软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(8):3059−3085 [doi: 10.13328/j.cnki.jos.006375]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

智能合约安全漏洞检测技术研究综述*

钱 鹏[1], 刘振广[1,2], 何钦铭[2], 黄步添[2], 田端正[1], 王 勋[1]

[1](浙江工商大学 计算机与信息工程学院, 浙江 杭州 310018)

[2](浙江大学 计算机科学与技术学院, 浙江 杭州 310058)

通信作者: 刘振广, E-mail: liuzhenguang2008@gmail.com

摘 要: 智能合约是区块链技术最成功的应用之一, 为实现各式各样的区块链现实应用提供了基础, 在区块链生态系统中处于至关重要的地位. 然而, 频发的智能合约安全事件不仅造成了巨大的经济损失, 而且破坏了基于区块链的信用体系, 智能合约的安全性和可靠性成为国内外研究的新关注点. 首先从 Solidity 代码层、EVM 执行层、区块链系统层这 3 个层面介绍了智能合约常见的漏洞类型和典型案例; 继而, 从形式化验证法、符号执行法、模糊测试法、中间表示法、深度学习法这 5 类方法综述了智能合约漏洞检测技术的研究进展, 针对现有漏洞检测方法的可检测漏洞类型、准确率、时间消耗等方面进行了详细的对比分析, 并讨论了它们的局限性和改进思路;最后, 根据对现有研究工作的总结, 探讨了智能合约漏洞检测领域面临的挑战, 并结合深度学习技术展望了未来的研究方向.

关键词: 区块链; 智能合约; 以太坊; 漏洞检测; 自动化工具

中图法分类号: TP311

中文引用格式: 钱鹏, 刘振广, 何钦铭, 黄步添, 田端正, 王勋. 智能合约安全漏洞检测技术研究综述. 软件学报, 2022, 33(8): 3059–3085. http://www.jos.org.cn/1000-9825/6375.htm

英文引用格式: Qian P, Liu ZG, He QM, Huang BT, Tian DZ, Wang X. Smart Contract Vulnerability Detection Technique: A Survey. Ruan Jian Xue Bao/Journal of Software, 2022, 33(8): 3059−3085 (in Chinese). http://www.jos.org.cn/1000-9825/6375.htm

Smart Contract Vulnerability Detection Technique: A Survey

QIAN Peng[1], LIU Zhen-Guang[1,2], HE Qin-Ming[2], HUANG Bu-Tian[2], TIAN Duan-Zheng[1], WANG Xun[1]

[1](School of Computer and Information Engineering, Zhejiang Gongshang University, Hangzhou 310018, China)

[2](School of Computer Science and Technology, Zhejiang University, Hangzhou 310058, China)

Abstract: Smart contract, one of the most successful applications of blockchain, provides the foundation for realizing various real-world applications of blockchain, playing an essential role in the blockchain ecosystem. However, frequent smart contract security events not only caused huge economic losses but also destroyed the blockchain-based credit system. The security and reliability of smart contract thus gain wide attention from researchers worldwide. This study first introduces the common types and typical cases of smart contract vulnerabilities from three levels, i.e., Solidity code layer, EVM execution layer, and blockchain system layer. Then, the research progress of smart contract vulnerability detection is reviewed and existing efforts are classified into five categories, namely formal verification, symbolic execution, fuzzing testing, intermediate representation, and deep learning. The detectable vulnerability types, accuracy, and time consumption of existing vulnerability detection methods are compared in detail as well as their limitations and improvements. Finally, based on the summary of existing researches, the challenges in the field of smart contract vulnerability detection are discussed and combined with the deep learning technology to look forward to future research directions.

Key words: blockchain; smart contract; Ethereum; vulnerability detection; automation tool

* 基金项目: 国家重点研发计划(2017YFB1401300, 2017YFB1401304); 浙江省自然科学基金(LQ19F020001); 国家自然科学基金(61902348); 浙江省重点研发计划(2021C01104)

收稿时间: 2020-08-13; 修改时间: 2021-01-18; 采用时间: 2021-05-08; jos 在线出版时间: 2021-05-20

区块链技术[1,2]本质上是一个分布式共享交易账本, 由区块链网络中的所有节点在共识协议[3]约束下共同维护. 区块链技术具有去中心化、防篡改、不可逆及可追溯等特点, 改变了传统行业的固有模式, 在医疗[4−6]、版权保护[7−9]、供应链管理[10−12]、能源互联网[13−15]、物联网服务[16−18]等诸多领域取得了突破性进展[19].

智能合约[20,21]是区块链技术最成功的应用之一, 已经成为学术界和工业界研究的新关注点. 早在 1994 年, 智能合约的概念就被 Szabo 提出[22], 但直到区块链技术的出现, 才为其提供了可靠的执行环境和应用基础. 智能合约本质上是一段运行在区块链上的计算机程序, 由图灵完备(Turing complete)语言编写, 可以在区块链网络中自动执行. 当前, 已有数以万计的智能合约部署在各类区块链平台上, 例如以太坊(Ethereum)[23]、EOS[24]、维特链(VNT Chain)[25]等, 并且其数量仍在迅速增长, 在过去几个月中, 仅在以太坊平台上就部署了超过 10 万个新的智能合约. 然而, 随着智能合约数量的增加, 智能合约安全问题也接踵而至. 据 Bcsec 和 Slowmist 统计[26,27], 智能合约安全漏洞导致的经济损失已经超过数十亿美元. 由于智能合约是一段程序代码, 其在设计和开发过程中难免会出现代码安全问题, 并且部署在公链上的智能合约通常暴露在开放网络环境中, 这进一步使得智能合约容易成为受攻击的目标. 由于区块链的不可篡改与不可逆等特性, 当黑客攻击某个智能合约时, 我们只能眼睁睁地看着资金流入攻击者账户而无法中断或阻止合约执行. 目前, 由智能合约漏洞引发的大规模安全案例不在少数, 并且每隔一段时间就会发生相关的安全漏洞事件. 例如: 2016 年 6 月, 黑客利用 DAO(decentralized autonomous organization)[28]合约的可重入漏洞, 窃取了价值约 6 000 万美元的以太币(即以太坊数字货币); 2017 年 7 月, 由于 Parity 多签名钱包合约的 Delegatecall 漏洞(parity multi-sig wallet delegatecall)[29], 价值近 3 亿美元的以太币被冻结; 2018 年 4 月, 恶意攻击者利用美链 BEC 合约[30]的整数溢出漏洞无限复制代币, 导致 BEC 代币的价值蒸发归零; 2018 年 11 月, 攻击者向 EOS.WIN 发起连续随机数攻击[31], 获利超过 20 000 枚 EOS(即 EOS 数字货币); 2019 年 5 月, Binance 交易所遭到黑客恶意攻击, 导致 7 000 多枚比特币被盗; 2020 年以来, 智能合约游戏 FarmEOS, Playgames, LuckBet, EOSPlaystation 等[32−35]都遭遇了不同程度的黑客攻击, 累计造成近百万美元的损失. 智能合约安全漏洞不仅造成了巨大的经济损失, 也破坏了人们对区块链与智能合约的信任基础, 智能合约的漏洞检测与安全防范已经成为亟待解决的关键问题和巨大挑战.

智能合约之所以容易受到安全漏洞影响的原因, 可以归纳为以下 4 个方面.

首先, 当前的智能合约编程语言和工具(如 Solidity)仍然是新颖且粗糙的, 而使用新的编程语言和运行环境编写的智能合约相对更难以测试, 尤其因为智能合约运行时被允许与外部合约函数或接口交互, 这可能导致重复的外部调用, 从而引发安全漏洞;

其次, 由于智能合约开发人员一时间无法完全理解新颖的智能合约编程语言和工具的基本执行逻辑, 因此无法预见合约在将来遇到的所有可能状态和环境,导致了开发人员容易编写出存在漏洞或易受攻击的合约;

第三, 区别于传统程序, 普通的应用程序发布后遇到安全问题时, 开发人员可以检测程序错误并进行修改. 然而, 由于智能合约二进制码及其状态是存储在不可篡改且不可变的区块链网络上, 智能合约一旦部署便是不可逆的, 也无法进行更新或修改, 这就导致区块链网络上可能存在一些有潜在安全问题而无法修补的合约;

第四, 由于区块链平台上的数字货币(如比特币[36]和以太币)被智能合约保管和操控, 使得智能合约成为极具诱惑力的被攻击目标, 吸引了很多恶意攻击者, 他们尝试各种手段挖掘并利用智能合约中可能存在的漏洞, 以窃取资金或阻塞区块链网络.

综上所述, 相较于其他的软件和应用程序, 暴露于开放环境下的智能合约更容易受到安全漏洞和恶意攻击的影响, 也进一步突显了智能合约漏洞检测和安全防范的重要性.

本文广泛收集了智能合约安全领域的相关研究, 文献选取标准为该文献与智能合约漏洞检测直接相关, 并在智能合约安全漏洞挖掘与检测领域提出了新方法、新技术. 本文收录时间为 2015 年以来符合要求的文献为标准, 收集范围涵盖 Web of Science, IEEE Xplore, SpringerLink 以及中国知网 CNKI 等计算机领域常用的国内外数据库. 截止到 2020 年 6 月, 共有 91 篇相关文献. 可以看出: 近 3 年来, 智能合约安全漏洞分析与检测

领域的研究论文数量迅速增长(如图 1 所示).

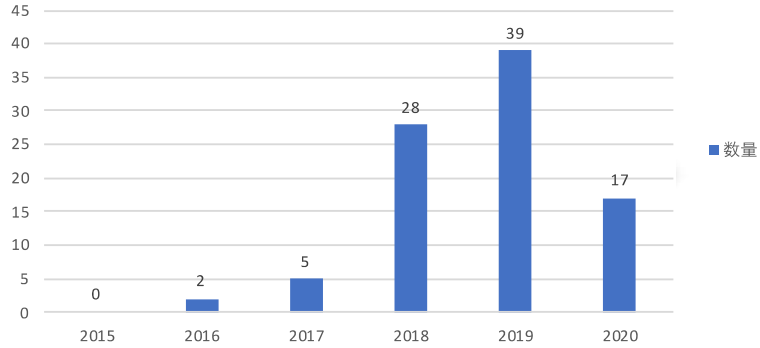

图 1 智能合约安全漏洞分析与检测领域文献分布情况

本文第 1 节从 Solidity 代码层、EVM 执行层、区块链系统层这 3 个层面介绍常见的智能合约漏洞类型和典型案例. 第 2 节阐述智能合约漏洞检测的常用方法及相应的漏洞检测工具. 第 3 节比较分析各种漏洞检测工具的特点、自动化程度、开源程度及检测性能, 并分析了它们的局限性和改进思路. 第 4 节总结全文, 讨论现有智能合约漏洞检测方法的不足之处, 并结合深度学习技术展望未来智能合约漏洞检测的研究方向.

1 智能合约安全漏洞类型和典型案例

随着区块链技术的发展、智能合约数量的增加以及去中心化应用(decentralized application, DAPP)的普及[37−40], 智能合约的安全漏洞被越来越多的攻击者发现并利用[41−44]. 相较于传统的应用程序, 智能合约引起的安全问题更加棘手、漏洞分析更加复杂. 本节归纳总结了当前主要的智能合约安全漏洞类型和典型案例.

1.1 智能合约安全漏洞类型

以太坊是当前影响力最大的开源区块链平台, 也是目前为止智能合约数量最多、漏洞类型最多、漏洞造成的损失最大的区块链平台. 从 2020 年前 2 个季度的 DAPP 市场占比来看, 以太坊 DAPP 占总创造价值的 82%, 其中 80%属于博彩、游戏等高风险类别; 其次, 据 Dune Analytics 数据统计[45], 仅 2020 年 3 月份, 部署在以太坊平台上的智能合约就超过200万份. 此外, 智能合约漏洞的典型案例大多由以太坊智能合约引发, 并且多数区块链平台上的智能合约是基于太坊智能合约进行设计与编写, 开发人员和研究人员通常以以太坊智能合约为研究对象进行实验与分析. 因此, 本文以以太坊为例, 详细阐述了智能合约安全漏洞并进行了分类.

根据以太坊智能合约漏洞发生的层次不同,可以将合约漏洞分为 3 个层面, 分别是 Solidity 代码层、EVM 执行层、区块链系统层. 本节中介绍了 15 种以太坊智能合约漏洞, 具体的漏洞分类与解释说明见表 1.

1.1.1 Solidity 代码层

(1) 可重入漏洞

程序执行具有原子性和顺序性. 一般来说, 当调用非递归函数时, 一次程序命令执行结束前将不会有新的执行命令进入. 然而智能合约的执行并非如此, 由于 Solidity 智能合约独特的回调(fallback)机制[46], 很可能使恶意攻击者在程序命令执行结束前再次进入被调用函数. 类似于大多数程序语言, 以太坊智能合约处理业务逻辑时会进行跨合约的函数调用, 不同的是, 智能合约经常涉及转账等敏感操作. 此外, 由于智能合约的固有特性, 转账操作必定会触发接收者合约中的回调函数. 当智能合约进行跨合约的转账操作时, 如果这些外部调用被恶意攻击者利用, 很可能会导致合约进一步地执行危险操作. 例如, 攻击者在其回调函数中设计恶意的攻击代码, 递归调用受害者合约的转账函数以盗取以太币. 以太坊可重入漏洞就是因为这种机制而产生的, 造成了有史以来最著名的智能合约安全漏洞事件(即 the DAO 攻击[28]), 不仅损失了价值近 6 000 万美元的以太币, 而且直接导致了以太坊硬分叉(hard-fork)[47].

