软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(11):5062−5081 [doi: 10.13328/j.cnki.jos.007379] [CSTR: 32375.14.jos.007379]

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

面向深度漏洞检测模型的黑盒对抗攻击*

曲豫宾[1,2,3], 黄 松[1], 陈 翔[4], 王兴亚[5], 李 龙[3], 王 丹[5], 姚永明[1], 鞠小林[4]

[1](中国人民解放军陆军工程大学 指挥控制工程学院, 江苏 南京 210007)

[2](江苏工程职业技术学院 信息工程学院, 江苏 南通 226001)

[3](广西可信软件重点实验室 (桂林电子科技大学), 广西 桂林 541004)

[4](南通大学 信息科学技术学院, 江苏 南通 226019)

[5](南京工业大学 人工智能学院, 江苏 南京 211816)

通信作者: 黄松, E-mail: huangsong@aeu.edu.cn

摘 要: 近年来, 基于深度学习的漏洞检测模型展示了令人印象深刻的漏洞检测能力. 通过对源代码的变量重命名实现扰动从而逃逸漏洞检测模型的对抗攻击已被广泛研究, 然而, 能否通过对源代码进行更多的扰动以提升对漏洞检测模型对抗攻击的效果并未被充分调研. 采用针对源代码的多种同义转换算子对其进行扰动, 并提出一种基于遗传算法的组合优化策略, 选择一组适应度最高的源代码同义转换算子, 以指导生成可以逃脱漏洞检测的对抗代码段. 在一个名为 NonVulGen (无漏洞代码生成器, non vulnerability generator) 的框架中实现了上述方法, 采用生成的对抗代码段对基于深度学习的漏洞检测模型进行评估. 考虑了最新提出的多种深度学习漏洞检测模型, 在攻击基于 CodeBERT 的漏洞检测模型时所提方法可以实现 91.38% 的平均攻击成功率, 在攻击基于 GraphCodeBERT 的漏洞检测模型时可以实现 93.65% 的平均攻击成功率, 以上结果相较于最先进的基线模型平均提升了 28.94% 和 15.52%. 为了检验所提攻击方法的泛化能力, 攻击 Devign、ReGVD 和 LineVul 等常见漏洞检测模型, 平均攻击成功率分别为 98.88%、97.85% 和 92.57%. 实验结果表明, 基于深度学习的漏洞检测模型无法有效区分由 NonVulGen 生成的对抗代码段. 进一步地, 基于训练数据生成的对抗代码段重训练漏洞检测模型, 在 CodeBERT, GraphCodeBERT, Devign, ReGVD 以及 LineVul 模型上攻击成功率均大幅度下降, 分别下降 96.83%, 97.12%, 98.79%, 98.57%, 以及 97.94%. 因此, 研究揭示基于深度学习漏洞检测模型存在亟待关注的对抗攻击问题并呼吁在发布漏洞检测模型之前做好模型加固工作.

关键词: 漏洞检测; 代码转换; 深度学习; 黑盒鲁棒攻击

中图法分类号: TP311

中文引用格式: 曲豫宾, 黄松, 陈翔, 王兴亚, 李龙, 王丹, 姚永明, 鞠小林. 面向深度漏洞检测模型的黑盒对抗攻击. 软件学报, 2025, 36(11): 5062–5081. http://www.jos.org.cn/1000-9825/7379.htm

英文引用格式: Qu YB, Huang S, Chen X, Wang XY, Li L, Wang D, Yao YM, Ju XL. Black-box Adversarial Attack for Deep Vulnerability Detection Model. Ruan Jian Xue Bao/Journal of Software, 2025, 36(11): 5062–5081 (in Chinese). http://www.jos.org.cn/1000-9825/7379.htm

Black-box Adversarial Attack for Deep Vulnerability Detection Model

QU Yu-Bin[1,2,3], HUANG Song[1], CHEN Xiang[4], WANG Xing-Ya[5], LI Long[3], WANG Dan[5], YAO Yong-Ming[1], JU Xiao-Lin[4]

* 基金项目: 国家自然科学基金 (U23A20280, 62172350); 广东省基础与应用基础研究基金 (2023A1515012846); 广西壮族自治区重点研发计划 (2023AB29051); 江苏省“青蓝工程”中青年学术带头人培养项目; 南京大学软件新技术国家重点实验室开放项目 (KFKT2022B10); 广西可信软件重点实验室 (kx202046); 南通市自然科学基金面上项目 (JC2023070)
收稿时间: 2024-01-04; 修改时间: 2024-07-24; 采用时间: 2024-12-18; jos 在线出版时间: 2025-05-14
CNKI 网络首发时间: 2025-05-15

[1](College of Command and Control Engineering, Army Engineering University of PLA, Nanjing 210007, China)
[2](School of Information Engineering, Jiangsu College of Engineering and Technology, Nantong 226001, China)
[3](Guangxi Key Laboratory of Trusted Software (Guilin University of Electronic Technology), Guilin 541004, China)
[4](School of Information Science and Technology, Nantong University, Nantong 226019, China)
[5](College of Artificial Intelligence, Nanjing Tech University, Nanjing 211816, China)

Abstract: In recent years, impressive capabilities have been demonstrated by deep learning-based vulnerability detection models in detecting vulnerabilities. Previous research has widely explored adversarial attacks using variable renaming to introduce disturbances in source code and evade detection. However, the effectiveness of introducing multiple disturbances through various transformation techniques in source code has not been adequately investigated. In this study, multiple synonymous transformation operators are applied to introduce disturbances in source code. A combination optimization strategy based on genetic algorithms is proposed, enabling the selection of source code transformation operators with the highest fitness to guide the generation of adversarial code segments capable of evading vulnerability detection. The proposed method is implemented in a framework named non-vulnerability generator (NonVulGen) and evaluated against deep learning-based vulnerability detection models. When applied to recently developed deep learning models, an average attack success rate of 91.38% is achieved against the CodeBERT-based model and 93.65% against the GraphCodeBERT-based model, representing improvements of 28.94% and 15.52% over state-of-the-art baselines, respectively. To assess the generalization ability of the proposed attack method, common models including Devign, ReGVD, and LineVul are targeted, achieving average success rates of 98.88%, 97.85%, and 92.57%, respectively. Experimental results indicate that adversarial code segments generated by NonVulGenx cannot be effectively distinguished by deep learning-based vulnerability detection models. Furthermore, significant reductions in attack success rates are observed after retraining the models with adversarial samples generated based on the training data, with a decrease of 96.83% for CodeBERT, 97.12% for GraphCodeBERT, 98.79% for Devign, 98.57% for ReGVD, and 97.94% for LineVul. These findings reveal the critical challenge of adversarial attacks in deep learning-based vulnerability detection models and highlight the necessity for model reinforcement before deployment.

Key words: vulnerability detection; code transformation; deep learning; black-box robustness attack

1 引 言

随着软件系统的复杂性不断增加, 传统的人工审查方法已无法满足高效、精准的漏洞检测需求. 漏洞检测模型对于确保软件安全至关重要, 基于深度学习的漏洞检测模型越来越被广泛研究与应用. 近年来, 深度学习漏洞检测工具取得了令人期待的结果. 最先进的模型报告了 0.9 的 $F1$ 得分[1,2], 并且优于静态分析工具[3,4]. 这些结果令人兴奋, 因为深度学习可能为软件保障带来变革性的改变. 因此, IBM、谷歌和亚马逊等工业公司非常感兴趣, 并且投入了大量资源来开发这些工具并共享了数据集[5,6]. 深度学习已经在漏洞检测领域展现了强大的能力, 近年来研究者们提出了多种基于深度学习的漏洞检测模型, 如 Devign、ReGVD 和 LineVul 等, 这些模型在自动化检测软件漏洞方面取得了显著进展, 比如 Devign 模型利用图神经网络从代码中提取结构化信息, 自动识别潜在的漏洞, ReGVD 结合代码的静态和动态特征, 提升了漏洞检测的准确性, LineVul 通过深度学习模型识别代码行级别的漏洞, 提高了检测的精细度.

随着越来越多的深度学习解决方案被用于软件工程社区的任务, 研究者们也提出了针对源代码处理模型鲁棒性的担忧, 即除了高精度之外, 当前 SOTA (state-of-the-art) 源代码处理模型的鲁棒性如何. 研究[7]表明, 像 code2vec[8] 和 code2seq[9]这样的代码模型, 对于两个共享相同操作语义的代码片段可能会输出不同的结果, 其中一个是通过对另一个代码片段中的一些变量进行重命名而生成的. 修改后的代码片段被称为对抗样本, 而受到攻击的模型被称为受害模型[10]. 这就意味着这些基于深度学习的漏洞检测模型也面临对抗攻击的威胁, 攻击者可以通过精心设计的对抗样本绕过检测, 从而隐藏漏洞.

与图形图像领域和自然语言处理领域鲁棒性攻击不同的是, 源代码处理模型需要处理的对象是代码片段. 图像和自然语言处理中的对抗攻击通常利用人眼无法察觉的微小扰动来迷惑模型, 而代码对抗攻击则需要在保持代码功能不变的前提下, 通过语法和语义上等价的变换来绕过模型检测. 具体来讲, 图像对抗攻击是通过将扰动直接应用于像素值, 生成的对抗样本在视觉上与原始样本几乎无异; 而在自然语言对抗攻击中, 通常通过替换、删除或插入单词等方式进行扰动, 保持句子语法正确且语义通顺. 代码对抗攻击则是通过重命名变量、修改注释、插入

无用代码等方式实现扰动, 必须确保代码的可编译性和功能性不受影响. 在源代码上下文中, 由于存在多种独特的约束和要求, 对抗性攻击和鲁棒性分析变得更具挑战性. 现有的图像或语音案例的鲁棒性分析主要基于欧几里得空间中的距离, 而程序源代码是离散的, 并严格遵循形式语法 (例如, 上下文无关语法), 这使得鲁棒性测量难以通过欧几里得空间中的距离来定义. 只有遵循语法的代码片段才有效, 可用于鲁棒性分析和估计. 以上对抗代码段的特性被称为语法有效性, 即生成的代码 (对抗性示例) 必须是可编译的 (即语法有效), 并保留原始代码片段的语义. 对抗代码段还需要考虑了以下属性: 生成效率, 即对抗攻击要能在复杂约束下, 在巨大的空间内快速找到对抗代码段. 自然性, 即对抗攻击方法生成的对抗代码段应该是程序员能够理解的和自然的. 攻击成功率, 对抗性攻击通常用于估计深度学习模型在最坏情况下的鲁棒性, 其中更强的对抗性攻击技术可以更准确地近似真实对抗攻击.

