# 基于符号执行的权限控制和转账漏洞检测方法

张艺琼，文伟平+，刘成杰

（北京大学 软件与微电子学院，北京 100091）

**摘 要**：针对权限控制漏洞和任意转账漏洞会让用户遭受经济损失的问题，提出一种基于符号执行的权限控制漏洞和任意转账漏洞检测方法。总结漏洞特征，设计漏洞检测算法，利用符号执行构建交易路径，通过漏洞特征构建约束条件并求解，实现工具 PTGuard。实验结果表明，PTGuard 的准确率、精确率、召回率和 F1 值相比其它漏洞检测工具有较大的提升；PTGuard 已应用于科技部国家重点项目，挖掘出 17 个漏洞，已成功上报到国家共享漏洞数据库，验证了 PTGuard 的实际应用价值。

**关键词**：符号执行；以太坊；智能合约；权限控制漏洞；任意转账漏洞；合约漏洞；漏洞检测工具

**中图法分类号**：TP309 **文献标识号**：A **文章编号**：1000-7024（2025）01-0081-07

**doi**：10.16208/j.issn1000-7024.2025.01.012

# Premission control and transfer vulnerability detection method based on symbolic execution

ZHANG Yi-qiong, WEN Wei-ping+, LIU Cheng-jie

(School of Software and Microelectronics, Peking University, Beijing 100091, China)

**Abstract**: In view of the problem that permission control vulnerabilities and arbitrary transfer vulnerabilities will cause users to suffer from economic losses, a symbolic execution-based detection method for permission control vulnerabilities and arbitrary transfer vulnerabilities was proposed. Vulnerability characteristics were summarized, vulnerability detection algorithms were designed, symbolic execution was used to construct transaction paths, constraint conditions were constructed and solved through vulnerability characteristics, and the tool PTGuard was implemented. Experimental results show that the accuracy, precision, recall and F1 value of PTGuard are greatly improved compared to that of other vulnerability detection tools. PTGuard has been used in national key projects of the Ministry of Science and Technology, and 17 vulnerabilities are discovered and successfully reported to the national shared vulnerability database, verifying the practical application value of PTGuard.

**Key words**: symbolic execution; Ethereum; smart contract; permission control vulnerability; arbitrary transfer vulnerability; contract vulnerability; vulnerability detection tools

## 0 引 言

区块链由加密数据块组成，具有不可修改、公开透明、去中心化等特点[1]。智能合约是去中心化的合约，能在没有可信的第三方时接收并执行交易[2]。但合约可能存在漏洞，受到攻击[3]，最严重的是，一旦智能合约成功部署到区块链上后，由于区块链的不可篡改性，合约代码无法被修改[4]，会导致用户损失巨大[5]。因此，保证智能合约无漏洞是非常有必要的。

现有工具虽然在智能合约的安全审计方面起了很大作用，但这些工具只覆盖了一些传统漏洞，对于一些新型漏洞缺乏漏洞检测算法，比如本文提到的权限控制漏洞和任意转账漏洞。基于此，本文针对权限控制漏洞和任意转账漏洞提出了一套检测算法，本文的贡献如下所示：

（1）本文总结出权限控制漏洞和任意转账漏洞的语义特征，构建漏洞模型，进一步提高检测精度和漏洞类型

**收稿日期**：2023-10-24；**修订日期**：2025-01-08

**基金项目**：国家自然科学基金项目（61872011）

**作者简介**：张艺琼（2000-），女，山西忻州人，硕士研究生，研究方向为区块链安全、漏洞挖掘、大模型安全；+通讯作者：文伟平（1976-），男，湖南益阳人，博士，教授，研究方向为系统与网络安全、大数据与云安全、智能计算安全；刘成杰（1998-），男，湖南衡阳人，博士研究生，研究方向为软件安全、漏洞挖掘、入侵检测。E-mail：weipingwen@pku.edu.cn

覆盖度；

（2）本文提出了针对权限控制漏洞和任意转账漏洞的检测工具 PTGuard，经实验验证，相比其它漏洞检测工具有着更高的精确率。且此工具已应用在科技部国家重点研发项目中，成功检测出 17 个智能合约漏洞，已上报到国家共享漏洞数据库并获得漏洞编号。

