软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2024,35(8):3647−3667 [doi: 10.13328/j.cnki.jos.007120]

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

面向漏洞检测模型的强化学习式对抗攻击方法 *

陈思然[1,2], 吴敬征[1,3], 凌 祥[1], 罗天悦[1], 刘镓煜[1,2], 武延军[1,3]

[1](中国科学院 软件研究所 智能软件研究中心, 北京 100190)

[2](中国科学院大学, 北京 100049)

[3](计算机科学国家重点实验室(中国科学院 软件研究所), 北京 100190)

通信作者: 吴敬征, E-mail: jingzheng08@iscas.ac.cn

摘 要: 基于深度学习的代码漏洞检测模型因其检测效率高和精度准的优势, 逐步成为检测软件漏洞的重要方法, 并在代码托管平台 GitHub 的代码审计服务中发挥重要作用. 然而, 深度神经网络已被证明容易受到对抗攻击的干扰, 这导致基于深度学习的漏洞检测模型存在遭受攻击、降低检测准确率的风险. 因此, 构建针对漏洞检测模型的对抗攻击不仅可以发掘此类模型的安全缺陷, 而且有助于评估模型的鲁棒性, 进而通过相应的方法提升模型性能. 但现有的面向漏洞检测模型的对抗攻击方法依赖于通用的代码转换工具, 并未提出针对性的代码扰动操作和决策算法, 因此难以生成有效的对抗样本, 且对抗样本的合法性依赖于人工检查. 针对上述问题, 提出了一种面向漏洞检测模型的强化学习式对抗攻击方法. 该方法首先设计了一系列语义约束且漏洞保留的代码扰动操作作为扰动集合; 其次, 将具备漏洞的代码样本作为输入, 利用强化学习模型选取具体的扰动操作序列; 最后, 根据代码样本的语法树节点类型寻找扰动的潜在位置, 进行代码转换, 从而生成对抗样本. 基于 SARD 和 NVD 构建了两个实验数据集, 共 14 278 个代码样本, 并以此训练了 4 个具备不同特点的漏洞检测模型作为攻击目标. 针对每个目标模型, 训练了一个强化学习网络进行对抗攻击. 结果显示, 该攻击方法导致模型的召回率降低了 74.34%, 攻击成功率达到 96.71%, 相较基线方法, 攻击成功率平均提升了 68.76%. 实验证明了当前的漏洞检测模型存在被攻击的风险, 需要进一步研究提升模型的鲁棒性.

关键词: 对抗攻击; 漏洞检测; 强化学习; 代码转换

中图法分类号: TP311

中文引用格式: 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

英文引用格式: 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). http://www.jos.org.cn/1000-9825/7120.htm

Reinforcement-learning-based Adversarial Attacks Against Vulnerability Detection Models

CHEN Si-Ran[1,2], WU Jing-Zheng[1,3], LING Xiang[1], LUO Tian-Yue[1], LIU Jia-Yu[1,2], WU Yan-Jun[1,3]

[1](Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: Deep learning-based code vulnerability detection models have gradually become an important method for detecting software vulnerabilities due to their advantages of high detection efficiency and accuracy, and play an important role in the code auditing service of

* 基金项目: 中国科学院战略性先导科技专项(XDA0320400); 国家自然科学基金(62202457); 源图重大基础设施资助
本文由“系统与网络软件安全”专题特约编辑向剑文教授、陈厅教授、王浩宇教授、罗夏朴教授、杨珉教授推荐.
收稿时间: 2023-09-10; 修改时间: 2023-10-30; 采用时间: 2023-12-15; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-04-03

the code hosting platform GitHub. However, deep neural networks have been proved to be susceptible to the interference of adversarial attacks, which leads to the risk of deep learning-based vulnerability detection models being attacked and reducing the detection accuracy. Therefore, building adversarial attacks against vulnerability detection models can not only uncover the security flaws of such models, but also help to evaluate the robustness of the models, and then improve the performance of the models through corresponding methods. However, the existing counter-attack methods for vulnerability detection models rely on generalized code transformation tools, and do not propose targeted code perturbation operations and decision algorithms, so it is difficult to generate effective counter-attack samples, and the legitimacy of the counter-attack samples relies on manual checking. To address the above problems, a reinforcement learning adversarial attack method for vulnerability detection model is proposed. The method firstly designs a series of semantically constrained and vulnerability-preserving code perturbation operations as a set of perturbations; secondly, the code samples with vulnerabilities are used as inputs, and the reinforcement learning model is used to select specific sequences of perturbation operations; finally, the code samples are used to search for potential locations of perturbations according to the types of nodes in the syntax tree, and then code transformations are carried out, thus generating the counteracting samples. Based on SARD and NVD, two experimental datasets with a total of 14 278 code samples are constructed, and four vulnerability detection models with different characteristics are trained as attack targets. For each target model, a reinforcement learning network is trained to counter the attack. The results show that the attack method leads to a 74.34% decrease in the recall of the models and a 96.71% success rate, which is an average increase of 68.76% compared to the baseline method. The experiment proves that the current vulnerability detection model has the risk of being attacked, and further research is needed to improve the robustness of the model.

Key words: adversarial attack; vulnerability detecting; reinforcement learning; code transformation

近年来, 随着计算机应用和软件供应链的发展, 软件产业规模日益庞大, 软件安全问题也愈发突出. 其中, 软件源代码漏洞是引发各类软件安全事件的主要原因之一. 美国国家漏洞数据库(national vulnerability database, NVD)的数据显示: 2016—2022 年, 每年披露的安全漏洞记录条目逐年增长. 截至 2023 年 6 月, 已披露的安全漏洞数目达到了 12 613 条. 数目日益增长的代码漏洞亟需进行检测和处理. 基于深度学习的代码漏洞检测技术[1−4]凭借其不需要安全专家手工定义漏洞的规则和模式并且检测效率高的特点, 成为近年学术界的主流研究方向. 基于深度学习的代码漏洞检测工作已经在检测准确性和检测效率方面取得了优异的效果. 代码托管平台 GitHub 基于已有的漏洞检测模型开发了工具 CodeQL 用以代码审计服务.

然而, 近年的很多研究工作已经证明, 深度神经网络(deep neural network, DNN)容易受到对抗样本的影响. Szegedy 等人[5]在其论文里首次提出了对抗样本(adversarial example)的概念, 即: 在数据集中, 通过故意添加细微的干扰形成模型的输入样本, 对抗样本会导致模型以高置信度给出一个错误的输出. 已知的深度神经网络在许多领域, 例如图像处理[6,7]和语音识别[8,9]领域, 都会受到对抗样本的影响. 而基于深度学习的漏洞检测模型也继承了 DNN 面对对抗样本的脆弱性. 基于此, 本工作通过添加扰动构造了一个漏洞代码的对抗样本. 图 1 展示了该案例, 图 1(a)为具备漏洞的原代码片段, 通过在第 4 行添加一个打印语句, 并在第 6 行添加一个虚假的错误处理语句, 得到图 1(b)中的代码片段作为对抗样本. 然后, 利用漏洞检测模型 Devign 对图 1(a)中的原代码片段和图 1(b)中的对抗样本进行了漏洞检测, 结果显示, 漏洞检测器能够识别原代码片段中的漏洞,但是将对抗样本识别为良好的代码片段. 这个案例表明, 精心构造的代码扰动操作确实能够使得基于深度学习的漏洞检测模型失效.

```
char *buf1;
buf1 = (char*) malloc(size);
free(buf1);
/* FLAW: use-after-free */
strncpy(buf1,"hack",5);
```

(a) 原代码

```
char *buf1;
buf1 = (char*) malloc(size);
free(buf1);
printf('No Vulnerability!');
/* FLAW: use-after-free */
if(buf1 != buf1){exit(-1);}
strncpy(buf1,"hack",5);
```

(b) 扰动后的代码

图 1 面向漏洞检测模型的对抗样本示例

通过构建针对漏洞检测模型的黑盒对抗攻击方法可以发掘此类模型的安全缺陷, 而且有助于验证模型的鲁棒性, 其重要性体现在如下 3 点: (1) 这种攻击属于黑盒攻击, 相较于白盒攻击更符合真实情形; (2) 通过对

抗攻击, 能揭示被评估模型的脆弱性, 改善当前评估此类模型局限于检测有效性的问题; (3) 有助于进一步研究模型如何提高防御能力.

但因为源代码本身具备复杂的语法结构和语义信息, 生成代码形式对抗样本的研究仍需改进. 针对漏洞检测模型, 给定一个漏洞代码作为输入, 其成功的对抗样本需要具备以下特性: (1) 与原代码语义一致; (2) 保留原代码中的漏洞; (3) 能有效地针对漏洞检测任务. 目前, 基于源代码生成对抗样本的工作主要应用在代码相似性检测[10]、代码预训练[11,12]等领域. 面对漏洞检测模型, 利用现有的代码转换工具进行对抗样本的生成, 随机性强, 难以针对性地生成有效对抗样本. 而 Li 等人[13]的工作专注于鲁棒性的提升, 未针对漏洞检测的对抗样本进行专门的扰动设计, 并且安全专家需要花费大约 105 个小时进行人工检查, 来保证转换后的代码保留了语义和漏洞. 目前, 尚未有针对漏洞检测模型的自动化的黑盒攻击方法, 这种攻击的有效性和成功率缺少可靠依据.

针对上述问题, 本文设计并实现了面向漏洞检测模型的强化学习式对抗攻击方法. 具体来说, 首先, 根据对抗攻击的目标——漏洞检测模型, 预先设计一组扰动操作作为扰动备选, 这组扰动操作是根据漏洞数据库上公开披露的代码漏洞及其补丁的特征设计而成, 并且是语义约束且漏洞保留的; 其次, 将漏洞代码样本转化为词向量表征, 将其作为强化学习模型的输入, 利用模型中的π网络输出一个概率分布, 根据此概率分布抽样得到一个扰动操作的序号, 并根据此序号选取某一具体的扰动操作; 最后, 将漏洞代码样本解析为语法树, 并且逐一遍历语法树的节点, 根据类型选取语法树节点作为应用扰动操作的潜在位置, 并提取语法树子树上的关键变量作为生成扰动内容的语料. 在潜在的代码扰动位置上, 根据所生成的扰动内容对原始代码进行转换, 从而生成对抗样本.

本文的主要贡献总结如下:

(1) 针对漏洞检测模型提出了一系列轻量的、语义约束和漏洞保留的原子扰动操作. 这些扰动操作的设计思路基于公开披露的代码漏洞及其补丁的特征, 因此, 扰动操作不仅不破坏原有代码的语义和漏洞信息, 还更具备针对性, 据此生成的对抗样本更容易逃避模型的检测;

(2) 设计并训练了一个强化学习模型作为指导扰动操作序列选择的决策模型. 给定一个具备漏洞的代码样本, 良好的决策模型能够针对样本特征选取更加合理且有效的扰动操作序列进行代码转换, 因此只需更少的扰动操作就使得生成的对抗样本能够逃避目标模型的检测. 该模型通过内部的近似策略优化(proximal policy optimization, PPO)[14]网络来选取正确的扰动操作, 指导对抗性样本的生成, 从而达到误导目标模型的目的;

(3) 通过使用本文所提出的方法, 在基于 SARD 和 NVD 构建的数据集上攻击 4 个不同类型且均训练良好的漏洞检测模型进行对比实验. 利用原本能够被检测出漏洞的样本, 经过扰动而生成对抗样本集合. 结果表明: 针对目标模型, 平均有 83.83%, 90.67%, 17.20%和 30.56%的对抗样本可以成功地逃避漏洞检测. 考虑到这些漏洞检测模型在两个数据集上 $F1$ 分数平均为 72.97%和 54.49%, 具备良好的检测能力, 本文的攻击方法工作显著地降低了这些基于深度学习漏洞检测模型的检测准确性.

