软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007643] [CSTR: 32375.14.jos.007643]

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

面向预训练漏洞检测模型的黑盒对抗攻击*

王霄东, 陈郅楷, 王一杰, 黄 荔, 关志涛

(华北电力大学 控制与计算机工程学院, 北京 102206)

通信作者: 关志涛, E-mail: guan@ncepu.edu.cn

摘 要: 预训练代码模型在代码理解与分析任务中展现出强大的能力, 逐渐成为源代码漏洞检测领域的重要工具与研究热点. 然而, 与传统深度学习模型类似, 预训练代码模型在面对精心构造的对抗代码输入时仍存在鲁棒脆弱性. 攻击者可通过添加语义保持的扰动, 误导模型将含漏洞代码识别为无漏洞代码, 从而威胁软件安全. 因此, 研究针对预训练漏洞检测模型的对抗攻击, 不仅有助于评估预训练代码模型的鲁棒性, 还能够为后续漏洞检测模型设计和防御机制构建提供重要参考. 针对硬标签黑盒攻击场景, 提出一种面向预训练漏洞检测模型的黑盒对抗攻击方法——VulBlurrer. 设计定向同义代码转换策略优先对漏洞邻近的高敏感区域和特定语句进行扰动, 并提出基于特征一致性、语义一致性与代码流畅度的逃逸分数, 以在无需访问目标模型内部信息的前提下量化候选样本的潜在攻击价值. 此外, 采用基于遗传算法的对抗代码优化策略, 通过动态调整逃逸分数计算权重, 并在迭代过程中采用精英保留机制, 进一步提升攻击准确率. 在基于 CodeBERT、GraphCodeBERT、CodeT5 和 UniXcoder 的预训练漏洞检测模型上, 对 VulBlurrer 及基线方法进行性能测试. 结果显示, VulBlurrer 在 4 种目标模型上的攻击成功率分别达到 85.51%、91.47%、93.14% 和 71.61%, 平均查询次数分别为 12.67 次、9.10 次、11.07 次和 19.44 次. 与现有方法相比, VulBlurrer 具有更高的攻击成功率, 且在攻击成功率与查询效率之间实现了更好的权衡, 其生成的对抗代码在语义一致性与代码流畅度方面亦表现更优. 进一步地, 在 ChatGPT、DeepSeek 和基于大语言模型的辅助编程工具 GitHub Copilot、TRAE 上开展实证研究, 验证了 VulBlurrer 在大语言模型上的有效性. 因此, 预训练代码模型在漏洞检测任务中仍面临对抗攻击带来的鲁棒性挑战, 基于预训练模型和大语言模型的漏洞检测工具需要进一步提升面对对抗性代码时的鲁棒性.

关键词: 漏洞检测; 黑盒对抗攻击; 预训练模型; 大语言模型

中图法分类号: TP311

中文引用格式: 王霄东, 陈郅楷, 王一杰, 黄荔, 关志涛. 面向预训练漏洞检测模型的黑盒对抗攻击. 软件学报. http://www.jos.org.cn/1000-9825/7643.htm

英文引用格式: Wang XD, Chen ZK, Wang YJ, Huang L, Guan ZT. Black-box Adversarial Attacks for Pretrained Vulnerability Detection Models. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7643.htm

Black-box Adversarial Attacks for Pretrained Vulnerability Detection Models

WANG Xiao-Dong, CHEN Zhi-Kai, WANG Yi-Jie, HUANG Li, GUAN Zhi-Tao

(School of Control and Computer Engineering, North China Electric Power University, Beijing 102206, China)

Abstract: Pretrained code models have demonstrated strong capabilities in code understanding and analysis and have become important tools and major research focuses in source code vulnerability detection. However, similar to traditional deep learning models, pretrained code models exhibit robustness vulnerabilities when exposed to carefully crafted adversarial code. Attackers can mislead the model into classifying vulnerable code as non-vulnerable by introducing semantically preserving perturbations, thus posing a significant threat to software security. Therefore, adversarial attacks for pretrained vulnerability detection models not only serve as an effective approach to

* 基金项目: 智能电网国家科技重大专项 (2025ZD0808500); 国家自然科学基金 (62372173)
收稿时间: 2025-11-15; 修改时间: 2025-12-16; 采用时间: 2026-01-23; jos 在线出版时间: 2026-03-25

evaluate the robustness of pretrained code models but also provide critical insights for the development of future vulnerability detection models and defense mechanisms. In the hard-label black-box attack scenario, this study proposes VulBlurrer, a black-box adversarial attack method targeting pretrained vulnerability detection models. VulBlurrer designs a directed synonymous code transformation strategy, prioritizing perturbations on highly sensitive regions adjacent to vulnerabilities and on specific statements. It also introduces an escape score based on feature consistency, semantic consistency, and code fluency, enabling the quantification of the potential attack value of candidate samples without accessing internal information of the target model. Furthermore, this study adopts a genetic-algorithm-based optimization strategy for adversarial code, in which the weights used to compute the escape score are dynamically adjusted and an elite retention mechanism is applied during the iterative process, thus further improving attack accuracy. VulBlurrer and baseline methods are evaluated on pretrained vulnerability detection models based on CodeBERT, GraphCodeBERT, CodeT5, and UniXcoder. The attack success rates of VulBlurrer on the four target models reach 85.51%, 91.47%, 93.14%, and 71.61%, with average query numbers of 12.67, 9.10, 11.07, and 19.44, respectively. Compared with existing methods, VulBlurrer achieves higher attack success rates and a better trade-off between attack success rate and query efficiency, while the generated adversarial code also exhibit superior consistency and fluency. Furthermore, empirical studies are conducted on ChatGPT, DeepSeek, and LLM-based programming assistants, including GitHub Copilot and TRAE, verifying the effectiveness of the proposed method against large language models. These results indicate that pretrained code models still face robustness challenges posed by adversarial attacks in vulnerability detection tasks, and vulnerability detection tools based on pretrained models and large language models require further improvements to enhance robustness against adversarial code.

Key words: vulnerability detection; black-box adversarial attack; pretrained model; large language model (LLM)

1 引 言

随着软件系统功能的不断扩展, 代码规模呈现爆炸式增长, 源代码漏洞等软件安全问题也日益凸显. 同时, 开源社区的发展使源代码能够在全球范围内协作开发、共享和复用, 代码复用场景愈发普遍, 进一步增加了代码审查和漏洞管理的难度. 面对海量的源代码, 传统人工审查难以逐行确认漏洞的存在, 基于深度学习的源代码分析技术为漏洞检测提供了自动化手段. 深度学习模型能够从大量代码数据中自动学习潜在特征和模式, 较传统规则或特征工程方法具有更强的泛化能力和自动特征提取能力. 在此基础上, 预训练语言模型的出现进一步提升了源代码语义理解和分析的效率. 在代码领域, 预训练编程语言模型或预训练代码模型发展迅速. 一系列先进的预训练代码模型, 如 CodeBERT[1]、GraphCodeBERT[2]等, 在源代码漏洞检测任务中取得了显著进展.

然而, 深度学习模型在多个领域均表现出鲁棒脆弱性, 尤其容易受到对抗攻击的干扰[3−5]. 在计算机视觉领域[4], 攻击者仅需对图像像素进行细微扰动, 即可误导模型预测; 在自然语言处理领域[5], 对输入文本进行词汇替换或语序调整, 也可能导致模型产生错误输出. 这些研究结果表明, 即便是性能优异的深度学习模型, 在面对精心构造的对抗性输入时仍缺乏足够的鲁棒性.

已有研究证明, 预训练代码模型与基于深度学习的漏洞检测模型同样存在鲁棒性问题, 即容易受到对抗性样本的影响, 产生错误的输出结果[6]. 在源代码分析领域, 攻击者可以在保持语法正确和语义等价的前提下[7], 对源代码进行诸如变量重命名、语句顺序重排或垃圾代码注入等修改, 即添加扰动, 从而干扰模型的预测结果[8]. 因此, 攻击者通过精心构造注入了对抗性扰动的漏洞代码, 使其绕过基于预训练代码模型的漏洞检测器, 即欺骗漏洞检测器将漏洞代码标记为无漏洞代码, 最终使其被成功注入正在开发或维护的项目中. 考虑到开源社区协作开发与代码复用的特性, 对抗性代码极易通过开源社区进行传播, 从而产生重大危害. 因此, 针对基于预训练模型的漏洞检测器开展黑盒对抗攻击的深入研究, 不仅能为预训练代码模型的鲁棒性评估提供手段, 还能通过生成高质量对抗代码, 为后续对抗性训练[8]和防御机制设计提供支持.

当前, 针对基于深度学习的源代码漏洞检测器的黑盒对抗攻击, 从扰动策略选择上可分为两类: 基于搜索与优化的方法[9−11]和基于强化学习的方法[12,13]. 基于搜索与优化的方法在代码同义转换操作的离散搜索空间中, 通过采样、优化等策略寻找能够干扰目标模型预测的对抗代码. 基于强化学习的方法将对抗代码生成过程建模为序列决策问题, 结合目标模型反馈与语义一致性等要素设计奖励函数[12]并进行模型训练, 最后通过策略网络选择具体扰动操作以生成对抗代码. 尽管现有源代码对抗攻击方法已获得一定成果, 但仍存在语义一致性和查询效率的挑战.

(1) 对抗代码需保持执行功能与原始代码一致. 与图像或文本任务不同, 源代码具有严格的语法规则和语义约

束, 其对抗扰动受语法合法性与功能不变性的双重约束[11]. 生成的对抗代码不仅需要保持可编译性与可执行性, 还必须确保扰动前后代码的功能等价, 否则将丧失实际攻击意义. 然而, 现有方法在扰动策略设计上往往采用非定向的、全局性的代码扰动方式, 如在整个函数范围内进行同义变换. 这类粗粒度扰动容易引入大量非必要修改, 不仅增加代码修改量, 还可能提升语义被破坏的风险, 限制对抗代码的实际可用性.

(2) 对抗代码需保持文本风格与原始代码一致. 除了功能正确, 对抗性代码还必须在语义表达与代码风格上保持一致, 并保证文本自然性, 即对抗代码在语义表达、编码风格和逻辑连贯上需符合真实编程习惯. 然而, 添加扰动的过程中, 现有方法存在冗余代码注入过于生硬或变量命名明显偏离常规习惯等问题, 从而削弱对抗代码的文本自然性与隐蔽性. 例如, 为逃避检测而注入的冗余代码 (如 print(“No vulnerability”) 或 print(“No bugs”)) 更容易引起经验丰富的程序员与安全人员的注意.

(3) 以较少的查询次数生成有效的对抗代码. 在黑盒场景中, 攻击者通常无法直接访问模型参数或梯度, 只能依赖有限的查询反馈来迭代构造对抗. 尤其在面对基于预训练模型或大语言模型的漏洞检测器时, 频繁查询不仅会产生高昂的时间与经济成本, 还会增加被监测和溯源的风险. 因此, 对抗攻击方法需要在有限的查询开销下实现较高的攻击成功率, 否则难以在实际场景中生效.

