefense 中设计了 13 类代码结构归一化规则 (共计 51 条规则). 具体来说, 我们系统地考虑了所有常见的代码结构, 即循环结构 (规则 1)、分支结构 (规则 2–4) 和顺序结构 (规则 5–13). 表 2 中详细解释了这 13 个类别, 并附有示例说明. 对于每一类转换规则, 可能包括若干条具体规则. 例如, $Branch_{s4}$ 包含了||的转换和&&的转换. 总的来说, CoDefense 针对 13 类转换制定了 51 条具体规则. 同样, 并非所有代码结构归一化规则都适用于所有编程语言. 例如, 规则 $In/Decrement_{s10}$ 中的++的转换和——的转换不适用于 Python. 此外, 在规则 $Constant_{s9}$ 中, 新定义的变量不能与代码中已有的变量名相同, 且应该按照变量名归一化的方式进行命名, 否则可能会导致语法错误或改变原始语义. 限于篇幅, 我们将所有具体规则的实现放在本项目开源主页上.

接下来, 说明如何引导这些代码结构转换规则来进行代码结构归一化. 由于每个规则涉及两种或多种代码结构, 我们需要设计一种评价指标来自动衡量转换前后的代码质量, 以引导代码结构归一化的过程. 受到代码重构相关工作的启发[58–60], 本文选择代码质量指标作为评价指标, 其被广泛用于识别设计缺陷 (例如代码异味) 以及作为重构推荐方法的适应度函数. 代码质量指标能够评估开发人员所感知的代码质量, 识别设计缺陷并推荐从开发人

员的角度来进行有意义的代码重构. 基于现有研究[58–60], 我们考虑两种类型的代码质量指标, 用于自动衡量转换前后的代码质量. 接下来, 我们将描述这两类代码质量指标.

表 2 CoDefense 的代码结构归一化规则的描述

ID	等价结构转换规则	描述	例子(转换前)	例子(转换后)
1	Loop_{s1}	for和while循环语句的等价转换	for (i=0;i < 9;i++){ Body;}	i=0; while (i<9) {Body;i++;}

2	Branch_{s2}	if-else和switch-case条件语句的等价转换	i=0;
			if (i==1){
			Body;
			}
				i=0;
				switch(i){
				case1:
				Body;}

3	Branch_{s3}	是否使用条件语句中的逆否表达式 (>, <, >=, <=, ==, !=, &&, ‖)	if (a>b){ Body;}	if(!(a<=b)){ Body;}
4	Brabch_{s4}	是否拆分多条件语句(‖, &&)	if (a > b){Body;} if (a==b){Body;}	if ((a > b)‖(a==b)){ Body;}
5	Type_{s5}	基础类型转换(int, float)	int *a*=0;	long *a*=0;
6	Variable_{s6}	局部变量声明的位置(代码开始处、首次使用处)	int a; Body; a=c+d;	Body; int a; a=c+d;
7	Variable_{s7}	局部变量声明和初始化是否在同一行	int a; a=0;	int a=0;
8	Variable_{s8}	相同类型的变量定义/声明是否在同一行	int a=0; int b=1;	int a, b=0, 1;
9	Constant_{s9}	常量和变量的等价转换(int, float, double, long, String)	int a=b+10;	int c=10; int a=b+c;
10	$\text{In/Decrement}_{s10}$	自增/减操作的等价转换(++, −−)	a++; −−b;	a+=1; b=b–1;
11	Operand_{s11}	二元操作符两端的操作数是否交换 (<, >, >=, <=, ==, !=, &&, ‖, +, *)	if (a > b){Body1;} if (c && d){Body2;}	if (b < a){Body1;} if (d && c){Body2;}
12	Combined_{s12}	是否使用复合赋值 (+=, −=, *=, /=, %=, <<=, >>=, &=, |=, ^=)	a*=10; b%=2;	a=a*10; b=b%2;
13	Curly_{s13}	是否省略花括号	if (a > b){ b+=1;}	if (a > b) b+=1;

(1) 代码复杂度. 降低代码复杂度是代码重构的主要目标之一[60]. 我们使用 WMC (weighted methods per class)[61]指标来评估类的复杂度. WMC 的计算方式为给定类内所有方法的 McCabe's cyclomatic complexity[62]的总和. 具体来说, 给定代码, 我们首先需要获取其控制流图, McCabe's cyclomatic complexity 的计算公式为 $(E-N+2)$, 其中 E 表示控制流图的边的数量, N 表示控制流图的节点的数量. WMC 越高, 代码复杂度越高.

(2) 代码可读性. 代码可读性是软件工程领域比较关注的一个重要代码质量属性, 可读性更高的代码能提高开发人员对于代码的可理解性. 为了测量可读性质量属性, 我们利用了两个最先进的指标. 第 1 个指标是 B&W[59], 它利用代码结构属性 (例如行的长度、分支的数量等) 来衡量代码的可读性. 第 2 个指标是 SRead[63], 它利用了一组完全基于代码词典分析的特征 (例如变量名的特异性、代码连贯性等) 来衡量代码的可读性.

与现有工作保持一致[15,16], CoDefense 会依此选择表 2 中定义的代码结构转换规则, 以执行代码结构归一化处理. 针对每一条选定的代码结构转换规则, 源代码中可能存在多个潜在的应用位置, CoDefense 则依据它们在代码中出现的位置顺序进行排序, 并依次对各个代码位置应用当前的代码结构转换规则. 对于每一条给定的代码结构转换规则与特定代码位置, CoDefense 首先会评估转换前的代码质量指标 $Quality_{\text{before}}=[WMC_{\text{before}}, B\ \&\ W_{\text{before}}, SRead_{\text{before}}]$, 然后在该代码位置应用特定代码结构归一化规则得到转换后的代码, 并重新计算相应的代码质量指标 $Quality_{\text{after}}=[WMC_{\text{after}}, B\ \&\ W_{\text{after}}, SRead_{\text{after}}]$. 为了量化代码结构转换对代码质量的影响, CoDefense 采用了一种代码质量平均提升率的指标, 即对每个质量指标分别计算其提升百分比, 并获取平均值, 该公式具体为:

$$\frac{1}{3}\times\sum_{i=1}^{3}\left(\frac{Quality_{\text{after}}[i]-Quality_{\text{before}}[i]}{Quality_{\text{before}}[i]}\right)\times 100\%,$$

其中, i 代表不同质量指标的索引. 这一计算不仅考虑了代码质量的相对提升, 还通过平均化处理, 使得不同维度上的提升能够公平地反映在整体改进效果上. 若计算结果显示, 经过转换后的代码在整体代码质量上相较于转换前有所提升, 则 CoDefense 决定保留该代码位置上的当前代码结构转换规则, 并继续迭代此过程, 直至所有预定义的代码结构归一化规则在所有可能的代码位置上都得到了充分的评估与可能的应用. 通过这种迭代优化策略, CoDefense 能够有效提升代码的整体质量, 最终输出一个经过精细的代码归一化处理且代码质量显著提升的代码版本.

3.4 CoDefense 的部署和应用

上文介绍了 CoDefense 的多粒度代码归一化过程, 通过 3 种代码归一化处理策略, 可以得到处理后的代码数据. 接下来, 我们将详细描述 CoDefense 方法的部署和应用流程. 如图 2 所示, CoDefense 作为代码模型的一个前置的数据处理模块, 与代码模型集成, 共同构成整体预测系统. 它的输入包括原始代码模型 M、原始训练集 $Data^{\text{train}}$ 和原始测试集 $Data^{\text{test}}$.

在训练阶段, CoDefense 方法首先利用代码归一化方法对原始训练数据迭代进行代码归一化预处理, 得到归一化的训练集 $Data^{\text{train}}_{\text{normal}}$. 然后, CoDefense 利用归一化后的训练集对原始模型进行训练以得到防御增强的代码模型 M_{defense}. 在推理阶段, 对于任意给定的代码输入 $data^{\text{test}}$, CoDefense 同样会将其进行代码归一化处理, 得到归一化后的代码输入 $data^{\text{test}}_{\text{normal}}$. 最后, 我们把归一化后的代码输入到防御增强的代码模型 M_{defense} 中, 得到正确的预测结果. 第 5 节和第 6.1 节将会深入分析 CoDefense 增强后的模型的性能表现和代码嵌入分布.

在整个工作流程中, CoDefense 与代码模型集成为一个整体系统运行. 通过将训练集和推理过程的代码输入都进行归一化处理, 以避免可能是对抗性样本的代码直接输入到代码模型, 进而从根本上提高了代码模型的对抗性防御能力.

4 实验分析

4.1 实验研究问题

为了全面探究 CoDefense 在防御对抗性攻击方面的有效性和效率, 并评估 CoDefense 对目标模型的原始性能影响, 本文设计了以下 3 个研究问题.

研究问题 1: 与现有方法相比, CoDefense 在防御不同对抗性攻击方面的效果如何?

研究问题 2: 与现有方法相比, CoDefense 在防御不同对抗性攻击方面的效率如何?

研究问题 3: 与现有方法相比, CoDefense 对目标代码模型的原始性能影响如何?

4.2 数据集和模型

为了充分评估 CoDefense, 本文考虑了最新研究[15]中的 3 个关键的基于代码的任务 (即漏洞检测[8]、代码克隆检测[9]和代码摘要生成[10]) 和 3 个流行的预训练代码模型 (即 CodeBERT[1]、CodeGPT[7]和 PLBART[6]), 共计 9 组实验对象. 表 3 展示了这些实验对象的具体信息, 其中第 2、3 列分别表示数据集和代码模型的组合, 第 4–8 列分别是训练集规模、验证集规模、测试集规模、测试指标以及模型测试集性能.

● 数据集: 与最新研究一致[16], 我们选择了 3 个代表性的数据集来进行实验评估, 分别涵盖了以代码为输入的分类任务和生成任务. 漏洞预测 (vulnerability detection) 任务旨在预测给定的代码片段是否存在漏洞. 对于该任务, 采用 OWASP (open Web application security project) 数据集作为实验数据, 它是一个用于评估漏洞检测工具的 Java 测试套件, 被广泛地应用于漏洞检测任务[64–66]. 与现有工作[16]保持一致, 我们也采用该数据集的 1.1 版本, 它包含了更多的数据, 更适合训练代码模型. 具体来说, 该版本包含 13 041 个训练样本、4 000 个验证样本和 4 000 个测试样本. 代码克隆检测 (code clone detection) 任务旨在检测两个给定的代码片段在语义上是否等价. 对于该任务, 采用 BigCloneBench[67]作为实验数据集, 它是一个被广泛使用的代码克隆检测数据集, 包含了多种克隆类型的项目内/项目间代码克隆片段. 为了更好地评估对抗性攻击和对抗性防御方法, 采用 Yang 等人[19]提供的过滤数据集.

他们的过滤策略包括删除未标记的数据样本、平衡两个标签 (克隆和非克隆) 的数据分布以及使数据处于计算友好的规模. 具体来说, 该数据集包含 90 102 个训练样本、4 000 个验证样本和 4 000 个测试样本. 代码摘要生成 (code summarization) 任务旨在针对给定的代码片段, 自动生成其对应的自然语言描述性摘要. 为实现这一目标, 本文选取了 CodeSearchNet[68]作为实验数据集, 该数据集作为代码摘要生成领域的重要基准, 广泛涵盖了 6 种编程语言的代码数据. 为保持与现有研究的可比性和一致性[7,16,21], 本文同样专注于 Java 编程语言的数据子集, 并实施了数据过滤与采样策略. 具体而言, 本文从 CodeSearchNet 中筛选出专门针对 Java 的数据子集, 进一步构建了包含 164 923 个训练样本、5 183 个验证样本以及 4 000 个测试样本的数据集. 这一数据划分确保了本文实验设置与先前两个分类任务中的测试集规模相当, 从而便于进行性能对比与结果分析.

