

基于程序过程间语义优化的深度学习漏洞检测方法

李妍^{1,2,3}, 羌卫中^{1,2,3}, 李珍^{1,2,3}, 邹德清^{1,2,3}, 金海^{1,4}

(1. 大数据技术与系统国家地方联合工程研究中心服务计算技术与系统教育部重点实验室, 湖北 武汉 430074;
2. 分布式系统安全湖北省重点实验室, 湖北 武汉 430074; 3. 华中科技大学网络空间安全学院, 湖北 武汉 430074;
4. 华中科技大学计算机科学与技术学院, 湖北 武汉 430074)

摘要: 近年来, 软件漏洞引发的安全事件层出不穷, 及早发现并修补漏洞能够有效降低损失。传统的基于规则的漏洞检测方法依赖于专家定义规则, 存在较高的漏报率, 基于深度学习的方法能够自动学习漏洞程序的潜在特征, 然而随着软件复杂程度的提升, 该方法在面对真实软件时存在精度下降的问题。一方面, 现有方法执行漏洞检测时大多在函数级工作, 无法处理跨函数的漏洞样例; 另一方面, BGRU 和 BLSTM 等模型在输入序列过长时性能下降, 不善于捕捉程序语句间的长期依赖关系。针对上述问题, 优化了现有的程序切片方法, 结合过程内和过程间切片对跨函数的漏洞进行全面的上下文分析以捕获漏洞触发的完整因果关系; 应用了包含多头注意力机制的 Transformer 神经网络模型执行漏洞检测任务, 共同关注来自不同表示子空间的信息来提取节点的深层特征, 相较于循环神经网络解决了信息衰减的问题, 能够更有效地学习源程序的语法和语义信息。实验结果表明, 该方法在真实软件数据集上的 F1 分数达到了 73.4%, 相较于对比方法提升了 13.6%~40.8%, 并成功检测出多个开源软件漏洞, 证明了其有效性与实用性。

关键词: 漏洞检测; 程序切片; 深度学习; 注意力机制

中图分类号: TP311

文献标志码: A

DOI: 10.11959/j.issn.2096-109x.2023085

Deep learning vulnerability detection method based on optimized inter-procedural semantics of programs

LI Yan^{1,2,3}, QIANG Weizhong^{1,2,3}, LI Zhen^{1,2,3}, ZOU Deqing^{1,2,3}, JIN Hai^{1,4}

1. Services Computing Technology and System Lab, National Engineering Research Center for Big Data Technology, Wuhan 430074, China
2. Hubei Key Laboratory of Distributed System Security, Wuhan 430074, China
3. School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China
4. School of Computer Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China

Abstract: In recent years, software vulnerabilities have been causing a multitude of security incidents, and the early

收稿日期: 2023-02-17; 修回日期: 2023-07-28

通信作者: 羌卫中, wzqiang@hust.edu.cn

基金项目: 国家自然科学基金(62272187); 国家通用技术基础研究联合基金(U1936211)

Foundation Items: The National Natural Science Foundation of China (62272187), The Joint Funds of the National Natural Science Foundation of China (U1936211)

引用格式: 李妍, 羌卫中, 李珍, 等. 基于程序过程间语义优化的深度学习漏洞检测方法[J]. 网络与信息安全学报, 2023, 9(6): 86-101.

Citation Format: LI Y, QIANG W Z, LI Z, et al. Deep learning vulnerability detection method based on optimized inter-procedural semantics of programs[J]. Chinese Journal of Network and Information Security, 2023, 9(6): 86-101.

discovery and patching of vulnerabilities can effectively reduce losses. Traditional rule-based vulnerability detection methods, relying upon rules defined by experts, suffer from a high false negative rate. Deep learning-based methods have the capability to automatically learn potential features of vulnerable programs. However, as software complexity increases, the precision of these methods decreases. On one hand, current methods mostly operate at the function level, thus unable to handle inter-procedural vulnerability samples. On the other hand, models such as BGRU and BLSTM exhibit performance degradation when confronted with long input sequences, and are not adept at capturing long-term dependencies in program statements. To address the aforementioned issues, the existing program slicing method has been optimized, enabling a comprehensive contextual analysis of vulnerabilities triggered across functions through the combination of intra-procedural and inter-procedural slicing. This facilitated the capture of the complete causal relationship of vulnerability triggers. Furthermore, a vulnerability detection task was conducted using a Transformer neural network architecture equipped with a multi-head attention mechanism. This architecture collectively focused on information from different representation subspaces, allowing for the extraction of deep features from nodes. Unlike recurrent neural networks, this approach resolved the issue of information decay and effectively learned the syntax and semantic information of the source program. Experimental results demonstrate that this method achieves an *F1* score of 73.4% on a real software dataset. Compared to the comparative methods, it shows an improvement of 13.6% to 40.8%. Furthermore, it successfully detects several vulnerabilities in open-source software, confirming its effectiveness and applicability.

Keywords: vulnerability detection, program slice, deep learning, attention mechanism

0 引言

近年来,随着软件行业的蓬勃发展,各式各样的软件给人们生产生活带来便利的同时引入了信息泄露的风险。黑客利用软件漏洞植入病毒、木马等,甚至绕过访问控制窃取高级权限,对数据和系统安全造成了巨大的威胁。通用漏洞披露(CVE, common vulnerabilities and exposures)^[1]所公布的漏洞数目逐年上升,2021年发布了20 175个CVE,比2020年超出10%,创下2018年以来的最大涨幅^[2]。截至2022年11月,CVE发布的漏洞总数已经超过18万个,这些漏洞年复一年积累起来,对企业的软件维护工作造成了巨大的负担。依赖于劳动密集型、被动的漏洞管理方式已然不足,必须提高其自动化效率,使用深度学习等方法提高软件漏洞的检测效率和效果,做到全面、系统、可持续地降低风险。

传统的机器学习在应用于漏洞检测时依赖于专家知识定义漏洞特征,需要投入大量的人力和时间成本,检测效果与工作者的实践经验及其专业领域知识深度息息相关^[3],且提取的特征可能只适用于特定的任务。新兴的深度学习在图像识别、自然语言处理等领域取得了有效的成果,在

面向源代码的软件漏洞检测方面显示出了不俗的潜力。一方面,深度神经网络的层状结构有利于学习抽象和高度非线性的特征,能够捕捉源代码中复杂的漏洞模式;另一方面,它能够自动提取样本的深层特征^[4],相较于人工提取特征,其更加全面,泛化性更强。

然而,多项研究^[5-6]表明,真实软件的数据远比人工合成的数据复杂,现有基于深度学习的漏洞检测工作在面对真实软件时存在精度下降的问题。Dam等^[7]强调,简单的代码级特征不足以捕获源代码实际的语义和语法表示,表示源代码的真实语义是非常重要的,它能够区分代码中有漏洞和无漏洞的部分。同时,目前应用深度神经网络进行漏洞检测的相关工作主要集中于函数级的漏洞识别,对于涉及多个函数的漏洞样本,这些方法无法表示过程间调用关系,存在漏报率和误报率较高的问题。为减少漏洞样本中包含的冗余信息,SySeVR^[8]从源代码中识别敏感的应用程序接口(API, application program interface)调用、数组使用、指针使用及数值运算这4种兴趣点,基于数据依赖和控制依赖关系执行双向切片提取与兴趣点相关的程序语句集合;MVP^[9]应用程序切片技术从漏洞函数及其补丁函数中提取漏洞和

补丁的语法和语义特征,但对函数调用信息的处理不够恰当,漏洞上下文信息存在缺失。本文对现有的程序切片方法进行优化,调整过程内切片规则,以平衡切片序列的语义完整性和包含的冗余信息量之间的关系,完善了缺失的函数调用关系,执行过程间分析以提取漏洞相关的全面上下文信息,揭示漏洞触发的根本原因。此外,本文研究了多种漏洞类型的 `source` 和 `sink` 语句特征,获取原始切片后根据自定义规则匹配对应语句再进行截断,能够进一步减少切片中的冗余信息,有利于模型学习潜在的漏洞模式。

深度神经网络能够有效地学习漏洞代码的复杂特征^[10],近期的相关工作^[8,11-12]将卷积神经网络(CNN, convolutional neural network)和循环神经网络(RNN, recurrent neural network)等神经网络模型应用于漏洞检测任务并取得了显著成果。2017年,谷歌基于多头注意力机制提出了Transformer神经网络模型,该模型在机器翻译、文本摘要等自然语言处理任务中展现出了卓越性能^[13]。考虑到编程语言与自然语言的相似性,本文应用基于Transformer神经网络模型学习程序切片中的长期依赖关系,相较于现有的静态漏洞检测器取得了更优的检测效果。