本文第 1 节对背景知识与相关技术进行介绍. 第 2 节定义威胁模型. 第 3 节介绍本文所提出方法的整体方案以及实现细节. 第 4 节对本文中所进行的实验进行阐述. 第 5 节对本文工作进行总结, 并对本方法的不足以及改进空间进行讨论.

1 相关工作

本节将对本文中所涉及的背景知识与相关技术进行介绍, 主要包括代码漏洞检测技术和深度神经网络的对抗样本生成两方面.

1.1 代码漏洞检测技术

针对源代码漏洞的检测方法近年来得到了广泛的研究. 常见的代码漏洞检测方法通常可划分为两类: (1) 动态分析方法[15,16], 此类方法需要对代码或程序进行编译、运行并观察运行中的信息和运行后的结果;

(2) 静态分析方法, 此类方法通过分析源代码中的特征和模式, 判断代码是否存在漏洞, 不需要对源代码进行编译和运行. 由于静态分析方法能处理无法编译但包含漏洞的代码片段, 本文主要关注基于静态分析的代码漏洞检测技术, 而静态分析方法可以进一步分为基于规则的检测技术和基于学习的检测技术.

1.1.1 基于规则的源代码漏洞检测技术

静态分析方法不需要构建复杂的编译环境对目标程序代码进行编译和运行, 具备检测效率高的特点. 其中, 基于代码规则和模式的静态检测方法通常由安全专家预定义代码具备漏洞的特征和规则, 并利用程序分析手段对代码的数据流和控制流进行分析和特征提取, 以此判断代码中是否存在漏洞. Lu 等人[17]面对安全检查漏洞缺乏标准、形式多样且难以识别的特点, 定义了安全检查的识别规则, 并且提出一种自动化的方法来识别操作系统内核中缺失安全检查的代码片段, 以此来检测操作系统内核中常见的 3 类漏洞. 在此工作的基础上, Lu 等人[18]进一步优化了对此类漏洞的检测方法, 定位操作系统内核代码中关键变量, 并结合语义和上下文对关键变量执行路径的差异性进行交叉检查, 以此检测出操作系统的内核之中缺失安全检查的漏洞. 但基于规则的方法需要专家总结并定义漏洞的识别规则, 人力成本高.

1.1.2 基于学习的源代码漏洞检测技术

近年来, 随着深度学习技术在安全领域的广泛应用, 利用深度学习对代码中的漏洞进行检测成为另一重要的代码漏洞静态分析方法. Li 等人[1]提出了基于循环神经网络(recurrent neural networks, RNN)的漏洞检测模型 VulDeePecker, 其工作将目标程序代码提取为一组程序切片, 并将切片级的表征输入循环神经网络之中, 以学习代码的漏洞特征并生成判别漏洞存在的分类器来进行漏洞检测. 该方法细化了漏洞检测的粒度, 能够在代码的切片级别上检测出漏洞. 进一步, Li 等人[2]又对代码的特征提取进行了优化, 提出了漏洞检测工具 SySeVR. SySeVR 提出了一种数据类型 SyVCs 来存储代码的漏洞特征, SyVCs 是通过对代码的语法和语义特征进行提取并转换为向量而生成的, 它改善了代码切片的质量, 提高了漏洞检测的效率. Zhou 等人[3]提出了智能漏洞检测框架 Devign, 该框架将代码提取为联合图, 该图以 AST 为中心, 包含不同级别的数据依赖和控制依赖编码. 基于联合图, Devign 利用图神经网络(graph neural networks, GNN)学习代码丰富的语义信息, 并对代码进行漏洞是否存在的检测. 为了更有效地检测细粒度的漏洞, Duan 等人[4]提出了检测框架 VulSniper. 该框架将源代码编码为特征张量从而避免语义信息的丢失, 并利用基于注意力机制(attention)的神经网络模型进行细粒度的代码漏洞检测. 在此工作基础上, 段旭等人[19]进一步提出了一种基于代码属性图及注意力双向 LSTM 的源码级漏洞挖掘方法, 对代码属性图进行切片以剔除与敏感操作无关的冗余信息, 并用基于双向 LSTM 和注意力机制的神经网络进行漏洞检测. Cao 等人[20]提出一种流敏感的图神经网络, 将源代码、控制流与数据流的信息进行联合嵌入用以训练网络, 以检测代码中内存相关漏洞. 以上基于深度学习的漏洞检测方法取得了极好的检测准确率, 并且避免了安全专家预定义漏洞规则和模式, 减少了工作量. 但是, 由于深度神经网络自身具备面对对抗攻击的脆弱性且无法对其预测行为进行合理解释的特点, 在安全攸关型应用中部署的深度学习模型会带来安全隐患[20]. 因此, 仍有必要开展针对此类模型脆弱性的攻击研究.

1.2 深度神经网络的对抗样本生成

对抗样本是指通过向正常样本中添加精细设计的、自然通道下无法感知的扰动所形成的输入样本[21]. 2015 年, Goodfellow 等人[6]提出了快速梯度符号方法, 通过对图片添加扰动形成对抗样本使得图片识别模型输出错误结果, 在此工作中, 正式提出了对抗攻击的概念. 对抗攻击能够使深度学习模型以高置信度的方式给出错误的输出, 实现针对深度学习检测服务的逃逸攻击[22]. 近年来, 研究者们针对不同领域的识别和分类任务开展了对抗样本生成的研究.

1.2.1 图像形式的对抗样本生成

在图像和计算机视觉领域, Wang 等人[23]提出了可迁移的目标对抗攻击(TTAA), 从标签和特征的角度捕获目标类的分布信息生成高度可迁移的目标对抗样本. Wei 等人[24]提出了一种新颖的基于迁移的目标攻击方法, 引入了一种特征相似性损失, 通过最大化对抗扰动的全局图像和随机裁剪的局部区域之间的特征相似性来生成对抗样本. 非图像的对抗样本生成的研究同样很多, 例如基于图结构[25,26]和基于程序代码[27−29]的研究.

1.2.2 代码形式的对抗样本生成

由于程序代码具备语法结构性和语义信息, 研究者们难以将其他领域的方法直接迁移到代码的对抗样本生成领域. 例如: 与图像、语音处理等领域不同, 代码的变化是离散而非连续的, 难以利用传统的梯度方法进行扰动生成. 同时, 对抗样本生成的一个原则是, 原样本和对抗样本的差别应该是难以察觉的. 区别于图像和音频对抗样本在自然通道上的难以察觉, 代码的对抗样本不能用与原样本的文本相似性这一指标来衡量, 而应该从语义一致性、代码执行结果一致性等方面进行研究.

在代码的对抗样本生成领域, 学者关注不同领域任务的深度学习模型的对抗攻击. 其中: Jha 等人[12]提出在代码的自然通道中检测预训练的语言模型针对对抗性攻击的脆弱性; Zhang 等人[27]通过使用 Metropolis-Hastings 抽样方法, 对代码中的命名标识符进行替换和重新生成, 以此来生成对抗样本; 进一步, Yang 等人[28]对该方法进行了改进, 通过使用贪婪和遗传算法提高对抗样本生成的效果. 在代码注释生成任务领域, Zhou 等人[29]提出了一种标识符替换方法 ACCENT, 以生成对抗样本, 来评估和提高基于 DNNs 的代码注释生成任务的鲁棒性. 在代码混淆检测领域, Zhang 等人[10]利用遗传算法、蒙特卡洛算法和强化学习网络作为优化策略来指导代码转换以生成混淆代码对, 针对基于深度学习的克隆检测器进行攻击; Li 等人[13]对基于深度学习的漏洞检测模型的鲁棒性进行了研究, 证明了这些模型能够受到对抗性漏洞样本的影响, 并利用特征学习和分类学习解耦的方法去提升这些模型的鲁棒性.

上述任务没有针对于漏洞检测任务的特点设计扰动操作规则和代码转换策略, 利用了已有的代码转换规则和工具, 并在一定程序上依赖于手工的检验和筛选. 因此, 有必要研究面向漏洞检测模型的对抗攻击方法.

2 威胁模型

本文参考代码对抗样本生成领域工作的建模原则[12]定义了相应的威胁模型, 具体包含了所提出攻击方法的攻击能力、攻击知识和攻击目标, 并进行了形式化的定义.

2.1 攻击能力

给定一个具备漏洞的代码样本作为输入, 攻击者的能力是基于一组扰动操作规则对此样本进行代码转换, 从而生成对抗样本以此扰乱目标模型. 面向基于深度学习的漏洞检测模型的对抗样本必须具备以下两个特征: (1) 对抗样本需要保留原代码的语义, 即扰动后的代码应该和原始代码具备同样的语义信息和相同的运行结果; (2) 对抗样本需要保留代码中的漏洞, 即对抗样本仍然具备与原代码样本相同的漏洞, 但是能够逃避目标模型的漏洞检测. 这两个特征保证了原代码样本和对抗样本之间的语义一致性和漏洞一致性, 从而保证了两者的相似性.

特别地, 给定一个具备漏洞的代码 $x\in X$ 作为输入, 其中, X 是样本的输入空间. 利用一组扰动操作Δ对 x 进行代码转换而得到漏洞代码的对抗样本 x_{adv}, 有效的对抗样本 x_{adv} 需要满足以下的要求:

$$x_{adv}=\Delta(x)\in A \tag{1}$$

$$\Delta=\delta_1\circ\delta_2\circ\ldots\circ\delta_n,\ n\leqslant N \tag{2}$$

其中, δ表示一个原子性的扰动操作, 此操作将一个具备漏洞的代码转化为另一个不改变语义和漏洞属性的代码; $\Delta=\delta_1\circ\delta_2\circ\ldots\circ\delta_n$ 表示为由 n 个原子扰动操作所构成的扰动操作序列, Δ逐一应用序列中的扰动操作将原始代码样本转化为对抗样本; N 代表扰动操作序列的最大长度, 由于原始代码样本及其对抗样本需要保证相似性, 故扰动操作序列保持一个长度上限, 即 $n\leqslant N$; 而 A 是衡量原始代码样本及其对抗样本两者一致性的特征空间, A 空间中的所有代码样本都具备相同的语义和漏洞属性, 只在个别词法和语法特征上存在差异, 以此满足对抗样本所具备的语义一致和漏洞保留的特征.

2.2 攻击知识

本工作的攻击环境是标准的黑盒攻击, 黑盒攻击代表攻击者不能够访问模型的参数、模型的梯度、模型的结构以及损失函数等信息, 只能够给定模型所需求的输入, 以此获取模型输出的得分或者分类标签. 由于

真实场景下, 待攻击的目标模型或者商用漏洞检测工具并不会开源自己的模型细节, 因此, 基于黑盒访问环境的攻击方法更加符合现实场景中对现有的漏洞检测模型的攻击和评估的环境.

2.3 攻击目标

给定一个具备漏洞的代码样本 x 作为输入, 攻击的目标是: 根据攻击能力生成的对抗样本 x_{adv} 不仅能被目标模型误检测为不具备漏洞的代码样本, 而且与原始的代码样本具备相同的语义和漏洞属性.

