软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007685] [CSTR: 32375.14.jos.007685]

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

SolRaceFuzz: Solana 链上程序并发缺陷检测*

高 旭, 马辰阳, 叶 盛, 李昭松, 宋 巍

(南京理工大学 计算机科学与工程学院, 江苏 南京 210094)

通信作者: 宋巍, E-mail: wsong@njust.edu.cn

摘 要: 随着 Solana 区块链应用日益广泛, 虽有工作针对其安全弱点、生态系统、商业应用等方面进行了研究, 但 Solana 区块链的并发缺陷问题尚未得到足够重视. 研究发现 Solana 链上程序中存在两类特殊的并发缺陷类型, 分别称为交易顺序依赖缺陷与时钟控制依赖缺陷. 这些缺陷允许区块链网络中的恶意节点合法地针对并发交易交换它们之间的执行顺序, 或推迟特定交易执行到特定时间, 以干涉这些并发交易执行结果. 因此, 具有这些缺陷的 Solana 链上程序的并发交易可能遭受攻击者的人为控制. 针对上述问题, 分别定义 Solana 链上程序的交易顺序依赖缺陷和时钟控制依赖缺陷, 并提出一种基于模糊测试与符号执行的检测方法. 该方法包含这两种缺陷的检测准则, 以及针对 Solana 区块链特点的账本模拟方法与交易生成方法. 基于该方法实现了原型工具 SolRaceFuzz, 并在 3 组数据集上对缺陷发现能力、效率和探索策略这 3 个方面进行实验, 验证所提方法的有效性.

关键词: Solana 链上程序; 并发缺陷; 区块链交易; 模糊测试; 符号执行

中图法分类号: TP311

中文引用格式: 高旭, 马辰阳, 叶盛, 李昭松, 宋巍. SolRaceFuzz: Solana链上程序并发缺陷检测. 软件学报. http://www.jos.org.cn/1000-9825/7685.htm

英文引用格式: Gao X, Ma CY, Ye S, Li XS, Song W. SolRaceFuzz: Concurrency Defects Detection in Solana On-chain Programs. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7685.htm

SolRaceFuzz: Concurrency Defects Detection in Solana On-chain Programs

GAO Xu, MA Chen-Yang, YE Sheng, LI Xuan-Song, SONG Wei

(School of Computer Science and Engineering, Nanjing University of Science & Technology, Nanjing 210094, China)

Abstract: With the rapid adoption of the Solana blockchain, prior research has examined its security vulnerabilities, ecosystem, and commercial applications. However, concurrency defects in Solana on-chain programs have not received sufficient attention. This study identifies two special types of concurrency defects in Solana on-chain programs: transaction-order dependency defects and clock-control dependency defects. These defects allow malicious nodes in the blockchain network to legitimately reorder concurrent transactions or delay specific transactions until a specific time, thus affecting the execution outcomes of these concurrent transactions. As a result, concurrent transactions in Solana on-chain programs containing these defects may be vulnerable to adversarial control. To address this problem, this study defines transaction-order dependency defects and clock-control dependency defects in Solana on-chain programs and proposes a detection method based on fuzz testing and symbolic execution. The method includes detection criteria for both defect types, as well as a ledger simulation method and a transaction generation method tailored to the characteristics of the Solana blockchain. Based on this method, a prototype tool, SolRaceFuzz, is implemented, and experiments are conducted on three datasets to evaluate defect discovery capability, efficiency, and exploration strategy, verifying the effectiveness of the proposed method.

Key words: Solana on-chain program; concurrency defect; blockchain transaction; fuzz testing; symbolic execution

随着区块链技术的推广以及 Web 3.0 技术的逐渐成熟, 以 Solana、Ethereum 等为代表的区块链平台得到越来越多的应用[1]. 区块链本质上是一种分布式账本技术, 其具有不可篡改、透明和去中心化的特点. 它由一系列记录

* 基金项目: 国家自然科学基金 (62472221)
收稿时间: 2025-09-03; 修改时间: 2026-02-07; 采用时间: 2026-04-20; jos 在线出版时间: 2026-07-01

数据的区块按时间顺序链接而成, 每个区块的数据通常是一组关于交易的记录, 这些区块通过哈希算法进行链式校验, 并在分布式网络节点之间共享和验证, 从而起到防篡改的作用[2].

Solana 链上程序 (或称智能合约、去中心化程序) 是一种部署在 Solana 区块链上的自动化程序, 具有去中心化、透明和不可篡改的特性, 其使用标准 ELF (executable and linkable format) 文件格式存储, 并使用专用的 SBPF (Solana Berkeley packet filter) 指令集来编码程序指令. Solana 区块链能够在用户提交交易时自动调用相应的 Solana 链上程序执行特定操作, 其核心思想是通过代码直接实现各方之间的协议或交易[3].

与传统的程序并发缺陷不同, 本文发现 Solana 链上程序具有两类独特的并发缺陷类型, 分别称之为交易顺序依赖 (transaction order dependency, TOD) 缺陷与时钟控制依赖 (clock-control dependency, CCD) 缺陷. 这是 Solana 链上程序执行以公有区块链为基础所导致的. 由于 Solana 链上程序在去中心化网络中的任意节点上运行, 且大量交易被同时处理, 无法预测并发交易之间的精确执行顺序与精确执行时间. 更重要的是, Solana 作为公有链, 任何人都可以不经许可加入区块链网络, 因此其中的网络节点并非都是可以绝对信任的. 特定节点可以出于自身的利益在不违反集群共识规则的情况下合法地推迟特定交易的执行, 这意味着 Solana 链上程序也存在安全问题. 例如, 恶意节点可针对并发交易交换它们之间的执行顺序, 或推迟特定交易执行到更有利的时间, 以改变这些调用 Solana 链上程序的交易执行结果. 因此, 具有这些缺陷的 Solana 链上程序相关交易执行可能遭受攻击者的人为干涉.

针对上述问题, 本文提出一种基于模糊测试与符号执行的 Solana 链上程序并发缺陷检测方法. 针对 Solana 链上程序中的交易顺序依赖缺陷和时钟控制依赖缺陷, 分别给出正式定义, 并提出缺陷检测准则. 此外, 由于 Solana 链上程序的执行依赖于区块链账本的即时状态的特点, 本文提出了其账本状态的模拟方法. 最后, 针对 Solana 链上程序具备接口描述语言 (interface description language, IDL) 信息的特点, 提出一种利用 IDL 信息高效生成合法 Solana 交易的方法.

本文主要贡献如下.

(1) 定义 Solana 链上程序在并发执行时潜在的两类缺陷: 交易顺序依赖缺陷与时钟控制依赖缺陷.

(2) 基于模糊测试与符号执行, 提出针对两类缺陷的检测方法.

(3) 针对系统变量与程序派生地址两大挑战, 提出 Solana 区块链账本的模拟方法;

(4) 提出一种充分利用 Solana 链上程序所包含的 IDL 信息来高效生成其合法区块链交易的方法.

(5) 实现针对 SBPF 字节码的模糊测试原型工具 SolRaceFuzz, 并在 3 个数据集上进行实验与评估.

本文第 1 节介绍 Solana 区块链的研究现状. 第 2 节介绍本文背景知识, 包括 Solana 账本与交易模型、Solana 链上程序及模糊测试与符号执行. 第 3 节给出问题的正式定义并描述本文构建的 Solana 链上程序并发缺陷的检测方法. 第 4 节通过一系列实验验证了所提方法的有效性. 第 5 节讨论本文与相近工作的区别. 第 6 节总结全文.

1 相关工作

随着 Solana 区块链的日趋流行和广泛应用, 当前已有众多研究人员针对 Solana 的不同方面开展研究.

在 Solana 区块链的应用研究方面, Ashraf 等人[4]提出了集成区块链到物联网设备的原型, 使用 Solana 区块链来实现供应链处理和商业逻辑; Duffy 等人[5]探讨了 Solana 区块链的高性能对物联网应用的推动作用; Kong 等人[6]从纵向计量和洗盘交易安全审计两个角度对 Solana 非同质化数字代币进行了系统研究; SaFaR[7]通过集成 Solana 区块链实现了一个去中心化的拼车系统.

在 Solana 区块链的系统设计研究方面, Sliwinski 等人[8]研究 Solana 的共识协议, 展示其精心操纵的恶意验证器能够使 Solana 区块链宕机, 揭示了 Solana 在共识方面的潜在缺陷; Li 等人[2]对比特币、以太坊、Solana 在内的区块链技术进行了详尽全面地研究, 分析了影响企业的区块链可扩展性问题和性能问题是如何解决的, 并发现 Solana 对数据结构、流程和算法进行的创新显著改善了这些问题. Hung 等人[9]提出了对 eBPF 运行时进行模糊测试的 BRF 模糊器, 可以满足 eBPF 子系统所需的语义和依赖项.

Solana 区块链的链上程序也得到了各种研究. Andreina 等人[10]对 Solana 链上程序的开发者进行实际调查, 着

手理解开发人员面临的挑战并探讨安全漏洞的普遍性; Pierro 等人[11]提出一个工具用于检查高级编程语言编写的程序源代码是否对应于 Solana 区块链上已部署的字节码; Cui 等人[12]提出了 Solana 的第 1 个自动化链上程序检测框架 VRust, 可验证检测链上程序的签名检查缺失、整数溢出等 8 种缺陷; Smolka 等人[13]提出 FuzzDelSol, 使用模糊测试分析 Solana 链上程序当中的签名检查缺失、所有者检查缺失、公钥检查缺失等漏洞; Wang 等人[14]设计了 Solana 链上程序的克隆检测工具 SolaSim, 基于中间代码表示 (mid-level intermediate representation, MIR) 对 Solana 链上程序进行相似性检测, 以了解 Solana 生态系统中的代码重复使用; Madhwal 等人[15]深入分析了用于虚拟化 Solana 区块链上交易执行的架构设计和开发, 该虚拟化系统有助于以本机速度执行 Solana 链上程序, 同时在受控隔离环境中评估其执行结果, 可推动更详尽地测试 Solana 链上程序.

在这些研究中, VRust[12]和 FuzzDelSol[13]是与本文最相近的研究. 然而, VRust 采用数据流分析规则对 Solana 链上程序进行静态分析, 必须在获得程序源代码的情况下才能进行分析. 据本文所知, FuzzDelSol 是当前唯一公开的针对 Solana 的模糊测试工具, 其使用模糊测试对 Solana 链上程序的二进制文件进行动态分析, 但其未考虑区块链的状态迁移问题, 且输入由随机字节生成而未充分考虑字节间的约束关系, 也没有结合符号执行. 最重要的是, 它们并不是从并发角度出发进行的研究, 试图检测的缺陷也不包含并发缺陷类型. 本文从并发角度出发进行研究, 对 Solana 链上程序二进制文件中的并发缺陷进行动态分析检测. SolRaceFuzz 在稍微牺牲缺陷检测效率的情况下, 整体上比 FuzzDelSol 展现出更好的缺陷发现能力.

2 背景知识

本节将介绍预备知识以更好地理解本文内容, 包括 Solana 账本与交易模型、Solana 链上程序、模糊测试与符号执行的相关概念与知识, 这些概念间的整体关系如图 1 所示.

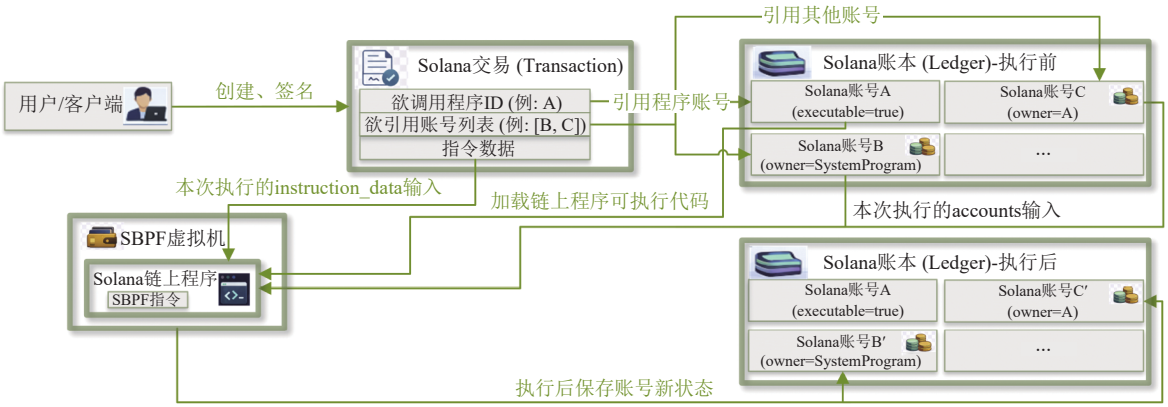

图 1 Solana 背景知识概念关系框图

2.1 Solana 账本与交易模型

区块链的本质是一种分布式账本技术, Solana 区块链也不例外. Solana 区块链账本中记录了 Solana 链上所有账号的数据信息, 账号是 Solana 区块链中数据的基本组织单位. 在 Solana 中, 账号具有多种不同的类型, 可以是存储用户资金的钱包账号, 也可以是存储链上程序的可执行程序账号, 还可以是存储程序状态的数据账号.

Solana 区块链运行时, 在逻辑上维护一个从账号地址到账号信息的映射:

$$s\colon address \Rightarrow (lamports, data, owner, executable, rent\_epoch),$$

该映射代表区块链账本在某时刻的状态. 其中, *address* 是账号的地址和唯一标识, 当账户是用户钱包或链上程序账号时, *address* 是 BIP44 标准生成的公钥; *lamports* 为整数, 代表账号以 Lamports 为单位所持有的加密数字货币余额; *data* 是变长字节数组, 代表账号中所包含的附加数据, 该数据可以是可执行程序本身或程序运行的状态数据, 由 Solana 内置程序或链上程序读取或写入, 当账号为用户钱包账户时, 该字段通常为空; *owner* 代表账号的所

有者, 其值等于另一个账号 *address* 字段的值; *executable* 是布尔值, 代表该账号 *data* 字段中的数据是可执行程序还是状态数据; *rent_epoch* 为整数, 代表该账号保持存在而被收取租金的下次 epoch 时间点.