综上所述,本文的主要贡献如下。

1) 结合过程间分析对现有程序切片算法进行改进,提取与漏洞相关的全面上下文语句,捕获漏洞触发的完整因果关系,更加准确地表示漏洞程序的语法和语义信息。

2) 应用基于Transformer神经网络模型执行漏洞检测任务,相较于经典RNN模型能够更有效地学习程序语句间的长期依赖关系。

3) 在17个真实软件的NVD漏洞数据集上进行实验,漏洞检测的精确率相较于现有系统提升了13.6%~40.8%,并成功检测出多个潜在的开源软件漏洞,证明了本文方法的有效性。

1 相关工作

传统的基于机器学习的方案通常依赖于静态代码分析手动提取特征,引导模型学习漏洞代码模式。Ghaffarian等^[14]将其分为软件度量、异常

检测及漏洞代码模式识别三大类。基于软件度量的方法借用软件工程领域的软件质量指标预测漏洞,这种方法建立在频繁修改的往往是错误的代码(它们更有可能存在漏洞)的假设之上。基于异常检测的方法并不直接寻找漏洞,而是考虑漏洞产生原因。相关研究^[15-17]从API的检查异常、函数调用、库引用中提取特征来进行漏洞检测,然而这些特征集通常只与特定的漏洞类型相关或者只适用于特定的项目,误报率较高。此外,部分研究应用文本挖掘技术从源码上下文中提取漏洞代码模式。相关研究^[18-19]指出,基于频率的文本挖掘技术无法捕捉到程序中丰富的语义信息,并且无法保留长期上下文依赖关系。Yamaguchi等^[20]提出了代码属性图的概念,结合抽象语法树(AST, abstract syntax tree)、控制流图(CFG, control flow graph)、程序依赖图(PDG, program dependency graph)生成更加全面的图结构来描述代码语义信息。除了结构化的程序特征集,程序动态分析得到的执行轨迹也可以作为特征^[21]。

与传统的机器学习方法相比,深度学习方法能够自动学习更加复杂和抽象的潜在特征,泛化性更强,并且能够针对不同的使用场景定制网络结构。Lin等^[11,22-23]通过深度优先遍历算法将函数的AST转化为序列,应用连续词袋模型将序列中的元素表示为嵌入层以恢复代码语义信息,使用双向长短时记忆(BLSTM, bidirectional long short-term memory)网络学习代码深层特征用于训练随机森林分类器。Li等^[8]提出的SySeVR系统先从程序的AST中识别基于语法的漏洞候选程序,再基于程序的PDG获取基于语义的漏洞候选程序,旨在更加准确地保留反映漏洞特征的代码语法和语义信息。然而,扁平的序列化特征不可避免地会丢失代码语义中的层次和结构信息,因此基于图或树的神经网络是一种更加理想的选择^[24]。Devign^[12]应用图神经网络(GNN, graph neural network)的Conv模块在代码属性图^[20]上,学习更加广泛的漏洞类型和模式。FUNDED^[25]将门控图神经网络(GGNN, gated graph sequence neural network)应用于漏洞检测,并利用迁移学习将漏洞检测模型移植到不同的编程语言上。IVDetect^[26]从PDG中提取信息作为代码的向量表

征,使用特征注意力图卷积网络(FA-GCN, graph convolution network with feature-attention)对向量表征进行分类,并使用解释模型 GNNExplainer^[27]进行漏洞语句定位。相关研究^[28-33]在 CNN、RNN、GNN 等深度学习模型上进行了广泛的探索,但现有大多数智能漏洞检测系统在函数级工作,缺乏自动化的过程间分析。

本文将程序的 PDG 和过程间调用信息结合起来获取程序切片,基于自定义的 source 和 sink

规则分离被漏洞影响的代码块,并将基于注意力机制的 Transformer^[13]神经网络模型应用到漏洞检测任务,取得更加有效的漏洞检测效果。

2 系统设计

本文的目标是设计一个能够有效检测真实软件漏洞的检测系统,系统分为源代码预处理和基于深度学习的漏洞检测两个阶段,系统架构如图1所示。

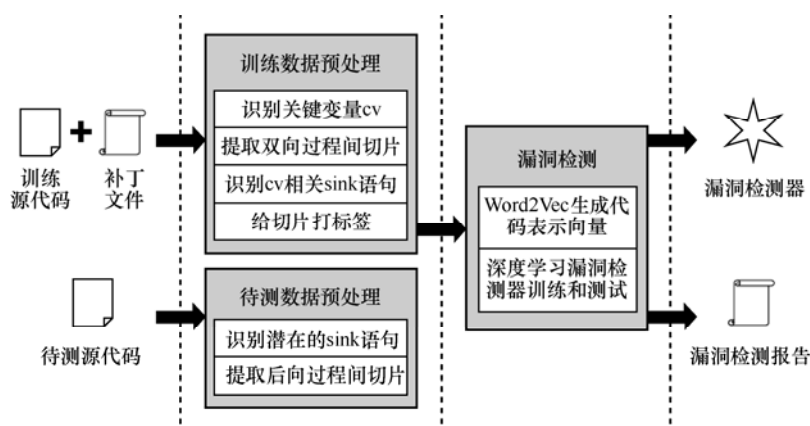


图1 系统架构

Figure 1 The architecture of system

在训练数据预处理阶段,系统输入为已知漏洞类型的修复前和修复后源代码及其补丁文件,首先解析补丁文件提取关键变量,提取与补丁语句具有数据依赖或控制依赖关系的语句集合作为程序切片,之后通过数据流分析识别与关键变量相关的 source 和 sink 语句,以此为依据截断切片中的冗余语句,为其设置标签后作为训练样本。在待测数据预处理阶段,系统输入为待测的软件源代码,本阶段不依赖于补丁文件识别与关键变量和提取切片,而是通过启发式的方法在源代码中识别潜在的 sink 语句,即漏洞触发语句,再执行后向切片提取与该语句相关的语句集合作为测试样本。在基于深度学习的漏洞检测阶段,先使用词向量模型 Word2Vec 将程序切片表示为向量,再将训练样本的表示向量输入 Transformer 深度神经网络学习程序语句间的长期依赖关系,捕获潜在的漏洞模式,模型训练完成后输入测试样本的表示向量执行检测任务并输出检测报告。

2.1 源代码处理

训练数据的预处理致力于对与漏洞相关的语

句执行全面的上下文分析,并减少切片中包含的噪声,平衡程序切片的语义完整性和包含的冗余信息之间的关系,提取高质量的代码特征,捕获漏洞代码及其补丁代码的实际语义差异。对于待测数据来说,直接识别潜在的 sink 语句再提取其后向过程间切片,切片终点为 sink 语句中所包含变量的 source 语句,该方法能够保证当存在漏洞时,待测样本与训练样本对漏洞的特征描述相似。

2.1.1 漏洞类型说明

本文重点分析了 10 种常见的漏洞类型:缓冲区溢出(CWE-119)、越界读(CWE-125)、越界写(CWE-787)、数值错误(CWE-189)、整数溢出或环绕(CWE-190)、输入校验不当(CWE-20)、除零错误(CWE-369)、Double Free(CWE-415)、Use After Free(CWE-416)和空指针解引用(CWE-476)。漏洞类型描述如表1所示。

2.1.2 识别并提取关键变量

漏洞样本的补丁语句中涉及的变量通常与漏洞触发语句存在数据依赖关系,本小节根据漏洞补丁识别与漏洞相关的关键变量。

表1 漏洞类型描述
Table 1 Description of vulnerability type

CWE-ID	漏洞类型描述
CWE-119	缓冲区溢出，内存缓冲区操作不当，读取或写入缓冲区预期边界之外的内存位置
CWE-125	越界读，软件读取的数据超过预期缓冲区的末尾或开头
CWE-787	越界写，软件将数据写入预期缓冲区的末尾或开头之前
CWE-189	数值错误，与不正确的数值计算或转换相关
CWE-190	整数溢出或环绕，当逻辑假定结果值总是大于原始值时，软件执行的计算可能会产生整数溢出或环绕。当计算结果用于资源管理或执行控制时，可能会引入其他漏洞
CWE-20	输入校验不当，软件接收输入或数据，但未验证或错误地验证输入是否安全或是否被正确处理，攻击者可能利用该漏洞修改控制流、控制任意资源和执行任意代码
CWE-369	除零错误，经常出现在涉及长度、宽度和高度等物理尺寸的计算中，可能导致系统崩溃
CWE-415	Double Free，在同一内存地址上两次调用 free()，可能导致修改意外的内存位置
CWE-416	Use After Free，释放内存后引用内存，可能导致程序崩溃、使用意外值或代码执行
CWE-476	空指针解引用，程序解引用预期有效但实际为 NULL 的指针，通常会导致崩溃或退出