针对上述挑战, 本文提出一种面向预训练漏洞检测模型的黑盒对抗攻击方法 VulBlurrer, 用于基于预训练模型的源代码漏洞检测器的鲁棒性测试. VulBlurrer 面向仅提供预测结果的硬标签黑盒攻击场景, 在保持语义一致性的前提下, 高效生成对抗代码. VulBlurrer 设计定向同义代码转换策略, 以减少代码改动量并提高攻击成功效率. 首先, 根据源代码的结构与漏洞位置特征, 优先在漏洞邻近的高敏感区域施加扰动, 而非直接对整个代码段进行同义转换操作; 其次, 优先对特定语句进行扰动, 如 if 语句、method 语句和 for 语句. 同时, VulBlurrer 提出综合性对抗代码评估指标——逃逸分数, 用于量化候选对抗代码的潜在攻击价值. 该分数由 3 部分构成, 特征一致性、语义一致性和代码流畅度. VulBlurrer 定义了特征一致性, 通过预训练模型的隐空间表征差异衡量对抗代码与原始代码在特征层面的偏移程度. VulBlurrer 在逃逸分数计算过程中无需访问目标模型, 从而减少对目标模型的查询次数. 此外, 在对抗代码优化阶段, 针对初始生成但未能成功绕过目标检测器的对抗代码, VulBlurrer 引入遗传算法进行迭代优化, 以提升攻击成功率. 不同于已有攻击方法依赖目标模型输出的置信度作为对抗代码的适应度来指导对抗代码的生成, VulBlurrer 以基于逃逸分数排序得到的高质量样本集作为初始种群, 从而支持在更严格的硬标签黑盒攻击场景下生成对抗代码. 本文以基于 CodeBERT、GraphCodeBERT、CodeT5 和 UniXcoder 微调的漏洞检测模型为目标模型, 对这些目标模型进行 NonVulGen 攻击[11]、CodeTAE 攻击[14]和 VulBlurrer 攻击. 此外, 本文还在流行的大语言模型 ChatGPT、DeepSeek 和基于大语言模型的辅助编程工具 GitHub Copilot、TRAE 上进行实证研究. 本文的贡献总结如下.

- 提出面向预训练漏洞检测模型的黑盒对抗攻击方法 VulBlurrer, 以在更严格的硬标签黑盒场景中生成对抗代码. 设计了定向同义代码转换策略和基于特征一致性、语义一致性和代码流畅度的逃逸分数, 从而减少代码改动量和目标模型查询次数.
- 在广泛使用的预训练模型 CodeBERT、GraphCodeBERT、CodeT5 和 UniXcoder 上进行对比实验. 结果表明, VulBlurrer 相较已有的针对深度学习的黑盒对抗攻击方法表现出更高的攻击成功率, 且更好地权衡了攻击成功率和查询效率, 攻击成功的对抗代码也具有更高的语义一致性和代码流畅度.
- 通过本文提出的 VulBlurrer, 对当前流行的大语言模型 ChatGPT、DeepSeek 和基于大语言模型的辅助编程工具 GitHub Copilot、TRAE 进行鲁棒性测试. 实验结果验证了 VulBlurrer 的有效性, 表明其在真实应用场景下能够有效评估大语言模型在漏洞检测任务上的鲁棒性.

2 相关工作

本节将介绍本文所涉及的背景知识与相关技术, 包括基于预训练模型与大语言模型的源代码漏洞检测和源代码对抗攻击方法两个方面.

2.1 基于预训练模型与大语言模型的源代码漏洞检测

随着深度学习技术的发展, 预训练模型已成为源代码分析与漏洞检测领域的重要研究方向. 近年来, 受自然语言处理领域预训练思想的启发, 研究者提出了多种面向代码的预训练模型, 如 CodeBERT[1]、GraphCodeBERT[2]和 CodeT5[15]等. 这些模型通过在大规模代码语料上进行自监督学习, 能够捕获源代码的词法、语法及语义层次特征, 为漏洞检测提供了更具泛化能力的表示基础. 例如, CodeBERT 能够基于双向 Transformer 架构学习代码与自然语言注释的联合语义空间; GraphCodeBERT 进一步引入数据流图 (data flow graph, DFG)[16]以增强对程序依赖关系的建模能力; 而 CodeT5 则通过编码-解码结构支持代码的生成与重构, 为漏洞修复等任务提供潜在支持. 在此基础上, 大量研究利用这些预训练模型在多种公开数据集 (如 Devign[16]、ReVeal[17]) 上进行微调 (fine-tuning), 实现自动化的漏洞检测任务[18].

随着大语言模型的迅速发展, 其在代码理解与生成任务上展现出前所未有的泛化能力与语义建模能力. 以 ChatGPT 和 DeepSeek 等为代表的模型在通用自然语言语料和大规模代码语料上进行预训练, 能够在零样本或少样本场景下, 通过简单的提示工程适配多种软件工程的下游任务, 包括漏洞检测、代码补全、函数摘要生成与代码修复等. 得益于大语言模型在跨模态的语义理解与上下文推理任务上的能力, 大语言模型能够捕获源代码语义模式、逻辑结构及潜在的安全风险模式. 此外, 模型在预训练过程中学习到的跨语言语义对齐 (如自然语言描述与代码实现的一致性) 使其能够借助自然语言提示对代码进行语义层面的漏洞识别与解释. 已有实证研究证明了大语言模型在源代码漏洞检测任务上的有效性. Zhou 等人[18]通过分析 58 篇相关研究指出, 当前基于预训练模型的漏洞检测方法中约有 73% 依赖于模型微调. 然而, 商用大语言模型 (如 ChatGPT) 仅通过提示工程即可实现与微调模型 (如 CodeBERT) 相当的性能, 并在精确率方面表现更优. Zhou 等人[19]聚焦于 ChatGPT, 设计了包括任务和角色描述、提供项目信息和通用弱点枚举 (common weakness enumeration, CWE) 类型信息以及训练集示例等的不同提示方法, 以探究大语言模型识别源代码漏洞的能力. 除通用对话式大语言模型外, 以 GitHub Copilot 为代表的模型体现了 LLM 在软件开发领域的工程化落地. GitHub Copilot 由 GitHub 与 OpenAI 共同开发, 能够实现函数自动补全、注释生成、代码审计、漏洞检测等功能. 目前, GitHub Copilot 已集成于主流的集成开发环境 (integrated development environment, IDE), 如 Visual Studio Code 和 JetBrains.

2.2 源代码对抗攻击方法

基于预训练模型和大语言模型的源代码漏洞检测方法的研究已经取得了显著进展, 但是与传统深度学习模型一样, 预训练模型在面对经过精心设计的对抗代码时表现出明显的脆弱性. 对抗攻击在保持代码语义不变的前提下引入扰动, 从而误导模型的预测结果. 根据攻击假设, 源代码对抗攻击可分为白盒攻击与黑盒攻击. 白盒攻击假设攻击者可以获取目标模型的完整信息, 包括模型结构、参数和梯度, 从而能够精确地构造扰动方向. 然而, 在实际应用场景中, 参数量较大的预训练模型通常以 API 等形式部署, 模型内部细节对用户不可见. 因此, 黑盒攻击因其更贴近现实而成为研究热点. 其中, 黑盒攻击者仅能将目标模型视为一个“黑匣子”, 通过查询输入并观察其输出来构造对抗代码. 根据攻击者能够从模型输出中获取的预测信息粒度, 可将其进一步细分为硬标签 (hard-label) 黑盒攻击与软标签 (soft-label) 黑盒攻击. 硬标签攻击指攻击者仅能够获取模型的最终预测类别 (例如, 代码片段是否被判定为“有漏洞”), 而无法得知其属于各个类别的置信度. 软标签攻击则假设攻击者能够获取模型输出的概率分布或置信度分数.

在源代码对抗攻击中, 语义保持是生成对抗代码的核心约束. 与图像和自然语言处理领域不同, 源代码的对抗扰动必须确保代码的语法正确且语义一致. 早期的研究主要集中于变量重命名这一简单操作. Zhang 等人[20]提出的 MHM 方法基于 Metropolis-Hastings 采样策略, 通过重命名局部变量生成对抗代码, 证明了使用代码数据集训练的深度学习模型在面对标识符替换时的脆弱性. 然而, 该方法未考虑对抗代码的文本自然性与代码流畅度. 为进一步提升攻击效果与隐蔽性, Yang 等人[9]提出了 ALERT 方法, 在变量重命名基础上引入遗传算法, 优化替换策略, 显著提高了攻击成功率与对抗代码的文本自然性. CodeAttack[10]通过识别代码中对模型预测影响最大的“敏感词元”或“脆弱词元 (vulnerable tokens)”来进行扰动. 为避免与漏洞或脆弱性 (vulnerability) 产生歧义, 本文后续将统一使用“敏感词元”来指代“vulnerable tokens”. CodeAttack 利用掩码语言模型生成符合代码语法与语义的替换词,

实现了对代码翻译、代码修复与代码摘要等多种任务的可迁移攻击. CodeTAE[14]首次探讨了源代码模型在跨领域迁移学习场景下的对抗脆弱性. CodeTAE 将预训练编码器的中间特征差异作为攻击目标, 结合遗传算法生成高迁移性的对抗代码, 在下游任务与模型架构对攻击者未知的设定下, 依然能维持较高的攻击成功率. 针对基于深度学习的漏洞检测模型, 曲豫宾等人[11]提出了 NonVulGen 黑盒攻击框架, 将多种代码同义转换算子引入对抗攻击中, 并将攻击过程建模为一个组合优化问题. NonVulGen 采用三阶段策略: 贪婪搜索初始化、遗传算法优化与深度搜索, 显著提升了对漏洞检测模型的攻击成功率. 除了上述基于搜索与优化的方法外, 陈思然等人[13]提出了一种强化学习式的对抗攻击方法, 在设计原子扰动操作的基础上, 引入强化学习模型作为扰动操作的决策机制, 通过近似策略优化算法动态选择最优扰动序列, 从而在黑盒设置下生成高质量的对抗代码. 但是, 该方法设置的奖励函数仅依赖于目标模型的预测结果, 即决策目标只考虑生成的对抗代码能否成功欺骗目标模型. Yao 等人[12]提出 CARL, 一种基于强化学习的无监督黑盒对抗攻击方法, 其奖励函数综合考虑攻击效果 (如目标模型性能下降和攻击成功率) 与生成样本质量 (如语义一致性、代码流畅度、扰动幅度), 以平衡攻击成功率与代码可用性. 尽管已有研究在源代码对抗攻击领域已取得了一些成果, 但是现有方法仍主要依赖软标签黑盒设定, 即假设攻击者能够获取目标模型的置信度向量. 综上, 现有方法在攻击场景、查询效率及扰动量控制方面仍存在局限, 这成为本研究的出发点.

3 VulBlurrer 对抗攻击方法

3.1 研究动机

近年来, 源代码对抗攻击逐渐成为软件安全领域的重要方向. 针对源代码漏洞检测器的对抗攻击能够生成语义保持但经过精心扰动的对抗代码, 能够误导深度学习模型的判断, 使其无法正确识别潜在漏洞, 从而对自动化漏洞检测系统提出新的挑战. 已有方法主要可分为基于搜索与优化的方法和基于强化学习的方法. 基于强化学习的方法在训练过程中需要频繁查询目标模型, 从而导致更高的计算成本和查询开销; 同时, 其模型训练过程往往伴随参数优化复杂、收敛不稳定等问题, 这限制了该类方法在严格的硬标签黑盒条件下的适用性. 因此, 本文聚焦于基于搜索与优化的方法.

通过第 2 节对现有工作的分析可知, 当前大多数研究仍基于软标签黑盒攻击的假设, 即攻击者可以访问目标模型输出的置信度, 并以此作为指导对抗代码生成的关键信号. 然而, 在诸如对话式大语言模型等日益普及的真实应用环境中, 模型通常仅提供最终的、离散化的分类结果 (例如“有漏洞”或“无漏洞”), 而不提供置信度信息. 在此类场景下, 攻击者难以利用概率分布或梯度相关信息指导攻击. 这种约束更强、挑战更高的硬标签黑盒攻击假设也更贴近真实系统的访问限制. 因此, 本文面向更为严苛的硬标签黑盒场景开展研究, 以支持面向大语言模型的漏洞检测对抗攻击.

此外, 已有的源代码对抗攻击方法在语义一致性和查询效率上仍存在不足. 首先, 已有方法的粗粒度扰动添加策略虽然在一定程度上扩大了对抗代码的搜索空间, 但容易引入大量与漏洞无关的非必要修改, 不仅增加代码修改量, 还可能在复杂控制流等情况下提高语义被破坏的风险. 其次, 与传统本地部署的深度学习模型不同, 当前基于大语言模型的漏洞检测系统往往以云服务或 API 接口的形式对外提供推理能力, 其模型推理过程通常运行在受控的服务器环境中. 在此类实际部署场景下, 模型查询不仅意味着额外的计算与时间开销, 还直接对应真实的服务调用成本. 同时, 对用户的请求频率、调用模式和行为特征通常处于服务提供方的管理与监控下. 因此, 源代码对抗攻击方法需进一步降低对模型的查询依赖以适应真实应用场景.