在运行时, Solana 区块链通过分布式网络节点接收并处理用户交易, 各网络节点会转发交易给当前 slot 的领导者节点, 然后经过领导者节点的打包和广播, 再由所有网络节点各自验证、执行并进行共识投票. 成功执行的交易可改变前述的 Solana 账本. Solana 区块链上的一个交易可被形式化定义为如下三元组:

$$t = (program\_id, instruction\_accounts, instruction\_data),$$

其中, *program_id* 代表本次交易将调用的 Solana 内置程序或链上程序的地址, 对应区块链账本中 *address* 字段; *instruction_accounts* 包含该次交易执行中所涉及所有账号元信息, 其中每个账号元信息可以表示为三元组:

$$(address, is\_writable, is\_signer) \in instruction\_accounts,$$

其中, *address* 是该账号在区块链账本中的地址, *is_writable* 代表该账号在本次交易执行中是否需要被写入, *is_signer* 代表该账号是否使用数字签名签署了本次交易; *instruction_data* 是要传递给所调用程序的变长字节数组, 其具体含义和解析方法由被调用的不同 Solana 内置程序或链上程序自行定义. Solana 节点从区块链账本中读取交易所涉及账号的数据和待执行的 Solana 链上程序, 将这些信息传递给被调用的 Solana 链上程序, 而 Solana 链上程序在 Solana 虚拟机中逐条执行指令, 完成账号间转账、读写账号中包含的状态数据等账号读取与修改操作. 在被调用的 Solana 链上程序执行完毕后, Solana 验证器节点将经过 Solana 链上程序修改的账号数据存储为 Solana 区块链账本的新区块, 形成区块链账本的新状态.

为便于后文分析, 本文将 Solana 链上程序的执行过程抽象为一个状态转移系统. 在该系统中, 设 A 为所有可能账号地址的集合, N 为非负整数集, B={TRUE, FALSE}, 则区块链账本状态 s 可定义为:

$$s\colon A \to N\times\{0,1\}^{*}\times A\times B\times N,$$

分别对应前述 *lamports*、*data*、*owner*、*executable*、*rent_epoch*. 状态 s 构成区块链账本状态空间 S, 交易 t 构成区块链账本状态空间 T. 账本状态转移可定义为:

$$\Delta\colon S\times T \to S.$$

账本转移函数 $\Delta(s_0,t) \to s$ 描述单笔交易执行对账本的影响, 约定简化记作 $s_0 \to s$. 状态 s_0 经过 0 次或多次交易执行到达状态 s, 可记为 $s_0 \to^{*} s$.

基于该抽象, Solana 链上程序的并发行为可形式化为交易序列在状态空间上的可达性与等价性问题.

2.2 Solana 链上程序

链上程序 (或称智能合约、去中心化程序) 是部署并运行在区块链上的程序, 也是 Web 3.0 技术的重要概念. 在 Solana 中, 链上程序的可执行指令和程序运行时的状态数据都存储于区块链账号中. 值得注意的是, Solana 采用所谓“无状态”程序模型, 即程序可执行指令并不与程序状态数据存储在一起, 而是分别独立存放在不同账号中. 因此, 同一链上程序可能拥有多组不同的程序状态数据. 用户在调用链上程序时, 需要在交易中显式地指定这些状态数据所存放的位置 (即数据账号的地址), 链上程序将根据交易引用的账号来读取和修改程序状态.

Solana 链上程序当前可以使用 Rust 语言和 C 语言编写, 仅 Rust 语言是当前主要受官方支持的语言[16]. 图 2 给出了一份 Rust 语言编写的计数器程序示例, 该链上程序接收一个账号作为计数器状态账号, 并根据所传入指令数据的第 1 个字节来初始化计数器账号或增加其中的计数值.

Solana 链上程序在编译后是一份标准的 ELF 文件, ELF 格式是 Linux 系统下的可执行与可链接格式. 与普通可执行程序不同的是, Solana 链上程序是一份动态链接库文件, 因此不能主动执行, 需要由 Solana 区块链加载并调用其中名为 entrypoint 的导出函数. 除此以外, Solana 链上程序的可执行指令采用 SBPF 指令集, 这是一个 Solana 定制修改的 eBPF 指令集. eBPF 指令集最初设计用于为 Linux 内核编写网络数据包过滤器并安全地加载, 后来用途逐渐扩大至其非网络部分, 允许用户将自定义的安全、监控或性能分析程序动态加载到内核中运行. 由于它在 Linux 内核中的成功, eBPF 成为一个通用的 RISC (reduced instruction set computer) 指令集, 其核心思想 (如动态加载、高效执行、安全性验证等) 也被引入区块链领域, 从而被 Solana 加以修改形成 SBPF 指令集.

```rust
use borsh::{BorshDeserialize, BorshSerialize};
use solana_program::{
    account_info::{next_account_info, AccountInfo}, entrypoint, entrypoint::ProgramResult,
    msg, program_error::ProgramError, pubkey::Pubkey,
};
#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct CounterAccount {
    pub count: u32,
} // 计数器账户数据结构
entrypoint!(process_instruction);
fn process_instruction(program_id: &Pubkey, accounts: &[AccountInfo], instruction_data: &[u8]) ->
ProgramResult {
    let accounts_iter = &mut accounts.iter();
    let counter_account = next_account_info(accounts_iter)?;
    if counter_account.owner != program_id { // 验证计数器账号的所有者是当前程序
        return Err(ProgramError::InvalidAccountOwner);
    }
    match instruction_data[0] {
        0 => { // 初始化计数器
            let mut counter = CounterAccount { count: 0 };
            counter_account.realloc(borsh::object_length(&counter).unwrap(), false);
            counter.serialize(&mut &mut counter_account.data.borrow_mut()[..])?;
            msg!("Counter initialized!");
        }
        1 => { // 修改计数器
            let mut counter = CounterAccount::try_from_slice(&counter_account.data.borrow())?;
            counter.count += 1;
            counter.serialize(&mut &mut counter_account.data.borrow_mut()[..])?;
            msg!("New count: {}", counter.count);
        }
        _ => {}
    }
    Ok(())
}
```

图 2　Rust 语言编写的简单 Solana 链上程序源代码示例

2.3　模糊测试与符号执行

模糊测试是一种动态程序分析技术, 经典符号执行是一种静态程序分析技术, 而混合符号执行是一种结合静态和动态的程序分析技术[17].

模糊测试的核心思想是通过创建大量随机或半随机的输入数据, 来观察程序的行为是否会产生异常 (例如崩溃、内存泄漏、逻辑错误或其他不符合预期的结果). 模糊测试通过模糊输入来覆盖尽可能多的代码路径, 从而尽可能地发现程序中的潜在的问题[18]. 根据创建模糊输入的方法不同, 大体上可分为基于生成的模糊测试和基于变异的模糊测试. 基于变异的模糊测试是通过对交易指令库中已有的测试用例进行变异, 从而得到新的测试用例. 基于生成的测试用例则是通过引入一定的测试用例生成规则, 按照生成规则进行测试用例的生成[19]. 模糊测试具有简洁易用的优势, 可快速发现明显的漏洞, 但其随机性可能导致代码覆盖率有限, 对于复杂的条件检查可能难以满足. 因此在使用时常结合不同方法提高其有效性[17].

符号执行是一种基于抽象解释理论的系统化方法, 它将程序的输入表示为符号而非具体值, 在执行过程中收集程序状态值的符号表示, 在遇到条件和循环语句时处理不同的路径, 以探索程序可能的执行路径, 并得到每个抽象路径的输入约束条件和抽象输出结果. 通过对每个路径的所有约束关系进行分析和求解, 可以得到满足特定条件的程序输入[20]. 经典符号执行完全静态地解析程序中的所有路径并获得输出, 然而对于大型程序而言其路径复杂, 且可能包含无限递归和无限循环, 从而导致路径数无限, 因而不可能完全遍历所有路径. 2000 年后, 研究者提出多种结合动态执行的改进方案, 其中较为典型的有 Cadar 等人[21]提出的根据符号执行结果生成测试用例的执行生成测试 (execution-generated testing, EGT) 技术, 并在此基础上改进而来的 KLEE[22]、Sen[23]提出的将符号执行和具体执行同时执行的混合执行 (Concolic) 技术、Chipounov 等人[24]提出的选择性符号执行技术.

在实际应用中, 模糊测试和符号执行常相互结合, 以达成更好的程序分析效果. 本文结合这两种技术进行 Solana 链上程序的分析与探索, 以发现其中的并发缺陷问题.

3 基于模糊测试与符号执行的 Solana 链上程序并发缺陷检测方法

3.1 问题定义

为统一刻画本文的检测目标, 首先形式化定义并发缺陷检测的问题空间.

定义 1. 程序可达状态空间. 给定 Solana 链上程序 P 和初始区块链账本状态 s_0, 其状态空间定义为:

$$S_p = \{s \mid s_0 \rightarrow^* s\}$$

即在 s_0 下执行程序 P, 0 次或多次能够到达的状态集合.

定义 2. 交易空间. 给定程序 P, 其交易空间定义为:

$$T_p = \{t \mid \exists s \in S_p,\ \Delta(s,t) = s'\}$$

即所有在某一可达状态 s 下能够被程序 P 接受并执行的交易集合.

定义 3. 交易序列的执行. 给定初始区块链账本状态 s_0 和交易序列 $\tau = \langle t_1, t_2, \ldots, t_n \rangle$, 其执行结果定义为:

$$Exec(s_0, \tau) = \Delta(\ldots\Delta(\Delta(s_0, t_1), t_2)\Delta, t_n)$$

其中, Δ 为第 2.1 节定义的单交易状态转移函数.

定义 4. 交易序列的等价性. 对于给定状态 s, 若两个交易序列 τ_1 与 τ_2 满足:

$$Exec(s, \tau_1) = Exec(s, \tau_2)$$

则称 τ_1 与 τ_2 在状态 s 下具备等价性.

然后, 本文分别给出交易顺序依赖缺陷和时钟控制依赖缺陷的定义.

定义 5. 交易顺序依赖缺陷 (TOD). 给定一个 Solana 链上程序 P 和初始状态 s_0, 若存在可达状态 $s \in S_p$ 和两个交易 $t_1, t_2 \subseteq T_p$, 它们满足以下条件:

(1) $t_1.PID = P.PID \cap t_2.PID = P.PID$;

(2) 交易序列 $\tau_1 = \langle t_1, t_2 \rangle$, $\tau_2 = \langle t_2, t_1 \rangle$ 在状态 s 下不具备等价性.

则称该 Solana 链上程序 P 在状态 s 下具有交易顺序依赖缺陷, 可由交易 t_1 和 t_2 的并发执行触发. 设 $s_F = Exec(s, \langle t_1, t_2 \rangle)$, $s'_F = Exec(s, \langle t_2, t_1 \rangle)$, 当 s_F 和 s'_F 满足 $AccountsKeys(s_F) \neq AccountsKeys(s'_F)$ 时, 则称该缺陷导致账本具有不同账号集合. 当 s_F 和 s'_F 满足 $\exists acc \in s_F \cap acc \in s'_F,\ s_F[acc].data \neq s'_F[acc].data$ 时, 则称该缺陷导致账本具有不同数据. 当 s_F 和 s'_F 满足 $\exists acc \in s_F \cap acc \in s'_F,\ s_F[acc].lamports \neq s'_F[acc].lamports$ 时, 则称该缺陷导致区块链账本中的账号具有不同的金额.

定义 6. 时钟控制依赖缺陷 (CCD). 给定一个 Solana 链上程序 P=(PID, $instlist$) 和交易 t, PID 是该程序的区块链地址, $instlist$=$\{i \mid i$ 是 Solana 链上程序 P 所包含的 SBPF 指令$\}$, 且满足 $t.PID == P.PID$. 如果在状态 s 下执行交易 t 的过程中, 存在被执行的指令 $i \in instlist$ 和 $i_c \in instlist$, 它们满足以下条件:

(1) i_c 先于 i 执行;

(2) $(i_c.op = syscall.sol\_get\_clock\_sysvar \cap i_c.arg1 = addr_{\text{clock}}) \cup (i_c.op = memcpy \cap regValue(i_c.src) \in [addrof(SysvarClockAcc.data), addrof(SysvarClockAcc.data) + CLOCKLEN) \cap regValue(i_c.dst) = addr_{\text{clock}})$;

(3) $i.op \in \{JEQ, JGE, JGT, JSET, JNE, JSGT, JSGE, JLT, JLE, JSLT, JSLE, CALL\} \cap i.class = REG$;

(4) $regValue(i.src)$ 数据依赖于值 val 的任一部分, $addrof(val) \in [addr_{\text{clock}}, addr_{\text{clock}} + CLOCKLEN)$.

则称 Solana 链上程序 P 在指令 i 处存在时钟控制依赖缺陷, 可在状态 s 下由交易 t 的执行触发. 其中, $regValue(x)$ 代表编号为 x 的寄存器中存储的值, $addrof(v)$ 代表变量 v 在内存中的地址. $CLOCKLEN$ 是时钟系统变量在内存中的字节长度, 是 Solana 运行环境预定义的常数, 通常为 40.

以上两种缺陷都是 Solana 链上程序中的并发缺陷. 其中交易顺序依赖缺陷可由恶意 Solana 节点调换并发交易 t_1 和 t_2 的相对顺序来攻击利用; 时钟控制依赖缺陷可由恶意 Solana 节点故意推迟交易 t 的执行并选择执行结果对自己有利的时钟值来攻击利用.

根据以上定义, 本文所提检测方法欲解决的研究问题如下.

研究问题 1 (TOD 检测问题): 给定 Solana 链上程序 P 和初始区块链账本状态 s_0, 是否存在可到达状态 $s \in S_p$, 以及交易 $t_1, t_2 \subseteq T_p$, 可以触发 P 的交易顺序依赖缺陷?

研究问题 2 (CCD 检测问题): 给定 Solana 链上程序 P 和初始区块链账本状态 s_0, 是否存在可到达状态 $s \in S_p$, 以及交易 $t \subseteq T_p$, 可以触发 P 的时钟控制依赖缺陷?

