计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
2025,34(9):1−10 [doi: 10.15888/j.cnki.csa.009932] [CSTR: 32024.14.csa.009932]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于函数依赖的智能合约 TOD 漏洞检测①

姜天琪[1,2], 严　飞[1,2]

[1](武汉大学 国家网络安全学院, 武汉 430072)
[2](空天信息安全与可信计算教育部重点实验室 (武汉大学), 武汉 430072)
通信作者: 严　飞, E-mail: yanfei@whu.edu.cn

摘　要: 随着区块链技术的广泛应用, 智能合约的安全性问题日益突出. 交易顺序依赖 (transaction order dependency, TOD) 漏洞是一种常见且危害性极大的漏洞, 可能引发严重的经济损失. 现有漏洞检测方法主要分为静态分析和动态分析, 但仍存在误报率高、关键路径覆盖不足及对固定规则依赖等局限性. 为此, 本文提出了一种基于函数依赖指导的 TOD 漏洞检测框架 FuncFuzz. 该框架通过静态分析模块提取合约的关键函数依赖, 精准定位脆弱区域, 提升测试用例生成的针对性; 设计多样化的交易变异策略, 扩展测试用例的覆盖范围; 并引入基于状态的一致性判定机制, 以突破传统固定模式的限制, 动态适应复杂或未知的漏洞场景. 实验结果表明, FuncFuzz 在检测 TOD 漏洞的有效性方面优于现有工具, 同时函数依赖指导有效增强了检测效果.

关键词: 交易顺序依赖; 静态分析; 模糊测试; 差分分析; 智能合约安全

引用格式: 姜天琪,严飞.基于函数依赖的智能合约 TOD 漏洞检测.计算机系统应用,2025,34(9):1–10. http://www.c-s-a.org.cn/1003-3254/9932.html

Function-dependency-based TOD Vulnerability Detection in Smart Contract

JIANG Tian-Qi[1,2], YAN Fei[1,2]

[1](School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China)
[2](Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education (Wuhan University), Wuhan 430072, China)

Abstract: Security issues in smart contracts have become increasingly prominent with the widespread use of blockchain technology. The transaction order dependency (TOD) vulnerability is a common and highly hazardous flaw that can lead to significant financial losses. Existing detection methods primarily consist of static and dynamic analysis methods but still face challenges such as high false positive rates, insufficient coverage of key paths, and reliance on fixed rules. To address these issues, this study introduces FuncFuzz, a novel TOD vulnerability detection framework guided by functional dependence. This framework uses a static analysis module to extract key functional dependence within contracts, accurately locates vulnerable areas, and enhances the specificity of generated test cases. In addition, it designs diverse transaction mutation strategies to expand test case coverage and introduces a state-based consistency determination mechanism to break through the constraints of traditional fixed patterns and dynamically adapt to complex or unknown vulnerability scenarios. Experimental results show that FuncFuzz outperforms existing tools in detecting TOD vulnerabilities, and guidance by functional dependence significantly enhances detection effectiveness.

Key words: transaction order dependency (TOD); static analysis; fuzz testing; differential analysis; smart contract security

① 基金项目: 湖北省重大科技攻关项目 (尖刀) (2023BAA027)
收稿时间: 2025-01-05; 修改时间: 2025-02-12; 采用时间: 2025-05-06; csa 在线出版时间: 2025-07-25
CNKI 网络首发时间: 2025-07-28

随着区块链技术的快速发展, 其应用场景日益广泛, 涵盖金融[1]、供应链管理[2]、跨境贸易[3]、物联网[4]、医疗保健[5]等多个关键领域. 智能合约作为一种自动化执行协议, 能够在无需可信第三方干预的情况下完成交易. 一旦预设条件满足, 合约条款将不可逆地执行, 从而显著提升了交易效率和可靠性. 然而, 智能合约的特点也使其面临独特的安全风险, 例如代码漏洞、逻辑错误和交互复杂性引发的安全问题[6]. 其中交易顺序依赖漏洞 (transaction order dependency, TOD) 是一类常见且危害性极大的漏洞[7].

TOD 漏洞使得恶意攻击者能够从智能合约中获利[8], 去中心化应用安全项目 (decentralized application security project, DASP) 已将 TOD 漏洞列为智能合约十大常见漏洞之一, 并指出其在智能合约安全中的重要性[9]. 在以太坊平台上, 验证者负责通过打包一组交易来创建和验证区块. 所有提交给以太坊的交易首先被放置在一个名为“内存池”的缓冲区中, 随后验证者从中选择交易, 并安排其在下一个区块中执行. 由于内存池中的交易顺序没有任何保证[10], 当多个交易在同一区块内针对同一合约批量执行时, 如果某笔交易修改了合约的状态变量, 可能导致其他交易结果出现不确定性. 这种不确定性被攻击者利用, 形成了前向攻击, 即通过操纵交易顺序来实现获利. 文献[11]表明以太坊上的前向攻击导致了巨大的经济损失, 在以太坊合并更新之前, 损失累计超过 6.75 亿美元. 此外, 自 2024 年年中以来, 比特币网络中的前向攻击现象急剧增加, 并开始威胁到普通用户参与 Ordinals 等交易的能力[12].

目前, 针对 TOD 漏洞的检测方法面临诸多挑战. 首先, 静态分析工具 (如 Securify[13]、ZEUS[14]) 在处理智能合约中的复杂依赖路径和状态交互时, 往往需要对合约的执行路径进行近似处理. 然而, 这种方法在面对多重函数调用等场景时, 难以准确捕捉漏洞的上下文信息, 导致误报率显著提高. 其次, 动态分析工具 (如 Smartian[15]、ConFuzzius[16]) 虽然通过模糊测试生成大量测试用例, 旨在覆盖广泛的代码路径, 但由于智能合约中的漏洞往往集中在特定代码区域[17], 随机测试难以精准定位并覆盖这些区域, 漏报问题较为严重. 此外, 许多现有方法 (如 TODChecker[8]、TODLER[18]) 依赖预定义的漏洞模式, 在应对未知或复杂场景时适应性有限, 难以检测新型或变种的 TOD 漏洞.

