软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(12):5755−5779 [doi: 10.13328/j.cnki.jos.007396] [CSTR: 32375.14.jos.007396]

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

基于目标制导符号执行的智能合约安全漏洞检测*

杨慧文[1], 崔展齐[1], 陈 翔[2], 郑丽伟[1], 刘建宾[1]

[1](北京信息科技大学 计算机学院, 北京 100101)

[2](南通大学 信息科学技术学院, 江苏 南通 226019)

通信作者: 崔展齐, E-mail: czq@bistu.edu.cn

摘 要: 智能合约是运行在区块链上的计算机程序, 在扩展区块链功能、实现复杂应用的同时, 其潜在的安全漏洞也带来巨大风险. 基于符号执行的安全漏洞检测方法具有精确度高、可生成能复现漏洞的测试用例等优势. 然而, 随着代码规模的增大, 符号执行技术容易受到路径爆炸、约束求解开销过大等问题的影响. 为此, 提出一种基于目标制导符号执行的智能合约安全漏洞检测方法, 首先将静态分析工具或人工标注的漏洞语句作为目标, 分析目标依赖语句, 补充事务序列以添加相关变量的符号约束. 然后, 基于智能合约字节码构建控制流图, 定位目标语句以及目标依赖语句所在的基本块, 剪枝控制流图以生成制导信息. 最后, 根据制导信息优化符号执行的路径探索策略, 减少需要分析的基本块数量以及求解路径条件的时间, 最终高效地检测目标语句是否存在安全漏洞, 并输出可复现漏洞的测试用例. 基于上述思路实现 Smart-Target 原型工具, 在 SB Curated 数据集上与符号执行工具 Mythril 进行对比. 实验结果表明 Smart-Target 在安全漏洞检测和漏洞复现场景分别减少 60.76% 和 92.16% 的时间开销, 大幅提高符号执行效率. 此外, Smart-Target 通过分析目标依赖语句使其多检测到 22.02% 的安全漏洞, 有效提升了漏洞检测能力.

关键词: 智能合约; 符号执行; 目标制导测试; 安全漏洞检测; 安全漏洞复现

中图法分类号: TP309

中文引用格式: 杨慧文, 崔展齐, 陈翔, 郑丽伟, 刘建宾. 基于目标制导符号执行的智能合约安全漏洞检测. 软件学报, 2025, 36(12): 5755–5779. http://www.jos.org.cn/1000-9825/7396.htm

英文引用格式: Yang HW, Cui ZQ, Chen X, Zheng LW, Liu JB. Smart Contract Security Vulnerability Detection Based on Target-guided Symbolic Execution. Ruan Jian Xue Bao/Journal of Software, 2025, 36(12): 5755–5779 (in Chinese). http://www.jos.org.cn/1000-9825/7396.htm

Smart Contract Security Vulnerability Detection Based on Target-guided Symbolic Execution

YANG Hui-Wen[1], CUI Zhan-Qi[1], CHEN Xiang[2], ZHENG Li-Wei[1], LIU Jian-Bin[1]

[1](Computer School, Beijing Information Science and Technology University, Beijing 100101, China)

[2](School of Information Science and Technology, Nantong University, Nantong 226019, China)

Abstract: Smart contracts are computer programs running on blockchain platforms, which extend the functionality of the blockchain and enable complex applications. However, the potential security vulnerabilities of smart contracts can lead to significant financial losses. Symbolic execution-based security vulnerability detection methods offer advantages such as high accuracy and the ability to generate test cases that can reproduce vulnerabilities. Nevertheless, as the code size increases, symbolic execution faces challenges such as path explosion and excessive constraint-solving overhead. To address those issues, a novel approach for detecting smart contract security vulnerabilities through target-guided symbolic execution is proposed. First, vulnerable statements identified by static analysis tools or manually are treated as targets. The statements that depend on these target statements are analyzed, and the transaction sequence is

* 基金项目: 江苏省前沿引领技术基础研究专项 (BK202002001); 北京信息科技大学“勤信人才”培育计划 (QXTCP B202406); 北京控制工程研究所高可信嵌入式软件工程技术实验室开放基金 (LHCESET202307)
收稿时间: 2023-01-05; 修改时间: 2024-05-29; 采用时间: 2025-01-13; jos 在线出版时间: 2025-06-11
CNKI 网络首发时间: 2025-06-11

augmented with symbolic constraints for the relevant variables. Second, the control flow graph (CFG) is constructed based on the bytecode of smart contracts, with the basic blocks containing the target statements and the dependentstatements located. The CFG is then pruned to generate guidance information. Third, path exploration in symbolic execution is optimized by reducing the number of basic blocks to be analyzed and reducing the time required for solving path constraints. With the guidance information, vulnerabilities are efficiently detected, and test cases capable of reproducing the vulnerabilities are generated. Based on this approach, a prototype tool named Smart-Target is developed. Experiments conducted on the SB Curated dataset in comparison with the symbolic execution tool, Mythril, demonstrate that Smart-Target reduces time overheads by 60.76% and 92.16% in vulnerability detection and replication scenarios, respectively. In addition, the analysis of target statementdependencies enhances vulnerability detection capability by identifying 22.02% more security vulnerabilities.

Key words: smart contract; symbolic execution; target-guided testing; security vulnerability detection; security vulnerability reproduction

1 引 言

随着区块链技术的发展, 众多领域都在探索利用区块链解决实际问题. 智能合约是运行在区块链平台上的计算机程序[1,2], 能够按照预先定义的代码自动执行, 具有无需第三方信任、去中心化以及不可篡改的特性[3], 在金融、物联网[4]以及智慧医疗[5]等领域被广泛应用. 然而, 由于其管理和存储数字资产, 吸引大量攻击者企图利用智能合约中的安全漏洞非法窃取利益, 如 2016 年 TheDAO 合约被攻击, 造成价值 6 000 万美元的以太币被恶意转移 (DAO 攻击事件, https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human); 2017 年 Parity 合约被攻击, 导致价值 3 000 万美元的以太币被冻结或窃取 (Parity 攻击事件, https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug). 此外, 与传统软件不同, 智能合约一旦部署在区块链平台后将无法更新代码[6], 难以通过补丁等方式对安全漏洞进行修复, 因此在部署智能合约前进行全面的安全漏洞检测尤为重要.

目前, 已有一些研究基于区块链以及智能合约的特点, 将静态分析[7–9]、模糊测试[10,11]、机器学习[12,13]、形式化验证[14]以及符号执行[15–19]等软件分析与测试中较为成熟的技术应用到智能合约安全漏洞检测中, 已取得了较好的检测效果. Durieux 等人[20]对 9 种静态分析和符号执行工具进行了实证研究, 结果表明静态分析工具 Slither[9]和符号执行工具 Mythril (https://github.com/ConsenSys/mythril) 能够检测出较多的漏洞, 具有较好的漏洞检测能力. Ren 等人[21]在 Durieux 等人的基础上更全面地从查准率、查全率等方面评估了 9 种静态分析、符号执行和模糊测试工具的漏洞检测能力, 发现虽然 Slither 能够找到较多的漏洞, 但具有较高的误报率.

结合其他相关实证研究[22]及综述[23–27]可以发现: 基于静态分析的方法通常分析控制流、数据流或调用关系等信息, 结合预先定义的漏洞模式或比较相似度等方式检测漏洞, 具有查全率高以及检测效率较快的优势, 但由于对合约状态、路径条件或调用方式等进行了假设, 导致误报率较高; 基于符号执行或模糊测试的方法符号化或实际部署并执行被测合约, 在检测过程中根据收集到的符号约束或执行信息判断是否存在安全漏洞, 具有更高的精确度且能够生成复现漏洞的测试用例, 但检测所消耗时间较多.

目标制导符号执行能够利用静态分析查全率高以及符号执行精确度高的优势, 将静态分析报告的语句或函数作为目标, 剪枝与目标无关的控制流路径, 优化符号执行的路径探索策略, 从而高效地覆盖和分析指定目标, 并检测安全漏洞. 此外, 在漏洞复现场景中, 还可利用符号执行能够生成测试用例的特点, 将人工标注的漏洞语句作为目标, 快速地生成能够触发漏洞的测试用例, 从而帮助开发或测试人员复现漏洞.

目标制导符号执行技术在缺陷检测以及确认等场景已有较多的研究与应用. 甘水滔等人[28]从程序功能文档提取功能标签, 结合控制流信息构建各功能标签和程序基本块之间的映射关系, 针对给定的目标点对程序进行切片, 并剪枝无关的功能分支提高制导效率. 鲍铁匀等人[29]根据静态分析的缺陷报告剪枝控制流图中与缓冲器溢出无关的路径, 制导符号执行沿着特定路径探索, 通过约束求解对静态分析的报告进行确认. 我们在前期研究中将静态分析所报告的可疑语句作为目标, 制导动态符号执行并生成覆盖待检测缺陷语句的测试输入, 提高了缺陷检测的效率[30]. 然而, 上述研究主要针对基于 C/C++等语言编写的软件, 智能合约则是由 Solidity 等程序设计语言编写, 且运行在区块链环境中, 与 C/C++等语言编写的程序在语法、语义、运行平台和运行机制上具有较大差异. 将目标制导符号执行技术应用到智能合约中面临两个研究挑战.

首先, 智能合约是具有状态的程序[15,31], 与 C/C++语言从单一程序入口 (即 main 函数) 开始执行程序不同, 智能合约对外界提供多个函数接口, 通过不同的事务序列 (即函数调用序列) 对合约内的状态变量进行修改, 从而使合约达到不同状态. 若仅将疑似漏洞所在语句作为目标, 可能会导致无法满足目标语句 (即漏洞语句) 所在分支的约束条件, 使符号执行错误地判断目标语句的可达性. 然而, 如果将状态变量假设为任意值, 会降低目标语句所在分支可达性判断的准确性, 导致产生误报增加人工确认的成本.

其次, 可达性分析是目标制导符号执行技术中的关键步骤, 该步骤对控制流图进行剪枝以优化符号执行的路径探索策略. 然而, 智能合约的控制流跳转信息无法直接获取, 需要根据字节码分析堆栈以构建控制流图; 并且, 静态分析或人工标注通常仅提供目标语句的行号, 因此需要对目标信息进行处理, 基于编译器提供的映射关系定位到目标基本块, 进而结合控制流图进行可达性分析.

针对上述挑战, 本文提出了一种基于目标制导符号执行的智能合约安全漏洞检测方法, 以 Solidity 语言编写的智能合约为例, 借鉴现有目标制导符号执行技术的思想, 将其适配并应用到智能合约领域中. 针对目标语句所在分支需要状态变量满足一定约束条件的情况, 本文通过分析目标依赖语句, 补充事务序列以添加相关变量的符号约束, 提高目标语句所在分支可达性判断的准确性. 在可达性分析中, 根据字节码划分基本块并构建控制流图, 解析源代码与字节码之间的映射关系, 定位目标语句并剪枝控制流图以生成制导信息, 优化符号执行的路径探索策略, 减少时间开销并快速生成测试用例.

本文的主要贡献如下.

- 提出一种基于目标制导符号执行的智能合约安全漏洞检测方法. 该方法能够降低符号执行的时间开销, 提高安全漏洞检测效率, 并能够快速生成覆盖指定目标的测试用例, 达到复现漏洞的目的.
- 提出目标依赖语句分析, 定位与目标语句存在控制依赖或数据依赖关系的语句, 补充事务序列以添加相关变量的符号约束, 提高目标语句所在分支可达性判断的准确性.
- 实现了原型工具 Smart-Target (https://github.com/yagol2020/Smart-Target), 在 SB Curated (https://github.com/smartbugs/smartbugs/tree/master/dataset) 人工标注数据集上与非目标制导的 Mythril 符号执行工具进行对比实验, 分析并评估了所提出方法的有效性.

本文第 2 节介绍相关工作. 第 3 节通过示例说明现有符号执行方法的局限性, 并引出本文所提方法的动机. 第 4 节给出智能合约语句、函数以及事务的符号化表示, 并详细介绍目标依赖语句分析、可达性分析以及目标制导符号执行的具体流程. 第 5 节描述实验设计, 并对实验结果进行分析. 第 6 节讨论方法的泛用性以及有效性威胁. 最后, 第 7 节总结全文, 并对未来工作进行展望.

2 相关工作

本节将从基于符号执行的智能合约安全漏洞检测方法和目标制导符号执行技术这 2 个方面介绍相关工作.

2.1 基于符号执行的智能合约安全漏洞检测方法

符号执行[32,33]在安全漏洞检测以及测试用例生成等领域被广泛应用[34]. 基于符号执行的智能合约安全漏洞检测方法将函数调用中的参数、调用者和携带以太币的数量等信息, 以及区块时间戳、块号和哈希值等区块链环境信息使用符号值作为替换, 符号化执行智能合约的字节码指令, 收集路径分支的约束条件, 通过约束求解器对路径条件进行求解, 判断路径分支的可达性, 并根据预先定义的漏洞模式判断路径条件是否满足漏洞触发条件, 从而进行安全漏洞检测并输出触发漏洞的测试用例.

Luu 等人[17]构建了 Oyente 工具, 对智能合约中的重入、整型溢出以及时间戳依赖等漏洞进行了描述, 并通过符号化表示各个漏洞模式, 从而能够利用符号执行对安全漏洞进行检测. 在 Oyente 的基础上, Chen 等人[18]构建了 DefectChecker 工具, 在符号执行过程中构建堆栈事件, 提取金钱调用、循环体和支付函数 3 种特征检测安全漏洞. Tsankov 等人[19]构建了 Securify 工具, 使用特定领域语言作为中间表示, 分析合约依赖关系以及语义信息, 利用符号执行检查是否违反了预先定义的模式, 相较于 Oyente 能够达到更高的代码覆盖率.

