软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007634] [CSTR: 32375.14.jos.007634]

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

基于静态与动态分析的智能合约漏洞检测实证研究*

祝 语[1,2,3,4], 李 珍[1,2,3,4], 张笑睿[1,2,3,4], 吴月明[1,2,3,4], 邹德清[1,2,3,4]

[1](大数据技术与系统国家地方联合工程研究中心 (华中科技大学), 湖北 武汉 430074)

[2](服务计算技术与系统教育部重点实验室 (华中科技大学), 湖北 武汉 430074)

[3](分布式系统安全湖北省重点实验室 (华中科技大学), 湖北 武汉 430074)

[4](华中科技大学 网络空间安全学院, 湖北 武汉 430074)

通信作者: 李珍, E-mail: zh_li@hust.edu.cn

摘 要: 近年来, 智能合约因其不可篡改性和可执行性在金融领域得到了广泛的应用. 与此同时, 智能合约引发的安全事件不断增多, 往往造成大规模的经济损失. 因此, 大多数研究人员致力于开发智能合约漏洞检测工具来检测智能合约的安全性. 然而, 不同的合约漏洞检测工具可能因为数据集的不一致使得研究人员无法客观评估其性能. 构建一个新的数据集, 并在统一的标准下系统地测试 9 个候选工具. 该数据集不仅包含真实世界的智能合约, 且涵盖了 5 类常见的智能合约漏洞. 从 5 个方面对工具进行细致的评估, 并提出一种新的智能合约分类方法来验证工具的鲁棒性. 实验结果表明: 1) 多数现有工具易于安装, 但同时存在停止维护等问题. 2) 静态检测工具在实际检测过程中仍然面临漏报率和误报率高的问题, 且依赖于智能合约不同版本的分析. 3) 静态检测工具的时间开销较小, 而基于符号执行的工具容易因状态爆炸而导致时间开销较大. 4) 部分静态检测工具不支持具有复杂继承关系的合约. 5) 采用多种漏洞检测技术融合的方式, 可以有效提高检测的精确率和召回率.

关键词: 智能合约; 漏洞检测; 基准数据集; 工具性能评测; 实证研究

中图法分类号: TP311

中文引用格式: 祝语, 李珍, 张笑睿, 吴月明, 邹德清. 基于静态与动态分析的智能合约漏洞检测实证研究. 软件学报. http://www.jos.org.cn/1000-9825/7634.htm

英文引用格式: Zhu Y, Li Z, Zhang XR, Wu YM, Zou DQ. Empirical Study on Smart Contract Vulnerability Detection Based on Static and Dynamic Analysis. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7634.htm

Empirical Study on Smart Contract Vulnerability Detection Based on Static and Dynamic Analysis

ZHU Yu[1,2,3,4], LI Zhen[1,2,3,4], ZHANG Xiao-Rui[1,2,3,4], WU Yue-Ming[1,2,3,4], ZOU De-Qing[1,2,3,4]

[1](National Engineering Research Center for Big Data Technology and System (Huazhong University of Science and Technology), Wuhan 430074, China)

[2](Key Laboratory of Services Computing Technology and System (Huazhong University of Science and Technology), Ministry of Education, Wuhan 430074, China)

[3](Hubei Key Laboratory of Distributed System Security (Huazhong University of Science and Technology), Wuhan 430074, China)

[4](School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China)

Abstract: In recent years, smart contracts have been widely used in the financial field due to their immutability and enforceability. At the same time, the number of security incidents caused by smart contracts has continued to increase, often causing large-scale economic losses. Therefore, most researchers focus on developing vulnerability detection tools to assess the security of smart contracts. However, the performance of different vulnerability detection tools cannot be objectively evaluated due to the use of inconsistent datasets. This study constructs a new dataset and systematically tests nine candidate tools under a unified benchmark. The dataset includes real-world smart

* 基金项目: 国家自然科学基金 (U2336203)
收稿时间: 2025-05-07; 修改时间: 2025-09-23; 采用时间: 2026-01-19; jos 在线出版时间: 2026-05-20

contracts and covers five common types of vulnerabilities. This study evaluates the tools from five aspects and proposes a new smart contract classification method to verify the robustness of the tools. The experimental results are as follows. 1) Most existing tools are easy to install, but there are also problems such as discontinued maintenance. 2) Static detection tools still face the problem of high false positive and false negative rates in the actual detection process, and rely on the analysis of different versions of smart contracts. 3) Static detection tools have a small time overhead, while tools based on symbolic execution are prone to large time overhead due to state explosion. 4) Some static detection tools do not support contracts with complex inheritance relationships. 5) The integration of multiple vulnerability detection technologies can effectively improve the detection precision and recall.

Key words: smart contract; vulnerability detection; benchmark dataset; tool performance evaluation; empirical study

智能合约是一套控制数字资产的数字承诺, 它包含合约参与方约定的权利和义务[1], 由计算机系统自动执行. 其特点是一旦满足合约规定的条件就可以强制执行, 整个过程不涉及第三方, 一定程度上可以避免人工干预. 早在 1994 年该技术就由密码学家 Szabo[2]提出, 但由于缺乏可信的执行环境, 其始终没有被运用于实践中. 区块链通过提供一套完整的信任机制, 为这项技术的大规模应用奠定了基础. 近年来, 智能合约被广泛应用于金融借贷、房屋租赁、虚拟货币交易等涉及金融属性的应用场景中[3]. 因此, 智能合约中的漏洞有时比传统语言中的漏洞更致命. DAO 事件[4]和 Parity multisig 钱包事件[5]充分体现了这一特点, 这两者都因智能合约漏洞而产生重大经济损失. 由于区块链[6]的不可篡改特性, 对于已经部署在链上的合约, 很难通过打补丁以提升其安全性.

为了应对这一挑战, 许多研究工作都集中在智能合约漏洞的离线检测上. 现有的智能合约检测方案在检测技术上存在差异. 根据是否部署和执行合约, 主要可以分为基于动态和基于静态两种. 其中, 动态检测技术是在合约执行过程中对合约行为进行监测. 通过这种方式, 可以准确地检测或触发相关漏洞. 目前智能合约漏洞检测中使用的动态检测技术主要是模糊测试 (如 ContractFuzzer[7]、ILF[8]、sFuzz[9]、Harvey[10]). 相反, 静态检测技术则偏向于获取合约的静态特征, 从而识别合约是否存在漏洞. 静态检测技术中的代表是符号执行 (如 Oyente[11]) 和污点分析 (如 Osiris[12]) 等.

由于工具的测试指标和实验数据集的不同, 这使得用户难以选择合适的检测工具. 针对这个问题, 已有部分工作通过构建统一的测试平台来评估检测工具的效果. 例如 Ghaleb 等人[13]提出了一种评估智能合约静态分析工具 SolidiFI, 并利用它构建有针对性的漏洞来验证工具的检测效果. Durieux 等人[14]提出 SmartBugs 框架, 将 9 个自动化分析工具集成到框架中, 并利用统一的数据集进行测试. Ren 等人[15]提出了一个统一的标准来消除评估过程中的偏差, 并构建了包含 46 186 个智能合约的数据集来测试 9 个代表性工具. 该工作中 45 622 个合约是无标签的, 因而无法准确体现工具的漏报率. 总的来说, 上述工作大多缺乏无漏洞数据或智能合约项目数据, 因此无法准确评估检测工具的漏报率和误报率.

本文收集了真实世界智能合约、包含 CVE 漏洞的智能合约以及智能合约的第三方审计数据并基于此构建了一个新的智能合约数据集. 为了更好地评估现有智能合约漏洞检测工具的能力, 本文从 5 个不同的角度对合约漏洞检测中的代表性工具进行了全面的实证研究. 1) 工具易用性: 通过判断工具是否容易安装部署, 来体现工具的用户友好性. 调研结果表明, 大多数漏洞检测工具提供了清晰且便捷的安装步骤. 然而, 它们中的多数已经不再支持维护. 2) 检测准确性: 在包含有漏洞和无漏洞合约的数据集上对这些工具进行评测, 并给出公平的有效性比较结果. 结果表明, 当面对访问控制漏洞等部分漏洞时, 现有工具检测效果不佳或不支持检测这些类型的漏洞. 3) 检测效率: 通过评估这些工具的运行时开销, 分析其效率. 实验结果表明, 使用符号执行等技术的检测工具在检测过程中具有相对较大的时间开销. 4) 复杂度敏感性: 本文设计了一种新的分类方法, 将数据集按照复杂程度进行分类. 通过分类测试, 可以判断哪些工具更具有鲁棒性. 通过分析结果, 本文发现这些检测工具在针对具有复杂调用关系的合约检测任务中均表现不佳. 5) 工具组合效应: 通过组合采用不同检测技术的工具进行漏洞检测, 评估不同工具之间的互补性. 实验结果表明, 不同工具的组合可以达到互补的效果, 仅损失极小的精确率或召回率就可以进一步提升整体检测效果.

综上所述, 本文的主要贡献如下.

(1) 收集并构建一个包含真实世界合约、含有 5 种常见漏洞类型的合约和无漏洞合约的智能合约数据集.

(2) 提出一种新的智能合约分类方法, 该方法结合了合约本身的特征以及继承调用关系. 与传统的智能合约特征指标相比, 它更能全面地反映合约样本的特征.

(3) 从 5 个方面对候选工具进行系统的评估. 通过实验结果, 指出了现有检测技术的局限性和未来可行的研究方向.

1 相关工作

近年来, 针对智能合约漏洞及其检测技术进行了大量的综述工作. Atzei 等人[16]首次系统地讨论了以太坊及其高级编程语言中的安全漏洞, 并对漏洞分类后的原理进行了分析. Chen 等人[17]将以太坊的漏洞分为 4 个级别, 主要分为应用层漏洞、数据层漏洞、共识层漏洞和网络层漏洞, 并重点研究了智能合约漏洞及其防御策略. 倪远东等人[18]分析了智能合约在不可更改、公开透明等特性下所衍生的全新安全风险, 提出了三层威胁模型. 以以太坊为例, 归纳了 15 类智能合约漏洞类型并总结了自动化漏洞挖掘、利用与防御的研究现状与挑战. 闫凯伦等人[19]从静态分析与动态分析两大方向系统介绍了 21 种检测技术和工具, 比较了它们在方法、对象和检测能力上的优劣, 展望未来可行研究方向. 董伟良等人[20]系统性梳理了 84 篇智能合约漏洞检测论文, 按照检测技术对这些工作进行分类, 深入分析了各技术的实现手段、目标漏洞类型与实验数据, 并对比了国内外研究现状的差异. 崔展齐等人[21]提出了一个涵盖漏洞发现与识别、漏洞分析与检测、数据集与评价指标这 3 方面的智能合约安全漏洞检测研究框架, 并基于漏洞基础特征归纳出 13 种漏洞类型. 在此基础上, 对 5 种检测技术进行了全面分析, 讨论了各自的优势与局限并对未来研究方向进行展望. 总的来说, 上述工作大多侧重于对漏洞原理与检测技术进行分类与原理阐述, 缺乏对工具实际性能的评估与对比.