针对上述问题, 本文提出了一种基于函数依赖指导的智能合约交易顺序依赖漏洞检测框架 (FuncFuzz). 该框架通过静态分析提取函数与全局变量的依赖关系, 构建依赖关系图. 这种依赖关系不仅用于生成覆盖关键路径的交易序列, 还用于指导后续的函数交换操作, 从而精准定位并覆盖关键脆弱区域, 解决了现有方法漏报率高的问题. 此外, 为了提升测试覆盖范围, 本文设计了一套高效的交易序列生成与变异策略, 通过优先覆盖高风险路径, 并在函数调用顺序、输入参数等多个维度进行变异, 丰富了测试用例, 弥补了随机测试在关键区域覆盖不足的问题. 最后, 本文提出了一种基于状态变化的动态漏洞判定机制, 通过对待测序列和重排序列的真实执行结果进行差分分析, 从真实执行中判定漏洞, 精准识别潜在漏洞, 避免固定规则带来的误判, 确保了极低的误报率.

实验结果显示, FuncFuzz 在人工生成的漏洞数据集和包含真实合约的多样化数据集上展现了显著优势. 与 ConFuzzius[15]、SAILFISH[19]和 TransRacer[20]等工具相比, FuncFuzz 在检测 TOD 漏洞时更为有效.

本文的主要贡献如下.

(1) 提出了一种融合静态分析与动态测试的函数依赖指导检测框架, 该框架利用函数依赖指导交易序列生成和序列重排操作, 从而精准定位脆弱路径.

(2) 设计了一套从随机测试转变为有针对性的高效交易序列生成与变异策略, 该策略基于函数依赖指导, 优先覆盖高风险路径, 并通过多维度交易变异, 提升了测试的针对性.

(3) 提出了一项从预定义模式匹配转变为基于真实执行状态的动态判定机制, 通过真实执行和状态差分分析, 从实际运行结果中动态判定漏洞.

(4) 通过实验评估了框架的有效性, 结果表明在人工生成的漏洞数据集以及包含真实合约的多样化数据集上, FuncFuzz 的漏洞检测有效性均优于现有工具.

1 相关工作

针对智能合约交易顺序依赖漏洞的研究主要集中在静态分析和动态分析两方面. 这些方法在技术实现和应用场景中各有侧重, 但在检测的准确性、覆盖率以及适用性方面仍存在明显不足.

静态分析方法通过对合约代码的建模和结构化分析, 能够快速定位潜在漏洞, 无需运行实际代码, 因而效率较高. Securify[13]利用依赖图建模分析全局变量与

交易顺序的关系, 可以高效检测交易冲突问题. 然而, 在处理跨函数调用等复杂依赖场景时, 其依赖关系模型的精度不足, 可能无法准确捕捉关键路径上的潜在风险, 导致误报. ZEUS[14]通过污点分析技术评估变量依赖关系及其对资金流转的影响, 在简单场景中表现较好, 但在复杂合约逻辑中, 静态建模可能无法充分揭示深层次依赖关系. Oyente[10]作为最早使用符号执行技术检测交易顺序依赖漏洞的工具, 通过路径条件验证交易依赖性, 但符号执行方法容易受路径爆炸问题的限制, 因此在大规模合约等场景中的效率较低.

一些改进的静态分析框架尝试结合符号执行与轻量化分析以提高检测效率和精度. 例如, SAILFISH[19]通过静态路径探索结合符号执行, 减少了部分误报, 但在面对动态交互场景时, 缺乏对运行时状态变化的全面支持, 导致关键路径覆盖不足. Nyx[21]利用静态依赖关系建模检测前置运行漏洞, 尽管效率较高, 但过于依赖已知模式, 难以覆盖未知或复杂场景.

静态分析方法的核心局限在于缺乏运行时动态信息的支持, 通常通过近似建模处理复杂状态交互, 这种方式在复杂函数依赖等场景中容易引发误报, 限制了其适用性和精度.

动态分析方法通过运行时模拟捕捉合约的真实行为, 适合处理复杂交互场景和多样化交易路径. 例如, Smartian[15]结合静态分析生成交易序列作为测试用例种子, 并通过模糊测试对其进行变异以检测交易依赖问题. 尽管提升了对复杂路径的检测能力, 但生成的测试用例往往过于依赖初始种子的质量, 对关键路径的覆盖仍显不足. ConFuzzius[16]通过结合数据依赖感知技术和模糊测试生成更具针对性的测试用例, 在一定程度上增强了覆盖率, 但其对已知漏洞模式的依赖限制了对未知场景的适应性. TransRacer[20]通过识别隐藏在特定合约状态中的交易竞争提升了检测能力, 但高度依赖合约的初始状态. EtherFuzz[22]利用简单的交易顺序变异策略生成测试用例, 适用于基本交易路径的分析, 但其变异策略过于单一, 每次仅交换最后两个函数的执行顺序.

此外, 一些链上合约分析器, 如 IcyChecker[23], 通过重放历史交易检测潜在的 TOD 漏洞, 能够捕捉真实交易环境中的动态行为. 然而, 该方法严重依赖已有历史数据, 对新型攻击方式和合约逻辑创新的适应性较差.

动态分析方法通常注重覆盖率, 但在生成测试用例时往往忽略对漏洞脆弱区域的针对性探索, 这种广覆盖策略可能导致资源的浪费.

漏洞判定是交易顺序依赖漏洞检测中的关键环节, 但现有的工具普遍依赖固定规则或模式化设计, 对未知漏洞的适应性有限, 难以覆盖新型的攻击模式. 例如, TODLER[18]利用预定义规则快速判定交易顺序依赖问题. 这种规则化设计虽然在简单场景中效率较高, 但在应对复杂的交易路径时, 容易出现误报和漏报.

