中国科学: 技术科学
SCIENTIA SINICA Technologica
techcn.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS
论 文
群体智能激发汇聚及应用专辑

基于流形鸽群优化的智能合约重入性漏洞检测方法研究

刘方青[1], 黄翰[1*], 向毅[1], 郝志峰[2,3]

1. 华南理工大学软件学院, 广州 510006;
2. 汕头大学计算机学院, 汕头 515063;
3. 广东工业大学计算机学院, 广州 510006
* E-mail: hhan@scut.edu.cn

收稿日期: 2021-08-10; 接受日期: 2021-11-04
国家自然科学基金项目(批准号: 61772225, 61876207)、广州科技项目(编号: 201802010007)和广东省重点领域研发计划(编号: 2018B010109003)资助

摘要 重入性漏洞在智能合约中普遍存在, 可能造成巨大的经济损失. 现有的基于符号执行的静态分析工具通过预设的规则判断漏洞是否存在, 然而预设规则不全面可能会导致重入性漏洞的误报. 为了避免误报, 本研究尝试从软件测试用例生成的动态分析角度解决这一难题. 本文将该应用场景抽象为存在重入性循环路径的路径覆盖测试用例自动生成问题, 通过生成并执行覆盖重入性循环路径的测试用例来检测重入性漏洞. 以鸽群算法为代表的群体智能算法是求解测试用例生成这类黑盒优化问题的常用方法. 鸽群算法在整个决策空间内围绕种群最优解邻域搜索, 然而, 问题的最优解可能并不在该邻域内, 导致路径覆盖率较低. 为了提升鸽群算法的路径覆盖率, 本文将利用流形启发式算子改进鸽群算法, 使其分配更多的算力搜索与优化目标相关的子空间, 从而提升鸽群算法求解效率, 覆盖重入性循环路径. 实验结果显示, 改进后的流形鸽群算法能够更加高效地生成覆盖重入性循环路径的测试用例, 检测出被测合约的重入性漏洞. 与Oyente, Securify和Smartcheck这三个智能合约测试工具相比, 本文提出的方法能够有效避免重入性漏洞的误报, 在实验的8个被测试智能合约中重入性漏洞识别准确率分别提升12.5%, 12.5%和25%.

关键词 智能合约, 重入性漏洞, 鸽群算法, 测试用例自动生成, 路径覆盖

1 引言

区块链由Nakamoto[1]于2008年提出, 作为一种重要的金融交易数据库应用于云计算[2]与智能合约[3,4]等多种场景. 近年来, Ethereum[1)]与Parity[2)]等区块链平台[5,6]在加密货币上的交易受到越来越多的关注. 然

1) https://www.ethereum.org/
2) https://www.parity.io/

引用格式: 刘方青, 黄翰, 向毅, 等. 基于流形鸽群优化的智能合约重入性漏洞检测方法研究. 中国科学: 技术科学, 2022, 52
Liu F Q, Huang H, Xiang Y, et al. Smart contract reentrancy vulnerability detection method based on manifold pigeon optimization algorithm (in Chinese). Sci Sin Tech, 2022, 52, doi: 10.1360/SST-2021-0365

而, 许多基于智能合约的加密货币平台交易却存在潜在安全漏洞, 可能造成重大经济损失. 例如, 著名的The DAO攻击就给Ethereum与Parity两个智能合约社区分别造成了3600000以太币(Ethers)和500000 Ethers的直接经济损失[7,8], 后续损失更是难以估计. 因此, 找出智能合约系统中潜在的漏洞至关重要.

智能合约作为一种新兴软件程序[9], 其原则与机制还不完善[10], 更容易产生漏洞. 此外, 与可以被删除或修改的一般程序代码不同, 基于区块链的智能合约系统一旦部署则不可修改[11], 无法对存在漏洞的已部署合约进行修复[12]. 这就需要测试人员在智能合约正式部署前, 对其进行充分的测试[13], 增强智能合约的可靠性. 在智能合约已知存在的漏洞中, 由于代码意外回调(callback)[14]造成资金流失的现象称为重入性漏洞, 正是该漏洞造成了The DAO攻击事件.

现有的智能合约重入性漏洞检测方法主要包括专家人工检测以及静态分析工具两类. 专家人工检测存在CoinMercenary[3], Decenter[4], SmartContractAudits[5]等许多面向智能合约的测试服务, 这些服务通过专家人工评估智能合约的可靠性. 由于需要人工参与审核, 服务会消耗大量的人力资源与评估时间, 若用户需求复杂, 则评估时间将更长. 因此, 能够自动评估和测试智能合约的服务或者工具是十分必要的.

为了检测出智能合约潜在的重入性漏洞, 相关研究人员提出了Oyente[8], Securify[15], Smartcheck[16]等基于符号执行的静态分析工具. 符号执行技术[17,18]采用约束求解器求得每一条路径的约束表达式, 得到路径覆盖测试用例, 并基于预设的规则检测潜在的重入性漏洞. 但符号执行方法由于没有实际执行合约代码, 只能通过预设的规则判断合约是否存在重入性漏洞, 对于部分已经修复的合约可能会出现误报的情况; 相对而言, 测试用例自动生成(automated test case generation, ATCG)以执行测试用例的方式检测重入性漏洞, 能够一定程度上避免误报这类问题.

ATCG作为一种结构测试方法, 属于单元测试的范畴[19~21], 其目标是在使用尽量少的测试用例开销的情况下, 生成满足覆盖准则的测试用例. 给定一个被测试程序(system under test, SUT), 其覆盖准则通常可以分为语句覆盖、分支覆盖和路径覆盖等类型, 路径覆盖是其中难度最高、有效性最强的准则[22,23]. 然而, 现有的路径覆盖测试用例自动生成(automated test case generation based on path coverage, ATCG-PC)问题模型[24,25]的目标路径中可能并不包含存在重入性漏洞的路径. 因此, 本文将改进ATCG-PC问题模型, 针对存在智能合约重入性漏洞的应用场景进行抽象.

群体智能算法是求解ATCG-PC问题的一类主要方法[26], 其中蚁群算法[27]、粒子群算法[28]与人工蜂群算法[29]等群体智能算法均应用于自动生成路径覆盖测试用例. 鸽群算法(pigeon-inspired optimization, PIO)[30,31,32]作为一种新兴的群体智能算法拥有较强的局部搜索能力, 成功应用于求解路径规划[33]、最优潮流问题[34]与飞行器气动设计[35]等大规模黑盒优化场景, 并展现出优于改进的粒子群算法、蚁群算法等群体智能算法的表现. 在求解ATCG-PC这一大规模黑盒优化问题时鸽群算法在收敛性上存在一定优势, 而提升鸽群算法求解ATCG-PC问题性能的关键在于找出与优化目标相关的子空间, 减少鸽群算法向其他空间分配的搜索算力, 节省算法的开销.

本文主要从ATCG-PC问题的智能合约重入性循环路径建模以及鸽群算法改进两个方向进行研究. 针对现有ATCG-PC问题模型无法检测智能合约重入性漏洞的现状, 本文提出存在重入性循环路径的ATCG-PC模型. 该模型为智能合约程序控制流图(control flow graph, CFG)新增额外边, 并为测试用例新增变量维度, 使修改后的ATCG-PC模型能够通过重入性循环路径模拟重入性漏洞攻击. 同时, 本文通过流形启发式算子[36]改进了鸽群算法, 并将其应用于生成重入性循环路径测试用例, 通过对比测试用例运行后实际结果与预期结果的差异来判断重入性漏洞是否存在. 为了检验本文提出的方法的准确性, 将在8个智能合约案例[7,8,25]上与Oyente等开源测试工具进行对比. 实验结果显示, 本文提出的方法在被测智能合约案例上的重入性漏洞识别率高于所有对比工具.

本文针对智能合约应用特征, 将重入性漏洞检测问题抽象为存在重入性循环路径的ATCG-PC问题, 在此基础上改进鸽群算法求解该问题. 第2节介绍智能测

3) https://www.coinmercenary.com/smart-contract-auditing/
4) https://www.decenter.com/
5) https://www.smartcontractaudits.com/

试相关的领域知识, 包括检测智能合约重入性漏洞定义、相关检测工具的研究现状. 第3节介绍存在重入性循环路径的ATCG-PC问题数学模型, 包括模型定义与重入性漏洞检测案例等内容. 第4节介绍求解存在重入性循环路径ATCG-PC问题的流形鸽群算法. 第5节通过实验验证的方式评估本文方法生成路径覆盖测试用例以及检测智能合约重入性漏洞的能力. 第6节总结论文工作并探讨未来研究方向.

2 智能合约重入性漏洞问题与检测方法

本节介绍智能合约重入性漏洞的定义与相关检测工具的现状.

定义1(重入性漏洞): 在智能合约中, 允许攻击者无限期调用call.value()语句从而恶意盗刷资金的漏洞称为重入性漏洞[8,37,38].

