

中国科学技术大学

University of Science and Technology of China

硕士学位论文



论文题目 EOSIO智能合约自动形式化漏洞检测

方法研究

作者姓名 涂才畅

学科专业 计算机应用技术

导师姓名 黄文超 副教授

完成时间 二〇二四年五月

中国科学技术大学

硕士学位论文



EOSIO 智能合约自动形式化漏洞检测 方法研究

作者姓名： 涂才畅

学科专业： 计算机应用技术

导师姓名： 黄文超 副教授

完成时间： 二〇二四年五月二十二日

University of Science and Technology of China
A dissertation for master's degree



Research on Automatic Formal Vulnerability Detection Method for EOSIO Smart Contracts

Author: Tu Caichang

Speciality: Computer Application Technology

Supervisor: Associate Prof. Huang Wenchao

Finished time: May 22, 2024

摘 要

随着区块链平台的迅猛发展,智能合约作为区块链平台功能的强大扩展,得到了广泛应用,EOSIO 平台正是其中之一。不过,近年来 EOSIO 平台智能合约屡次遭受漏洞攻击,导致了严重的经济损失。为确保 EOSIO 平台上的智能合约安全,学术界已提出了一系列安全性分析和漏洞检测方法。这些漏洞检测方法主要可划分为两类:基于形式化的方法和基于模糊测试的方法。基于形式化的方法主要采用符号执行,以搜索 EOSIO 平台智能合约运行状态空间中的异常。由于状态空间的复杂性,这类方法通常需要对其进行缩减。然而,这样的缩减导致对底层机制建模并不充分,从而无法识别一些复杂合约中的漏洞。相较而言,基于模糊测试的方法中,种子生成和变异过程具有随机性,导致方法对代码路径的检测覆盖率不足,可能导致对某些漏洞的检测遗漏。总体而言,目前的漏洞检测方法仍存在检测准确性不足的问题,需要进一步的改进。

因此,本文基于形式化方法,以高精度检测为目标,对 EOSIO 平台智能合约进行了如下的研究工作:

1) 提出了一种基于循环符号执行的高代码覆盖率漏洞检测方法。本研究通过添加对链上数据库机制的支持并采用循环结构,使得符号执行方法能够考虑不同链上数据库状态下的执行路径,从而提高代码覆盖率。同时,本研究通过内存优化、跳过交易反序列化等方法,缓解了符号执行的状态爆炸问题。实验表明,本方法可以有效检测 EOSIO 平台智能合约的漏洞,F1 值达到了 92.58%,优于当前代表性的开源方法 WANA 和 WASAI。

2) 提出了一种基于 FASVERIF 的深度漏洞检测方法。通过使用形式化验证的方法,本研究避免了使用启发式剪枝或固定分支深度剪枝等方法以缩减状态空间,从而能够实现对智能合约的深度漏洞检测。同时该方法通过以安全属性为目标的路径构建,避免了漏洞无关路径的探索,能够实现了检测深度和检测时间的平衡。实验表明,该方法能够有效检测前一方法难以检测的漏洞,使得总的检测准确率和 F1 值达到了 97.36% 和 95.17%。

综上所述,为了提高 EOSIO 平台智能合约漏洞检测的准确性,本文提出并实现了两种不同的漏洞检测方法。第一种方法侧重于提高漏洞检测的代码覆盖率,第二种方法侧重于实现对智能合约的深度漏洞检测。在实际检测场景下,通过结合两种方法,能够充分发挥其检测能力,从而实现更为全面、精确且高效的 EOSIO 平台智能合约漏洞检测。

关键词: 智能合约 EOSIO 形式化验证 符号执行 漏洞检测

ABSTRACT

With the rapid development of blockchain platforms, smart contracts, as a powerful extension of blockchain platform functionality, have been widely adopted. In order to enhance the execution performance of on-chain smart contracts, some blockchain platforms choose to use WebAssembly (WASM) format as the compilation target for smart contracts, with the EOSIO platform being one of the earliest and most representative blockchain platforms. However, in recent years, smart contracts on the EOSIO platform have been repeatedly exposed to various vulnerabilities, resulting in significant economic losses. One example is the EOSBet smart contract application, which suffered three vulnerability attacks within one month in the fall of 2018, resulting in a total loss of approximately \$1.23 million.

To ensure the security of smart contracts on the EOSIO platform, the academic community has proposed a series of methods for security analysis and vulnerability detection. These vulnerability detection methods can mainly be divided into two categories: formal methods and fuzz testing methods. Formal methods primarily utilize symbolic execution to search for anomalies in the execution state space of EOSIO platform smart contracts. Due to the complexity of the state space, such methods typically require reduction of the state space. However, such reduction may lead to incomplete modeling of the underlying mechanisms, thereby failing to identify vulnerabilities in some complex contracts. In contrast, fuzz testing methods involve random seed generation and mutation processes, resulting in insufficient coverage of code paths and potential oversight of certain vulnerabilities. Overall, current vulnerability detection methods still suffer from accuracy issues and require further improvement.

Therefore, this thesis proposes a research approach based on formal methods, aiming for high-precision vulnerability detection for EOSIO platform smart contracts. The research includes the following key aspects:

- 1) A vulnerability detection method based on loop symbolic execution is proposed and implemented. Considering that the actual execution path of the EOSIO platform smart contracts is influenced by the state of the on-chain database, this method adds support for the on-chain database mechanism in the symbolic execution engine. This enables the method to iteratively perform symbolic execution and update the on-chain database state to improve code coverage and thus enhance the accuracy of vulnerability detection. Additionally, the method mitigates the state explosion issue of symbolic ex-

ecution through memory model optimization and skipping transaction deserialization, without compromising the accuracy of vulnerability detection. Experimental results demonstrate that this method is effective in detecting vulnerabilities in EOSIO platform smart contracts, with a detection accuracy of 96.05% and an F1 score of 92.58%. These results surpass those of the current representative methods, WANA and WASAI.

2) A vulnerability detection method based on fine-grained modeling and verification is proposed. Leveraging the documentation of the WASM instruction set and the underlying virtual machine of the EOSIO platform, this method constructs an automated formal model of the EOSIO platform smart contracts through reverse engineering and static analysis. By using formal verification methods, this thesis avoids the need for heuristic pruning or fixed branch depth pruning to reduce the state space, enabling the constructed model to more accurately simulate the state space of smart contracts, thereby achieving fine-grained analysis of the EOSIO platform smart contracts. Furthermore, this thesis balances detection granularity and detection time by using the verifier to construct paths targeting security properties during verification, avoiding exploration of irrelevant paths. Experimental results show that this formal verification-based vulnerability detection method in this study is capable of effectively identifying vulnerabilities that were difficult to detect using the previous method. As a result, the overall detection accuracy and F1 score have reached 97.36% and 95.17%, respectively.

In summary, to improve the accuracy of vulnerability detection for EOSIO platform smart contracts, this thesis proposes and implements two different vulnerability detection methods. First, the method based on loop symbolic execution fully considers the execution of smart contracts under different on-chain database states, achieving high code coverage vulnerability detection for smart contracts. Second, the method based on formal verification avoids the vulnerability oversight issues caused by pruning methods in symbolic execution, thus achieving precise vulnerability detection for smart contracts. In practical detection scenarios, by combining both methods, their detection capabilities can be fully utilized, resulting in a more comprehensive, accurate, and efficient vulnerability detection for EOSIO platform smart contracts.

Key Words: Smart Contract, EOSIO, Formal Verification, Symbolic Execution, Vulnerability Detection

目 录

第 1 章 绪论.....	1
1.1 研究背景和意义.....	1
1.2 国内外研究现状.....	3
1.2.1 以太坊智能合约漏洞检测方法.....	3
1.2.2 EOSIO 智能合约漏洞检测方法.....	5
1.3 主要研究内容.....	6
1.4 论文组织结构.....	7
第 2 章 背景知识	9
2.1 引言.....	9
2.2 EOSIO 平台智能合约.....	9
2.3 WASM 指令集.....	12
2.4 EOSIO 平台智能合约漏洞.....	15
2.4.1 假 EOS 漏洞	15
2.4.2 假收据漏洞.....	16
2.4.3 链上数据依赖漏洞.....	17
2.4.4 回滚漏洞.....	18
2.4.5 缺少权限检查漏洞.....	18
2.4.6 整数溢出漏洞.....	19
2.5 软件形式化分析方法.....	20
2.6 模糊测试技术.....	22
第 3 章 基于循环符号执行的高代码覆盖率漏洞检测方法	23
3.1 引言.....	23
3.1.1 问题分析.....	23
3.1.2 解决思路.....	24
3.2 具体方法.....	25
3.2.1 相关定义.....	25
3.2.2 循环算法.....	26
3.2.3 符号执行算法.....	27
3.2.4 符号执行算法的优化.....	33
3.2.5 漏洞检测.....	37
3.2.6 路径选择和交易生成算法.....	38
3.3 实验结果与分析.....	39
3.3.1 实验目的与环境.....	39
3.3.2 代码覆盖率对比实验.....	40

3.3.3 算法检测效果实验.....41

3.3.4 优化策略对比实验.....45

3.3.5 真实智能合约检测效果实验.....46

3.4 本章小结.....47

第 4 章 基于 FASVERIF 的深度漏洞检测方法.....48

4.1 引言.....48

4.1.1 问题分析.....48

4.1.2 解决思路.....49

4.2 具体方法.....49

4.2.1 EOSIO 平台智能合约自动建模.....49

4.2.2 属性生成及相关模型修改.....58

4.2.3 自动验证.....62

4.3 实验结果与分析.....64

4.3.1 实验目的与环境.....64

4.3.2 复杂合约检测效果对比实验.....65

4.3.3 智能合约检测效果实验.....67

4.4 本章小结.....68

第 5 章 总结与展望70

5.1 工作总结.....70

5.2 未来工作.....70

参考文献.....72

插图清单

图 1.1	各区块链平台实际每秒吞吐量	1
图 2.1	apply 函数代码示例	10
图 2.2	EOSIO 转账过程	11
图 2.3	on_transfer 接口代码示例	11
图 2.4	WASM 主要语法	13
图 2.5	br_if 在控制块中的使用	15
图 2.6	假 EOS 漏洞的代码示例	15
图 2.7	假收据漏洞的代码示例	16
图 2.8	链上数据依赖漏洞的代码示例	17
图 2.9	回滚漏洞代码示例	18
图 2.10	缺少权限检查漏洞代码示例	19
图 2.11	整数溢出漏洞代码示例	19
图 3.1	存在缺少权限检查漏洞的智能合约示例	24
图 3.2	基于循环符号执行的高代码覆盖率漏洞检测方法框架图	25
图 3.3	算法 3.1 的运行示例	28
图 3.4	EOSIO 平台智能合约链上数据库示意图	31
图 3.5	请求数据反序列化代码	34
图 3.6	代码覆盖率对比实验结果	41
图 4.1	合约状态转移路径示意图	49
图 4.2	基于 FASVERIF 的漏洞检测方法框架图	50
图 4.3	多重集合重写规则	50
图 4.4	加法指令建模规则	52
图 4.5	比较指令建模规则	53
图 4.6	选择指令建模规则	53
图 4.7	入栈指令建模示意	54
图 4.8	出栈指令建模示意	54
图 4.9	跳转指令建模规则	55
图 4.10	if-else-end 控制块建模规则	55
图 4.11	内存初始化指令建模规则	55
图 4.12	函数调用指令建模规则	57

图 4.13	初始函数调用规则	57
图 4.14	库函数调用建模	57
图 4.15	自动建模方法示意图	59
图 4.16	假 EOS 漏洞对应的初始函数调用规则	60
图 4.17	假收据漏洞对应的初始函数调用规则	61
图 4.18	存在整数溢出风险的加法指令建模规则	62
图 4.19	FASVERIF 验证模块工作流程示意图	63
图 4.20	插入合约的循环代码示例	65
图 4.21	插入合约的回滚漏洞代码示例	66

表 格 清 单

表 1.1 受攻击智能合约金额损失 2

表 3.1 实验配置 40

表 3.2 数据集中漏洞合约的数量 42

表 3.3 实验评价指标 43

表 3.4 检测准确度对比实验结果 43

表 3.5 检测对比实验耗时 45

表 3.6 优化策略对比实验 45

表 3.7 真实智能合约检测效果实验 46

表 4.1 复杂智能合约检测效果实验结果 66

表 4.2 复杂智能合约检测时间 67

表 4.3 智能合约检测效果实验 68

第 1 章 绪 论

1.1 研究背景和意义

EOSIO 平台是最为活跃的 DPoS (Delegated Proof of Stake)^[1] 区块链平台之一。随着区块链的发展, 传统的区块链共识协议如比特币所采用的工作量证明 (Proof of Work, PoW) 共识协议, 因为吞吐量较小而难以满足高性能应用程序的需求。为此研究人员提出了不同的共识协议, 其中最为代表性的是权益证明共识协议 (Proof of Stake, PoS) 和委员会证明共识协议 (Delegated Proof of Stake, DPoS)^[1]。在理论情况和实际应用中, POS 和 DPoS 共识协议均表现出远高于传统区块链的吞吐率, 因此成为比特币之后大多数区块链项目选择的共识协议。以太坊作为最具代表性的采用 POS 共识协议的区块链平台之一, 而 EOSIO 则是最具代表性的采用 DPoS 共识协议的区块链平台之一。这两个平台目前都是全球最为活跃的社区之一。如图 1.1 所示, EOSIO 平台所采用的 DPoS 共识协议在理论上支持每秒百万级别的交易吞吐量, 在实际使用中已经达到每秒三千的交易吞吐量, 相较于以太坊平台的峰值高出约 100 倍, 比比特币平台的峰值高出约 400 倍^[2]。自 2018 年 6 月发布以来, EOSIO 仅用三个月就超过以太坊成为 DApp 交易量最大的平台^[3]。当前, EOSIO 的平均交易量是以太坊的 1.217 倍, 总市值已增长到 13.1 亿美元^[4]。

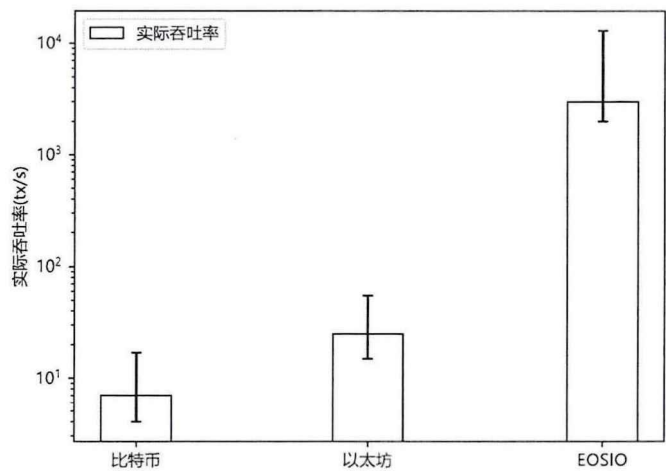


图 1.1 各区块链平台实际每秒吞吐量

EOSIO 平台智能合约是该平台不可或缺的组成部分, 对其他平台也产生了深远影响。智能合约最早由 Szabo 提出, 是一种电子化、自动化的交易协议, 旨在防止恶意或者意外篡改, 减少中介机构, 并保证执行条件和执行情况。尽管

智能合约的概念在早期提出^[5]，但直到进入以以太坊和 EOSIO 为代表的区块链 2.0 时代，才迎来了大规模的应用和发展。区块链上的智能合约使用代码描述协议的触发条件和处理规则，并将代码和运行结果都嵌入到区块链中，通过区块链的共识协议确保数字交互的正确性。EOSIO 平台上的智能合约采用 C++ 编写，随后编译成 WASM^[6]并部署到 EOS 平台的区块链上，由 EOSIO 平台的 WASM 虚拟机中执行。WASM 是一种基于栈式虚拟机的二进制指令集，最初提出是为了提高浏览器中代码执行的性能。相较于传统的区块链字节码如 EVM 字节码，WASM 指令具有更丰富的指令语义，能够支持更多的合约操作，并在执行时表现出更高的性能。因此，WASM 在许多区块链平台中备受青睐。例如，以太坊计划在以太坊 2.0^[7-8]中采用以太坊 WASM 虚拟机，Polkadot 平台目前也使用 WASM 作为其智能合约的最终编译目标^[9]。

EOSIO 平台智能合约正面临着巨大的安全威胁，并因为漏洞攻击造成了大量损失。如表 1.1 所示，2018 年秋季，在 EOSIO 平台上的赌博合约 EOSBet 先后由于 Fake EOS 漏洞和 Fake Notification 漏洞，遭受了两次攻击，损失了 56 万美元，之后又因对于 Fake Notification 漏洞的修复失误导致了约 67 万美元的损失^[10-11]。2018 年 10 月 EOSCast 合约因为 Fake EOS 漏洞损失了 37 万美元^[12]。

表 1.1 受攻击智能合约金额损失

被攻击合约	合约平台	预估损失金额
EOSBET	EOSIO	128 万美元
EOS Happy Slot	EOSIO	2.6 万美元
Newdex	EOSIO	5.8 万美元
EOS.WIN	EOSIO	2.1 万美元
EosRoyale	EOSIO	5.8 万美元
EOSCast	EOSIO	37 万美元

尽管 EOSIO 平台智能合约正面临巨大的安全风险，但相较于以太坊智能合约，其安全研究却相对较少。这主要源于对 EOSIO 平台智能合约的分析相对更为困难，这一困难主要体现在以下四个方面。首先，EOSIO 智能合约使用的 WASM 指令集相对于以太坊字节码更为复杂，涵盖更多种类和数量的指令，例如支持浮点运算和高级跳转等。其次，大多数 EOSIO 智能合约是闭源的，因此无法获取其 C++ 级别的源代码，只能获得区块链上的 WASM 级别代码。与之相对，以太坊平台存在大量开源合约，因此对以太坊智能合约的安全分析主要基于 Solidity 源代码而非字节码。与对源代码的分析相比，对字节码的分析需要考虑更多的因素。一方面，由于字节码的指令操作更加细粒度，因此相同操作需要更多的指令来实现，导致对字节码的分析需要处理更大规模的语句数量；另一方面，字节码

具有更复杂的语法结构，因此对字节码的分析需要处理更复杂的情况。此外，大多数已曝光的 EOSIO 漏洞依赖于 EOSIO 平台特有的特性，因此对其进行分析需要对 EOSIO 平台的内部实现有深入的了解。另外，EOSIO 平台的智能合约的逻辑往往更加复杂。这是因为以太坊平台智能合约的执行需要消耗 gas，消耗数量取决于执行时指令的数量，因此开发人员倾向于编写简洁的合约，而 EOSIO 平台没有这样的限制。

目前，对 EOSIO 平台智能合约的漏洞检测安全研究在准确率、覆盖率等方面尚有不足之处。在这些研究中，基于符号执行的漏洞检测方法为简化状态空间以提高执行效率，通常采用基于主观的提前剪枝以及固定分支深度剪枝等方法，并未充分考虑较为复杂的 EOSIO 内部机制。因此，这些方法往往难以发现路径较长的分支中以及具有复杂条件的分支中的漏洞。基于模糊测试的方法则需要依赖随机算法生成测试用例，以提升对待测试合约的漏洞覆盖率，然而这也导致了结果的准确性和可复现性受到限制，难以精确识别 EOSIO 平台智能合约的漏洞。因此，EOSIO 平台智能合约迫切需要新的漏洞检测方法，以更全面、高效且准确地保护 EOSIO 平台智能合约的安全。

1.2 国内外研究现状

1.2.1 以太坊智能合约漏洞检测方法

以太坊作为当前最受欢迎的区块链平台之一，也是最广泛部署智能合约的平台之一，在学术界研究智能合约安全问题时备受关注。对以太坊智能合约的研究经验对于研究其他平台的智能合约安全性具有重要的参考价值。关于以太坊智能合约的漏洞检测方法，可以划分为三类：自动化目标导向的方法、表达性目标导向的方法和综合目标导向方法。

(1) 自动化目标导向方法

自动化目标导向的方法可以进一步分为模式匹配、符号执行和模糊测试三种方法。

1) 模式匹配方法旨在通过预定义的漏洞模式与合约代码或其他形式的合约进行匹配，以便发现潜在的漏洞。SECURIFY^[13]和 SmartCheck^[14]是两个代表性的工具，前者将智能合约字节码转换为数据日志形式，设计合规和违规模式进行检测，而后者通过在从合同转换的 XML 语法树中搜索特定模式来标记潜在漏洞。

2) 使用符号执行的分析器将每个程序路径表示为命题公式，并识别与给定模式匹配或满足给定约束的路径。OYENTE^[15]和 Mythril^[16]是典型的符号执行工具，前者对 EVM 字节码进行符号执行，检查各种预定义的漏洞模式，而后者

采用污点分析和符号执行来查找漏洞模式。Manticore^[17]则能够分析二进制代码和智能合约源码。

3) 模糊测试方法通过输入大量随机和变异的数据来模拟现实世界中的各种异常情况,以触发合约的异常行为。以 ContractFuzzer^[18]为例,它使用模糊测试技术生成输入并执行合约,通过在 EVM 插桩中提取执行日志,并结合总结出的测试预言来分析漏洞。Harvey^[19]使用灰盒模糊测试技术,在生成输入数据之前预测新输入是否可能覆盖新的执行路径或发现漏洞。sFuzz^[20]结合了现有 C 程序模糊测试器的策略,并增加了自适应策略以选择种子,从而生成的输入能够覆盖更多原本难以覆盖的路径。

自动化目标导向方法虽然节省人力并易于使用,但漏洞检测效果都受限。模式匹配方法和符号执行方法为了实现自动化,必须缩减合约的运行细节。这使得它们无法全面抽象合约的运行状态,进而导致漏洞检测的覆盖率相对较低。模糊测试的方法通过大量的随机输入测试并进行真实执行的方法提高了漏洞检测的覆盖率。但因为模糊测试的方法依赖于运行日志和固定的测试预言进行漏洞检测,其仍然存在漏洞检测准确率较低的问题。为此,有部分工作采用面向表达式的方法以得到更优的漏洞检测效果。

(2) 表达性目标导向方法

表达性目标导向的方法使用逻辑公式定义合约的安全规范并进行验证,以获得更准确可靠的结果。其中,SMARTPULSE^[21]是一种用于自动检查给定时序属性的智能合约方法。类似地,VerX^[22]对使用 PastLTL 编写的时序安全规范进行半自动验证。KEVM^[23]使用 K 框架^[24]构建了 EVM 语言的可执行形式规范,而 Vandal^[25]将低层的字节码转换为逻辑关系,并结合预设的安全逻辑规范进行分析。尽管面向表达式的方法能够获得更准确可靠的结果,但它们只能自动生成简单的不变式用于验证,而对于更复杂的属性规范则需要人工定义,因此无法自动检测现有部分的漏洞。

(3) 综合目标导向方法

综合目标导向的智能合约分析方法旨在结合前述两类分析方法的优点,既保证结果的准确和可靠,又具有较高的自动化程度。代表性的工作包括 eThor^[26], ZEUS^[27]和 FASVERIF^[28]。eThor 是一种针对 EVM 字节码的静态分析器,它将 EVM 字节代码的语义抽象为一组 Horn 子句,并将属性表示为可达性查询,这些查询使用 Z3 来求解。ZEUS 先将 Solidity 智能合约转换为中间语言,然后转换为 LLVM 位代码,从而使用现有的符号模型检测器进行分析,使用预定义策略进行错误检测。FASVERIF 首先对 Solidity 源代码智能合约进行自动建模,然后根据需要自动生成安全属性,最后使用基于安全协议形式化验证工具 Tamarin Prover 的带数值约束求解的证明器进行安全属性的验证。

正如前文所述, EOSIO 平台智能合约在编程语言、底层指令集、运行机制以及漏洞原理等方面与以太坊平台智能合约存在显著差异。因此, 以太坊智能合约的漏洞检测方法虽然具有参考价值, 但无法直接应用于 EOSIO 平台智能合约的漏洞检测。

1.2.2 EOSIO 智能合约漏洞检测方法

目前, 对于 EOSIO 平台智能合约的漏洞检测方法主要可分为两种, 动态分析方法和静态分析方法。

动态分析方法主要包括基于模糊测试的漏洞检测方法。EOSFUZZER^[29]根据 EOSIO 智能合约的接口信息生成模糊测试输入, 使用修改后的 WASM 虚拟机执行合约并收集有价值的执行信息, 然后根据执行信息进行漏洞检测。WASAI^[30]采用混合模糊测试技术, 利用模糊测试生成的智能合约执行路径进行符号执行以构建路径的约束, 对于约束集合进行变异和约束求解以产生自适应种子, 改善 EOSFUZZER 的随机种子生成策略, 提高路径覆盖率。WASAI 采用的是对于智能合约进行插桩以获取执行日志的方式, 而不是 EOSFUZZER 对于 EOSIO 的 WASM 虚拟机进行修改的方式, 因此 WASAI 具有更好的可移植性。WASAI 使用了多种方法以缓解符号执行的状态爆炸问题。WASAI 的内存模型使用 z3 提供的 Array 结构, 将内存表示为字节数组, 能够比其他符号执行引擎更快地从内存中获取内存表达式。WASAI 在执行日志中省略了 EOSIO 平台智能合约统一的对于交易数据进行反序列化的过程, 从而缩短符号执行路径, 以缓解了符号执行的状态爆炸问题。尽管模糊测试虽然相对形式化方法实现简单, 并且测试效率高, 但由于其过程中合约输入的随机性, 导致漏洞检测率和可复现性都得不到保证。

静态分析方法主要涵盖符号执行方法、数据流分析方法和形式化验证方法。EOSAFE^[3]作为一种静态分析框架, 采用基于 WASM 字节码的符号执行来进行漏洞检测。该框架通过采用特定于漏洞的启发式剪枝策略, 缓解了符号执行的状态爆炸问题, 并对库函数采用按需和语义感知的方法进行模拟。WANA^[31]同样利用 WASM 字节码上的符号执行来检测漏洞, 且具有更好的扩展性。WANA 支持对以太坊平台和 EOSIO 平台智能合约的漏洞检测, 并且额外支持一种 EOSAFE 中不被覆盖的漏洞类型。EXGEN^[32]作为首个跨平台的静态漏洞利用生成框架, 通过将智能合约转换为 IR 表示, 并运用数据流分析和符号执行等方法来识别漏洞, 生成相应的漏洞利用合约。值得注意的是, 在这一转换过程中, 由于丢失了许多中间信息, EXGEN 仅支持对 EOSIO 平台智能合约整数溢出漏洞的检测。EOSIOAnalyzer^[33]将 EOSIO 平台合约的 WASM 代码转换为寄存器转移语言 (RTL), 然后进行数据流分析以获取变量间的数据传播关系, 从而进行漏洞检测。EOSDFA^[34]基于 Octopus^[35]设计并实现了一种数据流分析方法, 该方法

将目标函数及其变量转换为静态单一赋值 (SSA) 形式的中间语言, 从而支持对包含指针访问操作的 EOSIO 平台智能合约的数据流分析。EVulHunter^[36] 基于 Octopus 构建控制流图, 并选择控制流图中的特定基本块以进行模拟执行, 从而获取堆栈和内存信息以根据预定义模式检测漏洞。王自生等人^[37] 基于 Coq 实现了对于 EOSIO 平台智能合约 C++ 源码格式代码的形式化验证。然而, 该方法仅支持 C++ 格式的合约源码, 并且验证过程需要专家进行手动属性生成和证明过程交互, 因此仅进行了可行性分析的实验。

