Slice-GCN:基于程序切片与图神经网络的智能合约漏洞检测方法

张人娄, 吴　胜, 张　浩, 刘方宇

江苏师范大学 计算机科学与技术学院 徐州 中国 211116

摘要　智能合约是一段由计算机代码构成的程序。随着智能合约数量的暴涨, 如何利用漏洞检测方法来提升智能合约的安全性显得更加重要。已有的符号执行、模糊测试与形式化验证等漏洞检测方法自动化程度低, 而基于序列模型的深度学习方法由于对智能合约源代码的特征挖掘不足导致检测结果的精度偏低。因此, 本文提出一个基于程序切片与图神经网络的以太坊智能合约(简称智能合约)漏洞检测方法 Slice-GCN。该方法先对程序进行代码预处理简化程序, 再使用基于图可达性和数据流方程的程序切片方法对预处理后的程序进行切片, 并将切片结果输入长短期记忆网络(LSTM)中提取智能合约的程序语义特征。接着, 简化程序依赖图后将其输入图卷积神经网络中, 并提取智能合约的程序结构特征。然后, 将智能合约的程序语义特征和结构特征拼接后输入多层感知机(MLP)中, 并对智能合约进行漏洞检测。在提出 Slice-GCN 方法的基础上, 针对重入攻击、时间戳依赖及整数溢出三类漏洞, 本文对 Slice-GCN 方法与 Oyente、Osiris 和 Solidity check 三款智能漏洞检测工具进行了对比实验, 并且通过消融实验分析了程序切片、图神经网络及图收缩比例对实验结果的影响。实验结果表明本文提出的方法在各类指标上均有较大提升, 能有效提升检测准确度和精度, 降低误报率, 同时在检测速度上也明显优于传统的智能合约漏洞检测工具。

关键词　智能合约; 漏洞检测; 图神经网络; 程序切片

中图法分类号　TP311　DOI 号 10.19363/J.cnki.cn10-1380/tn.2025.01.08

Slice-GCN: Smart Contract Vulnerability Detection Based on Program Slicing and Graph Neural Networks

ZHANG Renlou, WU Sheng, ZHANG Hao, LIU Fangyu

College of Computer Science and Technology, Jiangsu Normal University, Xuzhou 211116, China

Abstract　A smart contract is a program made up of computer code. With the skyrocketing number of smart contracts, how to use vulnerability detection methods to improve the security of smart contracts becomes more important. Existing vulnerability detection methods such as symbolic execution, fuzz testing, and formal verification have a low degree of automation, while deep learning methods based on sequence models have low detection accuracy due to insufficient feature mining of smart contract source code. Therefore, this paper proposes a vulnerability detection method Slice-GCN for Ethereum smart contracts (smart contracts for short) based on program slices and graph neural networks. This method first preprocesses the code of the program to simplify the program, and then uses the program slicing method based on graph accessibility and data flow equations to slice the preprocessed program, and input the slicing results into the long short-term memory network (LSTM) to extract the program semantic features of the smart contract. Then, the simplified program dependency graph is fed into the graph convolutional neural network, and the program structure features of the smart contract are extracted. Then, the program semantic features and structural features of the smart contract are spliced and input into the multi-layer perceptron (MLP), and the smart contract is tested for vulnerabilities. On the basis of proposing the Slice-GCN method, aiming at the reentrancy attack, timestamp dependency and integer overflow three types of vulnerabilities, this paper compared the Slice-GCN method with three smart contract vulnerability detection tools Oyente, Osiris and Solidity check, and passed the ablation experiments analyze the effects of program slicing, graph neural network, and graph shrinkage ratio on the experimental results. The experimental results show that the method proposed in this paper has greatly improved various indicators, can effectively improve the detection accuracy and precision, and reduce the false positive rate. At the same time, the detection speed is also significantly better than the traditional smart contract vulnerability detection tools.

Key words　smart contract; vulnerability detection; graph neural network; program slicing

通讯作者: 吴胜, 博士, 副教授, Email: woodstone1978@jsnu.edu.cn。

本课题得到江苏师范大学科研与实践创新项目(No. 2022XKT1548)资助

收稿日期: 2023-03-15; 修改日期: 2023-08-08; 定稿日期: 2024-11-18

1 引言

智能合约最早由 Nick Szabo 在 20 世纪 90 年代提出[1], 是一种旨在以数字化形式制定、传播、验证及执行合同的计算机协议, 往往表现为一段由计算机代码构成的程序。区块链技术[2]为智能合约提供了去中心化和不可篡改的可信运行平台, 使得智能合约在金融、法律合同等领域得到了广泛的应用。

智能合约程序源代码难免会出现漏洞。一旦智能合约中的漏洞被不法分子恶意利用, 可能会造成严重的经济损失。例如, 2016 年的 The DAO[3]事件中价值超过 6000 万美元的以太币被盗。由于区块链具有不可篡改性, 智能合约一旦被部署就无法修改, 很有必要在部署智能合约之前对其进行漏洞检测。因此对智能合约的漏洞检测方法进行研究有一定的理论意义和应用价值。

传统的符号执行[4-5]、模糊测试[6-7]、形式化验证[8-9]等漏洞检测方法主要通过实际运行、模拟运行或形式化分析智能合约程序来检测智能合约是否存在漏洞。此类方法虽然在代码覆盖率上有一定的优势, 但缺点也十分明显。首先, 自动化程度低, 无法适应智能合约不断更新、迭代的现状。其次, 适用范围窄, 检测效率低。例如, 符号执行存在的路径爆炸问题会导致整个系统的执行效率变低。

近年来, 机器学习和深度学习等方法被应用于检测智能合约漏洞[10]。尽管这些检测方法自动化程度较高, 但是由于输入的数据不够丰富、不具有代表性, 以及无法充分挖掘智能合约的程序特征等原因导致检测结果的精度不高。

为了解决符号执行等漏洞检测方法的效率低和深度学习方法检测结果精度不高等问题, 本文提出了基于程序切片和图神经网络的以太坊智能合约(简称智能合约)漏洞检测方法 Slice-GCN。该方法先利用程序切片技术提取包含漏洞的关键代码, 再提取程序的语义特征和结构特征, 然后检测智能合约的漏洞。实验表明该方法提高了检测精度和准确度, 降低了误报率。本文的主要贡献包括:

1) 提出使用基于图可达性与数据流方程的程序切片方法来提取包含漏洞的关键代码, 再使用序列模型和图神经网络结合的方法进行漏洞检测。

2) 在由节点特征得到图的特征表示时使用了基于注意力分数的图收缩方法, 实验结果表明, 使用图收缩方法能够显著提高检测性能。

3) 实验结果表明, 对于智能合约的重入攻击、时间戳依赖、整数溢出三类漏洞, Slice-GCN 方法比工具 Oyente、Osiris 及 Solidity check 在准确率、精确率等方面表现更好, 检测速度更快。

2 智能合约漏洞

在智能合约的实际应用中, 重入攻击、时间戳依赖和整数溢出三类漏洞是相对严重的漏洞, 已经导致区块链领域遭受了严重的经济损失[11]。例如, 重入攻击漏洞给区块链应用领域造成了较大损失, 仅在 The DAO 事件中造成的损失就超过了 6000 万美元。这也是已有研究对此三类漏洞极为重视的原因。

本文的统计结果也反映了检测这三类漏洞的重要性, 本文检查了 85753 份智能合约, 其中有 4090 份合约调用了 call.value 函数, 调用该函数可能会导致重入漏洞; 42626 份合约调用了 block.timestamp 或 now, 存在时间戳依赖漏洞的风险。于是, 本文主要分析此三类漏洞。

2.1 重入攻击漏洞

重入攻击是智能合约中较常见的漏洞。重入攻击中被攻击合约的样例程序如程序 1 所示, 攻击合约的样例程序如程序 2 所示。程序 1 合约允许用户存储、取出以太币, 合约中的 withdraw 函数先判断用户取出的以太币是否超过用户剩余的以太币, 再通过调用 call.value 函数取出以太币, 并扣除调用者(即用户)的余额。程序 2 中 Attack 合约的 pwnEtherStore 函数调用了程序 1 中的 withdraw 函数, 在执行完 withdraw 函数中的第 2 条语句 require(msg.sender.call.value(*amount*)())之后将会执行 Attack 合约中的 fallback 函数(即程序 2 中第 10～12 条语句), 而 fallback 函数中再次调用了 withdraw 函数, 因此 EtherStore 合约的 withdraw 函数第 3 条语句不会被执行, 即调用者的余额不会减少, 仍然可以通过 withdraw 函数第 1 条语句的验证条件, 最终可能造成 EtherStore 合约中的以太币被盗取。

```
contract EtherStore{
    mapping (address=>uint) public balances;
    function depositFunds() public payable{
        balances[msg.sender] += msg.sender;
    }
    function withdraw(uint amount) public{
        require(balances[msg.sender]>= amount);
        require(msg.sender.call.value(amount)());
        balances[msg.sender] -= amount;
    }
}
```