表 1 智能合约漏洞分类与解释说明

漏洞分类	漏洞类型	漏洞原因	相关攻击	安全问题
Solidity 代码层	可重入漏洞	Fallback 函数中存在对外部合约函数的递归调用	The DAO 攻击	无法存储和保护合约代币或数据
	整数溢出漏洞	数值超出或低于定义的整数类型范围	整数溢出攻击 (美链 BEC 攻击)	整数范围错误
	权限控制漏洞	函数或变量的访问被限制为 public 类型	–	函数或变量被任意用户或变量调用
Solidity 代码层	异常处理漏洞	函数调用后未检查返回值和类型	The DAO 攻击, KoET 攻击	异常处理失败
	拒绝服务漏洞	意外执行自毁指令; 访问控制策略出错; Gas 达到区块上限; 非预期异常抛出	KoET 攻击	代币冻结;无法存储和保护合约代币或数据
	类型混乱漏洞	变量类型定义错误	–	无法存储和保护合约代币或数据
	未知函数调用漏洞	函数调用和转账操作引起 Fallback 函数自动触发	The DAO 攻击	无法存储和保护合约代币或数据
	以太冻结漏洞	合约被未经授权的用户销毁	Parity Multi-Sig Wallet 攻击	不适当的合约或函数访问
EVM 执行层	短地址漏洞	合约地址不符合规范 (小于 20 个字节)	–	无法存储和保护合约代币或数据
	以太丢失漏洞	合约地址错误或为空	–	无法存储和保护合约代币或数据
	调用栈溢出漏洞	合约或函数的调用次数超出 EVM 上限	–	缓存溢出问题
	Tx.Origin 漏洞	tx.origin 全局变量用于智能合约身份验证	–	不适当的合约或函数访问
区块链系统层	时间戳依赖漏洞	区块时间戳被赋值给可预测的变量	–	无法使用安全的随机数
	区块参数依赖漏洞	区块相关参数或信息赋值给可预测的变量	–	无法使用安全的随机数
	交易顺序依赖漏洞	交易顺序不一致	–	竞争条件/非法预先交易

(2) 整数溢出漏洞

整数溢出漏洞具有普遍性, 很多程序中都可能存在整数溢出问题. 整数溢出一般分为上溢和下溢, 智能合约中的整数溢出类型包括 3 种: 乘法溢出、加法溢出和减法溢出. 以太坊智能合约中, 一般指定整数为固定大小且无符号整数类型, 即表示整型变量只能是一定范围内的数值, 超过这个范围则会产生整数溢出错误. Solidity 语言的整型变量步长一般以 8 递增, 支持从 uint8 到 uint256. 以 uint8 类型为例, 其变量长度为 8 位, 支持存储的数字范围是[0,255]. 若试图将大小超过这个范围的数据存储到 uint8 类型变量中, 以太坊虚拟机 (Ethereum virtual machine, EVM)将会自动截断高位, 从而导致运算结果异常, 产生整数溢出错误. 不同于其他程序, 智能合约整数溢出漏洞造成的损失是巨大且不可弥补的, 例如美链 BEC 合约的整数溢出漏洞[30]导致其代币价值瞬间归零. 目前, 为了防止智能合约的整数溢出, 一方面可以在算数逻辑前后进行验证; 另一方面, 开发人员能使用 SafeMath 安全库[48]处理算术逻辑, 防止整数溢出.

(3) 权限控制漏洞

智能合约权限控制漏洞[41,49]产生的最根本原因是未能明确或未仔细检查合约中函数的访问权限, 从而允许恶意攻击者能进入本不该被其访问的函数或变量. 权限控制漏洞主要体现在两个层面.

1) 合约代码层面. Solidity 智能合约函数和变量的访问限制有 4 种, 即 public, private, external, internal. 如果函数未使用这些标识符, 那么默认情况下, 智能合约函数的访问权限为 public, 亦即该函数允许被本合约或其他合约的任何函数调用, 这种情况可能导致该函数被攻击者恶意调用;
2) 合约逻辑层面. 通常使用函数修饰器对函数或变量进行约束. 例如, 某些关键函数需要使用修饰器 onlyOwner 或 onlyAdmin 来约束. 若未给这些函数添加修饰器, 任何人都有权利访问并操纵这些关键函数, 则很有可能导致关键函数被恶意攻击者操纵, 从而进一步地破坏智能合约逻辑.

(4) 异常处理漏洞

以太坊智能合约中, 有 3 种情况会抛出异常: ① 执行过程中的 Gas(即部署或执行智能合约的费用)消耗殆尽; ② 调用栈溢出; ③ 执行语句中有 throw 命令. 一般来说, 如果被调用的合约抛出异常时, 合约将会通

过回滚的方式处理异常行为, 即终止当前合约执行并恢复到上一步状态, 同时返回一个错误标识符(即 false). 然而, 当一个合约以不同的方式调用另一个合约时, 以太坊智能合约却没有统一的方法处理异常, 发起调用的合约可能无法获取被调用合约中的异常信息. 例如, 智能合约中的子调用发生异常时, 通常会自动向上级传播, 但是一些底层调用函数(如 send, call, delegatecall)发生异常时只返回 False, 而不抛出异常[41,49]. 因此, 仅仅根据有无异常抛出就判断合约执行是否成功是不安全的, 在调用底层函数时必须严格检查返回值, 并且对异常采用一致性的处理方式.

(5) 拒绝服务漏洞

拒绝服务(denial of service, DOS)[41,43,44]是以太坊智能合约常见的漏洞, 攻击者通过破坏合约中原有的逻辑, 消耗以太坊网络中的资源(如以太币和 *Gas*), 从而使合约在一段时间内无法正常执行或提供正常服务. 针对智能合约的 DoS 攻击方式通常有 3 种.

1) 通过(unexpected)Revert 发动 DoS 攻击. 当智能合约状态的改变由外部函数的执行结果决定, 并且这个执行一直失败时, 若未对函数执行失败的情况进行处理, 将会使智能合约处于容易遭到 DoS 攻击的状态;
2) 通过以太坊区块的 *Gas* 限制发动 DoS 攻击. 以太坊网络中每个区块都设定了 *Gas* 上限, 如果交易花费的 *Gas* 超过上限, 会导致交易失败. 因此, 即使没有受到恶意攻击, 智能合约的运行也可能因为超过 *Gas* 限制而出现问题; 更严重的情况是, 若攻击者恶意操纵 *Gas* 消耗而导致其达到区块上限, 则会使合约的交易过程以失败告终;
3) 合约 Owner 账户发动 DoS 攻击. 很多智能合约都有 Owner 账户, 其拥有开启或停止合约交易的权限, 若没有保护好 Owner 账户, 导致其被攻击者操控, 很可能会使合约交易被永久冻结.

(6) 类型混乱漏洞

以太坊智能合约使用高级语言 Solidity 编写, 它是强类型编程语言, 会自动检查程序中是否有类型匹配错误[48], 例如在变量赋值时, 若把字符串赋值给整型变量, 则会产生类型匹配错误. 然而在智能合约中, 有些情况即使类型不匹配, 合约在执行过程中也不会引发异常. 因此, 开发人员有时候会默认合约可以自行检查程序中的类型匹配问题时, 往往会忽视人工检查, 从而导致意料之外的漏洞发生.

(7) 未知函数调用漏洞

类似于大多数编程语言, 智能合约通过函数名和函数参数类型确保函数的唯一性. 当一个智能合约调用另一个合约中的函数时, 若函数和参数类型无法匹配到被调用合约中的函数, 此时将会默认调用该合约中的 Fallback 函数. 若是该 Fallback 函数中隐藏了攻击者设计的恶意操作, 那么很可能会出现安全问题[44,49].

(8) 以太冻结漏洞

转账操作是智能合约最重要且最独特的功能之一. 智能合约可以接收以太币转账, 也可以转账给其他合约地址. 值得一提的是: 一些合约自身不实现转账函数, 而是通过 delegatecall 调用外部合约中的转账函数来实现转账功能. 然而, 若是这些提供转账函数的外部合约带有 Self-destruct 或 Suicide 等操作时, 那么通过 delegatecall 调用转账函数的合约很可能会发生因为被调用合约的自毁操作而导致自身以太币被冻结的情况[43,44,49].

1.1.2 EVM 执行层

(1) 短地址漏洞

智能合约短地址漏洞[41,44,49]其实是利用了以太坊虚拟机自动补 0 的特性. 在智能合约 ABI 规范中, 输入的合约地址长度必须为 20 字节, 当地址长度小于 20 字节时, 以太坊虚拟机会通过在末尾自动补 0 来满足地址长度的要求. 然而, 正是因为这个特性使得恶意攻击者有机可趁. 例如, 攻击者故意把账户地址少输一个字节, 以太坊虚拟机解析时就会从下一个参数(即以太币数量)取缺少的编码位数对地址进行补全, 然后在整串二进制码的末位补 0 至正常的编码位数, 这就意味着以太币数量这个参数被左移了一个字节, 此时若是执行的是转账操作, 则可能使合约转出超出实际应该转发的以太币数量给攻击者.

(2) 以太丢失漏洞

智能合约转账以太币时必须指定接收方的合约地址, 并且地址必须是规范的. 若是接收方的合约地址是完全独立的空地址, 即它们与任何其他用户或合约都没有关联, 如果将以太币转账给这样的合约地址, 将会导致以太币永远丢失[49].

(3) 调用栈溢出漏洞

智能合约每调用一次外部合约或者自身调用, 都会增加一次合约的调用栈深度[41,49]. 在以太坊虚拟机中, 调用栈的限制为 1 024, 若攻击者设计一系列的嵌套调用, 最终可能会成功引发调用栈的溢出, 从而进一步使智能合约处于不安全的状态.

(4) Tx.Origin 漏洞

以太坊智能合约有一个全局变量 tx.origin[44], 它能够回溯整个调用栈返回最初发起调用的合约地址. 若是合约使用这个变量做用户验证或授权操作时, 攻击者便可以利用 tx.origin 的特性创建相应的攻击合约盗取以太币. 例如, 攻击者在自己的 Fallback 函数中调用受害者合约的取钱函数, 通过诱导受害者合约转账以太币给攻击者合约, 而由于“tx.origin==owner”的缘故, 会导致无法检测出异常, 从而使受害者合约中的以太币全部转到攻击者合约账户中.

1.1.3 区块链系统层

(1) 时间戳依赖漏洞

智能合约通常使用矿工(即区块链网络中的节点或用户)确认的区块时间戳(block.timestamp)来实现时间约束[41−44,49], 合约可以检索区块的时间戳且区块中的所有交易共享同一个时间戳, 这保证了合约执行后状态的一致性. 然而, 创建区块的矿工可以在一定程度上刻意选择有利于他们的时间戳来攫取利益.

(2) 区块参数依赖漏洞

以太坊智能合约中无法直接创建随机数, 合约开发者往往会编写随机函数来产生随机数, 一般通过区块号(block.number)、区块时间戳(block.timestamp)或者区块哈希(block.blockhash)等相关的区块参数或信息作为产生随机数的基数种子[44]. 然而, 与时间戳依赖漏洞类似, 由于随机数生成依赖的这些区块参数可以被矿工提前获取, 这将导致生成的随机数是可预测的, 从而可能会被恶意攻击者利用并产生对他们有利的随机数.

(3) 交易顺序依赖漏洞

区块链网络中的交易执行顺序是由区块链网络中矿工决定的, 有些合约对交易执行顺序是有严格要求的, 错误的顺序可能对合约造成负面影响[43]. 如图 2 所示的交易顺序依赖案例, 用户 1 和用户 2 同时在 t 时刻分别提交了交易 $T1$ 和 $T2$, 然而 $T1$ 和 $T2$ 的执行顺序是被区块中的矿工决定, 如果 $T1$ 先执行, 则合约状态将由 S 变为 $S1$, 反之则变为 $S2$. 因此, 最终的合约状态依赖于矿工选定的交易执行顺序. 如果恶意的矿工监听到区块中对应的合约交易, 便可以通过提交恶意的交易改变当前合约状态, 从而有机会提前部署攻击.

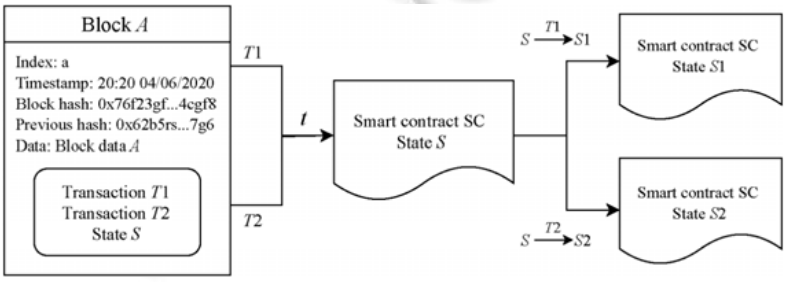

图 2 交易顺序依赖漏洞案例

1.2 典型案例

智能合约保管和操控着价值数千亿美元的数字货币. 面对如此巨大的财富的诱惑, 恶意攻击者尝试利用

智能合约漏洞来窃取利益. 本节详细阐述了现实世界中已经发生的智能合约安全漏洞事件的典型案例.

1.2.1 The DAO 案例[28]

DAO是一个去中心化的自治组织, 实现了用于众筹的The DAO合约, 该合约在遭到攻击前已经筹集了价值约 2.45 亿美元的以太币. 攻击者利用该合约的可重入漏洞窃取了价值约 6 000 万美元以太币, 直接引起了以太坊的硬分叉. 为了更好地理解 The DAO 攻击, 我们在图 3 中呈现了简化的 The DAO 攻击案例.

```
1  contract Bank {
2   mapping ( address => unit) private userBalance;
3
4   function deposit () payable{
5     userBalance[msg.sender]+=msg.value;
6   }
7   function withdraw() public{
8     uint amount = user Balance [msg.sender];
9     require(msg.sender .call.value (amount )());
10    user Balance [msg.sender ] = 0;
11  }
12 }
```

```
1  contract Attacker {
2    address bank_add=01f3x...32;
3
4    function attack (){
5      bank_add.deposit.value(10)();
6      bank_add.withdraw();
7    }
8    function () payable{
9      if(count ++ < 10)
10       bank_add .withdraw();
11   }
12 }
```

图 3 DAO 攻击简化版本

具体而言, 合约 Bank 存在两个函数: deposit 函数允许用户存入以太币, withdraw 函数允许用户提取存入合约的以太币. 攻击者的具体攻击步骤如下.

1) 攻击者 Attacker 首先调用合约 Bank 中的 deposit 函数存入 10 个以太币;
2) 然后利用 withdraw 函数取出以太币;
3) 当合约 Bank 调用 withdraw 函数, 并通过内置函数 call.value 将以太币转出给攻击者 Attacker 时, 合约 Attacker 中的 Fallback 函数(即“function () payable”)将会自动执行;
4) 攻击者 Attacker 在其 Fallback 函数中又递归调用了 10 次 withdraw 函数来取出以太币, 合约 Bank 将相应地执行 10 次转币操作, 共转出 100 个以太币给攻击者.