此外, 一些关于 WASM Web 应用的安全研究工作具有参考意义。Weikang 等研究人员^[38] 通过 WASM 指令执行路径对应用程序行为进行建模, 发现了加密货币挖矿 WASM 程序的异常行为, 并提出了一种基于浏览器的方法, 用于检测使用 WASM 的密码劫持攻击。Daniel 等研究人员^[39] 专注于 WASM 中二进制漏洞及其缓解措施。Jonathan 等研究人员^[40] 则将形式化验证技术应用于复杂密码组件的 WASM 实现, 提供了两个经过验证的 WASM 加密组件。另外, Wasabi^[41] 实现了一个通用的 WASM 动态分析框架。该框架采用二进制插入, 将用 JavaScript 编写的分析函数的调用插入到 WASM 二进制文件中。通过使用 Wasabi, 可以实现诸如指令计数、调用图提取、内存访问跟踪和污点分析等功能。

目前, 针对 EOSIO 平台智能合约的漏洞检测方法在检测效果上仍然存在不足, 特别是在漏洞检测的准确率和覆盖率等方面。基于模糊测试的漏洞检测方法依赖于对合约日志的分析, 但由于合约日志无法精确反映合约执行的详细信息, 因此难以准确识别潜在漏洞。此外, 基于模糊测试的方法通过随机或半随机算法生成测试输入, 因而漏洞识别准确率不稳定。基于符号执行方法或形式化验证方法的漏洞检测方法为缓解状态爆炸问题, 通常采用基于主观策略的剪枝或者固定分支深度剪枝等技术, 然而这也导致其难以识别深度较深的路径中的漏洞。而基于数据流分析的漏洞检测方法会对智能合约细节进行省略, 因此仅支持少部分漏洞种类的检测。WASAI^[30] 结合使用了符号执行和模糊测试方法, 但却并未深入考虑智能合约的链上数据库状态对漏洞检测的影响, 这限制了它在漏洞检测覆盖率和准确率上的表现。

1.3 主要研究内容

面对当前 EOSIO 平台智能合约漏洞检测方法中存在的问题, 本文以提高漏洞检测的准确率为目标, 从扩展漏洞检测的广度和深度出发, 提出并实现了对 EOSIO 平台智能合约的高覆盖率、高准确率的自动检测方法。本文的主要研究内容和研究成果包括以下两个方面:

- 1) 提出并实现了一种基于循环符号执行的高代码覆盖率漏洞检测方法。本

文发现当前漏洞检测方法缺少对链上数据库机制的考虑,这限制了它们检测使用该机制的智能合约时的代码覆盖率,并影响了漏洞检测的准确率。为此,本研究基于实际执行得到的数据进行链上数据库机制的精确模拟,并根据符号执行的结果进行链上数据库的状态迁移,这两步循环进行以尽可能探索不同链上数据库状态下的智能合约执行情况。同时,为了应对循环符号执行的状态爆炸问题,本研究采用了一系列策略,包括跳过交易反序列化流程和内存模型优化等,成功实现了一个高性能的符号执行方法。实验证明,本研究提出的方法在漏洞检测准确率方面优于其他方法,F1 值到达了 92.58%,准确率到达了 96.05%。

2) 提出并实现了一种基于 FASVERIF^[28] 的深度漏洞检测方法。本研究依托于 WASM 指令集的语义和 EOSIO 平台的源码,构建了一套 EOSIO 平台智能合约的深度形式化建模方案。借助多重集合重写语言,本研究对于建模方案进行了实际实现,并结合符号执行技术实现了建模的自动化。此外,本研究总结了 EOSIO 主流漏洞的安全属性,并实现了安全属性的自动生成。通过对 FASVERIF 的验证器的扩展和优化,本研究能够自动验证上述形式化模型和安全属性,从而实现对于漏洞的自动检测。与使用剪枝以缩减漏洞检测状态空间的符号执行方法不同,本研究的形式化验证方法能够在验证时自动规避与漏洞无关的路径,实现检测粒度和检测时间的平衡。实验结果表明,本研究的漏洞检测方法,能够有效检测前一方法在数据集中未发现的漏洞。综合使用两种方法,本文的漏洞检测方法的 F1 值和检测准确率分别达到了 95.17% 和 97.36%。

上述两个研究内容间是相互补充的关系。研究一的方法在不同链上数据库状态下进行符号执行,并依赖符号执行寻找最能提高代码覆盖率的链上数据库状态迁移,从而有效提高了检测的漏洞覆盖率。并且,由于研究一的符号执行方法中的多种优化策略,研究一的漏洞检测方法能够对 EOSIO 平台智能合约进行快速且高效的漏洞检测。然而,由于研究一采用了固定路径深度的策略以缩减状态空间,所以在漏洞检测的准确率上仍有进一步提升的空间。研究二基于形式化验证方法,通过对智能合约的自动建模和属性生成,消耗一定的时间成本,能够对智能合约进行高准确率的深度漏洞检测,从而为研究一的方法查漏补缺,并形成互补。

1.4 论文组织结构

本文分为 5 章,组织结构如下:

第一章,绪论。本章首先阐述研究的背景,着重介绍了 EOSIO 平台智能合约的研究价值、分析难点,以及确保全面且准确的安全保护的重要性。随后,详细探讨了对比以太坊平台智能合约,EOSIO 平台智能合约在国内外的研究现状,

明确了现有 EOSIO 平台智能合约漏洞检测方法的限制。最后，概述了本文的主要研究内容及整体框架。

第二章，相关背景知识。本章首先阐述 EOSIO 平台智能合约的定义和其特殊的内部机制。然后，介绍了 EOSIO 平台智能合约字节码格式中 WASM 指令集的指令和语法。最后，阐述了软件形式化分析方法和模糊测试方法这两种常见的程序分析方法。

第三章，基于循环符号执行的高代码覆盖率漏洞检测方法。本章针对现有漏洞检测方法对链上数据库机制考虑不足导致代码覆盖率不足、影响漏洞检测率的问题，提出并实现了一种基于循环符号执行的高代码覆盖率漏洞检测方法，以应对 EOSIO 平台智能合约漏洞检测的特殊需求。

第四章，基于 FASVERIF 的深度漏洞检测方法。本章面对现有基于形式化的漏洞检测方法使用启发式算法和固定循环深度导致漏洞检测遗漏的问题，提出并实现了一种对 EOSIO 平台智能合约进行自动建模和验证的方法，以提升漏洞检测的全面性和准确性。

第五章，总结与展望。本章对提出并实现的 EOSIO 平台智能合约漏洞检测方法进行总结，并探讨了未来研究的方向和展望。

第2章 背景知识

2.1 引言

智能合约是 EOSIO 平台重要的组成部分，管理着平台上大量的资金，但也面临着广泛的安全威胁。为了提供对 EOSIO 平台智能合约的全面且准确的安全保护，本章首先介绍了 EOSIO 平台的相关机制和智能合约所使用的 WASM 指令集，并详细介绍了目前主要的六类 EOSIO 平台智能合约漏洞。其次，本章介绍了针对各区块链智能合约的两类主要漏洞检测方法，包括软件形式化分析方法和模糊测试方法。

2.2 EOSIO 平台智能合约

账户名是指 EOSIO 平台上每个用户被分配的唯一标识，也被称作账户地址。该字符串的长度介于 1 到 12 个字符之间，字符可以是小写字母（'a'-'z'）、数字（1-5），以及可以放置在除最后一个位置外的点号（'.'）。账户名被设计为人类可读的形式，用于说明和用户交互。然而，在合约内部使用和链上数据存储时，EOSIO 会将账户名通过 base32 编码转换成 64 位的无符号整数。EOSIO 平台上，用户可以针对每个账户名自定义权限，以实现资源访问控制、智能合约接口调用、事务签名等操作的访问限制。EOSIO 账户的权限结构为分层的父子结构，即父权限具有子权限所有能力。

智能合约是指 EOSIO 平台上每个账户对应的一段 WASM 代码，通过账户名可以进行调用，从而实现代币转账、游戏行为等操作。EOSIO 平台提供一个官方的编译链 EOSIO.CDT，EOSIO.CDT 基于 Clang 7，支持将用户编写的 C++ 智能合约代码编译到 WASM 格式。EOSIO.CDT 中提供了官方的 C++ 代码库，以方便用户快速完成对于智能合约的功能实现，比如执行转账、链上数据库的增删改查以及调用其他合约等操作。在 EOSIO 平台上部署智能合约时，除了合约代码外，还需要合约接口文件。该文件描述了当前智能合约可供外部调用的接口名称和参数类型，其中的参数类型可以是支持嵌套的结构体类型。接口文件还可以描述当前智能合约所使用的链上键值数据库的键值类型，数据库的值类型同样可以是支持嵌套的结构体类型。EOSIO.CDT 通过对于编译器的修改，支持用户通过对接口函数进行简单标注，自动生成接口文件和合约入口函数。合约入口函数负责处理合约接收到的调用请求，判断请求中的接口名和参数类型是否匹配，并在检查完成后执行接口对应的相应函数。用户可以通过标注自动生成入口函数，也可以自定义入口函数，只需编写一个作用域为全局的名称为 `apply` 的函数。如

图 2.1 所示, `apply` 函数的参数是固定的三个 64 位的无符号整数 (`receiver`, `code`, `action`), 分别表示请求的接收账户, 请求的被调用账户, 请求的接口名。

```

1 void apply ( uint64_t receiver , uint64_t code , uint64_t action ) {
2     if( action == N( onerror ) ) {
3         check ( code == N( eosio ) , "exception captured" );
4     }
5     auto self = receiver ;
6     if( ( code == self || code == N( eosio . token ) ) ) {
7         switch( action ) {
8             case N( transfer ) : // action == N(transfer)
9                 //接下来进行判断:
10                // 是对于 transfer 接口的直接调用
11                // 还是被其他合约的 require_recipient 通知了
12                ...
13            }
14        }
15 }

```

图 2.1 `apply` 函数代码示例

在被调用时, 智能合约实际运行于构成 EOSIO 区块链的各个节点中。每个 EOSIO 节点都是 `Nodeos` 程序的一个进程, 负责维护区块链数据持久层、建立 P2P 网络和调度合约代码。EOSIO 平台的用户可以使用 `cleos` 命令行工具或者 HTTP 协议和 EOSIO 节点进行通信, 实现创建账户、发布合约、调用合约接口、查询余额和查询链上数据等操作。当 EOSIO 节点收到合约调用请求后, 如果该节点不是负责生产区块的节点, 它会将请求广播到区块链网络中; 否则它将根据调用请求获取对应账户的合约代码, 并在 WASM 虚拟机中以调用请求作为输入来执行合约。

智能合约请求调用的最小单位是动作 (`action`), 代表对合约的一个接口的一次调用。单个或多个动作可以合成一个交易 (`transaction`), 交易是被验证和打包进区块链的最小单元。动作除了可以由用户输入外, 还可以由合约在运行时, 使用序列化后的动作数据调用 `send_inline` 函数来产生, 这样产生的动作被称作内联动作 (`inline action`)。相关的函数还有 `send_context_free_inline` 和 `send_defered`, 前者产生的动作是上下文无关的, 后者需要传入序列化的交易数据, 并且交易会被推迟到之后再被打包放入区块链。前述所提的 `send_inline` 等几个函数, 来自 EOSIO 官方提供的函数库, 又被称为库函数。类似于系统调用, 库函数是 EOSIO 虚拟机向智能合约提供的一组底层 API, 用于扩展智能合约的能力, 提供了对于链上数据库的增删改查、查询区块链状态、获取请求信息、查询余额等功能。默认的内联动作会继承它的父动作 (触发产生内联动作的合约执行的动作) 的上下文 (包括权限信息), 而且当合约响应内联动作时出现错误时, 整个交易 (包括交易中的所有动作) 都会被回滚。而通过调用 `send_context_free_inline` 产生内联动作不会继承父动作的上下文, 但是回滚时同样是整个交易一起回滚。通过 `send_defered` 产生的交易不会继承动作的上下文, 并且因为产生了新的交易, 出

现错误也不会导致之前的交易被回滚。

除动作之外，智能合约中还存在一种称为通知（notification）的机制，也可用于合约间的调用。类似于内联动作，通知也只能在智能合约运行时通过调用库函数来产生，对应的库函数是 `require_recipient`。但与内联动作不同的是，通知的产生只需要指明通知的接收者，而内容则会自动设置为与当前输入智能合约的动作（或通知）相同。这也是为什么 `apply` 函数需要两个参数（`receiver, code`）分别用来表示请求的接收账户和请求的被调用账户：如果调用链中只存在动作，这两个参数会保持一样；一旦存在通知，接收通知的智能合约，其 `apply` 函数的 `receiver` 参数为接收通知的智能合约，而 `code` 参数为前一个动作的被调用账户地址。

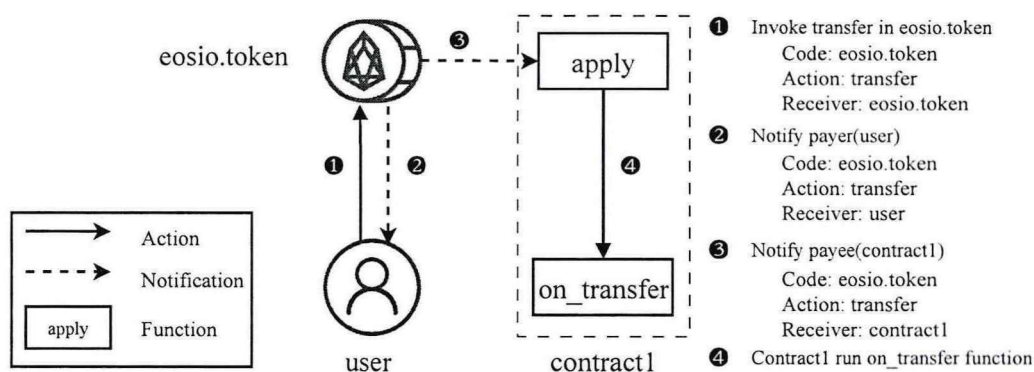


图 2.2 EOSIO 转账过程

```
1 [[eosio::on_notify("*::transfer")]]
2 void on_transfer( const name from, const name to, const asset quantity,
   const string memo )
3 {
4     require_auth( from );
5     // do something
6 }
```

图 2.3 on_transfer 接口代码示例

下面以 EOS 代币的转账过程为例，具体说明动作和通知的使用。EOS 代币是由 EOSIO 官方发行的代币，它发行于 eosio.token 这个账户的智能合约上。要进行 EOS 代币的转账，用户需要请求 eosio.token 的 transfer 接口，这就是图 2.2 中的步骤一：code（请求的被调用账户）为 eosio.token，receiver（请求的接收账户）为 eosio.token，action（请求的接口名）为 transfer。然后 eosio.token 账户上的合约会向转账的发起方和接收方先后发送通知，也就是图中的步骤二和三，可以看到，新产生的通知内容只是修改了转账操作的接收者。发送方或者接收方收到通知后，就可以认为这个转账已经完成，如果他们有设置 on_transfer 接口，那么 apply 函数会自动调用对应的函数，从而进行发送（接受）代币后需要进行的操作。如表 2.3 所示，on_transfer 接口的参数是固定的，分别表示转账发送方账户，

转账接收方账户，转账的代币信息（名称和数量）和转账的附言。

链上数据库机制是 EOSIO 平台智能合约中最主要的数据持久化方法。在 EOSIO 平台智能合约中，变量通常是非持久的，即在智能合约重复执行时，所有变量的值都会被重置。然而，这种无记忆的智能合约不能满足部分场景下的用户需求，例如智能合约中需要保存客户信息或者执行状态等关键信息的情况。因此，EOSIO 平台提供了链上数据库机制，允许合约开发者自定义数据库结构，并通过 API 进行数据的增删改查和排序操作。在智能合约中，可以调用数据库类库函数（如 `db_store`，`db_find` 等）以使用链上数据库。链上数据库的数据实际存储在 EOSIO 区块链上，因此具有高度的安全性和可靠性。由于链上数据库的易用性和数据可靠性，它成为了 EOSIO 平台智能合约开发者首选的数据持久化方式。

2.3 WASM 指令集

WebAssembly (WASM) 是一种可移植、高性能的二进制指令集，是 EOSIO 平台智能合约所采用的二进制格式。尽管 WASM 的诞生是为了提高网页应用的执行性能，但作为一种平台中立的中间表示，WASM 指令集也被同时应用于边缘计算、嵌入式系统和智能合约等多个领域中。WASM 作为一种二进制的指令集，虽然易于机器的读取和执行，但是并不适合开发者进行编码和调试，因此 WASM 存在对应的文本格式 (WAT, WebAssembly Text Format)。WASM 指令集的文本格式是一种具有更好可读性的 WASM 表示方式，通常用于 WASM 程序的开发、调试和分享，并且官方提供了工具用以相互转换二进制格式的 WASM 程序与 WAT 格式的 WASM 程序。除非特殊说明，下文都将使用 WASM 指代 WASM 程序的文本格式。

图 2.4 展示了 WASM 的主要语法结构。在 WASM 的文本表示中，WASM 的基本单位是模块 (module)。每个模块是一个独立的实体，包括了代码和数据，可以被其他模块引用。模块由多个域组成，各个域分别表示了模块的代码、数据、函数签名、导入导出信息等。内存域 (memory) 中定义了当前模块可以访问的内存空间的初始大小和最大大小。WASM 的内存空间是一块支持随机读写的顺序字节数组，偏移从 0 开始，每个模块最多只有一个内存。数据域 (data) 中定义了内存中初始数据的偏移和具体值，其中 `byte*` 表示用于初始化的二进制字符串。函数域 (function) 中定义了函数的标识符、函数的参数个数和类型、函数的返回值类型、函数需要的局部变量的个数和类型、以及函数内的指令等信息。函数域中的函数被执行时，除可以访问内存空间、全局变量空间和局部变量空间（函数的参数被放入局部变量空间）之外，还可以访问对应于当前函数调用的

```

module ::= funciton* global* start? table? memory? elem? data?
function ::= typefunc (import|code) export*
global ::= typeval (import|init) export*
start ::= idxfunc
table ::= import? idxfunc* export*
elem ::= instr import*
memory ::= import? byte* export*
data ::= instr "byte*"
import ::= "name", export ::= "name"
code ::= (local typeval)* instr*
init ::= instr*

instr ::= nop|drop|select
        |typeval.const value|unary|binary
        |typeval.load|typeval.store|memory.grow
        |(set|get|tee)_local idxlocal|(set|get)_global idxglobal
        |call idxfunc|call_indirect typefunc|return
        |block|loop|if|else|end
        |br label|br_if label|br_table label+
unary ::= i32.eqz|...|f32.neg|...
binary ::= i32.add|...|i32.eq|...

typeval ::= i32|i64|f32|f64
typefunc ::= typeval* → typeval*
label ∈ N, idxfunc|idxglobal|idxlocal ∈ N

```

图 2.4 WASM 主要语法

局部栈空间。全局域（global）中定义了全局变量的标识符和初始值，所有函数运行时都可以访问到这些全局变量。类似于内存域和数据域分别定义了内存和内存中的初始值，WASM 中还有表格域（table）和元素域（elem），分别定义了表格和它的初始值，不过不同于数据域中描述的字节数组，元素域描述为存放函数指针的数组。导入域（import）中定义了导入模块中的函数签名，包括函数标识符，函数参数的个数和类型等信息；导出域（export）中定义了导出给外部模块访问的函数、全局变量以及内存。WASM 主要包括四种变量类型（type_{val}）：i32, i64, f32, f64，分别对应于 32 位整数，64 位整数，32 位浮点数和 64 位浮点数。

WASM 中最主要的指令（instr）是算术指令，用于进行数据运算操作。算术指令会从操作数栈中弹出指定数量的操作数，然后将运算后的结果放回操作数栈中。根据操作数的不同类型（type），WASM 的算术指令会具有不

同的变种。算术指令中的主要二元操作包括：`type.add`加、`type.sub`减、`type.mul`乘、`type.div`除、`type.eq`比较是否相等、`type.neq`比较是否不等、`type.lt`比较是否小于、`type.gt`比较是否大于、`type.ge`比较是否大于等于、`type.le`比较是否小于等于、`type.eqz`比较是否为零、`type.rem`求余、`type.and`按位与、`type.or`按位或、`type.xor`按位异或、`type.shl`左移位、`type.shr`右移位、`type.min`计算最小值、`type.max`计算最大值等。算术指令的主要一元操作包括：`type.clz`计算二进制表示前导零的个数、`type.ctz`计算二进制计算末尾零的个数、`type.popcnt`计算二进制表示中位为一的数量、`type.abs`计算绝对值、`type.neg`计算负值、`type.ceil`计算向上取整、`type.floor`计算向下取整、`type.trunc`计算截断整数部分、`type.nearest`计算最接近的整数等。其中部分算术指令只对浮点数类型有效，如`type.floor`、`type.ceil`、`type.trunc`、`type.nearest`、`type.abs`、`type.neg`。部分算术指令则只对整数类型有效，如`type.and`、`type.or`、`type.rem`、`type.xor`、`type.shl`、`type.shr`、`type.clz`、`type.ctz`、`type.popcnt`。一些对整数类型有效的算术指令中还分别存在对于无符号数和对于有符号数的变种，需要分别为原有指令添加后缀u或者s。

在 WASM 中，压栈指令和出栈指令负责数据的复制和传递。压栈指令根据入栈值的不同来源可以分为：`type.const`压入常量；`get_local i`压入立即数*i*对应的局部变量的值；`get_global i`压入立即数*i*对应的全局变量的值；`type.load[len] offset`栈顶值加上偏移offset作为内存地址值，取的字节数根据len决定，如果没有len就根据type类型决定，最后会将取得的内存值转换为type类型，然后放入栈顶。出栈指令根据出栈值的不同存放位置可以分为：`drop`弹出后直接丢弃栈顶值；`type.set_local i`将栈顶值放入立即数*i*对应的局部变量；`type.set_global i`将栈顶值放入立即数*i*对应的全局变量；`type.store[len] offset`栈顶值是要存入的数据，栈顶下面的值加上偏移offset作为内存地址，将数据放到对应的内存位置，存入的字节数根据len决定，如果没有len就根据type类型决定。`select`指令可以看做特殊的出栈指令，它首先从栈中弹出三个操作数，如果第一个弹出的值是0，压入第二个弹出的值，否则压入第三个弹出的值。

WASM 中的控制流相关指令用于定义程序的分支和循环结构。如表 2.5所示，WASM 中可以使用`block`加上`end`定义一个控制块，在控制块中可以使用`br_if i`指令根据当前栈顶值决定是否要跳转，跳转的位置为跳出立即数*i*+1层控制块。`br i`指令则是直接跳出；`br_table`也是立即跳转，但会带有多个立即数参数，运行时根据栈顶的值决定使用哪个立即数作为跳出层数参数。类似的还有`if`加上可选的`else`以及一个`end`形成的控制块，执行到`if`指

```

1 block
2   block
3     get_local 0
4     i32.const 10
5     i32.eq
6     br_if 1 -----',
7     ;;当local0变量为0时不发生跳转
8     get_local 0
9     i32.const 1
10    i32.add
11    set_local 0
12  end;;第二个block对应的end
13end;;第一个block对应的end <-----,

```

图 2.5 br_if在控制块中的使用

令的时候会弹出栈顶的值，根据它决定接下来要执行if到else之间的指令还是else到end之间的指令，这两个分支控制块里面同样可以使用br_if等跳转指令。WASM中还可以使用loop加上end定义一个循环控制块。不同于前文所述的分支控制块，在循环控制块中，br_if i等跳转指令不是跳出控制块，而是跳到立即数i+1层循环控制块的开头。

在WASM中，函数调用指令用于调用模块内外函数。WASM中有两个函数调用指令，首先是call i，其中立即数i是被调用函数的序号。执行该指令时会从栈中弹出与被调用函数参数个数一致的操作数作为调用参数，并在调用完成后将被调用函数的返回值放回调用者函数的操作数栈中。其次是call_indirect指令，它会弹出一个栈顶的值作为当前模块表格（table）的索引，然后使用表格中的函数指针进行调用。

2.4 EOSIO 平台智能合约漏洞

本节将详细介绍EOSIO平台智能合约的六类主要漏洞，包括假EOS漏洞、假收据漏洞、链上数据依赖漏洞、回滚漏洞、缺少权限检查漏洞和整数溢出漏洞。为了清晰说明，本节将使用存在漏洞的智能合约C++源码作为示例，但实际漏洞存在于智能合约的二进制表示（WASM）中。

2.4.1 假EOS漏洞

```

1 void apply(name receiver , name code , name action) {
2   if(action==N(transfer)){
3     // 修复: assert(code==N(eosio.token), "");
4     run(on_transfer); // 执行on_transfer接口的处理
5   }
6   else if (code==receiver||code==N(eosio.token)){
7     EOSIO_API(action); // 执行其他函数
8   }
9 }

```

图 2.6 假EOS漏洞的代码示例

EOS 代币是 EOSIO 官方发行的代币，它通过 `eosio.token` 账户上的智能合约被发行。`eosio.token` 账户上的智能合约同时也是 EOSIO 官方提供的一个代币模板，用于想要自定义代币的开发者进行参考。因此 `eosio.token` 账户的智能合约是开源的，这也意味着任何人都可以通过在自己的地址上部署该智能合约，从而创建和发行一个同样名为 EOS 的代币。因此攻击者可以自己在一个地址上部署 `eosio.token` 账户上的智能合约，从而发行一个和官方代币同名的代币。在 2.2 本文描述过 EOS 代币的转账过程，其中转账的接收方是通过收到一个通知从而知晓自己收到了一笔转账。于是攻击者可以通过调用他的假代币合约的转账接口，来向任意账户发送通知，并且这对于他来说是没有开销的。如果转账接收方的智能合约中，对于他收到的通知，只判断了通知是否是调用 `on_transfer` 接口，并在判断成功后就进行相应的接收转账后的操作，那么该智能合约就存在假 EOS 漏洞。甚至如果转账接收方的智能合约中没有判断接收到是通知还是动作，那么攻击者不需要布置他的假代币合约，直接调用转账接收方合约的 `on_transfer` 接口即可。图 2.6 中的 `apply` 函数就存在假 EOS 漏洞，第 2 行中判断了调用接口是否是 `on_transfer` 之后就直接进行了对应接口的处理，攻击者只需要直接调用该合约的 `on_transfer` 接口即可完成攻击。代码中的 `N` 是一个 EOSIO 官方提供的宏定义，会将括号中包裹的内容在编译时替换为 `base32` 编码后的内容。

要修复智能合约的假 EOS 漏洞，需要在合约的 `apply` 函数中进行对于接收的通知是否来自于 `eosio.token` 合约的判断，更具体的说，是判断 `apply` 函数的第二个参数（`code`，请求的被调用账户）是否为 `N(eosio.token)`。如果攻击者使用直接调用漏洞合约的 `on_transfer` 接口的方法来进行攻击，那么漏洞合约执行时的 `apply` 函数的第二个参数是漏洞合约的地址；如果攻击者使用通过调用自己的假代币合约从而间接调用触发漏洞合约的 `on_transfer` 接口的攻击方式，那么漏洞合约执行时的 `apply` 函数的第二个参数是攻击者的假代币合约的部署地址。因此漏洞合约只需要添加对于 `apply` 函数的第二个参数的判断即可。对于图 2.6 中的 `apply` 函数，只需要按照第 3 行的修复代码，在判断是对于 `on_transfer` 接口的调用后，就进一步判断 `code` 参数是否是 `N(eosio.token)`。

2.4.2 假收据漏洞

```
1 [[eosio::on_notify("*::transfer")] void on_transfer(name from, name to,
  asset quantity, string memo){
2   // 修复: if (to != get_self()) return;
3   action(...).send(); // 敏感操作
4 }
```

