Journal of Computer Applications 计算机应用 | ISSN 1001-9081 CODEN JYIIDU | 2019-12-17 http://www.joca.cn

文章编号:1001-9081(****)**-0000-00 doi:10.11772/j.issn.1001-9081.2019111919

基于符号执行的智能合约漏洞检测方案

赵伟[1,2]，张问银[2*]，王九如[2]，王海峰[2]，武传坤[2]

1 临沂大学 信息科学与工程学院，山东 临沂 276002;

2 山东科技大学计算机科学与工程学院 山东 青岛 266000

(*zhangwenyin@lyu.edu.cn)

摘 要: 智能合约是区块链核心技术之一，其安全可靠性至关重要。随着区块链技术的应用推广，智能合约的数量呈现爆发式增长，但智能合约的漏洞将给用户带来巨大损失。在分析以太坊智能合约运行机制和常见漏洞原理的基础上，利用符号执行技术检测智能合约漏洞。首先以太坊字节码构建智能合约执行控制流图，再根据智能合约漏洞特点设计相应的约束条件，利用约束求解器生成软件测试用例，检测常见的整型溢出，权限控制，Call 注入，重入攻击等智能合约漏洞。实验结果表明，检测方案具有良好的检测效果，智能合约漏洞检测正确率达 85%。

关键词: 区块链；智能合约；符号执行；漏洞分析；以太坊

中图分类号: TP309 **文献标志码:** A

Smart Contract Bugs Detection Scheme Based on Symbol Execution

ZHAO Wei[1,2] , ZHANG Wen-yin[2], WANG Jiu-ru[2] , WANG Hai-feng[2] , WU Chuan-kun[2]

(1 School of Information Science and Engineering, Shandong Linyi University, 276002, Linyi ;

2 School of Computer Science and Engineering, Shandong University of Science and Technology, 266000, Qingdao)

Abstract: Smart contract is one of the core technologies of blockchain, and its security and reliability are very important. With the popularization of blockchain application, the number of smart contracts has increased explosively, but the bugs of smart contracts will bring huge losses to users. Based on the analysis of the operation mechanism and common vulnerabilities of Ethereum smart contract, this paper uses symbol execution technology to detect bugs in smart contract. By constructing the smart contract control flow graph of ethereum bytecode, the constraint conditions are designed according to the characteristics of smart contract bugs, and the constraint solver is used to generate test cases to detect the common bugs of smarts contracts such as integer overflow, owner control, call injection and reentry attack. The experimental results show that the detection scheme has good detection effect, and the accuracy rate of smart contract bugs detection is up to 85%.

Keywords: Blockchain; smart contract; symbol execution; vulnerability analysis; Ethereum

0 引言

随着区块链技术的快速发展，区块链已经由以比特币为主的区块链 1.0 时代进入以以太坊(Ethereum)、商用分布式设计区块链操作系统(Enterprise Operation System，EOS)等平台为主的区块链 2.0 时代。智能合约的广泛应用是区块链 2.0 时代的一大标志。智能合约作为区块链上一段自动触发、执行的程序代码，应用频率高，需要极高的安全性。但在代码设计过程中不免出现编码安全、设计缺陷等问题，将导致智能合约存在安全隐患，对合约使用者造成不可估量的损失。据统计，2018 年以太坊智能合约总量已经超过 300 万个，公开的合约超过 5 万个，其中 74.48%的公开合约都存在安全隐患[1]。

随着智能合约数量不断增多，功能复杂性日益增强，传统人工审计的方式已经不足以满足合约安全性的需要，这引起了诸多学者的关注。付梦琳[2]针对智能合约漏洞检测现状做了综述，讨论了形式化证明、模糊测试、符号执行、污点分析等主流检测方法，指出了形式化证明和静态分析是合约安全性分析的两种重要解决方案。

Grishchenko[3]提出了基于以太坊虚拟机(Ethereum virtual machine，EVM)字节码的小步骤语义与智能合约的核心安全

收稿日期: 2019-11-05; 修回日期:2019-11-18; 录用日期: 2019-11-21。

基金项目：山东省重点研发计划项目(2017CXGC0701，2019GNC106027)

作者简介: 赵伟(1994-),河南平顶山人，硕士研究生，主要研究方向为区块链技术；张问银(1972-),山东临沂人，博士，教授，主要研究方向为图像处理，信息隐藏，区块链技术；王九如(1983-),山东临沂人，博士，教授，主要研究方向为网络空间安全，区块链技术；王海峰(1976-),山东临沂人，博士，副教授，主要研究方向为计算机体系结构、高性能集群计算、复杂网络分析；武传坤(1964-),山东临沂人，博士，教授，主要研究方向为信息安全，移动网络安全，物联网安全。