最终,攻击者通过其 Fallback 函数窃取到了 90 个不属于他的以太币.

1.2.2 KoET 案例[50]

KoET(king of the ether throne)[50]是一款以太坊智能合约游戏, 参与的玩家通过竞争最终的以太坊“王座”赢取合约奖池中的所有奖金. 每当玩家向合约发送“竞争费”(即以太币)竞争“王座”时, 就会触发 KoET 合约中的 Fallback 函数, 该 Fallback 函数首先会检查玩家发送的“竞争费”是否足够: 如果不够, 合约将抛出异常并回滚本次交易; 如果足够, 玩家将成为新的“王座”, 且 KoET 合约会将玩家“竞争费”的一部分作为报酬发送给前任“王座”, 一部分则留在合约中作为最终的奖金. 24 小时后, 最后一位竞争成功的玩家将获得合约中所有的奖金.

这个游戏合约看起来很合理, 实际并非如此, 该合约容易受到拒绝服务(DoS)攻击. 如图 4 所示, 攻击者通过以下攻击步骤, 将会一直霸占“王座”直到夺得合约中所有奖金: 1) 攻击者 Malicious 执行 setKing 函数, 向 KoET 合约中转入足够的“竞争费”成为新的“王座”; 2) 当有新的玩家参加竞选并支付足够的“竞争费”后, KoET 合约将会向当前“王座”(即 Malicious)发送报酬, 这会触发 Malicious 中的 Fallback 函数. 然而, 攻击者 Malicious 的 Fallback 函数中仅有抛出异常的操作, 将会拒绝该次交易并导致交易回滚. 此时, KoET 合约将会重新尝试发送报酬给 Malicious, 会遇到同样的情况. 最终, 往返多次的发送报酬会让 *Gas* 消耗殆尽, 将没有人能再次竞选成为“王座”, 攻击者 Malicious 将一直霸占“王座”, 赢取 KoET 合约中所有的奖金.

```
contract KoET{
  address public king;
  uint public price = 10;
  ...
  function (){
    if (msg.value < price) throw;
    uint comp = generateCompensation();
    if (!king.call.value(comp)()) throw;
    king = msg.sender;
    price = generateNewPrice();
  }
}
```

```
contract Malicious {
  ...

  function setKing (address a, uint w) {
    a.call.value(w);
  }

  function () {
    throw;
  }
}
```

图 4 King of the ether throne 攻击案例

1.2.3 Parity Multi-Sig Wallet 案例[29]

Parity 多签名钱包是智能合约实现用来管理用户数字资产的公共库合约, 其中包括一些常用函数和逻辑代码, 用户可以在他们的钱包合约中调用该公共库合约中的函数来执行相应的业务逻辑. 然而, 公共库合约中的函数集中管理很容易成为攻击的目标. 由于以太坊中的大部分用户钱包都依赖于此公共库合约, 因此, 攻击者只要通过攻击公共库合约, 就可以影响所有依赖于它的钱包合约.

图 5 描述了多签名钱包冻结案例, 其中包括 WalletLibrary 和 MulitisigWallet 的合约片段.

合约 WalletLibrary 中, initWallet 函数用来初始化钱包的使用日限和所有者等参数, 任何使用 delegatecall 的钱包合约都可以公开调用 initWallet 函数, 如合约 MulitisigWallet 的第 9 行所示, 钱包实例使用 delegatecall 调用了公共库 WalletLibrary. 由于 WalletLibrary 中的所有公共函数(例如 initDayLimit 和 initMulitowned)都可以被未经授权的任何人调用, 攻击者只要通过调用 WalletLibrary 中的 initWallet 函数索取多签名钱包所有权, 就可以控制该多签名钱包. 在 Parity 多签名钱包冻结案例中, 攻击者获取该多签名钱包所有权后, 通过 Self-destruct 操作销毁了该钱包, 导致所有依赖于该公共多签名钱包的所有用户钱包被冻结, 总共冻结的以太币价值近 3 亿美元.

```
contract WalletLibrary {
  ...

  // set daylimit and multiple owners
  function initWallet(address [] owners,
  uint required, uint dayLimit){
    initDaylimit(dayLimit);
    initMultiowned(owners, required);
  }
}
```

```
contract MultisigWallet {
  ...

  // deposit an amount to sender's address
  function () payable{
    if (msg.value > 0)
      Deposit(msg.sender , msg.value);
    else if (msg.data.length > 0)
      W alletLibrary .delegatecall(msg.
      data);
  }
}
```

图 5 Parity 多签名钱包冻结案例

1.2.4 美链 BEC 合约整数溢出案例[30]

智能合约中, 如果对整型变量值操作不当, 极易产生整数溢出现象. 现实世界中的智能合约整数溢出案例很多, 例如在 Proof-of-Week-Hands(简称 POWH)合约中[51], 攻击者利用其整数溢出漏洞窃取了约 2 000 个以太币. 近年来, 最著名的智能合约整数溢出案例则是美链 BEC 合约漏洞, 攻击者通过 BEC 合约的批量转账方法无限生成代币, 导致整个美链 BEC 代币价值瞬间蒸发归零.

图 6 呈现了美链 BEC 合约中存在整数溢出问题的代码片段, 其中, batchTransfer 是一个批量转账函数, 通过传入多个地址和转账金额后, 实现 BEC 的批量转账. 然而, 这个转账函数中存在整数溢出漏洞, 问题出在图 6 中的第 6 行代码“uint256 amount=uint256(cnt)*_value”. 智能合约中, uint256 表示 256 位无符号整数, 数据范围为$[0,2^{256}-1]$,若传入的_value 值过大时, 则会使 amount 的值超过 uint256 的数据范围, 从而出现整数上溢

的问题. 攻击者正是利用这里的漏洞, 通过传入很大数值的_value, 最终导致 BEC 合约发生了整数溢出漏洞, 造成了不可逆转的损失.

```
1  contract PausableToken {
2    ...
3    function batchTransfer (address [] _receivers, uint256 _value      )
4      public whenNotPaused returns (bool){
5          uint cnt = _receivers.length;
6          uint256 amount = uint256(cnt) * _value;
7          require(cnt > 0 && cnt <= 20);
8          require(_value > 0 && balances[msg.sender] >= amount);
9          ...
10         return true
11   }
12 }
```

图 6 BEC 合约整数溢出漏洞案例

1.2.5 庞氏骗局合约 Rubixi 案例

庞氏骗局[52,53]是一种经典的投资欺诈手段, 其典型的特征在于将新加入的投资者的一部分参与费作为回报支付给现有的投资者. 庞氏骗局的发起者通常承诺加入投资能产生高回报且风险小, 以此吸引新的投资者. 然而, 这类骗局最终只会损害绝大多数参与者的利益.

随着智能合约的广泛运用, 庞氏骗局开始逐渐呈现新的特征[54]. 由于区块链的匿名性, 所有的参与者无法知道合约发起者的真实身份, 导致很多庞氏骗局可以在智能合约的掩饰下伪装起来, 我们将这类庞氏骗局称为智能合约庞氏骗局. 由于智能合约具有自动执行、不可篡改等特性, 这使得它成为庞氏骗局吸引受害者最有利的手段之一. 为了更好地了解智能合约庞氏骗局, 本节在图 7 中呈现了一个典型的智能合约庞氏骗局 Rubixi 案例.

从图 7 中可以看到, Rubixi 合约中分别包含 Rubixi 构造函数、Fallback 函数、collectAllFees 函数以及 addPayout 函数.

- Rubixi 构造函数会在合约创建时执行, 并且只执行一次;
- Fallback 函数(即“function () { addPayout(); }”)会在收到以太币转账时自动执行, 当参与者将以太币投入到 Rubixi 合约时, Fallback 函数将自动被触发;
- collectAllFees 函数用于合约创建者提取合约中存放的资金;
- addPayout 函数则是最关键的函数,它实现了庞氏骗局的主要逻辑: (1) 记录参与者的地址和参与费用; (2) 计算参与者的投资费用; (3) 当合约中余额足够时, 则支付给现有的参与者报酬费用.

显然, 从图 7 的代码片段中可以看出, pyramidMultiplier 是控制参与者能够获得多少利润的关键变量: 为了吸引早期的投资参与者, 合约所有者预先设置其值为 300; 当参与人数达到 10 和 25 时, pyramidMultiplier 的值降为 200 和 150. 从整个合约逻辑来看, 该合约发起者主要目的其实是为了窃取投资参与者的投资费用, 如“collectedFees+=(msg.value*fee)/100”(第 29 行)用来收取每笔投资 10%的参与费, 通过调用 collectAllFees 函数提取费用“creator.send(collectedFees)”(第 16 行), 这就是一个典型的智能合约庞氏骗局.

2 智能合约安全分析方法

智能合约安全漏洞造成的重大损失一方面严重影响了用户体验, 另一方面破坏了区块链智能合约的信用体系, 当前智能合约的安全问题正成为研究者和开发者共同关注的焦点. 为了防止恶意攻击者利用智能合约漏洞, 研究者们已经尝试使用各种方法对以太坊智能合约源代码以及 EVM 字节码进行全面分析. 传统的程序漏洞检测方法使用的是特征匹配[55], 即: 对一些恶意代码进行提取抽象, 通过匹配模块对静态源代码进行检测. 然而, 这种方法应用范围有限、漏报率高. 近年来, 智能合约漏洞检测方法主要有 5 种, 包括形式化验证法[56−58]、符号执行法[59−61]、模糊检测法[62−64]、中间表示法[65]和深度学习法[66−70].

```
1   contract  Rubixi {
2     uint private balance=0;
3     uint private collectedFees=0;
4     uint private feePercent=10;
5     uint private Order=0;
6     uint private pyramidMultiplier=300;
7     address private creator;
8     struct Participant {    uint payout;   }
9     Participant [] private participants;
10
11    function  Rubixi(){ creator=msg.sender; }
12    function (){ addPayout(); }
13
14    function  collectAllFees () onlyowner {
15      if (collectedFees==0) throw;
16       creator .send(collectedFees);
17      collectedFees=0;
18    }
19
20    function  addPayout  () private {
21      uint fee = feePercent;
22      participants.push(Participant(msg.sender       ,
23                          (msg.value*pyramidMultiplier)/100));
24      if (participants.length==10)
25         pyramidMultiplier=200;
26      else if (participants.length==25)
27         pyramidMultiplier=150;
28      balance += (msg.value*(100-fee)) / 100;
29       collectedFees += (msg.value * fee) / 100;
30      while (balance>participants[Order].payout) {
31         uint payoutT  oSend = participants[Order].payout;
32         participants[Order].etherAddress.send(payoutT       oSend);
33         balance -= participants[Order].payout;
34         Order += 1;
35       }
36     }
37  }
```

图 7　智能合约庞氏骗局 Rubixi 案例

2.1　形式化验证法

形式化验证是智能合约安全分析的重要技术之一. 通过形式化语言, 把合约中的概念、判断、推理转化成形式化模型, 从而消除合约中的歧义性和不通用性, 然后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. 目前, 形式化验证技术已经在核电、航天等高安全要求的领域取得了成功应用, 通过将形式化方法应用于智能合约漏洞检测, 可以使得智能合约的生成和执行有规范性约束, 从而保证合约的可信性和可靠性.

常见的形式化验证方法包括模型检测和演绎验证: 模型检测列举出所有可能的状态并逐一检验, 这种方法的基本思想是, 通过状态空间搜索来确认合约是否具有相应的性质; 演绎证明基于定理证明的思想, 采用逻辑公式描述系统及其性质, 通过一些推理规则证明系统具有某些性质. 简单来讲, 形式化验证通过数理逻辑证明方式, 验证代码在规范描述的前提下满足某些特性. 目前, 基于形式化验证的智能合约漏洞分析方法有以下 5 种.

(1) *F** framework[71].

*F**是一种形式化验证方法, 通过将 EVM 字节码的语义形式化并把 EVM 字节码编译成 Ocaml 形式, 最终将智能合约源码和字节码转化成函数编程语言 *F**, 以便分析和验证合约的安全性和功能正确性, 从而成功检测以太坊智能合约漏洞. 图 8 中描绘了 *F** framework 的总体架构和检测流程, 它实现了 Solidity*和 EVM*这 2 个模块来验证 Solidity 源代码和 EVM 字节码之间的功能等效性与正确性.

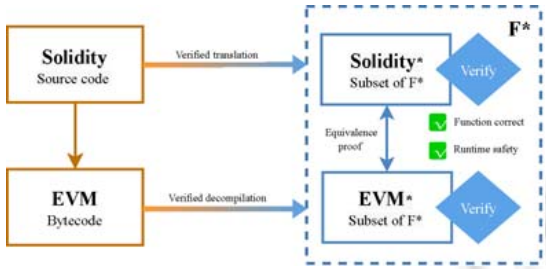

图 8 *F** Framework 总体架构和检测流程图

(2) KEVM framework[72].

KEVM 是一种形式化分析的框架, 它利用 *K* 框架构建了基于 EVM 字节码栈的可执行形式化规范, 提供了 EVM 的规范、参考解释器以及用于程序分析和验证的工具.

(3) Isabelle/HOL[73].

Isabelle/HOL 是一种 EVM 字节码级别的形式化验证方法, 其通过将字节码序列构造成直线代码块或将合约拆分成基本块, 并在此基础上创建逻辑程序进行推理验证.

(4) ZEUS[74].

ZEUS 是一种静态分析工具, 其基于形式化验证的方法能够正确且公平地分析智能合约. ZEUS 利用抽象解释和符号模型检查以及约束语句来快速验证合约的安全性, 支持多种智能合约漏洞的检测, 例如可重入漏洞、整数溢出漏洞、异常处理漏洞等.

(5) VaaS[75].

VaaS 是基于形式化验证方法的“一键式”智能合约安全检测平台, 它可以自动检测智能合约中 10 大项 32 小项的常规安全漏洞, 并且能够精准定位风险代码位置以及给出修改建议.