图 2.7 假收据漏洞的代码示例

针对假收据漏洞，攻击者会仿照 `eosio.token` 账户合约发送的通知，伪造一个通知发给受害者合约。对于存在假收据漏洞的智能合约来说，该通知和来自

eosio.token 账户合约的通知相同。具体来说，攻击者在自己的两个账户之间进行 EOS 代币转账，当接收代币的那个攻击者账户接收到来自 eosio.token 账户上智能合约的通知后，在该账户的智能合约中进行处理时（on_transfer 接口），它可以调用 require_recipient 库函数生成通知，通知的接收方设置为受害者合约地址，从而将 eosio.token 的通知转发出去。对于攻击者账户生成的这个通知，受害者合约的 apply 函数执行时，apply 函数的三个参数，包括通知的接收者合约、通知对应的动作的调用合约地址，通知对应动作的调用接口名，都和当受害者合约接收 EOS 转账时，来自 eosio.token 的通知一样，因此可以通过 apply 函数中的检查，进入到受害者合约的 on_transfer 接口对应的处理函数。如果受害者合约没有在 on_transfer 接口的处理函数中，进一步判断该转账的目标账户是否是自己，那么他就存在假收据漏洞，会被该假收据欺骗并进行收到 EOS 代币转账后的操作。图2.7中展示了存在假收据漏洞合约的 on_transfer 接口处理函数，其中并没有判断该转账的接收者账户地址是否和当前合约的地址匹配。

要修复智能合约的假收据漏洞，需要在合约的 on_transfer 接口的处理函数中，进行对于转账的目标账户是否是当前合约的判断，也就是判断 on_transfer 函数的 to 参数是否为当前合约的地址。对于图2.7中的合约而言，就是添加第 2 行中的检查语句，检查转账接收者账户地址是否和当前合约地址匹配，如果不匹配直接退出。

2.4.3 链上数据依赖漏洞

```
1 class EOSRoyale: public eosio::contract {
2     ...
3     void rand() {
4         checksum256 result;
5         auto mixedBlock = tapos_block_prefix() * tapos_block_num();
6         // 修复：使用正确的 PRNG 服务
7         const char * mixedChar = (const char *)(&mixedBlock);
8         sha256((char*)mixedChar, sizeof(mixedChar), &result);
9     }
10 }
```

图 2.8 链上数据依赖漏洞的代码示例

链上数据依赖多发于 EOSIO 平台上的赌博类的智能合约。这些合约有的需要使用合理的随机数生成算法进行赌博以保证其公平性，比如进行猜大小、猜奇偶等活动。有的赌博合约会通过库函数，获取区块链上运行时刻的信息（比如当前区块数，当前区块头等）作为种子来进行随机数的生成。因为 EOSIO 平台上的合约二进制都是公开的，攻击者可以获取其二进制编码并进行逆向和随机数生成算法的破解。如果这些合约只使用了这些区块链上的运行信息作为随机数生成算法的种子，而这些信息对于攻击者来说是可以预知甚至可以控制的，于是攻击者可以通过选择合适的时刻下注合适的金额，以破坏赌博的公平性。如

图2.8所示，rand 是合约的随机数生成函数，rand 函数中使用 tapos_block_prefix、tapos_block_num 获取链上状态以产生随机数，因此该合约存在链上数据依赖漏洞。

要修复智能合约的链上数据依赖漏洞，智能合约应该选择更安全的伪随机数生成算法，比如依靠去中心化的 oracle 或者使用后端服务器的随机数生成器来生成。在图2.8中，就是要使用随机数生成服务替换第 8 行。

2.4.4 回滚漏洞

```

1 void apply(name receiver , name code , name action) {
2     if (action == N(transfer))
3         run(reveal); // 执行 reveal 函数
4     ...
5 }
6 void reveal(name from , name to, asset quantity , string memo) {
7     eosio_assert(quantity >= asset("10.0000 EOS"), "exit");
8     int a = tapos_block_prefix () * tapos_block_num ();
9     if (a % 13) {
10        action(permission_level{ _self, N(active) },
11              // 使用 send_defered 进行转账
12              N(eosio.token), N(transfer),
13              std::make_tuple(_self , N(from), asset(1000) , std::string("test
14              "))).send();
15 }

```

图 2.9 回滚漏洞代码示例

本文在2.2中提过，当对于一个内联操作的响应出现错误时，整个交易（包括交易中的所有操作）都会被回滚。攻击者可以利用这一点，恶意回滚不利于他的交易。考虑一个赌博合约，他接收各个账户的转账，并在达到一定金额后进行开奖，将不同金额的代币发送给不同的参与者。如果这个赌博合约在开奖的时候，使用内联操作来调用 eosio.token 的转账接口，那么攻击者可以作为该赌博合约的用户，在他的智能合约中实现 on_transfer 接口，并在 on_transfer 接口的响应函数中查询自己的余额是否增长以获知开奖结果，当他如果发现余额没有增长就故意地产生错误而造成整个交易的回退，从而抵赖这次赌博。如图2.9所示，在 reveal 函数中，合约使用 action().send() 的方式向用户（N(from)）进行转账，而 action().send 内部调用的是 send_inline 库函数，因此合约存在回滚漏洞。

要修复智能合约的回滚漏洞，赌博类智能合约在开奖的时候，不应该使用 send_inline, send_context_free_inline 库函数生成内联操作，而是应该使用 send_defered 库函数生成一个交易。对于图2.9中的合约而言，就是要将第 8 行到第 11 行的 action().send() 进行转账的方式，改成使用 send_defered 进行转账。

2.4.5 缺少权限检查漏洞

EOSIO 平台智能合约中，如果要进行敏感操作（比如转账、更新链上数据

```

1 void on_transfer(name from, name to, asset quantity, string memo) {
2     // 修复: require_auth(from);
3     action(permission_level{from, N(active)},
4             N(eosio.token), N(transfer),
5             make_tuple(from, to, quantity, memo)
6             ).send();
7 }

```

图 2.10 缺少权限检查漏洞代码示例

库), 在这之前都需要检查调用者是否具有相应的操作权限。否则就意味着, 攻击者可以越过合约用户的权限设置和 EOSIO 平台的权限检查机制, 直接执行敏感操作。如果 EOSIO 平台智能合约中存在这样的缺少权限检查的路径, 那么该合约就存在缺少权限检查漏洞。图2.10中展示了一个存在缺少权限检查漏洞合约的 `on_transfer` 函数, 其中并没有进行权限的检查, 就直接执行了敏感操作 `action().send()` 以进行转账。

要修复智能合约的缺少权限检查漏洞, 开发者应该通过手动或者自动的检查, 确保每次进行敏感操作之前, 都完成了对于调用者携带权限的检查。在图2.10中, 就是需要在第 2 行加上 `require_auth(from)` 语句以进行对于调用者权限的检查。

2.4.6 整数溢出漏洞

```

1 void batchtransfer(symbol_name symbol, account_name from, account_names
   name0, account_names name1, account_names name2, account_names name3,
   uint64_t balance){
2     account fromaccount;
3
4     eosio_assert(balance > 0, "must transfer positive balance");
5
6     uint32_t amount = balance * 4; // 转账金额计算
7     // 修复: eosio::check(balance*4 <= std::numeric_limits<uint32_t>::max
   (), "Integer overflow");
8
9     int itr = db_find_i64(_self, symbol, N(table), from); // 获取当前数据
   库中 from 账户的行
10    eosio_assert(itr >= 0, "Bad Table Name");
11
12    db_get_i64(itr, &fromaccount, (account)); // 获取数据库中 from 账户的余
   额
13    eosio_assert(fromaccount.balance >= amount, "overdrawn balance"); //
   检查转账金额是否超出 from 账户的余额
14
15    // 修改账户余额
16    sub_balance(symbol, from, amount);
17
18    add_balance(symbol, name0, balance);
19    add_balance(symbol, name1, balance);
20    add_balance(symbol, name2, balance);
21    add_balance(symbol, name3, balance);
22 }

```

图 2.11 整数溢出漏洞代码示例

和传统的二进制代码相同, 由于 EOSIO 平台智能合约的 WASM 支持了定长

整数，因此会产生整数溢出的风险。更具体的说，在 EOSIO 平台智能合约执行过程中，当对于定长整数进行乘法、加法或是减法等算术操作的时候，计算结果的值可能超出定长整数的表示范围，而 WASM 写入结果采用的却仍然是和操作数相同长度的定长整数，这使得最后的计算结果可能会过大或过小，攻击者可以利用这一点来绕过一些合约内部的限制。如图 2.11 所示，这是一个存在整数溢出漏洞的合约。这个合约的 `batchtransfer` 函数用从将 `from` 账户向 `name0`, `name1`, `name2`, `name3` 等四个账户发送相同金额 (`balance`) 的自定义代币 (`symbol`)。在这个函数中，第 6 行计算了转账总金额，也就是单个转账金额乘四，因为这里结果数据类型是 32 位的无符号整数，因此这里可能发生整数溢出，导致 `amount` 成为一个较小的数。于是在第 12 行的检查转账发送方的余额是否足够时，就可以成功通过条件，从而从一个账户中成功得到超过它余额的转账。

要避免智能合约的整数漏洞，开发者应该检查用户的输入是否会在之后的运算过程中发生溢出。在图 2.11 中，就是如第 7 行所做，添加对于 `balance*4` 后是否会溢出的判断，并在出现整数溢出时终止合约执行。

2.5 软件形式化分析方法

软件形式化分析方法的主要目的是实现对于软件系统正确性的证明。它的基本流程有三步：

1) 首先是基于系统语义的建模，它需要根据软件系统的语义，使用形式化建模的语法以描述软件系统可能的行为。因为软件系统的语义多样性，建模的语法也是多种多样的。例如，串程序的语义通常可以使用状态机 (State Machine) 建模：状态机的状态包括程序中每个变量的值和当前需要执行的代码，每一行代码被建模为对于状态机的状态修改，串程序在执行中可能的行为可以描述为其状态机可达的状态。对于并行程序可以使用进程代数进行建模，它使用代数运算符表示进程的行为，并通过定义并发组合、通信、选择、迭代等运算符完成常见的进程并行行为的表示。

2) 其次是定义待证明的性质，也就是需要定义什么样的软件行为是正确的行为。这种对软件行为正确性的定义被称为规约 (Specification)，在使用形式化方法进行漏洞检测的工作中也被称作安全属性 (Security Property)。比如，程序员在代码里常常使用的 `assert` 等检查语法就是一种规约，但是这些规约只能在程序实际运行时才能进行被检查，并且用它能够表达出的软件行为也受限。因此，形式化方法中通常使用更加严谨的逻辑公式编写规约，这样的规约被称为形式化规约 (Formal Specification)。

3) 最后是执行对于性质的证明。基于之前定义出的形式化模型和形式化规

约，目标是完成性质的证明，或者判断软件系统无法满足性质。常用的证明方法主要包括定理证明和模型检验两种。下面将分别介绍这两种方法。

定理证明的基本思路是把待证明性质作为一个命题，把软件系统语义作为事实，使用推理规则进行命题是否成立的推导。按照自动化程度从低到高的顺序，定理证明可以分为手动定理证明、半自动定理证明和自动定理证明三类。手动定理证明要求用户手动进行命题的证明，包括选择适当的推理规则、应用引理等。手动定理证明为用户提供最高的证明灵活性和安全性保证，但会耗费大量的人力成本和时间成本，而且手动定理证明基本上无法完成复杂系统或者大规模系统的安全性证明。半自动定理证明中，自动工具会协助用户进行证明，比如推理规则的搜索和选择、自动的推理规则应用等，但用户仍然需要提供证明过程中的指导。相比于手动证明，自动工具的使用节省了研究者大量的精力，从而有了实现复杂或者大规模系统的安全性证明的可能。自动定理证明中，用户在手动定义形式化规约后，全部的证明过程都可以自动完成。

模型检验时的基本思路是自动遍历软件系统的所有可能状态，检查性质是否在某个状态中被违反，如果没有则证明成功，否则就找到了反例。模型检验的方法适用于系统的可能状态相对有限的情况，如果系统可能的状态过多，模型检验的方法可能无法在可接受的时间内终止，这样的问题可能是循环次数太大和输入太复杂导致的。模型检验方法的主要挑战就在于如何处理程序状态空间爆炸的问题。限界模型检验就是为了缓解这一办法而提出的。其基本想法是对于可能造成状态爆炸的模型参数设置边界，比如限制循环最大迭代次数等，从而有效避免状态爆炸问题。限界模型检验的主要缺点是它不能保证能够发现程序中所有的异常，因为它的限制缓解了状态爆炸也导致了部分状态被忽略。

符号执行也是最常见的形式化方法之一，其基本思想是将程序的输入看作可以取任意值的符号量，然后进行程序的模拟执行。当使用符号量作为判断时，就产生路径的分叉，分叉出的两条子路径分别具有一条不同的约束。模拟执行得到的路径集合可以表示程序所有可能的运行结果，将路径的当每条路径都满足形式化规约即可完成定理证明。面对状态爆炸问题，符号执行方法中往往采用固定路径（分支）深度、启发式策略剪枝等方法以缓解。虽然符号执行方法和限界模型检验的证明过程类似，但是模型检验一般使用以带命题的有限状态机描述的模型作为其输入，因此能够处理多线程系统；而符号执行则只需要程序，因此具有更高的自动化程度。

2.6 模糊测试技术

模糊测试是一种测试软件系统的方法，其基本思想是向软件系统输入随机或半随机的数据以发现软件中的潜在漏洞和缺陷。主要流程包括：

1) 生成模糊测试用例：取决于被测试程序的类型，生成随机或者半随机的输入数据，数据的类型可能是文本、二进制数据或者消息等。2) 注入模糊测试用例：依据程序的接口（命令行、通信端口等），将生成的模糊测试输入数据注入到程序中。3) 监控程序行为：通过自动化工具，收集、记录和分析程序的错误日志和输出等信息。4) 记录程序信息：当发现异常时，将程序的输入数据和错误报告等进行记录。5) 重复测试：再次回到第一步，循环进行模糊测试，直到不能发现更多的问题。

模糊测试的主要挑战是如何生成高效、高质量的测试用例。AFL (American Fuzzy Loop)^[42]是一种经典的模糊测试工具，由 Google 的 Michał Zalewski 开发。AFL 采用基于变异的模糊测试策略，它以程序覆盖率为导向，通过高效的变异引擎生成大量的测试用例，并识别不再有效的测试用例，集中测试更有可能产生新路径的测试用例。覆盖率是指测试用例能够触发到的程序代码和程序整体代码之间的比例，可以用来衡量当前测试用例发现软件漏洞或者异常的能力，因为测试能够覆盖的代码量越大，发现软件异常的可能性就越大。AFL 采用样本突变方式以产生新的测试用例，其原生的突变操作包括字节翻转、比特翻转、编解码、插入变异、导致变异、删除变异、轮换变异、字典变异、随机字节，同时也支持用户自定义变异方法。AFL 对于被测试程序进行插桩，为每个插桩单元生成一个随机标识符以标识对应的程序位置，并进一步捕获程序的分支执行情况，以计算代码覆盖率。Lemieux 等人提出了一种自动识别稀有分支的方法和一种新颖的突变掩码创建算法，该算法使得生成的突变偏向于生成命中给定稀有分支的输入。Ganesh 等人^[43]利用动态污点追踪来识别原始种子输入文件中影响程序关键攻击点的区域，然后对这些区域进行突变处理，以生成新的测试输入文件。新生成的文件保留了原始种子输入文件的底层语法结构，这种方法对于测试具有高度复杂、高度结构化的输入文件格式的程序效果显著。Ma 等人^[44]提出了一种基于遗传算法和 BI-LST 神经网络的模糊测试样本生成框架，重点关注了测试样本生成时的路径深度等指标，使探索能够发现原本无法发现的漏洞。Li 等人^[45]针对传统模糊测试工具，面对字符串比较（比如程序的幻数）难以突破，导致大量测试用例无效的问题，通过轻量级静态分析工具的使用，以可接受的执行速度下降为代价，基于程序状态进行特定的测试用例突变，从而有效实现幻数的匹配。

第3章 基于循环符号执行的高代码覆盖率漏洞检测方法

本章分析了当前漏洞检测方法对于链上数据库机制模拟的局限性，并结合实际案例分析了这对漏洞检测覆盖率和准确率的影响，从而引出本章的基于循环符号执行的漏洞检测算法。本章详细介绍了该算法的内部流程，包括符号执行、最优覆盖率路径选择、交易生成、交易执行和更新链上数据库状态等步骤。同时，本章在数据集中进行了实验，实验结果体现了本章方法相对于目前的开源方法的优越性。最后，本章在实际链上智能合约数据集上进行了测试，实验结果表明了本章方法的实用性。

3.1 引言

3.1.1 问题分析

对于使用链上数据库机制的智能合约，现有漏洞检测方法^[3,29-32,37]在检测时的代码覆盖率低，并可能遗漏漏洞。以图3.1中的智能合约为例，该合约是一个筹款合约，其中蓝色变量代表链上数据库中的数据，该合约是对实际 EOSIO 筹款合约的简化表示。合约中的 `on_deposit` 函数表示合约收到转账后的处理，而 `refund` 函数对应于用户请求退款的处理。这两个函数中都通过 `eosio_assert` 函数对合约当前状态（筹款当前阶段 `state.time`，总筹款金额 `state.total_tokens`）进行了检查。`eosio_assert` 库函数检查第一个参数的布尔值是否为真，若为假则退出执行。`state` 变量是合约存储在链上数据库中的当前状态变量，需要对链上数据库进行查询和读取才能获得。图中合约存在缺少权限检查漏洞，因为其在 `refund` 函数中进行了敏感操作（修改链上数据库），但未进行权限检查（`require_auth(investor)`）。然而，对基于模糊测试的方法^[29-30]而言，只有在测试 `refund` 函数时参数满足特定的条件，才能够执行到漏洞代码，从而检测到漏洞。这要求测试 `refund` 函数的 `investor` 参数之前曾被用作测试 `on_deposit` 函数的 `investor` 参数，从而在链上数据库中存在对应的数据，否则它无法通过代码中第 25 行的检查。然而，由于模糊测试方法中交易生成的随机性，这样的要求很难满足，因此该类方法无法有效检测该缺少权限检查漏洞。而当前的静态方法^[3,31-32,37]则于未考虑读取链上数据库操作的副作用，因此模拟中读取的链上数据库数据（如 `state.time`、`state.start` 等）是错误的或者不确定的。如果用于放置数据的缓冲区曾被写入过，那么这些旧数据会被错误地当成从链上数据库得到的数据，并进行解析以进行条件判断，从而导致符号执行到错误路径。如果缓冲区从未被写入过，则当前的基于符号执行的方法会对其填充符号量，并对符号量进行解析和条件判断。由于存放读取数

```

1 class [[eosio::contract]] Crowsale : public contract {
2 public:
3     using contract::contract;
4     [[eosio::action]]
5     void on_deposit(account_name investor, eosio::asset quantity) {
6         // 注资
7         // 合约当前状态检查
8         eosio_assert(state.time >= state.start, "Crowdsale hasn't started");
9         eosio_assert(state.time <= state.finish, "Crowdsale finished");
10        auto it = this->deposits.find(investor);
11        int64_t tokens_to_give = EOS2TKN(quantity.amount);
12        // 修改筹款金额
13        state.total_eoses += quantity.amount;
14        state.total_tokens += tokens_to_give;
15    }
16
17    [[eosio::action]]
18    void refund(account_name investor) {
19        // 退款
20        // 合约当前状态检查
21        eosio_assert(state.time > state.finish, "Crowdsale hasn't finished");
22        eosio_assert(state.total_tokens < SOFT_CAP_TKN, "Softcap reached");
23        // 修复: require_auth(investor);
24        auto it = this->deposits.find(investor);
25        eosio_assert(it != this->deposits.end(), "Nothing to refund");
26        this->asset_eos.set_amount(it->eoses);
27        this->inline_transfer(this->_self, investor, this->asset_eos, "Refund");
28        this->deposits.modify(it, investor, [](auto& d) {
29            d.eoses = 0;
30        });
31    }
32
33    [[eosio::action]]
34    void settime(time_t time) {
35        state.time = time;
36    }
37};

```

图 3.1 存在缺少权限检查漏洞的智能合约示例

据的缓冲区大小并不确定，直接使用其中符号量数据进行解析和条件判断可能产生过多分支，从而导致执行超时或被提前剪枝。这些情况都可能导致形式化方法无法检测到该缺少权限检查漏洞。

3.1.2 解决思路

面对上述问题，本章提出并实现了基于循环符号执行的高代码覆盖率漏洞检测方法。如图3.2所示，本章方法会循环地进行符号执行和链上数据库状态更新，直到无法继续通过循环进一步扩展代码覆盖率后输出结果。这种循环使得本章方法能够尽可能地探索智能合约的各个分支，从而提高漏洞检测的准确性。本章方法中主要包含两个模块，即符号执行模块和基于覆盖率的交易生成和执行模块。在符号执行模块中，本工作充分考虑了链上数据库函数调用的副作用，

并基于实际链上数据库数据模拟数据库接口层,以便更准确地建模执行路径。同时,符号执行模块会基于模拟执行的路径信息(如约束集合和执行状态等)进行同步的漏洞检测。在基于代码覆盖率的交易生成和执行模块中,本工作根据符号执行得到的路径集合选择最优路径,并生成和执行相应交易,从而进行代码覆盖率最优的链上数据库状态迁移。当无法进一步选择路径以生成交易时,本章方法输出检测结果。此外,为缓解多轮符号执行可能导致的更严重的状态爆炸问题,本章进一步优化了符号执行的内存模型,并简化了反序列化动作过程。

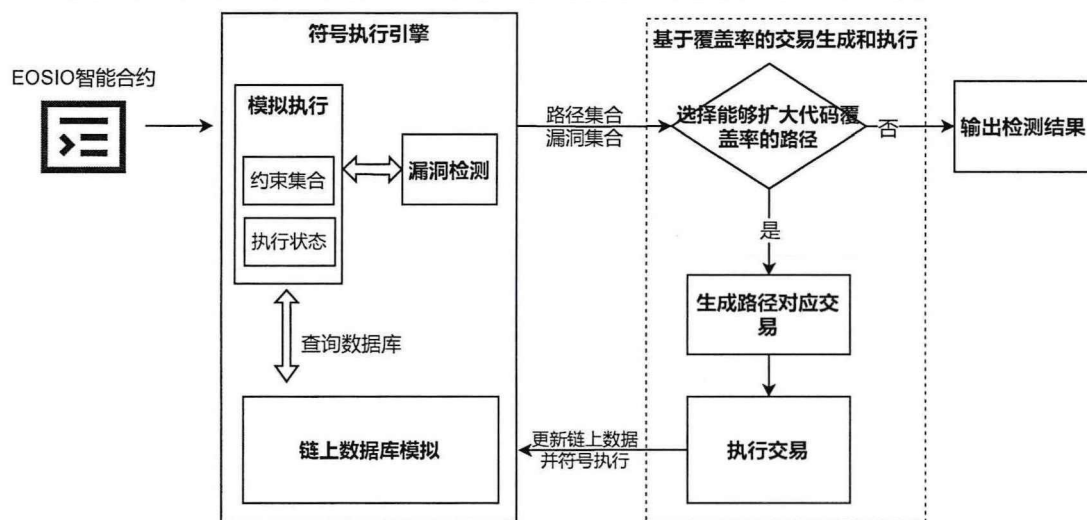


图 3.2 基于循环符号执行的高代码覆盖率漏洞检测方法框架图

综上所述,本章的主要工作包括:1)设计并实现了基于循环符号执行的漏洞检测算法。2)实现了细粒度建模链上数据库机制的高效符号执行算法。3)实现了以代码覆盖率为导向的路径选择和交易生成算法。

3.2 具体方法

3.2.1 相关定义

本节首先定义之后本章需要的相关概念,以便后续描述。

(1) 路径约束

定义符号量为一个具有取值范围和名称的变量,写作 (V, s) ,其中 V 是取值范围, s 是名称。表达式为常数值、符号量和运算符以符合运算规则的排列方式得到的组合。约束定义为两个表达式间的比较,用于表示路径的分支条件,记作 $e_1 \theta e_2, \theta \in \{=, \neq, <, >, \leq, \geq\}$ 。如果存在一组符号量的赋值使得约束集合成立,则称该约束集合为可满足的。路径约束定义为路径中所有分支指令对应约束的集合。

(2) 执行路径

定义智能合约中的函数名集合为 F 。定义执行路径为集合 $\mathcal{T} = \{(f, pc, inst, op, cons)\}$ ，集合元素五元组中， $f \in F$ 表示指令所在函数， pc 表示指令在函数中的相对偏移， $inst$ 是指令的具体内容， op 表示执行时指令具体的操作数， $cons$ 表示执行时指令对应的约束。记元组为 t ，定义 $t[i]$ 为元组中第 i 个元素。定义 $inst.name$ 、 $inst.type$ 和 $inst.i$ 分别为指令内容中的指令名称、指令类型和指令立即数。

(3) 调用路径

定义调用路径 $\mathcal{F}$ 为执行路径中函数调用指令的目标函数名数组，其中库函数调用的目标函数名为对应的库函数名。定义 $\mathcal{F}_i$ 表示数组中第 i 个被调用的函数名。

3.2.2 循环算法

为了更准确地检测使用了链上数据库机制的智能合约中的漏洞，本章结合形式化方法和模糊测试方法的优点，设计了一种基于循环符号执行的漏洞检测算法。本章设计的循环算法会根据实际执行的日志建立对链上数据库状态的模拟，并基于此进行符号执行，以尽可能使模拟执行产生路径和实际执行路径一致。因为符号执行算法使用符号量作为输入，所以能够考虑到输出参数的各种取值情况，从而避免了模糊测试随机化输入的问题。此外，本章的循环算法根据符号执行得到的路径集合，优先选择最能提高代码覆盖率的路径，生成并执行交易，从而更新链上数据库状态。这使得本章的循环算法相对于模糊测试方法能够考虑更多链上数据库状态，从而提高漏洞检测的准确率。

本章的循环算法具体如算法3.1所示。输入是智能合约 S 和最大循环次数 L 。最大循环次数 L 用于设置算法中更新链上数据库状态的最大次数。算法中首先初始化了一些变量，其中 $vuls$ 表示检测到的漏洞集合， $\mathbb{D}$ 表示链上数据库的状态， pcs 是实际执行中已经被覆盖的指令地址集合（第1到3行）。然后算法进入了具体的循环过程（第4到16行），循环以达到最大循环次数或者无法选出能扩大代码覆盖率的路径为止（第8到10行）。在每次循环中，算法首先根据当前链上数据库状态 $\mathbb{D}$ 对智能合约 S 进行符号执行，从而得到路径集合 $traces$ 和新的漏洞集合 $vuls_n$ （第5行）。然后，算法更新已检测到的漏洞集合并从路径集合 $traces$ 中选择最能扩大代码覆盖率的路径（第6, 7行）。如果能够选出最能扩大代码覆盖率的路径 $\mathcal{T}$ ，则算法更新被覆盖的指令地址集合并进行链上数据库状态的更新（第11到14行）。更新链上数据库状态的具体步骤包括根据路径 $\mathcal{T}$ 生成触发交易 tx （第12行），执行交易生成日志 log （第13行），并使用日志 log 更新链上数据库的状态 $\mathbb{D}$ （第14行）。

算法 3.1 基于循环符号执行的漏洞检测算法**Input:** 智能合约 S , 最大循环次数 L **Output:** 漏洞集合 vul

```

1  $vuls \leftarrow \Phi$ ;
2  $\mathbb{D} \leftarrow \text{initialDatabase}()$ ;
3  $pcs \leftarrow \Phi$ ;
4 for  $i \leftarrow 0$  to  $L$  do
5    $(traces, vuls_n) \leftarrow \text{symbolicExecute}(S, \mathbb{D})$ ;
6    $vuls \leftarrow vuls \cup vuls_n$ ;
7    $\mathcal{T} \leftarrow \text{chooseBestTraceByCoverage}(traces, pcs)$ ;
8   if  $\mathcal{T} = \text{None}$  then
9     return  $vuls$ ;
10  else
11     $pcs \leftarrow pcs \cup \text{getPcs}(\mathcal{T})$ ;
12     $tx \leftarrow \text{generateTransaction}(\mathcal{T})$ ;
13     $log \leftarrow \text{executeTransaction}(tx)$ ;
14     $\mathbb{D} \leftarrow \text{updateDatabase}(\mathbb{D}, log)$ ;
15  end
16 end
17 return  $vuls$ ;