智能合约中存在一个特殊的fallback函数, 当执行一些特定的函数, 如call.value(), send, transfer时, 会自动调用fallback函数. 如图1所示, 在攻击者发布的合约代码中, fallback和正常代码模块都会调用call.value()函数. 但是, call.value()与其他两个函数不同, 如果send或transfer函数被调用, 那么其触发的fallback函数最多只能消耗2300 gas[6]; 而call.value()则会尽可能使用gas, 直到合约账户金额耗尽.

智能合约重入性漏洞检测工具可以大致分为两类: 一类是动态测试工具, 另一类是静态测试工具. 在动态测试工具中, 智能合约代码必须经过编译后以执行代码的方式进行检测; 同时, 这类方法需要测试人员将相关工具部署在智能合约的运行环境中才能运行. 例如, Grossman等人[14]提出了无回调对象的在线监测方法, 重点检测重入性漏洞. 不同于通过执行代码分析程序的动态测试工具, 静态分析工具则是基于符号执行这类经典的静态分析技术. Luu等人[8]设计了Oyente自动安全分析工具, 该工具以符号执行方法检测智能合约的安全漏洞. Jiang等人[10]则设计了一款基于模糊分析的工具Contracufuzzer, 用于检测Ethereum智能合约的安全漏洞. Kim和Lee[39]提出了一种为智能合约分析工具自动生成测试用例的方法, 辅助工具检测重入性漏洞. 然而, 如果生成的测试用例中不包含重入性循环路径, 则可能导致检测误差.

不同于动态测试方法, 符号执行不需要模拟智能合约的运行环境, 因此, 在一些情况下, 符号执行[8]比动态测试更加容易实现. 但是, 符号执行方法存在工具的计算时间随程序路径数量的增加而指数增长的问题, 同时基于符号执行的静态分析方法还可能会出现重入性漏洞的误报[40]. 两种测试方式各有优劣. 上述智能合约测试工具总结如表1所示.

3 存在重入性循环路径的ATCG-PC问题模型

3.1 模型相关定义

本节介绍存在重入性循环路径的ATCG-PC问题模型与基于该模型的智能合约重入性漏洞检测方法.

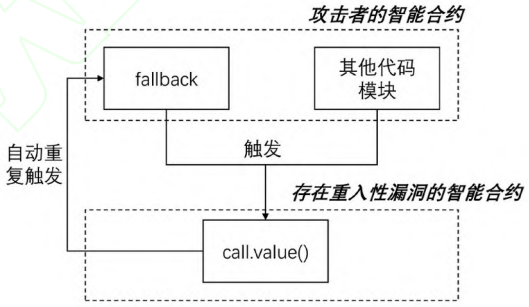

图 1 重入性漏洞示例

Figure 1 An example of reentrancy vulnerability.

表 1 智能合约检测工具总结

Table 1 Review of smart contract detection toolkits

工具名称	分析基础	主要技术
Oyente[8]	字节码	符号执行
MAIAN[7]	字节码	符号执行与具体验证器
Securify[15]	字节码与源代码	符号执行
Smartcheck[16]	源代码	静态分析
Mythril	字节码与源代码	并发分析、污点分析与控制流检查
Contractfuzzer[10]	ABI规范[7]	模糊测试
ZEUS[41]	源代码	抽象介绍与符号模型检查

6) Gas用于衡量执行一项操作需要向网络支付的费用.

7) 合约应用二进制接口(ABI)是与Ethereum生态系统中的合约交互标准, 既可以从区块链外部进行交互, 也可以实现合约之间的交互.

本文改进的鸽群算法在存在重入性循环路径ATCG-PC问题模型的基础上生成路径覆盖测试用例, 通过对比测试用例执行实际结果与预期结果, 检测智能合约是否存在重入性漏洞. 其主要流程如图2所示.

重入性漏洞攻击是智能合约的外部事件, 一般的软件工具很难检测到这种漏洞. 为了解决这一问题, 本文提出了一种能够反映这种外部事件的、存在重入性循环路径的ATCG-PC问题模型, 模型的相关定义如下所示.

定义2(测试用例): 一个测试用例$X_i=(x_{i,1}, x_{i,2}, \cdots, x_{i,n}, \pi)$可以由一串长度为$n+1$的整数与浮点数的混合向量表示, 其中$(x_{i,1}, x_{i,2}, \cdots, x_{i,n})$是原始ATCG-PC问题的测试用例, 而$\pi$是新增的维度, 用于模拟重入性漏洞攻击.

定义3(路径): 路径由一串节点的运行序列表示, 序列从初始节点开始, 到终点节点结束. 在测试的智能合约中, 每个顶点都有其分支条件, 在一条路径中不能出现两次.

定义4(路径覆盖): 测试用例X_i覆盖路径p_j意味着, 测试用例X_i与路径p_j在被测试函数中有相同的程序语句以及分支执行顺序.

定义5(原边): 原边是一对有序的节点, 这些边由原始智能合约代码生成.

定义6(额外边): 额外边是一对有序的节点, 由新增的测试用例维度π所触发重入性漏洞攻击节点为起点, 重复调用取款操作节点为终点.

定义7(重入性循环路径): 重入性循环路径是智能合约中经过额外边, 触发重入性漏洞的路径. 在重入性循环路径中额外边仅经过一次, 不会重复进入.

为了模拟攻击者行为, 论文提出的存在重入性循环路径的ATCG-PC模型将分别在原智能合约的控制流图与测试用例中增加一条额外边与额外的维度变量π. 该模型通过新增测试用例变量π触发模拟的智能合约重入性漏洞攻击, 以重复调用call.value()等取款操作造成重入性漏洞. 而本文方法将通过生成路径覆盖测试用例, 对比测试用例执行预期结果与实际结果的方式检测重入性漏洞.

模型仅考虑路径中首次出现的call.value()函数. 由于call.value()函数可能出现在两条完全不同的路径中, 额外边以模拟重入性漏洞攻击节点为起点, 以call.value()函数节点为终点. 当执行call.value()函数时, 程序以一定的概率选择沿原边或额外边继续执行, 选择概率由测试用例的新维度π的值决定.

3.2 存在重入性循环路径的ATCG-PC问题模型

本节描述程序路径编码以及存在重入性循环路径的ATCG-PC问题模型, 模型相关变量符号与描述如表2所示.

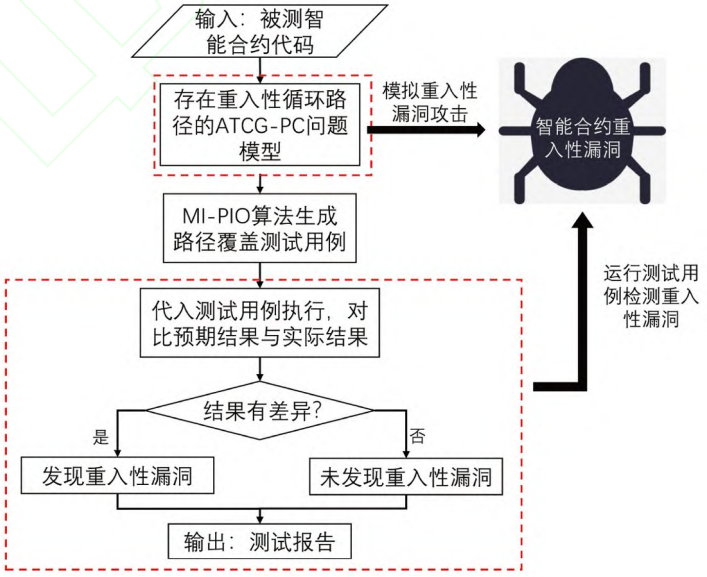

图 2 (网络版彩图)基于ATCG-PC问题求解的重入性漏洞检测方法

Figure 2 (Color online) Reentrancy vulnerability detection method based on ATCG-PC.

表 2 模型符号及其定义

Table 2 Definitions of variables in the model

符号	描述
k	SUT中分支节点的个数
c	测试用例集合的路径覆盖率
l	测试用例集合覆盖的路径数目
L	SUT中所有可达路径总数
m	算法生成的测试用例数目
N	SUT中可行测试用例数目
Max	算法测试用例开销上限
$\overline{X}$	SUT中所有可行测试用例的集合
$\overline{P}$	SUT中所有可达路径的集合
$\overline{V}$	SUT中所有分支节点的集合
$\mathrm{fitness}_j(X_i)$	测试用例X_i在路径p_j上的评估值
$f_j^v(X_i)$	X_i以SUT第j条路径为目标时, 在第v个分支节点的评估值
p_j	$\overline{P}$中第j条路径
X_i	$\overline{X}$中的一个测试用例
π	测试用例的新增变量, 模拟重入性工具的概率
n	原始测试用例变量维度的数目
$x_{i,j}$	测试用例X_i的第j维变量
ε	一个避免分母为零的常数
λ_j	路径p_j所经过的分支数量
θ_{ij}	一个中间变量, 记录X_i覆盖p_j的情况
K	评估函数计算的一个常数

路径编码用于表示SUT在本文所提出的ATCG-PC问题数学模型中的路径. 对于给定的智能合约函数, 将所有分支节点分别分配一个1~k的唯一编号, 其中k表示节点数量. 基于定义3, 智能合约函数中的所有路径都可以使用一个包含k个字符的字符串序列表示. 其中路径p的第v个字符值为“2”表示路径p没有经过第v个节点, 而字符值为“0”或“1”表示路径p在第v个顶点的不同运行方向(如Yes或No).