表 3　预训练代码模型在对应数据集上的评估结果

ID	数据集名称	模型	训练集规模	验证集规模	测试集规模	测试指标	测试集性能
1	Vulnerability detection	CodeBERT	13 041	4 000	4 000	Accuracy (%)	98.70
2		CodeGPT	13 041	4 000	4 000		97.45
3		PLBART	13 041	4 000	4 000		99.52
4	Code clone detection	CodeBERT	90 102	4 000	4 000	Accuracy (%)	96.33
5		CodeGPT	90 102	4 000	4 000		96.52
6		PLBART	90 102	4 000	4 000		96.82
7	Code summarization	CodeBERT	164 923	5 183	4 000	BLEU-4	18.69
8		CodeGPT	164 923	5 183	4 000		15.40
9		PLBART	164 923	5 183	4 000		17.54

● 预训练代码模型: 本文在第 2.1 节介绍了预训练代码模型的 3 个类别 (即 Encoder-only、Decoder-only 和 Encoder-Decoder). 现有研究[21]表明, 不同预训练代码模型的性能非常接近, 没有一个预训练代码模型可以在不同的任务或数据集上全部实现最佳性能. 例如, 对于 Encoder-only 架构的预训练代码模型, CodeBERT[1]在漏洞检测数据集上的表现要优于 GraphCodeBERT[2], 而在代码克隆检测数据集上则表现较差; 同样, 对于 Encoder-Decoder 架构的预训练代码模型, CodeT5[3]在漏洞测数据集上的表现优于 PLBART[6], 而在代码克隆检测数据集上则表现不佳. 因此, 与现有研究[16]保持一致, 对于每一类预训练代码模型, 我们选择一个代表性模型进行实验评估. 具体来说, 我们选择了 CodeBERT[1]作为 Encoder-only 模型的代表, CodeGPT[7]作为 Decoder-only 模型的代表, 以及 PLBART[6]作为 Encoder-Decoder 模型的代表. 本文分别基于相应的数据集, 对这 3 个预训练代码模型进行了微调. 微调过程中, 使用了现有研究[16]提供的相同超参数设置, 详细设置可参见本项目主页. 最终, 我们获得了 9 个微调后的代码模型作为研究对象. 表 3 的最后一列显示了这些代码模型在相应任务上的微调性能 (Accuracy 用于衡量漏洞检测和代码克隆检测任务、BLEU-4 用于衡量代码摘要生成任务). 尽管目前存在一些更先进的预训练代码模型 (例如 CodeT5+[69]、CodeGen[25]、StarCoder[67]和 Code Llama[70]), 但由于这些模型参数巨大, 受到计算资源和时间的限制, 难以在下游任务上进行微调和鲁棒性测试. 因此, 与最新的相关研究保持一致[16], 将本文的研究对象限定为这 3 个流行的预训练代码模型. 在未来的研究中, 我们会积极探索 CoDefense 在参数更多的预训练代码模型上的对抗性防御效果.

总体而言, 本文研究的实验对象是多样的, 涉及不同的任务、不同的数据规模、不同的预训练代码模型和不同的模型架构. 这有助于全面评估 CoDefense 的对抗性防御效果和效率.

4.3　对抗性攻击方法

表 4 总结了最近 5 年, 在主要会议和期刊上发表的最先进的针对以代码为输入的预训练模型的对抗性攻击方法. 与现有研究[16]保持一致, 本研究选择的对抗性攻击方法都是黑盒方法, 主要是因为白盒攻击方法具有两个限制: (1) 白盒攻击方法通常需要访问模型结构、参数信息、原始训练集等, 而这些信息在实际场景中往往很难获得, 攻击者通常只能通过访问提供的 API 来调用模型; (2) 白盒攻击往往是特定于模型和任务设计的, 而不同的模型具有不同的结构, 不同的任务侧重于不同的代码特征, 因此这些针对特定模型和任务而设计的白盒攻击方法难

以推广到其他模型和任务. 此外, Nguyen 等人[71]提出了恶意 API 插入 (malicious APIs insertion) 攻击, 将其定义为在正常代码中插入包含恶意代码的 API 或错误 API, 其中这些 API 可能是虚构的, 或是错误的 (来自伪造的第三方库). 在恶意 API 插入后, 含有这些 API 的代码在特定上下文或使用场景中执行时, 可能会触发中断或引发严重错误. 鉴于本文的研究重点在于保持代码语义一致的对抗性攻击方法, 因此假设任何破坏代码语义的攻击均会被识别. 本文所研究的对抗性攻击技术 (如重命名、死代码插入、等价结构转换等) 皆不改变代码的功能语义, 也不会在原始代码中引入缺陷或漏洞, 从而难以被漏洞扫描工具或测试用例检测到. 与之对比, 恶意 API 插入直接影响代码的语义一致性, 并可能引入新的漏洞, 因此不符合本文在对抗性攻击及其防御策略中的假设前提. 基于以上考虑, 并与现有对抗性攻击和防御研究[16,19,21]的常见假设一致, 本文未将恶意 API 插入攻击纳入实验考量. 未来工作中, 计划将恶意 API 插入攻击的防御视为潜在的扩展方向, 以进一步提升 CoDefense 的适用性和防御广度.

表 4　现有的针对代码输入的预训练模型对抗性攻击方法总结 (按出版物年份升序排列)

ID	对抗性攻击方法	出版物	白盒/黑盒	任务	对抗性攻击策略
1	DAMP[30]	OOPSLA 2020	白盒	Functionality classification Code completion	变量名替换 死代码插入
2	MHM[18]	AAAI 2020	黑盒	Functionality classification	变量名替换
3	Srikant等人[39]	ICLR 2021	白盒	Functionality classification	变量名替换 死代码插入
4	Rabin等人[72]	IST 2021	白盒	Method name prediction	变量名替换 代码结构转换
5	Pour等人[73]	ICST 2021	黑盒	Method name prediction Code captioning Code search Code Summarization	变量名替换 代码结构转换
6	AVERLOC[45]	SANER 2022	黑盒	Code summarization	变量名替换 代码结构转换
7	ACCENT[74]	TOSEM 2022	黑盒	Code summarization	变量名替换
8	WIR-Random[21]	ISSTA 2022	黑盒	Vulnerability detection Code summarization	变量名替换
9	RoPGen[50]	ICSE 2022	黑盒	Authorship attribution	变量名替换 代码结构转换
10	CARROT[20]	TOSEM 2022	白盒	Functionality classification Clone detection Vulnerability detection	变量名替换 死代码插入
11	ALERT[19]	ICSE 2022	黑盒	Authorship attribution Clone detection Vulnerability detection	变量名替换
12	MixCode[75]	SANER 2023	白盒	Functionality classification Bug detection	变量名替换 死代码插入
13	CODA[15]	ASE 2023	白盒	Vulnerability detection Clone detection Authorship attribution Functionality classification Defect Prediction	变量名替换 代码结构转换
14	StyleTransfer[16]	FSE 2023	黑盒	Vulnerability detection Clone detection Code Summarization	死代码插入 代码结构转换

具体而言, 在表 4 展示的 8 种黑盒攻击中, 我们选择了 WIR-Random[21]、ALERT[19]和 StyleTransfer[16]这 3 种方法. 这是因为 MHM[18]、ACCENT[74]、WIR-Random[21]和 ALERT[19]均是利用变量名替换作为攻击策略, 其中 WIR-Random 和 ALERT 作为其中最先进、最有效的方法, 被选为我们的研究对象. 其他 4 种方法 Pour 等人[73]、AVERLOC[45]、RoPGen[47]和 StyleTransfer[16]则在语句/代码块级别执行保留语义的代码转换 (例如插入死代码或

将 while 循环语句转换为 for 循环语句), 而 StyleTransfer 作为其中最先进的方法, 集成了其余 3 种方法的全部代码转换攻击策略, 因此在实验过程中选择 StyleTransfer 作为该类别的代表性方法. 总的来说, 本研究选择了 3 种最典型或最先进的针对代码模型的对抗性攻击方法来评估 CoDefense 和对比方法的防御效果. 下面是针对所选的 3 种对抗性攻击方法的描述.

(1) WIR-Random[21]. 该方法首先利用 WIR (word importance rank) 策略来确定变量名的替换顺序. 具体来说, WIR 策略将变量名依次替换为"*UNK*", 然后计算替换前后模型的生成概率的差异, 并据此对每个变量名进行重要程度排序. 最后, WIR-Random 按重要程度顺序以随机的方式选择候选变量名替换原始变量名. 在实验过程中, 将候选变量名的数量限制为 15, 以避免攻击过程中产生的巨大的计算成本和时间开销.

(2) ALERT[19]. 该方法利用上下文感知变量名预测方法, 迭代地进行变量名替换. 在变量名选择策略方面, ALERT 采用了贪心算法和遗传算法两种策略. 在实验过程中, 我们同样把候选变量名的数量设置为 15 (与 WIR-Random 保持一致), 把遗传算法的最大迭代次数设置为 $5\times Num_i$ 和 10 之间的较大者 (其中 Num_i 表示给定代码片段中变量名的数量).

(3) StyleTransfer[16]. 该方法执行特定的保留语义的代码结构转换. 具体而言, 该方法包含以下代码结构转换策略: 1) 随机添加日志语句; 2) for 循环语句和 while 循环语句的互相转换; 3) 随机更改两个独立代码行的位置; 4) 重新排序二元条件语句; 5) switch 条件语句和 if 条件语句的互相转换; 6) 随机添加 try-catch 代码块; 7) 随机添加一段死代码; 8) 用非逻辑替换原有的布尔变量值. 然后, StyleTransfer 应用这些代码结构转换规则来生成 N 个候选对抗性样本, 并利用这些候选样本迭代地攻击模型. 在实验过程中, 我们把 N 设置为了 15, 以避免过大的计算成本和时间开销.

4.4 评测指标

为了衡量 CoDefense 和对比方法在对抗性防御方面的有效性和效率, 我们使用了如下的评测指标用于实验验证.

(1) 攻击成功率 *ASR* (attack success rate). *ASR* 用于衡量对抗性攻击方法成功生成对抗性样本的能力. 具体而言, 对于给定测试集 *TestX* (需要注意的是, *TestX* 只包含能够被目标模型正确预测的测试集, 具体的原因已经在第 2.2 节讨论), 攻击成功率的计算公式如下:

$$ASR = \frac{|TestX_{\text{attack-success}}|}{|TestX|} \times 100\% \tag{1}$$

其中, $|TestX|$ 代表给定测试集 *TestX* 中代码片段的总数, 而 $|TestX_{\text{attack-success}}|$ 代表对抗性攻击方法能够成功生成对抗性样本的数量. 对于对抗性攻击方法, *ASR* 越高意味着对抗性攻击方法越有效.

(2) 防御成功率 *DSR* (defense success rate). *DSR* 用于衡量对抗性防御方法成功防御对抗性攻击的能力. 具体而言, 对于给定测试集 *TestX*, 防御成功率的计算公式如下:

$$DSR = \frac{|TestX_{\text{defense-success}}|}{|TestX_{\text{attack-success}}|} \times 100\% \tag{2}$$

其中, $|TestX_{\text{attack-success}}|$ 代表对抗性攻击方法生成的攻击成功的对抗性样本的总数, 而 $|TestX_{\text{defense-success}}|$ 代表对抗性防御方法能够成功防御攻击成功的对抗性样本的数量. 对于对抗性防御方法, *DSR* 越高意味着对抗性防御方法越有效.