与综述研究不同的是, 实证研究侧重于将检测工具在统一合约数据集上进行部署与测试, 量化评估这些工具的误报率、漏报率以及运行开销等指标. Ghaleb 等人[13]提出了 SolidiFI, 一种评估智能合约静态分析工具和系统的方法. 它通过自动插入漏洞构建测试样本, 解决了实验数据集不足的问题. Durieux 等人[14]提出了 SmartBugs, 它将 9 个自动化分析工具集成到框架中, 并利用统一的数据集进行测试. Ren 等人[15]构建了统一的测试套件来消除评估过程中的偏差, 并构建了包含 46 186 个智能合约的数据集来测试 9 个代表性工具. 钱鹏等人[22]系统介绍了 5 种不同的检测技术, 并构建了包含 300 个合约的数据集对工具可检测漏洞类型、精确率和时间消耗等指标进行了对比分析. 但上述工作大多缺乏无漏洞数据或智能合约项目数据, 因此无法准确体现检测工具在真实环境中的检测效果.

2 基础知识

本节将对涉及的基础知识进行介绍, 主要包括智能合约漏洞及其检测技术.

2.1 智能合约漏洞

智能合约有许多不同于传统程序的特征, 这些特征催生了新的攻击方法, 从而产生了新的漏洞类型. 根据智能合约以及运行实体的生命周期, 这些漏洞可以按照程序语言层、以太坊虚拟机 (EVM) 层和区块链层[16]进行分类, 如表 1 所示.

表 1 智能合约的漏洞分类

层级	智能合约漏洞
程序语言层	算术漏洞 未检查的低级调用 重入漏洞 访问控制漏洞
EVM层	栈大小限制漏洞 短地址漏洞
区块链层	时间戳依赖漏洞 差随机性漏洞 区块号依赖漏洞

程序语言层: Solidity 语言层级通常指开发以太坊智能合约时使用的高级语言造成的漏洞. 该级别的漏洞主要来自两个方面: (1) 智能合约程序开发设计过程中涉及的逻辑漏洞; (2) 智能合约编程语言本身存在的缺陷. 在编程语言层面, 存在重入漏洞 (DAO 事件)、整数溢出 (BEC 令牌) 等多种经典漏洞类型.

● 算术漏洞: 在执行数值计算时, 计算结果超过了合约类型所代表的上限或小于下限. 该漏洞通常会导致最大值变为最小值或者使得最小值变为最大值.

● 未检查的低级调用: 当使用低级函数调用如 call、send 和 delegatecall 等时, 调用的返回结果是不被检查的. 该漏洞可能导致转账失败未被感知、状态更新错误等问题.

● 重入漏洞: 在合约转账过程中, 先转账再更新状态. 攻击者劫持转账过程, 利用递归调用实现重复转账.

● 访问控制漏洞: 在合约开发过程中, 权限修饰符逻辑的错误使用导致敏感函数被攻击者非法调用.

EVM 层: EVM 级别通常是指编译后的智能合约字节码执行器造成的漏洞. 该级别的漏洞主要是由于 EVM 的设计缺陷. 其中最著名的是 EVM 调用栈限制攻击和短地址攻击.

● 栈大小限制漏洞: 合约每被调用一次, 与交易相关联的调用栈就增长一帧. 调用栈大小限制为 1 024 帧, 当调用次数达到上限时, 进一步的调用将会引发异常.

● 短地址漏洞: 攻击者构造一个以 0 结尾的地址来调用合约并在参数调用时省去地址尾部的 0, 利用虚拟机的自动补全机制对参数进行移位和放大.

区块链层: 区块链级别通常指区块链系统造成的漏洞. 该级别的漏洞主要是因为智能合约的一些关键控制流的状态变化与区块链本身的特性相关联. 其中, 最著名的是时间戳依赖漏洞、差随机性漏洞和区块号依赖漏洞等.

● 时间戳依赖漏洞: 智能合约使用代码中的区块时间戳来作为敏感操作 (例如 transfer) 的判断条件, 但时间戳在一定范围内是能被操纵的, 从而引入安全漏洞.

● 差随机性漏洞: 许多与区块链相关的变量在智能合约中被滥用为随机来源, 但这种生成的随机数是可以预测的.

● 区块号依赖漏洞: 与时间戳依赖类似, 智能合约在其代码中使用区块号来作为敏感操作 (例如 transfer) 的判断条件, 但矿工可以在一定范围内对其产生影响, 从而引入安全漏洞.

2.2 智能合约漏洞检测技术

本文聚焦于智能合约漏洞检测领域较著名的工作. 这些论文提出了自己的合约漏洞检测工具, 并开源了对应的项目源码. 根据检测技术, 这些工具主要分为 4 类: 模糊测试、符号执行、形式化验证和其他技术.

● 模糊测试: 这是一种动态检测技术, 主要用于传统软件工程中的漏洞挖掘. 它根据要测试的目标程序生成一系列输入, 这些输入大多是随机或半随机生成的. 这些输入可以驱动程序运行, 并在执行过程中监控并记录程序的运行状态. 通过分析状态, 可以收集目标程序的异常并发现潜在的漏洞. 对于智能合约而言, 模糊测试主要根据合约的接口函数生成一系列交易, 并记录执行结果. 通过将交易执行过程中的数据或结果与预定义的漏洞规则或检测器进行匹配, 可以判断合约中是否存在相关漏洞.

在智能合约漏洞检测领域, Jiang 等人[7]将传统程序中的模糊测试方法迁移到智能合约漏洞检测中, 首次提出了一套针对智能合约的模糊测试工具, 后来的研究者大多在此基础上优化交易生成和完善变异策略. 这类技术的代表作品包括 ContractFuzzer[7]、Harvey[10]、ILF[8]和 sFuzz[9]等. 目前, 智能合约领域的模糊测试技术仍存在诸多问题. 例如, 在交易生成方面, 生成的交易往往无法满足复杂分支的判定条件, 如哈希值匹配、签名验证等, 从而导致交易无效, 难以到达更深层的程序状态, 进而无法稳定触发潜在漏洞. 在漏洞判定方面, 由于漏洞规则本身不够准确, 检测结果可能出现漏报和误报.

● 符号执行: 该技术是一种静态检测技术, 其核心思想是在执行过程中使用符号作为输入, 而不是具体的数值[23]. 当程序执行过程中产生分支时, 符号执行引擎会将该分支的逻辑判断以符号表达式的形式记录下来, 形成相应的路径约束. 最后, 通过约束求解器对路径约束进行求解, 即可生成触发某个分支的输入. 在智能合约漏洞检测领域,

现有工作通常会利用源代码编译得到的字节码, 反汇编成操作码. 然后利用操作码构建基本块和控制流图, 并结合漏洞特征得到相关的路径约束. 最后使用约束求解器进行求解并判断是否存在漏洞. 代表性工具有 Oyente[11]、Osiris[12]、Mythril[24]等. 但是智能合约是一个有限状态机, 它的状态变化需要通过交易来实现. 因此要达到更深层次的程序状态, 可能需要构造更复杂的交易序列. 符号执行技术在处理这个问题时会遇到路径爆炸、约束求解难的情况, 这也是该技术在智能合约漏洞检测任务上出现漏报、误报的主要原因.

● 形式化验证: 形式化验证是利用数学知识来证明和验证一个系统的安全性[25]. 在智能合约领域, 合约通常会被转换成一种中间语言或一种易于形式化的语言. 根据漏洞原理确定相应的验证规则后, 将转换后的合约模型与规则结合从而生成需要证明的验证条件. 最后通过 SMT 求解、符号模型检查或交互式证明来判断合约是否存在对应漏洞. 形式化验证的代表性工具包括 ZEUS[26]、Securify[27]和 VerX[28]等. 但目前, 智能合约的形式化验证仍然依赖专家在规则定义等环节的人工干预, 难以实现全过程自动化. 与此同时, 随着智能合约规模和逻辑复杂度的增加, 状态空间和验证条件急剧膨胀, 这导致工具常常因计算量过大而无法在可接受的时间和资源内完成验证.

● 其他技术: 其他工具使用的技术类型包括污点分析[12]等静态检测技术. 其中, 污点分析通过对源点、汇点进行标记与追踪, 以实现更精确的数据流分析, 这在合约金额的转移中尤为关键. 此外, 还有不少工作将多种静态检测技术或静态与动态技术进行组合, 以进一步提升工具检测性能.

3 实验设计

3.1 工具选择

本文收集整理了近年在顶级软件工程会议 (ICSE、FSE、ISSTA、ASE)、安全会议 (S&P、USENIX Security、CCS、NDSS) 以及软件工程和安全领域权威期刊 (TDSC、TSE、TIFS、TOSEM) 上发表的智能合约相关论文, 共计 280 篇. 筛选掉与智能合约漏洞检测无关的文章后, 最终获得 42 篇与智能合约漏洞检测相关的论文. 针对这些文章, 从以下几个量化维度对已开源相关代码的工具进行筛选.

(1) 学术代表性: 通过统计 Google Scholar 上相关论文的总引用次数, 衡量工具在学术界的影响力.

(2) 社区流行度: 统计工具在 GitHub 上的星标数, 反映工具的实际使用情况.

(3) 技术覆盖面: 确保候选工具涵盖符号执行 (SE)、模糊测试 (F)、静态分析 (SA) 等主流分析方法.

(4) 漏洞覆盖度: 工具支持检测的漏洞类型, 包括重入漏洞 (RE)、时间戳依赖漏洞 (TD)、访问控制漏洞 (AC)、未检查的低级调用 (ULC) 等.

基于上述指标, 本文最终确定了 9 个具有代表性的工具. 如表 2 所示, 它们既代表了学术界和工业界的主流实践, 也涵盖了智能合约漏洞检测的主要技术路线. 表 2 中, 对于分析方法, SE 表示符号执行, TA 表示污点分析, SA 表示静态分析, F 表示模糊测试; RE、TD、AC、ULC 和 A 分别表示重入漏洞、时间戳依赖漏洞、访问控制漏洞、未检查的低级调用和算术漏洞. 特别地, Mythril 工具没有直接相关的技术论文, 因而无法统计其论文被引用次数.

表 2 所选工具及其量化特征

工具	方法	支持的漏洞类型	谷歌学术引用次数	GitHub星标数 (★)
Oyente	SE	RE, TD, A	3 039	1 300
Osiris	TA&SE	RE, TD, A	507	63
Securify	SE	RE, AC, ULC	1 383	603
Vandal	SA	RE, ULC	418	162
Mythril	TA&SE	RE, TD, A, ULC	—	4 100
SmartCheck	SA	RE, A, AC, ULC	1 056	375
Slither	SA	RE, TD, ULC, AC, A	908	5 700
sFuzz	F	RE, TD, ULC, A	410	101
ILF	F	RE, TD, ULC	375	154

Oyente[11]: Oyente 是智能合约漏洞检测领域的知名工具. 实验中使用其 0.2.7 版本. 该工具以智能合约字节码为输入, 构建合约的基本控制流图. 该图以基本块为节点, 以跳转关系为边. 在符号化执行过程中, 动态探索程序控制流图, 得到并求解相应的路径约束, 最后与漏洞规则进行匹配, 从而实现智能合约的漏洞检测.

Osiris[12]: 基于 Oyente 工具实现. 实验中使用其 0.0.1 版本. 它在 Oyente 的基础上增加了污点分析并标记了操作数的来源和操作结果的传递方向. 该改动过滤掉了不敏感的操作, 从而减少了工具对整数溢出的误报.