综上所述, 现有方法的不足主要集中在以下几个方面: 1) 静态分析在依赖关系建模精度上的局限; 2) 动态分析在关键路径覆盖策略上的不足; 3) 漏洞判定方法对固定模式的过度依赖. 这些问题显著限制了现有工具的检测能力和适用性.

2 FuncFuzz 检测框架介绍

2.1 威胁模型

区块链可以被视为一个状态转换系统, 其中每个状态σ由存储在区块链上的数据表示. 智能合约通过全局状态变量来存储和访问持久性数据, 这些变量是智能合约逻辑运转的核心. 一次交易可以修改状态变量, 从而引发合约状态的更新, 而这种更新会永久记录在区块链上以确保不可篡改性. 状态变量在存储合约关键数据的同时, 也成为攻击者潜在的攻击目标. 一旦攻击者能够操控这些关键状态变量, 可能导致智能合约逻辑被篡改, 从而对合约所有者或用户造成严重后果, 例如财务损失或权限滥用.

在本文的威胁模型中, 假设攻击者能够操控事务的执行顺序, 利用交易顺序依赖漏洞发起攻击. 这种攻击无需修改合约代码或区块链协议, 而是通过发送精心设计的事务并调整其执行顺序来影响合约状态. 例如, 攻击者可以通过调整两个函数的调用顺序导致合约逻辑的异常执行, 从而引发状态的不一致.

具体而言, 给定一个智能合约$c=\langle address, b, V, F, s_0\rangle$, 其中, $address$是用于标识智能合约的合约地址, b是合约余额, V是存储变量的集合, F是合约函数的集合; $s_0:\{b\}\cup V\to R^n$是初始合约状态, 它将b和V中的变量映射到R中的具体值. 本文考虑两个事务t_1和t_2, 分别调用合约中的两个函数f_1和f_2, 如果满足以下 3 个条件, 则表明存在 TOD 漏洞.

$$\left(R_{t_1}\cap W_{t_2}\right)\cup\left(W_{t_1}\cap R_{t_2}\right)\cup\left(W_{t_1}\cap W_{t_2}\right)\neq\varnothing \quad (1)$$

$$s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \text{和} s_0 \xrightarrow{t_2} s_1' \xrightarrow{t_1} s_2' \tag{2}$$

$$s_2 \neq s_2' \tag{3}$$

首先, 事务t_1和t_2的交叉依赖关系如式 (1) 所示, 其中, R和W分别表示事务的读写操作所访问的状态变量集合, 表明两个函数对状态变量的访问操作可能相互影响; 其次, 如式 (2) 所示, 不同的事务调用顺序可能引发状态更新路径的差异; 最后, 式 (3) 进一步表明调用顺序的变化会导致最终状态的不一致, 从而引发交易顺序依赖漏洞. 这种漏洞的根本原因在于事务依赖关系没有被充分考虑, 从而使得攻击者能够通过顺序调整引发系统执行结果的差异.

本文的研究旨在识别和检测这些由交易顺序依赖引发的竞争条件和安全风险. 通过分析冲突函数对并交换其调用顺序, 观察其是否会引发合约状态的不一致. 本文特别关注这些函数对在不同顺序下对状态变量的影响, 以及顺序变更是否导致逻辑异常. 通过这种分析方法, 能够有效检测由交易顺序依赖引发的漏洞, 从而避免智能合约在非确定性事务执行顺序下产生状态不一致, 防止潜在的安全隐患.

2.2 框架概述

为了应对交易顺序依赖漏洞检测中的关键问题, 本文提出了 FuncFuzz 框架, 通过结合函数依赖分析模块 (模块 1)、交易序列生成模块 (模块 2)、序列重排与差分分析模块 (模块 3) 来系统化地检测 TOD 漏洞. 框架的整体架构如图 1 所示.

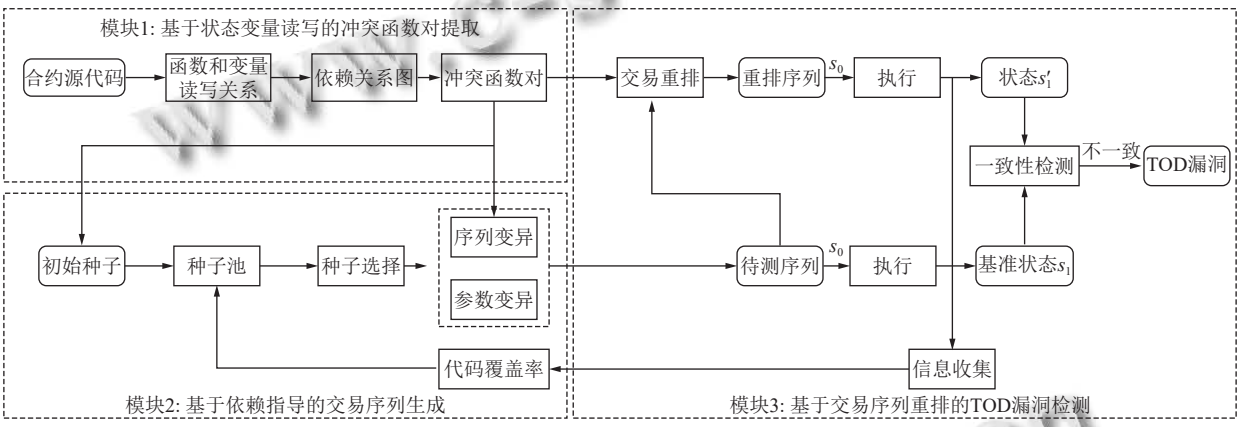

图 1 FuncFuzz 框架图

(1) 函数依赖分析模块

在模块 1 中, FuncFuzz 通过静态分析智能合约的源代码, 识别函数和状态变量间的读写关系, 并在此基础上构建“依赖关系图”来表示函数之间的依赖关系. 特别地, 基于函数对状态变量访问的建模方法能够更精准地捕捉到潜在的冲突函数对: 凡是访问相同状态变量并且至少有一个写操作的函数对, 都可能在不同调用顺序下导致合约最终状态出现差异, 从而引发 TOD 漏洞. 相比传统方法, 该方法可以有效缩小分析空间, 聚焦于真正可能触发漏洞的冲突函数对, 为后续的交易序列生成和重排提供了关键的指导信息.