(3) 错误防御率 *FDR* (false defense rate). 对抗性攻击方法生成的对抗性样本可能未能攻击成功, 即目标模型仍能正确预测某些对抗性样本. 我们把这些样本称为攻击失败的对抗性样本. 然而, 对抗性防御方法可能会导致这些原本攻击失败的对抗性样本错误地变为攻击成功的对抗性样本, 这种误报情况称为错误防御. 因此, 设计了 *FDR* 用于衡量对抗性防御方法造成的此类负面影响. 具体而言, 对于给定测试集 *TestX*, 错误防御率的计算公式如下:

$$FDR = \frac{|TestX_{\text{defense-fail}}|}{|TestX_{\text{attack-fail}}|} \times 100\% \tag{3}$$

其中, $|TestX_{\text{attack-fail}}|$ 代表对抗性攻击方法生成的攻击失败的对抗性样本数量, 而 $|TestX_{\text{defense-faild}}|$ 代表对抗性防御方法把原本攻击失败的对抗性样本误报为攻击成功的对抗性样本的数量. 对于对抗性防御方法, *FDR* 越低意味着误报

越少.

(4) Accuracy (准确率). 准确率是测试集中正确预测样本的比例, 用于衡量代码模型在分类任务上的性能. 在本实验中, 准确率还用于测量对抗性防御方法对于代码模型整体性能的影响, 以更直观的方式展示对抗性防御方法是否会对原始模型的性能造成负面影响.

(5) BLEU-4. BLEU 已经被广泛用于评估生成任务中预测的文本与参考文本之间的文本相似度. 与先前研究的保持一致, 本文选用 BLEU 的一个变体——BLEU-4[15]作为衡量代码摘要生成任务性能的关键测试指标. 此处, BLEU-4 特别强调了使用长度为 4 个连续词汇的 *n*-gram (即 4-gram) 作为基本的匹配单元. 在本实验中, BLEU-4 用于衡量代码模型的性能以及对抗性防御方法对于代码模型整体性能的影响.

(6) 总运行时间 TRT (total running time). 我们用对抗性防御所花费的总运行时间来衡量对抗性防御方法的效率. 更少的运行时间意味着更高的防御效率.

4.5 实验设计

针对本文的 3 个研究问题, 我们均采用了目前最广泛使用的对抗性训练方法作为对比方法, 评估 CoDefense 在防御对抗性攻击方面的有效性和效率. 与现有研究[16]保持一致, 对于每个任务, 将原始测试集 $Data^{\text{test}}$ 均分成两部分, 分别记为 $Data^{\text{test-1}}$ 和 $Data^{\text{test-2}}$. 其中, $Data^{\text{test-1}}$ 作为微调集, 而 $Data^{\text{test-2}}$ 作为评估集. 对于对抗性训练方法, 我们分别使用对抗性攻击方法为微调集 $Data^{\text{test-1}}$ 中的每个样本生成一个攻击成功的对抗性样本. 需要注意的是, 如果当前的对抗性攻击方法无法生成攻击成功的对抗性样本, 则选择正确预测的置信度 (对于分类任务) 或 BLEU-4 分数 (对于生成任务) 下降最大的对抗性样本. 因此对于相同的任务, 不同的对抗性攻击方法和代码模型构建的对抗性微调集的大小保持一致. 最终, 我们得到了对抗性微调集 $Data_{\text{adv}}^{\text{test-1}}$, 并将其与原始训练集结合, 形成增强训练集以训练新的模型 M_{adv}. 为了保证实验的公平性, 在代码模型的对抗性训练过程中, 超参数的设置与原始模型 M 的训练完全相同. 类似地, 对于 CoDefense, 首先对原始训练集 $Data^{\text{train}}$ 进行代码归一化, 生成数据规模完全一致的归一化训练集 $Data_{\text{normal}}^{\text{train}}$, 然后使用相同的超参数训练得到新的模型 M_{defense}.

对于研究问题 1, 我们首先评估不同的对抗性攻击方法对于原始模型 M 在评估集 $Data^{\text{test-2}}$ 上的攻击成功率 (即 *ASR*). 然后, 同样使用评估集 $Data^{\text{test-2}}$, 我们分别评估了 CoDefense 和对抗性训练策略对于不同对抗性攻击方法的防御成功率 (即 *DSR*) 和错误防御率 (即 *FDR*), 以比较两种对抗性防御方法的有效性. 对于对抗性训练策略, 我们分别使用 3 种对抗性攻击方法生成对抗性样本作为增强数据对模型进行训练.

对于研究问题 2, 我们分别评估了 CoDefense 和对抗性训练策略的总运行时间 (即 *TRT*), 以比较两种对抗性防御方法的效率.

对于研究问题 3, 我们分别评估了基于 CoDefense 和对抗性训练策略得到的增强后的模型 (即 M_{adv} 和 M_{defense}) 和原始模型 M 在评估集 $Data^{\text{test-2}}$ 上的性能 (即 Accuracy 和 BLEU-4). 这有助于探索两种对抗性防御方法是否会对原始模型的性能造成负面影响.

4.6 实现及环境配置

本文使用 Python 3.8 实现了 CoDefense, 并基于 Tree-Sitter 0.20.4[76]和 srcML 1.0[77]对代码片段进行解析与代码归一化. 对于对抗性攻击方法 WIR-Random[21]、ALERT[19]和 StyleTransfer[16], 我们采用了对应作者提供的开源代码. 所有预训练代码模型 (即 CodeBERT[1]、CodeGPT[7]和 PLBART[6]) 均从 Huggingface[78]下载. 超参数的详细设置可以在我们的项目主页中找到. 所有的实验均在 Intel Xeon Gold-6326 服务器上完成, 操作系统为 Ubuntu 22.04.3 LTS, 内存为 512 GB, 配备两个 24 GB 的 GeForce RTX 3090 GPU.

5 结果分析

- 研究问题 1: 与现有方法相比, CoDefense 在防御不同对抗性攻击方法的效果如何?

表 5 详细展示了 CoDefense 及其 3 个对比方法在防御对抗性攻击方面的防御成功率 (*DSR*) 和错误防御率 (*FDR*). 值得注意的是, 第 5–7 列 (Adv-Train 列) 分别代表了基于 3 种对抗性攻击方法 (即 WIR-Random、ALERT

和 StyleTransfer) 的对抗性训练策略. 具体而言, 第 4 列 (Original 列) 的每个单元格表示对抗性攻击方法在原始模型上的 *ASR*, 第 5–8 列的每个单元格均包含两个指标: *DSR* (↑) 和 *FDR* (↓). 这些指标为我们提供了 CoDefense 和对比方法在不同对抗性攻击下的防御能力的直观衡量结果.

表 5　CoDefense 和对比方法防御对抗性攻击的效果 (%)

数据集	模型	对抗性攻击方法	Original (*ASR*)	Adv-Train			CoDefense
				WIR-Random	ALERT	StyleTransfer	
Vulnerability detection	CodeBERT	WIR-Random	14.69	88.97/0.12	78.97/0.12	81.72/0.48	100.00/0.00
		ALERT	9.22	91.21/0.06	92.86/0.06	91.21/0.06	100.00/0.00
		StyleTransfer	13.12	79.15/0.64	79.92/0.23	98.66/5.42	82.81/0.24
	CodeGPT	WIR-Random	11.05	92.13/0.17	87.04/0.98	68.98/5.70	100.00/0.00
		ALERT	7.93	86.45/0.00	87.10/0.22	77.92/0.11	100.00/0.00
		StyleTransfer	6.35	58.06/0.38	49.19/1.26	94.44/3.77	36.07/1.10
	PLBART	WIR-Random	22.74	97.13/0.32	89.85/0.91	88.08/1.10	100.00/0.00
		ALERT	23.59	93.4/0.00	98.09/0.13	90.85/0.07	100.00/0.00
		StyleTransfer	21.69	83.56/0.51	62.04/1.60	98.11/17.07	64.97/2.63
Code clone detection	CodeBERT	WIR-Random	21.44	93.98/1.18	93.98/1.38	84.09/4.34	100.00/0.00
		ALERT	21.13	90.22/0.79	99.27/0.33	78.00/3.14	100.00/0.00
		StyleTransfer	0.20	25.00/0.83	100.00/0.16	75.00/1.55	100.00/0.00
	CodeGPT	WIR-Random	19.50	97.88/0.58	88.89/3.72	73.28/8.21	100.00/0.00
		ALERT	9.39	93.41/0.46	96.15/0.23	64.29/0.46	100.00/0.00
		StyleTransfer	0.05	100.00/0.21	100.00/0.15	100.00/0.15	100.00/0.26
	PLBART	WIR-Random	17.14	94.29/0.87	92.19/2.05	69.07/13.23	100.00/0.00
		ALERT	12.76	93.55/0.53	96.37/0.59	60.08/1.42	100.00/0.00
		StyleTransfer	0.62	100.00/0.98	100.00/0.83	33.33/0.98	100.00/0.05
Code summarization	CodeBERT	WIR-Random	19.24	92.07/0.81	94.33/0.40	96.03/0.54	100.00/0.00
		ALERT	57.76	92.56/0.25	93.30/0.51	94.05/0.25	100.00/0.00
		StyleTransfer	9.58	95.08/0.87	100.00/0.64	91.80/0.75	100.00/1.52
	CodeGPT	WIR-Random	8.72	81.44/1.14	83.83/1.09	84.43/0.86	100.00/0.00
		ALERT	32.36	73.62/1.36	71.95/1.28	74.96/1.04	100.00/0.00
		StyleTransfer	3.43	72.73/1.18	75.76/1.83	84.85/1.13	90.00/2.61
	PLBART	WIR-Random	21.88	94.76/0.77	94.76/0.56	94.76/0.56	100.00/0.00
		ALERT	64.81	95.45/1.24	95.37/1.09	95.37/1.09	100.00/0.00
		StyleTransfer	6.49	92.44/3.09	92.44/3.09	92.44/3.09	100.00/1.64
	Average		**16.92**	**86.98/0.72**	**88.65/0.94**	**82.81/2.84**	**95.33/0.37**

在总计 27 个实验场景 (3 个数据集×3 个代码模型×3 个对抗性攻击方法) 的广泛评估中, CoDefense 在防御效果方面展现出了显著优势. 具体而言, CoDefense 在 24 种情况下的 *DSR* 优于全部对比方法, 而在 21 种情况下的 FSR 优于全部对比方法, 这充分展现了 CoDefense 在防御对抗性攻击方面的有效性. 在少数特定场景下, CoDefense 未能达到最优的表现. 深入分析此现象, 可归因于两个主要原因: 其一, 这些对抗性训练策略用到了数量更多的增强训练集 (包括原始训练集 $Data^{\text{train}}$ 和特定对抗性微调集 $Data^{\text{test-1}}_{\text{adv}}$), 而 CoDefense 仅利用 (与原始训练集具有相同的数据规模的) 归一化训练集 $Data^{\text{train}}_{\text{normal}}$ 进行训练, 这种不公平的对比在一定程度上限制了 CoDefense 的性能; 其二, 当生成的对抗性样本与本文的代码质量指标保持一致时, 部分对抗性样本可能会成功欺骗 CoDefense 的防御机制, 我们将在未来工作章节进一步讨论 (详见第 6.3 节). 然而, 即便面临这样的不公平对比, 从全局角度审视, CoDefense 在所有 27 个实验场景上仍然实现了平均 95.33% 的 *DSR* 和仅 0.37% 的 *FDR*. 相比之下, 3 种对比方法的 *DSR* 仅为 82.81%–88.65%, *FDR* 则为 0.72%–2.84%. 这些数据显示, 与对比方法相比, CoDefense 不仅平均提升了 10.75% 的 *DSR*, 并且平均降低了 65.14% 的 *FDR*, 从而显著提升了代码模型的整体防御能力.

从实验结果中还可以观察到, 在面对基于变量名替换的对抗性攻击方法 (即 WIR-Random 和 ALERT) 时,

CoDefense 展现出了完美的防御能力. 具体而言, CoDefense 在所有 18 个实验场景中均实现了 100.00% 的 *DSR* 和 0.00% 的 *FDR*, 而对比方法的 *DSR* 平均仅为 87.85%, *FDR* 则为 1.28%. 这一显著的对比差异进一步表明了 CoDefense 在防御不同策略的基于变量名替换的对抗性攻击时的全面有效性. 这一发现不仅表明了 CoDefense 在面对基于变量名的对抗性攻击的卓越防御能力, 同时也为相关领域的研究提供了新的视角和思路.

