Computer Science and Application 计算机科学与应用, 2025, 15(4), 9-21
Published Online April 2025 in Hans. https://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2025.154073

Hans汉斯

RobinDetect：字节码级的漏洞规则提取工具

张居正

江西理工大学信息工程学院，江西 赣州

收稿日期：2025年2月28日；录用日期：2025年3月27日；发布日期：2025年4月3日

摘　要

随着区块链技术的广泛应用，智能合约的安全性问题日益凸显，尤其是跨链交易漏洞的检测成为当前研究的难点。本文提出了一种名为RobinDetect的字节码级漏洞规则提取工具，旨在通过对上链后的问题交易进行分析并快速提取其漏洞规则，实现对跨链桥漏洞的高效识别。RobinDetect通过交易收集器、交易分组器、调用流提取器、数据流提取器和规则提取器等组件协同工作，从交易数据中提取关键指令序列，并生成具有依赖关系的漏洞检测规则。实验表明，该工具比Aegis展现出了更高的检测精度，并且能够有效提取Xscope提供的交易漏洞规则，成功应用于跨链桥漏洞的检测。

关键词

跨链桥，漏洞规则提取，漏洞检测

RobinDetect: Bytecode-Level Vulnerability Rule Extraction Tool

Juzheng Zhang

School of Information Engineering, Jiangxi University of Science and Technology, Ganzhou Jiangxi

Received: Feb. 28th, 2025; accepted: Mar. 27th, 2025; published: Apr. 3rd, 2025

Abstract

With the widespread application of blockchain technology, the security issues of smart contracts have become increasingly prominent, especially the detection of vulnerabilities in cross-chain transactions, which has become a major challenge in current research. This paper proposes a bytecode-level vulnerability rule extraction tool named RobinDetect, which aims to analyze problematic transactions after they are on-chain and rapidly extract their vulnerability rules to achieve efficient identification of cross-chain bridge vulnerabilities. RobinDetect works through the collaborative

文章引用: 张居正. RobinDetect: 字节码级的漏洞规则提取工具[J]. 计算机科学与应用, 2025, 15(4): 9-21.
DOI: 10.12677/csa.2025.154073

efforts of several components, including a transaction collector, transaction grouper, call flow extractor, data flow extractor, and rule extractor. These components work together to extract key instruction sequences from transaction data and generate vulnerability detection rules with dependencies. Experiments have shown that this tool demonstrates higher detection accuracy than Aegis and can effectively extract vulnerability rules provided by Xscope, successfully applying to the detection of cross-chain bridge vulnerabilities.

Keywords

Cross-Chain Bridge, Vulnerability Rule Extraction, Vulnerability Detection

1. 引言

近年来，区块链技术的迅猛发展为其在金融、供应链、物联网等多个领域的广泛应用奠定了坚实基础[1] [2]，展现出巨大的潜力。然而，区块链技术的快速普及也带来了诸多安全挑战，其中智能合约的安全性问题尤为突出。智能合约作为区块链的核心应用之一，其代码一旦部署到区块链上，便自动执行且不可篡改。这种特性虽然带来了透明性和可信性，但也使得智能合约中的漏洞一旦被利用，可能导致不可挽回的资产损失和系统风险。

随着跨链技术[3] [4]的兴起，区块链的互操作性得到了显著提升，允许不同区块链网络之间的资产和数据转移。跨链桥作为跨链技术的核心实现，通过连接多个区块链网络，极大地促进了区块链生态的互联互通。然而，跨链桥的复杂性也引入了新的安全风险。由于跨链桥涉及多个区块链网络的交互，其安全性不仅取决于单个智能合约的代码质量，还依赖于跨链交易的整体逻辑和协议设计。一旦跨链桥出现漏洞，可能会导致资产被盗、网络中断等严重后果[5] [6]。

当前，智能合约漏洞检测工具主要集中在单个合约的漏洞检测上，而对跨合约漏洞的检测能力相对薄弱。现有的漏洞检测工具大多依赖于预定义的规则或静态分析，难以应对复杂多变的跨链桥漏洞。例如，SCG-Detector [7]提出了一种基于图注意力网络的智能合约漏洞检测方法，能够有效识别智能合约中的漏洞，但其对跨合约漏洞的检测仍存在局限性。此外，Contractsentry [8]作为一种静态分析工具，虽然能够检测智能合约中的漏洞，但在跨链桥场景下的适用性有限。SmartAxe [9]通过细粒度的静态分析结合机器学习的方法来识别和分类漏洞模式，但其规则定义的模糊性可能会对检测的精度产生一定的影响。

本文提出了一种名为 RobinDetect 的字节码级漏洞规则提取工具，旨在通过对上链后的问题交易进行分析，快速提取漏洞规则，实现对跨链桥漏洞的高效识别。RobinDetect 从交易数据中提取关键指令序列，并生成具有依赖关系的简易规则，通过少量的人工分析即可优化并提取漏洞规则。实验结果表明，RobinDetect 在检测精度上优于现有的先进工具 Aegis [10]，并能够有效提取由 Xscope [11]提供的跨链桥漏洞规则，为跨链桥漏洞的检测提供了一种新的解决方案。

本文的主要贡献包括：

1) 提出了一种基于交易的快速自定义漏洞规则提取方法，并设计了 RobinDetect 工具；
2) 通过与现有工具 Aegis 的对比，验证了 RobinDetect 提取规则的精准性；
3) 基于 Xscope 提供的跨链桥漏洞交易样本，证明了 RobinDetect 方法的可行性。

