RESEARCH ON VULNERABILITY DETECTION METHODS FOR ETHEREUM SMART CONTRACTS

A Thesis Submitted to

Southeast University

For the Professional Degree of Master of Engineering

BY

Supervised by

and

School of Cyber Science and Engineering

Southeast University

May 2022

摘　要

随着以太坊的快速发展，在该平台上部署的智能合约越来越多，而近年来不断曝出的智能合约漏洞致使以太坊投资者蒙受巨大的经济损失。智能合约一旦部署上链就无法修改，这种特性使得合约部署前的漏洞检测成为保障其安全的重要手段。现有的检测方法仍存在一系列的问题：传统方法依赖于专家规则，自动化程度较低且难以应用到大规模检测；基于机器学习的方法未能充分利用智能合约丰富的结构信息，导致检测结果的漏报率与误报率较高。为解决现有研究工作中难以利用智能合约中结构信息的问题，本文构建了一种能够表征智能合约的图中间表示形式，并将其与图神经网络技术相结合，提出了基于图匹配神经网络的智能合约漏洞检测方法。本文主要研究工作如下：

（1）为检测可获得源代码的智能合约中存在的漏洞，本文提出了一种面向智能合约源代码的图匹配神经网络漏洞检测方法。为利用智能合约源代码中的语法与语义信息，本文构建了一种能够充分表征智能合约源代码语法和语义信息的抽象语义图结构。首先，从源代码解析出抽象语法树，在抽象语法树的基础上构建控制流图和数据流图；其次，将抽象语法树、控制流图和数据流图结合生成抽象语义图；最后，为了充分使用抽象语义图中结构信息并实现自动化漏洞检测，使用自然语言模型将抽象语义图转化为图结构向量，并采用图匹配神经网络对智能合约进行检测。实验结果表明，本文提出的面向源代码的智能合约漏洞检测方法相比于现有面向源代码的漏洞检测方法（Slither、SmartCheck 和 DR-GCN）F1 分数平均提高了 10% 到 15%。

（2）以太坊中智能合约也会以字节码形式发布，本文针对此将上述方法进一步扩展到面向字节码的漏洞检测场景。为利用智能合约字节码中丰富的结构信息，本文基于字节码特性提出了一种表征智能合约字节码的抽象语义图结构。首先，将字节码反汇编为操作码，并根据操作码指令的语义将操作码划分成基本块；其次，模拟以太坊虚拟机执行操作码来构建基本块之间的控制流，并对操作码进行语义上的规范化从而生成面向字节码的抽象语义图；最后，采用图匹配神经网络来检测智能合约字节码中存在的漏洞。实验结果表明，本文提出的面向字节码的智能合约漏洞检测方法相较于现有面向字节码的漏洞检测方法（Oyente、Securify 和 ContractFuzzer）F1 分数平均提升 14% 到 20%。

（3）在提出的面向源代码和字节码的图匹配神经网络漏洞检测方法的基础上，本文设计并实现了一个可视化的智能合约漏洞检测系统，对系统的整体架构进行了介绍，并对系统各个模块划分和相关重要算法进行了说明。为了评估本文构建的智能合约漏洞检测系统在真实智能合约环境中漏洞检测效果，对部署在以太坊的智能合约进行了检测，并且对检测结果进行了人工分析，实验结果表明本文提出的漏洞检测系统能够提升 9% 到 20% 的漏洞检测准确率。

关键词： 以太坊，智能合约，漏洞检测，图神经网络，抽象语义图

Abstract

With the rapid development of Ethereum, more and more smart contracts are deployed based on Ethereum, and in recent years, the continuous exposure of smart contract vulnerabilities has caused huge financial losses to Ethereum investors. Once a smart contract is deployed on the chain, it cannot be modified, and this feature makes vulnerability detection before deployment be an important tool to ensure its security. Existing methods for detecting vulnerabilities in smart contracts have various problems: traditional methods rely on expert rules, which are less automated and difficult to apply to large-scale detection; machine learning-based methods do not make full use of the rich structural information of smart contracts, resulting in a high false negative report and false positive report in detection results. In order to solve the difficulty in using structural information in smart contracts, this paper constructs a graphic intermediate representation that can characterize smart contracts and combines it with graph neural network technology to propose a smart contract vulnerability detection method based on graph matching neural network. The main research work in this paper is as follows.

(1) In order to detect vulnerabilities in the source code of smart contracts, this paper proposes a graph matching neural network vulnerability detection method for smart contract source code. To exploit the syntactic and semantic information in the source code of smart contracts, an abstract semantic graph structure that can fully characterize the syntactic and semantic information of the source code of smart contracts is constructed. Firstly, the abstract syntax tree is parsed from the source code, and the control flow graph and data flow graph are constructed on the basis of the abstract syntax tree; secondly, the abstract syntax tree, control flow graph and data flow graph are combined to generate the abstract semantic graph; finally, in order to make full use of the structural information in the abstract semantic graph and to automate vulnerability detection, the abstract semantic graph is transformed into a graph structure vector using a natural language model, and a graph matching neural network is used to smart contracts. The experimental results show that the source code oriented smart contract vulnerability detection method proposed in this paper improves the F1 score by 10% to 15% on average compared with the existing source code oriented vulnerability detection methods (Slither, SmartCheck and DR-GCN).

(2) Smart contracts in Ethernet are also released in bytecode form, and this paper further extends the above approach to bytecode-oriented vulnerability detection scenarios. In order to exploit the rich structural information in the bytecode of smart contracts, this paper proposes an abstract semantic graph structure to characterize the bytecode of smart contracts based on the

characteristics of bytecode. Firstly, the bytecode is decompiled into opcodes, and the opcodes are divided into basic blocks according to the semantics of the opcode instructions; secondly, an Ethernet virtual machine is simulated to execute the opcodes to construct the control flow between the basic blocks, and the opcodes are semantically normalized to generate a bytecode-oriented abstract semantic graph; finally, a graph matching neural network is used to detect the vulnerabilities in the smart contract bytecodes. The experimental results show that the bytecode-oriented smart contract vulnerability detection method proposed in this paper improves the F1 score by 14% to 20% on average compared with the existing bytecode-oriented vulnerability detection methods (Oyente, Securify and ContractFuzzer).

(3) Based on the proposed bytecode- and source-code oriented graph matching neural network vulnerability detection method, this paper designs and implements a visual smart contract vulnerability detection system. The overall architecture of the system is described in this paper, and the division of each module of the system and related important algorithms are explained. In order to evaluate the vulnerability detection effect of the smart contract vulnerability detection system built in this paper in a real smart contract environment, the smart contracts deployed in Ethereum were tested and the results were analysed manually.

Keywords: Ethereum，Smart Contract, Vulnerability Detection, Graph Neural Network，Abstract Semantic Graph

目　录

插图目录

表格目录

算法目录

第一章 绪论

本章首先介绍了本课题的研究背景与研究意义，探讨了当前基于以太坊环境下的智能合约现状及其存在的问题，接着分别从传统代码审计技术和基于深度学习技术两个方面阐述了智能合约漏洞检测的国内外研究现状，并指出现有研究工作的不足，最后介绍了本文的主要研究工作和内容组织结构安排。

1.1 研究背景与意义

1.1.1 智能合约概念

上世纪九十年代 Szabo[1] 最早提出了智能合约的概念：智能合约是一种采用电子数据定义的共识，在这个共识之下所有的参与方会按照事先约定好的承诺来执行事务。Szabo 认为在这种约定的协议下能够使用一种计算机程序来完成现实中复杂的交易，然而当时缺少一种不依赖第三方就达成可信状态的执行环境，故而智能合约难以得到真实的应用。直到 2008 年中本聪发表的论文[2] 中提出了区块链概念，随后的 2014 年 Vitalik Buterin 受到区块链的启发，提出了一种全新的区块链平台——以太坊[3]，并将智能合约引入了区块链。由此，智能合约为区块链提供了除金融货币以外的另一种应用方式，并且也使得对区块链底层数据进行编程的操作成为可能。以太坊作为应用区块链技术的平台之一，不仅提供了数字货币的功能，还允许用户编写智能合约来管理区块链的底层数据，极大增强了区块链的应用范围，使得各种不同类型的应用运行在区块链上成为可能[4]。智能合约具有以下四个特点：（1）由交易触发，不需要人工交互；（2）智能合约一旦被启动，无法阻止其执行；（3）区块链网络中的每个节点都知道智能合约，因为智能合约的正确性必须经过大多数节点的验证；（4）可以根据不同的场景需求进行调整[5]。

随着时间的推移，智能合约的易用性受到了各行各业的青睐，智能合约开始在金融、管理、医疗、物联网和供应链等应用大放异彩。智能合约与金融行业天然就有着很高的契合度，智能合约的不可篡改、公开透明和去中心化特性能够避免引入强势第三方，点对点交易的特性可以简化交易流程，区块链提供的分布式账本为金融监管机构对所有交易行为的追溯和调查提供了便利，这些特性均可以保证投资者的资金安全[6]。此外，智能合约在管理领域可以说是方兴未艾。智能合约的可追溯特性使得其与选举投票等活动具有极高的适配性，在数字资产的版权管理方面也有着很好的应用，并且在具体的业务流程管理方面也有着广阔的应用前景[7]。智能合约也能助力医疗领域，其中一大应用就是电子病历。不同医生需要对患者的治疗过程进行回溯从而更好地实施诊疗，而智能合约能够为这种电子病历设置访问权限，使得医疗从业者对于某个病患的数据有着更加细粒度的访问权限，无需担心医疗记录被泄露与篡改[8]。物联网与具有去中心化特性的智

能合约结合也是一大发展趋势，智能合约能够解决物联网中复杂流程的这一痛点，为促进物联网中的资源共享提供了解决方案，从而提升行业的效率并保障信息安全，降低整体的应用成本[9]。

1.1.2 智能合约安全现状的研究

随着以太坊应用的普及，其所承载的金融价值也随之增长，由此引发的安全问题也日益突出。为了系统的稳健运营，以太坊的初始设计引入了一系列安全措施[10]：为保证以太坊中的数据不被轻易修改，使用了密码学技术对数据进行加密和校验；为避免因为中心化的主机数据丢失和篡改，使用了去中心化的点对点网络和共识算法；为使得以太坊中的程序隔离运行从而保障以太坊本身的安全，采用了虚拟机技术创建一个单独的沙盒环境提供给智能合约执行，从而限制了恶意智能合约的影响范围。智能合约本质上是一个在对等网络中自动执行的程序，所以与其他编程语言一样，它会受开发者个人水平和编程语言自身缺陷的影响，从而引入各种各样的漏洞。与其他程序不同的是，智能合约的执行环境通常直接承载着金融价值，对于那些有着经济利益诉求的别有用心者来说蕴含着巨大的吸引力。一旦智能合约中存在的漏洞遭到利用，将使得其他用户处于不公平的地位甚至直接蒙受经济损失。此外，针对智能合约成功发起攻击并获取到经济利益的案例，特别是那些在以太坊社区中被广泛宣传的案例，会直接降低用户对于智能合约和以太坊的信任，进而影响智能合约更为广泛的应用。