(2) 交易序列生成模块

在模块 2 中, FuncFuzz 以冲突函数对为核心, 动态构造覆盖脆弱代码的测试序列. 特别地, 本模块设计了基于依赖指导的交易序列生成和变异方法, 旨在尽可能多地覆盖冲突函数对, 从而提高测试用例的针对性. 具体而言, 初始种子生成阶段, 优先安排对冲突函数对的调用; 序列变异阶段, 采用冲突函数对替换/插入策略. 此外, 结合代码覆盖率信息, 动态扩展测试序列. 相比随机生成测试用例的方式, 该方法能够生成更具针对性的测试用例.

(3) 序列重排与差分分析模块

在模块 3 中, FuncFuzz 利用模块 1 得到的冲突函数对, 对生成的待测交易序列进行重排, 模拟不同的函数调用顺序. 为了提高交换效率, 本模块将函数映射到对应索引, 并同步交换函数参数, 保证重排的精度和有效性. 具体来说, 本模块交换待测序列中冲突函数对的位置, 并在同一初始状态下分别执行待测序列与重排序列. 特别地, 每次执行完成后, 记录最终状态并进行回滚, 对比两个序列最终的合约状态是否一致: 如果出现不一致, 则标记为潜在的 TOD 漏洞. 与依赖固定模式匹配的检测方法相比, 该策略更具灵活性和适应性.

通过 3 个模块的协同工作, FuncFuzz 框架能够高效、全面地识别智能合约中的交易顺序依赖漏洞.

3 系统实现

3.1 函数依赖分析模块

函数依赖分析模块旨在对智能合约源代码进行全面解析, 自动识别合约函数与全局状态变量之间的交互关系, 进而构建依赖图, 从而有效识别潜在的冲突函数对. 这一分析不仅能够揭示智能合约内部的潜在逻辑缺陷, 还为后续模糊测试和差分分析提供了坚实的技术支撑. 以下为模块的具体设计与实现流程.

3.1.1 依赖关系图的构建

依赖关系图的构建是函数依赖分析的核心环节. 该过程以智能合约源代码的解析为起点, 目标是自动提取公开函数及其对全局状态变量的操作行为. 公开函数 (包括公共函数和外部函数) 通常是智能合约的主要外部接口, 也是潜在攻击者利用的入口点, 因此分析这些函数至关重要.

为实现依赖关系的全面追踪, 使用 Slither[24]工具对合约源代码进行解析, 生成中间表示 (IR) 和控制流模型. 这些中间结果不仅提供了合约逻辑结构的抽象, 还为后续依赖分析提供了丰富的语义信息. 本模块重点提取与全局存储变量相关的操作, 包括变量的读取、写入及依赖关系. 接着, 这些操作被整合到过程间控制流图 (ICFG) 中, 捕捉函数之间的调用关系及其对存储变量的访问路径. ICFG 的构建不仅关注单个函数的行为, 还深入分析函数调用链, 确保对跨函数依赖关系的完整建模.

在此基础上, 系统构建了依赖关系图, 其中包括两类节点: 全局存储变量节点和对这些变量执行操作的函数节点. 在图的构建过程中, 每个语句与存储变量的关系被逐一分析. 如果某语句修改了存储变量, 就添加一条“写入边”; 如果某语句依赖变量的值, 则建立“数据依赖边”; 若语句之间存在顺序上的依赖关系, 则用“顺序边”表示. 通过这种设计, 依赖关系图能够清晰地表示存储变量与操作之间的相互作用及依赖关系.

3.1.2 冲突函数对识别

冲突函数对的识别是本模块分析的关键目标. 冲突函数对指两个对同一存储变量执行操作的函数组合, 这些函数因执行顺序的不确定性可能引发竞争条件, 从而导致智能合约的状态不一致. 本模块通过分析依赖关系图, 自动提取所有潜在的冲突函数对, 分类如下:

(1) 写-写冲突: 如果两个函数对同一个状态变量进行写入操作, 而这些写入操作之间的顺序无法保证, 则可能导致状态不一致.

(2) 读-写冲突: 当一个函数在另一个函数完成写入之前读取了同一个状态变量的值, 就会发生读写竞争.

在冲突函数对的识别过程中, 本模块通过查询依赖关系图, 提取所有涉及存储变量的相关函数对, 并记录为二元组的形式 $\langle s_1, s_2 \rangle$, 其中 s_1 和 s_2 都是对同一存储变量操作的两个函数, 且至少有一个函数对该变量执行写入操作.

如图 2 所示合约中, 函数 withdraw 依赖全局变量 investor, 并尝试向其发送资金, 而函数 setInvestor 修改了全局变量 investor 的值. 通过依赖关系图分析, 识别到 withdraw 和 setInvestor 之间存在读-写冲突. 在这种冲突中, withdraw 可能在变量 investor 的值未被正确设置前读取该值, 导致逻辑失效. 最终, 模块将检测到的冲突函数对记录为〈withdraw, setInvestor〉并输出到日志文件, 支持后续模糊测试和差分分析的深入研究.

```
contract Preallocation{
    address investor;
    function withdraw() onlyOwner{
        uint bal = this.balance;
        if (!investor.send(bal)) {
            throw;
        }
    }
    function setInvestor(address _investor) onlyOwner {
        investor = _investor;
    }
}
```

图 2 函数依赖示例

3.2 交易序列生成模块

交易序列生成模块旨在通过高效的测试序列构造与变异策略, 覆盖更多可能的交易场景, 为潜在的交易顺序依赖漏洞分析提供支持. 本模块在 sFuzz[25]的基础上进行了扩展, 不仅能够生成高质量的初始测试序列, 还通过灵活多样的变异策略, 显著扩展交易场景的覆盖范围, 为后续的漏洞分析提供全面的基础.