2. 背景和动机

在本节内容中，我们将对区块链上的关键知识进行介绍，以便于对我们的研究工作能有更深入的理解。

2.1. 以太坊与以太坊 RPC

以太坊是一个去中心化的公共区块链平台，由全球分布的节点网络维护。这些节点运行客户端软件，执行交易并参与区块的挖掘。矿工通过解决复杂的密码学问题竞争新区块的创建权。成功将区块加入区块链的矿工获得奖励和交易费。除了交易和挖矿，以太坊还支持智能合约的部署和执行。智能合约是自动执行的程序，它们存储在区块链上，能够在满足预设条件时自动执行相关操作。以太坊的交互性和灵活性部分得益于其远程过程调用(RPC)接口。以太坊 RPC 提供了一种标准化的通信协议，允许外部应用程序与以太坊区块链进行交互，无需运行自己的节点。通过 RPC，开发者可以发送交易、查询账户余额、调用智能合约，以及执行其他区块链操作。

2.2. EVM 与 EVM 指令

以太坊虚拟机 EVM 是一种纯堆栈、无寄存器的架构，它支持一系列图灵完备的操作码指令集。这些指令不仅包括基本的算术运算和控制流语句，还涵盖了对区块链状态的修改，如合约存储的变更和交易相关信息的查询。EVM 通过气体机制来管理智能合约的执行，确保合约能够安全地终止并防止潜在的拒绝服务攻击。气体作为一种计算资源的度量，与每条指令的执行成本相关联。智能合约的执行会影响其在区块链上的状态，这包括合约的余额和存储空间。EVM 在执行过程中维护着一个机器状态，包括可用气体、程序计数器、内存内容、内存中的有效字数和堆栈内容。EVM 的指令是字节码级的执行单元，是高级智能合约语言(如 Solidity)编译后的结果，能够在 EVM 中被解释和执行。本文提出的方法将围绕 EVM 指令展开。

2.3. 跨链与跨链桥

跨链技术是区块链领域的一项创新，它允许不同的区块链网络之间实现互操作性和通信。跨链桥作为这项技术的核心，是一种特殊的应用程序，它使用户能够在多个区块链之间转移代币。操作过程通常涉及在源链上锁定代币，然后桥接器在目标链上释放相应的代币，完成资产的跨链转移。跨链桥由链上路由器合约和链下中继器组成。路由器合约负责与代币合约交互，执行代币的锁定和解锁操作，并将交易记录为链上事件。链下中继器监听这些事件，并与目标链上的路由器合约协调，以完成跨链交易。跨链桥的实现促进了资产在不同区块链间的流动，但鉴于它们涉及大量资产转移，其安全性极为重要。

2.4. 智能合约安全漏洞

智能合约漏洞通常源于编程缺陷、逻辑错误或不充分的访问控制机制，这些漏洞可能被恶意利用，导致资金损失或合约功能被破坏。在智能合约的漏洞类别中，重入攻击、整数溢出、未授权访问、逻辑缺陷和不安全的外部调用是最为常见的几种。例如，重入攻击允许攻击者通过递归调用合约函数来操纵资金流，而整数溢出漏洞可能使攻击者获得超出预期的资产数量。近年来，随着跨链需求的增加，跨链桥作为智能合约的一个特殊应用进入人们视野。跨链桥通过连接不同的区块链网络，实现了资产和数据的互操作性。然而，跨链桥的复杂性引入了新的安全风险。Xscope [11]研究揭示了跨链桥在设计和实现过程中可能存在的安全漏洞，这些漏洞可能被利用来发起攻击，导致资产损失或网络中断。

2.5. 交易事件

交易事件是指在区块链上执行交易时触发的一系列动作和状态变化。每一笔交易在以太坊网络中都

是一个独立的事件，它可能包含对智能合约的调用或简单的金额转移。针对当前区块链的攻击威胁愈发频繁的情形下，我们的研究着重于关注针对智能合约调动的交易。这类交易都会产生一个事件，即对函数的调用执行，通常由 method 模块显示调用的函数名称，如图 1 所示。

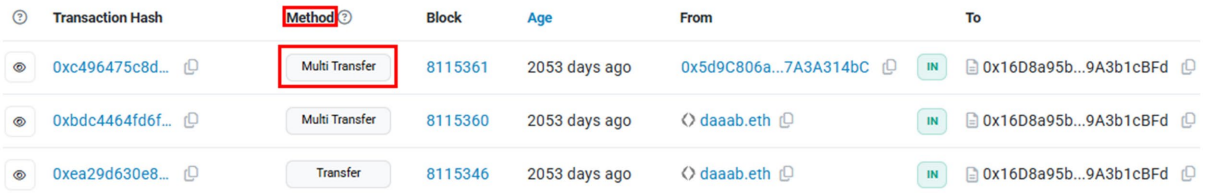

Figure 1. Name of the function called by the transaction event
图 1. 交易事件调用的函数名称

2.6. 动机

现有的智能合约漏洞检测工具和方法主要聚焦于单一合约的漏洞识别，然而在跨合约漏洞检测方面存在明显的局限性。这一缺陷在处理复杂的跨链桥智能合约时表现得尤为突出，因为当前的检测工具往往难以有效发现潜在的安全隐患。Contractsentry [8]作为一种静态分析工具，尽管能够检测智能合约中的漏洞，但在跨链桥场景中的适用性受到显著限制。跨链桥作为连接不同区块链网络的关键基础设施，其复杂的跨链逻辑和交互机制使得现有的检测方法过度依赖于人工分析，效率低下且容易遗漏关键漏洞。SmartAxe [9]和 Xscope [11]虽然通过定义复杂的安全模型来检测跨链桥的漏洞，但在应对不断出现的新攻击手段和漏洞类型时显得力不从心。鉴于此，我们迫切需要一种工具，能够迅速帮助研究人员针对问题交易漏洞进行快速分析和提取，并基于这些漏洞规则检测类似的异常交易行为。