近年来出现了诸多安全事件，如：2016 年 6 月，攻击者利用了分散自治组织 (Decentralized Autonomous Organization，DAO) 上存在的重入漏洞[11] 成功窃取了近 360 万个的以太坊货币，给投资者造成了近 3 亿人民币的损失；2017 年 7 月，Parity 多签名钱包分别发生了两起安全漏洞事件，分别造成了 2 亿人民币和 9 亿人民币的经济损失[12]；2018 年 4 月，BEC 智能合约中对用户进行转账操作的函数存在整数溢出漏洞[13]，黑客利用此漏洞创建了原本不应该存在的以太币，事后统计这次攻击事件造成了近 60 亿人民币的损失。在 DeFi 热潮的推波下，大量新项目与资金进入该市场，仅 2020 年 11 月就发生超过 13 起 DeFi 的智能合约漏洞事件[14]，损失估计约为 4 亿人民币。据不完全统计，截至 2020 年 12 月，区块链上发生的引起经济损失的重大安全事件至少发生了 358 起，造成了巨量的经济损失[15]。由此可见，智能合约的应用与发展有着十分急切的安全需求，面向智能合约的漏洞检测技术作为防范以太坊攻击者的直接手段显得尤为迫切。

1.1.3 本论文的研究意义

区块链技术一直是国内外央行和金融机构关注的焦点，智能合约作为区块链应用的重要组成部分，它在各领域的应用使得自动化流程和支付变得更加便捷，进而提高全球交易市场的效率。根据 Verified Market Research[16] 的数据，2020 年全球智能合约市场规模为 1.495 亿美元，预计到 2028 年将达到 7.7052 亿美元，从 2021 年到 2028 年将以 24.55 % 的复合年增长率增长。然而智能合约的安全问题始终困扰着全球，Durieux 等

人[17] 使用 9 种目前最先进的自动化分析工具对智能合约进行大规模分析，在评估相同类别的智能合约漏洞时，发现检测工具的结果存在较大差异，并且每种漏洞检测工具都存在着相当大的漏报率，其中只有 42% 包含漏洞的智能合约能够被所有检测工具都识别。同时，在分析真实世界的合约数据集时，有 97% 智能合约被这些检测工具标记为漏洞合约，表明结果中存在大量的假阳性。由此可见，现有的智能合约检测技术在效率、规模和扩展性方面仍难以满足合约飞速上涨的审计需要。

传统的人工检测漏洞方式看起来像一门费力不讨好的艺术，它需要有经验的安全专家仔细检查每一个合约的漏洞，效率低下。随着人工智能和机器学习的飞速发展，机器学习在分类识别的问题上展现出了极强的表现力[18]。在计算机软件安全领域，目前已经有众多研究人员开始将机器学习技术应用到漏洞挖掘和分析之中。在有着庞大输入数据集的情况下，相比于传统的形式化验证、符号执行和模糊测试等方法，机器学习在提升执行效率和降低漏洞检测成本方面有着显著的优势。此外，机器学习技术具有自动学习的能力，也就是说如果能够构建一个良好的模型，就可以让模型学习到智能合约漏洞的特征，从而使得智能合约漏洞检测摆脱对专家规则和人工作业的依赖[19]。然而，当前基于机器学习的智能合约漏洞检测技术并不完善，相较于更加得心应手的分类处理、自然语言处理等领域，面向智能合约漏洞检测的技术首当其冲的问题就是如何预处理智能合约和构建适合智能合约漏洞检测的模型。不同于图片的二维向量数据和文本的序列化数据，智能合约的源代码和字节码有着更加丰富的结构信息，这些结构信息在空间上可以看作一种非欧几里德数据[20]。现有基于机器学习的智能合约漏洞检测方案大多是将智能合约看作是文本序列，然后采用自然语言处理中较为常用的循环神经网络及其变种来提取漏洞特征，而这会使得智能合约中大量的语法和语义信息丢失，进而导致漏洞检测的效果难以令人满意。如何提取智能合约中的结构特征并最大限度地保留合约语义信息是漏洞检测的关键问题。图神经网络作为处理非欧几里得数据的一种特殊深度学习网络，可以最大限度地承载智能合约语法和语义信息，为解决现有工作中存在的局限性提供一种可能。因此，本文基于图神经网络技术开展智能合约漏洞检测研究具有较高的探索意义。

1.2　国内外研究现状

自智能合约应用于以太坊以来，不少专家学者对智能合约开展了安全研究。现有工作聚焦于应用软件漏洞检测领域已有的方法对智能合约来进行漏洞挖掘。作为一种新颖的去中心化应用程序，相较传统的软件程序，智能合约在程序运行环境、合约生存周期和软件特性上有着显著的不同，这为智能合约的漏洞检测工作带来了新的挑战。近几年发展出了多种针对智能合约的自动化漏洞检测方法，依据技术特性可以分为模糊测试、符号执行、形式化验证和静态分析等。

1.2.1 模糊测试

模糊测试（Fuzzing）是一种常用的动态漏洞检测技术。模糊测试在软件正常执行过程中为程序随机生成大量正常和异常的输入，通过对程序状态和行为进行监控，来检测软件漏洞[21]。模糊测试技术的关键就在于如何生成不同类型的随机输入，目前主要有两种方式来处理这个问题：基于生成和基于变异。基于生成的方式适用于输入格式比较严格的软件程序，根据一定格式生成输入用例；基于变异的方式是在初始输入的基础上依据程序执行过程反馈来对初始输入进行修改，从而生成不同的输入用例。模糊测试相比较于其他技术有着较好的伸缩性与易用性，能够在无法获得软件源代码的黑盒情况下进行漏洞挖掘。模糊测试技术在传统的软件漏洞检测领域被广泛使用，其有效性已经得到了验证。比如，面向 C++ 程序软件漏洞挖掘的 AFL[22] 就是一种基于变异的模糊测试工具，它通过覆盖率来调节输入的变异方向，使得测试用例能够尽可能地覆盖程序的所有分支，执行被中断的地方就是可能引发程序崩溃的位置。

ContractFuzzer[23] 是面向以太坊智能合约安全漏洞检测的模糊测试工具之一，主要由 EVM 虚拟机插桩工具和模糊测试工具组成，图 1-1 是 ContractFuzzer 的基本测试流程和整体架构。ContractFuzzer 首先通过分析智能合约的应用程序二进制接口来提取智能合约函数各个参数的数据类型，再利用提取到的信息生成符合智能合约二进制接口规范的输入用例。在执行过程中会记录智能合约状态，通过分析执行过程的日志来检测当前的智能合约是不是存在漏洞。ContractFuzzer 能够检测 gas 耗尽、重入漏洞、区块信息依赖、时间戳依赖和以太币冻结等七种类型的智能合约漏洞。

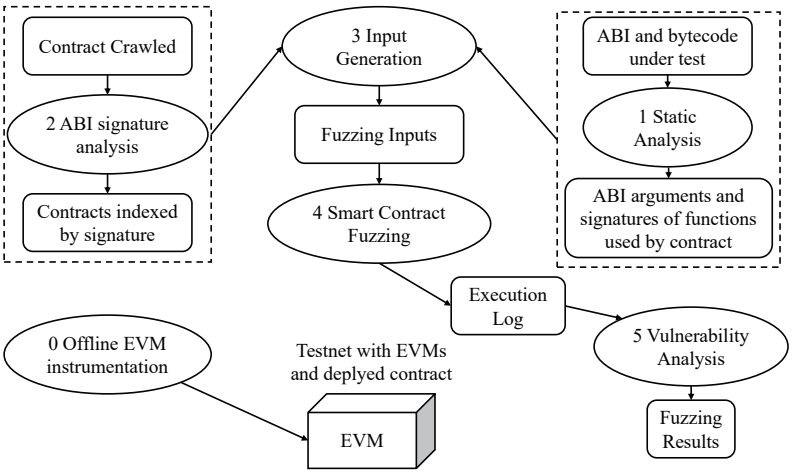


图 1-1 ContractFuzzer 整体架构

EthPloit[24] 生成基本的交易序列来测试智能合约是否可执行，利用动态种子策略解决模糊测试中无解的约束问题，同时将运行时间值作为反馈提高模糊测试的效率，利用传递硬约束的动态种子策略模拟区块链行为来检测是否存在漏洞。

ILF[25] 是一个将神经网络模型与模糊测试相结合的方法，ILF 使用符号执行引擎来生成随机的输入用例，并利用深度学习网络来学习那些可以提高覆盖率的用例特征，从

而提升模糊测试整体覆盖率。

1.2.2　符号执行

符号执行（Symbolic Execution）也是智能合约漏洞检测中的常用方法，主要用于评估程序是否按照预期工作，简单来说就是通过符号值来模拟程序的执行过程，查找那些触发漏洞的确切输入值或者值范围，从而查找程序中潜在的漏洞[26]。智能合约的代码量较小，路径数量相对较少，十分契合符号执行的特点。智能合约漏洞检测中应用符号执行技术的基本过程包括：首先将源代码中的输入符号化，从而将输入值转化为不定的符号向量，然后利用符号执行引擎对程序的所有路径进行求解，在执行过程更新路径中的状态并探索路径中的约束，直到将程序中的全部可执行路径都探索完毕为止。

Oyente[27] 是最早将符号执行技术应用到智能合约漏洞检测的研究工作之一，它实现了一个面向智能合约漏洞检测的符号执行引擎工具，通过模拟以太坊虚拟机来探索该智能合约的全部可执行路径，从而检测合约是否存在安全问题，并向用户输出有问题的符号路径。如图 1-2 所示，Oyente 在整个执行过程中利用 Z3 求解器来确定路径是否可执行，包含控制流图生成器、探索器和分析器等三个核心模块。Oyente 能够检测智能合约时间戳依赖、重入漏洞和未校验返回值等漏洞类型。