研究人员[7,10−17]做了初步研究, 发现了深度源代码处理模型中的鲁棒性问题. 这些研究通过对代码变量名重命名或者对代码段同义转换等生成对抗代码段, 这就保证了生成的对抗代码段满足语法有效性的需求. 然而这些研究并未在生成效率、自然性与攻击成功率之间找到平衡. 特别是到目前为止, 基于深度学习的漏洞检测模型的鲁棒性问题还没有得到足够的重视, 现有工作的进展有限. Yang 等人[10]提出了 ALERT, 它是一种在生成代码的对抗样本时考虑自然语义的黑盒攻击. 如图 1(b) 所示, 局部变量名 ctx 被重命名为 cas. 该对抗攻击方法充分考虑了生成对抗代码段的语法有效性、生成效率与自然性. 基于深度学习的漏洞检测模型的任务就是识别出有漏洞的代码片段, 恶意用户可能会编写易受攻击的代码片段, 并不希望它们被识别出来. 在这些安全相关的任务中, 包括恶意软件检测或漏洞检测, 攻击者通过利用精心设计的代码变体, 将有漏洞的代码发送到下游处理模块, 从而直接绕过漏洞检测模型, 这将对整个系统造成致命的错误. 特别是对抗攻击成功率越高, 即意味着有漏洞的代码段逃逸漏洞检测模型概率越大, 给软件质量保障带来安全隐患也就越高.

就基于深度学习的漏洞检测任务来讲, 尚没有针对基于深度学习的漏洞检测模型对抗攻击的深入研究. 为了弥合这一差距, 本文提出了一个通用框架 NonVulGen (non vulnerability generator), 用于源代码处理背景下的深度学习漏洞检测模型鲁棒性检测、增强, 这是大代码时代许多软件工程任务的必备. 我们首先提出了一种有效且高效的对抗性攻击技术, 专门设计用于基于深度学习的漏洞检测模型. 此外, 提出了基于对抗训练的对抗增强技术. 综合使用变量重命名、代码结构同义转换等算子对有漏洞代码段进行扰动, 将对抗代码段生成过程转换为一组代码同义转换算子向量的组合优化问题. 如图 1(c) 所示, 局部变量名 ctx 被重命名为 c_tx_756, 同时在选择结构 if 中插入了一段死代码, 改变了函数的语法树结构. 具体来讲, NonVulGen 将生成的对抗代码段用于对最先进的深度学习漏洞检测模型进行评估. 本文方法主要分为 3 个阶段: 基于贪婪搜索创建种子染色体, 基于遗传算法进行组合优化, 以及针对特定代码段的深度搜索, 用于遗传算法组合优化失败时更全面地搜索对抗样本. 我们将 5 种漏洞检测模型作为受害模型, 包括 Devign、ReGVD、LineVul、基于 CodeBERT[18]和 GraphCodeBERT[19]微调的受害者模型. 对 5 种受害模型进行随机搜索攻击、ALERT 攻击与 NonVulGen 攻击, 以挑战这 5 种最先进的深度学习漏洞检测模型. 本文实验结果表明, 基于深度学习的漏洞检测模型无法有效区分由 NonVulGen 生成的对抗代码段. 进一步地, 我们基于训练数据生成的对抗代码段重训练了漏洞检测模型, 攻击成功率在各个模型上均大幅度下降. 因此, 本文研究揭示了基于深度学习漏洞检测模型存在亟待关注的对抗攻击问题并呼吁在发布漏洞检测模型之前做好模型加固工作.

本文的贡献如下.

• 我们在为漏洞检测模型生成对抗样本时, 提出同时考虑自然性要求以及多种代码转换算子方法. 还提出了 NonVulGen, 相比之前的 SOTA 方法, NonVulGen 可以实现更高的攻击成功率.

• 对有代表性的深度学习漏洞检测模型实施大范围的对抗攻击实证研究, 并且展示了最先进漏洞检测模型容易受到对抗攻击的情况.

• 展示了 NonVulGen 生成的样本的价值: 通过对这些对抗样本对受害模型进行微调和加固, 攻击成功率在各个模型上均大幅度下降.

• 在项目主页 (https://github.com/qyb156/RobustnessAttackVunerbilityDetection) 上共享了所有实验数据和本文的源代码, 供实验结果复现、未来研究和实际使用.

```c
static ExitStatus trans_log(DisasContext *ctx, uint32_t insn, const DisasInsn *di)
{
    unsigned r2 = extract32(insn, 21, 5);
    unsigned r1 = extract32(insn, 16, 5);
    unsigned cf = extract32(insn, 12, 4);
    unsigned rt = extract32(insn,  0, 5);
    TCGv tcg_r1, tcg_r2;
    ExitStatus ret;
    if (cf) {
        nullify_over(ctx);
    }
    tcg_r1 = load_gpr(ctx, r1);
    tcg_r2 = load_gpr(ctx, r2);
    ret = do_log(ctx, rt, tcg_r1, tcg_r2, cf, di->f_ttt);
    return nullify_end(ctx, ret);
}
```

(a) 被 CodeBERT 模型正确分类的原始代码

```c
static ExitStatus trans_log(DisasContext *cas, uint32_t linssn,const DisasInsn *di)
{
    unsigned r2 = extract32(linssn, 21, 5);
    unsigned r1 = extract32(linssn, 16, 5);
    unsigned cf = extract32(linssn, 12, 4);
    unsigned rt = extract32(linssn,  0, 5);
    TCGv tcg_r1, tcg_R82;
    ExitStatus lit;
    if (cf) {
        nullify_over(cas);
    }
    tcg_r1 = load_gpr(cas, r1);
    tcg_R82 = load_gpr(cas, r2);
    lit = do_log(cas, rt, tcg_r1, tcg_R82, cf, di->f_ttt);
    return nullify_end(cas, lit);
}
```

(b) ALERT 通过替换变量名生成的对抗代码

```c
static ExitStatus tmjns_log (DisasContext *c_tx_756, uint32_t iyun, const  DisasInsn *d_i_716)
{
    unsigned  r_2_164 = extract32 (iyun, 21, (677 - 672));
    unsigned  r_1_999 = extract32 (iyun, 16, (994 - 989));
    unsigned  c_f_396 = extract32 (iyun, 12, 4);
    unsigned  r_t_485 = extract32 (iyun, (814 - 814), 5);
    TCGv tpf_r1, tuh_r2;
    ExitStatus r_et_59;
    if (c_f_396) {
        {
            if (0) {
                return 0;
            };
        }
        nullify_over (c_tx_756);
    }
    tpf_r1 = load_gpr (c_tx_756, r_1_999);
    tuh_r2 = load_gpr (c_tx_756, r_2_164);
    r_et_59 = do_log (c_tx_756, r_t_485, tpf_r1, tuh_r2, c_f_396, d_i_716->f_ttt);
    return nullify_end (c_tx_756, r_et_59);
}
```

(c) NonVulGen 通过替换变量名并在代码结构上实现同义转换来生成的对抗代码

图 1　对抗代码对比

本文第 2 节简要介绍研究背景和相关工作. 在第 3 节中, 提出设计对抗性攻击方法所面临的技术难题. 在第 4 节中, 我们详细阐述所提出方法 NonVulGen 的设计. 在第 5 节中描述实验的设置, 并在第 6 节中展示本文实验结果, 比较 ALERT 和一些基准的性能. 在第 7 节总结对有效性的威胁. 最后, 在第 8 节总结论文并提出未来的工作.

2 相关工作

本节介绍了基于深度学习的漏洞检测模型, 描述了针对基于深度学习的漏洞检测模型的对抗攻击.

2.1 基于深度学习的漏洞检测模型

识别潜在的易受攻击代码对于提高软件系统的安全性非常重要. 然而, 手动检测软件漏洞不仅需要专业知识, 而且非常耗时, 所以得到自动化技术的支持是非常有必要的. 基于深度神经网络 (DNN) 可以学习到高维度更加复杂和抽象的代码表征, 自动地学习更加通用的特征, 将研究者从主观的、耗费人力的、易错的任务中解放出来, 而且可以灵活地使用各种深度神经网络结构去获取不同应用场景的特征. 到目前为止, 已经提出了多种基于深度学习的方法, 以便从历史数据中自动学习漏洞模式.

Zhou 等人[20]构建了 Devign 模型, 该模型使用 GGNN 从代码属性图中学习功能级代码特征, 并添加了一维卷积, 用于功能级漏洞检测. Chakraborty 等人[21]提出了 ReVeal 模型, 并在漏洞检测数据集上进行了测试. 他们验证了数据集组合在漏洞检测任务中的影响. Devign 和 ReVeal 在集成控制流、数据依赖性和 AST 的属性图上使用了图神经网络 (GNN). Nguyen 等人[22]提出了基于滑动窗口机制构建标记之间的图关系的 ReGVD 漏洞检测. 随着预训练模型被广泛用于漏洞检测模型, Feng 等人[18]提出了 CodeBERT, 一个用于编程语言和自然语言的双峰预训练模型并将其应用于下游功能级漏洞检测任务上. Guo 等人[19]添加了边缘预测和节点对齐任务来训练一个跨语言预训练的编程语言模型, 名为 GraphCodeBERT. 他们的目标是学习代码序列的结构性信息, 同时, 还将 GraphCodeBERT 应用于下游功能级漏洞检测任务. 其他预训练模型还包括 VulBERTa-CNN[23]等. Li 等人[24]提出 IVDetect 来预测函数级别的漏洞, 但 IVDetect 方法仍然不准确且粗粒度, Fu 等人[1]提出了 LineVul, 一种基于 Transformer 的函数级漏洞预测方法, 以解决最先进的 IVDetect 方法的几个局限性.