Securify[27]: 是一种基于静态分析的智能合约安全分析工具, 实验中使用其 2.0 版本. 该工具通过分析合约程序表示, 推导控制流、数据流以及安全相关的语义事实, 并进一步利用 Datalog 规则检查这些事实是否满足预定义的安全属性或漏洞模式, 从而识别潜在的安全风险.

Vandal[29]: 是一个使用静态分析的静态检测工具. 实验中使用其唯一公开版本. 它从操作码中提取控制流图, 并将其编码为中间语言表示. 最后, 将这个表示形式输入到求解器中以提取合约特征.

Mythril[24]: 与 Securify 类似, 它是一种使用符号执行的静态检测工具. 实验中使用其 0.23.10 版本. 它利用符号执行、SMT 求解和污点分析来检测各种安全漏洞.

SmartCheck[30]: 与 Vandal 类似, 也被转换成中间语言进行后续处理. 实验中使用其 0.2 版本. 它将源代码转换为基于 XML 的中间表示, 然后根据定义的漏洞模式对其进行检查.

Slither[31]: 工作模式类似于 SmartCheck, 它还将代码特性转换为中间语言表示. 实验中使用其 0.9.1 版本. 它将源代码编译生成的 AST 作为输入, 获取合约的继承图、控制流图等信息. 然后将合约代码转换为 SlithIR, 并将中间表示与定义的分析规则进行匹配, 以检测合约漏洞.

sFuzz[9]: 整体上与 ContractFuzzer 的框架类似. 不同的是, 它引入了启发式距离策略来优化测试用例的生成, 使得测试用例的构建更有针对性. 本文使用其最新版本的 GitHub 项目.

ILF[8]: 是一种基于神经网络的智能合约模糊测试方案, 它利用模仿学习获取符号执行的调用序列特征并生成模型. 最后, 利用该模型指导模糊测试过程中的交易序列构建. 本文使用其最新版本的 GitHub 项目.

在本文的实验中, 各智能合约漏洞检测工具均采用其原论文或官方文档中推荐的默认参数进行设置, 以保证实验结果的可复现性和公平性. 未对工具参数进行额外调整或优化, 实验结果仅反映工具在默认配置下的性能和检测能力.

3.2 研究问题

基于上述选定的检测工具和漏洞类型, 本文提出了以下 5 项研究问题以系统评估这些智能合约漏洞检测工具的可用性、性能以及组合效应.

RQ1: 工具易用性——检测工具的友好程度如何?

本文对选定的合约漏洞检测工具进行了系统的整理与分析. 具体来说, 在统一服务器环境下将它们分别部署, 评估其依赖安装、环境配置等方面以反映其对用户的友好程度. 该问题旨在评估工具易用性, 为用户选择审计工具提供依据, 从而减少其在审计过程中因环境/工具配置带来的额外的负担.

RQ2: 检测准确性——检测工具的检测效果如何?

本文用查准率和查全率作为工具检测准确性的衡量标准. 将候选工具置于相同硬件配置下使用相同数据集比较各工具的检测效果. 将工具检测到的标签值与人工标注的结果进行比较, 并计算以下指标. *TP*: 实际有漏洞且工具检测为有漏洞; *FP*: 检测到漏洞, 但实际没有漏洞; *FN*: 实际有漏洞, 但工具没有检测出漏洞. 通过公式 (1)–(3), 可以计算出精确率、召回率和 *F*1 分数.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F1 = \frac{2 \times (Precision \times Recall)}{Precision + Recall} \tag{3}$$

更高的精确率意味着更低的误报率, 这表明检测工具更准确. 召回率越高, 则漏报率越低, 说明检测工具更加全面. $F1$ 分数越高, 意味着漏报率和误报率越低, 说明检测工具准确且全面.

RQ3: 检测效率——检测工具的检测效率如何?

本问题关注工具在实际审计过程中的时间开销. 将选定的工具部署在相同的硬件环境下, 比较每个工具在相同数据集下的时间开销. 这个问题直接反映了检测工具的效率以及工程适用性. 具体来说, 本文使用重入漏洞数据集来比较检测工具的检测时间.

RQ4: 复杂度敏感性——在不同复杂度的合约下, 工具的表现有何差异?

为评估各工具在面对不同复杂度的智能合约时的适应能力, 数据集中的合约按照复杂度将被分成不同的类别, 并在各子集上分别测试各工具的检测准确性. 该问题旨在揭示工具在处理高复杂度合约时可能面临的性能瓶颈或误报/漏报倾向.

RQ5: 工具组合效应——不同检测工具的组合效果如何?

考虑到单一工具可能存在检测盲区, 本文进一步对候选工具的输出结果进行组合分析, 探讨多工具融合是否能提升整体检测性能. 具体而言, 对不同工具的检测结果进行并集/交集运算, 并分析其对精确率、召回率及 $F1$ 分数的影响. 该问题有助于为未来多策略融合型漏洞检测工具的设计提供实证依据与参考思路.

3.3 数据收集

数据集是客观评价智能合约漏洞检测工具的关键因素之一. 然而, 由于当前缺乏统一的标准基准, 不同工具往往使用自定义数据集进行评测, 这导致评估结果存在可比性差、覆盖面不足等问题. 尤其, 多数工具评估时忽视了无漏洞合约的引入, 从而无法有效衡量其误报能力. 因此, 全面、结构化的数据集构建对于公正、真实地评价检测工具至关重要.

为了解决上述问题, 本文构建了一个基准数据集, 该数据集由 3 类子集组成.

● CVE 数据集[32]

(1) 数据来源及筛选: 以“smart contract”为关键词检索 CVE 官方数据库 (截至 2024 年 9 月), 排除与以太坊智能合约无关的条目, 获得 413 个智能合约漏洞相关的条目. 经过去重之后, 最终得到 43 份算术漏洞合约, 2 份重入漏洞, 2 份时间戳依赖漏洞合约, 3 份访问控制漏洞合约.

(2) 数据特征: 这些样本包含经官方披露的已知漏洞的真实攻击案例, 覆盖整数溢出、重入攻击、访问控制缺陷等常见漏洞类型. 由于其来源权威, 能够作为真实攻击样例.

● 漏洞检测工具实验数据集

(1) 数据来源及筛选: 收集近 5 年 (2019–2024 年) 发表在会议或期刊上的智能合约漏洞检测相关论文所公开的开源数据集, 仅保留提供完整源码、可编译且经过人工复现确认的漏洞合约和人工验证无漏洞的合约. 最终, 获得 486 份无漏洞合约, 17 份访问控制漏洞合约, 453 份算术漏洞合约, 99 份重入漏洞合约, 162 份时间戳依赖合约, 109 份未检查的低级调用合约.

(2) 数据特征: 这些样本已通过多种工具检测并由研究者或人工再次验证, 能够作为人工确认漏洞样本和无漏洞样本.

● 安全公司审计的数据集[33,34]

(1) 数据来源及筛选: 从 SlowMist 和 Etherscan 获得了人工审计的智能合约项目子集, 主要为已部署在以太坊主网上的智能合约项目, 且其审计项目覆盖了表 2 所示的常见漏洞类型, 最终获得 560 份合约文件.

(2) 数据特征: 由于这些合约在审计中未检测到已知漏洞, 可作为干净样本, 用于评估工具的误报率.

通过整合以上 3 类子集, 本文构建了一个包含真实漏洞、人工验证漏洞与无漏洞样本的多样化数据集. 该数据集包含 7 类合约数据, 分别是审计合约 (560 份)、无漏洞合约 (486 份)、访问控制漏洞 (20 份)、算术漏洞 (496 份)、重入漏洞 (101 份)、时间戳依赖漏洞 (164 份) 和未检查的低级调用 (109 份), 其分布情况如图 1 所示. 这一数据集不仅提升了评估工具准确性与覆盖率的客观性, 还为误报、漏报与检测效率等多个维度的分析提供了坚实基础, 从而更真实、全面地反映智能合约漏洞检测工具的实际表现.

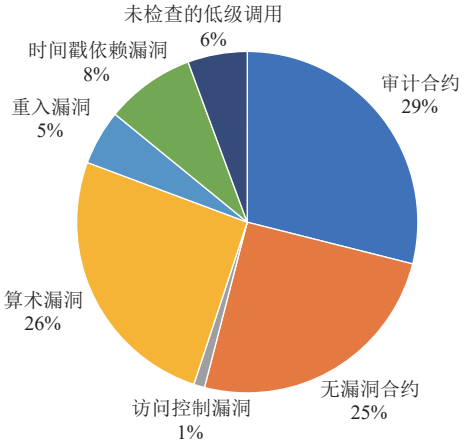

图 1　数据集分布情况

3.4　数据分类

早在 2018 年, Hegedüs[35]就开始研究智能合约的评估指标. 他将传统面向对象语言中的代码评估指标迁移应用于智能合约中, 并根据智能合约的特点补充了新的特征指标, 提出了面向智能合约的评估指标生成工具 SolMet. 所有的代码特征指标如表 3 所示.

表 3　智能合约的评估指标

名称	描述	名称	描述
SLOC	源代码行数	DIT	继承树深度
LLOC	逻辑代码行数	NOA	祖先合约数量
CLOC	源代码中注释的行数	CBO	对象之间的耦合
NF	函数数量	NL	合约中所有函数的最深嵌套层数之和
NA	属性数量	Avg.McCC	平均McCabe循环复杂度 (WMC/NF)
NOD	后继合约数量	Avg.NL	平均循环数量 (NL/NF)
NOI	传出调用数量	Avg.NLE	平均循环表达式数量 (NLE/NF)
WMC	所有函数的McCabe风格复杂度加权和	Avg.NUMPAR	平均参数数量 (NUMPAR/NF)
NLE	else-if的嵌套层数	Avg.NOS	平均语句数量 (NOS/NF)
NUMPAR	参数数量	Avg.N	平均接口数量 (NOI/NF)
NOS	陈述数量	—	—

尽管上述评估指标在分析单一智能合约时具有一定的适用性, 但是当一个 Solidity 文件 (sol 文件) 包含多个合约且合约之间存在复杂的继承与调用关系时, 这些指标已难以全面刻画合约的结构与行为特征. 在现实世界的智能合约开发实践中, 合约间调用及对外部库的依赖已十分常见. 例如, 如图 2 所示, MetaToken[36]合约继承自 4 个合约: ERC20、GovernedMinterRole、ERC20Detailed 和 ERC20Burnable. 这 4 个合约继承自其他合约并且也会调用第三方库, 例如 SafeMath.

当采用上述评估指标对 MetaToken 进行评估时, 主合约 MetaToken 只有一个函数, 有效代码行数为 25 行. 然而, 在将 MetaToken 部署在私有链上并进行功能验证后, 实际包含的功能数量达 11 项. 该现象表明现有的评估指标不能反映 MetaToken 的复杂性. 此外, 当尝试将所有与 MetaToken 相关的合约指标相加以表征 MetaToken 合约时, 结果也是不准确的. 其原因可能是合约的一些功能被覆盖或通过第三方库间接实现. 这一现象表明, 使用传统的代码度量不能真实地反映具有调用或继承关系的合约特征.

更有效的方式是从合约继承图中学习特征表示, 即在保持合约间继承依赖关系的图拓扑结构的同时, 建模节点之间的潜在联系.