此外, 本文进一步采用了 Wilcoxon Signed-Rank[79]检验来验证 CoDefense 与 3 个对比方法的实验结果之间是否存在显著性差异, 其分别比较不同方法在每个实验场景下的 *DSR* 和 *FDR* 数值. 当置信度为 0.05 时, p-value 均小于 1.05×10^{-3}. 这表明了 CoDefense 在统计学上显著优于 3 个对比方法, 进一步肯定 CoDefense 的有效性.

结论: 相比于对比方法, CoDefense 能够更有效地防御对抗性攻击. 在 27 个实验场景下, CoDefense 相较于对比方法平均提升了 10.75% 的 *DSR*, 并平均降低了 65.14% 的 *FDR*. 特别是对于基于变量名替换的对抗性攻击方法 (即 WIR-Random 和 ALERT), CoDefense 在所有 18 个实验场景上均实现了 100.00% 的 *DSR* 和 0.00% 的 *FDR* 的完美防御, 进一步表明了 CoDefense 的有效性.

● 研究问题 2: 与现有方法相比, CoDefense 在防御不同对抗性攻击方法的效率如何?

为了进一步探究 CoDefense 的时间效率, 本文进行了 CoDefense 与 3 个对比方法的总运行时间的对比分析. 表 6 详细展示了 CoDefense 及其对比方法的实验结果, 其中第 3 列 (Original 列) 展示了原始模型在训练和测试过程中的总运行时间.

表 6　CoDefense 和对比方法的总运行时间 (min)

数据集名称	模型	Original	Adv-Train			CoDefense
			WIR-Random	ALERT	StyleTransfer	
Vulnerability detection	CodeBERT	31.37	280.39	5 191.18	1 011.61	61.02
	CodeGPT	23.87	376.57	4 662.99	765.31	53.62
	PLBART	27.25	413.04	6 191.78	995.53	57.17
Code clone detection	CodeBERT	77.03	272.90	4 329.48	288.85	256.76
	CodeGPT	82.67	441.72	6 051.52	575.39	265.47
	PLBART	105.83	548.04	9 136.45	664.78	290.14
Code summarization	CodeBERT	552.13	1 023.26	12 143.32	1 033.74	588.82
	CodeGPT	1 096.86	2 800.46	3 479.07	1 669.75	1 133.55
	PLBART	846.01	1 677.01	8 682.91	1 418.81	882.70
Total		**2 843.02**	**7 833.39**	**59 868.70**	**8 423.77**	**3 589.27**

总体而言, 在全部 9 个实验场景下, CoDefense 的总运行时间显著优于对比方法. 具体而言, CoDefense 在所有 9 个实验场景下的总运行时间为 3 589.27 min, 相比之下, 3 个对比方法的总运行时间则为 7 833.39–59 868.70 min. 深入分析这一显著差异的根源, 主要归因于对比方法在运行过程中高度依赖于耗时且复杂的对抗性样本生成过程. 这一生成过程不仅调用代码模型密集度高, 而且往往涉及多次迭代与调整, 从而显著增加了总体运行时间. 相比之下, CoDefense 通过其创新的设计和优化策略, 平均提升了高达 85.86% 的时间效率. 这一数据不仅彰显了 CoDefense 在单次运行中的优势, 更预示着在需要更新防御规则集并可能多次执行对抗性防御过程的实际部署场景中, CoDefense 将展现出更为显著的长期效率优势.

此外, 本文进一步采用了 Wilcoxon Signed-Rank[79]检验来验证 CoDefense 与对比方法的实验结果之间是否存在显著性差异, 其分别比较不同方法在每个实验场景下的 TRT 数值. 当置信度为 0.05 时, p-value 均小于 3.19×10^{-2}, 本实验说明了 CoDefense 在统计学上优于对比方法, 进一步表明了 CoDefense 的效率.

结论: 相较于对比方法, CoDefense 在防御对抗性攻击方面展现了更高的效率. 具体而言, 在全部 9 个实验场景下, CoDefense 相较于对比方法平均提升了高达 85.86% 的时间效率.

● 研究问题 3: 与现有方法相比, CoDefense 对目标代码模型的原始性能影响如何?

表 7 详细展示了 CoDefense 及其对比方法对模型原始性能的影响. 分析表 7 中的数据, 我们可以观察到在全部 9 个实验场景下, CoDefense 策略非但未对原始模型的性能造成任何损害, 反而展现了其保持乃至提升模型性

能的能力. 相比之下, 3 个对比方法则在不同程度上对原始模型的性能产生了负面影响, 例如在 PLBART 的漏洞检测和代码摘要生成任务、CodeBERT 的代码摘要生成任务中即有所体现. 这一结果与现有研究[15,19]的结论一致, 进一步验证了对抗性防御策略在保持模型原始性能方面的局限性. 在少数特定场景下, CoDefense 的性能提升幅度略逊于其他的对抗性训练策略. 这一现象可以归因于这些对抗性训练策略用到了数量更多的训练集 (包括原始训练集 $Data^{\text{train}}$ 和特定对抗性微调集 $Data^{\text{test-1}}_{\text{adv}}$), 而 CoDefense 仅依赖于 (与原始训练集具有相同的数据规模的) 归一化训练集 $Data^{\text{train}}_{\text{normal}}$ 进行训练. 这种训练数据规模的不公平对比, 在一定程度上限制了 CoDefense 在部分特定场景下的性能. 然而, 即便面临这样的不利条件, 从全局角度审视, CoDefense 在全部 9 个实验场景上, 相较于原始模型, 平均实现了 3.41% 的性能提升; 同时, 与另外 3 种对比方法相比, 更是平均取得了 7.55% 的显著性能提升. 这一结果不仅彰显了 CoDefense 策略的有效性, 也为其在复杂多变的应用场景中的广泛应用提供了实证基础.

表 7　CoDefense 和对比方法对模型原始性能的影响

数据集名称	模型	测试指标	Original	Adv-Train			CoDefense
				WIR-Random	ALERT	StyleTransfer	
Vulnerability detection	CodeBERT	Accuracy (%)	98.60	99.05	98.90	99.00	98.60
	CodeGPT		97.70	98.75	99.00	98.85	98.60
	PLBART		99.60	99.50 (↓)	99.55 (↓)	99.45 (↓)	99.65
Code clone detection	CodeBERT	Accuracy (%)	96.05	96.25	96.95	96.05	96.05
	CodeGPT		96.20	97.50	97.50	97.40	96.45
	PLBART		96.55	97.70	97.75	96.60	96.70
Code summarization	CodeBERT	BLEU-4	18.52	15.01 (↓)	14.43 (↓)	15.52 (↓)	21.17
	CodeGPT		15.43	16.39	16.46	16.74	16.66
	PLBART		17.76	15.06 (↓)	14.67 (↓)	14.67 (↓)	19.00

在实验设计中, 由于 3 种对比方法使用了 50% 的原始测试数据作为微调集, 因此研究问题 3 的评估只在剩余 50% 的测试集上进行. 为进一步对比使用和不使用 CoDefense 归一化的训练集在完整原始测试集上的表现, 我们补充了相关实验, 结果如表 8 所示. 分析表 8 的数据可见, CoDefense 在 7 个实验场景下对模型的原始性能未产生不利影响 (甚至略有提升), 而在 CodeBERT 的两个实验场景下对模型性能有轻微影响. 我们推测, 这可能是由于 CodeBERT 作为参数规模最小的模型, 可能在学习归一化代码的语义表示方面有所局限. 相比之下, CoDefense 在参数规模更大、更先进的 CodeGPT 和 PLBART 的全部任务上均实现了性能提升. 总之, 这些实验结果进一步表明, CoDefense 在保持模型原始性能方面的显著优势.

表 8　CoDefense 在完整测试集上对模型原始性能的影响

数据集名称	模型	测试指标	Original	CoDefense
Vulnerability detection	CodeBERT	Accuracy (%)	98.70	98.78
	CodeGPT		97.45	98.60
	PLBART		99.52	99.62
Code clone detection	CodeBERT	Accuracy (%)	96.33	94.67 (↓)
	CodeGPT		96.52	96.65
	PLBART		96.82	96.98
Code summarization	CodeBERT	BLEU-4	18.69	18.46 (↓)
	CodeGPT		15.40	15.42
	PLBART		17.54	17.88

此外, 我们还发现 CoDefense 不仅增强了模型的防御能力, 还在一定程度上提升了模型的整体性能 (尤其是 CodeGPT 和 PLBART). 这一新发现为未来工作提供了潜在的研究方向, 即探索 CoDefense 作为一种数据质量提升工具, 在增强模型鲁棒性的同时, 是否也能作为提升模型性能的有效手段.

结论: 在全部 9 个实验场景中, CoDefense 相比于对比方法对模型性能的负面影响更小. 具体来说, CoDefense 相较于原始模型平均实现了 3.41% 的性能提升, 而相较于 3 种对比方法平均实现了 7.55% 的性能提升. 这进一步

表明 CoDefense 在保持模型原始性能方面的显著优势.

6 讨 论

6.1 CoDefense 对嵌入质量的影响

现有研究[80,81]已明确指出, 嵌入质量 (embedding quality) 对于代码模型学习深层代码语义特征以划定有效决策边界具有至关重要的作用. 为了深入对比原始模型与 CoDefense 增强后模型的嵌入质量差异, 我们采用了 t-SNE (t-distributed stochastic neighbor embedding) 技术[82], 该技术通过将目标代码模型生成的高维嵌入向量映射至二维空间, 以直观地揭示代码嵌入之间的相对关系. 鉴于先前研究[81]中, t-SNE 以离散程度作为关键衡量标准在评估分类任务嵌入质量方面展现出了卓越性能, 因此我们同样选择了两项分类任务 (即漏洞检测和代码克隆检测任务) 进行实验评估, 并覆盖了所有 3 个预训练代码模型. 与现有工作保持一致[80,81], 我们采用质心距离 (centroid distance) 作为量化嵌入离散程度及嵌入质量的度量标准. 质心距离数值越大, 则反映了嵌入空间内各类别界限更为清晰和明确, 进而表明嵌入质量的提升.

图 3 直观地展示了在 6 个实验场景下 (涉及 3 个代码模型和两个分类任务组合) 测试集样本的 t-SNE 可视化结果. 图中, “Original”标签指代原始模型, 而“w/CoDefense”则对应 CoDefense 增强后的模型, “VD”与“CCD”分别指代漏洞检测任务和代码克隆检测任务, “类别-0”和“类别-1”分别表示这两个分类任务中不同的类别. 在这 6 个实验场景的评估中, 针对质心距离的测量结果显示, CoDefense 增强后的模型在其中 4 个实验场景中都实现了更优异的分离效果. 进一步分析详细数据, 我们发现相较于原始模型, CoDefense 不仅未破坏原始模型的嵌入质量和模型性能, 反而平均提升了 1.96% 的质心距离 (反映嵌入质量增强) 和 0.23% 的 Accuracy (表明模型性能亦有小幅提升), 充分支持了研究问题 3 的实验结论.