给定一个被测试的智能合约函数, $\overline{X}=\{X_1, X_2, \ldots, X_N\}$表示所有可行测试用例的集合, $\overline{P}=\{p_1, p_2, \ldots, p_L\}$表示智能合约函数中所有可达路径的集合, $\overline{V}=\{V_1, V_2, \ldots, V_k\}$表示程序所有分支节点的集合, 其中$N$, L, k分别表示所有可行测试用例数目、程序可达路径的数目、分支节点数目. 面向智能合约函数的ATCG-PC问题目标可被重新表述为

Maximize c

满足约束:

$$c=\frac{l}{L}\times 100\%, \tag{1}$$

$$l=\sum_{j=1}^{L}\min\left\{1,\sum_{i=1}^{m}\theta_{ij}\right\}, \tag{2}$$

$$m\le Max, \tag{3}$$

$$\theta_{ij}=\begin{cases}1, \mathrm{fitness}_j(X_i)=\lambda_j\times\dfrac{1}{\varepsilon},\\ 0, \text{ otherwise},\end{cases} \tag{4}$$

$$\sum_{j=1}^{L}\theta_{ij}=1, \tag{5}$$

$$\mathrm{fitness}_j(X_i)=\sum_{v=1}^{k}f_j^v(X_i), \tag{6}$$

$$i=1,2,\ldots,m;\ j=1,2,\ldots,L;\ v=1,2,\ldots,k. \tag{7}$$

该数学模型的优化目标是在给定的测试用例开销范围内, 实现路径覆盖率c最大化. 约束式(1)定义了路径覆盖率的计算公式. 约束式(2)给出了生成测试用例所覆盖的路径数量统计方法, 其中变量L表示被测试智能合约中可达路径总数; θ_{ij}表示算法生成的第i个测试用例X_i覆盖第j条路径p_j的情况, 根据约束式(4)可知当X_i覆盖p_j时θ_{ij}值为1, 其他情况取值均为0. 约束式(3)定义了生成的测试用例数m不会超过预设的最大测试用例开销Max, 而Max是一个预设常数, 其取值参照相关文献设置[23,42]. 约束式(4)和(5)确保每个测试用例覆盖有且仅有一条路径. 约束式(6)定义了程序第j条路径的评估函数计算方法, 其中$f_j^v(X_i)$表示测试用例X_i以程序第j条路径为目标时, 在被测试程序第v个分支节点的评估值, 评估函数的详细计算方法方法将在第4.3节详细介绍. 约束式(7)则定义了测试用例的相关决策变量的取值范围.

3.3 基于ATCG-PC重入性漏洞检测的案例分析

基于存在重入性循环路径的ATCG-PC问题模型, 以下内容将展示论文所提出的方法检测智能合约重入性漏洞的具体案例. 图3(a)展示了一段智能合约示例程序的原始控制流图, 而图3(b)展示了基于存在重入性循环路径的ATCG-PC问题模型而修改的程序控制流图. 该示例合约取自一个典型的重入性漏洞分析案

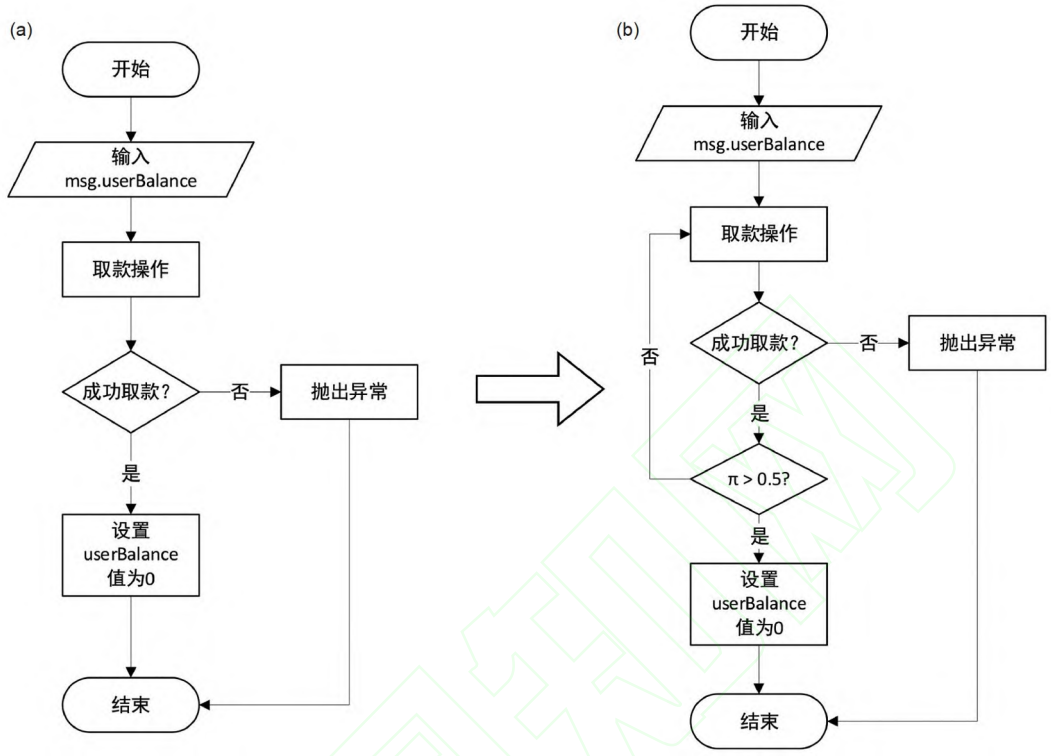

图 3 合约案例控制流图原始(a)与修改(b)对比

Figure 3 Original (a) and modified (b) control flow charts of a smart contract.

例[3], 该案例的代码可在相关开源社区[8]下载.

图3(a)展示的案例存在重入性漏洞. 图3(a)中msg表示调用该函数的合约地址, 调用者的目标是从该合约中提取余额. 作为输入维度之一的userBalance表示向目标账户取款的数值. 正常使用情况下, 若userBalance值或账户余额已经归零, 调用该函数时会报出提取余额失败的结果; 而合约攻击者却可以在提取出账户余额后, 从设计好的智能合约返回函数起点, 重复调用call.value()函数, 不断地套取资金.

图3(b)展示的是被测试程序修改后的控制流图, 该图能够模拟智能合约重入性漏洞攻击. 对比图3(a)和(b)两个控制流图, 可以发现图3(b)存在额外的一个分支节点与一条额外边, 该节点通过新输入的测试用例维度π模拟重入性漏洞攻击, π的取值范围为[0,1], 若$\pi \leq 0.5$则表示模拟黑客的重入性漏洞攻击, 此时路径将经过额外边重新跳转回到判断语句中, 并且调用call.value()重复套取资金. 在测试用例成功覆盖包含额外边的路径后, π的值将会设置为1, 保障测试用例在下一次循环执行过程中跳出循环, 并保障SUT路径数量固定. 由于受到重入性漏洞攻击的用户预期银行余额会与实际值不同, 可以在测试智能合约函数时发现漏洞.

表3展示了一组基于图3(b)的测试用例以及对应的路径编码等内容. 结合表3与图3(b)中的信息可以展示本文检测智能合约重入性漏洞的方法, 即通过对比预期银行账户余额与实际银行账户余额, 判断被测试的智能合约函数是否存在重入性漏洞. 所有账户初始余额均为1000 Ethers. 路径编码为“1222”的测试用例中userBalance值为零, 因此, 合约调用后取款不成功, 将直接抛出取款异常. 路径编码为“0022”与“0100”的两个测试用例对应账户预期银行余额均为900 Ethers, 但“0100”对应测试用例实际银行余额却是800 Ethers, 该测试用例对应账户余额与预期值不符, 可以断定该路径编码为“0100”的测试用例发现了被测试智能合约

8) https://github.com/crytic/not-so-smart-contracts

表 3 被测试智能合约路径编码与测试用例示例

Table 3 Tested smart contract path encoding and their test cases

路径编码	测试用例(userBalance, π)	预期账户余额	实际账户余额	是否检测出重入性漏洞
1222	0 Ether, 0.8	1000 Ethers	1000 Ethers	否
0022	100 Ethers, 0.8	900 Ethers	900 Ethers	否
0100	100 Ethers, 0.2	900 Ethers	800 Ethers	是

潜在的重入性漏洞.