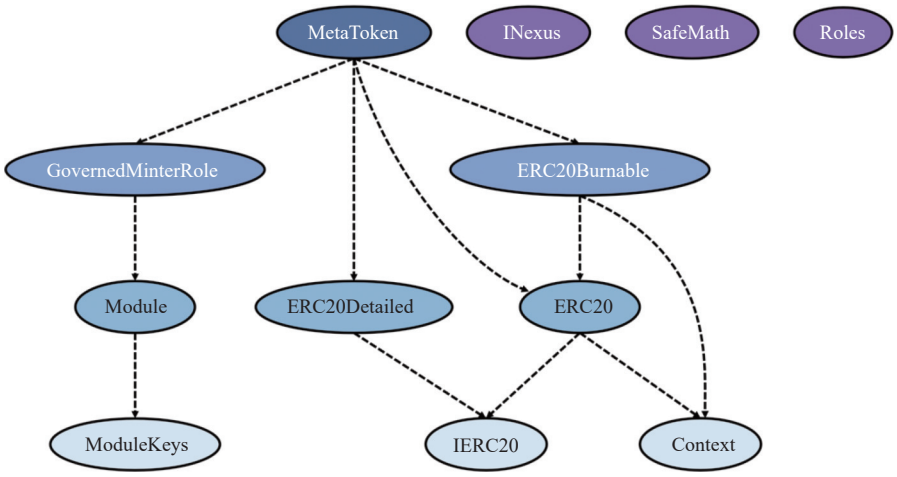

图 2　MetaToken 合约的继承图

为了更准确表征合约复杂程度, 在充分考虑合约本身特征以及合约继承关系的情况下, 本文设计了如图 3 所示的合约特征构建方法. 该方法主要由两部分组成: 第 1 部分是通过分析合约继承图来获取图特征, 第 2 部分是收集合约的代码度量指标.

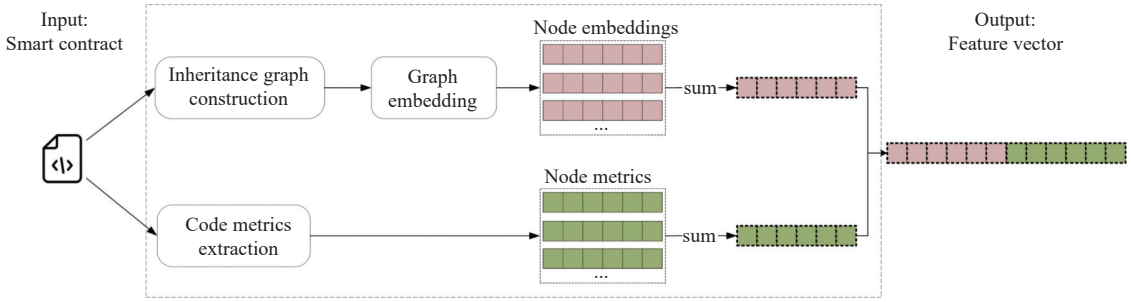

图 3　合约特征构建方法

设一个 sol 文件包含若干智能合约, 构成集合:

$$C = \{C_1, C_2, \ldots, C_n\} \tag{4}$$

每个合约 C_i 由若干函数和属性组成, 其代码特征可表示为向量:

$$X(C_i) = [x_1, x_2, \ldots, x_m] \in \mathbb{R}^m \tag{5}$$

其中, x_j 为合约代码特征指标 (如源代码行数 SLOC、逻辑代码行数 LLOC、McCabe 圈复杂性 WMC、继承深度 DIT 等, 具体见表 3). 这些指标能够在单一合约层面刻画其基本复杂性, 但在一个 sol 文件中包含多个合约且存在继承与调用关系时, 这些指标难以全面反映合约结构和行为特征.

为建模合约间的继承关系和依赖, 定义合约继承图:

$$G = (V, E) \tag{6}$$

其中, 节点集合 $V = C$ 表示文件中的所有合约. 边集合 $E \subseteq V \times V$ 表示继承关系, 即若合约 C_i 继承自 C_j, 则 $(C_i, C_j) \in E$. 图 2 即为 MetaToken 的合约继承图.

图嵌入技术作为一种有效的图结构建模工具, 已被广泛运用于节点分类、链路预测、聚类和可视化[37]等任务. 它可以将图数据等高维稠密矩阵映射到低维向量空间以便后续模型处理. 为捕捉合约间的结构信息, 引入图嵌入函数:

$$f_{\text{embed}}: G \to \mathbb{R}^{d\times|V|} \tag{7}$$

其中, d 为嵌入维度, 每个节点 C_i 得到嵌入向量:

$$Y(C_i) = f_{\text{embed}}(C_i) \in \mathbb{R}^d \tag{8}$$

本文采用 struc2vec 方法生成嵌入向量, struc2vec 是一种经典的图嵌入技术[38], 它致力于表征网络结构中节点的相似性. 该方法首先构建层次图, 通过定义节点之间的相似度图来表征边的权重, 然后基于权重进行随机游走, 最后使用 Skip-gram 模型实现节点向量化. 该嵌入向量能够编码合约在继承图中的结构信息, 补充单一代码度量的不足.

直接将两个向量拼接在一起, 从而得到每个合约的特征向量:

$$Z(C_i) = X(C_i) \oplus Y(C_i) \tag{9}$$

其中, $\oplus$ 表示向量拼接, 对于一个 sol 文件而言, 定义文件级特征向量 $Z(sol)$ 为:

$$Z(sol) = \sum_{C_i \in C} Z(C_i) \tag{10}$$

这种特征构建方法既保留了合约自身的度量信息, 又保留了继承图结构信息, 从而实现对复杂合约特征的全面表征. 基于所有 sol 文件的特征向量集合 $\{Z(sol_j)\}_{j=1}^{N}$, 采用 K-means 聚类算法[39]进行分类:

$$\min_{\{S_k\}_{k=1}^{K}} \sum_{k=1}^{K} \sum_{Z(sol_j)\in S_k} \left\| Z(sol_j) - \mu_k \right\|^2 \tag{11}$$

其中, k 为簇数, S_k 为第 k 个簇, μ_k 为第 k 个簇的质心. 聚类结果划分出若干个合约簇, 每个簇代表一类在结构和复杂度层面具有相似特征的智能合约.

在具体的实现过程中, 首先, 使用 surya[40]工具提取合约的继承图并利用 SolMet 工具获取每一个节点的代码评估指标. 然后, 使用 struc2vec 对继承图进行嵌入计算得到嵌入特征并对每个节点的特征向量求和得到整个特征向量. 接下来, 将这两个特征直接拼接为整个合约的特征向量. 最后, 基于得到的特征向量, 使用 K-means 算法对数据集中的所有合约文件进行聚类从而实现分类.

3.5 基础环境

实验的硬件环境采用 Intel(R) Xeon (R) GOLD 6234 处理器, 128 GB 内存的服务器.

4 实验分析

4.1 RQ1: 工具易用性

该研究问题的核心是评估智能合约检测工具的用户友好性. 为此, 依据每个工具在其 GitHub 项目提供的安装步骤, 将其分别部署于统一服务器环境, 并对工具的安装方法进行整理. 此外, 还对每个工具在 GitHub 中提出的问题 (issues) 进行跟踪分析. 若某一工具的最近一次用户提问在超过 6 个月的时间内未获得开发者回应, 则将该工具视为已停止维护. 整理结果如表 4 所示, 其中 Docker、Pip 和 SC 分别表示工具支持 Docker 安装、Pip 安装和从源码安装.

一方面, 由表 4 可见, 大多数智能合约检测工具均支持通过 Docker 方式进行部署. 这降低了合约开发者的部署门槛, 开发人员可以通过简单地部署 Docker 容器来审计合约. Oyente、Mythril、SmartCheck 和 Slither 都支持从 Pip 直接安装, 这提高了工具的可移植性. 通过这种方式, 开发人员可以快速地将审计工具嵌套到其他检测程序中. 在所有的工具中, 只有 Oyente 和 sFuzz 有 GUI 界面, 所以它们的检测报告是最直观的. GUI 界面能够降低检测结果的解读成本.

另一方面, 本研究结果揭示出一个值得深入讨论的问题: 多数智能合约检测工具已长期缺乏维护, 导致其不支持对新版本合约的检测. 这种维护不足不仅是技术问题, 还反映出学术研究中工具可持续开发模式所面临的挑战.

Solidity 语言的快速发展在一定程度上加剧了该问题. 自 0.4.x 至今, Solidity 经历了多轮语法和语义更新, 包括状态变量可见性规则的修改、EVM 字节码结构调整以及安全关键字 (如 immutable 与 receive) 的引入. 检测工具往往依赖静态语法解析与字节码匹配, 这意味着语言的微小变动都可能导致分析逻辑失效或误判. 例如, 当工具未及时更新抽象语法树 (AST) 解析模块时, 可能无法识别新语法节点, 导致潜在漏洞未被发现 (漏报) 或错误提示 (误报). 这使得语言更新越快, 工具维护成本越高, 进而导致更多工具被放弃更新.

表 4　工具的安装方法和维护情况

工具	Docker	Pip	SC	更新
Oyente	√	√	√	×
Osiris	√	×	√	×
Securify	√	×	√	×
Vandal	×	×	√	×
Mythril	√	√	√	√
SmartCheck	√	√	√	×
Slither	√	√	√	√
sFuzz	×	×	√	×
ILF	√	×	√	×

在分析几个停止维护更新的工具之后, 其维护中断可能原因主要有以下两方面: 其一, 缺乏持续的支持与保障. 多数工具是用于支撑阶段性科研任务, 其生命周期通常与项目进程或论文发表周期一致, 缺乏继续投入的动力和资源; 其二, 协作机制的缺乏. 工具间接口不统一, 缺乏通用的中间表示与标准测试集, 使得维护者需要针对语言的每次更新进行独立适配, 维护负担沉重. 此外, 部分工具未采用模块化设计, 使得局部代码更新可能导致整体代码重构, 进一步限制了社区参与后续维护的可行性.

相比之下, Mythril 与 Slither 是少数仍在持续更新的工具, 二者的持续更新主要得益于以下几个因素.

(1) 产业支持与价值驱动: Mythril 由 ConsenSys Diligence 团队维护, 服务于实际的区块链安全审计业务, 其更新具有直接的经济激励与用户需求驱动. Slither 则是由 Trail of Bits 团队开发并持续维护的.

(2) 模块化与可扩展架构: Slither 基于静态分析框架构建, 核心组件 (如 AST 解析、数据流分析、检测插件) 相互解耦, 使其能快速适配 Solidity 版本更新而无需整体重构.

(3) 开源社区与生态协作: 二者均拥有较为活跃的 GitHub 社区, 定期合并外部贡献. 此外, Slither 还提供了与 GitHub Actions 集成的功能, 允许用户在 CI/CD 流程中自动运行静态分析, 以提高开发效率和代码质量. 这种社区驱动的维护模式有效缓解了单一开发者的维护压力.

基于上述分析, 未来研究可从两方面提升工具可持续性: 通过模块化和插件化设计, 将语言解析层与检测逻辑解耦, 降低版本更新适配成本; 借鉴 Mythril 和 Slither 的经验, 依托开源社区协作或产学合作机制, 建立长期维护和保障体系.

