智能合约安全漏洞研究综述

倪远东, 张　超, 殷婷婷

清华大学 网络科学与网络空间研究院 北京 中国 100084

摘要　智能合约是一种基于区块链平台运行, 为缔约的多方提供安全可信赖能力的去中心化应用程序。智能合约在去中心化应用场景中扮演着重要的角色, 被广泛地应用于股权众筹、游戏、保险、物联网等多个领域, 但同时也面临着严重的安全风险。相比于普通程序而言, 智能合约的安全性不仅影响合约参与多方的公平性, 还影响合约所管理的庞大数字资产的安全性。因此, 对智能合约的安全性及相关安全漏洞开展研究显得尤为重要。本文系统分析了智能合约的特性及其带来的全新安全风险; 提出了智能合约安全的三层威胁模型, 即来自于高级语言、虚拟机、区块链三个层面的安全威胁; 并以世界上最大的智能合约平台——以太坊为例, 详细介绍了 15 类主要漏洞; 并总结了智能合约安全研究在漏洞方面的进展和挑战, 包括自动漏洞挖掘、自动漏洞利用和安全防御三个方面的研究内容; 最后, 本文对智能合约未来安全研究进行了展望, 提出了两个潜在的发展方向。

关键词　智能合约; 区块链; 安全; 漏洞

中图法分类号　TP301　DOI 号 10.19363/J.cnki.cn10-1380/tn.2020.05.07

A Survey of Smart Contract Vulnerability Research

NI Yuandong, ZHANG Chao, YIN Tingting

Institute for Network Science and Cyberspace, Tsinghua University, Beijing 100084, China

Abstract　Smart contract is a decentralized application that operates on a blockchain platform, providing secure and reliable capabilities to contract participants. Smart contracts play an important role in decentralized application scenarios. They are widely used in many fields, such as equity crowdfunding, games, insurance, and the Internet of Things, making them attractive to attackers. Compared to traditional programs, the security of smart contracts affects not only the fairness of contracts but also the safety of high volume digital assets on the blockchain managed by contracts. Therefore, analyzing the security of smart contracts and associated vulnerabilities is crucial. In this paper, we analyzed the characteristics of smart contracts and new security risks they bring. We propose a three-layer threat model, i.e., threats from high-level languages, virtual machines, and the blockchain, for characterizing smart contract security. We use the world's largest smart contract platform Ethereum as an example to illustrate 15 types of common vulnerabilities in smart contracts. We then summarize the main challenges and progress of smart contract security research on vulnerability, including automated vulnerability detection, automated exploit generation and mitigations for smart contracts. At the end of this paper, we highlight the future of smart contract security research, and proposed two potential research directions.

Key words　blockchain; smart contract; security; vulnerability

1　前言

智能合约是指一份可以在满足条件后自动履行承诺的合同。相比于传统的合约或者合同, 智能合约能够通过技术手段来强制保证: 在合同中的条件满足之后, 相应的合同条款会被强制自动地执行, 且整个过程不需要可信第三方的监督或者参与。智能合约这一概念最早由计算机学家和法学家尼克·萨博在 20 世纪 90 年代提出[1], 但是由于缺乏可信的执行环境, 智能合约这一概念一直没有得到广泛地应用。直到区块链技术的出现, 为智能合约提供了一个天然的去中心化、可信任且不可篡改的合约执行平台。智能合约这个概念现在也和区块链相绑定, 指的是一段可以在区块链平台上自动运行的代码, 用于描述并且自动执行一份合约。例如, 在一个公平的游戏合约中, 一旦参与者满足了合约中设置的获胜条件, 合约便会自动地向参与者账户发放奖励。整个合约执行的过程不需要依赖于对任何第三方的信任, 也不存在抵赖(不执行合约)或者欺骗(错误执行合约)的问题。

通讯作者: 张超, 博士, 副教授, Email:chaoz@tsinghua.edu.cn。

本课题得到自然科学基金(No.61772308, No.61972224, No.U1736209)项目资助。

收稿日期: 2020-01-22; 修改日期: 2020-03-15; 定稿日期: 2020-04-28

随着区块链技术的不断发展, 智能合约被广泛地应用于股权众筹[2-3]、游戏[4]、保险[5-6]、供应链[8-10]、物联网[11-12]等领域。对于区块链平台来说, 智能合约的运行极大提升了区块链的使用场景, 将区块链平台由简单的分布式账本系统扩展到了一个极为丰富的去中心化操作系统。相比于普通的程序而言, 智能合约更容易成为攻击者的目标。一方面, 智能合约通常可以用于管理区块链平台上的加密数字资产, 对智能合约的攻击可能会为攻击者带来更高的经济价值; 而更为重要的是, 引入智能合约的初衷在于借助区块链的特性来保证合约的可信赖, 而智能合约漏洞会使得合约出现非预期的行为, 从而可能使其变为一份“不平等合约”, 而失去了智能合约的意义。以太坊智能合约在诞生至今, 就曾发生过 The DAO 攻击[13]、Parity 钱包被盗[14]及多起因整数溢出漏洞造成的合约攻击事件[15], 其他诸如 EOS 等支持智能合约运行的区块链平台也曾被爆出大量的游戏合约被攻击事件[16], 不仅给用户造成了巨大的经济损失, 同时也使得智能合约的公平、可信赖属性受到挑战。一次又一次的攻击事件表明, 智能合约的安全形势十分严峻, 对于智能合约安全漏洞的研究也十分迫切, 越来越多的研究工作开始关注于智能合约的安全漏洞问题。

支持智能合约运行的区块链平台有很多, 其中最为主流的是以太坊区块链平台。比特币作为区块链技术诞生的代表, 可以执行简易的去中心化交易脚本, 但无法运行图灵完全的代码, 因此其并不被视为一个真正的智能合约平台[17]。而被视为区块链 2.0 的以太坊[18], 其最大的特点就是支持运行图灵完全的智能合约运行, 是首个支持智能合约运行、也是迄今为止最大的智能合约平台。随着以太坊的出现, 越来越多的区块链平台开始支持运行图灵完全的智能合约, 例如 EOS[19], NEO[20], Fabric[21], Libra[22], Zilliqa[23]等等。在众多支持智能合约运行的区块链系统中, 以太坊作为公认的最早实现智能合约的区块链平台, 已经成为目前世界上最大的区块链平台。近年来, 在智能合约的各类研究工作中, 大部分工作都集中于相对更加成熟的以太坊平台智能合约。本文对于智能合约的安全研究也将重点关注于以太坊平台上运行的智能合约。

以太坊智能合约本质上是一段程序, 由一门针对区块链运行环境而设计的程序语言编写, 运行在栈式的以太坊虚拟机中。以太坊区块链平台通过工作量证明(PoW)的共识机制[24]来维护了一个分布式、去中心化、可信赖的数据库, 而其上的智能合约则通过区块链交易来记录合约调用参数、通过区块链数据库来存储程序全局变量的方式来保证合约的可信赖。相比于普通程序而言, 智能合约有很多不同的特点。以太坊智能合约运行在区块链上, 其完整生命周期包括开发、编译、部署、调用和销毁, 其中多个步骤都是通过以太坊交易进行触发的。此外, 智能合约程序也存在 Gas 限制、委托调用、代码无法修改等特殊机制。智能合约在运行环境、生命周期和程序特性上与传统程序有较大的差异, 这些差异为智能合约了带来了全新的安全风险和攻击面。

本文以以太坊智能合约为例, 提出了一种全新的智能合约安全风险分析视角。通过对相关的智能合约特性及其对合约安全产生的影响进行分析, 将智能合约所遭受的 15 类重要安全漏洞系统性地划分为高级语言、虚拟机和区块链三个威胁层面, 对每个威胁层面的共性和单个漏洞进行了详细地分析。此外, 本文进一步分析了智能合约安全研究在安全漏洞方面的进展, 包括自动漏洞挖掘、自动漏洞利用和安全防御三方面, 总结了这些研究方向的挑战, 以及现有的研究工作在这些方面的最新进展。

本文在最后对智能合约未来的安全研究工作进行了展望。对于智能合约安全的研究, 一方面将继续着眼于如何改善现有的主流智能合约体系的安全性, 而另一方面则是通过吸收现有体系中安全设计的经验教训, 来设计更加适用于去中心化、安全公平的智能合约体系。智能合约这一技术还在不断发展, 相信未来两方面的研究都将有助于构建更加安全的智能合约。

2 智能合约特性

本章以以太坊智能合约为例, 介绍基于区块链的智能合约运行环境、智能合约的生命周期及与安全性研究有关的合约程序特性。本文认为, 智能合约在运行环境、生命周期和程序特性上与传统程序有较大的差异, 这些差异为智能合约了带来了全新的安全风险。

2.1 运行环境

智能合约这一概念尽管早在 20 世纪就被提出, 但其真正被应用还是依赖于区块链技术的发展。智能合约作为运行在区块链上的应用程序, 其通过去中心化、防篡改、可信赖的区块链来支撑其安全性。在以太坊中, 智能合约通过运行在以太坊节点上的以太坊虚拟机(Ethereum Virtual Machine, EVM)来完成智能合约程序的解释执行。与智能合约有关的各个组件在单个以太坊节点上的架构层级如图 1 所

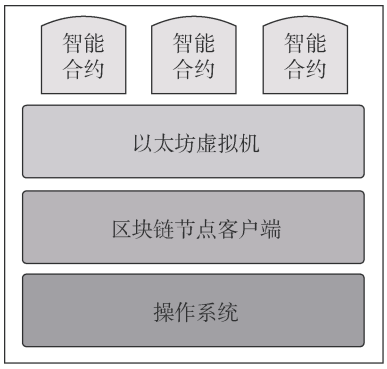

图 1　以太坊节点结构

Figure 1　Ethereum Node Architecture

示, 由底向上分别是操作系统、区块链节点客户端、以太坊虚拟机和智能合约程序。作为一个典型的去中心化、点对点的区块链网络, 以太坊通过成千上万个运行在不同主机上的以太坊客户端节点的互相通信来完成交易发送、交易确认、区块同步等机制, 从而推动区块数据的增长。以太坊客户端是一个运行在 Windows、Linux、Mac OSX 等通用操作系统上的软件, 常用的客户端有 Geth[25]、Parity[26]等, 这些客户端上层都运行着一个遵循太坊技术黄皮书规范的以太坊虚拟机[24]。当客户端需要对智能合约的调用交易进行确认或者校验时, 则会调用虚拟机来解释执行智能合约代码, 并校验计算结果是否正确。

以太坊虚拟机是智能合约执行的核心平台, 其实现规范定义在以太坊技术黄皮书中, 技术架构和运行模式如图 2 所示。以太坊虚拟机是一个无寄存器、基于栈式运行的虚拟机。以太坊虚拟机为智能合约程序提供了三种不用的存储空间, 分别为栈(Stack)、临时内存(Memory)和永久存储(Storage)。从使用场景来讲, Stack 和 Memory 是临时存储, 其存储结果仅在当前智能合约被调用期间有效, 调用结束后空间便会被回收, 仅作为程序运行时的临时存储区域使用; 而 Storage 的存储结果则是永久生效的, 其用于保存智能合约中需要被永久保存的重要全局变量, 是区块链状态的一部分。其中, Stack 和 Memory 的区别又在于, Stack 作为程序运行时的必要组件, 用于保存程序运行时的各种临时数据, 以 32 字节作为访问粒度; 而 Memory 则主要用于保存数组、字符串等较大的临时数据, 以单字节作为访问粒度, 更加灵活。智能合约代码存储在区块链状态中, 包含一个函数选择器和若干函数入口, 每个函数都有自己的参数列表。当虚拟机执行某个合约代码时, 会从交易数据中读取待执行的函数签名及其参数列表, 并启动合约代码的执行。虚拟机在合约的执行过程中, 便会使用 Stack、Memory 和 Storage 三种不同的存储空间。

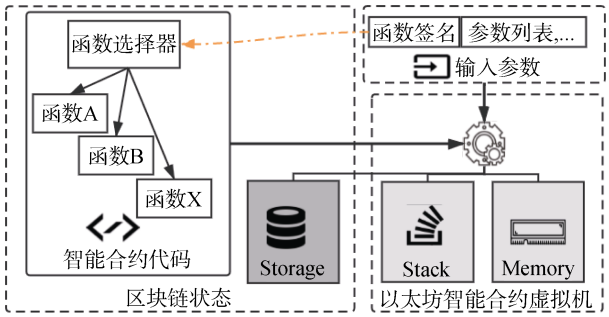

图 2　虚拟机架构

Figure 2　Virtual Machine Architecture

2.2　生命周期

一个典型的以太坊智能合约从开发到被使用、销毁的整个生命周期通常包含多个环节, 这些环节与普通程序的生命周期有很多区别。这些特有的智能合约生命周期的每个环节, 都可能带来不同的安全风险。