2.2 符号执行法

符号执行法的主要思想是, 将代码中的变量符号化. 通过符号化程序输入, 符号执行能够为所有的执行路径维护一组约束; 执行之后, 约束求解器将用于求解约束并确定导致该执行的输入; 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. 符号执行法应用于智能合约漏洞检测的执行过程为: 首先, 将合约中的变量值符号化; 然后逐条解释执行程序中的指令, 在解释执行过程中更新执行状态、搜集路径约束, 以完成程序中所有可执行路径的探索并发现相应的安全问题. 目前, 基于符号执行的智能合约漏洞检测工具有以下 6 种.

(1) Oyente[76]

Oyente 是最早的智能合约漏洞静态检测工具之一, 其在合约控制流图的基础上, 利用符号执行的方法检测智能合约漏洞. Oyente 以智能合约字节码和以太坊状态作为输入, 模拟 EVM 并且遍历合约的不同执行路径, 其支持检测的漏洞类型包括可重入漏洞、异常处理漏洞、交易顺序依赖漏洞等. Oyente 共有 4 个模块: CFGBuilder, Explorer, CoreAnalysis 和 Validator, 具体架构和检测流程如图 9 所示.

(2) Maian[77]

Maian 是一种基于符号分析的智能合约分析工具, 它通过长序列的合约调用过程来发现安全漏洞. 区别于一般的合约分析工具, Maian 只专注于 3 种类型的合约漏洞, 即资产无限期冻结的合约漏洞(greedy)、易泄露资产给陌生账户的合约漏洞(prodigal)、合约可以被任何人随意销毁的漏洞(suicidal).

(3) Securify[78]

Securify 是一种用于以太坊智能合约的静态安全分析器, 具有可伸缩、完全自动化、准确率高等特性, 其通过分析合约的依赖图以及从代码中提取精确的语义信息来检查合约的合规性与安全漏洞.

(4) Mythril[79]

Mythril 是一种智能合约静态分析工具, 其使用概念分析、污点分析以及控制流验证来检测以太坊智能合约中常见的漏洞类型, 包括可重入漏洞、整数溢出漏洞、异常处理漏洞等.

(5) TeEther[80]

TeEther 是一种智能合约静态分析工具, 区别于一般的漏洞检测工具, 它考虑了智能合约漏洞自动识别以及合约生成方法, 并通过分析合约字节码查找关键的执行路径, 以检测合约的安全问题.

(6) Sereum[81]

Sereum 是一种专注于智能合约可重入漏洞的新颖检测方案, 该解决方案利用动态污点跟踪监视智能合约执行过程中的数据流, 从而自动检测并防止状态不一致的情况, 有效地检测了可重入攻击.

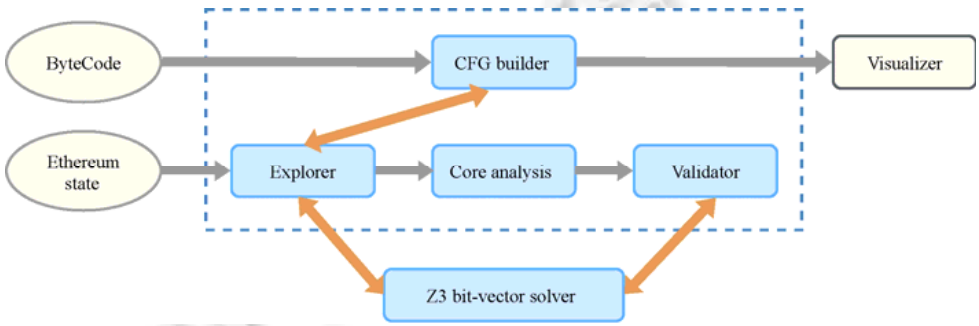

图 9 Oyente 总体架构和检测流程图

2.3 模糊测试法

模糊测试是当前流行的漏洞检测技术之一. 从概念上讲, 模糊测试从目标应用程序中生成大量正常和异常的测试用例, 尝试将生成的用例提供给目标应用程序, 并监视执行状态中的异常结果以发现安全问题. 与其他技术相比, 模糊测试具有良好的可扩展性和适用性, 可以在没有源代码的情况下执行. 目前, 运用在智能合约漏洞检测中的模糊检测方法有以下 3 种.

(1) ContractFuzzer[82]

ContractFuzzer 是第一个基于模糊测试的以太坊智能合约安全漏洞的动态分析方法, 其基于智能合约 ABI 规范生成模糊测试用例, 并定义测试方案来检测安全漏洞. ContractFuzzer 对 EVM 进行配置, 并记录智能合约运行时的行为. 通过分析这些日志并检测漏洞. 图 10 具体描述了 ContractFuzzer 的总体架构和检测流程.

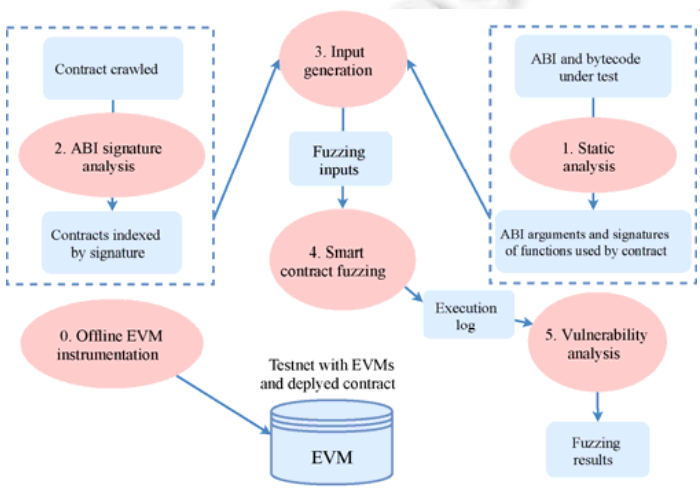

图 10 ContractFuzzer 总体架构和检测流程图

(2) Regurad[83]

Regurad 是一种专注于智能合约可重入漏洞的模糊测试分析器, 其通过迭代生成随机且多样化的测试用例, 对智能合约执行模糊测试, 从而在合约的执行过程中进行跟踪, 进一步地动态识别可重入漏洞.

(3) ILF[84]

ILF 是基于神经网络的智能合约模糊测试器, 它利用符号执行引擎生成有效的测试和调用序列, 以指导神经网络模型的特征学习, 从而实现有效的漏洞检测.

2.4 中间表示法

智能合约本质上是一种计算机程序,其区别于其他语言的方面在于: (1) 智能合约有其独特的 Fallback 函数; (2) 智能合约能操纵数字货币转账和交易. 因此, 相比于其他程序来说, 智能合约语言的分析更复杂, 安全漏洞造成的损失也更大. 为了能够准确地分析智能合约, 研究者们通过将智能合约源码或字节码转换成具有高语义表达的中间表示(intermediary representation, IR), 然后对合约的中间表示进行分析以发现安全问题. 目前, 利用中间表示法的智能合约分析工具有以下 6 种.

(1) Slither[85]

Slither 是一种以太坊智能合约的静态分析框架, 它将智能合约 Solidity 源代码转换为 SlithIR 的中间表示, SlithIR 使用静态单一分配(SSA)形式和精简指令集来简化合约分析过程, 同时保留了 Solidity 源代码转换为 EVM 字节码时丢失的语义信息. 图 11 描述了 Slither 的核心检测流程, Slither 不仅能用于检测智能合约的常见漏洞, 并且能给出合约代码优化的建议.

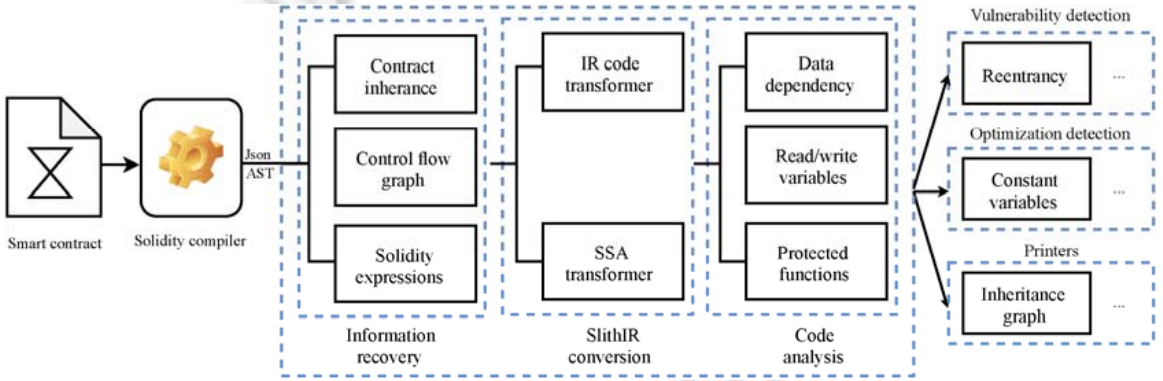

图 11 Slither 总体架构和检测流程图

(2) Vandal[86]

Vandal 是一种 EVM 字节码层面的智能合约静态分析工具, 它由一个分析管道和一个反编译器组成. 该反编译器执行抽象解释, 以逻辑关系的形式将字节码转换为更高级别的中间表示(IR), 然后使用新颖的逻辑驱动方法检测合约漏洞.

(3) Madmax[87]

Madmax 是一种专注于以太坊智能合约 *Gas* 相关的漏洞分析工具, 它基于 Vandal 实现了控制流分析和反编译器的程序结构性检测方法. 该工具同样将 EVM 字节码反编译成具有高语义信息的中间表示, 能够高精度地检测 *Gas* 相关的漏洞, 例如以太冻结漏洞等.

(4) Ethir[88]

Ethir 是一种基于 EVM 字节码层面的分析工具, 它基于 Oyente 生成控制流图(CFG), 然后将 CFG 转换为基于规则的中间表示(RBP), 从而分析和推断 EVM 字节码的安全属性.

(5) Smartcheck[89]

SmartCheck 是一种可扩展的智能合约静态分析工具, 它将智能合约 Solidity 源代码转换为基于 XML 的中

间表示, 然后利用 XPath 的模式来检测智能合约漏洞.

(6) ContractGuard[90]

ContractGuard 是一种面向以太坊智能合约的入侵检测工具, 它基于入侵检测系统(IDS)检测潜在攻击引发的异常控制流, 通过有效的上下文标记(context-tagged)无环路径实现入侵检测.

2.5 深度学习法

近年来, 深度学习在程序安全领域已经有越来越多的成功实践[66−70], 取得了令人鼓舞的效果. 深度学习技术的进步促进了各种安全检测方法的诞生, 对于新颖的安全漏洞类型, 深度学习方法具有良好的扩展性和适应性. 目前, 基于深度学习的智能合约漏洞检测方法有以下 5 种.

(1) SaferSC[91]

SaferSC 是第 1 个基于深度学习的智能合约漏洞检测模型, 其基于 Maian 划分的 3 类合约漏洞, 实现了比 Maian 更高的检测准确率. 此外, SaferSC 在智能合约操作码(operation code, opcode)层面进行分析, 利用 LSTM 网络构建了以太坊操作码序列模型, 实现了精准地智能合约漏洞检测.

(2) ReChecker[92]

ReChecker 是第 1 个基于深度学习的智能合约可重入漏洞检测方法, 其通过将智能合约 Solidity 源代码转换为合约块(contract snippet)的形式, 捕获了合约中基本的语义信息和控制流依赖信息. ReChecker 利用双向长短期记忆模型(bidirectional long short-term memory, BLSTM)和注意力机制(attention)[93]实现了以太坊智能合约可重入漏洞的自动化检测.

(3) DR-GCN[94]

DR-GCN 是第 1 个利用合约图(contract graph)的方式来检测智能合约漏洞, 其将智能合约源代码转换为具有高语义表示的合约图结构, 并利用图卷积神经网络构建了安全漏洞检测模型. DR-GCN 支持 2 个平台(即以太坊和维特链)的智能合约漏洞分析, 能够检测可重入漏洞、时间戳依赖漏洞以及死循环漏洞.

(4) TMP[94]

TMP 通过将智能合约中的关键函数和关键变量转换成具有高语义信息的核心结点来构建合约图, 关键的执行方式转换成控制流和数据流依赖的有向时序边. TMP 在 DR-GCN 的基础上考虑了合约图中边的时序信息, 并利用时序图神经网络[95,96]实现了相应的智能合约漏洞检测. 如图 12 所示, TMP 首先将源代码转换为图, 然后对合约图做归一化处理, 最终通过时序图神经网络模型输出漏洞检测结果.

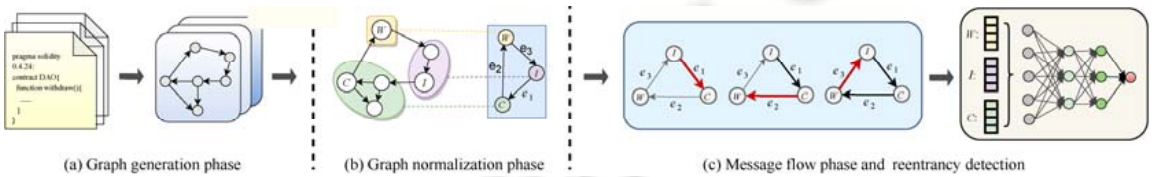

图 12 TMP 总体架构和检测流程图

(5) ContractWard[97]

ContractWard 从智能合约操作码中提取 bigram 特征, 利用多种机器学习算法和采样算法进行智能合约漏洞检测, 其总共支持 6 种漏洞类型, 包括可重入漏洞、整数溢出漏洞以及时间戳依赖漏洞.

3 智能合约漏洞检测工具分析和比较

本节首先概述了总共 25 种智能合约检测工具; 然后比较了不同检测工具的可检测漏洞类型; 接着评估了智能合约漏洞检测工具的性能, 包括准确率、*F*1-*Score* 和平均检测时间; 最后分析了当前漏洞检测方法的局限性, 并讨论了它们存在的研究挑战以及改进思路.

3.1 检测工具概览

表 2 概述了不同的智能合约漏洞检测工具.