通过上述案例可以发现, 本文方法检测出智能合约重入性漏洞的关键在于覆盖重入性循环路径. 而上述路径由于其分支约束更多, 覆盖难度也更高, 在实际智能合约中可行解的数量N将会十分巨大. 本文假设鸽群算法集中搜索的种群历史最优解邻域包含更优解的概率要大于其他群体智能算法搜索邻域包含更优解的概率. 因此, 本文选择在鸽群算法基础上进行改良, 确保算法能够在预设最大测试用例开销Max内覆盖重入性循环路径, 检测智能合约重入性漏洞.

4 流形鸽群算法

为了确保鸽群算法能够覆盖基于存在重入性循环路径的ATCG-PC问题模型中触发重入性漏洞的SUT路径, 一种可行的改进思路让鸽群算法在包含目标路径测试用例的局部空间内搜索, 避免其在整个决策空间内搜索产生的冗余开销. 因此, 本文选择以流形启发式算子[36]改进鸽群算法的方式, 求解存在重入性循环路径的ATCG-PC问题.

4.1 鸽群算法

原始鸽群算法主要包括地标算子与指南针算子两部分. 在指南针算子中, 算法的更新公式如下所示:

$$\lambda_i^t = \lambda_i^{t-1}\mathrm{e}^{-R\times t} + \mathrm{rand}\times\left(X_{\mathrm{gbest}} - X_i^{t-1}\right), \tag{8}$$

$$X_i^t = X_i^{t-1} + \lambda_i^t, \tag{9}$$

其中, X_i^t与λ_i^t分别表示种群个体在第t次迭代时的位置与速度, R是地标算子与指南针算子中的因数, 取值范围在[0, 1], rand是公示中取值范围在[0, 1]间的随机数, t表示当前算法迭代次数, 而X_{gbest}是算法在第$t-1$次循环迭代后种群个体中最优解的位置. 鸽群算法将会在指南针算子迭代次数达到预设最大值T_1后进入地标算子继续搜索.

在地标算子更新时, 每次迭代后将会选取种群个体中评估值更高的前一半个体, 舍弃评估值更低的一半个体. X_{center}表示被选中种群个体的中心位置, 其作为地标将指引后续的算法搜索, 更新公式如下所示:

$$X_{\mathrm{center}}^{t-1} = \frac{\sum_{i=1}^{NP^{t-1}} X_i^{t-1}\mathrm{fitness}\left(X_i^{t-1}\right)}{NP^{t-1}\sum_{i=1}^{t-1}\mathrm{fitness}\left(X_i^{t-1}\right)}, \tag{10}$$

$$X_i = X_i^{t-1} + \mathrm{rand}\times\left(X_{\mathrm{center}}^{t-1} - X_i^{t-1}\right), \tag{11}$$

其中, NP表示种群大小, $X_{\mathrm{center}}^{t-1}$是种群评估值靠前的一半个体依据式(10)计算得到的中心位置. 与指南针算子类似, 地标算子在迭代次数达到预设最大值T_2后停止工作.

通过式(8)~(11)可以发现, 鸽群算法利用指南针算子与地标算子向种群最优解收敛. 与向种群个体最优解与种群最优解两个方向收敛的粒子群算法相比, 鸽群算法搜索方向更加集中, 有更高的概率更快收敛到局部最优[43]. 然而鸽群算法在整个决策空间内进行局部搜索求解优化问题, 在面对ATCG-PC问题的大规模求解范围, 鸽群算法难以在预设的测试用例开销Max内收敛到目标路径. 本文将从找出与优化目标相关的子空间、约减鸽群算法的搜索范围的角度出发, 提升鸽群算法求解存在重入性循环路径的ATCG-PC问题性能.

4.2 改进后的鸽群算法

在机器学习领域存在普遍认同的流形分布假设[44,45], 即自然界中同一类别的高维数据往往集中在某个低维流形附近. 如图4所示, 该低维流形同胚某个低维欧氏空间, 该低维欧氏空间包含了所有目标特征. 受流形分布假设启发, 作者在先前的研究中提出了流形启发式算子[36]以增强群体智能算法求解ATCG-PC

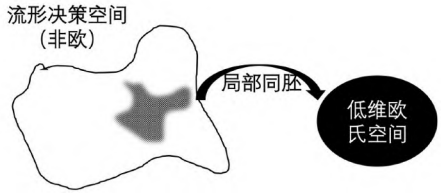

图 4 流形分布假设示意图
Figure 4 Manifold distribution hypothesis.

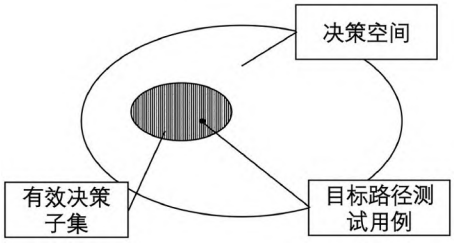

图 5 流形启发式算子示意图
Figure 5 Manifold-inspired operator.

问题的性能. 如图5所示, 该算子的主要思想是找出问题的有效决策子集(包含目标路径测试用例, 但求解范围远小于整个决策空间的子空间), 并自适应分配有效决策子集与其他决策空间的搜索开销, 从而覆盖目标路径的测试用例.

鸽群算法是通过地标算子与指南针算子在种群历史最优解的邻域进行局部搜索. 若鸽群算法搜索的局部空间不包含目标路径测试用例, 将会造成大量冗余计算开销. 因此, 本文将借助流形启发式算子的思想改进鸽群算法, 提出流形鸽群算法(MPIO)求解存在重入性循环路径的ATCG-PC问题.

MPIO算法的流程如图6所示, 在种群初始化后将利用鸽群算法的指南针算子或地标算子更新后代解, 若在此之后算法无法覆盖目标路径, 则调用流形启发式算子[36]进行搜索. 流形启发式算子主要通过两个策略实现: 策略1为问题流形性质挖掘, 其主要方式是在预设的子空间内搜索的同时更新测试用例-路径关系矩阵[24], 该矩阵记录了不同测试用例维度影响分支走向的次数, 能够辅助MPIO算法找出目标路径有效决策子集; 策略2则是基于流形性质的动态搜索策略, 其凭借测试用例-路径关系矩阵找出目标路径有效决策子集, 动态分配在该子空间与其他空间的搜索开销, 高效覆盖剩余目标路径.

MPIO流形性质挖掘是通过更新测试用例-路径关

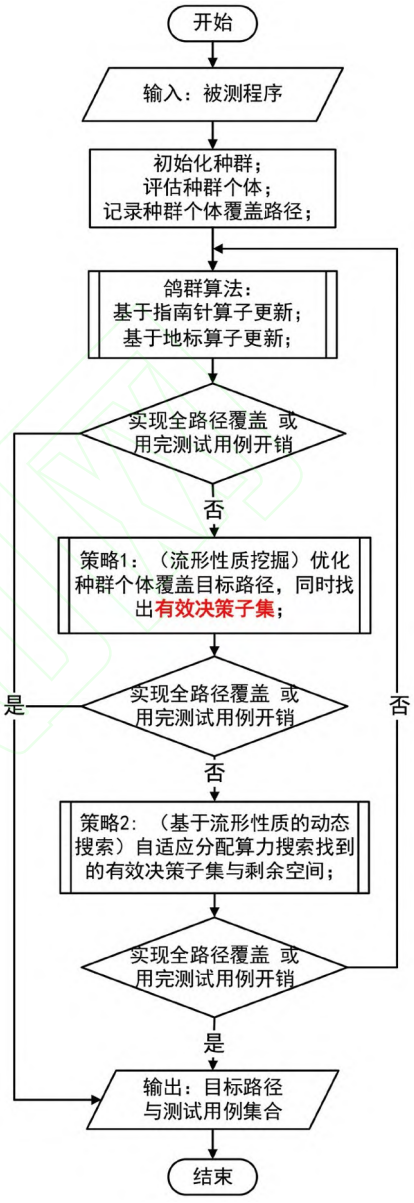

图 6 (网络版彩图)MPIO算法流程图
Figure 6 (Color online) Flow chart of MPIO algorithm.

系矩阵[24]实现, MPIO在策略1中每个种群个体X_i在其维度顺序最优组成的子空间S_{X_i}内搜索. 子空间$S_{X_i}=\cup_{r=1}^{n+1}\left\{(x_{i,1},x_{i,2},\ldots x_{i,r}\ldots,x_{i,n+1})\ \middle|\ x_{i,r}\in[lb_r,ub_r]\right\}$, 其中$x_{i,n+1}$即新增变量$\pi$, lb_r与ub_r分别表示第r维测试用例变量$x_{i,r}$取值范围的下界与上界. 在子空间S_{X_i}搜索过程中, MPIO每次产生新解仅改变一个维度的变量数值, 因此, 当更新后的X_i覆盖新路径时, 将会更新测试用

例-路径关系矩阵. 而在策略2中, MPIO将会根据关系矩阵精准地确定有效决策子集的范围, 通过对比X_i与目标路径编码找出不同分支, 结合关系矩阵找出相关维度并确定有效决策子集S'_{X_i}.