2.2 针对深度神经网络的对抗攻击

尽管 DNN 模型在许多任务上取得了巨大成功, 但之前的研究工作[25]表明, 基于 DNN 的模型容易受到对抗攻击的影响. 对抗攻击的目标是通过轻微扰动原始输入来生成对人类评判者来说自然的对抗性示例, 以欺骗 DNN 模型. 已经提出了许多技术来证明对抗性示例可以在不同领域的模型中找到, 包括图像分类[26]、强化学习[27]、语音识别[28]等. 根据攻击者是否可以访问的受害者模型信息, 对抗性攻击可以分为两种类型: 白盒和黑盒. 在白盒设置中, 攻击者可以访问受害者模型的所有信息, 例如使用模型参数计算梯度. 但是白盒攻击通常缺乏实用性, 因为受害者模型通常是远程部署的 (例如在云服务上), 通常攻击者只能访问 API 来查询模型以及相应的输出. 黑盒攻击意味着攻击者只知道受害者模型的输入和输出 (例如预测标签和相应的置信度). 对抗性攻击也可以分为无目标攻击和有目标攻击. 无目标攻击只是为了使受害模型产生错误的预测, 而有目标攻击则是强迫受害模型做出特定的预测. 比如图形图像处理领域中的后门攻击研究就属于有目标攻击, 在对抗图像中写入不可见的触发器, 即可使得受害模型输出攻击者想要的类别[29]. 我们针对漏洞检测模型的攻击既属于黑盒攻击, 也属于无目标攻击. 我们的攻击不需要知道模型的梯度信息、训练数据集等与模型相关的信息. 另外本文提出的攻击方法的目标是使得漏洞检测模型有错误的输出, 具体来讲将有漏洞代码段预测为其他无漏洞的类型即可.

2.3 针对深度漏洞检测模型的对抗攻击

深度学习在源代码的自动处理、分析和生成等任务方面表现出一定程度的能力, 但目前最先进的深度学习 (DL) 模型仍然存在非鲁棒的问题. 与对图像、音频和自然语言的对抗性攻击不同, 编程语言的结构化本质带来了新的挑战. 目前针对深度代码模型的对抗攻击被证明是有效的. Yang 等人[10]使用变量重命名作为对抗性示例生成技术, 并探索如何生成自然的对抗性示例. 因为基于变量重命名生成对抗性样例不需要梯度信息, 并且使用细粒度的模型输出 (即预测结果和相应的置信度) 来进行重命名, 并且实现了良好的攻击成功率. 为了确保重命名的变量对操作语义没有任何改变, 他们只对代码片段中的局部变量进行重命名.

除了本文研究中比较的最新技术 (ALERT[10]), 还有一些其他针对深度代码模型的敌对样本生成技术[7,11−17].

Zhang 等人[7]将对抗样本生成过程形式化为一个采样问题. 该问题可以分解为一个迭代过程, 包括 3 个阶段:

(1) 选择要重命名的变量, (2) 选择替换项, (3) 决定是否接受用所选替换项替换变量. 他们提出 MHM, 这是一种基于 Metropolis-Hastings 采样的代码模型识别重命名技术. 这种方法是一种黑盒攻击, 它随机选择局部变量的替换, 并根据预测标签和相应的置信度来决定是否接受或拒绝替换. 它利用受害模型的预测标签和相应的置信度更有效地选择对抗样本. Srikant 等人[11]提出, 生成一个成功的对抗程序被构建为一个优化问题. 解决优化问题并不是轻而易举的, 因为它具有组合性质, 存在双线性目标项和多个约束条件. 为了解决这个问题, 他们提出了基于投影梯度下降 (PGD) 的联合优化求解器, 并提出了可以更好地实现实证性能的替代方案. 在生成对抗示例的过程中, 他们还需要受害模型的内部梯度信息. Yefet 等人[12]提出了 DAMP, 通过梯度计算来改变目标输入中的变量. DAMP 需要访问受害模型的梯度等内部信息, 属于白盒攻击. Pour 等人[13]提出了一个基于搜索的测试框架, 用于源代码嵌入和其下游处理任务, 如代码搜索. 为了生成新的测试输入, 他们采用了流行的源代码重构工具来生成语义等效的变体. Henkel 等人[14]通过基于梯度的优化为深度代码模型生成了对抗性示例, 包括重命名转换和无效代码插入. 这也是一种白盒攻击. Jha 等人[15]提出了 CodeAnack, 采用掩码语言模型与贪婪搜索来预测易受攻击的标记的替代词. 他们从变量名替换的角度来解决代码扰动, 而不考虑其他代码扰动方法. Zhang 等人[16]提出了对抗性攻击技术 CARROT. 为了高效和有效, 攻击技术采用了基于优化的技术, 利用梯度信息对源代码进行扰动. 这也属于白盒攻击. Tian 等人[17]提出了对抗性示例生成技术 CODA, 其关键思想是利用目标输入 (即给定的代码片段作为模型输入) 与参考输入 (即具有小代码差异但与目标输入具有不同预测结果的输入) 之间的代码差异来引导对抗性示例的生成. 它考虑了结构差异和标识符差异, 以保留原始语义. 因此, 成分搜索空间可以大大减少, 因为它由这两种代码差异组成, 从而可以通过设计和引导相应的等效结构转换和标识符重命名转换来改进测试过程.

考虑到漏洞检测模型通常在远程部署, 使得模型参数无法访问, 我们将重点放在对深度漏洞检测模型的黑盒攻击上. 我们考虑的攻击场景是: 恶意攻击者无法获取漏洞检测模型的训练数据集, 能够与漏洞检测模型进行交互; 攻击生成的源代码要符合自然性; 而该攻击方法不但能适用于预训练模型, 也能适用于其他模型, 比如基于图神经网络的漏洞检测模型. MHM[7]对抗攻击方法在攻击的时候未考虑代码的自然性问题, 其他多种对抗代码生成方法[11,12,14,16]需要使用到受害模型的内部信息, 属于白盒攻击. CODA[17]需要用到训练数据集来实施对抗攻击, 以上这些方法与本文的攻击场景不同. 因此, 我们将攻击方法 ALERT 作为本文的基线方法进行对比. 我们观察到能够引起漏洞检测模型预测错误的非鲁棒性特征, 不但包括变量名, 还包括其他代码特征, 比如数值常量、逻辑运算符等. 本文采用了 Zhang 等人[30]提出的代码同义转换算子, 将经过同义转换后的代码作为对抗代码段, 与基于变量重命名的对抗攻击技术不同, 本文提出的对抗攻击技术不但要考虑变量重命名的自然性, 也考虑了其他代码转换方式. 实施该对抗攻击存在的技术问题是, 对于不同的代码段, 到底哪些特征是引起漏洞检测模型分类错误的有效非鲁棒性特征, 是我们要在第 3 节中要论述和解决的问题.

3 对抗攻击面临的技术问题

漏洞检测模型对抗攻击样本要满足: 语法有效性、生成效率、自然性、攻击成功率等约束, 搜索非鲁棒性源代码特征是实施对抗攻击的关键. 之前的研究表明, 对源代码中的变量进行重命名, 并对代码进行结构上的同义转换能够对代码模型实施对抗攻击[11,12,14,16,17,30]. 对代码进行结构上的同义转换并非本文的创新点, Tian 等人[17]与 Zhang 等人[30]都提出过对源代码进行同义转换从而实施对基于深度学习的代码模型的对抗攻击. 与他们的工作不同的是, 我们把针对漏洞检测模型的黑盒攻击转换为基于遗传算法的非鲁棒性代码特征的组合优化问题. 使用 NonVulGen 进行黑盒攻击的思路是直观的, 但是要解决几个技术问题才能对漏洞检测模型进行对抗攻击, 待解决的技术问题如下所述.

3.1 代码同义转换算子的选择问题

如 Zhang 等人[30]提出的, 应该选择代码同义转换算子以至于能够轻微修改代码的语法, 但保持代码的语义不变. 如表 1 所示, 我们在 NonVulGen 框架中采用了 Zhang 等人提出的 14 种代码转换算子, 包括 Op2-ChFor、Op3-ChWhile 等. 另外我们重新实现了代码变量重命名算子 Op1-ChRename, 实现过程如算法 1 所示.

表 1 NonVulGen 中 15 个同义转换算子的描述和示例

Transformation operator	Description	Original	Changed
Op1-ChRename	函数名和变量名重命名	int mov_read_strf;	int mtw_read_strf;

Op2-ChFor	for 循环被转换为 while 循环	for(i=0;i<10;i++)
		{ BodyA
		}
			i=0; while(i<10){
			BodyA
			i++;}

Op3-ChWhile	while 循环被转换为 for 循环	while(i<10) {BodyA }	for(;i<10;) BodyA}

Op4-ChDo	do 循环被转换为 while 循环	do{
		BodyA}
		while(i<10);
			BodyA
			while(i<10){
			BodyA}

Op5-ChIfElselF	转换 if elseif 到 ifelse	if(grad<60)BodyA
		else
		if(grad<80)
		BodyB
		else
		BodyC
			if(grad<60) BodyA
			else { if(grad<80)
			BodyB
			else
			BodyC }

Op6-Chif	转换 ifelse 为if elseif	if(grad<60) BodyA
		else{ if(grad<80)
		BodyB
		else
		BodyC }
			if(grad<60) BodyA
			else
			if(grad<80)
			BodyB
			else
			BodyC

Op7-ChSwitch	转换 switch 语句为 if elseif 语句	switch(a) {case 60: BodyA case 70: BodyB default: BodyC }	if(a==60) BodyA else if(a==70) BodyB else BodyC
Op8-ChRelation	关系表达式转换	a<b	b>a
Op9-ChUnary	对一元运算修改	i++;	i=i+1;
Op10-ChIncrement	增量操作修改	i+=1;	i=i+1;
Op11-ChConstant	修改常量	8	(a–b)//8=a–b
Op12-ChDefine	变量定义的修改	int b=0;	int b; b=0;

Op13-ChAddJunk	添加无用代码	if(a){BodyA}	if(a) {
			BodyA
			if(0)
			return 0; }

Op14-ChExchange	改变无控制依赖关系 语句块的顺序	a=b+10; c=d+10;	c=d+10; a=b+10:
Op15-ChDeleteComments	删除打印调试提示 和注释的语句	printf(“test”); //comments	//printf(“test”); //comments

算法 1. 自然感知函数名和变量名重命名算法.

Input: *i*: input function name or variable name;

Output: *ou*: Renamed function name or variable name.