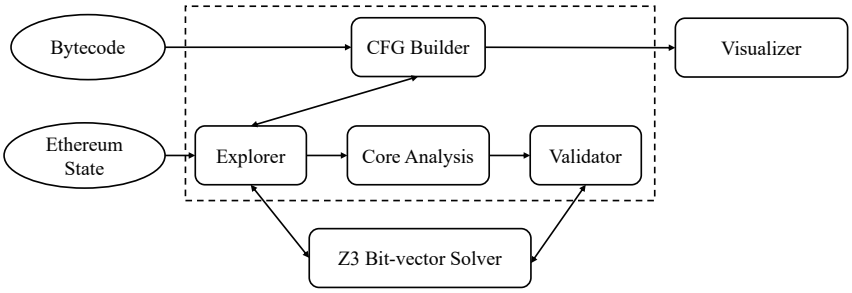


图 1-2 Oyente 整体架构

Mythril[28] 是面向以太坊智能合约的字节码进行漏洞分析的工具，它集混合符号执行、污点分析和控制流检查等技术于一体，支持对重入漏洞、时间戳漏洞、异常处理漏洞、区块信息依赖等多种漏洞类型的检测。

Manticore[29] 是一个动态符号执行引擎工具，它将整个符号执行过程划分为三个状态：准备、执行和终止，三个状态之间能够互相转换；它通过具体化变量值来提升漏洞检测效率和程序覆盖率。

Osiris[30] 在符号执行的技术基础之上结合了污点分析技术，能够跟踪并标记参与运算的变量和运算结果，从而使得数据变量不受无关变量的影响，专注于整数溢出漏洞的检测。

1.2.3 形式化验证

形式化验证方法（Formal Verification）[31] 可以分为模型检测和演绎验证，模型检测是证明特定的规范是否与模型相匹配，也就是应用有限状态模型来验证智能合约是否符合特定的状态，如果与规范相矛盾则可能产生漏洞；演绎证明通过定义一系列的逻辑规则，然后按照数学逻辑来证明智能合约存在着某些特征[32]。

F* 框架[33] 是由 Bhargavan 等人开发的用于程序验证的工具，这种方法需要 F* 的专业知识和对规范的理解。F* 框架首先将智能合约源代码和字节码转换成函数式编程语言 F* 来验证智能合约高级和低级属性，从而检测智能合约中存在的漏洞。

K 框架[34] 也是一种形式化验证框架，它通过执行基于以太坊虚拟机的形式化规范，从而实现了一系列与语义分析相关的工具，比如 Gas 调节工具、语义分析器和可执行路径分析的验证器等。

Securify[35] 是利用语义信息检测智能合约漏洞的形式化安全分析工具，其整体架构如图 1-3 所示。首先从智能合约字节码中提取智能合约的相互依赖关系，进而从智能合约获取精确的语义信息，接着将语义信息使用另一种描述性语言进行表述。最后将这些语义信息与事先定义好的模式进行匹配，验证智能合约是否符合这些预定义的模型，从而判断是否存在安全风险。

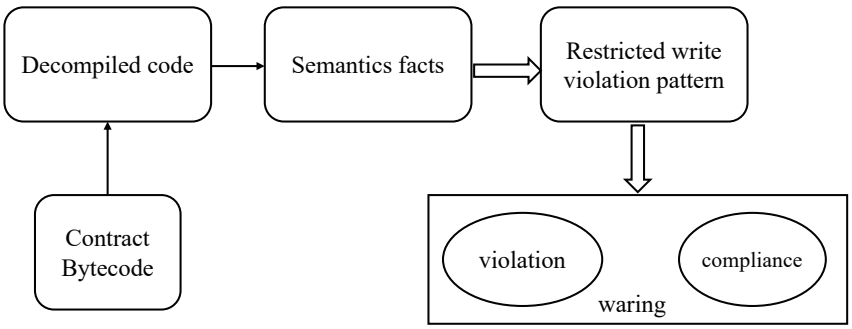


图 1-3 Securify 整体架构

1.2.4 静态分析

静态分析方法是通过分析软件代码，将源代码或者字节码转换为另外一种能够更好表达其语法和语义信息的中间表示形式，并从中提取智能合约的语义特征，进而对智能合约开展漏洞检测的方法。

Slither[36] 是一款面向智能合约源代码检测的静态分析框架，它支持四十多种不同漏洞类型的检测并在漏洞检测的基础上提供了一系列智能合约分析工具。其整体架构如图 1-4 所示，Slither 将智能合约源代码作为初始输入，首先将智能合约源代码编译为抽象语法树，并从抽象语法树中恢复合约的控制流图、继承关系等重要信息，接着将智能合约转换为 SlithIR 这种中间语言形式，通过静态单一评估促进对智能合约的分析，最后使用不同的模块来对智能合约进行漏洞检测。

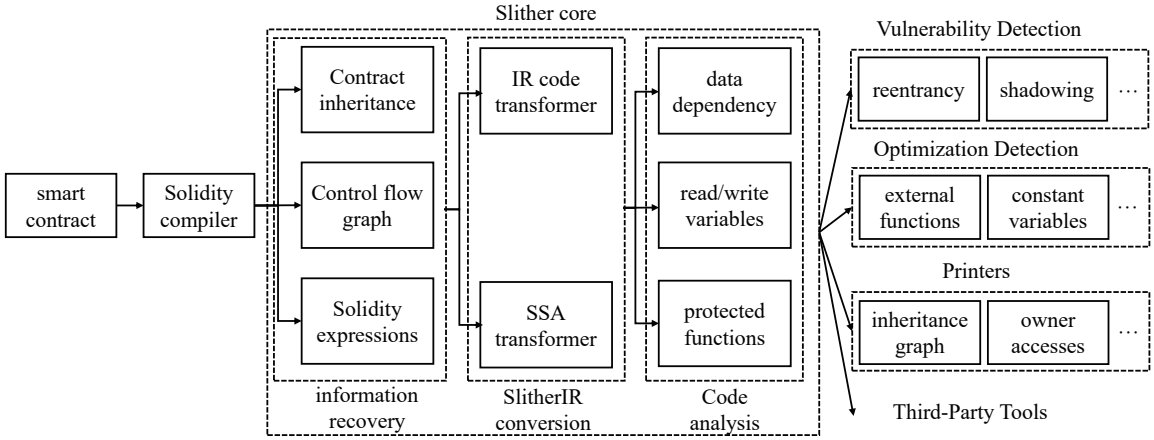


图 1-4 Slither 整体架构

SmartCheck[37] 是一种静态分析工具，将智能合约的源代码转化为基于 XML 的中间表示形式，然后使用预先定义的 XPath 模式来与智能合约生成的中间表示形式进行对比，从而检测智能合约是否存在漏洞。

DR-GCN[38] 是一个利用了图神经网络的智能合约漏洞检测工具。首先，将智能合约源代码转换为一种简单图结构，这种中间表示形式只保留了源代码的很小部分节点和边信息，未能充分利用智能合约的语法和语义信息。DR-GCN 定义了一个无度图神经网络，接着利用图神经网络学习智能合约漏洞的特征，进而实现对智能合约漏洞的检测。

1.3　本文的主要研究工作

通过对现有研究工作进行分析，发现模糊测试方法的漏洞覆盖率低，使得其难以应用到较大规模的漏洞检测中；符号执行方法的路径爆炸问题难以解决；形式化验证过多地依赖人工经验且存在不可达的程序路径；而现有基于机器学习的方法尚未充分利用智能合约中丰富的结构信息，检测效果仍存在进一步提升的空间。针对现有漏洞检测工具效率低，并且难以表征智能合约中语法和语义信息的问题，本文提出了面向智能合约的图神经网络漏洞检测方法，主要研究内容如下：

（1）构建能够表征智能合约源代码和字节码语法与语义特征的抽象语义图表征形式。智能合约源代码生成抽象语义图，首先将源代码转化为抽象语法树，在抽象语法树的基础上构建出控制流图和数据流图，将控制流与数据流结合从而构建智能合约源代码的抽象语义图。智能合约的字节码生成抽象语义图，先将字节码反编译生成操作码，然后模拟以太坊虚拟机执行操作码指令生成基本块之间的控制流，最后对操作码进行规范化从而构建智能合约字节码的抽象语义图。

（2）提出具有注意力机制的图匹配神经网络智能合约漏洞检测方法。将抽象语义图进行词嵌入生成图表示形式的数据，利用图神经网络学习智能合约的漏洞特征，采用图匹配神经网络对四种智能合约漏洞类型进行检测，通过与其他漏洞检测工具的实验比

对，论证本文提出的检测方案的有效性。

（3）设计实现面向智能合约漏洞检测系统。结合本文提出的智能合约源代码和字节码的漏洞检测方法，搭建可视化界面的漏洞检测系统来对近年来真实世界的智能合约进行检测，评估本文提出的方法在真实场景下的有效性。