属性，并对其进行形式化，获得了可执行代码，成功地对官方测试套件进行了验证，并在此基础上构建了一个EtherTrust[4]静态分析工具，并对 EVM 字节码静态分析进行了可靠性证明。Sukrit Kalra[5]提出了 ZEUS 工具，利用抽象解释和符号模型检查，以及约束 Horn 子句功能，快速验证契约的安全性。

Tsankov[6]开发了一种可扩展、完全自动化的 Ethereum 智能合约安全分析工具 Securify，通过分析合约的依赖关系图推导出语义事实，检查合约的合法与非法性，并利用特定域语言实现工具的可扩展性。Tikhomirov[7]开发了一种智能合约的静态分析工具 SmartCheck，通过将 solidity 解析生成 XML 解析树作为中间表示(Intermediate Representation，IR)，并根据 XPath 模式来检测漏洞。J Krupp[8]等开发了 TEETHER 工具，通过 Evm 字节码利用后向切片重构控制流图(Control Flow Graph, CFG)，利用路径遍历生成约束模块用于查找漏洞。Loi Luu[9]等提出了增强以太坊操作语义的方法，降低合约脆弱性，开发的 Oyente 工具可以无需访问 solidity 源码，直接基于 EVM 字节码检测时间戳依赖、重入漏洞、错误异常处理等漏洞。Slipher[10]和 Manticore[11]提出了基于符号执行的静态分析框架。Slipher 定义了一套 solidity 的 IR 表示，利用 IR 进行合约的漏洞检测，而 Manticore 是基于二进制文件的符号执行工具，提供了一套 API 供开发人员自定义漏洞检测方案。

上述研究工作主要侧重于以太坊智能合约的语义分析，符号执行的建模与优化等工作，没有详细描述利用符号执行技术检测智能合约漏洞流程，以及如何检测智能合约常见漏洞。本文在分析智能合约漏洞特点的基础上，利用符号执行技术遍历智能合约代码执行路径，并针对合约漏洞特点设计相应的路径约束条件，检测以太坊智能合约常见漏洞。基于 Awesome-Buggy-ERC20-Tokens 漏洞库，检测正确率达 85%。

1 智能合约概述

1.1 智能合约

智能合约是一种旨在以信息化方式传播、验证或执行合同的计算机协议。最先由 Nick Szabo 在 1995 年提出，“一个智能合约是一套以数字形式定义的承诺，包括合约参与方可以在上面执行这些承诺的协议”[12]。智能合约的具有达到执行条件自动触发、自动执行的特点，执行过程不需要任一参与方干预，这些特点使得智能合约必须在可信对等的环境下执行。而区块链的去中心化、数据不可篡改、可溯源等特点为智能合约提供了可信的执行环境。

智能合约有着广阔的应用前景，除了热门的加密货币场景外，智能合约在金融、投票、供应链、电子商务等领域都有着广泛应用。智能合约可以提高区块链事务的执行效率，但区块链的不可以篡改性也使得合约开发必须严谨，避免部署后出现不可挽回的损失。

1.2 以太坊智能合约

以太坊(Ethereum)是一个开源的区块链开发平台，致力于构建下一代加密货币与去中心化应用(Decentralized Application, DApp)平台。以太币(Ether)是仅次于比特币的全球第二大加密货币，通过智能合约可以在以太坊平台上快速构建去中心化应用。相比于比特币，以太坊支持图灵完备的智能合约，提供“叔块”的奖励机制并减少出块时间，支持 POW 共识的同时引入 POS 共识，减少能源消耗，采用支持复杂逻辑的账户模型。这些特点使得以太坊在加密货币、金融、非金融领域都有着广阔的应用前景。

以太坊通过帐户模型、交易过程、合约执行等方面更好地支持了智能合约。以太坊采用的账户模型，相比于比特币采用的 UTXO 模型，具有直接访问交易状态，较小的存储空间、易于编程等特点，较好地支持了智能合约。在以太坊交易中，平台为账户转帐、合约创建、合约执行构建了执行环境，以太坊虚拟机 EVM 提供一种完全隔离的运行环境，在 EVM 中运行的智能合约不能访问网络、文件系统、其他进程，不同合约之间的访问也受到限制[13]。