1. **if** *length*(*input*) ⩽ 3 **then**
2. **return** *input* + *randomNumber*()
3. **else**
4. *method* ← *randomChoice*(1, 2)
5. **if** *method* = 1 **then**

```
6.        output ← input
7.        pos1, pos2 ← randPos(1, length(input)–1, 2)
8.        output[pos1] ← randomChar()
9.        output[pos2] ← randomChar()
10.       return output + randomNumber()
11.   else
12.       output ← APTInterface.getSynonym(input)
13.       return output
14.   end if
15. end if
```

如 Yang 等人[10]与 Tian 等人[17]提出的, 开发人员关心代码的自然语义, 因为它可以帮助人类理解模型中检测到的错误[31]. 因此, 确保生成的对抗样本的自然性非常重要. Zhang 等人[30]采用的代码变量重命名算子 Op1-ChRename 采用随机重命名的方式实施变量名扰动, 不满足自然性的要求. Yang 等人[32,33] 发现, 程序员在书写代码的过程中, 可能会出现 typo 错误, 这些错误在程序员来看是自然的, 但是对于漏洞检测模型来讲是一种变量扰动, Belinkov 等人[34]采用这种打字错误和拼写错误对深度神经机器翻译进行对抗攻击. 基于这个观察, 我们重新实现代码变量重命名算子 Op1-ChRename. 该方法的思路是简单的, 对源代码中的长度大于 3 的变量名, 选择两个字符进行随机扰动, 模仿程序员可能会出现的 typo 错误. 比如, 原来的函数名称为 *mov_read_strf*, 经过对抗扰动以后, 函数名称为: *mtw_read_strf*. 进一步地, 采用了基于大语言模型的同义标识符重命名策略以提升代码的自然性, 并在一定程度上保持代码的可维护性和可理解性.

3.2 代码同义转换算子的编码问题

我们采用 15 种同义转换算子对源代码进行扰动, 如表 1 所示, 过多的扰动可能会导致对抗代码段自然性较差, 过少的扰动可能无法生成对抗代码段, 将是否选择某个代码同义转换算子转换为二进制变量, 当该变量取值为 0 的时候, 表示不使用该算子对源代码进行同义转换; 当取值为 1 的时候表示使用该算子对源代码进行同义转换, 那么 15 种代码同义转换算子编码为长度为 15 的一维向量, 每个变量取值为 0/1. 采用这种方法的直觉是, 某种非鲁棒性代码特征可能会对漏洞预测模型起到关键的作用, 使用同一种代码同义转换算子对代码段进行转换可以提升非鲁棒性代码特征的影响力. 比如, 一组编码后的代码同义转换算子向量为: [1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0], 表明对源代码执行的同义代码转换操作包括: Op1-ChRename、Op2-ChFor、Op4-ChDo、Op5-ChIfElseIF、Op7-ChSwitch、Op8-ChRelation、Op9-ChUnary、Op11-ChConstant.

3.3 代码同义转换算子的搜索策略问题

代码同义转换算子定义的成分空间是巨大的[30]. 例如, 所有有效的标识符名称都可以成为标识符重命名转换的成分. 而且, 随着代码转换算子的增加, 搜索空间大小呈现指数级变化, 以采用的同义转换算子编码为长度为 15 的一维向量为例, 代码转换算子搜索空间大小为 2^{15}. 在有限的搜索时间内搜索预测为无漏洞的对抗代码段是具有挑战性的. Zhang 等人[30]采用强化学习来构建对抗样本, 他们提出的 DRLSG 对抗攻击策略是其他搜索策略 (包括随机策略、基于演化算法的策略等) 的时间的 8–191 倍.

遗传算法受自然选择过程的启发, 迭代地将候选解集合演化为更好的解. 每次迭代的候选解集合被称为一个代. 在每一代中, 使用一个适应度函数评估候选解的质量. 较“适应”的解更有可能被选中繁衍下一代. 下一代通过交叉和变异的组合生成. 交叉是从多个父代解中产生一个子代解的过程; 它类似于生物的繁殖和交叉. 变异的目的是增加候选解集合的多样性, 并提供对搜索空间的更好探索. 遗传算法在解决组合优化问题方面表现良好[35], 并且由于采用了候选解集合, 这些算法可以在更少的修改下找到成功的对抗性示例[36], 我们提出基于遗传算法的代码同义转换算子的搜索策略.

4 NonVulGen 对抗攻击方法

本文设计了一种基于遗传算法的对抗攻击方法 NonVulGen, 图 2 展示了 NonVulGen 的整体架构. NonVulGen 包括 3 个阶段: 基于贪婪搜索创建种子染色体阶段、基于遗传算法进行组合优化阶段和针对特定代码段的深度搜索阶段. 该攻击方法分为如下 3 个步骤: 第 1 步基于贪婪搜索创建种子染色体, 针对待攻击代码段采用随机采样的策略生成若干代码同义转换算子向量, 在代码同义转换算子向量空间进行贪婪搜索. 初始化一个染色体家族. 按照适应度对该染色体组进行排序, 选择适应度最高的 k 个染色体; 第 2 步是以该组染色体为基础, 使用遗传算法进行组合优化, 找到适应度最佳的染色体; 第 3 步是, 针对对抗攻击失败的代码段选择更多的自适应扰动算子采用深度对抗攻击.

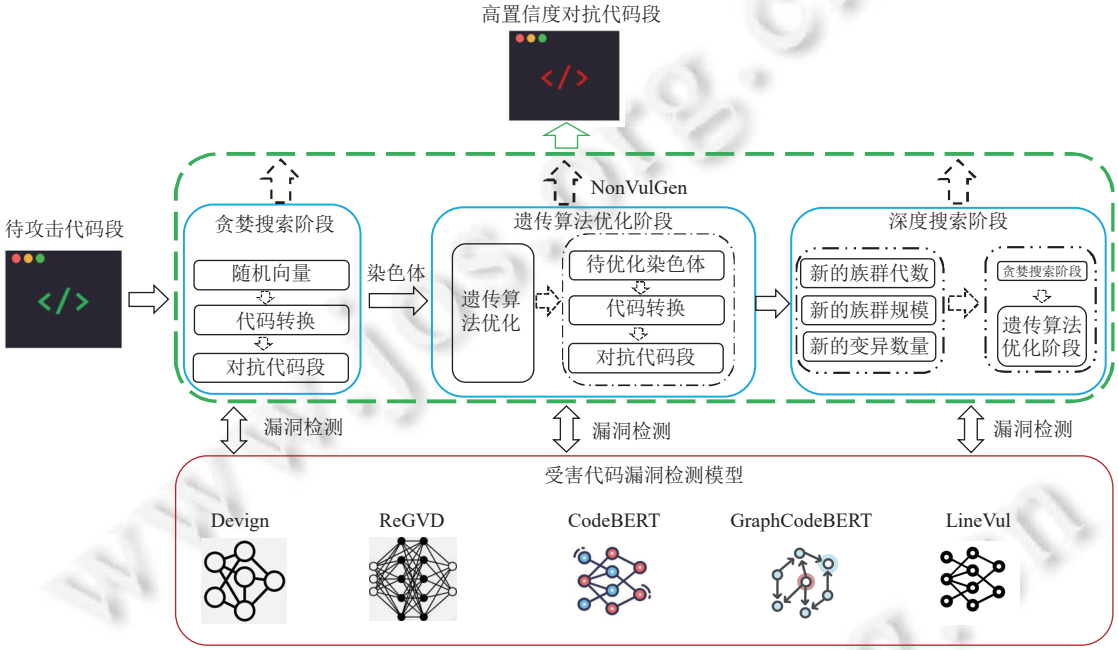

图 2 NonVulGen 整体架构

4.1 基于贪婪搜索创建种子染色体

NonVulGen 采用随机采样的策略生成若干代码同义转换算子向量, 算子向量搜索过程如算法 2 所示. 采用随机策略生成一族染色体是为了最大限度地从代码同义转换算子空间中进行采样, 减少搜索策略陷入局部最优解的问题. 接下来计算每个染色体所对应的适应度值, 选择适应度值最高的 K 个染色体作为种子染色体. 该适应度值是以代码漏洞检测模型返回的置信度值来表示, 将模型预测为无漏洞代码段的置信度作为适应度值, 从而能够指导 NonVulGen 从代码同义转换算子空间中选择满足要求的算子向量.

算法 2. 基于贪婪搜索创建种子染色体.

Input: *code*: input a code segment;

Output: *top_k_chromosomes*: *TopK* seed chromosomes.

1. *MAX* ← maximum number of iterations
2. *P* ← number of positions with random changes in the vector

3. K ← number of top chromosomes to select
4. *chromosomes* ← empty list
5. *count* ← 0
6. **while** *count* < *MAX* **do**
7. *rand_vector* ← random vector of length 15 with P random changes
8. *transformation* ← convert *rand_vector* into corresponding code transformation operator
9. *transformed_code* ← apply *transformation* on input code segment
10. *confidence* ← calculate the confidence of *transformed_code*
11. *chromosomes.append*(*rand_vector*, *confidence*)
12. *count* ← *count* + 1
13. **end while**
14. Sort *chromosomes* in descending order based on *confidence*
15. *top_k_chromosomes* ← select *TopK* chromosomes from the sorted list
16. **return** *top_k_chromosomes*

4.1.1 染色体表示

在遗传算法中, 染色体代表着解决目标问题的解, 而染色体由一组基因组成. NonVulGen 中将染色体表示为一组源代码同义转换算子. 例如, 以来自开放漏洞探测数据集项目 qemu 的有漏洞代码段为例, 该代码段如代码 1 所示.

代码 1. 漏洞探测数据集项目 qemu 的有漏洞代码段 mirror.

```
1. static inline int mirror(int v, int m)
2. {
3.    if(v<0)
4. return –v;
5.    else if(v>m)
6.            return 2×m–v;
7.         else
8.            return v;
9. }
```

4.1.2 族群初始化

在遗传算法中, 需要初始化一个种群, 基于该种群演化出满足适应度值的代码同义转换算子向量. 采用随机采样的策略生成代码同义转换算子向量, 比如, 当前生成的随机向量为: [1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0], 则按照代码同义转换算子编码规则, 转换生成的代码对应的代码段如代码 2 所示.