上述研究仅考虑了单一事务, 没有考虑对多个函数调用所组成的事务序列进行符号化分析. Mossberg 等人[16]基于 KLEE 构建了 Manticore 工具, 将事务的参数与携带的以太币数量等信息符号化, 重复地符号化执行事务从而探索合约各个状态. Nikolić等人[35]构建了 Maian 工具, 提出贪心、自杀以及挥霍 3 种类型漏洞, 构建事务序列并在符号执行过程中分析以太币转移和合约销毁相关的指令, 判断以太币转移的地址是否能被外部控制, 从而检测安全漏洞. So 等人[15]提出了 SmarTest, 从已知存在漏洞的事务序列中得到统计模型, 并基于该统计模型判断候选事务序列存在漏洞的概率, 选择概率较高的事务序列进行符号执行, 减少事务序列组合数量提高检测效率. ConsenSys 构建了 Mythril 工具, 结合污点分析与符号执行技术, 能够对运行在 Ethereum、Hedera 和 Quorum 等兼容 EVM 的区块链平台上的智能合约进行安全漏洞检测. Huang 等人[36]提出了动态符号执行框架, 构建了能够检测整型溢出漏洞的 NOVA 和多事务漏洞的 MTVD, 为解决状态变量非均匀数据访问 (nonuniform data accesses) 不精确的问题, 通过符号化状态变量的插槽序号, 生成具体值并结合哈希结果寻找状态变量存储的地址, 实验结果表明 NOVA 和 MTVD 优于现有工具, 具有较高的召回率和较低的误报率.

Durieux 等人[20]以及 Ren 等人[21]的实证研究结果表明, 考虑了事务序列的符号执行方法相较于仅检测单一事务的方法, 对代码的覆盖以及合约状态的探索更加充分, 通常能够检测出更多的漏洞. 但同时, 随着智能合约代码规模的增长, 程序代码的分支数量以及事务序列的组合数量快速增多, 符号执行可能遇到路径爆炸问题, 导致约束求解时间过长甚至失败, 对基于符号执行的安全漏洞检测方法的可扩展性以及漏洞检测能力产生较大的影响.

本文所提出的方法能够优化基于符号执行的智能合约安全漏洞检测, 通过目标制导符号执行技术缓解路径爆炸问题对符号执行效率以及检测能力造成的影响, 剪枝与目标无关的路径与基本块, 减少需要被分析的代码规模、调用约束求解的次数以及事务组合的数量, 提高了符号执行的检测效率.

2.2 目标制导符号执行技术

目标制导符号执行是一种能够有效缓解路径爆炸问题的技术手段, 利用静态分析等方式获得目标语句, 对控制流图中各基本块的可达性进行分析, 剪枝与目标语句无关的基本块, 优化符号执行的路径探索策略, 高效地遍历、分析并检测指定的基本块, 以提高符号执行的效率.

崔展齐等人[30]构建了 TARGET 工具, 将静态分析报告的疑似缓冲区溢出漏洞语句作为目标, 计算程序各分支到可疑语句的可达性, 插桩待检测程序并进行目标制导动态符号执行检测, 实验结果表明相较于无目标制导的动态符号执行方法, TARGET 能在更短的时间内发现更多的缓冲区溢出漏洞. 甘水滔等人[28]提出 OPT-SSE 方法, 根据功能标签对程序进行切片, 分析程序控制流图并构建功能标签与基本块之间的映射关系, 对于给定的代码目标点, 提取相关的功能标签切片并进行制导符号执行, 实验表明 OPT-SSE 能够提高符号执行的效率, 并且通过并行的方式对多个程序功能切片进行符号执行能够提高整体分支和指令的覆盖率. Guo 等人[37]提出断言制导的剪枝框架, 在符号执行过程中剪枝不会引发错误的路径, 并总结探索过程中无法覆盖错误语句的原因, 剪枝冗余路径减少搜索空间, 同时针对并发程序提出了静态并发程序切片, 提高符号执行的效率.

目标制导符号执行技术还能够被应用于漏洞报告确认中, 减少误报率以及人工确认的成本, 同时生成能够复现漏洞的测试用例. Ge 等人[38]构建了 DyTa 工具, 将静态分析和动态符号执行结合, 在静态分析阶段获得缺陷描述以及潜在缺陷的位置, 基于缺陷描述中的预定义条件插桩被测代码; 在动态符号执行阶段, 当遇到需要对路径条件翻转时, 根据静态分析的结果以及控制流图判断翻转后的路径是否能够达到存在潜在缺陷的代码, 若不能则不进行翻转, 从而对探索状态空间进行剪枝. Li 等人[39]提出了静态内存泄露警报自动确认方法, 针对静态分析报告的内存泄漏语句信息, 对目标程序进行控制流图分析并计算可达性, 生成制导信息, 在动态符号执行过程中监控内存对象的状态, 判断内存泄漏是否发生, 减少确认漏洞报告的人工成本. 李筱等人[40]在文献 [39] 所提出方法的基础上补充了与其他内存泄漏动态测试工具的对比试验, 并尝试将高覆盖率的测试用例作为动态符号执行的输入, 实验结果表明方法能够有效降低需要人工验证的警报数量, 并且在测试用例质量提高的情况下警报分类的准确性更高. 鲍铁匀等人[29]同样利用目标制导符号执行技术对静态缓冲区溢出报告进行自动确认, 剪枝与漏洞无关的路径, 制导符号执行沿特定确认路径执行, 能够减少 59.9% 的缓冲区溢出误报.

然而, 现有的目标制导符号执行技术通常以 C/C++程序作为分析对象, 难以直接应用到智能合约领域中, 一方面是因为智能合约具有状态变量且能够通过不同的事务序列对合约状态进行修改, 相较于单一入口的 C/C++程序具有更多程序状态. 另一方面从智能合约字节码中无法直接提取控制流信息以及定位目标基本块, 需要进一步结合编译器提供的信息进行分析. 本文通过分析目标依赖语句, 添加与目标相关变量的符号约束, 基于字节码构建控制流图并分析编译器提供的 Source Mappings 定位目标基本块, 将目标制导符号执行技术适配、改进并应用到了智能合约安全漏洞检测领域, 实验结果表明能够提高安全漏洞检测效率并高效生成测试用例.

3 研究动机示例

本节以一个真实的智能合约为例说明研究动机. 图 1 为存在整型溢出漏洞的智能合约代码片段 (PFG, https://etherscan.io/address/0xa2c4f961b93fc7bc200d2c8e55b6fe6958366787#code), 根据 CVE 以及相关引用的信息 (CVE-2018-13328, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-13328), 该示例中第 72 行和第 90 行存在整型溢出漏洞. 整型溢出漏洞指经过运算的结果超出了存储变量数据类型的最大存储范围, 第 41 行 *balances* 存储的数据类型为 uint256, 即存储范围为 0 至 $2^{256}-1$. 如果 *_value* 或 *amount* 中存储的数值过大, 又或 *balances* 已存储较大的数值, 都将可能引发整型溢出漏洞, 导致 *balances* 中记录的数据出现错误, 进而引发代码逻辑错误. 整型溢出漏洞是智能合约中常见漏洞之一[41], 且由于智能合约能够存储和管理区块链上的数字资产, 因此数值类型的数据更应被重视, 应在合约部署前对智能合约中的整型溢出、除零或算术符号错误等安全漏洞进行全面的检测.

```
41  mapping (address => uint256) balances;
42  mapping (address => mapping (address => uint256)) allowed;
70  function transferFrom(address _from, address _to, uint256 _value) public {
71      require(balances[_from] >= _value && allowed[_from][msg.sender] >= _value && _value > 0);
72      balances[_to] += _value;                    //整型溢出
73      balances[_from] -= _value;
74      allowed[_from][msg.sender] -= _value;
75      Transfer(_from, _to, _value);
76  }
78  function approve(address _spender, uint256 _value) public {
79      allowed[msg.sender][_spender] = _value;
81  }
87  function mint(uint256 amount) public {
88      assert(amount >= 0);
89      require(msg.sender == owner);
90      balances[msg.sender] += amount;             //整型溢出
91      totalSupply += amount;
92  }
```

图 1　存在整型溢出漏洞的智能合约代码片段

此外, 该 CVE 中没有提供相应的测试用例用于复现所提到的安全漏洞, 难以判断引发整型溢出漏洞的具体原因 (如未限制 *_value* 大小、*balances*[*_recipient*] 在其他函数被错误赋值等), 对安全漏洞的修复缺少有效的帮助信息.

符号执行能够有效地检测整型溢出等与数值相关的漏洞. 例如, 使用 Mythril 工具可在运行 560.8 s 后检测出第 72 行和第 90 行存在整型溢出漏洞, 并生成了对应的测试用例. Mythril 在符号执行分析过程中, 约束求解器分析路径条件所消耗的时间为 457.8 s.

然而, 并非该合约中所有的代码均与该漏洞相关, 该合约中未在图 1 展示的其余 2 个函数 (*balanceOf* 和 *allowance*) 以及 2 个事件 (*Transfer* 和 *Approval*) 用于输出合约的状态变量以及调用日志接口, 不会影响或触发漏洞. 直接使用符号执行进行安全漏洞检测将会对该合约中的所有代码进行分析, 不能充分利用 CVE、人工标注或静态分析提供的信息. 如果能在安全漏洞检测之前, 通过静态分析等方式获得疑似漏洞的位置, 将其作为目标, 剪枝控制流图中与目标无关的路径, 仅对目标语句相关的基本块或路径条件进行符号执行和约束求

解, 将会减少安全漏洞检测的时间开销. 此外, 针对静态分析工具仅提供疑似漏洞语句位置, 但缺少测试用例而难以复现漏洞的场景, 如果将漏洞语句作为目标, 可利用符号执行快速生成对应的测试用例, 以帮助开发或测试人员确认漏洞.

4 基于目标制导符号执行的智能合约安全漏洞检测方法

针对现有工作中符号执行检测效率较低问题, 本文提出了基于目标制导符号执行的智能合约安全漏洞检测方法. 图 2 为方法的流程图, 共包括 3 个步骤, 即目标分析、可达性分析以及目标制导符号执行. 首先进行目标分析, 从静态分析、人工标注等途径获取源代码中的漏洞目标信息, 包括目标语句所在的行号以及漏洞类型等, 再通过目标依赖语句分析, 获得与目标语句存在数据依赖或控制依赖的其他语句. 然后进行可达性分析, 将源代码编译为字节码并进一步翻译为指令, 根据指令模拟构建堆栈, 分析跳转指令并构建智能合约控制流图, 将目标分析步骤中得到的目标语句以及目标依赖语句作为分析对象, 定位其所在的基本块, 对控制流图进行剪枝以生成制导信息. 最后进行目标制导符号执行, 根据制导信息判断下一步跳转基本块的可达性, 制导符号执行跳过与目标语句无关的基本块, 并输出漏洞检测报告以及用于复现漏洞的测试用例.

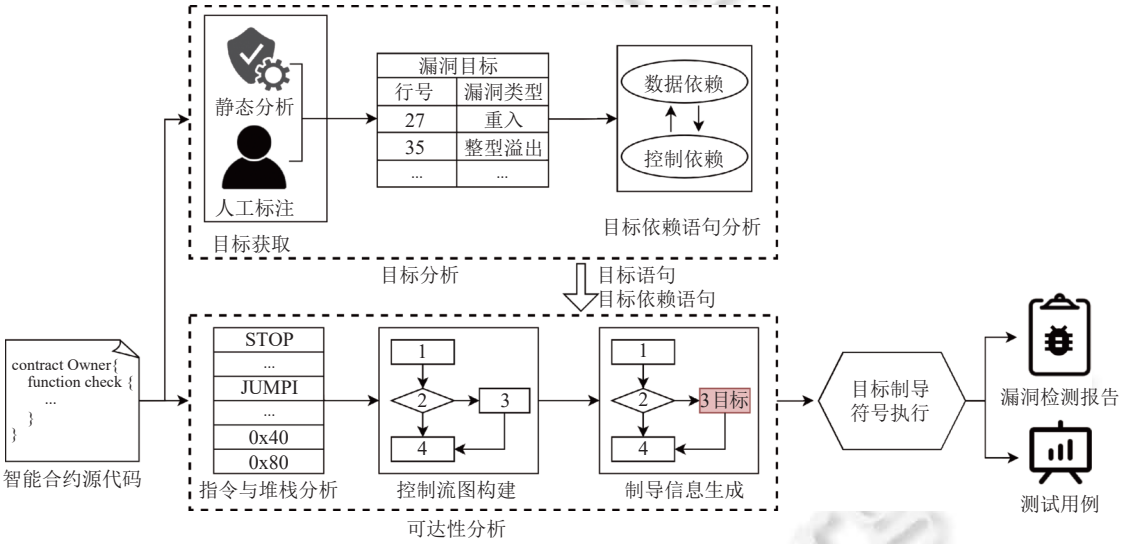


图 2 基于目标制导符号执行的智能合约安全漏洞检测方法流程图

本节的余下部分首先给出智能合约语句、函数以及事务的符号化表示, 然后介绍智能合约目标依赖语句分析、可达性分析以及目标制导符号执行.

4.1 智能合约语句、函数以及事务的符号化表示

Solidity 智能合约的代码由任意数量的状态变量、结构体、枚举、事件、函数以及函数修饰器组成. 表 1 为智能合约各代码结构的中英文名称和描述. 其中, 状态变量是用于持久存储数据的变量, 通常存储合约管理者、参与者或代币价格等信息; 结构体和枚举能够定义新的数据类型或有限的数据集合; 事件能够调用 EVM 的日志接口, 将合约的运行情况反馈给调用者; 函数和函数修饰器由任意数量的语句组成, 前者定义了智能合约的执行逻辑, 后者用于修饰函数的执行, 带有函数修饰器的函数需要先执行修饰器的内容, 例如在执行函数前检查执行条件是否满足.