- 第 1 列介绍了 5 种常见的智能合约漏洞检测方法;
- 第 2 列列举了相对应的检测工具;
- 第 3 列阐述了每种检测工具的自动化程度: 全自动指端到端的解决方法, 即输入一个合约输出对应的漏洞检测结果; 半自动指在检测过程中需要手动定义相关的合约属性等, 例如形式验证法使用定理证明智能合约的安全性, 由于这些证明是半自动化的, 因此形式化验证法需要大量的人工操作对智能合约进行分析和反馈;
- 第 4 列概括了相应的检测工具支持的智能合约语言和形式;
- 第 5 列说明了检测工具的开源程度,并且给出了相应的开源地址;
- 最后一列对应的是具体的参考文献.

表 2 智能合约漏洞检测工具概览

检测方法	检测工具	自动程度	语言支持	开源地址	文献
形式化验证法	*F** framework	半自动	EVM, Ocaml	未开源	[71]
	KEVM framework	半自动	Solidity, EVM	https://github.com/kframework/evm-semantics	[72]
	Isabelle/HOL	半自动	EVM, Ocaml	https://github.com/pirapira/eth-isabelle	[73]
	ZEUS	全自动	Solidity, EVM	未开源	[74]
	VaaS	全自动	Solidity, EVM	未开源	[75]
符号执行法	Oyente	全自动	Solidity, EVM	https://github.com/melonproject/oyente	[76]
	Maian	全自动	Solidity, EVM	https://github.com/MAIAN-tool/MAIAN	[77]
	Securify	全自动	Solidity, EVM	https://github.com/eth-sri/securify2	[78]
	Mythril	全自动	Solidity, EVM	https://github.com/ConsenSys/mythril	[79]
	TeEther	全自动	Solidity, EVM	https://github.com/nescio007/teether	[80]
	Sereum	全自动	Solidity, EVM	未开源	[81]
模糊测试法	ContractFuzzer	全自动	Solidity, EVM	https://github.com/gongbell/ContractFuzzer	[82]
	Regurad	全自动	Solidity, EVM	未开源	[83]
	ILF	全自动	Solidity, EVM	未开源	[84]
中间表示法	Slither	全自动	Solidity, EVM	https://github.com/crytic/slither	[85]
	Vandal	全自动	Solidity, EVM	https://github.com/usyd-blockchain/vandal	[86]
	Madmax	全自动	Solidity, EVM	https://github.com/nevillegrech/MadMax	[87]
	Ethir	全自动	Solidity, EVM	https://github.com/costa-group/ethIR	[88]
	Smartcheck	全自动	Solidity, XML	https://github.com/smartdec/smartcheck	[89]
	ContractGuard	全自动	Solidity	https://github.com/contractguard/experiments	[90]
深度学习法	SaferSC	全自动	Solidity, EVM	https://github.com/wesleyjtann/Safe-SmartContracts	[91]
	ReChecker	全自动	Solidity	https://github.com/Messi-Q/ReChecker	[92]
	DR-GCN	全自动	Solidity	https://github.com/Messi-Q/GraphDeeSmartContract	[94]
	TMP	全自动	Solidity	https://github.com/Messi-Q/GGNNSmartVulDetector	[94]
	ContractWard	全自动	Solidity	未开源	[97]

根据表 2 中的统计结果, 我们进行了以下分析.

- 相较于其他的检测方法, 形式化验证检测工具的自动化程度偏低且开源性较低;
- 基于符号执行和中间表示的漏洞检测工具类型比较多, 它们能够执行全自动的漏洞检测, 并且绝大多数都能找到开源代码;
- 基于模糊测试的检测工具比较少, 其原因可能是模糊检测的动态执行方法操作比较复杂和繁琐, 且由于测试用例的随机性, 其无法达到理想的测试路径覆盖率;
- 基于深度学习的智能合约漏洞检测大多数更专注于智能合约 Solidity 源代码的层面, 它们可以实现全自动化的检测, 并且开源程度较好.

3.2 检测工具性能分析

3.2.1 智能合约漏洞检测评估

表 3 统计了 25 种智能合约漏洞检测工具及它们支持的可检测漏洞类型, 总共包括 3 大类 15 种智能合约漏洞类型.

表 3 智能合约漏洞检测工具可检测漏洞类型概览

检测工具 \ 漏洞类型	Solidity 代码层								EVM 执行层				区块链系统层		
	可重入	整数溢出	权限控制	异常处理	拒绝服务	类型混乱	未知函数调用	以太冻结	短地址	以太丢失	调用栈溢出	Tx. Origin	时间戳依赖	区块参数依赖	交易顺序依赖
*F** framework	√			√			√						√	√	√
ZEUS	√	√											√	√	√
VaaS	√	√		√	√		√					√	√	√	
Oyente	√	√		√							√		√		√
Maian	√							√		√	√				
Securify	√			√		√	√	√	√	√	√	√	√	√	√
Mythril	√	√		√	√					√		√	√	√	
TeEther	√						√	√							
Sereum	√														
ContractFuzzer	√			√			√	√					√	√	
Reguard	√														
ILF	√			√				√		√			√	√	
Slither	√					√	√	√			√	√	√		
Vandal	√			√				√		√		√			√
Madmax		√						√							
Ethir	√			√									√		√
Smartcheck	√	√	√		√		√	√				√	√		
ContractGuard	√	√			√	√	√		√			√	√	√	√
SaferSC	√							√		√	√				
ReChecker	√														
DR-GCN	√												√		
TMP	√												√		
ContractWard	√	√									√		√		√
KEVM framework	KEVM framework 和 Isabelle/HOL 提供了验证条件用于合约程序分析和形式化验证的方法, 验证了合约执行时安全性,功能正确性和程序逻辑合理性,但未用作具体的智能合约漏洞检测														
Isabelle/HOL															

具体的分析从以下两个角度来看.

(1) 漏洞类型检测情况

- 从 Solidity 代码层的检测结果来看, 大多数检测工具都支持可重入漏洞的检测, 这可能是因为智能合约历史上最著名的“The DAO”事件由可重入漏洞引起, 因此研究者和开发者大多会关注此类漏洞的分析. 另外, 很多检测工具也支持整数溢出、以太冻结、异常处理和未知函数调用等漏洞的检测, 这些安全漏洞都曾引起过重大的合约攻击事件, 例如美链 BEC 合约整数溢出事件、Parity 多签名钱包冻结事件等. 值得指出的是: 支持权限控制、拒绝服务和类型混乱等漏洞检测的工具比较少, 这 3 类漏洞发生的频率较少且易于防范;
- 从 EVM 执行层的检测结果来看, 由于短地址漏洞发生概率小且容易校验, 因此支持该漏洞检测的工具比较少. 另外, 由于形式化验证法和深度学习法更关注于 Solidity 代码层的漏洞分析, 因此它们都缺乏对 EVM 执行层漏洞检测的考虑, 例如, ZEUS, *F** framework, ReChecker, DR-GCN 和 TMP 都不支持该层面的漏洞检测;
- 从区块链系统层的检测结果来看, 时间戳依赖是比较常见的漏洞且易于检测, 大多数检测工具都支持对时间戳依赖漏洞的检测. 值得注意的是, *F** framework, ZEUS 和 Securify 可以支持区块链层所有漏洞类型的检测.

综上所述, 尽管目前大多数智能合约漏洞都有相应的检测工具支持, 但有一些易于检测的合约漏洞反而

支持的漏洞检测工具很少, 例如权限控制和短地址漏洞, 它们只有2种漏洞检测工具可以检测. 这两类漏洞发生的概率较小, 若是发生此类漏洞, 造成的损失也是不可估计的. 因此, 漏洞类型的全面覆盖, 仍然是当前检测工具面临的关键挑战. 随着智能合约数量的跳跃式增长, 相应的漏洞数量和类型也在增加, 使用漏洞检测工具对合约进行更全面以及可拓展的漏洞检测, 是亟需解决的关键问题.

(2) 检测工具的支持程度

根据表 3 中的统计结果, 形式化验证工具支持检测的智能合约漏洞类型较少, 其中, KEVM 和 Isabelle/HOL 提供了程序分析与形式化验证的方法来验证合约执行时的安全性、功能正确性和程序逻辑合理性, 但未用作具体的智能合约漏洞检测. 此外, *F** framework, ZEUS 和 VaaS 几乎都不支持 EVM 执行层的漏洞检测. 形式化验证法大多使用数学定理证明及复杂机制进行验证, 因此, 使用形式化验证法分析智能合约并非易事.

基于符号执行的检测工具大多能支持较多类型的合约漏洞检测. 例如, Oyente, Securify 和 Mythril 分别可以支持 6 种、12 种和 8 种漏洞类型的检测, 其中, Securify 是 25 种检测工具中支持最多漏洞类型检测的工具. Maian, TeEther 和 Sereum 支持的漏洞类型较少, 其中, Maian 主要是对其划分的 3 类独特的漏洞进行检测(即 Greedy, Prodigal, Suicidal), 而 Sereum 只专注于可重入漏洞检测.

相比较于其他方法, 基于模糊测试法的 ContractFuzzer, Reguard 和 ILF 支持检测的漏洞类型较少, 其中, ContractFuzzer 和 ILF 支持 6 种常见的漏洞检测, 而 Reguard 只专注于可重入漏洞检测.

基于中间表示法的检测工具取得了不错效果, 其中, Vandal, Slither, Smartcheck 和 ContractGuard 支持检测的漏洞类型较多, 分别能检测 6 种、7 种、8 种、10 种漏洞类型, 而 Madmax 和 Ethir 分别只能检测 2 种和 4 种漏洞.

基于深度学习的方法支持检测的漏洞类型较少, SaferSC 和 ContractWard 分别能够支持 4 种和 5 种漏洞类型的检测, 而 ReChecker 只专注于可重入漏洞的检测; DR-GCN 和 TMP 分别支持 2 种漏洞类型的检测, 即可重入漏洞和时间戳依赖漏洞.

综上所述, 各类漏洞检测工具涵盖的漏洞类型仍然是不全面的, 其中大多数还只是能检测低级别的安全违规行为和漏洞, 缺少对合约可执行路径的判断与验证, 难以检测由外部合约调用导致的安全问题. 因此, 在当前智能合约数量巨大且日益增长的背景下, 使用单一的检测工具对合约漏洞进行全面验证, 仍是有挑战性的难题.

3.2.2 检测工具性能评估

表 4 详细比较分析了不同漏洞检测工具的性能. 我们从前文所述的 5 种智能合约漏洞检测方法中选择相应的具有代表性检测工具, 分别是 VaaS, Oyente, Smartcheck, ContractFuzzer 和 TMP, 并且从以太坊官方网站 Etherscan[23]中随机选取了 300 个真实的以太坊智能合约作为测试样本, 就可重入漏洞、整数溢出漏洞、时间戳依赖漏洞这 3 种合约漏洞, 从准确率、*F*1-*Score* 和平均检测时间这 3 个维度对检测工具进行了性能评估与比较.

根据表 4 中选择的漏洞类型以及相应的检测工具, 我们分别从不同的智能合约漏洞类型以及不同类别的漏洞检测方法两个角度进行讨论与分析.

- 从不同的智能合约漏洞类型来看. 当前的 5 类检测方法都支持检测的智能合约漏洞只有可重入漏洞和时间戳依赖漏洞, 这进一步表明, 不同类别的漏洞检测方法可支持检测的漏洞类型仍然不全面;
- 从不同类别的漏洞检测方法来看. 当前, 很多漏洞检测工具仍无可用的开源代码, 且存在源代码的检测工具很多处于半开源状态或只给出实验结果. 例如, 形式化验证法中的几个检测工具大多没有开源, 因此本文选择 VaaS 作为对比; 符号执行法中能够运行的工具有 Oyente, Maian, Securiy 和 Mythril, 因此本文选择 Oyente 作为对比; 模糊测试法中的 Reguard 和 ILF 也没有开源, 因此本文选择 ContractFuzzer 作为对比; 中间表示法中能够运行的工具有 Slither, Vandal 和 SmartCheck, 因此本文选择 SmartCheck 作为对比; 深度学习法中能够运行的工具有 ReChecker, DR-GCN 和 TMP, 其中, TMP 是它们中准确率最高的漏洞检测工具, 因此本文选择它作为对比.

表 4 检测结果比较:准确率、*F*1-*Score* 和平均检测时间

漏洞类型	检测工具	准确率(%)	*F*1-*Score*(%)	平均检测时间(s)
可重入漏洞	VaaS	82.54	73.95	159.4
	Oyente	61.62	44.96	29.6
	Smartcheck	52.97	30.10	14.5
	ContractFuzzer	67.89	52.67	352.2
	TMP	84.48	74.15	2.5
整数溢出漏洞	VaaS	86.80	80.10	159.4
	Oyente	66.85	59.64	29.6
	Smartcheck	58.48	54.96	14.5
	ContractFuzzer	–	–	–
	TMP	–	–	–
时间戳依赖漏洞	VaaS	89.20	82.46	159.4
	Oyente	59.45	41.53	29.6
	Smartcheck	51.32	40.18	14.5
	ContractFuzzer	68.08	52.49	352.2
	TMP	83.45	79.19	2.1

综上所述, 就当前漏洞检测工具的开源程度以及不同漏洞检测方法可支持检测的漏洞类型来看, 对不同漏洞的针对性检测方法进行对比仍是有难度的, 亟待进一步地研究与探讨.

智能合约漏洞检测工具的性能评估主要包括以下 3 个方面.

(1) 准确率

为了评估检测工具的优劣, 首先关注最常见的评估指标, 即准确率(accuracy). 准确率在某种意义上可以判定一个分类器是否有效, 能够客观地反映各类检测工具最直接的检测效果. 漏洞检测其实是一个二分类问题, 即检测工具判断合约是否存在某类漏洞. 对于二分类问题, 通常将真实情况与检测结果的匹配结果作为重要的评估指标, 其中包括以下 4 种情况.

- 真阳性(true positive, TP): 对于某一合约, 检测工具的检测结果是有漏洞且真实情况也是有漏洞, 即检测结果是正确的;
- 假阳性(false positive, FP): 对于某一合约, 检测工具的检测结果是有漏洞而真实情况却是无漏洞, 即检测结果存在误判;
- 假阴性(false negative, FN): 对于某一合约, 检测工具的检测结果是无漏洞而真实情况却是有漏洞,即检测结果存在漏判;
- 真阴性(true negative, TN): 对于某一合约, 检测工具的检测结果是无漏洞且真实情况也是无漏洞, 即检测结果是正确的.