为了确保MPIO更好地发挥其局部搜索特性, 其指南针算子与地标算子的搜索范围将会充分利用策略1与策略2搜索时挖掘的问题流形性质, 保证算法搜索范围在找到的有效决策子集内. 该子空间由流形启发式算子更新后的测试用例-路径关系矩阵[36]所确定, 通过该矩阵能够找出目前优化个体与目标路径间相关的输入变量维度, 从而约减鸽群算法两个算子的局部搜索范围, 提升收敛效果.

4.3 评估函数

为了使MPIO覆盖难度较高的目标路径, 需要设计一种针对特定目标路径的评估方法. 在该方法中, 测试用例X_i的评估值将会是其在所有目标路径p_{target}的分支距离之和. 具体来说, 测试用例X_i的评估值如式(6)所示. 而$f_j^v(X_i)$表示测试用例X_i在被测试程序第v个分支处的评估值, 其计算将基于式(12):

$$f_j^v(X_i)=\begin{cases}\dfrac{1}{\text{cost}\left(v_{X_i}^j\right)+\varepsilon}, & p_{X_i}\text{覆盖}p_j\text{第}v\text{个分支},\\ 0, & \text{otherwise},\end{cases} \tag{12}$$

其中, $\text{cost}\left(v_{X_i}^j\right)$表示$X_i$与$p_{\text{target}}$在被测试程序的第$v$个分支的分支距离, p_{X_i}表示测试用例X_i覆盖的路径, 而ε是一个用于避免分母为零的常数, cost的计算将基于表4列举的公式.

若生成的后代测试用例X_i与目标路径p_{target}在第v分支走向相同, 则$\text{cost}\left(v_{X_i}^j\right)$值将会为零, 此时$f_j^v(X_i)$取最大值$1/\varepsilon$, 当且仅当测试用例$X_i$覆盖目标路径$p_{\text{target}}$时, 才能取得最高的评估值. 该评估函数将在流形鸽群算法在有效决策子集内搜索目标路径测试用例时提供指引, 顺利覆盖目标路径.

5 算法验证

本节将验证MPIO算法求解存在重入性循环路径ATCG-PC问题的性能. 实验假设: (1) 鸽群算法搜索的种群最优解邻域存在更优解的概率要高于其他群体智能算法搜索邻域包含更优解的概率; (2) 基于提出的ATCG-PC问题模型, MPIO算法能够有效检测智能合约的重入性漏洞. 为了验证方法的有效性, MPIO算法将与粒子群算法、蜂群算法以及差分进化算法结合流形启发式算子后的变体(MPSO, MABC与MDE)进行对比; 同时, 本文方法的重入性漏洞检测结果将与三种先进的智能合约测试工具[8,15,16]进行对比.

表 4 代价函数cost(v)计算表

Table 4 Calculation of cost function cost(v)

状态v	cost(v)计算方程
布尔语句	若真值为0, 若假值为K
$A>B$	若$(B-A)<0$值为0; 否则值为$(B-A)+K$
$A\geq B$	若$(B-A)\leq 0$值为0; 否则值为$(B-A)+K$
$A<B$	若$(A-B)<0$值为0; 否则值为$(A-B)+K$
$A\leq B$	若$(A-B)\leq 0$值为0; 否则值为$(A-B)+K$
$A=B$	若$\lvert B-A\rvert=0$值为0; 否则值为$\lvert B-A\rvert+K$
$A\neq B$	若$\lvert B-A\rvert\neq 0$值为0; 否则值为K
$A\wedge B$	cost(A)+cost(B)
$A\vee B$	min(cost(A), cost(B))

5.1 算法参数设置

本文实验参数设置如下: MPIO算法种群规模为20, 其变异参数R设置为0.2, 针对一条目标路径, 指南针算子与地标算子的迭代次数上限T_1与T_2分别设置为15与20. 在对比算法MPSO, MABC与MDE算法中MPSO变异算子c_1, c_2分别取值1.5与2, 惯性权重ω设置为0.4[28], MABC蜂群限制参数limit取值为2[29], MDE的变异参数F设置为0.5, 交叉参数P_c设置为0.2[25]. 所有算法进行30次独立重复运行, 最大评估次数(最大测试用例开销)设置为3×10^5, 上述参数设置参照相关工作设置[24,31,46]. 显著性差异分析将基于α=0.05的Wilcoxon符号秩和检验进行.

同时本文选择了三个免费开源的智能合约测试工具(Oyente[8], Smartcheck[16], Securify[15]), 实验部分将对比这三种工具与本文方法在检查重入性漏洞上的性能表现, 这三种工具的详细信息详见表1.

5.2 基准测试函数

本文使用的基准测试函数选自相关研究论

表 5 智能合约基准测试函数[a)]

Table 5 Benchmark functions of smart contract

ID	函数名称	合约代码行数	测试用例维度数	路径数(可达路径数)	描述
F1	withdrawBalance	8	3	5(3)	该函数通过调用call.value()函数在存在重入性漏洞的智能合约中套取银行账户余额
F2	withdrawBalance_fixed_order	9	3	5(3)	该函数负责提取银行账户余额, 通过调整取款操作和设置操作间的顺序修复了重入性漏洞
F3	withdrawBalance_fixed_transfer	8	2	2(2)	该函数通过调用transfer函数代替call.value()进行取款操作, 修复了潜在重入性漏洞
F4	withdrawBalance_fixed_lock	10	4	5(4)	该函数负责提取银行账户余额, 通过使用mutex技术修复了潜在重入性漏洞
F5	withdrawBalance_fixed_send	8	2	2(2)	该函数通过调用send函数代替call.value()进行取款操作, 修复了潜在重入性漏洞
F6	withdrawBalance_fixed_gaslimit	8	3	5(2)	该函数负责提取银行账户余额, 通过固定gas开销修复了潜在重入性漏洞
F7	Puzzle_solved	13	7	5(4)	该函数用于验证漏洞是否被修复, 如果漏洞被修复, 那么该函数将调用call.value()函数, 并将余额转移到调用者银行账户(该函数存在重入性漏洞)
F8	transfer_to_someone	15	11	5(4)	该函数通过调用call.value()函数将一笔钱转入特定的账户(该函数存在重入性漏洞)

a) “描述”部分介绍了被测试函数在智能合约中的功能与是否存在重入性漏洞; “路径数(可达路径数)”分别表示被测试智能合约函数的路径数目与存在测试用例的路径数目

表 6 基准测试函数路径概率[a)]

Table 6 Probability of each path in benchmark functions

ID	函数名称	每条路径的概率
F1	withdrawBalance	0; 0; 1.16×10^{-10}; 4.99×10^{-1}; 4.99×10^{-1}
F2	withdrawBalance_fixed_order	0; 0; 1.16×10^{-10}; 4.99×10^{-1}; 4.99×10^{-1}
F3	withdrawBalance_fixed_transfer	2.32×10^{-10}; 9.99×10^{-1}
F4	withdrawBalance_fixed_lock	0; 1.16×10^{-10}; 2.49×10^{-1}; 2.49×10^{-1}; 0.50×10^{-1}
F5	withdrawBalance_fixed_send	2.32×10^{-10}; 9.99×10^{-1}
F6	withdrawBalance_fixed_gaslimit	2.32×10^{-10}; 9.99×10^{-1}
F7	Puzzle_solved	0; 6.84×10^{-49}; 1.16×10^{-10}; 1.16×10^{-10}; 9.99×10^{-1}
F8	transfer_to_someone	0; 6.84×10^{-49}; 3.42×10^{-49}; 3.43×10^{-49}; 9.99×10^{-1}

a) “每条路径的概率”指的是在问题决策空间内均匀分布随机初始化测试用例, 覆盖该路径的概率

文[7,8,25], 详细信息可见表5. 表6则展示了表5中测试集的每条路径被随机生成测试用例覆盖的概率.

F1是本文的一个函数实例, 它作为银行智能合约的重要函数, 能够通过调用call.value()函数, 实现用户从账户提取资金的功能. 许多研究人员将此函数作为重入性漏洞研究的经典案例[8], 因此, 本文将此函数选为基准测试函数, 观察本文方法与各对比工具在检测智能合约重入性漏洞上的表现.

F2展示了一种修复重入性漏洞后的合约函数案例. 该案例通过替换取款操作与设置余额为零操作的执行顺序, 攻击者在企图重复提取资金时, 会因为余额值已被设置为零而抛出异常, 从而终止函数执行. 由于潜在的重入性漏洞已被修复, 该函数被测试集被用于验证方法的准确度.

除去了替换账户清零与取款操作的执行顺序, 另一种修复重入性漏洞的方法是使用transfer函数或send

函数代替call.value()函数进行取款操作. 如定义1所述, transfer与send两个函数在调用时有严格的2300 gas的要求. 当攻击者试图提取金额超过上述限制时, 合约会报出异常. 基于上述修复措施, 可得到F3与F5两个基准测试函数, 其中F3使用transfer函数提取金钱, 而F5则调用send函数实现功能. F3与F5这两个函数的重入性漏洞已被修复, 因此被选为基准测试函数, 用于检测相关测试工具是否会存在误报的情况.