本文主要关注以下5种补丁语句。

1) 变量定义：关键变量是添加、删除或者修改的变量。

2) 赋值语句：关键变量是被赋值的变量。对于与数值计算有关的漏洞类型（如 CWE-189、CWE-190 和 CWE-369 等），由于漏洞可能在等号左边的赋值或者右边的数值运算处产生，整个表达式涉及的变量都被定义为关键变量。

3) 条件语句：关键变量是条件判断语句中增加、删除或者修改所涉及的变量，包含 if 语句和 while、for 等控制语句中的条件判断部分。

4) 函数调用语句：关键变量是添加、删除或者修改的函数实参。

5) 函数定义语句：关键变量是添加、删除或者修改的函数形参。

2.1.3 获取过程间程序切片

本小节执行过程间分析，获取与关键变量相关的全面上下文程序切片。首先使用 Joern^[34]工具解析源代码，对每个函数生成 PDG，然后应用 Neo4j^[35]图数据库查询函数节点，通过函数名匹配的方式获取源代码中的函数调用关系，之后分别执行过程内切片和过程间切片获取与关键变量相关的上下文信息。

(1) 过程内切片

首先通过解析补丁文件获取漏洞文件中被删

除的语句和修复后文件中增加的语句所在的函数名和行数，然后在 Neo4j 图数据库中查找这些语句对应的 PDG 节点作为切片的起点。当补丁语句是条件控制语句时，依据控制依赖，执行前向切片获取全部相关语句可能引入过多噪声，而仅仅依据数据依赖进行切片则可能不包含漏洞触发语句^[8]。对于 if、while、for 等条件语句，本文首先执行后向切片获取关键变量的定义语句，然后由定义语句出发依据函数内的数据依赖关系执行前向切片，最后从程序切片中删除关键变量定义语句和控制语句之间的冗余语句。其他语句则直接执行双向切片，后向切片保留全部的数据依赖和控制依赖语句，前向切片仅保留数据依赖语句。

(2) 过程间切片

获取函数调用图后，本文从漏洞函数出发执行过程间切片。漏洞函数可能调用其他函数，记为“调用类型”；也可能被其他函数调用，记为“被调用类型”。对于前者，漏洞函数将实参传递给被调用的函数，因此基于数据依赖关系获取参数相关的语句加入切片。“调用类型”的漏洞样例 CVE-2015-8662 如图 2 所示，漏洞函数 ff_dwt_decode 调用函数 dwt_decode53，将实参 s 传递给该函数，该参数未经检查就作为数组下标使用，可能导致越界读，该样例的 sink 语句为“int w = s->linelen[s->ndeclevels-1][0];”，漏洞在函数 dwt_de-

code53 内被触发。对于后者,其他函数调用漏洞函数并传入实参,漏洞函数对该参数进行处理,执行完毕后返回调用函数继续执行,漏洞触发语句可能位于调用函数内,因此需要获取漏洞函数返回之后的语句加入切片。“被调用类型”的漏洞样例 CVE-2018-10878 如图 3 所示,漏洞函数 ext4_check_descriptors 被函数 ext4_fill_super 调用,漏洞函数对关键变量执行检查逻辑后返回 sink 语句所在的函数 ext4_fill_super,该样例的 sink 语句为“memcpy(&sb->s_uuid, es->s_uuid, sizeof(es->s_uuid));”。

```
// source语句, 输入参数s
int ff_dwt_decode(DWTContext *s, void *t)
{
    /* diff语句, 关键变量为s->ndeclevels
    if (s->ndeclevels == 0)
        return 0;
    */
    switch (s->type) {
        ....
        case FF_DWT53:
            dwt_decode53(s, t);
            break;
        default:
            return -1;
    }
    return 0;
}
static void dwt_decode53(DWTContext *s, int *t)
{
    ....
    // sink语句, 数组越界访问
    int w = s->linelen[s->ndeclevels - 1][0];
    ....
}
```

图2 “调用类型”的漏洞样例 CVE-2015-8662

Figure 2 A vulnerability sample CVE-2015-8662 of “caller type”

```
// source语句, 输入参数sb
int ext4_fill_super(struct s_block *sb, void *data, int silent)
{
    ....
    if (!ext4_check_descriptors(sb,
        logical_sb_block, &first_not_zeroed)) {
        ext4_msg(sb, KERN_ERR, "...");
        ret = -EFSCORRUPTED;
        goto failed_mount;
    }
    // sink语句, 缓冲区溢出
    memcpy(&sb->s_uuid, es->s_uuid, sizeof(es->s_uuid));
    ....
}
static int ext4_check_descriptors(struct s_block *sb, ...)
{
    ....
    block_bitmap = ext4_block_bitmap(sb, gdp);
    /* diff语句, 关键变量为block_bitmap, 追溯到参数sb
    if (block_bitmap >= sb_block + 1 &&
        block_bitmap <= last_bg_block) {
        ext4_msg(sb, KERN_ERR, "...", i);
        if (!sb_rdonly(sb))
            return 0;
    }
    */
    return 1;
}
```

图3 “被调用类型”的漏洞样例 CVE-2018-10878

Figure 3 A vulnerability sample CVE-2018-10878 of “called type”

为提升系统识别过程间漏洞的能力,本文对 SySeVR 的过程间切片处理方法做出以下优化。

(1) 函数调用的隐式返回值处理

SySeVR 在处理函数调用语句时,如果函数调用没有显式返回有效数据或返回错误,则生成的程序切片将以函数调用语句结束。这是不合理的,因为函数调用中的指针类型参数实际上包含隐式的返回值,该函数调用语句实际上与函数中后续语句对指针参数的使用具有数据依赖关系。为了解决该问题,本文首先提取函数调用语句中的指针类型变量参数,通过后向切片到达该指针变量的定义语句,然后执行前向切片获取与该指针变量具有数据依赖关系的语句集合,最后从程序切片中删除指针变量定义语句和函数调用语句之间的语句,以排除不相关的冗余信息。

(2) 漏洞函数执行返回语句后的切片流程处理

SySeVR 在漏洞函数结束时将停止程序切片流程,然而在处理“被调用类型”的过程间漏洞时,需要在漏洞函数返回后继续执行程序切片以到达漏洞触发位置。为了实现这一点,本文基于源代码的函数调用图获得一组调用漏洞函数 f 的函数集合,表示为 H ,对于每个调用函数 $h \in H$,以其中调用漏洞函数的语句为起点提取漏洞函数返回之后与其具有数据依赖关系的程序语句。重复此过程,直到过程间切片层数达到给定阈值。

(3) 切片合并

执行过程内和过程间切片后,需要对同一个漏洞函数获取的切片内容进行合并。对于过程内切片,按照语句在源代码中所在的位置进行去重和排序;对于“调用类型”的过程间切片,从被漏洞函数 f 调用的函数 g 中获取的语句被填充在调用语句之后;对于“被调用类型”的过程间切片,漏洞函数 f 执行完毕后,返回调用它的函数 h 继续执行,该部分语句被填充在漏洞函数的返回语句之后。跨函数过程间的切片结构如图 4 所示。

(4) 特殊切片处理

当漏洞补丁仅包含添加语句而无删除语句时,首先根据修复后函数添加的语句获取切片,再删除切片中包含的添加语句作为漏洞函数对应的切片,因为这些语句在漏洞函数中并未出现;当漏洞补丁仅包含删除语句而无添加语句时,相应地,删除漏洞函数切片中包含的删除语句作为

修复后函数对应的切片；当补丁文件中既包含添加语句又包含删除语句时，直接由删除语句出发为漏洞函数生成切片，由添加语句出发为修复后函数生成切片即可。最后，将成对的漏洞切片和修复后切片输入神经网络进行训练和测试。

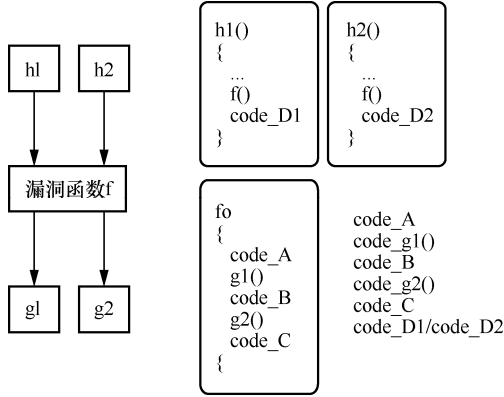


图4 跨函数过程间的切片结构
Figure 4 Cross function procedural slice structure

算法1描述了提取过程间切片的具体过程，首先基于PDG获取补丁语句相关的过程内切片，然后基于函数调用关系分别获取“调用类型”与“被调用类型”的过程间切片，最后按照程序语句的执行顺序对其进行合并。