显然, *TP*+*FP*+*FN*+*TN* 的结果是 300 个智能合约测试样本, 智能合约真实值和检测值的混淆矩阵(confusion matrix)如图 13 所示, 其中, 准确率的计算方法如公式(1)所示:

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

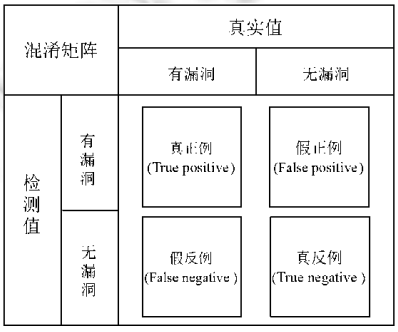

图 13 智能合约漏洞检测混淆矩阵

根据表 4 中的评估结果.

- 在可重入漏洞的检测上, 深度学习方法 TMP 实现了最高的准确率(84.48%); 形式化验证工具 VaaS 和模糊测试工具 ContractFuzzer 分别实现了 82.54%和 67.89%的准确率; 相比较来说, Oyente 和 Smartcheck 检测的准确率稍微不足, 分别只有 61.62%和 52.97%;
- 对于整数溢出漏洞: VaaS 的准确率高达 86.80%, 而 Oyente 和 Smartcheck 只有 66.85%和 58.48%的准确率, ContractFuzzer 和 TMP 则不支持该类漏洞的检测;
- 对于时间戳依赖漏洞: VaaS 和 TMP 实现了较高的检测准确率, 分别为 89.20%和 83.45%; 而 Oyente, Smartcheck 和 ContractFuzzer 检测的准确率很低, 分别只有 59.45%, 51.32%和 68.08%.

(2) *F*1-*Score*

*F*1-*Score* 是分类问题中重要的衡量指标, 它是精确率(precision)和召回率(recall)调和平均数, 常被用作一些分类问题的最终评估标准, 其计算方法如公式(2)–公式(4)所示:

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

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

$$F1\text{-}Score = 2 \cdot \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

根据表 4 的评估结果.

- 在可重入漏洞的检测上: TMP 实现了最高的 *F*1-*Score*(74.15%), 其次是 VaaS 的 73.95%, 其他几类检测工具的 *F*1-*Score* 相对较低;
- 在整数溢出漏洞的检测上, VaaS 则获得了 80.10%的 *F*1-*Score*, Oyente 和 Smartcheck 的 *F*1-*Score* 分别为 59.64%和 54.96%;
- 对于时间戳依赖漏洞的检测, VaaS 和 TMP 都获得了不错的 *F*1-*Score*, 分别为 82.46%和 79.19%.

(3) 平均检测时间

检测时间也是评估自动化检测工具的重要指标之一, 当前, 大多数检测工具审计时间较长导致漏洞分析的效率低下. 根据表 4 的评估结果: 形式化验证工具 VaaS 和模糊测试工具 ContractFuzzer 的平均检测时间较长, 分别为 159.4 s 和 352.2 s; 相比于来说, Oyente 和 Smartcheck 的平均检测时间为 29.6 s 和 14.5 s. 值得注意的是, TMP 对不同漏洞的检测时间是不同的: 对于可重入漏洞检测, 它的平均检测时间为 2.5 s; 而对于时间戳依赖漏洞, 它的平均检测时间为 2.1 s.

综合上述分析,我们对这 5 类检测工具进行了以下分析和总结.

- VaaS 在 3 种漏洞的检测上实现了比较高的准确率和 *F*1-*Score*, 但它的平均检测时间相对来说较长. VaaS 是“一键式”的形式化验证平台, 采用了多种形式化验证方法, 具有验证效率高、自动化程度高、精确率高等特点;
- Oyente 是基于符号执行的合约分析工具, 它对 3 种漏洞的检测效果仍有不足, 其通过简化合约中循环处理和基于规则匹配判断方法来提高效率, 但也因此导致很多漏报和误报情况;
- Smartcheck 是利用基于 XML 规则合约中间表示分析智能合约安全问题的工具, 然而, 由于它依赖于刻板简单的逻辑规则, 对于合约漏洞的检测存在很多误报情况, 从而导致了很低的准确率和 *F*1-*Score*. 值得一提的是: 由于其依赖于简单的逻辑规则, 相对来说花费的检测时间比较少;
- ContractFuzzer 是基于以太坊平台的智能合约安全漏洞模糊测试工具. 实验对比中, 它只支持可重入和时间戳依赖漏洞的检测. 由于模糊测试用例所能涵盖的系统行为有限, 其无法达到理想的路径覆盖率, 因此取得的效果并不好. 此外, 由于 ContractFuzzer 基于以太坊平台, 它在使用时还需要获取区块链网络的响应以执行动态验证, 因此执行一次检测需要花费大量时间;
- TMP 是新颖的基于图神经网络的智能合约漏洞检测模型, 具有可扩展性高、准确率高、批量检测等

特点. 实验对比中, TMP 支持可重入和时间戳依赖漏洞的检测, 并且取得了令人鼓舞的效果. 同时,由于利用已经训练好的检测模型, TMP 的平均检测时间非常少, 极大地提高了智能合约漏洞检测的效率.

3.3 检测方法评估与改进思路

3.3.1 局限性分析

当前的智能合约漏洞检测方法虽然能够较好地分析智能合约漏洞, 但它们仍然存在固有的局限性. 本节分别对前述的 5 类漏洞检测方法进行了具体地分析与讨论.

(1) 形式化验证法通过一些数学手段在智能合约生命周期内对其进行推导与证明, 需要交互式的验证与判断, 因此自动化程度较低, 并且依赖人工二次核验, 导致其无法较好地兼容 EVM 执行层漏洞. 同时, 由于形式化验证手段依赖于严谨的数学推导与验证, 它无法执行动态分析, 并且缺少对合约中可执行路径的检测与判断, 从而导致了较高的误报率和漏报率. 例如: *F** framework 和 KEVM 将智能化合约字节码转化为形式化模型, 以验证合约代码中的各种属性来检测漏洞, 它们仍然是半自动的; ZEUS 和 VaaS 较好地实现了全自动形式化验证, 但它们检测出来的漏洞不一定存在可达的执行路径, 因此存在较高的误报率;

(2) 符号执行法利用符号替代具体的执行程序指令、搜集路径约束、遍历合约程序中所有可执行路径, 这一方法虽然有效地改善了符号执行的检测效果, 但也显著地提高了漏洞分析过程中的计算资源和时间开销, 并且无法彻底解决状态空间爆炸与执行路径指数级增长等问题. 例如, Oyente 和 Maian 为了防止路径爆炸问题, 限制循环条件的次数来提高效率, 但也导致了较高的漏报率. 另外需要指出的是: 很多符号执行法其实并不能做到完全自动化, 同样需要人工协助与反馈;

(3) 模糊测试法很大程度上依赖于精心设计的测试用例, 其在动态执行过程中监测合约的异常行为以发现漏洞. 然而, 模糊测试对导致漏洞的具体语义代码洞察有限, 这使得其很难追踪到存在漏洞的确切代码位置. 例如, ContractFuzzer 虽然有效地降低了误报率, 但由于其测试用例生成的随机性, 无法达到理想的路径覆盖率, 因此很难找出所有的潜在威胁;

(4) 中间表示法通过将原始的智能合约转换为相应的中间表示, 利用控制流、数据流以及污点分析等手段对合约进行审查, 但它们往往依赖于预定义的语义规则或分析列表, 从而无法检测出智能合约复杂的业务逻辑问题, 且极易产生误报. 另外, 它们无法对合约中可能存在的执行路径进行遍历. 例如: Slither 的中间表示 SlithIR 依赖于固定的语义规则且缺乏形式化语义, 这限制其执行更详细的安全分析, 因此无法准确检测相应的漏洞; Smartcheck 依赖于刻板且简单的预定义规则, 因此无法检测出一些由污点分析或动态执行验证的合约漏洞;

(5) 深度学习法通常对智能合约进行预处理, 以构建有利于模型学习的数据集. 例如, 文献[92]利用 LSTM 模型处理智能合约源码序列片段, 文献[94]通过 GNN 模型处理智能合约图. 然而, 这些方法一方面无法突出智能合约源码中的关键变量而造成语义建模不足, 导致检测结果不理想, 并且缺少对 EVM 执行层漏洞的考虑; 另一方面, 由于神经网络的“黑箱性”, 因此其大多数情况下可解释性较差, 即无法像传统的检测工具一样给出可能存在漏洞的确切位置或代码行. 例如, TMP 是一个端到端的漏洞检测模型, 以合约测试集为输入, 输出相应的漏洞检测结果, 它的中间处理流程是黑盒的, 因此其可解释性较差, 且检测结果难以令人信服.

3.3.2 研究挑战与改进思路

针对现有的智能合约漏洞检测方法存在的问题, 本节分别就它们面临的研究挑战以及改进思路进行了讨论与分析, 主要围绕以下 5 个方面展开.

(1) 提高形式化验证自动化程度, 扩展应用范围

现有的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. 目前的形式化验证方法用数学推演来分析可能存在复杂漏洞的合约, 虽然有效地维护了智能合约的安全

但其难度很高. 另外, 使用形式化验证技术对更广泛的智能合约漏洞进行检测仍然面临着严峻挑战. 未来的研究应针对不同的漏洞检测目标定制对应的验证规范描述, 突破其不适应大规模合约及多漏洞类型等技术限制, 并扩展形式化验证的应用范围, 从验证一般功能属性和安全属性、检测常见漏洞到逐步实现商业场景中复杂业务逻辑的智能合约漏洞分析与验证[44].

(2) 提取符号执行重点路径, 缩减路径空间

符号执行当前面临的最主要挑战就是状态空间爆炸和执行路径指数级增长的问题. 未来可行的方法是: 结合现有符号执行工具的审计经验以及漏洞分析情况, 寻找智能合约中易产生漏洞的高危指令, 如 SUICIDE, CALL, DELEGATECALL, ORIGIN, ASSERT 等[41], 定义涉及这些操作码的路径为重点路径. 为了提高漏洞检测效率, 在具体的实践中不必对所有可能的执行路径进行检查, 仅符号执行关注的重点路径并进行漏洞验证, 从而有效地缩减路径空间.

(3) 完善测试用例, 改进模糊测试工具

相较于传统应用程序, 智能合约存在很多独特的变量和函数, 这给面向智能合约的模糊测试带来了全新的挑战.

首先, 由于智能合约全局状态与调用序列的特性, 导致生成有效的测试用例变得极为困难. 面向传统程序的模糊测试方案生成测试用例时仅考虑单测试用例, 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. 智能合约中的漏洞有较大差别, 它们既不会造成程序崩溃, 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, 这也为智能合约的漏洞检测带来了很大挑战[43].

具体而言, 模糊测试依赖于其测试用例的健壮性, 因此需要进一步改进现有的测试用例生成算法, 例如使用多目标优化算法. 另外, 模糊测试也可以考虑结合其他检测方法来提高检测效率, 采用静态分析、符号执行、模糊测试相结合的策略. 例如, 使用静态分析提取关键路径, 通过符号执行生成测试用例, 从而提高模糊测试的效率.

(4) 优化中间表现形式, 结合动态执行

中间表示法通常将智能合约源码或字节码转换为特有的中间表示形式来检测特定的几类漏洞, 同时, 它们也依赖于专家定义的漏洞规则. 但这些规则往往比较刻板简单, 且容易被攻击者绕过. 因此, 为了提高这类漏洞检测方法的拓展性及适应性, 研究者们应当专注于让智能合约的中间表现形式具备更好的通用性, 在检测多种类型漏洞的同时, 兼顾不同智能合约的统一表示形式. 另外, 静态分析与动态执行相结合, 是能够提高漏洞检测准确率的有效方法. 当前, 基于中间表示的检测方法大多是静态分析, 缺乏使用动态执行进行验证. 因此, 这既是中间表示法目前面临的关键挑战, 也是未来研究攻关的主要方向.

(5) 加强深度学习可解释性, 融合专家规则

现有的基于深度学习的智能合约漏洞检测方法大多是黑盒的检测流程, 它们通过训练漏洞检测模型来给出最终的漏洞检测结果. 由于深度学习模型固有的“黑箱性”, 其内部的具体工作状态和处理过程是不透明的, 因此缺乏对漏洞检测结果的合理解释(如标注可能存在漏洞的确切代码位置或代码行), 从而使得检测结果无法令人信服. 因此, 深度学习模型应该考虑在输出漏洞检测结果的同时进一步给出其合理的可解释性说明. 另外值得提出的是: 传统检测工具中定义的专家规则也是分析合约漏洞的利器, 未来的深度学习模型应当考虑融合传统检测方法中漏洞相关的专家规则, 从而更好地提高漏洞检测的准确率.

4 总结与展望

智能合约作为区块链最成功的应用之一, 极大地扩展了区块链的应用场景和现实意义, 在区块链生态环境中起着至关重要的作用. 随着区块链技术日益成熟以及智能合约发展与广泛应用, 智能合约的安全性和可靠性已经成为了新的研究焦点. 本文总结了以太坊智能合约中常见的安全漏洞类型, 并还原了智能合约安全漏洞历史上的典型案例. 目前, 研究者们已经提出了一系列的智能合约漏洞检测方法来检测智能安全漏洞,

本文将现有的智能合约漏洞检测方法归纳总结为形式化验证法、符号执行法、模糊测试法、中间表示法、深度学习法这 5 种, 并详细介绍和分析了各种方法的原理和特征, 对比评估了具有代表性的智能合约自动化检测工具的可检测漏洞类型以及检测性能, 讨论分析了现有的智能合约漏洞检测方法的局限性和改进思路.

智能合约自动化检测方法能够较为准确地、一键式地应对区块链网络中层出不穷的智能合约漏洞类型, 减少了人工核验和专家分析可能造成的误报和漏洞情况, 因此, 采用准确且高效的智能合约自动化检测工具来解决智能合约的漏洞挖掘与检测问题是有重要意义的工作和研究. 本文分析了现有的研究方法, 发现在智能合约安全漏洞检测领域中虽然已经取得了突破性进展和令人鼓舞的成果, 但是现阶段的检测方法仍然不够完善, 面临着以下待解决的关键问题.