```

图3.3展示了算法3.1的一个运行示例。首先, 算法在初始的链上数据库状态 ($\mathbb{D} = b_0$) 下进行符号执行, 符号执行产生了三条路径, 其中覆盖率最优的路径被选择用以生成触发交易 t_1 。初始链上数据库状态 b_0 即是链上数据库中不存在任何数据记录的状态, 算法在符号执行的过程中进行漏洞的检测。然后, t_1 在本地链上被执行, 从而产生新的链上数据库状态 b_1 。在新的链上数据库状态 b_1 下, 继续进行符号执行, 得到了四条路径, 其中覆盖率最优的路径被选择并生成交易 t_2 。执行交易 t_2 将使链上数据库状态进一步变为 b_2 。这样的循环将一直持续到无法产生能更新链上数据库状态的交易, 然后算法将输出检测到的漏洞集合。算法输出的漏洞集合中包含漏洞的触发交易序列, 它来源于实际执行交易序列。假设漏洞在第二次符号执行中被发现, 那么交易序列 t_1, t'_2 即可以触发该漏洞, 其中 t'_2 为第二次符号执行时漏洞所在的路径对应的交易。

3.2.3 符号执行算法

现有的符号执行方法缺少对链上数据库机制的建模, 并且模拟执行中并不保留搜索过的路径, 这无法满足本研究中循环算法的需求。为此, 本章实现了细粒度建模链上数据库机制的符号执行算法。如算法3.1中第5行所示, 循环算法所需的符号执行算法输入智能合约 S 和当前链上数据库状态 $\mathbb{D}$, 输出符号执行得到的路径集合 $traces$ 和发现的漏洞集合 $vuls_n$ 。符号执行得到的路径集合会被 $traces$ 会被循环算法用于后续的路径选择和交易生成, 从而更新链上数据库状态。本章的符号执行算法中基于链上数据库状态 $\mathbb{D}$ 和链上数据库的源码实现了链上数据库模拟, 以减少模拟执行的路径与实际执行路径的偏差。此外, 本章的

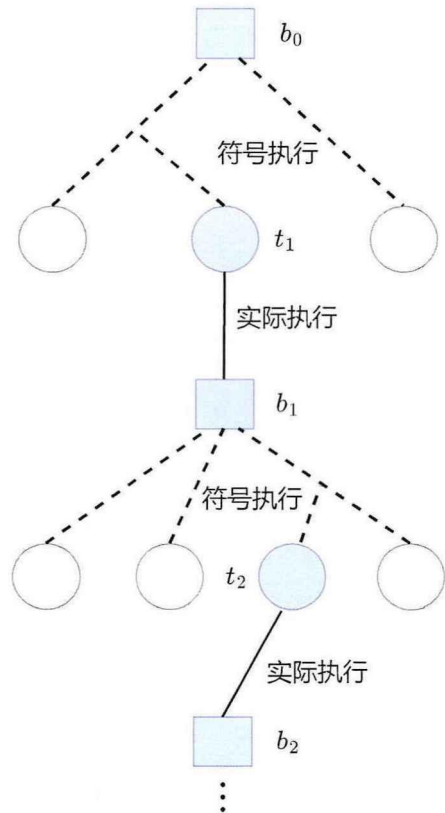


图 3.3 算法3.1的运行示例

符号执行算法通过动态维护当前路径集合和已搜索到的路径集合，并对影响控制流指令进行特殊处理，实现了对符号执行已搜索路径的记录。

算法 3.2 EOSIO 平台智能合约符号执行算法

Input: 智能合约 S ，链上数据库状态 $\mathbb{D}$

Output: 路径集合 $traces$ ，漏洞集合 $vuls$

```
1  $vuls \leftarrow \Phi$ ;  
2  $traces \leftarrow \Phi$ ;  
3  $(memory_0, table_0, global_0, expr) \leftarrow preProcess(S)$ ;  
4  $\mathcal{E} \leftarrow locateEntry(expr)$ ;  
5 for  $e \in \mathcal{E}$  do  
6    $(memory, table, global, expr) \leftarrow (memory_0, table_0, global_0, expr)$ ;  
7    $(stack, local, f) \leftarrow getExecuteEnvironment(e)$ ;  
8    $C \leftarrow \Phi$ ;  
9    $state \leftarrow (stack, local, memory, global)$ ;  
10   $traces', vuls' \leftarrow execute(f, pc, state, \mathbb{D}, C)$ ;  
11   $traces \leftarrow traces \cup traces'$ ;  
12   $vuls \leftarrow vuls \cup vuls'$ ;  
13 end  
14 return  $(traces, vuls)$ ;
```

本章符号执行算法的具体实现如算法3.2所示。首先，算法将已探索的路径集合和漏洞集合设置为空（第 1，2 行）。其次，算法对智能合约进行预处理，从而获取内存、表格等字段的初始化内容和指令集合 $expr$ （第 3 行）。然后，算法获取智能合约的入口点集合 $\mathcal{E}$ ，并对每个入口点 e 设置运行环境和初始约束 C 以

模拟执行（第4到12行）。最后，算法返回模拟执行中探索到的路径集合和漏洞集合（第13行）。

（1）模拟执行算法

算法3.3 模拟执行算法

Input: 函数标识 f , 模拟起始地址 pc , 执行状态 $state$, 链上数据库状态 $\mathbb{D}$, 起始约束集合 C

Output: 已探索路径集合 $traces$, 漏洞集合 $vuls$

```

1 ( $vuls, traces, \mathcal{P}$ )  $\leftarrow \{\Phi, \Phi, \{\Phi\}\}$ ;
2 while  $pc \neq -1$  do
3    $inst \leftarrow getInstruction(expr, f, pc)$ ;  $vuls \leftarrow vuls \cup vulnerabilityScanner(traces, state)$ ;
4   if  $isBranchInstruction(inst)$  then
5     for  $b \in getBranches(inst)$  do
6        $state' \leftarrow state$ ;  $(c, state') \leftarrow getBranchConditions(b, state')$ ;
7        $\mathcal{P} \leftarrow \mathcal{P} \times \{f, pc, inst, \Phi, c\}$   $C' \leftarrow C \cup \{c\}$ ;
8       if  $solve(C' = satisfiable)$  then
9          $pc' \leftarrow getBranchTarget(b)$ ;
10         $(traces'', vuls'') \leftarrow execute(f, state', \mathbb{D}, C')$ ;
11         $vuls \leftarrow vuls \cup vuls''$ ;  $traces \leftarrow traces \cup \mathcal{P} \times traces''$ ;
12      end
13    end
14    break
15  else if  $isCallInstruction(inst)$  then
16     $\mathcal{P} \leftarrow \mathcal{P} \times \{f, pc, inst, \Phi, None\}$   $state' \leftarrow state$ ;  $f_n \leftarrow locateFunc(inst)$ ;
17     $(traces'', vuls'') \leftarrow execute(f_n, state', \mathbb{D}, C)$ ;
18     $\mathcal{P} \leftarrow \mathcal{P} \times traces''$ ;
19     $vuls \leftarrow vuls \cup vuls''$ ;  $pc \leftarrow pc + 1$ ;
20  else if  $isEnd(inst)$  then
21     $\mathcal{P} \leftarrow \mathcal{P} \times \{f, pc, inst, \Phi, None\}$   $traces \leftarrow traces \cup \mathcal{P}$ ;
22    break
23  else
24     $state \leftarrow execInstruction(inst, state)$ ;
25     $pc \leftarrow pc + 1$ ;
26 end
27 return  $vuls, traces$ ;
```

符号执行中的模拟执行模块负责处理指令流并维持执行状态，本章符号执行算法主要关注路径信息。为了记录符号执行产生的路径，在进行模拟执行的过程中（算法3.3），需要同时维护当前路径集合 $\mathcal{P}$ 和已搜索到的路径集合 $traces$ 。 $\mathcal{P}$ 包含正在被模拟执行还未终止的路径，而 $traces$ 包含当前函数中已经搜索结束的路径。当前路径集合 $\mathcal{P}$ 被初始化为 $\{\Phi\}$ ，在模拟执行单条指令时，需要将指令的信息加入 $\mathcal{P}$ 中的每条当前路径中（第4行）。影响控制流的指令需要进行特殊处理，包括分支类指令、调用类指令和返回类指令。对这三类指令的处理如下：

1) 分支类指令：分支类指令包括 `br_if`、`br_table` 等指令，都是根据栈顶值选择跳转地址。模拟执行分支类指令时，需要依次模拟所有可能的分支，并在所有分支模拟完成后直接返回已搜索到的路径集合 $traces$ （第5行到15行）。模拟单个可能的分支时，首先在约束集合中添加新分支的条件，如果约束集合

仍然是可满足的, 则递归地模拟执行分支后续的指令, 并更新路径集合 $traces$ 。将递归模拟执行得到的路径集合记作 $traces^n$, 对路径集合的更新为 $traces \leftarrow traces \cup P \times traces^n$, 这里的 $P \times traces^n$ 表示分别将 $traces^n$ 中每条路径 $\mathcal{T}^n$ 添加到当前路径集合 P 中每条路径 $\mathcal{T}$ 后得到的路径集合。

2) 调用类指令: 调用类指令主要包括 `call` 和 `call_indirect` 指令, 用于进行函数调用。模拟执行调用类指令时, 需要定位被调用函数, 并递归模拟执行被调用函数 (第 16 行到 20 行)。记递归模拟执行得到的路径集合为 $traces^n$, 模拟完后更新当前路径为 $P \leftarrow P \times traces^n$ 。

3) 返回类指令: 返回类指令主要包括 `return` 指令, 用于结束当前函数执行并返回结果 (第 21 行到 23 行)。模拟执行返回类指令时, 将当前路径集合 P 加入 $traces$ 并返回 $traces$ 。

为了缓解状态爆炸问题, 在模拟执行过程中引入了两个隐藏参数, 分别表示分支深度上限和调用深度上限。在模拟执行的过程中, 算法会记录当前路径中分支类指令的调用次数和函数调用的深度, 并在达到设定的上限后停止进一步的模拟执行。一旦达到这两种上限, 对路径集合的处理类似于返回类指令, 即将当前路径集合 P 加入 $traces$ 并返回 $traces$ 。然而, 与返回类指令不同的是, 达到上限后需要回退到上一处分支类指令, 并执行另一分支, 以避免对当前分支的错误探索。

由于库函数的特殊性 (库函数的函数体不在智能合约中), 在模拟执行中处理库函数调用指令时, 本章会根据库函数的种类采取不同的处理策略:

1) 区块链信息相关库函数: 该类函数返回区块链相关的信息, 比如 `tapos_block_num` 会返回当前交易的区块数。该类函数的结果常被智能合约用作随机数生成, 并且不会引起副作用。模拟执行该类库函数时, 返回一个相应类型的符号量以维护调用栈的平衡。

2) 权限检查相关库函数: 该类函数检查对应账户是否是已授权账户, 包括 `require_auth` 和 `has_auth` 等。该类函数结合合理的权限分配, 保证了智能合约敏感操作的安全性。模拟执行该类库函数时, 同样是返回一个对应类型的符号量以平衡调用栈。

3) 合约调用相关库函数: 该类函数调用其他合约或当前合约的动作, 包括 `send_inline` 和 `send_context_free_inline` 等。该类函数没有返回值, 并会终止当前合约的执行。对该类函数的模拟等同于对返回类指令的处理。

4) 内存相关库函数: 该类函数进行内存的批量操作, 如 `memcpy` 进行内存块之间的复制, 并返回成功复制的字节数。该类函数调用存在副作用, 即为对内存的修改。当模拟执行该类库函数时, 需要根据库函数语义进行相应的内存修改, 并根据修改结果返回值。

- 5) 控制相关库函数: 该类函数进行条件的检验, 并根据结果判断是否终止控制流, 包括 `eosio_assert` 等。这类库函数不需要返回值, 在模拟时仅需根据语义进行条件判断和分支, 类似于对分支类指令的处理。
- 6) 链上数据库相关库函数: 该类函数对链上数据库进行增删改查等操作, 如 `db_find_i64` 根据主键获取对应数据记录的指针。在模拟执行时, 需要根据数据库操作的语义和链上数据库状态进行内存的修改, 并返回相应的结果。该类库函数对应的链上数据库机制是本工作重点关注的对象, 接下来将对其进行详细介绍。

(2) 链上数据库模拟

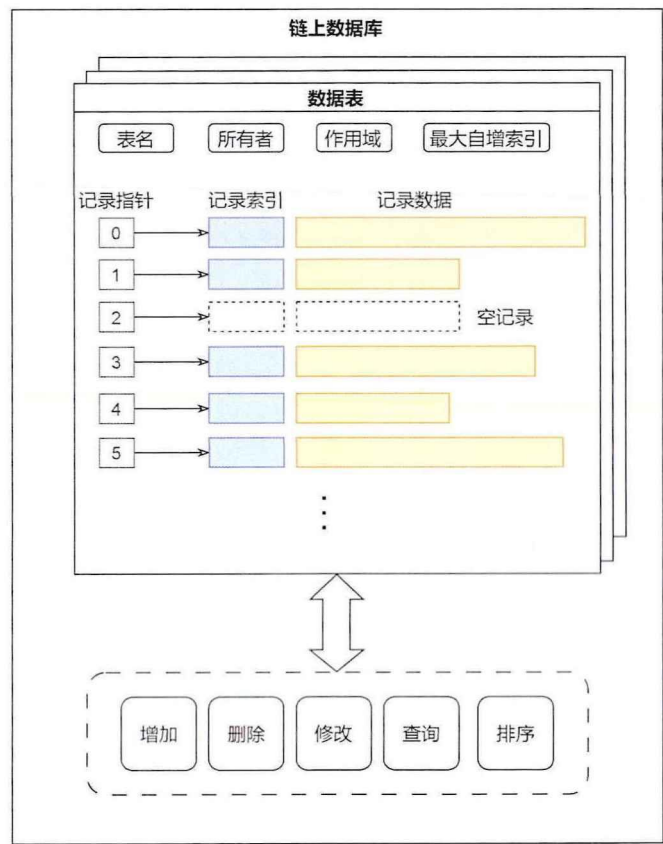


图 3.4 EOSIO 平台智能合约链上数据库示意图

为实现对链上数据库机制的精确建模, 在本工作的符号执行引擎中, 基于实际的链上数据库状态 ID, 实现了对链上数据库接口层的模拟。如图 3.4 所示, 在 EOSIO 平台智能合约中, 链上数据库的每条记录可被视为键值对, 其中键是自增索引, 初始为 0。当向链上数据库插入记录的时候, 当前自增索引作为新纪录的索引, 并将自增索引加一。因为每个自增索引唯一对应于一条记录, 所有它也可被视为是记录的指针, 当记录被删除时, 对应的自增索引仍然保留。每条记录的值中包含具体的数据和索引, 数据和索引都是用户自定义的结构, 在数据库中仅存储其序列化后的数据。索引是用户自定义的数据唯一标识, 用户可用它进行数据库数据的筛选和排序等操作。链上数据库中可以包含多个具有独立的自增

索引的数据表, 这些数据表由表名、所有者地址等字段唯一确定。本研究基于 $\mathbb{D}$ 模拟了链上数据库的内部实现, 包括数据库内部存储结构和查询修改等操作, 从而提供了对链上数据库接口层的模拟。

基于上述对链上数据库接口层的模拟, 本研究在符号执行引擎中实现了对智能合约中链上数据库机制的精确建模。根据用户自定义索引的不同长度, 链上数据库类库函数存在不同的变体, 例如, `db_get_i64` 库函数要求数据表的索引是 64 位整数。为了便于说明, 本节以索引为 64 位整数的情况为例。模拟执行到链上数据库库函数的调用, 如果库函数参数都是常量, 则根据库函数语义查询上述接口层模拟, 并修改内存和返回结果。对于 `db_find_i64`、`db_lowerbound_i64` 和 `db_upperbound_i64` 等库函数, 它们根据传入的索引来获取记录的指针, 其中后两者分别寻找大于等于 (大于) 传入索引的最小索引的指针, 而 `db_find_i64` 则寻找传入索引对应的指针。对于这三种库函数, 本研究根据其语义返回对应的指针, 并按照实际数据库接口层的实现, 在无法找到对应记录时返回表的结尾指针 (即当前自增索引)。对于 `db_get_i64`、`db_next_i64` 和 `db_previous_i64` 等库函数, 它们都是根据传入的指针来获取记录数据, 但后两者分别获取指针之后 (之前) 的数据的索引, 而 `db_get_i64` 则获取指针对应记录的具体数据。对于这三种库函数, 本研究分别根据其语义和实际数据修改其参数对应的缓冲区, 并在异常情况下返回符合接口层实现的值。对链上数据库进行更新、增加和删除操作的库函数, 如 `db_update_i64`、`db_store_i64` 和 `db_remove_i64` 等, 鉴于实际合约中未发现修改后立即进行查询的操作, 在模拟执行中未对这三种函数进行特殊处理。当库函数参数为符号量时, 对链上数据库类库函数的处理需要考虑符号量的各种取值情况。以 `db_find_i64` 为例, 如果传入的索引是符号量, 则类似于处理分支类指令, 本工作会修改路径集合 $\mathcal{P}$ 和约束集合 $\mathcal{C}$, 以表示产生两类分支。一类分支表示当前索引在数据库中, 对于该类分支, 约束集合 $\mathcal{C}$ 中需添加约束 $index = k$, 其中 $index$ 是传入的索引参数, k 是数据库中存在的索引值。另一类分支表示当前索引值不在数据库中, 需要修改约束集合为 $\mathcal{C} \leftarrow \mathcal{C} \cup \{index \neq k | k \in \mathbb{D}_k\}$, 其中 $\mathbb{D}_k$ 为链上数据库中的索引集合。对于产生的两类分支, 需要分别求解新的约束集合是否可满足, 如果可满足则根据语义修改内存并返回结果, 否则尝试其他分支。

在循环算法中, 交易的实际执行会改变链上数据库状态 $\mathbb{D}$, 符号执行引擎需要同步修改对链上数据库接口层的模拟。具体而言, 本研究通过分析执行日志中对于 `db_update_i64`、`db_store_i64` 和 `db_remove_i64` 等库函数的调用来实现这一目标。这些库函数分别负责对链上数据库进行更新、增加和删除操作。链上数据库模拟中包含基于链上数据库状态模拟的键值对数据存储, 因此对于这三类函数, 需要相应地更新链上数据库模拟中的键值对数据。对于 `db_update_i64` 库函

数, 其传入指针和新数据, 并使用新数据替代链上数据库中原有指针处的数据, 因此需更新链上数据库模拟中对应的键值对记录。对于 `db_store_i64` 库函数, 其传入索引和新数据, 因此需向链上数据库模拟中添加一个记录, 并增加自增索引的值。对于 `db_remove_i64` 库函数, 其根据传入的指针删除对应数据, 需删除链上数据库模拟中相应的记录, 并保持自增索引的值不变。

3.2.4 符号执行算法的优化

为了缓解多轮符号执行造成的更严重的状态爆炸问题, 本章进一步优化了符号执行算法, 以提高模拟执行的效率。本章的优化策略主要包括对请求数据反序列化流程的抽象和对内存模型的优化两种, 将在以下两节进行详细介绍。

(1) 请求数据反序列化抽象

请求数据反序列化是执行响应函数前不可缺少的步骤。如2.2节所述, EOSIO 平台智能合约具有统一的入口点 `apply` 函数。该函数根据其三个参数(接收请求账户、被调用的账户和请求的接口名称)来决定是否响应触发合约的请求, 并将请求分发给对应函数进行处理。在执行响应函数之前, 智能合约需要获取请求数据并对其进行反序列化, 从而使用反序列化的结果填充响应函数的参数。反序列化的过程如图3.5所示。图中的 `execute_action` 函数接收 `contract` 对象 `obj` (存储智能合约代码和数据) 和响应函数指针 `func`。 `execute_action` 函数中读取了请求数据(第10行)、对请求数据进行了反序列化(第13行)并最终调用了响应函数(第23行)。

在当前基于符号执行的漏洞检测方法^[3,31]中, 并未对该反序列化步骤进行特殊处理, 这引发了两个问题:

1) 首先, 因为获取请求数据是通过调用库函数(`read_action_data`)实现的, 而其他方法^[3,31]中并未考虑此类库函数执行时产生的副作用(写入内存)。因此, 实际获取到的请求数据可能是错误的或是不确定的(即符号量)。于是, 这些符号执行方法在模拟执行反序列化流程时, 类似于对链上数据库类函数的处理, 可能会耗费过多时间甚至产生错误的响应函数参数。

2) 其次, 该反序列化流程模糊了请求数据和响应函数参数之间的关系。这使得其他方法即使成功发现了漏洞, 也难以生成漏洞的触发请求以进行验证。因为它们不能确定漏洞所在的路径的响应函数对应的接口名和接口参数取值。

为解决上述问题, 本研究通过对智能合约实际执行日志的分析, 确定了各接口对应的响应函数, 并以其作为符号执行的入口。具体而言, 本研究首先使用 WASAI^[30]对待分析的合约进行插桩, 以便在执行时输出执行路径上的指令及指令操作数。随后, 根据 EOSIO 平台智能合约的 ABI 接口文件, 本研究生成多个交易, 每个交易对应一个智能合约接口, 并使用符合调用规范的随机值作为参

```

1  template<typename T, typename Q, typename... Args>
2  bool execute_action( T* obj, void (Q::*func)(Args...) ) {
3      size_t size = action_data_size(); // 获取请求数据大小
4
5      constexpr size_t max_stack_buffer_size = 512;
6      void* buffer = nullptr;
7      if( size > 0 ) {
8          buffer = max_stack_buffer_size < size ? malloc(size) : alloca(
9              size); // 为请求数据分配空间
10         read_action_data( buffer, size ); // 读取请求数据
11     }
12     auto args = unpack<std::tuple<std::decay_t<Args>...>>( (char*)
13         buffer, size ); // 反序列化请求数据为参数
14
15     if ( max_stack_buffer_size < size ) {
16         free(buffer);
17     }
18     auto f2 = [&]( auto... a ){
19         (obj->*func)( a... );
20     };
21
22     boost::mpl::tuple_apply( f2, args ); // 调用响应函数
23     return true;
24 }

```

图 3.5 请求数据反序列化代码

数。然后，本研究在本地链上执行这些交易，获取执行日志，并进一步分析日志以确定响应函数的唯一标识。最后，符号执行时可以直接以响应函数为入口点开始执行，并将符号量作为响应函数的参数。因为符号执行引擎以响应函数入口点为起点，它不需要模拟复杂的请求数据反序列化过程。这减少了符号执行过程中的时间消耗，从而提高了漏洞检测的效率。此外，当符号执行完成后，本研究可以根据响应函数对应的接口名和路径约束，生成触发路径需要的交易请求数据。

为了从日志中获取响应函数标识，设计了基于特征的响应函数定位算法，如算法3.4所示。文献^[30]中提到，执行路径中第一个 `call_indirect` 指令具体调用的函数就是响应函数。然而，在最新的编译器编译的智能合约中，这一特征并不存在。为此，本工作基于对智能合约的逆向分析经验总结了新的特征，并在算法3.4中采用新特征定位了响应函数。算法的输入是执行日志中的程序路径 $\mathcal{T}$ ，输出是响应函数的标识 f_{on} 。首先，算法在程序路径中定位第一个需要两个 32 位整数作参数的 `call` 指令，从而定位到中转函数 f_m （第 7 到 9 行）。中转函数中进行了请求数据的反序列化和 `contract` 结构体的初始化，`contract` 结构体存储了智能合约主要的数据结构和处理逻辑，响应函数都是 `contract` 结构体的成员函数。然后，算法定位到中转函数中初始化 `contract` 结构体的代码（第 13 到 17 行）。具体来说，是定位到初始化 `contract` 结构体第一个字段的位置，从而获取 `contract` 结构体的首地址 A 。最后，定位 `contract` 结构体完成初始化后的第一个成员函数调用（第 18 到 21 行）。成员函数调用的第一个参数是结构体首地址，调用指令的立即

算法 3.4 响应函数定位算法**Input:** 程序路径 $\mathcal{T}$ **Output:** 响应函数标识 f_{on}

```

1  $n \leftarrow$  The length of  $\mathcal{T}$ ;
2  $f_m \leftarrow \text{None}$ ;
3  $A \leftarrow \text{None}$ ;
4 for  $i \leftarrow 0$  to  $n$  do
5    $(f, pc, inst, op) \leftarrow \mathcal{T}[i]$ ;
6   if  $f_m = \text{None}$  then
7     if  $inst.name = \text{call}$  and  $inst.type = (i32, i32)$  then
8        $f_m \leftarrow inst.i$ ;
9     end
10  else
11    if  $f = f_m$  then
12      if  $A = \text{None}$  and  $inst.name = \text{local.get}$  and  $inst.i = 0$  then
13         $(f', pc', inst', op') \leftarrow \mathcal{T}[i+1]$ ;
14        if  $inst' = \text{i64.store}$  then
15           $A \leftarrow inst'.i + op[1]$ ;
16        end
17      end
18      if  $A \neq \text{None}$  and  $inst.name = \text{call}$  and  $op[0] = A$  then
19         $f_{on} \leftarrow inst.i$ ;
20        return  $f_{on}$ ;
21      end
22    end
23  end
24 end

```

数即是响应函数的标识 f_{on} 。

(2) 内存模型优化

内存模型即符号执行对内存空间的模拟，在模拟执行中具有不可或缺的地位。EOSIO 平台智能合约使用一块连续的内存空间，以一字节为单位，使用 32 位无符号整数作为地址。相对来说，内存空间是智能合约能使用的最大的存储空间。智能合约的 WASM 指令集中包含 23 种内存相关指令，可用于读取和写入内存，这些指令在实际合约中被大量应用。当符号执行引擎对内存空间进行模拟，需要同时考虑符号量和常量作为地址的情况。由于内存空间的范围大、使用频繁并且情况复杂，对内存空间的建模成为了缓解状态爆炸问题的关键。

目前的各种符号执行引擎^[3,30-31]采用了各异的内存模型，它们之间各有优劣。EOSAFE^[3]中考虑到内存的稀疏性，使用三元组 (l, h, D) 表示内存区域，其中 l 和 h 分别表示内存块的起始和终止地址， D 表示内存块的具体值。EOSAFE 在模拟执行内存相关指令时，同步进行内存区域的合并，以降低内存复杂性并缓解状态爆炸问题。而 WASAI^[30]则使用二元组 (a, s) 表示内存模型，其中 a 表示地址， s 表示单字节的内存变量值。WASAI 使用 z3 的数组实现内存模型，并通过 z3 的 Store 和 Select API 进行内存的更改和读取。相比之下，WASAI 的内存模

型能够更快地获取内存中的值,因为 EOSAFE 的内存模型在每次内存区域合并时需要搜索所有内存区域,这会耗费大量时间。WANA^[31]也使用二元组 (a, s) 表示内存模型,但要求内存地址 a 只可能是常量。当 WANA 在模拟执行时遇到使用符号量进行内存取值的情况时,它将这些符号量绑定到随机选择的内存地址。虽然 WANA 的内存模型显著提高了模拟执行时约束求解的速度,但在模拟执行时可能会出现将符号量地址绑定到其他已使用内存区域的情况,从而导致执行错误。