3. 框架设计

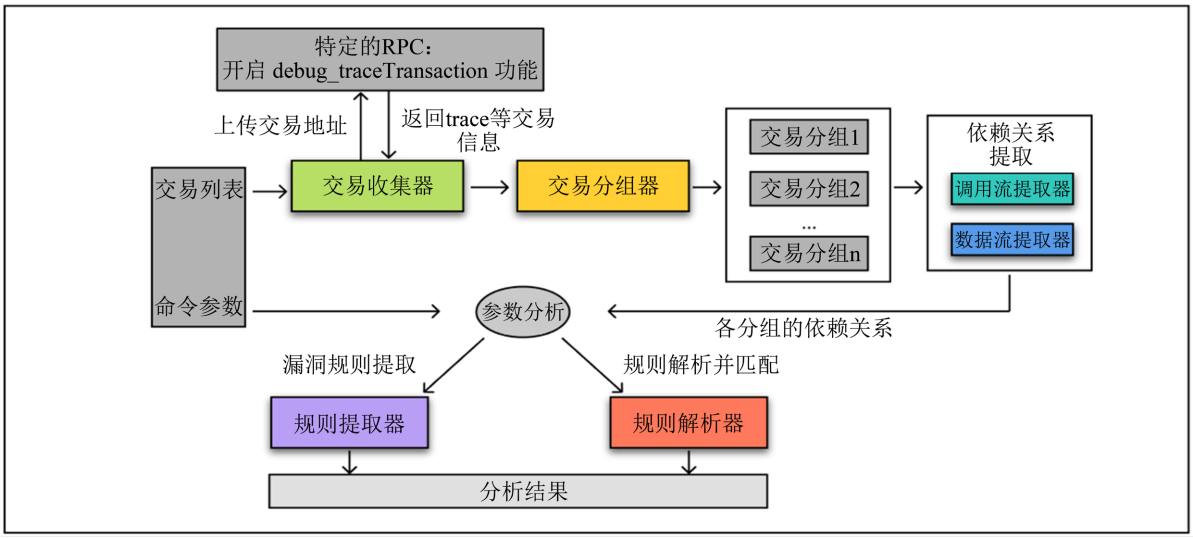

Figure 2. Overall framework of Robindetect
图 2. Robindetect 的总体框架

3.1. 交易收集器

交易收集器是 RobinDetect 的组件之一，其核心功能集中在链上交易信息的收集。该组件提供三种灵

活的收集模式：单笔交易收集、基于合约地址的交易收集以及基于区块号的交易收集，以满足不同场景下的数据分析需求。如图 2 所示，通过调用启用了特定调试功能的远程过程调用(RPC)接口，交易收集器能够高效地从区块链中提取交易执行时的调试数据。特别地，为了确保 RobinDetect 等后续分析工具能够获取充足的数据，所选的 RPC 接口必须启用 debug_traceTransaction 功能。debug_traceTransaction 是针对以太坊区块链的一项调试特性，它使得用户能够追踪和深入分析特定交易的执行细节。该功能通过模拟交易在以太坊虚拟机(EVM)中的执行过程，提供了一系列详尽的调试信息，包括操作码的执行情况、gas 消耗、执行深度以及潜在错误等关键数据，这些信息对于 RobinDetect 后续的漏洞规则生成至关重要。

3.2. 交易分组器

交易分组器是 RobinDetect 的一个关键组件，专门设计用于对按合约地址收集和按区块号收集的交易数据进行有效分组。该工具的分组策略如下所示：

A) 交易分组器依据交易的方法标识符(methodId)对交易序列进行初步分组。此过程中，具有不同 methodId 的交易被归为一组，而那些连续出现且 methodId 相同的交易则被单独分为一组。我们在实验中发现，漏洞触发的条件可以简单分为两类，一种是合约单笔交易时调用某个函数而触发，另一种是基于合约多笔交易的组合调用函数而触发。因而我们在分组时尽量将不同拥有 methodId 的交易分为一组。而对于连续的相同的 methodId 的交易则会单独分为一组。

B) 对于具有相同区块编号(blockNumber)但不同发送者地址(from)的交易，交易分组器会将它们划分为相同的组。对于同一组交易，通常会被打包至同一个区块上进行处理。在这个逻辑的基础上，我们将不同发送者地址的交易分在一组，同时还需要满足条件 A。由于我们会提取指令间的依赖关系，这样做可以避免对同类型交易在后续处理中造成的冗余依赖。

C) 若不满足条件 B，交易分组器还会将具有相同发送者地址(from)但不同输入数据(input)的交易归为一组。因为相同的输入往往意味着对同一个函数的调用，而我们将相同发送者地址且不同输入数据的交易归为一组，是为了尽可能避免同类型的交易分在一组。当然，我们也要同时满足条件 A，避免一些特殊情况造成的冗余分组。

D) 出于对实际交易情况的考虑，同一组的交易数量应该不多于 4 条。由于漏洞触发的条件苛刻，超出通过 3 次及以上交易进行调用触发的情况极其少见，因而我们对组合数量进行了限制，避免后期造成的冗余分析。

3.3. 调用流提取器

调用流提取器主要是用于分析指令间执行的依赖关系，最终生成一个依赖字典。智能合约在 EVM 的处理过程中，函数指令的调用大多会导致“depth”参数发生变化，但是也存在部分调用指令“depth”参数不发生变化的情况，因而我们要分为大致的两种情形进行处理。

A) depth 发生变化的情况

当前指令的 depth 大于前置指令的 depth 时，我们认为其存在依赖关系，即当前指令依赖于前置指令。当前指令的 depth 小于前置指令的 depth 时，我们也认为其与前置指令的依赖存在依赖关系，即当前指令的依赖等于前置指令依赖的依赖。

B) depth 未发生变化的情况

若当前指令为 JUMPDEST，则查找前置指令为 JUMP 或 JUMPI 的指令。若前置指令为 JUMP，且 stack 最后一位指向 JUMPDEST 的 PC，则关联起来作为依赖关系。若前置指令为 JUMPI，stack 的倒数