根据语句执行的操作不同, 可将语句分为变量操作语句、假定语句、函数调用语句以及跳转语句. 同时, 根据智能合约中状态变量存储结构的长度是否固定, 可分为定长类型状态变量 (如 uint、address 等) 和非定长类型状态变量 (如动态数组、映射表等, https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html). 由于上述 2

种类型的状态变量在读取和赋值时寻址方式不同[36], 因此本文将变量操作语句分为定长变量赋值语句和非定长变量赋值语句.

表 1 Solidity 智能合约代码结构

中文名称	英文名称	描述
状态变量	State variable	持久存储数据
结构体	Struct	定义由多种数据类型组成的结构
枚举	Enum	定义有限的数据集合
事件	Event	调用EVM日志接口
函数	Function	合约的可执行单元
函数修饰器	Function modifier	修改函数执行的语义

表 2 描述了 Solidity 智能合约语句类型, 并给出了对应的符号化表达式. 其中, x 表示变量, $x[i]$ 表示非定长变量 x 中位于位置 i 的元素, *value* 表示常量、变量或由变量组成的符号化表达式, $symb(s)$ 表示符号化执行语句 s 的结果, $symb_{\text{func}}(f)$ 和 $symb_{\text{lib}}(l)$ 分别表示符号化执行函数 f 和库合约函数 l 的返回值, REVERT 表示回滚合约中的状态变量. 根据表 2 中智能合约语句的符号化表达式, 可将函数调用 f 表示为六元组 $(sender, value, data, sign, param, S)$. 其中, *sender* 为函数的调用者地址, *value* 为函数调用携带的以太币数量, *data* 为函数调用携带的数据, *sign* 为函数名称, *param* 为调用函数的参数, S 为此次函数调用过程中所执行的语句的路径条件, 即 $S = s_1 \wedge s_2 \wedge \ldots \wedge s_n$, 其中 s_i 表示符号化执行第 i 条语句后的符号约束条件, 各语句之间通过“合取”逻辑符 (即 $\wedge$) 连接, 表示语句 s_i 在 $s_1 \wedge s_2 \wedge \ldots \wedge s_{i-1}$ 的基础上添加符号值约束. 对于变量操作类型中的赋值语句以及函数调用类型语句中涉及利用返回值赋值的语句, 对被赋值对象添加上标以区分符号化执行赋值语句前后变量的符号值, 例如函数内语句 $\{a = param; b = a + 2; a = b\}$ 经过符号化执行后, 其路径条件为 $(a' = param \wedge b' = a' + 2 \wedge a'' = b')$.

表 2 Solidity 智能合约语句类型、示例以及符号化表达式

语句类型		示例	符号化表达式
变量操作	变量声明	uint a; uint[] b;	—
	定长变量赋值	$a = 12$;	$x := value$
	非定长变量赋值	$b[2] = 3$;	$x[i] := value$
假定	条件判断	if($a == 20$)$\{s_1;\}$else$\{s_2;\}$	$\begin{cases} x = value \wedge symb(s_1) \\ x != value \wedge symb(s_2) \end{cases}$
	条件约束	require($a == 20$); s_1; assert $a == 10$; s_1;	$\begin{cases} x = value \wedge symb(s_1) \\ x != value \wedge \text{REVERT} \end{cases}$
函数调用	合约函数调用 库合约函数调用	$b = foo(a)$; $c = lib(a, b)$;	$x := symb_{\text{func}}(f)$ $x := symb_{lib}(l)$
跳转	返回值 抛出异常	return a; revert();	— REVERT

此外, 智能合约可通过事务序列改变合约的状态变量, 进而影响函数的执行逻辑. 我们将事务序列 T 表示为 $T = \{init, f^*\}$. 其中, *init* 表示合约状态变量的初始化, f^* 为该事务序列中函数调用的次序, 即 $f^* = \{f_1, f_2, \ldots, f_n\}$. 事务序列 T 执行的语句的路径条件为 $S_T = S_{init} \wedge S_1 \wedge S_2 \wedge \ldots \wedge S_n$, 其中, S_{init} 表示初始化合约状态变量的路径条件, S_i 表示函数调用 f_i 所执行的语句的路径条件, S_{init} 以及 S_i 之间通过“合取”逻辑符连接, 即 S_i 在 $S_{init} \wedge S_1 \wedge S_2 \wedge \ldots \wedge S_{i-1}$ 对状态变量操作的基础上添加符号值约束. 此外, 在 S_n 中根据预先定义的安全漏洞模型添加漏洞检测符号约束, 从而检测事务序列 T 是否存在相应的安全漏洞.

4.2 目标依赖语句分析

智能合约是具有状态的程序, 能够通过不同的事务序列改变合约的状态变量, 从而使合约处于不同的状态, 函

数将根据状态变量判断当前合约状态是否满足分支的条件约束[15,31]. 因此, 若仅符号化执行控制流图中可到达目标语句的基本块, 缺少相关变量的符号约束, 将无法满足目标语句所在分支的条件约束, 使符号执行错误判断路径的可达性, 影响对目标语句的安全漏洞检测.

本文将影响目标语句可达性判断的语句称为目标依赖语句, 接下来将以图 1 中第 72 行的整型溢出漏洞为例, 具体描述利用符号执行对智能合约进行安全漏洞检测的具体过程, 以及目标依赖语句的作用.

在符号化执行事务序列 T 的过程中, 会收集变量的符号约束. 当存在漏洞模型预定义的操作时, 将在路径条件中添加相应的漏洞检测符号约束, 如图 1 中在第 72 行进行加法运算前添加符号约束 $\neg(a+v>a)$. 如果事务序列 T 执行语句的路径条件约束求解的结果为可满足, 则该事务序列中存在整型溢出漏洞, 约束求解的结果可以作为复现该漏洞的测试用例.

若仅将漏洞语句作为目标, 由于漏洞语句位于函数 *transferFrom* 中, 因此事务序列 T_{only} 可表示为:

$$T_{\text{only}}=\{init, transferFrom(p_1,p_2,p_3)\},$$

其中, p_1、p_2 和 p_3 分别表示参数 *_from*、*_to* 和 *_value* 的符号化表示. 将 T_{only} 执行语句的路径条件 S_{only} 展开可得:

$$\begin{aligned}
S_{\text{only}} &= balances'=0\wedge allowed'=0 && \text{(s1)}\\
&\wedge balances[p_1]'\geqslant p_3 && \text{(s2)}\\
&\wedge allowd[p_1][sender(transferFrom(p_1,p_2,p_3))]'\geqslant p_3 && \text{(s3)}\\
&\wedge p_3>0 && \text{(s4)}\\
&\wedge\neg(balances[p_2]'+p_3>balances[p_2]') && \text{(s5)}
\end{aligned}$$

其中, s1 为第 41 行和第 42 行对合约状态变量的初始化, s2–s4 为第 71 行的 require 语句判断条件, s5 则为整型溢出漏洞的检测条件. 从 S_{only} 可以看出, 由于 s1、s2 和 s4 无法同时满足, 因此约束求解器将判断 S_{only} 无法满足, 导致符号执行未能检测出整型溢出漏洞.

若将漏洞语句作为目标的同时, 定位与其存在依赖关系的其他语句, 补充事务序列以添加相关变量的符号约束, 能够更准确地判断路径可达性. 从图 1 可以看出, 第 72 行目标语句受到第 71 行分支语句的控制, 分支语句读取了变量 *balances* 和 *allowed*, 第 79 行 (位于 *mint* 函数) 和第 90 行 (位于 *approve* 函数) 分别对这两个变量进行赋值. 将这 2 个函数作为事务序列 T_{only} 的补充, 可得到由 *mint*、*approve* 以及 *transferFrom* 共 3 个函数调用所组成的事务序列. 当事务序列的长度限制为 3 时共有 $3^3=27$ 种组合方式, 其中一种事务序列 T_{sv} 为:

$$T_{sv}=\{init, mint(p_4), approve(p_5,p_6), transferFrom(p_1,p_2,p_3)\},$$

其中, p_4 为函数 *mint* 的参数 *amount* 的符号化表示, p_5 和 p_6 为函数 *approve* 的参数 *_spender* 和 *_value* 的符号化表示. 将 T_{sv} 执行语句的路径条件 S_{sv} 展开可得:

$$\begin{aligned}
S_{sv} &= balances'=0\wedge allowed'=0 && \text{(s1)}\\
&\wedge p_4\geqslant 0\wedge sender(mint(p_4))=owner' && \text{(s2}^+\text{)}\\
&\wedge(balances[sender(mint(p_4))]'+p_4>balances[sender(mint(p_4))]') && \text{(s3}^+\text{)}\\
&\wedge balances[sender(mint(p_4))]''=balances[sender(mint(p_4))]'+p_4 && \text{(s4}^+\text{)}\\
&\wedge(totalSupply'+p_4>totalSupply')\wedge totalSupply''=totalSupply'+p_4 && \text{(s5}^+\text{)}\\
&\wedge allowed[sender(approve(p_5,p_6))][p_5]'=p_6 && \text{(s6}^+\text{)}\\
&\wedge balances[p_1]'\geqslant p_3 && \text{(s2)}\\
&\wedge allowd[p_1][sender(transferFrom(p_1,p_2,p_3))]'\geqslant p_3 && \text{(s3)}\\
&\wedge p_3>0 && \text{(s4)}\\
&\wedge\neg(balances[p_2]'+p_3>balances[p_2]') && \text{(s5)}
\end{aligned}$$

由于事务序列 T_{sv} 在 *transferFrom* 前执行了 *mint* 和 *approve* 函数, 因此在 S_{only} 的基础上, 新增了 $\text{s2}^+-\text{s6}^+$ 共 5 条路径条件, 添加了 *balances* 和 *allowed* 的符号约束, 使 S_{sv} 可以满足, 即检测出整型溢出漏洞. 经约束求解, 可得到 S_{sv} 的一个可满足解为: $mint(p_4)$ 的调用者为 *owner*, p_4 为 $2^{255}-1$; $approve(p_5,p_6)$ 的调用者为 *owner*, p_5 为任意地址, p_6 为大于 0 的整型数值; $transferFrom(p_1,p_2,p_3)$ 的调用者为 p_5, p_1 和 p_2 为 *owner*, p_3 为 p_6.

由 *mint*、*approve* 以及 *transferFrom* 这 3 个函数调用所组成的其他 26 种事务序列同样可能触发第 72 行所示的漏洞. 本文所提出的目标制导符号执行方法能够减少事务序列的组合数量, 具体描述和讨论见本文第 4.4 节.

上述示例中, 第 71、79 和 90 行的语句为第 72 行目标语句的目标依赖语句. 通过符号化执行目标依赖语句, 可以更准确地判断目标语句的可达性. 为了更准确地描述目标依赖语句, 本文参考 Wang 等人[42]关于依赖关系的定义, 对智能合约中的数据依赖和控制依赖进行了定义.

定义 1 (数据依赖). 对于智能合约中的语句 $stmt_i$ 和 $stmt_j$, 若 $stmt_i$ 对变量 v 进行赋值, $stmt_j$ 读取 v, 且由 $stmt_i$ 到 $stmt_j$ 的路径中不存在语句 $stmt_k$ 对 v 进行赋值, 即 $stmt_j$ 读取的变量数值依赖于 $stmt_i$ 的赋值操作, 称 $dataDep(stmt_i, stmt_j) = T$; 否则 $dataDep(stmt_i, stmt_j) = F$.

定义 2 (控制依赖). 对于智能合约中的语句 $stmt_i$ 和 $stmt_j$, 其中 $stmt_j$ 是函数 $func$ 中的语句, 若 (1) $stmt_i$ 为 $func$ 中的分支语句, 且对应判定条件满足时才能执行 $stmt_j$; 或 (2) $stmt_i$ 为函数修饰器中的分支语句, 且对应判定条件满足时才能执行函数 $func$, 即 $stmt_j$ 的执行受到 $stmt_i$ 的控制, 称 $controlDep(stmt_i, stmt_j) = T$; 否则 $controlDep(stmt_i, stmt_j) = F$.

需要注意的是, 智能合约的函数修饰器能够改变函数的行为, 从而影响语句的执行. 例如图 3 中函数 *abort* 函数被函数修饰器 *onlySeller* 所修饰 (https://docs.soliditylang.org/en/latest/structure-of-a-contract.html#function-modifiers). *onlySeller* 通过 require 判断函数的调用者是否为 *seller*, 若不是则引发异常并输出信息, 若满足条件则继续执行 *abort* 函数. 因此, 在智能合约中函数内的语句能否执行, 不仅受到函数内分支语句的控制, 同时还被函数修饰器的分支条件所影响.

```
1  address public seller;
2  modifier onlySeller() {
3      require(msg.sender == seller, "Only seller can call this.");
4      _;
5  }
6  function abort() public view onlySeller {
7      //do somethings…
8  }
```

图 3　包含函数修饰器的智能合约代码示例

由上文中智能合约符号执行的分析过程可知, 为了更准确地判断目标语句可达性, 需要预先执行合约中的部分语句, 其与目标语句之间存在由控制依赖和数据依赖构成的传递闭包关系. 本文称这类语句为目标依赖语句.

定义 3 (目标依赖语句). 对于智能合约中的语句 $stmt_i$ 和目标语句 $stmt_j$, 若 $transitiveDep(stmt_i, stmt_j) = T$, 其中 $transitiveDep$ 表示 $stmt_i$ 和 $stmt_j$ 之间存在由控制依赖和数据依赖构成的传递闭包, 则称 $stmt_i$ 为 $stmt_j$ 的目标依赖语句.

根据上述 3 个定义, 可对智能合约 C 中目标语句 $target$ 的目标依赖语句进行分析, 分析过程如算法 1 所示. 第 1, 2 行初始化目标依赖语句集合 *ret* 和待分析语句队列 *queue*, 第 3–9 行对 *queue* 中的待分析语句 $stmt_j$ 进行分析, 其中第 4–6 行获得与 $stmt_j$ 存在数据依赖的语句 $stmt_D$ 并添加到 *ret* 和 *queue*, 第 7–9 行则获得与 $stmt_j$ 存在控制依赖的语句 $stmt_C$ 并添加到 *ret* 和 *queue*, 通过循环进行迭代分析, 最后在第 10 行输出目标依赖语句集合.