特别地, 给定一个预先训练好的漏洞检测模型 $F:X\to Y$, 其中, X 是样本 x 的输入空间, Y 是输出空间, 代表了模型的输入 x 是否存在漏洞的标签. 该模型作为攻击的目标模型. 定义概率 g 作为模型内网络输出样本 x 具备漏洞的概率. 给定一个代码样本 $x\in X$ 作为模型的输入, 其输出可以表示为

$$F(x)=\begin{cases}1, & g(x)\geqslant 0.5\\ 0, & g(x)<0.5\end{cases} \tag{3}$$

当模型 F 的输出为 1 时, 代表样本 x 被模型判断为具备漏洞; 反之, 则代表模型判断样本 x 是不具备漏洞的良好代码. 而本文的攻击目标是: 尽可能地降低对抗样本被判断为具备漏洞的概率, 使得目标模型 F 将对抗样本误检测为一个良好的、无漏洞的代码样本. 本文将生成对抗样本的最终问题表述如下:

$$\begin{aligned}\Delta_{atk}&=\arg\max_{\Delta} g(x)-g(x_{adv})\\&=\arg\max_{\Delta} F(x_{adv})\\ \text{s.t. }&: F(x)=1,\ F(x_{adv})=0\end{aligned} \tag{4}$$

上述的目标函数是: 寻找一个扰动序列 Δ_{atk}, 使原始的代码样本 x 经过目标模型 F 输出的概率与对抗样本 x_{adv} 经过 F 所输出概率的差异最大化. 由于原始代码样本 x 经过目标模型 F 得到的结果为固定值, 因此上述目标同样可以表述为对抗样本 x_{adv} 经过目标模型 F 输出的概率最小化. 从而使得原本能被目标模型 F 检测出漏洞的代码样本 x, 根据扰动序列 Δ_{atk} 进行代码转换得到对抗样本 x_{adv}, 该对抗样本能够逃避目标模型 F 的检测.

3 面向漏洞检测模型的强化学习式对抗攻击方法

为了实现上一节中威胁模型中所定义的攻击目标, 生成能够逃避目标模型检测的对抗样本, 研究并实现了一种面向漏洞检测模型的强化学习式对抗攻击方法. 图 2 为本方法的整体框架示意图, 本方法通过以下 3 个阶段生成对抗样本, 对目标模型进行对抗攻击.

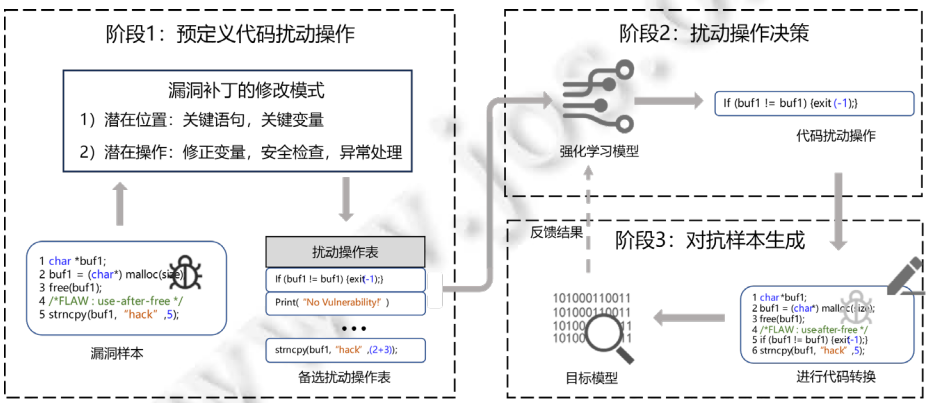


图 2 面向漏洞检测模型的强化学习式对抗攻击方法整体框架

给定一个具备漏洞的代码片段作为样本.

在第 1 阶段, 根据对抗攻击的目标——漏洞检测模型, 预先设计一组原子扰动操作规则作为扰动备选, 这组扰动操作根据漏洞数据库上公开披露的漏洞及其补丁所总结的特征设计而成, 是语义约束且漏洞保留的;

在第 2 阶段, 将漏洞代码样本, 利用词向量嵌入方法(Word2Vec)[30]转化为代码的词向量表征, 将其作为强

化学习模型的输入, 利用模型中的π网络输出一个概率分布, 根据此概率分布抽样得到一个扰动操作的序号, 并根据此序号选取某一具体的扰动操作;

在第 3 阶段, 利用程序分析工具 Tree-Sitter 将漏洞代码样本解析为语法树, 并且逐一遍历语法树的节点, 根据类型选取语法树节点作为应用扰动操作的潜在位置, 并提取语法树子树上的关键变量和关键字面量作为生成扰动内容的语料. 在潜在的代码扰动位置上, 根据所生成的扰动内容对原始代码进行转换, 从而生成对抗样本.

以上步骤完成了对抗样本的生成. 因为一个漏洞代码样本可能存在多个潜在的扰动位置, 在每个潜在扰动位置上进行代码转换均会生成一个对抗样本, 因此所生成的对抗样本会组成一个样本集合. 在攻击阶段, 将对抗样本作为目标模型的输入进行漏洞检测, 并选取结果偏移最大的样本作为最优对抗样本. 根据最优对抗样本的检测结果判断此轮攻击是否完成: 若攻击未完成, 则利用本次生成的最优对抗样本替换最初的漏洞代码样本, 并重复第 2 阶段和第 3 阶段的步骤. 第 3.1—3.3 节描述了本方法的 3 个实现步骤.

3.1 设计代码扰动操作

本工作利用一系列代码扰动操作来指导输入的样本进行代码转换得到对抗样本. 本工作的代码转换有两个要求: (1) 不能改变原代码的语义; (2) 必须保留原代码中的漏洞. 因此, 本文将这些操作定义为具备语义约束和漏洞保留特点的等价扰动操作. 同时, 为了满足转换后的代码能够被目标模型误检测为良好代码, 扰动操作需要给代码带来一定的“无漏洞”的特征以绕过目标模型.

通过对 NVD 所披露的真实漏洞及其补丁的代码特征进行分析, 本文总结了代码补丁对漏洞代码的修改模式. 在被分析的 20 个真实漏洞中, 18 个漏洞在其补丁代码里增加了“安全检查”和“异常处理”的代码语句, 其余 1 个补丁在临界区添加了对自旋锁的请求, 另一个补丁重写了存在漏洞的整段代码. 这些补丁语句均出现在引发漏洞的代码语句之前, 对变量进行修正、安全检查和异常处理. Lu 等人[17]定义了安全检查的识别规则, 认为安全检查位于容易出现漏洞的关键语句之前, 而被检查的变量即为关键变量.

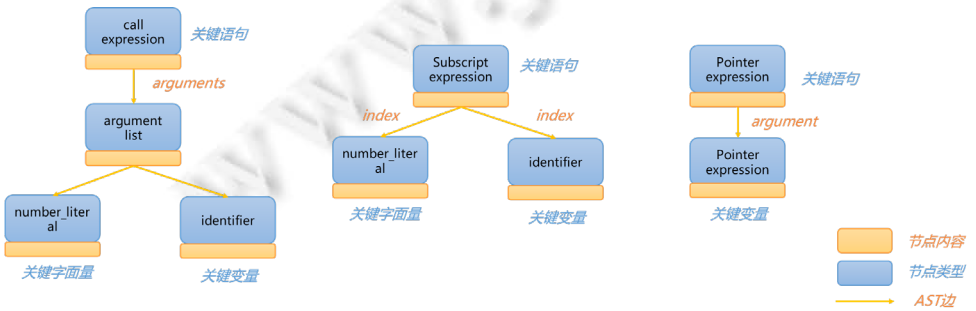
综上所述, 本文总结了补丁语句出现的潜在位置、潜在对象和潜在操作.

潜在位置: 容易出现漏洞的关键语句之前. 本工作代码识别“函数调用语句”“数组访问语句”和“指针引用语句”作为关键语句, 其中, “数组访问语句”对应了“数组访问越界”的漏洞类型, “指针引用语句”对应了“空指针引用”“对象字段异常”和“释放后使用”的漏洞类型, 而“函数调用语句”根据函数类型和功能对应了多种常见的漏洞类型;

潜在对象: 关键语句中的关键变量(含字面量), 本工作识别数组的访问下标、指针变量、函数调用的实际参数中的变量和字面量作为关键变量;

潜在操作: 在关键语句之前对关键变量进行修改、安全检查和异常处理.

图 3 展示了“函数调用语句”“数组访问语句”和“指针引用语句”的部分语法树结构, 并标识了树结构中的关键语句、关键字面量和关键变量节点.



(a) 函数调用语句 (b) 数组访问语句 (c) 指针引用语句

图 3 关键语句的语法树结构

本文所定义的扰动操作借鉴了漏洞补丁的思路, 目的是在原始代码样本上添加具备补丁特征的扰动. 与真实的漏洞补丁不同, 扰动操作所添加的关键变量修改、安全检查和异常处理是“虚假”的: 对于前者, 本文只改变字面量类型的关键变量, 并且使得更改后的字面量值与更改前保持一致, 例如将字面量 4 转变为字面量表达式“(2+2)”; 对于后两者, 本文通过改变条件语句, 使得“安全检查”的条件恒满足, “异常处理”的条件恒不满足, 以此来保证根据扰动操作所转换后的代码仍然与原代码具备语义和漏洞上的一致性.

表 1 总结了本文所定义的所有扰动操作, 第 1 栏列出了扰动操作的序号和名称, 第 2 栏列出了扰动操作的类型, 第 3 栏简要描述了每个扰动操作的定义, 第 4 栏显示了扰动前的代码片段示例, 最后一栏给出了扰动后的代码片段示例. 本文将在第 3.3 节中描述了如何根据这些扰动操作的规则对代码进行转换.

表 1 扰动操作表

名称	类型	描述	扰动前示例	扰动后示例
op-1 变量类型升级	常规扰动	升级变量类型	float *a*;	double *a*;
op-2 类名替换	常规扰动	类名,结构体名转化为自定义名称	structa {…}	struct *b* {…}
op-3 二元运算转换	常规扰动	替换等价的二元运算语句	*a*>*b*;	*b*<*a*;
op-4 控制语句转换	常规扰动	替换等价的控制语句	switch (*a*) {case 1: …}	if (*a*==1) {…}
op-5 无关信息打印	常规扰动	打印与代码无关语句	/	*print*(“*NoVulnerability*!”);
op-6 字面量替换	补丁特征扰动	关键字面量替换为变量	*add*(*a*+5);	*int b*=5, *add*(*a*+*b*)
op-7 字面量拆分	补丁特征扰动	关键字面量拆为等价字面量表达式	*add*(*a*+5);	*add*(*a*+(2+3));
op-8 安全检查	补丁特征扰动	在关键语句前添加虚假安全检查代码	*free*(*p*);	if (*p*==*p*){*free*(*p*);}
op-9 异常处理	补丁特征扰动	在关键语句前添加虚假异常处理代码	*free*(*p*);	If (*p*!=*p*){*exit*(−1);} *free*(*p*);

本文共定义了 9 个扰动操作, 覆盖了对代码中变量类型、运算语句、控制语句、字面量类型及无关语句的修改和增添. 表 1 所示的 9 个扰动操作中, op-1–op-5 是代码混淆等任务中常见的、具备代表性的代码等价转换, 这些常规扰动操作的设计参考了现有的代码扰动工作[10,11]; op-6–op-9 是本工作所提出的具备补丁特征的扰动操作. 这些代码的扰动操作均是轻量且细粒度的, 从扰动粒度的角度进行区分, op-1–op-3, op-6 和 op-7 是词级的扰动操作, 而 op-4, op-5, op-8 和 op-9 是语句级的扰动操作. 表中扰动操作对代码的改变只在词法和语法层面上进行, 不改变程序代码的执行流程和调用关系. 表 1 所显示的扰动操作对代码的修改有两种方式.