第二位为 1 且 stack 最后一位指向 JUMPDEST 的 PC，则关联起来作为依赖关系。

部分可能不会导致 depth 发生变化的前置指令，例如，“CALL”，“CALLCODE”，“DELEGATECALL”，“STATICCALL”，“CREATE”，“CREATE2”，当其 stack 最后一位的数值大于 0 或者是布尔类型，则当前指令与其关联起来作为依赖。不满足上述条件的，则将当前指令与前一个指令的依赖关联起来，即当前指令的依赖等同于前一个指令的依赖。

对于指令调用的依赖，我们全面地考虑到了特殊情况的依赖关系，使得调用流提取器提取的依赖字典更为完善。其中需要说明的是，当前指令和前置指令是根据 step 参数进行区分的，若当前指令的 step 为 n，则前置指令的 step 则为 n − 1。若涉及到多个交易，其指令数量为 n1 和 n2，我们会将指令集按照交易顺序进行合并，总的分析指令数量则为 n1 + n2。

3.4. 数据流提取器

数据流提取器主要负责提取数据流的依赖关系，这里主要是通过污点传播技术来获取指令与指令之间的数据流依赖字典。简单来说，我们会从源指令出发进行污点传播，后续的指令作为目的指令，进而检测是否存在污点。污点传播主要遵循跨堆栈 stack、内存 memory 和存储 storage 的 EVM 处理。其中，存储在 stack 和 memory 的污点会随着单笔交易的完成而清除，而存储在 storage 中的污点则会留在存储器中，因而对于多笔交易是否能联合分析的着重点在于对 storage 中污点的处理。而在 EVM 中，sload 和 sstore 指令用来处理在 storage 中数据的读取和写入，污点传播将模拟 sload 和 sstore 指令在 evm 中的执行过程，根据 storage 读取的数据来判断不同交易的指令是否存在依赖关系。同样的，对于 stack 和 memory 相关的操作的指令我们也会进行额外处理，最终我们从指令集中提取出一个基于数据流依赖的字典。

3.5. 规则提取器

规则提取器主要负责对交易的指令集的依赖关系进行提取，并在此基础上提取出每次交易事件的关键动作指令，从而生成具有依赖关系的指令序列。对于一组交易来说，规则提取器会将其指令集看作为一个事件，并尝试从中提取最长指令序列来代表此次交易事件的动作。当然，并非所有的指令都需要提取其依赖，规则提取器只从关键指令中提取依赖关系。以下是我们标记的关键指令，如“CALL”、“CALLER”、“CALLDATACOPY”、“CALLDATALOAD”、“DELEGATECALL”、“CREATE”、“SLOAD”、“SSTORE”、“MSTORE”、“MLOAD”、“GASLIMIT”、“GAS”、“JUMPI”、“SELFDESTRUCT”、“ADD”、“MUL”、“SUB”、“ISZERO”、“TIMESTAMP”、“JUMPDEST”，它们可以大致分为函数调用指令、合约创建/销毁指令、数据读写指令、GAS 相关指令、运算指令和跳转指令这几类。需要补充的是，若相邻指令的 depth 发生了变化，我们也会将其临时补充到关键指令的列表里。

在依赖关系中我们使用 step 来代表不同的指令，为了获取最长指令序列，我们会先从提取的依赖字典中统计依赖频率最高的指令作为切入点。需要注意的是，并非所有的高频依赖指令适合作为事件动作的切入点。根据测试和观察，我们最终选择“CALL”和“DELEGATECALL”作为切入指令。之后我们会在依赖字典中提取关键指令到切入指令的最长指令序列，同时还会提取从切入指令到后续关键指令的最长指令序列。最终将两者进行合并生成最长的指令序列集，该指令序列集将表示此次交易事件的动作。

为了更快地获取最长指令序列，我们通过依赖字典生成了有向图，并使用 DFS(深度优先)算法进行处理。该算法地优点在于可以快速查询特定节点之间的最长路径，同时还通过使用记忆化(Memoization)技术来进行优化，它会存储已经计算过的结果以避免重复计算，这将大大提升最大路径的提取速度。

由于我们只针对部分的指令进行依赖关系提取，最长的指令序列并不能代表合约的有效行为。为了弥补这一不足点，我们继续进行了研究，并在后续实验中发现指令的 GasCost 更能反馈整个合约的运行情况，其中 GasCost 是指执行某个指令操作所需的 GAS 数量。因而在最长指令的基础上，我们补全了针对 GasCost 消耗的计算，最终要提取的是消耗最大的指令序列，这样就可以转换为有向加权图的问题。整个指令序列的消耗 $Cost_L$ 可以表示为：

$$Cost_L = \sum_0^N GasCost_i \tag{1}$$

在这里，我们采用动态规划的算法，该算法能够高效地计算最大权重路径。以下图 3 为例，其中指令序列 Instruction Sequence 是指令的 step 的集合，而 step 代表当前指令在整个指令集中所处的位置。$Transcation_1$ 的指令范围是 0 到 420，$Transcation_2$ 的指令范围在 $Transcation_1$ 的基础上从 421 到 875，而 $Transcation_3$ 包含了 $Transcation_1$ 和 $Transcation_2$ 中的所有指令。$Transcation_1$ 和 $Transcation_2$ 的 Cost 之和远大于 $Transcation_3$，这是因为相对于提取的代表交易重要操作的指令序列，我们还需要考虑到不同交易之间的关联操作的指令序列。当然，具有关联的指令序列的消耗必定要比单个交易提取的指令序列要高，这样才能代表该组交易的主要行为。