1) 开发。开发人员使用高级语言进行以太坊智能合约的开发, 可以使用的高级语言多达数十种, 包括 Solidity[27], Vyper[28], idris[29]等。其中, Solidity 是使用人数最多、最活跃的以太坊智能合约开发语言。大部分针对以太坊智能合约的源代码级别的程序分析、漏洞挖掘等工具也是面向 Solidity 来设计的。由于智能合约的高级语言尚在不断完善之中, 语言设计的不完善、用户编写程序的不规范也会引发合约的很多安全问题。

2) 编译。尽管可用于智能合约开发的高级语言有多种, 但是这些高级语言编写 的合约源代码都将被编译为统一规范的智能合约字节码(bytecode), 才能在以太坊虚拟机上运行。字节码的规则与虚拟机的规范相匹配, 同样按照以太坊技术黄皮书的规范进行设计。此外, 合约被编译之后, 还会生成相应的合约调用接口(Application Binary Interface, ABI), 该接口定义了合约所有可以被调用的外部函数及其参数列表。

3) 部署。编译之后的智能合约字节码需要被部署在以太坊区块链平台中。合约的部署通常由一笔合约部署交易来完成, 其中交易的数据(data)字段将被设置为合约部署字节码, 而交易的接收方被设置为空。矿工在进行交易打包时, 将会按照交易发送者的地址(address)和交易序列号(nonce)信息来生成一个新的地址, 并将合约的字节码部署到该地址。这个新生成的地址就是合约地址, 是合约的唯一识别, 之后对于该合约的所有操作都将使用这个地址进行。

4) 调用。合约被部署之后, 区块链上的用户可以通过合约地址对合约进行调用。对合约的调用可

以通过两种形式发起: 一种是由普通地址发起一笔合约调用交易, 这种调用称为交易调用(Transaction Call), 会在区块链的区块数据中留下直接的调用信息; 另一种是由某个合约发起的对另一个合约中函数的调用, 称为消息调用(Message Call), 这种调用的详细调用参数不会留在区块数据中。不同的调用方式有不同的使用场景, 合约中也经常会通过判断调用的发起者是普通用户还是合约, 来进行调用方式的限制以保障安全性。无论是哪种调用方式, 都需要按照合约规定的 ABI 信息来生成调用数据。然而, 由于两种调用方式的产生机制、调用者类型不同, 其也会产生不尽相同的调用后果。这要求合约设计时需要充分考虑两种不同调用的后果, 以避免出现安全漏洞。

5) 销毁。以太坊允许合约进行“自我销毁”, 不过并不是所有的合约都可以销毁, 而是需要开发者在进行合约编写时加入该功能。开发者在编写合约时, 可以规定合约在达到特定的条件时进行销毁。合约在执行过程中, 到达提前设置的条件, 便会进行自我销毁。需要注意的是, 由于区块链数据都是公开且不可篡改的, “销毁”只是意味着该合约在当前的区块状态(state)中被标记为删除, 且不能被后续调用, 但并不意味着合约代码和 Storage 存储被删除。相反, 通过遍历并执行历史上的以太坊交易, 任何被“销毁”的合约及其 Storage 存储都可以被恢复和查看。合约一旦销毁便不可再被调用, 因此销毁功能的调用必须进行足够的权限设置, 否则可能会被攻击者恶意销毁而产生拒绝服务攻击。

由于智能合约基于区块链运行, 其生命周期与普通程序并不相同。正是智能合约不同的生命周期和特性, 给智能合约带来了全新的安全威胁。而在一个智能合约生命周期的每个环节, 都有可能引入安全问题。

2.3 程序特性

为了适应于基于区块链交易运行、管理加密数字货币资产、防篡改等特性, 相比于普通程序, 智能合约程序本身有很多不同的特性。这些程序特性与智能合约面向的主要功能有关, 也为智能合约增加了与众不同的攻击面。

1) Gas 机制。以太坊节点在对涉及到合约调用的交易进行打包或者校验时, 都需要调用以太坊虚拟机来执行合约代码以得到最终的运算结果。如果恶意的攻击者发起了一个包含无限循环或者开销巨大的合约调用交易, 将导致矿工无法完成这笔交易的打包, 或者导致节点耗费大量的资源来执行合约程序。为了防止这种资源滥用情况的发生, 以太坊设计了 Gas 机制来为合约的执行计算费用。每一个以太坊字节码指令都根据其运算的复杂程度被标记了对应的需要消耗的 Gas 花费。合约调用方在发起一次合约调用时, 需要指定本次合约程序执行最高能花费的 Gas 数量, 并为这个最大数量先行付费。如果合约程序的执行开销超过了最大花费还没有停止, 以太坊虚拟机将会抛出一个 Out-Of-Gas 异常以停止合约执行。此时, 由于合约并未正常退出, 合约程序执行过程中对区块链状态的更改(Storage 变量更新和 ETH 转账)将会被回滚, 但是所消耗的 Gas 费用不会退回。Gas 机制保障了合约程序的可终止性, 但也可能被攻击者恶意地用于对合约发起拒绝服务攻击。

2) 异常传递机制。与普通程序一样, 智能合约程序允许进行函数调用, 因此也会形成一个函数调用栈, 以记录函数调用结束的返回地址。然而不同的是, 智能合约中的函数调用有两种形式, 一种是对于本合约或者父合约的内部函数的调用, 这种情况称为内部函数调用; 另一种是对于指定地址的外部合约函数的调用, 称为外部函数调用。两种函数调用在实现上有较大的差别。内部函数调用只需要在以太坊虚拟机执行时进行指令跳转, 而外部函数调用需要使用 CALL 指令向外部合约发送消息, 这种调用也称为低级别的调用。对于所有的低级别调用来说, 如果被调用函数执行过程中出错而抛出异常, 则异常并不会被沿着函数调用栈进行传递, 而是仅使用布尔类型的返回值来表示函数调用是否正常完成。由于智能合约对外部合约的函数调用、转账等在字节码层面都使用 CALL 指令来完成, 均属于低级别调用, 其与众不同的异常传递机制也引发了很多状态不一致性和其它安全问题。

3) 委托调用。智能合约中有一种特殊的使用 DELEGATECALL 指令进行的外部函数调用方式, 称为委托调用。委托调用属于外部函数调用的一种, 其与函数调用一样都体现了代码复用的优点, 但不同之处在于其外部函数中的指令在执行的过程中, 将会使用并改变函数调用者的上下文信息。委托调用的本质上是对当前合约函数注入外部代码, 以支持函数库机制的实现, 并能够弥补智能合约代码无法修改的弱点。然而, 一旦委托调用的目标地址被攻击者可控, 则攻击者可能获得在当前合约上任意代码执行的能力, 安全风险极大。

4) 合约代码无法修改。以太坊智能合约的部署阶段, 编译后的合约字节码将会被存储到以太坊账

户状态中。为了保证合约的安全可信, 以太坊合约一旦部署之后, 便无法再修改代码。代码无法修改的特点, 尽管保证了合约代码部署后的唯一性, 但是也为漏洞修补带来了困难。

5) 全局状态与调用序列。每个合约都有一个长期的 Storage 存储区域, 为合约存储可跨函数使用的全局变量状态。由于合约的多入口调用方式, 因此在不同函数内部的变量关系、约束结果会随着全局变量进行跨函数的传递, 最直接的体现特征就是特定功能或者漏洞的触发需要多笔交易组成的调用序列来完成。更进一步, 在传统的程序分析中, 函数内控制流信息和过程间的控制流信息都是十分重要的程序信息。而智能合约这一特点带来的深远影响在于, 智能合约程序仅仅提供了多个函数入口, 即只提供了函数内部的控制流信息, 而缺乏过程间的控制流信息。触发特定功能的更加全面的过程间控制流信息, 则需要用户通过一个调用序列来提供。这一特性为智能合约的高效率程序分析与漏洞挖掘带来了不少的挑战。

以太坊智能合约作为一种特殊的运行在区块链上的去中心化应用程序, 其在设计之初便被赋予了很多独特的程序特性。这些程序特性也是一把双刃剑, 一方面赋予了智能合约特有的安全可信赖的属性, 另一方面则为智能合约带来了前所未有的安全风险。

2.4 本章小结

以太坊智能合约是目前智能合约应用和研究的典范。以太坊智能合约运行在以太坊的栈式虚拟机中, 有三种不同类型的存储区域。一个完整的智能合约生命周期包括开发、编译、部署、调用和销毁等, 其中每个环节都可能引入安全问题。智能合约面向去中心化、安全可信赖的合约管理而设计, 具备 Gas 机制、特殊函数调用、代码无法修改、全局状态等特点, 这些特点面向特定的应用场景而设计, 但也为智能合约带来了新的安全威胁。

3 智能合约安全威胁

智能合约有很多区别于普通程序的特性, 这些特性也为智能合约引入了与众不同的攻击面, 并带来了新的安全威胁。智能合约的安全威胁主要来自于安全漏洞, 而基于智能合约的运行环境、生命周期和程序特性, 本文认为智能合约的安全漏洞主要来自于高级语言、虚拟机和区块链三个层面。本章将详细分析这三个层面共 15 类主要安全漏洞, 并介绍四个典型的因为安全漏洞而引发的安全事件。

3.1 智能合约漏洞

作为运行在区块链上的去中心化应用程序, 智能合约所面临的安全威胁与其运行环境有紧密的关联。智能合约由高级语言所编写, 之后被编译为字节码, 并由区块链交易驱动, 在以区块链作为存储基础的虚拟机上运行, 整个过程中都会面临不同的安全威胁, 并伴随着安全漏洞的发现。通过对智能合约的运行机制和已经被发现的漏洞进行梳理, 本文认为智能合约的安全威胁模型, 自顶向下主要为高级语言、虚拟机和区块链三个层面。本节对重要的以太坊智能合约漏洞按照上述模型的分类进行了介绍。详细的威胁模型和安全漏洞如表 1 所示。

表 1 威胁模型与安全漏洞

Table 1 Threat model and security vulnerabilities

威胁层面	安全漏洞	备注
高级语言层面	变量覆盖	高级语言层面引入, 与高级语言设计模式、用户程序编写等有关
	整数溢出	
	未校验返回值	
	任意地址写入	
	拒绝服务	
	资产冻结	
虚拟机层面	未初始化变量	虚拟机层面引入, 与虚拟机及其字节码设计规范、虚拟机实现有关
	影子变量	
	重入	
	代码注入	
	短地址攻击	
区块链层面	不一致性攻击	区块链层面引入, 主要与区块链本身的特性有关
	时间戳依赖	
	条件竞争	
	随机性不足	

3.1.1 高级语言层面

高级语言是开发者进行智能合约编写的工具。以太坊智能合约开发有多种可以使用的高级语言, 其中最为常用的是 Solidity 语言。高级语言层面为智能合约带来的安全威胁主要有两个原因, 一个是高

级语言自身设计的缺陷所引入的安全问题, 另一个则是开发者在编写高级语言过程中因为代码质量而引入的安全漏洞。

1) 变量覆盖。变量覆盖漏洞是一个典型的由高级语言设计缺陷所导致的安全漏洞, 其影响特定版本的 solidity 编写的智能合约。使用 Solidity 编写智能合约时, 其函数中的变量如果没有特殊的类型声明, 默认应为 Memory 类型的临时变量, 只会在函数被调用时生效, 函数返回后即被回收。然而, 在受影响版本的 Solidity 语言编写的智能合约函数中, 其默认声明的数组或者结构体类型变量会被编译器误用为 Storage 类型的变量。从而, 对这些变量的操作将导致对智能合约 Storage 存储区的非法覆盖。由于合约的 Storage 变量访问操作 Gas 费用高昂, 一般仅用于存储管理员地址、余额等重要信息, 这些变量一旦被覆盖, 将会导致合约运行出错。而恶意的攻击者更是可以利用这个漏洞来构造恶意的攻击载荷, 将特定的变量覆盖为指定的值, 从而实现管理员篡改、增加攻击账户余额等攻击。

2) 整数溢出。整数溢出是计算机程序设计语言的通用特性, 其主要是因为编程语言对于整数类型的存储空间是有特定长度限制的, 因此其所能表示的整数类型也有大小限制。一旦整数运算结果超过这个范围, 就会发生整数溢出。智能合约中的整数溢出问题, 主要是因为用户没有对运算结果进行安全地溢出检测而造成的。非预期的整数溢出将导致智能合约程序运行出错, 也可能被攻击者利用于绕过检查、篡改余额等重要数据。整数溢出是智能合约, 尤其是 ERC20 合约中非常常见的一类安全漏洞, 这类漏洞主要因为开发者编程不规范引入。3.2 节介绍了多个整数溢出攻击, 这些攻击带来了大量的经济损失, 也颠覆了合约的公平性。