1) 代码替换: 这种替换是等价的, 例如 op-6 和 op-7 将字面量替换为相同的值, op-3 将运算语句中的运算符和操作数替换为等价形式, op-4 将 swith-case 语句等价替换为 if-else 语句;
2) 代码添加: 这种添加与原代码是无关的, 例如 op-5 只打印了“No Vulnerability!”这一字符串, op-8 和 op-9 所添加的 if 语句, 设置了特别的条件判断, 不改变原代码的执行.

因而, 上述扰动操作不会对程序代码的控制流、数据流和调用关系进行实质的改变, 从而保证了转换后的代码仍然保持与原代码语义和漏洞上的一致性.

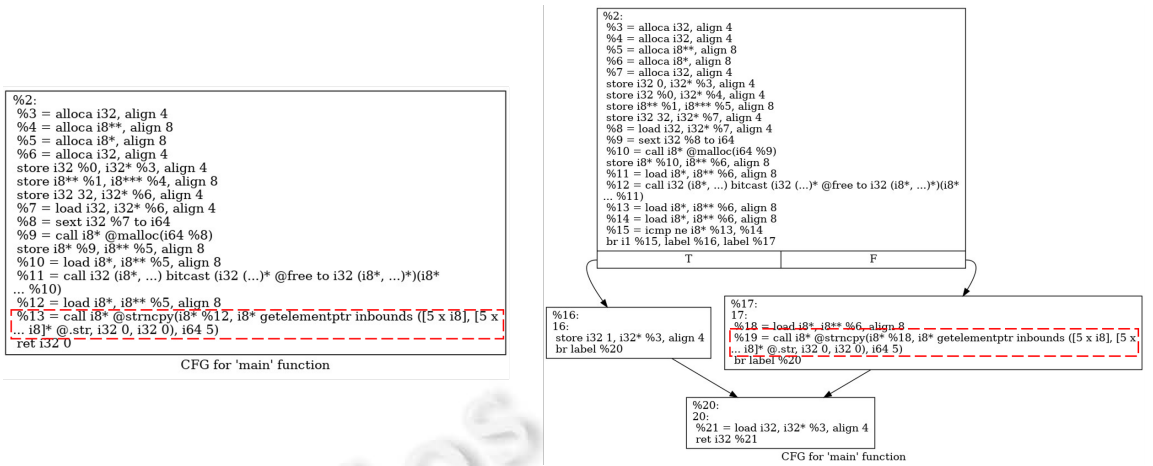
本文参考代码克隆工作[31], 将代码的一致性分为 4 种类型.

1) 类型 I: 除注释、缩进和布局外, 两个代码片段基本相同;
2) 类型 II: 除了类型 I 所导致的差异外, 两个代码片段之间有且仅有标识符、字面量、类型的差异;
3) 类型 III: 除类型 I 和类型 II 所导致的差异外, 两个代码片段还存在细微的修改, 如更改、添加、删除或重新排列语句;
4) 类型 IV: 类型 IV 只保留语义一致性. 因此, 两个代码片段可能功能相似, 但代码结构不同.

类型 I~类型 III 针对代码的文本的一致性, 而类型 IV 针对代码的语义一致性. 具体而言, 对于具备漏洞的代码片段, 其语义一致性表现为: 修改后的代码不改变原代码中漏洞的触发. 这表明修改后的代码执行结果相同, 具备相同的功能, 表现出语义的一致性. 本文所定义的扰动操作所生成的修改后代码均符合上述代码一致性类型, 其中, op-1, op-2, op-6, op-7 属于类型 I; op-3–op-5 属于类型 II; op-8, op-9 属于类型 IV. 图 4 展示了图 1 案例中代码的控制流图(control flow graph, CFG)的对比.

由于图 4(b)中的变量%13 和%14 相等, 控制流执行%17 基本块, 因此扰动前后的代码均会执行触发漏洞

的语句, 即代码执行结果相同. 经过 op-9 的扰动, 原代码中触发漏洞的语句执行的结果并未发生改变. 因此, 根据 op-9 所生成的修改后的代码符合类型 IV 关于代码一致性的定义.



(a) 原控制流图 (b) 扰动后控制流图

图 4 原代码与扰动后代码的控制流图对比

3.2 扰动操作决策模型

攻击方法接收一个代码样本作为输入后, 需要有效的策略来指导扰动操作的选择, 从而让对抗样本以更大的可能性绕过目标模型的检测.

选择扰动操作可以被视为一个最优化的搜索问题. 与常规的搜索问题不同, 本工作对最优扰动操作的搜索目标是生成对抗样本, 以绕过目标模型的漏洞检测. 因此, 优化目标需要考虑目标模型的反馈, 而常规的优化算法, 例如遗传算法和蒙特卡洛算法所设置的优化目标是泛化的, 并没有考虑到被评估的目标模型的反馈. Zhang 等人[10]在针对代码相似性检测模型对抗攻击的实验结果表明, 不考虑目标模型反馈的优化算法所取得的效果并不佳. 基于遗传算法和蒙特卡洛算法的优化策略, 相较于随机进行扰动的策略在目标模型的 $F1$ 指标上, 只取得相同得分或 0.002 的得分优势.

本工作提出一种基于强化学习的策略方法, 将待攻击的目标模型的反馈纳入本文的优化策略之中, 以更加高效地生成能绕过目标模型漏洞检测的对抗样本. 根据第 1.2 节的定义, 本文的攻击方法是基于黑盒访问的, 但攻击方法仍可以获取目标模型的输出结果作为反馈. 并且本文在实验环节证明, 考虑目标模型的反馈在策略的优化方面是有效的.

在强化学习中, 模型所学习的策略是在某个给定状态(state, s)下应该采取什么行动(action, a). 这个决策的过程由一个神经网络实现, 被称为π网络. 该网络输出一个概率分布, 并基于此概率分布抽样获取特定的行动 a 作为决策结果. 用θ表示π网络的参数. 奖励函数(reward function)决定了在状态 s 下采取行动 a 能够获得的奖励值(reward, r), 奖励值通常是一个数值, 奖励值的累加和被称为收益或回报(return). 接下来会从模型结构和训练目标两个方面对本文所采取的强化学习模型的具体设计进行描述.

(1) 模型结构

如图 5 所示, 本文所述强化学习模型由代理(agent)和环境(environment)组成. 其中, 代理即π网络, 功能是实现决策, 根据当前的状态 s_t 选择具体的动作 a_t; 而环境的功能是根据所选的动作 a_t 进行相应的响应, 转移到新状态 s_{t+1}, 同时计算一个奖励值 r_t. 强化学习模型的输入是代码样本经过处理后的词向量表征, 以此表示当前的状态 s_t, 代理会根据代码表征 s_t 从预定义的扰动操作表中选取一个扰动操作, 作为动作 a_t. 环境会根据所决策的扰动操作 a_t 进行代码转换, 生成新的代码表征 s_{t+1}, 并利用目标模型对该代码表征 s_{t+1} 进行漏洞检测, 最后根据目标模型的检测结果计算当前决策的奖励值 r_t. 图中的 h_t 代表π网络中长短期记忆网络(long short

term memory, LSTM)中第 t 时间步的隐藏状态(hiddenstate), 用以和输入状态 s_t 共同计算 LSTM 第 t+1 步的输出 o_{t+1}. 而π网络中的线性层和 *softmax* 函数根据 LSTM 最后一个时间步 t=n 的输出 o_n 采样出动作 a_t.

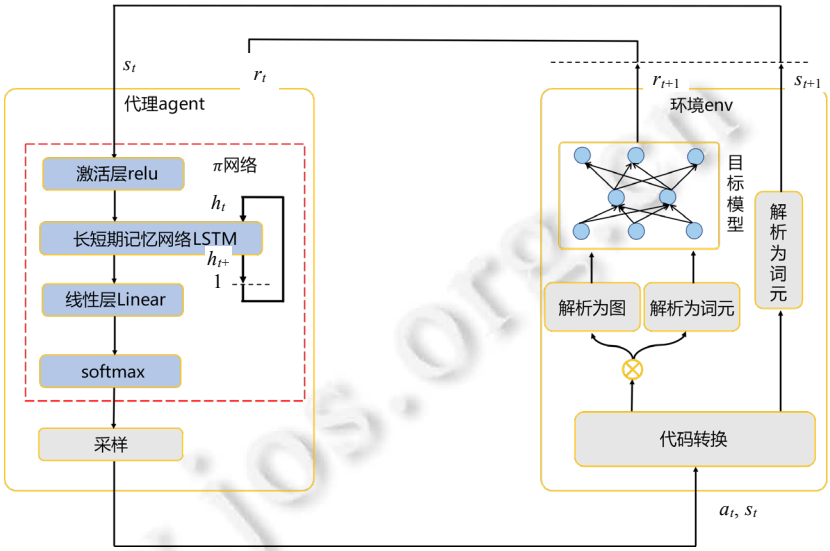


图 5 强化学习模型结构

下面给出本模型中动作空间、状态、抽样方法和奖励的描述.

(1.1) 动作空间(action space)

强化学习模型中, 代理会从给定的动作空间中, 计算概率分布并抽样以选取动作. 本工作使用第 3.1 节所定义的扰动操作集合作为模型的动作空间, 这些扰动操作显示在表 1 中.

(1.2) 状态(State)

本工作使用代码样本的词向量表征来表示强化学习模型的状态. 对一个给定的代码样本, 本文首先将其处理为词向量, 并用 Word2Vec 对其进行编码, 从而得到代码的嵌入(embedding)表征. 环境根据扰动操作进行代码转换后, 得到的对抗样本同样是此类型的代码表征;

(1.3) 抽样方法(sampling method)

代理中的π网络学习如何根据当前的代码样本选取合适的扰动操作, 该网络的输出是关于扰动操作的概率分布, 需要依分布抽样得到具体的扰动操作. 因此, 本文采用抽样方法对π网络输出的概率分布进行抽样. 在此, 本文将介绍选取扰动操作所采用的 3 种抽样方法——NoSample, Sample 和 ProbSample.