算法1 提取过程间切片

输入 包含多个函数的源程序 $P = \{f_1, \dots, f_n\}$ ；补丁语句集合 $Y = \{e_1, \dots, e_r\}$ 及其所属函数 $Y_f = \{f_1, \dots, f_r\}$

输出 过程间切片集合 C

- 1) $C = \emptyset$;
- 2) for 每个函数 $f_i \in P$ do
- 3) 为 f_i 生成程序依赖图 $G_i = (V_i, E_i)$;
- 4) end for
- 5) for 每个补丁语句 $e_i \in Y$ & $f_i \in Y_f \rightarrow G_i$ do
- 6) if e_i 是控制类型的语句 then
- 7) 解析 e_i 中的变量并回溯至其定义语句 $e'_i \rightarrow e_i$ ，使用 e'_i 替换 e_i ;
- 8) end if
- 9) 为 e_i 生成函数 f_i 内前向切片 fs 及后向切片 bs ;
- 10) 过程内切片 $ps = fs \cup bs$ ， $\cup$ 表示有序集合，按照函数内语句顺序进行去重并排序；

- 11) if e_i 是控制类型的语句 then
- 12) 删除 e'_i 与 e_i 之间的冗余语句，结果作为 ps
- 13) end if
- 14) for 出现在 ps 中的每个函数调用语句 $s \in f_i$ do
- 15) 进入被调用的函数 $g \in P \rightarrow G_g$
- 基于 s 相关数据依赖关系生成前向切片 fs_g ;
- 16) “调用类型”的过程间切片 $ps' = ps_i$ before $s \cup fs_g \cup ps_2$ after s ;
- 17) end for
- 18) for 每个调用函数 f_i 的其他函数 $h \in P \rightarrow G_h$ do
- 19) 识别 h 中调用 f_i 的语句 $s \in h$;
- 20) if f_i 具有返回值 then
- 21) 进入函数 $h \in P \rightarrow G_h$ 基于 s 相关的数据依赖关系生成前向切片 fs_h ;
- 22) else
- 23) 解析 s 中的参数并回溯至其定义语句 $s' \rightarrow s$;
- 24) 基于 s' 相关的数据依赖关系生成前向切片;
- 25) 删除 s' 与 s 之间的冗余语句，结果作为 fs_h ;
- 26) end if
- 27) 合并“调用类型”与“被调用类型”的切片构成完整过程间切片， $ps' = ps' \cup fs_h$;
- 28) end for
- 29) $C = C \cup \{ps'\}$;
- 30) end for
- 31) return C

2.1.4 Source-sink 特征定义与匹配

本小节总结与漏洞类型相关的 source 和 sink 特征，在初步获得的过程间程序切片中基于数据依赖关系匹配关键变量对应的 source 和 sink 语句，对原始切片进行截断，最大程度地去除漏洞样本中的冗余信息，仅保留与漏洞最为相关的核心语句作为最终的切片样本输入深度学习模型进行训练。在污点分析中，通常使用三元组 (source, sink, sanitizer) 的形式表示漏洞：source 即污点

源，代表系统中不受信任的数据或者机密数据；sink 即污点汇聚点，代表安全敏感或者泄露隐私数据的不当操作；sanitizer 即无害处理，代表采用数据加密或者移除危害操作等手段使数据传播不再对软件系统造成危害。简单来说，source 为污点数据输入的位置，sink 为污点数据传播导致漏洞触发的位置，sanitizer 为过滤操作，通过定义以上内容可以获取一条污点数据传播链。本文研究的漏洞类型均与关键变量的不安全操作密切相关，本节提取关键变量的传播链作为重要特征以促进漏洞检测效果。

通过人工分析漏洞样例，本文总结了 3 种与关键变量相关的 source 语句特征，如表 2 所示。针对不同的漏洞类型，本文总结了与漏洞类型相关的关键变量和 sink 语句特征，如表 3 所示。

表 2 source 语句特征
Table 2 Feature of source statement

source 语句特征	样例
漏洞函数参数输入	vul_func(CV,...)
局部变量声明与定义	CV = func/fread/fopen/...
全局变量声明与定义	extern int CV = ...

sink 语句样例如图 5 所示。对于缓冲区溢出的漏洞类型，本节以漏洞 CVE-2014-8884 为例进行说明，如图 5(a)所示，在将 cmd->msg 复制给 b 时，未对缓冲区长度进行检查，会导致越界访问漏洞，该例的 sink 语句为“memcpy(&b[4],

cmd->msg, cmd->msg_len)”；出现漏洞的根本原因是敏感 API 使用不恰当。对于整数溢出的漏洞类型，以漏洞 CVE-2018-12896 为例，如图 5(b)所示，两个整数相加导致整数上溢，未进行越界检查，会进一步导致内核调度错误，该例的 sink 语句为“info->si_overrun += timr->it_overrun_last;”。对于空指针解引用的漏洞类型，以漏洞 CVE-2015-8956 为例，如图 5(c)所示，使用变量 addr 初始化结构体指针 sa，指针值可能为 NULL，在访问 sa 的成员变量时漏洞被触发，该例的 sink 语句为“int chan = sa->rc_channel;”。

在程序切片中识别 source 和 sink 语句时，首先对从补丁文件中直接提取的关键变量按照漏洞类型特征进行匹配，如果匹配失败，需要依据函数中的数据依赖关系和函数调用图对关键变量进行转化再重新匹配，考虑以下 3 种情况。

1) 结构体或类。当关键变量是结构体或类中的某个成员变量时，将结构体或类对象本身作为新的关键变量。

2) 赋值。关键变量通过赋值语句将污点数据传播给其他变量，此时将被赋值的变量定义为新的关键变量。

3) 函数调用。当漏洞函数调用其他函数时，关键变量被作为实参传递给其他函数，此时将关键变量对应的形参名作为新的关键变量；当漏洞函数被其他函数调用且关键变量为漏洞函数形参

表 3 与漏洞类型相关的关键变量和 sink 语句特征
Table 3 Feature of critical variables and sink statement related to vulnerability type

漏洞类型	CWE-ID	关键变量与 sink 语句特征
内存操作不当	CWE-119	
	CWE-125	内存敏感的 API，如 malloc、memcpy、memset 等，关键变量为内存大小相关的参数；数组使用，关键变量为数组下标；指针使用，关键变量为指针本身或与指针运算相关的参数
	CWE-787	
	CWE-20	
数值运算不当	CWE-189	整数运算，关键变量为参与整数运算的参数，可能进一步导致缓冲区溢出漏洞
	CWE-190	
	CWE-369	除法运算或者模运算，关键变量为除数
指针使用不当	CWE-415	double free，关键变量为指针，漏洞在第二次调用 free/delete 重复释放内存时被触发
	CWE-416	use after free，关键变量为释放内存后的空指针，漏洞在其调用 free/delete 释放内存之后再次使用时被触发
	CWE-476	空指针解引用，关键变量为未初始化或被赋值为 NULL 的指针，如未初始化的结构体、函数指针等，漏洞在其首次使用时被触发

时, 漏洞可能在当前函数返回原函数后被触发, 此时将原函数调用漏洞函数时的实参名作为新的关键变量。

```
@@ -156,6 +156,9 @@ static int ttusbdecfe_dvbs_diseqc_send_master_cmd
(struct dvb_frontend* fe, struct
0x00, 0x00, 0x00, 0x00,
0x00, 0x00 );

+ if (cmd->msg_len > sizeof(b) - 4)
+ return -EINVAL;
+
+ memcpy(&b[4], cmd->msg, cmd->msg_len);

state->config->send_command(fe, 0x72,
```

(a) CVE-2014-8884

```
@@ -283,6 +283,17 @@ static __init int init_posix_timers(void)
}
__initcall(init_posix_timers);

+static inline int timer_overrun_to_int(struct k_itimer *timr, int baseval)
+{
+ s64 sum = timr->it_overrun_last + (s64)baseval;
+ return sum > (s64)INT_MAX ? INT_MAX : (int)sum;
+}
+

@@ -321,10 +331,10 @@ void posixtimer_rearm(struct siginfo *info)

timr->it_active = 1;
timr->it_overrun_last = timr->it_overrun;
timr->it_overrun = -1;
timr->it_overrun = -1LL;
++timr->it_requeue_pending;

+ info->si_overrun += timr->it_overrun_last;
+ info->si_overrun = timer_overrun_to_int(timr, info->si_overrun);
}

unlock_timer(timr, flags);
```

