计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(9):257−264 [doi: 10.15888/j.cnki.csa.009249]

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

基于交易序列分层变异的 EVM 模糊测试①

周潮晖, 赵易如, 刘　佩, 王笑克, 童铃皓, 赵　磊

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

摘　要: 以太坊虚拟机是以太坊区块链中关键组成部分, 其缺陷会导致交易的执行结果出现偏差, 给以太坊生态带来严重问题. 现有的以太坊虚拟机缺陷检测工作仅将虚拟机视为独立的智能合约执行工具, 没有完整测试其工作流程, 从而导致缺陷检测存在盲点. 针对上述问题, 提出了一种以太坊虚拟机运行全过程的缺陷检测方法 (ETHCOV). ETHCOV 首先结合权重策略指导智能合约、合约接口参数输入和交易序列按不同粒度变异, 然后将其与区块状态以及世界状态打包作为测试用例, 最后将测试用例输入到以太坊虚拟机中触发运行并对比检验运行结果, 以此来检测以太坊虚拟机的漏洞缺陷. 基于上述方法实现了一个原型系统, 并以 2 万多个真实智能合约作为为输入对以太坊虚拟机进行缺陷检测测试. 实验结果表明, 相较于现有工具 EVMFuzzer, ETHCOV 的测试效率提升了 339%, 代码覆盖率提升了 125%, 并检测出 3 组用例的不一致输出. 这些结果表明 ETHCOV 能有效检测以太坊虚拟机的缺陷.

关键词: 以太坊虚拟机; 模糊测试; 智能合约; 交易序列

引用格式: 周潮晖,赵易如,刘佩,王笑克,童铃皓,赵磊.基于交易序列分层变异的 EVM 模糊测试.计算机系统应用,2023,32(9):257–264. http://www.c-s-a.org.cn/1003-3254/9249.html

Fuzzer for EVM Based on Hierarchical Variation of Transaction Sequences

ZHOU Chao-Hui, ZHAO Yi-Ru, LIU Pei, WANG Xiao-Ke, TONG Ling-Hao, ZHAO Lei

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

Abstract: The Ethereum virtual machine (EVM) is a key component of the Ethereum blockchain, and its defects will cause deviations in the execution results of transactions, which will bring serious problems to the Ethereum ecosystem. The existing work on EVM defect detection only treats the virtual machine as an independent smart contract execution tool and does not fully test its workflow, resulting in blind spots in defect detection. To solve the above problems, a defect detection method for the whole process of EVM operation (ETHCOV) is proposed. ETHCOV first combines the weight strategy to guide smart contracts, contract interface parameter inputs, and transaction sequences to vary at different granularities. It then packages them with block state and world state as test cases and finally inputs the test cases into the EVM to trigger the run and compare the test run results, so as to detect the vulnerabilities in the EVM. Based on the above method, a prototype system is implemented and more than 20 000 real smart contracts are tested as input to the EVM for defect detection. Experimental results show that compared with the existing tool EVMFuzzer, ETHCOV improves the test efficiency by 339% and the code coverage by 125%, and the inconsistent output of three sets of test cases is detected. These results show that ETHCOV can effectively detect defects in the EVM.

Key words: Ethereum virtual machine (EVM); fuzzer; smart contract; transaction sequence

① 基金项目: 国家自然科学基金 (62172305); 湖北省重点研发计划 (2021BAA027)
收稿时间: 2023-03-04; 修改时间: 2023-04-04; 采用时间: 2023-04-28; csa 在线出版时间: 2023-07-14
CNKI 网络首发时间: 2023-07-17

以太坊虚拟机 (EVM)[1,2] 用来执行以太坊[3] 的交易和更改以太坊状态, 是以太坊架构中的关键部分. EVM 定义了从一个区块计算产生下一个区块新的有效状态的规则, 包括智能合约的部署、调用、运行、结果写回多个执行环节, 以及这些行为对世界状态产生的影响.