综上所述, 工具版本依赖性与维护不足不仅影响当前检测效果, 也揭示了学术软件在可持续性方面所面临的挑战. 研究应在算法创新之外, 更关注工具生命周期管理与可复用性机制建设, 以促进智能合约安全分析领域的长期发展. 从合约开发者角度来看, 在选择审计工具时, 应优先考虑仍在活跃维护且具备良好安装支持 (如 Docker 或 Pip) 的工具, 以保证对新版本智能合约的适配能力.

发现 1: 大多数智能合约检测工具安装便捷, 但很少提供 GUI 界面, 影响用户体验. 同时, 多数工具已停止维护, 可能无法适配新版本合约, 存在漏报或误报风险.

4.2 RQ2: 检测准确性

针对本研究问题, 本文构建了一个全新的数据集来全面客观地评估检测工具的检测效果. 该数据集包含 7 类合约数据, 其中 5 类为有漏洞合约, 剩下的两种是审计合约 (560 份) 和无漏洞合约 (486 份). 这 5 种漏洞类型分别是访问控制漏洞 (20 份)、算术漏洞 (496 份)、重入漏洞 (101 份)、时间戳依赖漏洞 (164 份) 和未检查的低级调用 (109 份). 这些漏洞数据包括 CVE 的真实漏洞合约和一些人工审核得到的漏洞合约.

与既有研究相比, 以往工作中很少提到合约版本问题. 事实上, 智能合约对编译版本很敏感, 编译使用的 solc 版本不一致会导致工具无法正常运行. 另外部分合约版本书写不规范, 导致这些合约在解析和编译时会出现错误. 针对这个问题, 本文手动分析了实验数据集并提取了新的合约版本解析模式. 基于这种模式, 实现了对所有合约数据的编译, 以消除因编译版本错误而无法解析的问题. 本实验主要聚焦于可成功编译并具备检测价值的漏洞合约样本, 以确保检测工具在有效输入条件下的性能评估具有参考性与可比性.

如表 5 所示, 部分检测工具在处理数据集时因编译错误而无法正常解析合约, Oyente 是其中的典型代表. 在测试所有数据集时, Oyente 有 25.01% 的合约解析失败, 最常见的原因是编译错误. 深入分析这个异常发现, 如果编译器版本正确且合约可以通过 solc 指令进行正常编译, 但该工具仍然可能会提示编译错误. 这个问题可能是检测工具本身嵌套了 solc 造成的. Osiris 是基于 Oyente 迭代开发的, 因此也存在类似的问题. 基于此, 实验中将字节码作为 Osiris 的输入. 相比之下, Vandal、SmartCheck 和 Slither 主要基于静态分析方法实现, 因而在处理过程中不涉及编译环节, 具备较高的合约兼容性.

表 5　每个工具的故障分析汇总

类别	Oyente	Osiris*	Securify	Vandal	Mythril	SmartCheck	Slither	sFuzz	ILF
访问控制漏洞 (20)	—	—	0	—	—	0	0	—	—
算术漏洞 (496)	0	0	—	—	1	0	0	7	—
重入漏洞 (101)	48	0	0	0	0	0	0	6	56
时间戳依赖漏洞 (164)	33	3	—	—	1	—	0	8	96
未检查的低级调用 (109)	—	—	—	0	0	0	0	7	54
无漏洞 (486)	10	20	33	0	3	0	0	21	266
审计合约 (560)	361	1	169	0	5	0	0	56	455
总数	452/1 807	24/1 807	202/1 167	0/1 256	10/1 916	0/1 772	0/1 936	105/1 916	927/1 420
占比 (%)	25.01	1.32	17.31	0	0.52	0	0	5.48	65.28

注: *表示该工具以字节码作为输入; “—”表示该工具不支持或未测试该类漏洞; 总数行中“a/b”表示该工具分析失败的合约数量/该工具实际参与测试的合约总数, 占比为 a/b×100%

在两种模糊测试工具中, sFuzz 比 ILF 具有更低的故障率. 理论上 sFuzz 不支持 0.5 及以上版本, 因为用于攻击的合约版本仅限于 0.4 版本. 为了探究其检测性能, 对攻击者合约进行了修改以适应 0.5 以上的编译环境. 由于许多合约在部署前需要初始化或传入构造参数, 对于这类合约 ILF 提供的示例无法对这类合约进行部署. 在表 5 的审计合约数据集中, ILF 分析失败 455 次, 占该数据集总量的 81.25%.

从表 6 中可以看到, Slither 是唯一支持所有 5 种漏洞检测的工具, 它检测到的漏洞占其正常分析合约的 77.52%. Oyente 检测到的漏洞占比 79.55%, 但其检测到的漏洞类型和数量都少于 Slither. 从漏洞检测比例来看, SmartCheck 检测到的漏洞比例相对最低. 这意味着 SmartCheck 的假阴性率比较高.

表 6 中的精确率、召回率和 $F1$ 值仅基于各工具成功完成分析的样本计算; 未成功分析的样本主要由于合约源码不完整、无法编译或字节码反编译失败等原因导致, 其运行情况已在表 5 中单独统计. 基于表 5 与表 6 中的实验数据, 计算了每个工具在其能够检测到的漏洞数据集上的精确率、召回率和 $F1$ 分数, 如图 4 所示. 在重入漏洞的实验中, Slither 保持了较高的召回率, 但其精确率较低. 基于此, Slither 工具的假阴性率较低但假阳性率极高, 这与 Slither 工具的重入检测粒度较粗有关. Oyente 在重入实验中表现更好, 同时保持了较高的精确率和召回率. 在算术漏洞实验中, 可以看到 Osiris 的精确率优于 Oyente, 这意味着 Osiris 增加的污点分析模块可以大大提高工具在算术漏洞检测中的精度. 在时间戳依赖漏洞检测中, Mythril 的召回率较低, 表明该工具具有较高的假阴性. 这种不足源于 Mythril 的检测机制, 它会生成相应的交易来触发漏洞, 如果无法触发就会导致漏洞合约的漏报. 在未检查的低级别调用中, Slither 的精确率更高. 这类漏洞更易通过静态程序分析精准定位, 因此多数工具在该任务中均表现良好, Vandal 是其中表现相对较差的个例. 对于访问控制漏洞, 很少有工具支持它们的检测, 表现不佳. ILF 作为模糊测试中的代表性工具, 在大多数情况下都能保持较高的检测精度. 但在重入漏洞检测中, 它和 sFuzz 取得

了最低的 *F*1 分数, 这与触发重入漏洞的难度有关. 在真实合约场景中, 往往需特定条件或预设状态才能成功触发重入路径, 说明现有测试用例生成策略尚需进一步优化与增强.

表 6　每个工具识别的漏洞

类别	Oyente	Osiris*	Securify	Vandal	Mythril	SmartCheck	Slither	sFuzz	ILF
访问控制漏洞 (20)	—	—	13/20	—	—	19/20	2/20	—	—
算术漏洞 (496)	434/496	293/496	—	—	148/495	14/496	389/496	225/489	—
重入漏洞 (101)	45/53	30/101	35/101	45/101	35/101	94/101	100/101	19/95	8/45
时间戳依赖漏洞 (164)	62/131	69/161	—	—	23/163	—	154/164	103/156	38/68
未检查的低级调用 (109)	—	—	—	74/109	54/109	54/109	45/109	33/102	11/55
总数	541/680	392/758	48/121	119/210	260/868	181/726	690/890	380/842	57/168
占比 (%)	79.55	51.72	39.67	56.67	29.95	24.93	77.52	45.13	33.93

注: *表示该工具以字节码作为输入; “—”表示该工具不支持或未测试该类漏洞; 表中“a/b”表示该工具成功识别出的漏洞合约数量/该工具成功完成分析的该类漏洞合约数量; 总数行中的“a/b”为各漏洞类型结果汇总, 占比为 a/b×100%

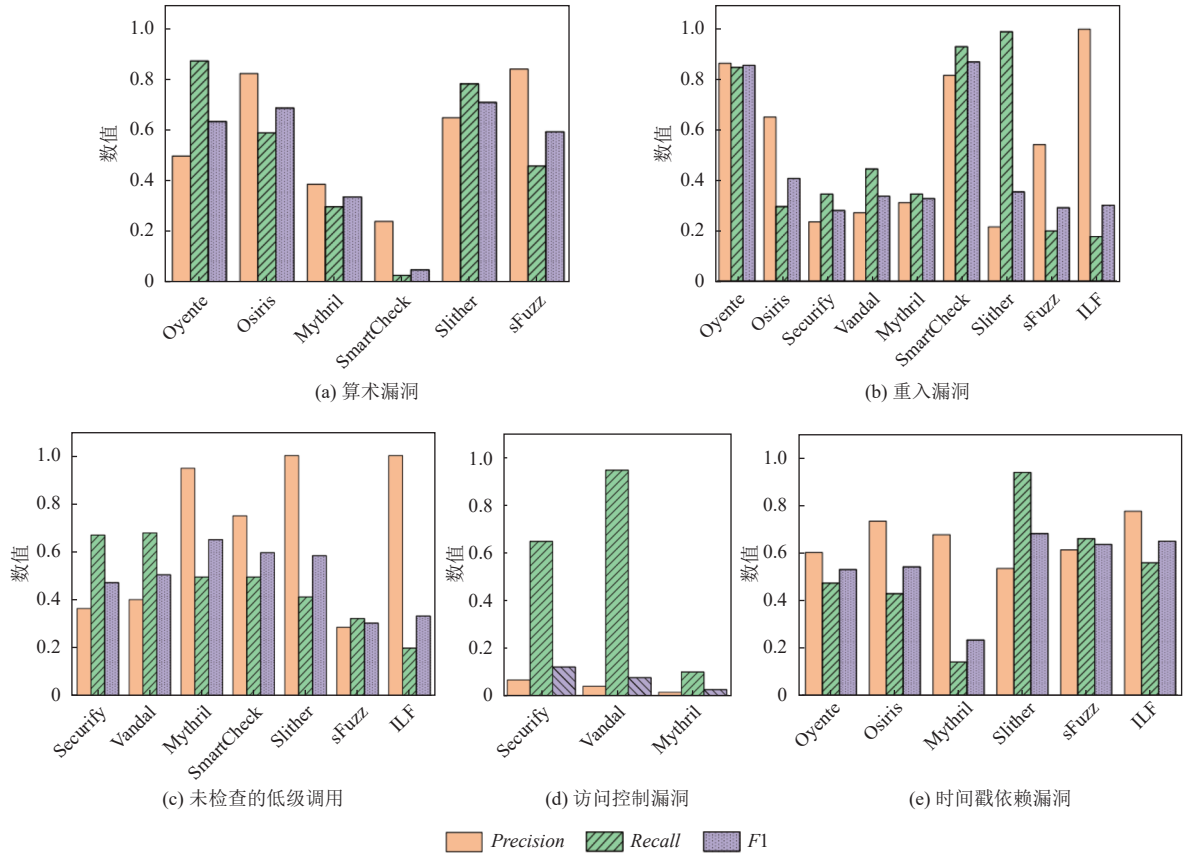

图 4　每个工具的精确率、召回率和 *F*1 分数

发现 2: 一些工具内置 solc 编译器, 这使得工具在实际检测过程中很容易因为合约编译失败而检测失败. 此外在面对具体漏洞时, 如访问控制漏洞, 现有工具无法有效检测这些漏洞或不支持对这些漏洞的检测.