## 1 相关工作

最近几年出现的主流的智能合约自动化安全检测方法主要包括特征代码匹配、形式化验证、基于符号执行和符号抽象、模糊测试、程序静态分析这 5 种[6]。其中最常用的是基于符号执行和符号抽象的方法。

目前有不少使用符号执行进行智能合约漏洞检测的研究：Muller[7]开发出一款漏洞检测工具 Mythril，它的核心原理是符号执行，目前官方提供了整数溢出、代码重入等 14 种安全问题的检测方法[8]。Torres 等[9]提出一个漏洞检测框架 Osiris，主要结合了符号执行和污点分析，能检测整数错误包括整数截断、算数溢出等问题。Mossberg 等[10]研究出一个开源动态符号执行框架 Manticore，可用于检测智能合约及一些二进制文件。目前可以有效检测出包括时间戳依赖、重入漏洞、整数溢出等问题。Tsankov 等[11]提出了一种基于符号执行和符号抽象的静态漏洞检测工具 Securify，该工具对智能合约依赖图进行符号化分析并从代码中提取特征信息来进行安全性分析。

## 2 相关知识

### 2.1 以太坊与智能合约

以太坊是基于区块链实现的智能合约应用平台，它使用基于账户的模型记录每个账户的余额，转账是否合法只需判断转账者的余额是否足够。以太坊中存在两类账户：外部账户和合约账户[12]。外部账户使用公私钥控制，拥有私钥就拥有控制权。合约会在创建时会返回一个地址，可利用这个地址对其调用，它不可以主动发起交易，只有受到外部账户调用后才能发起交易或调用其它合约账户。以太坊账户间所有交互都是通过交易发生的。调用智能合约和转账交易相似。

智能合约运行是靠以太坊虚拟机即 EVM[13]进行的，它是基于堆栈的，因此所有的运算行为都是通过栈来进行的，并且出栈操作一般可以得到指令的参数或运算结果。

对于以太坊虚拟机中存放的数据，其中 storage 里存储的数据是非易失的，而 stack、args、memory 里存储的数据是易失的。若要操作存储结构里的数据，就要用到虚拟机指令。这些指令包括密码学运算、算术指令、栈操作、storage 操作等。后续漏洞模型的构建即是基于 EVM 字节码进行语义特征提取，表 1 为以太坊虚拟机中常用的 EVM 字节码。

表 1 常用的 EVM 字节码

| 汇编码 | 助记符 | 含义 |
|---|---|---|
| 10 | LT | 小于 |
| 20 | SHA3 | keccak256 哈希 |
| 30 | ADDRESS | 当前执行合约的地址 |
| 31 | BALANCE | 指定地址的余额 |
| 33 | CALLER | 消息调用方地址 |
| 54 | SLOAD | 从存储读取一个(u)int256 |
| 55 | SSTORE | 向存储写入一个(u)int256 |
| 56 | JUMP | 无条件跳转 |
| F1 | CALL | 调用另一个合约中的方法 |

### 2.2 符号执行概述

符号执行[14]是一种程序分析技术，它的关键目标之一是在一定时间内尽可能探索更多的不重复程序路径，并对于每一条程序路径都生成一组输入值来执行此路径，同时检查每一条路径是否存在一些错误，比如未捕获的异常、断言冲突、一些漏洞或内存损坏。

符号执行背后的核心思想[14]是通过使用符号值来替代具体的数值作为程序的输入，并将程序输入的变量值表示为抽象符号值。在智能合约的应用中，符号执行提供了一种自动生成输入的方法来模拟执行智能合约，它会收集所有的可达路径，如果某个输入会触发软件错误或者说是该路径出现漏洞标识，则可认为该智能合约中存在某个漏洞[15]。

## 3 漏洞检测算法

### 3.1 权限控制漏洞

权限控制漏洞是一类相当严重的漏洞，由于编写者的疏忽，攻击者可以通过非法手段窃取合约控制权。因此针对这类型漏洞归纳漏洞特征，设计并提出对应的漏洞检测算法。

#### 3.1.1 权限控制漏洞原理

当合约部署到以太坊上时，合约的创建者即合约的拥有者拥有最高控制权限。因此合约的权限控制对于合约的安全性来讲是非常重要的。

权限控制漏洞是指合约没有设置合理的访问控制限制，以及没有对合约进行有效的身份校验，从而导致攻击者非法窃取合约控制权。针对智能合约的权限控制漏洞可以从逻辑层面的权限约束进行考虑。合约通常是指通过修饰器 modifier 对函数执行前进行各种权限检查，以达到对关键函数进行权限控制的作用。权限控制漏洞主要是由于某些对权限有要求的函数修饰符发生逻辑错误，导致合约中的一些关键函数被攻击者非法调用。