由于 EVM 在庞大的以太坊区块链生态中运行并承载着巨大的经济利益, 所有节点都基于 EVM 处理交易, 它的可靠性至关重要, 已有多次由于 EVM 代码漏洞引发了安全事故. 2017 年 4 月, Golem 团队发现 EVM 的设计缺陷导致的短地址攻击漏洞[4], 可能会导致用户的财产损失. 具体而言, 在 ERC20 代币标准定义的转账操作中, 当传入的是末端缺省的短地址时, EVM 会自动用后续的字节补足地址, 而转账的字节不足则用 0 填充, 导致实际转出的代币数值倍增. 2020 年 8 月 Saad 等人发现特定的交易序列可能导致某些版本的客户端共识失败[5]. 作为以太坊交易执行的可信平台和标准, 如果 EVM 的实现存在漏洞, 必然会导致严重的后果. 因此 EVM 的安全问题关乎整个以太坊生态, 不容忽视.

针对 EVM 的安全问题, Fu 等人[6,7] 提出了一种基于差分模糊测试的 EVM 缺陷检测方法, 并实现了 EVMFuzzer 工具. 该方法将智能合约以及根据合约函数的接口所生成的合约接口参数 (ABI) 输入作为测试用例, 并设计了 8 种变异算子来对智能合约的 Solidity 源码进行变异, 随后将生成的测试用例分别输入到多个不同语言版本的 EVM 中执行, 对比各 EVM 运行差异, 引导智能合约向使 EVM 执行差异最大化的方向变异, 从而检测各语言版本 EVM 实现不一致的缺陷. Cassez 等人[8] 基于 Dafny 实现 EVM, 并提供了 EVM 语义的可读性、完整和正式规范, 以及给出了验证 EVM 字节码正确性的框架. Yang 等人[9] 使用 RASP 技术来确保 EVM 操作的安全性. 它嵌入在 EVM 核心模块中用以跟踪交易输入, 监控状态和账户变量, 并使用 EVM 上下文作为输入来准确分析智能合约代码目的. 一旦发现漏洞利用行为, 将立即停止并回滚交易, 以确保区块链世界的安全.

现有的相关工作在检测 EVM 缺陷时仅将 EVM 视为独立的智能合约执行工具, 无法全面测试 EVM 作为状态转换机的功能, 这可能会导致 EVM 的某些缺陷被忽视, 并且现有方法没有考虑到 EVM 运行机制与智能合约结构的特殊性, 直接采用模糊测试将导致测试效率低下. 具体而言, 在真实的以太坊区块链环境下, EVM 是一个状态转换机, 如果要完整测试 EVM 的工作流程, 即测试完整的智能合约部署、执行流程以及完成智能合约间的调用, 需要通过交易来触发 EVM 的运行. 以太坊的交易输入具有高结构化的特征并包含丰富的语义信息. 想要使得输入能到达 EVM 内部的深层逻辑, 就需要在符合其语法的情况下, 尽可能地变异出包含各种语义的输入.

针对上述问题, 本文设计了一种交易序列分层变异方法, 指导智能合约、ABI 输入和交易序列按不同粒度变异, 面向以太坊客户端 Go Ethereum (Geth) 中的 EVM 实现了缺陷检测框架——ETHCOV. ETHCOV 首先基于抽象语法树对智能合约进行解析与剪枝, 然后在模糊测试模块中将智能合约、ABI 输入和交易序列作为种子进行选择和分层变异. 最后 ETHCOV 将种子与区块状态以及世界状态打包作为测试用例输入到 EVM 中运行, 并通过比对不同版本 EVM 执行的结果和世界状态以生成检测报告, 从而对 EVM 进行全面的缺陷检测.

本文的研究工作贡献如下.

(1) 针对如何完整测试 EVM 的工作流程的问题, 提出了针对以太坊虚拟机运行全过程的缺陷检测方法.

(2) 提出了针对交易序列的分层变异方法, 该方法通过对智能合约不同粒度的变异和调用字段的策略性变异, 可以使生成输入到达虚拟机内部的深层逻辑.

(3) 设计并实现了 ETHCOV 方法, 经评估表明, ETHCOV 的测试效率和代码覆盖率相对于 EVMFuzzer 有显著提高, 并检测出 3 组用例的不一致输出.

1 以太坊虚拟机缺陷检测框架

在本节我们将介绍 EVM 及其运行机制, 并据此给出相应的模糊测试框架. EVM 是以太坊架构的关键部分, 定义了从区块到区块改变状态的具体规则. 具体而言, 在链中的任何给定块上, 以太坊只有一个“规范”状态, 而 EVM 定义了从一个块到另一个块计算新的有效状态的规则.