(b) CVE-2018-12896

```
@@ -334,16 +334,19 @@ static int rfcomm_sock_create(struct net *net, struct socket *sock,
static int rfcomm_sock_bind(struct socket *sock, struct sockaddr *addr, int addr_len)
{
struct sockaddr_rc *sa = (struct sockaddr_rc *) addr;
+ struct sockaddr_rc sa;
struct sock *sk = sock->sk;
+ int chan = sa->rc_channel;
int err = 0;

BT_DBG("sk %p %pMR", sk, &sa->rc_bdaddr);
int len, err = 0;

if (!addr || addr->sa_family != AF_BLUETOOTH)
return -EINVAL;

+ memset(&sa, 0, sizeof(sa));
len = min_t(unsigned int, sizeof(sa), addr_len);
+ memcpy(&sa, addr, len);
BT_DBG("sk %p %pMR", sk, &sa.rc_bdaddr);

lock_sock(sk);
```

(c) CVE-2015-8956

图5 sink 语句样例
Figure 5 sink statement samples

当关键变量多次转化后仍未成功定位 source 或 sink 语句时, 本文辅以人工分析对语句特征进行完善以提升匹配效果。

2.1.5 切片标记

本节对截取 source-sink 后的切片样本进行标记。本文以漏洞函数为单位生成切片, 源文件分为“OLD”文件和“NEW”文件, 其中“OLD”表示修复前文件, “NEW”表示修复后文件。从“OLD”文件中提取的切片标记为“1”, 表示有漏洞的切片样本, 从“NEW”文件中提取的切片标

记为“0”, 表示无漏洞的切片样本, 两者共同输入神经网络进行训练。

2.2 基于深度学习的漏洞检测

本文基于 Transformer 神经网络模型自动学习程序切片中的语法和语义信息实现漏洞检测。首先将程序切片表示为特征向量, 然后将其输入神经网络模型进行训练, 最后使用训练完成的神经网络模型检测目标程序中是否存在漏洞。

2.2.1 切片表征

为了避免程序中用户自定义的变量名和函数名对神经网络的自动特征学习产生干扰, 需要对程序切片中包含的自定义符号进行映射, 变量名被映射为 VAR1, VAR2... 函数名被映射为 FUNC1, FUNC2... 在不同的样本中, 不同的变量名或函数名可能被映射为相同的名称。

程序切片需要表征为向量才能够输入神经网络进行训练和测试。本文应用 Word2Vec^[36]词向量模型进行文本嵌入, 该模型首先将每个词牌映射为指定维度的数字向量, 然后将各个词牌对应的向量连接起来作为程序切片样本的特征向量。为了使切片样本长度一致, 本文对切片样本进行截断和补全, 样本截断所选取长度的准则为覆盖 95% 的切片样本。具体来说, 首先使用全部的切片样本所包含的词牌预训练一个 Word2Vec 模型, 然后将程序切片输入该预训练的词向量模型进行文本嵌入, 输出大小为 $m \times n$ 的矩阵作为程序切片样本对应的特征向量, 其中 m 表示样本中包含的词牌数, n 表示特征向量的维度。

2.2.2 神经网络模型

本节的核心思想是利用 Transformer 模型的自注意力机制自动选择与漏洞特征相关的特征集合, 将 Transformer 的多层编码器与线性分类层结合起来实现切片级的漏洞样本分类任务。神经网络模型架构如图 6 所示。

模型的输入为切片样本中每一个单词的表示向量 X , X 由词向量模型直接提取的特征向量和单词位置的特征向量相加得到, 大小为 $n \times d$, n 是样本中的单词个数, d 是表示向量的维度; 然后应用多个编码器块进行自注意的特征学习, 第一个编码器块的输入为样本的表示向量矩阵, 后续编码器块的输入是前一个编码器块的输出; 最后添加全连接层和 softmax 层进行分类。

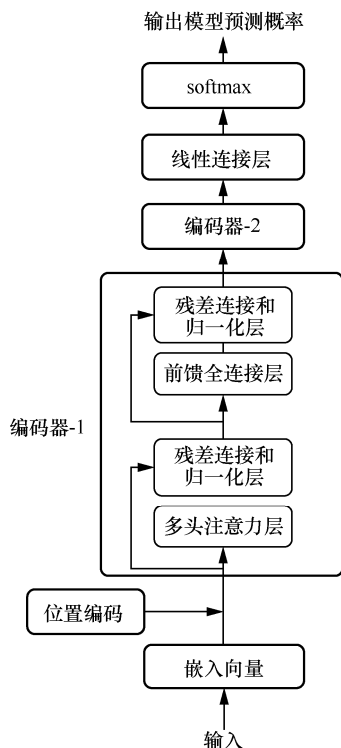


图6 神经网络模型架构
Figure 6 The architecture of neural network model

每个编码器块由两个子层连接结构组成，第一个子层连接结构包括一个多头注意力层和一个残差连接和归一化层，残差连接用于防止网络退化，归一化层用于对每一层的激活值进行归一化。第二个子层连接结构包括一个前馈全连接层和一个残差连接和归一化层。其中，多头注意力是编码器层的核心逻辑，它通过并行计算进一步优化了自注意力机制，增强了模型的表达能力。多头注意力结构如图7所示。

在自注意力机制中，对于每个输入的样本向量，分别乘以权重矩阵 $\mathbf{W}^Q$ 、 $\mathbf{W}^K$ 、 $\mathbf{W}^V$ 创建 **Query**、**Key**、**Value** 这3个向量，**Query**、**Key**、**Value** 的概念来自信息检索系统，如在某电商平台搜索某件商品时，在搜索引擎上输入的内容是 **Query**，搜索引擎根据 **Query** 匹配 **Key**（如商品的种类、颜色、描述等），然后根据 **Query** 和 **Key** 的相似度得到匹配的内容，即 **Value**。在一个句子中，每个单词都有 $\mathbf{Q}$ 、 $\mathbf{K}$ 、 $\mathbf{V}$ 这3个向量， $\mathbf{Q}$ 表示与该单词相匹配的单词的属性， $\mathbf{K}$ 表示该单词本身的属性， $\mathbf{V}$ 表示该单词本身的信息，这3个向量被应用于注意力机制计算该单词的注意力分数。

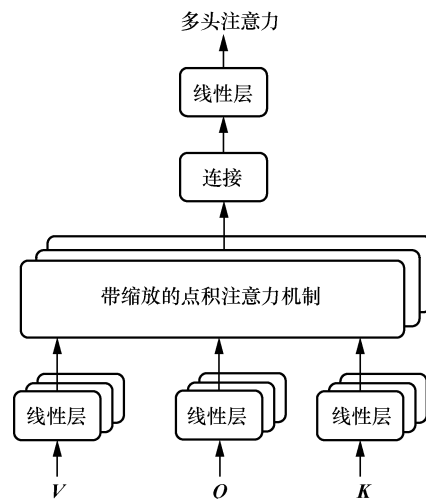


图7 多头注意力结构
Figure 7 The structure of multi-head attention

注意力分数的计算首先将矩阵 $\mathbf{Q}$ 与 $\mathbf{K}$ 转置相乘，通过向量点积表示单词间的关联程度，由此得到注意力权重矩阵，然后对权重矩阵进行缩放并执行 **softmax** 函数保证注意力权重的非负性，最后乘以矩阵 $\mathbf{V}$ ，即使用每个单词的权重与每个单词的特征进行加权求和，结果作为其注意力分数。多头注意力机制相当于对 n 个注意力头的集成，首先通过 n 次线性变换得到 n 组 $\mathbf{Q}$ 、 $\mathbf{K}$ 、 $\mathbf{V}$ ，对于每一组 $\mathbf{Q}_i$ 、 $\mathbf{K}_i$ 、 $\mathbf{V}_i$ ，分别计算得到相应的输出 Head_i ，拼接所有的 $\text{Head}_0 \sim \text{Head}_{n-1}$ ，把拼接后的矩阵和一个权重矩阵 $\mathbf{W}^O$ 相乘得到矩阵 $\mathbf{Z}$ ， $\mathbf{Z}$ 包含所有注意力头的信息，最后执行线性映射作为最终输出，具体计算过程如下。

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_0, \dots, \text{head}_{n-1}) \mathbf{W}^O \quad (1)$$

$$\text{head}_i = \text{Attention}(\mathbf{Q} \mathbf{W}_i^Q, \mathbf{K} \mathbf{W}_i^K, \mathbf{V} \mathbf{W}_i^V) \quad (2)$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q} \mathbf{K}^T}{\sqrt{d_K}} \right) \mathbf{V} \quad (3)$$

3 实验设计与结果分析

本节阐述具体的实验设计与实现，首先对本文所使用的数据集、评估指标等进行说明，然后引出4个研究问题以验证本文的实验效果，并对实验结果进行分析。

3.1 实验准备

3.1.1 数据集

为评估本文方法在现实世界中检测漏洞的有

效性,本文从国家安全漏洞数据库(NVD, national vulnerability database)收集已公布 CVE 编号的漏洞数据,共包含 2 338 个 CVE,漏洞样本包含修复前后的完整源文件及其漏洞修补提交信息,收集 Linux Kernel、FFmpeg、OpenSSL、FireFox 等 17 个开源软件,主要关注 10 种 CWE 漏洞类型,数据集信息如表 4 所示。本文系统最终生成 2 941 个有漏洞切片和 2 893 个无漏洞切片,在模型训练和测试阶段,切片样本按照 8:1:1 的比例随机划分为训练集、测试集和验证集。