3.2.1 初始测试序列生成

本模块以函数依赖分析模块识别的冲突函数对为核心, 优先生成可能覆盖高风险场景的调用序列, 同时通过参数的随机化填充进一步提升测试序列的多样性和适应性.

在生成初始测试序列时, 首先提取冲突函数对, 并

优先将这些函数对组合成调用序列. 例如, 给定一个合约, 其中包含了图 2 所示的冲突函数对, 由于现有初始序列生成方法往往采用随机生成方式, 生成的初始测试序列可能不包含冲突函数对, 从而无法触发 TOD 漏洞. 本模块能够确保检测到的冲突函数对首先被添加到初始测试序列中. 这一设计确保了初始测试用例能够集中覆盖可能引发交易顺序依赖的关键场景. 在此基础上, 系统进一步随机生成非冲突函数索引, 将其与冲突函数对融合, 形成更加复杂和多样化的调用序列. 通过动态化和随机化生成策略, 每次运行生成的序列均具有唯一性, 从而有效扩大了场景覆盖范围.

在参数填充方面, 采用了一种分层填充策略, 能够根据参数类型和结构灵活生成对应的输入数据. 对于标量类型参数, 系统根据函数签名随机生成值; 而对于数组或多维数组, 通过递归生成的方式构造出复杂的嵌套结构, 模拟真实场景中的多样化输入. 此外, 动态长度参数采用随机长度生成策略, 使测试用例能够更加贴合实际应用场景.

为了模拟真实的区块链环境, 还引入了链上环境参数的填充功能, 包括发送者地址、区块号和时间戳等. 这些参数以 32 字节对齐方式统一添加到序列中, 确保生成的用例能够反映真实环境下的调用情况.

通过高风险路径优先覆盖、参数填充灵活性以及真实场景模拟的结合, 本模块能够生成具有高度针对性和多样性的初始测试序列, 为后续变异操作奠定了坚实的基础.

3.2.2 测试序列多样化变异

在初始测试序列的基础上, 通过引入多维度的变异策略进一步扩展测试场景, 生成更多可能的调用序列. 这些变异策略涵盖了函数参数和函数调用序列两个方面, 通过对输入数据和调用结构的调整, 提高了测试场景的复杂性和覆盖范围.

(1) 函数参数变异策略

参数变异策略主要针对单个交易的输入参数, 通过修改其内容来模拟各种可能的输入场景. 具体变异方式如表 1 所示.

(2) 函数调用序列变异策略

除了针对输入参数的变异, 系统还支持对交易调用序列的多样化修改. 在变异过程中, 该策略优先采用函数依赖分析模块提供的冲突函数对, 替换或插入原有序列中的函数调用, 从而构造出更易触发交易顺序依赖漏洞的用例. 调用序列的变异策略通过调整交易顺序或结构, 生成更复杂的交易场景, 进一步扩展测试用例的覆盖范围, 具体变异方式如表 2 所示.

表 1 函数参数变异策略

变异操作	细节
单/双/四比特翻转	随机选择输入参数以比特为单位0/1翻转
单/双/四字节翻转	随机选择输入参数以字节为单位0/1翻转
单/双/四字节算术变异	随机选择输入参数对字节进行加或减运算
特殊值替换	根据具体的数据类型选择特殊值作为输入
字典覆盖	利用预定义字典或地址字典覆盖输入参数
Havoc模式	随机组合比特翻转、特殊值替换和字典覆盖等多种变异操作

表 2 函数调用序列变异策略

变异操作	细节
替换函数调用	随机选择交易序列中的函数调用进行替换
增加函数调用	向交易序列中插入新的函数调用
删除函数调用	随机删除某些函数调用
数据拼接	结合多个测试用例的数据片段生成新的交易序列

通过高风险初始测试序列的生成与多样化变异策略的结合, 交易序列生成模块能够构建多样化的交易场景, 从而全面提升对可能场景的覆盖范围. 这些策略的协同作用不仅提高了测试覆盖的深度和广度, 还为后续的漏洞检测模块提供了更加丰富的输入基础.

3.3 序列重排与差分分析模块

序列重排与差分分析模块旨在验证交易顺序依赖漏洞. 其设计理念基于智能合约的全局状态应与交易调用顺序无关, 即便交易的调用顺序发生变化, 最终状态也应保持一致. 通过结合静态分析和动态测试, 系统地挖掘因调用顺序变动可能引发的安全隐患.

3.3.1 基于索引映射的序列重排

序列重排通过调整函数调用顺序, 模拟潜在的风险场景, 发现可能导致全局状态不一致的漏洞. 变异过程以函数索引解析为基础, 并结合灵活的变异策略, 确保生成的序列语义一致且结构合理.

首先, 模块从函数依赖分析模块中获取冲突函数对的集合. 基于这些函数对进一步提取其调用索引, 确保序列重排的操作具有针对性和准确性. 例如, 假设合约中存在两个函数 A 和 B, 其索引分别为 indexA 和 indexB, 通过调整调用顺序, 可以构造出 A→B 或 B→A 的测试序列. 采用索引映射函数能精确地将每个函数调用与其在原始序列中的位置对应起来, 从而在交换冲突函数对的函数位置过程中确保顺序调整的精度,

提高交换效率, 并避免无关部分的随机变动.

在序列重排过程中, 本模块在调整函数调用顺序的同时对参数位置进行同步更新, 以确保数据结构的完整性. 具体而言, 提取每个函数对应的参数, 在交换函数调用顺序的同时调整参数的位置和长度, 确保参数与对应的函数保持匹配关系. 这一操作保证了重排序列的语义一致性, 即便处理动态长度参数时, 也能通过重新计算偏移与容器长度, 避免潜在数据错误. 借助参数同步机制, 本模块得以有效避免在函数顺序调整时出现潜在的参数错位风险, 进而为后续的测试提供准确且可靠的执行序列.