为了提高符号执行效率并保持正确性,本研究综合了 WASAI 和 WANA 的内存模型,实现了二元的内存模型。本研究的内存模型分为两部分,分别使用二元组表示 $M = \{(a, s)\}$ 和 $M' = \{(a', s')\}$,其中 M 使用常量地址, M' 使用符号量地址。当符号执行引擎使用常量地址进行访存时,本研究通过 M 快速返回结果;当引擎使用符号量 (a') 作为地址时,本研究判断符号量是否在 M' 中,若在,则直接返回结果 (s'),否则添加记录 ($M' = M' \cup (a', s'')$) 并返回新符号量 (s'')。从而,本研究的内存模型能够兼具 WANA 内存模型的高效和 WASAI 内存模型的正确性。

为了进一步提高符号执行的速度,本研究优化了符号内存 M' 的实现。WASAI 采用 $z3$ 的数组实现内存模型主要是考虑到内存地址覆盖的问题。假设 a 和 b 是两个符号量,当模拟执行时先在地址 a 写入 $0x00$,然后在地址 b 写入 $0x11$ 。如果符号量 a 等于符号量 b ,则实际地址 a 处的值为 $0x11$,否则地址 a 处为 $0x00$,这就是内存地址覆盖的问题。使用 $z3$ 的数组实现内存模型可以解决内存地址覆盖的问题,但这会导致求解速度的下降,因为 $z3$ 数组取址时需要考虑地址等于任何符号量的情况。然而,在本研究的符号执行算法中这是不必要的。由于直接从响应函数开始模拟执行,本研究符号执行算法中的符号量仅有两种来源:来自响应函数的参数或来自库函数调用结果。由于库函数的返回结果不会作为内存地址,符号内存地址只能是响应函数的参数的表达式,而响应函数的参数是结构体的首地址,称其初始符号地址。因此,不同的初始符号地址应对应不同的结构体,并且它们之间不应该相互访问。假设两个参数对应的初始符号地址分别为 a 和 b 。 a 不可能等于 b ,因为 C++ 编译时会为每个参数结构体生成地址不同的实参。 $a+d$ 也不应该等于 b ,其中 d 是一个常量,因为这样超过结构体范围的访问是危险且应该避免的。通过对符号量的化简(将 $a+3-2$ 化简为 $a+1$),本研究即可区分初始符号地址不同的符号地址和初始符号地址相同而偏移不同的符号量地址,从而避免内存地址覆盖问题。因此,本研究的符号内存 M' 仅使用键值对实现,其中键为化简后的符号地址。

3.2.5 漏洞检测

本研究在符号执行过程中同步进行漏洞检测（算法3.3第3行）。具体而言，每次对指令进行模拟执行后，算法基于智能合约的当前执行信息进行漏洞检测，并更新漏洞集合。漏洞检测模块使用的执行信息主要包括约束集合 C 、调用路径 F 和执行路径 P 。由于链上数据库状态仅影响对响应函数中路径的探索，本章不考虑出现在响应函数之外的漏洞，例如仅可能在 `apply` 函数中发生的假收据漏洞和假 EOS 漏洞。针对 EOSIO 平台智能合约中的其他四种主流漏洞，本章在模拟执行的过程中进行漏洞的检测，具体检测方法如下：

1) 缺少权限检查漏洞：如果在漏洞检测时，当前指令进行了敏感操作，则进一步判断当前路径之前是否调用过权限检查相关库函数，如果没有，则存在权限检查漏洞。敏感操作包括对链上数据库的更新和删除，以及调用合约调用相关库函数等。存在漏洞的路径条件为

$$Authenticate \cup F \neq \emptyset \wedge Sensitive \cup F \neq \emptyset \quad (3.1)$$

其中 *Authenticate* 表示权限检查相关库函数名集合，包括 `require_auth`、`has_auth`、`require_auth2` 等。而 *Sensitive* 表示链上数据库更新和合约调用相关库函数名集合，包括 `db_update_*`、`db_remove_*`、`db_store_*`、`send_inline`、`send_defered` 等。

2) 回滚漏洞：回滚漏洞只发生在赌博类智能合约中，这类合约的一个特征是使用区块链状态作为求余指令的操作数^[3]。当模拟执行到求余指令 (`rem`) 时，漏洞检测模块判断操作数对应的表达式中是否存在由区块链信息相关库函数返回的符号量，并将判断结果记录在路径状态中。如果在漏洞检测时，路径状态中同时存在对内联动作类库函数的调用和符合条件的求余指令，则智能合约存在回滚漏洞。存在漏洞的路径条件为

$$Send \cup F \neq \emptyset \wedge (\exists i(i \in P \wedge i[3] = rem \wedge i[4] \cup B \neq \emptyset)) \quad (3.2)$$

其中 *Send* 表示内联动作类库函数名集合，包括 `send_inline`、`send_context_free_inline` 等，*B* 表示区块链信息相关函数返回的符号量集合。

3) 链上数据依赖漏洞：如果路径中同时存在对区块链信息相关库函数的调用和敏感操作，则存在链上数据依赖漏洞，对应的路径条件为

$$Sensitive \cup F \neq \emptyset \wedge Block \cup F \neq \emptyset \quad (3.3)$$

其中，*Block* 表示区块链信息相关库函数名集合，包括 `tapos_block_num`、`tapos_block_prefix` 等。

4) 整数溢出漏洞: 当模拟执行定长整数的加 (type.add), 减 (type.sub) 和乘 (type.mul) 指令时, 如果结果表达式是符号量, 则本章向路径约束集合中添加一个约束并进行约束求解, 并在求解完成后弹出该约束。该约束为 $r = bound$, 其中 r 表示指令运算结果, $bound$ 表示定长整数表达边界, 如果该约束可以满足, 则存在整数溢出漏洞。存在漏洞的路径需满足的条件为

$$\exists i(i \in \mathcal{P} \wedge i[3] \in \mathcal{Calc} \wedge \text{Sat}(C \cup r = bound)) \quad (3.4)$$

其中 $\mathcal{Calc}$ 表示定长整数的加减乘指令名集合, Sat 谓词表示对应约束集合是否可满足。

3.2.6 路径选择和交易生成算法

为了探索 EOSIO 平台智能合约在不同链上数据库状态下的执行路径, 提出了一种基于覆盖率的路径选择算法。该算法将在符号执行得到的路径集合中选择合适的路径, 以生成相应的交易从而更新链上数据库状态。在路径选择的过程中, 为了使生成的交易能实现链上数据库状态的转移, 首先需要筛选出路径集合中包含链上数据库状态更新操作的路径。这样的操作包括对链上数据库增加和修改等库函数的调用。为了分析更多的链上数据库状态以提高代码覆盖率, 本算法优先考虑了路径集合中对代码覆盖率提升更多的路径。这些路径和之前已选路径的差异更为显著, 从而路径对应的交易更可能产生新的链上数据库状态。

本章所提出的路径选择算法如算法3.5所示。对于路径集合中的每条路径 t , 该算法评估路径中是否包含对链上数据库状态进行修改的库函数调用, 如 `db_store_*`、`db_update_*` 和 `db_remove_*` 等, 并检查路径的约束集合是否可满足 (第4行)。如果满足这两个条件, 算法将路径添加到数组 $traces'$ 中, 并计算该路径增加的实际执行代码覆盖率 $\text{getLength}(pcs' - pcs)$ (第5到8行)。然后, 算法根据路径增加的代码覆盖率对数组 $traces'$ 进行排序 (第11行), 并返回排序结果中覆盖率增加最多的路径 $\mathcal{T}$ 。

为了更新链上数据库的状态, 本研究根据选择出的路径 $\mathcal{T}$ 生成用以触发该路径的请求交易。请求交易中包括请求名和请求参数, 算法3.4得到了请求名和响应函数标识的映射, 因此只需获取路径 $\mathcal{T}$ 的初始函数即可得到请求名。如前2.2所述, 智能合约的接口规范记录于 ABI 文件中, 其中包括接口名和接口参数的数据结构。如果参数是简单数据类型 (如定长整数, 定长浮点数等), 则求解路径的约束集合即可获取接口参数对应符号量的赋值。如果参数是复杂数据类型 (如字符串, 数字资产等), 那么在 WASM 格式的智能合约中, 实际参数是具体数据值的内存首地址, 单纯求解约束只能得到一个可能的内存地址。针对这种情况, 本研究会获取这些首地址符号量在符号内存中的连续内存值, 并进一步通过求

算法 3.5 基于代码覆盖率的路径选择算法**Input:** 路径集合 $traces$, 实际执行已覆盖代码地址集合 pcs **Output:** 最优路径 $\mathcal{T}$

```

1  $\mathcal{T} \leftarrow \Phi$ ;
2  $traces' \leftarrow []$ ;
3  $C \leftarrow []$ ;
4 for  $t \in traces$  do
5   if  $isModifyDataBase(t)$  and  $solve(getConstrains(t)) == satisfiable$  then
6      $traces'.add(t)$ ;
7      $pcs' \leftarrow getPcs(t)$ ;
8      $C.add(getLength(pcs' - pcs))$ ;
9   end
10 end
11  $traces' \leftarrow sortByCoverage(traces', C)$ ;
12  $\mathcal{T} \leftarrow traces'[0]$ ;
13 return  $\mathcal{T}$ ;

```

解约束集合以获取其中符号值的取值, 从而使用这些内存值构造请求的参数。

3.3 实验结果与分析

3.3.1 实验目的与环境

为了应对当前 EOSIO 平台智能合约的漏洞检测方法覆盖率低的问题, 本工作聚焦于 EOSIO 平台智能合约的链上数据库机制, 实现了一种基于循环符号执行的漏洞检测工具 EOSL。EOSL 主要由符号执行引擎和基于覆盖率的交易生成两个模块组成, 前者在固定链上数据库状态下进行符号执行以检测漏洞, 而后者负责生成并执行交易, 以更新链上数据库状态。通过符号执行和实际执行的循环迭代, EOSL 能够实现对智能合约的高代码覆盖率漏洞检测。

为了验证 EOSL 的检测效果和检测性能, 本节设计了以下四个实验内容, 本节的执行环境都如表 3.1 所示:

1) 实验一: 代码覆盖率对比实验。EOSL 旨在通过对链上数据库机制的精确建模以提高检测时的代码覆盖率, 从而提升漏洞检测效果。同时, EOSL 中存在分支深度限制和执行时间限制等参数, 这些参数用于缓解状态爆炸问题, 可能影响代码覆盖率。实验一将测试 EOSL 在使用不同参数时在实际合约中的检测代码覆盖率效果, 并选择合适的参数设置用于后续实验。

2) 实验二: 算法检测效果对比实验。本工作基于 WANA^[31] 构建了包含 133 个智能合约的验证数据集, 并通过人工验证的方式确定了各个智能合约中存在的漏洞。实验二将以该数据集为基础, 测试 EOSL 的漏洞检测效果, 并和其他工具的检测效果进行对比。

2) 实验三: 优化策略对比实验。实验三使用和实验二相同的数据集, 以对比

分析在采用不同内存模型和执行策略下的 EOSL 漏洞检测效果和检测时间，从而评估本章对符号执行引擎使用的两种优化策略的优化效果。

3) 实验四：真实智能合约检测效果实验。本工作基于 XBlock-EOS^[46] 的数据集构建了包含 5000 个真实智能合约的数据集。实验二将测试 EOSL 在真实智能合约中的检测效果。

表 3.1 实验配置

配置类型	配置参数
CPU	Intel(R) Xeon(R) Gold 5215 CPU @ 2.50GHz
内存	128G
显卡	NVIDIA GTX3090
操作系统	Ubuntu 18.04 LTS

3.3.2 代码覆盖率对比实验

本实验从链上随机选择了 100 个智能合约作为数据集，旨在测量本方法的代码覆盖率，其中代码覆盖率被定义为漏洞检测中被探索的指令地址数量。作为对比，本实验选择了具有代表性的混合模糊测试工具 WASAI^[30]。考虑到本方法中存在分支深度上限和执行时间上限等参数可能会影响代码覆盖率，因此在实验中对于本方法的多种参数设置分别进行了实验。由于本方法仅考虑响应函数内部的路径，为了公平比较，本实验仅保留了 WASAI 输出路径中位于响应函数调用之后的部分。

图3.6展示了代码覆盖率对比实验的结果，其中横轴表示运行时间，纵轴表示所有智能合约被探索的代码地址数量的累计。初始阶段，EOSL 的代码覆盖率低于 WASAI，主要是由于 EOSL 模拟执行的性能损耗和约束求解耗时的影响。随着时间推移，WASAI 的代码覆盖率的增长速度逐渐放缓，并缓慢接近其上限。相比之下，在 15 分钟后，EOSL 在三种参数设置下开始表现出优势，达到了高于 WASAI 的代码覆盖率。这主要因为 EOSL 相比 WASAI 对链上数据库机制的建模更加精确，从而能够探索更多链上数据库状态下的合约执行路径。同时，这也因为 EOSL 的符号执行引擎能够考虑几乎所有的智能合约输入情况，而 WASAI 采取的基于种子变异的输入生成方法存在不确定性。最终结果表明，EOSL 在这 100 个智能合约上的最高的代码覆盖率达到 84,199，比 WASAI 的峰值高出约 36%。这证明了 EOSL 采用的基于循环符号执行的漏洞检测方法能够实现比 WASAI 更高的代码覆盖率。

图3.6中 EOSL 在不同参数设置（分支深度限制 d ）下的实验结果显示，随着 d 的增加，EOSL 的代码覆盖率的增长速度变缓。增长速度减缓的原因在于随着 d 的增加，符号执行引擎付出的时间代价大于所提升的代码覆盖率。随着 d 的增

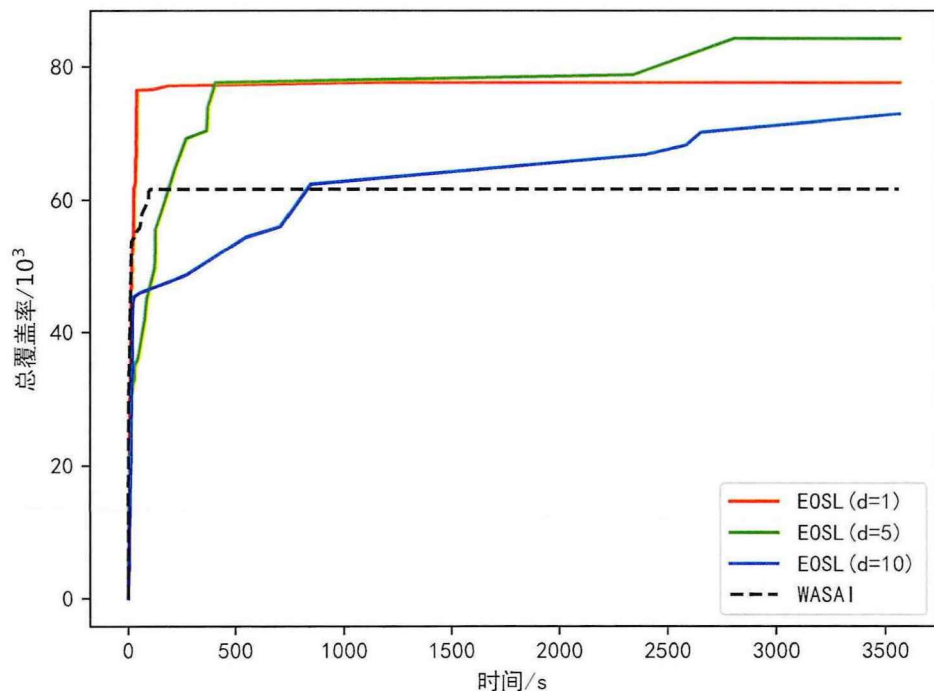


图 3.6 代码覆盖率对比实验结果

加, 符号执行引擎探索的路径会更多并且更长, 导致约束求解和模拟的耗时增加, 这带来的时间增长是指数级的。然而, 增加 d 所提升的代码覆盖率却无法和代价匹配。例如, 当路径中存在多重循环时, 即使增加分支深度限制, 新探索的代码之前都已经被探索过, 并不能实际提升代码覆盖率。同时实验结果表明, 增加 d 后, EOSL 的代码覆盖率并不一定增加。适当地提高 d , 会使得 EOSL 探索的路径的总数量和总长度都会增加, 从而提高了代码覆盖率的上限。然而当 d 过大时, EOSL 会侧重于探索路径的深度而忽略探索的广度, 从而可能导致在相同时间限制内代码覆盖率不增反减, 如 EOSL ($d=10$) 的覆盖率峰值低于 EOSL ($d=1$)。综上所述, 为了平衡检测效果和检测时间, 在后续实验中将 EOSL 的分支深度限制设置为 5, 超时时间设置为 1 小时。

3.3.3 算法检测效果实验

本工作的数据集是基于 WANA^[31] 的带源码智能合约构建的验证数据集。鉴于当前 EOSIO 平台智能合约漏洞检测领域缺少广泛认可的数据集, 本工作采用漏洞注入的方法生成数据集。WANA 提供了一个包含 84 个智能合约的数据集, 其中每个智能合约都有对应的源码。本实验在这些智能合约源码中随机地注入漏洞或漏洞修补, 并对源码进行编译, 以生成 WASM 格式的测试智能合约。随后, 通过人工验证的方式对注入的漏洞或修复进行确认。本工作的四种目标漏洞

对应的修改方法为:

1) 缺少权限检查漏洞: 缺少权限检查漏洞的修复需要在敏感操作前执行 `require_auth` 等权限检查库函数。因此, 为了注入该漏洞, 本实验在智能合约源码中随机删除已有的对 `require_auth` 等库函数的调用。为了进行漏洞修补, 本实验识别智能合约源码中的敏感操作, 并在其中每个敏感操作之前添加对 `require_auth` 库函数的调用。

2) 回滚漏洞和链上数据依赖漏洞: 本实验根据这两类漏洞特征生成固定的代码模板, 并在智能合约源码的响应函数中随机地插入。代码模板详见2.4小节中两函数的代码示例。为了修改这两个漏洞, 本实验将删除智能合约中所有内联操作相关函数的调用。

3) 整数溢出漏洞: 如果智能合约中存在参数为定长整数的响应函数, 则本实验在该响应函数中随机地插入对这些参数的加法运算。为了得到无整数溢出漏洞的智能合约, 本实验将删除合约中所有对定长参数的加、减和乘运算。

对于 WANA 数据集集中的每个智能合约, 随机使用上述的漏洞插入方法或漏洞修补方法, 本实验得到了一个包含 133 个智能合约的漏洞合约数据集。表3.2展示了本实验数据集中存在各种漏洞的合约的数量。

表 3.2 数据集中漏洞合约的数量

漏洞名称	合约数量
缺少权限检查漏洞	57
回滚漏洞	35
链上数据依赖漏洞	7
整数溢出漏洞	53
总计	133

本实验选择了多种指标以评测本工作的漏洞检测效果, 如表3.3所示。表中的 TP、TN、FP 和 FN 等分别代表真阳性、真阴性、假阳性和假阴性样本数。而表中 ACC、P 和 R 等英文标识分别表示漏洞检测的准确率、精确率和召回率, F1 值则是对召回率和精确率的综合评价。

为了探索本工作的检测能力, 本节选择了研究界一些具有代表性的开源工作作为对比方案:

1) WANA^[31]: WANA 是基于符号执行方法进行 EOSIO 平台智能合约漏洞检测的代表工作, 它对 WASM 格式的智能合约进行符号执行, 并使用代码特征以判断漏洞是否存在。

2) WASAI^[30]: WASAI 是使用模糊测试方法进行 EOSIO 平台智能合约漏洞检测的代表工作, 它搜集智能合约的执行路径, 并根据执行路径进行输入种子的

表 3.3 实验评价指标

评价指标	英文标识	计算公式
准确率	ACC	$\frac{TP + TN}{TP + TN + FP + FN}$
精确率	P	$\frac{TP}{TP + FP}$
召回率	R	$\frac{TP}{TP + FN}$
F1 值	F1	$\frac{2 \times P \times R}{P + R}$

变异以探索更多路径。WASAI 根据执行路径和执行造成的用户余额变动判断智能合约是否存在漏洞。

表 3.4 检测准确度对比实验结果

漏洞类型	方法	ACC	P	R	F1
缺少权限检查	EOSL	96.99%	100.00%	92.98%	96.36%
	WASAI	61.65%	75.00%	15.79%	26.09%
	WANA	-	-	-	-
回滚	EOSL	96.24%	100.00%	85.71%	92.31%
	WASAI	78.95%	100.00%	20.00%	33.33%
	WANA	-	-	-	-
链上数据依赖	EOSL	100.00%	100.00%	100.00%	100.00%
	WASAI	95.49%	60.00%	42.86%	50.00%
	WANA	97.74%	100.00%	57.14%	72.73%
整数溢出	EOSL	90.98%	100.00%	77.36%	87.23%
	WASAI	-	-	-	-
	WANA	-	-	-	-
总计	EOSL	96.05%	100.00%	86.18%	92.58%
	WASAI	78.70%	79.17%	19.19%	30.89%
	WANA	97.74%	100.00%	57.14%	72.73%

漏洞检测结果如表3.4所示。综合来看，EOSL 的检测效果优于其他漏洞检测工具。EOSL 标记了智能合约数据集中合约的 131 个漏洞，标记结果中包含 0 个 FP 和 21 个 FN。EOSL 的检测准确率达到了 96.05%，F1 值达到了 92.58%，高于 WASAI 约 22.05% 和 299.71%，准确率低于 WANA 约 1.72%，F1 值高于 WANA 约 27.29%。实验结果表明，与其他基于常规符号执行的方法相比，EOSL 使用的基于循环符号执行的漏洞检测方法，更精确地模拟了使用链上数据库机制的智能合约执行路径，从而能够更准确地检测该类智能合约中的漏洞；与基于模糊测试的方法相比，EOSL 在符号执行中进行漏洞检测，从而能够考虑更多的智

能合约输入情况，并具有更高的检测准确率。同时，EOSL 的基于代码覆盖率的路径选择和交易生成算法也使其能考虑更多的链上数据库状态，从而减少漏报。经过对结果的人工审查，EOSL 漏洞检测的假阴性结果的主要原因有两点：首先，分支深度限制使得 EOSL 难以覆盖具有复杂分支结构的智能合约中的深度分支，从而遗漏这些分支路径中的漏洞；其次，交易生成中 EOSL 对于复杂结构体参数采用了随机生成的方法，这可能导致实际执行中无法到达一些分支条件所需要的链上数据库状态。对于第一个问题，可以通过调整 EOSL 的分支深度限制参数（d）以覆盖更多分支来减少漏报，但该参数过大也可能导致漏报的增多，如图 3.6 中所示。因此，在第4章中本文提出了一种更合理的方法用以解决该问题。对第二个问题，可以通过实现更细致的交易生成算法来解决，这将作为未来工作的一部分。

WASAI 在检测准确率和精确率上和 EOSL 相近，但召回率和 F1 值偏低。WASAI 的漏洞检测结果中包含 5 个 FP 和 80 个 FN，检测准确率和 F1 值分别为 78.70% 和 30.89%。WASAI 召回率低的原因之一在于其难以处理复杂的事务依赖关系，尤其是当事务涉及复杂的链上数据库结构时。这一困难降低了 WASAI 的代码覆盖率，导致其无法探索到一些对链上数据库状态具有特定要求的漏洞路径，从而导致了假阴性样本多且召回率低。另一方面，WASAI 的漏洞检测依赖于智能合约执行时输出的执行日志和余额变动情况，这在发现较为明显的漏洞时表现良好，如假 EOS 漏洞、假收据漏洞和回滚漏洞等。然而，对于链上数据依赖漏洞等在执行日志中特征不明显的漏洞，其检测效果存在偏差。例如，WASAI 将日志中 `tapos_block_prefix` 和 `tapos_block_num` 等库函数的调用视为链上数据库依赖漏洞的标志，这导致了假阳性检测结果的出现。可以看到，在回滚漏洞检测时，WASAI 的精确率达到了 100.00%，而在对缺少权限检查漏洞和链上数据依赖漏洞的检测中，该指标仅为 75.00% 和 60.00%。

WANA 的总的漏洞检测准确率高于 EOSL，但其仅支持四种漏洞中的链上数据依赖漏洞。在链上数据依赖漏洞的检测中，WANA 的检测结果中包含 0 个 FP 和 3 个 FN，漏洞检测的精确率和 F1 分别为 97.74% 和 72.73%，分别低于 EOSL 约 2.30% 和 37.49%。这一结果的产生主要有三个方面的原因：首先，作为跨平台的符号执行引擎，WANA 并未对 EOSIO 平台智能合约的链上数据库机制进行细致建模，因此难以探索到智能合约的深度分支；其次，WANA 未考虑复杂的反序列化过程对符号执行的影响，即使在相同的分支深度限制下，由于需要模拟反序列化过程，其对响应函数内部代码的探索较为有限，导致漏洞检测的遗漏；最后，为了提高模拟执行的效率，WANA 过估计了 WASM 内存的稀疏性并以此建立了内存模型，这可能导致在模拟复杂分支时产生内存错误覆盖，进而导致模拟中断。

表 3.5 检测对比实验耗时

方法	平均检测时间	最小检测时间	最大检测时间	检测时间中位数
EOSL	409.74s	1.52s	3600.00s	33.65s
WASAI	54.67s	0.19s	301.92s	28.52s
WANA	9.26s	0.13s	314.91s	0.30s

各方法的漏洞检测时间如表3.5所示。相对而言，WANA 的检测速度最快，它具有最小的平均漏洞检测时间和最小的检测时间中位数。正如前文所述，这是因为 WANA 并未考虑 EOSIO 平台智能合约独有的特殊机制，并且采用了效率最高但牺牲精度的内存模型。比较 EOSL 和 WASAI，EOSL 的平均检测时间和最大检测时间都大于 WASAI，但两者的检测时间中位数的差距并不大。这样的结果是因为 WASAI 和 EOSL 都采用了多轮循环的漏洞检测算法，并且都需要对交易的实际执行和约束求解。而 EOSL 相对于 WASAI 需要更多次的循环次数以覆盖更多的链上数据库状态，这使得 EOSL 的最大检测时间大于 WASAI，并导致 EOSL 具有更高的平均检测时间。总的来说，EOSL 具有和 WASAI 相近的漏洞检测性能，达到了检测效果和检测时间的平衡。

3.3.4 优化策略对比实验

为验证 EOSL 的符号执行引擎中采用的两种优化策略的有效性，本实验在实验二的数据集上进行了对比实验。对比实验包括了不使用优化策略、使用一种优化策略以及同时使用两种优化策略的情况。在不使用内存模型优化策略的情况下，实验分别考虑了使用 WANA 的内存模型和 WASAI 的内存模型，因此总情况有 6 种。由于在不采用请求数据反序列化抽象策略时，难以生成触发请求交易，EOSL 无法更新链上数据库状态并进行下一轮符号执行。为了公平起见，在这 6 种情况下，实验都只进行了一轮符号执行。实验统计了各种情况下 EOSL 的平均检测时间和 ACC、P、R 和 F1 值等指标，分别用于表示漏洞检测的效率和效果。

表 3.6 优化策略对比实验

策略	平均检测时间	TP	FP	TN	FN	ACC	P	R	F1
M_0A_0	13.05s	0	0	380	152	71.43%	-	0.00%	-
M_0A_1	25.41s	46	0	380	106	80.08%	100.00%	30.26%	46.46%
M_1A_0	231.54	0	0	380	152	71.43%	-	0.00%	-
M_1A_1	1267.81s	77	0	380	75	85.90%	100.00%	50.66%	67.25%
M_2A_0	16.22s	3	0	380	149	71.99%	100.00%	1.97%	3.87%
M_2A_1	103.38s	114	0	380	38	92.86%	100.00%	75.00%	85.71%