1.4 论文组织结构

依据各个章节的内容不同，如图 1-5 所示本文一共划分为六个章节，各个章节的主要内容介绍如下：

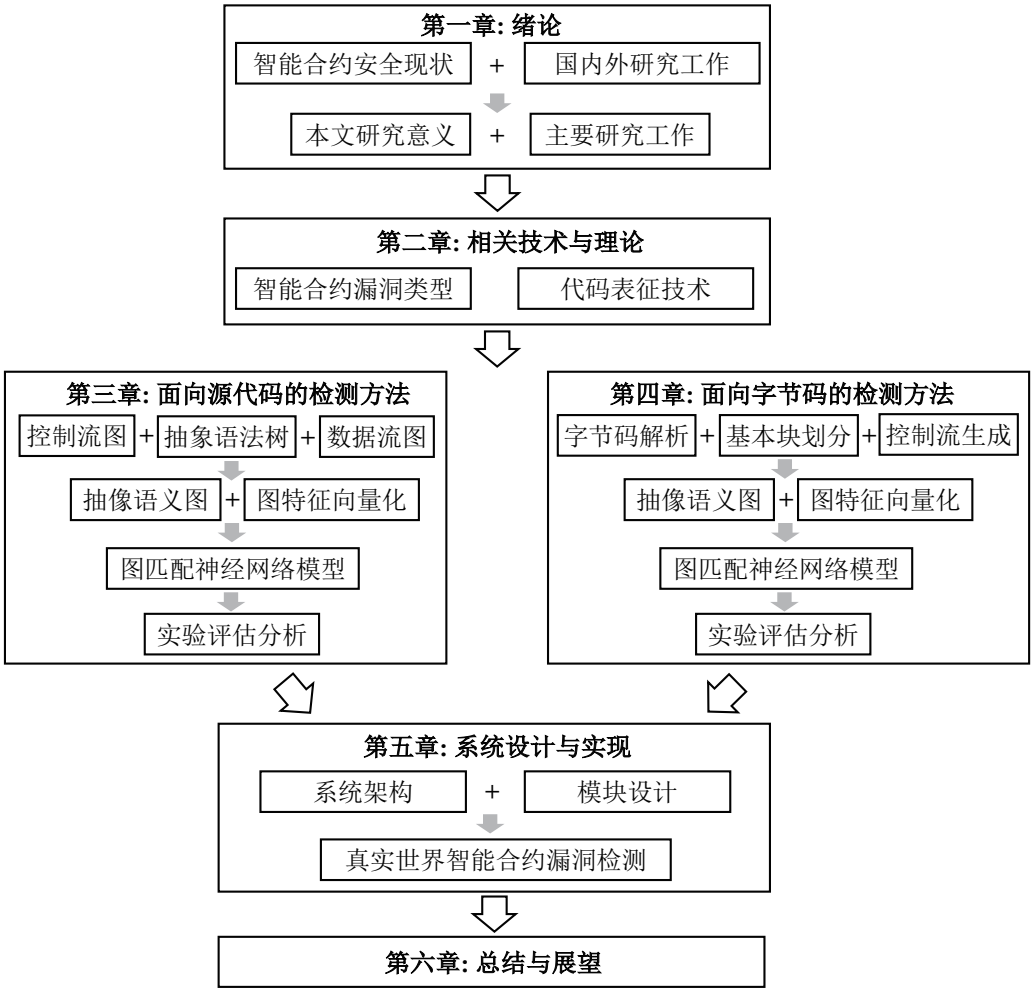


图 1-5 论文组织结构

第一章：介绍本文的研究背景与意义。概述了当前以太坊智能合约的应用生态和面临的安全挑战，总结了目前智能合约漏洞检测的技术路线，讨论了国内外研究现状及当前研究工作的不足，最后介绍了本文的主要研究内容和论文组织结构。

第二章：阐述本文所涉及到的相关技术和基本理论。介绍了四种基于以太坊平台的智能合约漏洞且进行了简要的代码分析，并重点阐述了智能合约漏洞检测中十分重要的

程序表征技术。

第三章：提出面向智能合约源代码的漏洞检测方法。首先，说明了如何从源代码中提取抽象语法树，并基于抽象语法树获取源代码的控制流图与数据流图；然后，生成包含语法和语义信息的抽象语义图，再通过词嵌入技术将抽象语义图转换为图数据结构；最后，再使用图匹配神经网络来检测漏洞，并在源代码形式的公开数据集上进行了实验验证。

第四章：介绍面向智能合约字节码的漏洞检测方法。首先，描述了如何从智能合约字节码中生成操作码，并从操作码中提取出控制流结构；接着，对操作码进行语义化表达，生成面向字节码的抽象语义图；最后，使用图匹配神经网络来检测智能合约字节码中存在的漏洞，并在字节码形式的公开数据集上进行了实验验证。

第五章：阐述本文提出的智能合约漏洞检测系统的设计实现。分析了系统的整体架构，介绍了系统的模块划分和各个模块的实现细节，并实现了一个在线漏洞检测系统，最后对真实世界的无标签智能合约进行了漏洞检测。

第六章：总结了本论文的工作和贡献。对本文研究工作进行分析，指出了本文提出的漏洞检测方法中存在的不足之处，并展望了未来研究工作的方向。

第二章　相关理论与技术

本章首先介绍了以太坊平台智能合约的四种漏洞实例，并对实例代码进行了简要的分析，然后详细介绍了与智能合约漏洞检测相关的代码表征技术，其中包括自然语言表征、深度学习表征和图神经网络表征。

2.1　智能合约漏洞类型

软件漏洞一般是指程序中出现的一个或多个缺陷，别有用心者可以利用该缺陷访问或修改程序数据、中断正常执行和执行需要授权才能进行的操作。智能合约漏洞是一种特殊的缺陷，这些漏洞包括资源被锁定或被盗取、身份丢失、数据完整性丢失以及执行过程中的智能合约状态变化，而这会给其他参与交互的用户带来不公平优势或劣势。智能合约漏洞可以划分成三个层级：在 Solidity 代码层，包含可重入漏洞、整数溢出漏洞等；在 EVM 执行层，包含 Tx.Origin 漏洞、调用栈溢出漏洞等；在区块链系统层，包含时间戳漏洞、区块参数依赖等。本文选取重入漏洞、Tx.Origin 漏洞、时间戳依赖漏洞和区块信息依赖漏洞这四种漏洞类型作为本文评估智能合约漏洞检测工具效果的基准。首先，这四种漏洞十分常见，现有的漏洞检测工具大多声称能够检测到它们，从而便于进行不同漏洞检测工具的性能比较；其次，包含这四种漏洞类型的智能合约有着较为明确语义结构，通过人工分析智能合约源代码或字节码生成的操作码可以相对轻松地识别这些漏洞，从而为后续面向真实世界智能合约漏洞检测中的人工分析工作减轻负担；最后，现有漏洞检测工具针对这四种漏洞类型的检测结果中依然有着较高的误报率和漏报率。

2.1.1　重入漏洞

重入漏洞（Reentrancy）也称为递归调用漏洞，如果外部调用导致被调用合约的控制流被接管，则表明被调用的合约中存在着重入漏洞[39]。智能合约中独特的回退（Fallback）机制使得攻击者能在程序执行结束前再次进入受害者合约的被调用函数，由此产生的意外递归行为将会导致循环调用。

The DAO 项目[40] 就是由于重入漏洞而导致了当时价值约 6000 万美元的以太币被盗，此次事件也直接导致了以太坊区进行了一个硬分叉（Hard-fork）。为了深入理解重入漏洞发生过程，这里对 The Dao 攻击案例进行简化。如图 2-1 所示的合约 Reentrancy 记录了许多外部地址的余额 Balance，并允许用户使用其公开的 withdraw() 函数来从相对应 userBalance 中取回资金。从直觉上看，合约 Reentrancy 允许这种情况只发生一次，因为需要检查是否包含足够的余额才能进行转账，而 call.value(amount)() 这种低级函数

```
contract Reentrancy {
    mapping (address => uint) public userBalance;
    function withdraw () public {
        uint amount = userBalance [msg.sender];
        if (amount > 0){
            msg.sender.call.value (amount)();
            // INSECURE: user account balance must be reset before external
            userBalance[msg.sender] = 0 ;
        }
    }
}
```

图 2-1 重入漏洞合约实例

会触发调用者的回退函数（即 function() payable ），从而导致重入漏洞发生。

如图 2-2 所示的攻击者合约 Attacker 可以通过它的回退函数来调用 withdraw () 再次进入合约 Reentrancy。具体的攻击步骤如下：1）攻击者利用 withdraw() 函数来提取以太币；2）受害者合约调用 withdraw() 函数并通过内置的 call.value (amount)() 将以太币发送给攻击者，合约 Attacker 中的回退函数会自动执行；3）攻击者在其回退函数中再次递归调用合约 Reentrancy 的 withdraw() 函数来提取以太币。因为从第一次提款起，合约 Attacker 的余额就未更新，所以攻击者会提取合约 Reentrancy 中剩余的全部以太币。

```
contract Attacker {
    // Withdraw account balance
    function() payable {
        Reentrancy(msg.sender).withdraw ();
    }
    function reentrancy (address addr) {
        Reentrancy(addr).withdraw () ;
    }
}
```

图 2-2 重入漏洞攻击者合约

更具体地说，合约 Reentrancy 预期整个生命周期的执行逻辑是：

$$\text{Attacker} \overset{\text{withdraw}}{\longrightarrow} \text{Reentrancy} \overset{\text{fallback}}{\longrightarrow} \text{Attacker} \longrightarrow \text{end}$$

而由于自身逻辑上的缺陷，攻击者可以利用这个缺陷使得包含重入漏洞的合约实际执行逻辑为：

$$\text{Attacker} \overset{\text{withdraw}}{\longrightarrow} \text{Reentrancy} \overset{\text{fallback}}{\longrightarrow} \text{Attacker} \overset{\text{withdraw}}{\longrightarrow} \ldots \longrightarrow \text{Reentrancy} \overset{\text{fallback}}{\longrightarrow} \text{Attacker} \longrightarrow \text{end}$$

在以太坊中，有 3 种调用方法可用于发送以太币，即 address.send()、address.transfer() 和 address.Call.value()。如果接收者是合约帐户，address.send() 和 address.transfer() 将最大 gas 限制更改为 2300 单位，而 2300 单位的 gas 不足以传输以太币，这也意味着 address.send() 和 address.transfer() 不会触发重入漏洞。

为避免重入漏洞，开发者应遵循以下编码实践：（1）Solidity 中，应使用 transfer() 或 send() 而不是 call() 将以太币发送给未知地址；（2）在执行外部调用之前，应对内部状态变更进行控制流假设；（3）只有在绝对必要的情况下才能进行外部调用。（4）外部调用应始终处理调用异常；（5）标记不受信任的合同。

2.1.2　Tx.Origin 漏洞

以太坊智能合约有一个全局变量 tx.origin，它可以遍历调用栈并返回发出调用的账户地址，tx.origin 变量可用于身份验证或授权操作事务。攻击者也可以借助 tx.origin 的特性来从受害者的合约中攫取以太币[41]。如图 2-3 所示的合约 Phishable 中的 withdarw() 函数在 require 语句里使用了 tx.origin，意为只有当 tx.origin 就是合约所有者的地址时，才可以发送以太币。

```
1  contract Phishable {
2      address public owner;
3      constructor () public {
4          owner = msg.sender;
5      }
6      function () public payable {}
7          function withdraw(address _recipient) public {
8              require(tx.origin == owner);
9              _recipient.transfer(this.balance);
10         }
11 }
```

图 2-3 Tx.Origin 漏洞合约实例

如图 2-4 所示的攻击者合约 Attacker 在自己的回退函数中调用合约 Phishable 的 withdraw() 函数就将再次进入合约 Phishable 中执行，此时 tx.origin==owner 条件被满足，从而使得合约 Phishable 向攻击者的合约发送以太币。

为避免 Tx.Origin 漏洞，开发者应遵循以下的编码实践和预防措施：（1）在智能合约鉴权逻辑中不使用 tx.origin，以防止使用中间合约来调用当前合约；（2）如果想要控制外部合约调用本合约，可以使用 require(tx.origin == msg.sender) 这个语句来防止一些中间的合约来调用本合约。

```
contract Attacker {
 Phishable victim;
 address attacker;
 constructor (Phishable _victimContract, address _attackerAddress) {
    victim = _victimContract;
    attacker = _attackerAddress;
 }
 function () {
    Phishable.withdraw(attacker);
 }
}
```