本文的研究动机可总结为 3 个方面: (1) 更严苛的黑盒攻击场景: 针对无法获得目标模型置信度信息的硬标签黑盒攻击场景开展对抗攻击研究. (2) 更低的查询开销: 在有限的模型交互条件下提升攻击效率, 降低对目标模型的查询依赖. (3) 更小的扰动量: 在保证攻击成功率的前提下, 尽可能减少对源代码的修改, 保持代码语义与功能的一致性, 从而生成更具隐蔽性的对抗代码.

3.2 威胁模型

为明确本文研究所针对的攻击场景与假设, 以及方便读者理解, 本文定义威胁模型如下.

针对基于预训练模型的源代码漏洞检测器, 设漏洞检测器为一个判别函数 $f: \mathcal{X} \to \mathcal{Y}$, 其中 $\mathcal{X}$ 为代码空间, $\mathcal{Y} = \{0,1\}$ 为代码标签空间, 1 表示漏洞代码, 0 表示无漏洞代码. 攻击者无法访问该漏洞检测器的模型参数、置信度输出或梯度信息. 此外, 攻击者具有漏洞代码片段的位置知识. 在实际应用中, 攻击者可能通过已披露的 CVE 漏洞信息、补丁差异或漏洞利用程序获取漏洞位置信息.

具体而言, 在给定一个被漏洞检测器判定为存在漏洞的代码 x (即 $f(x) = 1$) 的情况下, 攻击者的目标是构造一个新的代码 x_{adv}, 使得:

(1) $f(x_{\text{adv}}) = 0$.

(2) x_{adv} 与 x 在执行功能上等价, 且漏洞保留.

(3) x_{adv} 在语义风格如编码风格、命名及可读性上与原始代码相近, 具备隐蔽性.

攻击者利用一组语义保持的代码变换操作算子集合 $\mathcal{T}$ 对 x 进行操作来得到 x_{adv}. 每个操作算子 $t \in \mathcal{T}$ 定义为一个从代码空间到代码空间的映射, 即 $t: \mathcal{X} \to \mathcal{X}$, 如变量名替换、函数名替换、垃圾代码注入等. 多个算子的组合构成一个代码同义转换的操作序列 $\tau = (t_1, t_2, \ldots, t_k)$, 其作用结果为公式 (1), $\circ$ 表示按序操作代码.

$$x_{\text{adv}} = \tau(x) = t_1(x) \circ t_2(x) \circ \ldots \circ t_k(x) \tag{1}$$

因此, 攻击者实际上是在离散的变换空间 $\mathcal{T}^*$ 上进行搜索, 以寻找能使模型判定结果发生改变的代码变换操作路径. 定义 x_{adv} 需要满足的约束条件为一个联合约束函数 $\Phi(\cdot)$, 该约束要求生成的对抗代码必须语法正确且执行功能与原始代码等价. 若满足约束, 则 $\Phi(\cdot) = 1$; 否则, $\Phi(\cdot) = 0$. 因此, 本文研究的面向预训练漏洞检测模型的黑盒对抗攻击可以形式化为以下优化问题.

对于给定漏洞代码 x (即 $f(x) = 1$):

$$\begin{cases} \text{find } \tau \in \mathcal{T}^* \\ \text{s.t.} \begin{cases} x_{\text{adv}} = \tau(x) \\ f(x_{\text{adv}}) = 0 \\ \Phi(x_{\text{adv}}, x) = 1 \end{cases} \end{cases} \tag{2}$$

3.3 方法概述

从研究动机和威胁模型所定义的攻击目标出发, 为应对预训练漏洞检测模型在硬标签黑盒环境下的鲁棒性评估问题, 提出一种面向预训练漏洞检测模型的黑盒对抗攻击方法 VulBlurrer. 后文图 1 为 VulBlurrer 的整体框架图, 主要通过 4 个阶段来生成对抗代码.

阶段 1. 定向同义代码转换: 以漏洞位置为中心施加代码同义变换操作以生成初始候选样本. 该阶段利用指向性扰动降低搜索空间, 并通过设计语法和操作语义等价的同义代码转换操作算子保证生成对抗代码的可用性.

阶段 2. 逃逸分数计算: 提出一种对生成的对抗代码进行评估的复合评分机制, 在无需访问目标模型置信度的条件下评估候选样本的潜在攻击价值. 该评分兼顾特征一致性、语义一致性和代码流畅度, 从而在无查询条件下提供可靠的对抗代码质量判断.

阶段 3. 基于逃逸分数的排序与查询: 依据第 2 阶段计算得到的逃逸分数对所有候选样本进行降序排序, 并选取得分最高的前 k 个样本进行目标模型查询. 若任一被查询样本 x_i^k 的预测标签为“无漏洞” (即 $f(x_i^k) = 0$), 则找到 x_{adv}, 攻击成功并终止后续流程; 若未能在该阶段实现预测标签翻转, 则进入下一阶段的优化过程.

阶段 4. 基于遗传算法的对抗代码优化: 以第 3 阶段中得分较高的候选样本作为初始种群, 利用遗传算法在代码变换空间中进行操作算子组合的迭代优化.

3.4 定向同义代码转换

为生成初始的候选对抗代码, VulBlurrer 采用定向同义代码转换策略. 该策略的目标为: (1) 将扰动集中在与漏洞密切相关的代码区域, 以提高攻击针对性和攻击成功率; (2) 通过事先定义的代码同义变换操作算子组成受控的代码变换集合, 从而约束代码扰动空间, 保持对抗代码与原始代码在执行功能上的一致性, 以提高对抗代码的可用性和语义一致性. 具体地, 定向同义代码转换包含漏洞位置定向转换和关键语句定向转换, 详细过程如算法 1 所示.

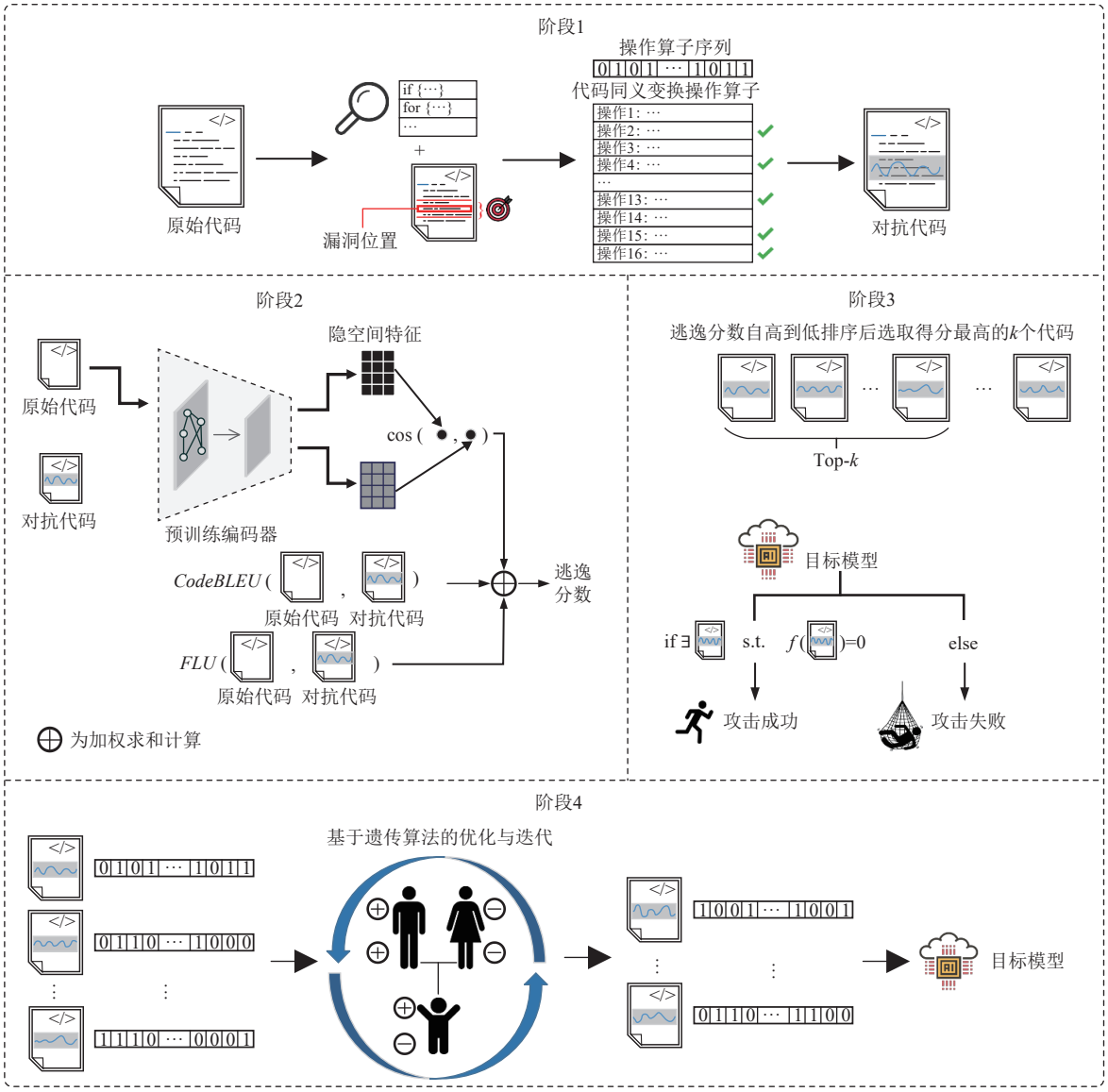


图 1　面向预训练漏洞检测模型的黑盒对抗攻击方法框架图

算法 1. 定向同义代码转换算法.

输入: 原始代码 x, 漏洞位置表示 p, 邻域半径 r, 同义变换算子集合 $\mathcal{T}=\{t_0,t_1,\ldots,t_{|\mathcal{T}|-1}\}$, 优先选择算子集合 $\mathcal{P}=\{t_{p,1},t_{p,2},\ldots,t_{p,q}\}\subset\mathcal{T}$, 优先算子权重 w_high, 非优先算子权重 w_low, 选中算子数量 op_count, 生成的对抗代码数量 $code\_count$;
输出: 候选对抗代码集合 S_{adv}, 操作算子序列索引向量集合 V.