4.3　RQ3: 检测效率

在本研究问题中, 基于动态检测技术的工具的检测时间是可以人工控制的, 所以模糊测试的执行效率不在此讨论. 为了比较其他检测工具的效率, 选择特定的漏洞模式和非漏洞数据来比较检测效率工具. 由于所有工具都可

以检测重入漏洞, 因此将重入漏洞作为待检测的漏洞类型, 并且只记录所有检测工具的执行时间, 不包括结果解析时间. 为确保比较的公平性, 将所有工具的实验参数设置为默认参数, 不添加额外的执行时间限制.

表 7 显示了所有工具的检测时间. 在 101 个重入漏洞合约的检测中, Slither 的检测效率最高而 Securify 的检测效率最低. 在 Slither 的检测过程中平均每个合约文件只需要 0.89 s, 完成整个数据集测试只需要 90 s. 相较之下, Securify 的检测时间较长, 平均每个合约需 267.32 s, 处理全部数据集约需 7.5 h, 为所有评估工具中耗时最长者.

表 7　每个工具的执行时间和合约吞吐量

工具	执行时间 (s)		合约吞吐量 (合约/s)
	平均时间	总时间	
Oyente*	2.3142	233.7400	0.43
Osiris*	14.9802	1513.0082	0.067
Securify	267.3181	26999.120	0.0037
Vandal	0.9884	99.8371	1.01
Mythril	42.0799	4250.0705	0.024
SmartCheck	2.6817	270.8556	0.37
Slither	0.8985	90.7518	1.11

注: *表示该工具以字节码作为输入

从检测技术的角度进行分析可发现, 采用静态分析方法的工具在检测效率方面优于其他技术. 例如, Slither、SmartCheck 和 Vandal 均表现出较高的检测速度. 这一因素主要归功于静态分析在进行合约检测的过程中不需要执行合约, 从而节省了大量的时间开销.

在以符号执行为主要检测技术的 4 种工具中, Securify 的时间开销最大, 其次是 Mythril. 根据工具日志分析可见, 这两种工具在检测过程中都存在大量的超时分析. 造成这种现象的主要原因在于, 符号执行的路径探索深度越大, 约束求解空间越复杂, 越容易触发状态爆炸. 若深度设定过小, 又可能导致分析过程无法覆盖实际的合约执行路径. 因此在实际的检测工作中, 需要根据实际情况对该参数进行调整, 在保证检测结果准确性的前提下, 尽可能提高检测效率.

此外, 大多数工具直接将源代码作为输入, 而 Oyente 和 Osiris 在本研究问题中使用字节码作为输入. 因此相比之下, Oyente 和 Osiris 将比其他工具更多地减少部分合约编译的时间开销, 从而略微提高其效率. Osiris 的平均执行时间比 Oyente 工具高 12.67 s. Osiris 在 Oyente 的基础上进行了优化, 增加了一个污点分析模块. 在相同的数据集下, 该技术显著增加了 Osiris 的时间开销.

在实际部署场景中, 评估工具的检测效率不仅依赖于单个合约的平均检测时间, 还需要考虑其在大规模任务下的处理能力. 为此, 本文引入合约吞吐量 (contracts per second, *CPS*) 作为衡量指标, 定义为在单位时间内工具可完成的合约检测数量. 具体计算方式如下:

$$CPS = \frac{N}{T} \tag{12}$$

其中, N 为检测合约总数, T 为总检测时间. 该指标借鉴自区块链和数据库等领域常用的事务吞吐量 (transactions per second, TPS) 概念, 用于提供直观的可扩展性比较. 如表 7 所示, 从实际部署角度来看, 静态分析类工具 (如 Slither、SmartCheck、Vandal) 在检测效率上表现突出. 例如, Slither 和 Vandal 的吞吐量分别为 1.11 与 1.01 合约/s, 意味着在 1 h 内可处理超过 3600 个合约, 完全能够满足大规模自动化审计和在线快速筛查的需求. 相比之下, 符号执行类工具 (如 Securify、Mythril) 虽能探索更深层次的执行路径, 具备更强的潜在漏洞检测能力, 但其吞吐量仅为 0.0037 合约/s (Securify) 和 0.024 合约/s (Mythril), 即每小时仅能处理约 13 个和 86 个合约, 时间开销超过静态分析工具数百倍, 因而难以直接应用于大规模场景. 对于资源受限或需要快速响应的应用环境, 这类工具更适合作为关键合约的深度审计手段, 而非通用检测工具.

从实际部署角度来看, 静态分析类工具 (如 Slither、SmartCheck、Vandal) 检测效率高, 适合大规模自动化审计和在线快速筛查; 而符号执行类工具 (如 Securify、Mythril) 虽然能探索更深层次的执行路径, 具备更强的潜在

漏洞检测能力, 但其极高的时间开销使其不适合实时或大规模应用. 对于资源受限或需要快速响应的场景, 这类工具难以直接部署, 更适合作为关键合约的深度审计方案.

对耗时较长的工具进行深入分析后, 未来在实际检测任务中, 可以考虑从以下几个方面进行优化: 设置合理的分析深度与超时时间, 在检测效率与覆盖率之间取得平衡; 在大规模任务中, 先用轻量级工具进行初筛, 再对高风险合约应用符号执行类工具做深度检测; 针对已审计过的合约, 仅对更新部分进行检测, 以减少重复计算开销.

综上所述, 工具的时间开销不仅反映了检测技术的差异, 也决定了其实际应用场景. 轻量工具可支持高效的批量检测, 而符号执行类工具则应定位于高价值目标的精细化审计.

发现 3: 使用符号执行等技术的检测工具在检测过程中普遍存在较大的时间开销. 在实际应用中, 需要根据精度与效率的平衡对工具参数进行合理调整, 以提升可用性. 此外, 部分工具通过直接处理字节码作为输入, 以提高整体检测效率.

4.4 RQ4: 复杂度敏感性

为评估不同漏洞检测工具在应对不同复杂度合约时的检测能力, 本研究问题选取了包含重入漏洞与无漏洞的合约样本, 旨在考察工具的检测鲁棒性.

在实验中, 首先提取要测试的合约继承图和代码度量指标, 然后采用 struc2vec 对合约继承图进行嵌入, 具体参数设置为: 嵌入维度为 128 维, 随机游走长度为 10, 每个节点的随机游走次数为 80, Skip-gram 窗口大小为 5. 此外, 针对每个合约文件使用 SolMet 提取代码度量指标从而得到 21 维向量. 将两部分向量拼接, 构建每份合约文件的最终特征向量, 共计 149 维. 这一特征向量既包含合约继承图的结构信息, 又包含代码度量特征, 能够全面表征合约复杂度.

对 587 份合约进行分析, 其中部分合约没有调用关系, 因此这部分合约无法生成继承关系图. 对合约进行筛选和过滤, 最终得到 284 个具有继承或调用关系的合约. 之后提取合约特征向量, 并使用特征向量对这些合约进行分类. 采用 K-means 对其进行聚类, 并使用肘部法则[41]确定最佳聚类类数为 3, 同时计算轮廓系数为 0.558 7, 表明聚类内部紧密、簇间分离较明显, 验证了将数据集划分为 3 类的合理性. 其中第 0 类有 173 份合约, 第 1 类有 16 份合约, 而第 2 类有 95 份合约. 对于这 3 类数据集, 分别在这 3 类数据集上计算每个工具的 $F1$ 分数, 如表 8 所示.

表 8　检测工具在不同类别上的 $F1$ 分数 (%)

类别	Oyente	Osiris*	Securify	Vandal	Mythril	SmartCheck	Slither	sFuzz	ILF
0 (173)	55.55	60.00	96.38	14.63	19.35	36.11	92.30	94.80	96.07
1 (16)	—	—	—	—	—	—	—	—	—
2 (95)	19.34	26.30	88.24	24.99	5.55	47.70	92.30	84.15	84.15

注: *表示该工具以字节码作为输入

从实验结果可以观察到, 第 1 类样本均为有漏洞合约, 不包含负样本, 无法完整计算涉及 FP 的精确率和 $F1$ 分数, 因此表 8 未报告该类结果. 在第 0 类和第 2 类合约中, 大多数工具在第 0 类数据集上的表现优于第 2 类数据集. 特别地, Securify 在第 0 类和第 2 类的检测精确率均为 1, 因为 Securify 只检测出一个漏洞并且没有产生误报. 为进一步探究这一现象的成因, 对这两类合约进行手动分析. 研究发现第 2 类的合约继承关系比较复杂, 这使得检测工具无法有效检测这类数据集. 更加复杂的继承与调用结构可能使得检测工具难以有效地解析其代码结构或构建完整的控制流图 (CFG). 此外, 复杂的路径约束也可能加剧状态空间爆炸问题, 增加符号执行与静态分析的难度.

基于上述结果, 现有检测工具在复杂调用关系的合约检测方面效果不佳. 这可能源于无法对编译后的复杂合约字节码进行反汇编或构建 CFG. 此外, 符号执行路径中存在复杂约束而无法有效求解, 这将导致状态空间爆炸. 现有的静态分析工具可能难以审计完整的以太坊合约项目. 该发现与审计合约数据集中呈现的趋势保持一致, 表明对具有复杂继承与调用关系的合约的有效检测仍是当前智能合约审计研究中亟待解决的重要问题, 亦为后续研究提供了有价值的方向指引.

发现 4: 现有的检测工具在面对具有复杂调用关系的合约时表现不佳. 未来, 如何有效处理此类合约中的复杂

调用关系, 可能是一个可行的研究方向.

4.5 RQ5: 工具组合效应

为进一步提升漏洞检测的整体效果, 本研究对不同检测工具的结果进行了组合分析, 旨在探讨多工具融合策略在智能合约漏洞检测中的可行性. 实验选取了重入漏洞数据集与无漏洞数据集作为测试基础, 选取了在重入漏洞检测任务中表现最佳的 3 种漏洞检测工具 (Oyente、Osiris、SmartCheck) 构成候选工具集合, 分别对任意两个工具的检测结果进行并集操作, 组合检测结果如表 9 所示, 其中 Oy-Os 表示 Oyente 和 Osiris 构成的组合, Oy-Sc 表示 Oyente 和 SmartCheck 构成的组合, Os-Sc 表示 Osiris 和 SmartCheck 构成的组合.

表 9 检测工具组合分析结果

指标	Oy-Os	Oy-Sc	Os-Sc	Oyente	Osiris	SmartCheck
合约总数	587	587	587	529	587	587
检出漏洞数	64	122	126	52	46	115
TP	45	99	98	45	30	94
FP	19	23	28	7	16	21
FN	56	2	3	8	71	7
精确率 (%)	70.31	81.15	77.78	86.53	65.21	81.73
召回率 (%)	44.55	98.02	97.03	84.90	29.70	93.06
*F*1分数 (%)	54.55	88.79	86.34	85.71	40.81	87.03

图 5 展示了 SmartCheck 与 Osiris 两种具备重入漏洞检测能力的工具的组合策略, 其中, T 表示对应工具将该合约判定为存在漏洞. 具体而言, 当两个工具中任意一个将合约判定为存在漏洞时, 组合结果即判定该合约存在漏洞, 本质上对应于两个工具检测结果的并集. 在本研究问题中, 将 3 种漏洞检测工具进行两两配对, 以构建 3 种组合方案, 并计算组合工具在检测任务中的精确率与召回率指标.