NoSample, 即不进行抽样, 直接选取概率最大的扰动操作. 由于迭代过程中对抗样本的变动很小, 因此根据样本特征所输出概率分布相似, 采取 NoSample 方法很可能会陷入重复选取同一扰动操作的陷阱;

Sample, 为了克服 NoSample 的局限性, 在给定代码样本的情况下, 对针对该样本的扰动操作进行抽样. 本文采取 Top$K$ 抽样方法, 即, 在概率前 K 大的扰动操作中随机选取;

ProbSample, 基于 Top$K$ 的抽样方法同样忽略了那些输出概率较小的扰动操作, 因此本文提出利用依概率抽样的方法来选择扰动操作, 不仅保证了能够高效地选取合适的扰动操作, 也避免陷入重复选取无效扰动操作的情形;

(1.4) 奖励(reward)

模型的环境会根据奖励函数来计算动作 a 所带来的奖励, 奖励之和被称为收益, 而模型的最终目标是为了最大化该收益. 本文的决策目标是为了产生能够绕过目标模型漏洞检测的对抗样本, 为此, 本文所设计的每一个步长 t 的奖励函数如下面公式所示:

$$R_i(t)=\begin{cases}R_{adv}, & f(s_{t+1})<0.5\\ i*(f(s_t)-f(s_{t+1})), & f(s_{t+1})\geqslant 0.5\end{cases} \tag{5}$$

$$\beta=\begin{cases}100, & f(s_t)-f(s_{t+1})<0\\ 10, & f(s_t)-f(s_{t+1})\geqslant 0\end{cases} \tag{6}$$

其中,f表示目标模型F的神经网络. 给定模型F的输入样本,f计算后输出一个概率q, 当q<0.5时, 代表模型认为该样本不存在漏洞. 而奖励函数由两个部分组成, 当$f(s_{t+1})$<0.5时, 对抗样本s_{t+1}绕过了目标模型的漏洞检测, 本文给予一个大的正值奖励R_{adv}并终止当前的学习过程, 代表此时本文已经得到一个有效的代码扰动操作序列Δ; 当$f(s_{t+1})\geqslant$0.5时, 本文计算目标模型对原代码表征s_t和对抗样本代码表征s_{t+1}检测结果的差值为$i*(f(s_t)-f(s_{t+1}))$, 该值反映了对抗样本在逃避模型检测的进步效果.

i代表修正参数, 当i=1时, 表示不进行参数修正. β代表一个具体的参数, 因为检测结果差值常为小数点后3位的小数, 利用β能放大奖励值. 当$f(s_t)-f(s_{t+1})<0$时, 代表当前对抗样本取得较差的检测结果, 本文加大对它的惩罚, 这是为了提高模型的学习效率, 使模型尽快完成收敛.

(2) 训练目标

将强化学习任务中每一步(step)出现的状态s和动作a按序组成的序列称为轨迹, 每个轨迹出现的概率为

$$p_\theta(\tau)=p(s_1)\prod_{t=1}^{T}p_\theta(a_t\mid s_t)p(s_{t+1}\mid s_t,a_t) \tag{7}$$

强化学习的目标是: 调整π网络的参数θ, 促使π网络做出正确的决策, 使得最终的收益尽可能的大. 计算收益函数R的期望, 并利用梯度上升法来修正参数θ:

$$\theta\leftarrow\theta+\eta\nabla\overline{R_\theta} \tag{8}$$

$$\overline{R_\theta}=\sum\nolimits_\tau R(\tau)p_\theta(\tau) \tag{9}$$

本工作中所采用的近似策略优化(proximal policy optimization, PPO)[14]算法是对于置信域策略优化(trust region policy optimization, TRPO)[32]的改进, 是对上述梯度算法的一个变种. π网络根据参数θ'完成动作的决策之后, 参数会被更新为θ, 因此, 根据梯度修改的参数θ和最初进行决策所使用的参数θ'并不相同. 这种代理进行决策所使用的参数和根据梯度进行学习时使用的参数不相等的情况, 被称为非一致策略(off-policy). 常见的解决方法是, 通过重要性抽样(importance sampling)对公式进行修改以应对此情况. 本文根据最终收益函数计算一个损失函数(loss function), 该损失函数和最终收益函数呈负相关; 然后, 基于该损失函数采取梯度下降算法, 修改参数θ:

$$Loss=-\min\left(\frac{p_\theta(\tau)}{p_{\theta'}(\tau)}R(\tau),clip\left(\frac{p_\theta(\tau)}{p_{\theta'}(\tau)},1-\varepsilon,1+\varepsilon\right)R(\tau)\right) \tag{10}$$

其中, $p_\theta(\tau)/p_{\theta'}(\tau)$代表修正新旧参数$\theta$和$\theta'$之间的差距的比值, 以此计算真实的收益期望. 当新旧参数θ和θ'取得同一轨迹τ的概率差别过大时, 利用$clip$函数约束用以更新参数的梯度大小, 以此控制更新的步长, 增加训练的稳定性. 其中, ε代表截取的范围.

3.3 对抗样本生成

本节将介绍根据扰动操作进行代码转换从而生成对抗样本的过程. 本文在第3.1节中预定义的扰动操作规定了进行代码转换的潜在位置、潜在对象和潜在操作, 因此进行代码转换生成对抗样本的过程可以分为如下3个步骤: (1) 寻找潜在的扰动位置; (2) 利用潜在对象生成扰动内容; (3) 对扰动位置原内容进行替换、删除或添加. 每个潜在的扰动位置都会生成一个新的扰动样本, 将其作为备选对抗样本. 所有的备选对抗样本成为一个样本集合, 根据第3.1节和第3.2节中对代码表征和扰动操作的定义, 将该集合表示为

$$\delta(s_t,a_t)=\{s_1,s_2,\ldots,s_i\} \tag{11}$$

其中, t表示强化学习模型的当前步长, i表示该集合内样本的数量. 将该集合内的样本作为目标模型的输入,从

而得到目标模型对它们的检测得分, 选取得分最佳的样本作为最佳对抗样本 s_{t+1}:

$$s_{t+1}=s^i,\ i=\text{argmin}_i f(s^i) \tag{12}$$

最后, 根据最佳对抗样本的得分情况 $f(s_{t+1})$和当前扰动操作序列Δ的长度判断是否完成对目标模型的攻击. 若攻击未完成, 则将当前步骤生成的对抗样本作为决策模型的输入, 进行迭代攻击.

本文基于代码解析工具 Tree-Sitter 实现代码的转换. Tree-Sitter 是一个代码解析工具, 它能够将一个源代码解析为具体的语法树. 并且 Tree-Sitter 具备健壮性, 即使在代码有漏洞和语法错误的情况下, 也能够提供有效的解析结果. 因其面对漏洞的健壮性优势, Tree-Sitter 成为本工作漏洞代码的解析工具.

图 1 展示了根据扰动操作进行代码转换所生成对抗样本的案例. 以此为例, 并根据图 6 展示的定位过程, 介绍本工作的代码转换方法. 首先, 决策模型根据代码(a)输出一个动作 a=9, 在表 1 中根据序号选取了扰动操作 op-9. 其次, 利用 Tree-Sitter 将代码转化为结构化的语法树表示, 语法树上的节点保留了类型(type)字段, 该字段代表了节点的语法类型. 逐一遍历该语法树上的节点, 根据类型选取语法树节点作为应用扰动操作的潜在位置. 本案例中, 根据类型“*call_expression*”类型定位到关键语句——函数调用语句“*strncpy*(*buf*1,‘*hack*’,5);”并将其作为潜在的扰动位置. 再次, 在语法树的子树上提取关键变量作为生成扰动内容的潜在对象, 函数调用语句的关键变量是函数实参中的变量和字面量. 于是, 在子树中提取到标识符“*buf*1”作为关键变量. 最后, 根据 op-9 的扰动规则, 利用关键变量“*buf*1”生成虚假的异常处理语句“if (*buf*1!=*buf*1) {*exit*(−1);}”并将其添加至扰动的潜在位置, 从而生成扰动后的代码(b).

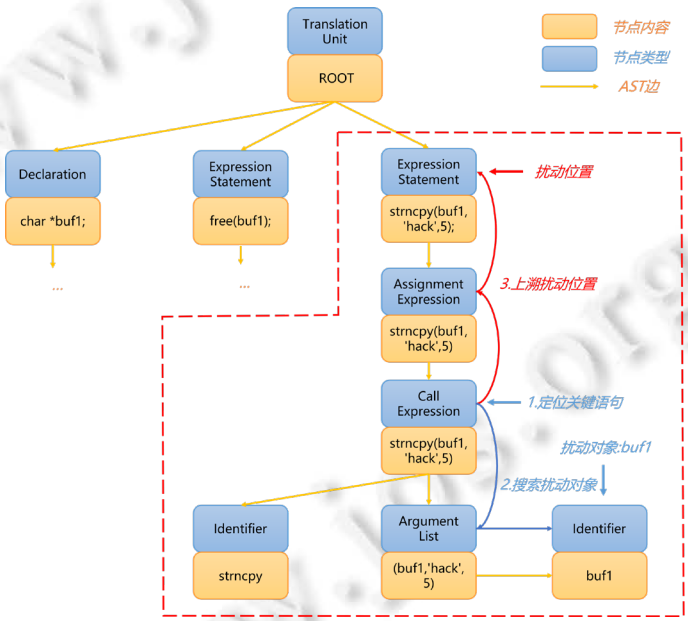


图 6 代码扰动定位示例图

4 实验评估

本节将对实验环节进行阐述. 首先对实验准备工作进行说明, 包括实验数据集、目标模型、评价指标和实验环境; 其次, 分别对本文中进行的实验内容进行阐述, 并对实验结果进行分析. 特别地, 为了研究本文提出的对抗攻击方法的有效性, 本文希望通过实验评估来回答如下 4 个研究问题(research questions, RQ).

- RQ1: 本文提出的强化学习式对抗攻击方法对现有的基于深度学习的漏洞检测模型的攻击效果如何? 与基线方法相比是否有优势?
- RQ2: 本文在第 3.1 节中所设计的扰动操作在生成对抗样本的任务中是否有效? 不同类型扰动操作的

效果如何?

- RQ3: 本文在第 3.2 节中提出的强化学习模型在扰动操作的决策上是否有效? 与随机策略相比是否有优势?
- RQ4: 本文所采取的不同抽样方法和奖励函数对实验结果有什么影响? 应该如何选取抽样方法和奖励函数?

4.1 实验准备

4.1.1 数据准备

本工作需要一个能满足以下要求的漏洞代码数据集: (1) 数据集中的漏洞代码可以被解析用以代码转换; (2) 数据集中的漏洞应该贴近真实世界软件中的漏洞. 基于以上考虑, 本文基于如下两个被广泛采用的漏洞数据源构造了实验数据集: 软件保障参考数据集(software assurance reference dataset, SARD)和美国国家漏洞数据库(national vulnerability database, NVD). SARD 数据集中包含了不同代码形式(源代码或二进制代码)、不同语言(C, Java, Python 等)、不同漏洞类型(缓冲区溢出、资源管理错误、注入等)的漏洞数据. SARD 的样本丰富、标注准确、权威性高, 并且有良好的标签: 每个程序都被标记为好(无漏洞)、坏(有漏洞)或混合(有漏洞的功能及其补丁版本). SARD 中的大量样本通过变量数据类型转换、添加无效控制流、多语句提取为函数等代码转换生成, 样本间存在相似性. NVD 中包含软件程序中的真实漏洞, 并且 NVD 中每一个漏洞样例都由一个独特的公共漏洞和暴露编号(common vulnerabilities and exposures identity document, CVE ID)所标识, 部分漏洞包括其补丁文件或用差异文件来描述漏洞代码和补丁之间的不同.

本工作的目的是针对漏洞检测模型进行黑盒对抗攻击. 本文认为在真实场景下, 漏洞代码会在不同软件版本中进行代码修改和更新, 这与 SARD 数据集使用代码修改生成数据的模式类似. 而 NVD 数据库则由所披露的真实历史漏洞所构成. 因此, 本文实验数据集中的代码在一定程度上和具备真实漏洞的代码有相似性.并且由于数据集中存在经过转换生成的相似代码样本, 这与通过代码转换而生成对抗样本的情况相似, 待攻击的目标模型在此数据集上训练可以提高对于对抗样本的防御阈值, 以证实本方法的攻击有效性.