#### 3.1.2 权限控制漏洞实例

如图 1 所示的权限控制漏洞代码，在第 13 行构造函数进行初始化合约权限的时候，设置了 public 权限，却没有使用 onlyowner 进行保护，任意调用者可以调用此函数，

因此攻击者就可以直接调用 initOwner 函数，将自己的合约地址作为函数参数，从而获得合约控制权，触发了权限控制漏洞。

```
1. pragma solidity ^0.4.25;
2.
3. contract Access{
4.   uint totalSupply = 0;
5.   address public owner;
6.   mapping (address => uint256) public balances;
7.
8.   modifier onlyOwner{
9.       if(msg.sender != owner)
10.          revert();
11.      _;
12. }
13. constructor() public{
14.      initOwner(msg.sender);
15. }
16. function initOwner(address _owner) public{
17.      owner = _owner;
18. }
19. function SendBouns(address re, uint bouns) public onlyOwner return (uint){
20.      require(balances[re] < 1000);
21.      require(bouns < 200);
22.      balances[re] += bouns;
23.      totalSupply += bouns;
24.      return balances[re];
25. }
26. }
```

图 1　权限控制漏洞示例代码

### 3.1.3　权限控制漏洞检测模型

在 solidity 语言中，modifier 是代表修饰符，它会告诉编译器这不是一个函数，不会被直接调用，而是添加到函数定义的末尾，用来改变函数的行为。onlyowner 修饰符会用来判断当前的调用者是不是合约的拥有者，即 *msg. sender*==*owner* 这个语句，所以在修改合约 *owner* 的语句前有 onlyowner 修饰符时，合约是没有权限控制漏洞的。

模型基于上面所述的观点构建，经过对大量合约字节码调研，结果显示：

（1）*owner* 主要是通过构造函数进行初始化的，值通常为 *msg. sender*（对应的字节码是 CALLER），*owner* 值会存储到 storage 槽位中（对应的字节码是 SSTORE），因此对获取 *owner* 值的字节码特征可以总结为：SSTORE[*key*] = CALLER → JUMP；

（2）onlyowner 是安全性保障，它会将调用者地址与合约所有者地址（即 *owner* 值）进行对比，*owner* 值会通过 SLOAD 字节码从 storage 槽位中获取，因此 onlyowner 修饰符的字节码特征可以总结为：EQ(SLOAD[*key*], CALLER) → ISZERO；

（3）SSTORE 字节码可存储变量也可修改变量，因此若后续检测到 SSTORE，判断它存储变量的槽位是否为 *owner* 所在的槽位，若是且没有经过 onlyowner 判断，那么当前合约存在权限控制漏洞。

图 2 是权限控制漏洞检测模型的流程图。具体检测流程如下：

（1）输入智能合约代码开始检测

对输入的智能合约代码进行编译，生成合约字节码。显然，与检测相关的关键字节码是“CALLER”和“SSTORE”，因此模型设置 hook 点为“CALLER”和“SSTORE”，然后检测模型判断合约字节码中是否存在这两个指令，若存在则继续以下的流程。

（2）对 hook 点进行判断

首先判断一下当前地址对应的指令是否分析过，若已分析过，则寻找下一个 hook 点。若未分析过，则判断当前的指令是哪一个指令。若是 SSTORE 指令，且在构造函数阶段，说明正在初始化 *owner* 值，当前栈顶元素的值正是 *owner* 值所在的槽位地址，存储该值以供后续检测。若是 CALLER 指令，求解 onlyowner 是否存在，并存储结果。若是 SSTORE 指令，且不在初始化阶段，判断当前修改的位置是否为 *owner* 值的槽位，若是且 onlyowner 修饰符不存在，那么当前合约存在权限控制漏洞。