Solc 是以太坊常用的编译器，以太坊合约通过 solc 生成汇编代码，汇编代码包括 3 部分：部署代码、runtime 代码、auxdata。部署代码是创建合约是时的运行的代码；runtime 代码是合约运行时的代码；auxdata 是合约的指纹验证，不会被 EVM 执行。将 runtime 代码反编译可以获取到以太坊的字节码。以太坊字节码长度设定为 1 个字节，最大可以有 256 个操作码，目前已经定义 144 种操作码，支持算术、逻辑、比较、跳转等操作，表 1 展示了部分常用操作码。

表1 以太坊部分常用操作码

Tab. 1 common opcodes in Ethereum

操作码	汇编指令	介绍
0x00	STOP	停止执行
0x01	ADD	从栈中取出 arg0、arg1，将 arg0 + arg1 结果入栈
0x02	MUL	从栈中取出 arg0、arg1，将 arg0 * arg1 结果入栈
0x03	SUB	从栈中取出 arg0、arg1，将 arg0 - arg1 结果入栈
0x55	SSTORE	从栈中取出 arg0、arg1,将 arg1 存放到 Storage 的 arg0 处
0x57	JUMPI	从栈中取出 arg0、arg1，当 arg1 为真时，跳转到 arg0 处
0xf0	CREATE	创建合约并返回合约地址
0xf1	CALL	调用某个地址的合约

0xf3	RETURN	结束执行，返回数据
0xfd	REVERT	结束执行，程序异常，回滚所有状态

2 智能合约漏洞分析

智能合约漏洞主要分为 5 种类型：编码规范问题、设计缺陷问题、编码安全问题、编码设计问题、编码问题隐患[14]。主要问题如表 2 所示。

表 2 以太坊智能合约漏洞类型

Tab. 2 Ethereum smart contract vulnerability types

问题类型	主要描述
编码规范	编译器版本、构造函数问题、返回标准、时间标准、假充值问题
设计缺陷	Approve 授权函数条件竞争问题、循环消耗问题、循环安全问题
编码安全	溢出问题、重入问题、Call 注入、权限问题
编码设计	地址初始化、判断函数、余额判断、转账函数、代码外部调用、错误处理、弱随机数、变量覆盖等问题
编码隐患	语法特性、数据隐私、数据可靠性、gas 消耗、回调函数、Owner 权限、条件竞争等问题

编码规范、设计缺陷、设计问题、编码隐患漏洞类型主要由以太坊底层设计、开发标准存在缺陷产生。编码安全问题主要由开发人员在合约编写时疏忽导致，因此智能合约常见漏洞集中于编码安全问题，本文将主要关注编码安全问题。

2.1 整型溢出

整型溢出属于以太坊智能合约高危漏洞，此漏洞会导致任意铸币,超额铸币，超额购币，任意定向分配，下溢增持等多种漏洞场景。Solidity 最大支持整型变量为 256bit，uint256 支持取值范围为[0,2^256-1]，当数值超出这个范围就会产生数值异常。在智能合约中对变量进行算术运算时极易产生整型溢出漏洞。

程序 1 Totalsupply-Overflow 漏洞：

```
function mintToken(address target, uint256 mintedAmount)
onlyOwner
{
  balanceOf[target] += mintedAmount;
  totalSupply += mintedAmount;
  …
}
```

如程序 1 所示，在 mintToken 函数中，如果传入一个过大的 mintedAmount 值，从而可能造成 target 地址的余额溢出变为一个很小的值，同时代币总量 totalSupply 也可能产生溢出。

程序 2 BatchTransfer-Overflow 漏洞：

```
function batchTransfer(address[] _receivers, uint256 _value)
public returns (bool)
{
  uint cnt = _receivers.length;
  uint256 amount = uint256(cnt) * _value;
  require(cnt > 0 && cnt <= 20);
  require(_value > 0 && balances[msg.sender] >= amount);
  ...
  return true;
}
```

如程序 2 所示，batchTransfer 是一个批量转账的函数，当传入_value 值过大时，uint256(cnt) * _value 可能导致 amount 溢出变为一个极小的值，从而绕过 balances[msg.sender] >= amount 余额判断，转出大量超过用户余额的币。

实际案例：美链发布的 BEC 合约出现过整型溢出漏洞，漏洞原因为程序 2 所述，攻击者利用漏洞转出约 64 亿的 BEC Token，导致 BEC 代币急速贬值，最终市值近乎为 0，对 BEC 市场造成了毁灭性的打击。

2.2 权限控制

以太坊合约开发者(Owner)具有合约的超级权限，包括冻结代币、增发代币、销毁代币、铸造新的代币、终止合约运行等，当合约权限被攻击者窃取将造成严重影响。以太坊合约权限漏洞主要由合约开发者疏忽导致，开发者一处笔误将导致合约权限被任何人控制。权限漏洞产生主要有 2 点：函数修饰符使用不当与构造函数书写错误。