本文选取了 SARD 中 C/C++语言的资源管理错误类型(CWE-399)的漏洞数据集中 C 语言源程序子集作为数据集 D_{SARD}. 资源管理错误类型是 C/C++语言常见的漏洞类型, 并且该类型拥有包括 CWE-122, CWE-121 和 CWE-789 等多种常见的子类型, 涵盖多种代码结构, 可以代表不同类型的漏洞代码. 表 2 展示了数据集内样本的数量. 同时, 参考以往的漏洞检测工作[1,2,20], 本文选取了 NVD 中的部分 C/C++开源软件的漏洞程序作为数据集 D_{NVD}, 其中包含了 Linux Kernel, QEMU 和 FFmpeg 等多款软件程序的代码文件. 由 D_{SARD} 和 D_{NVD} 共同构成了本文的实验数据集.

表 2 数据集中的数据量

数据集	样本总数量	漏洞样本数量	非漏洞样本数量
D_{SARD}	12 611	4 154	8 457
D_{NVD}	1 667	1 347	320

4.1.2 目标模型

基于深度学习的漏洞检测模型可以通过如下 3 个维度进行分类: (1) 检测粒度, 例如程序切片级和程序函数级等; (2) 基于何种代码表征, 例如基于词元序列、基于抽象语法树、基于代码属性图(code property graph, CPG)等; (3) 基于何种神经网络类型, 例如基于双向门控循环单元(bidirectional gate recurrent unit, BGRU)、双向长短期记忆网络(bi-directional long short-term memory, BLSTM)和图神经网络等. 其中, 检测粒度代表漏洞检测模型可以在代码的哪个层级上检测出漏洞, 代码表征代表漏洞检测模型在数据预处理阶段将代码转换为何种数据结构的表示作为模型的输入, 神经网络类型代表漏洞检测模型内部的神经网络是何种结构.

本方法考虑选取开源的、检测效果良好并且在以上 3 个维度的类别各异的漏洞检测模型作为目标模型, 这让选取的目标模型更具代表性.

基于以上考虑, 本文选取了 Devign[3], SySeVR[2], VulSniper[4]和 VulDeepecker[1]这 4 个漏洞检测模型作为

目标模型. 接下来, 本文将根据所提出的 3 个维度, 介绍目标模型的信息.

1) Devign: 程序函数+CPG+GNN. Devign 将代码编码为以 AST 为中心、融合不同级别的数据依赖和控制依赖的联合图, 以完成对代码语义信息的完整提取, 并使用 GNN 学习代码表征进行函数粒度漏洞预测;
2) SySeVR: 程序切片+词元序列+BGRU. SySeVR 将代码提取为抽象语法树, 并遍历抽象语法树的节点, 根据规则生成语法漏洞候选(SyVCs)和语义漏洞候选(SeVCs), 这两个候选均由代码的词元组成. 并使用 BGRU 网络对 SeVCs 提取特征, 并进行切片粒度的漏洞检测;
3) VulSniper: 程序函数+特征张量(基于 CPG)+注意力神经网络. VulSniper 将代码提取为 CPG, 并将 CPG 转换为节点数为 128、编码为 128*128*144 的特征矩阵, 其中, 144 为设定的 144 种特征关系(使用 0, 1 表示). 并使用具备注意力机制的神经网络对该特征矩阵进行学习, 在函数粒度预测漏洞;
4) VulDeePecker: 程序切片+词元序列+BLSTM. VulDeePecker 定位程序中的关键语句, 把与关键语句具备语义关联的代码语句组合形成程序切片, 并训练 BLSTM 网络进行漏洞检测.

本文利用 D_{SARD} 和 D_{NVD} 这 2 两个数据集对 4 个目标模型进行了训练和评估, 遵循其论文所描述的训练步骤, 采取了作者建议的参数, 以此得到训练好的目标模型. 本文用准确率(precision, *P*)、召回率(recall, *R*)和 *F*1 分数(*F*1-score, *F*1)来评估模型的训练效果, 结果见表 3.

表 3 目标模型在实验数据集的检测效果

检测模型	数据集	*P* (%)	*R* (%)	*F*1 (%)
Devign	D_{SARD}	92.09	76.87	83.79
	D_{NVD}	53.48	64.74	58.57
VulSniper	D_{SARD}	80.40	70.66	75.22
	D_{NVD}	47.60	66.07	55.33
SySeVR	D_{SARD}	90.04	46.71	61.51
	D_{NVD}	55.14	67.19	60.57
VulDeepecker	D_{SARD}	84.05	61.99	71.35
	D_{NVD}	52.66	37.05	43.50

Devign 和 VulSniper 在数据集 D_{SARD} 上的表现良好, 其 *F*1 得分均高于原论文实验中最佳 *F*1 得分. 而 SySeVR 和 VulDeepecker 因为召回率值较低的原故, *F*1 得分不高. 同时, 4 个目标模型在 D_{NVD} 上的检测效果相较 D_{SARD} 有所下降. 这是因为 D_{NVD} 中的样本来自真实漏洞代码, 其代码量更大, 数据更为复杂, 这与以往漏洞检测工作[20]所展示的结果具有相似性. 但应该注意的是: 本方法只会选取目标模型能够准确识别的漏洞样本作为待攻击数据集 $D_{detected}$, 因此模型的召回率指标对攻击方法的影响不大. 相反, 4 个模型在准确率的得分上表现良好, 这反映出模型能够在很大程度上避免误报情况的产生, 模型的检测能力不会受到相似代码的混淆. 因此, 本文认为: 训练后的 4 个模型是训练情况良好, 具备漏洞检测能力的目标模型.

4.1.3 评价指标

本文考虑设置合理的评价指标来评估本方法所生成对抗样本的攻击有效性和质量这两个方面的情况.

针对攻击有效性, 参考了其他代码形式对抗样本生成工作所使用的指标[12], 本文定义了如下两个指标.

1) Δ_{drop}: 测量攻击前后目标模型检测性能的下降. 定义Δ_{drop}:

$$\Delta_{drop}=Recall_{before}-Recall_{after} \tag{13}$$

其中, $Recall_{before}$ 和 $Recall_{after}$ 分别代表攻击前后, 目标模型对数据集中漏洞样本的召回率(查全率, *R*), 以此衡量目标模型检测含有漏洞的样本的能力是否下降;

2) 攻击成功率(attack success rates, ASR): 测量攻击方法的成功率. 给定目标漏洞检测模型, 本文选取数据集 D 中能够被目标模型正确识别的漏洞样本, 作为待攻击数据集 $D_{detected}$. 然后, 验证该数据集内的样本经过本文的攻击方法后能否逃避目标模型的检测. 将成功逃避检测的样本数量与待攻击数据集内样本数量的比值视为攻击成功率, 该值越高, 本攻击方法就越有效.

针对生成对抗样本的质量, 本文定义如下指标.

扰动序列长度(perturbation length, *#Perturb*): 测量攻击方法的效率和隐蔽性.

本文利用强化学习模型指导对抗样本迭代生成, 每一次扰动会对原代码样本进行转换生成新的代码样本, 扰动序列的长度反映了每个对抗样本所采取的扰动操作次数, 也即代码转换的次数. 在黑盒访问的攻击环境下, 强化学习模型每次选择扰动操作后都要查询目标模型的结果以计算收益, 因此, 扰动序列的长度也反映了每个对抗样本所需的查询次数. 本文用扰动序列长度的均值来衡量对抗样本的质量, 该值越低, 攻击就越难以察觉, 攻击效率也就越高.

4.1.4 基线对抗攻击方法

目前, 尚未有针对漏洞检测模型已开源的、自动化的黑盒对抗攻击方法. 目前, 相关领域的工作中, Li 等人[13]未针对漏洞检测模型提出专门的攻击方法, 并且需要领域专家通过人工检查来保证转换后的代码是否保留了语义和漏洞. 而针对漏洞检测模型的对抗攻击, 要求攻击方法的解析阶段能兼顾非完整的漏洞代码, 因此, 其他代码领域的对抗攻击方法不适用于漏洞检测任务. 本文选取了针对代码预训练模型的攻击方法 ALERT[28]作为基线方法, 该工作主要针对预训练代码模型 CodeBERT 和 GraphCodeBERT, 具备良好的攻击效果; 同时,可以很好地兼容包含漏洞检测在内的多个下游任务. 本文在第 4.2 节中进行了对比实验和结果分析.

4.1.5 实验环境及参数设置

本文中涉及的实验在配有 Nvidia Tesla P100 PCIE 12 GB GPU, Intel Xeon Silver 4116 CPU@2.10 GHz CPU 和 64 GB 内存的设备上进行.

本文在 D_{SARD} 和 D_{NVD} 数据集上训练了 4 个目标模型. 遵循其论文所描述的训练步骤, 采取了论文中作者建议的参数.

同时, 针对每个目标模型, 训练了攻击方法中的强化学习决策模型. 将强化学习模型奖励函数中的正值奖励 R_{adv} 设置为 20, 将计算损失函数公式(10)中 *clip* 函数的截取范围ε设置为 0.1, 将优化器的学习率设置为 0.000 5, 将扰动序列的最大长度 N 设置为 30.

4.2 实验结果与分析

(1) RQ1

为了证明本文攻击方法的有效性, 本文对 4 个目标模型进行了攻击实验. 首先, 给定一个训练好的目标模型, 本文选取它在实验数据集上能够正确检测出的漏洞代码样本集合, 将这些样本划分为目标模型的可检数据集 $D_{detected}$; 其次, 从每个目标模型的可检数据集 $D_{detected}$ 随机选取 80%的数据进行攻击方法中强化学习模型网络的训练, 用剩余 20%的数据进行评估.

表 4 展示了 Devign, VulSniper,SySeVR 和 VulDeepecker 在 D_{SARD} 和 D_{NVD} 数据集上经过本方法攻击后的评估指标, 分别展示了攻击后模型的召回率(括号内为模型召回率的下降值)和攻击成功率. 面对本方法所生成的对抗样本, 4 种目标模型的召回率都发生了一定程度的降低. 基于 D_{SARD} 数据集的实验结果中, 指标Δ分别为 74.34%, 64.54%, 12.25%和 20.42%. 其中, Devign 的召回率从 76.87%下降到 2.53%, 是最脆弱的. 从攻击成功率来看, 针对 4 个模型, 本方法攻击成功率分别达到 96.71%, 91.33%, 26.22%和 32.93%. 表明绝大部分的对抗样本都可以绕过 Devign 和 VulSniper 的检测, 有超过 1/4 的对抗样本可以绕过 SySeVR 和 VulDeepecker 的检测. 基于 D_{NVD} 数据集的实验结果中, 指标Δ分别为 45.96%, 59.47%, 5.50%和 10.40%, 并且攻击成功率平均达到了 49.33%. 本方法在指标上均优于基线方法 ALERT. 针对 VulSniper 模型, ALERT 成功地进行了攻击, 本文方法较 ALERT 提高了 66.16%和 68.76%的攻击成功率, 在两个实验数据集上的攻击效果均优于 ALERT. 此外, 表格显示了 ALERT 针对 Devign, SySeVR 和 VulDeepecker 的攻击成功率均为 0. 这是因为此类模型在数据的预处理阶段都进行了代码标识符的标准化, 即将变量和函数的命令映射为统一的形式, 而 ALERT 的攻击方法仅仅针对代码标识符进行替换, 这导致 ALERT 对部分模型的攻击是无效的. 本文所提出的攻击方法包含多种类型的扰动操作. 针对目标模型而言, 每个待检测的代码对在检测结果上具备极大的差异性, 但在代码的文本和语义层面具备极大的相似性. 因此, 本工作通过轻量级的扰动达成了很好的攻击效果. 总之, 本方法可以针对目标模型进行高效且隐蔽的有效攻击, 揭示了目标模型的脆弱性.