如图 1 所示, EVM 是一种交易驱动的状态机[10], 其状态是一个大型的数据结构, 被称为世界状态 (world state), 世界状态包含了以太坊中所有账户地址与对应的账户状态的映射. EVM 在其中起到的作用为对状态转换[11] 的处理和计算. 如图 2 所示, EVM 的完整工作

流程包括智能合约部署、调用、运行和写回. 在由已部署的合约及其状态与账户状态共同构成 EVM 执行的上下文中, 用户通过交易调用合约触发 EVM 运行计算, 最后 EVM 将结果写回世界状态.

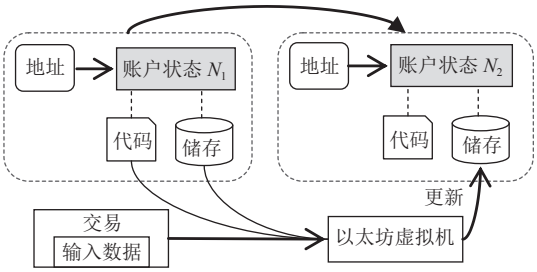

图 1 以太坊状态的转换过程

据此, 本文提出了一种针对以太坊虚拟机运行全过程的缺陷检测方案, 整体框架如图 3 所示. 在静态分析模块中, ETHCOV 基于抽象语法树对智能合约进行解析与剪枝, 生成调用智能合约函数的 ABI 输入. ETHCOV 将智能合约、ABI 输入和交易序列作为种子放入种子队列, 对种子选择和分层变异, 将变异后的交易序列与区块状态以及世界状态一起组成以太坊状态文件集[12], 作为 EVM 的测试用例, 交于 EVM 在本地部署[13] 执行, 最后进行异常检测对比[14,15] 和测试结果的输出.

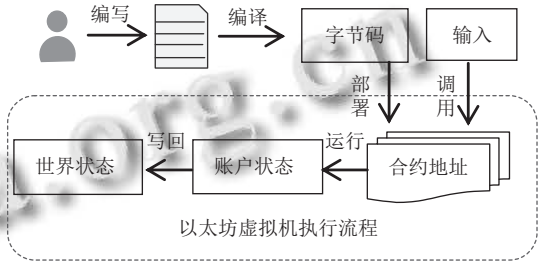

图 2 EVM 工作流程

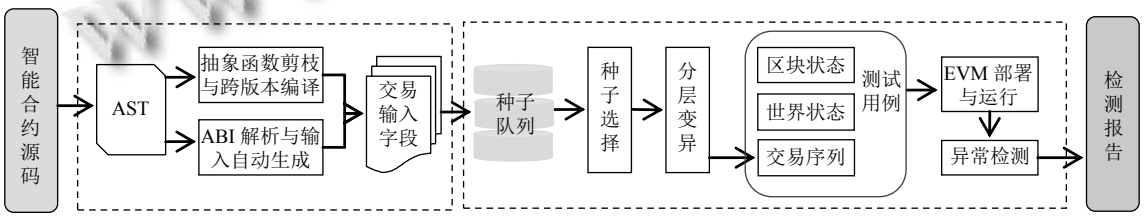

图 3 以太坊虚拟机运行全过程缺陷检测

2 设计思路

本节我们介绍 ETHCOV 的设计思路, 包括抽象函数剪枝、交易分层变异和种子调度[16,17] 3 个关键部分.

2.1 抽象函数剪枝

为了提高测试效率, 本文分析了智能合约抽象函数在抽象语法树 (AST) 中的结构特征[18], 基于此设计了抽象函数识别算法, 算法流程如算法 1 所示.

算法 1. 基于 AST 的抽象函数识别算法

输入: *AST*: 智能合约的抽象语法树; *Sc*: 当前执行合约的子合约; *ABI*: 当前执行合约的 *ABI* 输入.
输出: *Dict*: *ABI* 输入与是否为抽象函数的映射.

```
1  for each contract in AST do
2    If Sc.fullyImplemented == false then
3      for each ABI in Sc do
4        Dict ⇓ Sc.ABI is abstract
5      end for
6    elif Sc.contractKind == library then
7      continue
8    else then
9      for each ABI in Sc do
10       if ABI.body == null then
11         Dict ⇓ Sc.CurrentABI is abstract
12       elif ABI.body.statements == null then
13         Dict ⇓ Sc.ABI is abstract
14       else then
15         Dict ⇓ Sc.ABI is not abstract
16       end for
17   end for
```

本文定义了两种抽象函数, 显性抽象函数与隐性抽象函数; 同时定义了 3 种抽象合约, 显性抽象合约、隐性抽象合约以及特殊抽象合约. 具体来说, 显性抽象函数是指, 该函数没有代码块, 在 AST 中该函数的 body 字段为空值; 隐性抽象函数是指, 该函数存在代码块, 但代码块中不存在可执行的代码, 在 AST 中表现为该函数的 body 字段下的 statements 字段为空.