表 4 数据集信息
Table 4 Information of dataset

CWE-ID	CVE 数量	CWE-ID	CVE 数量
CWE-119	970	CWE-190	95
CWE-125	220	CWE-369	30
CWE-787	88	CWE-415	18
CWE-20	366	CWE-416	129
CWE-189	241	CWE-476	181

3.1.2 评估指标

本文使用以下指标对模型效果进行评估。

假阳性 (FP, false positive): 模型预测结果为有漏洞,但实际为无漏洞的样本数量。

假阴性 (FN, false negative): 模型预测结果为无漏洞,但实际为有漏洞的样本数量。

真阳性 (TP, true positive): 模型预测结果为有漏洞,且实际为有漏洞的样本数量。

真阴性 (TN, true negative): 模型预测结果为无漏洞,且实际为无漏洞的样本数量。

准确率 (ACC, Accuracy): 模型预测结果与实际情况相符的样本数量占全部样本数量的比例。

$$Acc=(TP+TN)/(TP+TN+FP+FN)$$

精确率 (P, precision): 模型预测结果为有漏洞且与实际情况相符的样本数量占模型预测为有漏洞的全部样本数量的比例。

$$P=TP/(TP+FP)$$

召回率 (R, recall): 模型预测结果为有漏洞且与实际情况相符的样本数量占实际有漏洞的全部样本数量的比例。

$$R=TP/(TP+FN)$$

F1: 结合精确率和召回率,更加均衡地反映系统的漏洞检测性能。

$$F1=2PR/(P+R)$$

3.1.3 实验配置

本文的深度学习模型基于 PyTorch 实现,使用 Transformer 的编码器学习代码序列的深层特征,再使用全连接的线性层实现分类任务;使用网格搜索法进行调参,穷举搜索学习率、批大小、隐藏层数等超参数的最优组合。本文所使用的模型训练超参数设置如表 5 所示。

表 5 模型训练超参数设置
Table 5 Hyper-parameter settings of model training

参数	设置	参数	设置
编码器层数	3	注意力头数	8
损失函数	交叉熵损失	优化算法	Adam
学习率	0.000 5	随机失活率	0.5
批量大小	16	训练轮次	50

在经过 50 个轮次 (epoch) 训练后,模型在验证集上的损失和准确率基本稳定,能够成功收敛,模型训练的准确率和损失曲线如图 8 所示。

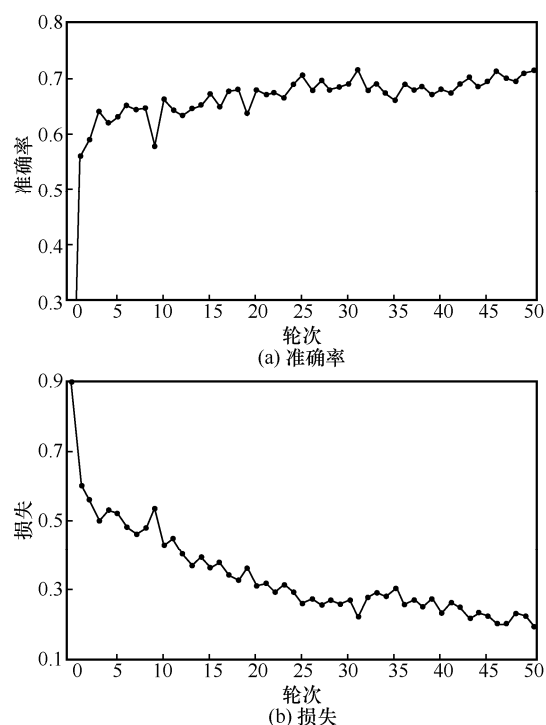


图 8 模型训练的准确率和损失曲线
Figure 8 Accuracy and loss curve of model training

本文方法应用 Python 语言在 Linux 操作系统

下实现,深度学习模型基于 PyTorch 后端实现,所使用的实验环境配置如表 6 所示。

3.2 实验结果与分析

本文实验用于回答以下 4 个研究问题。

RQ1: 本文方法所应用的源程序处理方法是能够有效提升切片质量,促进神经网络模型学习潜在的漏洞模式,提升漏洞检测效果?

RQ2: 本文方法所应用的神经网络模型相较于其他 RNN 是否具有更优的漏洞检测性能?

RQ3: 本文方法相较于其他源代码漏洞检测系统是否具有更优的漏洞检测性能?

RQ4: 本文方法是否能够有效检测真实软件中潜在的漏洞?

SySeVR^[8]提出的程序切片方法启发了大量相关工作^[30]的开展,本文在其基础上优化了切片算法。MVP^[9]将程序切片应用于函数的语义签名提取。

为了回答 RQ1,针对相同的数据集分别应用 SySeVR、MVP 和本文的过程间切片方法处理源程序,输入 Transformer 神经网络模型进行训练和测试,切片方法效果对比如表 7 所示。

程序切片的目的是提取目标程序中与漏洞相关的语句,以更加准确地描述漏洞语义信息。SySeVR 提出的切片方法将内存敏感的 API 函数调用、数组使用、指针使用及数值运算这 4 类语句定义为与漏洞相关的兴趣点,先在目标程序中识别这 4 类兴趣点,再由兴趣点出发依据 PDG 的数据依赖和控制依赖关系获取程序切片,生成的切片是 PDG 子图。然而, SySeVR 生成的切片存在以下两个问题:一是生成的切片大多与漏洞无关,二是上下文关系不够完整。目标程序中包含大量符合 4 类兴趣点的语句,识别的每个兴趣点会生成对应的切片,简单的筛选会导致切片数量过多、正负样本不平衡的现象。此外, SySeVR 未关注“被调用类型”的过程间切片,对于跨函数的漏洞样本无法准确描述漏洞触发的完整语义

信息,模型难以学习潜在的漏洞模式。

表 7 切片方法效果对比

切片方法	Acc	P	R	F1
SySeVR	60.2%	63.8%	59.6%	61.6%
MVP	61.8%	64.7%	61.5%	63.1%
本文	71.4%	75.0%	71.9%	73.4%

MVP 工作在函数级,基于补丁文件提取漏洞函数和补丁函数的语义签名用于复制检测任务。考虑到函数的语义完整性与包含的噪声信息之间的平衡, MVP 对控制语句、赋值语句、函数调用语句等不同类型的程序语句定义了不同的切片规则,如对于控制语句,则先回溯到变量定义语句再基于数据依赖关系获取前向切片;对于无返回值的函数调用语句,则基于参数的数据流关系获取前向切片,但其并未考虑过程间切片,无法处理跨函数的漏洞样本,具有较大的局限性。

本文对 SySeVR 的切片方法进行了改进,对于过程内切片,改变了控制语句的前向切片规则以平衡程序切片的语义完整性和包含的噪声之间的关系;对于过程间切片,引入了函数调用的隐式返回值和漏洞函数返回后继续执行的语句部分,进一步完善了程序切片的语义信息,使其能够表示漏洞触发的完整因果关系。为了观察与漏洞位置直接相关的代码片段,本文将补丁语句作为兴趣点提取程序切片,避免了训练集中包含大量与漏洞无关的样本问题。此外,本文在初步生成的切片中通过自定义的规则识别 source 和 sink 语句,能够进一步减少切片中的噪声,有利于模型学习潜在的漏洞模式。由表 7 可知,应用本文方法处理源程序生成的切片能够更加有效地表示漏洞的语义特征,本文方法的 F1 分数相较于 SySeVR 和 MVP 切片方法分别提升了 11.8%和 10.3%,漏报率和误报率都较低。

表 6 实验环境配置

Table 6 Experimental environment configuration

硬件配置				软件配置	操作系统
CPU 型号	GPU 型号	运行内存	硬盘大小		
Intel Xeon Gold 6234 CPU @ 3.30 GHz	Quadro RTX 5000	132 GB	4 TB	Python2.7, Joern-0.3.1, Neo4j-community-2.1.5, Python3.6, PyTorch-1.5.1, VS Code	Linux version 5.4.0-77-generic

为了回答 RQ2, 本文生成程序切片, 将其分别输入 BGRU、BLSTM 及 Transformer 神经网络模型进行训练和测试, 模型效果对比如表 8 所示。

表 8 模型效果对比
Table 8 Comparison with model effect

模型	Acc	P	R	F1
BGRU	57.4%	55.0%	60.4%	57.5%
BLSTM	59.1%	56.5%	63.3%	59.7%
Transformer	71.4%	75.0%	71.9%	73.4%