程序 1 重入攻击漏洞被攻击合约样例

Program 1 Example of contract for reentrancy vulnerability being attacked

```
contract Attack{
    EtherStore public etherStore;
    constructor(address _etherStoreAddress){
        etherStore = etherStore(_etherStoreAddress);
    }
    function pwmEtherStore() public payable{
        require(msg.value >= 1 ether);
        etherStore. withdraw(1 ether);
    }
    function () payable{
        etherStore. withdraw(1 ether);
    }
}
```

程序 2 重入攻击漏洞攻击合约样例

Program 2 Example of reentrancy vulnerability attack contract

2.2 时间戳依赖漏洞

时间戳依赖是指智能合约中调用了时间戳, 并将时间戳等与区块相关的变量值作为随机种子, 而且将其用在关键操作的判断条件中。矿工可以在一定程度上修改或控制时间戳来实施攻击以获得意外的利益或影响合约的执行结果。该漏洞可能会导致合约出现逻辑漏洞, 进而导致合约的安全性受到威胁。如程序 3 所示的第 3 条语句将时间戳作为判断条件, 如果攻击者将时间戳设置为 15 的整数倍, 就能通过程序 3 中的第 4 条语句转移合约中的以太币。

```
fallback() external payable{
    require(msg.value == 1 ether);
    if(block.timestamp % 15 == 0){
        payable(msg.sender)
          .transfer(address(this).balance);
    }
}
```

程序 3 时间戳依赖漏洞合约样例

Program 3 Example of timestamp dependency vulnerability contract

2.3 整数溢出漏洞

整数溢出漏洞是一种常见的漏洞, 许多程序(不仅仅是智能合约)中都存在整数溢出的潜在风险。整数溢出漏洞的样例如程序 4 所示。程序 4 中第 2 条语句为条件判断语句。假设 *balances[msg.sender]*值为 0, *_value* 为 1, 则 *balances[msg.sender]*小于*_value*, 在实际执行时这两个变量相减后的结果不是–1, 而会发生溢出, 使得运算结果变为一个极大的正数, 即 *balances[msg.sender]-_value* 大于 0。这样会影响到后面语句的执行情况。

```
function transfer(address _to, uint _value) {
    require(balances[msg.sender] - _value >= 0);
    balances[_to] += _value;
}
```

程序 4 整数溢出漏洞合约样例

Program 4 Example of integer overflow vulnerability contract

3 方案设计

如图 1 所示为 Slice-GCN 方法的总体结构图, 该方法分为三个阶段: 1)代码预处理阶段, 为了便于后续操作, 先对数据集(即智能合约)的代码进行了预处理; 2)提取程序语义特征阶段, 此阶段先用基于图可达性和数据流方程的程序切片方法提取包含漏洞的关键代码, 并通过 LSTM 提取智能合约的语义特征; 3)提取程序结构特征阶段, 此阶段先将程序源代码转换为程序依赖图(Program dependence graph, PDG), 并使用图卷积神经网络(Graph convolutional network, GCN)和基于注意力的边收缩的方法提取程序的结构特征。

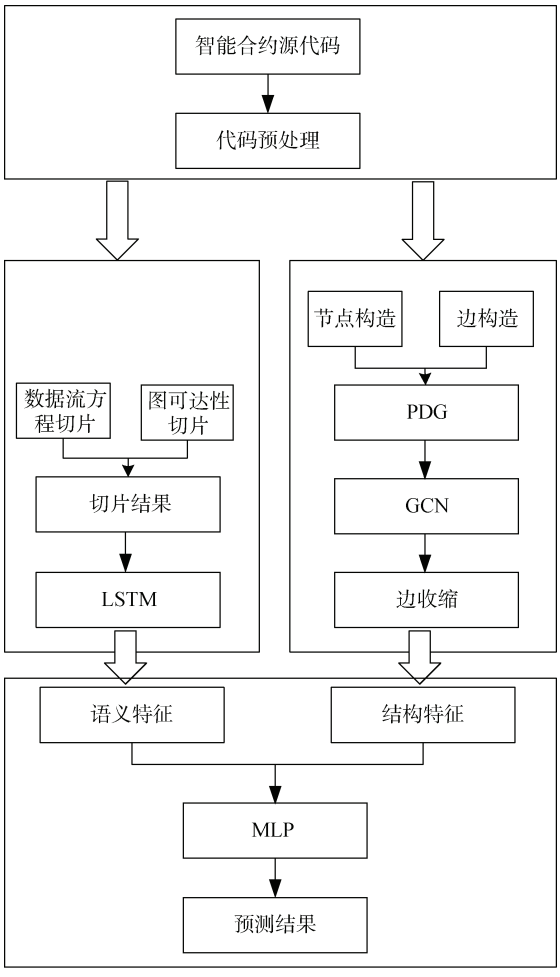


图 1 Slice-GCN 总体结构

Figure 1 The Structure of Slice-GCN

4 代码预处理

由于开发人员的开发习惯(如函数命名方式)和智能合约内容不同等方面的原因, 智能合约源代码中除了智能合约语言本身的关键字之外, 用户自定义的函数名和变量名等可能不相同, 但程序的功能可能相同。因此本文对智能合约代码进行以下四种规范化(预处理)操作:

1) 关键字规范化: 在智能合约中存在着许多含义相近的关键字, 如 uint8 和 uint16 等都表示整型, 只是取值范围和所占空间大小不同。于是, 将具有相近功能的关键字统一转换成同一个关键字, 如 uint8、uint16 和 uint32 等都统一转换成 uint。

2) 变量名规范化: 智能合约中变量主要分为普通变量和状态变量, 状态变量一般由关键字 public 或者 private 修饰。如图 2 所示的程序中, 第 3 行代码中的变量 *_value* 为普通变量, 第 4 行的变量 *cnt* 为状态变量。为了统一用户定义的变量名, 将所有的普通变量改为 *NORMALVAR_X* 的形式, 将所有的状态变量改为 *STATEVAR_X*的形式, 其中 *X*为该类型变量在智能合约源代码中出现的次序。

3) 操作符与操作数规范化: 由于操作符的变体形式较多, 因此先将操作符进行变换。例如, 将加法、减法和乘除法等操作符转换成 OPERATE; 将赋值操作符转换成 EQUAL,再将操作数转换成 NUMBER。

4) 失活变量重定义: 在静态单赋值技术(Static single assignment form, SSA)中每个变量只能被赋值一次, 从而保证了每个变量都拥有唯一的定义。在程序中对同一变量出现互不影响的多次调用, SSA 将这些调用转换为对不同变量的调用, 以此来消除大量多余且不相关的数据与控制依赖关系。假设第 *i* 条语句定义了变量 *x*, 之后 *x* 又在第 *j* 条语句(*j*>*i*)被重新赋值, 且 *x* 在第 *i* 语句和第 *j* 条语句中的取值互不影响, 那么称第 *i* 条语句定义的变量 *x* 为失活变量。由于失活变量在赋值前后互不影响, 因此可以将失活变量进行重定义。如图 2 中的程序中将变量 *_value* 从 *NORMALVAR*_2 重新定义为 *NORMALVAR*_4。

如图 2 所示为示例程序规范化前后的结果。

```
address[] _receivers;
mapping(address=>uint) balances;
uint256 _value;
uint public cnt= _receivers.length;
uint256 amount = uint256(cnt) * _value;
require(cnt>0 && cnt<=20);
_value = _value.add(10);
balances[msg.sender] = balances[msg.sender].Sub(amount);
for(uint i = 0;i<cnt;i++){
balances[_receivers[i]] = balances[msg.sender].add(_value);
Transfer(msg.sender, _receivers[i], _value);}
```

```
01  address[] NORMALVAR_0;
02  mapping(address=>uint) NORMALVAR_1;
03  uint NORMALVAR_2;
04  uint public STATEVAR_0 EQUAL NORMALVAR_0.length;
05  uint NORMALVAR_3 EQUAL uint(STATEVAR_0) * NORMALVAR_2;
06  require(STATEVAR_0 OPERATE NUMBER OPERATE STATEVAR_0 OPERATE NUMBER);
07  uint NORMALVAR_4 EQUAL NORMALVAR_2.add(NUMBER);
08  NORMALVAR_1[msg.sender] EQUAL NORMALVAR_1[msg.sender].Sub(NORMALVAR_3);
09  for(uint NORMALVAR_5 EQUAL NUMBER ;
10  NORMALVAR_5 OPERATE STATEVAR_0{;
11  NORMALVAR_1[NORMALVAR_0[NORMALVAR_5]] EQUAL
    NORMALVAR_1[NORMALVAR_0[NORMALVAR_5]].add(NORMALVAR_4);
12  Transfer(msg.sender, NORMALVAR_0[NORMALVAR_5], NORMALVAR_4);
13  NORMALVAR_5++;}
```