显性抽象合约是指, 合约中的函数不全都是有具体实现的函数, 具体表现为其 contractKind 字段值为 contract, 同时 fullyImplemented 字段为 false. 显性抽象合约通常用于给子合约提供外部合约的调用接口, 或

作为子合约的父合约类型. Solc 编译器在将 solidity 源代码编译成 EVM 字节码时, 显性抽象合约生成的代码为空值, 此类合约无法直接部署到以太坊上, 其功能性体现在与其相关的非抽象合约中, 因此对其剪枝可以提高 EVM 的运行效率. 隐性抽象合约是指, 合约中的所有函数都为隐性抽象函数, 调用该类合约中的函数将开启 EVM, 但不会运行任何有效代码. 特殊抽象合约是指, 合约的 contractKind 字段值不为 contract, 常见的类型有 interface 和 library 等, interface 类型本身作为接口, 其 fullyImplemented 字段为 false, 且内部都是显性抽象函数, 类似于全显性抽象合约; library 类型是引用的库合约, 用以减少重复消耗 gas, Solc 编译器在解析时将不生成 ABI, 则无需生成 ABI 输入. 我们通过对智能合约的静态解析, 在合约和函数层面识别与修剪不体现功能性的子树, 避免冗余开销[19], 可以有效提高测试效率.

2.2 ABI 输入生成

本文通过获取当前 ABI 的函数选择器[20], 然后根据 ABI 所需的参数类型生成输入数据. 函数选择器是函数签名的 Keccak-256 哈希值的前 4 个字节, 用于指定需要调用的函数. 在生成输入时, 首先通过哈希生成函数选择器, 然后根据 ABI 参数类型生成参数, 并将其拼接到函数选择器之后. 参数生成规则如表 1 所示.

2.3 交易分层变异

本文将种子定义为元组 (C, I, S), 其中 C, I, S 分别代表智能合约、ABI 输入和交易序列. 本文对种子分层变异统一调度, 在生成符合语法、包含各种语义的输入的同时, 提高变异的效率.

表 1 ABI 输入参数生成规则参考表

参数类型	生成规则
int/unit/unit(2^n)	按一定概率选择边界输入或随机
string	随机, 不满32字节则用0填充
address	选择外部账户地址或合约地址
bool	选择true或false
byte(1⇒32)/unit[]	若没有给定则默认生成10元素数组

2.3.1 分层变异

如表 2 所示, 本文将智能合约的变异维度[21] 分为 4 层: 函数层、语句层、变量层和运算符层. 其中函数层面的变异算子包括: 函数状态增删与替换、函数修饰增删与替换、返回值删除等 3 种; 语句层面的变异算子包括: 循环控制语句增删与替换、循环控制条件修改、delete 语句的删除、assert 与 require 语句的增删; 变量层面的变异算子包括: 变量存储位置删除与替换、数据类型替换、地址变量替换、布尔变量替换等 4 种; 运算符层面的变异算子包括: 算术运算符替换、条件判断符替换等两种. 在实际变异过程中, 由于部分智能合约的变异点位较少, 可能出现遍历所有变异点位后未对表 2 中定义的变异点位进行操作的情况, 此时将变异模块将随机选择一个点位进行变异. 合约分层变异 ABI 输入按一定概率变异或重新生成, 防止陷入局部最优, 变异策略包括位反转、加减操作、替换.