（3）输出漏洞信息

记录符合上述过程的执行路径，进行约束求解，若求解成功，则构造漏洞信息进行上报。

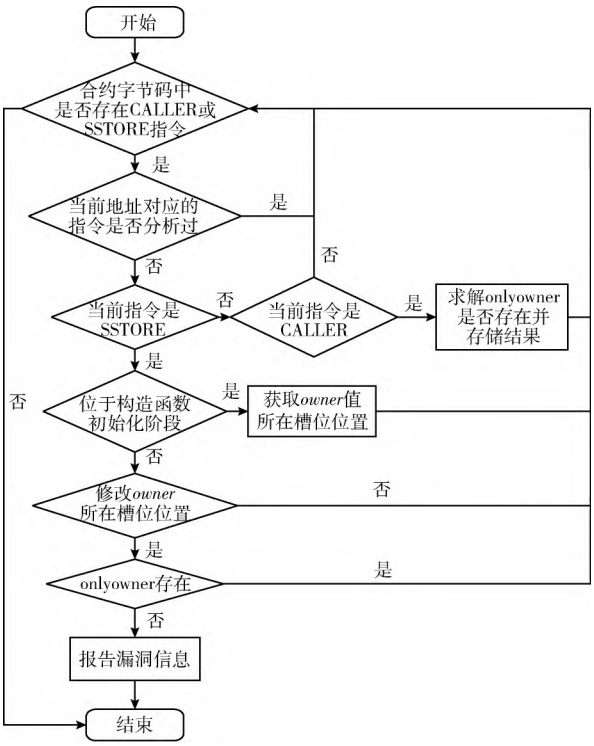


图 2　权限控制漏洞检测模型流程

### 3.2　任意转账漏洞

任意转账漏洞是合约的转账函数前缺少合理的限制条件，使得攻击者能够影响合约正常运行，甚至从合约的账

户中随意提取以太币，引发用户资产的损失。因此针对这类型漏洞归纳漏洞特征，并提出对应的漏洞检测算法。

3.2.1 任意转账漏洞原理

智能合约可以实现以太币的转入和转出，所以这部分合约的代码安全至关重要，若出现漏洞，容易让用户的财产被非法转移，造成不可逆的损失。

任意转账漏洞是缺少对转账人的限制，使得合约运行异常，攻击者可以从合约账户中提取部分甚至全部的以太币。合约转账一般有两种情况，一种是分账，也就是该合约允许将以太币按权重转给一组账户，因此在调用转账函数时，需对调用者在合约中能提取的余额值进行判断，如果输入非法，会导致转账操作异常甚至合约运行异常；一种是指定受益人，也就是合约只允许指定的一个受益人获得转账，因此在调用转账函数时，需要对调用者账户进行判断，如果没有对账户进行判断，则会导致财产被任意转移。

3.2.2 任意转账漏洞实例

如图 3 所示的任意转账漏洞的代码，代码第 19 行的外部函数 returnEth 含有任意转账的漏洞。该函数功能是向 *msg. sender* 进行转账，但在第 20 行调用转账函数前未对调用者账户地址进行判断，因此即使调用者账户地址不是 *ceoAddr*，也可以非法转移合约中存有的以太币。

```
1 pragma solidity ^0.4.19;
2 contract Adoption {
3   address ceoAddr = 0x1AEA2d3709bB7CFf5326a4Abc44c45Aa2629C626;
4   struct Pepe {
5     address owner;
6     uint256 price;
7   }
8
9   Pepe[16] data;
10
11  function Adoption() public {
12    for (uint i = 0; i < 16; i++) {
13
14      data[i].price = 10000000000000000;
15      data[i].owner = msg.sender;
16    }
17  }
18
19  function returnEth(uint256 price) public payable {
20    msg.sender.transfer(price);
21  }
22 }
```

图 3 任意转账漏洞示例代码

3.2.3 任意转账漏洞检测模型

总结来讲，合约转账时，需要在转账函数前对余额值或者调用者账户进行判断。

模型基于上面观点构建，经过对大量合约字节码调研，结果显示：

（1）变量 *balances* 通常是映射类型，可通过 *key* 值来查询对应的 *value* 值，在这里 *key* 为合约账户地址 *address*，*value* 为账户对应的金额，*value* 会存储在 storage 槽位中，槽位地址是对 *key* 值进行 SHA3 哈希得到的。得到槽位地址后，再通过 SLOAD 获取具体的 *value* 值进行判断。因此对 *balances* 值判断的字节码特征可以总结为：SLOAD[SHA3(*key*)] → GT → ISZERO；