序列重排与参数同步的伪代码见算法 1, 描述了本模块生成重排序列以及同步调整参数的过程.

算法 1. 序列重排与参数同步

输入: 待测序列 Seq_{test}, 冲突函数对集合 $Pairs$.
输出: 重排序列 $ReorderedSeqs$.

1. $ReorderedSeqs \leftarrow \varnothing$
2. for each (f_x, f_y) in $Pairs$ do
3. $index_x \leftarrow get\_index(f_x)$
4. $index_y \leftarrow get\_index(f_y)$
5. $param_x \leftarrow get\_params(f_x)$
6. $param_y \leftarrow get\_params(f_y)$
7. $swap(f_x, f_y)$
8. $adjust\_params(param_x, param_y)$
9. $Seq_{reordered} \leftarrow create\_sequence(f_x, f_y)$
10. $ReorderedSeqs \leftarrow ReorderedSeqs \cup \{Seq_{reordered}\}$
11. end for
12. return $ReorderedSeqs$

3.3.2 基于合约状态的差分分析

差分分析用于验证交易顺序依赖漏洞是否真实存在. 通过对比待测序列和重排序列的执行结果, 判断函数调用顺序对智能合约全局状态的一致性影响.

本模块为每组测试序列 (包括待测序列和重排序列) 创建独立的虚拟执行环境, 每个执行环境的初始状态相同. 执行过程中, 模块持续记录全局变量的状态变化、读写行为及调用上下文. 通过在独立环境中分别执行原始和重排后的序列, 能够直观地比较二者的状态变化.

通过对比待测序列和重排序列的最终状态, 可以判断函数调用顺序是否对合约全局状态产生了影响. 如果全局变量值在两个序列中出现不一致, 则表明存在交易顺序依赖漏洞. 这种基于状态差分对比的方法直接反映了函数调用顺序变化所带来的影响, 为发现漏洞提供了明确的证据, 有效避免了模式匹配方法的局限性.

序列差分验证的伪代码见算法 2, 描述了本模块执行并对比序列以验证交易顺序依赖漏洞的过程.

算法 2. 序列差分验证

输入: 初始状态 S_0, 待测序列 Seq_{test}, 重排序列 $ReorderedSeqs$.

1. $Env_{test} \leftarrow create\_environment(S_0)$
2. $Env_{reordered} \leftarrow create\_environment(S_0)$
3. $execute\_sequence(Env_{test}, Seq_{test})$
4. $S_{test} \leftarrow get\_final\_state(Env_{test})$
5. for each $Seq_{reordered}$ in $ReorderedSeqs$ do
6. $execute\_sequence(Env_{reordered}, Seq_{reordered})$
7. $S_{reordered} \leftarrow get\_final\_state(Env_{reordered})$
8. if $S_{reordered} \neq S_{test}$ then
9. $log\_difference(Seq_{reordered})$
10. end if
11. end for

4 实验评估

为了验证 FuncFuzz 在智能合约 TOD 漏洞检测方面的有效性和性能表现, 本文进行了系统的实验, 以回答以下研究问题.

RQ1: 与现有工作相比, FuncFuzz 在检测 TOD 漏洞方面的有效性如何?

RQ2: FuncFuzz 在漏洞检测中的性能开销如何?

RQ3: 静态分析阶段指导漏洞检测的作用如何?

为了确保实验结果的公平性, 所有实验均在统一环境下进行, 具体运行环境如表 3 所示.

表 3 实验环境

名称	版本
CPU	Intel Core i7-10875H
内存 (GB)	16
存储 (GB)	512
操作系统	Ubuntu 20.04.6 LTS
Linux Core	5.4.0-200-generic

4.1 数据集与评价指标

本文实验在以下两个数据集上进行.

(1) 数据集 D1: 本文采用 SolidiFi[26]工具, 通过漏洞注入的方式在 50 条无漏洞合约的基础上人工生成了 50 条漏洞合约, 共计 100 条合约.

(2) 数据集 D2: 由于目前以太坊官方尚未提供含 TOD 漏洞的合约数据集, 本文通过整合 ScrawLD 数据集[27]和 CGT 数据集[28], 构建了含 TOD 漏洞的 1 132 个

真实合约.

本文采用以下评价指标: 正确检测出的漏洞合约数量 (*TP*)、漏报数量 (*FN*)、正确检测出的安全合约数量 (*TN*)、误报数量 (*FP*)、误报率 (*FPR*=*FP*/(*TN*+*FP*))、召回率 (*TPR*=*TP*/(*TP*+*FN*))、准确率 (*Acc*=(*TP*+*TN*)/(*TP*+*TN*+*FP*+*FN*)) 和平均检测时间 (AVT).

4.2 实验结果

4.2.1 有效性评价

为了验证 FuncFuzz 在检测 TOD 漏洞方面的有效性, 本文分别在两个数据集上进行了实验, 并与现有工作 SAILFISH、TransRacer 和 ConFuzzius 进行对比. 其中 TransRacer 需要以太坊上的初始合约状态, 仅在真实数据集上进行了对比实验.

实验 1: 在数据集 D1 上, 本文统计了 SAILFISH、ConFuzzius 和 FuncFuzz 在注入漏洞合约中的检测结果, 实验结果如表 4 所示.

表 4 注入漏洞合约检测结果

检测工具	*TP* (个)	*FP* (个)	*TPR* (%)	*FPR* (%)	*Acc* (%)
FuncFuzz	50	0	100	0	100
SAILFISH	49	1	98	2	98
ConFuzzius	5	1	10	2	54

实验 2: 在数据集 D2 上, 本文深入分析了 SAILFISH、TransRacer、ConFuzzius 和 FuncFuzz 这 4 种工具在处理真实合约中的效果, 实验结果如表 5 所示.