图 2 程序规范化示例

Figure 2 Example of procedure standardization

在代码预处理阶段, 只是进行变量以及关键字的替换, 不会改变程序的语义和结构特征。

5 程序语义特征提取

通常情况下, 漏洞涉及到的语句可能只占整个程序的小部分, 如果将整个合约程序作为神经网络的输入不仅会降低检测的准确率, 而且可能会导致模型训练和检测速度降低。删除无关代码、提取包含漏洞的关键代码, 不仅能够增强漏洞的表达力度, 还能够提高检测效率。

由于数据的变化取决于数据的运算和赋值(即数据依赖), 而语句之间的执行逻辑取决于语句之间的执行关系(即控制依赖)。于是, 基于图可达性与数据流方程的程序切片以数据为中心对程序切片, 保证了对漏洞语句的覆盖率。

5.1 切片规则

切片规则为一个二元组<*n*,*V*>, 其中 *n* 表示程序中的某个点, 一般为某条语句或程序基本块, *V* 表示语句或程序基本块中定义和使用的变量集合。

本文基于对漏洞发生环境的分析来确定切片规则, 保证了切片规则的合理性。智能合约多数漏洞是由数据使用不当引起的, 使用不当的数据主要分为三类, 与转账相关的数据、区块数据和参数数据。围绕这三类数据, Slice-GCN 方法定义了如下切片规则:

1) 与转账数据相关的语句: 在智能合约中经常需要转移以太币。一旦涉及到转账(转移以太币)就有可能存在重入漏洞, 因此将与转账数据相关的语句作为切片规则。

2) 与区块数据相关的语句: 任何人都能随意调用区块上的数据, 甚至能够在一定范围内对其进行

修改。如果智能合约将区块数据(如 *block.timestamp*)作为判断条件, 就可能会引起漏洞。因此将与区块数据相关的语句作为切片规则。

3) 与参数数据相关的运算语句: 智能合约对外部调用传入的参数数据进行运算时可能发生整数溢出等漏洞, 因此智能合约在与外部传入的参数数据进行运算之后通常需要对运算结果进行验证。因此将与参数数据运算的语句作为切片规则。

5.2 基于图可达性的程序切片

在程序依赖图中, 主要涉及到以下定义:

定义 1(数据依赖): 如果语句 P_u 定义了变量 V, 而且在语句 P_u 之后的某条语句 P_i 中使用了变量 V, 则称语句 P_i 数据依赖于语句 P_u。

定义 2(控制依赖): 对于语句 P_i 和 P_j, 如果语句 P_j 的执行与否受语句 P_i 影响, 则称 P_j 控制依赖于 P_i。

通过对智能合约源代码进行静态分析, 可以由程序代码中语句之间的的数据依赖和控制依赖关系得到程序依赖图。程序依赖图为有向图, 其中节点表示语句, 边表示对应节点(即语句)之间的数据依赖或控制依赖关系。获得程序依赖图之后, 再根据 5.1 节提出的切片规则对其进行深度优先搜索, 并将遍历到的语句依次加入切片集合。

如图 3 所示为基于图可达性的程序切片过程示例。该方法先遍历整个程序, 并根据定义的切片规则收集切片语句, 如图 3 左边程序所示, 第 05 条语句为经过遍历程序得到的一条切片规则语句。得到切片语句之后再根据定义 1 与定义 2 对整个程序进行分析, 得到如图 3 中间所示的程序依赖图, 其中虚线表示数据依赖, 实线表示控制依赖。然后, 以第 05 条语句为起始节点对程序依赖图进行深度优先搜索, 将遍历到的语句依次加入切片程序集合得到如图 3 右边所示的切片结果。

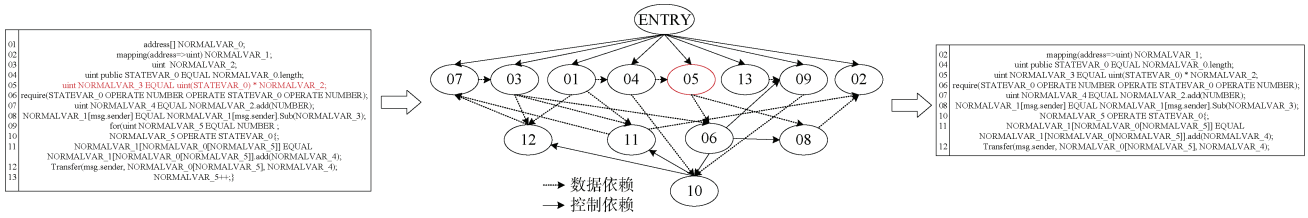


图 3 基于图可达性的程序切片

Figure 3 Program slicing based on graph accessibility

5.3 基于数据流方程的程序切片

在介绍数据流方程切片之前, 给出相关定义:

定义 3(活跃变量): 对于变量 x 和程序点 p, 如果在程序流程图中存在从 p 开始的某条路径引用了变量 x 在 p 点定义的值, 则称变量 x 在点 p 是活跃的(即 x 是活跃变量), 否则称变量 x 在点 p 不活跃。

定义 4(def 集): 程序基本块或语句中定义的所有变量集合。

定义 5(use 集): 程序基本块或语句中被使用的所有变量的集合。

定义 6(redef 集): 程序基本块或语句中失活变量集合。

基于数据流方程的程序切片主要通过对程序进行活跃变量分析得到每条语句执行前后的数据流值来进行切片。

程序基本块的数据流分析一般以多条语句集合为粒度, 而本方法以单条语句为粒度, 计算程序中每条语句 S 执行前后的数据流值, 将其分别记为 $In[S]$ 和 $Out[S]$, 公式(1)和公式(2)分别为 $In[S]$和 $Out[S]$的计算公式, 其中公式(1)中语句 P 为语句 S 在程序控制流程图中的所有前驱。

$$In[S]=\cup_{p\text{为s的前驱}} Out[P] \tag{1}$$

$$Out[S]=f_s(In[S]) \tag{2}$$

其中 f_s 为转换函数, 其定义如公式(3)所示。

$$Out[S]=use_s \cup def_s \cup (In[S]\text{-}redef_s) \tag{3}$$

基于数据流方程的切片如算法 1 和算法 2 所示。

算法 1. 数据流分析

输入: 每条语句的 *use* 集、*def* 集、*redef* 集
程序控制流程图 $S=\{S_1, S_2, \ldots\}$
程序控制流程图节点数量 *Length*

输出: 每条语句的 *Out* 集和 *In* 集

1.$In[ENTRY]=\varnothing$
2.*changeFlag*=true
3.FOR $s_i \in S$:
4. $In[s_i]=\varnothing$
5.END
6.WHILE(*chanegFlag*):
7. *COUNT=0*
8. FOR $s_i \in S$:
9. $inTemp= In[s_i]$
10. $In[s_i]=\cup In[s_{i-1}]$
11. $Out[s_i]=use_{si} \cup def_{si} \cup (In[s_i]\text{-}redef_{si})$

```
12.     IF inTemp== In[s_i]
13.         COUNT++
14.     END
15.   END
16.   IF COUNT ==Length
17.     changeFlag=false
18.   END
19.END
```

(注: *ENTRY* 表示程序入口; s_{i-1} 表示语句 s 在程序控制流程图中的所有前驱)

算法 2. 基于数据流方程的程序切片

输入: 程序语句集合 $S=\{S_1, S_2, \ldots\}$
切片规则每条语句的 *def* 集、*use* 集
$Out=\{V_1, V_2, \ldots\}$
输出: 切片程序集合 *sliceProgram*

```
1. sliceProgram=∅
2.FOR S_i∈S
3.   FOR v_i∈Out_i
4.     IF v_i∈def || v_i∈use
5.       sliceProgram=sliceProgram∪{S_i}
6.       BREAK
7.     END
8.   END
9.END
```

算法 1 的功能是计算程序中每条语句执行前后的 *In/Out* 集。算法的输入包括程序每条语句的 *use*、*def*、*redef* 集, 程序控制流程图及图中节点的数量。