1. 初始化权重向量和操作算子序列 $w,\tau\leftarrow$ empty list
2. 初始化长度为 $|\mathcal{T}|$ 的全零向量 $v\leftarrow[0,0,\ldots,0]$
3. 初始化候选对抗代码集合和操作算子序列索引向量集合 $S_{\text{adv}},V\leftarrow\varnothing$
4. $x'\leftarrow GetNeighborFragment(x,p,r)$

5. **for** t in $\mathcal{T}=\{t_0,t_1,\ldots,t_{|\mathcal{T}|-1}\}$ **do**
6. **if** $t\in\mathcal{P}$ **then**
7. $w.append(w\_high)$
8. **else**
9. $w.append(w\_low)$
10. **for** i=0 to $code\_count$–1 **do**
11. $index\_list \leftarrow WeightedSampling(\{0,1,\ldots,|\mathcal{T}|-1\},w,op\_count)$
12. **for** j=0 to $|\mathcal{T}|-1$ **do**
13. **if** $j\in index\_list$ **then**
14. $v[j]=1$
15. $t \leftarrow GetOperatorByIndex(i)$
16. $\tau.append(t)$
17. $x_{adv} \leftarrow MutateCodeWithOperatorIndex(x,\mathcal{T},\tau)$
18. $S_{adv}.append(x_{adv})$
19. $V.append(v)$
20. **Return** S_{adv}, V

首先, 漏洞位置定向转换, VulBlurrer 借鉴已有源代码对抗攻击研究的敏感词元思想[10,12], 将扰动添加位置聚焦于漏洞邻域. 敏感词元定义为在模型表征或判决过程中对输出结果影响较大的词元, 而攻击这些敏感词元能够增加改变模型预测结果的可能性[10]. 对漏洞检测任务, 漏洞位置天然构成敏感词元位置, 该位置往往承载关键的安全语义信息, 因此 VulBlurrer 优先针对漏洞所在区域的代码进行扰动, 即在以漏洞所在位置为中心的半径窗口内进行同义代码转换操作. 针对关键语句定向转换, 即优先对指定的结构性语句进行扰动. 其次, Du 等人[21]将对多种语句类型 (return、if、throw、try 和 for) 进行对抗攻击与仅修改函数名和变量名的对抗攻击对比发现, 其中对模型判定最为敏感、扰动效果最显著的前 3 类分别是对 if 语句的攻击、仅修改函数名和变量名的对抗攻击和对 for 语句的攻击. 因此, VulBlurrer 在生成初始候选样本时, 优先选择对 if 和 for 语句进行扰动的同义代码转换操作算子和修改函数名与变量名的操作算子. 另外, Li 等人[7] 在针对代码搜索与代码摘要生成任务的对抗攻击研究中发现, 相较于在给定代码中间位置插入冗余或无效代码, 在给定代码片段的前后位置插入垃圾代码对模型性能的干扰更为显著. 在定向同义代码转换策略中, VulBlurrer 采用了曲豫宾等人[11]使用的 14 种同义代码转换操作算子 (表 1 中的操作 1–14), 并定义了在漏洞位置前后插入垃圾代码的操作算子 (表 1 中的操作 15 和 16). 同时, 在随机组合定向同义代码转换操作算子时, 进行带偏随机数生成, 实现对操作 1–6 的优先选择, 对应算法 1 的第 11 行.

表 1　代码同义变换操作算子的描述和示例

操作	转换算子	描述	原始代码	转换后代码
操作1	Rename	函数名和变量名重命名	int cnt;	– int cnt; + int count;

操作2	ForToWhile	转换for循环为while循环	for(i=0; i<5; i++){
			sum+=i;}
				– for(i=0; i<5; i++)
				+ i=0; while(i<5){
				sum+=i;
				+ i++;}

操作3	WhileToFor	转换while循环为for循环	while(n>0){
			n– –;}
				– while(n>0){
				for(;n>0;){
				+ n– –;}

操作4	DoToWhile	转换do循环为while循环	do{
			x++;}
			while(x<10);
				– do{
				x++;
				while(x<10){
				+ x++;}

表 1　代码同义变换操作算子的描述和示例 (续)

操作	转换算子	描述	原始代码	转换后代码

操作5	ElseifToElse	转换if elseif为if else	if(x>90) A;
			else if(x>60) B;
			else C;
				if(x>90) A;
				– else if(x>60) B;
				+ else{
				+ if(x>60) B;
				else C;}

操作6	ElseToElseif	转换if else为if elseif	if(x>80) A;
			else{
			if(x>50) B;
			else C;}
				if(x>80) A;
				– else{
				– if(x>50) B;
				– else C;}
				+ else if(x>50) B;
				+ else C;

操作7	SwitchToIF	转换switch语句为if elseif语句	switch(opt){ case 1: A; case 2: B;}	– switch(opt){ – case 1: A; – case 2: B;} + if(opt==1) A; + else if(opt==2) B;
操作8	RelationChange	关系表达式转换	a<=b	– a<=b + b>=a
操作9	UnaryChange	对一元运算修改	– –k;	– – –k; + k=k–1;
操作10	IncrementChange	增量操作修改	x–=2;	– x–=2; + x=x–2;
操作11	ConstantChange	修改常量	size=100;	– size=100; + size=50*2;
操作12	DefinitionChange	变量定义的修改	float y=3.14;	– float y=3.14; + float y; y=3.14;
操作13	OrderChange	改变无控制依赖关系语句块的顺序	a=1; b=2;	+ b=2; a=1; – b=2
操作14	CommentsDelete	删除打印调试提示和注释的语句	//debug info return val;	– return val; – //debug info

操作15	AddJunkCodeBefore	在漏洞位置前添加无用代码	if(flag) res=1;
			A //漏洞位置
				if(flag){
				res=1;
				if(0)
				+ int a=1
				+ return a;}
				+ A //漏洞位置

操作16	AddJunkCodeAfter	在漏洞位置后添加无用代码	A //漏洞位置
			if(flag) res=1;
				A //漏洞位置
				if(flag){
				res=1;
				+ if(0)
				+ int a=1
				+ return a;}

注: “–”表示删去该行代码; “+”表示新增该行代码

3.5 逃逸分数

在完成定向同义代码转换后, VulBlurrer 将生成多个执行功能等价但扰动不同的候选代码. 为评估生成的候选样本, VulBlurrer 引入逃逸分数用于量化对抗代码质量, 并为后续排序与优化提供依据.

已有方法大多将查询目标模型获得的输出置信度分数直接作为适应度 (fitness) 来衡量对抗代码使目标模型预测结果发生偏移的可能性. 然而, 在更为严格的硬标签黑盒场景下, 无法获得目标模型的梯度信息或者置信度. 因此, 基于置信度的样本评估方法不再适用. 逃逸分数的设计动机在于, 一方面保证对抗代码的文本自然性与语义一致性, 以提高对抗代码的隐蔽性与可部署性; 另一方面追求对抗代码在模型表征空间中产生足够偏移, 以增加误

导模型的概率. 因此, VulBlurrer 提出一种基于特征一致性、语义一致性和代码流畅度的复合评估指标. 下面详细介绍用于计算逃逸分数的 3 个核心要素.

(1) 特征一致性

深度学习模型存在从具体特征到抽象特征的层级化学习规律[22], 即前层网络主要负责基本特征的提取, 而后层网络聚焦于高层特征建模与任务判别. 在这种分层结构下, 输入代码在模型中的传播过程可形式化为:

$$h = \varphi(x),\ y = f(h) \tag{3}$$

其中, $\varphi(\cdot)$ 表示特征提取网络, 负责将输入代码 x 映射到隐空间特征表示 h; $f(\cdot)$ 表示任务映射网络, 将隐空间特征表示 h 投影到任务空间并输出预测结果 y.

代码对抗攻击的核心是在保持代码执行功能不变的前提下, 引导样本在特征空间中跨越决策边界. 因此, 攻击成功的关键在于模型在隐空间中如何“看待”这段代码. 即使代码在语法与功能上保持一致, 只要扰动引起了模型隐空间表征的显著偏移, 就可能导致模型输出的改变. 另一方面, 结构类似的不同模型也具有相似的隐空间表征, 即浅层特征共享, 从而导致对抗攻击的可迁移性[23,24], 这就使得特征一致性指标具有一定的模型无关性, 因此能够在黑盒攻击场景下奏效. 延续针对深度学习模型的成员推理攻击研究[25,26]中的影子模型或代理模型思想, VulBlurrer 引入基于预训练模型的编码器 $\phi(\cdot)$, 计算对抗代码与原始代码的特征一致性如下:

$$fea\_sim(x, x_{\text{adv}}) = \cos(\phi(x), \phi(x_{\text{adv}})) \tag{4}$$

其中, $\cos(\cdot,\cdot)$ 表示余弦相似度. 定义特征一致性指标为:

$$\Delta_{\phi}(x, x_{\text{adv}}) = 1 - fea\_sim(x, x_{\text{adv}}) \tag{5}$$

(2) 语义一致性

定向同义代码转换过程中, 一个核心要求是生成的对抗代码在语法和语义层面上仍与原始代码保持一致. 在逃逸分数中, VulBlurrer 采用 *CodeBLEU*[27]来衡量代码扰动前后在语法和语义上的相似性. *CodeBLEU* 综合了 n-gram 匹配、抽象语法树匹配、数据流匹配及关键字匹配这 4 个维度的加权指标. 定义语义一致性指标如下:

$$\Delta_{\text{CB}}(x, x_{\text{adv}}) = CodeBLEU(x, x_{\text{adv}}) \tag{6}$$

(3) 代码流畅度

代码流畅度用于衡量对抗代码在文本层面上的自然性和可读性. VulBlurrer 采用 Yao 等人[12]提出的基于掩码语言模型 (MLM) 的代码流畅度计算方法. 利用预训练的掩码语言模型计算每个 x_{adv} 中的词元在上下文中的预测置信度, 来度量文本的连贯性与自然性. 设对抗代码为 $x_{\text{adv}} = (x^1_{\text{adv}}, x^2_{\text{adv}}, \ldots, x^{|x_{\text{adv}}|}_{\text{adv}})$, 则对于其中的每个词元 (token), x^j_{adv} 构造掩码词元序列 $x^{(j)}_{\text{adv}} = (x^1_{\text{adv}}, \ldots, x^{j-1}_{\text{adv}}, [\text{MASK}], x^{j+1}_{\text{adv}}, \ldots, x^{|x_{\text{adv}}|}_{\text{adv}})$, 并将其输入掩码语言模型 $M(\cdot)$, 模型输出在掩码位置的预测分布为 $P_M(x^j_{\text{adv}} | x^{(j)}_{\text{adv}})$, 计算代码流畅度指标为:

$$FLU = \frac{1}{|x_{\text{adv}}|} \sum_{j=1}^{|x_{\text{adv}}|} P_M(x^j_{\text{adv}} | x^{(j)}_{\text{adv}}) \tag{7}$$

由于 3 个指标的数值尺度不同, VulBlurrer 在第 1 阶段生成的候选样本集合 S_{adv} 内对各项指标分别采用 min-max 归一化, 以特征一致性指标为例:

$$\widehat{\Delta}_{\phi, S_{\text{adv}}}(x, x_{\text{adv}}) = \frac{\Delta_{\phi}(x, x_{\text{adv}}) - \min\limits_{x'_{\text{adv}} \in S} \Delta_{\phi}(x, x'_{\text{adv}})}{\max\limits_{x'_{\text{adv}} \in S} \Delta_{\phi}(x, x'_{\text{adv}}) - \min\limits_{x'_{\text{adv}} \in S} \Delta_{\phi}(x, x'_{\text{adv}})} \tag{8}$$

同样地, 计算得到 $\widehat{\Delta}_{\text{consistency}}(x, x_{\text{adv}})$ 和 $\widehat{FLU}(x_{\text{adv}})$. 计算逃逸分数为:

$$Esc\_Score(x, x_{\text{adv}}) = \alpha \cdot \widehat{\Delta}_{\phi, S_{\text{adv}}}(x, x_{\text{adv}}) + \beta \cdot \widehat{\Delta}_{\text{CB}, S_{\text{adv}}}(x, x_{\text{adv}}) + \gamma \cdot \widehat{FLU}(x_{\text{adv}}) \tag{9}$$

其中, $\alpha, \beta, \gamma \geqslant 0$ 且 $\alpha + \beta + \gamma = 1$.

3.6 基于逃逸分数的排序与查询

对于第 1 阶段生成的候选样本集合 S_{adv}, VulBlurrer 根据逃逸分数对所有生成样本进行降序排序. 然后采用

Top-k 策略, 选择前 k 个代码作为查询候选集 $S_{\text{adv},k}$ 进行验证. 若其中存在预测标签为正常的样本, 则判定攻击成功; 否则, 将继续进入下一阶段的优化操作. 通过基于逃逸分数的排序与查询策略, VulBlurrer 能够在保证攻击成功率的同时, 降低查询次数. 图 2 为 $|S_{\text{adv}}|=3$ 且 $k=3$ 时的逃逸分数计算示例, 以 ChatGPT (GPT-4.1 版本) 为目标模型, 对抗代码 1 攻击成功, 对抗代码 2 和 3 攻击失败.