表 2 智能合约变异参考表

维度	变异对象	操作描述	示例 (原始代码→变异后代码)
函数	函数修饰	增删与替换	function demo(uint a) public→ function demo(uint a) public payable
	函数状态	增删与替换	function demo(uint a) public/external/internal/private → function demo(uint a)
	函数返回值	删除	function demo(uint a) public returns (uint) {… return a;} → function demo(uint a) public {…}
语句	循环控制语句	增删与替换	for(i=0; i < 10; ++i) {… break;} → for(i=0; i < 10; ++i) {… continue;}
	循环控制条件	修改	for(i=0; i < 10; ++i)→ for(i=0; i < 100; ++i)
	delete语句	删除	function demo(uint a) public {… delete b;}→ function demo(uint a) public {…}
	assert语句	增加与删除	function demo(uint a) public {…assert(a<5);}→ function demo(uint a) public {…}
	require语句	增加与删除	function demo(uint a) {… require(a==1);}→ function demo(uint a) {…}
变量	变量存储位置	替换	string memory y;→ string storage y;
	数据类型	替换	function demo(uint a) {uint8 x;}→ function demo(uint a) {uint256 x;}
	地址变量	替换	require(msg.sender == addr);→ require(tx.origin == addr);
	布尔变量	替换	return True;→ return False;
运算符	算术运算符	替换	uint c = a + b;→ uint c = a - b
	条件判断符	替换	if (a < b) {…}→ if (a > b) {…}

交易序列变异[22] 如图 4 所示, 序列 1 逻辑是依次部署智能合约, 并顺序对其调用, 默认情况下选择序列 1. 基于智能合约抽象函数在抽象语法树中的结构特征, 若合约 A 中存在 selfdestruct 函数则按一定概率选择

序列 2, 序列 2 是在序列 1 基础上插入触发 A 中 selfdestrut 函数调用的交易 1 和对 A 发送 ether 交易 2. 若合约 A 没有定义回退函数则按一定概率选择序列 3, 序列 3 是在 1 的基础上插入对 A 的转账, 此转账交易的位置在对合约 A 调用之后.

本文动态地给变异种子中的智能合约和 ABI 输入分配权重[23], 并依照权重策略进行选择, 实现对算子的动态调度, 以提高变异效率. 在对智能合约和 ABI 输入变异前, 首先给所有变异算子都分配相同的初始分数, 并按照每个算子的分数在所有算子的总分数中的占比进行权重计算, 如果采用某算子生成的种子提升了覆盖率, 则提高该算子的分数. 我们通过结合权重策略按不同粒度的变异, 在尽量保证输入合法性[24] 的情况下使得变异更加充分, 能更有效探索难以到达的分支, 因此能更完整更深入地测试 EVM 的逻辑.

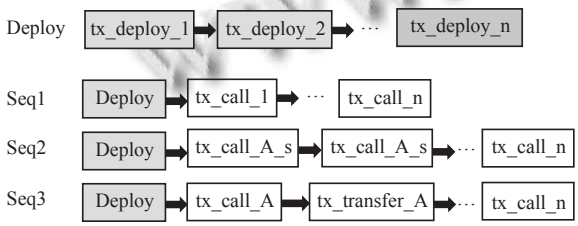

图 4 交易序列

2.3.2 种子调度

种子调度算法如算法 2 所示, 本文通过根据每一轮模糊测试中 EVM 代码覆盖率的变化情况和测试用例的运行时间对种子选择, 达到提高变异效率的目的.

算法 2. 种子调度算法

输入: *seed*: 合约, 输入, 序列; *iterMax*: 最大变异次数.
输出: *seedQueue*: 种子队列.

```
for iter in range(maxIter) do
    pri_new = pri(seed_iter)
    if pri_new > 1 do
        seedQueue ⇓ seed_iter
        sort(seedQueue)
        return seedQueue
        record.cov = get_cov[seed_iter]
end for
```

在每轮变异后对种子的评估值计算[25], 评估值基于其对覆盖率和测试效率带来的正向影响的估量, 评估值计算如下:

$$pri(seed)=\frac{cov'}{cov}+n\left(1-\frac{cov'}{cov}\right)\times\frac{t}{m+t'} \tag{1}$$

其中, *cov* 为代码覆盖率, *t* 为单个合约运行时间, *m*、*n* 为经验数值, 本文分别将其设定为 0.8 和 5. 若 *pri* 大于 1 则对 *record* 更新并将其加入种子队列, 再从种子队列选择新种子开始下一轮变异. 以覆盖率和测试效率作为对种子的选择的正向反馈能明显提高覆盖率增长速率, 但由于对所有路径分支分配的权重是相同的, 可能导致 ETHCOV 更倾向 fuzz 高频路径.

3 实验

本文方案的交易执行后端是面向 Geth 实现的, 同时为了实现 EVM 的快速执行, 实验中不执行 RPC 调用, 而是修改为本地发送的方式.