3) 未校验返回值。未校验返回值漏洞由不规范的高级语言代码编写引入, 也与程序特性一节中介绍的以太坊智能合约中特殊的异常传递机制有关。对于低级别的函数调用, 虚拟机并不会将函数调用中抛出的异常状态沿着函数调用栈进行传递, 而是只会在返回值中表示是否发生了异常。因此, 开发者在进行合约代码开发时, 需要对低级别调用的返回值进行校验, 以确定该调用是否成功, 而不应期待调用失败的异常信息自动影响控制流。由于低级别调用本质是是对其它合约账户地址的一个“消息发送”, 因此虚拟机设计的该异常传递机制是合理的。而由于高级语言设计中没有显式告知这一特性, 若开发者缺少这个背景, 则极易在合约中引入未校验返回值漏洞。该漏洞会导致程序的执行控制流与预期相悖, 从而造成状态混乱。

4) 任意地址写入。任意地址写入是指合约中包含用户可控的对任意 Storage 地址写入数据的漏洞。Storage 是重要且开放的合约存储空间, 对于特定 Storage 地址的数据读写权限完全由合约开发者自行控制。合约中的重要 Storage 变量的修改需要设置严格的访问限制, 才能保证合约的安全性。Storage 的存储模型可以视为一个 Key-Value 的映射, 如果用户可以控制 Storage 写入时的 Key, 则可以对任意的 Storage 变量进行修改, 这是极其危险的行为。

5) 拒绝服务。拒绝服务是指合约无法按照预期设计完成响应的功能, 是一类通用的程序漏洞。智能合约中可能造成拒绝服务的原因有很多, 常见的有触发非预期的异常、达到区块 Gas 上限、意外自毁、硬编码错误的管理员地址等。这些原因通常都是因为用户在进行合约代码开发时不安全的代码编写规范所引入的。智能合约中的拒绝服务漏洞有多种类型, 根据拒绝服务漏洞的危害主体, 一般将引起以太坊网络阻塞的拒绝服务称为主动式拒绝服务, 而将合约开发者引入、引发合约无法正常运行的拒绝服务称为被动式拒绝服务[30]。本文所介绍的拒绝服务漏洞主要为被动式拒绝服务。和一般程序中的拒绝服务不同, 智能合约中的拒绝服务状态一旦被触发, 通常是不可逆的, 且无法被通过打补丁来修复。部分合约一旦受到拒绝服务攻击, 合约将会永久性地瘫痪, 再也无法被调用。

6) 资产冻结。智能合约的一个重要作用是管理区块链平台上的数字资产。以太坊智能合约管理通常通过以太币 ETH 作为数字资产, 很多合约可以通过与用户进行数字资产的交易来实现丰富的数字资产使用场景。相比于以太坊上的普通账户, 智能合约账户没有自己的私钥, 只能通过合约代码来管理账户中的数字资产。如果开发者在进行智能合约开发时, 仅设置了接收 ETH 的功能, 但没有任何允许 ETH 转出的操作, 或转出操作无法被触发, 将导致合约接收到的ETH资产被永久冻结, 无法再被使用。

7) 未初始化变量。使用高级语言进行智能合约开发时, 没有被初始化的 Storage 变量可能会指向未知的 Storage 存储内容, 如果对其直接进行访问将会引发程序意外。对于未初始化变量的处理是智能合约开发者的责任, 一个逻辑安全的程序应当避免对未初始化的变量进行直接的读取访问。

8) 影子变量。大多数智能合约高级语言是面向对象的, 允许合约类之间的继承。影子变量主要指子

合约中声明了与父合约中相同的 Storage 变量, 也可以指同一个合约内部的全局 Storage 变量和函数内部的局部变量重名。在复杂的程序中, 多个有关联的作用域中的相同命名变量极易引起合约编写、阅读和理解的逻辑问题, 从而引发合约漏洞。

3.1.2 虚拟机层面

虚拟机是编译后的智能合约字节码执行器。以太坊的虚拟机及其字节码的设计规范被定义在以太坊技术黄皮书中, 是各种以太坊客户端实现以太坊虚拟机的标准指导手册。虚拟机层面的安全威胁主要有两个方面, 一是以太坊黄皮书设计智能合约字节码规范和运行机制本身的一些缺陷, 二是不同的以太坊客户端在实现虚拟机的过程中, 因没有严格按照手册实现而引入的问题。

1) 重入。重入漏洞是以太坊智能合约一个典型的漏洞。重入漏洞是指合约在本该属于原子性事务的“修改 Storage 变量并转账”的操作中, 采用了先转账再修改 Storage 变量的顺序。如果转账的目标是一个带有恶意 fallback 函数的合约, 则可能会被恶意合约对受害合约发起递归调用, 从而破坏操作的原子性, 绕过检查以重复获得转账收益。例如, ERCToken 合约有一个典型的重入漏洞, 代码如下:

```
contract ERCToken{
...

function withdraw(uint amount) public returns (bool) {
        if (credit[msg.sender]>= amount) {
            msg.sender.call.value(amount)();
            credit[msg.sender]-=amount;
            emit Withdraw(msg.sender, amount);
            return true;
        }
        return false;
    }
}
```

ERCToken 合约的 withdraw 函数的功能为让用户提取资产, 当用户调用该函数时, 合约先检查用户余额是否大于提取的资金数, 若检查通过, 则会将请求提取的资产以 ETH 的形式转账给用户, 并在用户账户上扣除相应余额。由于以太坊智能合约的运行机制, 如果收到转账的地址是一个合约地址, 便会触发该地址的 fallback 函数, 该机制则可能被恶意的攻击者用于发起重入攻击。Attack 是一个攻击合约, 其代码如下所示:

```
contract Attack{
...

function hack() public {
        erctoken.withdraw(1);
    }

function() public payable { //fallback 函数
        erctoken.withdraw(1);
    }
}
```

攻击者只需要通过 hack 函数调用受害合约的 withdraw 函数, 受害合约 withdraw 函数中 mag.sender.call.value(amount)() 语句执行时, 会触发攻击合约中的 fallback 函数, 攻击者在该函数中再次发起对受害合约 withdraw 的调用, 则会在用户余额减少之前, 重复地发起递归调用, 直至 Gas 耗尽, 从而不断偷窃受害合约中的 ETH 资产。整个攻击的过程如图 3 所示, 攻击者通过构造一个第 2 步和第 3 步之间的循环, 来不断窃取受害合约中的 ETH。重入漏洞的出现需要满足多个条件, 即转账和 Storage 修改具有逻辑上的原子性绑定且转账在前, 以及使用 CALL 指令进行转账。重入漏洞虽然可以由开发者在高级语言层面修改操作顺序从而被避免, 但是其根源是以太坊虚拟机 CALL 指令调用的 Gas 机制、fallback 函数机制、允许低级别调用的递归访问等特性共同导致的。在 3.2 节中介绍的 The DAO 攻击即由重入漏洞引起, 并直接导致了以太坊的分叉。

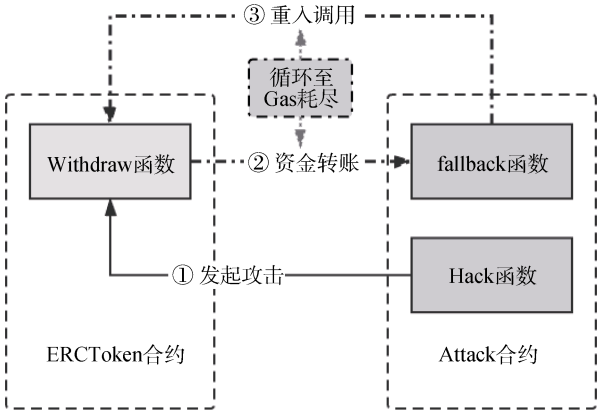

图 3 重入漏洞

Figure 3 Reentrancy vulnerability

2) 代码注入。代码注入漏洞是指智能合约可以被攻击者利用于注入任意的代码并执行, 从而修改合约中的重要 Storage 变量或者偷窃合约账户中的数字资产。代码注入的主要原因是由以太坊智能合约

设计中的委托调用, 这个机制我们在“程序特性”一节中已经有过介绍。委托调用所使用的DELEGATECALL 字节码, 允许合约在自己的上下文环境中执行一段其他合约的代码片段。如果外部合约的地址是由攻击者可以控制的, 攻击者便可以控制受害合约来执行一段恶意合约中的代码, 从而实现攻击行为。代码注入使得合约的行为可以被任意控制, 从而变得完全不安全不可信。

3) 短地址攻击。短地址攻击是指攻击者通过构造末尾为零的地址进行合约调用, 并在调用参数中故意将地址末尾的零舍去, 从而利用虚拟机对于数据的自动补全机制来将第二个参数进行移位放大。短地址攻击通常的发生场所在交易所, 若用户在交易所发起对合约进行转账的操作, 并使用了恶意构造的短地址作为目标。交易所如果没有对用户输入长度进行校验, 便会因为短地址漏洞而使得实际转账的金额被扩大若干倍, 从而造成大量的资金损失。短地址漏洞的根源在于虚拟机在读取合约调用输入时, 对长度不符合要求的字段进行了末尾自动补零的操作, 从而造成了数据的歧义和参数的移位扩大。

4) 不一致性攻击。不一致性攻击是指智能合约因虚拟机实现不一致从而导致的智能合约状态混乱。智能合约的可靠性依赖于各个节点对于智能合约的执行基于同样的 Storage 状态, 并在同样的状态与调用参数的情况下, 执行完全相同的功能。但是由于各种客户端对于虚拟机的实现并非完全一致, 可能会与以太坊黄皮书的设计出现偏差, 从而导致同样的一次合约调用在不同的区块链节点中产生了不一样的运行结果。由于区块链的节点之间只同步合约的调用交易信息, 并不会同步运行结果, 因此运行结果的差异非常隐蔽, 并会进一步导致合约执行的异常, 更有可能造成区块的分叉。

3.1.3 区块链层面

智能合约依靠区块链来提供去中心化、不可篡改和信任等特性, 区块链平台对智能合约的运行也有很多影响。区块链对于智能合约来说, 虽然是其安全可信任的根基, 但区块链本身的很多特性也会给智能合约带来安全风险。

1) 时间戳依赖。时间戳依赖是指智能合约在代码中使用严格的区块时间戳来进行重要的控制流决策, 从而引入的安全漏洞。区块时间戳是指当前的合约调用交易所属的区块被打包的时间戳。区块时间戳看似具有一定的偶然性, 但却是可以被矿工在一定的取值范围内操纵的。若精确的时间戳被当作合约中的一个重要决策参数, 虽然对于普通攻击者有一定的不可违抗性, 但是矿工身份的攻击者则可以通过构造取值范围内的恶意时间戳来轻易绕过合约中使用时间戳设计的限制。

2) 条件竞争。条件竞争漏洞是指智能合约中仅通过交易顺序来作为决策条件的程序逻辑所引起的漏洞。智能合约的调用是通过发起交易进行的, 然而交易的发起时间和交易被确认从而使合约调用生效的时间并不具有严格相关性。例如, 某个悬赏合约承诺给第一个提交答案的账户基于奖励, 但是首个提交正确答案的用户并不一定能真正拿到奖励。原因在于这笔首个提交正确答案的交易, 在用户发起交易之后, 这笔交易便可以被网络中的部分节点所观察到, 但此时离交易被打包尚有一段时间。由于区块链中矿工通常打包手续费更高的交易, 因此攻击者可以快速发起提交同样答案的交易, 并通过提高手续费的方式以让自己的交易被优先打包。条件竞争漏洞的根源在于区块链的交易打包和手续费机制, 使得交易的确认顺序与发起顺序并不具有相关性。对这一性质的误用将导致条件竞争漏洞, 著名的 ERC20 合约标准中的 Approve 函数就曾存在这个漏洞[31]。

3) 随机性不足。随机性不足是指智能合约中误用了很多与区块链有关的变量作为随机源, 但是这样的做法将导致随机数可被预测。智能合约中常使用区块编号、区块时间戳、区块打包矿工、区块哈希等与区块链中区块打包有关的变量。这些变量看似对于普通的用户是随机且不可控制的, 但是若攻击者通过驱使攻击合约来调用受害合约的方式进行合约调用, 由于对于攻击合约和受害合约的调用都来自于攻击者发起的同一笔交易, 自然处在同一个区块中, 因此攻击合约中可以读取到和受害合约中使用的全部区块变量, 从而预测受害合约中的随机数。一旦随机数被预测, 那基于随机数的随机性而产生的各项安全保障将荡然无存。随机性不足的主要原因在于区块链本身特殊的共识打包机制, 以及区块链数据的公开性。仅依靠区块链本身的一些信息作为随机源, 智能合约很难产生良好随机性的随机数。

3.2 典型安全事件

智能合约旨在依靠区块链提供的去中心化、防篡改等特性, 提供安全平等可信任的可编程合同, 然而智能合约中的漏洞将可能使智能合约的这些特性难以实现。智能合约中的安全漏洞一旦被攻击者所利用于发起攻击, 一方面可能为使用智能合约进行数字资产管理的用户带来巨大的经济损失, 另一方面则可能破坏智能合约参与多方的公平性, 使智能合约变得不可被信赖。以太坊智能合约在面世以