程序 3 Setowner-Anyone 漏洞：

```
function setOwner(address _owner) returns (bool success)
{
  owner = _owner;
  return true;
}
```

如程序 3 所示，Solidity 函数修饰符默认为 public，表明此函数可以被外部、合约内部和子合约直接调用。当合约开发者为合约设置 Owner 权限时，没有设置权限检查，将导致所有人都可以获取 Owner 权限。setOwner 函数用于设置合约管理员，此处函数修饰符为 public，并且没有任何权限检查，导致任何人都可以调用此函数更改合约管理员。

程序 4 Constructor 漏洞：

```
contract Owned
{
    address owner;
    function owned() public
    {
        owner = msg.sender;
    }
}
```

如程序 4 所示，构造函数在合约部署时会对变量初始化，构造函数必须与合约名称同名，当开发者误将构造函数写错将导致此函数变为可以被外部调用的普通函数，从而产生权限漏洞。Owned 合约中的构造函数 owned()与合约名称不一致，导致 owned()可以被任意调用造成权限漏洞。而在 0.4.22 版本的 Solc 发布后，将构造函数统一命名为 constructor，减少了此类漏洞的发生。

实际案例： Bancor 合约和 KickICO 合约都出现过合约权限被盗的问题，分别损失了 1250 万美元和 770 万美元。主要原因是 Owner 的私钥泄露导致合约权限被盗。虽然没有上述漏洞造成损失的直接案例，但 Owner 拥有权限过大，权限被盗依然属于高危漏洞。

2.3 Call 注入

Solidity 提供了 3 种合约间交互的方式：call、callcode 和 delegatecall。这 3 种函数调用过程中会引起全局变量 msg 的变化，msg 包括一些可以被合约访问的区块链属性，如 gas，消息调用者(msg.sender)等属性，结合一些特定场景将产生漏洞风险。表 3 比较了 3 种函数的异同点。

表 3. call、callcode、delegatecall 异同点

Table 3. Similarities and differences between call, callcode and delegatecall

	Msg 值	运行环境
Call	修改为调用者	被调用者
Callcode	修改为调用者	调用者
Delegatecall	不会修改	调用者

如程序 5 所示，在 CallBug 合约中，authority 函数只允许合约自我调用，当攻击者对 callFunc 函数中 data 参数传入 bytes4(keccak256(“authority()”))时，便可通过 call 函数特性修改调用者(msg.sender)，当前 call 的调用者为 this，从而绕过 require 的权限检查。

程序 5 Call 注入漏洞：

```
contract CallBug
{
    function callFunc(bytes data)
    {
        this.call(data);
    }
    function authority() public
    {
        require(this == msg.sender);
        ...
    }
}
```

如程序 6 所示，在 DelegatecallBug 合约中，当攻击者调用 delegatecallFunc 时传入 Attacker 合约的地址，构造 data 参数为 bytes4(keccak256(“attack()”))时。由于 delegatecall 不会修改 msg 的值，此时 msg.sender 为攻击者账户地址，但 delegatecall 的运行环境却是 Attacker 合约的，当攻击者在 attack 函数中对 DelegatecallBug 合约进行转账或者修改权限等攻击就有可能成功。

程序 6 Delegatecall 注入漏洞：

```
contract DelegatecallBug
{
    function delegatecallFunc(address addr, bytes data)
    {
        addr.delegatecall(data);
    }
}
contract Attacker
{
    function attack()
    {
        ... // 对 DelegatecallBug 合约转账、修改权限等攻击
    }
}
```

实际案例：Parity Multisig 钱包曾存在 delegatecall 漏洞，攻击者利用漏洞窃取了价值 3000 万美元的 Ether。漏洞的主要原因是钱包 initMultiowned 函数可以多次调用。虽然 initMultiowned 函数位于 WalletLibrary 合约下，无法直接调用，但攻击者利用 Wallet 合约中的 delegatecall 函数调用，修改合约管理员为攻击者自己并进行大量转账操作从而窃取了 15 万的 Ether。

2.4 重入攻击

以太坊 EOA 和 CA 账户都可以拥有 Ether，当向 CA 账户转账时会触发合约内的回调函数(fallback)。当正常合约向攻击合约转账时会触发攻击合约的回调函数，迫使合约回调自身代码，造成重入漏洞。