F4通过调用call.value()函数进行账户取款操作, 但不同于F1的是, F4在取款操作前实现了一个锁. F4的有限状态机[47](finite state machine, FSM)如图7所示. 该FSM显示, 当用户成功取款后, F4的状态将会被锁定(S1), 只有当balance值(取款金额)设置为零时, F4才会被解锁(S0). 该锁定操作保证了函数完全执行前不会被多次调用. 加入锁定是修复智能合约重入性漏洞最有效、最常用的方法之一, 因此, 本文将F4加入基准评估函数来检测相关测试工具的有效性.

F6与F1相似, 两者的不同之处在于F6新增了一个参数用于限制call.value()函数中的gas开销, 该参数能够避免call.value()被无限期调用, 避免攻击者偷取资金.

F7和F8与F1类似, 这两个函数都有重入性漏洞, 但它们的应用环境有所不同. F7是一个奖金合同, 如果有人解决了网络上悬赏的难题, 那么可以通过调用该合约获取奖励; 而F8是一个负责转账的合约, 可以实现向指定地址转账一定金额的功能.

5.3 实验结果与分析

流形启发式算子能够显著提升PIO算法求解存在重入性循环路径ATCG-PC问题的性能. 如表7所示, 与PIO算法相比, MPIO算法在所有测试集上均取得更高的平均路径覆盖率ave.c, 且能够在约束的测试用例开销内覆盖所有可达路径. 特别地, 在F1~F6这6个测试集上, MPIO算法的平均测试用例开销ave.m均低于1.00×10^3. 通过表6可以发现, F7与F8都存在比较难覆盖的路径, 均匀分布随机生成的测试用例能够将其覆盖的概率仅为6.84×10^{-49}. PIO仅能在这两个测试集上分别达到43.3%与25%的平均路径覆盖率, 而MPIO不仅能够覆盖这两个测试集的所有路径, 其测试用例开销平均值ave.m均低于2.00×10^4.

本文假设生成重入性路径测试用例后, 通过代入测试用例对比预期结果与实际结果就能够发现程序潜在的重入性漏洞. 因此, 重入性循环路径被覆盖时将被视为重入性漏洞检测成功. 如表7与8所示, MPIO算法覆盖了所有路径, 其中也包括模拟重入性漏洞攻击的路径, 在对比了合约测试用例实际执行与预期结果后, 这些基准测试函数潜在的重入性漏洞最终都能被检测到. 为了进一步检验本文方法检测重入性漏洞的效率, 本文方法将与其他工具进行对比, 详细结果见表9.

为了验证本文提出的假设1, 即鸽群算法搜索的邻域包含更优解的概率高于其他群体智能算法搜索邻域包含更优解的概率. 本文设计MPIO与MPSI, MABC以

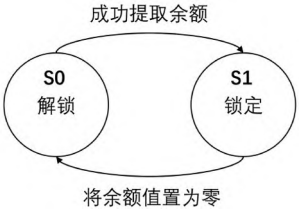

图 7 F4的有限状态机

Figure 7 Finite state machine of F4.

表 7 MPIO与PIO在基准测试函数上的对比结果[a)]

Table 7 Comparison results of MPIO and PIO in benchmark functions

函数ID	PIO		MPIO	
	ave.c	ave.m (std.m)	ave.c	ave.m (std.m)
F1	71.6%	2.51×10^5 (8.53×10^4)+	100%	5.37×10^2 (2.71×10^2)
F2	71.6%	2.50×10^5 (8.54×10^4)+	100%	4.52×10^2 (2.79×10^2)
F3	50%	3.00×10^5 (0)+	100%	4.91×10^2 (2.52×10^2)
F4	81.7%	2.21×10^5 (1.20×10^5)+	100%	4.78×10^2 (2.73×10^2)
F5	50%	3.00×10^5(0)+	100%	6.32×10^2 (3.71×10^2)
F6	50%	3.00×10^5(0)+	100%	5.67×10^2 (3.95×10^2)
F7	43.3%	3.00×10^5(0)+	100%	5.57×10^3 (2.44×10^3)
F8	25%	3.00×10^5(0)+	100%	1.68×10^4 (3.83×10^3)
+/=/–		8/0/0		–

a) "+/=/–"表示MPIO算法在测试集上显著优于、无显著差异与显著劣于PIO的次数

表 8 MPIO与MPSO, MDE, MABC在基准测试函数上的对比结果[a)]

Table 8 Comparison results of MPIO, MPSO, MDE and MABC in benchmark functions

函数ID	MPSO		MPIO	
	ave.c	ave.m (std.m)	ave.c	ave.m (std.m)
F1	100%	5.67×10^{2} (4.10×10^{2})=	100%	4.91×10^{2} (2.93×10^{2})
F2	100%	5.09×10^{2} (3.43×10^{2})=	100%	5.80×10^{2} (3.81×10^{2})
F3	100%	5.99×10^{2} (3.86×10^{2})=	100%	4.37×10^{2} (2.81×10^{2})
F4	100%	4.98×10^{2} (3.16×10^{2})=	100%	4.27×10^{2} (2.53×10^{2})
F5	100%	6.84×10^{2} (4.53×10^{2})=	100%	5.45×10^{2} (3.47×10^{2})
F6	100%	6.13×10^{2} (3.15×10^{2})=	100%	7.43×10^{2} (4.35×10^{2})
F7	100%	1.01×10^{4} (6.41×10^{3})+	100%	5.72×10^{3} (2.79×10^{3})
F8	100%	4.65×10^{4} (2.08×10^{4})+	100%	1.71×10^{4} (4.66×10^{3})
+/=/–	2/6/0		–	

函数ID	MDE		MABC	
	ave.c	ave.m (std.m)	ave.c	ave.m (std.m)
F1	100%	4.87×10^{2} (2.98×10^{2})=	100%	6.03×10^{2} (3.98×10^{2})=
F2	100%	6.88×10^{2} (5.55×10^{2})=	100%	5.78×10^{2} (3.87×10^{2})=
F3	100%	6.10×10^{2} (3.55×10^{2})=	100%	4.97×10^{2} (2.49×10^{2})=
F4	100%	5.90×10^{2} (4.20×10^{2})=	100%	6.34×10^{2} (4.38×10^{2})=
F5	100%	7.41×10^{2} (4.41×10^{2})=	100%	6.36×10^{2} (4.11×10^{2})=
F6	100%	5.42×10^{2} (2.87×10^{2})=	100%	7.12×10^{2} (4.84×10^{2})=
F7	100%	5.61×10^{3} (2.36×10^{3})=	100%	6.72×10^{3} (2.25×10^{3})=
F8	100%	1.93×10^{4} (4.79×10^{3})=	100%	2.15×10^{4} (5.84×10^{3})+
+/=/–	0/8/0		1/7/0	

a) “+/=/–”表示MPIO算法在测试集上显著优于、无显著差异与显著劣于MPSO, MDE与MABC的次数

及MDE在F1~F8上进行统计实验对比. 与MPSO以及MABC相比, MPIO更加适合求解存在重入性循环路径的ATCG-PC问题, 如表8所示, MPIO, MPSO与MABC在F1~F6这6个测试集上都能够在1.00×10^{3}的测试用例开销内覆盖所有可达路径, 且两种算法的效果无显著性差异. 虽然三种算法都能在F7与F8两个测试集上以低于5.00×10^{4}的测试用例开销覆盖全部可达路径, 但MPIO对应开销显著低于MPSO. 在测试集F8上MPIO对应开销也显著低于MABC. 造成该现象的原因是F1~F6求解难度低于F7与F8, 导致MPIO需要进行多轮迭代循环才能覆盖所有路径. 在MPIO中, 鸽群算法的指南针算子与地标算子能够在假设的有效决策子集内局部搜索, 而MPSO的更新策略收敛性更差, 使得MPIO仅在F7与F8这类难度较高的测试集上的表现显著优于MPSO.

如表8所示, MPIO与MDE在F1~F8所有测试集上的表现都没有显著性差异, 两种算法的开销都保持在较低的水平. 该实验结果说明, 鸽群算法以及差分进化算法搜索的邻域包含更优解的概率相近.

为了验证实验假设2, 即本文方法能够有效检测出智能合约重入性漏洞, 本文方法将与其他不同工具方法进行对比.如表7与8所示, MPIO算法在F1, F7, F8这3个测试函数均上达到了100%的路径覆盖率, 通过执行这些存在重入性漏洞测试集的测试用例, 可以发现实际银行余额与预期余额不同, 因此本文方法可以发现这三个函数潜在的重入性漏洞, 而其他函数上不会存在金额的差异, 因此不会产生误报. 如表9所示, 用于对比的智能合约测试工具Oyente, Securify与Smartcheck也能发现F1, F7, F8存在重入性漏洞.