算法 1 中第 1 行至第 5 行初始化每条语句 *In* 集, 变量 *changeFlag* 用于标记语句在数据流分析前后的 *In* 集是否发生变化。获得当前语句的 *In* 集后, 算法 1 再通过公式(1)和公式(2)计算当前语句经过数据流分析之后的 *In* 集与 *Out* 集, 该过程对应第 9 行到第 11 行的代码。第 12 行到 14 行代码判断当前语句分析前后 *In* 集是否发生变化, 若没有发生变化则将计数器 *COUNT* 累加; 第 16 到第 18 行代码判断经过一轮数据流分析后是否存在某条语句的 *In* 集发生变化, 如果有则跳出 WHILE 循环结束整个算法。

算法 2 的输入为算法 1 计算的每条语句的 *In* 集和 *Out* 集及切片规则所有语句对应的 *def* 集与 *use* 集, 伪代码中的第 2 行到第 6 行通过遍历每条语句的 *Out* 集中的每个变量, 如果该变量存在于切片规则对应的 *use* 集、*def* 集中则将当前语句加入切片结果集合。

如图 4 所示为图 1 所示程序经过算法 1 数据流分析每条语句对应的 *In* 集和 *Out* 集, 与此同时通过算法 2 即可得到如图 5 所示的数据流方程切片结果。

	def	use	redef	IN	OUT
01	NORMALVAR_0	ø	NORMALVAR_0	ø	NORMALVAR0
02	NORMALVAR_1	ø	ø	NORMALVAR_0	NORMALVAR_1, NORMALVAR_0
03	NORMALVAR_2	ø	ø	NORMALVAR_1, NORMALVAR_0	NORMALVAR_2,NORMALVAR_1 NORMALVAR_0
04	STATEVAR_0	ø	ø	NORMALVAR2,NORMALVAR_1 NORMALVAR_0	NORMALVAR_0,STATEVAR_0 NORMALVAR_2, NORMALVAR1
05	NORMALVAR_3	ø	ø	STATEVAR_0,NORMALVAR_2 NORMALVAR_3,NORMALVAR_0 NORMALVAR_1	STATEVAR0,NORMALVAR_2 NORMALVAR_3,NORMALVAR_0 NORMALVAR_1
06	ø	STATEVAR_0	ø	STATEVAR_0,NORMALVAR_2 NORMALVAR_3,NORMALVAR_0 NORMALVAR_1	STATEVAR_0,NORMALVAR_2 NORMALVAR_3,NORMALVAR_0 NORMALVAR_1
07	ø	NORMALVAR_4 NORMALVAR_2	ø	STATEVAR_0,NORMALVAR_2 NORMALVAR_3,NORMALVAR_0 NORMALVAR_1	NORMALVAR_3,NORMALVAR_4 STATEVAR_0,NORMALVAR_0 NORMALVAR_1
08	ø	NORMALVAR_1 NORMALVAR_3	ø	NORMALVAR_3,NORMALVAR_4 STATEVAR_0,NORMALVAR_0 NORMALVAR_1	NORMALVAR_1,NORMALVAR_3 NORMALVAR_4,STATEVAR_0 NORMALVAR_0
09	NORMALVAR_5	NORMALVAR_5	ø	NORMALVAR_1,NORMALVAR_3 NORMALVAR_4,STATEVAR_0 NORMALVAR_0	NORMALVAR_5,NORMALVAR_1 NORMALVAR_3,NORMALVAR_4 STATEVAR_0, NORMALVAR_0
10	ø	NORMALVAR_5 STATEVAR_0	ø	NORMALVAR_5,NORMALVAR_1 NORMALVAR_3,NORMALVAR_4 STATEVAR_0, NORMALVAR_0	[NORMALVAR_5,STATEVAR_0 NORMALVAR_1,NORMALVAR_3 NORMALVAR_4, NORMALVAR_0
11	ø	NORMALVAR_0 NORMALVAR_1 NORMALVAR_5 NORMALVAR_4	ø	NORMALVAR_5,STATEVAR_0 NORMALVAR_1,NORMALVAR_3 NORMALVAR_4, NORMALVAR_0	NORMALVAR_1,NORMALVAR_0 NORMALVAR_5,NORMALVAR_4 STATEVAR_0, NORMALVAR_3
12	ø	NORMALVAR_0 NORMALVAR_5 NORMALVAR_4	ø	NORMALVAR_1,NORMALVAR_0 NORMALVAR_5,NORMALVAR_4 STATEVAR_0, NORMALVAR_3	NORMALVAR_5,NORMALVAR_4 NORMALVAR0,NORMALVAR_1 STATEVAR_0, NORMALVAR_3
13	ø	NORMALVAR_5	ø	NORMALVAR_5,NORMALVAR_4 NORMALVAR_0,NORMALVAR_1 STATEVAR_0, NORMALVAR_3	NORMALVAR_4,NORMALVAR_0 NORMALVAR_1,STATEVAR_0 NORMALVAR_3

图 4 数据流分析结果

Figure 4 Data flow analysis results

```
03    uint  NORMALVAR_2;
04    uint public STATEVAR_0 EQUAL NORMALVAR_0.length;
05    uint NORMALVAR_3 EQUAL uint(STATEVAR_0) * NORMALVAR_2;
06  require(STATEVAR_0 OPERATE NUMBER OPERATE STATEVAR_0 OPERATE NUMBER);
07    uint NORMALVAR_4 EQUAL NORMALVAR_2.add(NUMBER);
08  NORMALVAR_1[msg.sender] EQUAL NORMALVAR_1[msg.sender].Sub(NORMALVAR_3);
10    NORMALVAR_5 OPERATE STATEVAR_0{;
```

图 5　基于数据流方程程序切片结果示例

Figure 5　Example of program slicing results based on data flow equations

5.4　语义特征提取

通过基于图可达性、数据流方程的程序切片方法获得程序切片结果之后将这两部分结果拼接并删除重复的语句得到如图 6 所示最终的程序切片结果。本文使用 TF-IDF(Term Frequency-nverse Document Frequency)将代码转换为向量表示, 由于转换后的向量维度较高且过于稀疏, 因此本文先采用 PCA (Principal component analysis)技术将其降至 128 维, 再将降维后的向量输入 LSTM 中经过训练获取程序语义特征, 如图 7 所示为提取程序语义特征的过程, 其中左边为所有训练集的程序列表, 中间为经过 TF-IDF 获得的所有样本的向量表示的矩阵, 右边为对中间矩阵使用 PCA 技术进行降维后的向量矩阵。

```
02    mapping(address=>uint) NORMALVAR_1;
03    uint  NORMALVAR_2;
04    uint public STATEVAR_0 EQUAL NORMALVAR_0.length;
05    uint NORMALVAR_3 EQUAL uint(STATEVAR_0) * NORMALVAR_2;
06  require(STATEVAR_0 OPERATE NUMBER OPERATE STATEVAR_0 OPERATE NUMBER);
07    uint NORMALVAR_4 EQUAL NORMALVAR_2.add(NUMBER);
08  NORMALVAR_1[msg.sender] EQUAL NORMALVAR_1[msg.sender].Sub(NORMALVAR_3);
10    NORMALVAR_5 OPERATE STATEVAR_0{;
11    NORMALVAR_1[NORMALVAR_0[NORMALVAR_5]] EQUAL
        NORMALVAR_1[NORMALVAR_0[NORMALVAR_5]].add(NORMALVAR_4);
12    Transfer(msg.sender, NORMALVAR_0[NORMALVAR_5], NORMALVAR_4);
```

图 6　最终切片结果

Figure 6　Final slice results

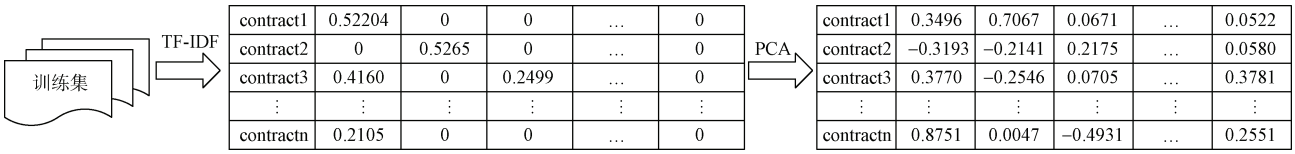


图 7　程序语义特征提取

Figure 7　Program semantic feature extraction

由于程序之间存在着控制依赖关系, 而 LSTM 能够学习到程序语句之间的控制依赖关系, 能最大程度地保留源程序的语义特征。

LSTM 利用遗忘门、输入门和输出门三个门控机制, 有效的传递了长时间序列的信息, 保证长时间前的关键信息不会被遗忘, 因此能够保存和传输大范围内的上下文信息。门控机制的计算方法如公式(4)~(9)所示。

$$f_t = \sigma(W_f \bullet [h_{t-1}, x_t] + b_f) \tag{4}$$

$$I_t = \sigma(W_I \bullet [h_{t-1}, x_t] + b_I) \tag{5}$$

$$O_t = \sigma(W_o \bullet [h_{t-1}, x_t] + b_o) \tag{6}$$

$$c = \tanh(W_c \bullet [h_{t-1}, x_t] + b_c) \tag{7}$$

$$C_t = f_t \times C_{t-1} + I_t \times c \tag{8}$$

$$h_t = \tanh(C_t) \times O_t \tag{9}$$

在上述公式中,x 和 h 分别表示输入和输出特征; W 和 b 分别表示权重矩阵和偏置项; c 表示存储单元; $tanh$ 为双曲正切激活函数; $\sigma(\bullet)$ 为非线性激活函数; f_t、I_t 和 O_t 分别表示遗忘门、输入门及输出门。