（2）对调用者账户与受益人账户进行判断，受益人账户地址会存储到 storage 槽位中，通过 SLOAD 进行获取，调用者账户就是 *msg. sender*（对应字节码是 CALLER），获取值后就会进行值的判断。因此对账户判断的字节码特征可以总结为：EQ(SLOAD[*key*],CALLER) → ISZERO；

（3）转账函数有 transfer，send，call 等，但在字节码层面上都为 CALL，因此判断合约函数是否存在转账行为，即判断是否出现字节码 CALL。

图 4 是任意转账漏洞检测模型的流程图。具体检测流程如下：

（1）输入智能合约代码开始检测

对输入的智能合约代码进行编译，生成合约字节码。显然，与检测相关的关键字节码是“CALLER”、“SHA3”和“CALL”，因此模型设置对应的 hook 点“CALLER”、“SHA3”和“CALL”，然后检测模型判断合约字节码中是否存在这 3 个指令，若存在则继续以下的流程。

（2）对 hook 点进行判断

首先判断一下当前地址对应的指令是否分析过，若已分析过，那么寻找下一个 hook 点；若未分析过，就判断当前的指令是哪一个指令。若是 CALLER 指令，判断是否满足上述给出的账户判断字节码特征，存储对调用者账户与受益人账户的判断结果，以供后续检测。若是 SHA3 指令，判断是否满足上述给出的 *balances* 值判断字节码特征，存储判断结果以供后续检测。若是 CALL 指令，说明存在转账行为，判断是否存在限制条件，若不存在，那么当前合约存在任意转账漏洞。

（3）输出漏洞信息

记录符合上述过程的执行路径，进行约束求解，若求解成功，则构造漏洞信息进行漏洞上报。

## 4 整体框架设计与实现

漏洞检测工具整体架构流程如图 5 所示，分别是编译转换、静态分析、符号执行、漏洞检测、漏洞上报 5 个模块。

（1）编译转换模块

负责将输入的智能合约代码进行预处理，使用 solc 编译器将源码编译为 EVM 字节码。

（2）静态分析模块

负责将字节码划分为基本块，构建控制流图，流图的边表明了哪些基本块可能紧随一个基本块之后执行。

（3）符号执行模块

智能合约经过上面两步处理后，会部署到模拟的以太

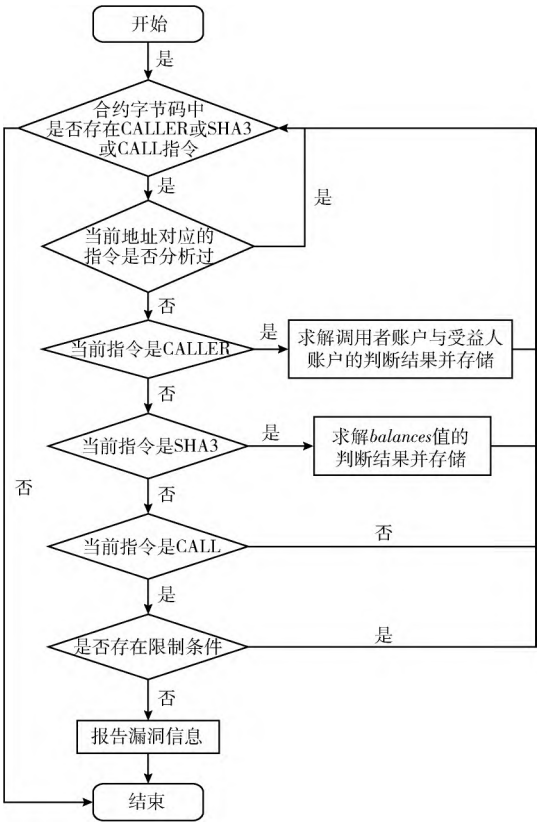


图 4　任意转账漏洞检测模型流程

坊虚拟机环境上来，部署完成后，以太坊虚拟机的模拟环境会存放一些合约相关的状态变量，比如机器状态、世界状态、全局状态等，通过改变这些状态来模拟智能合约进行交易。然后将智能合约的交易执行符号化，形成若干条执行的程序路径。

（4）漏洞检测模块

漏洞检测模块包含编写的漏洞检测算法。检测智能合约漏洞需要根据漏洞的原理以及特征来建立漏洞的检测模型，通过检测模型来检测合约中是否包含对应的漏洞。具体步骤包括分析漏洞的触发条件，构造自动化的检测方案，设置 hooks，在代码的执行路径上获取漏洞的 hook，当 hook 出现时，判断是否满足设定的漏洞约束，进行求解，通过判断求解值来确认有无漏洞。