来, 已经遭遇了大量的攻击事件, 在造成了巨额经济损失的同时, 也破坏了合约的公平性。在智能合约的发展历史上, 发生过难以计数的安全事件, 其中影响较大的安全事件及其相关的安全漏洞如表 2 所示。这些安全事件虽然由不同层面的智能合约漏洞引发, 但都带来了巨大的经济损失。接下来将介绍这几大典型的安全事件。

表 2 典型安全事件及相关漏洞

Table 2 Typical Security Events and associated vulnerabilities

安全事件	安全漏洞	威胁层面
The DAO 被攻击事件	重入	虚拟机层面
Parity 钱包被攻击事件	代码注入	虚拟机层面
BEC/SMT 整数溢出事件	整数溢出	高级语言层面
KotET 合约拒绝服务	拒绝服务	高级语言层面

1) The DAO 被攻击事件。2016 年 4 月, 部分区块链开发者利用以太坊平台成立了一个名叫 The DAO 的去中心化自治组织。该组织是一个去中心化的风投基金, 其通过以太坊上的智能合约来众筹募集资金, 并将募集后的资金用于项目投资。所有的众筹参与人都将按照自己的出资份额来分配投票权, 以对投资项目进行表决[32]。整个环节中, 接收资金的形式为以太坊平台上的加密数字货币以太币, 并使用智能合约来进行全流程的管理, 以确保资金使用的公开透明可信任。The DAO 项目作为当时以太坊诞生以来最成功的区块链项目, 在接下来一个多月的时间里成功募集了超过一亿六千万美元。而就在 2016 年 6 月, The DAO 智能合约中存在的重入漏洞被黑客发现并被用于发起攻击, 致使该组织损失了超过 5000 万美元[13]。该事件是以太坊诞生以来最轰动的安全事件, 甚至进一步导致了以太坊的分叉。随着 The DAO 攻击的出现, 智能合约安全也逐渐受到更多的关注。

2) Parity 钱包被攻击事件。Parity 是以太坊上广受欢迎的一个多重签名钱包, 其提供了一些公用的智能合约调用库, 用于为其他智能合约开发者提供便捷地多重签名钱包工具。2017 年 7 月, 黑客利用了 Pairty 公共合约库中的一个代码注入漏洞, 盗取了超过 3000 万美元的以太币[33]。随后 Parity 官方在对该合约漏洞的修复过程中, 却又再次引入了一个新的漏洞。2017 年 11 月, 黑客利用新引入的漏洞对合约再次发起攻击, 导致了大约 2.3 亿美元的以太币被永久冻结, 给使用该公共合约库的开发者带来了巨大的经济损失[34]。

3) BEC/SMT 整数溢出事件。BEC 和 SMT 是以太坊中的两个符合 ERC20 标准的股权众筹合约。2018 年 4 月, 这两个合约相继被发现存在整数溢出漏洞[15]。黑客利用漏洞对这两个合约发起攻击, 可以凭空造出至多 2^256 量级的天量 Token, 并在交易所大量抛售, 从而导致其市场价值跌破归零。随后越来越多运行中的合约被发现存在整数溢出漏洞。更可怕的是, 由于智能合约代码一旦上链便无法被更改的特性, 导致这些漏洞即便被发现, 也难以被及时修补。

4) KotET 合约拒绝服务。KotET 合约是一个多方参与的游戏合约, 游戏允许胜利的玩家通过使用以太币来从当前的“国王”玩家手中购买王位, 以成为新的“国王”。该合约因为存在拒绝服务漏洞从而被攻击者所利用[35]。攻击者利用该漏洞, 使得自己编写的恶意合约账户成为“国王”, 该合约包含一个复杂的回退函数, 使得任何向其转账的合约调用都会执行失败, 从而让 KotET 的其他玩家无法购买王位, 而让自己成为永久的“国王”。这个攻击也体现了, 智能合约漏洞一旦被利用, 将使得智能合约应用的公平性不复存在。

以太坊作为最大的智能合约平台, 其上的智能合约管理着大量的加密数字货币资产, 也极大增加了其遭受攻击的风险。除了上述介绍的这几大典型的安全事件之外, 还有 Fomo3D 游戏合约被攻击[36]、多个 EOS 游戏合约被攻击等诸多的真实攻击发生[16], 可以说面对智能合约的攻击愈演愈烈, 从来没有停下脚本。因此, 面向智能合约的安全研究, 作为对抗黑客攻击的重要手段, 也显得尤为重要。

3.3 本章小结

本章介绍了引发智能合约安全漏洞的三个层面及 15 类漏洞, 并介绍了 4 个典型的智能合约安全事件。从智能合约的架构来看, 智能合约遭受的安全威胁可能来自于高级语言、虚拟机和区块链三个层面。而智能合约存在的漏洞通常情况下是多个层面安全威胁互相影响的结果, 我们按照主要的影响因素, 对漏洞所处的主要威胁层面进行了分类, 并详细介绍了每一类漏洞的原理和危害。智能合约漏洞一旦被利用, 即可能为合约的参与方带来巨额经济损失, 也会影响合约参与多方的公平性。

4 智能合约自动化漏洞挖掘

自动化漏洞挖掘是软件漏洞挖掘的重要研究领域, 采取的主要方法有模糊测试、符号执行、形式化验证、污点分析等技术。智能合约作为一种新兴的

去中心化应用程序, 与传统程序在运行环境、生命周期、程序特性上有较多的不同, 这也对智能合约的自动化漏洞挖掘提出了新的挑战。现有的大量研究工作关注于如何在智能合约上应用现有的技术来实现更好的自动化漏洞挖掘效果。我们对目前比较有代表性的研究工作和漏洞挖掘工具进行了整理并按照主要类型进行分类, 如表 3 所示。此外, 由于不同的工具会报告智能合约中不同等级的漏洞, 且在定义上会有所不同, 因此表格中所统计的漏洞类型数量以本文 3.1 节中所总结的 15 类智能合约漏洞为主。

表 3　漏洞挖掘研究工作列表

Table 3　Vulnerability Detection Research List

主要技术	研究工作	辅助技术	分析类型	序列分析	漏洞类型	研究进展总结
模糊测试	Echidna	-	源代码	×	-	模糊测试可以被用于有效地挖掘智能合约安全漏洞, 但还面临着无法自动化测试无源码或无调用接口信息的合约、调用序列的有效性差、漏洞检测精确度低等挑战。
	ContractFuzzer	-	字节码	×	6	
	ILF	符号执行/深度学习	字节码	√	4	
符号执行	Harvey	程序分析	源代码	√	4	符号执行是目前智能合约漏洞挖掘的主流方案, 但分析多层深度的调用序列时状态空间爆炸的问题难以解决。
	Oyente	-	字节码	×	3	
	Osiris	污点分析	字节码	×	1	
	Mythril	污点分析	字节码	√	6	
形式化验证	Manticore	-	源代码	√	-	形式化验证通常被用于检测合约是否满足程序设计人员预定义的安全属性, 但目前的研究工作大多自动化程度相对较低, 且检测出的漏洞并不一定存在可达的执行路径。
	Securify	-	源代码	√	3	
	ZEUS	-	源代码	√	4	
	VerX	符号执行	源代码	√	-	
其他	SASC	程序分析	源代码	×	3	利用程序分析等其他技术的方案通常开销较小, 但是检测误报较高; 融合多种技术的漏洞检测方案也是一个研究趋势。
	SmartCheck	程序分析	源代码	×	5	
	Slither	程序分析	源代码	×	7	

本章后续的小节中, 将介绍主流的智能合约自动化漏洞挖掘技术, 及其相关研究工作的研究现状。

4.1　模糊测试

模糊测试是一种较为高效的软件分析技术。其核心思想是为程序提供大量测试样例, 在程序执行过程中监控程序异常行为, 以发现程序漏洞[37]。测试用例的生成是模糊测试的关键环节之一, 通常来讲可以有两种方式, 基于生成和基于变异。基于生成的模糊测试适用于待测输入格式较为明确, 或格式要求较为严格的情况, 如文件处理系统或用协议进行通讯的程序, 测试器根据输入格式生成测试用例, 以提高有效输入的比例。基于变异的模糊测试工具则在原始输入的基础上根据程序反馈对种子进行变异, 变异方法包括比特翻转、增减、拷贝等。

模糊测试可以被分为黑盒、灰盒、白盒三种模式。其中, 黑盒模糊测试不对程序内部结构进行分析, 而是通过生成随机输入触发程序的缺陷。白盒模糊测试则利用符号执行等程序分析技术对程序结构进行分析, 以提高覆盖率和漏洞挖掘能力。灰盒模糊测试不对程序进行分析但会根据程序反馈调整输入。

模糊测试这一技术被广泛地应用于传统程序的漏洞挖掘, 且被证明十分有效。AFL 是一个非常流行的针对运行在 Linux 平台上的 C/C++应用程序的模糊测试工具, 其主要采取基于覆盖率反馈来指导种子变异的模糊测试方案, 且已经被用于挖掘出至少数百个流行应用程序、公共函数库和操作系统内核

的内存破坏漏洞[38]。研究工作[39-43]等在此基础上更进一步, 研究如何在种子选择策略、变异策略、路径记录策略等方面进行优化以提高漏洞挖掘效率, 都取得了非常不错的实验效果。这些研究工作表明, 模糊测试是一种非常有效的漏洞挖掘方法。

相比于传统应用程序来说, 智能合约有很多不同的特点, 这些特点为面向智能合约的模糊测试带来了全新的挑战。首先, 智能合约全局状态与调用序列的特性, 导致模糊测试中生成有效的测试用例变得极为困难。传统应用程序在执行测试用例时, 针对不同的测试用例的执行结果互相独立, 不存在关联。而智能合约的特殊性质, 导致智能合约程序对于不同的测试用例的执行状态会通过全局 Storage 存储进行传递, 从而互相影响, 甚至大多数的功能需要特定的调用序列才可以完成。面向于传统程序的模糊测试方案, 在测试用例生成、覆盖率反馈等方面仅仅考虑单测试用例, 因此并不适用于智能合约。其次, 智能合约基于虚拟机运行, 其漏洞的形成原因与传统程序有较大的不同, 导致智能合约的漏洞检测更加困难。传统应用程序中的常见漏洞类型, 包括栈溢出、堆溢出、空指针引用等, 大多数都为内存破坏型漏洞, 其最大的特点是程序执行时指令指针寄存器可能被攻击者所控制。针对传统漏洞的模糊测试工具通常旨在发现这一类的内存破坏漏洞, 并通过将指令指针寄存器的值修改为一个非法地址以触发程序崩溃, 从而进行漏洞检测。因此, 程序崩溃是面向传统应用程序的模糊方案中的典型漏洞检测特征。然而智能合约中的漏洞有较大的差别。3.1 节中介绍的智能合约十五大漏洞, 既不会产生程序的崩溃, 也没有很多共同的特征用于漏洞检测。这些漏洞的产生根源可能来自于区块链、虚拟机和高级语言等不同的层面, 且彼此之间也有较多的差异, 这为智能合约的漏洞检测带来了很大的挑战。面向智能合约漏洞挖掘的模糊测试方案, 在传统模糊测试方案的基础上, 大多数旨在解决上述的两个挑战, 以更好地挖掘智能合约中的漏洞。

Echidna 是最早开源的智能合约模糊测试方案之一, 由安全研究组织 Trail of Bits 在其博客上发布[44]。Echidna 提供了一个完善的以太坊智能合约模糊测试框架, 其可以对智能合约源代码进行分析和模拟执行, 并生成符合合约调用规范的随机的交易数据来对合约进行模糊测试。Echidna 引入了覆盖率信息来检测模糊测试的执行效率, 但并没有深入探讨更加有效的种子生成策略。此外, Echidna 并没有提供通用的漏洞检测手段, 而是需要测试人员自行在合约源代码中增加特定的漏洞检测代码, 并通过返回值状态来指示是否发生漏洞。Echidna 是一个完善的智能合约工业化模糊测试框架, 但并没有解决智能合约模糊测试的两大主要挑战。

ContractFuzzer[45]是早期的智能合约模糊测试研究方案之一, 主要解决了智能合约的漏洞检测问题。ContractFuzzer 基于以太坊 Go 语言版本的客户端 Geth 进行修改, 通过记录智能合约执行时的指令日志来进行离线的漏洞检测。其定义了 Gas 不足、异常传递、重入、时间戳依赖、交易序列依赖、代码注入、资产冻结等七种漏洞的检测模型, 并通过指令日志来检测当前的合约执行是否触发了漏洞。在输入生成方面, ContractFuzzer 主要通过随机生成调用参数、交易金额、交易发送地址的方式来生成随机交易。此外, 其针对重入漏洞的触发, 专门设计了一种独特的攻击代理合约, 并通过该攻击代理合约来调用被测试合约的方式, 来尝试触发被测试合约中的重入漏洞。