6　程序结构特征提取

Slice-GCN 使用图卷积神经网络来提取程序的结构特征, 并将图卷积神经网络模型的构建分为基于节点构造和边构造的程序依赖图(PDG)构造(简称图构造)、图卷积神经网络(GCN)、注意力边收缩三个阶段。

由于程序依赖图能够更加充分地反映程序的结构信息以及执行依赖关系, 于是, 本文基于程序依赖图提取结构特征。再基于程序切片的结果简化程序依赖图, 并重点关注漏洞语句之间的依赖关系。

6.1　图构造

如果程序比较复杂, 则会导致其程序依赖图也比较庞大, 因此先通过节点消除来简化程序依赖图以提高效率。在简化程序依赖图之前, 将程序依赖图中的节点分为回调节点、主节点和其他节点, 其定义如下。

定义 7(回调节点): 如果某个语句为外部函数调用语句, 则将该节点记为回调节点。

定义 8(主节点): 漏洞相关节点, 本文结合 5.1 和 5.2 节程序切片的结果, 将切片结果中的所有语句都记为主节点。

定义 9(其他节点): 除回调节点和主节点之外的节点都记为其他节点。

将节点进行分类之后, 将每个节点转换为向量表示, 再删除其他节点(定义 9), 并将其特征融合到与之相邻的主节点或回调节点的方法来简化程序依赖图。

如图 8 所示为示例程序图构造及节点消除过程, 左边程序中语句 02、04-08、10-12 为程序切片的结果语句, 中间为程序对应的程序依赖图, 其中节点 02、04-08、10-12 为主节点, 再将程序依赖图中每个节点转换为向量表示, 为了简化程序依赖图提高模型的训练速度, 本文通过将其他节点与回调节点(节点 01、03、09、13)的特征与主节点融合得到右边简化之后的程序依赖图。

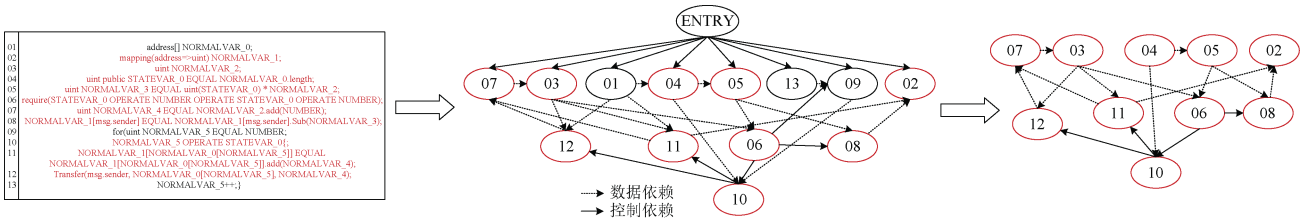


图 8　节点消除示例图

Figure 8　Sample diagram of node elimination

6.2　图卷积神经网络

普通的序列模型较难提取程序之间的结构依赖特征, 而基于图卷积的方式能够从程序依赖图中提取较为充分的程序结构特征。图卷积神经网络在有向图或无向图中使用卷积计算, 其每层之间的传播计算公式如公式(10)所示。

$$\boldsymbol{X}^{(l+1)}=\sigma(\hat{D}^{-\frac{1}{2}}\hat{\boldsymbol{A}}\hat{D}^{-\frac{1}{2}}\boldsymbol{X}_l\boldsymbol{W}_l) \tag{10}$$

其中, $\boldsymbol{A}$ 表示图的邻接矩阵; $\hat{\boldsymbol{A}}=\boldsymbol{A}+\boldsymbol{I}$, $\boldsymbol{I}$ 为单位矩阵, $\tilde{\boldsymbol{A}}$ 为邻接矩阵加入自循环后的矩阵, 用于将所有邻居节点的信息和自身的信息聚合; $\boldsymbol{X}_l$ 为特征矩阵, $\boldsymbol{W}_l$ 为第 l 层的权重矩阵; $\sigma(\bullet)$ 为非线性激活函数。

与对节点进行分类不同, Slice-GCN 方法是对整个图进行分类, 因此需要由所有节点得到整个图的表示, 本文提出通过基于注意力的边收缩方法来获取图的全局特征。

6.3　基于注意力的边收缩

节点消除之后, 将图中每个节点转换为向量表示后得到节点的特征, 再将所有节点的特征进行融合得到整个图的特征表示。常见的融合方法就是直接将所有的节点特征进行求和或者求平均值, 但这类方法会忽略每个节点的重要程度。为了更加充分的提取图的结构特征, 本文提出基于注意力的边收缩方法逐步融合所有节点的特征, 从而得到整个图的特征表示。

基于注意力的边收缩方法主要是通过归并操作来逐步学习图的全局特征, 不断地对每条边上的节点进行两两归并形成一个新的节点, 同时保留合并前两个节点的连接关系到新节点上。图边收缩的过程如图 9 所示, 其中边上的数字即为该条边的注意力分数。由于每个节点可能有多条边, 而边收缩只能选择一条边进行收缩。本文通过计算每条边的注意力分数来选择要收缩的边, 注意力分数计算公式如公式(11)所示。

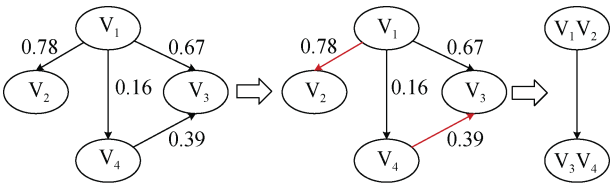


图 9　基于注意力的边收缩

Figure 9　Attention based edge contraction

$$Z=\sigma(\hat{D}^{-\frac{1}{2}}\hat{\boldsymbol{A}}\hat{D}^{-\frac{1}{2}}\boldsymbol{X}\theta_{att}) \tag{11}$$

其中 $\boldsymbol{X}$ 为输入的特征矩阵, θ_{att} 为可训练参数。

在计算完每条边的注意力分数之后, 根据注意力分数的大小选择注意力分数较大的边进行收缩, 再根据超参数 K 决定每次保留边数的比例, 最后获得需要保留的边的索引。

$$idx=TopRank(Z,\lceil KN\rceil) \tag{12}$$

如公式(12)所示, TopRank 函数根据每条边的注意力分数 Z 和收缩比例 K、图的边数 N 返回索引位置。

6.4　特征融合

假设有两个层次的特征 $x\in\mathbb{R}^n, y\in\mathbb{R}^m$, 可以将它们融合后得到的新特征 $z\in\mathbb{R}^k$。目前常用的特征融合方法包括特征拼接、特征求和及对应元素相乘等方法, 这些方法的计算公式如公式(13)~(15)所示。

$$z=[x,y]\in\mathbb{R}^{m+n}, k=m+n \tag{13}$$

$$z=x+y\in\mathbb{R}^k \tag{14}$$

$$z=x\bullet y\in\mathbb{R}^k \tag{15}$$

其中, 特征求和与对应元素相乘要求融合前的两个

特征维度相同, 而特征拼接允许两个特征维度不一致。另外, 特征求和与对应元素相乘会丧失原来两个特征的个性。于是, 本文使用特征拼接方法融合程序语义特征 F_s 和结构特征 F_g, 然后将它们输入多层感知机(MLP)得到预测结果, 如图 10 所示。图 10 计算过程如公式(16)和(17)所示。