算法 1. 分析目标依赖语句.

输入: 智能合约 C; 目标语句 $target$;
输出: 目标依赖语句集合.

1. $ret = \varnothing$
2. $queue = Queue(target)$
3. **while** $stmt_j = queue.pop()$!= None:
4. $\quad stmt_D = getDataDep(stmt_j, C)$ //数据依赖
5. $\quad ret.add(stmt_D)$ //添加到目标依赖语句集合

6. $queue.push(stmt_D)$
7. $stmt_C = getControlDep(stmt_j, C)$ //控制依赖
8. $ret.add(stmt_C)$
9. $queue.push(stmt_C)$
10. **return** *ret*

在可达性分析中, 目标语句 *target* 和目标依赖语句集合 *ret* 将用于标记智能合约控制流图中各基本块的可达性, 达到剪枝控制流图的目的.

4.3 可达性分析

可达性分析指判断控制流图中各基本块是否与目标相关, 剪枝与目标无关的基本块生成制导信息, 用于指导符号执行优化路径探索策略. 首先解析智能合约字节码并翻译为指令集合, 将指令集合根据终止和跳转指令划分为多个基本块, 并根据跳转指令分析控制流的跳转信息, 以生成控制流图; 然后, 基于源代码和字节码之间的映射关系, 定位目标语句和目标依赖语句所在基本块; 最后, 遍历控制流图并标记各基本块的可达性, 剪枝无法到达目标语句的基本块以生成制导信息.

4.3.1 控制流图生成

根据表 3 所示的终止和跳转指令可将 Solidity 智能合约的指令集合划分为多个基本块. 其中, 跳转指令除了用于划分基本块外, 还用于计算各个基本块之间的跳转关系, 结合基本块和跳转关系可生成控制流图.

表 3 智能合约字节码的终止指令和跳转指令

指令类型	名称	控制流跳转	描述
终止指令	STOP	—	终止合约的执行
	RETURN	—	读取内存值, 作为返回值并结束此次调用
	REVERT	—	回滚此次调用的状态变量修改
	SELFDESTURT	—	销毁合约, 返还以太币给予栈顶地址
跳转指令	JUMP	$pc = pop()$	无条件跳转
	JUMPI	$pc = \begin{cases} pc+1 \\ pop() \end{cases}$	条件跳转

Solidity 智能合约中跳转指令有 2 种, 即无条件跳转 JUMP 指令和条件跳转 JUMPI 指令. 表 3 中 *pc* (program counter, 程序计数器) 用于表示当前指令的序号. 对于无条件跳转 JUMP, 当执行该指令时弹出栈顶元素, 该元素所表示的 16 进制数值即为跳转的目的地基本块中的首个指令的序号. 对于条件跳转 JUMPI, 当执行该指令时连续弹出 2 个栈顶元素, 分别记为 *destination* 和 *condition*, 判断 *condition* 是否为真, 若为真则跳转并执行指令序号为 *destination* 的指令 (即跳转到首个指令序号为 *destination* 的基本块), 否则顺延执行下一个指令 (即跳转到首个指令序号为 *pc*+1 的基本块).

4.3.2 制导信息生成

制导信息的生成需要遍历控制流图, 标记各基本块的可达性, 然后将各基本块的可达性信息输出为便于符号执行工具识别的存储形式.

首先, 由于目标语句以及目标依赖语句均仅提供源代码行号, 因此需要通过源代码与字节码之间的映射关系定位目标基本块. 源代码与字节码之间的映射关系可通过解析编译器 (如 solc) 提供的 Source Mappings (https://docs.soliditylang.org/en/latest/internals/source_mappings.html) 得到. 具体地, 先将源代码转换为 *pos* : *length* 的形式, *pos* 为源代码语句首个字符在代码文件中的位置, *length* 为源代码语句的字符长度. Source Mappings 经过处理后可得到指令序号与 *pos* : *length* 之间的映射关系. 得到该映射关系后, 可记录目标语句以及目标依赖语句所对应的指令序号集合作为目标指令集合.

然后, 遍历控制流图并判断各个基本块是否为目标基本块, 并对控制流图进行剪枝, 具体流程如算法 2 所示.

算法 2. 控制流图剪枝.

输入: 控制流图 *cfg*, 目标指令集合 *target_ins*;
输出: 剪枝过的控制流图.

```
1. visited = Set()
2. check(cfg.entrancy)
3. return cfg
4. Procedure bool check(bb):
5.    if bb in visited:
6.       return bb.targeted //若基本块被遍历过则返回目标值
7.    visited.add(bb)
8.    temp = false
9.    for ins in target_ins:
10.      if bb.start.pc ≤ ins ≤ bb.end.pc: //若目标指令位于该基本块中
11.         bb.targeted = true //标记基本块的目标值为 true
12.         temp = true //标记临时目标值为 true, 用于递归判断
13.         break
14.      for next_bb in bb.outgoing_bbs:
15.         temp = check(next_bb) or temp //递归遍历, 并更新临时目标值
16.      bb.targeted = temp //更新该基本块的目标值
17.      return bb.targeted
```

算法 2 中第 1–3 行是算法的主体, 第 4–17 行是遍历控制流图的递归过程 check. 其中, check 以 *cfg* 的入口基本块作为输入, 第 5–7 行判断该基本块是否被遍历过, 若遍历过则返回该基本块是否可达; 第 8 行声明临时变量 *temp* 用于表示基本块是否可达, 并赋初值为 false; 第 9–13 行遍历目标指令集合, 若该基本块包含目标指令, 则定位该基本块为目标基本块, 标记该基本块为可达 (第 11 行), 并更新 *temp* 值为 true; 第 14, 15 行遍历跳转基本块, 递归调用 check 函数对其余基本块进行分析和标记可达性; 第 16, 17 行更新并返回该基本块的可达性.

算法 2 在递归过程中, 如果定位到目标基本块, 则 check 函数将返回 true. 根据 check 函数的返回值与临时变量 *temp* 进行“或”操作 (第 15 行), 从而将目标基本块的所有父基本块均标记为可达, 得到从控制流图入口基本块到目标基本块之间的路径. 各基本块的可达性信息可用于生成制导信息.

4.4 目标制导符号执行

目标制导符号执行指在符号执行判断跳转基本块是否需要被符号化执行前, 根据制导信息中对各基本块可达性的标记情况, 跳过与目标语句无关的基本块, 减少求解路径条件的次数, 以及需要被分析的基本块数量, 达到提高符号执行效率的目的.

如第 3.3 节所述, 智能合约中的控制流跳转由 JUMP 和 JUMPI 指令实现. 在符号执行的过程中, 遇到 JUMPI 指令时会复制当前路径条件, 分别添加分支条件为 true 和 false 的符号约束, 调用约束求解器分别对两个路径条件的可满足性进行求解, 并将能够满足的路径条件所指示的基本块作为可达基本块, 进一步对该基本块进行符号化执行. 本文在求解路径条件是否可满足之前, 根据制导信息中基本块的可达性进行判断, 若跳转基本块的可达性为 false, 则不对其路径条件进行求解, 即剪枝该控制流路径.

智能合约目标制导符号执行的流程如算法 3 所示. 算法 3 的输入为第 3.3.2 节中得到的制导信息 *target_info*、合约创建成功后的事务序列队列 *trans_queue* 以及事务序列的最大长度 *limit*, 算法 3 的输出为能够覆盖目标语句的测试用例集合.

算法 3. 目标制导符号执行.

输入: 制导信息 *target_info*, 事务序列队列 *trans_queue*, 事务序列长度 *limit*;
输出: 能够覆盖目标的测试用例集合.

```
1. solutions = dict()
2. while trans_seq = trans_queue.pop() != None or limit– – > 0:
3.     state_queue = Queue(trans_seq.state)
4.     while state = state_queue.pop() != None:
5.         for ins in state.bb.ins:
6.             switch JUMP: //若为跳转指令 JUMP
7.                 jumpdest = pop_stack()
8.                 if target_info[jumpdest] == true: //判断 jumpdest 是否被标记为可达
9.                     state_queue.add(generate_state(state, jumpdest, JUMP, None)) //生成新状态, 并添加到队列
10.            switch JUMPI: //若为跳转指令 JUMPI
11.                jumpdest, condition = pop_stack(), pop_stack() //弹出 2 次堆栈, 作为跳转目的地和条件
12.                if target_info[jumpdest] == true:
13.                    if isSatisfy(state.condition, condition): //判断路径条件是否满足
14.                        state_queue.add(generate_state(state, jumpdest, JUMPI, condition))
15.                if target_info[state.pc+1] == true:
16.                    if isSatisfy(state.condition, !condition):
17.                        state_queue.add(generate_state(state, state.pc+1, JUMPI, !condition))
18.            switch STOP or RETURN or REVERT or SELFDESTURT: //若为终止指令
19.                trans_queue.add(trans_seq.update(state)); //更新事务序列并添加至事务序列集合
20.                solutions[state.pc] = (solve(state.condition), trans_seq) //求解状态的约束条件并添加到测试用例
21.            default: //若为其他指令
22.                symbolic_execution_ins(state, ins) //符号化执行该指令
23. return solutions
```

算法 3 第 1 行初始化测试用例集合, 第 2–22 行循环读取事务序列队列 *trans_queue* 直到为空, 或达到事务序列的最大长度 *limit*; 第 3 行将事务序列 *trans_seq* 的合约状态 *state* 将其放入 *state_queue* 队列中, 其中 *state* 表示 *trans_seq* 经过符号化执行后的合约状态, 包含状态变量等符号约束信息; 第 4–22 行从 *state_queue* 中取出队首的状态进行分析, 逐行符号化执行基本块指令: 对于 JUMP (第 6–9 行), 弹出 1 次堆栈并将其作为跳转目的地 *jumpdest*, 根据制导信息 *target_info* 判断 *jumpdest* 是否可达, 若可达则调用 *generate_state* 函数生成新的状态. 函数 *generate_state* 的参数中, JUMP 表示跳转指令, 用于计算 *gas* 消耗, None 表示当前跳转需要满足的条件约束, 由于 JUMP 为无条件跳转, 因此约束条件为空. 对于 JUMPI (第 10–17 行), 连续弹出 2 次堆栈, 分别作为跳转目的地 *jumpdest* 和跳转条件 *condition*, JUMPI 的跳转地址有 2 种, 满足跳转条件 *condition* 则跳转至 *jumpdest*, 否则跳转至 *state.pc*+1. 对于这 2 个跳转地址, 首先通过 *target_info* 判断是否可达, 然后分别将 *condition* 以及!*condition* (*condition* 的取反) 添加到路径条件中, 调用约束求解器判断路径条件是否满足, 若满足则调用 *generate_state* 函数生成相应的新状态, 并添加到 *state_queue* 中; 对于 STOP 等终止指令 (第 18–20 行), 首先更新 *trans_seq* 中状态变量的符号约束等信息, 用以记录在 *trans_seq* 的基础上符号化执行该状态后的合约状态, 并将其添加至 *trans_queue* 以继续组合和探索其他事务, 然后调用约束求解器求出满足当前约束条件的测试用例, 并添加到测试用例集合 *solutions* 中. 对于其他指令 (第 21, 22 行), 如 ADD、SUB 或 MUL 等, 符号化执行该指令并更新 *gas* 数量以及堆栈信息等.

需要注意的是, 算法 3 中第 2 行和第 4 行的 *trans_queue.pop* 函数和 *state_queue.pop* 函数分别对应事务序列的生成策略和合约状态的探索策略, 这 2 种策略的优化方法是符号执行以及智能合约研究工作中的重点[43], 如 SmarTest[15]基于语言模型判断事务序列触发漏洞的概率, 优先执行较高概率的事务序列以缓解组合爆炸问题; Smartian[31]基于数据依赖关系生成事务序列, 通过随机插入函数调用等策略对事务序列进行变异, 以增加模糊测试过程中对智能合约代码的覆盖率; Mythril 可选择深度优先、广度优先或随机路径选择等策略遍历合约状态; KLEE[44]采用随机路径选择并结合覆盖率优化搜索策略探索程序路径空间等.

本文所提出的方法不依赖于特定的事务序列生成策略或合约状态探索策略, 在算法 3 中可尝试应用不同的事务序列生成策略或合约状态探索策略以达到更高的安全漏洞检测效果. 在算法 3 中, 目标制导符号执行策略主要体现在第 8 行和第 12 行, 即在探索合约状态时剪枝无法到达目标语句的基本块, 减少需要被探索的合约状态数量, 以达到优化合约状态探索策略的目的. 同时, 若通过可达性分析判断某一函数与目标语句无关, 则制导信息中会将该函数对应的基本块设置为不可达, 不会将该函数添加至事务序列, 从而减少事务组合的数量以提高安全漏洞检测效率.

接下来以 Mythril 使用的事务序列生成策略为例, 描述目标制导符号执行方法如何优化现有的事务序列生成策略. Mythril 默认采用广度优先策略, 根据字节码中函数基本块出现的顺序对各函数进行探索, 在遇到 REVERT 或 RETURN 等终止指令后, 记录当前符号状态并选择另一个函数继续执行. 当合约内所有函数均被符号化执行后, Mythril 还原首个函数执行后的符号状态, 并再次根据函数基本块的顺序进行探索, 直到代码覆盖率无法提升或达到事务序列的最大长度.

图 4 中红色矩形表示已经符号化执行的函数调用, 蓝色矩形表示候选函数调用. 若蓝色矩形右上角存在“⊗”符号标记, 则表示该函数与目标语句无关, 无需对其进行符号化执行. 由图 4 可以看出, *func*2 函数被标记为与目标语句无关, 因此在符号执行过程中不会对该函数进行探索. 具体来讲, 当事务序列长度为 2 时, 若采用 Mythril 默认策略, 则需要执行 $3^2=9$ 种事务序列; 而通过目标制导过滤无关函数后, 仅需执行 $2^2=4$ 种事务序列即可. 随着事务序列长度的增加, 目标制导所带来的效率提升会更加明显.