ILF[46]是一个基于神经网络的智能合约模糊测试方案, 其致力于在智能合约的模糊测试中生成更好的测试用例及交易序列。由于智能合约具备全局状态, 因此针对单个函数的测试用例进行覆盖率引导反馈很难提高全局的覆盖率, 因为部分函数中的约束条件是由全局状态变量引进的, 这些函数变量的值可能依赖于其他的函数进行修改。因此要想提高全局的程序覆盖率, 实现更好的漏洞挖掘效果, 就需要生成合理的调用序列。在解决全局约束的问题上, 符号执行可以直接求解出约束的解空间状态, 相比于模糊测试的效果更好。正因如此, ILF 的解决方案是, 首先借助符号执行引擎来产生大量优秀的调用序列, 再通过神经网络来学习这些优秀的调用序列的特征, 来指导模糊测试引擎生成优秀的调度策略。ILF 的架构如图 4 所示, 其首先选取部分的智能合约程序, 并使用符号执行引擎工具对这写合约进行多次的符号执行, 以产生出覆盖率足够高的调用序列。接下来, ILF 使用神经网络对这些调用序列特征进行训练, 并生成可以模仿该序列生成规则的模型以指导模糊测试。最后, 使用这个训练好的模型来提供模糊测试策略, 即可以对任意的未学习过的合约进行高效的模糊测试, 生成更好的调用序列。ILF 的测试效果表明, 训练好的神经网络模型产生的交易序列对于程序覆盖率的提升效果, 甚至超过了符号执行引擎。ILF 方案背后重要的逻辑在于, 大多数智能合约都遵循相似的接口规范和开发标准, 因此不同合约之间接口或者代码逻辑的相似性较高。

Harvey[47]是一个尚未正式发表的智能合约模糊测试研究工作, 其主要关注于如何生成更加有效的

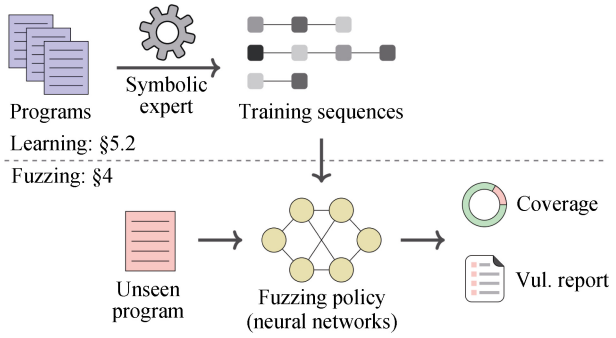

图 4 ILF 系统架构

Figure 4 ILF Architecture

输入和调用序列。Harvey 通过对在智能合约程序中条件判断语句之前进行插桩的方式, 来测量合约的每一次输入是否有利于产生新的路径, 即判断输入中相关的参数是否有利于解决当前的约束并进入到新的分支中, 从而使生成的输入更加有利于解决约束并进入新的路径。此外, Harvey 还尝试通过不同函数间对全局变量的读写依赖关系进行来生成简单的调用序列, 以提高序列对程序覆盖率的影响。

4.2 符号执行

符号执行是一种传统的自动化漏洞挖掘技术, 目前也被广泛用于智能合约的漏洞挖掘。符号执行引擎为目标代码提供符号化的虚拟运行环境, 将程序所需的外部输入抽象为取值不固定的符号值, 并通过不断求解路径约束, 来尽可能的探索程序分支。符号执行的主要思想是把程序执行过程中不确定的输入转换为符号值, 以推动程序执行与分析我们以图 5 为例, 对符号执行的基本流程进行解释。图 5 对应的示例代码如下:

```
contract sample{
    uint g_var;

    function foo1(uint m, uint n) public{
        if(m > n){
            g_var = m;
        }else{
            g_var = n;
        }
    }

    function foo2(uint x) public{
        if(g_var < x){
            g_var = x;
        }
    }
}
```

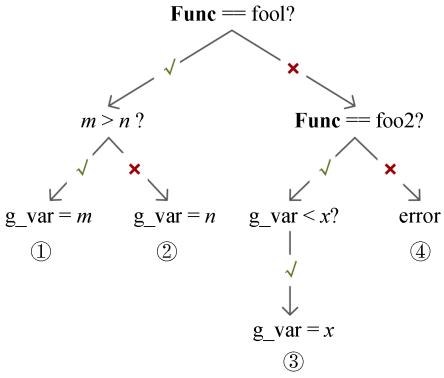

图 5 符号执行示例

Figure 5 Symbolic Execution Sample

无论是智能合约还是传统平台中的程序, 都可以被抽象为一棵执行树。在正常的执行流程中, 由于程序输入为定值, 每个条件判断都可以得到确定的答案, 因此仅有一条分支被探索。而在符号执行中, 输入值为不定的符号变量, 当遇到条件判断时, 符号执行引擎会利用约束求解器对包含符号变量的表达式进行求解。对于所有有解的分支, 符号执行都会进行分析, 并记录路径中的约束。在上图中, 合约调用的输入有两个, 一个是待调用的函数, 一个是函数的输入。对于图 4 中的路径 1 来说, 当程序执行到叶子节点时, 该路径的约束有两个: [*Func* == *foo1*, $m > n$], m 和 n 皆为符号化的输入值。对于符号执行来说, 智能合约与传统程序的差异主要在于合约中全局变量的取值是不确定的。因此通常情况下除了输入值, 智能合约的全局变量也需要被处理为符号值。

相比于模糊测试和静态分析, 符号执行能够对程序进行更精确和全面的分析。符号执行过程中收集到的丰富信息能够服务于漏洞识别。然而受分支、循环等结构的影响, 符号执行常常面临路径爆炸等难题, 在进行大型程序分析时, 会产生较大的时间开销。相比于传统应用程序来说, 智能合约代码量更小, 其路径数量更少, 长度更短, 更加适用于符号执行进行分析。然而, 由于智能合约的约束在不同的路径之间进行传递, 因此面向传统程序的符号执行思路在智能合约中仅能收集单个合约路径上的约束, 而非全局约束。这也为智能合约的符号执行工作带来了挑战。

Oyente[48]是最早关注到自动化合约漏洞挖掘的工作之一, 并为其他工作提供了一个实现较为精简的合约符号执行引擎。Oyente 以智能合约字节码作为输入, 包含 4 个核心组件, 控制流图生成器(CFG Builder), 探索器(Explorer), 分析器(Core Analysis)和验证器(Validator)。控制流图生成器对合

约进行预分析, 为合约构建基本的控制流图, 以基本块为节点, 跳转关系为边。然而部分跳转关系并不能由生成器完全确定。因此, 探索器会对智能合约进行符号执行, 并在执行过程中将这些信息补齐。探索器承担着收集合约信息的重要责任, 它本质上是一个循环, 依次执行合约控制流图中各个基本块的代码。它利用 Z3 求解器对合约中的条件跳转进行求解, 探索器根据求解结果决定对哪个分支进行分析, 当条件跳转的两个分支条件都有解时, 两个分支都会被探索。分析器是 Oyente 的另一个重要组件, 用于根据探索器收集的信息, 识别合约漏洞。Oyente 在论文中介绍了对条件竞争、时间戳依赖、未校验返回值以及重入漏洞等四种合约漏洞的检测方案, 并在开源代码中进一步补充了整数上溢、整数下溢、调用栈溢出等常见漏洞的检测代码。最终, oyente 增加了一个验证器, 用于过滤分析器所产生的误报。整体来说, Oyente 是一个字节码层面的合约漏洞挖掘工具, 在符号执行的过程中对程序控制流图进行动态探索, 并通过路径约束、变量来源等信息对合约漏洞进行检测。尽管 Oyente 的部分检测方案并不完善, 涉及的漏洞也不够全面, 但依旧作为开创性的工作, 为后续研究提供了很好的支持。

Osiris[49]利用符号执行技术对合约中的整数类漏洞进行检测, 包括整数溢出、类型转换导致的整数截断和符号错误等问题。以整数溢出为代表, 这类漏洞在合约中广泛存在且危害较大。而算数运算指令在合约中被大量使用, 除了用户主动定义的算数操作, 这类指令还服务于参数传递、数组索引等场景。如果仅关注算数运算操作本身, 可能会导致带来大量误报和巨大检测开销。此外, 并非所有的整数溢出漏洞都会带来危害。因此, 想要对此类漏洞进行准确的检测, 应当对溢出结果进行跟踪, 仅有直接或间接污染了合约全局变量, 或影响程序控制流的溢出, 才是真正有害的。基于这样的考虑, Osiris 通过污点分析技术, 对操作数来源和运算结果的传递方向进行标记, 过滤掉了无害溢出操作, 相比于其他工具在误报上有了明显的改善。在代码实现上, Osiris 在 Oyente 的基础上进行开发, 并通过维护一个了影子栈的方式进行污点传播。

Mythril[50]是由 Consensys 公司开发的符号执行引擎, 可以对各类基于 EVM 字节码的合约进行分析。Mythril 提供了丰富的扩展接口, 开发者可以在 Mythril 的基础上编写自定义的漏洞检测逻辑。目前 Mythril 官方提供了整数溢出、任意地址写入、时间戳依赖等 14 种漏洞检测功能。和 Oyente 与 Osiris 不同, Mythril 通过多次符号执行的方式模拟现实中合约被多次调用的情况(默认为 2 轮)。这种方法避免了将全局变量初始化为任意符号值带来的误差, 真实刻画了合约执行的实际情况。然而每轮符号执行都需要在上一轮符号执行所产生的所有分支状态上继续进行, 导致需要探索的路径数量指数级增长, 带来了较大的时间开销。Maian[51] 在处理需要多笔交易才能触发的深路径时也采用了相似的思路。

Manticore [52]是一种智能合约动态符号执行引擎。它通过将变量值具体化提高分析效率和和代码覆盖率。Manticore 将合约的执行过程抽象为: 就绪(Ready)、忙碌(Busy)、终止(Terminated)三个状态, 三个状态之间可以进行转换。当符号变量需要被转换为一个或多个具体值时, 就会有一个相应的就绪状态被创建并处理, 在程序退出或发生异常时终止。与其他工具针对单个合约进行分析不同, Manticore 可以支持对多个合约进行分析, 相当于对以太坊世界进行了符号化。此外, Manticore 也支持对其他二进制程序的符号执行。

4.3 形式化验证

形式化验证技术是一种有效的验证程序是否符合预期的设计属性和安全规范的技术。形式化, 是指通过严谨可验证的描述语言或逻辑来描述程序, 以便对其进行严格的数学推理和验证。形式化验证技术通常使用形式化的描述语言来描述一个系统的属性和特点, 为其构造出形式化规范, 再运用严格的数学逻辑证明来对其实际的运行时行为进行推理, 以检测其是否符合预期设计的功能要求, 从而来比较系统实现和系统设计之间的差异性。

形式化验证技术中, 对程序指令和逻辑的形式化描述是验证的前提。对于以太坊智能合约的指令设计和描述中, 最为严谨的是以太坊社区推出的技术黄皮书[24]。技术黄皮书中使用自然语言对智能合约指令设计进行了较为严格的定义和描述, 但其并不是可以被直接应用于形式化验证的形式化描述语言, 无法被机器用于进一步描述程序逻辑, 从而进行推理或者证明。Hirai 等人[53-54]的研究工作是早期尝试对以太坊智能合约指令进行形式化描述的研究工作之一。其利用 Isabelle/HOL 对以以太坊虚拟机中的指令语义进行形式化描述, 以将其用于手动地证明某个程序的安全性。但是其仅仅支持以太坊中的部分指令, 并且无法描述指令完整的语义。例如, 如果合约通过 CALL 指令执行了一次外部调用, 则将会被认为是执行了任意的代码, 可以对任意的 Storage 和全局状态进行更改, 这与实际的语义有较

大的差别, 会影响验证的结果。此外, 还有多个研究工作致力于构建出一个形式化描述的以太坊虚拟机, 以便对智能合约指令进行形式化的表述。KEVM[55]使用K框架来形式化描述智能合约的语义。K框架[56]是一个基于可达性逻辑且高级语言无关的验证框架, KEVM 尝试使用其来实现智能合约上一些语义相关的分析工具, 诸如 Gas 分析工具、语义调试器、基于可达性逻辑的程序验证器等等, 但是很难实现完整的程序分析, 且过多人工工作的投入也使得其不利于扩展到大型的程序分析中。Bhargavan 等人[57]的工作尝试将以太坊指令翻译为 F*语言。F*[58]是一种具有交互式证明功能的函数式编程语言, 可以进行程序验证。但其工作也仅仅支持少部分的字节码。

对智能合约指令的形式化描述只是形式化验证的第一步, 一些研究工作开始关注在形式化描述的基础上对智能合约开展安全性证明。Grishchenko 等人[59]的研究工作在使用 F*语言翻译大部分以太坊智能合约指令的基础上, 定义了外部调用完整性、原子性、可变账户状态独立性和交易环境独立性等四个较为通用的安全属性, 并对其进行了证明。该工作虽然引入了安全属性的证明, 但是证明过程仍然需要较多的人工投入。