3.2 Solana 链上程序的动态探索

本文基于模糊测试来动态探索 Solana 链上程序, 算法 1 展示了 Solana 链上程序缺陷检测算法的自动化探索总体过程. 要强调的是, 本方法对编译后的 SBPF 字节码直接进行测试, 不要求被测程序的源代码.

该算法在交易的生成方法上属于遗传算法的变种. 由于单纯的模糊测试可能无法通过一些难以满足的条件检查, 该方法还结合混合符号执行技术来满足这些条件检查, 以更好地探索 Solana 链上程序. 该算法的输入是待测的 Solana 链上程序 P, 输出是所检测到的交易顺序依赖缺陷和时钟控制依赖缺陷的报告信息.

算法 1. 基于模糊测试的 Solana 链上程序缺陷检测自动化探索.

输入: P: 待测程序;
输出: *TOD*: 检测到的交易顺序依赖缺陷; *CCD*: 检测到的时钟控制依赖缺陷.

```
1. global TOD ← ∅, CCD ← ∅
2. s_0 ← createInitialState(P)
3. S ← {s_0}, T ← createSeedTransactions(P, s_0)
4. while 终止条件未满足 do
5.    tx ← randomTransactionFromCorpus(T)
6.    if random() < K then
7.       tx.state ← selectStateFromCorpus(S)
8.    else
9.       (tx.usedAcc, tx.instData) ← mutateTx(tx.usedAcc, tx.instData)
10.   evalTransaction(tx, P)
11.   if fuzz stuck too long then
12.      txFromConcolic ← generateTxWithConcolic(tx, P)
13.      evalTransaction(txFromConcolic, P)
14.   return (TOD, CCD)
15. function evalTransaction(tx, P)
16.   (reverted, interesting, newState) ← execuateInSvm(tx.state, tx, P)
17.   if reverted == FALSE then
18.      S ← S ∪ {newState}
19.   if checkTODOracle(state.prevState, tx.prevTx, tx, newState, P) then
20.      TOD ← TOD ∪ {(tx.prevTx, tx, newState, P)}
21.   if interesting then
22.      T ← T ∪ {tx}
23.   (hasCCD, ccdBuggyInst) ← checkCCDOracle(tx)
24.   if hasCCD then
25.      CCD ← CCD ∪ {(tx, ccdBuggyInst)}
```

算法 1 接收待测程序 P 作为输入, 首先进入初始化阶段, 创建 *TOD*、*CCD* 空集合, 二者分别代表所检测到的交易顺序依赖缺陷和时钟控制依赖缺陷集合, 然后该算法根据待测程序 P 创建初始的区块链账本状态 s_0 并插入

状态集合 S 中, 接着基于状态 s_0 及待测程序 P 创建种子交易并插入交易集合 T 中 (第 1–3 行). 随后进入循环探索阶段, 直到终止条件满足才退出 (第 4 行), 此处的终止条件可由使用者自行定义, 通常为时间预算是否超出或发现缺陷是否足够多 (本文在实验部分使用前者). 在循环探索阶段, 每次从集合 T 中按顺序轮询选择其中的一个交易 tx, 然后通过函数 *random*() 获取一个范围在 [0, 1) 的随机值, 根据随机值以概率 K 随机替换交易 tx 的起始状态为 S 中随机选择的状态, 否则对交易所使用的输入账号和输入的指令数据进行变异 (第 6–9 行), 此时交易 tx 已经与最初从集合 T 中选取的交易不同, 因此对交易进行评估 (第 10 行). 当发现模糊测试太久没有探索到新的程序区域时, 调用符号执行引擎对 tx 进行符号执行, 该符号执行引擎在执行过程中维护输入 (即 *tx.instData*) 的符号变量, 当遇到条件跳转指令的符号约束时便尝试取反并求解, 以生成新的可探索未探索分支的交易输入 *txFromConcolic*, 然后对符号执行生成的新交易也进行评估 (第 11–13 行). 算法中的 K 为用户预定义的常数.

(1) 交易评估函数

函数 *evalTransaction* 在得到新的交易时对其进行评估, 具体有 3 个方面, 第一是判断该交易执行是否可生成新的区块链账本状态; 第二是判断该交易是否“有趣”; 第三是判断该交易是否能触发缺陷.

交易评估首先在 Solana 虚拟机中执行交易 tx, 基于区块链账本状态 *tx.state,* 该交易被待测程序 P 处理 (第 16 行). 若该交易可以生成新的区块链账本状态, 则将新状态保存到集合 S (第 17、18 行), 以便在随后的交易变异中使用. 若该交易的执行可探索到从未执行的 SBPF 程序指令, 或缩小还未满足的条件比较指令的历史最小距离, 则认为该交易“有趣”. 由于每个条件比较指令具有真和假两个结果, 本文分别计算真分支和假分支不满足时的历史最小距离, 分别记为 $Distance_{\text{True}}$ 和 $Distance_{\text{False}}$, 计算公式如表 1 所示. 一个“有趣”的交易将有助于探索程序中未探索到的区域, 否则该交易的作用可被其他交易所代替. 对于“有趣”的交易, 该算法将其保存到集合 T 中 (第 21、22 行), 以便在随后的交易变异中使用. 判断该交易是否能够触发 Solana 链上程序的缺陷则是通过调用具体缺陷类型的检测准则, 这里分别调用交易顺序依赖缺陷的检测准则 (第 19、20 行) 和时钟控制依赖缺陷的检测准则 (第 23–25 行). 需要注意的是, 只有在该交易成功创建了新的区块链账本状态时, 我们才会尝试检测交易顺序依赖缺陷, 而会对所生成的任何交易尝试检测时钟控制依赖缺陷. 这是因为交易顺序依赖缺陷的本质是并发交易的顺序不确定性导致区块链账本执行后状态的不确定性, 因此只有在区块链账本状态成功发生变更时才有可能触发交易顺序依赖缺陷, 而时钟控制依赖缺陷在任意的交易执行中都可能触发. 第 3.3 和 3.4 节将会介绍两种缺陷的具体检测准则.

表 1　计算 SBPF 条件比较指令分支满足距离的公式

SBPF Op	$Distance_{\text{True}}$	$Distance_{\text{False}}$
==	abs(*dst* – *src*)	*bool2int*(*dst* == *src*)
!=	*bool2int*(*dst* == *src*)	abs(*dst* – *src*)
<	*dst* – *src* + 1	*src* – *dst*
<=	*dst* – *src*	*src* – *dst* + 1
>	*src* – *dst* + 1	*dst* – *src*
>=	*src* – *dst*	*dst* – *src* + 1
&	*cntmindt*(*dst*, *src*)	*cntbothsetbit*(*dst*, *src*)

在表 1 中, abs() 是绝对值函数; *bool2int*(*b*) 在 *b* 为 TRUE 时返回 1, 否则为 0; *cntmindt*(*dst*, *src*) 逐个遍历 *dst* 和 *src* 中相同位置比特位, 并计数 *dst* 或 *src* 中该比特位为 0 的条件满足个数的最小值; *cntbothsetbit*(*dst*, *src*) 计算 *dst* 和 *src* 中相同位置比特位都为 1 的位个数.

(2) 符号执行模块

本文方法使用混合符号执行方法辅助模糊探索进行更深入的探索. 为此, 本文基于 Z3 求解器实现了支持 SBPF 指令集的符号执行引擎. 根据混合符号执行方法[23], 该模块同时维护程序输入数据和程序状态数据的一份具体值与一份符号值, 输入具体值来自模糊测试之前得到的交易, 对于每个 SBPF 指令, 同时更新维护程序状态的具体值与符号值. 当遇到条件跳转指令且对当前具体值不满足的另一分支未探索过时, 根据符号值, 取反相应的符号约束条件并进行求解. 对于循环而言, 其在 SBPF 汇编层面表现为反向跳回的条件跳转指令, 我们未与普通的条

件判断进行特殊区分, 循环编译后产生的条件跳转指令两分支分别对应退出循环与继续循环, 根据前述的处理方法, 应当至少探索到从未进入循环与进入循环体执行 1 次这 2 种情况. 混合符号执行一定程度上缓解了路径爆炸问题, 因为其仅在当前具体值输入的程序执行路径上取反条件约束并求解, 另外本文限制每个条件分支语句的 2 个分支仅分别取反求解一次.

SBPF 具有 12 个寄存器 (含 10 个通用目的寄存器), 本文符号化了除 R11 外的所有寄存器 (R11 用于记录 PC, 且不可被链上程序直接访问), 并在发生函数调用与返回时维护一个符号调用栈以正确跟踪跨函数调用. 对于内存访问, 符号执行引擎在字节级别建模内存并读写符号值, 根据 SBPF 规范, 其内存划分为 4 个区域: 程序只读区、栈区、堆区、输入区, 虚拟地址空间分别起始于 0x100000000、0x200000000、0x300000000、0x400000000.

3.3 交易顺序依赖缺陷的检测准则

本节介绍交易顺序依赖缺陷的检测准则. 交易顺序依赖缺陷是并发交易的顺序不确定性导致的 Solana 链上程序缺陷, 区块链网络中的恶意节点可能干涉并发提交交易的处理顺序, 以选择有利于自身的执行结果. 一个足够健壮的 Solana 链上程序应当能够在并发交易的任意处理顺序下都达成相同的区块链账本状态.

对并发交易进行顺序重排, 然后检测执行结果是否相同, 这是一个很直观的检测思路. 但随之而来的问题是, 这种做法要求重排后重新进行交易重放执行, 即使本文仅针对两个并发交易进行重排, 对于具有规模 n 的交易池也需要进行 $2\times\mathrm{C}_n^2$ 次重排后的交易重放执行. 交易执行需要收集区块链账本前状态、创建 SVM 虚拟机、收集区块链账本后状态等多个复杂步骤, 具有较大的性能开销. 为减少交易执行的次数, 提高交易顺序依赖缺陷的检测效率, 本文的一个观察是, 只有当两个交易的输入账号有交集且其中至少一个为可写入权限时, 本节的检测准则使用命题 1 作为必要条件进行剪枝.

命题 1. 账号冲突必要条件. 基于定义 4 中的顺序等价性, 设交易 t_1, t_2, 若满足:

$$(UseAccounts(t_1)\cap WriteAccounts(t_2))\cup(UseAccounts(t_2)\cap WriteAccounts(t_1))=\varnothing,$$

则两交易的顺序交换不可能构成交易顺序依赖缺陷. 其中, $UseAccounts(t)$ 代表交易 t 执行中所使用的所有账户地址集合, $WriteAccounts(t)$ 代表在交易 t 执行中可写入的账户地址集合.

证明: 根据定义 3, 交易执行仅通过状态转移函数 Δ 修改账本中的账号状态. 在 t_1, t_2 无共享可写账户的情况下, 可知两交易对账本的影响是相互独立的, 因此满足交换律, 在任意账本状态 $s\in S_p$ 下均有:

$$Exec(s,\langle t_1,t_2\rangle)=Exec(s,\langle t_2,t_1\rangle),$$

满足定义 4 中的交易序列等价性. 进而根据定义 5, 不满足交易顺序依赖缺陷的定义.

算法 2 展示了完整的交易顺序依赖缺陷的检测准则. 该算法选择只检测两个并发交易的顺序重排是否可触发交易顺序依赖缺陷, 因为两个并发交易是交易顺序依赖的最小单元, 多个并发交易的顺序依赖可由两个并发交易的顺序依赖扩展而来.

算法 2. Solana 链上程序交易顺序依赖缺陷的检测准则.

输入: *state*: 区块链账本状态; *firstTx*、*secondTx*: 用于检测的 2 个交易; *originalFinalState*: 在 *state* 按顺序先后执行 *firstTx* 和 *secondTx* 后的状态;

输出: *existTOD*: 交易 *firstTx* 和交易 *secondTx* 的并发提交是否会在状态 *state* 触发交易顺序依赖缺陷.

```
1. function checkTODOracle(state,firstTx,secondTx,originalFinalState,P)
2.     accountsMayRace ← FALSE
3.     foreach a1 ∈ firstTx.usedAcc do
4.         foreach a2 ∈ secondTx.usedAcc do
5.             if a1.addr == a2.addr and (a1.writable or a2.writable) then
6.                 accountsMayRace ← TRUE
7.     if accountsMayRace == FALSE then
```

8. **return** FALSE
9. $reorderInterState \leftarrow executeInSvm(state, secondTx, P)$
10. $reorderFinalState \leftarrow executeInSvm(reorderInterState, firstTx, P)$
11. **if** $accNum(originalFinalState) \neq accNum(reorderFinalState)$ **then**
12. **return** TRUE
13. **foreach** $a_1 \in originalFinalState$ **do**
14. $a_2 \leftarrow reorderFinalState[a_1.addr]$
15. **if** $a_1.lamports \neq a_2.lamports$ **then**
16. **return** TRUE
17. **if** $a_1.data.length \neq a_2.data.length$ or $!bytesEq(a_1.data, a_2.data)$ **then**
18. **return** TRUE
19. **return** FALSE

算法 2 首先根据命题 1 进行剪枝 (第 2–8 行), 若不存在这样的重叠账号, 则两个交易不可能触发交易顺序依赖缺陷, 因此直接返回 FALSE 以快速完成判定. 若存在这样的重叠账号, 则进行进一步判定. 该算法通过调换顺序进行交易重排, 首先执行交易 t_2, 再执行交易 t_1, 得到重排后的最终区块链账本状态 *reorderFinalState* (第 9、10 行), 然后对重排前的最终区块链账本状态 *originalFinalState* 和重排后的最终区块链账本状态 *reorderFinalState* 进行一系列比较: 首先检测两个区块链账本状态所包含的账号数量是否相同, 若数量不同, 则两个账本状态必定不同 (第 11、12 行), 这样的预判方法可以加速判定过程, 在这种情况下避免遍历整个区块链账本状态. 当两个区块链账本状态具有相同的账号数量时, 才对包含的账号进行逐一比较, 分别检测每个账号的 *lamports* 余额是否相同, 账号的数据长度和数据内容是否相同, 发现任一不同则认为发生了交易顺序依赖缺陷 (第 13–18 行). 若全部检测后仍然没有发现不同, 则认为没有发生交易顺序依赖缺陷, 并结束该缺陷检测准则的执行流程 (第 19 行).