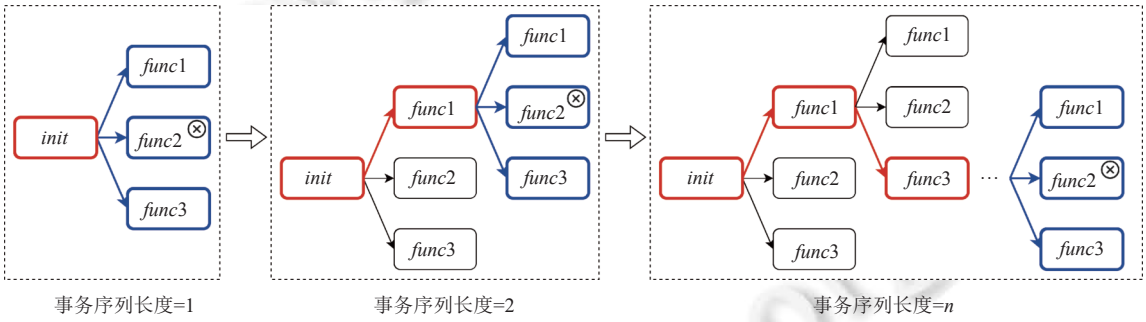


图 4 目标制导符号执行事务序列生成策略示意图

4.5 示例分析

本节以第 2 节中图 1 所示的代码为例, 说明本方法的主要流程. 根据 CVE 的信息, 我们将第 72 行和第 90 行语句作为目标, 然后进行目标依赖语句分析、可达性分析以及目标制导的符号执行.

首先进行目标依赖语句分析, 通过分析源代码得知, 第 72 行的目标依赖语句为第 71、79、88、89 和 90 行, 第 90 行的目标依赖语句为第 88 行和第 89 行. 由此可知, 第 72 和 90 行这 2 条目标语句的目标依赖语句为第 71、79、88 和 89 行.

第 2 步进行可达性分析, 将源代码编译为字节码并划分基本块, 通过分析跳转指令构建控制流图, 图 5(a) 为该合约代码的控制流图, 其中每个节点对应各个基本块, 节点上的序号为基本块中首个指令的序号. 然后, 通过编译器提供的 Source Mappings 定位目标语句以及目标依赖语句的基本块, 定位的结果如表 4 所示, 即 1371、1826、1837、2529、2542 和 2634 共 6 个基本块为目标基本块. 根据上述定位的 6 个目标基本块遍历并剪枝控制流图, 剪枝后的控制流图如图 5(b) 所示, 其中红色节点表示目标语句所在的基本块, 绿色节点表示目标依赖语句所在的

基本块, 蓝色节点表示由入口 (即 0 号节点) 出发到达红色或绿色节点所经过的基本块. 根据剪枝后的控制流图生成制导信息.

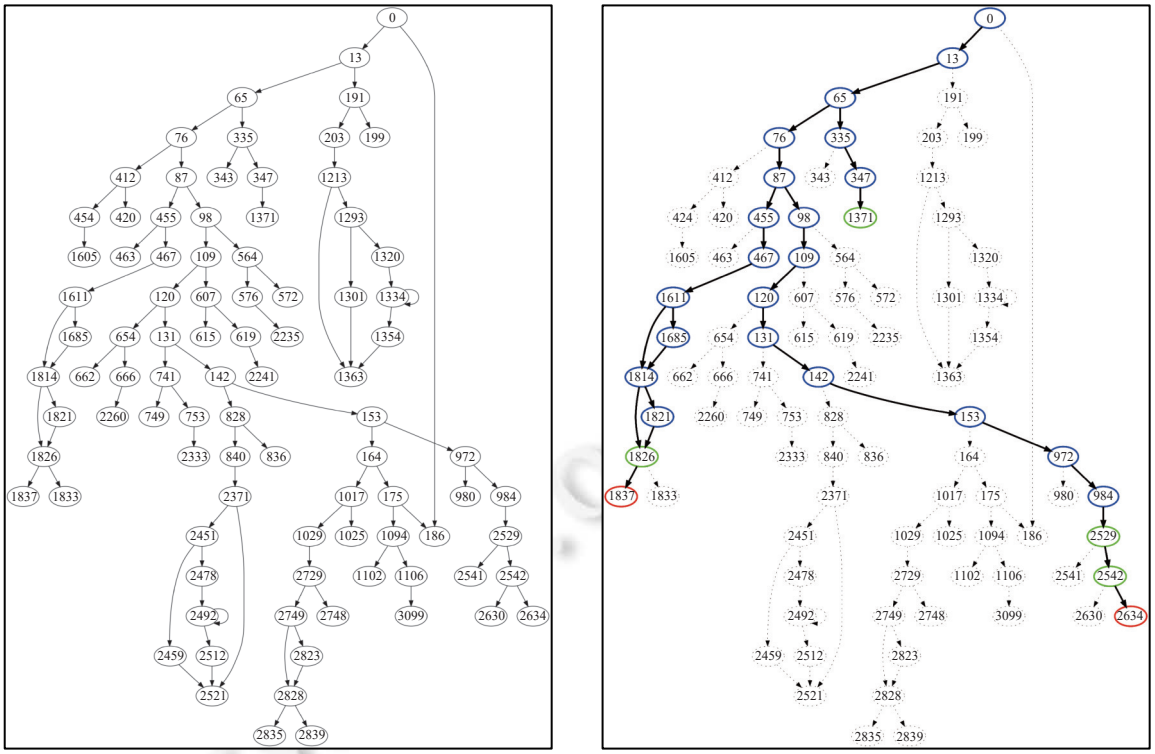


图 5　图 1 所示代码片段控制流图

表 4　图 1 代码片段的定位结果

类型	行号	源代码位置	指令序号	基本块序号
漏洞语句	72	2645:23	1904–1914	1837
	90	3172:30	2701–2711	2634
目标依赖语句	71	2547:88	1827–1836	1826
	79	2859:38	1497–1500	1371
	88	3117:19	2535–2541	2529
	89	3140:28	2642–2633	2542

最后进行目标制导符号执行, 在符号执行过程中根据制导信息判断跳转基本块是否被标记为可达, 跳过与目标语句或目标依赖语句无关的基本块. 目标制导符号执行的结果显示, 仅用时 107.4 s 即完成了安全漏洞检测, 并成功检测出第 72 行和第 90 行的整型溢出漏洞, 检测结果与 Mythril 相同.

对比图 5(a) 和 (b) 两张子图可以看出, 符号执行仅需对红色、绿色和蓝色共 35 个 (总基本块的 30%) 基本块进行遍历和分析, 对 28 条 (总基本块跳转的 28.6%) 路径的约束条件进行求解, 即可对该智能合约代码进行安全漏洞检测.

若直接使用 Mythril 对全部代码进行测试, 用时 560.8 s 可检测出第 72 行和第 90 行的整型溢出漏洞. 总体上看, 本文提出的方法减少了 80.85% 的符号执行时间开销, 提高了安全漏洞检测的效率, 并成功生成了能够复现漏洞的测试用例.

5　实验设计及评估

基于所提出方法, 我们在智能合约符号执行工具 Mythril 的基础上实现了原型工具 Smart-Target. Mythril 结合

了符号执行和污点分析等技术, 能够检测出包括整型溢出、重入以及未检测低级调用等安全漏洞, 具有较强的漏洞检测能力[20–22]. 在事务序列生成以及合约状态探索策略的选择上, 本文沿用 Mythril 中的生成和探索策略, 即根据字节码中函数基本块出现的顺序生成和执行事务序列, 并采用广度优先策略探索合约状态. 此外, Smart-Target 基于 Python 第三方库 evm-cfg-builder (https://github.com/crytic/evm_cfg_builder) 识别基本块并生成控制流图以进行可达性分析, 基于静态分析工具 Slither (https://github.com/crytic/slither) 对智能合约源代码进行解析, 获得状态变量类型、状态变量操作语句以及结构体定义等信息.

为了验证本文提出的方法的有效性, 我们提出了 3 个研究问题, 分别从安全漏洞检测、安全漏洞复现以及目标依赖语句对漏洞检测能力的影响 3 个角度出发, 对 Smart-Target 进行实验和评估. 具体研究问题如下.

RQ1: 在安全漏洞检测场景中, 使用静态分析的结果作为目标, Smart-Target 能否减少符号执行的时间开销? 是否会影响安全漏洞检测性能?

安全漏洞检测场景指在部署智能合约前进行全面的安全漏洞检测. 静态分析检测效率高但可能存在误报, 且无法生成测试用例; 符号执行精确度较高且能生成对应的测试用例, 但由于路径爆炸以及约束求解难度等问题, 检测效率较低. 因此, 安全漏洞检测场景中可将静态分析与符号执行结合, 使用本文提出的目标制导符号执行方法快速生成测试用例. RQ1 将静态分析报告的多个漏洞同时作为目标, 确认静态分析的报告并生成测试用例.

RQ2: 在安全漏洞复现场景中, Smart-Target 能否快速复现指定漏洞并生成测试用例?

安全漏洞复现场景指合约的参与者、用户或其他安全专家发现或报告源代码中存在安全漏洞问题, 但未提供具体的测试用例用于复现. 本文提出的目标制导符号执行方法可将人工标注的漏洞语句作为目标, 生成能够复现漏洞的测试用例, 帮助智能合约开发或测试人员复现、分析和修复安全漏洞. RQ2 将数据集中人工标注存在漏洞的语句逐一作为目标, 生成复现指定漏洞的测试用例.

RQ3: 目标依赖语句分析是否提升了 Smart-Target 的安全漏洞检测能力?

为了评估目标依赖语句分析对 Smart-Target 检测安全漏洞能力的影响, RQ3 与 RQ2 类似, 将数据集中人工标注存在漏洞的语句逐一作为目标, 比较分析目标依赖语句分析对目标制导符号执行检测安全漏洞的能力以及时间开销的影响.

5.1 实验设计

本文选取 Durieux 等人[20]的实证研究中使用的 SB Curated 公开数据集 (https://github.com/smartbugs/smartbugs/tree/master/dataset) 作为实验对象. 该数据集中包含 10 种漏洞类型, 即权限控制 (access control, AC)、算术相关 (arithmetic, ARM)、弱随机性 (bad randomness, BD)、拒绝服务 (denial of service, DoS)、前端运行 (front running, FR)、重入 (reentrancy, RE)、短地址 (short addresses, SA)、时间操作 (time manipulation, TM)、未检查低级调用 (unchecked low level calls, ULLC) 以及其他尚未分类的漏洞 (other, OT), 各漏洞类型的文件数量以及漏洞数量如表 5 所示.

表 5 SB Curated 数据集相关信息

漏洞类型	文件数	漏洞数	简介
权限控制	18	24	函数或语句能够被外界任意地执行
算术相关	15	23	整型上溢或下溢
弱随机性	8	34	基于区块链环境变量生成随机数, 但环境变量能够被外界 (如矿工) 控制
拒绝服务	6	14	代码中存在大量耗时操作, 如循环中发起外部调用
前端运行	4	7	事务执行的结果与当前合约状态有关
重入	31	32	合约函数非预期地外界再次被调用
短地址	1	1	利用短地址使EVM读取非地址数据, 造成数据异常
时间操作	5	7	基于区块链时间戳执行某些函数或语句, 而时间戳能够被矿工影响
未检查低级调用	52	75	使用call等函数但未检查返回值, 当出现调用失败情况时将影响合约执行逻辑
其他	3	5	其他不属于以上9种类型的漏洞, 如未初始化变量等

本文选择静态分析工具 Slither 和符号执行工具 Mythril 作为实验的比较对象, 这 2 个工具开源且更新频率较快, 同时实证研究表明这 2 个工具的漏洞检测能力较优于其他工具[20–22]. 需要注意的是, 由于 Slither 和 Mythril 均不支持对弱随机性、拒绝服务、前端运行、短地址以及其他这 5 种类型漏洞的检测, 且 Slither 不支持对算术相关漏洞的检测, 为了保证实验的正确性与公平性, 在实验中不使用弱随机性、拒绝服务、前端运行、短地址以及其他这 5 种类型的漏洞, 且在 Slither 相关的实验中不使用算术相关漏洞.

根据提出的 3 个实验问题设计了 3 组实验, 各个实验问题对应的实验设计如表 6 所示. 其中, 对于 RQ1 (安全漏洞检测场景), 采用静态分析工具 Slither 检测结果作为目标, 并将报告的多个漏洞语句位置同时作为目标, 评估检测时间、TP (即与人工标注的漏洞位置和类型一致的报告数)、FP (即与人工标注的漏洞位置或类型不一致的报告数)、FN (即没有报告出人工标注漏洞语句的数量) 以及 *F*1 值等; 对于 RQ2 (安全漏洞复现场景), 采用数据集中标注的漏洞语句位置作为目标, 并逐一尝试复现漏洞, 评估复现时间以及复现成功率; 对于 RQ3 (验证目标依赖语句分析有效性), 为了避免其他语句对实验的干扰, 同样将数据集中标注的漏洞语句逐一作为检测漏洞目标, 评估在分析目标依赖语句分析前后检测时间以及检测出的漏洞数量变化.

表 6　各个研究问题对应的实验设计

序号	漏洞类型选择	目标来源	目标方式	评价指标
RQ1	AC、RE、TM、ULLC	Slither	多目标	检测时间、*F*1值等
RQ2	AC、ARM、RE、TM、ULLC	人工标注	单目标	复现时间、复现成功率
RQ3	AC、ARM、RE、TM、ULLC	人工标注	单目标	检测时间、漏洞数量

上述实验均运行在 Ubuntu 20.04 操作系统、Intel(R) Xeon(R) E5-2680v4@2.40 GHz 处理器以及 16 GB 内存的环境中. Smart-Target 基于 Mythril v0.23.3、Slither v0.8.3 以及 evm_cfg_builder v0.3.1 实现, 实验比较的对象选择相同版本的 Mythril 和 Slither.