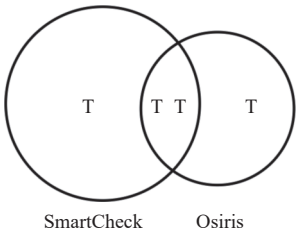

图 5 组合检测样例

实验结果如表 9 所示, 从 *F*1 数值上可以观察到 Oyente-SmartCheck (Oy-Sc) 组合的检测性能最优. 该组合在精确率和召回率上均表现突出. 原因可能在于 Oyente 基于符号执行擅长发现复杂逻辑漏洞, SmartCheck 依赖模式匹配对模板漏洞敏感. 该结果表明 Oyente 和 SmartCheck 可在漏洞检测任务上互补, 从而显著提升了整体检测性能. 相比之下, Oyente-Osiris (Oy-Os) 组合召回率较低, 这可能由于 Osiris 虽在 Oyente 基础上增加了污点分析, 增强了针对算术漏洞检测能力, 但对重入漏洞的检测没有显著提升, 因此该组合在重入漏洞检测上的互补性有限.

此外, 本研究问题对 3 种智能合约漏洞检测工具 (Oyente、Osiris、SmartCheck) 进行了投票组合实验, 以评估三工具联合检测的效果. 具体方法如下: 首先, 对每份合约分别使用 3 个工具进行独立检测, 记录每个工具对重入漏洞的判定结果, 即存在漏洞或不存在漏洞. 然后, 按照投票组合原则对检测结果进行整合, 设计了 3 种投票策略: 1-工具投票: 任意一个工具判定存在漏洞, 即认为该合约存在漏洞; 2-工具投票: 至少两个工具判定存在漏洞, 才认为该合约存在漏洞; 3-工具投票: 只有 3 个工具都判定存在漏洞, 才认为该合约存在漏洞. 因 Oyente 只能正常检测其中 529 个合约文件, 58 个合约文件检测失败, 所以在联合检测实验中, 考虑将三者能检测的最小公共子集作为实验的数据集, 即 529 个合约文件, 其中有漏洞的文件为 53 份, 无漏洞的文件为 476 份.

实验结果如表 10 所示, 采用投票组合原则时, 检测结果随着投票门槛的变化呈现明显趋势. 1-工具投票能够

捕获几乎所有真实漏洞, 召回率最高, 但误报较多, 精确率较低, *F*1 分数为 77.04%; 2-工具投票在平衡精确率与召回率方面表现最佳, 精确率提升至 81.48%, 召回率为 83.02%, *F*1 分数为 82.24%. 这说明中等投票门槛能够有效过滤部分误报, 同时保持较高的漏洞检测能力. 3-工具投票最严格, 仅捕获多数真实漏洞, 召回率下降至 49.06%, 但精确率最高, *F*1 分数下降至 62.65%. 说明严格投票策略适合对误报敏感的场景, 但会增加漏报风险.

表 10　3 种工具联合检测结果

指标	1-工具投票	2-工具投票	3-工具投票
合约总数	529	529	529
检出漏洞数	82	54	30
TP	52	44	26
FP	30	10	4
FN	1	9	27
精确率 (%)	63.41	81.48	86.67
召回率 (%)	98.11	83.02	49.06
*F*1分数 (%)	77.04	82.24	62.65

该结果说明, 在实际使用过程中, 可根据具体的安全分析需求灵活选择投票策略: 在对误报敏感的场景 (例如人工审计或安全认证) 中, 可采用较高的投票门槛 (如 2-或 3-工具投票), 以减少误报并提高结果的可靠性; 而在对漏报敏感的场景 (如自动化安全监控或大规模漏洞筛查) 中, 则宜采用较低的门槛 (如 1-工具投票), 以尽可能捕获潜在漏洞、降低漏报风险.

综上所述, 实验结果充分验证了不同检测技术的协同优势. 未来在智能合约安全工具的设计与开发中, 可进一步探索多种检测技术 (如符号执行、静态分析与模式匹配等) 的深度融合, 以构建更加全面、高效且鲁棒的漏洞检测框架, 为复杂区块链应用场景下的智能合约安全保障提供有力支持.

发现 5: 两两组合或多工具投票策略能够显著提升漏洞检测性能, 不同工具间具有互补优势可提高召回率和 *F*1 分数. 在多工具联合检测中, 采用中等投票门槛在平衡精确率与召回率方面表现最佳.

4.6 案例分析

在本节中, 为验证输入类型对漏洞检测效果的影响, 本研究分别使用合约字节码与源代码作为输入, 对检测性能进行了对比分析. 一些漏洞检测工具在漏洞检测中提供了将合约源代码和字节码作为输入的选项. 以 Oyente 检测工具为例, 人工编译了重入漏洞数据集和审计合约数据集, 并使用 Oyente 对这两个数据集的字节码和源代码进行检测. 结果如表 11 所示, 其中覆盖率表示的是数据集中代码的平均覆盖率, P 表示检测失败所占百分比. 实验结果表明, 虽然 Oyente 对合约源代码的检测有一定的检测失败率, 但该检测方法的精确率、召回率和 *F*1 分数都高于使用字节码的检测方法. 这意味着对源代码进行检测可以取得更好的检测结果.

表 11　字节码和源代码执行结果比较 (%)

类别	P	覆盖率	精确率	召回率	*F*1分数
重入 (源代码)	47.52	92.68	86.54	84.91	85.71
重入 (字节码)	0	26.92	81.08	29.70	43.47

从代码覆盖率来看, Oyente 对重入漏洞源代码的平均覆盖率为 92.68%, 对字节码的平均覆盖率为 26.92%. 这意味着, 以字节码为输入来生成和解决有效路径约束将具有更大的挑战性. 同时, 在 Oyente 的两种检测方法下, 以源代码为输入的检测方法可以覆盖更多的漏洞类型, 检测结果也更加全面. 测试结果也会更具体、更能标明具体的漏洞位置, 方便开发人员快速定位. 进一步分析显示, 字节码形式在编译过程中可能丢失诸如语法结构和变量信息等关键内容, 导致工具在建模和数据流分析方面受限, 进而影响部分漏洞类型的检测. 而源代码保留了完整的语义信息, 使得工具可以更充分地利用程序结构与上下文信息, 从而实现更全面和精准的漏洞识别.

综上所述, 本研究建议在智能合约漏洞检测任务中优先采用源代码作为分析输入, 以获得更高的检测准确性.

5 讨 论

5.1 有效性威胁

● 外部有效性. 实验中所使用的智能合约样本可能在规模、复杂度及实际应用场景上与现实世界中的合约项目存在显著差异, 这构成了对外部有效性的潜在威胁. 为尽可能缓解该问题, 本研究从以太坊区块链和部分链下审计机构收集了经过人工审计的真实合约项目, 以增强数据集的真实性与代表性. 通过引入高质量的真实世界样本, 实验尽可能模拟实际审计环境, 从而提高工具性能评估的现实适用性.

● 内部有效性. 由于本研究涵盖多类漏洞及多种检测工具, 实验过程中存在操作复杂、出错可能性较高的风险. 例如, 部分工具即便使用正确的 solc 编译器版本, 仍可能无法成功编译合约. 为降低此类干扰因素对实验结论的影响, 本文针对每项实验均进行了多次重复测试, 排除因误操作导致的异常结果, 从而将问题更准确地归因于工具自身的稳定性和兼容性. 此外, 检测工具的运行时开销也可能受到机器资源状态 (如 CPU 占用率) 波动的影响, 进而造成轻微偏差. 为缓解该风险, 所有实验均执行 3 次并记录平均运行时间, 以增强实验结果的可靠性.

● 结构有效性. 实验中部分工具允许用户自定义检测参数, 不同参数设置可能会影响检测结果的准确性与效率. 为消除参数配置对评估公平性的干扰, 本研究统一采用各工具的默认参数进行测试. 此举旨在保障不同工具在相同实验条件下的可比性, 从而提高评估结论的结构有效性.

● 数据分类. 传统的智能合约评估指标在衡量合约结构复杂性方面存在一定的局限性, 尤其对于存在函数调用关系或继承关系的复杂合约, 其表达能力较弱. 为更准确评估合约的复杂度, 本文引入合约继承图作为辅助特征, 通过结构化图表示合约之间的继承和依赖关系. 在图嵌入阶段, 采用 struc2vec 对继承图进行特征提取, 因为 struc2vec 更注重网络结构. 该算法能够保留图中节点的结构相似性, 适用于捕捉不同合约在继承层次上的共性与差异. 未来, 将考虑采用其他图嵌入算法来提取合约特征, 以便更准确地衡量合约的复杂性. 在分类阶段, 采用 K-means 对提取的特征进行分类, 从而实现根据合约的复杂度进行分类. 未来, 将使用更多不同的聚类方法进行更详细的分类以获得每个工具更完整的鲁棒性结果, 实现对智能合约检测工具更准确、客观的评估.

6 启 示

基于上述实验与分析, 在统一的数据集条件下对多种主流智能合约检测工具进行了系统性评估. 结果揭示了各工具在检测能力、运行效率、鲁棒性等方面的优劣势. 通过对比分析, 可以为工具使用者和开发者提供以下参考信息.

6.1 给用户的启示

首先, 从安装方面, 建议优先选择安装和使用方便的工具, 例如 Mythril 和 Slither. 它们支持多种快捷安装方式, 例如 Pip. 特别是这些工具还提供接口函数, 因此可以轻松地嵌入到用户的项目中. 此外更重要的是, 这些工具仍然处于更新状态, 这意味着其有潜力检测新类型的漏洞.

其次, 从可检测的主流漏洞数量来看, Mythril、SmartCheck、Slither 和 sFuzz 均支持 4 种以上类型的漏洞检测; 优先使用这些工具以便用户可以对合约进行尽可能详细的安全检查. 在这些工具中, Slither 不仅支持检测主流漏洞, 还支持检测一些在合约设计阶段可能出现的风险操作, 如 transferFrom 函数中任意来源等.

第三, 从检测精度和效率方面来看, Oyente 和 Slither 都有较好的检测效果. 因为两者是静态检测工具, 它们的检测时间开销比其他工具要小. 因此选择这两种工具可以以更少的时间开销进行更准确的检测. 特别是结合多种检测技术可以进一步提高检测结果的准确性.

第四, 现有的检测工具在具有复杂调用或继承关系合约上表现不佳. 此外, 这些工具很少考虑跨合约安全问题. 这意味着当用户需要对复杂的合约项目进行安全性测试时, 除了使用这些漏洞检测工具外, 应辅以人工审计手段, 以弥补工具在路径分析和语义建模方面的局限, 从而提升整体安全检测的效果.

6.2 给研究人员的启示

首先, 从实验结果来看, 传统的智能合约检测技术仍有一定的提升空间. 大多数静态检测工具存在较高的误报率. 静态解决方案通过静态分析收集漏洞相关信息, 然后将收集到的信息与漏洞规则进行比较, 判断是否存在漏洞. 因此静态检测工具可以考虑结合漏洞的数据流信息或语义信息, 进一步提高解决方案的准确性. 动态检测方案通过构造和执行事务来收集事务执行信息. 最后将这些信息进行测试和预测. 其中基于模糊测试的工具难以生成有效的交易. 因此模糊测试方案可以专注于更优化的种子生成设计, 同时考虑构建长交易序列以探索更深层次的漏洞状态. 这可以进一步降低方案的漏报率.