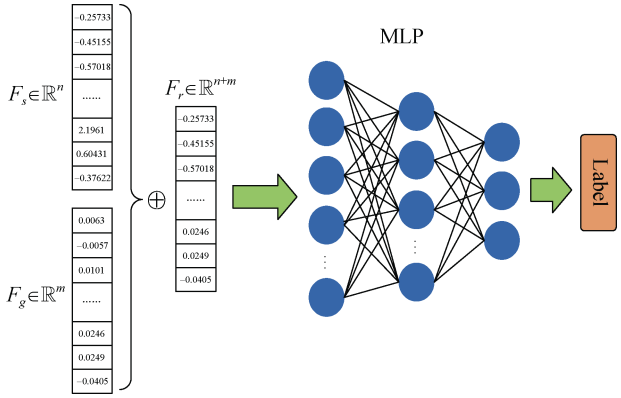


图 10　特征融合与漏洞检测

Figure 10　Feature fusion and vulnerability detection

$$F_r = F_s \oplus F_g \tag{16}$$

$$\hat{y} = FC(F_r) \tag{17}$$

其中 F_s 即为公式(13)中的 x, F_g 为其中的 y, $\oplus$ 表示拼接操作, FC 为多层感知机, 标签 Label 即为预测值 $\hat{y}$。其中多层感知机为三层结构, 第一层为输入层, 该层包含 32 个神经元, 即 F_r 的维度; 第二层为隐藏层, 包含 16 个神经元; 最后一层为输出层, 包含 3 个神经元, 即类别个数。

7　实验评估

7.1　数据集

本文数据集来源于以下两个部分:

1) 开源数据集: ContractWard[12]数据集中的数据来源于真实世界发生过交易的智能合约; SmartBug 数据集由一部分真实世界具有漏洞的智能合约和一部分人工构造的漏洞合约组成。为了保证实验的准确性, 本文整合了两个数据集, 并删除了重复数据。

2) 真实世界数据集: 除了开源数据集之外, 本文还从 Etherscan 网站上收集了部分数据, 根据 Durieux 等人[13]的研究表明, Slither[14]兼具准确性和效率, 因此本文选择 Slither 对合约进行标注。

最后通过删除重复或相似度较高的数据得到 964 条数据, 并以 6:4 划分训练集与测试集设计对比实验。实验环境为: Intel(R) Xeon(R) Platinum 8269CY CPU, 内存 8GB, 系统 Centos8.2。

本文漏洞检测分为两个模块, 第一个模块为数据处理模块, 主要功能包括预处理智能合约代码(代码的规范化处理)、提取程序中间表示(数据依赖图、控制依赖图等)、程序切片(数据流切片以及图可达性切片)等; 第二个模块为预测模块, 主要功能为程序语义特征提取、程序结构特征提取、特征融合以及结果预测。

7.2　评价标准

实验采用准确率、精确率、召回率和 $F1$ 值来衡量本文方法效果和对比实验结果分析。每个指标的计算方法为:

1) 准确率(*Accuracy*): 表示准确分类的样本数量占总样本数量的比例, 计算方法如公式(18)所示。

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{18}$$

2) 精确率(*Precision*): 又称为查准率, 衡量模型对预测的正样本的准确程度。精确率越高, 说明在被预测为正的样本中, 真实标签也为正的概率越大, 计算方法如公式(19)所示。

$$Precision = \frac{TP}{TP + FP} \tag{19}$$

3) 召回率(*Recall*): 又称为查全率, 能够衡量模型捞出正样本的能力, 召回率越高, 说明真实标签为正的样本被预测为正的概率越大, 其计算方法如公式(20)所示。

$$Recall = \frac{TP}{TP + FN} \tag{20}$$

4) $F1$ 值: 表示精确率和召回率的调和均值, 反映了模型的整体表现效果, 其计算方法如公式(21)所示。

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recal} \tag{21}$$

在上述 4 个评价指标中, *TP*(*True Positive*)表示漏洞样本检测为漏洞样本的数量, *TN*(*True Negative*)表示无漏洞样本检测为无漏洞样本的数量, *FP*(*False Positive*)表示漏洞样本检测为无漏洞样本的数量, *FN*(*False Negative*)表示无漏洞样本检测为漏洞样本的数量。

7.3　实验结果分析

为了全面分析 Slice-GCN 在智能合约缺陷检测上的性能, 设置了以下 4 个研究问题。

研究问题 1: Slice-GCN 方法与工具 Oyente、Osiris 和 Solidity check 相比, 在准确率、精确率、召回率及 $F1$ 值方面, 孰优孰劣?

研究问题 2: Slice-GCN 方法与现有漏洞检测工具 Oyente、Osiris 和 Solidity check 的检测速度相比, 孰优孰劣?

研究问题 3: Slice-GCN 方法中所提出的程序切片与图神经网络对方法的性能有何影响?

研究问题 4: 在本文提出的基于注意力的边收缩方法中, 保留比例 K 值对 Slice-GCN 方法的性能有何影响?

7.3.1 对研究问题 1 的分析

本文选取 Osiris[15]、Solidity check[16]及 Oyente[17]三款智能合约漏洞检测工具进行对比实验。

Oyente: 是基于符号执行的智能合约漏洞检测工具, 该工具将整个系统分为 CFGBuilder、Explorer、CoreAnalysis 及 Validation 4 个模块, CFGBuilder 模块负责将操作码转换为程序控制流程图; Explorer 模块通过符号输入并依据程序控制流程图模拟执行程序, 与此同时收集每条路径的符号路径; CoreAnalysis 模块通过约束求解器求解符号路径得到检测结果; 最后, Validation 模块能够在一定程度上消除误报。

Osiris: 基于符号执行的漏洞检测框架, 该工具由符号分析模块、污点分析模块及漏洞检测模块 3 个模块组成。其中, 符号分析模块首先构造控制流程图, 然后符号执行智能合约的不同路径, 并将每一条指令的执行结果反馈给其他 2 个模块; 污点分析模块负责引入、传播并检查内存及存储中的污点数据; 漏洞检测模块检查执行的指令中是否存在漏洞。

Solidity check: 基于正则表达式的智能合约漏洞检测工具, 该工具通过总结归纳每种漏洞类型的漏洞模式预定义对应的正则表达式, 其次遍历待检测的程序检测是否存在与预定义正则表达式匹配的语句, 如果存在则说明存在漏洞。

对比实验结果如表 1 所示。从表 1 中可以看出, 三款工具(传统的非深度学习的智能合约漏洞检测方法)在对重入攻击漏洞检测时的准确率和召回率较低, 导致误报率较高; 与此同时, Slice-GCN 方法在对重入攻击漏洞的检测时准确率为 93.29%, 比传统方法中表现最好的 Solidity check 还高出 20.08%。

表 1 Slice-GCN 方法与传统检测工具对比结果

Table 1 Comparison results between Slice-GCN and traditional detection tools (%)

检测方法	重入攻击				时间戳依赖				整数溢出			
	准确率	精确率	召回率	*F*1 值	准确率	精确率	召回率	*F*1 值	准确率	精确率	召回率	*F*1 值
Oyente	41.07	15.97	41.07	22.98	20.73	66.66	20.73	31.61	74.28	54.37	74.28	63.01
Osiris	75.00	27.81	75.00	40.57	23.17	53.52	23.17	32.33	50.00	51.47	50.00	50.72
Solidity check	73.21	82.00	73.21	77.37	**98.78**	93.71	**98.78**	**96.17**	78.57	84.61	78.57	81.47
Slice-GCN	**93.29**	**91.07**	**91.07**	**91.07**	91.07	**97.45**	93.29	95.33	**94.28**	**85.71**	**94.29**	**89.80**

在对时间戳依赖漏洞的检测中, Oyente 和 Osiris 的准确率及召回率都比较低, 因为这两个工具只能检测与转账相关的时间戳依赖漏洞, 而 Solidity check 的表现效果明显优于上述两种方法, 或许是因为 Solidity check 定义了较多的正则表达式来检测漏洞; 而 Slice-GCN 方法在四种指标上都具有较高的值, 且精确率最高。

Oyente 和 Solidity check 对整数溢出漏洞的检测效果相比 Osiris 要更好, 准确率分别为 74.28%和 78.57%, 而 Slice-GCN 方法对整数溢出漏洞的检测准确率达到了 94.28%, 比表现最好的工具 Solidity check 还高出 15.71%。

7.3.2 对研究问题 2 的分析

表 2 展示了 Slice-GCN 方法与 Oyente、Osiris 及 Solidity check 3 种工具平均检测每份合约所用的时间对比结果。

如表 2 所示, Slice-GCN 方法平均检测每个合约的时间为 2.485 s, 而 Oyente、Osiris 及 Solidity check 平均检测每个合约消耗的时间分别为 4.044 s、13.793 s 和 7.962 s。可以看出, Slice-GCN 方法检测智能合约漏洞的速度要明显快于其他 3 款检测工具。

表 2 Slice-GCN 方法与传统检测工具时间对比结果

Table 2 Time comparison results between Slice-GCN and traditional detection tools