图 2-4 Tx.Origin 漏洞攻击者合约

2.1.3 时间戳依赖漏洞

一个区块中的所有交易共享相同的时间戳，从而保障合约在执行后的状态一致性。创建新区块的矿工（区块链网络中的节点或用户）可以将区块时间戳进行小范围的修改[42]，当智能合约执行某些逻辑用到区块时间戳 (block.timestamp)，那么矿工将有能力影响合约逻辑从而获得不公平的优势。

如图 2-5 所示的合约 Game 能够通过发出交易的所在区块时间戳来决定是否获奖，每个区块中只允许第一笔交易获奖，若区块时间戳的十进制表示最低位是 5，则交易发送者即可获奖。由于矿工有 0 到 900s 设置时间戳的范围权限，这使得矿工可以设置符合自己利益的区块时间戳。

```
contract Game{
    uint public lastBlockTime;
    function guess() public payable{
        require(msg.value == 100 wei);
        // Malicious miners can influence the blockchain timestamp
        require(lastBlockTime != block.timestamp);
        lastBlockTime = block.timestamp;
        if(lastBlockTime % 10 == 5){
            msg.sender.transfer(address(this).balance);
        }
    }
}
```

图 2-5 时间戳依赖漏洞合约实例

为避免时间戳依赖，开发者应遵循以下编码实践：（1）在合约中使用 block.timestamp 时，应该充分考虑该变量被矿工操纵的可能性；（2）block.timestamp 不仅可以被操纵，还可以被同一区块中的其他合约读取，因此也不应直接或通过某种推导间接成为赢得一

场比赛或改变一个重要的状态的决定性因素；（3）设计转账交易等操作时，如果对于时间操纵比较敏感，应当使用近期区块平均时间、区块高度等数据来进行资金锁定，而这些数据不能被矿工操纵。

2.1.4　区块信息依赖漏洞

智能合约开发中，在 Solidity 程序中使用较好的伪随机数是很难的，很多看起来难以被预言的随机数种子或变量，实际上被攻击者预测难度很低。如果在智能合约开发中采用了随机性不足的随机数作为关键条件，就可能遭遇预言攻击[43]。

当通过区块哈希（block.blockhash）、时间戳（block.timestamp）或区块号 (block.number) 等与区块参数相关信息作为生成随机数的种子就可能会导致随机性不足的问题，因为这些区块参数能够被矿工提前获知，故而可能被攻击者利用以获取不正当的利益。如图 2-6 所示的合约 Roulette 是轮盘赌合约的代码片段，该合约利用当前区块的哈希值作为选择本轮胜者的判断条件。如果一个矿工（或矿池）挖出下一个区块并发现区块哈希值不能使得他们获胜，他们就会丢弃这个不符合他们利益的块，继续挖矿直到挖出一个符合自身利益的块哈希值，以此从漏洞合约中获利。

```
1  contract Roulette {
2     constructor() public payable {}
3     function () public payable {
4       address [] participators;
5       // Malicious miners can influence the blockchain hashcode
6     uint winnerid = uint(block.blockhash) % participators.length;
7     participators[winnerId].transfer(1 eths);
8     }
9  }
```

图 2-6 区块信息依赖漏洞合约实例

为避免区块信息依赖漏洞，开发者应遵循以下的编码实践和预防措施：（1）随机数生成的方法有很多，并不一定要依赖区块参数，随机数应当尽可能地来源于区块链之外，例如 commit-reveal 之类的系统的对等体；（2）可以将信任模型改变为一组参与者，或者通过中心化的实体来完成，由这个实体生成随机数；（3）根据智能合约的官方最佳实践，合约开发者可以使用链外的第三方服务，比如 Oraclize 来生成随机数。

2.1.5　其他类型漏洞

目前已经有很多工作对智能合约中的漏洞进行归纳整理。Luu 等人[27] 的研究是最早对智能合约漏洞进行分类整理的工作之一，它总结了如交易顺序依赖性、时间戳依赖漏洞、错误处理异常和重入漏洞等常见漏洞。Atzei 等人[44] 是最早创建基于以太坊的智

能合约漏洞分类法的工作之一，Dika 等人[45] 在 Atzei 等人的基础上更新并添加了更多智能合约漏洞类型，并对每种漏洞进行了严重程度的评估。开源社区项目的智能漏洞分类集有 DASP TOP 10[46] 和 SWC Registry[47]，DASP TOP 10 将智能合约漏洞分为了 10 个等级，SWC Registry 收集了 37 种开发中常见的智能合约安全漏洞。据不完全统计，目前已知的智能合约漏洞类型超过了六十种[48]。

2.2 代码表征形式

机器学习作为一种表征学习的技术，当机器学习用于代码漏洞检测时，应该将代码表征为向量作为模型的输入。因此要将机器学习应用到智能合约的漏洞检测中，首先应该将智能合约代码编码为合适的向量形式，本节主要说明代码表征粒度与层次和机器学习中常用的表征模型。

2.2.1 代码表征粒度与层次

代码的表征是对智能合约漏洞检测的第一步，在此基础上设计漏洞特征的提取方法，并实现智能合约漏洞的检测。代码表征决定了对智能合约特征提取的程度，从而对后续的漏洞检测效果产生影响。如图 2-7 所示，可以将代码表征技术从表征粒度和表征层次两个方面来进行说明。

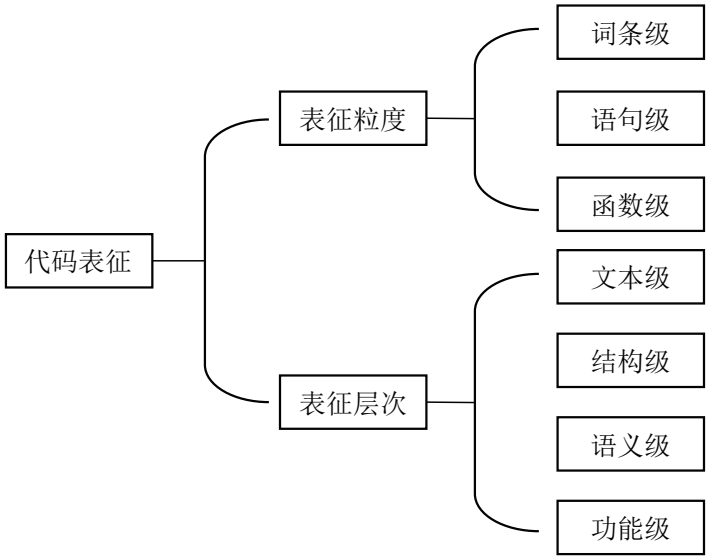


图 2-7 代码表征分类

根据代码特征的提取粒度的不同，可以将代码表征分为词条级、语句级和函数级三个级别[49]：

（1）词条级：词条是表达程序的最细粒度，但难以表达词和词之间的联系。代码自身就是词条序列，但由于程序设计中命名规范十分自由，例如同一个变量可以命名为

num、num1、number123、n，因此面向词条级的建模难以把握词表的大小。

（2）语句级：语句是代码片段的基本组成成分，而代码是由不同种类的语句组成并按照语句顺序依次执行。各种程序语言的一条语句可以表述不同的内容，比如某一条语句可以是一个运算，也可以是调用某个函数的算法。故而在语句级上可以采用抽象语法树（Abstract Syntax Tree，AST）、数据流图（Data Flow Graph，DFG）等方式来进行表达。语句级的粒度介于词条级和函数级之间，能结合上下文表达来表达代码的信息。

（3）函数级：函数级的抽象层次比词条级和语句级要更高，能够更好地表达语义信息，但是复杂度也随之提高。目前直接对程序中的函数进行表征的工作较少，常用的手段是将函数拆分为粒度更细的段，然后对这些小段进行组合，从而表征出函数的特征。例如将函数转换为抽象语法树，然后对抽象语法树进行加权组合学习，从而获取函数的表征向量。

依据抽象程度的不同，可以将代码表征的层次分为文本级、结构级、语义级和功能级四个层次，层次级别越高，其抽象的程度就越高，从中获取的代码信息量就越多[50]。

（1）文本级：代码可以被看作为一个普通的文本，自然语言模型能够将文本转换为任意的词序列，进而将代码映射为词向量。根据是否对代码进行词法分析可以将代码的词汇级表征分为两种类型，一种是基于文本形式的词汇，也就是直接将文本拆分为一个个词，然后应用 Word2vec[51] 等词嵌入技术将各个词映射为向量，然后通过加权平均向量来表征代码；另一种是基于语义的词汇序列。首先对代码进行词法分析得到词汇序列，在转换的过程中移除掉了无意义的空白、制表符、注释等，从而增强了词汇序列对代码的抽象，不足之处在于缺少了代码的结构特征。

（2）结构级：如果两个代码片段只是在标识名、变量名、变量类型和函数名等代码书写风格的部分有一定不同，并且修改、增加或者删除了一些不影响程序逻辑的语句，而在执行结构和结果上保持一致，那么就可以说两段代码在结构上看是一致的。抽象语法树是源代码抽象句法结构的一种抽象表示形式，一般是由编译器生成，是源代码构建其他更加复杂中间表示形式的基础中间表示。抽象语法树在结构上可以表现出源代码语句和表达式相互影响的联系，而这种语句与表达式之间的相互关系构建了程序。抽象语法树中并不会体现智能合约源代码真实语法中的各个细节，换句话说就是不使用源代码中存在的实际语法来表现程序，而只是表达智能合约代码的结构或者跟内容有关联的信息。基于抽象语法树的代码表征方法是经典的结构级别表征形式，将代码进行词法和语法分析之后转换为抽象语法树，它可以摆脱代码的词汇差异的影响，从而更加专注于两段代码之间的结构特征。

（3）语义级：代码不单单有着静态的结构信息，即使是结构信息相同的代码，执行的路径很大可能并不相同。目前通过图的形式来对代码表征是语义级表征的主要形式，它是通过控制流图（Control Flow Graph，CFG）、数据流图等方式来承载比结构级表征方式更加详细的信息。控制流图一方面反映了程序每条语句的执行路径，另一方面也直观地展现了语句在执行过程中达到某种独特的路径需要符合的条件。控制流图呈现了