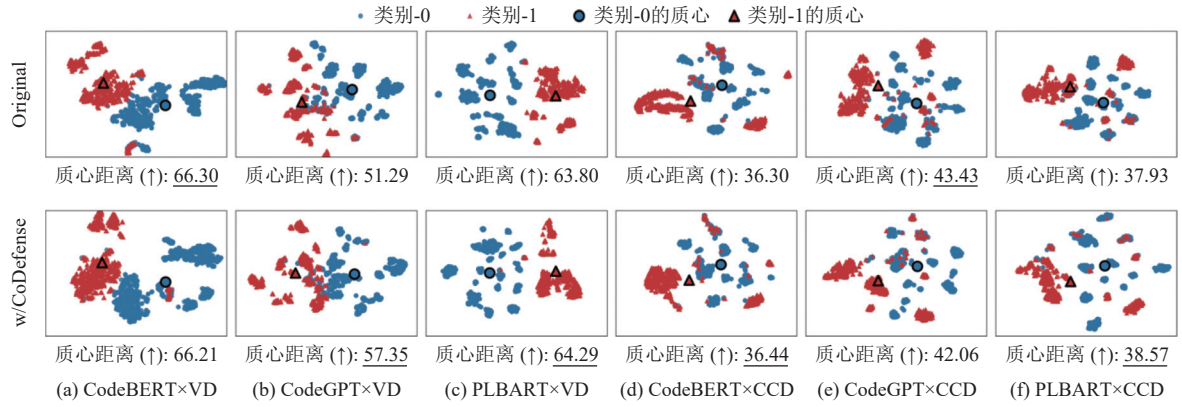


图 3 原始模型和 CoDefense 增强后模型的嵌入质量对比

6.2 CoDefense 在大模型上的泛化性

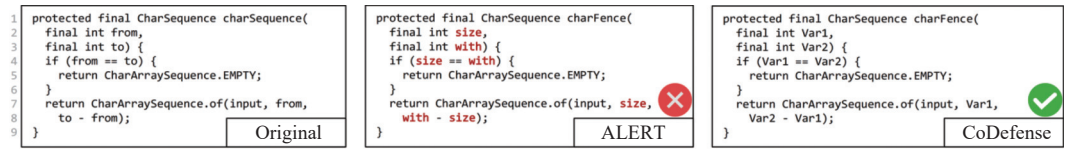
关于大规模模型的对抗性攻击有效性问题, 我们深入分析了相关研究成果[83–85], 当前的研究表明, 即便是最先进的大语言模型在应对对抗性攻击时仍存在一定的脆弱性. 例如, Zhang 等人[84]展示了在较小规模代码模型上生成的对抗性样本在大模型 (如 GPT-3.5、GPT-4、Claude-2 等) 上仍然有效. 此外, Li 等人[83]的研究验证了代码变异策略 (如代码重命名和等价结构转换) 在大型代码模型 (如 StarCoder 和 CodeGen2) 上的有效性, 而 Yang 等人[85]发现, 通过插入函数名或关键字的对抗性扰动可以诱导诸如 Code Gemma、Code Llama 和 CodeGeeX2 等模型生成恶意代码片段. 这些研究表明, 本文所针对的变量名替换、死代码插入和结构等价转换等对抗性防御方法在当前研究中仍具备实际价值.

为进一步评估 CoDefense 在更大规模模型上的泛化能力, 我们增加了 DeepSeek-Coder-1.3B[86]作为新的研究

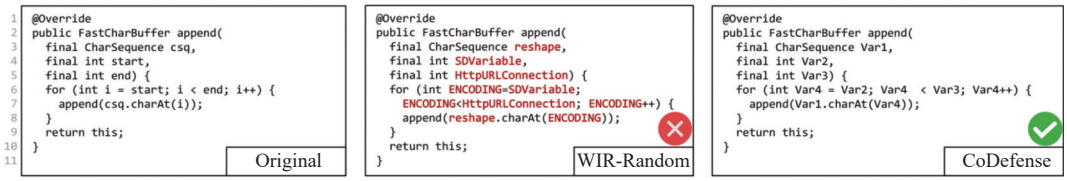
对象, 以探索其在更大规模模型上的表现. 由于大语言模型的训练开销及对抗性攻击实验耗时较高 (约需 3 851.28 h), 我们选择了 Vulnerability detection 任务作为代表性实验, 耗时仍然高达 187.36 h, 详细的实验数据可查阅本文的开源主页. 未来, 我们计划扩展至更多大语言模型和数据集. 基于对 DeepSeek-Coder 在 3 个研究问题的分析, 得出以下发现: (1) 对比其他方法, CoDefense 在对抗性防御方面更具优势, 平均提升 14.22% 的 *DSR*, 并减少了 85.46% 的 *FDR*. 特别是在变量名替换的对抗性攻击 (WIR-Random 和 ALERT) 中, CoDefense 达到了 100.00% 的 *DSR* 和 0.00% 的 *FDR*, 实现了完美防御. (2) CoDefense 在时间效率方面表现显著, 防御对抗性攻击时平均提升了 59.32% 的效率. (3) CoDefense 在保持模型原始性能方面, 相较于原始模型提升了 0.71% 的性能, 相较于 3 种对比方法平均提升了 1.26%. 这些结果均与现有结论保持一致, 进一步验证了 CoDefense 在更大规模模型上具有较强的泛化性.

6.3 CoDefense 的防御通用性

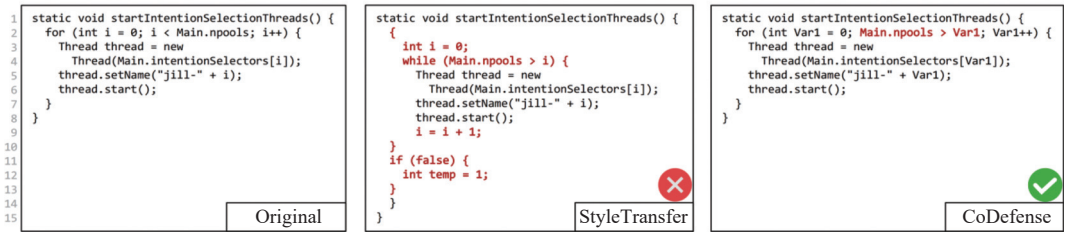
图 4 展示了在 PLBART 模型和 Code summarization 任务中, 不同对抗性攻击方法生成的对抗性样本, 以及 CoDefense 生成的相应的归一化代码. 具体而言, 图 4(a) 显示了 ALERT 通过变量名替换生成的对抗性样本, CoDefense 针对此样本实施了相应的变量名归一化防御; 图 4(b) 展示了 WIR-Random 利用变量名替换生成的对抗性样本, CoDefense 亦通过变量名归一化有效抵御了该攻击; 图 4(c) 则展示了 StyleTransfer 利用死代码插入和代码结构转换生成的对抗性样本, CoDefense 对该样本进行了相应的死代码消除和代码结构归一化的防御. 基于这些示例, 我们总结出了两个关键发现.



(a) ALERT 在 PLBART 模型和 Code summarization 任务上的示例



(b) WIR-Random 在 PLBART 模型和 Code summarization 任务上的示例



(c) StyleTransfer 在 PLBART 模型和 Code summarization 任务上的示例

图 4 3 个对抗性攻击与防御的示例

(1) 在应对 ALERT 和 WIR-Random 等基于变量名替换的对抗性攻击方法时, CoDefense 展现出高度的通用性与显著的防御效果, 支持了我们在研究问题一中的结论. 具体而言, CoDefense 在所有 18 个实验场景中均达到了 100.00% 的 *DSR* (防御成功率) 和 0.00% 的 *FDR* (错误防御率), 而对比方法的平均 *DSR* 仅为 87.85%, *FDR* 为 1.28%. 这一显著对比进一步验证了 CoDefense 在抵御多种变量名替换攻击策略时的全面有效性.

(2) 针对 StyleTransfer 这类基于死代码插入和代码结构转换的对抗性攻击, CoDefense 成功消除了相应的死代

码并破坏了大部分代码结构转换攻击, 确保了模型预测未被恶意改变. 然而, 我们观察到 CoDefense 尚不能完全防御所有代码结构转换攻击 (如图 4(c) 中的“Main.npools > Var1”), 其原因在于当前 CoDefense 的代码结构归一化策略主要依赖预设的代码转换规则及两类常用代码质量指标 (即代码复杂度与代码可读性). 随着对抗性攻击技术的演进, 攻击者可能利用这些指标的局限性构造出隐蔽性更强的对抗性样本. 特别是当生成的对抗性样本符合上述代码质量指标时, 它们可能成功绕过 CoDefense 的防御机制.

在深入探讨对抗性防御策略面对未知攻击的通用性时, 我们不得不正视一个核心挑战: 即便是当前最先进的对抗性训练技术, 仍然难以全面防御所有已知乃至未知的对抗性攻击策略. 对抗性攻击策略的快速更新和预训练代码模型的不可解释性加剧了这个问题. 因此, 构建一个能够一劳永逸地防御所有粒度和策略的对抗性攻击的完美防御技术, 在现有的技术框架下, 显得尤为不切实际. 这种普遍存在的局限性, 凸显了对抗性防御领域持续探索与创新的紧迫性.

尽管如此, 我们提出的 CoDefense 策略仍然展现出了显著的优势, 特别是在防御基于变量名替换的对抗性攻击方面. 通过创新性的变量名归一化方法, CoDefense 完全避免了含有潜在对抗性变量名的对抗性样本输入到代码模型, 从根本上消除了此类攻击, 实现了对所有基于变量名替换的对抗性攻击的全面防御. 研究问题一的实验结果也充分表明了变量名归一化策略面对基于变量名替换攻击的完美防御能力.

对于代码模型面临的另外两种粒度的对抗性攻击——死代码插入攻击与代码结构转换攻击, 本文系统地总结了最近 5 年来, 最具有代表性或最先进的针对预训练代码模型的对抗性攻击方法. 在此基础之上, 精心设计并实现了对应的防御规则, 该规则集是目前我们所知的最详尽的防御规则集, 包括 19 条针对死代码插入攻击的防御规则以及 51 条针对代码结构转换攻击的防御规则. 这些对抗性防御规则集显著提升了 CoDefense 在面对这两种粒度复杂攻击时的防御能力, 相较于现有的对抗性训练策略, 展现出了更为卓越的性能表现. 经过人工检查, 可以发现 CoDefense 可以防御所有已知的死代码插入攻击和大部分已知的代码结构转换攻击. 当面临未知攻击时, 只需要更新防御规则集并重新执行 CoDefense 即可提升模型的防御能力, 考虑到 CoDefense 更新的成本要远远低于最先进的对抗性重训练, 展现了 CoDefense 显著的长期效率优势. 此外, 对于这两种粒度的对抗性攻击, 在未来工作 (第 6.4 节) 中提供了两个极具潜力的研究方向, 旨在未来研究中进一步优化和完善应对这两类未知攻击的防御策略.

6.4 未来工作

CoDefense 作为一种新颖的对抗性防御策略, 显著增强了预训练代码模型的鲁棒性, 并有效抵御了多粒度的对抗性攻击, 同时保持模型的原始性能和嵌入质量, 这一成果为代码安全领域带来了重要突破. 尽管当前的实证研究已充分表明了其有效性和效率, 但 CoDefense 仍有进一步拓展的潜力, 未来工作将围绕以下 4 个方面展开, 以进一步提升其性能与适用范围.

第一, 优化死代码消除策略. 针对日益复杂的死代码插入攻击, CoDefense 已初步总结并实现了一套相应的死代码消除规则集, 有效地应对了近年来最为常见的死代码插入攻击. 然而, 如现有研究[30,53]所指出的, 死代码检测本质上是一个不可判定的任务, 这使得当前的 CoDefense 在消除复杂死代码时仍面临挑战. 为克服此限制, 我们拟引入编译优化技术[87,88], 通过深度整合这些技术, 优化并增强 CoDefense 在识别与消除复杂死代码方面的能力, 从而构建更加全面的防御策略.