本文方法可以在F2上实现100%的路径覆盖. 对比预期输出与实际输出后发现F2不存在重入性漏洞. 但是, Smartcheck却得出该函数存在重入性漏洞的结论, 得出假阳性的结果. 通过分析表10中#FP以及其他统计指标的结果可以发现, Oyente与Securify都得出该函数是安全的正确结论.

本文方法在F3, F5和F6上都实现了100%的路径覆盖. 由于这三个测试函数都不存在重入性漏洞的路径, 因此本文方法并不会给出检测到重入性漏洞的报告. 用于对比的测试工具Oyente与Securify得出了相同结论, 但Smartcheck在检测F6时会报出假阳性的错误报告.

根据表9的实验结果, 用于对比的三个工具均表示F4存在重入性漏洞. 但事实上, F4通过新增一个mutex变量, 实现了一个状态锁, 避免了函数被多次调用的情况, 修复了重入性漏洞, 所以这三个工具在F4上得出的

表 9 不同工具与本文方法在检测重入性漏洞的对比结果[a)]

Table 9 Comparison results of different toolkits and the proposed method in detecting reentrancy vulnerability

函数ID	Oyente[9]	Securify [34]	Smartcheck[35]	本文方法	标准答案
F1	T	T	T	T	T
F2	F	F	T	F	F
F3	F	F	F	F	F
F4	T	T	T	F	F
F5	F	F	F	F	F
F6	F	F	T	F	F
F7	T	T	T	T	T
F8	T	T	T	T	T

a) T表示工具找到重入性漏洞; F表示工具没有发现重入性漏洞

表 10 不同工具与本文方法在实验指标上的对比结果

Table 10 Comparison results of different toolkits and the proposed method in experimental indicators

指标	Oyente[9]	Securify[34]	Smartcheck[35]	本文方法	标准答案
#TP	3	3	3	3	3
#FN	0	0	0	0	0
#TN	4	4	3	5	5
#FP	1	1	2	0	0
TRP	100%	100%	100%	100%	100%
FRP	20%	20%	40%	0%	0%
ACC	87.5%	87.5%	75%	100%	100%

结论是错误的.

在本小节中, 分析和对比了相关检查工具在各类合约函数上的表现. 本文提出的方法, Securify, Oyente与Smartcheck在检测重入性漏洞上都有着不错的表现. 但是, 上述对比工具[8,15,16]在面对不同修复后的合约函数上可能会出现一些误判, 比如Oyente, Securify与Smartcheck都不能够识别出F4的重入性漏洞已被修复, 而且Smartcheck在检测F2和F6时也会给出误报. 为了更加直观地比较各方法的表现, 下一节将通过一些统计指标进一步分析实验数据.

5.4 实验指标分析

如表10所示, 本文给出了4个实验指标[48]来进一步对比本文方法与三个相关测试工具的性能:

(1) #TP: 方法正确识别出重入性漏洞, 且被测智能合约存在重入性漏洞的合约数量;

(2) #FN: 方法未能识别出重入性漏洞, 但被测智能合约存在重入性漏洞的合约数量;

(3) #TN: 方法未检测出重入性漏洞, 且被测智能合约不存在重入性漏洞的合约数量;

(4) #FP: 方法检测出重入性漏洞, 但被测智能合约不存在重入性漏洞的合约数量.

在上述4种指标的基础上, 进一步通过TPR, FPR, ACC这三个数值来刻画工具的性能.

$$\mathrm{TPR}=\frac{\#\mathrm{TP}}{\#\mathrm{TP}+\#\mathrm{FN}}. \tag{13}$$

TPR的计算如式(13)所示, 该数值表示工具检测过程中真正的阳性率, 代表了包含潜在重入性漏洞的合约中被正确识别的比例. 实验结果如表10所示, 三种对比工具与本文方法都实现了100%的TPR值.

$$\mathrm{FPR}=\frac{\#\mathrm{FP}}{\#\mathrm{FP}+\#\mathrm{TN}}. \tag{14}$$

FPR的计算可见式(14), 该数值表示工具检测过程中的假阳性率, 代表了被错误识别的实际阳性比例. 如表10所示, Oyente与Securify都达到了20%的FPR值,

Smartcheck更是高达40%, 而本文方法没有出现假阳性的检测报告, 因此其TPR值为零.

$$\mathrm{ACC}=\frac{\#\mathrm{TP}+\#\mathrm{TN}}{\#\mathrm{TP}+\#\mathrm{FN}+\#\mathrm{TN}+\#\mathrm{FP}}. \tag{15}$$

ACC值的计算如式(15)所示, 该数值代表方法的整体正确率, 用于衡量正确识别的结果占所有合约的比例. 如表10所示, Oyente和Securify达到了87.5%的ACC值, Smartcheck的ACC值仅有75%, 是对比方法中最低的, 而本文方法由于没有误判且重入性漏洞检测准确, 其ACC值为100%.

通过分析对比表10所展示的本文方法与对比工具在TPR, FPR与ACC三个实验指标上的数据, 可以得出以下结论: (1) 当存在重入性漏洞时, 本文方法与对比三种工具都能够识别重入性漏洞; (2) Smartcheck的FPR值比较高, 意味着该工具可能存在重入性漏洞误报的情况; (3) 本文方法拥有最低的FPR值, 且ACC值最高, 说明与Oyente, Securify与Smartcheck相比, 本文方法在测试集内能够更加准确地识别智能合约的重入性漏洞.

6 结论与讨论

本文将智能合约存在重入性漏洞的应用场景抽象为存在重入性循环路径的ATCG-PC问题模型, 并通过流形鸽群算法生成该模型路径覆盖测试用例, 检测智能合约的重入性漏洞. 具体来说, 本文所提出的方法以存在重入性循环路径的ATCG-PC问题数学模型为基础, 通过向合约控制流图添加额外边, 形成重入性循环路径, 实现重复调用call.value()等取款操作, 从而模拟智能合约重入性漏洞攻击. 本文方法通过执行流形鸽群算法搜索得到的路径覆盖测试用例, 对比测试用例执行后实际银行余额与预期值, 判断被测试合约是否存在潜在的重入性漏洞. 由于本文方法属于动态测试, 能够避免符号执行等静态分析方法可能发生的重入性漏洞误报的情况. 实验结果表明, (1) 流形启发式算子能够使鸽群算法在与目标路径相关的有效决策子集内搜索, 缩小鸽群算法搜索范围, 从而提升ATCG-PC问题的求解效率; 与粒子群算法和蜂群算法等群体智能算法相比, 鸽群算法与流形启发式算子结合后能够更有效地求解ATCG-PC问题. (2) Oyente, Smartcheck与Securify这些智能合约测试工具在处理修复后的智能合约时会出现误报的情况(误报率分别为20%, 20%与40%), 而本文方法在测试集内没有出现假阳性的误报问题. (3) 本文所提出的通过求解存在重入性循环路径的ATCG-PC问题, 进而对比账户执行结果与预期值的方法能够准确检测智能合约重入性漏洞.

在下一步的工作中, 我们将研究多合约并行调用场景的ATCG-PC问题建模与求解, 实现更复杂的应用场景下的测试效率优化. 同时, 利用流形启发式算子思想优化其他群体智能算法, 从而高效求解ATCG-PC问题也是未来需要重点研究的方向.

致谢 感谢广东外语外贸大学何莉怡同学以及华南理工大学的冯夫健、康力、徐粤婷、苏俊鹏为论文提供修改意见与建议.

参考文献

1 Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. Manubot, 2019
2 Li H Y, Baoyin H X. Sequence optimization for multiple asteroids rendezvous via cluster analysis and probability-based beam search. Sci China Tech Sci, 2021, 64: 122–130
3 Li R, Song T, Mei B, et al. Blockchain for large-scale internet of things data storage and protection. IEEE Trans Serv Comput, 2018, 12: 762–771
4 Xu Q, Dai P C, Wang L, et al. Distributed consensus-based algorithm for social welfare in smart grid with transmission losses. Sci China Tech Sci, 2020, 63: 44–54
5 Christidis K, Devetsikiotis M. Blockchains and smart contracts for the internet of things. IEEE Access, 2016, 4: 2292–2303
6 Zheng P, Zheng Z, Luo X, et al. A detailed and real-time performance monitoring framework for Blockchain systems. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE, 2018. 134–143

7 Nikolić I, Kolluri A, Sergey I, et al. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference. San Juan, 2018. 653–663

8 Luu L, Chu D H, Olickel H, et al. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, 2016. 254–269

9 Chen W, Zheng Z, Cui J, et al. Detecting Ponzi schemes on Ethereum: Towards healthier Blockchain technology. In: Proceedings of the 2018 World Wide Web Conference. Lyon, 2018. 1409–1418

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

11 Gatteschi V, Lamberti F, Demartini C, et al. Blockchain and smart contracts for insurance: Is the technology mature enough? Future Internet, 2018, 10: 20