- 漏洞检测准确率低. 目前, 大多数智能合约检测工具仍然存在很高的误报率和漏报率, 以本文中测试的 300 个智能合约为例, 在 3 种智能合约漏洞的检测上, 除了 VaaS 和 TMP 的准确率能达到 80%以上, 其他 3 种检测工具的准确率仅仅刚达到 60%左右, 这远不能满足当前智能合约漏洞类型多、合约数量大的应用场景. 因此, 提高智能合约漏洞检测工具的准确率是当前面临的关键挑战;
- 漏洞类型覆盖率低. 由于智能合约漏洞类型繁多且复杂, 大多数检测工具无法涵盖所有的漏洞类型. 以本文统计的 25 种检测工具和 15 种漏洞类型为例, 其中支持漏洞类型最多的工具是 Securify,能够检测 12 种安全漏洞类型. 然而大多数检测工具检测类型单一, 且只能验证低级别的安全漏洞,缺乏对合约整个执行过程中的监测和推断, 难以发现跨合约的漏洞问题. 因此, 完善检测工具对智能合约漏洞类型更全面的检测, 也是亟待解决的关键问题;
- 漏洞审计时间较长. 智能合约漏洞的及时审计也是保证合约安全的关键要素, 当前的检测工具漏洞挖掘效率低, 阻碍了智能合约的开发和使用. 例如, Mythril 的平均检测时间为 225.6 s, VaaS 大约为 159.4 s, 而 ContractFuzzer 大约需要 352.2 s. 面对着与日俱增的合约数量, 保证智能合约漏洞检测工具的漏洞审计效率, 也是需要解决的难题;
- 漏洞检测完全自动化. 智能合约工具完全自动化是保证漏洞检测效率的重要一环. 现有的检测工具还不能做到完全自动化, 例如形式化验证方法 *F**和 KEVM framework. 此外, 有些“一键式”的检测工具无法明确输出合约是否存在漏洞, 而要对检测出来的疑似漏洞进行手动分类, 例如 Securify 和 Smartcheck, 这极大地增加了智能合约检测过程中的工作量, 降低了检测效率. 因此, 如何实现更全面的自动化检测流程和方案, 需要进一步研究;
- 智能合约语言多样性. 现实世界中的编程语言类型繁多且更迭迅速, 目前, 支持编写智能合约的语言就有几十种(如 Solidity, Go, C, Java), 不同的语言有不同的语法和语义规则. 如何使智能合约漏洞检测工具能够适配大多数编程语言, 也存在一定的挑战和难度.

虽然智能合约漏洞检测现阶段的发展仍然存在很多挑战, 但是也恰逢新的机遇. 随着深度学习技术的发展, 近几年, 研究人员开始利用深度学习模型进行智能合约漏洞的检测, 并且取得了不错的进展. 下面结合深度学习和智能合约漏洞检测技术对未来的研究方向进行展望, 以供研究探讨.

- 构建统一且规范的智能合约漏洞数据集. 首先, 基于深度学习的智能合约漏洞检测方法能够取得突破性的进展必定依赖于统一且全面的智能合约漏洞数据集. 目前, 正是因为数据集贫乏且不规范,已有的深度学习方法(如 ReChecker, TMP)只能支持少数的合约漏洞检测. 因此, 只有基于统一规范的、涵盖漏洞类型全的漏洞数据集, 才能让深度学习模型发挥更好的效应, 更好地推动该领域的研究;
- 构建基于深度学习的动静态分析综合模型. 现阶段, 基于深度学习的智能合约安全漏洞检测工具刚刚起步(如 SaferSC, ReChecker, TMP), 它们在静态的源代码或字节码层面进行分析. 然而, 这种静态的分析方法会遗漏可能存在的执行路径; 同时, 由于缺少与外部合约动态的交互过程, 往往会导致出现漏报或误报的情况. 因此, 为了应对大规模应用场景的需求, 可以在构建深度学习模型的时候考虑结合动态执行和静态分析;
- 构建统一且可扩展的深度学习模型. 随着智能合约数量的爆炸式增长, 相应的安全漏洞类型也越来

越复杂且无法预料. 目前, 基于深度学习的智能合约漏洞检测方法仍然在已发掘的漏洞类型构建模型, 其是否能够快速适应新的漏洞类型还亟待研究. 本文认为: 应充分利用开源生态中丰富的智能合约安全漏洞来构建统一且可扩展的深度学习漏洞检测模型, 以应对层出不穷的智能合约漏洞;

- 构建基于深度学习的可解释性漏洞检测模型. 虽然基于深度学习的智能合约漏洞检测模型有效地提高了漏洞检测的准确率, 但它们仍然存在可解释性差的关键问题, 并且缺乏融合专家定义的经典漏洞规则. 因此, 为了使深度学习模型的检测结果更有说服力, 本文认为: 深度学习法一方面要进一步融合经典的专家规则, 另一方面要准确地给出漏洞检测的可解释性说明;
- 构建统一的漏洞检测工具性能评估体系. 根据已经出现的智能合约安全事件以及相关的合约漏洞审计经验, 综合考虑漏洞检测的漏报率、误报率、检测时间、可检测漏洞类型等因素, 最终构建统一的漏洞检测工具性能评价体系, 对已有的相关工具进行对比分析, 以验证其有效性, 并为新的智能合约漏洞检测工具的研发和改进提供参考与指导.

区块链技术的快速发展, 给智能合约提供了可靠的执行环境. 随着智能合约普及到各种各样的去中心化应用中, 智能合约的安全问题也变得愈发重要. 本文梳理了当前智能合约面临的主要安全漏洞类型, 针对现阶段研究工作中存在的问题给出了建议, 展望了智能合约安全漏洞检测的研究方向, 以期能够启发未来的研究工作.

References:

[1] Swan M. Blockchain: Blueprint for a New Economy. O'Reilly Media, Inc., 2015.

[2] Zheng Z, Xie S, Dai HN, *et al*. Blockchain challenges and opportunities: A survey. Int'l Journal of Web and Grid Services, 2018, 14(4): 352−375. [doi: 10.1504/IJWGS.2018.095647]

[3] Sankar LS, Sindhu M, Sethumadhavan M. Survey of consensus protocols on blockchain applications. In: Proc. of the 4th Int'l Conf. on Advanced Computing and Communication Systems (ICACCS). IEEE, 2017. 1−5. [doi: 10.1109/ICACCS.2017.8014672]

[4] Xia Q, Sifah EB, Smahi A, *et al*. BBDS: Blockchain-based data sharing for electronic medical records in cloud environments. Information, 2017, 8(2): 44. [doi: https://doi.org/10.3390/info8020044]

[5] Azaria A, Ekblaw A, Vieira T, *et al*. Medrec: Using blockchain for medical data access and permission management. In: Proc. of the 2nd Int'l Conf. on Open and Big Data (OBD). IEEE, 2016. 25−30. [doi: 10.1109/OBD.2016.11]

[6] Zhang P, Schmidt DC, White J, *et al*. Blockchain technology use cases in healthcare. Advances in computers. Elsevier, 2018, 111: 1−41. [doi: https://doi.org/10.1016/bs.adcom.2018.03.006]

[7] Meng Z, Morizumi T, Miyata S, *et al*. Design scheme of copyright management system based on digital watermarking and blockchain. In: Proc. of the IEEE 42nd Annual Computer Software and Applications Conf. (COMPSAC), Vol.2. IEEE, 2018. 359−364. [doi: 10.1109/COMPSAC.2018.10258]

[8] Holland M, Nigischer C, Stjepandic J. Copyright protection in additive manufacturing with blockchain approach. Transdisciplinary Engineering: A Paradigm Shift, 2017, 5: 914−921. [doi: 10.3233/978-1-61499-779-5-914]

[9] Qian P, Liu Z, Wang X, *et al*. Digital resource rights confirmation and infringement tracking based on smart contracts. In: Proc. of the IEEE 6th Int'l Conf. on Cloud Computing and Intelligence Systems (CCIS). IEEE, 2019. 62−67. [doi: 10.1109/CCIS48116.2019.9073733]

[10] Saberi S, Kouhizadeh M, Sarkis J, *et al*. Blockchain technology and its relationships to sustainable supply chain management. Int'l Journal of Production Research, 2019, 57(7): 2117−2135. [doi: https://doi.org/10.1080/00207543.2018.1533261]

[11] Abeyratne SA, Monfared RP. Blockchain ready manufacturing supply chain using distributed ledger. Int'l Journal of Research in Engineering and Technology, 2016, 5(9): 1−10. [doi: https://doi.org/10.15623/ijret.2016.0509001]

[12] Chen S, Shi R, Ren Z, *et al*. A blockchain-based supply chain quality management framework. In: Proc. of the 14th IEEE Int'l Conf. on e-Business Engineering (ICEBE). IEEE, 2017. 172−176. [doi: 10.1109/ICEBE.2017.34]

[13] Mengelkamp E, Notheisen B, Beer C, *et al*. A blockchain-based smart grid: Towards sustainable local energy markets. Computer Science-Research and Development, 2018, 33(1−2): 207−214. [doi: https://doi.org/10.1007/s00450-017-0360-9]

[14] Pop C, Cioara T, Antal M, *et al*. Blockchain based decentralized management of demand response programs in smart energy grids. Sensors, 2018, 18(1): 162. [doi: https://doi.org/10.3390/s18010162]

[15] Knirsch F, Unterweger A, Eibl G, *et al*. Privacy-preserving smart grid tariff decisions with blockchain-based smart contracts. In: Proc. of the Sustainable Cloud and Energy Services. Cham: Springer-Verlag, 2018. 85−116. [doi: https://doi.org/10.1007/978-3-319-62238-5_4]

[16] Christidis K, Devetsikiotis M. Blockchains and smart contracts for the internet of things. IEEE Access, 2016, 4: 2292−2303. [doi: 10.1109/ACCESS.2016.2566339]

[17] Bahga A, Madisetti VK. Blockchain platform for industrial internet of things. Journal of Software Engineering and Applications, 2016, 9(10): 533−546. [doi: 10.4236/jsea.2016.910036]

[18] Kshetri N. Can blockchain strengthen the Internet of Things. IT Professional, 2017, 19(4): 68−72. [doi: 10.1109/MITP.2017.3051335]

[19] Wang R, Lin Z, Luo H. Blockchain, bank credit and SME financing. Quality & Quantity, 2019, 53(3): 1127−1140. [doi: https://doi.org/10.1007/s11135-018-0806-6]

[20] Buterin V. A next-generation smart contract and decentralized application platform. Ethereum White Paper, 2014, 3(37).

[21] Ouyang LW, Wang S, Yuan Y, *et al*. Smart contracts: Architecture and research progresses. Acta Automatica Sinica, 2019, 45(3): 445−457 (in Chinese with English abstract). [doi: 10.16383/j.aas.c180586]

[22] Szabo N. Smart contracts: Building blocks for digital markets. Journal of Transhumanist Thought, 1996, 18(16): 2.

[23] Etherscan. 2014. https://etherscan.io/

[24] EOS Official Portal. 2019. https://eos.io/

[25] Vntchain. 2018. https://scan.vntchain.io/

[26] Bcsec. 2018. https://bcsec.org/

[27] Slowmist. 2018. https://hacked.slowmist.io/

[28] The DAO. 2016. https://en.wikipedia.org/wiki/TheDAO(organization)

[29] Parity multisig bug. 2017. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/

[30] BeautyChain integer overflow. 2018. https://etherscan.io/token/0xc5d105e63711398af9bbff092d4b6769c82f793d

[31] EOS WIN 2018. https://eos.win/

[32] FarmEOS. 2019. https://bcsec.org/index/detail/id/456

[33] Playgames. 2019. https://bcsec.org/index/detail/id/459

[34] LuckBet. 2019. https://bcsec.org/index/detail/id/461

[35] EOSlots. 2019. https://bcsec.org/index/detail/id/477

[36] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. In: Proc. of the Consulted. 2008. [doi: http://dx.doi.org/10.2139/ssrn.3440802]

[37] Mohanta BK, Panda SS, Jena D. An overview of smart contract and use cases in blockchain technology. In: Proc. of the 9th Int'l Conf. on Computing, Communication and Networking Technologies (ICCCNT). IEEE, 2018. 1−4. [doi: 10.1109/ICCCNT.2018.8494045]

[38] Zheng Z, Xie S, Dai H, *et al*. An overview of blockchain technology: Architecture, consensus, and future trends. In: Proc. of the IEEE Int'l Congress on Big Data (BigData congress). IEEE, 2017. 557−564. [doi: 10.1109/BigDataCongress.2017.85]

[39] Macrinici D, Cartofeanu C, Gao S. Smart contract applications within blockchain technology: A systematic mapping study. Telematics and Informatics, 2018, 35(8): 2337−2354. [doi: https://doi.org/10.1016/j.tele.2018.10.004]

[40] Xu MX, Yuan C, Wang YJ, *et al*. Mimic blockchain—Solution to the security of blockchain. Ruan Jian Xue Bao/Journal of Software, 2019, 30(6): 1681−1691 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5744.htm [doi: 10.13328/j.cnki.jos.005744]

[41] Fu ML, Wu LF, Hong Z, *et al*. Research on vulnerability mining technique for smart contracts. Journal of Computer Applications, 2019, 39(7): 1959−1966 (in Chinese with English abstract). [doi: 10.11772/j.issn.1001-9081.2019010082]

[42] Wang HQ, Zhang F, Li T, *et al*. Security and privacy protection technologies in smart contract. Journal of Nanjing University of Posts and Telecommunications, 2019, 39(4): 63−71 (in Chinese with English abstract). [doi: 10.14132/j.cnki.1673-5439.2019.04.009]

[43] Ni YD, Zhang C, Yin TT. A survey of smart contract vulnerability research. Journal of Cyber Security, 2020, 5(3): 78−99 (in Chinese with English abstract). [doi: 10.19363/J.cnki.cn10-1380/tn.2020.05.07]

[44] Zheng ZB, Wang CD, Cai JH. Analysis of the current status of smart contract security research and detection methods. Information Security and Communications Privacy, 2020, (7): 93−105 (in Chinese with English abstract).