源代码在执行的过程中有可能会经过的所有路径，也以图的形式展现了一个程序中基本块的所有流向。数据流图是从数据的视角来展现源代码的，它显示了数据在程序中的定义、修改和使用过程，仿佛数据在程序中流转一样，而数据流转情况描述了程序中各个语句之间对于同一数据的依赖关系。随着图嵌入技术的出现，语义级表征代码正在成为一种研究趋势。

（4）功能级：如果两段代码片段通过不同语法方式实现了相同功能，这就达到了功能级上的相似。通过对代码进行功能级别上的表征，可以最大限度保留语法和语义信息。

2.2.2 自然语言表征模型

在自然语言处理领域，传统的将文本转化为向量的模型有 one-hot 模型、N-Gram 模型和 Word2vec 模型。one-hot 编码[52] 又称为一位有效编码，是将程序中的代码进行表征的最简单直接的方法，它的主要思路是将向量中的每一个元素都与词库中的一个词条关联，每个词条占据向量的唯一一个位置，在任意情况下词向量中有且只有一个元素为 1，其他元素都为 0。one-hot 编码使用了最简便直观的方式来使得不同编码之间保持正交，不过这样也损失了编码之间的上下文信息，而且随着词表的增加，one-hot 向量的维度也随之增加，导致向量变得极为稀疏。

N-Gram 模型是一种基于概率的语言模型，它根据单词的顺序序列，以大小为 N 的宽度预测下一个输出序列的概率[53]。每一个单词片段称为 gram, 对所有的 gram 出现频率进行统计，并且按照一定的阈值过滤，从而生成关键的 gram 列表（即这个文本的向量特征空间）。N-Gram 模型基于第 N 个单词只与前面 N-1 个单词相关而不与其他任何词相关的假设之上，如式 (2.1) 所示，语句出现的概率就是语句中每个词出现概率的乘积，而每个词的出现概率可以从事先准备好的语料库中统计得到。

$$\mathrm{p}\left(w_1, w_2, \cdots w_N\right)=\mathrm{p}\left(w_1\right) \mathrm{p}\left(w_2 \mid w_1\right) \cdots \mathrm{p}\left(w_t \mid w_1, w_2, \cdots w_{N-1}\right) \tag{2.1}$$

受限于计算机的算力，N 一般取 2 或者 3。如果一个单词的出现仅依赖于它前面出现的一个词，那这样的模型就称之为二元模型（bigram model），即 $\mathrm{p}\left(w_1, w_2, \cdots, w_m\right)=\prod_{i=1}^{m} P\left(w_i \mid w_{i-1}\right)$；仅依赖于前面出现的两个词的模型就称之为三元模型（trigram model），即 $\mathrm{p}\left(w_1, w_2, \cdots, w_m\right)=\prod_{i=1}^{m} P\left(w_i \mid w_{i-2} w_{i-1}\right)$。

Word2vec 模型是自然语言处理领域常用的词嵌入模型，它的主要思想是使用神经网络预测每个词的向量分布。Word2vec 模型分别有连续词袋 (Continuous bag-of-word model, CBOW)[54] 和 skip-gram[55] 模型这两种来对词向量进行训练。CBOW 模型是利用上下文来预测某个词的出现概率，文本中某个单词的上下文环境是由其左右的 N 个词组成，那么首先就需要将该词左右 2N 个词作为输入层的输入，接着将该词的上下文环境词向量进行平均求和，最后利用 Softmax 技术将词库中所有出现的词组按照文本出现

的频率构建一棵霍夫曼树。Skip-gram 模型是依据当前词来预测词组对应的上下文，因此某个词组出现的概率就转化为 p(Context(w) | w) 。

2.2.3　深度学习表征模型

传统表征代码特征的模型在训练的难易程度上有一定的优势，但对于代码的上下文反映能力较弱，而随着深度学习技术在图像识别和自然语言处理领域的不断成熟，各种基于神经网络的代码表征模型也如雨后春笋般出现。当前深度学习在代码表征领域所采用的神经网络模型有循环神经网络模型（Recurrent Neural Network, RNN）、卷积神经网络模型（Convolutional Neural Network,CNN）和 Seq2seq 模型[56]。深度学习网络的一般过程是直接将代码以及标签过的数据作为模型的输入，使用深度学习模型来挖掘数据中深层次的特征，从而将高维度的稀疏表示转化为低维度的连续空间向量表示，进而使用这些向量特征来处理各种后续工作。

神经网络一般包含输入层、隐藏层和输出层三层结构，每层结构依次递进，每层中间的节点相互之间没有连接，不同层之间也并不是全连接，因此无法处理随着递归出现的权重指数爆炸和梯度消失的问题，也难以捕捉序列化数据的特征。循环神经网络可以把每层内的节点通过时序的方式联合起来，从而有能力建立起序列数据之间的依赖关系。RNN 模型[57] 的模型架构如图 2-8 所示，通过将序列数据作为输入生成固定大小的向量，接着输出到其他网络结构。循环神经网络在处理和预测序列数据方面有着很好的效果，因此适合那些需要泛化上下文相关信息的工作，当上下文序列过于冗余时会有梯度消失的风险。

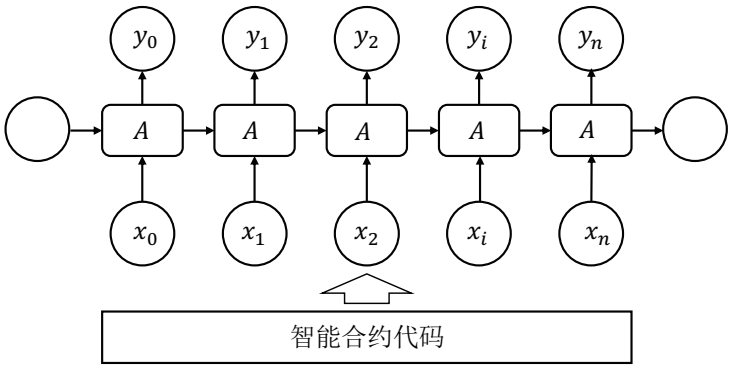


图 2-8 循环神经网络模型架构

卷积神经网络[58] 是一种前馈的神经网络，主要使用了卷积层来替代传统神经网络中的全连接层，从而降低了卷积神经网络中需要训练的参数个数，加快了网络的收敛速度。CNN 模型的一般结构如图 2-9 所示，卷积层有着获取数据特征的能力，它可以将一些底层的特征可以不断地映射成人类可以理解的高层特征。软件工程领域的一些应用就借鉴了卷积神经网络在图像识别领域的能力：代码片段可以被抽象地理解为一张二维图片，每个代码元素对应着图片中的一个像素，卷积神经网络就可以像捕捉图片的边角信

息一样捕捉代码中的底层特征，然后通过不断地卷积映射获取代码中的高层语义特征，从而获取到程序代码中隐含着的结构信息，但 CNN 模型中同样存在着梯度消失的问题。

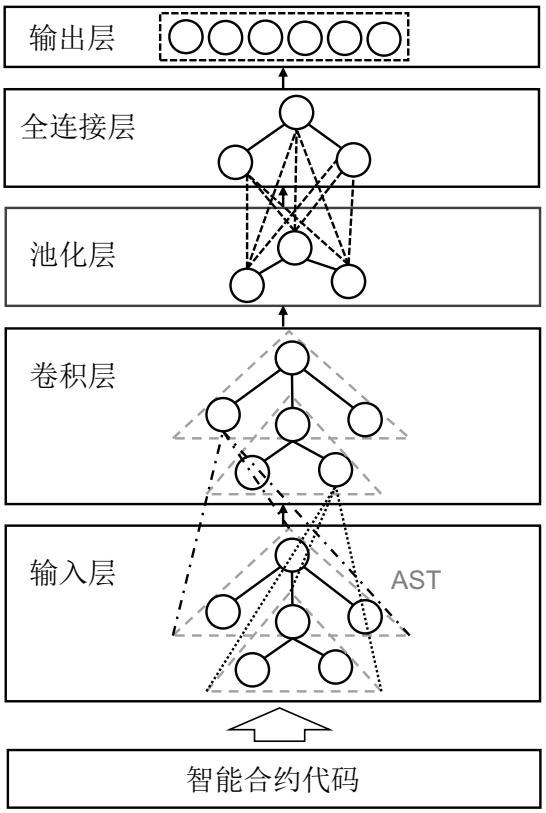


图 2-9 卷积神经网络模型架构

Seq2seq 问题在机器语言翻译和文档摘要等应用场景中出现地十分频繁，也就是将一个输入的序列文本转化为另一种形式的输出文本。Seq2seq 模型[59] 就是为了应对 seq2seq 问题而产生的，如图 2-10 所示的 Seq2seq 模型本质上就是一种编-解码的结构。

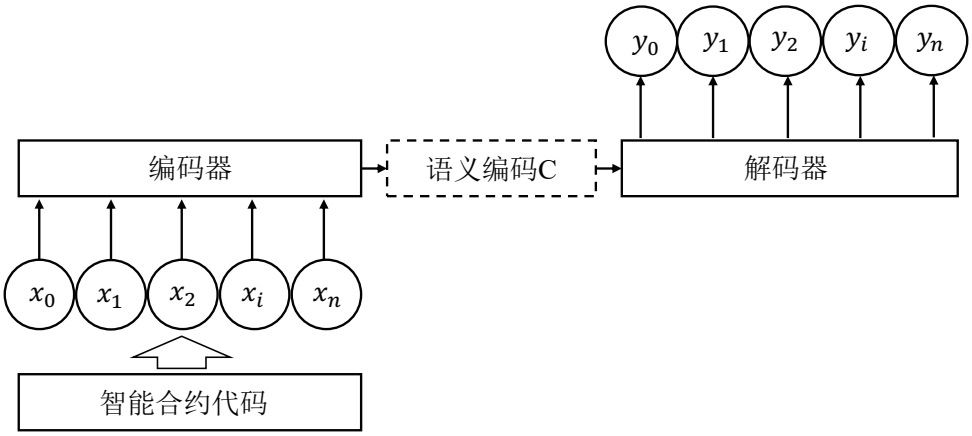


图 2-10 Seq2seq 模型架构

编码器将输入的文本序列转化为固定大小的语义编码 C，而解码器就是提供一个解

码器来将这个语义编码向量转化为一个输出序列。Seq2seq 模型有着极大的自由度，因为它的编码和解码器可以是任何一种类型的结构，从而为不同的任务提供解决方案。

2.2.4　基于图神经网络的代码表征

图是一种十分典型的非欧几里得数据类型，包含节点和边两种组成成分。传统的对图数据的研究主要聚焦于图节点分类、节点链接预测和图聚类等，而图网络嵌入（Graph Network Embedding）算法在此类研究过程中发挥了重要作用。图网络嵌入算法是将图数据中的节点进行映射，从而将图映射为一个固定大小的向量。随着面向图的研究逐步深入，图神经网络[60] 的模型也被研究者提出，并在后续的研究中确定了图神经网络采用类似循环神经网络的结构，该结构将图中信息向相邻节点传播[61]，直到图中所有的节点都到达一种稳定的状态。随后提出的消息传递神经网络（Message Passing Neural Network，MPNN）[62] 确立了当前图神经网络基本的三个阶段：首先，传播阶段将图神经网络的节点和边自身信息向相邻的节点进行传播；随后，更新阶段则将这些从相邻节点传播而来的信息更新到节点和边上，此后不断往复直到所有节点达到稳定；最后，读出阶段输出到整个图的向量信息。当前，图神经网络中的门控图神经网络（Gated Graph Neural Network，GGNN）[63] 在代码语义表征领域有着较好的性能表现。