12 Atzei N, Bartoletti M, Cimoli T. A survey of attacks on Ethereum smart contracts (SoK). In: Maffei M, Ryan M, eds. Principles of Security and Trust. POST 2017. Lecture Notes in Computer Science, vol 10204. Berlin, Heidelberg: Springer, 2017. 164–186

13 Juels A, Kosba A, Shi E. The ring of Gyges: Investigating the future of criminal smart contracts. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, 2016. 283–295

14 Grossman S, Abraham I, Golan-Gueta G, et al. Online detection of effectively callback free objects with applications to smart contracts. Proc ACM Program Lang, 2018, 2: 1–28

15 Tsankov P, Dan A, Drachsler-Cohen D, et al. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto, 2018: 67-82

16 Tikhomirov S, Voskresenskaya E, Ivanitskiy I, et al. SmartCheck: Static analysis of Ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. Gothenburg, 2018. 9–16

17 Cadar C, Sen K. Symbolic execution for software testing. Commun ACM, 2013, 56: 82–90

18 Qin X J, Gan S T, Chen Z N. A static detecting technology of software code secure vulnerability based on first-order logic (in Chinese). Sci Sin-Inf, 2014, 44: 108–129 [秦晓军, 甘水滔, 陈左宁. 一种基于一阶逻辑的软件代码安全性缺陷静态检测技术. 中国科学: 信息科学, 2014, 44: 108–129]

19 Ali S, Briand L C, Hemmati H, et al. A systematic review of the application and empirical investigation of search-based test case generation. IIEEE Trans Software Eng, 2009, 36: 742–762

20 Anand S, Burke E K, Chen T Y, et al. An orchestrated survey of methodologies for automated software test case generation. J Syst Software, 2013, 86: 1978–2001

21 Sun H Y, Liu J, Chen X H, et al. A safety verification approach for changed real time implementation based on formal testing (in Chinese). Sci Sin-Inf, 2014, 44: 70–90 [孙海英, 刘静, 陈小红, 等. 基于形式化测试的实时系统变更后安全性验证. 中国科学: 信息科学, 2014, 44: 70–90]

22 Horgan J R, London S, Lyu M R. Achieving software quality with testing coverage measures. Computer, 1994, 27: 60–69

23 Fraser G, Arcuri A. Whole test suite generation. IIEEE Trans Software Eng, 2012, 39: 276–291

24 Huang H, Liu F, Yang Z, et al. Automated test case generation based on differential evolution with relationship matrix for IFOGSIM toolkit. IEEE Trans Ind Inf, 2018, 14: 5005–5016

25 Huang H, Liu F, Zhuo X, et al. Differential evolution based on self-adaptive fitness function for automated test case generation. IEEE Comput Intell Mag, 2017, 12: 46–55

26 Wu H, Nie C, Petke J, et al. An empirical comparison of combinatorial testing, random testing and adaptive random testing. IIEEE Trans Software Eng, 2018, 46: 302–320

27 Li H, Lam C P. Software test data generation using ant colony optimization. In: International Conference on Computational Intelligence. Istanbul, 2004. 1–4

28 Ding R, Feng X, Li S, et al. Automatic generation of software test data based on hybrid particle swarm genetic algorithm. In: 2012 IEEE Symposium on Electrical & Electronics Engineering (EEESYM). IEEE, 2012. 670–673

29 Jeya Mala D, Mohan V, Kamalapriya M. Automated software test optimisation framework—An artificial bee colony optimisation-based approach. IET Softw, 2010, 4: 334–348

30 Duan H, Huo M, Yang Z, et al. Predator-prey pigeon-inspired optimization for UAV ALS longitudinal parameters tuning. IEEE Trans Aerosp Electron Syst, 2018, 55: 2347–2358

31 Duan H, Wang X. Echo state networks with orthogonal pigeon-inspired optimization for image restoration. IEEE Trans Neural Netw Learn Syst, 2015, 27: 2413–2425

32 Xiang S, Xing L, Wang L, et al. Comprehensive learning pigeon-inspired optimization with tabu list. Sci China Inf Sci, 2019, 62: 70208

33 Zhong Y, Wang L, Lin M, et al. Discrete pigeon-inspired optimization algorithm with Metropolis acceptance criterion for large-scale traveling salesman problem. Swarm Evolary Computat, 2019, 48: 134–144

34 Chen G, Qian J, Zhang Z, et al. Application of modified pigeon-inspired optimization algorithm and constraint-objective sorting rule on multi-objective optimal power flow problem. Appl Soft Computing, 2020, 92: 106321

35 Cao R, Liu Y B, Shen H D, et al. Optimization algorithm of the surrogate model structure based on pigeon swarm search strategy (in Chinese). Sci Sin Tech, 2020, 50: 1612–1624 [曹瑞, 刘燕斌, 沈海东, 等. 基于鸽群搜索策略的代理模型结构优化方法. 中国科学: 技术科学, 2020, 50: 1612–1624]

36 Liu F, Huang H, Su J, et al. Manifold-inspired search-based algorithm for automated test case generation. IEEE Trans Emerg Top Comput, 2021, doi: 10.1109/TETC.2021.3070968

37 Samreen N F, Alalfi M H. Reentrancy vulnerability identification in Ethereum smart contracts. 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, 2020. 22–29

38 Liu C, Liu H, Cao Z, et al. Reguard: Finding reentrancy bugs in smart contracts. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). IEEE, 2018. 65–68

39 Kim K B, Lee J. Automated generation of test cases for smart contract security analyzers. IEEE Access, 2020, 8: 209377–209392

40 Antunes N, Vieira M. Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples. IEEE Trans Serv Comput, 2014, 8: 269–283

41 Kalra S, Goel S, Dhawan M, et al. ZEUS: Analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium (NDSS18), 2018

42 Yao X, Gong D, Wang W. Test data generation for multiple paths basedon local evolution. Chin J Electron, 2015, 24: 46–51

43 Duan H B, Ye F. Progresses in pigeon-inspired optimization algorithms (in Chinese). J Beijing Univ Technol, 2017, 43: 1–7 [段海滨, 叶飞. 鸽群优化算法研究进展. 北京工业大学学报, 2017, 43: 1–7]

44 Gu X, Luo F, Sun J, et al. Variational principles for Minkowski type problems, discrete optimal transport, and discrete Monge–Ampère equations. Asian J Math, 2016, 20: 383–398

45 Lei N, An D, Guo Y, et al. A geometric understanding of deep learning. Engineering, 2020, 6: 361–374

46 Mansour N, Salame M. Data generation for path testing. Software Qual J, 2004, 12: 121–136

47 Matrosova A Y, Levin I, Ostanin S A. Self-checking synchronous FSM network design with low overhead. VLSI Des, 2000, 11: 47–58

48 Parizi R M, Dehghantanha A, Choo K. K. R. et al. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In: Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering. IBM Corp., 2018. 103–113

Smart contract reentrancy vulnerability detection method based on manifold pigeon optimization algorithm

LIU FangQing[1], HUANG Han[1], XIANG Yi[1] & HAO ZhiFeng[2,3]

[1] *School of Software Engineering, South China University of Technology, Guangzhou 510006, China;*
[2] *School of Computer, Shantou University, Shantou 515063, China;*
[3] *School of Computer, Guangdong University of Technology, Guangzhou 510006, China*

Reentrancy vulnerability commonly exists in smart contracts and results in serious economic losses. The existing symbolic execution-based static analyzing tools detect the reentrancy vulnerability by evaluating the default rules. However, the incompleteness of the default rules can lead to false positive judgments. We attempt to solve this problem from the perspective of test case generation based on dynamic execution. In this paper, the application scenario is abstracted as a mathematical model of the automated test case generation for path coverage (ATCG-PC) with reentrancy loop paths. The reentrancy vulnerability can be detected by executing the test cases of the reentrancy loop paths. The swarm intelligence algorithm represented by the pigeon optimization algorithm is a common method for solving the black-box optimization problem. The pigeon-inspired optimization algorithm searches in the neighbor of the population optimal solution; however, the optimal solution of the large-scale black-box optimization problem may not be in this neighbor. An improved pigeon-inspired optimization algorithm is proposed herein to improve the path coverage rate of the pigeon-inspired optimization algorithm for the ATCG-PC. The proposed algorithm allocates more computational resources to the subspace related to the target path, consequently improving the effectiveness of the pigeon-inspired optimization algorithm. It helps the pigeon-inspired optimization algorithm to cover the reentrancy loop path. The experimental results show that the improved pigeon-inspired optimization algorithm can effectively generate path coverage test cases in different smart contracts. The proposed method can also find all possible paths and accurately detect the reentrancy vulnerabilities when other tools (i.e., Oyente, Securify, and Smartcheck) make false positive judgments in the eight selected benchmarks. The recognition accuracy of the reentrancy vulnerabilities is improved by 12.5%, 12.5%, and 25%.

smart contract, reentrancy vulnerability detection, pigeon-inspired optimization algorithm, automated test case generation, path coverage

doi: 10.1360/SST-2021-0365