第二, 增强代码结构归一化策略. 当前 CoDefense 采用的代码结构归一化策略主要依赖于设计的代码转换规则以及两类常用的代码质量指标 (即代码复杂度和代码可读性). 然而, 随着对抗性攻击技术的不断演进, 攻击者可能会利用这些指标的局限性来构造难以察觉的对抗性样本. 特别是当生成的对抗性样本与这两类代码质量指标保持一致时, 它们可能会成功绕过 CoDefense 的防御机制. 为应对此挑战, 我们计划设计并实施一套更为多样化、精细化的代码质量评估体系, 涵盖但不限于代码语义完整性、逻辑一致性及异常处理效率等维度, 以全面提升 CoDefense 对潜在威胁的识别与抵御能力.

第三, 拓展 CoDefense 的应用范围. CoDefense 不仅在防御对抗性攻击方面展现出卓越效能, 其潜在的应用价

值亦不容忽视. 实验结果显示, 该策略在提升代码模型性能和代码嵌入质量方面亦具备一定潜力. 因此, 我们计划将其延伸至代码数据质量优化领域, 研究如何将 CoDefense 作为一种高效的数据预处理工具, 通过优化输入代码数据质量, 以期在增强模型鲁棒性的同时, 促进模型性能和嵌入质量的进一步提升. 这一研究方向有望为代码模型性能优化开辟新的途径.

第四, 其他编程语言验证与扩展. 受限于计算资源和时间成本, 当前研究主要聚焦于 Java 语言环境下的 CoDefense 性能评估. 为全面评估其泛化能力, 我们计划在未来工作中将 CoDefense 工具扩展至更多编程语言, 包括但不限于 C/C++、Python 等, 并在这些编程语言上进行广泛的实验验证. 通过比较与分析, 我们将进一步揭示 CoDefense 在不同编程环境下的适用性, 为其在实际应用中的广泛部署提供坚实支撑.

6.5 有效性威胁

内部有效性威胁. 内部有效性威胁主要来自 CoDefense 方法的实现和对比方法的实现. 为了有效减少此类威胁, 对于 CoDefense 方法的实现, 本文依赖于 Python、Tree-Sitter 和 srcML 等成熟框架进行实现确保实现过程的稳定性和可靠性; 对于对比方法的实现, 本文采用了现有研究的开源代码, 保证对比方法的实现准确无误. 此外, 所有实验结果的分析脚本均经过作者的多次校对, 以确保结果的准确性, 我们将所有数据、模型和代码开源给社区, 以供同行进一步检查和验证.

● 外部有效性威胁. 外部有效性威胁主要来源于研究的实验对象, 包括实验任务、数据集以及代码模型. 为了有效减少此类威胁, 对于任务, 所选任务涉及分类和生成任务, 已经被广泛应用于现有的对抗性攻击相关工作[15,18], 具有较高的代表性; 对于数据集, 不仅使用了在许多对抗性攻击相关研究中常用的 CodeXGLUE 基准数据集, 还包括了新的 OWASP 基准数据集, 以评估方法的普遍适用性; 对于代码模型, 与最新研究保持一致, 选择了最流行且性能较高的不同架构的代码模型, 确保实验结果的普适性. 在未来的工作中, 我们将在更多的实验对象上进行验证, 以进一步表明 CoDefense 的有效性和可扩展性. 此外, 现有研究表明[7,10,21,68], 在代码摘要生成任务中, BLEU-4 值是衡量模型性能的主要指标, 其取值范围为 0–1, 完美匹配的得分为 1, 完全不匹配的得分为 0. 在对抗性攻击效果的判定方面, 现有方法主要采用两种判定标准: (1) BLEU-4 值发生任意下降即判定为攻击成功[89,90]; (2) BLEU-4 值降为 0 时即判定为攻击成功[16,21]. 本文采用了第 2 种判定方法, 因为相比 BLEU-4 值任意下降的判定, BLEU-4 值降为 0 作为攻击成功的标准更为严格且差异更大. 相应地, 我们也在评估对抗性防御技术时采用了类似标准. 尽管当前的这种常用的攻击和防御判定指标可能存在一定的局限性, 为了保证评估结果具有良好的可比性, 仍然采用了这种最常用的判定方式. 在未来, 我们会进一步研究更高差异性、更严格的判定方式.

● 结构有效性威胁. 结构有效性的威胁主要来自实验过程中方法运行时指定的参数设置, 尤其是训练时的超参数设置 (如批量大小、学习率、优化器等). 为了有效减少此类威胁, 我们参照相关研究[16]中公开的预训练代码模型初始训练时的超参数设置, 并在后续 CoDefense 和对抗性训练的训练过程中与之保持一致, 以确保实验结果的公平性和可靠性.

7 相关工作

为了提升代码模型的鲁棒性, 目前主要有两类方法: 设计更先进的神经网络以增强模型, 以及结合更多代码数据来微调或训练模型. 在第 1 类方法中, Bielik 等人[42]引入了 (m+1)-class 策略到代码模型中, 通过重新训练模型使其能够在面对不确定输入时放弃预测. 另外, Bui 等人[91]将基于树的卷积神经网络与胶囊网络相结合, 以更好地学习代码的抽象语法树表示, 从而提高代码模型的泛化能力. 最近的一些工作[40,92–94]则采用对比学习来增强代码模型的鲁棒性. 对于第 2 类方法, 一些研究提出了构建对抗性代码对代码模型进行对抗性训练, 以提高代码模型的鲁棒性[15,19]. 类似地, Yu 等人[52]、Allamanis 等人[95]、Li 等人[96]和 Li 等人[83]提出了利用等价的代码转换规则来进行数据增强, 从而提高代码模型的鲁棒性. Mi 等人[97]则将代码片段视为图像, 通过辅助生成对抗性网络生成增强数据集. 最近, Wang 等人[50]结合课程学习来增强代码数据, 以优化代码模型的微调过程, 减少在目标数据集上的过拟合, 从而提高模型鲁棒性.

最近, 大语言模型 (如 ChatGPT) 展现了强大的推理能力, 但仍然容易受到基于提示的攻击 (如越狱攻击、后门攻击和对抗性提示攻击)[98,99]. 针对这一问题, 有研究提出了基于输入或输出过滤的大语言模型防御方法, 但这些方法主要用于保护大语言模型免受越狱攻击[100–102]. 此外, Sheshadri 等人[103]提出了一种基于目标潜在对抗性训练 (target latent adversarial training) 的方法, 通过帮助大语言模型忘记不良的知识记忆, 来有效应对后门攻击. 近期, Zhang 等人[84]提出了基于提示的自我防御策略, 利用大语言模型的上下文学习和人类指令理解的能力, 自动修改提示中的对抗性扰动, 并添加有效信息以增强大语言模型应对对抗性提示的能力. 类似地, Lin 等人[104]提出了基于大语言模型代理 (LLM agent-based) 的防御方法, 在将对抗性提示输入到大语言模型之前对其进行字符级修改, 并提供修改之后干净的提示, 从而提升模型的鲁棒性. 然而, 这些方法主要针对基于提示的大语言模型 (prompting-based LLM), 在本文的研究场景中无法直接应用, 因此我们考虑在未来工作中进一步研究这些技术.

我们的研究表明, 与广泛使用的对抗性训练策略相比, CoDefense 在防御对抗性攻击方面表现出更高的有效性和效率. 此外, 基于特定对抗性攻击方法的对抗性训练策略大多无法防御其他的对抗性攻击, 但 CoDefense 可以防御不同粒度和策略的对抗性攻击, 这是因为 CoDefense 与对抗性攻击的策略无关. 更为关键的是, 相较于其他的对抗性防御方法, CoDefense 不会损害模型的原始性能和嵌入分布. 此外, 在计算机视觉领域, 还存在一些技术可以通过处理输入来提高模型性能[87,105,106]. 最近, Tian 等人[31]首次尝试通过即时输入去噪来提升已部署代码模型的性能, 然而他们的方法尚未在对抗性攻击场景下验证对模型鲁棒性的影响.

8 总 结

本研究致力于增强预训练代码模型在面对对抗性攻击时的防御能力. 为此, 提出了一种基于代码归一化的预训练代码模型对抗性防御方法 (即 CoDefense). 该方法通过多粒度代码归一化将训练过程的原始训练集和推理过程的代码输入进行归一化预处理, 以避免对抗性样本直接输入到代码模型, 并具备防御不同粒度和策略的对抗性攻击的能力. 我们对 CoDefense 进行了全面地实证研究, 涵盖了 27 个不同的实验场景. 实验结果表明, 与当前最先进的对抗性训练方法相比, CoDefense 能够更有效地防御对抗性攻击. 具体而言, CoDefense 能够成功防御高达 95.33% 的对抗性攻击, 相较于最先进的对抗性训练策略平均提升 10.75%. 更为关键的是, 不同于对抗性训练, CoDefense 对原始模型的性能和嵌入质量影响更小. 此外, CoDefense 在运行效率上也大幅超越了最先进的对抗性训练, 其时间效率相较于对比方法平均提升了 85.86%. 综上所述, CoDefense 不仅显著提升了预训练代码模型在对抗性攻击面前的防御能力, 而且在保证模型性能和嵌入质量的同时大幅提升了时间效率, 为维护代码模型的安全性和实用性提供了有力的保障.

References

[1] Feng ZY, Guo DY, Tang DY, Duan N, Feng XC, Gong M, Shou LJ, Qin B, Liu T, Jiang DX, Zhou M. CodeBERT: A pre-trained model for programming and natural languages. In: Proc. of the 2020 Findings of the Association for Computational Linguistics. ACL, 2020. 1536–1547. [doi: 10.18653/v1/2020.findings-emnlp.139]

[2] Guo DY, Ren S, Lu S, Feng ZY, Tang DY, Liu SJ, Zhou L, Duan N, Svyatkovskiy A, Fu SY, Tufano M, Deng SK, Clement C, Drain D, Sundaresan N, Yin J, Jiang DX, Zhou M. GraphCodeBERT: Pre-training code representations with data flow. arXiv:2009.08366, 2020.

[3] Wang Y, Wang WS, Joty S, Hoi SCH. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing. ACL, 2021. 8696–8708. [doi: 10.18653/v1/2021.emnlp-main.685]

[4] Wang Z, Yan M, Liu S, Chen JJ, Zhang DD, Wu Z, Chen X. Survey on testing of deep neural networks. Ruan Jian Xue Bao/Journal of Software, 2020, 31(5): 1255–1275 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5951.htm [doi: 10.13328/j.cnki.jos.005951]

[5] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010.

[6] Ahmad W, Chakraborty S, Ray B, Chang KW. Unified pre-training for program understanding and generation. In: Proc. of the 2021 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. ACL, 2021.

2655–2668. [doi: 10.18653/v1/2021.naacl-main.211]

[7] Lu S, Guo DY, Ren S, Huang JJ, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang DX, Tang DY, Li G, Zhou LD, Shou LJ, Zhou L, Tufano M, Gong M, Zhou M, Duan N, Sundaresan N, Deng SK, Fu SY, Liu SJ. CodexGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664, 2021.

[8] Zhou YQ, Liu SQ, Siow J, Du XN, Liu Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Red Hook: Curran Associates Inc. , 2019. 10197–10207.