（5）漏洞上报模块

如果求解器显示有解，则说明这个漏洞确实存在，最后保存漏洞信息，并进行漏洞的上报。

## 5　实验及分析

### 5.1　实验环境

对于本文编写的权限控制漏洞和任意转账漏洞的检测算法，需要对其性能进行测试，测试算法是否能从线上智能合约中检测到漏洞，以及本文提出的 PTGuard 工具相对于其它漏洞检测工具的有效性。表 2 和表 3 为实验环境介绍。

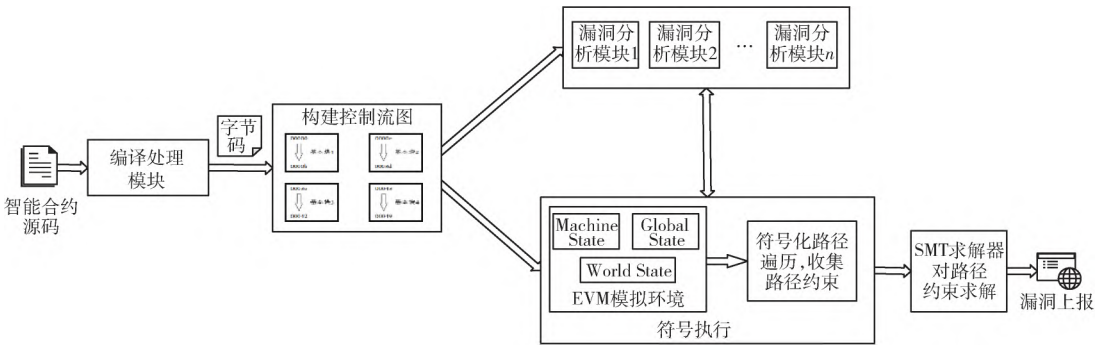


图 5　整体架构

表 2　硬件环境

| 名称 | 配置 |
|---|---|
| CPU | I5-8250 |
| 内存 | 8 GB |
| 显卡 | 独立显卡 UHD Graphics 620 |

表 3　软件环境

| 名称 | 配置 |
|---|---|
| 系统 | Ubuntu 18.04.5 LTS |
| 内核版本 | 5.4.0-150-generic |
| Python | 3.6.9 |

### 5.2　实验结果

#### 5.2.1　漏洞扫描结果

为了对漏洞检测算法的有效性进行测试，本文从以太坊上爬取了最近已部署的 2000 个合约。最终成功检测出了 CNVD-2022-31809、CNVD-2022-31808 等 13 个权限控制漏洞和 4 个任意转账漏洞，测试结果表明本文提出的漏洞检测模型具有高效的漏洞检测能力。

此处选取两个漏洞来进行具体介绍：

（1）CNVD-2022-31809

该漏洞出现在 RenBTC 项目中，该项目运行在以太坊主链的 ERC20 合约（合约地址为 0xEB4C2781e4eb-

A804CE9a9803C67 d0893436bB27D)。摘取代码片段如图 6 所示。

```
1 contract CanReclaimTokens is Claimable {
2   using SafeERC20 for ERC20;
3   mapping(address => bool) private recoverableTokensBlacklist;
4
5   function initialize(address _nextOwner) public initializer {
6     Claimable.initialize(_nextOwner);
7   }
8
9   function blacklistRecoverableToken(address _token) public onlyOwner {
10     recoverableTokensBlacklist[_token] = true;
11  }
12
13  function recoverTokens(address _token) external onlyOwner {
14    require(
15      !recoverableTokensBlacklist[_token],
16      "CanReclaimTokens: token is not recoverable"
17    );
18
19    if (_token == address(0x0)) {
20      msg.sender.transfer(address(this).balance);
21    } else {
22      ERC20(_token).safeTransfer(
23        msg.sender,
24        ERC20(_token).balanceOf(address(this))
25      );
26    }
27  }
28}
```

图 6 CNVD-2022-31809 代码片段

可以看到代码中的 initialize 函数（第（5）行～第（7）行），可以初始化合约 *owner* 值，此函数并未设置 onlyOwner 等权限控制条件，因此攻击者可将自己的合约地址通过参数传入 initialize 函数中，然后调用 initialize 函数就可以将 *owner* 修改为自己的合约地址，获取合约权限。攻击者就可以调用包含 onlyowner 修饰符 recoverTokens 函数（第（13）行～第（27）行），将合约内所有余额转到自己账户中。