	Slice-GCN	Oyente	Osiris	Solidity check
时间(s)	2.485	4.044	13.793	7.962

7.3.3 对研究问题 3 的分析

本文主要采用了程序切片和图神经网络方法进行漏洞检测, 为了评估这 2 种方法的结合优势, 本文设计了对比实验: 1)只采用程序切片, 而不采用图神经网络(Slice-only); 2)只采用图神经网络, 而不采用程序切片(GCN-only); 3)Slice-GCN 方法。表 3 展示了上述 3 种方法在本文数据集上的表现效果。

表 3 Slice-GCN 方法与 Slice-only、GCN-only 对比结果

Table 3 Comparison results between Slice-GCN and Slice-only、GCN-only (%)

检测方法	重入攻击				时间戳依赖				整数溢出			
	准确率	精确率	召回率	*F*1 值	准确率	精确率	召回率	*F*1 值	准确率	精确率	召回率	*F*1 值
Slice-only	**93.90**	65.85	48.21	55.67	48.21	78.97	**93.90**	85.79	38.57	50.00	38.57	43.55
GCN-only	93.29	**95.92**	83.93	89.52	83.92	88.44	93.29	90.80	72.85	75.00	72.86	73.91
Slice-GCN	93.29	91.07	**91.07**	**91.07**	**91.07**	**97.45**	93.29	**95.33**	**94.28**	**85.71**	**94.29**	**89.80**

根据表 3 中的实验数据, Slice-GCN 方法对重入攻击漏洞、时间戳依赖和整数溢出漏洞检测的 *F*1 值高于 Slice-only 与 GCN-only, 即 Slice-GCN 的整体效果(*F*1 值)优于 Slice-only 与 GCN-only, 表明在图神经网络中加入程序语义特征能够有效提高模型性能, 且 Slice-GCN 方法能够有效的提取程序语义和结构特征, 降低噪声(即与漏洞的无关代码)对模型的影响。

7.3.4 对研究问题 4 的分析

如图 11 给出了不同保留比例 *K* 对实验结果的影响。

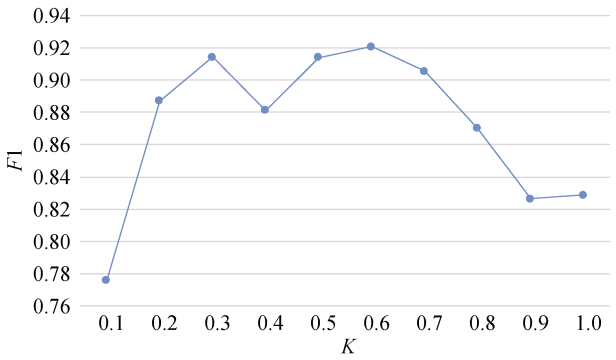


图 11 不同 *K* 值对实验结果的影响

Figure 11 Effect of different *K* values on experimental results

可以看出, 随着 *K* 值的增大 *F*1 值总体呈上升趋势, *K* 值到达 0.6 时 *F*1 值达到最大, 之后随着 *K* 值的增大, *F*1 值会慢慢下降。其可能的原因是, *K* 值过小时, 图神经网络模型携带的信息较少, 使得网络效果不佳; *K* 值过大时, 网络携带的初始信息过多, 导致模型无法学习到深层网络信息, 从而降低了模型的表达能力。当保留比例为 1.0 时, 表示保留所有节点的信息(即不采用边收缩机制)。由图 11 可以看出采用边收缩机制(*K* 值在 0.2～0.8 之间时)能够提高模型的效果。

7.4 案例研究

本文选择如程序 5 所示 Serpent 合约作为案例探讨本文方法的可理解性。该合约为本文数据集之外的合约, 使用本文方法发现该合约存在时间戳依赖漏洞。

```
contract Serpent {
  mapping(address=> uint256) public investorReturn;
  uint256 public SerpentCountDown;
  function CollectReturns () external {
    uint256 currentTime=uint256(block.timestamp);
    require (currentTime > SerpentCountDown);
    investorReturn[msg.sender] = 0;
  }
}
```

程序 5 Serpent 合约

Program 5 Serpent contract

经过人工审计该合约的确存在时间戳依赖漏洞, 其中第 5 行代码将当前时间戳 *block.timestamp* 赋值给 *currentTime*, 第 6 行间接使用当前时间戳作为 require 语句的判断条件, 根据时间戳依赖漏洞的特点, 攻击者能够在一定范围内篡改时间戳使得条件成立, 进而对合约进行攻击。

本文方法之所以检测时间戳依赖漏洞关键在于能发现 *block.timestamp*、*now* 等与漏洞相关的特征, 并提取与其相关语句的语义与结构特征。本文方法在一定程度上符合人工审计的习惯, 具有一定的可解释性。但是, 在具体的算法(比如 LSTM、GCN)细节上, 可解释性较差问题依然存在。

7.5 对本文实验效果影响因素分析

1) 内部因素: 由于当前智能合约漏洞检测没有一个统一且完善的数据集, 因此本文构建了自己的数据集。本文数据集的标签工作是通过 Slither 这款工具检测得来的, 虽然该工具具有较高的检测准确率和精度, 但仍无法保证能准确检测每份合约, 因此在检测其他数据集时与本文实验中的效果可能会存在偏差。

2) 外部因素: 智能合约是一个更新较快的程序语言, 每次发布新的版本时可能会增加、删除或修改之前版本的部分功能, 而本文方法研究的智能合约版本为 0.4.*X*, 这也是大多数智能合约漏洞检测文献使用的版本。因此, 使用本文方法检测由更高版本编写的智能合约, 效果可能会有所下降, 甚至无法正

常运行。

8 相关工作

已有的智能合约自动化检测方法大致可以分为三类: 基于符号执行和模糊测试的方法、基于形式化验证的方法及基于机器学习或深度学习的方法。

符号执行、污点分析及模糊测试等技术主要通过获取测试用例的执行信息来检测漏洞。例如, Oyente[17]就是基于动态符号执行的智能合约漏洞检测工具, 之后的 Osiris[15]及 Maian[18]也是在 Oyente 的基础上进行改进开发的漏洞检测工具。除了使用符号执行外, EASYFLOW[19]通过污点分析技术对智能合约二进制代码进行分析, 它通过污点标记跟踪交易的整个过程, 并通过生成与原始交易具有相同的输入数据的交易去触发潜在的溢出漏洞。Slither[14]通过将源代码转化成抽象语法树生成继承图和控制流程图, 并将源代码转换为 SlithIR 中间表示, 最后使用 SlithIR 中间表示进行漏洞分析。

符号执行和模糊测试等漏洞检测技术通常需要实际或模拟程序的运行来获取程序的执行信息, 因此随着程序规模的扩大, 该类方法的检测效率也会降低。

形式化验证已经被广泛应用于程序的漏洞检测和验证, 也是智能合约安全验证的重要技术之一。其主要通过形式化语言, 将程序中的概念、判断和推理转化成形式化模型, 从而消除程序中的歧义性和不通用性, 与此同时配合严谨的逻辑和证明,验证智能合约中函数功能的正确性和安全性。ZEUS[20]是一基于形式化验证的智能合约静态分析工具, 该工具利用抽象解释和模型检查来快速验证合约的安全性, 支持重入漏洞、整数溢出漏洞、异常处理漏洞等漏洞类型的检测。杨慧文等人[21]提出一组针对智能合约特有的变量、函数、结构及语言特性的度量元集 SC-Sol(Smart Contract-Solidity), 实验结果表明, 结合 COOP(Code complexity and features of object-oriented program)和 SC-Sol 的 COOP-SC-Sol 度量元集具有较好的缺陷预测性能。赵颖琪等人[22]通过对智能合约时间约束的不同表现形式进行梳理, 提取相应的时间约束模式并对其进行形式化, 再定义智能合约到时间自动机的转换规则, 并实现规则到实时模型检测工具 UPPAAL 入口模型的自动转换, 最后利用 UPPAAL 验证合约的时间相关性质。

形式化验证往往需要大量的公式推导和证明过程, 因此要求需要掌握专业的数学知识和证明原理, 导致基于该种方法的检测工具发展缓慢。同时由于复杂的推导和证明过程, 造成其适用范围窄且自动化程度低。