其次, 事实上无论是静态检测方案还是动态检测方案, 都依赖于专家定义的规则. 在实际应用中这样的规则会限制工具的可扩展性. 一方面, 这将使其在应用于检测新类型漏洞时具有挑战性; 另一方面, 这样的规则依赖于人工知识很容易造成一定的漏报或误报. 因此智能合约漏洞检测可以考虑采用深度学习. 深度学习等技术可从漏洞样本中学习漏洞相关模式从而实现更高精度的检测. 同时还可以通过提供新的漏洞数据, 支持对新漏洞的检测. 当然这需要构建一个标准的漏洞数据集.

第三, 随着智能合约应用场景的增加, 智能合约能够实现的功能变得越来越复杂. 由于传统的检测技术只检测单一合约, 满足实际的审计需求逐渐变得困难. 由于复杂合约项目的搜索状态空间较大, 现有检测工具大多难以构建需要长交易序列才能触发的攻击. 因此如何检测具有复杂调用关系的多合约或合约项目将成为一个潜在的研究方向. 当检测对象从合约变为合约项目时, 传统的检测方案也需要进行相应的更新.

7 总 结

智能合约为区块链平台提供了一个繁荣、安全、可信的去中心化应用场景, 是区块链平台的重要组成部分. 一旦智能合约中的漏洞被利用, 不仅可能造成巨大的经济损失, 还可能动摇智能合约的公平基础. 智能合约虽然有各种各样的测试工具, 但由于缺乏统一的评价标准, 很难科学地评价其性能. 为了帮助后续研究者更理性地进行对比实验, 本文收集了包括 5 类漏洞和智能合约项目在内的共计 1 936 份合约并从工具易用性、检测有效性、检测效率、复杂度敏感性和工具组合效应这 5 个方面对经典的智能合约检测工具进行了详细的测试和评估. 实验过程中发现, 现有的工具大多已经停止了维护, 且大多难以支持对合约项目进行审计. 因此, 后续研究应重点提升工具对复杂合约项目和跨合约交互的建模与检测能力. 本文结果可为后续研究提供参考, 帮助研究者开发更安全的智能合约检测工具.

References

[1] Buterin V. A next-generation smart contract & decentralized application platform. 2014. https://ethereum.org/whitepaper/

[2] Szabo N. Formalizing and securing relationships on public networks. First Monday, 1997, 2(9). [doi: 10.5210/fm.v2i9.548]

[3] Zou WQ, Lo D, Kochhar PS, Le XBD, Xia X, Feng Y, Chen ZY, Xu BW. Smart contract development: Challenges and opportunities. IEEE Trans. on Software Engineering, 2021, 47(10): 2084–2106. [doi: 10.1109/TSE.2019.2942301]

[4] AI Incident Database. Incident 50: The DAO hack. 2016. https://incidentdatabase.ai/cite/50

[5] OpenZeppelin. The parity wallet hack explained. 2017. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[6] Jin H, Xiao J. Towards trustworthy blockchain systems in the era of “Internet of value”: Development, challenges, and future trends. Science China Information Sciences, 2022, 65(5): 153101. [doi: 10.1007/s11432-020-3183-0]

[7] 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. Montpellier: IEEE, 2018. 259–269. [doi: 10.1145/3238147.3238177]

[8] He JX, Balunović M, Ambroladze N, Tsankov P, Vechev M. Learning to fuzz from symbolic execution with application to smart contracts. In: Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security. London: ACM, 2019. 531–548. [doi: 10.1145/3319535.3363230]

[9] Nguyen TD, Pham LH, Sun J, Lin Y, Minh QT. sFuzz: An efficient adaptive fuzzer for Solidity smart contracts. In: Proc. of the 42nd ACM/IEEE Int’l Conf. on Software Engineering. Seoul: ACM, 2020. 778–788. [doi: 10.1145/3377811.3380334]

[10] Wüstholz V, Christakis M. Harvey: A greybox fuzzer for smart contracts. In: Proc. of the 28th ACM Joint Meeting on European Software

Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 1398–1409. [doi: 10.1145/3368089.3417064]

[11] 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]

[12] Torres CF, Schütte J, State R. Osiris: Hunting for integer bugs in Ethereum smart contracts. In: Proc. of the 34th Annual Computer Security Applications Conf. San Juan: ACM, 2018. 664–676. [doi: 10.1145/3274694.3274737]

[13] 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. ACM, 2020. 415–427. [doi: 10.1145/3395363.3397385]

[14] 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]

[15] Ren M, Yin ZJ, Ma FC, Xu ZY, Jiang Y, Sun CN, Li HZ, 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. ACM, 2021. 566–579. [doi: 10.1145/3460319.3464837]

[16] Atzei N, Bartoletti M, Cimoli T. A survey of attacks on Ethereum smart contracts (SoK). In: Proc. of the 6th Int'l Conf. on Principles of Security and Trust. Uppsala: Springer, 2017. 164–186. [doi: 10.1007/978-3-662-54455-6_8]

[17] Chen HS, Pendleton M, Njilla L, Xu SH. A survey on Ethereum systems security: Vulnerabilities, attacks, and defenses. ACM Computing Surveys (CSUR), 2021, 53(3): 67. [doi: 10.1145/3391195]

[18] 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]

[19] Yan KL, Diao WR, Guo SQ. A survey of smart contract vulnerabilities and detection techniques. Information Countermeasure Technology, 2023, 2(3): 1–17 (in Chinese with English abstract). [doi: 10.12399/j.issn.2097-163x.2023.03.001]

[20] Dong WL, Liu Z, Liu K, Li L, Ge CP, Huang ZQ. Survey on vulnerability detection technology of smart contracts. Ruan Jian Xue Bao/Journal of Software, 2024, 35(1): 38–62 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6810.htm [doi: 10.13328/j.cnki.jos.006810]

[21] Cui ZQ, Yang HW, Chen X, Wang LZ. Research progress of security vulnerability detection of smart contracts. Ruan Jian Xue Bao/Journal of Software, 2024, 35(5): 2235–2267 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7046.htm [doi: 10.13328/j.cnki.jos.007046]

[22] 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]

[23] Yang GW, Filieri A, Borges M, Clun D, Wen JY. Advances in symbolic execution. Advances in Computers, 2019, 113: 225–287. [doi: 10.1016/bs.adcom.2018.10.002]

[24] Mueller B. Smashing Ethereum smart contracts for fun and actual profit. 2018. https://consensys.io/diligence/files/D1T2%20-%20Bernhard%20Mueller%20-%20Smashing%20Ethereum%20Smart%20Contracts%20for%20Fun%20and%20ACTUAL%20Profit.pdf

[25] Murray Y, Anisi DA. Survey of formal verification methods for smart contracts on blockchain. In: Proc. of the 10th IFIP Int'l Conf. on New Technologies, Mobility and Security (NTMS). Canary Islands: IEEE, 2019. 1–6. [doi: 10.1109/NTMS.2019.8763832]

[26] Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: Analyzing safety of smart contracts. In: Proc. of the 2018 Network and Distributed Systems Security (NDSS) Symp. San Diego: NDSS, 2018. 1–12. [doi: 10.14722/ndss.2018.23082]

[27] 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]

[28] 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 (SP). San Francisco: IEEE, 2020. 1661–1677. [doi: 10.1109/SP40000.2020.00024]

[29] Brent L, Jurisevic A, Kong M, Liu E, Gauthier F, Gramoli V, Holz R, Scholz B. Vandal: A scalable security analysis framework for smart contracts. arXiv:1809.03981, 2018.

[30] 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. ACM, 2018. 9–16. [doi: 10.1145/3194113.3194115]

[31] 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). Montreal: IEEE, 2019. 8–15. [doi: 10.1109/WETSEB.2019.

00008]
[32] CVE. CVE™ program mission. https://cve.mitre.org/
[33] Slowmist. Slowmist—Focusing on blockchain ecosystem security (in Chinese). https://cn.slowmist.com/
[34] Etherscan. Ethereum verified contracts. 2025. https://etherscan.io/contractsVerified
[35] Hegedűs P. Towards analyzing the complexity landscape of Solidity based Ethereum smart contracts. In: Proc. of the 1st Int'l Workshop on Emerging Trends in Software Engineering for Blockchain. Gothenburg: ACM, 2018. 35–39. [doi: 10.1145/3194113.3194119]
[36] GitHub. SmartContractSe/dataset. 2025. https://github.com/SmartContractSe/dataset/blob/main/MetaToken.sol
[37] Cai HY, Zheng VW, Chang KCC. A comprehensive survey of graph embedding: Problems, techniques, and applications. IEEE Trans. on Knowledge and Data Engineering, 2018, 30(9): 1616–1637. [doi: 10.1109/TKDE.2018.2807452]
[38] Ribeiro LFR, Saverese PHP, Figueiredo DR. struc2vec: Learning node representations from structural identity. In: Proc. of the 23rd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. Halifax: ACM, 2017. 385–394. [doi: 10.1145/3097983.3098061]
[39] Sinaga KP, Yang MS. Unsupervised K-means clustering algorithm. IEEE Access, 2020, 8: 80716–80727. [doi: 10.1109/ACCESS.2020.2988796]
[40] GitHub. ConsenSysDiligence/surya. 2025. https://github.com/ConsenSysDiligence/surya
[41] Nainggolan R, Perangin-angin R, Simarmata E, Tarigan AF. Improved the performance of the K-means cluster using the sum of squared error (SSE) optimized by using the elbow method. Journal of Physics: Conf. Series, 2019, 1361(1): 012015. [doi: 10.1088/1742-6596/1361/1/012015]

附中文参考文献

[18] 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]
[19] 闫凯伦, 刁文瑞, 郭山清. 智能合约安全漏洞及检测技术综述. 信息对抗技术, 2023, 2(3): 1–17. [doi: 10.12399/j.issn.2097-163x.2023.03.001]
[20] 董伟良, 刘哲, 刘逵, 黎立, 葛春鹏, 黄志球. 智能合约漏洞检测技术综述. 软件学报, 2024, 35(1): 38–62. http://www.jos.org.cn/1000-9825/6810.htm [doi: 10.13328/j.cnki.jos.006810]
[21] 崔展齐, 杨慧文, 陈翔, 王林章. 智能合约安全漏洞检测研究进展. 软件学报, 2024, 35(5): 2235–2267. http://www.jos.org.cn/1000-9825/7046.htm [doi: 10.13328/j.cnki.jos.007046]
[22] 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]
[33] 慢雾科技. 专注区块链生态安全: 慢而有为, 雾释冰融. https://cn.slowmist.com/

作者简介

祝语, 博士生, 主要研究领域为区块链, 智能合约安全.
李珍, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为软件安全, 人工智能安全.
张笑睿, 博士生, 主要研究领域为智能合约安全, 协议安全.
吴月明, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为软件供应链安全, 移动安全, 人工智能安全.
邹德清, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为大数据安全与人工智能安全, 云计算安全, 软件定义安全与主动防御, 软件漏洞检测与网络攻防.