3.1 ETHCOV 与 EVMFuzzer 对比测试

为了验证 ETHCOV 工具的检测效果, 本文同时使用 ETHCOV 和 EVMFuzzer 在相同系统环境下, 使用相同的实验数据集进行测试, 对 EVM 的代码覆盖率与测试效率进行比较.

3.1.1 实验环境

为了保证对比实验结果的公平性, 本文中所有的实验都在统一的实验环境下进行, 实验的硬件环境如表 3 所示.

表 3 实验主机硬件参数

配置项目	规格参数
CPU型号	Intel(R) Core(TM) i7-10750H
CPU主频	2.60 GHz
内存	16 GB
硬盘	800 GB

3.1.2 实验结果

本文通过构建以太坊状态测试用例对 EVM 的完整执行流程进行了测试, 相较于 EVMFuzzer 而言, 更全面地覆盖了 EVM 的工作流程, 明显地提升了代码覆盖率. 此外, 本文相较于 EVMFuzzer 采用了更丰富的变异算子, 设计了对应的 ABI 输入生成与变异算法, 并通过策略权重和种子调度算法提高了变异的利用率[26], 能够提高在测试过程中获得的代码覆盖率增长量. 在本文的实际实验中, 每个智能合约执行带来的覆盖率增长通常会在 20 轮左右变异收敛, 因此本文设置测试轮数为 20 轮, 以提高测试效率.

实验结果如表 4 所示, 执行过程中的覆盖率变化情况如图 5 所示. 在相同实验条件下, 我们将 ETHCOV 与 EVMFuzzer 进行了实验对比. 实验结果表明, 对于

相同目标程序 ETHCOV 的代码覆盖率为 39.35%, 相较于 EVMFuzzer 提升了 131.61%. 在执行过程中, EVMFuzzer 的初始覆盖率为 13.31%, 最终覆盖率为 16.99%, 共增长了 3.68 个百分点; ETHCOV 的初始覆盖率为 31.08, 最终覆盖率为 39.35, 共增长了 8.27 个百分点, 为 EVMFuzzer 的 2.25 倍.

表 4 代码覆盖率对比 (%)

实验对象	代码覆盖率
EVMFuzzer	16.99
ETHCOV	39.35

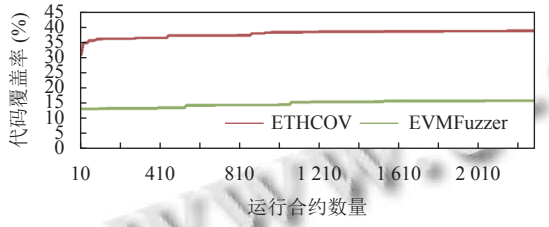

图 5 EVM 覆盖率变化曲线对比

上述实验结果表明, 相对于 EVMFuzzer, ETHCOV 具有更高的代码覆盖率和代码覆盖率增量[27,28], 从而证明了本文的方法能够覆盖更完整的 EVM 工作流程, 并有效地提升变异的利用率.

对于 EVM 的核心组件集 (core/vm), 我们通过对组件中函数被调用层级与函数内部覆盖率计算, 来量化对比输入是否能达到虚拟机内部的深层逻辑, 计算公式见式 (2). 其中 $f(i)$ 代表函数被调用层级, $q(i)$ 代表函数内部覆盖率. 如图 6 所示, ETHCOV 生成的输入明显优于 EVMFuzzer.

$$score = \sum_{i=1}^{n} \ln(f(i)+1) \times q(i) \tag{2}$$

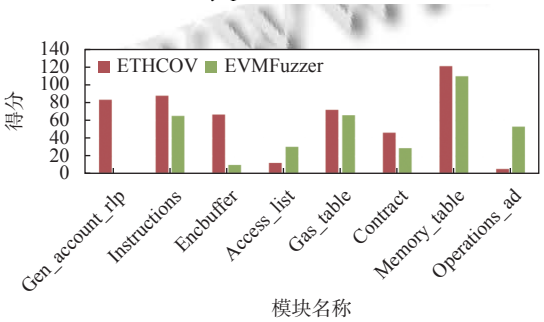

图 6 深层逻辑触发率量化对比

与 EVMFuzzer 的差分模糊测试方法相比, 本文的方法在两个方面提升了测试的效率.

(1) 基于 AST 的抽象函数剪枝.

(2) 智能合约层次结构的流程优化.