原始代码

```
static int _hid_wcslen(WCHAR *str)
{
    int i = 0;
    while (str[i] && (str[i] != 0x409)) {
        i++;
    }
    return i;
}
```

对抗代码 1

```
- static int _hid_wcslen(WCHAR *str) {
+ static int _zkd_wcslen(WCHAR *s_tr_644) {
-     int i = 0;
+     int i_163 = 0;
-     while (str[i] && (str[i] != 0x409)) {
+     while (s_tr_644[i_163] && (s_tr_644[i_163] != 0x409)) {
-         i++;
+         i_163++;
      }
-     return i;
+     return i_163;
  }
```

Δ_{φ}=0.0922
Δ_{CB}=0.8217
FLU=7.2754×10^{-7}
Esc_Score=0.6953

对抗代码 2

```
  static int _hid_wcslen(WCHAR *str) {
      int i = 0;
      while (str[i] && (str[i] != 0x409)) {
-         i++;
+         i = i + 1;
      }
      return i;
  }
```

Δ_{φ}=0.0171
Δ_{CB}=0.8674
FLU=1.1067×10^{-6}
Esc_Score=0.4934

对抗代码 3

```
  static int _hid_wcslen(WCHAR *str) {
-     int i = 0;
+     int i;
+     i = (323 – 323);
      while (str[i] && (str[i] != 0x409)) {
          i++;
      }
      return i;
  }
```

Δ_{φ}=0.0311
Δ_{CB}=0.5750
FLU=1.3883×10^{-6}
Esc_Score=0.3245

图 2　当 $|S_{\text{adv}}|=3$ 且 $k=3$ 时的逃逸分数计算示例 ($\alpha=0.4, \beta=0.35, \gamma=0.25$)

3.7　基于遗传算法的对抗代码优化

在完成逃逸分数的计算与排序和对目标模型的查询后, 若尚未有候选对抗代码能够绕过目标模型的检测, VulBlurrer 则执行下一阶段, 引入基于遗传算法的对抗代码优化策略 (见算法 2), 通过模拟自然选择和进化过程, 从高质量的初始样本出发, 逐步产生更优的对抗代码.

遗传算法开始时, VulBlurrer 将查询候选集 $S_{\text{adv},k}$ 中的样本作为遗传算法的初始种群, 每条染色体对应一组代码同义转换操作算子的索引序列向量 $v\in\{0,1\}^{|\mathcal{T}|}$. 对于每条染色体, VulBlurrer 通过其对应的代码同义转换操作生成对抗代码, 并计算逃逸分数作为适应度值. 在每一代遗传迭代中, 使用锦标赛选择从当前种群中选取两条父本染色体 p_1 和 p_2. 然后对选出的父本 p_1, p_2, 随机选择单点切割位置 l, 交叉后生成子代 $child_1=[p_1[1:l], p_2[l+1:|\mathcal{T}|]]$ 和 $child_2=[p_2[1:l], p_1[l+1:|\mathcal{T}|]]$. 在交叉后, 对生成的子代染色体进行变异操作, 即随机选择子代染色体中的基因进行翻转. 然后将每个子代染色体映射为实际代码操作序列, 并应用到原始代码以生成对抗代码. 将生成的对抗代码输入目标模型进行查询, 若出现成功逃逸的样本, 则攻击成功; 否则, 继续进行下一轮迭代.

如果本轮迭代未攻击成功, 则进行逃逸分数权重动态更新和基于逃逸分数的精英保留策略. 首先, VulBlurrer 逃逸分数权重动态更新策略, 设初始权重为 α_0, β_0, γ_0, 满足 $\alpha_0+\beta_0+\gamma_0=1$, 定义第 i 轮迭代的更新公式为:

$$\alpha_i=\alpha_0+(1-\alpha_0)(1-\mathrm{e}^{-\lambda i}),\ \beta_i=(1-\alpha_i)\frac{\beta_0}{\beta_0+\gamma_0},\ \gamma_i=(1-\alpha_i)\frac{\gamma_0}{\beta_0+\gamma_0} \tag{10}$$

然后, 将上代种群与子代合并, 形成扩展候选集. 根据公式 (8) 和 (9) 及更新后的权重重新计算所有候选样本的逃逸分数. 按照逃逸分数进行降序排序, 并选取前 k 条样本对应的染色体构成下一代种群继续迭代.

算法 2. 基于遗传算法的对抗代码优化策略.

输入: 原始代码 x, 迭代最大轮数 $max\_iteration$, 初始种群大小 k, 初始种群及其对应的染色体 $S_{\text{adv},k}$, $V_{\text{adv},k}$, 生成子代数量 $child\_count$, 同义变换算子集合 $\mathcal{T}$, 逃逸分数初始权重 α_0, β_0, γ_0;
输出: 对抗代码 x_{adv}.

初始化子代列表 $child\_list \leftarrow$ empty list
$S^0_{\text{adv},k} \leftarrow S_{\text{adv},k}$
$V^0_{\text{adv},k} \leftarrow V_{\text{adv},k}$
for i=0 to $max\_iteration$–1 do
  $p_1, p_2 \leftarrow SelectParentsUsingTournament(S^i_{\text{adv},k}, V^i_{\text{adv},k})$
  for j=1 to $child\_count$/2 do
    $child_1, child_2 \leftarrow Crossover(p_1, p_2)$
    $child_1, child_2 \leftarrow Mutation(child_1, child_2)$
    $child\_list.append(child_1)$
    $child\_list.append(child_2)$
  $x_{\text{adv}} \leftarrow QueryTargetModel(child\_list)$
  //将生成的子代输入目标模型进行查询, 若攻击成功则返回成功的样本, 否则返回 None
  if x_{adv} is not None then
    Return x_{adv}
  $\alpha_i, \beta_i, \gamma_i \leftarrow UpdateWeights(\alpha_0, \beta_0, \gamma_0, i)$
  $S_{\text{child}} \leftarrow MutateCodeWithOperaterIndex(x, \mathcal{T}, child\_list)$
  $S^i_{\text{adv},k} \leftarrow S^i_{\text{adv},k} \cup S_{\text{child}}$
  $V^i_{\text{adv},k} \leftarrow V_{\text{adv},k} \cup child\_list$
  $esc\_score\_list \leftarrow CalculateEscapeScore(\alpha_i, \beta_i, \gamma_i, S^i_{\text{adv},k})$
  $S^i_{\text{adv},k}, V^i_{\text{adv},k} \leftarrow SelectTopKElementsByEscapeScore(k, esc\_score\_list, S^i_{\text{adv},k}, V^i_{\text{adv},k})$
  $x_{\text{adv}} \leftarrow$ GetElementWithHighestEscapeScore($esc\_score\_list, S^i_{\text{adv},k}$)
Return x_{adv}

4 实验设计与结果分析

4.1 实验数据集

本文使用 Fan 等人[28]提出的 Big-Vul 数据集, 该数据集在源代码漏洞检测领域被广泛使用[29−31]. Big-Vul 是一个高质量的真实漏洞数据集, 包含来自开源项目的真实漏洞函数, 收录了 2002–2019 年间公开披露的 CVE 漏洞信息, 覆盖 348 个不同的开源项目. 本文使用 Fan 等人[28]提供的经过数据清洗后的版本 (https://GitHub.com/ZeoVan/MSR_20_Code_Vulnerability_CSV_Dataset), 共包含 188 636 个函数样本, 其中 10 900 个为漏洞函数, 177 736 个为非漏洞函数. 按照 8:1:1 的比例将其划分为训练集、验证集和测试集. 本文选择 Big-Vul 数据集的原因为: (1) 该数据集中的样本包含真实世界的漏洞, 均来源于真实项目的历史提交记录, 具有较高的代表性与可靠性, 且被广泛用于源代码漏洞检测模型的性能评估; (2) 该数据集提供了较为全面的样本信息, 包括每个漏洞样本对应的 CVE 编号、漏洞修复前后的函数代码; (3) 该数据集反映了真实世界的漏洞分布情况, 即非漏洞代码数量远多于漏洞代码.

4.2 目标模型

本文选择了 4 个被广泛使用的预训练代码模型作为对抗攻击的目标模型, 包括 CodeBERT[1]、GraphCodeBERT[2]、CodeT5[15]和 UniXcoder[32]. 这 4 种模型分别代表了序列建模、图结构建模和序列到序列生成这 3 类典型的代码表示学习范式, 其在源代码漏洞检测任务中均取得了显著性能. CodeBERT 采用基于 Transformer 的编码器堆栈结构, 能够捕获代码与自然语言注释的上下文依赖; GraphCodeBERT 在此基础上进一步引入数据流图信息, 用以增强对代码语义结构的建模能力. CodeT5 作为一种编码-解码结构的预训练模型, 兼具理解与生成能力, 被用于多种代码理解与转换任务; UniXcoder 是一种基于 Transformer 的代码预训练语言模型, 在多种代码理解与分析任务中表现出色.

此外, 为了进一步评估 VulBlurrer 在大语言模型及基于大语言模型的商用工具上的通用性, 本文还在当前流行的大语言模型及其衍生工具上进行了实证研究, 包括 2 个大语言模型 ChatGPT、DeepSeek, 以及基于大语言模型的辅助编程工具 GitHub Copilot、TRAE. ChatGPT 是由 OpenAI 开发的通用对话式大语言模型, 使用大规模自然语言与代码语料混合训练, 具备较强的代码生成、语义理解和逻辑推理能力. 而 DeepSeek 是近年来开源社区中发展迅速的多模态大语言模型之一, 经过专门的代码语料强化预训练, 特别增强了其在程序分析、代码补全与错误检测方面的能力. 已有研究证明了 ChatGPT 和 DeepSeek 在源代码漏洞检测上具有较为出色的性能[19,33]. GitHub Copilot 是由 GitHub 与 OpenAI 共同开发的人工智能编程工具, 其中整合了 GPT 系列模型, 能够在用户编写代码的过程中实时生成补全建议、函数实现或注释描述. 而 GitHub Copilot 的官方文档中提及了其具备辅助识别代码中常见漏洞的功能, 并给出了进行漏洞检测的提示词 (prompt) 范式 (https://docs.GitHub.com/en/copilot/tutorials/copilot-chat-cookbook/analyze-security/find-vulnerabilities). TRAE 为字节跳动发布的基于大语言模型的辅助编程工具, 提供了可以部署在 Visual Studio Code (VS Code) 及 JetBrains 等主流编辑器中的插件, 并且提供了智能问答功能. 本文所使用目标模型的统计信息如表 2 所示.

表 2　本文所使用目标模型的统计信息

模型	版本	参数量	使用方式	测试集召回率 (%)
CodeBERT	—	120M	微调	78.04
GraphCodeBERT	—	120M	微调	73.86
CodeT5	—	220M	微调	58.11
UniXcoder	—	125M	微调	78.68
ChatGPT	GPT-4.1	—	API	78.59
DeepSeek	DeepSeek-V3.1	685B	API	60.47
GitHub Copilot	GPT-4.1	—	VS Code 插件	71.64
TRAE	Gemini-2.5-Flash	—	VS Code 插件	70.44

针对 ChatGPT、DeepSeek、GitHub Copilot 和 TRAE, 本文均采用构造对话提示词的方式实现源代码漏洞检测. 采用基于任务描述和角色描述的方法[19,34]构造提示词, 定义模板如下:

“You are an experienced developer who knows the security vulnerability very well. Now you need to identify whether a method contains any potential vulnerability or not. If it has any potential vulnerability, output: ‘this code is vulnerable’. Otherwise, output: ‘this code is non-vulnerable’. You are only to give the prior two answers. The Code is: **[X]**. Let’s Start:”

其中, **[X]** 为输入的代码, 并且明确规定了模型是二分类任务 (输出“this code is vulnerable”或“this code is non-vulnerable”), 并严格限制其仅返回这两个预设答案, 以确保输出格式的统一性, 同时便于后续的自动化处理与结果分析. 由于大语言模型 (如 ChatGPT) 可能会产生与预先定义的标签不同的回答[19], 因此当大语言模型生成的答案偏离指定的标签时, 需人工检查其预测类别.