5.2 实验结果及分析

5.2.1 RQ1 的结果与分析

RQ1 以静态分析工具 Slither 的漏洞报告作为目标, 使用重入、权限控制、未检查低级调用以及时间操作这 4 类漏洞相关的智能合约进行实验, 从时间消耗和安全漏洞检测能力两个方面比较 Slither、Smart-Target 以及 Mythril 这 3 种工具的性能差异. 实验结果如表 7 所示.

表 7　安全漏洞检测场景中各工具性能比较

漏洞类型	工具	时间 (s)	TP	FP	FN	Recall	Precision	*F*1	测试用例生成数量	事务序列执行数量
重入	Slither	6.88	32	10	0	1	0.762	0.865	—	—
	Smart-Target	17268.95	28	2	4	0.875	0.933	0.903	30	952
	Mythril	63434.55	28	2	4	0.875	0.933	0.903	30	1595
权限控制	Slither	3.31	5	3	19	0.208	0.625	0.315	—	—
	Smart-Target	333.96	6	0	16	0.273	1	0.429	6	38
	Mythril	7957.689	6	0	16	0.273	1	0.429	6	167
未检查低级调用	Slither	14.79	71	3	4	0.947	0.959	0.953	—	—
	Smart-Target	74434.35	55	0	20	0.733	1	0.846	55	1414
	Mythril	162747.77	56	0	19	0.747	1	0.855	56	2772
时间操控	Slither	0.85	3	3	4	0.429	0.5	0.462	—	—
	Smart-Target	137.51	2	2	5	0.286	0.5	0.364	4	10
	Mythril	744.43	2	2	5	0.286	0.5	0.364	4	101
总计	Slither	25.83	111	19	27	0.804	0.854	0.828	—	—
	Smart-Target	92174.76	91	4	45	0.669	0.958	0.788	95	2414
	Mythril	234884.44	92	4	44	0.676	0.958	0.793	96	4635

在时间消耗方面, Slither 漏洞检测所需时间为 25.83 s, Smart-Target 所需的时间为 92 174.76 s, Mythril 所需的时间为 234 884.44 s. 相较于 Slither 仅分析源代码的语法语义, Mythril 逐条指令进行符号化执行, 收集路径条件并对其进行求解以分析可达性, 因此 Mythril 所需的时间远大于 Slither. 而 Smart-Target 通过目标制导优化符号执行的路径探索策略, 跳过了与目标语句无关的基本块, 减少了需要分析的指令数量以及路径条件求解的次数, 因此相对于非目标制导的 Mythril, Smart-Target 节省了 60.76% 的时间开销.

在安全漏洞检测能力性能比较上, Slither 的 $F1$ 值为 0.828, Mythril 的 $F1$ 值为 0.793, Smart-Target 的 $F1$ 值为 0.788. 其中, Smart-Target 与 Mythril 的 FP 相同, TP 相差 1. 分析发现, Smart-Target 比 Mythril 少检测到的漏洞为未检查低级调用漏洞, 该漏洞语句 (ClockAuction#1496, https://github.com/smartbugs/smartbugs/blob/master/dataset/unchecked_low_level_calls/0x663e4229142a27f00bafb5d087e1e730648314c3.sol#L1496) 没有被 Slither 报告为漏洞, 因此 Smart-Target 没有将其设置为目标, 导致漏报. 当把该漏洞语句作为目标时, Smart-Target 能够检测出该漏洞. 上述结果表明, Smart-Target 在安全漏洞检测能力上与 Mythril 相同, 并没有因为剪枝控制流图而降低漏洞检测能力.

从表 7 可以看出, Slither 比 Mythril 和 Smart-Target 能够检测出更多漏洞. 但是在安全漏洞分析、定位和修复的应用场景上, 由于 Slither 仅能报告漏洞类型以及漏洞所在行数, 相较于 Mythril 和 Smart-Target 缺少生成测试用例复现漏洞的能力. 实验结果显示, Smart-Target 和 Mythril 分别生成了 95 和 96 个测试用例, 其中除了生成触发人工标注的漏洞 (即 TP) 之外, 还在重入以及时间操作的漏洞检测过程中为 4 个未被数据集标注的语句生成了测试用例. 分析发现, 其中 1 个测试用例触发了未涉及以太币转移的重入行为, 另外 3 个触发了人工标注的漏洞, 但由于人工标注的语句不完整 (如仅标注函数调用语句或时间戳的声明语句), 导致报告的漏洞位置未能与数据集中人工标记的语句匹配. 上述结果表明, 相较于 Slither 仅提供漏洞所在位置, Smart-Target 和 Mythril 能够生成测试用例触发相应的安全漏洞, 并且 Smart-Target 能够为 Slither 检测出的 111 个安全漏洞中的 91 个生成对应的测试用例, 成功生成测试用例的比例达 81.9%.

表 7 中最后一列统计了各工具生成和执行事务序列的数量, 从中可以看出, 静态分析工具 Slither 相较于符号执行工具 Mythril 和 Smart-Target 缺少生成并执行事务序列的能力, 因此在检测存在多个函数的复杂智能合约时, 难以为开发或测试人员复现、理解和修复漏洞提供有效的帮助. 相较于 Mythril 共执行了 4 635 条事务序列, Smart-Target 在仅执行了 2 414 条测试用例的情况下, 检测出与 Mythril 相同数量的安全漏洞, 这是因为 Smart-Target 能够通过可达性分析将与目标语句无关的函数基本块设置为不可达, 因此在目标制导符号执行过程中能够过滤掉无关函数, 从而减少事务的组合数量以及需要执行事务序列数量. 随着需要执行的事务序列数量减少, 符号执行的时间开销也相应减少, Smart-Target 的时间开销相较于 Mythril 减少了 60.76%.

此外, 由于 Slither 未考虑上下文约束, 相较于 Mythril 和 Smart-Target 产生了较多误报. 图 6 为一个 Slither 误报的重入漏洞代码示例 (spank_chain_payment, https://github.com/smartbugs/smartbugs/blob/master/dataset/reentrancy/spank_chain_payment.sol). Slither 报告了第 803–806 行存在重入漏洞, 但实际上 804 行的外部函数调用需要满足第 802 行的 if 条件, 该条件需要 *tokenbalanceA* 或 *tokenbalanceI* 不为 0. 当 *channel.token.transfer* 函数存在重入行为时 (*channel.token* 存储 HumanStandardToken 类, 该类的 *transfer* 函数可能由攻击者重写, 从而引发重入), 由于第 794 行和第 795 行已将值设置为 0, 因此 *tokenbalanceA* 和 *tokenbalanceI* 均为 0, 无法满足第 802 行的函数条件, 即无法引发重入漏洞. 由于 Slither 未考虑语句的上下文, 因此误报了 803 行的漏洞. 而 Mythril 和 Smart-Target 能够收集 *channel* 结构体中各个成员变量的符号值, 在第 1 次执行第 792–795 行时, 将 *channel* 中相关的成员变量数组的符号值置为 0, 在重入过程中判断第 802 行的路径条件无法满足, 因此能够正确判断该智能合约中不存在重入漏洞.

还需要注意的是, Smart-Target 是基于 Mythril 实现的安全漏洞检测工具, 仅在 Mythril 的基础上优化了路径探索的策略, 而没有改变或增强漏洞检测逻辑 (Oracle), 因此 Smart-Target 的安全漏洞检测能力依赖于 Mythril. 若将本文所提出的目标制导符号执行方法应用到其他符号执行工具中, 如 SmartTest[15]、Manticore[16]或 Oyente[39]等, 可能会取得更高的安全漏洞检测能力.

综上所述, 我们对 RQ1 的回答是: 在安全漏洞检测场景中, 相较于静态分析工具 Slither, Smart-Target 减少了

误报并成功为 81.9% 的 Slither 报告漏洞生成了测试用例; 相较于非目标制导的 Mythril, Smart-Target 在没有损失安全漏洞检测能力的情况下减少了 60.76% 的检测时间.

```
787  uint256 ethbalanceA = channel.ethBalances[0];
788  uint256 ethbalanceI = channel.ethBalances[1];
789  uint256 tokenbalanceA = channel.erc20Balances[0];
790  uint256 tokenbalanceI = channel.erc20Balances[1];
792  channel.ethBalances[0] = 0;
793  channel.ethBalances[1] = 0;
794  channel.erc20Balances[0] = 0;
795  channel.erc20Balances[1] = 0;
797  if(ethbalanceA != 0 || ethbalanceI != 0) {
798      channel.partyAddresses[0].transfer(ethbalanceA);
799      channel.partyAddresses[1].transfer(ethbalanceI);
800  }
802  if(tokenbalanceA != 0 || tokenbalanceI != 0) {
803      require(
804          channel.token.transfer(channel.partyAddresses[0], tokenbalanceA),
805          "byzantineCloseChannel: token transfer failure"
806      );
807      require(
808          channel.token.transfer(channel.partyAddresses[1], tokenbalanceI),
809          "byzantineCloseChannel: token transfer failure"
810      );
811  }
```

图 6　Slither 误报示例

5.2.2　RQ2 的结果与分析

为模拟漏洞复现场景, RQ2 使用重入、权限控制、未检查低级调用、时间操作以及算术相关 5 类漏洞相关的智能合约进行实验, 将 SB Curated 数据集中标注了上述 5 类漏洞的 161 条漏洞语句作为目标. 在实验过程中每隔 60 s 对漏洞复现情况进行一次采样, 统计并比较在不同时刻 Smart-Target 和 Mythril 复现漏洞数量以及复现成功率.

实验结果如图 7 所示, 其中横轴为复现时间, 纵轴表示截至该时刻各个工具能够复现的漏洞数量. 从图中可以看出, Smart-Target 在 60–3 600 s 之间的任何时刻, 复现漏洞的数量均高于 Mythril. 在复现时间为 120 s 时 (绿色竖线), Smart-Target 能够复现 85 个漏洞, 复现成功率达 52.80%, 此时 Mythril 仅能复现 47 个漏洞, 复现成功率为 29.19%. 在复现时间为 1 200 s 时 (红色竖线), Mythril 仍仅能复现 73 个漏洞. 这表明 Mythril 在多花费 10 倍时间的情况下仍无法复现与 Smart-Target 相同数量的漏洞.

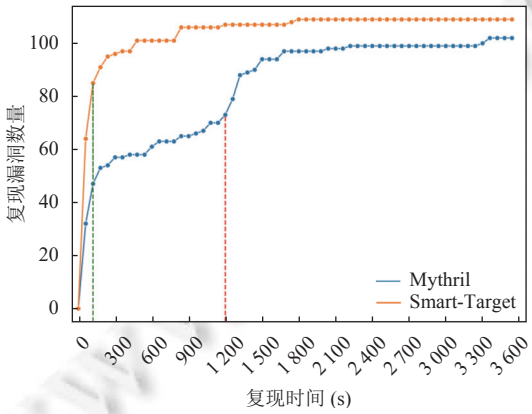


图 7　各工具在不同时刻的复现漏洞数量比较

此外, 在复现时间为 1 800 s 时, Smart-Target 达到最大复现漏洞数量 (109 个), 而此时 Mythril 相较于 Smart-Target 仍有 11 个漏洞无法复现. 总体上看, Smart-Target 复现 109 个漏洞所需的总时间为 16 340.69 s, 其中复现单

个漏洞所需最长时间为 1 789.98 s. Mythril 复现 109 个漏洞所需的总时间为 208 555.23 s, 比 Smart-Target 多花费 12.8 倍的时间, 其中复现单个漏洞所需最长时间为 35 226.65 s, 而 Smart-Target 仅需 1 166.21 s 即可成功复现该漏洞, 减少了 96.69% 的时间开销.

对于未能复现的 52 个漏洞中, Smart-Target 的分析时间为 5 396.06 s, Mythril 的分析时间为 175 824.48 s, 即 Smart-Target 减少了 96.93% 的分析时间, 这有助于及时更换其他检测工具或考虑人工复现漏洞.

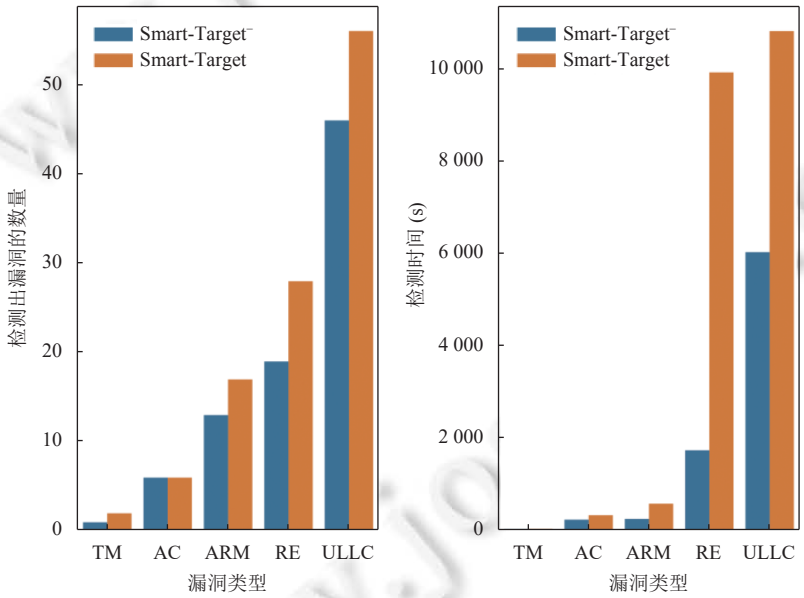
综上所述, 我们对 RQ2 的回答是: 在漏洞复现场景中, Smart-Target 的复现漏洞能力明显优于非目标制导的 Mythril. 在复现相同数量的安全漏洞情况下, Smart-Target 相较于 Mythril 减少了 92.16% 的复现时间.