程序 7 ReEntrancy 漏洞：

```
contract ReEntrancyBug {
  ...
  function withdraw(address addr, uint256 amount) public
  {
    require(balances[msg.sender] >amount);
    require(address(this).balance >amount);
    addr.call.value(amount)();
    balances[msg.sender] -= amount;
  }
}
contract Attacker
{
  constructor(address _reAddr)
  {
    re= ReEntrancyBug (_reAddr);
  }
  ...
  function () public payable
  {   //fallback 函数
    if(re.balance >1 ether )
    {
      re.withdraw(addr,amount);
    }
  }
}
```

如程序 7 所示，ReEntrancyBug 是一个拥有存取 Ehter 功能的合约，攻击者窃取此合约的 Ether 步骤如下：

1. 攻击者首先向 ReEntrancyBug 合约中存入 1Ether；
2. 攻击者调用 ReEntrancyBug 合约中的 withdraw 函数向 Attacker 合约转入 1Ehter；
3. withdraw 函数执行到 addr.call.value(amount)() 会向 Attacker 合约转账并触发 Attacker 合约中的 fallback 函数；
4. 此时 withdraw 函数并没有执行到 balances[msg.sender] -= amount 这一步，因此 Attacker 合约账户仍然有 1Ether，继续调用 ReEntrancyBug 合约中的 withdraw 函数转账，产生重入漏洞。
5. 5.当合约账户小于等于 1Ether 时，结束回调，执行 balances[msg.sender] -= amount。

实际案例：The DAO 是基于以太坊的一个众筹项目，攻击者利用 The DAO 合约中的重入漏洞进行了 200 多次的攻击并成功向其他地址转出 360 万的 Ether，对众筹参与者造成了巨大的损失。并且由于这次攻击造成了以太坊历史上的首次硬分叉。

3 基于符号执行的检测方案

3.1 符号执行原理

符号执行的主要思想是将变量符号化，通过将符号化变量作为程序的输入，探索程序执行路径并收集路径约束，最后利用约束求解器得到新的测试输入，检测符号值是否可以产生漏洞[15]。而符号化主要是要建立符号与内存、寄存器之间的映射关系。传统符号执行在面对复杂路径时求解困难，无法生成新的测试用例。Godefroid[16]和 Sen K[17]提出了动态符号执行的方法，将具体执行和符号执行结合，利用具体值代替符号值作为程序的输入，分析精度较高且实现较为容易，近些年来被广泛使用。图 1 给出了动态符号执行流程。

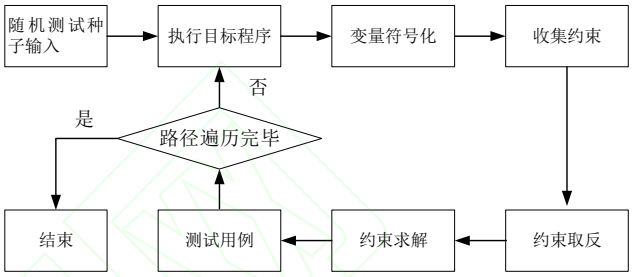

图 1 动态符号执行流程

Fig.1 Dynamic symbol execution flow

利用符号执行检测合约漏洞的基本过程为：

1. 利用 solc 编译器对合约源码进行编译生成汇编代码，汇编代码包含：部署代码、runtime 代码和 auxdata。
2. 利用 solc 编译器对 runtime 代码进行反编译生成以太坊合约字节码。
3. .通过以太坊合约字节码构建控制流图。
4. 随机生成测试数据，遍历控制流图可达路径，收集路径约束
5. 利用约束求解器对路径约束求解，生成测试用例。

通过合约字节码构建控制流图基本过程如下所示，其中 P 代表智能合约字节码指令集合，S 代表字节码，B 代表代码块，E 代表代码块之间的边。

伪代码：字节码构建控制流图

```
Input: smart contract bytecode P
FOR each instruction of P
  IF S is the first instruction of P THEN
    Create a new B;
    Insert S into B;
  END IF
  IF S = “JUMPI” THEN
    Insert S into B;
    End current B;
  END IF
  Insert S into B;
END FOR
Build E by jumping between B
```