对于 ChatGPT, 通过 *client.chat.completions.create*() 方法调用其对话补全 API (https://api.openai.com/v1/chat/completions). 在该方法中, 影响代码漏洞检测效果的关键参数包括 model 和 message. 其中, model 参数指定所使用的具体模型版本, 由于不同模型在参数量与结构上存在差异, 其漏洞检测能力也有所不同. message 参数用于向模

型提供输入, 实验中采用固定的提示模板. 对于 DeepSeek, 使用其进行漏洞检测的方式与 ChatGPT 类似, 本文通过调用对话补全的 API (https://api.deepseek.com/v1/chat/completions) 来实现. 对于 GitHub Copilot 和 TRAE, 均采用基于 IDE 的插件实现对话功能. 由于 GitHub Copilot 和 TRAE 均未提供 API, 实验中逐条将处理后的包含样本的提示, 输入模型对话窗口并记录输出结果. 每次提问均开启新对话, 使模型不会基于历史对话内容进行学习, 从而保障实验结果的准确性. 图 3 展示了使用 GitHub Copilot 进行源代码漏洞检测的示例.

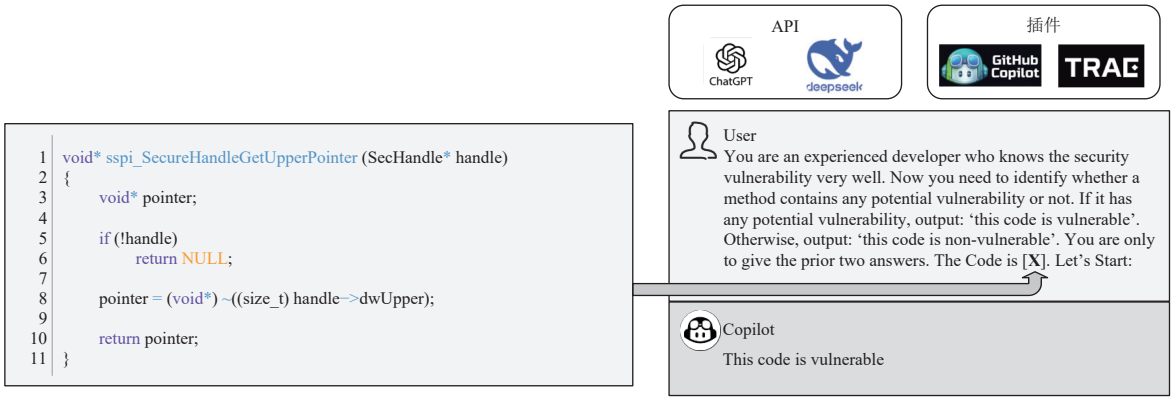


图 3 基于 GitHub Copilot 的源代码漏洞检测示例

4.3 方法实现

本节介绍方法实现过程中的主要超参数及设置. 对于算法 1 中的邻域半径 r, 由于存在长度较短 (行数小于 10) 和长度较长 (行数大于 1 000) 的代码, 根据代码长度采用分段设置. 对行数小于 10 行的短代码, 设置固定的邻域半径 $r=2$; 对于行数在 10–20 之间的代码, 设置固定的邻域半径 $r=3$; 对于行数大于 20 的样本, 则按行数比例设置 $r=0.1\times L$, 其中 L 为样本的行数. 其余参数设置见表 3. 此外, 对于基线方法 NonVulGen 和 CodeTAE, 本文均采用与原文献相同的超参数设置. 实验相关代码开源在项目主页 (https://github.com/SwithunWang/VulBlurrer).

表 3 实验设置表

参数设置	模型	设定值
训练轮次epoch	CodeBERT、GraphCodeBERT、CodeT5、UniXcoder	20
块大小block size		400
训练批大小train batch size		4
评估批大小eval batch size		8
学习率learning rate		2×10^{-6}
遗传算法的最大迭代次数*max_iteration*	CodeBERT、GraphCodeBERT、CodeT5、UniXcoder	10
	ChatGPT、DeepSeek、GitHub Copilot、TRAE	5
选择的操作算子数量*op_count*	–	8
生成的对抗代码数量*code_count*	–	15
初始种群大小k	–	5
生成子代数量*child_count*	–	4
逃逸分数的初始权重 α_0, β_0, γ_0	–	0.4, 0.35, 0.25
逃逸分数动态调整速率 λ	–	0.05
用于计算特征一致性预训练模型	–	CodeT5.encoder
用于代码流畅度的预训练模型	–	CodeBERT

4.4 研究问题及评估指标

本文聚焦于以下 4 个研究问题 (research question, RQ) 并展开实验.

RQ1: 在基于 CodeBERT、GraphCodeBERT、CodeT5 和 UniXcoder 微调的预训练漏洞检测模型上, 相比于两个先进的基线攻击方法 NonVulGen 和 CodeTAE, 本文提出的攻击方法 VulBlurrer 是否具有更好的性能?

RQ2: 不同攻击方法生成的对抗代码的质量如何?

RQ3: VulBlurrer 在当前流行的大语言模型和基于大语言模型的辅助编程工具上的有效性如何?

RQ4: VulBlurrer 中不同组件对其攻击效果的影响如何?

本文引入了两个指标来评估对抗攻击的性能, 包括攻击成功率和查询次数.

(1) 攻击成功率: 表示生成的带有漏洞的对抗代码成功使目标模型产生错误分类的比例, 更高的攻击成功率说明方法在使漏洞代码绕过检测模型方面具有更强的攻击效果. 定义攻击成功率的计算过程如下.

对于测试集 S_{Test} 和目标模型 M, 首先获得在测试集中被模型正确识别为有漏洞的代码集合, 即真阳性 (true positive, TP) 代码集合 S_{TP}.

$$S_{\text{TP}} = \{x \in S_{\text{Test}} \mid f_M(x) = 1\} \tag{11}$$

然后对 S_{TP} 中的代码施加扰动, 对于每个 $x \in S_{\text{TP}}$, 若能生成满足公式 (2) 的对抗代码 x_{adv}, 则将 x_{adv} 加入攻击成功的对抗代码集合 S_{success}. 攻击成功率定义为:

$$\text{攻击成功率} = \frac{|S_{\text{success}}|}{|S_{\text{TP}}|} \times 100\% \tag{12}$$

其中, $|S_{\text{success}}|$ 为 S_{success} 中的样本数量, $|S_{\text{TP}}|$ 为 S_{TP} 中的样本数量.

(2) 平均查询次数: 表示对 S_{TP} 进行对抗攻击的过程中对目标模型的平均查询次数, 更低的平均查询次数说明攻击方法具有更高的查询效率, 耗费的时间和经济成本更低, 在实际黑盒攻击场景中被检测到的可能性也更低. 本文中的对抗扰动仅施加于 S_{TP}, 即仅针对原本能够被目标模型正确识别的漏洞代码进行攻击. 本文关注攻击方法在完整攻击阶段中的总体查询成本定义. 一方面, 在攻击失败的情况下, 模型查询同样已经发生, 并构成真实开销; 另一方面, 在实际黑盒攻击场景中, 不管攻击成功与否, 被攻击方只能观测到攻击过程中产生的全部模型查询行为, 过多的模型查询对被攻击方来说是不正常和可疑的. 因此, 将平均查询次数定义为总查询次数相对于 $|S_{\text{TP}}|$ 的平均值, 计算公式为:

$$\text{平均查询次数} = \frac{\#Queries}{|S_{\text{TP}}|} \tag{13}$$

其中, *#Queries* 为整个攻击过程中的总查询次数.

在保证攻击成功率的同时, 所生成的对抗代码应尽量保持与原始代码具有语义一致性及文本自然性. 针对生成的对抗代码质量, 本文从语义一致性和代码流畅度两个角度进行评估.

(1) *Avg_CodeBLEU*: 在 BLEU 的基础上, 微软提出了 *CodeBLEU* 以评价生成代码的质量, *CodeBLEU* 同时结合了 n-gram 匹配、语法树匹配、数据流匹配以及代码关键词匹配这 4 个维度. 较高的 *CodeBLEU* 值表示生成代码在语法与语义层面均较好地保持与原始代码一致. 本文采用 *CodeBLEU* 来衡量对抗代码与原始代码的一致性, *CodeBLEU* 计算如第 3.5 节中公式 (6) 所示. 定义 *CodeBLEU* 的平均值计算过程如下:

$$Avg\_CodeBLEU = \frac{1}{|S_{\text{success}}|} \sum_{j=1}^{|S_{\text{success}}|} CodeBLEU(x^j, x_{\text{adv}}^j) \tag{14}$$

其中, x_{adv}^j 为 S_{success} 中的对抗代码, x^j 为 x_{adv}^j 的原始代码.

(2) *Avg_FLU*: 用于评估生成的对抗代码在文本层面的自然性和可读性, 更高的代码流畅度表明生成的对抗代码更不易被察觉, 代码流畅度的计算如第 3.5 节中公式 (7) 所示. 定义 *FLU* 的平均值计算过程如公式 (15) 所示, 更高的 *Avg_FLU* 表示攻击方法生成的对抗代码在代码流畅度上表现更优.

$$Avg\_FLU = = \frac{1}{|S_{\text{success}}|} \sum_{j=1}^{|S_{\text{success}}|} FLU(x_{\text{adv}}^j) \tag{15}$$

4.5 实验结果与分析

(1) RQ1

本文以 CodeBERT、GraphCodeBERT、CodeT5 和 UniXcoder 作为目标模型, 评估了 VulBlurrer 相较于两个基线方法 NonVulGen 和 CodeTAE 的性能, 包括攻击成功率和平均查询次数, 结果如图 4 所示, 图中 Δ 表示差值.

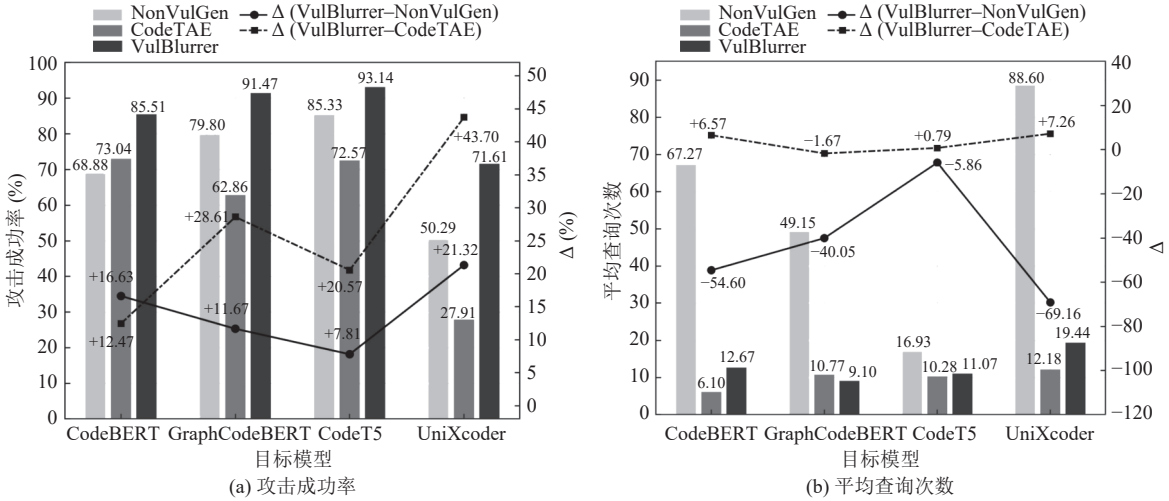


图 4 NonVulGen、CodeTAE 和 VulBlurrer 对不同目标模型攻击的结果

攻击成功率方面, VulBlurrer 在 4 种模型上均表现最优, 分别达到 85.51%、91.47%、93.14% 和 71.61%, 显著高于 NonVulGen (68.88%、79.80%、85.33%、50.29%) 和 CodeTAE (73.04%、62.86%、72.57%、27.91%). 特别地, VulBlurrer 相较于 CodeTAE 在 GraphCodeBERT 和 UniXcoder 上的提升分别超过 28% 和 43%. 这些结果说明 VulBlurrer 在不同架构的预训练漏洞检测模型上产生了对抗影响, 能够使更多的漏洞代码逃逸这 4 种预训练漏洞检测模型的检测. 在平均查询次数方面, VulBlurrer 同样展示了其在攻击效率上的优势. 相较于 NonVulGen, VulBlurrer 在 CodeBERT、GraphCodeBERT、CodeT5 和 UniXcoder 上的平均查询次数分别由 NonVulGen 的 67.27 次、49.15 次、16.93 次和 88.60 次下降至 12.67 次、9.10 次、11.07 次和 19.44 次, 显著提高了查询效率. 对于 CodeTAE, 在 GraphCodeBERT 和 CodeT5 上, VulBlurrer 的平均查询次数分别为 9.10 次和 11.07 次, 与 CodeTAE (10.77 次和 10.28 次) 接近. 在 GraphCodeBERT 上 VulBlurrer 的查询次数略低于 CodeTAE; 而在 CodeBERT、CodeT5 和 UniXcoder 上, VulBlurrer 的平均查询次数略高于 CodeTAE, 但 VulBlurrer 的攻击成功率明显高于 CodeTAE, 其在 CodeBERT 上提升 12.47%, CodeT5 上提升 20.57%, UniXcoder 上提升 43.70%.