[45] Dune analytics. 2020. https://www.duneanalytics.com/

[46] Grossman S, Abraham I, Golan-Gueta G, *et al*. Online detection of effectively callback free objects with applications to smart contracts. In: Proc. of the ACM on Programming Languages (POPL). 2017. 1−28. [doi: https://doi.org/10.1145/3158136]

[47] Hard-Fork. 2020. https://www.investopedia.com/terms/h/hard-fork.asp

[48] SafeMath. 2020. https://docs.statechannels.org/contract-api/natspec/SafeMath

[49] Atzei N, Bartoletti M, Cimoli T. A survey of attacks on Ethereum smart contracts (SOK). In: Proc. of the Int'l Conf. on Principles of Security and Trust. Berlin, Heidelberg: Springer-Verlag, 2017. 164−186. [doi: https://doi.org/10.1007/978-3-662-54455-6_8]

[50] KoET. 2017. https://www.kingoftheether.com/thrones/kingoftheether/index.html

[51] POWH. 2018. https://etherscan.io/token/0x317eb3b357e5cb2c94dca5586f018d594d3d8091

[52] Chen W, Zheng Z, Cui J, *et al*. Detecting ponzi schemes on Ethereum: Towards healthier blockchain technology. In: Proc. of the 2018 World Wide Web Conf., Int'l World Wide Web Conf. on Steering Committee. 2018. 1409−1418. [doi: 10.1145/3178876.3186046]

[53] Chen W, Zheng Z, Ngai ECH, *et al*. Exploiting blockchain data to detect smart ponzi schemes on Ethereum. IEEE Access, 2019, 7: 37575−37586. [doi: 10.1109/ACCESS.2019.2905769]

[54] Bartoletti M, Carta S, Cimoli T, *et al*. Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact. Future Generation Computer Systems, 2020, 259−277. [doi: https://doi.org/10.1016/j.future.2019.08.014]

[55] Han SM, Liang B, Huang JJ, *et al*. DC-Hunter: Detecting dangerous smart contracts via bytecode matching. Journal of Cyber Security, 2020, 5(3): 100−112 (in Chinese with English abstract). [doi: 10.19363/J.cnki.cn10-1380/tn.2020.05.08]

[56] Bhargavan K, Delignat-Lavaud A, Fournet C, *et al*. Formal verification of smart contracts: Short paper. In: Proc. of the 2016 ACM Workshop on Programming Languages and Analysis for Security. 2016. 91−96. [doi: https://doi.org/10.1145/2993600.2993611]

[57] Bai X, Cheng Z, Duan Z, *et al*. Formal modeling and verification of smart contracts. In: Proc. of the 2018 7th Int'l Conf. on Software and Computer Applications. 2018. 322−326. [doi: https://doi.org/10.1145/3185089.3185138]

[58] Abdellatif T, Brousmiche KL. Formal verification of smart contracts based on users and blockchain behaviors models. In: Proc. of the 9th IFIP Int'l Conf. on New Technologies, Mobility and Security (NTMS). IEEE, 2018. 1−5. [doi: 10.1109/NTMS.2018.8328737]

[59] Mossberg M, Manzano F, Hennenfent E, *et al*. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2019. 1186−1189. [doi: 10.1109/ASE.2019.00133]

[60] Shishkin E. Debugging smart contract's business logic using symbolic model checking. Programming and Computer Software, 2019, 45(8): 590−599. [doi: https://doi.org/10.1134/S0361768819080164]

[61] Zhao W, Zhang WY, Wang JR, *et al*. Smart contract vulnerability detection scheme based on symbol execution. Journal of Computer Applications, 2020, 40(4): 947−953 (in Chinese with English abstract). [doi: 10.11772/j.issn.1001-9081.2019111919]

[62] Takanen A, Demott JD, Miller C, *et al*. Fuzzing for Software Security Testing and Quality Assurance. Artech House, 2018.

[63] Liang H, Pei X, Jia X, *et al*. Fuzzing: State of the art. IEEE Trans. on Reliability, 2018, 67(3): 1199−1218. [doi: 10.1109/TR.2018.2834476]

[64] Zhang X, Li ZJ. Survey of fuzz testing technology. Computer Science, 2016, 43(5): 1−8 (in Chinese with English abstract). [doi: 10.11896/j.issn.1002-137X.2016.5.001]

[65] Zhao J, Nagarakatte S, Martin MMK, *et al*. Formalizing the LLVM intermediate representation for verified program transformations. In: Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 2012. 427−440. [doi: https://doi.org/10.1145/2103656.2103709]

[66] Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. In: Proc. of the IEEE/ACM 38th Int'l Conf. on Software Engineering (ICSE). IEEE, 2016. 297−308. [doi: 10.1145/2884781.2884804]

[67] Shi Y, Sagduyu YE, Davaslioglu K, *et al*. Vulnerability detection and analysis in adversarial deep learning. Cham: Springer-Verlag, 2018. 211−234. [doi: https://doi.org/10.1007/978-3-319-92624-7_9]

[68] Wu F, Wang J, Liu J, *et al*. Vulnerability detection with deep learning. In: Proc. of the 3rd IEEE Int'l Conf. on Computer and Communications (ICCC). IEEE, 2017. 1298−1302. [doi: 10.1109/CompComm.2017.8322752]

[69] Li Z, Zou D, Xu S, *et al*. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv: 1801.01681, 2018.

[70] Russell R, Kim L, Hamilton L, *et al*. Automated vulnerability detection in source code using deep representation learning. In: Proc. of the 17th IEEE Int'l Conf. on Machine Learning and Applications (ICMLA). IEEE, 2018. 757−762. [doi: 10.1109/ICMLA.2018.00120]

[71] Grishchenko I, Maffei M, Schneidewind C. A semantic framework for the security analysis of ethereum smart contracts. In: Proc. of Int'l Conf. on Principles of Security and Trust. Cham: Springer-Verlag, 2018. 243−269. [doi: https://doi.org/10.1007/978-3-319-89722-6_10]

[72] Hildenbrandt E, Saxena M, Rodrigues N, *et al*. Kevm: A complete formal semantics of the Ethereum virtual machine. In: Proc. of the IEEE 31st Computer Security Foundations Symp. (CSF). IEEE, 2018. 204−217. [doi: 10.1109/CSF.2018.00022]

[73] Amani S, Bégel M, Bortin M, *et al*. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: Proc. of the 7th ACM SIGPLAN Int'l Conf. on Certified Programs and Proofs. 2018. 66−77. [doi: https://doi.org/10.1145/3167084]

[74] Kalra S, Goel S, Dhawan M, *et al*. ZEUS: Analyzing safety of smart contracts. In: Proc. of the NDSS. 2018. 1−12. [doi: 10.14722/ndss.2018.23082]

[75] VaaS. Automated formal verification platform for smart contract. 2019. https://www.lianantech.com/

[76] Luu L, Chu DH, Olickel H, *et al*. Making smart contracts smarter. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. 2016. 254−269. [doi: https://doi.org/10.1145/2976749.2978309]

[77] Nikolić I, Kolluri A, Sergey I, *et al*. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proc. of the 34th Annual Computer Security Applications Conf. 2018. 653−663. [doi: https://doi.org/10.1145/3274694.3274743]

[78] Tsankov P, Dan A, Drachsler-Cohen D, *et al*. Securify: Practical security analysis of smart contracts. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. 2018. 67−82. [doi: https://doi.org/10.1145/3243734.3243780]

[79] Mythril. A framework for bug hunting on the Ethereum blockchain. 2017. https://mythx.io/

[80] Krupp J, Rossow C. Teether: Gnawing at Ethereum to automatically exploit smart contracts. In: Proc. of the 27th USENIX Security Symp. 2018. 1317−1333.

[81] Rodler M, Li W, Karame GO, *et al*. Sereum: Protecting existing smart contracts against re-entrancy attacks. arXiv: 1812.05934, 2018.

[82] Jiang B, Liu Y, Chan WK. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. 2018. 259−269. [doi: https://doi.org/10.1145/3238147.3238177]

[83] Liu C, Liu H, Cao Z, *et al*. Reguard: Finding reentrancy bugs in smart contracts. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering: Companion (ICSE-Companion). IEEE, 2018. 65−68. [doi: https://doi.org/10.1145/3183440.3183495]

[84] He J, Balunović M, Ambroladze N, *et al*. Learning to fuzz from symbolic execution with application to smart contracts. In: Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security. 2019. 531−548. [doi: https://doi. org/10.1145/3319535.3363230]

[85] Feist J, Grieco G, Groce A. Slither: A static analysis framework for smart contracts. In: Proc. of the 2nd IEEE/ACM Int'l Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). IEEE, 2019. 8−15. [doi: 10.1109/WETSEB.2019.00008]

[86] Brent L, Jurisevic A, Kong M, *et al*. Vandal: A scalable security analysis framework for smart contracts. arXiv: 1809.03981, 2018.

[87] Grech N, Kong M, Jurisevic A, *et al*. Madmax: Surviving out-of-gas conditions in Ethereum smart contracts. Proc. of the ACM on Programming Languages, 2018, 2(OOPSLA): 1−27. [doi: https://doi.org/10.1145/3276486]

[88] Albert E, Gordillo P, Livshits B, *et al*. Ethir: A framework for high-level analysis of Ethereum bytecode. In: Proc. of Int'l Symp. on Automated Technology for Verification and Analysis. Cham: Springer-Verlag, 2018. 513−520. [doi: https://doi.org/10.1007/978-3-030-01090-4_30]

[89] Tikhomirov S, Voskresenskaya E, Ivanitskiy I, *et al*. Smartcheck: Static analysis of ethereum smart contracts. In: Proc. of the 1st Int'l Workshop on Emerging Trends in Software Engineering for Blockchain. 2018. 9−16. [doi: https://doi.org/10.1145/3194113.3194115]

[90] Wang X, He J, Xie Z, *et al*. ContractGuard: Defend Ethereum smart contracts with embedded intrusion detection. IEEE Trans. on Services Computing, 2019, 13(2): 314−328. [doi: 10.1109/TSC.2019.2949561]

[91] Tann WJW, Han XJ, Gupta SS, *et al*. Towards safer smart contracts: A sequence learning approach to detecting security threats. arXiv: 1811.06632, 2018.

[92] Qian P, Liu ZG, He QM, *et al*. Towards automated reentrancy detection for smart contracts based on sequential models. IEEE Access, 2020, 8: 19685−19695. [doi: 10.1109/ACCESS.2020.2969429]

[93] Zhou P, Shi W, Tian J, *et al*. Attention-based bidirectional long short-term memory networks for relation classification. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics. 2016. 207−212. [doi: 10.18653/v1/P16-2034]

[94] Zhang Y, Liu ZG, Qian P, *et al*. Smart contract vulnerability detection using graph neural networks. In: Proc. of the 29th Int'l Joint Conf. on Artificial Intelligence (IJCAI-20). 2020. [doi: https://doi.org/10.24963/ijcai.2020/454]

[95] Gilmer J, Schoenholz SS, Riley PF, *et al*. Neural message passing for quantum chemistry. In: Proc. of the 34th Int'l Conf. on Machine Learning (PMLR 70). 2017. 1263−1272.

[96] Li YJ, Tarlow D, Brockschmidt M, *et al*. Gated graph sequence neural networks. arXiv: 1511.05493, 2015.

[97] Wang W, Song J, Xu G, *et al*. ContractWard: Automated vulnerability detection models for ethereum smart contracts. IEEE Trans. on Network Science and Engineering, 2020. [doi: 10.1109/TNSE.2020.2968505]

附中文参考文献:

[21] 欧阳丽炜, 王帅, 袁勇, 等. 智能合约: 架构及进展. 自动化学报, 2019, 45(3): 445−457. [doi: 10.16383/j.aas. c180586]

[40] 徐蜜雪, 苑超, 王永娟, 等. 拟态区块链——区块链安全解决方案. 软件学报, 2019, 30(6): 1681−1691. http://www.jos.org.cn/1000-9825/5744.htm [doi: 10.13328/j.cnki.jos.005744]

[41] 付梦琳, 吴礼发, 洪征, 等. 智能合约安全漏洞挖掘技术研究. 计算机应用, 2019, 39(7): 1959−1966. [doi: 10.11772/j.issn.1001-9081.2019010082]

[42] 王化群, 张帆, 李甜, 等. 智能合约中的安全与隐私保护技术. 南京邮电大学学报(自然科学版), 2019, 39(4): 63−71. [doi: 10.14132/j.cnki.1673-5439.2019.04.009]

[43] 倪远东, 张超, 殷婷婷. 智能合约安全漏洞研究综述. 信息安全学报, 2020, 5(3): 78−99. [doi: 10.19363/J.cnki.cn10-1380/tn.2020.05.07]

[44] 郑忠斌, 王朝栋, 蔡佳浩. 智能合约的安全研究现状与检测方法分析综述. 信息安全与通信保密, 2020(7): 93−105.

[55] 韩松明, 梁彬, 黄建军, 等. DC-Hunter: 一种基于字节码匹配的危险智能合约检测方案. 信息安全学报, 2020, 5(3): 100−112. [doi: 10.19363/J.cnki.cn10-1380/tn.2020.05.08]

[61] 赵伟, 张问银, 王九如, 等. 基于符号执行的智能合约漏洞检测方案. 计算机应用, 2020, 40(4): 947−953. [doi: 10.11772/j.issn.1001-9081.2019111919]

[64] 张雄, 李舟军. 模糊测试技术研究综述. 计算机科学, 2016, 43(5): 1−8. [doi: 10.11896/j.issn.1002-137X.2016.5.001]

钱鹏(1996－), 男, 硕士, 主要研究领域为智能合约安全, 图数据处理.

刘振广(1988－), 男, 博士, 研究员, CCF 专业会员, 主要研究领域为智能合约代码数据处理, 图像数据处理, 多媒体技术.

何钦铭(1965－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为机器学习, 区块链技术.

黄步添(1981－), 男, 博士, 高级工程师, CCF 专业会员, 主要研究领域为联盟区块链.

田端正(1983－), 男, 工程师, 主要研究领域为区块链架构.

王勋(1967－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为计算机图形学与虚拟现实, 计算机视觉, 智能信息处理.