代码 2. 有漏洞代码段 mirror 被对抗攻击后的代码段, 函数名转换为 mohror.

```
1. static inline int mohror(int v_241, int m_9)
2. {
3.    if((659–659)>v_241)
4.       return –v_241;
5.    else {
6.       if(v_241 > m_9)
```

```
7.            return (180–178)×m_9–v_241;
8.      else
9.            return v_241;
10.   }
11. }
```

经过计算, 该代码段对应的适应度值: 0.741 75. 重复随机采样 *MAX* 次数, 则可以初始化一个原始种群, 对该种群中的每个生成代码段计算适应度值, 并按照降序排列, 取该种群中 *K* 个代码段作为遗传算法的种子种群, 该种群的大小为 *K*.

4.2 基于遗传算法进行组合优化

寻找一组代码同义转换算子生成对抗性样本是一个组合优化问题, 其目标是同义代码转换算子的最佳组合, 以最小化受害模型对含有漏洞代码段的置信度. 在第 4.1 节中, 采用贪婪搜索的方式找到一组代码同义转换算子向量, 然而这种方式仍然存在陷入局部最优解的问题. 如果贪婪攻击无法找到一组代码同义转换算子以生成对抗性样本, 我们将应用遗传算法对第 1 步生成的种子染色体进行演化计算. 遗传算法偏向于寻找代码语义相同而结构变化较大的代码同义转换向量[37–39].

算法 3 展示了基于遗传算法的对抗攻击算法的工作概述. 首先使用算法 2 生成的种子染色体来初始化种群 (第 4 行), 然后执行遗传操作生成新的解决方案 (第 5–13 行). 计算每个新染色体的适应度值 (第 14–19 行) 并保留具有较大适应度值的解决方案 (第 23 行). 最后, 算法返回具有最高适应度值的解决方案 (第 26 行).

算法 3. 基于遗传算法的对抗攻击.

Input: *top_k_chromosomes*: *TopK* seed chromosomes, *child_size*: number of generated children in each iteration;
Output: *code_adv*: adversarial code segment.

```
1. num_generations ← number of generations
2. population_size ← size of the population
3. mutation_rate ← rate of mutation
4. population ← top_k_chromosomes
5. for i ← 1 to num_generations do
6.    new_population ← empty list
7.    for j ← 1 to population_size/2 do
8.       parent _1, parent _2 ← select a chromosome from the population using roulette wheel selection based on fitness
9.       child_1, child_2 ← crossover on parent _1 and parent _2
10.      child_1, child_2 ← mutation on child_1, child_2 with probability mutation_rate
11.      new_population.append(child_1)
12.      new_population.append(child_2)
13.   end for
14.   for j ← 1 to population_size do
15.      chromosome ← new_population [j]
16.      adversarial_code ← generate adversarial code segment based on chromosome
17.      chromosome.fitness ← evaluate fitness of adversarial_code
18.      new_population.append(chromosome)
19.   end for
```

20. *population* ← *new_population*
21. **if** any *chromosome* in *population* has fitness greater than 0.5 **then**
22. **return** *adversarial_code* with highest fitness
23. **end if**
24. **end for**
25. **return** *adversarial_code* with highest fitness

4.2.1 演化算子

基于遗传算法的演化过程分为多个迭代, 在每个迭代中, 每次都要从种子染色体集合中取出两个染色体, 依据各个染色体的适应度值采用轮盘的方式选择两个染色体, 保证最优秀的染色体有较大的概率被选中, 也就是说适应度值高的代码同义转换算子向量被选中的概率较高 (算法 3 第 8 行). 使用交叉算子对两个染色体演化生成新的染色体 (即子代, 第 9 行). 我们以 *mutation_rate* 的概率应用变异 (第 10 行). 交叉与变异算子在两个染色体 *parent*_1, *parent*_2 上的工作原理如下: 首先随机选择一个切割位置 *h*, 然后将 *parent*_1 在位置 *h* 之后的基因替换为 *parent*_2 相应位置的基因. 例如, 两个染色体 *parent*_1=[1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0], *parent*_2=[0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0]. 切割位置 *h*=2, 交叉生成的子代为 *child*_1=[0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0] 和 *child*_2=[1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0].

给定种群中的染色体, 变异算子随机选择一个基因, 然后对该基因进行变异, 即按位取反. 例如, 在 *child*_1 中, 随机选择的基因位置为 0, 则变异后的基因为 1–0=1, 染色体演化为 *child*_1=[1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0].

4.2.2 对抗代码段生成以及适应度函数

对当前的新的种群 *new_population*, 将每个染色体转换为对应的一组同义代码转换算子, 对输入代码段上进行计算, 获得对应的对抗代码段 (第 16 行). 将该代码段输入受害漏洞检测模型, 将预测值为无漏洞类别的置信度作为当前染色体的适应度值 (第 17 行). 如果当前的种群 *new_population* 中存在适应度值大于阈值 (0.5) 的染色体, 则终止该演化算法 (第 21 行).

4.3 针对特定代码段的深度搜索

在基于黑盒的代码段对抗攻击过程中, 为了减少查询受害漏洞探测模型, 只有部分代码同义转换算子被使用对输入代码进行转换, 在本文实验中, 该经验值设置为 8. 也就意味着, 在创建种子染色体阶段, 只有部分基因被设定为有效基因. 然而, 存在某些代码段本身不含有被选择的有效基因的情况, 需要针对性地对这些代码段进行深度搜索. 深度搜索过程如算法 4 所示. 该深度搜索过程思路是简单的, 就是通过在更大的代码同义转换向量空间中进行搜索并对更多的基因进行变化, 以获取更加具有扰动性的对抗样本 (第 1–3 行). 把这些种子染色体通过贪婪搜索 (算法 2) 与基于遗传算法的组合优化 (算法 3) 来生成对抗代码段.

算法 4. 针对特定代码段的深度搜索.

Input: *code*: code segment;
Output: *code_adv*: adversarial code segment.

1. *num_generations* ← number of generations
2. *population_size* ← size of the population
3. *mutation_count* ← count of mutation gene
4. Create seed chromosomes based on greedy search in 算法 2
5. *adversarial_code* ← evolutionary algorithm-based attack in 算法 3
6. **return** *adversarial_code* with highest fitness

5 实验设计

5.1 数据集

本节介绍了实验中使用的数据集. 我们使用了 Devign[20]和 MSR[40]数据集. Steenhoek 等人[41]研究了 11 个开源漏洞检测模型中的数据集, 他们发现 Devign 数据集已经在 11 个模型中的 8 个中进行了评估. 这是一个平衡的数据集, 包括大约相同数量的易受攻击和非易受攻击的例子, 共 27 318 个样本, 每个函数代码段被标记为包含漏洞或干净. 该数据集是从两个流行的开源 C 项目中提取的: FFmpeg (https://www.ffmpeg.org/) 和 Qemu (https://sites.google.com/view/devign). 这个数据集作为 CodeXGLUE 基准测试的一部分[5], 已经被用来研究 CodeBERT 在漏洞预测方面的有效性. LineVul 使用了 MSR 数据集, 这是一个最近可用的数据集. 它是一个不平衡的数据集, 包括 10 900 个易受攻击的样本和 177 736 个非易受攻击的样本. 这些样本标有它们的源项目和它们的公共弱点枚举 (CWE) 条目, 这表明了漏洞的类型.

我们选择这两个数据集, 原因如下: (1) 这两个数据集都包含了真实世界的项目和漏洞; (2) 大多数模型在其论文中都是使用 Devign 数据集进行评估和调优的; (3) MSR 数据集包含 310 个项目, 其数据对漏洞类型有注释.

5.2 目标模型

为了收集 SOTA 深度学习模型, Steenhoek 等人[41]研究了 2018–2022 年的论文, 并使用了微软的 CodeXGLUE 排行榜 (https://microsoft.github.io/CodeXGLUE) 和 IBM 的漏洞检测 D2A 排行榜 (https://ibm.github.io/D2A). 他们处理了所有能找到的开源模型, 并成功复现了 11 个模型 (https://doi.org/10.6084/m9.figshare.20791240). 我们选择了多个有代表性的漏洞检测模型作为我们的目标模型. 目标模型的统计数据在表 2 中呈现.

表 2 漏洞检测目标模型的统计信息

模型	年份	架构	数据集
Devign	2019	GNN, property graph	Devign
ReGVD	2022	GNN, token	Devign
CodeBERT	2020	Transformer	Devign
GraphCodeBERT	2021	Transformer	Devign
LineVul	2022	Transformer	MSR

5.2.1 Devign

Devign[20]是一个基于 GNN 的源代码漏洞检测模型, 使用 GNN 学习丰富的代码语义信息, 该模型包括一个卷积模块, 其功能是提取有用的特征来进行 graph-level 的分类. Devign 在属性图上使用了 GNN, 集成了控制流、数据依赖和抽象语法树.

5.2.2 ReGVD

受预训练编程语言模型 (如 CodeBERT 和图神经网络) 的成功应用的启发, Nguyen 等人[22]提出了 ReGVD, 这是一种通用新颖的基于图神经网络的漏洞检测模型. ReGVD 使用 GNN 对源代码进行处理. ReGVD 是一个通用、简单而又有效的基于图神经网络的漏洞检测模型.

5.2.3 CodeBERT

CodeBERT[18]是一个预训练模型, 能够从编程语言和自然语言的双模态数据中进行学习. 在下游漏洞检测任务中, Yang 等人[10]在微调 CodeBERT 时采用了 CodeXGLUE[5]中相同的参数设置, 只是增加了最大输入长度到 512.

5.2.4 GraphCodeBERT

GraphCodeBERT 是一种考虑程序的固有结构并利用数据流表示的预训练模型. Yang 等人[10]将 GraphCodeBERT 的最大输入长度设置为 512, 并对 GraphCodeBERT 论文中的其他超参数采用相同的设置, 以便在 3 个漏洞检测任务上进行微调.

5.2.5 LineVul

Fu 等人[1]提出了 LineVul, 一种基于 Transformer 的函数级漏洞预测方法, 以解决 IVDetect 方法仍然不准确粗

粒度的问题.