基于 Transformer 的神经网络模型已被应用于文本翻译等自然语言处理任务, 相较传统的循环神经网络表现出了更加优越的性能。基于 BGRU 或者 BLSTM 的模型存在以下两个问题, 一是当输入序列过长时性能下降, 二是在处理序列时无法并行化计算。循环神经网络在计算隐藏状态时, 包含其他词的信息与当前词的距离相关, 存在信息衰减的现象, 而 Transformer 在对当前词进行计算时, 包含其他词的信息不取决于距离, 而取决于两者的相关性。此外, 循环神经网络是时序相关的顺序结构, 计算效率较低, 而 Transformer 通过多头注意力机制同时处理整个句子, 且通过位置编码信息弥补了时序信息, 相较于循环神经网络有明显优势。由表 8 可知, Transformer 神经网络模型的 F1 分数相较于传统的 BGRU 模型和 BLSTM 模型分别提升 15.9% 和 13.7%, 验证了 Transformer 模型在智能漏洞检测领域的有效性。

为了回答 RQ3, 本文针对相同的数据集使用多种静态漏洞检测方法进行实验, 包括两种基于规则的漏洞检测方法 (Checkmarx^[37] 和 FlawFinder^[38]) 和两种基于深度学习的漏洞检测方法 (SySeVR^[8] 和 Devign^[12])。SySeVR 是切片级漏洞检测的代表工作, 应用循环神经网络 Bi-LSTM 实现, Devign 是函数级漏洞检测的代表工作, 应用图形神经网络 GGNN 实现。漏洞检测方法效果对比如表 9 所示。

基于规则的漏洞检测方法依赖于专家根据经验定义的漏洞规则, 结合词法和语法分析技术通过模式匹配识别目标程序中的漏洞。由于真实软件中漏洞存在的形式多样, 专家知识无法覆盖全部的漏洞模式, 该类方法存在较大的局限性。由表 9 可知, 基于规则的漏洞检测方法均表现不佳, Checkmarx 的召回率仅为 24.2%, FlawFinder 的召

回率为 26.6%, 存在漏报率较高的问题。基于深度学习的漏洞检测方法能够自动挖掘程序中潜在的漏洞模式, 避免了高额的人工成本, 近年来被广泛应用于漏洞检测。SySeVR 和 Devign 是其中的代表性工作, Devign 工作在函数级, 相较于切片级的方法, 存在更多与漏洞无关的语句干扰模型训练, 且完全无法处理跨函数的漏洞样本, 不完整的上下文信息导致其模型效果欠佳。本文对 SySeVR 的程序切片方法进行了改进, 在对漏洞样本执行完整上下文分析的同时进一步减少了程序样本中的噪声语句, 并应用了更加先进的 Transformer 神经网络模型进行漏洞检测, F1 分数相较于 SySeVR 模型和 Devign 模型分别提升了 19.2% 和 13.6%, 检测效果更佳。

表 9 漏洞检测方法效果对比
Table 9 Comparison with vulnerability detection methods effect

检测方法	Acc	P	R	F1
Checkmarx	50.2%	50.3%	24.2%	32.6%
FlawFinder	50.4%	50.7%	26.6%	34.9%
SySeVR	60.0%	51.2%	57.7%	54.2%
Devign	58.8%	58.5%	61.1%	59.8%
本文方法	71.4%	75.0%	71.9%	73.4%

为了回答 RQ4, 本文使用 NVD 数据集之外的多个真实软件漏洞样本进行实验, 并以开源关系型数据库管理系统 openGauss 作为检测对象进一步验证本文系统的实用性。具体来说, 首先根据表 3 所示的目标漏洞类型的 sink 语句特征在源代码中识别潜在的 sink 语句, 如对于 double free 或者 use after free 的漏洞类型, 识别源代码中包含的 free/delete 语句并提取其后向切片, 输入训练完成的深度学习模型进行检测。当检测结果为阳性, 即有漏洞时, 本文使用人工分析漏洞因果关系的方法确定模型的检测结果是否正确。

本节对 Linux Kernel 中的一个漏洞实例进行详细分析。该漏洞类型为 CWE-119 缓冲区溢出漏洞, 函数 brcmf_cfg80211_mgmt_tx 允许本地用户通过精心设计的 NL80211_CMD_FRAME 网络链接数据包造成拒绝服务攻击。具体来说, 漏洞函数在调用 memcpy 执行内存复制时, 未对缓冲区大小进行验证, 继而引发越界访问, 修复后的函数通过添加验证条件避免漏洞被触发。本文方法

首先在源代码中识别到敏感的 API 调用，然后基于控制依赖和数据依赖关系提取其后向切片，转化为向量后输入深度学习模型进行检测，最终能够正确识别该切片样本中存在的漏洞。真实漏洞检测实例分析如图 9 所示。

此外，通过自动化检测再进行人工分析验证的方式，本文方法在目标软件 openGauss 中检测出 3 个潜在的漏洞，分别为空指针解引用、double free 及除零漏洞。表 10 为通过人工分析验证的漏洞列表，详细信息已被提交给厂商进行审核。

4 结束语

本文针对现有基于深度学习的漏洞检测方法在面对真实软件时检测效果不佳的问题进行优

化，对补丁语句执行全面的上下文分析以提取漏洞程序的完整语义信息，能够有效处理跨函数的漏洞样本，并平衡了程序切片的语义完整性与包含的冗余信息之间的关系，有利于方法学习潜在的漏洞模式。此外，本文将 Transformer 神经网络模型应用于漏洞检测领域，通过与其他静态漏洞检测方法的实验对比，证实了本文方法的有效性。

目前的工作还存在一定的局限性。本文仅研究了 10 种漏洞类型，无法全面反映真实软件中漏洞模式的多样性，未来需要研究更加广泛的漏洞语法特征。此外，数据集标注是一项重要的工作，本文所应用的数据集规模较小，未来需要持续关注来自真实软件的漏洞样例，构建基准数据集以帮助研究人员进行漏洞挖掘工作。

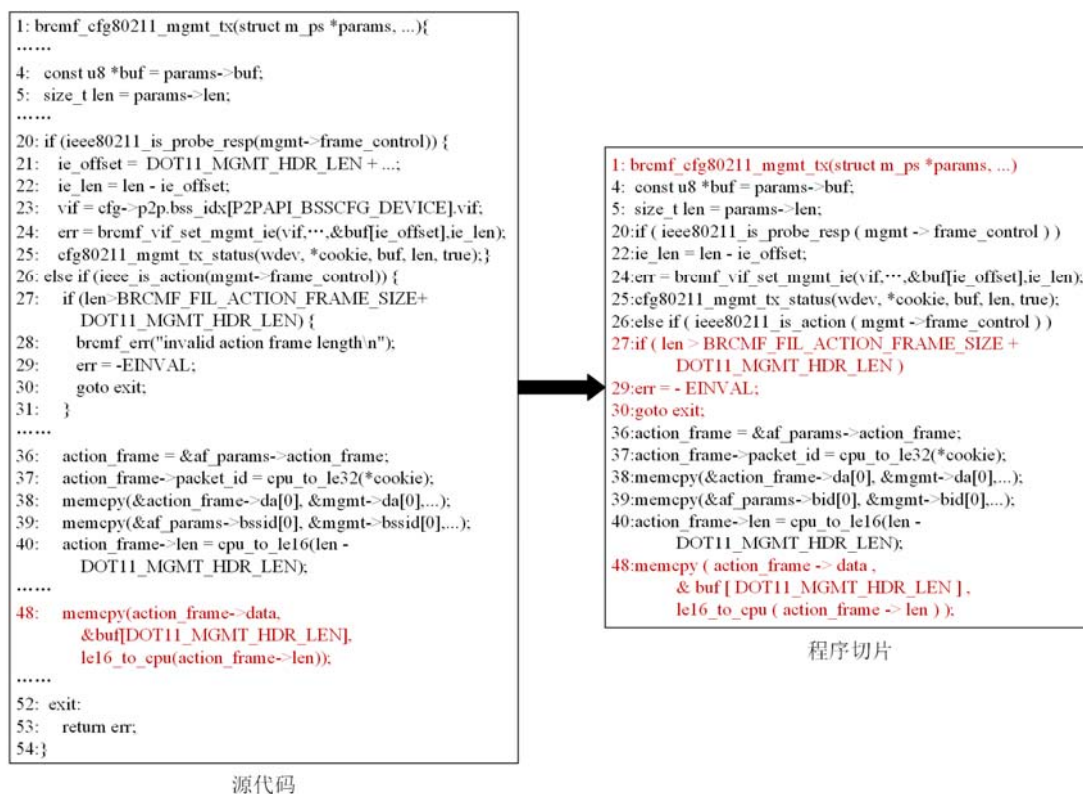


图 9 真实漏洞检测实例分析
Figure 9 A detection case of a real vulnerability

表 10 通过人工分析验证的漏洞列表

Table 10 List of vulnerabilities found in openGauss and verified by manual analysis