程序语言是一种高度结构化的文本信息，目前基于深度学习的智能合约漏洞检测方法一般只能将智能合约最基本文本特征用于漏洞检测，而忽略了智能合约中更加丰富的结构信息，比如抽象语法树中的语法信息和控制流图中的结构化信息。图神经网络可以挖掘出漏洞研究者都难以察觉到的漏洞潜在特征，能够有效地拓展漏洞的搜索范围。如图 2-11 所示，图神经网络被用于学习智能合约生成的非欧式空间图数据信息时，它既能够学习到各个节点的局部文本信息，也可以学习到诸如控制流图、数据流图和程序依赖图等所包含的全局信息，从而更加全面地表征一个智能合约的语法和语义特征。

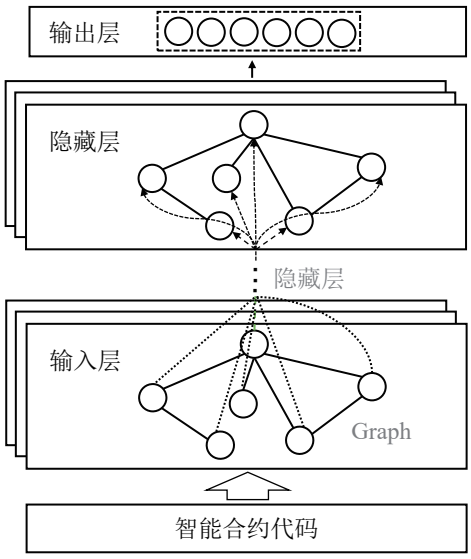


图 2-11 图神经网络模型架构

2.3 本章小结

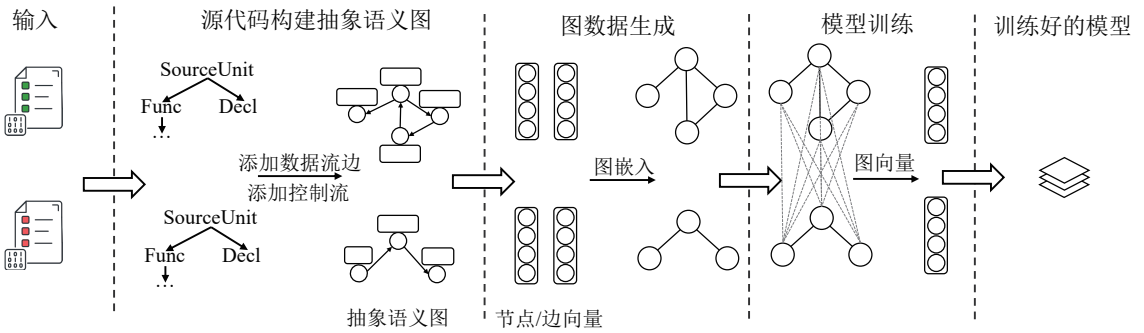
本章主要对研究中所涉及的相关理论和技术进行了介绍。首先，介绍了以太坊智能合约的四种类型漏洞，并简要分析了它们形成的原因和在智能合约实践中的避免措施。其次，介绍了代码表征的粒度、层次的分类，并详细介绍了抽象语法树、控制流图和数据流图等表征形式。然后，说明了几种常用的代码表征模型，并介绍了它们的主要特点和模型架构。最后，介绍了研究中所要使用到的图神经网络表征智能合约的基本思路。

第三章　面向源代码的检测方法

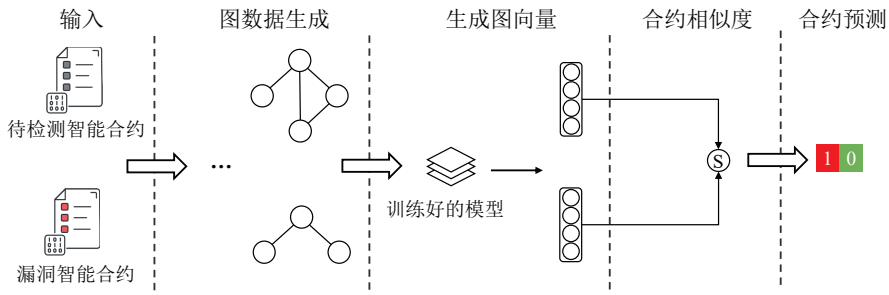
本章面向可获得智能合约源代码的场景介绍本文提出的智能合约漏洞检测方法。现有基于机器学习的漏洞检测方法直接将源代码转化为序列，这会丢失代码原有的结构信息，而图结构数据能够表现节点之间的关系从而反映源代码中的语法和语义特征。本章介绍了从智能合约源代码转化为图结构数据的过程，并利用图匹配神经网络来检测智能合约源代码中存在的四种类型漏洞。

3.1　基本框架

本文面向智能合约源代码漏洞检测的主要思想是将源代码的图向量化嵌入技术与图神经网络相结合。首先将智能合约源代码转化为图结构数据，并对包含不同漏洞类型的智能合约进行标注，利用图神经网络学习源代码图结构数据中隐含的语法与语义信息。使用训练好的图神经网络对智能合约进行检测，判断该合约是否包含相应的漏洞。图 3-1 是面向源代码的图神经网络的训练过程以及利用图神经网络对智能合约进行漏洞检测过程示意图。



(a) 模型训练



(b) 模型检测

图 3-1 面向源代码漏洞检测模型框架

面向智能合约源代码漏洞检测系统主要分为了训练和检测两个阶段。训练阶段如图3-1 (a) 所示，首先为不同的漏洞准备不同的数据集，在训练检测某一类型漏洞的图神经网络时从该漏洞类型的训练集中取出一对智能合约，并且分别从它们身上提取抽象语法树、数据流图、控制流图，进而生成抽象语义图。然后，对抽象语义图生成的节点属性和边属性进行图向量化嵌入，将生成的一对图结构数据作为图匹配神经网络的输入。根据输入的一对智能合约标签和指定的损失函数来对图神经网络模型中的参数进行调整，直到图神经网络模型充分学习到漏洞的特征。

每种漏洞类型的检测流程都是一样的，需要使用训练好的不同图神经网络模型来检测智能合约特定漏洞。如图 3-1 (b) 所示，如果要检测智能合约是否包含某种漏洞类型，首先需要从该漏洞类型的训练集中随机选取一个包含该漏洞类型的智能合约，通过抽象语义图构图模块和图向量嵌入模块生成图结构数据，并将待检测的智能合约采用同样的方式转化为图结构数据。对于已经训练好的检测该漏洞类型的图神经网络，可以将待检测智能合约的图结构数据和包含该漏洞类型智能合约的图结构数据作为一对数据输入，两个图数据经过图匹配神经网络之后会分别生成两个图向量。利用余弦相似度函数计算两个图向量在空间上相似度系数，根据设定好的相似度阈值系数来判断待检测合约是否包含该漏洞，如果相似度系数超过阈值则说明待检测合约包含该漏洞；如果相似度系数低于阈值，则从该漏洞的数据集中再取出一个漏洞合约进行判断。重复上述步骤，直到待检测的智能合约与某个包含漏洞的智能合约相似度超过阈值为止。若待检测合约与该漏洞的数据集中所有合约都不相似，则说明待检测智能合约不包含该漏洞。

3.2 基于源代码的图数据结构生成

智能合约源代码的控制流信息和数据流信息隐含着代码各语句之间的关联，因此图结构数据相较于文本序列形式能更有效地反映代码数据和控制依赖关系。本节将介绍一种从智能合约源代码中生成，聚合了抽象语法树、数据流图和控制流图的新结构——抽象语义图，以如图 3-2 所示的合约 Simple 为例来说明构建抽象语义图的过程。

```
1  contract Simple {
2     mapping (address => uint) public userBalance;
3     function withdraw () public {
4        uint amount;
5        amount = userBalance [msg.sender];
6        if (amount > 0){
7           msg.sender.call.value (amount)();
8           userBalance[msg.sender] = 0 ;
9        }
10    }
11 }
```

图 3-2 一个简单的智能合约

3.2.1　基础语义图

抽象语法树是一棵有先后顺序的树，它的叶子节点通常对应于操作数，例如标识符或者常量；非叶子节点通常对应于运算符，比如减加乘除法运算或者赋值运算。智能合约源代码使用了 Solidity 语言编写，Solidity 中声明变量可以有多种方式，比如可以采用逗号隔开两个变量达到同时声明两个变量，也可以使用两条语句分别声明两个变量，而源代码这两种不同的声明方式在抽象语法树中是相同的，图 3-3 中所示的抽象语法树是由合约 Simple 展示的智能合约代码生成。