优化策略对比实验的结果如表3.6所示。在表中， M_2 表示使用 EOSL 的混合

内存模型， M_I 表示使用 WASAI 的内存模型， M_0 表示使用 WANA 的内存模型， A_0 和 A_I 分别代表不使用请求数据反序列化抽象策略和使用该策略的情况。表中 M_0A_0 和 $M_I A_0$ 行数据中的 P 和 F1 值为空，因为这两种情况下没有被标记为阳性的样本，所以 P 和 F1 值无意义。实验结果证明了 EOSL 符号执行引擎中两种策略的有效性。首先，当 EOSL 采用固定的内存模型时，采用 A_I 策略比采用 A_0 策略的漏洞检测效果更好。例如，与采用 $M_I A_I$ 策略相比，采用 $M_I A_0$ 策略的 EOSL 的准确率提升了约 20.25%，F1 值也达到了 67.25%。类似的情况也出现在第三行和第四行数据、第五行和第六行数据中。这表明了 EOSL 采用的请求数据反序列化抽象的优化策略能有效提高检测效果。其次，当 EOSL 采用固定的反序列化处理策略时，采用 M_2 策略能够实现检测时间和检测效果的平衡。例如，在使用请求数据反序列化抽象策略，并分别使用了 WANA、WASAI 和 EOSL 的内存模型的情况下（即 M_0A_I 、 $M_I A_I$ 和 M_2A_I ）， M_2 策略使得平均检测时间相比 M_I 下降约 91.85%，而在检测效果方面则优于 M_I 策略，F1 值提升了约 27.45%，检测准确率提升了约 8.10%；同时，相比 M_0 ， M_2 策略的平均检测时间仅增加 77.97s，而 F1 值提升了约 84.48%，检测准确率提升了约 25.96%。因此，EOSL 内存模型采用的 M_2 策略能够更好地实现检测效果和检测效率的平衡。

3.3.5 真实智能合约检测效果实验

本实验的数据集包含来自链上的 5000 个真实智能合约。XBlock-EOS^[46]搜集了 2019 年 11 月 14 日之前 EOS 主链上所有的 55735 个智能合约，这些合约分布在 5594 个不同的地址上，构成了目前公开的最大规模的 EOSIO 平台智能合约数据集。需要注意的是，该智能合约数据集中可能包含相同智能合约的多个版本，因为 EOSIO 平台允许对链上的智能合约进行重新部署。从这些智能合约中，本实验随机选择了 5000 个智能合约以测试 EOSL 在真实智能合约中的漏洞检测效果。

表 3.7 真实智能合约检测效果实验

漏洞类型	脆弱合约数量
缺少权限检查	307
回滚	510
链上数据依赖	58
整数溢出	2141
总计	2485

EOSL 在真实智能合约上的漏洞检测结果如表3.7所示。在检测结果中，共发现了 2485 个智能合约存在漏洞，其中包括 307 个缺少权限检查漏洞、510 个回

滚漏洞、2141 个整数溢出漏洞和 58 个链上数据依赖漏洞。这四种漏洞影响了超过 48% 的智能合约，这凸显了 EOSIO 平台智能合约漏洞的普遍性，并彰显了对 EOSIO 平台智能合约进行高精度漏洞检测的迫切性。根据实验结果，整数溢出漏洞和回滚漏洞是影响最为广泛的漏洞，其中整数溢出漏洞的数量更是几乎为回滚漏洞的 4.2 倍。这是因为整数溢出漏洞相对更加隐蔽，并且 EOSIO 平台智能合约的开发者包含大量依赖库，即使经验丰富的开发者也难以手工检测到这种漏洞。这进一步表明了 EOSL 对 EOSIO 平台智能合约开发的意义。

为验证 EOSL 结果的准确性，本实验从数据集的 5000 个智能合约中随机选取了 40 个智能合约作为手工验证样本。具体而言，针对 EOSL 检测出的各类漏洞，本节在数据集中抽取 5 个标记为存在该漏洞的智能合约和 5 个被标记为不存在该漏洞的智能合约。对于这 40 个样本，本节使用 Ghidra^[47] 进行手工逆向工程验证。在这 40 个样本中，发现了 3 个假阴性的检测结果，本节进一步对它们进行了深入分析。其中，两个智能合约分别存在回滚漏洞和缺少权限检查漏洞，并且具有极为复杂的控制流结构。通过对 EOSL 执行日志的分析本节发现，由于合约中的控制流结构复杂，EOSL 在模拟执行过程中过早结束，未能探索到漏洞代码所在位置，导致漏洞未被成功检测到。在将分支深度限制设置为 20 并将超时设置为 5 小时后，EOSL 成功识别了这两个智能合约中的漏洞。另一个智能合约中存在回滚漏洞，根据对其执行日志的分析，EOSL 构建的部分触发请求交易未能成功引起链上数据库状态迁移，从而使得漏洞代码所在分支未被探索，导致将该样本误报为阴性。EOSL 的触发交易构建失败的原因在于，EOSL 在求解满足路径约束的字段值时只考虑了交易参数中的基础结构字段，而对其他复杂字段则采用随机生成的方法。综上所述，EOSL 达到了本工作的目标，能够对 EOSIO 平台智能合约进行高代码覆盖率的漏洞检测。

3.4 本章小结

本章针对当前漏洞检测方法无法准确模拟链上数据库机制的问题，提出了一种基于循环符号执行的漏洞检测方法。该方法在实际链上数据库状态下进行符号执行，并利用符号执行得到的路径信息进行链上数据库状态的更新。通过循环符号执行的形式，该方法既能够考虑各种链上数据库状态下的智能合约的执行路径，又尽可能避免了建模链上数据库机制带来的状态爆炸问题，从而实现了对该机制智能合约的高代码覆盖率的漏洞检测。实验结果表明，本章提出的基于循环符号执行的漏洞检测算法的漏洞检测 F1 值达到了 92.58%，优于 WASAI^[30] 和 WANA^[31] 等开源漏洞检测方法。

第4章 基于FASVERIF的深度漏洞检测方法

第三章详细介绍了一种基于循环符号执行的漏洞检测方法，该方法能够更好地模拟 EOSIO 平台智能合约的链上数据库机制，从而得到更好的漏洞检测效果。本章将分析第三章及大部分相关工作^[3,30-31]所采用的符号执行方法的局限性，并提出了本文的解决思路，详细介绍了一种基于 FASVERIF 的漏洞检测方法。与第三章方法的对比实验结果表明，该方法能够检测到第三章方法无法检测到的漏洞，并在包含复杂分支的智能合约中具有更高的检测效率。

4.1 引言

4.1.1 问题分析

基于符号执行的方法在应对状态爆炸问题时，采用了剪枝和分支深度限制等措施，然而这些策略可能会影响漏洞检测的准确率。如图4.1所示，图中展示了一个智能合约在符号执行中的状态转移路径。符号执行等方法使用的基于经验的剪枝方法存在可靠性不足的问题，可能在检测时遗漏危险状态，如图4.1中的状态4。因为符号执行的剪枝，探索到状态4的路径被阻断，从而导致漏洞未被检测到。同时，符号执行采用的固定分支深度的措施，同样可能导致对危险状态的遗漏，如图4.1中状态2所示。假设状态2需要循环指定次数（例如5次）才能到达，否则只能到达状态1。当符号执行的循环探索深度上限设置为4的时候，符号执行最多探索到循环4次的路径，无法到达状态2，从而导致检测时对漏洞的遗漏。

对于符号执行类的方法来说，即使不使用基于经验的剪枝策略，固定分支深度上限也是必要的。如果在符号执行方法中不固定分支深度上限，那么考虑图4.1中状态5对应的路径。可以看到，这条路径上存在一个循环，并且路径中不存在危险合约状态。当符号执行方法不固定分支深度上限时，符号执行方法需要探索该路径经过各种次数循环的情况，这将消耗大量时间。并且，这些探索实际上都是无意义的，因为该路径不可能到达危险合约状态。同时，如果该路径上的循环次数可以为无限次，那么符号执行方法在该路径上消耗的时间是没有上限的，如果存在总的检测时间限制，则将无法探索其他路径上可能的漏洞（如状态4所在的路径）。因此，符号执行方法实际不能将分支深度的上限设为无穷大。那么，即使符号执行方法将该上限设置为极大，它仍可能无法发现危险状态（如状态2）。

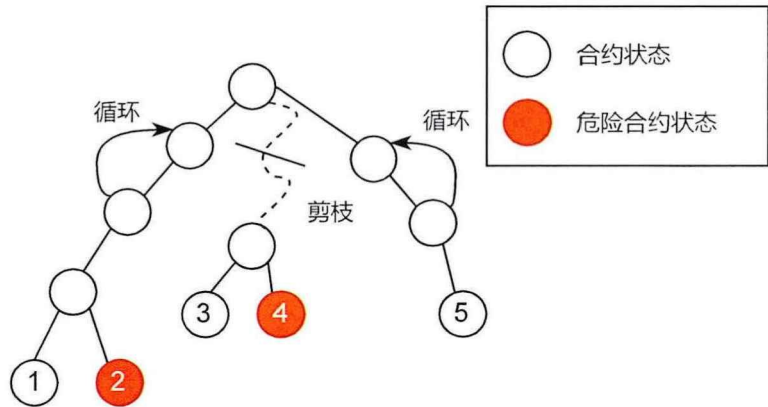


图 4.1 合约状态转移路径示意图

4.1.2 解决思路

为了解决前述问题，本章采用了基于反向路径构建的验证方法（FASVERIF）进行漏洞检测。该验证方法从危险状态开始逆向构建状态转移路径，以确定危险状态是否可从初始状态到达。由于反向路径构建的验证方法避免了对不存在危险状态路径的探索，因此无需使用剪枝和固定分支深度的措施。在使用反向路径构建的验证方法时，需要将安全目标定义为路径属性，并使用多重集合重写规则对智能合约进行建模。

综上所述，本章的主要工作包括：1）对 EOSIO 平台智能合约使用多重集合重写规则进行建模，并实现了建模过程的自动化。2）提出了涵盖 EOSIO 平台智能合约主流六种漏洞的路径安全属性，并给出了模型所需的额外修改。3）对 FASVERIF 的验证模块进行了扩展和优化，以实现对模型和属性的自动化验证。

4.2 具体方法

如图4.2所示，基于 FASVERIF 的 EOSIO 平台智能合约漏洞检测方法由模型构建、安全属性生成和验证三个模块组成。模型构建模块负责使用多重集合重写规则，将输入的 EOSIO 平台智能合约表示为状态转移系统。安全属性生成模块负责生成漏洞所对应的安全属性，并对已构建的模型进行修改以适配生成的安全属性。验证模块负责判断模型是否满足安全属性，并根据验证结果输出合约安全（满足安全属性）或者攻击路径（不满足安全属性）。

4.2.1 EOSIO 平台智能合约自动建模

为了建模 WASM 格式的 EOSIO 平台智能合约，本章将其状态表示为其执行期间的变量集合。根据2.3节的描述，这些变量集合根据应用场景可划分为四种：全局变量、局部变量、内存变量和栈变量。局部变量和栈变量的作用域限

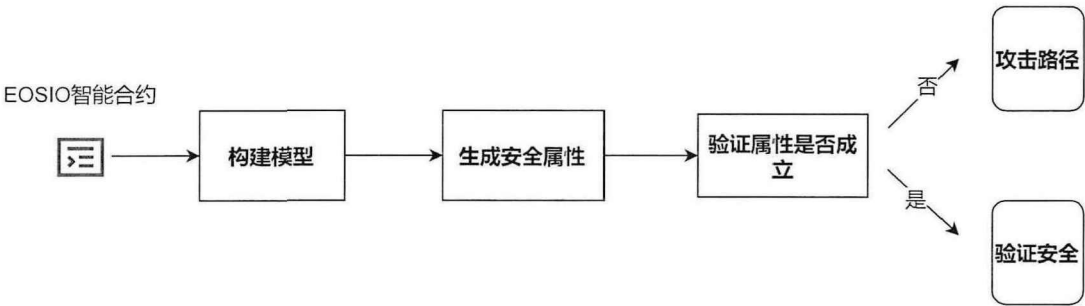


图 4.2 基于 FASVERIF 的漏洞检测方法框架图

定在其所定义的函数内部，不同函数间无法相互访问各自的局部变量和栈变量，同一函数也无法访问不同调用深度的局部变量和栈变量。因此，局部变量和栈变量分别对应于其所在的函数和函数调用深度。局部变量集合和栈变量集合分别写作 $Stack = \{stack_{f,d,i}, f \in F, d \in D_f, i \in N^s_{f,d}\}$ 和 $Local = \{local_{f,d,i}, f \in F, d \in D_f, i \in N^l_{f,d}\}$ 。其中 F 表示函数名集合， D_f 表示函数名对应的调用深度集合， $N^s_{f,d}$ 表示函数名和调用深度对应的栈变量编号集合， $N^l_{f,d}$ 表示函数名和调用深度对应的局部变量编号集合。内存变量和全局变量则在所有函数中都能访问，分别写作 $Memory = \{memory_i, i \in N^m\}$ 和 $Global = \{global_i, i \in N^g\}$ ，其中 N^m 和 N^g 分别表示内存字节变量编号集合和全局变量编号集合。

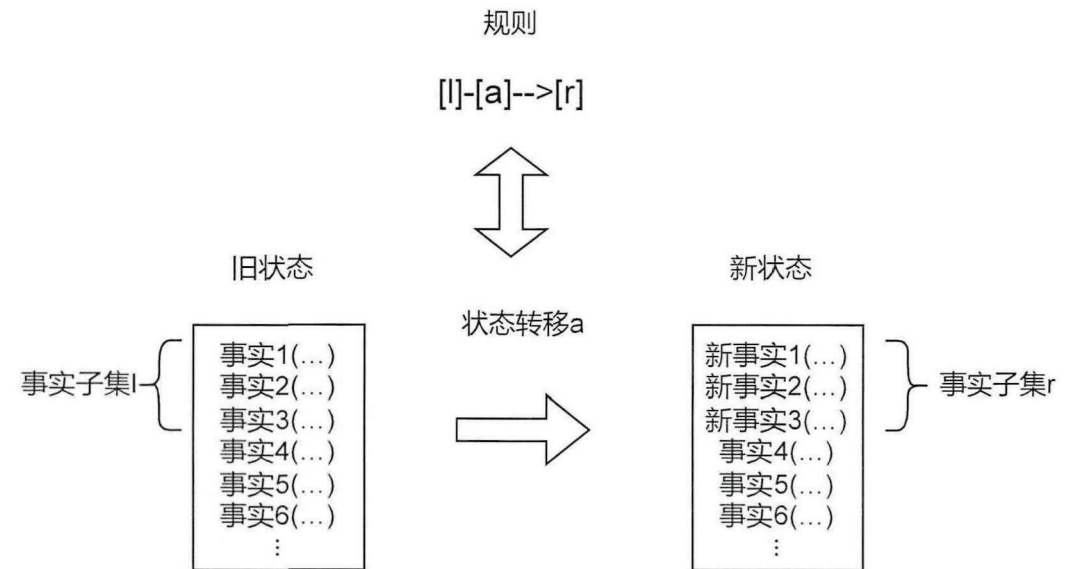


图 4.3 多重集合重写规则

为了形式化定义 EOSIO 平台智能合约的运行，本章使用多重集合重写语言对 EOSIO 平台智能合约进行建模。多重集合重写规则描述的是一个状态转移系统。在该系统中，状态被表示为事实的多重集合。事实是一个由项构成的谓词，事实集合形式化为 $Facts = S(t_1, \dots, t_k)$ ，其中 S 是谓词符号， t_i 表示项。事实具有固定的参数个数，模型中不应存在不同参数个数的相同事实。项包括双引号包裹

的字符串或者变量，项之间支持一些运算符（字符串的拼接等）以及一些内置的函数。如图4.3所示，多重集合重写规则是一个三元组 (l, a, r) ，其中 l 和 r 分别表示前一个状态和下一个状态的事实子集（称为前提和结论）， a 表示当前状态转移的标识符（称为动作），简写为 $[l] - [a] \rightarrow [r]$ 。多重集合重写规则使用推理规则符号表示为公式4.1。

$$\frac{l_1, \dots, l_k}{r_1, \dots, r_n} [a_1, \dots, a_m], k, n, m \in N \quad (4.1)$$

因为本章将合约状态定义为变量集合，所以合约执行时存在两种状态变化，分别对应于变量个数的变化和变量值的变化。值得注意的是，全局变量和内存变量虽然会被指令读取或修改值，但是并不会被指令添加或删除。因此，变量个数的变化不包括全局变量和内存变量。

本章使用 *Var* 事实表示函数内指令执行时变量个数的变化，如公式4.2所示。

$$\begin{aligned} &Var_{pc}(f, d, Local_{f,d,pc}, Stack_{f,d,pc}) \rightarrow \\ &Var_{pc'}(f, d, Local_{f,d,pc'}, Stack_{f,d,pc'}) \\ &, f \in F, d \in D_f \end{aligned} \quad (4.2)$$

公式4.2中的 pc 和 pc' 分别对应当前指令和下一条指令的相对地址， $Stack_{f,d,pc}$ 表示对应函数、调用深度和指令相对地址处的栈变量集合， $Local_{f,d,pc}$ 类似。为了突出重点，下文只在函数标识 (f) 和调用深度 (d) 出现变化时明确注明这两者。

对于变量值的变化，本章使用表示数值约束的事实进行表示。为了简化说明，本章使用运算符代替这些事实的名称，如公式4.3所示。

$$expr_l \theta expr_r, \theta \in \{<, <_s, =, \neq, :=\} \quad (4.3)$$

公式4.3中 $<$ 、 $<_s$ 、 $=$ 、 $\neq$ 和 $:=$ 分别表示无符号数比较小于、有符号数比较小于、比较相等、比较不等和赋值五种数值约束。其中赋值约束表示产生一个新变量对应于 $expr_l$ ，并且该变量的值和 $expr_r$ 对应的表达式值相等。

本章接下来将详细介绍如何使用多重集合重写规则对 WASM 格式的 EOSIO 智能合约进行建模，并探讨相应的自动化建模方案。本章将 EOSIO 平台智能合约的执行行为分为了五类进行建模：1) 函数内常规指令：常规指令是 EOSIO 平台智能合约中最普遍的指令，实现了智能合约的主要运算逻辑。常规指令可以分成算术指令、出栈指令和入栈指令三类。2) 分支跳转：定义了控制块或循环块的指令，加上跳转指令，实现了 EOSIO 平台智能合约的分支和循环功能。3) 初始化建模：在 EOSIO 平台智能合约运行之前，需要进行内存和表格等数据结构的初始化。4) 函数调用：通过函数调用指令，EOSIO 平台智能合约可以进行函数间的相互调用，并进一步实现递归等复杂操作。5) 库函数调用：库函数是 EOSIO 平台为智能合约提供的底层 API，用于扩展智能合约的功能。

(1) 建模函数内常规指令

常规指令中的基本算术指令先从操作数栈中弹出所需数量的操作数，再将运算结果放回栈顶。这些指令的运算操作包括加、减、乘、除、模、左移、右移、位与、位或和位异或等。由于这些指令间仅在所需操作数的数量和具体运算上存在差异，因此它们建模为的规则类似。以加法操作（`type.add`）为例，其建模规则如图4.4所示。加法指令被建模为一条规则，规则中包含一个赋值约束，表示将栈顶变量和次栈顶变量相加的结果赋值给新的栈顶变量。规则中的 top 表示执行指令前栈顶的高度， $stack_{top}$ 表示指令执行前的栈顶变量， $stack_{top-1}$ 同时表示次栈顶变量和新的栈顶变量。栈变量空间高度的降低在规则中体现为 Var 事实中的栈变量集合的更新，即 $Stack_{pc} - Stack_{pc+1} = \{stack_{top}\}$ 。

$$\begin{aligned} type.add \rightarrow [Var_{pc}(\dots, Stack_{pc}), \\ stack_{top-1} := stack_{top} + stack_{top-1}] - [] \rightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \end{aligned}$$

图 4.4 加法指令建模规则

算术指令中的比较指令对操作数进行比较，根据比较结果将对应的值压入栈顶（比较成立压入 1，比较不成立压入 0）。这些指令主要包括 `type.eq`（比较是否相等）、`type.neq`（比较是否不等）、`type.lt`（比较是否小于）、`type.eqz`（比较是否为零）等。如图4.5所示，每条比较指令会被建模为多条规则，这些规则间只有数值约束不同，分别表示比较结果的不同情况。规则中的数值约束可以分成两类，一类约束用于表示操作数之间的比较关系（如相等，小于等），一类约束用于表示对栈顶变量进行赋值。图中四条指令除 `type.eqz` 需要一个操作数外，都需要两个操作数，因此 $Stack_{pc} - Stack_{pc+1} = \{stack_{top}\}$ 。而 `type.eqz` 对应的栈变量集合的变化是 $Stack_{pc} - Stack_{pc+1} = \emptyset$ 。

除图4.5中所表示的比较指令的建模外，其他比较指令的建模都是类似的：大于比较指令（`type.gt/type.ge`）和小于比较指令（`type.lt/type.le`）的建模规则类似，只需反转比较约束的两个参数；不等比较指令（`type.neq`）和等于比较指令（`type.eq`）的建模规则类似，只需将两条规则中的 $=$ 和 $\neq$ 相互替换；指定操作数是有符号数的比较指令（`type.lts`，`type.les`等）和基本比较指令的建模规则类似，只需将 $<$ 替换为 $<_s$ 。

算术指令中的选择指令（`select`）需要三个操作数，如果栈顶变量（ $stack_{top}$ ）等于 0，压入次栈顶变量（ $stack_{top-1}$ ），否则压入次栈顶下的变量（ $stack_{top-2}$ ）。如图4.6所示，选择指令会被建模为两条数值约束不同的规则。这两条规则分别表示根据栈顶变量值的不同，对新的栈顶变量的赋值。因为选择指令需要三个操作数而压入一个变量，因此 $Stack_{pc} - Stack_{pc+1} = \{stack_{top}, stack_{top-1}\}$ 。

$$\begin{aligned}
type.eq &\rightarrow \begin{cases} [Var_{pc}(\dots, Stack_{pc}), stack_{top} = stack_{top-1}, stack_{top-1} := 1] - [] \\ \quad \rightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\ [Var_{pc}(\dots, Stack_{pc}), stack_{top} \neq stack_{top-1}, stack_{top-1} := 0] - [] \\ \quad \rightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \end{cases} \\
type.lt &\rightarrow \begin{cases} [Var_{pc}(\dots, Stack_{pc}), stack_{top-1} < stack_{top}, stack_{top-1} := 1] - [] \\ \quad \rightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\ [Var_{pc}(\dots, Stack_{pc}), stack_{top} = stack_{top-1}, stack_{top-1} := 0] - [] \\ \quad \rightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\ [Var_{pc}(\dots, Stack_{pc}), stack_{top} < stack_{top-1}, stack_{top-1} := 0] - [] \\ \quad \rightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \end{cases} \\
type.le &\rightarrow \begin{cases} [Var_{pc}(\dots, Stack_{pc}), stack_{top-1} < stack_{top}, stack_{top-1} := 1] - [] \\ \quad \rightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\ [Var_{pc}(\dots, Stack_{pc}), stack_{top} = stack_{top-1}, stack_{top-1} := 1] - [] \\ \quad \rightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\ [Var_{pc}(\dots, Stack_{pc}), stack_{top} < stack_{top-1}, stack_{top-1} := 0] - [] \\ \quad \rightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \end{cases} \\
type.eqz &\rightarrow \begin{cases} [Var_{pc}(\dots, Stack_{pc}), stack_{top} = 0, stack_{top} := 1] - [] \\ \quad \rightarrow [Var_{pc}(\dots, Stack_{pc+1})] \\ [Var_{pc}(\dots, Stack_{pc}), stack_{top} \neq 0, stack_{top} := 0] - [] \\ \quad \rightarrow [Var_{pc}(\dots, Stack_{pc+1})] \end{cases}
\end{aligned}$$

图 4.5 比较指令建模规则

$$select \rightarrow \begin{cases} [Var_{pc}(\dots, Stack_{pc}), stack_{top} = 0, stack_{top-2} := stack_{top-1}] - [] \rightarrow \\ \quad [Var_{pc+1}(\dots, Stack_{pc+1})] \\ [Var_{pc}(\dots, Stack_{pc}), stack_{top} \neq 0, stack_{top-2} := stack_{top-2}] - [] \rightarrow \\ \quad [Var_{pc+1}(\dots, Stack_{pc+1})] \end{cases}$$

图 4.6 选择指令建模规则

入栈指令根据立即数或栈顶值获取对应的变量，并将变量值压入栈中。根据压入栈中变量的不同来源，入栈指令可以分为四类：type.const i 压入立即数、get_local i 压入局部变量、get_global i 压入全局变量和type.load[len] offset压入内存变量。对入栈指令的建模规则如图4.7所示，其中内存加载指令（type.load[len] offset）以加载32位整数为例。除内存加载指令需要从栈中弹出一个值作为内存取址基址外，其他指令不使用栈变量。因此除内存加载指令外，规则中 $Stack_{pc+1} - Stack_{pc} = \{stack_{top+1}\}$ ；而内存加载指令对应的规则中， $Stack_{pc+1} - Stack_{pc} = \emptyset$ 。建模入栈指令的规则中需要一个赋值约束用于表示对新栈顶值的赋值。因为内存加载指令可能加载多字节数据，而内存变量是单字节的，所以可能需要进行结果的拼接。图中的CAT操作符用于表示字节变量的拼接。

$$\begin{aligned}
&type.const\ i \rightarrow [Var_{pc}(\dots, Stack_{pc}), stack_{top+1} := i] - [] \dashrightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\
&get_local\ i \rightarrow [Var_{pc}(\dots, Stack_{pc}), stack_{top+1} := local_func_id_i] - [] \dashrightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\
&get_global\ i \rightarrow [Var_{pc}(\dots, Stack_{pc}), stack_{top+1} := global_i] - [] \dashrightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\
&i32.load\ o \rightarrow [Var_{pc}(\dots, Stack_{pc}), stack_{top} := memory_{stack_{top}+o+3} \text{ CAT} \\
&\quad memory_{stack_{top}+o+2} \text{ CAT} \\
&\quad memory_{stack_{top}+o+1} \text{ CAT} \\
&\quad memory_{stack_{top}+o}] - [] \dashrightarrow [Var_{pc+1}(\dots, Stack_{pc+1})]
\end{aligned}$$

图 4.7 入栈指令建模示意

$$\begin{aligned}
&drop \rightarrow [Var_{pc}(\dots, Stack_{pc})] - [] \dashrightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\
&type.set_local\ i \rightarrow [Var_{pc}(\dots, Stack_{pc}), local_func_id_i := stack_{top+1}] \\
&\quad - [] \dashrightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\
&type.set_global\ i \rightarrow [Var_{pc}(\dots, Stack_{pc}), global_i := stack_{top+1}] - [] \dashrightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\
&i32.store\ o \rightarrow [Var_{pc}(\dots, Stack_{pc}), memory_{stack_{top-1}+o} := stack_{top} \text{ ECT } 1, \\
&\quad memory_{stack_{top-1}+o+1} := stack_{top} \text{ ECT } 2, \\
&\quad memory_{stack_{top-1}+o+2} := stack_{top} \text{ ECT } 3, \\
&\quad memory_{stack_{top-1}+o+3} := stack_{top} \text{ ECT } 4] - [] \dashrightarrow [Var_{pc+1}(\dots, Stack_{pc+1})]
\end{aligned}$$

图 4.8 出栈指令建模示意

出栈指令将删除栈顶变量，并将栈顶值直接丢弃或者赋值其他变量。图4.8展示了对出栈指令的建模规则，其中内存写入指令以写入 32 位整数为例 (*i32.store offset*)。对出栈指令的建模规则，除 *drop* 指令对应的规则外，其他规则中都使用赋值约束表示将栈顶值赋值给目标变量。因为内存写入指令可能要写超过一字节的数据，因此其规则中可能存在多个赋值约束。图中 *ECT* 操作符用于表示字节截取操作，*ECT i* 表示截取从低到高第 *i* 个字节。除内存写入指令外，栈变量集合的变化为 $Stack_{pc} - Stack_{pc+1} = \{stack_{top}\}$ 。而内存写入指令对应的栈变量集合变化为 $Stack_{pc} - Stack_{pc+1} = \{stack_{top}, stack_{top-1}\}$ 。