我们在实现这些目标模型的时候使用了开源的漏洞检测模型[41], 相应论文中的原始数据集和设置被用于重现这些目标模型. 模型重现的结果如表 3 所示. 再现结果报告为 3 个随机种子的平均值, "—"表示该指标的结果在他们的论文中没有报告, 数字以百分比表示. *A*, *P*, *R* 和 *F* 的列代表深度漏洞检测模型中常用的指标, 包括精度 (*A*), 准确性 (*P*), 召回率 (*R*) 和 *F*1 (*F*). 这些复现结果与原始论文相比存在 2% 的差异. 例外的情况是 Devign, Steenhoek 等人[41]使用了由 Chakaborthy 等人[21]发布的第三方重现版本 (https://github.com/saikat107/Devign), 因为原始的 Devign 代码不是开源的. 我们得到的结果与他们原始论文和另一篇最近的论文[5,18,19]报道的结果非常接近, 这也突显出本文实验中所使用的受害者模型经过了充分的微调. 另外, 在本文研究中, 我们使用了 Yang 等人[10]在漏洞检测任务上对 CodeBERT 和 GraphCodeBERT 微调的受害者模型 (https://github.com/soarsmu/attack-pretrain-models-of-code).

表 3 模型在它们原始数据集上的再现 (%)

模型	论文结果				本文再现结果			
	A	*P*	*R*	*F*	*A*	*P*	*R*	*F*
Devign	59	54	62	57	56	50	71	59
ReGVD	63	—	—	—	62	62	46	52
CodeBERT	62	—	—	—	64	59	54	55
GraphCodeBERT	61	—	—	—	64	59	59	58
LineVul	—	97	89	91	99	96	88	92

5.3 针对深度漏洞检测模型的对抗攻击基线

在本文实验中, 我们选择两种对抗攻击策略作为我们的基线, 与本文提出的 NonVulGen 攻击方法进行对比.

Random: 随机搜索策略使用随机值 0 或 1 填充位向量中的位, 概率相等. 采用随机搜索策略生成的对抗代码段表示实施的是完全随机地对漏洞检测模型进行黑盒攻击, 生成的代码同义转换向量具有随机性[42,43]. 这就使得每种代码同义转换算子都有同样的转换机会. 同时, 为了模仿攻击者的行为, 随机生成若干个随机对抗代码段, 并选择置信度值最高的代码段作为 Random 策略的攻击结果.

ALERT: Zhang 等人[7]首先提出一种针对代码模型的黑盒攻击. 他们提出了一种名为 Metropolis-Hastings 修改器 (MHM) 的 Metropolis-Hastings 采样技术, 用于生成专门用于源代码处理的深度学习模型的对抗样本. Yang 等人[10]提出 ALERT, 在数据集 Devign[20]进行的实验证明了 ALERT 的有效性, 其生成的示例的自然性, 以及在对抗性微调后增强模型鲁棒性的能力均要优于 MHM[7]. 因此, 我们将 ALERT[10]视为本文的基准线, Yang 等人共享了他们的源代码和实验数据, 我们使用与作者完全一样的超参数.

5.4 我们的攻击方法设置

NonVulGen 有一些需要设置的超参数. 在贪婪搜索阶段中的算法 2 中, 贪婪搜索的最大迭代次数 *MAX* 设置为 40, 表明需要最多从代码同义转换算子向量空间中最多抽样次数为 *MAX*. 在算法 3 中, 我们将 *num_generations* 设置为 20, 表明遗传算法演化的代数为 *num_generations*, 超参数 *population_size* 设置为 20, *mutation_count* 设置为 8. 而为了对特定代码进行更大范围的扰动, 在针对特定代码段的深度搜索阶段, 算法 4 中的超参数 *population_size* 设置为 40, *mutation_count* 根据代码段中的结构特征动态设置, 最小值为 8, 最大值为 15, 当设置为 15 的时候, 则意味着对代码段进行 15 种不同的代码扰动.

6 实验结果与分析

在本节中, 我们通过实证研究来回答与对抗攻击相关的研究问题. 实证研究聚焦于攻击成功率、攻击方法的泛化能力、不同阶段在对抗攻击的时候有什么样的重要性以及如何进行有效的防御等问题, 这些问题分别通过回答 4 个研究问题来讨论.

RQ1. 与最先进的攻击方法 ALERT 相比, 本文的攻击方法是否能取得更好的攻击性能?

在本节中, 我们回答 RQ1, 即 NonVulGen 对绕过漏洞检测模型的有效性. 为了达到这个目标, 我们将原始的测试数据集作为两个预训练漏洞检测模型的目标数据集. 然后, 我们针对测试数据集使用两种不同的策略, 即 ALERT 和 NonVulGen, 对这两个基于预训练的漏洞检测模型进行攻击. 为了回答这个问题, 我们针对漏洞检测任务的特点, 提出了评估攻击方法的评价指标. 采用攻击成功率与查询次数来评估本文方法. 这两个度量标准, 在不同的维度上衡量对抗性示例生成方法的性能.

● 攻击成功率 (attack success rate, *ASR*): 我们引入 Li 等人[44]在针对代码模型的投毒攻击上使用的“攻击成功率”作为我们的攻击指标. 在漏洞检测任务中, 通过黑盒攻击希望能把有漏洞的代码段通过扰动以后预测为无漏洞的代码段. 给定一个干净的漏洞检测模型的测试集, 首先获得所有被毒害模型正确预测为有漏洞的样本 $|\mathbb{C}_{\text{non-target}}|$. 在 $|\mathbb{C}_{\text{non-target}}|$ 中的所有样本被黑盒攻击以后, 我们测试这些改变后的样本, 并得到被预测为无漏洞的样本 $\mathbb{C}_{\text{flipped}}$. 因此, *ASR* 可以计算为:

$$ASR = \frac{|\mathbb{C}_{\text{flipped}}|}{|\mathbb{C}_{\text{non-target}}|},$$

其中, $|\cdot|$表示集合的样本数量. 这个攻击成功率与 Yang 等人[10]所采用的评价指标略有不同. 我们考虑的是测试数据集中, 所有被毒害模型正确预测为有漏洞的样本 $\mathbb{C}_{\text{non-target}}$, 因为有漏洞的代码段被成功攻击更加有现实意义, 被成功攻击的漏洞代码段如果成功逃逸了漏洞检测模型检测, 将会对软件系统安全带来重大的隐患.

● 查询次数 (number of queries, *NoQ*): 在对抗攻击中, 尤其是黑盒攻击, 需要尽量减少对受害模型的查询次数. 在实践中, 受害模型通常是远程部署的, 过多地查询模型会很昂贵, 也可能引起怀疑. 计算每次攻击在数据集 $\mathbb{C}_{\text{flipped}}$ 上生成对抗样本时对受害模型的平均查询次数 (*NoQ*), *Query_times* 用于表示黑盒攻击过程中总的查询次数. 该评价指标与 Yang 等人[10]使用的评价指标保持一致.

$$NoQ = \frac{|Query\_times|}{|\mathbb{C}_{\text{flipped}}|}.$$

与基线方法 ALERT 相比, 本文提出的 NonVulGen 攻击成功率如表 4 所示 (括号中的数字表示多次实验的标准偏差 (standard deviation)), 平均攻击次数如表 5 所示 (括号中的数字表示相对于 ALERT 查询数量的绝对改进). 就攻击成功率而言, NonVulGen 在 CodeBERT 上比 ALERT 高出 28.94%, 在 GraphCodeBERT 上也高出 15.52%. 除了达到更高的攻击成功率外, 本文方法还能减少查询次数, NonVulGen 在 CodeBERT 上比 ALERT 减少查询 92.07%, 在 GraphCodeBERT 上减少 91.74%.

表 4 CodeBERT 和 GraphCodeBERT 的攻击成功率 (*ASR*) 在漏洞检测攻击中的结果比较

模型	Random	ALERT	NonVulGen
CodeBERT	48.84%	70.87%	**91.38% (±0.03)**
GraphCodeBERT	52.53%	81.07%	**93.65% (±0.05)**

表 5 攻击 CodeBERT 和 GraphCodeBERT 在漏洞检测上的查询数量 (*NoQ*) 比较结果

模型	ALERT	NonVulGen
CodeBERT	566.8	**44.9 (−92.07%)**
GraphCodeBERT	478.5	**39.5 (−91.74%)**

NonVulGen 方法相较于最先进的攻击方法 ALERT 显示出了更好的攻击性能, 鲁棒性攻击性能大幅度提升的原因如下.

1) 多样化的代码扰动策略: NonVulGen 采用了多种代码同义转换算子, 这些算子在组合使用时能产生更为多样和复杂的对抗样本, 增加了攻击成功的概率.

2) 遗传算法的优化能力: 通过遗传算法进行组合优化, NonVulGen 能够选择出适应度最高的算子组合, 这一过程能够有效地识别和利用代码中的脆弱点, 从而提高攻击性能.

3) 攻击隐蔽性: NonVulGen 在生成对抗样本时考虑了代码的自然性和可读性, 这种隐蔽性使得攻击样本更难被检测模型识别, 从而提高了攻击成功率.

RQ1 的答案: 针对两种预训练模型: CodeBERT 以及 GraphCodeBERT 的对抗攻击, 从攻击成功率来讲, 本文

提出的攻击方法 NonVulGen 比基线方法 ALERT 有了较大的提升; 从对抗攻击过程中的查询次数来讲, 本文提出的攻击方法 NonVulGen 比基线方法 ALERT 大为减少.

RQ2. 我们提出的对抗攻击方法是否具有泛化能力?

具有较强的泛化攻击能力是检验漏洞检测模型对抗攻击方法的关键指标. 除了能够对预训练模型 CodeBERT 和 GraphCodeBERT 进行对抗攻击以外, 我们能否对其他漏洞检测模型进行对抗攻击. 我们选择了 Devign[20], ReGVD[22]以及 LineVul[1]作为我们的目标模型. 在第 5.2 节中, 对 3 种模型已经做了简要的介绍.

在数据集上的实验结果如表 6 所示 (括号中的数字表示多次实验的标准偏差 (standard deviation)). 本文提出的对抗攻击方法 NonVulGen 在漏洞检测模型 Devign, ReGVD 以及 LineVul 上的攻击成功率比随机搜索分别高出 87.10%, 78.27% 与 73.06%.