ZEUS[60]是一个定义支持五种安全漏洞验证的自动化智能合约形式化验证工具。ZEUS 将 Solidity 源代码翻译为 LLVM 中间语言, 并在其上使用 XACML 来编写验证规则, 进一步使用 SeaHorn[61]验证器进行形式化验证。与之前的一些方案一样, 将 Solidity 代码转为 LLVM 中间语言的好处是可以借助针对传统程序的形式化验证解决方案, 但是缺点在于转换的过程可能难以表达智能合约程序所有的语义, 尤其是智能合约一些特有的性质。ZEUS 设计了五种安全漏洞的检测规则, 可以在形式化验证的过程中对目标程序的安全性进行判定。

Securify[62]是另一个自动化安全属性验证方面比较典型的工作。其从智能合约程序的字节码中推断出语义事实, 并将语义事实用 Datalog 语法进行描述。Datalog 是一种基于逻辑的编程语言规范, 分为事实和规则两个部分。在推断出程序的语义事实后, Securify 将其与预先定义好的安全属性规则进行检查。安全属性规则分为服从模式和违反模式, 通过语义事实与两种模式的匹配情况来检测合约的安全性。

VerX[63]是一个针对合约安全性功能属性的形式化验证工作, 其结合符号执行技术, 提出了一种延迟谓词抽象技术。延迟谓词抽象技术将交易执行期间的符号执行信息与交易之间的抽象信息相结合, 以对智能合约的时间安全属性进行自动化验证。其基本的出发点是所有的智能合约程序必须满足于其自定义的安全属性, 从而将智能合约在时间上的无限状态证明转换为可达性证明。此外, 为了实现更加精确的程序分析, VerX 设计了一个全新的面向 Solidity 语言的符号执行引擎。VerX 还意识到, 所有的智能合约一定需要满足其开发者规定的自定义安全属性, 因此也为用户设计了方便地规则定义入口, 以便对合约的自定义安全属性进行证明。

除此之外, 还有一些研究工作[64-66]也使用了不同的形式化验证技术对智能合约的漏洞挖掘进行了探索。

4.4 其他技术

除了上述模糊测试、模糊执行和形式化验证技术之外, 程序分析技术和污点分析技术也经常被用于自动化漏洞挖掘中。程序分析是一门通用的计算机技术, 其旨在通过对程序进行自动化的分析处理, 来获取程序的特征、属性, 以进一步确定程序的安全性和正确性。程序分析技术通常可以分为静态程序分析和动态程序分析, 其中静态程序分析主要利用程序的静态控制流和数据流信息进行分析, 而动态程序分析则进一步可以收集到程序的运行时信息[67]。而污点分析则是一种特殊的程序分析技术, 其通过在程序执行的过程中去标记感兴趣的关键数据并追踪其流向, 来实现精确化的程序分析。在漏洞挖掘中, 通常将来自于程序外部的输入数据标记为污点, 然后通过跟踪和污点数据相关的信息的流向, 可以知道它们是否会影响某些关键的程序操作, 进而挖掘程序漏洞。

SASC[68]是一个基于静态程序分析技术的智能合约漏洞挖掘挖掘。SASC 通过对智能合约源代码的自动分析, 来确定合约中的函数调用拓扑图和函数内部的控制流特征, 并从中检测对于区块链时间戳、交易来源地址(tx.origin)等变量的非法使用, 从而进行安全漏洞的标记和报告。SmartCheck[69]是另一个以太坊智能合约的静态分析工具。SmartCheck 同样工作在智能合约的 Solidity 源代码层面, 其进一步对源代码进行语法分析和词法分析, 并使用 XML 来描述分析后的抽象语法树结果。在此基础上, 其利用 Xpath[70]来检查智能合约的安全属性, 以检测合约中的重入、时间戳依赖、拒绝服务、资金锁定等漏洞。

Slither[71]则是一个专注于智能合约程序分析的静态分析框架, 其在对智能合约源代码静态分析的基础上, 为用户提供了信息打印(Printer)和异常探测(Detector)两大模块用于智能合约的分析。其异常探

测可以探测多达四十余种智能合约的异常, 其中就包括多种安全漏洞。Slither 创建了一种名为 SlitherIR 的中间语言, 将所有的静态程序分析工作都放在中间语言层面实现, 有助于分析框架拓展到不同的高级语言和类型。

污点分析技术的作用是实现更加精确的数据流分析, 通常不会被独立用于漏洞挖掘, 而需要和其他技术想结合。Sereum[72]则是利用了污点分析技术对影响控制流的 Storage 变量的数据流向进行追踪, 从而有效检测出三种不同的重入漏洞模式, 这个工作将在安全防御章节进行详细介绍。Oyente、Mythril 等研究工作也是利用了污点分析技术来辅助数据流分析, 以提高漏洞检测的准确性。

4.5 漏洞挖掘技术对比

模糊测试、符号执行和形式化验证是三种主流的智能合约自动化漏洞挖掘技术, 大量基于这些技术的研究工作在智能合约的自动化漏洞挖掘中取得了很好的效果, 但也还存在诸多的挑战。相关研究工作的特点总结如表 3 所示。

在现有的面向智能合约自动化漏洞挖掘的研究工作中, 如何应用模糊测试技术进行更加高效的漏洞挖掘还有很多挑战尚未解决, 也正涌现出越来越多的研究工作。Echidna[44]和 ContractFuzzer[45]是早期使用模糊测试进行智能合约漏洞挖掘的代表性作品, 但其所采用的模糊测试策略较为初级, 难以实现较高的代码覆盖率。ILF[46]和 Harvey[47]则分别通过深度学习和程序插桩的方式, 尝试生成更好的测试用例序列以提高覆盖率, 但其解决方案仅面向特定的合约, 且需要合约的源代码或调用接口规范, 难以进行大规模的扩展。如何使用模糊测试对无源码智能合约进行大规模测试、如何更加精确地在测试中发现漏洞, 仍然是智能合约模糊测试需要解决的问题。

相比之下, 符号执行则是相对成熟的漏洞挖掘技术。Oyente[48]实现了针对智能合约的符号执行引擎, 是最早的智能合约自动化漏洞挖掘工作之一。在此之后, 诸如 Osiris[49]、Mythril[50]、Manticore[52]等越来越多采用符号执行技术的智能合约自动化漏洞挖掘研究工作开始出现, 其主要通过增加符号执行分析的交易深度来模拟更加真实的合约执行, 探索更深层次的状态空间, 并通过影子栈等方式引入污点分析, 以降低误报, 实现更加精确的漏洞检测。这些措施有效地改善了符号执行在漏洞挖掘中的效果, 但也显著地提高了分析工作的计算资源和时间开销, 并且无法彻底解决状态空间爆炸的问题。尽管如此, 在考虑漏洞挖掘效率与计算开销的平衡之后, 符号执行仍然是目前相对主流的智能合约漏洞挖掘方案。

形式化验证技术用于智能合约的漏洞挖掘也经过了较为成熟的发展。Hirai、Bhargavan 等人的研究工作[53-54, 57]是早期比较有代表性的相关研究, 其主要关注如何对智能合约的指令进行通用的形式化建模, Grishchenko 等人的工作[59]则进一步引入安全属性验证, 对形式化后的智能合约程序进行验证以找出其中的安全漏洞, 但这些工作均主要关注于对智能合约程序的正确形式化, 而不是安全性验证。在此之后, ZEUS[60]、Securify[62]、Verx[63]等研究方案则提出了针对智能合约更加有效的安全属性验证方式, 以提高漏洞检测的精确度, 并提高检测过程的自动化程度, 减少人工参与。形式化验证技术被广泛用于检测合约是否满足合约设计人员所定义的安全属性规则, 但在通用漏洞的检测上不如模糊测试和符号执行应用广泛。相比基于其他漏洞挖掘技术的方案, 现有的采用形式化验证技术的研究工作大多数自动化程度不高, 且检测出来的漏洞不一定存在可达的程序路径, 因此使用形式化验证技术进行智能合约通用漏洞挖掘还面临着挑战。

除了这三种主流的技术, 一些研究工作, 诸如本章中所介绍的 ILF[46]等, 尝试通过深度学习为模糊测试训练出更好的输入生成模型来进行智能合约的漏洞挖掘, 以充分结合不同技术的优势, 实现更好的漏洞挖掘效果。此外, 也有研究工作提出了符号执行辅助的模糊测试框架等解决方案[73], 这些多种技术相结合的漏洞挖掘方案也是一个值得探究的研究方向。

4.6 本章小结

自动化漏洞挖掘是软件安全领域的重要研究方向, 也是智能合约安全研究中最活跃的研究方向。在智能合约的自动化漏洞挖掘中, 模糊测试、符号执行和形式化验证是三种最常用的技术。这三种技术各有优劣: 模糊测试技术依赖于智能合约程序的运行时信息, 其发现的程序漏洞一定是路径可达的, 漏洞检测更加精确, 而其难点在于如何产生更加优秀的输入, 来提高程序分析的覆盖率, 以分析更多的程序片段; 符号执行技术借助于约束收集和求解, 可以更精确地探索程序的执行路径, 甚至分析交易之间的依赖来求解出合适的交易序列, 但是随之而来的状态空间爆炸的问题导致复杂的约束难以求解且实际可处理的交易深度非常有限; 形式化验证则可以理解更多语义层面的信息, 保证合约程序与预期的设计相符, 但用于自定义漏洞检测的建模需要一定的人工经验, 且其探测出的漏洞并不一定是实

际可达的。

5 智能合约自动化漏洞利用

自动化漏洞利用是软件自动化攻防研究的重要方向, 其在自动化漏洞利用的基础上更进一步, 研究如何自动化生成漏洞利用脚本或漏洞利用辅助信息, 以提高漏洞利用的自动化水平。APEG[74]是提出了基于程序补丁分析的自动化漏洞利用方案, 是最早的自动化漏洞利用研究工作之一。在此之后, FUZE[75]、Revery[76]、CRAX[77]等研究工作与 APEG 一样, 主要关注于传统应用程序中内存破坏漏洞的自动化利用生成。

随着智能合约安全研究的进一步深入, 一些工作开始关注如何生成智能合约漏洞的利用信息。由于智能合约程序通常通过区块链交易来进行交互, 因此这类工作主要关注如何自动化生成恶意的交易数据, 以利用智能合约中的漏洞来完成特定的攻击。teEther[78]是智能合约自动化漏洞利用的典型代表, 其主要关注于如何自动化生成恶意的交易数据, 来从合约中盗取 ETH。teEther 首先通过静态分析来定位合约中所有可能进行 ETH 转账的指令, 并通过反向数据流分析来探测这些指令的参数是否可以来自于外部输入。对于可能受到外部输入影响的转账指令, teEther 进一步通过符号执行对程序进行路径探索, 求解出函数入口到达该指令所在基本块的约束信息, 并篡改转账地址为攻击者可控的地址, 从而完成对于智能合约中任意转账漏洞的自动化利用生成。

此外, 一些基于符号执行的分析工具能够在分析漏洞的同时求解出触发漏洞所需的交易数据, 包括函数签名、函数参数、交易发起地址等。不过受符号执行工具自身局限性影响, 这种自动化利用方式往往不能构造需要较长交易序列才能触发的攻击。例如, Oyente 只进行单次符号执行, 只能处理由单个交易触发的漏洞。Mythril 可以进行多轮符号执行, 但受路径爆炸问题的影响, 也只能处理较短交易序列。相比于 teEther, 符号执行分析工具生成的恶意交易数据能对目标漏洞进行精确利用, 但具体的攻击效果以及是否能够带给攻击者受益是不确定的, 需要根据合约语义进行具体分析。

自动化漏洞利用技术挑战重重, 面向智能合约尤其如此。在智能合约中, 漏洞的分类众多, 产生的原因各不相同, 因此对应利用方式也有显著的差别。而一次成功的攻击往往需要多个漏洞的相互配合, 并涉及到多种合约调用方式, 这些都增加了自动化漏洞利用生成的难度。现有的研究工作中, Oyente[48]、Mythril[50]等使用符号执行的智能合约漏洞挖掘工具可以生成简易的漏洞触发数据, 但是由于其并不具备构造较长交易序列的能力, 因此其难以实现真实的自动化漏洞利用。比较有代表性的方案则是 teEther[78], 其仅面向于 ETH 盗取漏洞, 来生成恶意的交易数据以窃取合约中的 ETH。如何自动化生成更多面向其他类型漏洞的自动化漏洞利用载荷, 仍然是当前智能合约自动化漏洞利用的一个研究难点。