EVMFuzzer 在设计实现中没有考虑到智能合约的层次结构问题, 在其实现方案中, 采用了以合约为变异范围、以函数为运行粒度、以合约为运行范围的组合方式, 在对单个合约执行中 EVM 的实际启停次数将与合约的函数个数相同, 即每个智能合约的执行平均需要启停 54 次 EVM. 而本文以智能合约为运行粒度, 每个智能合约的运行仅需经过 1 次 EVM 启停, 因此可以让虚拟机启停的时间开销显著减少.

实验结果如表 5 所示, ETHCOV 的测试效率是 EVMFuzzer 的 4.39 倍, 说明基于本文的方法可以有效提高测试效率.

表 5 测试效率对比 (s)

测试对象	测试耗时
EVMFuzzer	682 276.18
ETHCOV	155 565.90

3.2 对比分析

将 4 214 个 solidity 源文件, 27 712 个合约作为 ETHCOV 的输入, 在 ETHCOV 对 EVM 输入相同的交易序列后, 两个不同版本的 EVM 出现不同的输出.

对 Geth1.10.6 版本的测试, 返回 alloc 中的 storage 字段如下.

```
"storage": {
"0x49fd7bc01184b95fc5bec7ba1b9e37cb69fa39a85e1c9795bac9bb1402cc39ad":
"0x49fd7bc01184b95fc5bec7ba1b9e37cb69fa39a85e1c9795bac9bb1402cc39ad" }
```

不同于对 Geth1.10.16 版本的测试结果, alloc 中的 storage 字段如下.

```
"storage": {
"0x7fc59c42eff19b618e9158be1222e8b755a47254b319e824c37b4a10ee7539b5":
"0x7fc59c42eff19b618e9158be1222e8b755a47254b319e824c37b4a10ee7539b5" }
```

以上对比结果说明对于相同的交易, 两个版本的客户端分别写回了不同的结果. 根据我们对真实以太坊环境调研, 以太坊区块链非最新版本客户端节点占比超过 50%, 若节点存储的数据不一致可能导致严重的结果如共识错误.

除此之外, ETHCOV 还发现了 Geth1.10.16 版本

的 EVM 在本地运行时的未知问题.

(1) 其编译模块在某种情况下将本地目录写入编译生成的 bincode 中, 导致运行报错.

(2) 在某次 EVM 运行交易过程中丢失了区块 hash 信息.

3.3 小结

在相同实验条件下, 我们将 ETHCOV 与 EVM-Fuzzer 进行了实验对比. 实验结果表明, 对于相同目标程序, ETHCOV 的代码覆盖率为 39.35%, 较于 EVM-Fuzzer 提升了 131.61%. 同时, ETHCOV 的测试效率是 EVMFuzzer 的 4.39 倍. 在对 EVM 多版本对比测试中, ETHCOV 通过生成和变异交易序列触发 EVM 的执行, 并对输出的结果进行对比检验, 发现 Geth1.10.6 版本的执行错误以及 Geth1.10.16 在本地运行时的未知问题.

4 结束语

本文设计了交易序列分层变异方法, 结合权重策略使得生成有效的输入能完整地测试 EVM 的运行机制并能到达虚拟机内部的深层逻辑, 同时, 本文通过抽象函数剪枝方法与检测流程优化, 极大提高了测试效率. 最后, 我们期待该工具能发现更多不同版本 EVM 和基于 EVM 实现的区块链虚拟机的安全问题, 帮助开发者修复区块链虚拟机缺陷.

参考文献

1 Hildenbrandt E, Saxena M, Rodrigues N, *et al*. KEVM: A complete formal semantics of the Ethereum virtual machine. Proceedings of the 31st IEEE Computer Security Foundations Symposium. Oxford: IEEE, 2018. 204–217. [doi: 10.1109/CSF.2018.00022]

2 Buterin V. Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/zh/whitepaper/. [2022-09-06].

3 Wood G. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, 2014, 151: 1–32.

4 Antonopoulos AM, Wood DG. Mastering Ethereum: Building Smart Contracts and DApps. Sebastopol: O'Reilly Media, 2018. 204–206.