Transcation1 : 0x05f71e1b2cb4f03e547739db15d080fd30c989eda04d37ce6264c5686e0722c9
Instruction Range : (0, 420)
Instruction Sequence :
[12, 109, 112, 114, 186, 188, 190, 199, 201, 203, 215, 225]
12(CALLDATALOAD)~~>109(JUMPI)==>112(ISZERO)~~>114(JUMPI)==>186(SLOAD)~~>188(ISZERO)~~>190(JUMPI)==>199(SLOAD)
~~>201(ISZERO)~~>203(JUMPI)==>215(TIMESTAMP)~~>225(SSTORE)
Cost : 20428

Transcation2 : 0x47f7cff7a5e671884629c93b368cb18f58a993f4b19c2a53a8662e3f1482f690
Instruction Range : (421, 875)
Instruction Sequence :
[433, 525, 528, 530, 539, 875]
636(SLOAD)~~>650(SSTORE)~~>707(SLOAD)~~>714(JUMPI)==>718(MSTORE)-->863(JUMP)==>875(SELFDESTRUCT)
Cost : 25421

Transcation3 : 0x05f71e1b2cb4f03e547739db15d080fd30c989eda04d37ce6264c5686e0722c9 +
0x47f7cff7a5e671884629c93b368cb18f58a993f4b19c2a53a8662e3f1482f690
nstruction Range : (0, 875)
Instruction Sequence :
[12, 109, 112, 114, 120, 128, 153, 354, 355, 357, 379, 386, 593, 597, 598, 600, 636, 650, 707, 714, 718, 863, 875]
12(CALLDATALOAD)~~>109(JUMPI)==>112(ISZERO)~~>114(JUMPI)==>120(CALLDATALOAD)~~>128(CALLDATALOAD)~~>153(MSTORE)
~~>354(ISZERO)~~>355(ISZERO)~~>357(JUMPI)==>379(MSTORE)~~>386(SSTORE)~~>593(SLOAD)~~>597(ISZERO)~~>598(ISZERO)
~~>600(JUMPI)==>636(SLOAD)~~>650(SSTORE)~~>707(SLOAD)~~>714(JUMPI)==>718(MSTORE)-->863(JUMP)==>875(SELFDESTRUCT)
Cost : 30637

Figure 3. Rule extraction for combination transcation
图 3. 组合交易的规则提取

3.6. 规则解析器

规则解析器是用于检测目标交易指令集中是否存在指定的漏洞类型，这里我们参考了 Aegis 设计思路，用特定领域语言(DSL)进行描述漏洞检测规则。如图 4 所示，我们将 DSL 主要分为若干个模块来描述漏洞检测规则。模型(Model)是 DSL 的顶层结构，包含多个模式(Pattern)和注释(Comment)。模式(Pattern)，描述一个具体的模式，包含描述(Description)和条件(Condition)。而条件(Condition)定义了多种条件判断的语法，包括比较、算术运算、依赖关系等。注释(Comment)则用于在 DSL 中添加注释。执行(Execution)是一个关键组件，可用于表示执行过程中的属性(如操作码、栈、内存、交易信息等)，用于构建条件逻辑。

```
模型(Model): patterns*=Pattern | Comment;

模式(Pattern): 'Description' ':' description=STRING
               'Condition'   ':' condition=Condition;

注释(Comment): /\/\/.*$/;

条件(Condition): ArithmeticOperation | ComparativeOperation | Relation;

算数运算(ArithmeticOperation): add | subtract | multiply | ...

比较运算(ComparativeOperation): Equal | NotEqual | LessThan | ...

依赖关系(Relation):      ControlDependency | DataDependency | Follows;

执行(Execution):      ('opcode' | 'depth' | 'pc' | 'address' | Stack | Memory | Transaction | 'executeResult');
...
```

Figure 4. Text structure 1 based on DSL rules
图 4. 基于 DSL 规则的文本结构 1

```
加法(add):   '(' (x=ArithmeticExpression | x=Source | x=Destination | x=Execution | x=INT) '+' (y=ArithmeticExpression |
y=Source | y=Destination | y=Execution | y=INT) ')';
...

不等于(NotEqual):      '(' (x=ArithmeticExpression | x=Execution | x=Source | x=Destination | x=INT | x=STRING) '!='
(y=ArithmeticExpression | y=Execution | y=Source | y=Destination | y=INT | y=STRING) ')';
...

数据流依赖(DataDependency):   '(' source=Condition '~~>' destination=Condition ('where' condition=Condition)? ')';

...
源(Source):            'src' '.' property=Execution;
目的(Destination):     'dst' '.' property=Execution;

算数表达式(ArithmeticExpression): add | subtract | multiply;

交易(Transaction):     'transaction' '.' ('hash' | 'value' | 'from' | 'to');
```

Figure 5. Text structure 2 based on DSL rules
图 5. 基于 DSL 规则的文本结构 2

以图 5 中的加法为例，其中 x 和 y 分别代表参与加法运算的两个参数。x 和 y 可以是算数表达式、源、目的、执行或者一个整数。源和目的则是指令对应的步数(step)，而执行里包含了自定义的参数，例如，用 src.depth 则可以表述源指令所处的深度。在不等于的规则里，它对应的参数 x 的类型多出了一个字符串类型的比较。在数据流依赖的规则里，源和目的需要用关系符号进行连接，同时可以使用 where，指定需要满足的其他条件。交易里定义了交易的基本属性，包括交易的哈希、数值、来源、目的。

基于上述的规则，Robindetect 在对漏洞进行检测时，会优先提取源指令和目的指令之间的依赖关系。在提取符合预定的指令关系后，还会根据条件进行进一步的判断，并将符合条件的指令对存入条件数组，接着再以当前目的指令作为新的源指令，重复上述步骤，直至提取到符合完整规则的指令序列，最终触