（2）CNVD-2022-31808

该漏洞出现在 UserfeedsClaimWithConfigurableValueMultiTransfer 项目中，该项目运行在以太坊主链上的 ERC20 合约（合约地址为 0xfad31a5672fBd8243E-9691E8a5F958699CD0AaA9）。摘取代码片段如图 7 所示。

```
1 contract UserfeedsClaimWithValueMultiSend {
2    function send(address[] recipients) public payable {
3        uint amount = msg.value / recipients.length;
4        for (uint i = 0; i < recipients.length; i++) {
5            recipients[i].send(amount);
6        }
7    msg.sender.transfer(address(this).balance);
8    }
9 }
```

图 7 CNVD-2022-31808 代码片段

代码第 2 行外部函数 send 函数含有任意转账的漏洞。任意账户调用该函数，都可以将合约中存有的以太币全部转走。攻击者向参数给定的地址数组发送一定数量的以太币，但合约并没有对以太币的数值进行约束，攻击者便可以用极小的金额套取合约的全部以太币。

5.2.2 漏洞检测工具对比

（1）数据集介绍

漏洞样本来源于 CVE、CNVD 等漏洞库，经扫描总共获取 48 个漏洞样本，其中包含 31 个权限控制漏洞，17 个任意转账漏洞。

正常样本来源于以太坊线上经审查无漏洞的智能合约，总共 92 个正常样本，其中包含 49 个正常样本用于权限控制漏洞检测，43 个存在交易转账行为的正常样本用于任意转账漏洞检测。

（2）评价指标介绍

为评估本文提出的算法的性能，采用 4 个指标对算法进行评估，即准确率（*Accuracy*）、精确率（*Precision*）、召回率（*Recall*）以及 F1 分数（*F1_score*），分别如式（1）～式（4）所示

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$F1\_score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

$TP$ 代表真阳性，表示将正样本预测为正样本的数量；$TN$ 代表真阴性，表示将负样本预测为负样本的数量；$FP$ 代表假阳性，表示将负样本预测为正样本的数量；$FN$ 代表假阴性，表示将正样本预测为负样本的数量。这些指标结果是计算其它指标的基础。

准确率表示预测正确的数据占总样本的比例，精确率表示实际正确样本被判定为正样本的比例，召回率表示预测为正样本占实际为正样本的比例。F1 分数是精确率和召回率的调和平均值，同时兼顾这两个指标。

（3）实验结果评估

本文将 PTGuard 工具与其它主流的智能合约开源检测工具进行对比，对比的对象有：Muller[7] 开发的 Mythril 检测工具、Feist 等[16] 开发的 Slither 检测框架、Tikhomirov 等[17] 开发的 SmartCheck 检测工具。结果见表 4。

由表 4 可以看出，与其它检测工具相比，本文模型算法具有明显的优势，可以看到在 4 个指标上都有明显的提升，准确率、精确率、召回率和 F1 值分别为 86.25%、85.71%、77.42%、81.35%。这是因为这些工具只覆盖了一些传统漏洞，对于一些新型漏洞缺乏漏洞检测算法，本文提出的检测算法针对这些漏洞提取语义特征，构建多层判断条件，提高漏洞被召回的概率，同时由于特征提取准确，也降低了算法的误报率。

表 4　漏洞检测工具测试结果

| 检测工具 | 权限控制漏洞 | | | | 任意转账漏洞 | | | |
|---|---|---|---|---|---|---|---|---|
| | 准确率/% | 精确率/% | 召回率/% | F1 值/% | 准确率/% | 精确率/% | 召回率/% | F1 值/% |
| Mythril | 67.50 | 72.73 | 25.80 | 38.09 | 81.67 | 71.43 | 58.82 | 64.51 |
| Slither | 70.00 | 73.33 | 35.48 | 47.82 | 85.00 | 75.00 | 70.59 | 72.73 |
| SmartCheck | 71.25 | 83.33 | 32.26 | 46.51 | 75.00 | 62.50 | 29.41 | 40.00 |
| PTGuard | 86.25 | 85.71 | 77.42 | 81.35 | 91.67 | 83.33 | 88.24 | 85.71 |