5.2.3 RQ3 的结果与分析

为评估目标依赖语句分析对 Smart-Target 安全漏洞检测能力以及时间开销的影响, RQ3 在 RQ2 的基础上进行了一组对比试验, 比较不启用目标依赖语句分析的 Smart-Target$^-$ 与启用目标依赖语句分析的 Smart-Target 之间的性能差异.

RQ3 的实验结果如图 8 所示. 从图 8(a) 可以看出, 除了权限控制漏洞中两种策略检测到相同数量的漏洞外, 启用目标依赖语句分析的 Smart-Target 均能检测到更多安全漏洞. 具体地, Smart-Target 检测到 109 个漏洞, 而 Smart-Target$^-$ 检测到 85 个漏洞, 即开启目标依赖语句分析能够多检测 24 个 (22.02%) 漏洞, 有效增强了安全漏洞检测能力.

目标依赖语句分析在增强漏洞检测能力的同时, 也增加了符号执行的时间开销. 具体地, Smart-Target$^-$ 需要 8 361.01 s 完成安全漏洞检测, Smart-Target 增加了 13 375.74 s (61.54%) 的分析时间. 这是因为目标依赖语句分析引入了更多的语句作为符号执行需要遍历的路径和基本块. 如图 8(b) 所示, 在启用目标依赖语句分析后, 检测重入和未检测低级调用两种类型漏洞所花费的时间增幅最大.



(a) 安全漏洞检测数量对比 (b) 检测时间消耗对比

图 8 目标依赖语句对性能影响情况对比图

表 8 统计了 Smart-Target 执行目标依赖语句的数量. 如表 8 所示, 发现在存在重入和未检查低级调用两类漏洞的合约中执行了较多结构体赋值状态变量的语句. 智能合约中, 当状态变量通过非定长类型结构 (如动态数组或 mapping) 存储结构体时, 能够将存储的结构体以内存引用的形式赋值给局部变量, 通过对该局部变量的访问与修改, 达到赋值状态变量的目的. 例如图 9 中第 7 行从状态变量 *Acc* 中取出用户信息 *acc*, 并在第 8 和 9 行对

Holder 结构体类型的成员变量进行修改. 由于 mapping 中存储的是结构体地址, 因此修改 *acc* 会改变状态变量 *Acc* 中的数值. 由于第 14 行目标语句与第 13 行存在控制依赖, 且第 13 行与第 8 和 9 行赋值存在数据依赖, 因此第 8 和 9 行的赋值语句需要被符号化执行, 以更准确地判断目标语句的可达性. 然而, 结构体在符号执行分析的过程中所需要的测试资源较多, 需要对结构体中各个成员变量进行符号化表示, 导致分析结构体相关的赋值操作需要花费较多的时间. 需要注意的是, 结合 RQ1 和 RQ2 的分析可知, 相较于非目标制导的 Mythril, 启用了目标依赖语句分析的 Smart-Target 仍能在安全漏洞检测和复现场景中分别减少 60.76% 和 92.16% 的时间开销.

表 8　Smart-Target 执行的目标依赖语句数量统计

漏洞类型	直接赋值状态变量	结构体赋值状态变量
时间操作	8	0
权限控制	11	0
数值相关	73	0
重入	77	35
未检查低级调用	290	29
总计	459	64

```
1  struct Holder {                //结构体声明，定义用户相关信息
2      uint256 unlockTime;
3      uint256 balance;
4  }
5  mapping(address => Holder) public Acc;        //状态变量，记录用户相关信息
6  function Put(uint256 _unlockTime) public payable { //用户存储以太币的函数
7      var acc = Acc[msg.sender];                //从 mapping 中取出用户信息
8      acc.balance += msg.value;                 //记录用户发送的以太币金额
9      acc.unlockTime = _unlockTime > now ? _unlockTime : now;    //记录解锁时间
10 }
11 function Collect(uint256 _am) public payable {
12     var acc = Acc[msg.sender];
13     if (acc.balance >= MinSum && acc.balance >= _am && now > acc.unlockTime) {
14         if (msg.sender.call.value(_am)()) {       //重入漏洞
15             acc.balance -= _am;
16         }
17     }
18 }
```

图 9　通过结构体赋值状态变量的以太坊智能合约片段

综上所述, 我们对 RQ3 的回答是: 目标依赖语句能够有效提升 Smart-Target 的安全漏洞检测能力, 多检测到 22.02% 的安全漏洞; 在时间开销上, 当代码中存在较多结构体状态变量赋值操作时, 启用目标依赖语句分析会增加较多的符号执行分析时间.

6 讨　论

6.1 泛化性分析

用于开展 Smart-Target 的实验评估需要数据集提供行级别的漏洞语句信息, 即漏洞语句所在的代码行. 而 Ren 等人[21]、So 等人[41]以及 Luo 等人[45]的数据集仅对漏洞位置进行了文件级别的标注, 无法直接用于评估 Smart-Target 的有效性和泛用性. 为了进一步评估本文所提方法的泛用性, 我们参考并简化了 Zhang 等人[46]的测试框架, 在 Luo 等人[45]经过人工标注的 SCVHunter 数据集 (https://doi.org/10.6084/m9.figshare.24566893.v1) 基础上, 使用 GPT-4o 分析智能合约源代码, 并根据 GPT-4o 的回答标注疑似漏洞语句.

具体来讲, SCVHunter 数据集中包含重入、区块信息依赖、嵌套调用以及事务状态依赖 4 种类型漏洞. 根据 Luo 等人对上述漏洞的定义和描述, 区块信息依赖和事务状态依赖漏洞与 SB Curated 数据集中弱随机性以及权限控制漏洞的定义相似, 嵌套调用漏洞指在无限制循环中执行 CALL 指令, 导致 *gas* 不足引发异常, 进而影响合约正

常执行. 根据第 5.1 节中对 Mythril 支持检测的漏洞类型, 我们选择重入和事务状态依赖这 2 种类型的漏洞进行实验.

首先, 从上述 2 种漏洞类型中分别随机选取了 10 个智能合约文件. 然后, 使用图 10 所示的提示词 (Prompt) 向 GPT-4o 提问, 其中#Vul#为 SCVHunter 提供的漏洞标签, #Define#为 SCVHunter 论文中对#Vul#的定义和描述, #SourceCode#为待检测的智能合约源代码. 最后, 根据 GPT-4o 的回答, 标注该源代码中存在#Vul#类型漏洞的行号, 从而获得可用于实验的数据集.

Prompt:

Assuming you are an expert in Solidity smart contract vulnerability testing, now I will provide you with a Solidity source code that contains vulnerabilities of type #Vul#. The definition and description of this vulnerability are #Define#.

Please tell me (1) the line number in the source code where the #Vul# vulnerability exists, (2) the content of that line of source code, and (3) the name of the function in which that line is located.

The following is the source code for Solidity smart contracts:

#SolSourceCode#

Response:

...

图 10 用于向 GPT-4o 提问的 Prompt

将数据集中的漏洞语句作为 Smart-Target 的目标, 可用于检测 GPT-4o 所报出的疑似漏洞, 进而帮助开发和测试人员对其进行修复. 实验结果如表 9 所示, 其中 Mythril 的 $F1$ 值为 0.564, Smart-Target 的 $F1$ 值为 0.579. Smart-Target 在检测出相同数量的安全漏洞的同时, 其所需时间相较于 Mythril 减少了 71.05%. 实验结果表明, Smart-Target 在未减少漏洞检测能力的同时, 有效减少了测试所需的时间开销. 该实验结果与第 5.2.1 节中基于 SB Curated 数据集的实验结果相同, 表明 Smart-Target 的实验评估具有泛化性.

表 9 Smart-Target 和 Mythril 检测漏洞性能比较 (SCVHunter)

漏洞类型	工具	时间 (s)	TP	FP	FN	Recall	Precision	$F1$	测试用例生成数量	事务序列执行数量
重入	Smart-Target	12 448.29	3	0	7	0.3	1	0.462	3	228
	Mythril	38 850.62	3	0	7	0.3	1	0.462	3	484
权限控制	Smart-Target	243.54	8	0	9	0.471	1	0.640	8	24
	Mythril	4 990.93	8	1	9	0.471	0.889	0.615	9	98
总计	Smart-Target	12 691.83	11	0	16	0.407	1	0.579	11	252
	Mythril	43 841.55	11	1	16	0.407	0.917	0.564	12	582

6.2 有效性威胁

内部有效性分析主要与影响实验正确性的因素相关. 在本文中, 内部有效性主要体现在 Smart-Target 的实现是否正确. 本文基于 Slither 获得状态变量的操作语句 (读取或赋值)、修饰器的调用关系等信息, 分析目标语句与状态变量之间的依赖关系. Slither 开源并且更新频繁, 能够快速响应智能合约版本的变化, 提高了本方法的适用性. 此外, 本文使用 Solidity 编译器 solc (v0.4.25) 和 Python 第三方库 crytic-compile 编译源代码并获得 Source Mappings 信息, 利用 evm_cfg_builder 构建控制流图, 尽可能减少内部有效性威胁. 需要注意的是, 由于 evm_cfg_builder 没有完全模拟 EVM 堆栈, 因此无法分析部分 JUMP 指令的跳转地址, 导致可达性分析中构建的 CFG 不完整, 缺失部分基本块的制导信息. 本文的实验表明即使制导信息存在不完整的情况, 仍能够达到与非制导符号执行方法相同的安全漏洞检测能力, 未来工作可尝试在 evm_cfg_builder 的基础上完善 CFG 的构建, 从而更精确地制导符号执行.

外部有效性分析主要关注实验结果的一般性. 本文实验中所使用的数据集为 Durieux 等人[20]的实证研究中公开的 SB Curated 数据集以及 Luo 等人[45]的 SCVHunter 数据集. SB Curated 数据集包含 10 种漏洞类型, 并且漏洞信息由人工进行标注. 在实验过程中我们发现数据集在部分合约文件中不仅标记了触发漏洞的语句, 同时标记了漏洞相关变量声明或其他信息, 存在冗余标记的情况. 例如时间戳类型漏洞中不仅标记了存在时间戳判断条件的语句, 同时标记了获得时间戳的变量声明语句 (lottopollo, https://github.com/smartbugs/smartbugs/blob/master/dataset/time_manipulation/lottopollo.sol). 对于这类标记我们没有做特殊处理, 可能导致部分漏洞能够被 Slither、

Mythril 或 Smart-Target 检测, 但是实验结果记录为漏报的情况. 需要说明的是, 在实验过程中我们检查了 Slither 等工具的漏洞报告, 均没有在冗余标记的语句位置产生相应的漏洞报告, 因此冗余标记不会对实验结果中的 TP 或 FP 产生影响. SCVHunter 数据集同样为人工标注数据集, 但仅提供文件级的漏洞标注信息, 为了进一步确定漏洞语句的具体行号, 我们利用 GPT-4o 自动确认漏洞目标并得到行级别标注的数据集. 上述 2 种数据集包含了重入、权限控制以及整型溢出等智能合约中常见的安全漏洞, 可以在一定程度上确保数据集的代表性以及本文所提方法的泛化性, 未来工作考虑使用更多数据集进行实验验证, 以更全面的评估本文所提方法的有效性. 此外, 本文仅在 Mythril 的基础上实现了目标制导符号执行, 对于 Oyente (https://github.com/enzymefinance/oyente)、Manticore (https://github.com/trailofbits/manticore) 等其他符号执行工具, 可能由于符号执行的实现不同导致方法的性能有所差异. 本文所提出的目标依赖语句分析和可达性分析等步骤独立于符号执行的具体实现形式, 未来工作可尝试将本方法应用到其他基于符号执行的智能合约安全漏洞检测方法中, 比较不同符号执行技术对安全漏洞检测性能的影响.

构造有效性分析主要关注实验使用的评估指标. 本文使用安全漏洞检测领域常用的查全率、查准率以及 $F1$ 值作为评估指标, 并且增加了符号执行时间开销的统计与对比. 在安全漏洞复现场景的相关实验中, 使用漏洞复现领域常用的复现数量、复现成功率以及复现时间作为方法的评价指标[47,48], 尽可能从多个角度评价本方法的有效性.

7 总结与展望

本文提出了一种基于目标制导符号执行的智能合约安全漏洞检测方法. 首先根据静态分析报告或人工标注提供的目标语句位置, 分析与目标语句存在依赖关系的目标依赖语句; 然后进行可达性分析, 通过字节码构建控制流图, 定位目标语句以及目标依赖语句所在的基本块, 遍历并剪枝控制流图生成制导信息; 最后基于制导信息优化符号执行的路径探索策略, 跳过与目标无关的基本块, 减少需要被分析的基本块数量以及求解路径条件的时间, 高效地对目标语句进行安全漏洞检测. 实验结果表明, Smart-Target 在漏洞检测以及漏洞复现场景中均优于非目标制导的 Mythril 工具, 减少了 60.76% 的安全漏洞检测时间以及 92.16% 的复现时间, 提高了符号执行效率. 此外, 目标依赖语句分析能够使 Smart-Target 多检测到 22.02% 的安全漏洞, 有效提高了漏洞检测能力.

未来工作计划进一步针对复杂合约或存在嵌套调用的合约进行安全漏洞检测, 这类合约中可能存在多个漏洞目标, 可考虑使用多目标制导优化本文所提方法. 一方面, 可以根据各目标点的相关性、路径距离或探索难度等信息计算智能合约中各基本块的适应度, 在符号执行中基于适应度优化探索策略; 另一方面可以逐一进行目标制导符号执行, 借鉴现有工作中对约束求解的优化方法[49], 在符号执行过程中记录各个语句对应状态的路径条件, 在对其余目标点进行分析时, 根据状态信息匹配已知的路径条件以及约束求解结果, 提高符号执行效率. 此外, 还可将本文所提方法与并行符号执行、重用约束求解[50]等策略相结合, 进一步提高安全漏洞检测和复现的效率.