漏洞类型	漏洞文件路径	漏洞成因分析
CWE-476	src/storage/gstor/zekernel/kernel/table/*.c	置空指针可能引起空指针解引用风险，需对其进行初始化
CWE-415	src/gausskernel/optimizer/commands/*.cpp	存在 double free 安全隐患，在释放指针后应及时将其置为 NULL
CWE-369	src/common/backend/utills/adt/*.cpp	被除数被初始化为 0，但在除以该变量时未判断其是否为零

参考文献:

- [1] CVE [EB].
- [2] SKYBOX SECURITY[EB].
- [3] WU T M, WEN S, XIANG Y, et al. Twitter spam detection: survey of new approaches and comparative study[J]. *Computers & Security*, 2018, 76(7): 265-284.
- [4] LECUN Y, BENGIO Y, HINTON G. Deep learning[J]. *Nature*, 2015, 521(7553): 436-444.
- [5] LI Z, ZOU D Q, XU S H, et al. VulDeePecker: a deep learning-based system for vulnerability detection[C]//The Network and Distributed System Security Symposium (NDSS). 2018: 18-21.
- [6] CHAKRABORTY S, KRISHNA R, DING Y, et al. Deep learning based vulnerability detection: are we there yet[J]. *IEEE Transactions on Software Engineering (TSE)*, 2021, 48(9): 3280-3296.
- [7] DAM H K, TRAN T, PHAM T, et al. Automatic feature learning for predicting vulnerable software components[J]. *IEEE Transactions on Software Engineering (TSE)*, 2018, 47(1): 67-85.
- [8] LI Z, ZOU D Q, XU S H, et al. SySeVR: a framework for using deep learning to detect software vulnerabilities[J]. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2021, 19(4): 2244-2258.
- [9] XIAO Y, CHEN B H, YU C D, et al. MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures[C]//Proceedings of USENIX Security Symposium. 2020.
- [10] LIN G J, WEN S, HAN Q L, et al. Software vulnerability detection using deep neural networks: a survey[J]. *Proceedings of the IEEE*, 2020, 108(10): 1825-1848.
- [11] LIN G J, ZHANG J, LUO W, et al. POSTER: vulnerability discovery with function representation learning from unlabeled projects[C]//Proceedings of the Conference on Computer and Communications Security (CCS). 2017.
- [12] ZHOU Y Q, LIU S Q, SIOW J, et al. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks[C]//Proceedings of Annual Conference on Neural Information Processing Systems (NeurIPS). 2019.
- [13] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is all you need[C]//Proceedings of Advances in Neural Information Processing Systems (NIPS). 2017.
- [14] GHAFARIAN S M, SHAHRIARI H R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques[J]. *ACM Computing Surveys (CSUR)*, 2017, 50(4): 1-36.
- [15] ACHARYA M. Mining API patterns as partial orders from source code: from usage scenarios to specifications[C]//Proceedings of Joint Meeting of the European Software Engineering Conference & the ACM Sigsoft Symposium on the Foundations of Software Engineering. 2007.
- [16] SCANDARIATO R, WALDEN J, HOVSEPYAN A, et al. Predicting vulnerable software components via text mining[J]. *IEEE Transactions on Software Engineering (TSE)*, 2014, 40(10): 993-1006.
- [17] YAMAGUCHI F, WRESSNEGGER C, GASCON H, et al. Chucky: exposing missing checks in source code for vulnerability discovery[C]//Proceedings of the Conference on Computer & Communications Security (CCS). 2013.
- [18] WHITE M, VENDOME C, LINARES-VASQUEZ M, et al. Toward deep learning software repositories[C]//Proceedings of IEEE/ACM Working Conference on Mining Software Repositories. 2015.
- [19] WANG S, LIU T Y, TAN L. Automatically learning semantic features for defect prediction[C]//Proceedings of IEEE/ACM 38th International Conference on Software Engineering (ICSE). 2016.
- [20] YAMAGUCHI F, GOLDE N, ARP D, et al. Modeling and discovering vulnerabilities with code property graphs[C]//IEEE Symposium on Security and Privacy (S&P). 2014.
- [21] SHAR L K, BRIAND L, TAN H. Web application vulnerability prediction using hybrid program analysis and machine learning[J]. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2015, 12(6): 688-707.
- [22] LIN G J, ZHANG J, LUO W, et al. Cross-project transfer representation learning for vulnerable function discovery[J]. *IEEE Transactions on Industrial Informatics*, 2018, 14(7): 3289-3297.
- [23] LIN G J, XIAO W, ZHANG J, et al. Deep Learning-Based Vulnerable Function Detection: A Benchmark[C]//Proceedings of International Conference on Information and Communications Security (ICICS). 2019.
- [24] 李韵, 黄辰林, 王中锋, 等. 基于机器学习的软件漏洞挖掘方法综述[J]. *软件学报*, 2020, 31(7): 2040-2061.
- LI Y, HUANG C L, WANG Z F, et al. Survey of software vulnerability mining methods based on machine learning[J]. *Journal of Software*. 2020, 31(7): 2040-2061.
- [25] WANG H T, YE G X, TANG Z Y, et al. Combining graph-based learning with automated data collection for code vulnerability detection[J]. *IEEE Transactions on Dependable and Secure Computing (TIFS)*, 2020, 16: 1943-1958.
- [26] LI Y, WANG S H, NGUYEN T N. Vulnerability detection with fine-grained interpretations[C]//Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). 2021.
- [27] YING R, BOURGEOIS D, YOU J X, et al. GNNExplainer: generating explanations for graph neural networks[C]//Proceedings of International Conference on Neural Information Processing Systems (NIPS). 2019.
- [28] SONNEKALB T. Machine-learning supported vulnerability detection in source code[C]//Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). 2019.
- [29] RUSSELL R L, KIM L, LEI H H, et al. Automated vulnerability detection in source code using deep representation learning[C]//2018 IEEE 17th International Conference on Machine Learning and Applications (ICMLA). 2018.
- [30] CHENG X, WANG H Y, HUA J Y, et al. DeepWukong: statically detecting software vulnerabilities using deep graph neural network[J]. *ACM Transactions on Software Engineering and Method-*

ology (TOSEM 2021), 2021, 30(3): 1-33.

- [31] 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[J]. 软件学报, 2020, 31(11): 3404-3420.
DUAN X, WU J Z, LUO T Y, et al. Vulnerability mining method based on code property graph and attention BiLSTM[J]. Journal of Software, 2020, 31(11): 3404-3420.
- [32] 陈肇炫, 邹德清, 李珍, 等. 基于抽象语法树的智能化漏洞检测系统[J]. 信息安全学报, 2020, 5(4): 1-13.
CHEN Z X, ZOU D Q, LI Z, et al. Intelligent vulnerability detection system based on abstract syntax tree[J]. Journal of Cyber Security, 2020, 5(4): 1-13.
- [33] 胡雨涛, 王溯远, 吴月明, 等. 基于图神经网络的切片级漏洞检测及解释方法[J]. 软件学报, 2023, 34(6): 65-82.
HU Y T, WANG S Y, WU Y M, et al. A Slice-level vulnerability detection and interpretation method based on graph neural network[J]. Journal of Software, 2023, 34(6): 65-82.
- [34] Joern[EB]
- [35] Neo4j graph platform[EB]
- [36] MIKOLOV T, CHEN K, CORRADO G, et al. Efficient estimation of word representations in vector space[J]. Computer Science, 2013(1).
- [37] Checkmarx[EB].
- [38] Flawfinder[EB].

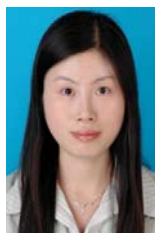
[作者简介]



李妍(1998-), 女, 陕西渭南人, 华中科技大学硕士生, 主要研究方向为深度学习和漏洞检测。



羌卫中(1977-), 男, 江苏南通人, 华中科技大学教授、博士生导师, 主要研究方向为机密计算、云计算安全、软件安全。



李珍(1981-), 女, 河北保定人, 博士, 华中科技大学副教授, 主要研究方向为软件安全和人工智能安全。



邹德清(1975-), 男, 湖南湘潭人, 博士, 华中科技大学教授、博士生导师, 主要研究方向为云计算安全、网络攻防与漏洞检测、软件安全、隐私保护。



金海(1966-), 男, 上海人, 博士, 华中科技大学教授、博士生导师, 主要研究方向为计算机体系结构、虚拟化技术、集群计算和云计算、存储与安全。