表 4 对目标模型攻击实验的结果

数据集	目标模型	本文方法		ALERT	
		$Recall_{after}(\Delta_{drop})$ (%)	*ASR* (%)	$Recall_{after}(\Delta_{drop})$ (%)	*ASR* (%)
D_{SARD}	Devign	2.53 (−74.34)	**96.71**	76.87 (−0)	0
	VulSniper	6.12 (−64.54)	**91.33**	52.89 (−17.77)	25.17
	SySeVR	34.46 (−12.25)	**26.22**	46.71 (−0)	0
	VulDeepecker	41.57 (−20.42)	**32.93**	61.99 (−0)	0
D_{NVD}	Devign	18.78 (−45.96)	**70.95**	64.74 (−0)	0
	VulSniper	6.60 (−59.47)	**90.00**	52.04 (−14.03)	21.24
	SySeVR	61.69 (−5.50)	**8.18**	67.19 (−0)	0
	VulDeepecker	26.65 (−10.40)	**28.18**	37.05 (−0)	0

通过本实验结果可以归纳出目标模型工作的薄弱点, 例如: VulSniper 利用注意力机制来捕捉漏洞的细粒度特征, 然而实验结果显示, 通过扰动在漏洞样本中添加简单的无关语句就能改变模型的检测结果. 这表明该模型神经网络所采取的注意力机制并不能正确地判断代码中何种特征决定了漏洞的存在, 神经网络的注意力关注了与代码漏洞无关的特征. Devign 利用一个卷积模块来筛选代码中有效的特征, 用于判断漏洞的存在. 实验结果表明: 该卷积模块仍然无法精准地排除本方法所添加的扰动特征, 即使这些特征并不对样本中的漏洞产生影响.

其次, 本文观察到: 两个基于代码属性图及其变种表征的漏洞检测模型(Devign, VulSniper)误判了更多的漏洞样本, 它们在攻击有效性和生成对抗样本的效率两种指标上都高于基于词元序列的模型(SySeVR, VulDeepecker), 其中, 攻击成功率分别高出 70.49%和 65.11%, 表明此类漏洞检测模型更容易被绕过. 本文推测, 这是由以下原因造成的:

为了避免对代码中存在的漏洞产生影响, 本方法定义的扰动操作避免了对标识符的替换, 大部分扰动操作改变代码中的表达式语句. 根据这些扰动操作对代码进行转换后, 相较于代码的词元序列, 代码属性图的变化更为明显. 因此, 基于代码属性图及其变种的漏洞检测器对对抗样本的变化更加敏感, 模型在攻击前后的评估指标变化更大. 此外, SySeVR 和 VulDeepecker 对代码进行了切片操作, 只保留了和触发漏洞的语句具备语义相关性的代码片段. 实验结果表明: 这有助于剔除冗余的代码信息, 对提升模型的鲁棒性是有效的.

综上, 实验结果表明: 本方法产生的对抗样本能够有效逃避目标模型的漏洞检测, 尤其针对基于代码属性图的目标模型, 能够达到 90%以上的攻击成功率.

(2) RQ2

为了证明本工作提出的具备补丁特征的代码扰动操作的有效性, 基于数据集 D_{SARD}, 本文将现有的常规扰动和本文提出的具备补丁特征的扰动进行对比实验. 具体来说, 参考代码混淆的相关研究[33], 本文将表 1 中的代码扰动操作分为了两组: (1) 常规扰动; (2) 具备补丁特征的扰动. 其中, op-1−op-5 为常规扰动, 剩余为具备补丁特征的扰动. 为了研究这两组扰动操作对漏洞检测模型的影响, 本工作利用 D_{SARD} 生成了两个新的漏洞样本集合: 只应用第 1 组操作的漏洞样本集合 D_{normal} 和只应用第 2 组操作的漏洞样本集合 D_{key}. 本文在目标模型的可检数据集 $D_{detected}$ 中随机选取 200 个样本, 对每个样本分别逐一应用两组扰动操作得到 D_{normal} 和 D_{key}. 本文不使用强化学习模型来指导扰动操作的决策, 而是直接将根据组内的扰动操作逐个进行代码转换, 这样做的目的是排除强化学习模型给两组扰动操作对比带来影响.

表 5 展示了两种不同的测试数据集面对目标模型的攻击成功率和所采取扰动序列的平均长度. 总的来说, 在不使用策略指导、仅逐一应用组内扰动操作的情况下, 两组扰动操作的攻击成功率的平均值分别为 36.38%和 47.00%. 针对扰动序列平均长度这一指标, D_{key} 在 3 个目标模型上都有更优的表现, 表明具备补丁特征的扰动操作在绕过模型的漏洞检测上表现更好.

综上, 实验结果表明, 相较于代码混淆等领域的所采取的常规代码转换操作, 本方法所设计的具备补丁特征的扰动操作能够更加有效地生成对抗样本以逃避目标模型的漏洞检测.

表 5 扰动操作有效性实验的结果

目标模型	测试数据集	*ASR* (%)	*#Perturb*
Devign	D_{normal}	53.00	4.20
	D_{key}	**68.50**	**2.94**
VulSniper	D_{normal}	75.00	4.16
	D_{key}	**79.00**	**2.03**
SySeVR	D_{normal}	5.00	4.33
	D_{key}	**13.50**	**3.62**
VulDeepecker	D_{normal}	12.50	3.92
	D_{key}	**27.00**	**4.03**

(3) RQ3

为了证明本工作所设计的强化学习模型能够更好地指导对抗样本的生成, 基于数据集 D_{SARD}, 本文评估了基于强化学习模型和基于随机算法的对抗攻击方法的差异. 具体来说, 本文设计了一个随机选择算法作为对比, 该算法每次会随机地从表 1 的扰动操作中选取一个扰动操作. 相对而言, 当给定一个漏洞代码作为输入时, 强化学习模型会根据输入, 利用 π网络决策一个最优的扰动操作. 本文生成了两个新的漏洞样本集合: 应用随机选择算法进行扰动操作决策的漏洞样本集合(D_{random})和应用强化学习模型进行扰动操作决策的漏洞样本集合(D_{rl}), 并使用这两个数据集对 3 种目标模型进行了评估.

表 6 显示了基于两种不同的决策算法所生成的对抗样本对 4 个目标模型的攻击结果, 分别展示了模型召回率的下降值、攻击成功率和所采取扰动序列的平均长度这 3 个指标. 在攻击有效性方面, 相较于基于随机选择算法的方法, 采取强化学习模型的方法在攻击成功率上的指标分别高出 16.27%, 28.83%, 7.20%和 3.16%. 在生成对抗样本的质量方面, 采取强化学习模型的方法同样更优, 其扰动序列的平均长度分别低了 6.02, 1.77, 8.97 和 2.79.

表 6 扰动决策算法有效性实验的结果

目标模型	攻击策略	Δ_{drop} (%)	*ASR* (%)	*#Perturb*
Devign	强化学习	**74.34**	**96.71**	**2.31**
	随机搜索	61.83	80.44	8.33
VulSpiner	强化学习	**64.54**	**91.33**	**4.65**
	随机搜索	44.16	62.50	6.42
SySeVR	强化学习	**12.25**	**26.22**	**15.62**
	随机搜索	8.88	19.02	24.59
VulDeepecker	强化学习	**20.42**	**32.93**	**11.23**
	随机搜索	18.64	29.77	15.02

综上, 采取强化学习模型的攻击方法在攻击有效性和生成对抗样本质量的各项指标上均好于采取随机选择算法的攻击方法. 这表明: 本文采取的基于强化学习的攻击方法只需要更少的扰动次数, 就能取得更好的攻击效果.

图 7 展示了以 Devign 为目标模型进行攻击时, 扰动次数随决策模型训练轮次变化关系. 随着强化学习模型训练轮次的增加, 对抗样本的扰动序列长度逐渐降低. 决策模型能够逐步学习到如何正确地选取扰动操作, 以更高效地生成对抗样本. 这表明采取强化学习模型进行扰动决策, 能够取得更高的攻击成功率、更少的扰动次数和查询次数. 因此, 通过强化学习模型所指导生成对抗样本具备更好的攻击效果、生成效率和隐蔽性. 本文提出的强化学习式对抗攻击方法优于基线方法.

(4) RQ4

为了研究不同抽样方法和奖励函数对实验结果的影响, 本文采取不同的抽样方法和奖励函数进行了对照实验. 抽样方法决定了如何根据模型预测的分布选择合理的扰动操作, 并影响最终的攻击效果. 因此, 本文通过对比实验收集了采取不同的抽样方法进行对抗攻击的实验数据. 图 8 展示了当采取 R_β作为奖励函数时, 分别利用 3 种抽样方法对模型 Devign 在数据集 D_{SARD} 上进行对抗攻击的实验结果.

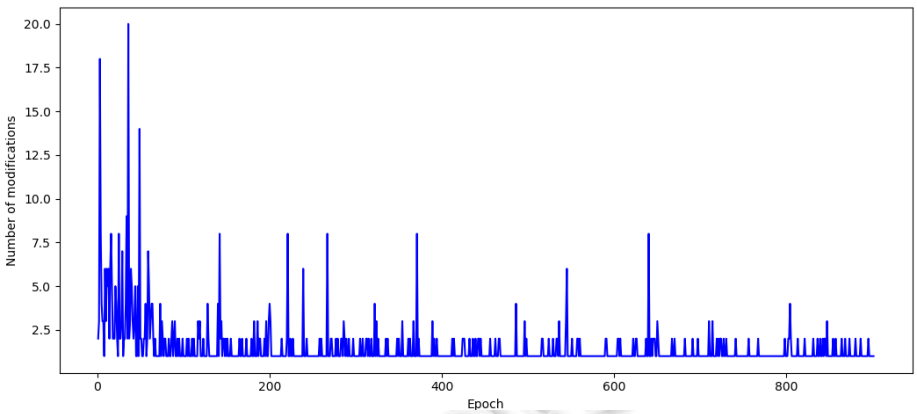


图 7　扰动次数随决策模型训练轮次变化关系

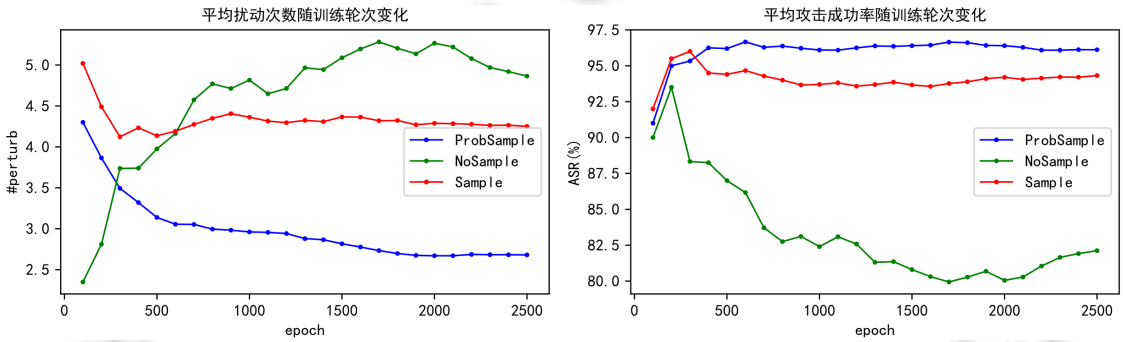


图 8　不同抽样方法的攻击结果

如图 8 所示: ProbSample 抽样方法在*#perturb* 和 *ASR* 两项指标上都优于其他两种抽样方法, 能达到 96.71% 的攻击成功率和 2.31 的平均扰动次数. NoSample 抽样方法在*#perturb* 指标上出现了波动上升, 而随着训练轮次的上升, 其 *ASR* 指标反而出现了波动下降的情况, 这证实了该抽样方法陷入了重复选取无效扰动操作的陷阱. 而 Sample 抽样方法虽然在 *ASR* 指标上同样稳定于较高值, 但*#perturb* 下降幅度小, 逊于 ProbSample 方法, 代表其平均扰动次数多, 扰动操作的选取效率并不高. 因此, 本实验决定选取 ProbSample 作为抽样方法.