表 6　在模型 Devign、ReGVD 和 LineVul 上的攻击成功率比较

模型	Random	NonVulGen
Devign	48.84%	**98.88% (±0.01)**
ReGVD	52.53%	**97.85% (±0.03)**
LineVul	53.49%	**92.57% (±0.02)**

NonVulGen 在对抗不同的漏洞检测模型时, 均展示了较强的泛化攻击能力, 有效提升泛化攻击能力的因素如下.

1) 广泛的代码覆盖: NonVulGen 生成的对抗样本涵盖了多种代码结构和风格, 通过多样的同义转换算子, 能够有效针对不同模型的特性进行攻击.

2) 模型无关性: NonVulGen 在生成对抗样本时不依赖于特定模型的内部结构, 而是通过优化算子组合来提升攻击效果. 这种方法具有较强的通用性, 能够对不同的深度学习模型产生有效攻击.

3) 基于遗传算法的适应性: 遗传算法具有较强的适应性和进化能力, 能够在不同环境下不断优化对抗样本, 使其对多种模型都能保持高效攻击.

RQ2 的答案: 实证研究的结果表明, 我们提出的对抗攻击方法 NonVulGen 具有较强的泛化攻击能力.

RQ3. NonVulGen 的不同组件对于对抗攻击的影响如何?

我们要回答每种组件对对抗攻击影响. 将 NonVulGen 划分为 3 个阶段, 第 1 个阶段进行贪婪搜索, 简称为 Greedy, 第 2 个阶段使用遗传算法进行组合优化, 简称为 GA, 第 3 个阶段采用深度搜索去生成对抗代码段, 简称为 DeepSearch.

我们进行了消融实验, 实验结果如表 7 所示. 每一行都代表了采用漏洞检测模型成功生成对抗代码段的个数. 括号中的数字表示在下个阶段比 Greedy 阶段多生成的对抗代码段. 以 Devign 模型为例, 在 Greedy 阶段, 生成对抗样例为 428 个; 使用遗传算法组合优化以后, 另外生成对抗样例为 9 个; 第 3 阶段通过深度对抗搜索生成对抗样例 4 个.

表 7　NonVulGen 的消融实验分析

攻击方案	Devign	ReGVD	CodeBERT	GraphCodeBERT	LineVul
Greedy	428	598	404	526	876
Greedy+GA	437 (+9)	609 (+11)	414 (+10)	538 (+12)	889 (+13)
Greedy+GA+DeepSearch	441 (+4)	618 (+9)	439 (+25)	559 (+21)	911 (+12)

在 GA 阶段, 比 Greedy 阶段生成了更多的对抗样本, 对不同的漏洞检测模型: Devign, ReGVD, CodeBERT, GraphCodeBERT, LineVul, 提升比例分别为: 2.10%, 1.84%, 2.48%, 2.28%, 1.48%; 在 DeepSearch 阶段, 比 Greedy 阶段进一步生成了更多的对抗样本, 对不同的漏洞检测模型: Devign, ReGVD, CodeBERT, GraphCodeBERT, LineVul, 提升比例分别为: 3.04%, 3.34%, 8.66%, 6.27%, 3.99%.

RQ3 的答案: 基于贪婪搜索, 使用代码重命名以及代码段结构扰动的方式能生成有效的对抗代码段; 采用贪婪搜索无法获取对抗攻击代码段的情况下, 采用遗传算法进行组合优化以及深度搜索能进一步提升对抗攻击的成

功比例.

RQ4. 基于对抗样本重训练的模型能够抵挡我们方法的攻击吗?

在本节中, 我们试图使用实验结果来回答 RQ4, 具体探讨通过使用对抗样本增强漏洞检测模型的训练是否可以对抗 NonVulGen. 对于 RQ1 和 RQ2 的回答突出了漏洞检测模型的鲁棒性问题. 加强这些模型的一个常见方法是将矛融入盾牌, 这意味着我们可以使用 NonVulGen 中的对抗性代码段来重新训练漏洞检测模型.

Yang 等人[10]提出使用对抗样本对漏洞检测模型进行重训练作为一种防御攻击的手段, ALERT 生成的对抗性示例对提高受害模型的鲁棒性非常有价值. 使用 ALERT 生成的对抗性示例对受害模型进行对抗微调, 可以分别将 CodeBERT 和 GraphCodeBERT 的准确度提高 87.59% 和 92.32%. 他们利用变量重命名的方式生成每个受害模型在其相应的训练集上的对抗性示例, 将这些生成的对抗性示例添加到原始训练集中, 形成“对抗性训练集”. 然后, 对受害模型在对抗性训练集上进行微调. 我们也采用同样的方法, 在原始训练数据集上生成对抗代码段, 分别得到 5 个微调的受害模型: CodeBERT-Adv, GraphCodeBERT-Adv, Devign-Adv, ReGVD-Adv 以及 LineVul-Adv. 表 8 显示了对抗攻击方法对被加固的漏洞检测模型的攻击成功率. 使用基线方法和本文方法 NonVulGen 在训练数据集上生成对抗性示例, 微调漏洞检测模型, 然后验证在测试数据集上攻击成功率, 较低的攻击成功率以粗体显示. 我们选择攻击成功率来衡量对抗训练后的模型的攻击能力, 这个指标能够更加直观地反映对抗训练是否有效.

表 8　使用基线方法和本文方法 NonVulGen 在测试数据集上的攻击成功率对比 (%)

方法	Devign	ReGVD	CodeBERT	GraphCodeBERT	LineVul
Random	2.71	2.60	12.24	12.41	2.70
NonVulGen	**1.20**	**1.40**	**2.90**	**2.70**	**1.90**

从表 8 中, 针对被加固后的漏洞检测模型的攻击成功率都大幅度下降, 这样说明了对漏洞预测模型进行对抗训练是有效的. 在预训练模型 CodeBERT 和 GraphCodeBERT 上, 我们观察到本文提出的方法 NonVulGen 比 ALERT 具有更低的攻击成功率, 这可以推断出本文的攻击方法 NonVulGen 生成了更有代表性的非鲁棒性特征. 经过对抗训练以后, 这些非鲁棒性特征被漏洞检测模型学到, 因此 NonVulGen 在对测试数据集进行攻击的时候, 导致攻击成功率大幅度下降. 在 Devign 模型, ReGVD 模型与 LineVul 模型上, 我们看到采用随机搜索策略与 NonVulGen 方法攻击成功率都大幅度下降, 而两种策略的攻击成功率大致一致. 那么, 为什么基于训练数据集上的对抗样本对漏洞检测模型重训练师可以加固模型的? 在使用表 1 中的代码同义转换算子对源代码进行扰动的时候, 对抗模型学习到了这些扰动的特征. 以 Op13-ChAddJunk 为例, 扰动代码段“if(0) return 0;”表明当前的源代码是根据代码同义转换算子生成的. 这就为基于深度学习的漏洞检测模型从字符序列中学习到扰动特征提供了潜在的线索.

RQ4 的答案: NonVulGen 创建的对抗样本对提升目标模型的鲁棒性非常有用. 使用 NonVulGen 生成的对抗样本对受害模型进行对抗微调可以将 CodeBERT, GraphCodeBERT, Devign, ReGVD 和 LineVul 的鲁棒性分别提高 96.83%, 97.12%, 98.79%, 98.57% 和 97.94%.

7　有效性威胁

内部威胁: 我们的实验中需要对 5 种漏洞检测模型进行重训练, 我们保持重训练的参数与 Steenhoek 等人[41]重现的模型中使用的超参数一致. 另外本文提出的对抗攻击方法 NonVulGen 存在一些需要微调的超参数, 比如在贪婪搜索阶段, 创建的初始种子染色体的数目; 在基于遗传算法的组合优化阶段, 需要设定的参数包括演化的代数、变异的比例等超参数; 在深度搜索阶段, 需要设定的超参数包括深度搜索的种子数目等. 另外在对代码变量名进行对抗攻击的时候, 我们使用的是部分单词随机变化的方式来模拟程序员的 typo 错误或者基于大语言模型对变量名进行同义转换的方式, 还要考虑更多的满足变量名自然性转换的对抗攻击方法.

外部威胁: 目前我们在 5 种漏洞检测模型上进行了实证研究, 这 5 种模型在漏洞检测任务上拥有较强的代表性, 涵盖了图神经网络、预训练模型等. 本文攻击方法 NonVulGen 在其他漏洞检测模型上泛化性能还需要进一步去验证和研究.

8 结论以及未来工作

深度漏洞检测模型存在对抗攻击的问题, 对抗攻击成功率越高, 意味着有漏洞的代码段逃逸漏洞检测模型概率越大, 给软件质量保障带来安全隐患也就越高. 需要研究如何生成对抗样本提升漏洞检测模型鲁棒性. 漏洞检测模型对抗攻击样本要满足: 语法有效性、生成效率、自然性、攻击成功率等约束, 搜索非鲁棒性源代码特征是实施对抗攻击的关键. 为了提高深度漏洞检测模型的鲁棒性, 我们提出同时使用多种代码同义转换算子对源代码进行扰动, 将代码同义转换算子搜索问题转换为组合优化问题, 从这个角度出发, 我们设计了 NonVulGen, 该方法通过使用遗传算法生成一组具有高适应度值的代码同义变换操作算子, 创建对抗代码段. 我们对 5 个深度漏洞检测目标模型在两个被广泛采用的真实数据集上进行了实证研究. 结果表明, NonVulGen 比 SOTA 基线方法 ALERT 发现了更多的对抗样本并证实了其增强模型鲁棒性的能力; NonVulGen 在漏洞检测模型上有较强的泛化攻击能力; 基于对抗样本重训练的模型能够抵挡本文提出的对抗攻击方法 NonVulGen 的攻击.

未来将从如下几个方面改进 NonVulGen. 首先, 研究代码同义转换算子攻击的机理并选择更多的代码同义转换算子. 由于深度神经网络的黑盒特性, 引起深度漏洞检测模型的鲁棒性问题的代码特征需要被深入调研, 研究角度包括代码学习、漏洞检测模型可解释性分析等. 鲁棒性攻击的机理研究是确保软件质量保障的重要途径. 同时探索更多的代码同义转换算子, 提升对深度漏洞检测模型的攻击成功率, 提升重训练模型的鲁棒性. 其次, 随着越来越多深度漏洞检测模型的出现, 我们计划研究更多易受攻击的检测模型, 提高 NonVulGen 的攻击能力, 并进一步增强深度漏洞检测模型的强大性. 另外计划把研究对象从漏洞检测扩展到恶意软件检测等领域, 对基于深度学习的恶意软件检测模型进行广泛的深入的实证研究.