发检测预警。

需要注意的是，对于指令的依赖关系我们将其分为三类，调用流关系(⇒)、数据流关系(~>)以及跟随流关系(->)。调用流依赖和数据流依赖分别由调用流提取器和数据流提取器提取，而跟随流依赖则是指一条指令在另一条指令之后执行，两条指令之间并非要求具有强依赖性。跟随流依赖本质是对漏洞检测规则的一种补充，是一种弱依赖性，它更多依赖于研究人员的经验分析。参考图 3 中的“718 (MSTORE) --> 863 (JUMP)”，两个指令间是跟随流关系，因为我们希望在提取指令序列时，能够兼容一条指向特别的指令的序列进行辅助分析，例如 SELFDESTRUCT、SUICIDE。

4. 实验设计

4.1. 已知漏洞检测规则的提取实验

本节通过与现有工具 Aegis 的对比，验证了 RobinDetect 在已知漏洞检测规则提取方面的准确性和有效性。我们选取了 Aegis 提供的 Parity Wallet Hack 2 漏洞检测规则作为对比对象。其中 Aegis 提供的 Parity Wallet Hack 2 的漏洞检测规则如下图 6 所示：

Parity Wallet Hack 2
Aegis 提供的规则
(opcode = CALLDATACOPY) ~~> (opcode = SSTORE) ~~> (opcode = JUMPI) where (src.transaction.hash != dst.transaction.hash)
--> ((opcode = CALLDATALOAD) ~~> (opcode = SELFDESTRUCT))

测试出现的问题
(opcode = CALLDATACOPY) ~~> (opcode = SSTORE) ~~> (opcode = JUMPI) where (src.transaction.hash != dst.transaction.hash)
--> ((opcode = CALLDATALOAD) ~~> (opcode = SELFDESTRUCT))

Figure 6. Vulnerability detection rules provided by Aegis and their problems
图 6. Aegis 提供的漏洞检测规则及其问题

Aegis 的检测规则要求 CALLDATACOPY、SSTORE 和 JUMPI 分别具有控制流关系，且 JUMPI 指令要满足条件指向新的交易 hash，即 JUMPI 是新的交易中的指令，后面要出现的指令 CALLDATALOAD 和 SELFDESTRUCT 也要具有数据流关系。这里可以参考图 3 中 Transcation_3，也是 Aegis 提供的问题交易组。然而在实际测试中，我们发现 Aegis 并不能正确检测漏洞 Parity Wallet Hack 2，因为在 Transcation_3 中并不存在既满足 CALLDATACOPY 到 SSTORE 的数据流关系，又满足跨越合约的 SSTORE 到 JUMPI 的数据流依赖关系，这导致在 Aegis 并不能正确的从 Transcation_3 中检测到漏洞。

Parity Wallet Hack 2
新提取的规则
(opcode = CALLDATALOAD) ~~> (opcode = JUMPI) ==> (opcode = ISZERO) ~~> (opcode = JUMPI) ==> (opcode = CALLDATALOAD)
~~> (opcode = CALLDATALOAD) ~~> (opcode = MSTORE) ~~> (opcode = ISZERO) ~~> (opcode = ISZERO) ~~> (opcode = JUMPI)
==> (opcode = MSTORE) ~~> (opcode = SSTORE) ~~> (opcode = SLOAD) ~~> (opcode = ISZERO) ~~> (opcode = ISZERO)
~~> (opcode = JUMPI) ==> (opcode = SLOAD) ~~> (opcode = SSTORE) ~~> (opcode = SLOAD) ~~> (opcode = JUMPI)
~~> (opcode = MSTORE) --> (opcode = JUMP) ==> (opcode = SELFDESTRUCT)

优化后的规则
(opcode = CALLDATALOAD) ~~> (opcode = JUMPI) ==> (opcode = MSTORE) ~~> (opcode = SSTORE)
~~> (opcode = SLOAD) where (src.transaction.hash != dst.transaction.hash) ~~> (opcode = JUMPI) ==> (opcode = SLOAD)
~~> (opcode = SSTORE) ~~> (opcode = JUMPI) ==> (opcode = MSTORE) --> (opcode = JUMP) ==> (opcode = SELFDESTRUCT)

Figure 7. Vulnerability rules extracted by Robindetect and their optimization
图 7. Robindetect 提取的漏洞规则及其优化

表 1 详细展示了 $Transcation_3$ 提取的指令序列，其中 SSTORE 到 SLOAD 具有流依赖关系，其 Gas 值的巨大差异明显地凸显出是两个不同交易中地指令。参考图 6 中从 $Transcation_3$ 提取的指令序列及其依赖关系，我们将重新定义 Parity Wallet Hack 2 的漏洞检测规则，最终优化的规则如图 7 所示。

Table 1. Sequence of instructions extracted from the combined transactions of Parity Wallet Hack 2
表 1. 从 Parity Wallet Hack 2 的组合交易中提取的指令序列

$Transcation_3$ [0, 875]					
Instruction Sequence 1					
Step	**PC**	**Operation**	**Gas**	**GasCost**	**Depth**
12	23	CALLDATALOAD	117273	3	1
109	234	JUMPI	116854	10	1
112	1007	ISZERO	116841	3	1
114	1011	JUMPI	116835	10	1
120	1021	CALLDATALOAD	116812	3	1
128	1030	CALLDATALOAD	116788	3	1
153	1060	MSTORE	116711	6	1
354	4024	ISZERO	10292	3	1
355	4025	ISZERO	10289	3	1
357	4029	JUMPI	10283	10	1
379	4058	MSTORE	10153	3	1
386	**4067**	**SSTORE**	**10093**	**5000**	**1**
593	**4476**	**SLOAD**	**84481**	**200**	**1**
597	4480	ISZERO	84272	3	1
598	4481	ISZERO	84269	3	1
600	4485	JUMPI	84263	10	1
636	4536	SLOAD	73717	200	1
650	5722	SSTORE	73278	20000	1
707	4561	SLOAD	48066	200	1
714	4570	JUMPI	47848	10	1
718	4576	MSTORE	47831	3	1
Instruction Sequence 2					
Step	**PC**	**Operation**	**Gas**	**GasCost**	**Depth**
863	4812	JUMP	5103	8	1
875	4144	SELFDESTRUCT	5000	5000	1