[9] White M, Tufano M, Vendome C, Poshyvanyk D. Deep learning code fragments for code clone detection. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: IEEE, 2016. 87–98. [doi:]

[10] Ahmad W, Chakraborty S, Ray B, Chang KW. A Transformer-based approach for source code summarization. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. ACL, 2020. 4998–5007. [doi: 10.18653/v1/2020.acl-main.449]

[11] Lu ZP, Hu HC, Huo SM, Li SY. Ensemble learning methods of adversarial attacks and defenses in computer vision: Recent progress. In: Proc. of the 2021 Int'l Conf. on Advanced Computing and Endogenous Security. Nanjing: IEEE, 2022. 1–10. [doi: 10.1109/IEEECONF52377.2022.10013347]

[12] Wang ZB, Guo HC, Zhang ZF, Liu WX, Qin Z, Ren K. Feature importance-aware transferable adversarial attacks. In: Proc. of the 2021 IEEE/CVF Int'l Conf. on Computer Vision. Montreal: IEEE, 2021. 7619–7628. [doi: 10.1109/ICCV48922.2021.00754]

[13] Zhang WE, Sheng QZ, Alhazmi A, Li CL. Adversarial attacks on deep-learning models in natural language processing: A survey. ACM Trans. on Intelligent Systems and Technology, 2020, 11(3): 24. [doi: 10.1145/3374217]

[14] Morris J, Lifland E, Yoo JY, Grigsby J, Jin D, Qi YJ. TextAttack: A framework for adversarial attacks, data augmentation, and adversarial training in NLP. In: Proc. of the 2020 Conf. on Empirical Methods in Natural Language Processing: System Demonstrations. ACL, 2020. 119–126. [doi: 10.18653/v1/2020.emnlp-demos.16]

[15] Tian Z, Chen JJ, Jin Z. Code difference guided adversarial example generation for deep code models. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering. Luxembourg: IEEE, 2023. 850–862. [doi: 10.1109/ASE56229.2023.00149]

[16] Du XH, Wen M, Wei ZC, Wang SW, Jin H. An extensive study on adversarial attack against pre-trained models of code. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 489–501. [doi: 10.1145/3611643.3616356]

[17] Cao XJ, Chen JJ, Yan M, You HM, Wu Z, Wang Z. Test case selection for neural network via data mutation. Ruan Jian Xue Bao/Journal of Software, 2024, 35(11): 4973–4992 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7005.htm [doi: 10.13328/j.cnki.jos.007005]

[18] Zhang HZ, Li Z, Li G, Ma L, Liu Y, Jin Z. Generating adversarial examples for holding robustness of source code processing models. In: Proc. of the 2020 AAAI Conf. on Artificial Intelligence. New York: AAAI Press, 2020. 1169–1176. [doi: 10.1609/aaai.v34i01.5469]

[19] Yang Z, Shi JK, He JD, Lo D. Natural attack for pre-trained models of code. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 1482–1493. [doi: 10.1145/3510003.3510146]

[20] Zhang HZ, Fu ZY, Li G, Ma L, Zhao ZH, Yang HA, Sun YZ, Liu Y, Jin Z. Towards robustness of deep program processing models—Detection, estimation, and enhancement. ACM Trans. on Software Engineering and Methodology, 2022, 31(3): 50. [doi: 10.1145/3511887]

[21] Zeng ZR, Tan HZ, Zhang HT, Li J, Zhang YQ, Zhang LM. An extensive study on pre-trained models for program understanding and generation. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. ACM, 2022. 39–51. [doi: 10.1145/3533767.3534390]

[22] Karmakar A, Robbes R. What do pre-trained code models know about code? In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2021. 1332–1336. [doi: 10.1109/ASE51524.2021.9678927]

[23] Shi ES, Wang YL, Zhang HY, Du L, Han S, Zhang DM, Sun HB. Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 39–51. [doi: 10.1145/3597926.3598036]

[24] Zhu QH, Liang QY, Sun ZY, Xiong YF, Zhang L, Cheng SY. GrammarT5: Grammar-integrated pretrained encoder-decoder neural model for code. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 1–13. [doi: 10.1145/3597503.3639125]

[25] Nijkamp E, Pang B, Hayashi H, Tu LF, Wang H, Zhou YB, Savarese S, Xiong CM. CodeGen: An open large language model for code with multi-turn program synthesis. arXiv:2203.13474, 2022.

[26] Lewis M, Liu YH, Goyal N, Ghazvininejad M, Mohamed A, Levy O, Stoyanov V, Zettlemoyer L. BART: Denoising sequence-to-

sequence pre-training for natural language generation, translation, and comprehension. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. ACL, 2020. 7871–7880. [doi: 10.18653/v1/2020.acl-main.703]

[27] Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou YQ, Li W, Liu PJ. Exploring the limits of transfer learning with a unified text-to-text Transformer. The Journal of Machine Learning Research, 2020, 21(1): 140.

[28] Wang SQ, Li Z, Qian HF, Yang CH, Wang ZJ, Shang MY, Kumar V, Tan S, Ray B, Bhatia P, Nallapati R, Ramanathan MK, Roth D, Xiang B. Recode: Robustness evaluation of code generation models. In: Proc. of the 61st Annual Meeting of the Association for Computational Linguistics. Toronto: ACL, 2022. 13818–13843. [doi: 10.18653/v1/2023.acl-long.773]

[29] Karampatsis RM, Babii H, Robbes R, Sutton C, Janes A. Big code != big vocabulary: Open-vocabulary models for source code. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 1073–1085. [doi: 10.1145/3377811.3380342]

[30] Yefet N, Alon U, Yahav E. Adversarial examples for models of code. Proc. of the ACM on Programming Languages, 2020, 4(OOPSLA): 162. [doi: 10.1145/3428230]

[31] Tian Z, Chen JJ, Zhang XY. On-the-fly improving performance of deep code models via input denoising. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering. Luxembourg: IEEE, 2023. 560–572. [doi: 10.1109/ASE56229.2023.00166]

[32] Brown WH, Malveau RC, McCormick HWS, Mowbray TJ. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. New York: John Wiley & Sons, Inc., 1998.

[33] Mantyla M, Vanhanen J, Lassenius C. A taxonomy and an initial empirical study of bad smells in code. In: Proc. of the 2003 Int'l Conf. on Software Maintenance. Amsterdam: IEEE, 2003. 381–384. [doi: 10.1109/ICSM.2003.1235447]

[34] Wake WC. Refactoring Workbook. Boston: Addison-Wesley Longman, 2003.

[35] Martin RC. Clean code: A Handbook of Agile Software Craftsmanship. Upper Saddle River: Pearson, 2008.

[36] Romano S, Vendome C, Scanniello G, Poshyvanyk D. A multi-study investigation into dead code. IEEE Trans. on Software Engineering, 2020, 46(1): 71–99. [doi: 10.1109/TSE.2018.2842781]

[37] Nguyen TD, Zhou Y, Le XBD, Thongtanunam P, Lo D. Adversarial attacks on code models with discriminative graph patterns. arXiv:2308.11161, 2023.

[38] Bui NDQ, Yu YJ, Jiang LX. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In: Proc. of the 44th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval. ACM, 2021. 511–521. [doi: 10.1145/3404835.3462840]

[39] Srikant S, Liu SJ, Mitrovska T, Chang SY, Fan QF, Zhang GY, O'Reilly UM. Generating adversarial computer programs using optimized obfuscations. arXiv:2103.11882, 2021.

[40] Jia JH, Srikant S, Mitrovska T, Gan C, Chang SY, Liu SJ. ClawSAT: Towards both robust and accurate code models. In: Proc. of the 2023 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Macao: IEEE, 2023. 212–223. [doi: 10.1109/SANER56733.2023.00029]

[41] Li J, Li Z, Zhang HZ, Li G, Jin Z, Hu X, Xia X. Poison attack and defense on deep source code processing models. arXiv:2210.17029, 2022.

[42] Bielik P, Vechev M. Adversarial robustness for code. In: Proc. of the 37th Int'l Conf. on Machine Learning. JMLR.org, 2020. 896–907. [doi: 10.48550/arXiv.2002.04694]

[43] Zhuo TY, Yang Z, Sun ZS, Wang YF, Li L, Du XN, Xing ZC, Lo D. Data augmentation approaches for source code models: A survey. arXiv:2305.19915, 2023.

[44] Mercuri V, Saletta M, Ferretti C. Evolutionary approaches for adversarial attacks on neural source code classifiers. Algorithms, 2023, 16(10): 478. [doi: 10.3390/a16100478]

[45] Henkel J, Ramakrishnan G, Wang Z, Albarghouthi A, Jha S, Reps T. Semantic robustness of models of source code. In: Proc. of the 2022 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Honolulu: IEEE, 2022. 526–537. [doi: 10.1109/SANER53432.2022.00070]

[46] Gopstein D, Iannacone J, Yan Y, DeLong L, Zhuang YY, Yeh MKC, Cappos J. Understanding misunderstandings in source code. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. Paderborn: ACM, 2017. 129–139. [doi: 10.1145/3106237.3106264]

[47] Li Z, Chen GQ, Chen C, Zou YY, Xu SH. RoPGen: Towards robust code authorship attribution via automatic coding style transformation. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering. Pittsburgh: IEEE, 2022. 1906–1918. [doi: 10.1145/3510003.3510181]

[48] Chakraborty S, Ahmed T, Ding Y, Devanbu PT, Ray B. Natgen: Generative pre-training by "naturalizing" source code. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 18–30. [doi: 10.1145/3540250.3549162]

[49] Li Z, Zhang RQ, Zou DQ, Wang N, Li YT, Xu SH. Robin: A novel method to produce robust interpreters for deep learning-based code classifiers. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering. Luxembourg: IEEE, 2023. 27–39. [doi: 10.1109/ASE56229.2023.00164]

[50] Wang DZ, Jia ZY, Li SS, Yu Y, Xiong Y, Dong W, Liao XK. Bridging pre-trained models and downstream tasks for source code understanding. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 287–298. [doi: 10.1145/3510003.3510062]

[51] Zhang WW, Guo SJ, Zhang HY, Sui YL, Xue YX, Xu Y. Challenging machine learning-based clone detectors via semantic-preserving code transformations. IEEE Trans. on Software Engineering, 2023, 49(5): 3052–3070. [doi: 10.1109/TSE.2023.3240118]

[52] Yu SW, Wang T, Wang J. Data augmentation by program transformation. Journal of Systems and Software, 2022, 190: 111304. [doi: 10.1016/j.jss.2022.111304]

[53] Gao FJ, Wang Y, Wang K. Discrete adversarial attack to models of code. Proc. of the ACM on Programming Languages, 2023, 7(PLDI): 172–195. [doi: 10.1145/3591227]

[54] Nakamura K, Ishiura N. Random testing of C compilers based on test program generation by equivalence transformation. In: Proc. of the 2016 IEEE Asia Pacific Conf. on Circuits and Systems. Jeju: IEEE, 2016. 676–679. [doi: 10.1109/APCCAS.2016.7804063]

[55] Quiring E, Maier A, Rieck K. Misleading authorship attribution of source code using adversarial learning. In: Proc. of the 28th USENIX Conf. on Security Symp. Santa Clara: USENIX Association, 2019. 479–496.

[56] Cheers H, Lin YQ, Smith SP. Spplagiarise: A tool for generating simulated semantics-preserving plagiarism of Java source code. In: Proc. of the 10th IEEE Int'l Conf. on Software Engineering and Service Science. Beijing: IEEE, 2019. 617–622. [doi: 10.1109/ICSESS47205.2019.9040853]

[57] Ding YRB, Chakraborty S, Buratti L, Pujar S, Morari A, Kaiser G, Ray B. CONCORD: Clone-aware contrastive learning for source code. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 26–38. [doi: 10.1145/3597926.3598035]

[58] Sellitto G, Iannone E, Codabux Z, Lenarduzzi V, De Lucia A, Palomba F. Toward understanding the impact of refactoring on program comprehension. In: Proc. of the 2022 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Honolulu: IEEE, 2022. 731–742. [doi: 10.1109/SANER53432.2022.00090]

[59] Buse RPL, Weimer WR. Learning a metric for code readability. IEEE Trans. on Software Engineering, 2010, 36(4): 546–558. [doi: 10.1109/TSE.2009.70]

[60] Pantiuchina J, Lanza M, Bavota G. Improving code: The (Mis) perception of quality metrics. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution. Madrid: IEEE, 2018. 80–91. [doi: 10.1109/ICSME.2018.00017]

[61] Chidamber SR, Kemerer CF. A metrics suite for object oriented design. IEEE Trans. on Software Engineering, 1994, 20(6): 476–493. [doi: 10.1109/32.295895]

[62] McCabe TJ. A complexity measure. IEEE Trans. on Software Engineering, 1976, SE-2(4): 308–320. [doi: 10.1109/TSE.1976.233837]

[63] Scalabrino S, Linares-Vásquez M, Poshyvanyk D, Oliveto R. Improving code readability models with textual features. In: Proc. of the 24th IEEE Int'l Conf. on Program Comprehension. Austin: IEEE, 2016. 1–10. [doi: 10.1109/ICPC.2016.7503707]

[64] Hough K, Welearegai G, Hammer C, Bell J. Revealing injection vulnerabilities by leveraging existing tests. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 284–296. [doi: 10.1145/3377811.3380326]

[65] Sayar I, Bartel A, Bodden E, Le Traon Y. An in-depth study of Java deserialization remote-code execution exploits and vulnerabilities. ACM Trans. on Software Engineering and Methodology, 2023, 32(1): 25. [doi: 10.1145/3554732]

[66] Spoto F, Burato E, Ernst MD, Ferrara P, Lovato A, Macedonio D, Spiridon C. Static identification of injection attacks in Java. ACM Trans. on Programming Languages and Systems, 2019, 41(3): 18. [doi: 10.1145/3332371]

[67] Li R, Allal LB, Zi YT, *et al*. StarCoder: May the source be with you! arXiv:2305.06161, 2023.