3.4 时钟控制依赖缺陷的检测准则

时钟控制依赖缺陷是 Solana 链上程序执行过程对区块链集群给出的时钟系统变量具有控制依赖导致的缺陷, 区块链网络中的恶意节点可合法地故意推迟交易的处理, 并选择有利于自身的时钟值. 因此, 若程序的条件控制指令依赖于时钟系统变量数据, 则执行结果可被恶意攻击者控制, 以选择有利于攻击者的执行结果.

在 Solana 链上程序中, 时钟系统变量的获取共有两种方法, 一种是通过将时钟系统变量的特殊账号作为交易的输入并在执行中读取账号数据字段获取, 另一种通过系统调用直接获取, 示例代码片段如图 3 所示.

```
/// 方法 1：通过交易传入的 Account 读取系统变量 (需交易发起方显式传入 Sysvar account)
fn get_clock_via_account(clock_account: &AccountInfo) -> Result<Clock, ProgramError> {
    // 验证交易传入的 Account 是否为 Clock sysvar account
    if !clock_account.key.eq(&solana_program::sysvar::clock::ID) {
        msg!("Error: Invalid Clock sysvar account");
        return Err(ProgramError::InvalidArgument);
    }
    // 从 Account 数据反序列化 Clock
    let clock = Clock::from_account_info(clock_account)?;
    Ok(clock)
}
/// 方法 2: 直接通过 SysvarName::get()读取系统变量 (无需显式传入 Account)
fn get_clock_from_syscall() -> Result<Clock, ProgramError> {
    let clock = Clock::get()?;
    Ok(clock)
}
```

图 3 缺陷检测需支持的两种 Solana 时钟系统变量的获取方法代码示例

两种方法均通过 Solana 运行时访问数据, 区别为一个间接获取一个直接获取, 用途和最终结果没有明显差异. 为尽可能完整地检测时钟控制依赖缺陷, 需要对两种时钟系统变量的获取方法均进行支持.

算法 3 展示了时钟控制依赖缺陷的检测准则. 算法 3 基于污点分析对时钟控制依赖进行检测, 主要分析过程是在交易执行过程中通过 SVM 虚拟机的回调函数完成, 算法共设置了两个回调函数 *SvmCallbackBeforeTxExecute* 和 *SvmCallbackOnTxInstStep*.

算法 3. Solana 链上程序时钟控制依赖缺陷的检测准则.

输入: *state*: 区块链账本状态; *firstTx*、*secondTx*: 用于检测的两个交易; *originalFinalState*: 在 *state* 按顺序先后执行 *firstTx* 和 *secondTx* 后的状态;
输出: *existTOD*: 交易 *firstTx* 和交易 *secondTx* 的并发提交是否会在状态 *state* 触发交易顺序依赖缺陷.

```
1. global tracingTx
2. global registersTaint, memoryTaint
3. global buggyInstList
4. function SvmCallbackBeforeTxExecute(tx)
5.     tracingTx ← tx
6.     registersTaint ← ∅, memoryTaint ← ∅
7. function SvmCallbackOnTxInstStep(inst)
8.     if inst.op is memload then
9.         registersTaint[inst.dst] ← memoryTaint[inst.src]
10.    if inst.op is memstore then
11.        memoryTaint[inst.dst] ← registersTaint[inst.src]
12.    if inst.op is alu then
13.        registersTaint[inst.dst] ← registersTaint[inst.src]
14.    if inst.op is syscall.sol_ get_ clock_ sysvar then
15.        for addr in [inst.arg1, inst.arg1 + CLOCKLEN] do
16.            memoryTaint[addr] ← TRUE
17.    if inst.op is compareKey and key_1 ∈ tracingTx and key_2 == CLOCKID then
18.        accDataAddr ← lookupAccDataAddr(tracingTx, addressof(key_1))
19.        for addr in [accDataAddr, accDataAddr + CLOCKLEN] do
20.            memoryTaint[addr] ← TRUE
21.    if inst.op is condJump and (registersTaint[dst] or registersTaint[src]) then
22.        buggyInstList ← buggyInstList ∪ {inst.pc}
23. function checkCCDOracle(tx)
24.    result ← buggyInstList
25.    buggyInstList ← ∅
26.    return (result.length > 0, result)
```

回调函数 *SvmCallbackBeforeTxExecute* 在虚拟机准备开始执行交易时被调用, 此时算法记录该交易信息用于随后的污点分析, 并初始化并重置相关污点标志变量 (第 4–6 行). 回调函数 *SvmCallbackOnTxInstStep* 在虚拟机每次准备执行下一条 SBPF 指令时进行调用, 此时算法判断 SBPF 指令的类型, 并分别执行不同的操作. 若指令是内存读写指令, 则在寄存器与内存地址间进行污点标记的传播 (第 8–11 行). 若指令是算术运算指令, 则在寄存器间进行污点标记的传播 (第 12、13 行). 若指令是系统调用, 且系统调用的类型是获取时钟系统变量 (系统调用名称为 *sol_get_clock_sysvar*), 则为该系统变量写入结果的内存区域引入污点 (第 14–16 行). 若指令正在比较账号地址且源地址位于交易输入, 同时目标地址为时钟系统变量账号地址, 则引入污点源, 为该输入账号的数据区域内存地

址引入污点 (第 17–20 行). 若指令是条件跳转指令, 且用作比较的寄存器具有污点标记, 此时发现污点汇聚点, 视为发现时钟控制依赖, 并记录当前指令的信息 (第 21、22 行). 在交易执行结束后, 调用函数 *checkCCDOracle* 汇总返回执行过程中发现的所有指令信息 (第 23–26 行).

3.5 区块链账本模拟方法

与普通传统程序相比, Solana 链上程序的一个特点就是, 它的状态数据保存在区块链账本上, 执行过程中的读写操作也是针对区块链账本上的账号数据进行读写, 执行完成后的状态也是保存回区块链账本. Solana 的区块链账本中具有一些特殊账号, 例如系统变量账号、程序派生账号. 这些账号的地址或数据具有一定的特殊规则. 如果只是简单地随机生成, 那么所生成的账号大多是无效的, 从而影响探索 Solana 链上程序的充分性. 因此, 本文进一步提出对这些特殊账号做针对性的处理: 对系统变量账号的数据基于预定义的结构和规则进行生成, 对程序派生地址的创建种子进行提取再针对性创建程序派生账号. 通过这些方法能够解决该问题, 能够更充分地探索 Solana 链上程序并检测其中的缺陷.

3.5.1 系统变量的模拟

系统变量 (Sysvar) 是一系列 Solana 链上程序可访问的区块链集群状态数据, 这些数据一般具有预定义的特殊区块链账号地址. 对于 Solana 区块链, 当前最新版本共具有 11 个系统变量, 如表 2 所示.

表 2 需要模拟的 Solana 区块链账本中系统变量及其模拟时约束条件

系统变量名	更新间隔	约束字段名与类型	约束条件
Clock	每slot	slot (u64)	slot ⩾ slot_{old}
		epoch_start_timestamp (i64)	epoch_start_timestamp ⩾ $\text{epoch\_start\_timestamp}_{\text{old}}$
		epoch (i64)	epoch ⩾ $\text{epoch}_{\text{old}}$
		leader_schedule_epoch (u64)	leader_schedule_epoch ⩾ epoch
		unix_timestamp (i64)	unix_timestamp ⩾ $\text{unix\_timestamp}_{\text{old}}$ 且unix_timestamp ⩾ epoch_start_timestamp
EpochRewards	每epoch	—	—
EpochSchedule	静态	slots_per_epoch (u64)	(static) 43 200
		leader_schedule_slot_offset (u64)	(static) 43 200
Fees	每slot	lamports_per_signature (u64)	(static) 0
Instructions	每交易	无需模拟	无需模拟
LastRestartSlot	每次重启	last_restart_slot (u64)	last_restart_slot ⩾ $\text{last\_restart\_slot}_{\text{old}}$
RecentBlockhashes	每slot	—	empty
Rent	静态	lamports_per_byte_year (u64)	(static) $10^7 \times 365 / 2^{20}$
		exemption_threshold (f64)	(static) 2.0
SlotHashes	每slot	无需模拟	无需模拟
SlotHistory	每slot	无需模拟	无需模拟
StakeHistory	每epoch	无需模拟	无需模拟

在这些系统变量中, Instructions 是最特殊的一个, 不存在于区块链账本中, 也没有相应的区块链地址. 而 SlotHashes、SlotHistory 和 StakeHistory 这 3 者仅可通过验证器节点 RPC 方法访问而无法在链上程序中访问, 所以也无需进行模拟. 总结而言, 共有 7 个系统变量需要在 Solana 链上程序的探索中进行模拟. 然而, 这些系统变量的值如何设定是一个问题. 如果仅提供零值或者随机生成显然是不高效的, 且有些组合在实际中是不可能出现的 (例如当前 slot 的时间戳不可能小于当前 epoch 的开始时间戳), 因此本文根据这些系统变量的功能特点与语义得到一系列约束条件. 表 2 的最后两列也进一步展示了本文对需要模拟的 7 个系统变量所设定的约束条件. 这些约束条件是根据各系统变量及其字段的语义设定, 且参考了当前 Solana 主网的运行参数设定.

这些系统变量约束主要在模糊测试阶段发挥作用, 确保所生成系统变量不违反基本的语义合理性.

3.5.2 程序派生地址的种子提取与账号创建

程序派生地址 (program derived address, PDA) 是从 Solana 链上程序的地址确定的派生, 且格式上与标准公钥相似的地址, 但是没有相关联的私钥. 其原理为: Solana 密钥对是指向 Ed25519 密码学曲线的一个点, 因此计算一个偏离 Ed25519 曲线的点, 就可得到格式上像是标准公钥但其实不存在对应私钥的地址, 称作程序派生地址. 这意味着没有外部用户可以为这个地址通过数字签名算法实在地生成有效的签名. 然而, Solana 运行时从编程角度允许 Solana 链上程序虚拟地为该地址“签名”而无需私钥, 也就是说, 虽然没有真正生成数字签名, 但 Solana 运行时在检验到该地址确实从该链上程序派生之后, 虚拟地将该 PDA 账号的 *is_signer* 字段设置为 TRUE. 由于 PDA 从密码学原理上就不存在相关联的私钥, 因此也避免了私钥泄漏的风险, 只能通过上述机制进行“签名”.

生成 PDA 需为其指定程序地址和一组种子, 种子是不超过 32 字节的一串二进制字节, 可为静态常量或动态输入. 同一程序可拥有多个程序派生地址, 不同的种子组合生成不同的程序派生地址, 而给定相同的种子和程序地址, 所生成的程序派生地址总是不变. 一个程序派生地址可以用作创建一个 Solana 账号时的地址, 作为一种存储、映射和获取 Solana 链上程序状态的方法. 可以简单地认为, 程序派生地址是一种为区块链上创建类似哈希表结构的方法, (链上程序地址, 种子)-账号数据形成了类似 key-value 的关系.

PDA 为 Solana 链上程序的动态探索带来了挑战. 这是由于 Solana 在设计上要求交易开始执行前就显式指定用到的所有账号地址, 即使 PDA 对应的账号还不存在时, 也需要交易发起者预先计算好 PDA 并传递给交易. 若传入的不是合法的 PDA, 则通常会导致交易执行失败.

由于本文方法直接在 Solana 链上程序的二进制文件上进行分析和测试, 通常没有可用的源代码和文档说明这一信息. 为解决这一挑战, 本文发现虽然交易会传入已经计算好的账号地址, 但 Solana 链上程序在使用程序派生账号时往往还会重新生成一次程序派生地址以对比验证交易传入的账号地址是正确的程序派生地址, 而在链上计算程序派生地址就意味着会进行相关的系统调用, 此外, 进行 CPI 调用时也会传入账号的种子, 并发起系统调用. 因此本文得出以下观察.

● 观察 1. Solana 链上程序使用程序派生地址时, 往往伴随着系统调用 sol_invoke_signed_rust、sol_invoke_signed_c、sol_create_program_address 和 sol_try_find_program_address, 它们的参数均携带着用于生成程序派生地址的种子组合.

本文方法监控交易执行中的系统调用, 若发现执行以上 4 个系统调用, 则提取其参数中的种子组合信息, 并进行记录. 程序派生账号生成模块将根据所记录的种子组合信息生成程序派生账号, 以供随后的交易使用.

虽然通过监控系统调用可以简单地获得当次执行生成程序派生地址所使用的种子组合, 但是如上文所述, 种子可以是静态编码的常量, 也可以来自运行时的动态输入. 若仅记录当次执行的种子信息, 输入一旦变化就可能引起种子变化. 为识别出种子组合中的动态输入部分, 本文通过观察实际的 Solana 链上程序, 得出观察 2.

● 观察 2. Solana 链上程序使用程序派生地址的典型种子模式是 (*prefix*, *pubkey*,…, *bump*). 其中, *prefix* 是静态编码的字符串常量, *pubkey*,…是存在于区块链账本上的一个或多个账号的公钥, *bump* 是用于保证程序派生地址偏离 Ed25519 曲线的额外种子.

综上所述, 本文跟踪账号的公钥地址数据在交易执行中的数据流, 当发现执行 PDA 相关系统调用时, 对所使用的种子进行数据来源判别, 若其来源于公钥地址, 则标记为动态输入, 否则, 标记为静态数据并记录其字节数据. 而后, 程序派生账号生成器使用所记录的种子为区块链账本中的所有钱包账号生成该程序的程序派生账号.

3.6 区块链交易生成方法

Solana 链上程序的执行入口点在设计上接收 3 个参数: 输入账号、指令数据、程序 ID. 其中, 指令数据是在交易中指定的一个结构不透明的变长字节数组, 其结构与意义完全由链上程序本身自行定义和解析.

由于 Solana 限制每个交易必须可编码在单个网络数据包内, 而单个数据包受到网络数据链路层 MTU (最大传输单元) 的限制, Solana 指令数据的理论长度为 1 492 字节. 进一步地, 可生成空间总规模为 256^{1492}. 如果完全进行随机测试, 其中大量的指令数据代表了无效或重复的程序路径, 这导致探索效率低下.