图 2 展示了一段智能合约生成控制流图的过程。

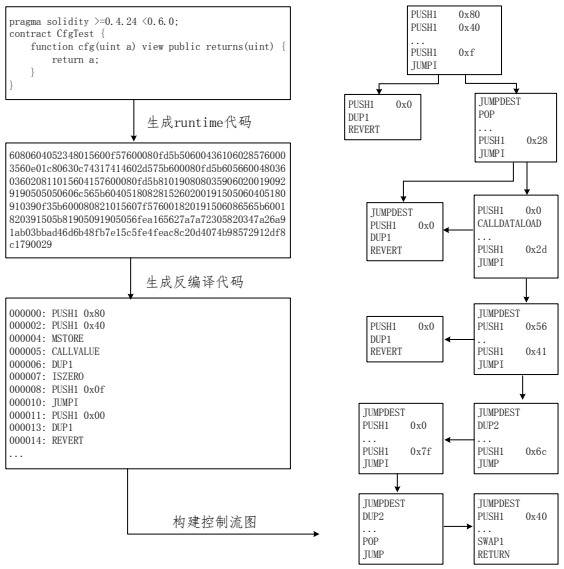

图 2 智能合约生成控制流图过程

Fig. 2 Process of smart contract generation control flow graph

3.2 典型漏洞检测

本文基于以太坊 ERC20 开发标准，针对智能合约整型溢出、权限控制、Call 注入，重入漏洞给出检测方案。

3.2.1整型溢出检测

表 4 整型溢出判断方案

Tab. 4 Integer overflow judgment scheme

判断类型	指令	运算表达式	约束
加法溢出	ADD(a,b)	a+b=result	UGE(result,2^256)
减法溢出	SUB(a,b)	a-b=result	UGT(b,a)
乘法溢出	MUL(a,b)	a*b=result	UGE(result,2^256)

整型溢出会导致算术运算后的结果(result)变成一个极小值甚至归 0，对执行结果检测很容易判断是否溢出，图 3 给出了整型溢出检测的基本流程。例如判断 uint a+ uint b = result 是否溢出，随机构建一个极大的值作为程序的输入，对执行结果添加约束 UGT（a,result）OR UGT（b,result），即 result 比 a 或 b 小，则可能发生了整型溢出。利用 a 和 b 的符号值和路径信息进行符号运算，当遇到 ADD 指令时，对执行结果添加约束 UGE(result，2^256)，即 result 大于等于 2^256，通过约束求解器求解 result，当 result 有解时发生溢出。表 3 给出整型溢出判断方案，当相应约束有解时，报告溢出漏洞。

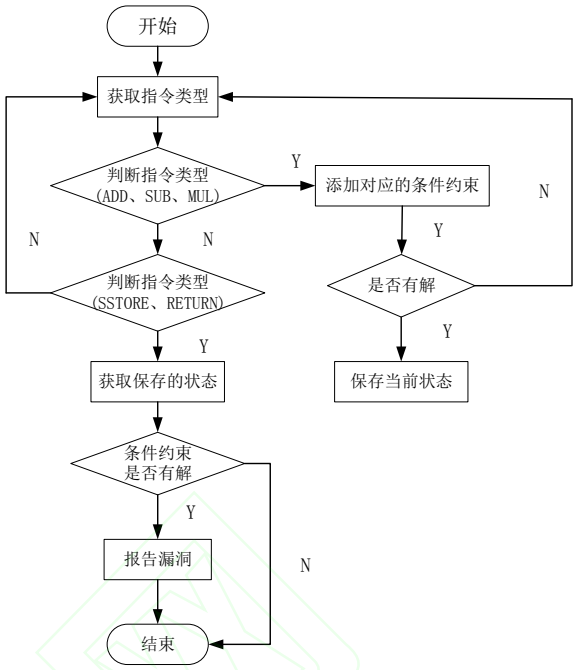

图 3 整型溢出检测的基本流程

Fig. 3 Basic flow of integer overflow detection

对程序 1 的测试关键信息如下：

Type: CALL (0)
From:normal0(0xe3ff20050f78738dd774ab17ddf84e78f2799598)
To:contract0(0x85c88d0f7118f76235f6acf81641f129dc767534)
Data:0x79c650680000000000000000000000003cecda360e75cce17a3adbee61b57373847cc565fffffff

权限控制漏洞发生主要原因是在设置权限时没有进行管理员权限检查。随机构建管理员地址(address_owner)创建合约，并随机构建一个地址(address_A)作为程序的输入，地址(address_B)作为程序的调用者。当程序的返回值不为 REVERT 时程序执行成功，此时 address_owner != address_B, 并且判断 Owner 的值发生变化，说明任意地址都可以修改合约的管理员权限，报告漏洞。对程序 3 的测试关键信息如下：

```
Type: CALL (0)
From:normal1(0x9ba625491215b0f3dc169f4f86d3022d39106c11)
To:contract0(0xbd82ab8120a86b7b426a230bb9daa97c57dfe6b4)
Return_data:0x0000000000000000000000000000000000000000000000000000000000000001 (*)
Function call:
setOwner(88859874119988220123807148980727697252260147508 9) -> RETURN
return: 1 (*)
```