6 智能合约安全防御

安全防御是安全研究的重要部分, 也是对抗攻击的重要环节。由于智能合约一旦部署便不能修改的特点, 对于漏洞程序的直接补丁变得不太可行, 这也导致其安全防御工作相比其他程序来说困难不少。对于智能合约的安全防御方案, 主要分为三个方向, 分别是智能合约的安全编程, 智能合约的热升级和智能合约运行时攻击检测和阻断。本章主要介绍这些防御方案的相关研究工作。

6.1 安全编程

由于以太坊智能合约代码一旦部署便无法被修改, 因此如何编写出更加安全可靠的智能合约代码就变得尤为重要。现有的研究工作中对于以太坊智能合约的安全编程主要有两种不同的方案, 一是为最为流行的智能合约编写语言 Solidity 提供安全便捷的第三方库, 二是设计比 Solidity 更加安全易用的智能合约编写语言。这两种方案都是为了减轻开发者的安全开发风险, 提高代码质量。

OpenZeppelin 在第一种方案上做了诸多的工作, 其提供了大量经过安全审计的标准化 Solidity 代码库, 包含了 ERC 标准令牌、管理员权限访问控制、加解密、安全算术运算等, 用于帮助开发者快速构建出一个安全可靠的智能合约应用程序。开发者只需要在合约开发时继承或者导入这些代码库, 便可以使用响应的库函数进行智能合约的开发[79]。OpenZeppelin 最著名的一个代码库名为 Safemath, 实验表明其可以防止智能合约算术运算中的整数溢出漏洞, 目前已经被广泛地应用于各种合约的安全开发中[80]。

而有更多的研究和工业界工作集中于第二种方案, 即提供更加安全的合约编写语言。Vyper[28]是一个测试中的以太坊智能合约高级语言, 其旨在通过缩减诸如类继承、函数重载、无限循环和递归等易于引发安全问题的高级编程特性, 来让智能合约开发变得更加简洁。另一方面, 其增加了数组边界检查、整数溢出检测等编译时优化工具, 以构建更加安

全、更加易于审计的智能合约。Flint[81]是一个还未正式发布的类型安全型语言, 面向高鲁棒性的以太坊智能合约开发。Flint 语言内置了函数调用保护功能, 允许合约函数设置调用权限, 仅面向特定的合法调用者开放。此外, 其还设计了单独的资产类型变量, 并将所有的资产转移操作都视为原子性操作进行保护, 以减少资产转移过程中的安全风险。Flint 还提供诸如静态类型检查、仅允许有限次循环等特性, 以提高以太坊智能合约的安全性。

6.2 热升级方案

无论是提供更安全的合约库或者使用更加安全的高级语言, 智能合约的安全编程方案要求在合约发布前就尽可能消灭合约中所有的漏洞, 以保证部署合约的安全性。然而, 新的智能合约漏洞在不断被发现, 即使合约在部署前已经被消除了所有的已知漏洞, 其对于部署之后被发现的新漏洞仍然无能为力。对于普通的软件程序来说, 可以通过不断的版本迭代和更新补丁来防御新出现的漏洞, 但是智能合约上链代码不可更改的特性让合约漏洞修复变得极为困难。

有很多的研究人员提出了“代理合约”的概念, 通过代理合约来实现间接的合约热升级功能, 以进行合约漏洞的修复, 其中做的比较全面的研究工作是 OpenZeppelin 实验室提出的代理合约机制[82]。智能合约最重要的两个部分分别是合约的 Storage 存储和合约的代码逻辑。基于这样的观察, 代理合约机制将合约部署分为两个步骤, 其通过一个相对安全简洁的代理合约作为智能合约的入口, 代理合约管理着合约的 Storage 存储, 并通过委托调用的方式来调用逻辑合约中的合约代码。一旦实现合约主要逻辑的代码被发现存在漏洞, 开发者可以修复该漏洞后重新部署一个新的逻辑合约, 并将代理合约中的逻辑合约地址指向这个新的合约, 从而实现合约的热升级。

通过代理合约来提供的热升级方案可以有效缓解合约在部署之后, 对于新发现漏洞的应对能力, 但是由于其改变了智能合约一旦部署便不可更改的特性, 恶意的开发者可能在升级过程中改变合约规则、引入不平等的合约条款等, 使合约丧失了不可篡改性, 也降低了合约的可信任程度, 因此也引起了比较多的争议。

6.3 攻击检测与阻断

智能合约在部署之前的安全开发无法防御所有的漏洞, 而使用代理合约的热升级模式又存在较多的争议。更重要的是, 对于那些早已经部署在以太坊上, 正在活跃运行的合约, 面对已知漏洞将显得无能为力。由于其无法以任何方式被更改, 新漏洞的发现将显著增加其受攻击的风险。当然, 漏洞的存在并不一定会给合约带来危害, 而是需要攻击者发起针对漏洞的攻击和利用。一些研究提出了智能合约的运行时攻击检测和阻断方案, 这些方案旨在通过在虚拟机运行合约时检测当前的合约调用是否是一次攻击行为, 从而对检测到的攻击行为进行阻断, 以保护已经被部署且无法被修改的智能合约免收攻击。

Sereum[72]设计了一个可以进行攻击阻断和安全检测的虚拟机, 并总结了三种不同类型的面向重入漏洞的攻击模型。其通过动态污点分析技术进行实时的程序数据流分析, 并设计合约动态调用树的结构对重入攻击进行建模, 可以在虚拟机运行时实时监测发生的重入攻击, 并进行阻断。EVM*[83]也是一个类似的研究工作, 不过其支持检测和阻断的攻击类型与 Sereum 不同, 其支持整数溢出和时间戳依赖这两类漏洞引发的攻击。ÆGIS[84]同样通过修改虚拟机的方式进行攻击检测, 但将可检测的攻击类型进行了进一步的扩展。其设计了一种领域特定语言(DSL)用于描述更加通用的攻击模型, 将其部署在虚拟机中以在检测到攻击发生时回滚当前的交易。除此之外, 其还实现了攻击模型的投票和打分机制, 经过多个节点投票产生的有效攻击检测规则可以通过共识机制在网络中进行分发和传播。

6.4 安全防御技术对比

对于智能合约漏洞的安全防御, 现有的研究工作分别从安全编程、热升级和攻击检测与阻断三个不同的方面进行了探索。

大部分的研究工作从高级语言设计的层面, 研究如何进行更加安全的智能合约编程, 以尽可能降低合约的安全风险。安全编程的代表性工作有 OpenZeppelin 提供的多个安全共享库[79]和 Vyper[28]、Flint[81]等为代表的多种安全编程语言。这些方案能够大大降低开发人员在新的程序开发中可能引入的安全风险, 但无法解决现有智能合约的安全防御问题, 且这些共享库或编程语言自身一旦出现安全问题, 将会影响到所有相关的合约程序。

第二个思路是尝试对已经部署的智能合约进行热升级, 已修补被发现的安全漏洞。这方面的代表性研究工作是 OpenZeppelin 提出的代理合约方案[82]。但是这种方案使合约的不可篡改特性被破坏, 也进一步破坏了合约参与多方的公平性, 因此很难得到广泛地应用。

还有一些研究工作则关注对智能合约的恶意攻

击交易进行实时检测和阻断, 以保护智能合约免受攻击。Sereum[72]、EVM*[83]提出了有效的实时攻击检测方案, ÆGIS[84]则进一步提出了如何改进区块链中的共识算法以让全网节点对恶意交易的阻断达成共识, 但是其算法设计较为粗糙, 难以被真实应用。攻击检测与阻断的防御方案研究, 在如何降低攻击检测误报、如何在区块链网络中对攻击阻断形成共识上, 还需要更进一步的探索。

对于智能合约的安全防御, 以上三种不同的解决方案均可以实现有效防御, 但是从实际的应用效果来看, 安全编程方案和攻击检测与阻断方案具有更加优秀的防御性能和更低的防御代价。

6.5 本章小结

安全防御是对抗攻击的重要手段, 也是安全研究的重要目标。智能合约受制于代码无法修改、运行在区块链环境中等特点, 其安全防御相比普通程序来说有不少挑战。本章介绍了智能合约安全防御的三个研究方面, 分别是智能合约安全编程、热升级方案和运行时的攻击检测与阻断。

7 智能合约安全展望

以太坊是首个大规模应用智能合约的平台, 被称为区块链 2.0 的开端。自以太坊开始, 越来越多的区块链的平台开始支持智能合约的运行。作为区块链平台上管理数字资产的重要组成部分, 智能合约的安全性必将受到越来越多的关注。智能合约安全研究的未来发展方向, 本文认为主要可以分为两个大类, 一是提升当前主流合约开发、部署、运行方式的安全性, 二是设计全新的更加安全的合约开发、部署、运行架构。

第一, 提升主流智能合约方案的安全性。当下以太坊仍然是最大的智能合约运行平台, Solidity 则是最为主流的以太坊智能合约编写语言。由于 Solidity 编写智能合约仍然是当下的主流, 本文所提到的大多数研究工作仍然着力于关注该智能合约方案下的安全性。未来, 对于该方案下的安全性研究仍有很多挑战尚未解决, 包括如何提高智能合约漏洞挖掘的有效性、如何设计更加广泛的智能合约漏洞检测模型, 以及如何更好地保护现有的智能合约程序免遭攻击。这些方向的安全研究工作都将助力于推动当下的以太坊智能合约开发者构建出更加安全的智能合约。

第二, 设计全新的智能合约方案。以太坊作为最早出现的智能合约平台, 随着越来越多的智能合约漏洞被发现, 其在设计之初留下的很多弊端开始显现。而由于区块链平台的共识机制, 大规模升级全网客户端节点中的智能合约字节码、虚拟机设计方案将是一件非常困难的事情。一些新兴的区块链平台在设计智能合约语言时, 充分总结了以太坊智能合约的弱点, 开始去设计更加安全、面向资产管理的智能合约, 其中最为典型的例子就是 Libra 区块链平台上的 Move 智能合约语言[85]。Move 语言是一个智能合约的中间语言, 其面向资产而设计, 提出了“资源优先”的概念, 并引入了全新的数字资产变量。与普通变量不同的是, 数字资产变量所表示的是资产, 正如真实世界中的资产, 不能随意复制或者凭空消失, 只能被安全地转移。此外, Move 还通过静态类型绑定、强制类型检查、不支持无限循环递归等方式, 来提供更加安全使用的面向智能合约的中间语言。之所以将这些特性设计在中间语言上, 是因为 Libra 还计划推出面向不同领域的域特定高级语言, 以帮助不同领域的开发者开发更加简洁安全的智能合约, 并通过将这些不同的高级语言编译成统一的中间语言 Move 的方式, 来提供统一的安全性保障。Libra 的做法代表着智能合约安全研究的另一种趋势, 即放下对现有平台的安全修补, 提供一种天然适用于数字资产、更加安全的全新智能合约方案。

未来, 对于智能合约安全的研究工作在这两方面均会有所展开。面向现有方案的研究工作能够解决眼下最为急迫的安全问题, 也可以在不断的软件升级中逐渐提高现有方案的安全性, 但是其最基本的以太坊设计机制很难改变, 这也让安全性的提升必须考虑兼容性, 从而带来很多性能上的损失。而智能合约并不是一个完全成熟的技术, 其自身也在不断进步之中。吸收现有方案中的问题、设计全新的面向数字资产管理的智能合约在安全性和可靠性上将会有更多的优势, 也是未来安全研究的一大方向。本文认为, 这两种安全研究的方向将会共同发展, 助力于构建出更加安全的智能合约技术。

8 结论

智能合约为区块链平台提供了丰富、安全、可信的去中心化应用使用场景, 是区块链平台的重要组成部分。智能合约因其特殊的运行环境和程序特性, 面临着全新的安全威胁。而智能合约中的漏洞一旦被利用, 不仅可能造成巨大的经济损失, 还会破坏智能合约的公平性基础。本文以目前最大的智能合约平台以太坊为例, 先介绍了智能合约的主要特性, 并分析了这些特性可能带来的潜在安全风险。基于智能合约的架构和程序特性分析, 本文提出了全新的智能合约安全威胁分析视角, 认为其可能遭受

的安全威胁主要来自于高级语言、虚拟机和区块链这三个层面共 15 类漏洞。之后, 本文介绍了现有智能合约研究工作在自动化漏洞挖掘、自动化漏洞利用和安全防御方案这几方面的进展, 并对智能合约安全研究未来的发展进行了展望, 提出了提升主流智能合约安全性和设计全新的智能合约方案这两个潜在的发展方向。

参考文献

[1] The Idea of Smart Contracts. S. Nick, http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html, 1997.

[2] Fenu G, Marchesi L, Marchesi M, et al. The ICO Phenomenon and Its Relationships with Ethereum Smart Contract Environment[C]. *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2018: 26-32.

[3] Chen Y. Blockchain Tokens and the Potential Democratization of Entrepreneurship and Innovation[J]. *Business Horizons*, 2018, 61(4): 567-575.

[4] k26dr/ethereum-games. k26dr, https://github.com/k26dr/ethereum-games, Jan. 2020.