本文在算法 1 中已结合符号执行, 针对程序在执行中遇到的条件指令进行约束求解来生成指令数据, 这在一定程度上缓解了该问题. 但是, 符号执行技术本身具有路径爆炸、复杂约束难求解、内存模型难以完全建模等局限性, 这导致符号执行的加入仍未能完全解决这一问题.

为此, 本文观察到 Solana 链上程序通常提供了 IDL 信息, 且可基于 IDL 信息构建指令数据的抽象语法树结构, 本文使用 IDL 信息指导构建有效的指令数据信息. 基于 IDL 信息构建的抽象语法树, 可指导解决指令数据的生成长度、各字节数据的有效范围、字节之间的约束关系等问题. 表 3 展示了 Solana IDL 可表达出的数据类型清单, 表 3 中的“变长”表示该数据类型在运行时具有动态长度, 可以动态改变自身的字节长度; “定长”表示固定长度, 在运行时不能改变自身的字节长度, 但仍可以由于内嵌数据类型 T 的不同而静态地具有不同长度, 其长度在编译时确定.

表 3　Solana IDL 支持表达的数据类型清单

数据类型	解释	字节长度
bool	逻辑布尔值	1
u8/i8	无符号/有符号8位整数	1
u16/i16	无符号/有符号16位整数	2
u32/i32	无符号/有符号32位整数	4
f32	32位浮点数	4
u64/i64	无符号/有符号64位整数	8
f64	64位浮点数	8
Bytes	字节数组	变长 (4字节表示数组长度 + 字节数据)
String	动态字符串	变长 (4字节表示字符串长度 + 字符数据)
Pubkey	账号公钥地址	32
Array<*T*>	包含类型为*T*元素的数组	定长 (数组长度 × *T*的长度)
Vec<*T*>	包含类型为*T*元素的动态数组	变长 (4字节表示数组长度 + 数组长度 × *T*的长度)
Option<*T*>	包含类型为*T*的元素的可空值	定长 (1字节表示是否为空 + 内嵌值数据)
DefinedStruct{…}	用户定义的结构体	取决于用户定义
DefinedEnum{…}	用户定义的枚举量	取决于用户定义

此外, Solana 链上程序对其接受的输入账号往往也具有一定的约束, 这些约束在程序执行过程中进行检查, 违反约束将导致交易执行失败. 尽管模拟中的区块链账本规模较小, 且区块链中的账号通常不会太多, 但通过多次尝试也可以较快地找到有效的输入账号组合, 因此充分利用 IDL 信息中对输入账号的约束信息也可加速构建有效的输入账号信息.

Solana 链上程序的 IDL 信息可视为三元组 $IDL = (I, T, A)$, 其中 I 代表该链上程序所支持的指令的集合, T 代表该链上程序自定义数据类型的集合, A 代表该链上程序所期待和解析的账号数据结构.

指令信息 $i = (i_{\text{Disc}}, i_{\text{Accs}}, i_{\text{Args}}) \in I$, 其中 i_{Disc} 代表该指令的标识符, 为 8 字节长的哈希值; i_{Accs} 代表该指令接收的输入账号信息; i_{Args} 代表该指令接收的参数信息; $accMeta = (acc_{\text{Name}}, acc_{\text{writable}}, acc_{\text{signed}}) \in i_{\text{Accs}}$, $arg = (arg_{\text{Name}}, arg_{\text{TypeName}}) \in i_{\text{Args}}$, 这二者代表该指令接收的参数信息.

自定义数据类型 $t = (type_{\text{Name}}, type_{\text{Kind}}, type_{\text{Def}}) \in T$, $type_{\text{Name}}$ 为数据类型名称字符串, 与指令信息 i 中的 arg_{TypeName} 对应, $type_{\text{Kind}} \in$ {STRUCT, ENUM}. 当 $type_{\text{Kind}}$ 为 STRUCT 时, $typeField = (field_{\text{Name}}, field_{\text{TypeName}}) \in type_{\text{Def}}$; 当 $type_{\text{Kind}}$ 为 ENUM 时, $enumVariant = (variant_{\text{Name}}, variant_{\text{Fields}}) \in type_{\text{Def}}$, $variantField = (field_{\text{Name}}, field_{\text{TypeName}}) \in variant_{\text{Fields}}$, $field_{\text{TypeName}}$ 的有效值如表 3 第 1 列所示.

账号数据结构 $a = (acc_{\text{Name}}, acct_{\text{Def}}) \in A$, 其中 acc_{Name} 为账号的名称字符串, 与指令信息 i 中的 acc_{Name} 对应, $typeField = (field_{\text{Name}}, field_{\text{TypeName}}) \in acct_{\text{Def}}$.

本节将主要关注 Solana 链上程序的 IDL 信息中的 I 和 T, 也即指令信息和定义类型信息, 利用这两个信息, 可以有效生成合法结构的交易指令数据和交易输入账号. 接下来分别详细介绍从 IDL 生成二者的具体方法.

3.6.1 基于 AST 的交易指令数据生成方法

Solana 交易除了指定所调用的链上程序和所使用的账号外, 还包含一段长度可变化的字节数组, 称为指令数据. 指令数据的含义完全由链上程序自行定义, 不同的链上程序定义的指令数据的结构完全不同. 因此, 本文利用 IDL 信息中对指令数据的语法结构描述, 构建指令数据的抽象语法树结构, 以促进生成合法的指令数据.

IDL 中的数据类型可分为基本数据类型和复合数据类型. 此处的基本数据类型指的是本身作为独立类型、定义变量时不依赖其他数据类型、不可再分解的数据类型, 复合数据类型则不是.

定义基本数据类型为 T_{basic} = {bool, u8, i8, u16, i16, u32, i32, f32, u64, i64, f64, Bytes, String, Pubkey}.

定义复合数据类型为 T_{nested} = {Array, Vec, Option, DefinedStruct, DefinedEnum}.

本方法在抽象语法树中使用叶子节点表示基本数据类型, 使用子树表示复合数据类型及其依赖的子类型. 定义叶子节点 $N_{leaf} = (t, val)$, 其中 t 代表基本数据类型的具体种类, 此处 $t \in T_{basic}$; val 为具体值 (例如, t 为 bool 时, val 为 TRUE 或 FALSE 之一). 定义以非叶子节点为根的子树 $N = (t, val, children)$. 其中 t 代表复合数据类型的具体种类, 此处 $t \in T_{nested}$; val 为该复合类型的额外数据 (例如, Array、Vec 携带元素个数、Option 是否携带子元素、DefinedStruct 字段个数); $children$ 代表其依赖的子类型, 可为叶子节点或子树.

当模糊测试开始时, 本方法首先基于 IDL 信息为每种指令生成其对应的 AST, 附加指令的标识符信息后, 作为种子输入, 并逐个评估. val 字段在初始种子中取类型的零值 (例如, bool 取 FALSE, u8 取 0, f32 取 0.0, String 取空串). AST 中每个节点的 t 字段可指示类型对应的字节长度, 具体如表 3 最后一列所示. 然后根据 AST 计算对应的指令数据字节, 具体方法为: 首先是 8 字节的指令标识符信息. 然后从 AST 根节点开始先序遍历所有节点: 1) 若遇到节点代表基本数据类型, 则直接将 val 字段转换为对应的字节; 2) 若遇到节点代表复合数据类型, 则先写入复合数据类型本身所需要的数据 val 字段, 再按顺序递归访问 $children$ 字段中的所有节点. 最后, 当 AST 遍历完成, 就构造出一段符合 AST 所描述语法结构的指令数据.

在模糊测试循环中, 每当随机选取交易并进行变异时, 不直接改变字节, 而是通过遍历 AST, 随机选择 AST 中的某个节点, 并根据节点的 t 字段相应改变 val 字段, 而后根据 AST 重新计算交易所使用的指令数据字节.

通过以上方法, 可利用 IDL 中关于指令数据的语法结构信息, 以确保生成有效语法结构的交易指令数据.

3.6.2 基于约束限制的交易输入账号生成方法

Solana 链上程序的 IDL 信息主要包含关于交易输入账号的 3 个约束信息: 账号名称、是否需要签名、是否需要账号的写入权限. 下面分别介绍这三者可提供的不同角度的约束信息.

账号名称是对账号类型的简短文字描述, 如图 4(a) 第 1 列所示. 根据该信息可推断出链上程序所期待的账号类型, 并为交易中输入账号的地址有效范围提供约束. 然而, 其蕴含的一个挑战为文字描述不精确, 这表现为重复性和多样性. 例如图 4 中 systemProgram 和 system_program 实际上均代表系统程序. 事实上, 本文还发现少量链上程序使用 system 代表系统程序, Token2022 程序甚至使用 4 种不同的名称 (“token2022Program”“tokenProgram2022” “token_2022_program”“token_program_2022”) 来指代.

为解决这一问题, 本文对主网中已部署的 1 045 个链上程序具有的 IDL 信息进行了解析和统计, 统计结果如图 4(a) 所示. 通过人工解读其中出现频率较高且具有明显指向性的名称的具体含义, 然后将名称与其应该指向的账号集合建立映射关系, 从而得到图 4(b). 本文在交易生成过程中利用图 4(b) 根据账号名称对输入账号地址建立约束, 以提高对 Solana 链上程序的探索效率.

IDL 信息还表明账号是否需要签名和写入权限, 二者都是简单的布尔值, 然而二者提供不同角度的约束.

若 IDL 信息表明某交易的输入账号需要签名, 则该交易需要携带该账号的数字签名, 而数字签名的生成必须使用私钥生成. 在 Solana 的账号模型中, 不是所有的账号都具有私钥. 大体上, Solana 账号可分为内置程序账号、链上程序账号、用户钱包账号、PDA 账号、系统变量账号这 5 种. 其中, 内置程序账号、系统变量账号都不具备私钥, 也无法进行签名. PDA 账号虽然可虚拟地由编程手段“签名”, 但必须经由跨程序调用产生, 而本文对交易的生成是直接调用被测程序而不是跨程序调用, 因此也无需考虑. 链上程序账号虽然具有私钥, 也可以为交易签名, 但通常只有管理链上程序时才会签名且其调用的是链上程序加载器, 而本文的测试目标是链上程序而非链上程序

加载器, 因此也不需要考虑. 综上所述, 当 IDL 信息表明某交易的输入账号需要签名, 本文将该账号的地址生成限制在用户钱包账号集合内.

名称	出现次数
systemProgram	5 412
tokenProgram	4 176
authority	2 766
payer	2 279
rent	1 993
admin	1 356
owner	1 333
associatedTokenProgram	960
mint	959
pool	946
user	941
signer	859
system_program	809
vault	795
state	739
config	615
program	508
token_program	504
clock	471
market	463
…	

(a) 1045份真实IDL信息中频率前20的账号名称

账号名称集合	目标账号地址 (或目标账号地址集合)
systemProgram	11111111111111111111111111111111
system_program	
system	
tokenProgram	TokenkegQfeZyiNwAJbNbG KPFXCWuBvf9Ss623VQ5DA
token_program	
token2022Program	TokenzQdBNbLqP5VehdkA S6EPFLC1PHnBqCXEpPxuEb
tokenProgram2022	
token_2022_program	
token_program_2022	
authority	{Pubkey$_i$ | Pubkey$_i$ 是用户钱包的公钥} (动态生成)
payer	
wallet	
feeAndRentPayer	
rentPayer	
signer	
vault	{Address$_k$ | Address$_k$ 是程序派生地址}(动态生成)
state	
clock	SysvarC1ock11111111111111111111111111111111
clockSysvar	
rent	SysvarRent111111111111111111111111111111
sysvarRent	

(b)典型账号名称到账号范围映射关系表

图 4　1 045 份真实 IDL 信息中频率前 20 的账号名称和典型账号名称到账号范围映射关系表

若 IDL 信息表明某交易的输入账号需要写入权限, 则需确保该账号所有者是交易中所指定链上程序. 在 Solana 的链上程序机制中, 对程序更改账号内容具有严格限制. 具体而言, 每个账号具有一个 *owner* 字段, 即账号所有者, 它指向区块链中的另一个账号, 通常是一个程序. 只有账号的所有者才可以更改账号的 *data* 字段内容. 因此, 链上程序只能直接更改其拥有的账号, 这从来源上通常分为两类: 一类是通过 PDA 机制创建的账号, 另一类是通过用户钱包机制创建账号之后又转移所有权给链上程序. 此外, 链上程序还可以通过跨程序调用机制来调用其他程序, 以间接更改其他程序所拥有的账号. 用户钱包账号的所有者是系统程序, 因此链上程序可通过跨程序调用系统程序来间接更改用户钱包账号. 综上所述, 当 IDL 信息表明某交易的输入账号需要写入权限, 本文将该账号的地址生成限制在用户钱包账号集合和 PDA 账号集合的并集之中.

4　实验评估

为评估本文方法的效果, 我们根据上述方法实现了一个原型工具 SolRaceFuzz, 并针对以下 3 个问题进行实验评估.

● RQ1: 缺陷发现能力. SolRaceFuzz 能否有效检测出基准数据集中的 Solana 链上程序并发缺陷? 能否有效检测真实世界中的 Solana 链上程序并发缺陷?

● RQ2: 效率. SolRaceFuzz 检测基准数据集和真实世界 Solana 链上程序中缺陷的效率如何? 它能应用到真实世界的大型 Solana 链上程序中吗?

● RQ3: 策略有效性. SolRaceFuzz 的各种策略确实提高了检测效果吗?

在本节中, 我们还与 FuzzDelSol[13]方法进行对比. 由于其未开放源代码, 本研究根据其论文重新实现. 据我们所知, FuzzDelSol 是当前 Solana 唯一公开的模糊测试工具, 且支持检测的缺陷类型并不含本文定义的两种缺陷.

为尽可能与同类工作对比, 本研究将这两种缺陷的检测准则移植到 FuzzDelSol 上, 并进行实验. 至于其他通用 Rust 模糊测试工具或传统区块链测试工具, 由于调用 Solana 链上程序除了接收一段字节输入以外, 还涉及模拟区块链账本与选择输入账号的问题, 通用 Rust 模糊测试工具或传统区块链测试工具并不具备这两大能力.

4.1 实验设置

实验对象为两个数据集, 分别可称作基准数据集和真实链上程序数据集, 下面分别介绍.