5 Saad M, Spaulding J, Njilla L, *et al*. Exploring the attack surface of blockchain: A comprehensive survey. IEEE Communications Surveys & Tutorials, 2020, 22(3): 1977–2008. [doi: 10.1109/COMST.2020.2975999]

6 Fu Y, Ren M, Ma FC, *et al*. EVMFuzzer: Detect EVM vulnerabilities via fuzz testing. Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Tallinn: ACM, 2019. 1110–1114. [doi: 10.1145/3338906.3341175]

7 Ma FC, Ren M, Fu Y, *et al*. Security reinforcement for Ethereum virtual machine. Information Processing & Management, 2021, 58(4): 102565.

8 Cassez F, Fuller J, Ghale MK, *et al*. Formal and executable semantics of the Ethereum virtual machine in Dafny. Proceedings of the 25th International Symposium on Formal Methods. Lübeck: Springer, 2023. 571–583.

9 Yang WC, Peng J. Research on EVM-based smart contract runtime self-protection technology framework. Proceedings of the 34th International Conference on Web, Artificial Intelligence and Network Applications. Caserta: Springer, 2020. 617–627. [doi: 10.1007/978-3-030-44038-1_57]

10 Kalodner HA, Goldfeder S, Chen XQ, *et al*. Arbitrum: Scalable, private smart contracts. Proceedings of the 27th USENIX Security Symposium. Baltimore: USENIX Association, 2018. 1353–1370.

11 贺海武, 延安, 陈泽华. 基于区块链的智能合约技术与应用综述. 计算机研究与发展, 2018, 55(11): 2452–2466.

12 Liao JW, Tsai TT, He CK, *et al*. SoliAudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS). Granada: IEEE, 2019. 458–465.

13 Liu J, Li PL, Cheng R, *et al*. Parallel and asynchronous smart contract execution. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(5): 1097–1108. [doi: 10.1109/TPDS.2021.3095234]

14 Mehar MI, Shier CL, Giambattista A, *et al*. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. Journal of Cases on Information Technology, 2019, 21(1): 19–32. [doi: 10.4018/JCIT.2019010102]

15 Zhou SF, Yang ZM, Xiang J, *et al*. An ever-evolving game: Evaluation of real-world attacks and defenses in Ethereum ecosystem. Proceedings of the 29th USENIX Security Symposium. USENIX Association, 2020. 2793–2810.

16 张阳, 佟思明, 程亮, 等. 模糊测试改进技术评估. 计算机系统应用, 2022, 31(10): 1–14. [doi: 10.15888/j.cnki.csa.008680]

17 杨克, 贺也平, 马恒太, 等. 有效覆盖引导的定向灰盒模糊

测试. 软件学报, 2022, 33(11): 3967–3982. [doi: 10.13328/j.cnki.jos.006331]

18 He JX, Balunović M, Ambroladze N, *et al*. Learning to fuzz from symbolic execution with application to smart contracts. Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. London: ACM, 2019. 531–548.

19 Manès VJM, Han H, Han C, *et al*. The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering, 2021, 47(11): 2312–2331. [doi: 10.1109/TSE.2019.2946563]

20 Jiang B, Liu Y, Chan WK. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). Montpellier: IEEE, 2018. 259–269.

21 Jiang B, Li ZC, Huang YH, *et al*. WasmFuzzer: A fuzzer for WasAssembly virtual machines. Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering. Pittsburgh: KSI Research Inc., 2022. 537–542.

22 Lin IC, Liao TC. A survey of blockchain security issues and challenges. International Journal of Network Security, 2017, 19(5): 653–659. [doi: 10.6633/IJNS.201709.19(5).01]

23 林敏, 张超. 针对 WebAssembly 虚拟机的模糊测试方案. 网络安全技术与应用, 2020, (6): 15–18.

24 欧阳丽炜, 王帅, 袁勇, 等. 智能合约: 架构及进展. 自动化学报, 2019, 45(3): 445–457.

25 王文硕, 程亮, 张阳, 等. 基于函数重要度的模糊测试方法. 计算机系统应用, 2021, 30(11): 145–154. [doi: 10.15888/j.cnki.csa.008127]

26 Peng H, Shoshitaishvili Y, Payer M. T-Fuzz: Fuzzing by program transformation. Proceedings of the 2018 IEEE Symposium on Security and Privacy. San Francisco: IEEE, 2018. 697–710. [doi: 10.1109/SP.2018.00056]

27 Li J, Zhao BD, Zhang C. Fuzzing: A survey. Cybersecurity, 2018, 1(1): 6. [doi: 10.1186/s42400-018-0002-y]

28 Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna: ACM, 2016. 1032–1043.

(校对责编: 孙君艳)