随着机器和深度学习的迅速发展, 已有一些研究将深度学习应用于漏洞检测领域。Peng 等人[23]提出了使用包含注意力机制的双向长短期记忆模型(BLSTM-ATT)合约的漏洞检测, 为了简化输入该方法使用程序切片技术提取重入漏洞的相关代码。但该方法只是针对重入漏洞的检测方案。为了对更多类型的漏洞进行检测, Oliver Lutz 等人[24]提出了可扩展性的深度学习框架, 只需对深度神经网络模型架构进行较少的修改便可支持对新型漏洞的迁移学习。Yuan Zhuang[25]提出从源代码的数据流和控制流中提取主节点、次节点及回馈节点, 与此同时构造各个节点的边来构建图神经网络的输入, 并提出了一种扩展的图神经网络用以解决图的归一化问题, 从而充分地发挥各个节点的作用。相比于上述深度学习方法, 本文通过结合图神经网络和序列模型来构建模型, 更大程度的保留了源程序的语义特征和结构特征, 具有较高的检测准确率和精度。

9 结论

区块链上具有漏洞的智能合约可能会导致用户的资产损失, 因此智能合约的漏洞检测是一项重要工作。基于深度学习的智能合约漏洞检测工作目前主要存在着数据噪声过多、模型无法充分挖掘程序的结构特征等问题。于是, 本文提出了一种基于程序切片和图神经网络的智能合约漏洞检测方法 Slice-GCN, 该方法使用程序切片技术来降低噪声的影响, 并使用图神经网络来提取程序的结构特征, 进一步提升了漏洞检测的能力。实验表明, 本文方法能够有效提升检测准确度和精度, 降低误报率和漏报率。在未来的研究工作中, 将研究更多类型漏洞具体特点, 对这些漏洞更具针对性地进行语义信息提取, 以达到更细致分类的目的。

参考文献

[1] Ni Y D, Zhang C, Yin T T. A Survey of Smart Contract Vulnerability Research[J]. *Journal of Cyber Security*, 2020, 5(3): 78-99.
(倪远东, 张超, 殷婷婷. 智能合约安全漏洞研究综述[J]. *信息安全学报*, 2020, 5(3): 78-99.)

[2] Sankar L S, Sindhu M, Sethumadhavan M. Survey of Consensus Protocols on Blockchain Applications[C]. *2017 4th International Conference on Advanced Computing and Communication Systems*, 2017: 1-5.

[3] Fu M L, Wu L F, Hong Z, et al. Research on Vulnerability Mining Technique for Smart Contracts[J]. *Journal of Computer Applications*, 2019, 39(7): 1959-1966.

(付梦琳, 吴礼发, 洪征, 等. 智能合约安全漏洞挖掘技术研究[J]. *计算机应用*, 2019, 39(7): 1959-1966.)

[4] Krupp J, Rossow C. teether: Gnawing at ethereum to automati-cally exploit smart contracts[C]. *27th Security Sympo-sium,* 2018: 1317-1333.

[5] Rodler M, Li W T, Karame G O, et al. Sereum: Protecting Existing Smart Contracts Against re-Entrancy Attacks[EB/OL]. 2018: 1812.05934. https://arxiv.org/abs/1812.05934v1.

[6] Jiang B, Liu Y, Chan W K. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection[C]. *The 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018: 259-269.

[7] Liu C, Liu H, Cao Z, et al. ReGuard: Finding Reentrancy Bugs in Smart Contracts[C]. *2018 IEEE/ACM 40th International Conference on Software Engineering*: *Companion*, 2018: 65-68.

[8] Tsankov P, Dan A, Drachsler-Cohen D, et al. Securify: Practical Security Analysis of Smart Contracts[C]. *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 67-82.

[9] Park D, Zhang Y, Saxena M, et al. A Formal Verification Tool for Ethereum VM Bytecode[C]. *The 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018: 912-915.

[10] Tann W J W, Han X J, Gupta S S, et al. Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Security Threats[EB/OL]. 2018: 1811.06632. https://arxiv.org/abs/1811.06632v3.

[11] Liu Z G, Qian P, Wang X Y, et al. Combining Graph Neural Networks with Expert Knowledge for Smart Contract Vulnerability Detection[J]. *IEEE Transactions on Knowledge and Data Engineering*, 2023, 35(2): 1296-1310.

[12] Wang W, Song J J, Xu G Q, et al. ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts[J]. *IEEE Transactions on Network Science and Engineering*, 2021, 8(2): 1133-1144.

[13] Durieux T, Ferreira J F, Abreu R, et al. Empirical Review of Automated Analysis Tools on 47, 587 Ethereum Smart Contracts[C]. *The ACM/IEEE 42nd International Conference on Software Engineering*, 2020: 530-541.

[14] Feist J, Grieco G, Groce A. Slither: A Static Analysis Framework for Smart Contracts[C]. *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019: 8-15.

[15] Torres C F, Schütte J, State R. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts[C]. *The 34th Annual Computer Security Applications Conference*, 2018: 664-676.

[16] Xiao F, Zhang P C, Luo X P. Ethereum Smart Contract Bug Detection and Repair Approach Based on Regular Expressions, Program Instrumentation and Code Replacement[J]. *Computer Science*, 2021, 48(11): 89-101.
(肖锋, 张鹏程, 罗夏朴. 基于正则表达式、程序插桩和代码替换的以太坊智能合约 bug 检测和修复方法[J]. *计算机科学*, 2021, 48(11): 89-101.)

[17] Luu L, Chu D H, Olickel H, et al. Making Smart Contracts Smarter[C]. *The 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016: 254-269.

[18] Nikolić I, Kolluri A, Sergey I, et al. Finding the Greedy, Prodigal, and Suicidal Contracts at Scale[C]. *The 34th Annual Computer Security Applications Conference*, 2018: 653-663.

[19] Gao J B, Liu H, Liu C, et al. EASYFLOW: Keep Ethereum Away from Overflow[C]. *2019 IEEE/ACM 41st International Conference on Software Engineering*: *Companion Proceedings*, 2019: 23-26.

[20] Kalra S, Goel S, Dhawan M, et al. ZEUS: Analyzing Safety of Smart Contracts[C]. *Proceedings 2018 Network and Distributed System Security Symposium*, 2018: 1-12.

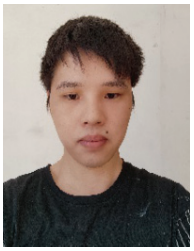
[21] Yang H W, Cui Z Q, Chen X, et al. Defect Prediction for Solidity Smart Contracts Based on Software Measurement[J]. *Journal of Software*, 2022, 33(5): 1587-1611.
(杨慧文, 崔展齐, 陈翔, 等. 基于软件度量的 Solidity 智能合约缺陷预测方法[J]. *软件学报*, 2022, 33(5): 1587-1611.)

[22] Zhao Y Q, Zhu X Y, Li G Y, et al. Time Constraint Patterns of Smart Contracts and Their Formal Verification[J]. *Journal of Software*, 2022, 33(8): 2875-2895.
(赵颖琪, 朱雪阳, 李广元, 等. 智能合约的时间约束模式及其形式化验证[J]. *软件学报*, 2022, 33(8): 2875-2895.)

[23] Qian P, Liu Z G, He Q M, et al. Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models[J]. *IEEE Access*, 2020, 8: 19685-19695.

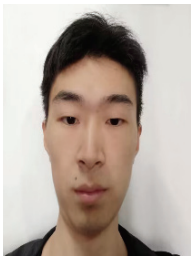
[24] Lutz O, Chen H L, Fereidooni H, et al. ESCORT: Ethereum Smart COntRacTs Vulnerability Detection Using Deep Neural Network and Transfer Learning[EB/OL]. 2021: 2103.12607. https://arxiv.org/abs/2103.12607v1.

[25] Zhuang Y, Liu Z, Qian P, et al. Smart Contract Vulnerability Detection using Graph Neural Network[C]. *IJCAI*, 2020: 3283-3290.



张人娄 于 2021 年在湖南应用技术学院物联网工程专业获得学士学位。现在江苏师范大学计算机科学与技术学 院电子信息专业攻读硕士学位。研究领域为软件安全分析。研究兴趣包括: 漏洞挖掘、漏洞定位。Email: zhangrenlou@jsnu.edu.cn

吴胜 于 2009 年在南京林业大学农林经济管理专业获得博士学位。现任江苏师范大学计算机科学与技术学院副教授。研究领域为软件工程。研究兴趣包括: 智能合约、会计信息化等。 Email: woodstone1978@jsnu.edu.cn



张浩 于 2021 年在徐州工程学院计算机科学与技术专业获得学士学位。现在江苏师范大学计算机科学与技术学院电子信息专业攻读硕士学位, 研究领域为区块链技术。研究兴趣包括: 代码生成、软件安全分析。Email: elder@jsnu.edu.cn

刘方宇 于 2021 年在重庆师范大学计算机科学与技术专业获得学士学位。现在江苏师范大学计算机科学与技术学院电子信息专业攻读硕士学位。研究领域为区块链。研究兴趣包括: 跨链协议、智能合约。Email: 1763272783@qq.com