(2) 建模控制分支

WASM 的跳转指令 (*br_if i*) 根据栈顶的值判断是否进行跳转。该类指令通常与比较指令结合使用，比较指令用于检查是否满足跳转条件，并根据比较结果设置栈顶值，然后跳转指令进一步决定是否跳转。如图4.9所示，跳转指令被建模成两条数值约束和结果状态都不同的规则。这两条规则的数值约束不同，分别对应栈顶变量的不同取值。根据栈顶值的不同，结论中的 *Var* 事实的指令相对地址也不一样。跳转发生时是跳转目标的地址 *jump_pc*，跳转不发生后一条指令的地址 *pc + 1*。因为从栈中弹出了一个变量，所以栈变量集合的变化是

$Stack_{pc} - Stack_{pc+1} = \{stack_{top}\}$, 或者 $Stack_{pc} - Stack_{jump_pc} = \{stack_{top}\}$ 。

$$\begin{aligned} br_if\ i \rightarrow & \begin{cases} [Var_{pc}(\dots, Stack_{pc}), stack_{top} = 0] - [] \dashrightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\ [Var_{pc}(\dots, Stack_{pc}), stack_{top} \neq 0] - [] \dashrightarrow [Var_{jump_pc}(\dots, Stack_{jump_pc})] \end{cases} \\ br\ i \rightarrow & [Var_{pc}(\dots, Stack_{pc})] - [] \dashrightarrow [Var_{jump_pc}(\dots, Stack_{jump_pc})] \end{aligned}$$

图 4.9 跳转指令建模规则

其他跳转指令的建模类似于条件跳转指令的建模规则。无条件跳转指令 $br\ i$ 的建模不存在对于栈顶值的判断, 而直接进行跳转。因此无条件跳转指令对应的规则中不需要判断栈顶值的数值约束, 并且结论中 Var 事实的指令地址是 $jump_pc$ 。因为不用从栈中弹出变量, 所以 $Stack_{pc} - Stack_{jump_pc} = \emptyset$ 。if、else 和 end 定义的控制块分支建模规则如图 4.10 所示。图中前两条规则分别表示根据栈顶值选择 if 分支还是 else 分支, 类似于条件跳转指令建模规则。第三条规则表示 if 对应的分支的最后一条指令 ($else_pc - 1$) 建模时, 下一条指令的地址是 end 指令的位置 (end_pc), 而不是 else 指令的位置 ($else_pc$)。

$$if \rightarrow \begin{cases} [Var_{pc}(\dots, Stack_{pc}), stack_{top} = 0] - [] \dashrightarrow [Var_{else_pc}(\dots, Stack_{else_pc})] \\ [Var_{pc}(\dots, Stack_{pc}), stack_{top} \neq 0] - [] \dashrightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\ [Var_{else_pc-1}(\dots, Stack_{else_pc-1}), \dots] - [] \dashrightarrow [Var_{end_pc}(\dots, Stack_{end_pc})] \end{cases}$$

图 4.10 if-else-end 控制块建模规则

(3) 智能合约初始化建模

在每次智能合约执行之前, 都要根据 WASM 文件内容进行内存、全局变量和表的初始化。因为这些初始化的操作不是函数内指令, 所以建模初始化时不需要考虑局部变量集合。本章使用事实 $GVar$ 表示初始化时全局的状态, 并用递增的索引进行标识。以一条对于内存的初始化指令为例, 如图 4.11 所示, 它被建模成一条具有多个赋值约束的规则, 前后状态以一个递增的索引标识。因为状态转移间没有全局变量的增加和减少, 所以规则的状态中不需要指明所有全局变量。对全局变量和对表的初始化指令的建模规则是类似的, 只需修改赋值约束的目标变量。

$$\begin{aligned} data\ (i32.const\ addr)\ str \rightarrow & [GVar_{id}(), memory_{addr} := str[0], \\ & memory_{addr+1} := str[1], \\ & memory_{addr+2} := str[2], \dots] - [] \dashrightarrow [GVar_{id+1}()] \end{aligned}$$

图 4.11 内存初始化指令建模规则

(4) 建模函数调用

WASM中的函数调用(`call i`)指令,会根据被调用函数的参数个数,将调用者函数的栈顶值弹出;当被调用函数执行完成后,将被调用函数的栈顶值将作为调用者函数的新栈顶值。如图4.12所示,函数调用指令会被建模为产生四条规则:

1) 第一条规则,它的前提是调用者函数的 *Var* 事实,结论中包含一个 *Var_Temp* 事实以及一个 *Call* 事实。规则中的 f 表示调用者函数标识, f' 表示被调用者函数标识, d 表示调用者函数的调用深度。*Var_Temp* 事实用于暂存调用者函数的当前状态,并在调用结束后进行恢复。*Var_Temp* 事实使用调用者函数标识符、调用者函数调用深度和指令地址进行标识,以区分递归调用中不同深度被暂存的状态;暂存的状态中的栈变量不包括被弹出的变量,即 $Stack_{f,d,pc} - Stack'_{f,d,pc} = \{stack_{top-i}, i \in [0, L_f]\}$, 其中 L_f 是被调用函数的参数个数。*Call* 事实用于表示对于指定函数的调用。*Call* 事实使用调用者函数标识,被调用者函数标识和被调用函数调用深度进行标识,从而区分递归调用中不同的调用。规则一的前提中还包含多条赋值约束,以建模将调用者函数的栈顶值弹出作为调用函数参数的操作。

2) 第二条规则,其前提是一个 *Call* 事实,结论是被调用函数的 *Var* 事实。该规则中的 *Var* 事实表示被调用函数的初始状态,使用被调用函数第一条指令的地址作为标识。

3) 第三条规则,其前提是被调用函数的 *Var* 事实,结论包括一个 *Return* 事实。该规则的 *Var* 事实表示被调用函数的终止状态,使用被调用函数的终止指令的地址作为标识。如果被调用函数存在多个终止指令,第三个规则也会有对应多个。

4) 第四条规则,其前提是 *Return* 事实和 *Var_Temp* 事实,结论是调用者函数的 *Var* 事实。规则中的 *new_top* 表示调用者函数弹出参数并压入返回值的栈顶高度, *caller_top* 表示被调用函数返回值对应的栈顶高度。该规则中的 *Var_Temp* 事实是之前被暂存的调用者函数状态, *Var* 事实表示调用者函数新的状态。因为压入了返回值,所以被调用函数的栈顶高度减少 $L_f - 1$ 。因此 $new_top = top + L_f - 1$, 且 $Stack_{f,d,pc+1} - Stack_{f,d,pc} = \{stack_{f,d,top-k}, k \in [0, L_f]\}$ 。规则四中的前提中包含一条赋值约束,以建模将运行结果返回到调用者函数的栈顶的操作。

间接函数调用指令(`call_indirect`)的建模类似于函数调用指令,同样是四种规则以建模调用和返回的过程。与普通的函数调用指令不同的是,间接函数调用指令需要使用栈顶值作为索引,去表格域中获取对应的被调用函数指针。因此间接调用指令会被建模多次,分别对应于不同的被调用函数。这多次建模中,分别使用不同的等于约束,以限定栈顶变量的值为对应函数指针的索引。

$$\begin{aligned}
& [Var_{f,d,pc}(\dots, Stack_{f,d,pc}), local_{f,0} := stack_{f,d,top}, \\
& \quad local_{f,1} := stack_{f,d,top-1}, \dots, Stack_{f,top-1}, \dots] \\
& \quad - [] \rightarrow [Var_Temp_{f,d,pc}(\dots, Stack'_{f,d,pc}), Call_{f,f,d+1}()] \\
& [Call_{f,f,d+1}()] \\
& \quad - [] \rightarrow [Var_{f,d+1,0}(\dots, Stack_{f,d+1,0})] \\
& [Var_{f,d+1,end_pc}(\dots, Stack_{f,d+1,end_pc})] \\
& \quad - [] \rightarrow [Return_{f,f,d+1}()] \\
& [Return_{f,f,d+1}(), \\
& \quad Var_Temp_{f,d,pc}(\dots, Stack'_{f,d,pc}), \\
& \quad stack_{f,d,new_top} := stack_{f,d+1,caller_top}] \\
& \quad - [] \rightarrow [Var_{f,d,pc+1}(\dots, Stack_{f,d,pc+1})]
\end{aligned}$$

图 4.12 函数调用指令建模规则

EOSIO 平台智能合约初始化完成后, 会从智能合约的入口函数 (apply 函数) 开始执行。如图4.13所示, 本章生成一条规则用来表示对智能合约的初始函数调用。规则中 end_id 表示完成初始化时的 $GVar$ 标号, f 是 apply 函数的唯一标识。该规则以最后的初始化状态为前提, 以 apply 函数的第一条指令执行前的状态为结论。因为是最初始的函数调用, 规则中 apply 函数的调用深度为 0。

$$[GVar_{end_id}()] - [] \rightarrow [Var_{f,0,0}(f, 0, Local_{f,0,0}, Stack_{f,0,0})]$$

图 4.13 初始函数调用规则

(5) 建模库函数调用

库函数是 EOSIO 平台向合约提供的底层 API, 提供了对于链上数据库的增删改查、时间戳获取、合约调用等功能。对库函数的调用和对普通函数的调用在形式上是一致的 (call i), 但是库函数的具体实现不在智能合约中, 合约中只存在库函数的函数签名。图4.14展示了对于库函数调用的建模规则。规则中, 需要根据库函数签名中是否存在返回值, 更改栈变量空间的大小。即 $Stack_{pc+1} - Stack_{pc} = \{stack_{top+1}\}$ (存在返回值) 或者 $Stack_{pc+1} - Stack_{pc} = \emptyset$ (无返回值)。

$$call\ i \rightarrow [Var_{pc}(\dots, Stack_{pc})] - [] \rightarrow [Var_{pc+1}(\dots, Stack_{top+1})]$$

图 4.14 库函数调用建模

(6) 建模的自动化实现

对于上述建模方法进行自动化时, 主要面临两个问题:

1) 栈顶高度的确定: 栈顶高度即栈变量空间的大小, 上文中使用 top 符号进行表示。栈顶高度主要用于表示栈变量空间的大小变化和确定栈顶变量 ($stack_{top}$)。然而栈顶高度只有在确定了函数内执行路径和指令相对地址后才能确定, 并且同一指令在不同的执行路径中栈顶高度可能不同。

2) 内存地址的确定: 内存加载和写入指令都需要使用栈顶变量以计算具体的内存地址。本文的建模方法需要具体的内存地址以确定对应的内存变量, 因此需要在建模时确定内存地址。对于基址定义和内存指令在同一函数中的情况, 可以通过数据流分析、控制流分析等方法确定内存指令使用的基址的具体值。然而如果内存基址是作为函数参数传入的, 则无法仅通过分析当前函数确定内存基址, 需要进一步确定函数的调用路径。将内存基址作为函数参数的情况在 EOSIO 平台智能合约中很常见, 这主要因为 EOSIO 平台智能合约的编写语言是 C++ 语言。C++ 语言中的指针类型就是内存基址, 而指针类型可以作为函数参数。并且在 C++ 语言中, 如果将非基本数据类型变量作为函数参数, 编译中会自动将函数参数设置为变量的首地址。

为了解决上述问题, 本章采用了基于符号执行的方法进行自动化建模。本章的自动化建模流程如图4.15所示: 首先, 基于 WANA 的合约预处理模块, 对于二进制格式的 EOSIO 智能合约进行解析, 获取其合约信息, 如函数签名、初始化内容、函数指令等信息, 从而确定各个指令的操作数个数和类型、跳转目标地址、入口函数标识符、库函数标识符等。然后, 从入口点函数进行指定调用深度和分支深度的符号执行, 从而获取指令对应的栈顶高度和内存指令需要的基址值等信息。最后, 使用以上搜集到的信息, 基于前述的建模方案, 完成 EOSIO 智能合约的自动建模。特别地, 对于内存指令建模时, 需要在之前建模方案的基础上, 在规则的前提中加入基址实际值和基址对应变量间的等于约束, 以保证验证时路径生成的正确性。

4.2.2 属性生成及相关模型修改

使用上述多重集合重写规则, 本章将智能合约的执行形式化建模为一个状态转移系统。该状态转移系统的路径是从初始状态开始, 使用不同的规则进行状态转移的转移路径, 如公式4.4所示。

$$T = \emptyset \xrightarrow{a_1} S_1 \xrightarrow{a_2} S_2 \cdots \xrightarrow{a_n} S_n \quad (4.4)$$

公式4.4中, $\emptyset$ 为初始状态, $\emptyset, S_1, S_2, \dots, S_n$ 是该路径上的状态转移序列, $a_1, a_2, \dots, a_n$ 为对应的状态转移标识 (动作) 序列。

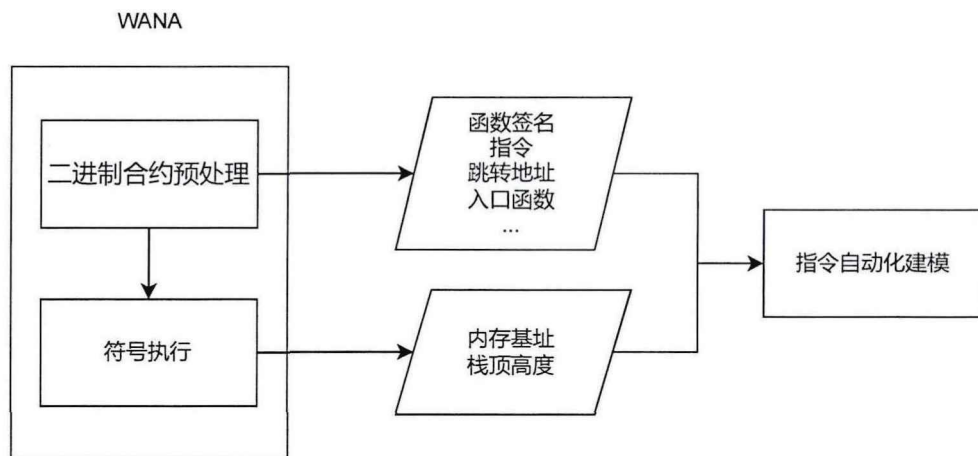


图 4.15 自动建模方法示意图

为了表示合约执行中的漏洞，本章将使用量化时间点的一阶逻辑公式定义路径安全属性，以表示上述状态转移系统中的危险状态。路径安全属性使用规则中的状态转移标识，定义动作发生的先后及因果关系。如果路径 T 对应的动作序列满足路径安全属性定义的动作先后顺序和因果关系，则称路径 T 满足安全属性。对于一组多重集合重写规则 R 定义的状态转移系统，如果其任意路径 T 都满足安全属性公式 Φ ，则称 P 满足安全属性 Φ ，记作 $P \models \Phi$ 。

量化时间点的一阶逻辑公式由三种逻辑连接词 $\wedge$ （也可写作 $\&$ ）， $\vee$ （也可写作 $|$ ）， $\neg$ （也可写作 *not*）、两种量词 $\exists$ （也可写作 Ex ）， $\forall$ （也可写作 All ）和六种原子命题共同组成。六种原子命题包括：

1) $f@i$ ，其中 f 是模型中的状态转移标识， i 表示这个规则在状态转移路径中发生的时间点。

2) $i = j$ ，表示 i 和 j 项的值相同。

3) $\#x = \#y$ ，表示时间点 x 等于 y 。

4) $x < y$ ，表示时间点 x 在时间点 y 之前。

5) $s_1 \sqsubset s_2$ 表示集合 s_1 是 s_2 的子集（也写作 $s_1 \ll s_2$ ）。

6) $\perp$ 和 $\top$ 分别表示逻辑真和假（也写作 T 和 F ）。

本章接下来将对于 EOSIO 平台智能合约主流的六种漏洞分别给出其安全属性和对于模型的进一步修改。之后本文将使用量化时间点的一阶逻辑公式表示安全属性，并继续使用多重集合重写规则表示对模型的进一步修改。六种漏洞包括：假 EOS 漏洞、假收据漏洞、链上数据依赖漏洞、回滚漏洞、缺少权限检查漏洞和整数溢出漏洞。本文在 2.4 节中给出了这些漏洞的原理和修复方法。

(1) 假 EOS 漏洞

假 EOS 漏洞对应的安全属性是：在合约的 `apply` 函数中，不存在一条路径，在该路径中能够到达 `on_transfer` 接口的处理函数，并且这条路径中 `code` 参数不

等于 $N(\text{eosio.token})$ 。安全属性形式化为公式4.5:

$$\text{"not (Ex \#t1 \#t2. Fake_EOS_Constranit()@t1 \& On_Transfer()@t2 \& t2 < t1)"}$$

(4.5)

为了支持该属性, 需要对模型进行两项修改:

1) 模型中需要额外添加一条规则用以表示 `apply` 函数的 `code` 参数不等于 $N(\text{eosio.token})$ 的情况, 如图4.16所示。该规则和初始函数调用规则4.13比较, 前提中多了一条不等约束用于表示 `code` 参数不等于 $N(\text{eosio.token})$, 并使用 `Fake_EOS_Constranit()` 标识这条规则的使用。规则中 `f` 表示 `apply` 函数, `locall` 即 `apply` 函数的 `code` 参数, 6138663591592764928 即为 $N(\text{eosio.token})$ 的值。

2) 模型中需要在 `on_transfer` 接口的处理函数的第一条指令对应的规则中, 使用 `On_Transfer()` 作为状态转移标识。

$$\begin{aligned} &[GVar_{end_id}(), local_l \neq 6138663591592764928] - \\ &[Fake_Constranit()] --> [Var_{f,0,0}(f, 0, Local_{f,0,0}, Stack_{f,0,0})] \end{aligned}$$

图 4.16 假 EOS 漏洞对应的初始函数调用规则

(2) 假收据漏洞

假收据漏洞对应的安全属性是: 在合约的 `on_transfer` 函数中, 不存在一条路径, 在该路径中 `to` 参数不等于合约当前地址, 但能够进行敏感操作 (如发送内联操作、调用函数、修改链上数据库等)。安全属性形式化为公式4.6:

$$\text{"not (Ex \#t1 \#t2. Fake_Notify_Constranit()@t1 \& SensitiveOp()@t2 \& t2 < t1)"}$$

(4.6)

为了支持该属性, 需要对模型进行两项修改:

1) 额外添加一条规则用以表示 `on_transfer` 函数的 `to` 参数不等于当前合约地址的情况, 如图4.17所示, 其中 `f` 是 `on_transfer` 接口对应的处理函数标识。该规则和原始的 `on_transfer` 函数的第一条指令建模的规则相比, 前提中多了一条不等约束用于表示 `to` 参数不等于当前合约地址。图中 `locall` 是 `on_transfer` 函数的 `to` 参数, `_self` 表示智能合约地址。

2) 对于进行敏感操作的指令, 在建模时使用 `SensitiveOp()` 进行标识。敏感操作包括转账和更新链上数据库, 转账需要调用 `send_inline`、`send_deferred` 等库函数, 而写入和修改链上数据库需要调用 `db_update_*` 和 `db_store_*` 类库函数。建模对于这些库函数调用时, 使用 `SensitiveOp()` 进行标识。

$$[Var_{f,d,0}(\dots), local_l \neq _self] - [Fake_Notify_Constraint(), On_Transfer()] \\ \longrightarrow [Var_{f,d,l}(\dots)]$$

图 4.17 假收据漏洞对应的初始函数调用规则

(3) 链上数据依赖漏洞

链上数据依赖漏洞对应的安全属性是：在合约的响应函数中，不存在一条路径，在该路径中获取了链上的状态，并在之后进行了敏感操作。安全属性形式化为公式4.7：

$$"not (Ex \#t1 \#t2. Get_Block_State()@t1 \& SensitiveOp()@t2 \& t2 < t1)" \quad (4.7)$$

为了支持该属性，需要在模型中使用 `Get_Block_State()` 标识对于 `tapos_block_num` 和 `tapos_block_prefix` 等库函数的调用建模规则。

(4) 回滚漏洞

回滚漏洞对应的安全属性是：不存在一条路径，该路径先使用区块链状态值作为 `rem` 指令的一个操作数，然后生成了内联操作类别的消息来执行转账操作。回滚漏洞只会出现在赌博智能合约中，而这类合约的一个特征是使用区块链状态值作为 `rem`（求余）指令的操作数^[3]。安全属性形式化为公式4.8：

$$"not (Ex \#t1 \#t2. Send()@t1 \& Block_State()@t2 \& t2 < t1)" \quad (4.8)$$

为了支持该安全属性，需要进行两项模型修改：

- 1) 使用 `Send()` 标识 `send_inline` 库函数调用所对应的规则。
- 2) 通过数据流分析，确定使用了区块链状态值作为操作数的 `rem` 指令，并在建模时使用 `Block_State()` 标识这些 `rem` 指令对应的规则。

(5) 缺少权限检查漏洞

缺少权限检查漏洞对应的安全属性是：在所有进行了敏感操作的路径上，进行敏感操作之前都进行了权限检查。安全属性形式化为公式4.9：

$$"All \#t1. SensitiveOp()@t1 \implies Ex \#t2. Authenticate()@t2 \& t2 < t1" \quad (4.9)$$

该安全属性需要在建模时，使用 `Authenticate()` 标识使用进行权限检查操作的指令规则。权限检查操作包括调用 `require_auth`、`require_auth2` 和 `has_auth` 这三个库函数。

(6) 整数溢出漏洞

对于整数溢出漏洞，对应的安全属性是：在所有可能发生整数溢出的操作时，结果不会到达定长整数的表示边界。可能存在整数溢出的操作即进行 32 位

或 64 位整数的加法、减法和乘法操作时，操作的至少一个操作数来自攻击者可控制的输入或链上状态。安全属性形式化为公式 4.10:

$$\text{"not (Ex \#t1. Overflow()@t1)"} \quad (4.10)$$

为了支持该安全属性，对于可能存在整数溢出的操作建模时，添加一条额外的规则。图4.18展示了对于存在整数溢出风险的 32 位整数加法指令的建模规

$$i32.add \rightarrow \begin{cases} [Var_{pc}(\dots, Stack_{pc}), & stack_{top-1} := stack_{top} + stack_{top-1} - [] \\ & \text{---} \rightarrow [Var_{pc+1}(\dots, Stack_{pc+1})] \\ [Var_{pc+i}(\dots, Stack_{pc+i}), & stack_{top-1} = 2147483647 - [Overflow()] \\ & \text{---} \rightarrow [Overflow\ End()] \end{cases}$$

图 4.18 存在整数溢出风险的加法指令建模规则

则。图4.18中的第二条规则就是额外添加的规则。该规则的前提是第一条规则中结论的 *Var* 事实，并使用一个等于约束，表示计算结果变量的值等于当前数位可表示的最大值。32 位有符号整数的最大值即为 2147483647。将该规则的标识为 *Overflow()*。

4.2.3 自动验证

本章的验证模块以 FASVERIF 的验证模块为基础。FASVERIF 验证模块负责根据模型和安全属性进行证明，最后生成违反安全属性的状态转移路径或者证明安全属性成立。图4.19展示了 FASVERIF 的验证模块的工作流程，它对模型文件进行语法解析以获得规则集合和安全属性。对于进行有效性检查的安全属性，验证模块将其转化为检查否命题的可满足性，从而寻找安全属性的反例；对于表示某些行为在执行中不会发生的安全属性，验证模块通过简化命题，以寻找证明该安全属性的路径。因此可以将 FASVERIF 的证明过程看成是状态转移树中搜索路径的过程。不同于常规树搜索算法中自顶向下的搜索，FASVERIF 的验证器首先根据安全属性得到目标状态，然后反向的搜索可以到达起始状态 ($\emptyset$) 的状态转移路径。在每一步的搜索中，根据目前所必须的事实集合，FASVERIF 的验证器查询可能产生这些事实的规则，并逐一尝试这些规则。当验证器尝试一条规则时，如果规则中存在数值约束，验证器会将约束加入到约束集合的开头，并进行约束集合的求解。如果验证器约束集合无法满足，验证器会尝试其他的规则；如果当前所有规则都无法满足，则回退到上一对应多条规则的事实集合。如果无法回退或发现完整路径，则输出对应的验证结果。

本文的验证模块对 FASVERIF 的验证模块的扩展包括两部分:

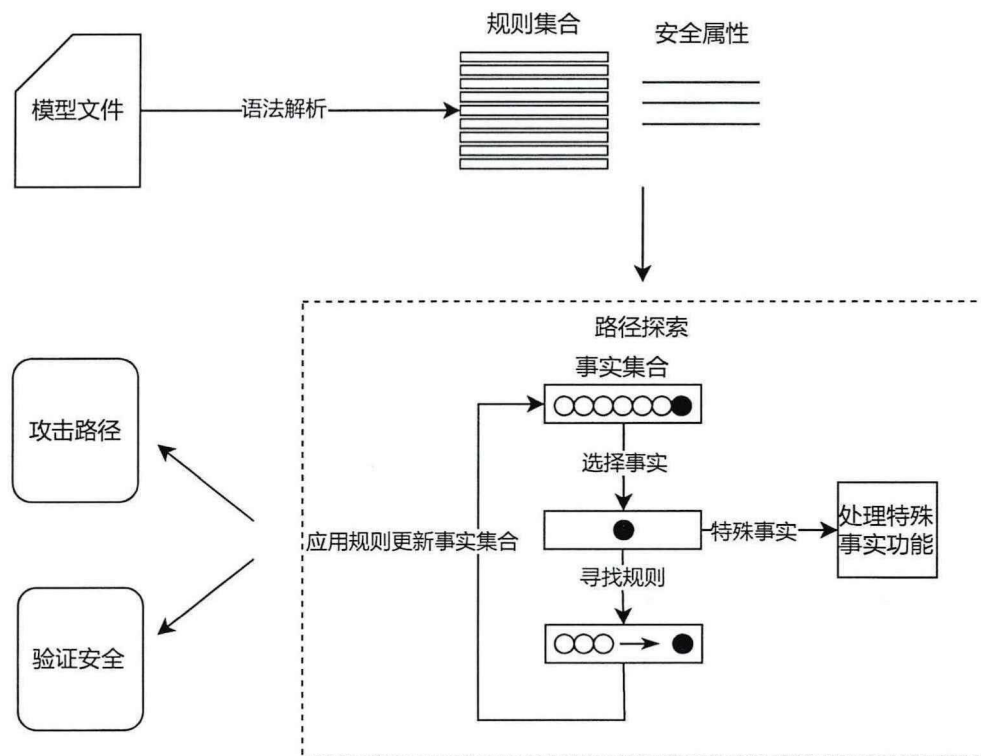


图 4.19 FASVERIF 验证模块工作流程示意图

1) 扩展了表达式的运算符集合。根据本章的自动化建模方案，除了FASVERIF支持的常规四则运算和取模取余外，还需要额外的算术操作如位或、位与、移位等。同时还需要加入本文在图4.7和图4.8中自定义的字节拼接运算符（CAT）和字节截取运算符（ECT）。

2) 扩展了数值约束的种类。根据公式4.3中定义的本章的建模方案需要的数值约束，除了FASVERIF验证模块中已提供的数值约束外，还需要符号数比较小于的数值约束。