表 5 真实合约检测结果

检测工具	*TP* (个)	*FN* (个)	*TPR* (%)
FuncFuzz	907	225	80.12
SAILFISH	616	516	54.42
TransRacer	579	553	51.15
ConFuzzius	142	990	12.54

实验 1 中, FuncFuzz 未产生任何漏报或误报, 准确率达 100%; SAILFISH 出现了 1 个漏报和 1 个误报, 准确率为 98%; 而 ConFuzzius 仅检测到 5 个漏洞, 召回率为 10%, 伴有 1 个误报, 准确率仅为 54%. 实验 2 中, FuncFuzz 的召回率达到 80.12%, 显著优于 SAILFISH 的 54.42%、TransRacer 的 51.45% 和 ConFuzzius 的 12.54%.

实验结果表明, FuncFuzz 检测的有效性优于现有工作. FuncFuzz 通过结合静态分析和模糊测试, 展现了在漏洞检测中的优势. 静态分析识别的冲突函数对能够定位潜在漏洞的代码区域, 为模糊测试提供精准的输入. 在模糊测试阶段, FuncFuzz 优先执行包含冲突函数对的交易序列, 并利用动态变异策略扩展场景覆盖范围, 提升了对关键路径的检测能力.

与之相比, SAILFISH 的规则模板化方法在简单场景 (如数据集 D1) 中表现良好, 但其检测方法依赖于固定模式, 无法应对数据集 D2 中复杂的场景, 导致召回率下降. TransRacer 通过函数间的关系检测部分 TOD 漏洞, 但其路径生成策略倾向于简单序列, 未能涵盖所有可能的交易冲突, 复杂场景中的路径覆盖不足. 此外, 其对初始状态的依赖进一步限制了其检测范围. ConFuzzius 的路径探索能力较弱, 检测策略未能有效覆盖关键路径, 而 TOD 漏洞代码往往集中于有限逻辑部分, 这导致其在两个数据集上的表现均最差.

对于 FuncFuzz 在数据集 D2 上的漏报问题, 本文分析了主要原因. 一是静态分析的局限性: 部分冲突函数对未被识别, 导致模糊测试覆盖不足; 二是路径复杂性: 模糊测试在处理深度依赖路径时, 因执行时间限制未能完成所有探索. 尽管如此, FuncFuzz 能够提供引发状态不一致的原始序列和重排序列, 这些序列均经过实际执行验证, 保证了检测结果的准确性. 由于结果基于实际执行反馈, 因此 FuncFuzz 的误报率可忽略.

4.2.2 性能开销

FuncFuzz 的性能开销在于如下两个部分.

(1) 静态分析: 用于收集冲突函数对. 本文在实验 2 中统计了静态分析时间, 根据合约行数将合约分为大、中、小 3 种, 并统计平均检测时间. 实验结果见表 6.

表 6 静态分析时间开销

合约规模	合约行数	合约数量	AVT (s)
小	(0, 500)	917	7.63
中	[500, 1000)	165	24.72
大	[1000, ∞)	51	85.10

(2) 模糊测试: 用于交易序列的生成、变异和执行. 模糊测试阶段的开销主要与交易序列的复杂性和覆盖范围相关. 本文在实验中设置的超时时间为 5 min, 作为探索交易序列的限制条件. 模糊测试在实际运行中表现出较高的灵活性, 可动态调整序列生成和变异策略, 从而在性能与有效性之间取得平衡.

考虑到 FuncFuzz 在漏洞检测方面的有效性, 本文认为这样的开销是可以接受的.

4.2.3 消融研究

为更好地理解冲突函数对的指导作用, 本文进行了消融实验, 将冲突函数对替换为随机选择的函数对,

并分析检测结果.

实验 3: 在数据集 D2 中, 本文从具有冲突函数对且 FuncFuzz 检测出漏洞的合约中随机选择 100 个合约进行测试. 在相同实验环境下运行, 实验结果如表 7 所示.

表 7 无静态分析指导检测结果

配置	*TP* (个)	*FN* (个)	*TPR* (%)
仅模糊测试	67	33	67

消融实验结果显示, 在没有静态分析指导的情况下, FuncFuzz 的召回率有所下降. 由于随机选择的函数对可能未包含冲突操作, 模糊测试的路径探索受限, 影响了关键场景的覆盖. 然而, 即使在这种情况下, 模糊测试依然表现出一定的鲁棒性, 证明了其基础策略的有效性.

5 结论与展望

交易顺序依赖漏洞是智能合约中常见的安全问题, 可能导致状态变量的非确定性修改, 进而引发严重的经济损失. 现有检测方法虽然有所成效, 但仍存在以下不足: 静态分析方法在复杂函数依赖等场景中易产生误报. 动态分析工具则缺乏针对性, 导致关键路径覆盖不足. 此外, 依赖固定规则的漏洞判定方法在应对新型或复杂场景时表现出明显的适应性局限.

为解决这些问题, 本文提出了 FuncFuzz 框架, 通过静态分析提取合约关键函数依赖关系, 快速定位脆弱区域, 提升测试用例生成的针对性; 设计多样化交易变异策略, 扩展测试场景的覆盖范围; 并引入基于状态变化的一致性判定机制, 摆脱了传统规则化设计的局限性. 实验结果表明, FuncFuzz 在检测有效性方面优于现有方法, 能够精准定位交易顺序依赖漏洞, 并在真实场景中展现出良好的适用性.

未来研究将进一步提升检测效率和适用范围. 一方面, 优化静态分析方法, 综合考虑函数重要性及偏序依赖关系; 另一方面, 探索更高效的路径探索和交易序列生成策略, 提高对关键路径的覆盖能力. 此外, 还将拓展 FuncFuzz 对其他类型智能合约漏洞的检测能力.