## 6　结束语

本文针对权限控制和任意转账漏洞会造成用户经济损失，以及当前工具对于新型漏洞缺乏漏洞检测算法，检测精度低的问题，提出了基于符号执行的漏洞检测算法，实现了针对合约漏洞自动化检测的工具 PTGuard，最后通过对线上智能合约的检测，发现并上报了 13 个权限控制漏洞和 4 个任意转账漏洞。与其它漏洞检测工具进行实验对比，表明了本文设计的漏洞检测算法在准确率、精确率、召回率和 F1 值上有较大的提升。

在未来的工作中，将探索总结更多的合约特征，提高漏洞类型覆盖度以及检测精度，并且会针对符号执行效率问题进行深入研究，更好平衡精度和性能。

## 参考文献：


[1] HU Tianyuan，LI Zecheng，LI Bixin，et al. Contractual security and privacy security of smart contract：A system mapping study [J]. Chinese Journal of Computers，2021，44 (12)：2485-2514 (in Chinese). [胡甜媛，李泽成，李必信，等. 智能合约的合约安全和隐私安全研究综述 [J]. 计算机学报，2021，44 (12)：2485-2514.]

[2] Grishchenko I，Maffei M，Schneidewind C. A semantic framework for the security analysis of ethereum smart contracts [C] //International Conference on Principles of Security and Trust. Springer，2018：243-269.

[3] Ghaleb A，Pattabiraman K. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection [C] //Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis，2020：415-427.

[4] Nguyen T D，Pham L H，Sun J. SGUARD：Towards fixing vulnerable smart contracts automatically [C] //IEEE Symposium on Security and Privacy. IEEE，2021：1215-1229.

[5] Raj A，Maji K，Shetty S D. Ethereum for internet of things security [J]. Multimedia Tools and Applications，2021，80 (12)：18901-18915.

[6] ZHANG Guanghua，LIU Yongsheng，WANG He，et al. Smart contract vulnerability detection scheme based on BiLSTM and attention mechanism [J]. Netinfo Security，2022，22 (9)：46-54 (in Chinese). [张光华，刘永升，王鹤，等. 基于 BiLSTM 和注意力机制的智能合约漏洞检测方案 [J]. 信息网络安全，2022，22 (9)：46-54.]

[7] Mueller B. MYTHRIL：A framework for bug hunting on the ethereum blockchain [EB/OL]. [2017-07-11]. https://mythx.io/.

[8] NI Yuandong，ZHANG Chao，YIN Tingting. A survey of smart contract vulnerability research [J]. Journal of Cyber Security，2020，5 (3)：78-99 (in Chinese). [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.]

[9] Torres C F，Schütte J，State R. Osiris：Hunting for integer bugs in ethereum smart contracts [C] //Proceedings of the 34th Annual Computer Security Applications Conference，2018：664-676.

[10] Mossberg M，Manzano F，Hennenfent E，et al. Manticore：A user-friendly symbolic execution framework for binaries and smart contracts [C] //34th IEEE/ACM International Conference on Automated Software Engineering. IEEE，2019：1186-1189.

[11] Tsankov P，Dan A，Drachsler-Cohen D，et al. Securify：Practical security analysis of smart contracts [C] //Proceedings of the ACM SIGSAC Conference on Computer and Communications Security，2018：67-82.

[12] Norvill R，Pontiveros B B F，State R，et al. Visual emulation for Ethereum's virtual machine [C] //IEEE/IFIP Network Operations and Management Symposium. IEEE，2018：1-4.

[13] Pinna A，Ibba S，Baralla G，et al. A massive analysis of ethereum smart contracts empirical study and code metrics [J]. IEEE Access，2019，7：78194-78213.

[14] Baldoni R，Coppa E，D'elia C D，et al. A survey of symbolic execution techniques [J]. ACM Computing Surveys，2018，51 (3)：1-39.

[15] Zheng P，Zheng Z，Luo X. Park：Accelerating smart contract vulnerability detection via parallel-fork symbolic execution [C] //Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis，2022：740-751.

[16] Feist J，Grieco G，Groce A. Slither：A static analysis framework for smart contracts [J]. CoRR，2019：8-15.

[17] Tikhomirov S，Voskresenskaya E，Ivanitskiy I，et al. Smartcheck：Static analysis of ethereum smart contracts [C] //Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain，2018：9-16.