第 1 个实验数据集由大语言模型 DeepSeek-R1 671B 版生成, 包含 Solana 链上程序两种特定缺陷的基准数据集, 包含 65 个 Solana 链上程序. 本文这样构建数据集是因为本研究面临这样一种挑战: 尽管我们相信本文中两种缺陷类型真实存在, 然而据我们所知, 截至本文撰写时尚未有其他工作针对 Solana 链上程序并发缺陷开展研究, 因此无现成可用基准数据集. 此外, 从真实世界缺陷报告构建数据集也是不太可能的, 因为 Solana 链上程序来源多样且通常不可获得源代码. Smolka 等人[13]先前的研究声称仅有少于 2% 的 Solana 链上程序可获得源代码. 本研究从 Solana 主网随机抓取了 8 195 份 Solana 链上程序并发现仅有 139 份 (1.7%) 链上程序提供了源代码, 这与他们的结果相符. 综上所述, 为解决这一问题, 本文利用大语言模型 DeepSeek-R1 671B 版生成包含两种特定类型缺陷的 Solana 链上程序代码, 以构建一个关于 Solana 链上程序交易顺序依赖缺陷和时钟控制依赖缺陷的基准数据集, 该数据集中植入了 32 个交易顺序依赖缺陷和 33 个时钟控制依赖缺陷.

第 2 个实验数据集由 Solana 主网上随机抓取的真实链上程序组成, 作为评估 SolRaceFuzz 在真实世界程序中表现的大规模实验数据集, 该数据集包含 1 045 个 Solana 链上程序且均具有 IDL 信息, 但是否包含缺陷是未知的. 这些链上程序平均每个具有 0.98 MB 的 ELF 文件大小、425.93 个函数、64 352.86 条 SBPF 指令、11.79 种系统调用. 不失一般性地, 该数据集中包含的 Solana 链上程序 ELF 文件体积最小 182 KB, 最大 9.84 MB; 最少的仅包含 8 111 条 SBPF 指令, 最多的包含 822 321 条 SBPF 指令, 95% 的程序 SBPF 指令数量位于范围 [8 111, 149 268].

第 3 个实验数据集是一个半合成数据集, 以真实 Solana 开源项目的代码为基础, 我们人工手动在其中注入缺陷 (fault injection). 这种“半合成”的数据集比纯 LLM 生成的数据集更具说服力, 也能更好地模拟真实开发场景. 该数据集基于 5 个开源的 Solana 链上程序, 每个程序分别注入 1 个 TOD 缺陷与 1 个 CCD 缺陷. 为避免同时注入多个缺陷产生意外影响, 两种缺陷并非同时注入而是分别注入, 在 5 个原始程序上注入 2 类缺陷, 共得到 10 个经缺陷注入的程序.

基准数据集具有已知的源代码和确定的缺陷, 但代码量较少, 程序功能相对单一. 真实程序数据集代码量较大, 种类较多, 具有较大的多样性, 但未必提供了源代码, 所具有的缺陷也不能事先确定. 两个数据集可以相互补充, 以便更好地评估本文方法及原型工具. 半合成数据集则兼具贴近真实开发场景和所包含缺陷确定的优点.

本方法和对比方法的实验均在云主机平台 DigitalOcean 提供的云主机上进行, CPU 型号为 Intel Xeon 4 核, CPU 主频为 2.49 GHz, 内存大小为 8 GB, 内存主频为 2 933 MHz, 硬盘为 25 GB 的 SSD 及 100 GB 的存储卷, 分别用于系统盘和实验数据存储盘, 操作系统为 Debian 12 64 bit, Solana 命令行工具的版本号为 2.1.14, Cargo 和 Rustc 的版本号均为 1.84.1. 虽然该工具实现为单线程计算, 但该工具支持同时启动多个不同的实例并指定不同的输入输出路径. 这允许在需要测试多个待测链上程序时并行地进行, 以充分利用硬件提供的多核 CPU 计算资源.

实验中符号执行的超时相关设置为: 对于每次约束求解, 最多花费 500 ms, 若超出时间预算则中止并跳过当前试图求解的约束; 每个输入用例分配最多 3 s 的求解时间, 若超出预算则该测试用例不再进行符号约束求解.

4.2 缺陷发现能力

本节以缺陷报告数量作为衡量指标, 分别在 3 个数据集上开展实验评估, 并与 FuzzDelSol 方法进行对比.

首先在基准数据集上运行 SolRaceFuzz, 评估其对基准数据集中已知缺陷的发现能力, 并与 FuzzDelSol 方法进行对比, 工具执行检测的时间预算统一设为 1 min, 当超出时间预算时, 保存已发现的缺陷信息后退出. 实验结果如表 4 所示.

SolRaceFuzz 在 TOD 分类上达到 100.00% 的检测率, 表现出较好的缺陷发现能力, 但在 CCD 分类上仅有 93.94%. 研究表明, 两个 CCD 类型缺陷未能检出的原因在于, 系统变量取值的条件检查语句难以满足.

表 4 基准数据集上的 SolRaceFuzz 缺陷发现能力及对比

分类	已知缺陷	SolRaceFuzz			FuzzDelSol		
		平均函数覆盖率 (%)	平均指令覆盖率 (%)	发现缺陷	平均函数覆盖率 (%)	平均指令覆盖率 (%)	发现缺陷
TOD	32	**58.46**	**46.16**	**32 (100.00%)**	17.97	17.21	0 (0.00%)
CCD	33	42.81	39.09	31 (93.94%)	42.81	39.09	31 (93.94%)
ALL	65	**50.52**	**44.06**	**63 (96.92%)**	30.58	23.72	31 (47.69%)

注: 加粗表示SolRaceFuzz的数据优于对比方法FuzzDelSol; 括号中的数据表示发现缺陷占已知缺陷的比例

对比方法 FuzzDelSol 在 TOD 缺陷分类上未能检测出任一缺陷, 原因是不能充分探索多次交易执行之间的状态转移. FuzzDelSol 在 CCD 缺陷分类上达成了 93.94% 的检测率, 与 SolRaceFuzz 等同; FuzzDelSol 未能检出的 2 个 CCD 缺陷与 SolRaceFuzz 未能检出的 2 个 CCD 缺陷是相同的, 且未能检出的原因也一致.

我们在经过缺陷注入的半合成数据集上运行 SolRaceFuzz, 并与 FuzzDelSol 方法进行对比, 工具执行检测的时间预算统一设为 5 min. 实验结果如表 5 所示, “LOC (注入前)”代表相应开源程序在注入缺陷前的 Rust 源代码行数. “发现缺陷?”列中, “A/B”分别表示是否发现了本文所注入的 TOD 缺陷/CCD 缺陷, Y 表示发现, N 表示未发现.

表 5 经过缺陷注入的半合成数据集上 SolRaceFuzz 缺陷发现能力及对比

程序名	LOC (注入前)	SolRaceFuzz			FuzzDelSol		
		函数覆盖率 (%)	指令覆盖率 (%)	发现缺陷?	函数覆盖率 (%)	指令覆盖率 (%)	发现缺陷?
binary_option	1437	**42.86**	**18.96**	**Y/Y**	22.86	6.57	N/N
spl_managed_token	1270	**41.33**	**17.19**	**Y/Y**	33.95	11.75	N/N
spl_name_service	752	**37.86**	**23.29**	**Y**/Y	31.28	15.02	N/Y
spl_token_lending	6644	25.59	9.26	N/Y	25.59	9.27	N/Y
stateless_asks	640	**31.70**	**13.52**	**Y/Y**	28.57	9.78	N/N
合计	—	**35.87**	**16.44**	**4/5**	28.45	10.48	0/2

注: 加粗表示SolRaceFuzz的数据优于对比方法FuzzDelSol

SolRaceFuzz 检出所注入的 5 个 TOD 缺陷中的 4 个, 以及全部 5 个 CCD 缺陷, 并且无误报, 表现出较高的精度与召回率. 对比方法 FuzzDelSol 未检出任何 TOD 缺陷, 检出 2 个 CCD 缺陷.

我们还在大规模真实程序数据集上运行 SolRaceFuzz 以评估真实世界程序中未知缺陷发现能力, 同时与 FuzzDelSol 方法进行对比. 工具执行检测的时间预算均设为 5 min, 当超出时间预算时, 保存已发现的缺陷信息, 然后退出. 实验结果如表 6 所示. “函数”表示程序包含的可达 SBPF 函数个数, “指令”表示程序包含的可达 SBPF 指令个数. 需要强调的是, 这两列仅计入从入口点开始可达的函数和指令. “覆盖函数”和“覆盖指令”分别对应分析工具成功覆盖的 SBPF 函数和 SBPF 指令个数, 括号中的百分数为覆盖比例. “发现缺陷”表示成功发现的缺陷个数, 该列中的“A(B)/C(D)”代表成功发现 A 个 (B 个真阳性) TOD 型缺陷/C 个 (D 个真阳性) CCD 型缺陷. 由于该数据集包含 1045 个受分析程序而无法逐一列出, 表 6 仅列出成功发现缺陷的 45 个所分析程序的详细数据和全体程序的统计信息.

表 6 真实程序数据集上的 SolRaceFuzz 缺陷发现能力及对比

ProgramID	函数	指令	SolRaceFuzz			FuzzDelSol		
			覆盖函数	覆盖指令	发现缺陷	覆盖函数	覆盖指令	发现缺陷
4X2JL…D7aeY	656	72928	**305 (46.5%)**	**14738 (20.2%)**	0/**3**(**3**)	196 (29.9%)	6166 (8.5%)	0/0
6QXNN…eGyrM	413	28914	173 (41.9%)	**6399 (22.1%)**	0/**5**(**5**)	196 (47.5%)	6102 (21.1%)	0/0
86pSX…tbYEX	449	98757	**156 (34.7%)**	**11861 (12.0%)**	0/**1**(**1**)	108 (24.1%)	5559 (5.6%)	0/0
AxeE1…Ap5Du	513	32306	265 (51.7%)	9146 (28.3%)	0/**5**(**5**)	284 (55.4%)	9868 (30.5%)	0/0
AXEPm…r59KB	590	32306	**198 (33.6%)**	**9149 (18.8%)**	0/**4**(**4**)	183 (31.0%)	5865 (12.8%)	0/0
CyZ1S…pKaxt	412	29536	**207 (50.2%)**	**7798 (26.4%)**	0/**5**(**5**)	182 (44.2%)	5641 (19.1%)	0/0
FARM1…yJzyc	580	43772	**206 (35.5%)**	**9143 (20.9%)**	0/**5**(**5**)	126 (21.7%)	3508 (8.0%)	0/0
HajXY…aNnUf	407	63113	**137 (33.7%)**	**7611 (12.1%)**	0/**1**(**1**)	95 (23.3%)	5005 (7.9%)	0/0
KJ6b6…D2Y6E	563	38054	172 (30.6%)	**5472 (14.4%)**	0/**1**(**1**)	181 (32.1%)	5106 (13.4%)	0/0
SLend…FHJSh	636	155400	**191 (30.0%)**	**9502 (6.1%)**	0/**1**(**0**)	51 (8.0%)	1977 (1.3%)	0/0

表 6　真实程序数据集上的 SolRaceFuzz 缺陷发现能力及对比 (续)