[5] R. Hans, H. Zuber, A. Rizk, et al. Blockchain and Smart Contracts: Disruptive Technologies for the Insurance Market[C]. *AMCIS 2017* 2017:236-241.

[6] Bader L, Burger J C, Matzutt R, et al. Smart Contract-Based Car Insurance Policies[C]. *2018 IEEE Globecom Workshops (GC Wkshps)*, 2018: 1-7.

[7] Gatteschi V, Lamberti F, Demartini C, et al. Blockchain and Smart Contracts for Insurance: Is the Technology Mature Enough?[J]. *Future Internet*, 2018, 10(2): 20.

[8] Tian F. A Supply Chain Traceability System for Food Safety Based on HACCP, Blockchain & Internet of Things[C]. *2017 International Conference on Service Systems and Service Management*, 2017: 1-6.

[9] Kim H M, Laskowski M. Toward an Ontology-driven Blockchain Design for Supply-chain Provenance[J]. *Intelligent Systems in Accounting, Finance and Management*, 2018, 25(1): 18-27.

[10] K. Korpela, J. Hallikas, Dahlberg. Digital Supply Chain Transformation toward Blockchain Integration. http://hdl.handle.net/10125/41666, 2017.

[11] Christidis K, Devetsikiotis M. Blockchains and Smart Contracts for the Internet of Things[J]. *IEEE Access*, 2016, 4: 2292-2303.

[12] Y. Tian, N. Zhang, Y.-H. Lin, et al. SmartAuth: User-Centered Authorization for the Internet of Things[C]. *26th USENIX Security Symposium*, 2017: 361-378.

[13] M. I. Mehar, C. L. Shier, A. Giambattista, et al. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack[J]. *J. Cases Inf. Technol. JCIT*, 2019, 21(1): 19–32.

[14] The Parity Wallet Hack Explained. OpenZeppelin blog, https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/, Jul. 2017.

[15] Batch Overflow bug on Ethereum ERC20 token contracts and SafeMath. Medium, S. Hessenauer, https://blog.matryx.ai/batch-overflow-bug-on-ethereum-erc20-token-contracts-and-safemath-f9ebcc137434, Jan. 2019.

[16] Hacker Spends $1K to Win Over $110K in EOS Betting Game Using REX. Cointelegraph, H. Partz, https://cointelegraph.com/news/hacker-spends-1k-to-win-over-110k-in-eos-betting-game-using-rex, Sep. 2019.

[17] M. Conti, S. K. E, C. Lal, , S. Ruj, A Survey on Security and Privacy Issues of Bitcoin. *IEEE Commun. Surv. Tutor*., 2018:1-1.

[18] Ethereum white paper: a next generation smart contract & decentralized application platform. GitHub, https://github.com/ethereum/wiki.

[19] B. Xu, D. Luthra, Z. Cole, et al. Eos: An architectural, performance, and economic analysis. Bitmex. Retrieved from https://www.whiteblock. io/library/eos-test-report. pdf, 2018.

[20] Elrom E. NEO Blockchain and Smart Contracts[M]. The Blockchain Developer. Berkeley, CA: Apress, 2019: 257-298.

[21] C. Cachin, Architecture of the hyperledger blockchain fabric. *Workshop on distributed cryptocurrencies and consensus ledgers*,2016:4-9.

[22] M. Baudet, A. Ching, A. Chursin, et al. State machine replication in the Libra Blockchain. Technical Report. Calibra. https:// developers. libra. org/docs/state-machine 2019.

[23] A. Secure, The Zilliqa Project: A Secure, Scalable Blockchain Platform. 2018.

[24] G. Wood. Ethereum: A secure decentralised generalised transaction ledger[J]. *Ethereum Proj. Yellow Pap*., 2014, 151(1): 1-32.

[25] Go Ethereum. https://geth.ethereum.org/.

[26] Parity Ethereum Client. Blockchain Infrastructure for the Decentralised Web, https://www.parity.io/ethereum/.

[27] Dannen C. Solidity Programming[M]. Introducing Ethereum and Solidity. Berkeley, CA: Apress, 2017: 69-88.

[28] Vyper—Vyper documentation. https://vyper.readthedocs.io/en/ latest/.

[29] Idris | A Language with Dependent Types. https://www.idris-lang.org/.

[30] G. ZHAO, Z. XIE, X. WANG, et al. A survey on smart contract:Vulnerability analysis[J]. *Guangzhou Univ. Sci. Ed*., 2019, 18(03): 59-67.
(赵淦森, 谢智健, 王欣明, 等. 智能合约安全综述:漏洞分析[J]. *广州大学学报(自然科学版)*, 2019, 18(03): 59-67.)

[31] Rahimian R, Eskandari S, Clark J. Resolving the Multiple With-

drawal Attack on ERC20 Tokens[C]. *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2019: 320-329.

[32] The DAO (organization). Wikipedia, https://en.wikipedia.org/w/index.php?title=The_DAO_(organization)&oldid=930240181, Dec. 2019.

[33] The Multi-sig Hack: A Postmortem. Blockchain Infrastructure for the Decentralised Web, https://www.parity.io/the-multi-sig-hack-a-postmortem/, Jul. 2017.

[34] A Postmortem on the Parity Multi-Sig Library Self-Destruct. Blockchain Infrastructure for the Decentralised Web, https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/, Nov. 2017.

[35] KotET-Post-Mortem Investigation. https://www.kingoftheether.com/postmortem.html.

[36] Largest Smart Contract Attacks in Blockchain History Exposed—FOMO3D Part 2. Medium, AnChain.AI, https://medium.com/@AnChain.AI/largest-smart-contract-attacks-in-blockchain-history-exposed-fomo3d-part-2-a5e4a117f44c, Nov. 2019.

[37] Li J, Zhao B D, Zhang C. Fuzzing: A Survey[J]. *Cybersecurity*, 2018, 1: 6.

[38] M. Zalewski, American fuzzy lop. URL Httplcamtuf Coredump Cxafl, 2017.

[39] C. Aschermann, S. Schumilo, T. Blazytko, et al. REDQUEEN: Fuzzing with Input-to-State Correspondence[C]. *NDSS*, 2019:15.

[40] Chen P, Chen H. Angora: Efficient Fuzzing by Principled Search[C]. *2018 IEEE Symposium on Security and Privacy (SP)*, 2018: 23-34.

[41] S. Gan, C. Zhang, X. Qin,et al. CollAFL: Path Sensitive Fuzzing[C]. *2018 IEEE Symposium on Security and Privacy (SP)*, 2018:356-361.

[42] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware Evolutionary Fuzzing[C]. *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, 2017: 25-32.

[43] D. She, K. Pei, D. Epstein, et al. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. arXiv:1807.05620.

[44] Echidna, a smart fuzzer for Ethereum. Trail of Bits Blog, https://blog.trailofbits.com/2018/03/09/echidna-a-smart-fuzzer-for-ethereum/, Mar. 2018.

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

[46] He J X, Balunović M, Ambroladze N, et al. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts[C]. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, London United Kingdom, 2019: 531-548.

[47] V. Wuestholz, M. Christakis. Harvey: A Greybox Fuzzer for Smart Contracts. arXiv:1905.06944.

[48] L. Luu, D.-H. Chu, H. et al. Making Smart Contracts Smarter[C]. 2016: 254-269.

[49] Torres, Christof Ferreira , Julian Schütte, et al. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts[C]. *34th Annual Computer Security Applications Conference (ACSAC)*, 2018:256-271.

[50] B. Mueller, Mythril—Reversing and Bug Hunting Framework for the Ethereum Blockchain https://pypi.org/project/mythril/0.8.2/.

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

[52] Mossberg M, Manzano F, Hennenfent E, et al. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts[C]. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019: 1186-1189.

[53] Y. Hirai, Formal verification of Deed contract in Ethereum name service[C]. *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2019: 1-6.

[54] Hirai Y. Defining the Ethereum Virtual Machine for Interactive Theorem Provers[M]. Financial Cryptography and Data Security. Cham: Springer International Publishing, 2017: 520-535.

[55] Hildenbrandt E, Saxena M, Rodrigues N, et al. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine[C]. *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, July 9-12, 2018. Oxford. Piscataway, NJ: IEEE, 2018: 204-217.

[56] K Framework. http://www.kframework.org/index.php/Main_Page.

[57] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, et al. Formal verification of smart contracts: Short paper[C]. the *2016 ACM Workshop on Programming Languages and Analysis for Security*,2016: 91-96.

[58] FStarLang/FStar. GitHub, https://github.com/FStarLang/FStar.

[59] Grishchenko I, Maffei M, Schneidewind C. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts[M]. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018: 243-269.

[60] S. Kalra, S. Goel, M. Dhawan,et al. ZEUS: Analyzing Safety of Smart Contracts[C]. *Network and Distributed System Security Symposium*. 2018: 26-35.

[61] SeaHorn | A Verification Framework. https://seahorn.github.io/.

[62] P. Tsankov, A. Dan, D. D. Cohen, et al. Securify: Practical Security Analysis of Smart Contracts. ArXiv180601143 Cs, Jun. 2018.

[63] A. Permenev, D. Dimitrov, P. Tsankov, et al. VerX: Safety Verification of Smart Contracts, https://seahorn.github.io/.

[64] Abdellatif T, Brousmiche K L. Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models[C]. *2018*

9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), 2018: 1-5.

[65] Liu Z T, Liu J. Formal Verification of Blockchain Smart Contract Based on Colored Petri Net Models[C]. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, 2019: 555-560.

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

[67] Zhang R Y, Huang S Q, Qi Z W, et al. Static Program Analysis Assisted Dynamic Taint Tracking for Software Vulnerability Discovery[J]. *Computers & Mathematics With Applications*, 2012, 63(2): 469-480.

[68] Zhou E C, Hua S, Pi B F, et al. Security Assurance for Smart Contract[C]. *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018: 1-5.

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

[70] xpath cover page - W3C. https://www.w3.org/TR/xpath/all/.

[71] crytic/slither. https://github.com/crytic/slither, Jan. 2020.

[72] Rodler M, Li W T, Karame G O, et al. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks[C]. *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, 2019: 12-19.

[73] M. FU, L. WU, Z. HONG, et al. Research on vulnerability mining technique for smart contracts[J]. *Comput. Appl.* 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.)

[74] Brumley D, Poosankam P, Song D, et al. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications[C]. *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 2008: 143-157.

[75] W. Wu, Y. Chen, J. Xu, et al. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities[C]. *USENIX Security*, 2018:16.

[76] Y. Wang, C. Zhang, X. Xiang, et al. Revery: from Proof-of-Concept to Exploitable (One Step towards Automatic Exploit Generation)[C]. *ACM Conference on Computer and Communications Security (CCS)*, 2018:26-35.

[77] Lance Chen, CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations[C]. *SERE* 2012:654-661.

[78] J. Krupp, C. Rossow. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts[C]. t*he 27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018: 1317-1333.

[79] Contracts-OpenZeppelin Docs. https://docs.openzeppelin.com/contracts/2.x/.

[80] X. Bi, Z. Ma, M. Xu, Research and Implementation of Blockchain Smart Contract Security Development Technology[C]. *Inf. Secur. Commun. Priv*. 2018(12): 63-73.
(毕晓冰, 马兆丰, 徐明昆. 区块链智能合约安全开发技术研究与实现[J]. *信息安全与通信保密*, 2018(12):63-73.)

[81] F. Schrans, S. Eisenbach, S. Drossopoulou, Writing safe smart contracts in Flint[C]. *Programming'18 Companion*, 2018:123-132.

[82] Proxy Patterns–OpenZeppelin blog. https://blog.openzeppelin.com/proxy-patterns/.

[83] Ma F C, Fu Y, Ren M, et al. EVM*: From Offline Detection to Online Reinforcement for Ethereum Virtual Machine[C]. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019: 554-558.

[84] Ferreira Torres C, Baden M, Norvill R, et al. Ægis[C]. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019: 2589-2591.

[85] Getting Started With Move. Libra, https://developers.libra.org/, 2019.

倪远东 于 2017 年在哈尔滨工业大学计算机科学与技术专业取得学士学位。现在清华大学网络空间安全专业攻读硕士研究学位。研究领域为系统安全, 研究兴趣为二进制程序分析、自动化漏洞挖掘、区块链安全等。Email: nyd17@mails.tsinghua.edu.cn

张超 于 2013 年在北京大学大学计算机应用专业获得博士学位。现任清华大学网络科学与网络空间研究院副教授。研究领域为软件与系统安全。研究兴趣包括: 漏洞挖掘、漏洞利用、漏洞防利用、人工智能安全等。Email: chaoz@tsinghua.edu.cn

殷婷婷 于2018年在北京交通大学信息安全专业获得学士学位。现在清华大学网络空间安全专业攻读硕士学位。研究领域为系统安全。研究兴趣包括: 二进制安全、自动化漏洞挖掘、区块链安全等。Email: ytt18@mails.tsinghua.edu.cn