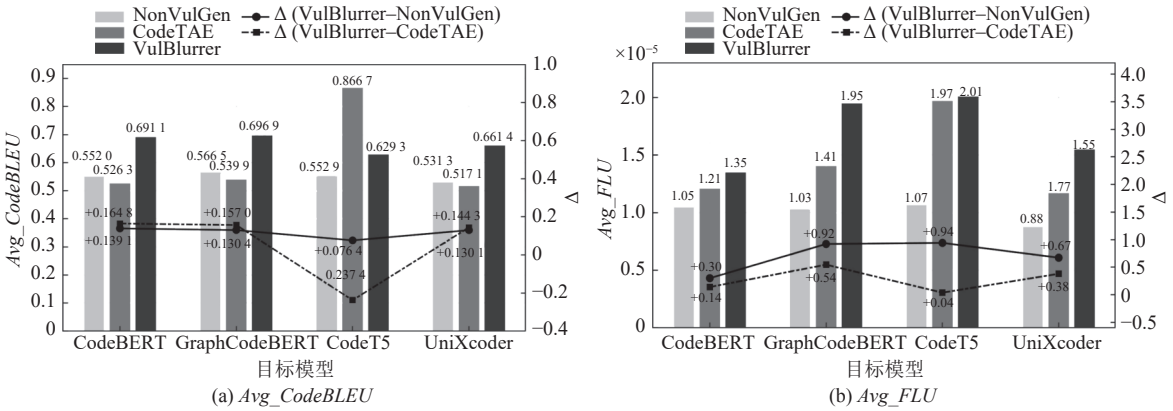
综上可见, VulBlurrer 在攻击成功率与查询效率之间实现了更优的平衡, 在大幅提升攻击成功率的同时仍保持了可接受甚至优越的平均查询次数. VulBlurrer 相较基线方法显示出更优攻击性能的主要原因在于: 1) VulBlurrer 基于漏洞位置和关键语句优先施加同义变换, 使得扰动集中于与安全语义密切相关的区域, 从而提高攻击的针对性. 2) 在选择扰动算子时结合带偏随机选择算子与多算子组合, 既保证扰动的多样性以增加越过模型决策边界的概率, 又通过优先选择经验上更易触发模型预测变化的算子提升效率. 3) 利用结合了特征一致性、语义一致性与代码流畅度的复合逃逸分数, 对候选的扰动样本进行评估与排序, 从而减少查询次数, 并且逃逸分数的权重动态调整, 使 VulBlurrer 能够在早期探索多样扰动, 在后期聚焦特征一致性, 实现在扰动空间中更高效的对抗搜索.

(2) RQ2

本文对不同攻击方法生成的对抗代码质量进行了评估, 实验结果如图 5 所示.

在代码一致性上, VulBlurrer 在 CodeBERT、GraphCodeBERT 和 UniXcoder 上获得了最高的 *Avg_CodeBLEU* (分别为 0.6911、0.6969 和 0.6614), 相较于 NonVulGen 的 0.5520、0.5665 和 0.5313 均有提升. CodeTAE 在这 3

个模型上的表现均低于 NonVulGen, 其 *Avg_CodeBLEU* 分别为 0.526 3、0.539 9 和 0.517 1. 在 CodeT5 上, 虽然 CodeTAE 的 *Avg_CodeBLEU* 达到 0.866 7, 但 VulBlurrer 的 *CodeBLEU* 为 0.629 3, 仍显著优于 NonVulGen (0.552 9), 表明 VulBlurrer 在不同模型上均能生成质量稳定的对抗代码. 在代码流畅度上, VulBlurrer 攻击生成的对抗代码在所有目标模型上均具有最高的 *Avg_FLU* 值. CodeTAE 生成的对抗代码的 *Avg_FLU* 值次之. VulBlurrer 在 CodeBERT、GraphCodeBERT、CodeT5 和 UniXcoder 上, 其 *Avg_FLU* 值分别为 1.35×10^{-5}、1.95×10^{-5}、2.01×10^{-5} 和 1.55×10^{-5}, 高于 NonVulGen (1.05×10^{-5}、1.03×10^{-5}、1.07×10^{-5} 和 0.88×10^{-5}), 并在 CodeT5 上略高于 CodeTAE (1.97×10^{-5}). 综合 *Avg_CodeBLEU* 与 *Avg_FLU* 这两个指标, VulBlurrer 相较于其他方法在生成对抗代码的质量上具有明显优势, 在保持代码语义一致性的同时, 生成的对抗代码 *Avg_FLU* 值更高, 且在多种预训练漏洞检测模型上表现稳定, 验证了 VulBlurrer 在黑盒攻击环境下生成高质量对抗代码的能力.



(a) *Avg_CodeBLEU* (b) *Avg_FLU*

图 5 NonVulGen、CodeTAE 和 VulBlurrer 生成的对抗代码质量

(3) RQ3

为验证 VulBlurrer 在当前主流大语言模型及基于大语言模型的商用代码工具上的有效性, 本文选择了 4 个具有代表性的目标模型进行评估: 包括开源模型 DeepSeek, 闭源模型或工具 ChatGPT、GitHub Copilot 及 TRAE. 实验中分别统计了每个模型在未受攻击时的漏洞检测召回率 (见表 2), VulBlurrer 的攻击成功次数和平均查询次数.

实验结果如图 6 所示, VulBlurrer 能够在所有目标模型上成功实现漏洞代码的逃逸, 即使含漏洞的代码被模型误判为无漏洞代码, 表明 VulBlurrer 在不同模型上均具有良好的攻击有效性与一定程度的跨模型迁移能力. 从攻击成功次数来看, 针对 TRAE 的攻击成功次数最高, 为 333 次, 即有 333 个含漏洞的代码在经过 VulBlurrer 扰动后被误判为无漏洞代码. 其次是 GitHub Copilot (186 次) 与 ChatGPT (176 次); 而 DeepSeek 的成功次数最少 (162 次). VulBlurrer 在 ChatGPT、DeepSeek、GitHub Copilot 和 TRAE 上的平均查询次数分别为 16.94 次、15.90 次、16.49 次和 14.02 次. 另一方面, 本文的实验结果证明了现有的基于大语言模型的漏洞检测在面对语义保持的对抗性扰动时仍表现出一定的脆弱性, 并且不同模型在对抗鲁棒性上存在显著差异.

(4) RQ4

本文以基于贪婪搜索的同义代码转换作为消融实验的基准方法, 构建了 4 组消融实验以评估 VulBlurrer 中不同组件对攻击效果的影响, 实验结果如图 7 所示. 具体而言, 该基准方法首先在第 3.4 节定义的 16 种代码同义变换算子中随机选择 8 个算子 (*op_count*=8), 并基于所选算子为每个原始代码生成 15 个对抗代码 (*code_count*=15); 随后, 将生成的 15 个对抗代码逐一输入目标漏洞检测模型进行查询, 以获得攻击结果. 引入定向同义代码转换后, 攻击成功率有明显提升, 其中在 GraphCodeBERT 上从 57.34% 提升至 79.67%, 表明目标定向策略能够有效增加攻击成功率. 进一步结合基于逃逸分数的排序与查询后, 在 GraphCodeBERT、CodeT5 和 UniXcoder 上攻击成功率分别达到 80.93%、84.85% 和 68.15%, 显示逃逸分数排序能够更精确地选择高效对抗代码, 但在 CodeBERT 上造成了攻击成功率的略微下降 (从 58.79% 到 58.19%), 说明逃逸分数在提高生成样本质量和降低查询次数的同时会

降低生成对抗代码的多样性, 从而影响攻击成功率. 最终, 结合遗传算法优化的 VulBlurrer 在 4 种模型上实现了攻击成功率的较大提升, 说明遗传算法能够通过扩大搜索空间, 探索更多算子组合, 增加找到能够攻击成功的对抗代码的机会.

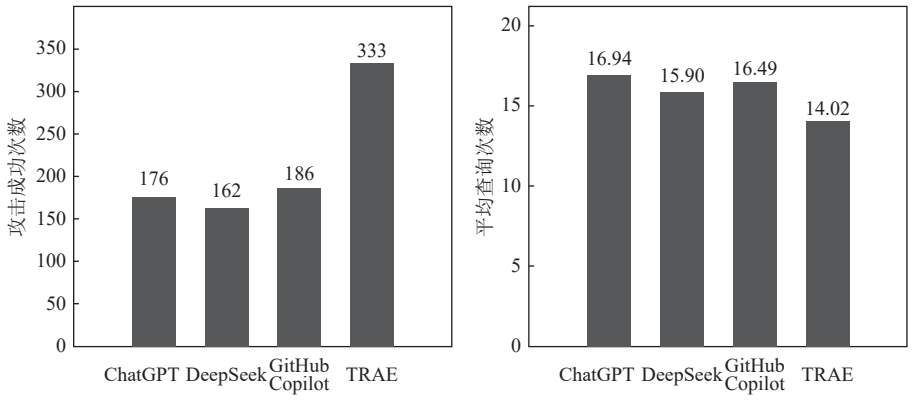


图 6 VulBlurrer 在 ChatGPT、DeepSeek、GitHub Copilot 和 TRAE 上的攻击成功次数和平均查询次数

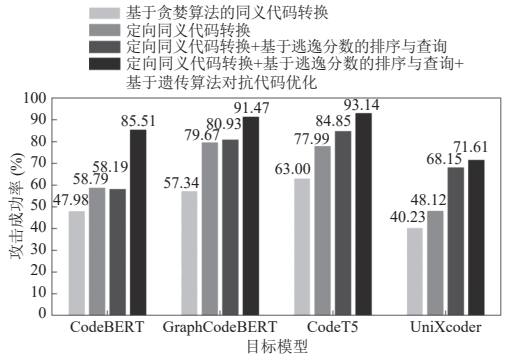


图 7 VulBlurrer 中不同组件对攻击成功率的影响

4.6 有效性威胁

● 内部威胁. 内部有效性威胁主要来源于本文提出对抗攻击方法 VulBlurrer 中存在的一些需要微调的超参数, 以及目标模型和对比方法的实现细节. VulBlurrer 涉及的超参数, 例如选择的操作算子数量、生成的对抗代码数量、遗传算法的最大迭代次数、初始种群大小和生成子代数量等, 可能在不同取值下对实验结果产生影响. 为缓解上述威胁, 本文在实验设计中采取了如下措施: 所有实验均在统一参数配置下进行, 避免针对特定模型或数据集进行额外的参数调优, 以确保实验对比的公平性; 其次, 关键参数采用随机搜索的方式, 并综合攻击成功率和平均查询次数来确定合适的值; 再次, 对于对比方法, 采用原文献中公开的参数设置与实现方式.