4.2. 未知漏洞规则的提取实验

为了验证 RobinDetect 在未知漏洞检测规则提取方面的有效性，我们选取了 Xscope 检测到的跨链桥漏洞进行实验。Xscope 作为一种基于自定义跨链交易行为的检测工具，其检测结果具有一定的独特性。我们以 QubitBridge 的交易为例，对其进行了详细的漏洞规则提取实验。

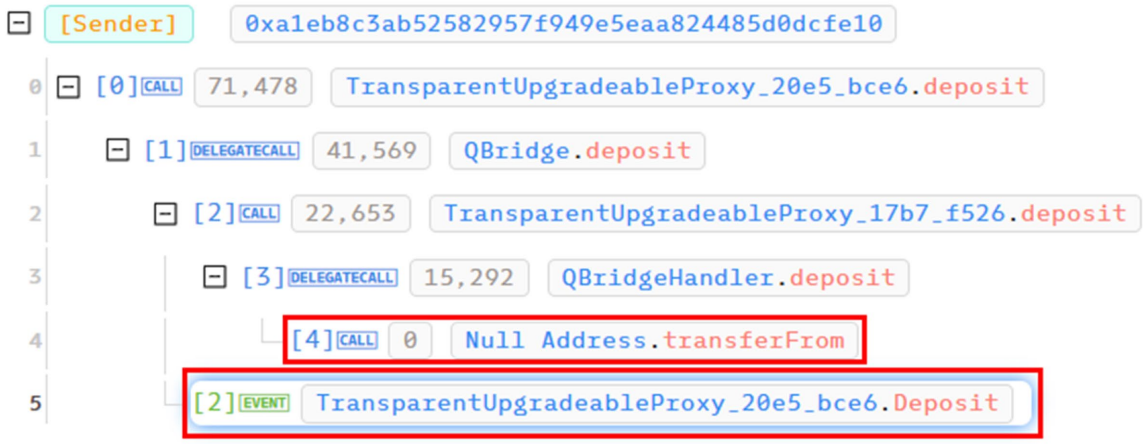

Figure 8. Command call analysis for Phalcon Explorer
图 8. Phalcon Explorer 的指令调用分析

在实验过程中，我们还使用 Phalcon Explorer [12]进行辅助分析和参考。如图 8 所示，QubitBridge 的交易行为在执行过程中发生多次调用，且调用的指令深度不断增大，而最终触发了一个深度为 2 的事件。我们提取的交易指令序列如表 2 所示，需要注意的是这里只展示了部分指令序列，以便观察和对比。其中，Relation 是当前 step 的指令与下一个 step 的指令之间的关系，而 ExecResult 特指 ISZERO 指令执行后的结果，为 0 的话，说明用于判断的结果是 True，若为 1 的话则实际判断结果为 False。

```
// 深度为4的CALL指向的代码行
function safeTransferFrom(
...
) internal {
    ...
    require(success && (data.length == 0 || abi.decode(data, (bool))),
"!safeTransferFrom");
}

// 深度为2的EVENT触发时的代码行
function deposit(uint8 destinationDomainID, bytes32 resourceID, bytes calldata
data) external payable notPaused {
    ...
    emit Deposit(destinationDomainID, resourceID, depositNonce, msg.sender,
data);
}
```

Figure 9. The line of contract code that corresponds to the instruction
图 9. 指令对应所在的合约代码行

我们先筛选出 COST 较高的指令作为提取规则的一部分，它们分别是深度为 1 的 DELEGATECALL、深度为 2 的 CALL、深度为 3 的 DELEGATECALL 以及深度为 4 的 CALL。由于这几个指令之间的其他

指令都有不同的依赖关系，因而这四个指令用“-->”关系链接。参考图 8 的指令调用分析，正好能与这四个指令对应上。

在这个基础上，我们还重点关注了深度为 4 的 CALL 所在行代码的运行逻辑。如图 9 所示，“require”用于验证条件是否成立，若失败则交易回滚，并返回错误信息。该笔交易并没有在此处停下表明“require”的验证条件成立了。ISZERO 经常用于判断条件是否成立，若执行后返回为 1，则表明条件失败，反之若返回为 0，则条件成立，因此我们需要选择一条“ExecResult 值”为 0 的 ISZERO。

再者，深度 2 的 EVENT 的触发时，“emit”会将事件及其参数记录到交易日志之中，以便这些日志能被监听和解析，而记录日志则需要 LOG 及其相关的指令，例如：LOG2、LOG4D 等。这里我们在表 2 中提取到了 LOG2 指令，由于 LOG2 和 ISZERO 之间同样存在着不同依赖关系的指令，因此同样用“-->”进行链接。通过综合考虑这些指令的执行顺序及其依赖关系，我们最终提取出了 QubitBridge 漏洞的检测规则，如图 10 所示。该规则不仅能够准确地识别出 QubitBridge 漏洞的特征，还能够为类似跨链桥漏洞的检测提供参考。

Table 2. Sequence of instructions extracted from QubitBridge transactions
表 2. 从 QubitBridge 的交易中提取的指令序列