对程序 3 添加管理员验证的测试关键信息如下：

```
Type: CALL (0)
From:normal1(0x628f813953395e1133edfbbed9dae7d7857529bd)
To:contract0(0xa2f03eb836874d657078d8fc72697a3c160ddab2)
Return_data: 0x
Function call:
setOwner(56268135909573005652993498801263902817662619 8973) -> REVERT
```

可以对比发现，当存在权限控制漏洞时，任意地址调用 setOwner 函数都可以成功返回。当对程序添加管理员验证后，非管理员地址调用 setOwner 函数会验证不通过，触发 REVERT 指令。

3.2.3Call 注入检测

Call 注入发生条件：

1. 地址可以被调用方控制
2. 地址可以被调用方控制

当遇到 call、callcode 和 delegatecall 指令时，检测地址是否可以有多个不同值，通过约束求解器解出的结果数如果大于 1，那么地址就可以被调用方控制。同样检测调用参数是否可以有多个不同值，如果有则表明调用参数可以被调用方控制，报告漏洞。

对程序 6 检测的关键信息：

```
possible_addresses:[1490158532180528465619443358847974843357075474970, 0, 1]
possible_func:[b'\x00\x00\x00\x00', b'\x01\x01\x01\x01', b'\x00\x02\x00\x02']
```

对地址和传入参数检测，经过约束求解器可以解出多个地址值和参数值，说明地址和参数可以被用户任意控制，报告 call 注入漏洞。

3.2.4重入漏洞检测

发生重入漏洞所需条件：

1. 函数调用者为合约账户；
2. gas 大于 2300（gas 不足 2300 时不够一个合约调用另一合约的手续费）；
3. Call 的调用地址可以为任何用户；
4. Call 操作执行成功；
5. Call 执行完后执行 SSTORE 操作。

利用路径所包含的信息来检查可重入性漏洞。在符号执行过程中遇到满足上述 1-4 条件路径时，保存执行的路径信息。当 Call 操作执行完后，如果有 SSTORE 操作，即执行向 storage 写入操作时，读取当前状态的执行信息，如果有信息则报告漏洞。因为被调用方有可能在完成调用之前重新执行调用,导致 SSTORE 操作未执行。对程序 7 执行关键信息如下：

```
Type: CALL (0)
From:attacker(0xcb23551b14667759e98f805c589caa76a78b7cc4)
To:contract0(0x4da25de5ba6b9ae0426e2060427e988fb60e5279)
Value: 490 (*)
Gas used: 3000000
Return_data: 0x
Function call:deposit() -> STOP (*)
Type: CALL (0)
From:attacker(0xcb23551b14667759e98f805c589caa76a78b7cc4)
To:contract0(0x4da25de5ba6b9ae0426e2060427e988fb60e5279)
Value: 0 (*)
Gas used: 3000000
Return_data: 0x
Function call:
withdrawAll(235) -> STOP (*)
tx.is_human:False
tx_result:STOP
tx.gas:2999431
Location:(407461482571921275407577332650820263379991131684, 375, 'Reentrancy bug', False)
```

发现在当前状态中，交易为合约调用(tx.is_human)，交易成功(tx_result),交易的 gas(tx.gas)大于 2300 时，保存了当前执行信息(location), 并在最终执行的写入操作中读出了当前信息并报告漏洞。

3.3 测试结果

本文从 Awesome-Buggy-ERC20-Tokens[18]项目中选取了 70 份含有漏洞的智能合约，对其进行了分析。最终检测结果如表 4 所示，正确警告约为 85%，未检测出漏洞约为 10%，不完整警告约为 5%。由于符号执行路径遍历效率低、String 类型符号化复杂、循环处理存在缺陷等问题，导致检索结果出现漏检率偏高，并且检索时间较长。

表 5 智能合约漏洞检测结果

Tab.5 smart contract vulnerability detection results

漏洞名称	数量	正确警告	未检测出漏洞	不完整警告
BatchTransfer-Overflow	5	5	0	0
Totalsupply-Overflow	20	13	3	4
Mint-Token-Overflow	10	8	2	0
Setowner-Anyone	5	5	0	0
CustomCall-Abuse	20	18	2	0
Re-Erntrancy	10	10	0	0

4 总结

智能合约是区块链技术普及应用的关键环节，对区块链的技术集成创新、交易模式创新起到了推动作用，有助于互联网发展的信用改造和价值传递。智能合约在以区块链为驱动的新一代互联网中的应用越来越广泛，所以其安全性需要高度关注。本文剖析了以太坊智能合约几种常见的安全性漏洞，提出了基于符号执行的漏洞检测实施方案，并进行了实验验证，取得了良好的检测效果。下一步我们将在符号执行路径搜索优化方面继续研究，提高合约检测效率和准确性。

参考文献

[1] 王化群,张帆,李甜,等.智能合约中的安全与隐私保护技术[J].南京邮电大学学报：自然科学版,2019,39(4):63-71.(WANG H Q,ZHANG F,LI T, et al. Security and privacy protection technologies in smart contract[J]. Journal of Nanjing University of Posts and Telecommunications,2019,39(4):63-71.)