References:

[1] Yuan Y, Wang FY. Blockchain: The state of the art and future trends. Acta Automatica Sinica, 2016, 42(4): 481–494 (in Chinese with English abstract). [doi: 10.16383/j.aas.2016.c160158]

[2] Zheng ZB, Xie SA, Dai HN, Chen XP, Wang HM. An overview of blockchain technology: Architecture, consensus, and future trends. In: Proc. of the 2017 IEEE Int'l Congress on Big Data. Honolulu: IEEE, 2017. 557–564. [doi: 10.1109/BigDataCongress.2017.85]

[3] Ouyang LW, Wang S, Yuan Y, Ni XC, Wang FY. 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]

[4] 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]

[5] Azaria A, Ekblaw A, Vieira A, Lippman A. MedRec: Using blockchain for medical data access and permission management. In: Proc. of the 2nd Int'l Conf. on Open and Big Data. Vienna: IEEE, 2016. 25–30. [doi: 10.1109/OBD.2016.11]

[6] Alharby M, Aldweesh A, van Moorsel A. Blockchain-based smart contracts: A systematic mapping study of academic research (2018). In: Proc. of the 2018 Int'l Conf. on Cloud Computing, Big Data and Blockchain. Fuzhou: IEEE, 2018. 1–6. [doi: 10.1109/ICCBB.2018.8756390]

[7] Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. SmartCheck: Static analysis of ethereum smart contracts. In: Proc. of the 1st Int'l Workshop on Emerging Trends in Software Engineering for Blockchain. Gothenburg: ACM, 2018. 9–16. [doi: 10.1145/3194113.3194115]

[8] Xue YX, Ma ML, Yun L, Sui YL, Ye JM, Peng TY. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2020. 1029–1040.

[9] 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. Montreal: IEEE, 2019. 8–15. [doi: 10.1109/WETSEB.2019.00008]

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

[11] Torres CF, Iannillo AK, Gervais A, State R. ConFuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In: Proc. of the 2021 IEEE European Symp. on Security and Privacy. Vienna: IEEE, 2021. 103–119. [doi: 10.1109/EuroSP51992.2021.00018]

[12] Zhuang Y, Liu ZG, Qian P, Liu Q, Wang X, He QM. Smart contract vulnerability detection using graph neural network. In: Proc. of the 29th Int'l Conf. on Int'l Joint Conf. on Artificial Intelligence. Yokohama: ijcai.org, 2021. 454.

[13] Yang HW, Cui ZQ, Chen X, Jia MH, Zheng LW, Liu JB. Defect prediction for Solidity smart contracts based on software measurement. Ruan Jian Xue Bao/Journal of Software, 2022, 33(5): 1587–1611 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6550.htm [doi: 10.13328/j.cnki.jos.006550]

[14] Permenev A, Dimitrov D, Tsankov P, Drachsler-Cohen D, Vechev M. VerX: Safety verification of smart contracts. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 1661–1677. [doi: 10.1109/SP40000.2020.00024]

[15] So S, Hong S, Oh H. SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution. In: Proc. of the 30th USENIX Security Symp. USENIX Association, 2021. 1361–1378.

[16] Mossberg M, Manzano F, Hennenfent E, Groce A, Grieco G, Feist J, Brunson T, Dinaburg A. 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. San Diego: IEEE, 2019. 1186–1189. [doi: 10.1109/ASE.2019.00133]

[17] Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: ACM, 2016. 254–269. [doi: 10.1145/2976749.2978309]

[18] Chen JC, Xia X, Lo D, Grundy J, Luo XP, Chen T. DefectChecker: Automated smart contract defect detection by analyzing EVM bytecode. IEEE Trans. on Software Engineering, 2022, 48(7): 2189–2207. [doi: 10.1109/TSE.2021.3054928]

[19] Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Bünzli F, Vechev M. Securify: Practical security analysis of smart contracts. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. Toronto: ACM, 2018. 67–82. [doi: 10.1145/3243734.3243780]

[20] Durieux T, Ferreira JF, Abreu R, Cruz P. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 530–541. [doi: 10.1145/3377811.3380364]

[21] Ren M, Yin ZJ, Ma FC, Xu ZY, Jiang Y, Sun CN, Li HG, Cai Y. Empirical evaluation of smart contract testing: What is the best choice? In: Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. New York: ACM, 2021. 566–579. [doi: 10.1145/3460319.3464837]

[22] Ghaleb A, Pattabiraman K. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In: Proc. of the 29th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. New York: ACM, 2020. 415–427. [doi: 10.1145/3395363.3397385]

[23] 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 with English abstract). http://www.jos.org.cn/1000-9825/6375.htm [doi: 10.13328/j.cnki.jos.006375]

[24] Fu ML, Wu LF, Hong Z, Feng WB. 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]

[25] 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]

[26] 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). [doi: 10.3969/j.issn.1009-8054.2020.07.012]

[27] Tu LQ, Sun XB, Zhang JL, Cai J, Li B, Bo LL. Survey of vulnerability detection tools for smart contracts. Computer Science, 2021, 48(11): 79–88 (in Chinese with English abstract). [doi: 10.11896/jsjkx.210600117]

[28] Gan ST, Wang LZ, Xie XH, Qin XJ, Zhou L, Chen ZN. Guiding symbolic execution analysis by leveraging program function label slice.

Ruan Jian Xue Bao/Journal of Software, 2019, 30(11): 3259–3280 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5562.htm [doi: 10.13328/j.cnki.jos.005562]

[29] Bao TY, Gao FJ, Zhou Y, Li Y, Wang LZ, Li XD. Automatically validating static buffer overflow warnings based on guided symbolic execution. Journal of Cyber Security, 2016, 1(2): 46–60 (in Chinese with English abstract). [doi: 10.19363/j.cnki.cn10-1380/tn.2016.02.005]

[30] Cui ZQ, Wang LZ, Li XD. Target-directed concolic testing. Chinese Journal of Computers, 2011, 34(6): 953–964 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.00953]

[31] Choi J, Kim D, Kim S, Grieco G, Groce A, Cha SK. SMARTIAN: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2021. 227–239. [doi: 10.1109/ASE51524.2021.9678888]

[32] Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 80–109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[33] Mei H, Wang QX, Zhang L, Wang J. Software analysis: A road map. Chinese Journal of Computers, 2009, 32(9): 1697–1710 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2009.01697]

[34] Cadar C, Godefroid P, Khurshid S, Păsăreanu CS, Sen K, Tillmann N, Visser W. Symbolic execution for software testing in practice: Preliminary assessment. In: Proc. of the 33rd Int'l Conf. on Software Engineering. Honolulu: IEEE, 2011. 1066–1071. [doi: 10.1145/1985793.1985995]

[35] Nikolić I, Kolluri A, Sergey I, Saxena P, Hobor A. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proc. of the 34th Annual Computer Security Applications Conf. San Juan: ACM, 2018. 653–663. [doi: 10.1145/3274694.3274743]

[36] Huang JJ, Jiang JS, You W, Liang B. Precise dynamic symbolic execution for nonuniform data access in smart contracts. IEEE Trans. on Computers, 2022, 71(7): 1551–1563. [doi: 10.1109/TC.2021.3092639]

[37] Guo SJ, Kusano M, Wang C, Yang ZJ, Gupta A. Assertion guided symbolic execution of multithreaded programs. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo: ACM, 2015. 854–865. [doi: 10.1145/2786805.2786841]

[38] Ge X, Taneja K, Xie T, Tillmann N. DyTa: Dynamic symbolic execution guided with static verification results. In: Proc. of the 33rd Int'l Conf. on Software Engineering. Honolulu: IEEE, 2011. 992–994. [doi: 10.1145/1985793.1985971]

[39] Li MC, Chen YJ, Wang LZ, Xu GQ. Dynamically validating static memory leak warnings. In: Proc. of the 2013 Int'l Symp. on Software Testing and Analysis. Lugano: ACM, 2013. 112–122. [doi: 10.1145/2483760.2483778]

[40] Li X, Zhou Y, Li MC, Chen YJ, Xu GQ, Wang LZ, Li XD. Automatically validating static memory leak warnings for C/C++ programs. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 827–844 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5189.htm [doi: 10.13328/j.cnki.jos.005189]

[41] So S, Lee M, Park J, Lee H, Oh H. VERISMART: A highly precise safety verifier for Ethereum smart contracts. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 1678–1694. [doi: 10.1109/SP40000.2020.00032]

[42] Wang HJ, Liu T, Guan XH, Shen C, Zheng QH, Yang ZJ. Dependence guided symbolic execution. IEEE Trans. on Software Engineering, 2017, 43(3): 252–271. [doi: 10.1109/TSE.2016.2584063]

[43] Chen T, Zhang XS, Guo SZ, Li HY, Wu Y. State of the art: Dynamic symbolic execution for automated test generation. Future Generation Computer Systems, 2013, 29(7): 1758–1773. [doi: 10.1016/j.future.2012.02.006]

[44] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 209–224.

[45] Luo F, Luo RJ, Chen T, Qiao A, He ZY, Song SW, Jiang Y, Li X. SCVHUNTER: Smart contract vulnerability detection based on heterogeneous graph attention network. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: IEEE, 2024. 2098–2110.

[46] Zhang CY, Liu H, Zeng JT, Yang KJ, Li YH, Li H. Prompt-enhanced software vulnerability detection using ChatGPT. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering: Companion Proc. Lisbon: IEEE, 2024. 276–277. [doi: 10.1145/3639478.3643065]

[47] Li SY, Guo JQ, Fan M, Lou JG, Zheng QH, Liu T. Automated bug reproduction from user reviews for Android applications. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering: Software Engineering in Practice. Seoul: ACM, 2020. 51–60. [doi: 10.1145/3377813.3381355]

[48] Zhao Y, Yu TT, Su T, Liu Y, Zheng W, Zhang JZ, Halfond WGJ. ReCDroid: Automatically reproducing Android application crashes from bug reports. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 128–139. [doi: 10.1109/ICSE.2019.00030]

[49] Romano A, Engler D. Expression reduction from programs in a symbolic binary executor. In: Proc. of the 20th Int'l Symp. on Model Checking Software. Stony Brook: Springer, 2013. 301–319. [doi: 10.1007/978-3-642-39176-7_19]

[50] Visser W, Geldenhuys J, Dwyer MB. Green: Reducing, reusing and recycling constraints in program analysis. In: Proc. of the 20th ACM SIGSOFT Int'l Symp. on the Foundations of Software Engineering. Cary North: ACM, 2012. 58. [doi: 10.1145/2393596.2393665]

附中文参考文献:

[1] 袁勇, 王飞跃. 区块链技术发展现状与展望. 自动化学报, 2016, 42(4): 481–494. [doi: 10.16383/j.aas.2016.c160158]

[3] 欧阳丽炜, 王帅, 袁勇, 倪晓春, 王飞跃. 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]

[13] 杨慧文, 崔展齐, 陈翔, 贾明华, 郑丽伟, 刘建宾. 基于软件度量的 Solidity 智能合约缺陷预测方法. 软件学报, 2022, 33(5): 1587–1611. http://www.jos.org.cn/1000-9825/6550.htm [doi: 10.13328/j.cnki.jos.006550]

[23] 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 [doi: 10.13328/j.cnki.jos.006375]

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

[25] 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]

[26] 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. [doi: 10.3969/j.issn.1009-8054.2020.07.012]

[27] 涂良琼, 孙小兵, 张佳乐, 蔡杰, 李斌, 薄莉莉. 智能合约漏洞检测工具研究综述. 计算机科学, 2021, 48(11): 79–88. [doi: 10.11896/jsjkx.210600117]

[28] 甘水滔, 王林章, 谢向辉, 秦晓军, 周林, 陈左宁. 一种基于程序功能标签切片的制导符号执行分析方法. 软件学报, 2019, 30(11): 3259–3280. http://www.jos.org.cn/1000-9825/5562.htm [doi: 10.13328/j.cnki.jos.005562]

[29] 鲍铁匀, 高凤娟, 周严, 李游, 王林章, 李宣东. 基于目标制导符号执行的静态缓冲区溢出警报自动确认技术. 信息安全学报, 2016, 1(2): 46–60. [doi: 10.19363/j.cnki.cn10-1380/tn.2016.02.005]

[30] 崔展齐, 王林章, 李宣东. 一种目标制导的混合执行测试方法. 计算机学报, 2011, 34(6): 953–964. [doi: 10.3724/SP.J.1016.2011.00953]

[32] 张健, 张超, 玄跻峰, 熊英飞, 王千祥, 梁彬, 李炼, 窦文生, 陈振邦, 陈立前, 蔡彦. 程序分析研究进展. 软件学报, 2019, 30(1): 80–109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[33] 梅宏, 王千祥, 张路, 王戟. 软件分析技术进展. 计算机学报, 2009, 32(9): 1697–1710. [doi: 10.3724/SP.J.1016.2009.01697]

[40] 李筱, 周严, 李孟宸, 陈园军, Xu GQ, 王林章, 李宣东. C/C++程序静态内存泄漏警报自动确认方法. 软件学报, 2017, 28(4): 827–844. http://www.jos.org.cn/1000-9825/5189.htm [doi: 10.13328/j.cnki.jos.005189]

杨慧文(1997－), 男, 硕士生, 主要研究领域为智能合约漏洞检测技术.

郑丽伟(1979－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为需求软件工程, 社交网络, 数据质量增强.

崔展齐(1984－), 男, 博士, 教授, CCF 高级会员, 主要研究领域为智能化软件工程, 可信人工智能.

刘建宾(1963－), 男, 博士, 教授, 主要研究领域为智能软件技术, 模型驱动开发.

陈翔(1980－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件缺陷预测, 软件缺陷定位, 组合测试.