References:

[1] Fu M, Tantithamthavorn C. LineVul: A Transformer-based line-level vulnerability prediction. In: Proc. of the 19th IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR). Pittsburgh: IEEE, 2022. 608–620. [doi: 10.1145/3524842.3528452]

[2] Song ZH, Wang JF, Liu SL, Fang ZY, Yang KY. HGVul: A code vulnerability detection method based on heterogeneous source-level intermediate representation. Security and Communication Networks, 2022, 2022: 1919907. [doi: 10.1155/2022/1919907]

[3] Cao SC, Sun XB, Bo LL, Wu RX, Li B, Tao CQ. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering. Pittsburgh: IEEE, 2022. 1456–1468. [doi: 10.1145/3510003.3510219]

[4] Ding YRB, Suneja S, Zheng YH, Laredo J, Morari A, Kaiser G, Ray B. VELVET: A novel ensemble learning approach to automatically locate vulnerable statements. In: Proc. of the 2022 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Honolulu: IEEE, 2022. 959–970. [doi: 10.1109/SANER53432.2022.00114]

[5] 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. In: Proc. of the 35th Conf. on Neural Information Processing Systems. NeurIPS, 2021. 1–16.

[6] Zheng YH, Pujar S, Lewis B, Buratti L, Epstein E, Yang B, Laredo J, Morari A, Su Z. D2A: A dataset built for AI-based vulnerability detection methods using differential analysis. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Madrid: IEEE, 2021. 111–120. [doi: 10.1109/ICSE-SEIP52600.2021.00020]

[7] 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 34th AAAI Conf. on Artificial Intelligence. New York: AAAI Press, 2020. 1169–1176.

[8] Alon U, Zilberstein M, Levy O, Yahav E. code2vec: Learning distributed representations of code. Proc. of the ACM on Programming Languages, 3(POPL): 1–29. [doi: 10.1145/3290353]

[9] Alon U, Brody S, Levy O, Yahav E. code2seq: Generating sequences from structured representations of code. In: Proc. of the 7th Int'l Conf. on Learning Representations. New Orleans: ICLR, 2019. 1–22.

[10] 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. New York: ACM, 2022. 1482–1493. [doi: 10.1145/3510003.3510146]

[11] Srikant S, Liu SJ, Mitrovska T, Chang SY, Fan QF, Zhang GY, O'Reilly UM. Generating adversarial computer programs using optimized obfuscations. In: Proc. of the 9th Int'l Conf. on Learning Representations. Vienna: ICLR, 2021. 1–16.

[12] 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]

[13] 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 (ICST). Porto de Galinhas: IEEE, 2021. 36–46. [doi: 10.1109/ICST49551.2021.00016]

[14] 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 (SANER). Honolulu: IEEE, 2022. 526–537. [doi: 10.1109/SANER53432.2022.00070]

[15] Jha A, Reddy CK. CodeAttack: Code-based adversarial attacks for pre-trained programming language models. In: Proc. of the 37th AAAI Conf. on Artificial Intelligence. Washington: AAAI Press, 2023. 14892–14900. [doi: 10.1609/aaai.v37i12.26739]

[16] Zhang HZ, Fu ZY, Li G, Ma L, Zhao ZH, Yang HZ, 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]

[17] 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]

[18] 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: EMNLP 2020. ACL, 2020. 1536–1547. [doi: 10.18653/v1/2020.findings-emnlp.139]

[19] 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. In: Proc. of the 9th Int'l Conf. on Learning Representations. Vienna: ICLR, 2021. 1–18.

[20] 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. Vancouver: Curran Associates Inc., 2019. 915.

[21] Chakraborty S, Krishna R, Ding YRB, Ray B. Deep learning based vulnerability detection: Are we there yet? IEEE Trans. on Software Engineering, 2022, 48(9): 3280–3296. [doi: 10.1109/TSE.2021.3087402]

[22] Nguyen VA, Nguyen DQ, Nguyen V, Le T, Tran QH, Phung D. ReGVD: Revisiting graph neural networks for vulnerability detection. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering: Companion Proc. (ICSE-Companion). Pittsburgh: IEEE, 2022. 178–182. [doi: 10.1145/3510454.3516865]

[23] Hanif H, MaffeisS. VulBERTa: Simplified source code pre-training for vulnerability detection. In: Proc. of the 2022 Int'l Joint Conf. on Neural Networks (IJCNN). Padua: IEEE, 2022. 1–8. [doi: 10.1109/IJCNN55064.2022.9892280]

[24] Li Y, Wang SH, Nguyen TN. Vulnerability detection with fine-grained interpretations. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 292–303. [doi: 10.1145/3468264.3468597]

[25] Yang Z, Shi JK, Asyrofi MH, Lo D. Revisiting neuron coverage metrics and quality of deep neural networks. In: Proc. of the 2022 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Honolulu: IEEE, 2022. 408–419. [doi: 10.1109/SANER53432.2022.00056]

[26] Madry A, Makelov A, Schmidt L, Tsipras D, Vladu A. Towards deep learning models resistant to adversarial attacks. In: Proc. of the 6th Int'l Conf. on Learning Representations. Vancouver: ICLR, 2018. 1–23.

[27] Gleave A, Dennis M, Wild C, Kant N, Levine S, Russell S. Adversarial policies: Attacking deep reinforcement learning. In: Proc. of the 8th Int'l Conf. on Learning Representations. Addis Ababa: ICLR, 2020. 1–16.

[28] Carlini N, Wagner D. Audio adversarial examples: Targeted attacks on speech-to-text. In: Proc. of the 2018 IEEE Security and Privacy Workshops (SPW). San Francisco: IEEE, 2018. 1–7. [doi: 10.1109/SPW.2018.00009]

[29] Qu YB, Huang S, Chen X, Wang XY, Yao YM. Detection of backdoor attacks using targeted universal adversarial perturbations for deep neural networks. Journal of Systems and Software, 2024, 207: 111859. [doi: 10.1016/j.jss.2023.111859]

[30] 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]

[31] Casalnuovo C, Barr ET, Dash SK, Devanbu P, Morgan E. A theory of dual channel constraints. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering: New Ideas and Emerging Results. Seoul: IEEE, 2020. 25–28.

[32] Yang G, Zhou Y, Yang WH, Yue T, Chen X, Chen TL. How important are good method names in neural code generation? A model robustness perspective. ACM Trans. on Software Engineering and Methodology, 2023, 33(3): 60. [doi: 10.1145/3630010]

[33] Yang G, Zhou Y, Zhang XY, Chen X, Han TT, Chen TL. Assessing and improving syntactic adversarial robustness of pre-trained models

for code translation. arXiv:2310.18587, 2023.

[34] Belinkov Y, Bisk Y. Synthetic and natural noise both break neural machine translation. In: Proc. of the 6th Int'l Conf. on Learning Representations. Vancouver: ICLR, 2018. 1–13.

[35] Anderson EJ, Ferris MC. Genetic algorithms for combinatorial optimization: The assemble line balancing problem. ORSA Journal on Computing, 1994, 6(2): 161–173. [doi: 10.1287/ijoc.6.2.161]

[36] Alzantot M, Sharma Y, Elgohary A, Ho BJ, Srivastava M, Chang KW. Generating natural language adversarial examples. In: Proc. of the 2018 Conf. on Empirical Methods in Natural Language Processing. Brussels: ACL, 2018. 2890–2896. [doi: 10.18653/v1/D18-1316]

[37] Devore-McDonald B, Berger ED. Mossad: Defeating software plagiarism detection. Proc. of the ACM on Programming Languages, 2020, 4(OOPSLA): 138. [doi: 10.1145/3428206]

[38] Navarro G. A guided tour to approximate string matching. ACM Computing Surveys, 33(1): 31–88. [doi: 10.1145/375360.375365]

[39] Ren XL, Ho M, Ming J, Lei Y, Li L. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. ACM, 2021. 142–157. [doi: 10.1145/3453483.3454035]

[40] Fan JH, Li Y, Wang SH, Nguyen TN. A C/C++ code vulnerability dataset with code changes and CVE summaries. In: Proc. of the 17th IEEE/ACM Int'l Conf. on Mining Software Repositories. Seoul: IEEE, 2020. 508–512. [doi: 10.1145/3379597.3387501]

[41] Steenhoek B, Rahman MM, Jiles R, Le W. An empirical study of deep learning models for vulnerability detection. In: Proc. of the 45th Int'l Conf. on Software Engineering. Melbourne: IEEE Press, 2023. 2237–2248. [doi: 10.1109/ICSE48619.2023.00188]

[42] Andonie R, Florea AC. Weighted random search for CNN hyperparameter optimization. Int'l Journal of Computers Communications & Control, 2020, 15(2): 1–11. [doi: 10.15837/ijccc.2020.2.3868]

[43] Florea AC, Andonie R. Weighted random search for hyperparameter optimization. Int'l Journal of Computers Communications & Control, 2019, 14(2): 154–169.

[44] Li J, Li Z, Zhang HZ, Li G, Jin Z, Hu X, Xia X. Poison attack and poison detection on deep source code processing models. ACM Trans. on Software Engineering and Methodology, 2024, 33(3): 62. [doi: 10.1145/3630008]

曲豫宾(1981－), 男, 博士生, 主要研究领域为机器学习, 软件工程, 软件安全性, 软件测试与质量评估.

李龙(1989－), 男, 博士, CCF 高级会员, 主要研究领域为机器学习, 软件工程.

黄松(1970－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件工程, 软件安全性, 软件测试与质量评估.

王丹(2001－), 女, 硕士生, 主要研究领域为智能软件测试, 智能合约测试.

陈翔(1980－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为机器学习, 软件缺陷预测.

姚永明(1987－), 男, 博士, CCF 高级会员, 主要研究领域为机器学习, 软件工程.

王兴亚(1990－), 男, 博士, 副研究员, 主要研究领域为智能软件测试, 智能合约测试.

鞠小林(1978－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为机器学习, 软件缺陷预测.