[2] 付梦琳,吴礼发,洪征,冯文博.智能合约安全漏洞挖掘技术研究[J].计算机应用,2019,39(07):1959-1966.(Fu M L,Wu L F,Hong Z,Feng W B. Research on Smart Contracts Vulnerability Mining Technique[J]. Journal of Computer Applications,2019,39(07):1959-1966.)

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

[4] Grishchenko I, Maffei M, Schneidewind C. Foundations and tools for the static analysis of ethereum smart contracts[C]. ICCAV 2018: International Conference on Computer Aided Verification. Cham:Springer, 2018: 51-78.

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

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

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

[8] Krupp J, Rossow C. teether: Gnawing at ethereum to automatically exploit smart contracts[C]. 27th Security Symposium ({USENIX} Security 18). 2018: 1317-1333.

[9] Luu L, Chu D H, Olickel H, et al. Making smart contracts smarter[C]. 2016 ACM SIGSAC conference on computer and communications security. Vienna: ACM, 2016: 254-269.

[10] R M, Dehghantanha A, Choo K K R, et al. Empirical vulnerability analysis of automated smart contracts security testing on blockchains[C]. 2018 Annual International Conference on Computer Science and Software Engineering. Ontario:IBM Corp, 2018: 103-113.

[11] MOSSBERG M, IVNITSKIY Y, SMITH J P, et.al. trailofbits/manticore [EB/OL]. [2019.8.15] https://github.com/trailofbits/manticore

[12] Wright C, Serguieva A. Sustainable blockchain-enabled services: Smart contracts[C]. 2017 IEEE International Conference on Big Data. Boston:IEEE, 2017: 4255-4264.

[13] 范吉立,李晓华,聂铁铮,于戈.区块链系统中智能合约技术综述[J].计算机科学,2019, 46 (11): 1-10.(Fan J L,Li X H,Nie T Z,Yu G. Survey on Smart Contract based on Blockchain System[J].Computer Science,2019, 46 (11): 1-10.)

[14] 邱欣欣,马兆丰,徐明昆.以太坊智能合约安全漏洞分析及对策[J].信息安全与通信保密,2019(02):44-53.(Yue X X,Ma Z F,Xu M K. Ethereum Smart Contract Security Vulnerability Scenario Analysis[J]. China Information Security. 2019(02):44-53.)

[15] 牛伟纳,丁雪峰,刘智,张小松.基于符号执行的二进制代码漏洞发现[J].计算机科学,2013,40(10):119-121+138.(Niu W N,Ding X F, Liu Z,Zhang X S. Vulnerability Finding Using Symbolic Execution on Binary Programs[J].Computer Science,2013,40(10):119-121+138.)

[16] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing[C].PLDI 2005:Programming language design and implementation, Chicago:ACM,2005:213-223.

[17] Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C[C]. ESEC/FSE 2005: The 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, Lisbon:ACM, 2005:263-272.

[18] SECBIT. sec-bit/awesome-buggy-erc20-tokens[EB/OL]. [2019.8.15] https://github.com/sec-bit/awesome-buggy-erc20-tokens.

This work is partially supported by the Key Research and Development Program of Shandong Province(2017CXGC0701, 2019GNC106027)

ZHAO Wei, born in 1994, Postgraduate, not Member of China Computer Federation (CCF). His main research interests include blockchain technology.

ZHANG Wenyin, born in 1972, Ph. D.,Professor, is Member of China Computer Federation (CCF). His main research interests include image processing, information hiding and blockchain technology.

WANG Jiuru, born in 1983, Ph. D., Professor, is Member of China Computer Federation (CCF). His main research

interests include cyberspace security and blockchain technology.

WANG Haifeng, born in 1976, Ph. D., Professor, is Member of China Computer Federation (CCF). His main research interests include computer architecture, high performance cluster computing and .complex network analysis.

WU Chuankun, born in 1964, Ph. D.,Professor, is Member of China Computer Federation (CCF). His main research interests include information security, mobile network security and Internet of things security.