ProgramID	函数	指令	SolRaceFuzz			FuzzDelSol		
			覆盖函数	覆盖指令	发现缺陷	覆盖函数	覆盖指令	发现缺陷
strmd…8aCGw	330	62 467	**127 (38.5%)**	**8 421 (13.5%)**	0/**29**(**27**)	94 (28.5%)	4 860 (7.8%)	0/0
TCVk8…GL9Gg	662	72 176	**313 (47.3%)**	**14 612 (20.2%)**	0/**3**(**3**)	148 (22.4%)	4 547 (6.3%)	0/0
TLPv2…hQxtX	2 117	805 266	**383 (18.1%)**	**31 255 (3.9%)**	0/**10**(**9**)	83 (3.9%)	2 851 (0.4%)	0/0
zF2vS…bLQ25	1 249	379 840	**383 (30.7%)**	**32 196 (8.5%)**	0/**478**(**419**)	78 (6.2%)	3 833 (1.0%)	0/0
2eDLE…LdJ9R	372	17 594	**206 (55.4%)**	**6 446 (36.6%)**	**6** (**6**)/0	54 (14.5%)	1 204 (6.8%)	0/0
3wwB1…ooibn	348	16 679	168 (48.3%)	4 780 (28.7%)	**2**(**2**)/0	180 (51.7%)	4 957 (29.7%)	0/0
4tdmk…GZhSt	873	307 714	**234 (26.8%)**	**14 752 (4.8%)**	**1**(**1**)/0	51 (5.8%)	2 398 (0.8%)	0/0
5Dq9P…Y3p5r	169	17 863	**154 (91.1%)**	**8 454 (47.3%)**	**1**(**1**)/0	96 (56.8%)	5 271 (29.5%)	0/0
6kDvz…xihCt	358	50 751	**191 (53.4%)**	**12 041 (23.7%)**	**3**(**3**)/0	80 (22.3%)	3 314 (6.5%)	0/0
6vBg1…c99qU	195	23 443	**167 (85.6%)**	**11 652 (49.7%)**	**1** (**1**)/0	96 (49.2%)	5 273 (22.5%)	0/0
7Keyd…n7W65	225	25 779	**151 (67.1%)**	**7 688 (29.8%)**	**7**(**7**)/0	32 (14.2%)	1 174 (4.6%)	0/0
8fSeR…8xEVe	192	22 962	**161 (83.9%)**	**10 174 (44.3%)**	**1**(**1**)/0	96 (50.0%)	5 282 (23.0%)	0/0
B6inX…t7tip	165	17 712	**63 (38.2%)**	**2 534 (14.3%)**	**1**(**1**)/0	32 (19.4%)	995 (5.6%)	0/0
BrpV1…AyhsK	193	23 026	**167 (86.5%)**	**11 660 (50.6%)**	**1**(**1**)/0	77 (39.9%)	3 513 (15.3%)	0/0
chatG…EX4To	439	67 626	**199 (47.5%)**	**14 783 (21.9%)**	**2**(**2**)/0	95 (22.7%)	4 885 (7.2%)	0/0
crypt…WmUcs	294	37 264	**162 (55.1%)**	**10 328 (27.7%)**	**7**(**7**)/0	98 (33.3%)	5 369 (14.4%)	0/0
DbDTN…5JRy5	168	16 823	**137 (81.5%)**	**7 100 (42.2%)**	**4**(**4**)/0	99 (58.9%)	5 312 (31.6%)	0/0
dca6x…xHB4W	260	33 439	**158 (60.8%)**	**8 816 (26.4%)**	**1**(**1**)/0	94 (36.2%)	4 859 (14.5%)	0/0
DeStf…2UugA	286	42 354	**175 (61.2%)**	**8 435 (19.9%)**	**3**(**3**)/0	57 (19.9%)	2 110 (5.0%)	0/0
dstK1…3meVd	211	25 182	**140 (66.4%)**	**9 260 (36.8%)**	**7**(**7**)/0	32 (15.2%)	1 172 (4.7%)	0/0
DyQWF…dWZS8	231	30 496	**150 (64.9%)**	**7 469 (24.5%)**	**1**(**1**)/0	31 (13.4%)	1 122 (3.7%)	0/0
Fd87T…CmowN	201	24 537	**147 (73.1%)**	**9 134 (37.2%)**	**3**(**3**)/0	69 (34.3%)	2 798 (11.4%)	0/0
FH2hm…7x5cg	160	16 425	**133 (83.1%)**	**6 732 (41.0%)**	**3**(**3**)/0	97 (60.6%)	5 298 (32.3%)	0/0
GovyJ…X1Dir	218	23 801	**150 (68.8%)**	**7 029 (29.5%)**	**5**(**5**)/0	31 (14.2%)	1 148 (4.8%)	0/0
hadeK…CNbvu	641	167 515	**257 (40.1%)**	**19 532 (11.7%)**	**2**(**2**)/0	33 (5.1%)	1 222 (0.7%)	0/0
JUP2j…uJvfo	1 088	98 262	**400 (36.8%)**	**35 059 (35.7%)**	**1**(**1**)/0	196 (18.0%)	5 784 (5.9%)	0/0
JUP3c…s32Ph	457	140 992	**233 (51.0%)**	**41 394 (29.4%)**	**1**(**1**)/0	91 (19.9%)	4 420 (3.1%)	0/0
KodBs…3JUav	739	54 848	**360 (48.7%)**	**15 622 (28.5%)**	**1**(**1**)/0	193 (26.1%)	5 664 (10.3%)	0/0
pantx…KiEmR	260	33 439	**131 (50.4%)**	**7 051 (21.1%)**	**1**(**1**)/0	31 (11.9%)	1 077 (3.2%)	0/0
porcS…mBegy	209	25 276	**143 (68.4%)**	**8 306 (32.9%)**	**5**(**5**)/0	31 (14.8%)	997 (3.9%)	0/0
SBAPy…694Ha	884	51 695	**452 (51.1%)**	**14 590 (28.2%)**	**6**(**6**)/0	85 (9.6%)	1 566 (3.0%)	0/0
sbatt…gDshx	1 301	121 797	**524 (40.3%)**	**20 458 (16.8%)**	**2**(**2**)/0	78 (6.0%)	1 531 (1.3%)	0/0
SoarN…hJYkk	429	49 896	**186 (43.4%)**	**11 652 (16.7%)**	**7**(**7**)/0	31 (7.2%)	1 177 (1.7%)	0/0
vEafW…ByhCM	471	22 508	**257 (54.6%)**	**8 354 (37.1%)**	**1**(**1**)/0	185 (39.3%)	5 089 (22.6%)	0/0
whirL…ctyCc	623	124 991	**245 (39.3%)**	**14 312 (11.5%)**	**4**(**4**)/0	32 (5.1%)	1 156 (0.9%)	0/0
平均 (发现缺陷)	488.4	81 007.1	**211.8**	**12 189.9**	**2.0**/**12.0**	98.1	3 674.0	0/0
平均 (所有)	369.2	57 518.8	**123.1**	**6 319.6**	**0.1**/**0.5**	68.1	2 679.7	0/0

注: 加粗表示SolRaceFuzz的数据优于对比方法FuzzDelSol

SolRaceFuzz 在 1 045 个被分析的真实世界链上程序中检测出了 45 程序含有缺陷, 检测出 TOD 缺陷 91 个、CCD 缺陷 551 个, 共计报告 642 个缺陷. 经过误报分析, 其中 91 个 TOD 缺陷为真阳性、488 个 CCD 缺陷为真阳性, 共计 579 个缺陷为真阳性. 平均每个受分析程序含有 0.1 个 TOD 缺陷和 0.5 个 CCD 缺陷; 对比方法 FuzzDelSol 在该数据集上未能检测出任何一个缺陷, 本文对此进行了进一步探究, 结果表明原因是 FuzzDelSol 随机生成的指令数据基本不能成功通过指令的标识符检查, 指令的标识符是一段 8 字节的哈希值, 随机生成仅有 $1/256^8$ 的概率生成特定的指令标识符. SolRaceFuzz 得益于 IDL 信息和符号执行的指导, 可以更有效地突破指令的标识符检查.

综合以上两个数据集的实验结果, SolRaceFuzz 均表现出较好的缺陷发现能力, 且效果好于对比方法 FuzzDelSol. 这主要得益于 SolRaceFuzz 对 Solana 链上程序的自动化探索策略更具有针对性.

对 RQ1 的回答: SolRaceFuzz 能有效检测基准数据集中 63/65 的 Solana 链上程序并发缺陷, 也在约 4.3% 的真

实世界 Solana 链上程序中检测出了交易顺序依赖缺陷或时钟控制依赖缺陷.

● 误报分析. 为确认所发现缺陷是否为误报, 我们实际执行原型工具报告触发缺陷的交易, 并查看执行后结果是否触发缺陷, 以确认原型工具的报告是否为真阳性. 结果如表 7 所示.

表 7　SolRaceFuzz 在真实程序数据集上的误报分析

分类	Report	*TP*	*FP*	精度 (%)
TOD	91	91	0	100.0
CCD	551	488	63	88.6
合计	642	579	63	90.2

表 7 中, Report、*TP*、*FP* 分别表示报告总数、真阳性个数、假阳性个数. 精度 (*Precision*) 的计算公式如下:

$$Precision = \frac{TP}{TP+FP}.$$

SolRaceFuzz 在真实程序数据集上, 检测 TOD 的精度达到 100.0%, 无误报; 检测 CCD 的精度达到 88.6%. 本文检查了剩下 11.4% 的 CCD 缺陷为何未能触发, 发现其控制流路径发生了改变, 我们认为这是由于检测时所模拟的系统变量与重放执行时的系统变量取值不同引起的, 也就是说这 11.4% 也有可能并不是假阳性. 由于无法完全控制 Solana 真实环境的系统变量, 我们未能通过重放执行来复核这 11.4% 的真假. 综上所述, 至少 88.6% 的 CCD 缺陷是真阳性. 整体上, 由于本文方法采用动态分析, 精度较好, 可达到 90.2%.

● 覆盖率分析. 本文注意到部分被测程序的覆盖率偏低. 经过探究, 这主要是由于我们在二进制层面而非源码层面对 SBPF 指令进行测试与计算覆盖率. 同类工作 FuzzDelSol 也部分提到了这一点. 我们发现, 链上程序编译后的可执行文件不仅包含了开发者编写的函数代码, 还包含了一些库函数和宏展开后插入的检查代码. 我们对此进行了简单的实验, 编写一份源码层面仅包含 1 个 entrypoint 函数且无条件判断分支的 Solana 链上程序, 编译后再反汇编可在 SBPF 层面发现多达 52 个 SBPF 函数与众多条件判断, 执行 entrypoint 仅覆盖 2 个 SBPF 函数与 28.27% 的 SBPF 指令. 也就是说, 即使我们在源码级别已覆盖了唯一的 entrypoint 函数与其代码, 但是未完全覆盖编译后自动插入的条件检查分支与库函数, 特别是其作为通用函数/代码处理各种情况的分支, 我们认为这些函数与分支有一部分在当前程序中甚至应为死代码.

目前在二进制层面暂难以有效区分开发者编写的条件检查与自动插入的条件检查 (经过编译优化后可能基本块都被打乱甚至函数调用被内联), 也难以有效地区分开发者编写的函数与所用到的库函数 (链上版本剥离了全部函数符号名称与调试信息), 因此本文在计算覆盖率时也未能有效筛除, 这些非链上程序开发者编写的代码部分计入分母应当会造成所统计覆盖率系统性低于应有的覆盖率.

4.3　缺陷检测效率

评估 SolRaceFuzz 的检测效率. 本节以平均每秒执行交易次数、缺陷发现时间、函数覆盖数、指令覆盖数等作为衡量指标, 分别在基准程序和真实程序上开展实验评估.

本节首先在基准数据集上评估 SolRaceFuzz 的缺陷检测效率, 运行时间设定为 1 min, 运行过程中即时记录交易执行次数、是否发现缺陷等信息. 实验结果如表 8 所示. “平均每秒执行交易”代表每秒执行交易次数, “发现缺陷平均用时”指成功发现这些缺陷的平均用时, 不计入未能发现的缺陷项, 保留 1 位小数.

表 8　基准数据集上的 SolRaceFuzz 缺陷发现能力及对比实验结果

分类	已知缺陷	SolRaceFuzz			FuzzDelSol		
		平均每秒执行交易	发现缺陷平均用时 (s)	发现缺陷	平均每秒执行交易	发现缺陷平均用时 (s)	发现缺陷
TOD	32	683.0	1.6	**32 (100.00%)**	1 700.6	N/A	0 (0.00)
CCD	33	1 029.8	0.0	31 (93.94%)	1 565.9	0.0	31 (93.94%)
ALL	65	800.5	0.77	63 (96.92%)	1 632.2	0.0	31 (47.69%)

注: 加粗表示SolRaceFuzz的数据优于对比方法FuzzDelSol; 括号中的数据表示发现缺陷占已知缺陷的比例

由实验结果可知, SolRaceFuzz 的交易执行次数相比 FuzzDelSol 出现了一定的下降, 研究发现这是 SolRaceFuzz

的探索策略更加复杂所导致的. SolRaceFuzz 发现缺陷的平均用时也较 FuzzDelSol 略有增加, 研究发现这是因为 FuzzDdelSol 未能发现隐藏较深的难探索缺陷. 两者发现缺陷的数量佐证了这一观点.

本研究在真实程序上评估 SolRaceFuzz 的缺陷检测效率时, 运行时间设定为 5 min, 运行过程中即时记录 SBPF 指令覆盖数、函数覆盖数、缺陷发现数量等信息, 并记录总交易执行次数. 本实验评估了 1 045 个真实程序, 由于总数较多, 本文选择在图 5 中仅展示其中 6 个真实程序以代表性地展示具体趋势, 在表 9 中列出受分析对象全体的统计信息. 这 3 张折线图分别展示了 6 个真实程序在 5 min 内的覆盖函数数量、覆盖指令数量、发现缺陷数量的变化趋势. 此外, 这 6 个程序的平均每秒执行交易次数如表 9 所示.

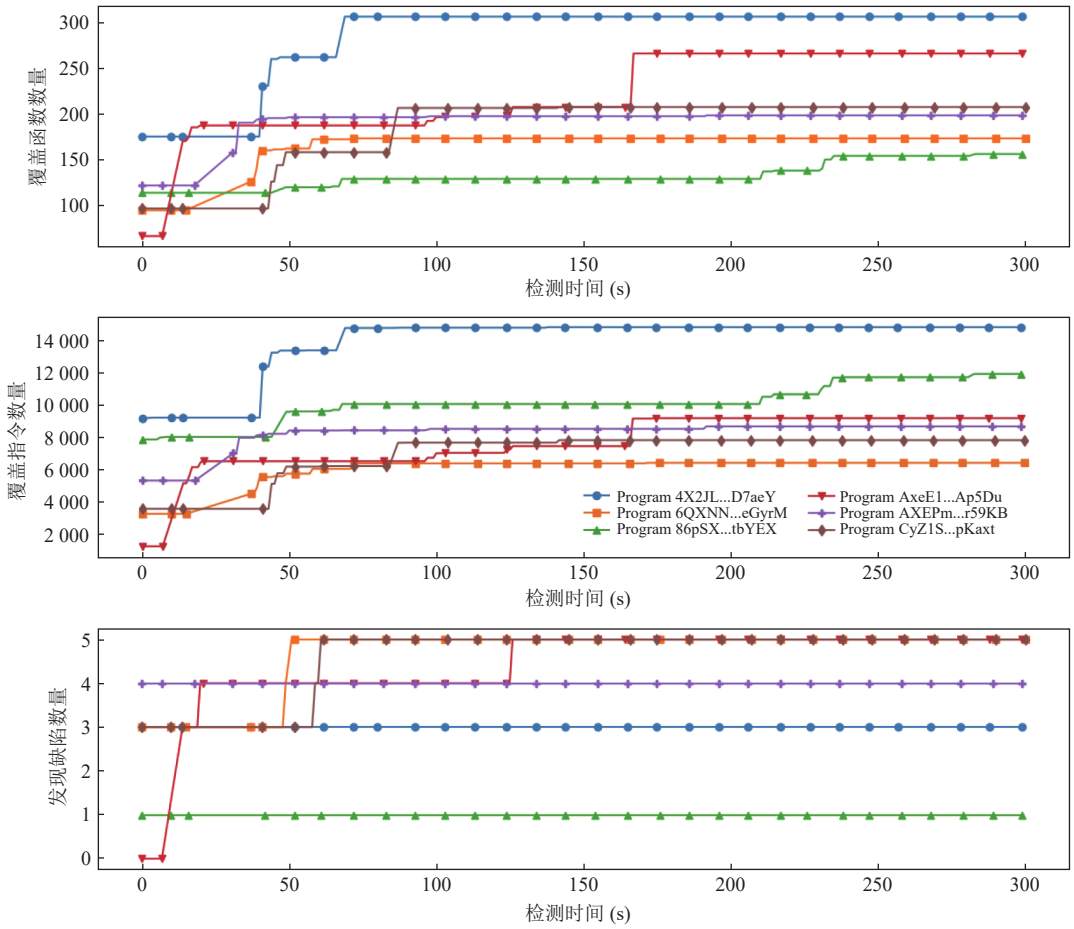

图 5 SolRaceFuzz 在真实程序上的检测指标随时间变化趋势

表 9 SolRaceFuzz 在真实程序上的每秒交易执行次数

程序ID	SBPF函数数量	SBPF指令数量	平均每秒执行交易
Program 4X2JL…D7aeY	656	72 928	447.9
Program 6QXNN…eGyrM	413	28 914	399.5
Program 86pSX…tbYEX	449	98 757	340.2
Program AxeE1…Ap5Du	513	32 306	610.4
Program AXEPm…r59KB	590	45 877	492.8
Program CyZ1S…pKaxt	412	29 536	369.9
平均 (图5中的6个)	505.5	51 386.3	443.5
平均 (共计1 045个)	369.2	57 518.8	328.7

综合以上实验结果, SolRaceFuzz 的交易执行速度相比对比方法 FuzzDelSol 略有降低, 本文认为这是引入更复杂策略带来的代价. 但总体来说仍然可以接受, 且本文方法实际上具有更好的最终缺陷发现能力.