[68] Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M. Codesearchnet challenge: Evaluating the state of semantic code search. arXiv:1909.09436, 2019.

[69] Wang Y, Le H, Gotmare A, Bui N, Li JN, Hoi S. CodeT5+: Open code large language models for code understanding and generation. In: Proc. of the 2023 Conf. on Empirical Methods in Natural Language Processing. Singapore: ACL, 2023. 1069–1088. [doi: 10.18653/v1/2023.emnlp-main.68]

[70] Rozière B, Gehring J, Gloeckle F, *et al*. Code Llama: Open foundation models for code. arXiv:2308.12950, 2023.

[71] Nguyen PT, di Sipio C, di Rocco J, di Penta M, di Ruscio D. Adversarial attacks to API recommender systems: Time to wake up and smell the coffee? In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2021. 253–265. [doi: 10.1109/ASE51524.2021.9678946]

[72] Rabin MRI, Bui NDQ, Wang K, Yu YJ, Jiang LX, Alipour MA. On the generalizability of neural program models with respect to semantic-preserving program transformations. Information and Software Technology, 2021, 135: 106552. [doi: 10.1016/j.infsof.2021.106552]

[73] Pour MV, Li Z, Ma L, Hemmati H. A search-based testing framework for deep neural networks of source code embedding. In: Proc. of the 14th IEEE Conf. on Software Testing, Verification and Validation. Porto de Galinhas: IEEE, 2021. 36–46. [doi: 10.1109/ICST49551.2021.00016]

[74] Zhou Y, Zhang XQ, Shen JJ, Han TT, Chen TL, Gall H. Adversarial robustness of deep code comment generation. ACM Trans. on Software Engineering and Methodology, 2022, 31(4): 60. [doi: 10.1145/3501256]

[75] Dong ZM, Hu Q, Guo YJ, Cordy M, Papadakis M, Zhang ZY. MixCode: Enhancing code classification by mixup-based data augmentation. In: Proc. of the 2023 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Macao: IEEE, 2023. 379–390. [doi: 10.1109/SANER56733.2023.00043]

[76] Tree-sitter. 2024. https://tree-sitter.github.io/tree-sitter/

[77] srcML. 2024. https://www.srcml.org/

[78] Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, Cistac P, Rault T, Louf R, Funtowicz M, Davison J, Shleifer S, von Platen P, Ma C, Jernite Y, Plu J, Xu CW, Le Scao T, Gugger S, Drame M, Lhoest Q, Rush AM. Huggingface's Transformers: State-of-the-art natural language processing. arXiv:1910.03771, 2019.

[79] Rosner B, Glynn RJ, Lee MLT. The Wilcoxon signed rank test for paired comparisons of clustered data. Biometrics, 2006, 62(1): 185–192. [doi: 10.1111/j.1541-0420.2005.00389.x]

[80] Burkart N, Huber MF. A survey on the explainability of supervised machine learning. Journal of Artificial Intelligence Research, 2021, 70: 245–317. [doi: 10.1613/jair.1.12228]

[81] Tian Z, Shu HL, Wang D, Cao XJ, Kamei Y, Chen JJ. Large language models for equivalent mutant detection: How far are we? In: Proc. of the 33rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Vienna: ACM, 2024. 1733–1745. [doi: 10.1145/3650212.3680395]

[82] van der Maaten L, Hinton G. Visualizing data using t-SNE. Journal of Machine Learning Research, 2008, 9(11): 2579–2605.

[83] Li XY, Meng GZ, Liu SQ, Xiang L, Sun K, Chen K, Luo XP, Liu Y. Attribution-guided adversarial code prompt generation for code completion models. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Automated Software Engineering. Sacramento: ACM, 2024. 1460–1471. [doi: 10.1145/3691620.3695517]

[84] Zhang C, Wang ZF, Zhao RS, Mangal R, Fredrikson M, Jia LM, Păsăreanu CS. Attacks and defenses for large language models on coding tasks. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Automated Software Engineering. Sacramento: IEEE, 2024. 2268–2272.

[85] Yang Y, Yao H, Yang B. TAPI: Towards target-specific and adversarial prompt injection against code LLMs. arXiv:2407.09164, 2024.

[86] Guo DY, Zhu QH, Yang DJ, Xie ZD, Dong K, Zhang WT, Chen GT, Bi X, Wu Y, Li YK, Luo FL, Xiong YF, Liang WF. DeepSeek-Coder: When the large language model meets programming—The rise of code intelligence. arXiv:2401.14196, 2024.

[87] Xiao Y, Lin Y, Beschastnikh I, Sun CS, Rosenblum D, Dong JS. Repairing failure-inducing inputs with input reflection. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 85. [doi: 10.1145/3551349.3556932]

[88] Karer HH, Soni PB. Dead code elimination technique in eclipse compiler for Java. In: Proc. of the 2015 Int'l Conf. on Control, Instrumentation, Communication and Computational Technologies. Kumaracoil: IEEE, 2015. 275–278. [doi: 10.1109/ICCICCT.2015.7475289]

[89] Jha A, Reddy CK. CodeAttack: Code-based adversarial attacks for pre-trained programming language models. In: Proc. of the 2023 AAAI Conf. on Artificial Intelligence. Washington: AAAI Press, 2023. 14892–14900. [doi: 10.1609/aaai.v37i12.26739]

[90] Liu DX, Zhang SK. ALANCA: Active learning guided adversarial attacks for code comprehension on diverse pre-trained and large language models. In: Proc. of the 2024 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering. Rovaniemi: IEEE, 2024. 602–613. [doi: 10.1109/SANER60148.2024.00067]

[91] Bui NDQ, Yu YJ, Jiang LX. TreeCaps: Tree-based capsule networks for source code processing. In: Proc. of the 2021 AAAI Conf. on Artificial Intelligence. AAAI Press, 2021. 30–38. [doi: 10.1609/aaai.v35i1.16074]

[92] Wang X, Wang YS, Mi F, Zhou PY, Wan Y, Liu X, Li L, Wu H, Liu J, Jiang X. SynCoBERT: Syntax-guided multi-modal contrastive pre-training for code representation. arXiv:2108.04556, 2021.

[93] Liu SQ, Wu BZ, Xie XF, Meng GZ, Liu Y. ContraBERT: Enhancing code pre-trained models via contrastive learning. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2476–2487. [doi: 10.1109/ICSE48619.2023.00207]

[94] Wang X, Wu Q, Zhang HY, Lyu C, Jiang X, Zheng ZR. HELoC: Hierarchical contrastive learning of source code representation. In: Proc. of the 30th IEEE/ACM Int'l Conf. on Program Comprehension. Pittsburgh: IEEE, 2022. 354–365. [doi: 10.1145/3524610.

3527896]

[95] Allamanis M, Jackson-Flux H, Brockschmidt M. Self-supervised bug detection and repair. In: Proc. of the 35th Int'l Conf. on Neural Information Processing Systems. Curran Associates Inc., 2021. 27865–27876.

[96] Li Z, Zhang C, Pan MX, Zhang T, Li XD. AACEGEN: Attention guided adversarial code example generation for deep code models. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Automated Software Engineering. Sacramento: ACM, 2024. 1245–1257. [doi: 10.1145/3691620.3695500]

[97] Mi Q, Xiao Y, Cai Z, Jia XB. The effectiveness of data augmentation in code readability classification. Information and Software Technology, 2021, 129: 106378. [doi: 10.1016/j.infsof.2020.106378]

[98] Zou J, Zhang SG, Qiu MK. Adversarial attacks on large language models. In: Proc. of the 17th Int'l Conf. on Knowledge Science, Engineering and Management. Birmingham: Springer, 2024. 85–96. [doi: 10.1007/978-981-97-5501-1_7]

[99] Kumar P. Adversarial attacks and defenses for large language models (LLMs): Methods, frameworks & challenges. Int'l Journal of Multimedia Information Retrieval, 2024, 13(3): 26. [doi: 10.1007/s13735-024-00334-8]

[100] Jain N, Schwarzschild A, Wen YX, Somepalli G, Kirchenbauer J, Chiang PY, Goldblum M, Saha A, Geiping J, Goldstein T. Baseline defenses for adversarial attacks against aligned language models. arXiv:2309.00614, 2023.

[101] Robey A, Wong E, Hassani H, Pappas GJ. SmoothlLM: Defending large language models against jailbreaking attacks. arXiv:2310.03684, 2023.

[102] Kumar A, Agarwal C, Srinivas S, Li AJ, Feizi S, Lakkaraju H. Certifying LLM safety against adversarial prompting. arXiv:2309.02705, 2023.

[103] Sheshadri A, Ewart A, Guo P, Lynch A, Wu C, Hebbar V, Sleight H, Stickland AC, Perez E, Hadfield-Menell D, Casper S. Latent adversarial training improves robustness to persistent harmful behaviors in LLMs. arXiv:2407.15549, 2024. [doi: 10.48550/arXiv.2407.15549]

[104] Lin G, Zhao QB. Large language model sentinel: Advancing adversarial robustness by LLM agent. arXiv:2405.20770, 2024.

[105] Xiao Y, Beschastnikh I, Rosenblum DS, Sun CS, Elbaum S, Lin Y, Dong JS. Self-checking deep neural networks in deployment. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 372–384. [doi: 10.1109/ICSE43902.2021.00044]

[106] Selvaraju RR, Cogswell M, Das A, Vedantam R, Parikh D, Batra D. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In: Proc. of the 2017 IEEE Int'l Conf. on Computer Vision. Venice: IEEE, 2017. 618–626. [doi: 10.1109/ICCV.2017.74]

附中文参考文献

[4] 王赞, 闫明, 刘爽, 陈俊洁, 张栋迪, 吴卓, 陈翔. 深度神经网络测试研究综述. 软件学报, 2020, 31(5): 1255–1275. http://www.jos.org.cn/1000-9825/5951.htm [doi: 10.13328/j.cnki.jos.005951]

[17] 曹雪洁, 陈俊洁, 闫明, 尤翰墨, 吴卓, 王赞. 基于数据变异的神经网络测试用例选择方法. 软件学报, 2024, 35(11): 4973–4992. http://www.jos.org.cn/1000-9825/7005.htm [doi: 10.13328/j.cnki.jos.007005]

作者简介

田朝, 博士生, 主要研究领域为软件测试.

邝仕琦, 本科生, 主要研究领域为代码大模型测试与增强.

闫明, 博士生, 主要研究领域为深度学习系统测试, 芯片设计程序测试.

王海弛, 博士生, 主要研究领域为系统软件测试.

陈俊洁, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件分析与测试.