本文引入了不同的两种奖励函数. 为了分析不同奖励函数下攻击成功率和平均扰动次数的变化情况, 本文通过对比实验收集了采取两种不同奖励函数进行对抗攻击的实验结果. 表 7 展示了针对 Devign 模型在数据集 D_{SARD} 上的实验结果, 分别展示了攻击成功率和攻击所采取扰动序列的平均长度. 在以 ProbSample 作为抽样方法的基础上, 采取 R_1 作为奖励函数的 *ASR* 和*#perturb* 分别为 93.74%, 3.1, 采取 R_β作为奖励函数的 *ASR* 和 *#perturb* 分别为 96.71%, 2.31. 通过比较基于两种不同奖励函数所得出的指标, 本文得出结论: 在 ProbSample 下的效果更好. 因此, 本实验决定选取 R_β作为奖励函数.

表 7　不同抽样方法和奖励函数实验的结果

抽样方法+奖励函数	*ASR* (%)	*#perturb*
Sample+R_β	94.10	4.28
NoSample+R_β	81.99	4.87
ProbSample+R_1	93.74	3.14
ProbSample+R_β	**96.71**	**2.31**

5　结论与未来工作

本文提出了一种面向漏洞检测模型的强化学习式对抗攻击方法, 该方法针对漏洞检测模型提出了 9 种轻

量的、语义约束和漏洞保留的原子扰动操作, 利用强化学习模型指导代码扰动操作的决策, 并基于扰动操作进行代码转换以生成有效的对抗样本, 以此来攻击基于深度学习的漏洞检测模型. 为验证攻击方法的效果, 本文基于SARD和NVD构建了两个数据集对目标模型进行了攻击评估. 实验结果表明: 本文所提出的对抗攻击方法面对4个不同类型、训练良好的漏洞检测模型在攻击成功率指标上分别达到92.8%, 91.33%, 26.2%和32.93%, 在扰动序列平均长度上分别取得2.31, 4.65, 15.62和11.23. 考虑到目标模型在实验数据集上取得的良好检测效果, 本攻击方法能够对4个目标模型进行有效、高效且隐蔽的攻击. 并且通过对比实验, 证实了本文所提出的具备补丁特征的扰动操作, 相较于常规的代码转换操作, 能够针对漏洞检测模型生成更佳的对抗样本. 同时, 通过与随机算法的对比, 证实了本文所提出的基于强化学习的策略模型的有效性. 本工作证实了当前的漏洞检测模型存在被攻击的风险, 需要进一步研究以提升模型的鲁棒性.

在下一步的研究工作中, 本文计划从以下3个方面进行探索.

- 第一, 本方法基于代码的抽象语法树进行代码转换, 攻击针对关注代码结构的漏洞检测模型的效果较好, 但针对关注代码词句和标识符的漏洞检测模型效果较差, 需进一步优化已有的扰动操作和代码转换方法;
- 第二, 对抗攻击能够暴露被评估模型的鲁棒性和脆弱性指标, 有助于进一步研究模型如何进行防御. 需进一步利用本文的攻击方法和生成的对抗样本训练漏洞检测模型以增强其鲁棒性;
- 第三, 本文第4.2节的实验结果表明: 相较于真实漏洞数据集, 本文的攻击方法在实验性数据集上能够取得更好的攻击效果, 需进一步优化基于真实漏洞样本的攻击效果.

未来会针对以上3个问题进行研究.

References:

[1] Li Z, Zou D, Xu S, *et al*. VulDeePecker: A deep learning-based system for vulnerability detection. In: Proc. of the Annual Network and Distributed System Security Symp. (NDSS). 2018. 1−15.

[2] Li Z, Zou D, Xu S, *et al*. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. on Dependable and Secure Computing, 2021, 19(4): 2244−2258.

[3] Zhou Y, Liu S, Siow J, *et al*. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Advances in Neural Information Processing Systems (NeurIPS). 2019. 32.

[4] Duan X, Wu J, Ji S, *et al*. VulSniper: Focus your attention to shoot fine-grained vulnerabilities. In: Proc. of the Int'l Joint Conf. on Artificial Intelligence (IJCAI). 2019. 4665−4671.

[5] Szegedy C, Zaremba W, Sutskever I, *et al*. Intriguing properties of neural networks. arXiv:1312.6199, 2013.

[6] Goodfellow I, Shlens J, Szegedy C. Explaining and harnessing adversarial examples. In: Proc. of the 2015 Int'l Conf. on Learning Representations (ICLR Poster). 2015.

[7] Rozsa A, Rudd EM, Boult TE, *et al*. Adversarial diversity and hard positive generation. In: Proc. of the 2016 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR). IEEE, 2016. 410−417.

[8] Kreuk F, Adi Y, Cisse M, *et al*. Fooling end-to-end speaker verification with adversarial examples. In: Proc. of the 2018 IEEE Int'l Conf. on Acoustics, Speech and Signal Processing. 2018. 1962−1966.

[9] Taori R, Kamsetty A, Chu B, *et al*. Targeted adversarial examples for black box audio systems. In: Proc. of the 2019 IEEE Security and Privacy Workshops. 2019. 15−20.

[10] Zhang, WW, Guo SJ, Zhang HY, *et al*. Challenging machine learning-based clone detectors via semantic-preserving code transformations. IEEE Trans. on Software Engineering, 2023, 49(5): 3052−3070.

[11] Yefet N, Alon U, Yahav E. Adversarial examples for models of code. ACM on Programming Languages, 2020, 4(OOPSLA): 162:1−162:30.

[12] Akshita J, Reddy CK. CodeAttack: Code-based adversarial attacks for pre-trained programming language models. arXiv:2206.00052v3, 2023.

[13] Li Z, Tang J, Zou D. Towards making deep learning-based vulnerability detectors robust. arXiv:2108.00669v2, 2021.

[14] Schulman J, Wolski F, Dhariwal P, *et al*. Proximal policy optimization algorithms. arXiv:1707.06347, 2017.

[15] Liu X, Li X, Prajapati R, *et al*. Deepfuzz: Automatic generation of syntax valid C programs for fuzz testing. In: Proc. of the AAAI Conf. on Artificial Intelligence, 2019, 33(1): 1044−1051.

[16] Lee S, Han HS, Cha SK, *et al*. Montage: A neural network language model-guided javascript engine fuzzer. In: Proc. of the 29th USENIX Conf. on Security Symp. 2020. 2613−2630.

[17] Lu K, Pakki A, Wu Q. Automatically identifying security checks for detecting kernel semantic bugs. In: Proc. of the 24th European Symp. on Research in Computer Security. Part II 24. Luxembourg: Springer Int'l Publishing, 2019. 3−25.

[18] Lu K, Pakki A, Wu Q. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In: Proc. of the 28th USENIX Security Symp. 2019. 1769−1786.

[19] Duan X, Wu JZ, Luo TY, *et al*. Vulnerability mining method based on code property graph and attention BiLSTM. Ruan Jian Xue Bao/Journal of Software, 2020, 31(11): 3404−3420 (in Chinese with English abstract). http://www.jos.org.cn/ 1000-9825/ 6061.htm [doi: 10.13328/j.cnki.jos.006061]

[20] Cao S, Sun X, Bo L, *et al*. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In: Proc. of the 44th Int'l Conf. on Software Engineering. 2022. 1456−1468.

[21] Ji SL, Du TY, Deng SG, *et al*. Robustness certification researchon deep learning models: A survey. Chinese Journal of Computers, 2022, 45(1): 190−206 (in Chinese with English abstract).

[22] Liu H, Zhao B, Guo JB, *et al*. Survey on adversarial attacks towards deep learning. Journal of Cryptologic Research, 2021, 8(2): 202−214 (in Chinese with English abstract).

[23] Wang Z, Yang H, Feng Y, *et al*. Towards transferable targeted adversarial examples. In: Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition. 2023. 20534−20543.

[24] Wei Z, Chen J, Wu Z, *et al*. Enhancing the self-universality for transferable targeted attacks. In: Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition. 2023. 12281−12290.

[25] Zügner D, Günnemann S. Adversarial attacks on graph neural networks via meta learning. In: Proc. of the Int'l Conf. on Learning Representations. 2019.

[26] Dai H, Li H, Tian T, *et al*. Adversarial attack on graph structured data. In: Proc. of the Int'l Conf. on Machine Learning. PMLR, 2018. 1115−1124.

[27] Zhang ZW, Zhang HY, Shen BJ, *et al*. Diet code is healthy: Simplifying programs for pre-trained models of code. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. 2022. 1073−1084.

[28] Yang Z, Shi J, He J, *et al*. Natural attack for pre-trained models of code. In: Proc. of the 44th Int'l Conf. on Software Engineering (ICSE 2022). 2022. 1482−1493.

[29] Zhou Y, Zhang X, Shen J, *et al*. Adversarial robustness of deep code comment generation. ACM Trans. on Software Engineering and Methodology (TOSEM), 2022, 31(4): 1−30.

[30] Church KW. Word2Vec. Natural Language Engineering, 2017, 23(1): 155−162.

[31] Roy CK, Cordy JR. A mutation/injection-based automatic framework for evaluating code clone detection tools. In: Proc. of the Int'l Conf. on Software Testing, Verification, and Validation Workshops. 2009. 157−166.

[32] Schulman J, Levine S, Abbeel P, *et al*. Trust region policy optimization. In: Proc. of the Int'l Conf. on Machine Learning. PMLR, 2015. 1889−1897.

[33] Xu H, Zhou Y, Kang Y, *et al*. On secure and usable program obfuscation: A survey. arXiv:1710.01139, 2017.

附中文参考文献:

[19] 段旭, 吴敬征, 罗天悦, 等. 基于代码属性图及注意力双向 LSTM 的漏洞挖掘方法. 软件学报, 2020, 31(11): 3404−3420. http://www.jos.org.cn/1000-9825/6061.htm [doi: 10.13328/j.cnki.jos.006061]

[21] 纪守领, 杜天宇, 邓水光, 等. 深度学习模型鲁棒性研究综述. 计算机学报, 2022, 45(1): 190−206.

[22] 刘会, 赵波, 郭嘉宝, 等. 针对深度学习的对抗攻击综述. 密码学报, 2021, 8(2): 202−214.

陈思然(1997－), 男, 硕士生, CCF 学生会员, 主要研究领域为智能系统安全, 漏洞挖掘.

吴敬征(1982－), 男, 博士, 研究员, 博士导师, CCF 高级会员, 主要研究领域为系统安全, 漏洞挖掘, 操作系统安全.

凌祥(1992－), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为智能软件安全.

罗天悦(1990－), 男, 高级工程师, CCF 专业会员, 主要研究领域为操作系统安全分析, 代码漏洞挖掘, 人工智能安全.

刘镓煜(1998－), 男, 博士生, 主要研究领域为计算机系统安全, 人工智能.

武延军(1979－), 男, 博士, 研究员, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 系统安全.