参考文献

1 Treleaven P, Gendal Brown R, Yang D. Blockchain technology in finance. Computer, 2017, 50(9): 14–17. [doi: 10.1109/MC.2017.3571047]

2 Wang SP, Li DY, Zhang YL, *et al*. Smart contract-based product traceability system in the supply chain scenario. IEEE Access, 2019, 7: 115122–115133. [doi: 10.1109/ACCESS.2019.2935873]

3 李云鹏, 姜茸, 梁志宏. 基于区块链的跨境贸易数据共享与访问控制方案. 计算机系统应用, 2024, 33(10): 97–105. [doi: 10.15888/j.cnki.csa.009648]

4 Huh S, Cho S, Kim S. Managing IoT devices using blockchain platform. Proceedings of the 19th International Conference on Advanced Communication Technology. Pyeongchang: IEEE, 2017. 464–467. [doi: 10.23919/ICACT.2017.7890132]

5 Jamil F, Hang L, Kim KH, *et al*. A novel medical blockchain model for drug supply chain integrity management in a smart hospital. Electronics, 2019, 8(5): 505. [doi: 10.3390/electronics8050505]

6 Jiao TY, Xu ZY, Qi MF, *et al*. A survey of Ethereum smart contract security: Attacks and detection. Distributed Ledger Technologies: Research and Practice, 2024, 3(3): 23. [doi: 10.1145/3643895]

7 崔展齐, 杨慧文, 陈翔, 等. 智能合约安全漏洞检测研究进展. 软件学报, 2024, 35(5): 2235–2267. [doi: 10.13328/j.cnki.jos.007046]

8 Munir S, Taha W. Statically checking transaction ordering dependency in Ethereum smart contracts. Proceedings of the 6th ACM International Symposium on Blockchain and Secure Critical Infrastructure. Singapore: ACM, 2024. 1–8. [doi: 10.1145/3659463.3660013]

9 NCC Group. Decentralized application security project (or DASP) top 10 of 2018. http://dasp.co/. [2024-12-1].

10 Luu L, Chu DH, Olickel H, *et al*. Making smart contracts smarter. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna: ACM, 2016. 254–269. [doi: 10.1145/2976749.2978309]

11 The Merge. https://ethereum.org/en/roadmap/merge/. [2024-12-1].

12 Qi MF, Wang Q, Li NR, *et al*. BRC20 snipping attack. arXiv:2501.11942, 2025.

13 Tsankov P, Dan A, Drachsler-Cohen D, *et al*. Securify: Practical security analysis of smart contracts. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto: ACM, 2018. 67–82. [doi: 10.1145/3243734.3243780]

14 Kalra S, Goel S, Dhawan M, *et al*. ZEUS: Analyzing safety of smart contracts. Proceedings of the 25th Annual Network and Distributed System Security Symposium. San Diego:

The Internet Society, 2018. 1–12. [doi: 10.14722/ndss.2018.23082]

15 Choi J, Kim D, Kim S, *et al*. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering. Melbourne: IEEE, 2021. 227–239. [doi: 10.1109/ASE51524.2021.9678888]

16 Torres CF, Iannillo AK, Gervais A, *et al*. ConFuzzius: A data dependency-aware hybrid fuzzer for smart contracts. Proceedings of the 2021 IEEE European Symposium on Security and Privacy. Vienna: IEEE, 2021. 103–119. [doi: 10.1109/EuroSP51992.2021.00018]

17 Böhme M, Szekeres L, Metzman J. On the reliability of coverage-based fuzzer benchmarking. Proceedings of the 44th International Conference on Software Engineering. Pittsburgh: ACM, 2022. 1621–1633. [doi: 10.1145/3510003.3510230]

18 Munir S, Reichenbach C. TODLER: A transaction ordering dependency analyzer—For Ethereum smart contracts. Proceedings of the 6th IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain. Melbourne: IEEE, 2023. 9–16. [doi: 10.1109/WETSEB59161.2023.00007]

19 Bose P, Das D, Chen YJ, *et al*. SAILFISH: Vetting smart contract state-inconsistency bugs in seconds. Proceedings of the 2022 IEEE Symposium on Security and Privacy. San Francisco: IEEE, 2022. 161–178. [doi: 10.1109/SP46214.2022.9833721]

20 Ma CY, Song W, Huang J. TransRacer: Function dependence-guided transaction race detection for smart contracts. Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. San Francisco: ACM, 2023. 947–959. [doi: 10.1145/3611643.3616281]

21 Zhang WQ, Zhang Z, Shi QK, *et al*. Nyx: Detecting exploitable front-running vulnerabilities in smart contracts. Proceedings of the 2024 IEEE Symposium on Security and Privacy. San Francisco: IEEE, 2024. 2198–2216. [doi: 10.1109/SP54263.2024.00146]

22 Wang XY, Sun JZ, Hu CY, *et al*. EtherFuzz: Mutation fuzzing smart contracts for TOD vulnerability detection. Wireless Communications and Mobile Computing, 2022, 2022(1): 1565007. [doi: 10.1155/2022/1565007]

23 Ye MX, Nan YH, Zheng ZB, *et al*. Detecting state inconsistency bugs in DApps via on-chain transaction replay and fuzzing. Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle: ACM, 2023. 298–309. [doi: 10.1145/3597926.3598057]

24 Feist J, Grieco G, Groce A. Slither: A static analysis framework for smart contracts. Proceedings of the 2nd IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain. Montreal: IEEE, 2019. 8–15. [doi: 10.1109/WETSEB.2019.00008]

25 Nguyen TD, Pham LH, Sun J, *et al*. sFuzz: An efficient adaptive fuzzer for Solidity smart contracts. Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering. Seoul: ACM, 2020. 778–788. [doi: 10.1145/3377811.3380334]

26 Ghaleb A, Pattabiraman K. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2020. 415–427. [doi: 10.1145/3395363.3397385]

27 Yashavant CS, Kumar S, Karkare A. ScrawLD: A dataset of real world Ethereum smart contracts labelled with vulnerabilities. arXiv:2202.11409, 2022.

28 Di Angelo M, Salzer G. Consolidation of ground truth sets for weakness detection in smart contracts. Proceedings of the 2023 International Conference on Financial Cryptography and Data Security. Brač: Springer, 2023. 439–455. [doi: 10.1007/978-3-031-48806-1_28]

(校对责编: 王欣欣)