● 外部威胁. 外部有效性威胁主要来源于数据集、目标模型以及编程语言. 首先, 本文采用当前漏洞检测与对抗攻击研究中广泛使用的公开基准数据集, 但实验中使用的目标模型 (如大语言模型) 在预训练阶段可能已经接触过相关代码, 从而对对抗攻击效果产生潜在影响. 本文方法与所有对比方法均在相同数据集和目标模型下进行评估, 因此实验结果在相对比较意义上仍然是公平和有效的. 此外, 不同模型的训练、微调设置以及提示工程均可能影响攻击效果. 本文在实验中对同一模型采用统一的参数与提示设置, 以减少模型配置差异带来的影响, 但尚未系统分析不同提示工程策略对攻击性能的影响. 其次, 本文在计算逃逸分数时, 特征一致性的计算使用了预训练模

型 CodeT5, 代码流畅度的计算使用了预训练模型 CodeBERT. 尽管 CodeT5 和 CodeBERT 已被广泛用于代码语义建模, 并在已有工作中验证了其有效性, 但是不同预训练模型因架构与训练语料的不同存在代码语义建模能力上的差异, 从而可能对特征一致性与代码流畅度的数值产生影响. 再次, 本文以 C 语言代码为研究对象验证 VulBlurrer 的有效性, 不同编程语言在语法结构、代码风格以及同义变换规则上存在差异, 本文方法在其他编程语言上的效果可能有所不同. 尽管本文所采用的攻击流程和整体框架具有一定的通用性, 但相应的同义代码变换算子仍需针对具体语言进行适配.

结构有效性: 结构有效性威胁主要体现在对抗代码质量评估方式上. 本文在实验中主要采用自动化评价指标生成的对抗代码进行评估, 这些指标虽然在现有研究中被广泛采用, 但可能无法全面反映对抗代码在可读性、语义一致性和隐蔽性等方面的质量. 当引入人工评估时, 可能会对对抗代码质量给出基于人类理解和专家知识的判断. 未来工作将考虑引入人工评估机制, 例如邀请具备相关漏洞知识的专业编程人员, 对对抗代码的可读性、语义一致性和漏洞的可识别性等方面进行评估, 以更全面地刻画对抗代码的质量和隐蔽性.

5 总 结

本文提出了一种面向预训练漏洞检测模型的黑盒对抗攻击方法, 通过定向同义代码转换优先在漏洞邻近的敏感区域和特定语句类型施加扰动, 并引入结合特征一致性、语义一致性及代码流畅度的逃逸分数, 对候选对抗代码进行量化评估, 利用遗传算法优化生成的对抗代码以进一步提高攻击成功率. 实验结果表明, VulBlurrer 在 CodeBERT、GraphCodeBERT、CodeT5 和 UniXcoder 上分别达到 85.51%、91.47%、93.14% 和 71.61% 的攻击成功率, 明显高于 NonVulGen (68.88%、79.80%、85.33%、50.29%) 和 CodeTAE (73.04%、62.86%、72.57%、27.91%), 显示其在不同架构的预训练漏洞检测模型上均具有较为稳定的攻击能力, 并且生成的对抗代码具有较高的语义一致性和代码流畅度. 在查询效率方面, 相较于 NonVulGen, VulBlurrer 在 4 种模型上的平均查询次数显著降低 (从 67.27、49.15、16.93、88.60 次降至 12.67、9.10、11.07、19.44 次). 在 CodeBERT、CodeT5 和 UniXcoder 上 VulBlurrer 的平均查询次数虽然略高于 CodeTAE, 但其攻击成功率提升明显 (CodeBERT 提升 12.47%, CodeT5 提升 20.57%, Unixcoder 提升 43.70%), 更好地平衡了攻击成功率与查询效率. 同时, VulBlurrer 在 ChatGPT、DeepSeek、GitHub Copilot 及 TRAE 等大语言模型和辅助编程工具上同样表现出有效性. VulBlurrer 不仅在多种预训练漏洞检测模型上实现了高攻击成功率, 还保证了对抗代码的质量和低查询次数, 充分证明了现有基于预训练模型的漏洞检测模型在面对语义保留型对抗攻击时存在鲁棒性不足的问题, 同时为后续模型防御与鲁棒性提升提供了依据.

未来工作将从以下 3 个方面展开: (1) 进一步优化同义代码转换算子的设计, 提出更具针对性和有效性的同义代码转换算子, 并探索方法在其他编程语言上的有效性; (2) VulBlurrer 在大语言模型上的攻击效果仍有提升空间, 针对大语言模型, 探索结合上下文信息的攻击策略; (3) 利用高质量对抗代码进行对抗训练或鲁棒性增强, 设计针对预训练模型和大语言模型的防御策略, 提高漏洞检测系统在面对对抗性代码时的安全性和鲁棒性.

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: Findings of the Association for Computational Linguistics: EMNLP 2020. 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. In: Proc. of the 9th Int'l Conf. on Learning Representations. Vienna: OpenReview.net, 2021.

[3] Yuan XY, He P, Zhu QL, Li XL. Adversarial examples: Attacks and defenses for deep learning. IEEE Trans. on Neural Networks and Learning Systems, 2019, 30(9): 2805–2824. [doi: 10.1109/TNNLS.2018.2886017]

[4] Wei H, Tang H, Jia XM, Wang ZX, Yu HX, Li ZB, Satoh S, van Gool L, Wang Z. Physical adversarial attack meets computer vision: A decade survey. IEEE Trans. on Pattern Analysis and Machine Intelligence, 2024, 46(12): 9797–9817. [doi: 10.1109/TPAMI.2024.3430860]

[5] 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 (TIST), 2020, 11(3): 24. [doi: 10.1145/3374217]

[6] 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]

[7] Li YX, Qi SY, Gao CY, Peng Y, Lo D, Lyu MR, Xu ZL. Understanding the robustness of Transformer-based code intelligence via code transformation: Challenges and opportunities. IEEE Trans. on Software Engineering, 2025, 51(2): 521–547. [doi: 10.1109/TSE.2024.3524461]

[8] Li Z, Huang X, Li YR, Chen G. A comparative study of adversarial training methods for neural models of source code. Future Generation Computer Systems, 2023, 142: 165–181. [doi: 10.1016/j.future.2022.12.030]

[9] 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]

[10] 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]

[11] 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 with English abstract). http://www.jos.org.cn/1000-9825/7379.htm [doi: 10.13328/j.cnki.jos.007379]

[12] Yao KC, Wang H, Qin C, Zhu HS, Wu YJ, Zhang LB. CARL: Unsupervised code-based adversarial attacks for programming language models via reinforcement learning. ACM Trans. on Software Engineering and Methodology, 2024, 34(1): 22. [doi: 10.1145/3688839]

[13] Chen SR, Wu JZ, Ling X, Luo TY, Liu JY, Wu YJ. Reinforcement-learning-based adversarial attacks against vulnerability detection models. Ruan Jian Xue Bao/Journal of Software, 2024, 35(8): 3647–3667 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7120.htm [doi: 10.13328/j.cnki.jos.007120]

[14] Yang YL, Fan HR, Lin CH, Li Q, Zhao ZY, Shen C. Exploiting the adversarial example vulnerability of transfer learning of source code. IEEE Trans. on Information Forensics and Security, 2024, 19: 5880–5894. [doi: 10.1109/TIFS.2024.3402153]

[15] 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. Punta Cana: ACL, 2021. 8696–8708. [doi: 10.18653/v1/2021.emnlp-main.685]

[16] 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. 10197–10207.

[17] 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]

[18] Zhou X, Cao SC, Sun XB, Lo D. Large language model for vulnerability detection and repair: Literature review and the road ahead. ACM Trans. on Software Engineering and Methodology, 2025, 34(5): 145. [doi: 10.1145/3708522]

[19] Zhou X, Zhang T, Lo D. Large language model for vulnerability detection: Emerging results and future direction. In: Proc. of the 44th ACM/IEEE Int'l Conf. on Software Engineering: New Ideas and Emerging Results. Lisbon: ACM, 2024. 47–51. [doi: 10.1145/3639476.3639762]

[20] 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. [doi: 10.1609/aaai.v34i01.5469]

[21] 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]

[22] Jawahar G, Sagot B, Seddah D. What does BERT learn about the structure of language? In: Proc. of the 57th Annual Meeting of the Association for Computational Linguistics. Florence: ACL, 2019. 3651–3657. [doi: 10.18653/v1/P19-1356]

[23] Wallace E, Feng S, Kandpal N, Gardner M, Singh S. Universal adversarial triggers for attacking and analyzing NLP. In: Proc. of the 2019 Conf. on Empirical Methods in Natural Language Processing and the 9th Int'l Joint Conf. on Natural Language Processing (EMNLP-IJCNLP). Hong Kong: ACL, 2019. 2153–2162. [doi: 10.18653/v1/D19-1221]

[24] Wei XX, Liang SY, Chen N, Cao XC. Transferable adversarial attacks for image and video object detection. In: Proc. of the 28th Int'l Joint Conf. on Artificial Intelligence. Macao: AAAI Press, 2019. 954–960. [doi: 10.24963/ijcai.2019/134]

[25] Hu L, Yan AL, Yan HY, Li J, Huang T, Zhang YY, Dong CY, Yang CS. Defenses to membership inference attacks: A survey. ACM Computing Surveys, 2023, 56(4): 92. [doi: 10.1145/3620667]

[26] Ye JY, Maddi A, Murakonda SK, Bindschaedler V, Shokri R. Enhanced membership inference attacks against machine learning models. In: Proc. of the 2022 ACM SIGSAC Conf. on Computer and Communications Security. Los Angeles: ACM, 2022. 3093–3106. [doi: 10.1145/3548606.3560675]

[27] Ren S, Guo DY, Lu S, Zhou L, Liu SJ, Tang DY, Sundaresan N, Zhou M, Blanco A, Ma S. CodeBLEU: A method for automatic evaluation of code synthesis. arXiv:2009.10297, 2020.

[28] 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]

[29] Fu M, Nguyen V, Tantithamthavorn CK, Le T, Phung D. Vulexplainer: A Transformer-based hierarchical distillation for explaining vulnerability types. IEEE Trans. on Software Engineering, 2023, 49(10): 4550–4565. [doi: 10.1109/TSE.2023.3305244]

[30] Steenhoek B, Gao HY, Le W. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 16. [doi: 10.1145/3597503.3623345]

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

[32] Guo DY, Lu S, Duan N, Wang YL, Zhou M, Yin J. UniXcoder: Unified cross-modal pre-training for code representation. In: Proc. of the 60th Annual Meeting of the Association for Computational Linguistics (Vol. 1: Long Papers). Dublin: ACL, 2022. 7212–7225. [doi: 10.18653/v1/2022.acl-long.499]

[33] Qin WT, Suo LJ, Li LC, Yang F. Advancing software vulnerability detection with reasoning LLMs: DeepSeek-R1's performance and insights. Applied Sciences, 2025, 15(12): 6651. [doi: 10.3390/app15126651]

[34] Chen C, Su JZ, Chen JC, Wang YL, Bi TT, Yu JX, Wang YL, Lin XW, Chen T, Zheng ZB. When ChatGPT meets smart contract vulnerability detection: How far are we? ACM Trans. on Software Engineering and Methodology, 2025, 34(4): 100. [doi: 10.1145/3702973]

附中文参考文献

[11] 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 [doi: 10.13328/j.cnki.jos.007379]

[13] 陈思然, 吴敬征, 凌祥, 罗天悦, 刘镓煜, 武延军. 面向漏洞检测模型的强化学习式对抗攻击方法. 软件学报, 2024, 35(8): 3647–3667. http://www.jos.org.cn/1000-9825/7120.htm [doi: 10.13328/j.cnki.jos.007120]

作者简介

王霄东, 博士生, 主要研究领域为深度学习, 漏洞检测.

陈郅楷, 硕士生, CCF 学生会员, 主要研究领域为系统安全, 软件缺陷检测, 大语言模型.

王一杰, 硕士生, CCF 学生会员, 主要研究领域为机器学习, 软件安全性.

黄荔, 本科生, CCF 学生会员, 主要研究领域为物联网, 系统安全.

关志涛, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为人工智能安全, 系统安全, 隐私计算, 区块链, 应用密码.