对 RQ2 的回答: SolRaceFuzz 能够以每秒 800.5 次交易执行的效率检测 Solana 链上程序基准数据集中的缺陷, 也能对真实世界中的 Solana 链上程序达到约每秒 328.7 次的交易执行效率, 总体来说效率可接受.

4.4 消融实验

本节评估 SolRaceFuzz 的探索策略有效性. 本节针对 SolRaceFuzz 的各种探索策略开展消融实验. 根据第 3 节, SolRaceFuzz 主要具有系统变量模拟、程序派生地址 (PDA) 生成、IDL 指导的探索、结合混合符号执行 (Concolic) 的探索等策略. 本节从完整的 SolRaceFuzz 中单独移除每个功能模块, 并通过实验重新测试其执行结果, 以探究这些策略每个所产生的作用.

实验结果如表 10 所示. “检出缺陷程序”代表该项在整个数据集中检出含缺陷程序的个数, “累计覆盖 SBPF 指令” 表示所覆盖的 SBPF 指令条数总和. 实验结果表明: 在基准数据集上, 只有消除系统变量模拟引起了较大的缺陷发现能力下降. 消除 PDA 生成功能没有引起缺陷发现能力下降, 本文认为这是因为基准数据集中的程序不强制要求使用 PDA 作为状态账号, 可以接受普通的账号作为状态账号. 消除符号执行的辅助探索只引起了少许的缺陷发现能力下降, 本文认为这是因为基准数据集中的程序较为简单, 基本不存在复杂和难以满足的条件检查, 因此符号执行没有起到太大的帮助; 在真实程序数据集上, 消除各策略功能模块均引起了不同程度的缺陷发现能力下降. 其中 IDL 引导的探索对缺陷发现能力的影响最大, 其次是对系统变量的模拟. 在覆盖 SBPF 指令数量上, 消除各功能模块均引起了较大的覆盖减少, 这表明了所使用策略的必要性.

表 10　SolRaceFuzz 分别在不同数据集上进行消融实验的结果

消融项	基准数据集		真实程序数据集		缺陷注入半合成数据集	
	检出缺陷程序	累计覆盖 SBPF指令	检出缺陷程序	累计覆盖 SBPF指令	检出缺陷程序	累计覆盖 SBPF指令
SolRaceFuzz (完整)	63	79 105	45	6 604 004	9	24 729
–系统变量模拟	49	72 990	32	1 135 590	9	19 098
–PDA 生成	63	79 125	44	1 156 608	9	23 184
–IDL引导生成	—	—	16	945 769	—	—
–Concolic 探索	62	77 743	43	869 466	7	22 567

对 RQ3 的回答: SolRaceFuzz 在基准数据集上各探索策略没有起到太显著的作用, 但对于真实程序的检测而言, 各探索策略均有效地提升了对于 Solana 链上程序的探索能力.

5 讨　论

我们注意到之前的一些研究[25−31]在以太坊的智能合约上关注了类似缺陷, 但这些研究与本文并不完全相同. 在 Solana 区块链上检测这些缺陷面临不同的挑战: 以太坊将程序与状态存放在一起, 而 Solana 采用程序与状态分离存放的模型, 从而衍生出程序派生账号创建、交易输入账号选择等问题; 以太坊在 EVM 指令集中为智能合约提供特殊操作码 (如 TIMESTAMP、NUMBER 等) 来访问时间戳、块高度等信息, 而 Solana 通过提供特殊的系统变量 (Sysvar) 为链上程序提供访问此类集群状态信息的能力. 此外, Solana 链上程序中的 CCD 缺陷通常使用与块相关的逻辑时钟编号 (称之为 slot), 而以太坊中的类似缺陷使用的是真实时间戳或区块高度.

6 总　结

Solana 区块链是当前日趋火热的分布式高性能公有区块链, 本文发现 Solana 链上程序中存在着两类独特的并发缺陷类型, 分别称之为交易顺序依赖缺陷与时钟控制依赖缺陷. 为有效检测这两类缺陷, 本文基于模糊测试与符号执行提出了一种 Solana 链上程序缺陷检测方法. 本文首先针对 Solana 链上程序中的交易顺序依赖缺陷和时

钟控制依赖缺陷分别给出正式定义, 并提出缺陷检测准则. 针对 Solana 区块链特点, 本文提出账本模拟与交易生成方法来高效触发缺陷. 在 3 组数据集上, 从缺陷发现能力、效率和探索策略这 3 个方面进行实验验证了所提方法的有效性.

References

[1] Solana Foundation. State of the Solana Ecosystem, January 2024. 2024. https://solana.com/2024outlook

[2] Li XY, Wang XY, Kong TL, Zheng JH, Luo M. From bitcoin to Solana—Innovating blockchain towards enterprise applications. In: Proc. of the 4th Int'l Conf. on Blockchain (ICBC 2021). Springer, 2021. 74–100. [doi: 10.1007/978-3-030-96527-3_6]

[3] Yakovenko A. Solana: A new architecture for a high performance blockchain v0.8.13. 2017. https://solana.com/solana-whitepaper.pdf

[4] Ashraf M, Heavey C. A prototype of supply chain traceability using Solana as blockchain and IoT. Procedia Computer Science, 2023, 217: 948–959. [doi: 10.1016/j.procs.2022.12.292]

[5] Duffy F, Bendechache M, Tal I. Can Solana's high throughput be an enabler for IoT? In: Proc. of the 21st IEEE Int'l Conf. on Software Quality, Reliability and Security Companion (QRS-C). Hainan: IEEE, 2021. 615–621. [doi: 10.1109/QRS-C55045.2021.00094]

[6] Kong DC, Li XQ, Li WK. Characterizing the Solana NFT ecosystem. In: Companion Proc. of the 2024 ACM Web Conf. Singapore: ACM, 2024. 766–769. [doi: 10.1145/3589335.3651478]

[7] Bhattacharya P, Verma A, Obaidat MS, Tanwar S, Sadoun B. SaFaR: Solana blockchain-based optimal route selection scheme for cab aggregators. In: Proc. of the 2022 Int'l Conf. on Computer, Information and Telecommunication Systems (CITS). Piraeus: IEEE, 2022. 1–5. [doi: 10.1109/CITS55221.2022.9832995]

[8] Sliwinski J, Kniep Q, Wattenhofer R, Schaich F. Halting the Solana blockchain with epsilon stake. In: Proc. of the 25th Int'l Conf. on Distributed Computing and Networking. Chennai: ACM, 2024. 45–54. [doi: 10.1145/3631461.3631553]

[9] Hung HW, Amiri Sani A. BRF: Fuzzing the eBPF runtime. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 52. [doi: 10.1145/3643778]

[10] Andreina S, Cloosters T, Davi L, Giesen JR, Gutfleisch M, Karame G, Naiakshina A, Naji H. Defying the odds: Solana's unexpected resilience in spite of the security challenges faced by developers. In: Proc. of the 2024 ACM SIGSAC Conf. on Computer and Communications Security. Salt Lake City: ACM, 2024. 4226–4240. [doi: 10.1145/3658644.3670333]

[11] Pierro GA, Amoordon A. A tool to check the ownership of Solana's smart contracts. In: Proc. of the 2022 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Honolulu: IEEE, 2022. 1197–1202. [doi: 10.1109/SANER53432.2022.00140]

[12] Cui SW, Zhao G, Gao YF, Tavu T, Huang J. VRust: Automated vulnerability detection for Solana smart contracts. In: Proc. of the 2022 ACM SIGSAC Conf. on Computer and Communications Security. Los Angeles: ACM, 2022. 639–652. [doi: 10.1145/3548606.3560552]

[13] Smolka S, Giesen JR, Winkler P, Draissi O, Davi L, Karame G, Pohl K. Fuzz on the beach: Fuzzing Solana smart contracts. In: Proc. of the 2023 ACM SIGSAC Conf. on Computer and Communications Security. Copenhagen: ACM, 2023. 1197–1211. [doi: 10.1145/3576915.3623178]

[14] Wang C, Li Y, Gao JB, Wang K, Zhang JS, Guan Z, Chen Z. SolaSim: Clone detection for Solana smart contracts via program representation. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Program Comprehension. Lisbon: ACM, 2024. 258–269. [doi: 10.1145/3643916.3644406]

[15] Madhwal Y, Yanovich Y, Zhintiy G. Portable blockchain for system testing of Solana smart contracts. In: Proc. of the 7th Int'l Conf. on Blockchain Technology and Applications. Xi'an: ACM, 2024. 49–54. [doi: 10.1145/3708622.3708633]

[16] Solana Foundation. Programs on Solana. 2025. https://solana.com/zh/docs/core/programs

[17] Tao J, Mi XY, Wang BS, Wang PF. Evaluation and analysis of concolic execution optimizations in hybrid fuzzing. Journal of National University of Defense Technology, 2023, 45(2): 45–54. [doi: 10.11887/j.cn.202302005]

[18] Wang QY, Xu JC, Li YW, Pan ZL, Zhang YQ, Zhang C, Ji SL. A review of smart fuzzing: Problem exploration and method classification. Chinese Journal of Computers, 2024, 47(9): 2059–2083. [doi: 10.11897/SP.J.1016.2024.02059]

[19] Cui ZQ, Zhang JM, Zheng LW, Chen X. A survey of research on coverage-guided greybox fuzzing. Chinese Journal of Computers, 2024, 47(7): 1665–1696 (in Chinese with English abstractin Chinese). [doi: 10.11897/SP.J.1016.2024.01665]

[20] 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 with English abstract). http://www.jos.org.cn/1000-9825/7396.htm [doi: 10.13328/j.cnki.jos.007396]

[21] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. EXE: Automatically generating inputs of death. ACM Trans. on Information and System Security (TISSEC), 2008, 12(2): 10. [doi: 10.1145/1455518.1455522]

[22] 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.

[23] Sen K. Concolic testing. In: Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering. Atlanta: ACM, 2007. 571–572. [doi: 10.1145/1321631.1321746]

[24] Chipounov V, Georgescu V, Zamfir C, Candea G. Selective symbolic execution. In: Proc. of the 5th Workshop on Hot Topics in System Dependability (HotDep 2009). 2009.

[25] 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 (EuroS&P). Vienna: IEEE, 2021. 103–119. [doi: 10.1109/EuroSP51992.2021.00018]

[26] Kolluri A, Nikolic I, Sergey I, Hobor A, Saxena P. Exploiting the laws of order in smart contracts. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 363–373. [doi: 10.1145/3293882.3330560]

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

[28] Dong CT, Huang H, Shang Y. Erinys: Efficient fuzzing by function invoke sequence generation for smart contracts. In: Proc. of the 8th Int'l Conf. on Big Data and Internet of Things. Macau: ACM, 2024. 236–241. [doi: 10.1145/3697355.3697394]

[29] Han Q, Wang L, Zhang HY, Shi LY, Wang DX. Ethchecker: A context-guided fuzzing for smart contracts. The Journal of Supercomputing, 2024, 80(10): 13949–13975. [doi: 10.1007/s11227-024-05954-9]

[30] Ye MX, Nan YH, Dai HN, Yang S, Luo XP, Zheng ZB. FunFuzz: A function-oriented fuzzer for smart contract vulnerability detection with high effectiveness and efficiency. ACM Trans. on Software Engineering and Methodology, 2024, 33(7): 191. [doi: 10.1145/3674725]

[31] Liang RC, Chen J, Wu C, He K, Wu YM, Cao RC, Du RY, Zhao ZM, Liu Y. Vulseye: Detect smart contract vulnerabilities via stateful directed graybox fuzzing. IEEE Trans. on Information Forensics and Security, 2025, 20: 2157–2170. [doi: 10.1109/TIFS.2025.3537827]

附中文参考文献

[17] 陶静, 糜娴雅, 王宝生, 王鹏飞. 混合模糊测试中混合符号执行优化策略评估与分析. 国防科技大学学报, 2023, 45(2): 45–54. [doi: 10.11887/j.cn.202302005]

[18] 王琴应, 许嘉诚, 李宇薇, 潘祖烈, 张玉清, 张超, 纪守领. 智能模糊测试综述: 问题探索和方法分类. 计算机学报, 2024, 47(9): 2059–2083. [doi: 10.11897/SP.J.1016.2024.02059]

[19] 崔展齐, 张家铭, 郑丽伟, 陈翔. 覆盖率制导的灰盒模糊测试研究综述. 计算机学报, 2024, 47(7): 1665–1696. [doi: 10.11897/SP.J.1016.2024.01665]

[20] 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 [doi: 10.13328/j.cnki.jos.007396]

作者简介

高旭, 硕士, 主要研究领域为软件工程, 软件缺陷检测, 区块链技术.

马辰阳, 博士, 讲师, 主要研究领域为软件工程, 软件缺陷检测.

叶盛, 博士生, 主要研究领域为软件工程, 软件安全性, 服务计算.

李晅松, 博士, 副教授, CCF 专业会员, 主要研究领域为软件方法学, 形式化技术, 普适计算技术, 物联网安全.

宋巍, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件工程与方法学, 形式化方法, 服务计算.