除了上述这些必要的扩展外，为了提高验证速度，本章的验证模块采取了两种优化措施：

1) 修改了数值约束的整数表示，以优化约束求解时间。FASVERIF的验证器中进行数值约束求解时，对于有限长度的整数，一律使用的是256位长度进行求解。这对于FASVERIF来说是合理的，因为以太坊合约中并不能使用不同长度的整数进行运算。然而将有限长度整数都认为是256位，会使得约束求解时需要耗费更多的时间，因为模型求解器需要考虑整数更多可能的情况。本文对于EOSIO智能合约的内存建模时，每个字节被建模为一个变量，如果仍然将每个字节变量扩展到256位，会使得内存变量空间实际上扩大到64倍，导致验证时间过长。因此本章修改了验证器内部对于定长整数的表示和计算，以加速约束求解。同时，本章对于建模方案中提到的数值约束，添加了它们的变种，以支持

不同的整数位宽。以等于约束(=)为例,添加了 $=_8$ 、 $=_{16}$ 、 $=_{32}$ 、 $=_{64}$ 等数值约束,分别表示不同位宽的整数间的等于比较关系。特别的,本章将 $=$ 定义为和 $=_{32}$ 等价,因为32位是EOSIO平台智能合约最常使用的位宽。

2) 使用了延迟约束求解的方法,以优化约束求解次数。FASVERIF的验证模块中,每使用一条存在数值约束的规则扩展路径,就进行一次约束求解。然而,在对本章的模型进行路径探索时,一些规则即使存在数值约束也没有必要进行约束求解。如算术指令对应的建模规则,这些规则中只存在赋值约束,并且赋值约束的左右表达式中都不含常量,如图4.4所示。由于赋值约束右表达式中的变量值都还未被确定,这样的赋值约束添加进约束集合中,是不会使得约束集合求解出现矛盾的。因此,本文添加了`Solve_Cons`事实,只有当路径生成时遇到这个事实后才会进行一次约束求解。在建模时,在两类规则的前提中加入`Solve_Cons`事实。一类是第一条初始化指令对应规则,以保证对每条生成的路径至少进行了一次约束求解。一类规则中含赋值约束,并且赋值约束的右表达式中包含常数。

为了支持上述扩展和优化中提到的新运算符和特殊事实,本章对FASVERIF验证模块内部进行了修改。首先修改了验证器的语法解析模块,在其中添加了新的关键字并调整了运算符之间的优先级。其次,为了支持新运算符的语义,本文在处理数值约束类的特殊事实生成约束集合时,识别新的运算符并根据运算符的定义调用相应的z3功能。为了支持新的特殊事实,在路径探索时解析并处理特殊事实时,本章添加了新的特殊事实的识别条件和功能实现。本文新添加的特殊事实的功能分为两类,一类是用于添加约束集合,一类进行约束求解并根据结果决定是否终止这条路径继续生成。并且,本章去掉了数值约束类特殊事实处理时的自动约束求解。

4.3 实验结果与分析

4.3.1 实验目的与环境

为了解决当前符号执行方法因主观策略剪枝和固定分支深度剪枝方法造成的漏洞遗漏问题,本研究提出了一种基于FASVERIF的漏洞检测方法,从而实现了EOSIO平台智能合约的深度漏洞检测工具EOSVERIF。

为了验证EOSVERIF的检测效果和检测性能,本节设计了以下两个实验,实验配置和本文3.3节相同:

1) 实验一:复杂合约检测效果对比实验。鉴于当前符号执行因采用固定分支深度剪枝等策略而导致漏洞遗漏的问题,EOSVERIF采用了自动建模和反向路径构建的验证方法。为了验证EOSVERIF在具有复杂分支的智能合约中的漏洞检测效果,本实验构建了包含不同循环个数的智能合约,并分别使用EOSVERIF

和 EOSL 进行漏洞检测，以评估 EOSVERIF 的检测效果的检测性能。

2) 实验二：智能合约检测效果实验。EOSVERIF 作为前一工作的补充，旨在综合实现对 EOSIO 平台智能合约的高覆盖率和高精度的漏洞检测。因此，本实验使用 EOSVERIF 对 EOSL 检测智能合约验证数据集的阴性样本进行了检测，以评估 EOSVERIF 在实际合约中的检测效果。

4.3.2 复杂合约检测效果对比实验

本实验将人工构造具有不同复杂分支程度的智能合约。具体来说，本实验在合约的响应函数中插入循环和漏洞，插入的循环个数反映了该智能合约的复杂程度。图4.20展示了插入的循环代码。在此示例中，变量 `sym_input` 表示合约的可变输入值，仅当此类输入作为分支条件时，符号执行的路径才会产生分叉，因此代码中使用 `sym_input` 作为是否结束循环的判断条件。循环代码中改变 `condition_var` 变量的值，它的取值决定了漏洞是否触发。为简单考虑，循环代码中将 `condition_var` 的改变设置为每次循环值乘以 2，而将漏洞触发条件设置为 `condition_var` 变量值等于一个常数（2 的整数幂）。为了避免编译器对代码进行优化，代码中将常量值 2 替换为变量 `p`。在实际合约执行中，循环的结果必然导致 `condition_var` 等于 `sym_input`。然而，由于符号执行需要同时考虑 `sym_input` 的所有取值，因此在符号执行中，`condition_var` 的取值实际上取决于模拟执行循环体内部代码的次数，而该次数受符号执行分支深度参数限制。为增加智能合约的复杂程度，本实验在智能合约中插入了更多的循环代码，并分别使用不同的 `sym_input` 和 `condition_var` 变量。只有当所有 `condition_var` 变量分别和某个常数相等时才会触发漏洞。

对上述的智能合约，本实验进一步将回滚漏洞插入到响应函数中。具体而言，需要在函数开头插入模拟赌博智能合约的特征代码^[3]，并在所有循环代码之后插入对 `condition_var` 变量值的判断，取值满足条件时会调用 `send_inline` 库函数。以插入了三个循环的情况为例，插入的回滚漏洞代码如图4.21所示。其中，第 1 行为插入响应函数开头的代码，第 4 到 8 行为对于 `condition_var` 的条件判断和对应的 `send_inline` 库函数的调用。可以观察到，代码第 4 行中对 `condition_var` 的条件判断之间使用了 `&&` 作为连接，因此需要分别对 `condition_var1`、`condition_var2` 和 `condition_var3` 更新 `r1` 次、`r2` 次和 `r3` 次后才能满足条件。插入完成后，本实验使用 EOSIO 官方编译器将智能合约编译到 WASM 格式。

```
1 int condition_var=0;
2 for(int i=0;i<sym_input;i++){
3     condition_var=condition_var*p;
4 }
```

图 4.20 插入合约的循环代码示例

```
1 int a = tapos_block_num()*tapos_block_prefix()%23;
2 ...
3
4 if(condition_var1==(1<<r1)&&condition_var2==(1<<r2)&&condition_var3
   == (1<<r3)){
5     action(permission_level{ _self, N(active) },
6             N(eosio.token), N(transfer),
7             make_tuple(_self, N(from), asset(1000), string("test"))).
              send();
8 }
```

图 4.21 插入合约的回滚漏洞代码示例

在本实验中 EOSL 被选作对比方法，并对其在不同的分支深度限制参数下的漏洞检测效果进行了测试。具体而言，EOSL 的分支深度限制被设为 1、5、10 和 20，并分别对插入循环代码的函数进行模拟执行。为了避免耗费过多时间，本实验将漏洞发生条件中的常数（如 r_1 、 r_2 和 r_3 等）都设置为 3。鉴于 EOSL 采用了代码反序列化省略策略而不考虑响应函数调用之前的代码，为了比较的公平，本实验中的 EOSVERIF 仅对插入了循环代码的响应函数进行建模和自动验证。对于 EOSVERIF 和 EOSL 漏洞检测的超时时间，本实验都设置为 20 个小时，若超时则视为未能成功检测到漏洞。

复杂智能合约检测效果实验的结果见表 4.1。表中 d 表示 EOSL 的分支深度限制参数，EOSL ($d=1$) 表示分支深度参数设置为 1 时的 EOSL 工具。表中的 $\checkmark$ 表示对应方法成功检测到漏洞， $\times$ 表示对应方法未能成功检测到漏洞。实验结果表明：随着智能合约复杂度的增加，EOSL 需要更大的分支深度限制参数才能检测到漏洞。如仅包含一个循环的智能合约，EOSL ($d=1$) 无法检测其中的漏洞，而当 d 为 5 或以上时，EOSL 可以检测该合约中的漏洞。类似的，包含两个循环的智能合约可以在 d 为 10 或以上时被检测到，而包含 3 个循环的智能合约可以在 d 为 20 时被检测到。对比 EOSVERIF 和 EOSL 的实验数据，发现 EOSVERIF 能够检测到循环数量为 6 的智能合约中的漏洞，而这是 EOSL 在实验中各种参数 d 设置下都未能检测到的。总而言之，EOSVERIF 能够实现对 EOSIO 平台智能合约的高精度漏洞检测。

表 4.1 复杂智能合约检测效果实验结果

循环数	方法				
	EOSVERIF	EOSL ($d=1$)	EOSL ($d=5$)	EOSL ($d=10$)	EOSL ($d=20$)
1	$\checkmark$	$\times$	$\checkmark$	$\checkmark$	$\checkmark$
2	$\checkmark$	$\times$	$\times$	$\checkmark$	$\checkmark$
3	$\checkmark$	$\times$	$\times$	$\times$	$\checkmark$
4	$\checkmark$	$\times$	$\times$	$\times$	$\checkmark$
5	$\checkmark$	$\times$	$\times$	$\times$	$\checkmark$
6	$\checkmark$	$\times$	$\times$	$\times$	$\times$

表 4.2 展示了各方法进行漏洞检测时所消耗的时间。从实验结果可以观察到，

对于具有相同循环数量的智能合约，在使用 EOSL 并分别设置分支深度限制为 1、5、10 和 20 时，漏洞检测所需时间呈现上升趋势。这一现象归因于参数 d 的增加导致 EOSL 探索的分支长度增加，进而增加了模拟时间和约束求解时间。此外，随着分支深度限制参数的增加，EOSL 在使用相同参数 d 下所需的检测时间也呈现增长趋势，而这种趋势在参数 d 更大时更为显著。这是因为循环数量越多的合约，EOSL 需要模拟更多的分支路径，进而增加时间成本。另外，当参数 d 增大时，增加相同数量的分支会导致更多的时间消耗，因为新分支需要进行更深的探索。当智能合约的复杂程度较低时，EOSVERIF 的漏洞检测时间并没有优势。然而，随着智能合约复杂度的增加，EOSVERIF 漏洞检测的时间增长相对 EOSL ($d=20$) 的增长更为平缓。这使得 EOSVERIF 在处理复杂程度较高的智能合约（例如循环数为 5）时，相较于同样能检测出漏洞的 EOSL ($d=20$)，具有更短的漏洞检测时间。因此实际使用中本文建议结合两种方法，使用 EOSL 进行大规模的漏洞检测，使用 EOSVERIF 对重点智能合约进行更精确的分析。

表 4.2 复杂智能合约检测时间

循环数	方法				
	EOSVERIF	EOSL($d=1$)	EOSL($d=5$)	EOSL($d=10$)	EOSL($d=20$)
1	4716.26s	95.31s	113.79s	130.05s	81.52s
2	4876.72s	85.37s	94.45s	125.32s	173.72s
3	5550.28s	93.46s	105.67s	173.35s	708.28s
4	7633.31s	97.16s	109.17s	278.19s	2607.25s
5	8924.13s	100.99s	113.84s	417.74s	9252.13s
6	15756.51s	100.23s	115.51s	11077.31s	18750.86s

4.3.3 智能合约检测效果实验

本工作旨在作为第一个工作提出的漏洞检测方法的补充，以减少漏报从而提高检测准确率。因此，本实验使用第一个工作的阴性样本作为智能合约数据集，从中随机选择了 5 个检测结果为真阴性的合约和 5 个具有假阴性检测结果合约样本。为了验证安全属性，本实验设定了最大使用内存限制为 126G，并将验证时间上限设置为 24 小时。修改内存限制和物理内存大小一致，以避免验证器使用交换区带来的性能损失。若在内存限制和时间限制内未能成功验证安全属性，则认为智能合约满足相应的安全属性，也即不存在漏洞。

实验结果如表 4.3 所示。EOSVERIF 成功发现了 EOSL 未能检测到的 6 个漏洞，这使得 EOSVERIF 在整数溢出、回滚和缺少权限检查等漏洞的检测效果方面与采用各种参数设置的 EOSL 相比均有所提升，并且在链上数据依赖漏洞的检测方面与 EOSL 达到了相同的最优结果。尤其值得注意的是，在回滚漏洞的检测方面，EOSVERIF 相较于 EOSL 的最优结果，检测准确率提高了 42.86%，F1 值提高了 150.00%。综合本实验中 EOSVERIF 的检测结果与 EOSL 原本的实验

表 4.3 智能合约检测效果实验

漏洞类型	方法	TP	FP	TN	FN	ACC	F1
缺少权限检查	EOSVERIF	4	0	6	0	100.00%	100.00%
	EOSL(d=5)	2	0	6	2	80.00%	66.67%
	EOSL(d=10)	2	0	6	2	80.00%	66.67%
	EOSL(d=20)	1	0	6	3	70.00%	40.00%
回滚	EOSVERIF	4	0	6	0	100.00%	100.00%
	EOSL(d=5)	1	0	6	3	70.00%	40.00%
	EOSL(d=10)	1	0	6	3	70.00%	40.00%
	EOSL(d=20)	0	0	6	4	60.00%	-
链上数据依赖	EOSVERIF	1	0	9	0	100.00%	100.00%
	EOSL(d=5)	1	0	9	0	100.00%	100.00%
	EOSL(d=10)	1	0	9	0	100.00%	100.00%
	EOSL(d=20)	0	0	9	1	90.00%	-
整数溢出	EOSVERIF	2	0	7	1	90.00%	80.00%
	EOSL(d=5)	0	0	7	3	70.00%	-
	EOSL(d=10)	1	0	7	2	80.00%	50.00%
	EOSL(d=20)	1	0	7	2	80.00%	50.00%

结果，漏洞检测准确率和 F1 值达到了 97.36% 和 95.17%。特别地，两种方法的结合使得对回滚漏洞的检测准确率提高了 2.34%，F1 值提高了 5.14%。这些实验结果表明，EOSVERIF 作为 EOSL 的有力补充，能够实现对 EOSIO 平台智能合约的高覆盖率、高精度漏洞检测。对实验中出现的假阴性样本，人工分析结果显示，其产生原因是 EOSVERIF 在验证过程中达到了内存限制而被迫终止，导致未能成功识别漏洞。因此，通过增加物理内存的大小或启用虚拟内存并增加超时时间等方法，可以进一步提升 EOSVERIF 的漏洞检测效果。

4.4 本章小结

本章介绍了对于 EOSIO 平台智能合约主要指令和机制的建模方案，并提出了一种借助符号执行的自动化建模方法。本章分析并总结了 EOSIO 平台智能合约主流的六种漏洞对应的安全属性，以及这些安全属性对模型的修改需求。本章还对 FASVERIF 的验证器进行了扩展和优化，以支持对本章提出模型和属性的自动验证，从而实现对 EOSIO 平台主流漏洞的自动检测。最后，本章实现了该漏洞检测方法 EOSVERIF，在实际合约上进行了测试，并与前一工作的 EOSL 工具进行了对比。实验结果表明，EOSVERIF 能够检测 EOSL 难以检测的漏洞，并对于包含复杂分支的智能合约具有相对更快的漏洞检测时间；结合两种方法，

漏洞检测准确率和F1值分别达到了97.36%和95.17%。因此，EOSVERIF能够作为EOSL的有力补充，综合实现对EOSIO平台智能合约的深度漏洞检测。

第5章 总结与展望

5.1 工作总结

EOSIO 平台作为广泛流行的开源区块链平台,其上的智能合约也饱受安全威胁。因此,EOSIO 平台智能合约迫切需要准确、高效且全面的漏洞检测方法。本文考虑到 EOSIO 平台智能合约的特性和当前学术界现有漏洞检测方法的局限,提出了一种基于循环符号执行的漏洞检测方法,并据此实现了漏洞检测系统 EOSL,以实现 EOSIO 平台智能合约的高覆盖率的漏洞检测。此外,本文针对 EOSL 在检测复杂智能合约时受限于代码深度的问题,使用基于反向路径构建的验证方法,实现了细粒度建模和验证系统 EOSVERIF,以进一步提高对 EOSIO 平台复杂智能合约的漏洞检测能力。本文的工作内容和成果总结如下:

1) 提出了一种基于循环符号执行的漏洞检测方法,并据此实现了漏洞检测系统 EOSL。该系统以固定的链上数据库状态作为输入进行符号执行,并借助所获得的符号执行信息对链上数据库状态进行更新。该系统的符号执行引擎基于真实的链上数据构建了对链上数据库机制的模拟,从而缓解了其他符号执行工具在建模链上数据库机制时所存在的偏差和时间消耗问题。因此,该系统能够高效而全面地检测使用该类机制的智能合约的漏洞。据实验结果显示,该系统对 EOSIO 智能合约漏洞检测的准确率达到了 96.05%、F1 值达到了 92.58%,总体优于当前代表性的开源漏洞检测方案 WANA^[31]和 WASAI^[30]。

2) 提出了一种基于形式化验证的高精度漏洞检测方法,并以此实现了自动建模和验证系统 EOSVERIF。通过对 EOSIO 智能合约的自动建模、安全属性构建和基于反向路径构建的属性验证,本方法实现了对 EOSIO 平台智能合约的高精度漏洞检测,避免了符号执行方法因剪枝策略带来的问题。通过本研究的努力,由 EOSL 和 EOSVERIF 共同组成的系统成为了一个全面、高效且精确的漏洞检测系统。实验结果表明,EOSVERIF 能够准确识别 EOSL 未能成功识别出的漏洞,两种方法综合使用使得漏洞检测准确率达到了 97.56%、F1 值达到了 95.53%。

5.2 未来工作

本论文对 EOSIO 平台智能合约漏洞检测课题进行了深入研究,提出并实现了两个漏洞检测系统,综合实现了准确、高效且全面的漏洞检测目标。然而,在研究内容中,仍存在一些可以进一步改进和提高的内容,将在后续工作中进一步的完善。

1) EOSL 会在链上数据库状态改变后进行符号执行,然而符号执行可能包

含对相同路径的重复探索，如何有效优化符号执行以避免这种重复探索是下一步工作的重点之一。此外，EOSL 根据符号执行得到的路径约束构建相应的触发请求交易，但是由于性能考虑，EOSL 采用随机生成的方法处理触发请求中的复杂结构参数，这可能影响对某些智能合约的深度探索。因此，在下一步工作中，将尝试解析和生成接口中的复杂结构参数，以进一步提高 EOSL 的漏洞检测效果。

2) EOSVERIF 将智能合约执行中的所有变量都视作符号量以进行建模，然而这会导致 EOSVERIF 的约束求解相对于 EOSL 更频繁且耗时，并且会消耗大量内存。针对上述问题，下一步工作将对 EOSVERIF 的建模和验证模块进行优化，以进一步提高 EOSVERIF 的漏洞检测效率。

参考文献

- [1] Proof of stake[EB/OL]. [2023-04-11]. https://en.wikipedia.org/w/index.php?title=Proof_of_stake&oldid=1147139711.
- [2] A Complete Guide to EOS with Price Analysis - Phemex Academy[EB/OL]. Phemex[2023-04-11]. <https://phemex.com/academy/what-is-eos-blockchain>.
- [3] HE N, ZHANG R, WANG H, et al. EOSAFE: Security Analysis of EOSIO Smart Contracts [C]//2021: 1271-1288.
- [4] Crypto Research, Data, and Tools[EB/OL]. [2023-04-11]. <https://messari.io/asset/eos>.
- [5] Smart Contracts[EB/OL]. [2024-02-14]. <https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html>.
- [6] WebAssembly[EB/OL]. [2023-04-11]. <https://webassembly.org/>.
- [7] Diving into Ethereum's Virtual Machine(EVM): the future of Ewasm[EB/OL]. [2023-04-11]. <https://hackernoon.com/diving-into-ethereums-virtual-machine-the-future-of-ewasm-wrk32iy>.
- [8] SCHWARZ C. Ethereum 2.0: A Complete Guide. Ewasm.[EB/OL]. Medium(2021-02-05T17:55:34)[2023-04-11]. <https://blog.chainsafe.io/ethereum-2-0-a-complete-guide-ewasm-394cac756baf>.
- [9] WebAssembly (Wasm) • Polkadot Wiki[EB/OL]. (2022-11-21)[2023-04-11]. <https://wiki.polkadot.network/docs/learn-wasm>.
- [10] How EOSBET attacked by aabbccddeefg : eos[EB/OL]. [2023-04-11]. https://www.reddit.com/r/eos/comments/9fpcik/how_eosbet_attacked_by_aabbccddeefg/.
- [11] EOSBet Got Hacked Again; Lost 65000 EOS To Hackers[EB/OL]. [2023-04-11]. <https://latesthackingnews.com/2018/10/19/eosbet-got-hacked-again-lost-65000-eos-to-hackers/>.
- [12] 直接调用合约 transfer 攻击[EB/OL]. [2024-02-14]. <https://d1nn3r.github.io/2018/12/10/transferattack/>.
- [13] TSANKOV P, DAN A, DRACHSLER-COHEN D, et al. Securify: Practical Security Analysis of Smart Contracts[C]//CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2018: 67-82.
- [14] TIKHOMIROV S, VOSKRESENSKAYA E, IVANITSKIY I, et al. SmartCheck: static analysis of ethereum smart contracts[C]//WETSEB '18: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. New York, NY, USA:

- Association for Computing Machinery, 2018: 9-16.
- [15] LUU L, CHU D H, OLICKEL H, et al. Making Smart Contracts Smarter[C]//CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2016: 254-269.
- [16] Mythril[CP/OL]. ConsenSys(2023-05-28T15:01:57Z)[2023-05-29]. <https://github.com/ConsenSys/mythril>.
- [17] MOSSBERG M, MANZANO F, HENNENFENT E, et al. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts[C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2019: 1186-1189.
- [18] JIANG B, LIU Y, CHAN W K. ContractFuzzer: fuzzing smart contracts for vulnerability detection[C]//ASE '18: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. New York, NY, USA: Association for Computing Machinery, 2018: 259-269.
- [19] WÜSTHOLZ V, CHRISTAKIS M. Harvey: a greybox fuzzer for smart contracts[C]//ESEC/FSE 2020: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, NY, USA: Association for Computing Machinery, 2020: 1398-1409.
- [20] NGUYEN T D, PHAM L H, SUN J, et al. sFuzz: an efficient adaptive fuzzer for solidity smart contracts[C]//ICSE '20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. New York, NY, USA: Association for Computing Machinery, 2020: 778-788.
- [21] STEPHENS J, FERLES K, MARIANO B, et al. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts[C]//2021 IEEE Symposium on Security and Privacy (SP). 2021: 555-571.
- [22] PERMENEV A, DIMITROV D, TSANKOV P, et al. VerX: Safety Verification of Smart Contracts[C]//2020 IEEE Symposium on Security and Privacy (SP). 2020: 1661-1677.
- [23] HILDENBRANDT E, SAXENA M, RODRIGUES N, et al. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine[C]//2018 IEEE 31st Computer Security Foundations Symposium (CSF). 2018: 204-217.
- [24] K | Runtime Verification Inc[EB/OL]. [2023-06-03]. <https://kframework.org/>.
- [25] BRENT L, JURISEVIC A, KONG M, et al. Vandal: A Scalable Security Analysis Framework for Smart Contracts[EB/OL]. (2018-09-11)[2024-03-31]. <http://arxiv.org/abs/1809.03981>.
- [26] SCHNEIDEWIND C, GRISHCHENKO I, SCHERER M, et al. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts[C]//CCS '20: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA:

- Association for Computing Machinery, 2020: 621-640.
- [27] KALRA S, GOEL S, DHAWAN M, et al. ZEUS: Analyzing Safety of Smart Contracts[C]// Proceedings 2018 Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2018.
- [28] WANG W, HUANG W, MENG Z, et al. Automated Inference on Financial Security of Ethereum Smart Contracts[C]//32nd USENIX Security Symposium (USENIX Security 23). Anaheim, CA: USENIX Association, 2023: 3367-3383.
- [29] HUANG Y, JIANG B, CHAN W K. EOSFuzzer: Fuzzing EOSIO Smart Contracts for Vulnerability Detection[C]//Internetware '20: Proceedings of the 12th Asia-Pacific Symposium on Internetware. New York, NY, USA: Association for Computing Machinery, 2021: 99-109.
- [30] CHEN W, SUN Z, WANG H, et al. WASAI: uncovering vulnerabilities in Wasm smart contracts[C]//ISSTA 2022: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. New York, NY, USA: Association for Computing Machinery, 2022: 703-715.
- [31] JIANG B, CHEN Y, WANG D, et al. WANA: Symbolic Execution of Wasm Bytecode for Extensible Smart Contract Vulnerability Detection[C]//2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS). 2021: 926-937.
- [32] JIN L, CAO Y, CHEN Y, et al. EXGEN: Cross-platform, Automated Exploit Generation for Smart Contract Vulnerabilities[J]. IEEE TDSC, 2022.
- [33] LI W, HE J, ZHAO G, et al. EOSIOAnalyzer: An Effective Static Analysis Vulnerability Detection Framework for EOSIO Smart Contracts[C]//2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC). 2022: 746-756.
- [34] LI L T, ZHANG M. Poster: EOSDFA: Data Flow Analysis of EOSIO Smart Contracts[C]// CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2022: 3391-3393.
- [35] octopus[CP/OL]. FuzzingLabs(2024-03-25T11:00:19Z)[2024-03-31]. <https://github.com/FuzzingLabs/octopus>.
- [36] QUAN L, WU L, WANG H. EVulHunter: Detecting Fake Transfer Vulnerabilities for EOSIO's Smart Contracts at Webassembly-level[EB/OL]. (2019-06-25)[2024-03-31]. <http://arxiv.org/abs/1906.10362>.
- [37] 王自升. 针对EOS智能合约的形式化验证系统解释器模型研究[D]. 电子科技大学, 2020.
- [38] BIAN W, MENG W, WANG Y. Poster: Detecting WebAssembly-based Cryptocurrency Mining[C]//CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: Association for Computing Machinery, 2019:

- 2685-2687.
- [39] LEHMANN D, KINDER J, PRADEL M. Everything Old is New Again: Binary Security of WebAssembly[C]//2020.
- [40] PROTZENKO J, BEURDOUCHE B, MERIGOUX D, et al. Formally Verified Cryptographic Web Applications in WebAssembly[C]//2019 IEEE Symposium on Security and Privacy (SP). 2019: 1256-1274.
- [41] LEHMANN D, PRADEL M. Wasabi: A Framework for Dynamically Analyzing WebAssembly[C]//ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2019: 1045-1058.
- [42] AFL[CP/OL]. Google(2024-03-30T09:46:16Z)[2024-03-31]. <https://github.com/google/AFL>.
- [43] LEMIEUX C, SEN K. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage[C]//2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2018: 475-485.
- [44] MA M, HAN L, QIAN Y. CVDF DYNAMIC—A Dynamic Fuzzy Testing Sample Generation Framework Based on BI-LSTM and Genetic Algorithm[J]. Sensors, 2022, 22(3): 1265.
- [45] LI Y, CHEN B, CHANDRAMOHAN M, et al. Steelix: program-state based binary fuzzing [C]//ESEC/FSE 2017: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. New York, NY, USA: Association for Computing Machinery, 2017: 627-637.
- [46] ZHENG W, ZHENG Z, DAI H N, et al. XBlock-EOS: Extracting and exploring blockchain data from EOSIO[J]. Information Processing & Management, 2021, 58(3): 102477.
- [47] ghidra[CP/OL]. National Security Agency(2024-03-09T13:26:43Z)[2024-03-09]. <https://github.com/NationalSecurityAgency/ghidra>.