QubitBridge Transcation: 0x5af141b2c19cc8cb77a5583654575a6811664ebc304b75e687a6e356b7dd7cf7							
Step	**PC**	**Operation**	**Gas**	**GasCost**	**Depth**	**Relation**	**ExecResult**
...	...	...	...	...	...	...	...
105	926	SLOAD	140786	2100	1	~~>	/
135	**867**	**DELEGATECALL**	**138556**	**136432**	**1**	==>	**/**
145	15	CALLDATALOAD	133790	3	2	~~>	/
...	...	...	...	...	...	...	...
518	2179	GAS	118594	2	2	~~>	/
519	**2180**	**CALL**	**118592**	**116741**	**2**	-->	**/**
575	926	SLOAD	116416	2100	3	~~>	/
601	703	ISZERO	114240	3	3	~~>	0 × 1
603	707	JUMPI	114234	10	3	==>	/
625	926	SLOAD	114148	2100	3	~~>	/
655	**867**	**DELEGATECALL**	**111918**	**110210**	**3**	==>	**/**
665	15	CALLDATALOAD	107568	3	4	~~>	/
...	...	...	...	...	...	...	...
1174	5511	JUMPI	95196	10	4	==>	/
1186	5542	MSTORE	95148	3	4	~~>	/
1209	**3862**	**CALL**	**95082**	**93637**	**4**	~~>	**/**
1232	**3922**	**ISZERO**	**92420**	**3**	**4**	~~>	**0 × 0**
1234	3926	JUMPI	92414	10	4	-->	/
...	...	...	...	...	...	...	...
1356	11093	CALLDATACOPY	95593	12	2	~~>	/
1400	**2274**	**LOG2**	**95457**	**3173**	**2**	/	**/**

QubitBridge
优化后的规则

```
( opcode = DELEGATECALL ) --> ( opcode = CALL ) where ( src.depth == 1 && dst.depth == 2 )
 --> (( opcode = DELEGATECALL ) --> ( opcode = CALL ) where ( src.depth == 3 && dst.depth == 4 )
 ~~> (( opcode = ISZERO ) where ( dst.executeResult == 0x0 )
 -->  (( opcode = LOG2 ) where ( dst.depth == 2 ))))
```

Figure 10. Vulnerability rules for QubitBridge
图 10. QubitBridge 的漏洞规则

5. 总结

本文聚焦于区块链技术中智能合约的安全性问题，尤其是跨链桥漏洞检测这一关键难点，提出了一种名为 RobinDetect 的字节码级漏洞规则提取工具。该工具通过字节码级漏洞规则提取和多组件协同工作，能够高效、准确地提取漏洞规则，实现对跨链桥漏洞的高效识别。RobinDetect 的主要特点在于其灵活和轻便的设计，这使得它在处理复杂的跨链桥漏洞检测任务时表现出显著的优势。然而，RobinDetect 的检测能力在一定程度上依赖于对特定指令的检测，面对不断更新的智能合约，要求使用人员对新型合约具有一定的敏感性，并能够及时补充新的指令，以确保工具的有效性和准确性。未来，我们将继续优化 RobinDetect，探索使用机器学习等先进技术，进一步提高安全分析人员的工作效率，为区块链技术的安全发展贡献更多的力量。

参考文献

[1] Guo, L., Chen, J., Li, S., Li, Y. and Lu, J. (2022) A Blockchain and IoT-Based Lightweight Framework for Enabling Information Transparency in Supply Chain Finance. *Digital Communications and Networks*, **8**, 576-587. https://doi.org/10.1016/j.dcan.2022.03.020

[2] Rejeb, A., Keogh, J.G. and Treiblmaier, H. (2019) Leveraging the Internet of Things and Blockchain Technology in Supply Chain Management. *Future Internet*, **11**, Article 161. https://doi.org/10.3390/fi11070161

[3] Ou, W., Huang, S., Zheng, J., Zhang, Q., Zeng, G. and Han, W. (2022) An Overview on Cross-Chain: Mechanism, Platforms, Challenges and Advances. *Computer Networks*, **218**, Article 109378. https://doi.org/10.1016/j.comnet.2022.109378

[4] Lan, R., Upadhyaya, G., Tse, S., *et al*. (2021) Horizon: A Gas-Efficient, Trustless Bridge for Cross-Chain Transactions.

[5] (2021) pNetwork. https://medium.com/pnetwork/pnetwork-post-mortem-pbtc-on-bsc-exploit-170890c58d5f

[6] (2021) Thorchain. https://medium.com/thorchain/eth-parsing-error-and-exploit-3b343aa6466f

[7] Tang, Q., Sang, N. and Liu, H. (2020) Learning Nonclassical Receptive Field Modulation for Contour Detection. *IEEE Transactions on Image Processing*, **29**, 1192-1203. https://doi.org/10.1109/tip.2019.2940690

[8] Wang, S. and Zhao, X. (2024) Contractsentry: A Static Analysis Tool for Smart Contract Vulnerability Detection. *Automated Software Engineering*, **32**, Article No. 1. https://doi.org/10.1007/s10515-024-00471-8

[9] Liao, Z., Nan, Y., Liang, H., Hao, S., Zhai, J., Wu, J., *et al*. (2024) Smartaxe: Detecting Cross-Chain Vulnerabilities in Bridge Smart Contracts via Fine-Grained Static Analysis. *Proceedings of the ACM on Software Engineering*, **1**, 249-270. https://doi.org/10.1145/3643738

[10] Ferreira Torres, C., Baden, M., Norvill, R., Fiz Pontiveros, B.B., Jonker, H. and Mauw, S. (2020) ÆGIS: Shielding Vulnerable Smart Contracts against Attacks. *Proceedings of the* 15*th ACM Asia Conference on Computer and Communications Security*, Taipei, 5-9 October 2020, 584-597. https://doi.org/10.1145/3320269.3384756

[11] Zhang, J., Gao, J., Li, Y., Chen, Z., Guan, Z. and Chen, Z. (2022) Xscope: Hunting for Cross-Chain Bridge Attacks. *Proceedings of the* 37*th IEEE/ACM International Conference on Automated Software Engineering*, Rochester, 10-14 October 2022, 1-4. https://doi.org/10.1145/3551349.3559520

[12] (2025) Phalcon Explorer. https://app.blocksec.com/