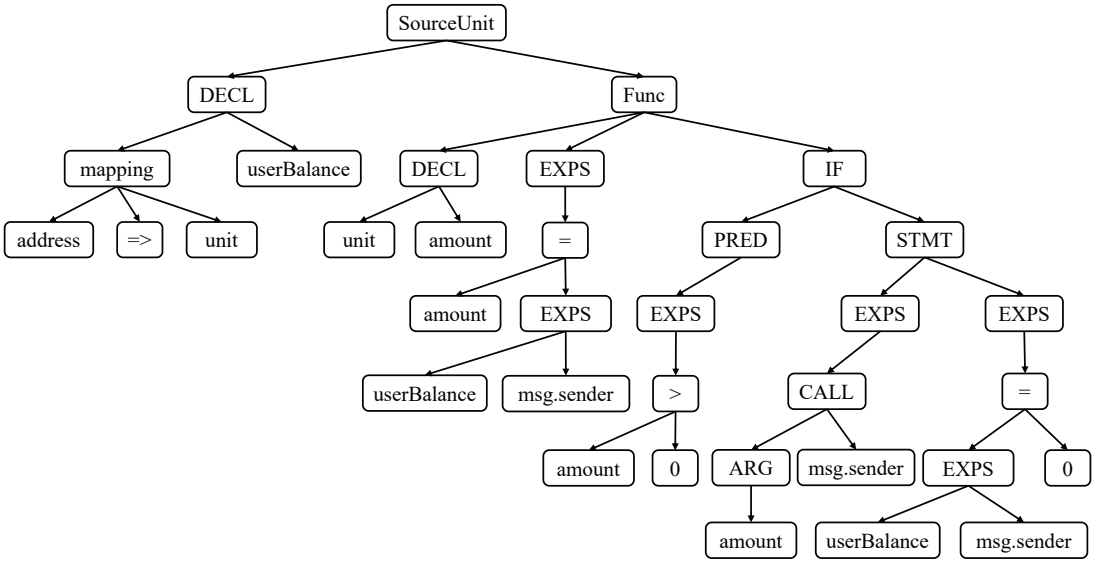


图 3-3 抽象语法树

从图 3-3 中所展现的抽象语法树可知，该合约包含一个函数定义和一个变量定义。函数体中有四条语句，分别为一条变量声明语句（声明局部变量 amount）、两条赋值语句（分别为 amount 和 userBalance 赋值）和一条关系判断语句（用于判断变量 amount 的值是否大于 0），在关系判断语句之后包含着语句内要执行的其他语句，从而形成了嵌套关系。从上面简单分析可知，抽象语法树不仅能够很大程度地留存智能合约源代码的语法结构，而且可以明确地表达合约源代码中语句之间的层次关系。

然而，源代码中抽象语法树的表现方式难以直观地展现源代码中各个语句之间的数据依赖、控制依赖以及不同语句之间的顺序关系。程序的循环关系有 for 循环、while 循环、do-while 循环，分支关系有 if 结构、if...else 结构、if...else if 结构、switch...case 结构，顺序关系是语句按照程序依次执行的逻辑关系。每个语句节点都会有顺序流边，而只有条件语句会有相对应的 True 边、False 边。利用 Solidity 抽象语法树中的基本控制语句（例如 If、While、For 等控制依赖关系）来构建基本的控制流图，然后根据 goto、continue、break 等控制语句来对控制流图进行调整。图 3-4 就是合约 Simple 中提取的控制流图，控制流图从开始节点进入，最后从退出节点结束，图中每个矩形框都表示着一个控制流图节点，同时也对应着智能合约程序中一个语句。从图中可以分析出整个合约

只有一个条件语句，条件语句有两条分支，故而整个程序有两条执行路径。在 If 条件语句中判断 amount 值是否大于 0，如果 amount 大于 0 则进行转账操作并将该用户的余额改为 0，如果余额不大于 0 则直接退出程序。

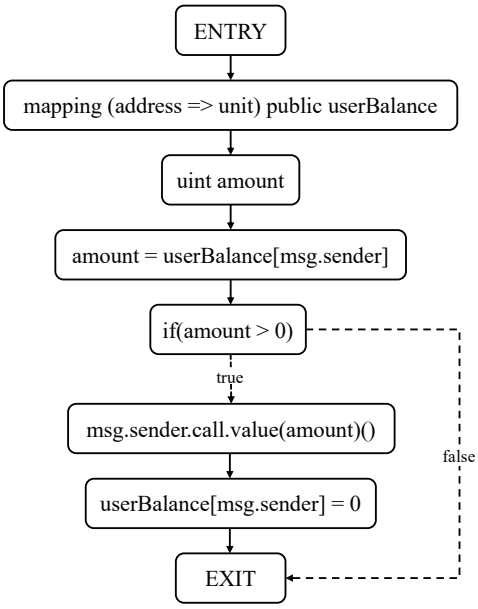


图 3-4 控制流图

控制流图虽然包含了源代码的控制流依赖关系，但是并不包含源代码的数据流依赖关系。智能合约源代码的数据流图能够从控制流图中生成，首先对程序进行一次遍历来获取程序中所有数据变量，之后再次遍历程序来获取所有使用到数据变量的语句。通过遍历上面生成的控制流图，在使用到同一个数据的语句之间添加数据依赖边，同时在有向边上添加该数据作为此边的标签，从而构建出整个程序的数据流图。图 3-5 是合约 Simple 生成的数据流图，图中的矩形框表示程序中的一个表达式，数据流边体现了变量在不同表达式之间的传递过程和语句之间所蕴含的数据依赖关系。从图中不难看出，userBlance 变量从定义开始先后被语句调用了两次，首先在生成变量 amount 时被调用一次，然后变量 amount 被用作条件语句的判断条件时被再次调用。观察数据流图可以十分清晰地发现 userBlance 与条件判断语句之间存在着关联，而这种数据依赖关系在抽象语法树和控制流图中是难以获取的。

3.2.2 抽象语义图生成

从抽象语法树中获取到智能合约源代码的控制流图和数据流图等基础语义图之后，就需要将它们进行组合，从而生成一种能够表征该合约的全新结构。这种能够更大限度表征源代码原有语法和语义信息的表示形式被本文称为抽象语义图，它能够进一步地利用源代码中的控制依赖关系与数据依赖关系来检测智能合约中存在的漏洞。其中，控制流图中的控制依赖边反映了程序中控制语句对于数据的限制，而数据流图中的数据依赖对应着程序中语句通过数据变量对另一语句的执行产生逻辑上的影响。

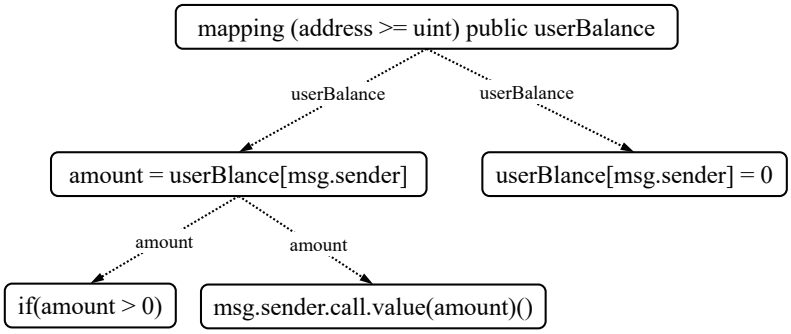


图 3-5 数据流图

将数据流、控制流图和抽象语法树组合起来构建基于源代码的抽象语义图。抽象语义图以抽象语法树为基础，将控制流图中的 If 条件结构、While 循环结构和 For 循环结构添加到抽象语法树，首先找出控制流起始语句和目的语句对应的抽象语法树节点，然后根据控制流的类型与方向，在抽象语法树的两个节点之间添加相应的控制流边。数据流图也按照同样的方式加入到抽象语法树中，在数据流边的起始语句到目的语句对应于抽象语法树的节点上添加数据流边，由此构建起抽象语义图。如图 3-6 所示，它是由合约 Simple 的控制流图和数据流图添加到抽象语法树上生成的抽象语义图。从图中能够看出节点 1 指代的是全局变量 userBlances, 之后通过数据流图指向了节点 2 的局部变量 amount, 然后节点 2 中的变量受到了节点 4 的条件判断语句限制，之后节点 5 调用了节点 2 进行转账操作，最后经过顺序流指向节点 6，节点 6 中对全局变量进行了更新。从整个过程能够发现，在进行转账操作的时候，并没有做出应有的限制，这可能导致触发智能合约重入漏洞，也就是说合约 Simple 源代码的第 7 行语句可能会导致重入漏洞。

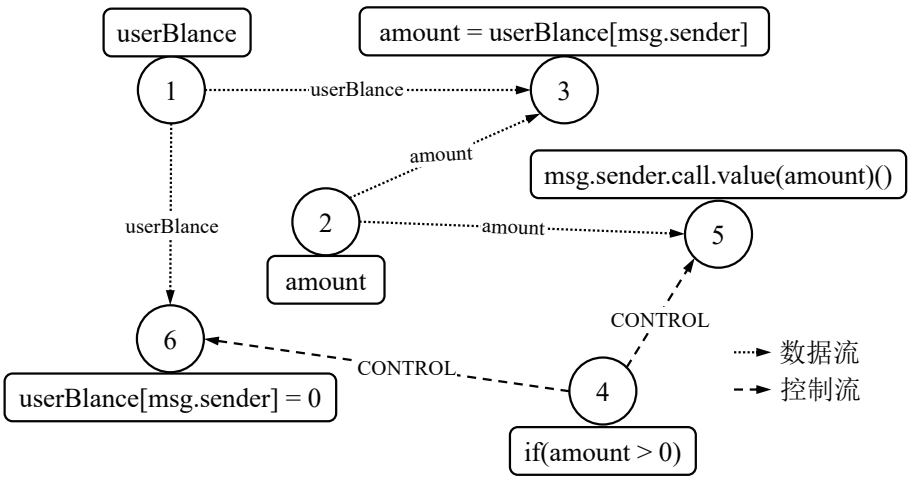


图 3-6 抽象语义图

算法 3.1 描述了由智能合约源代码生成抽象语义图的全部过程。其中，第 1 行将智能合约编译为抽象语法树，第 2 行初始化要构建的抽象语义图，第 4 - 8 行为抽象语义图添加控制流边，第 9 - 14 行标记智能合约中所有的数据变量，第 16 -21 行为抽象语义

图添加数据流边。

算法 3.1 源代码构图过程算法

输入: 智能合约源代码 C

输出: 源代码的抽象语义图 ASG

```
将源代码 C 编译为抽象语法树 AST
初始化抽象语义图 ASG
for 节点 in AST do                                          ▷ 添加控制流边
    if 节点 is 控制流节点 then                               ▷ 如 For、While 和 If 语句等
        在对应的 ASG 中节点添加对应的控制流边
    else
        在对应的 ASG 中节点添加对应的顺序流边
    end if
    if 节点 is 数据定义节点 then
        将节点添加到 DataDefinition 中
    end if
    if 节点 is 数据使用节点 then
        将节点添加到 DataUse 中
    end if
end for
for 数据变量 1 in DataDefinition do                         ▷ 添加数据流边
    for 数据变量 2 in DataUse do
        if 数据变量 1 == 数据变量 2 then
            在 ASG 中两个数据变量所在的节点之间添加数据流边
        end if
    end for
end for
```

3.2.3 图特征向量化

在获取到包含着语法和语义信息并隐含智能合约漏洞特征的抽象语义图之后，需要进一步构建图嵌入网络来向量化抽象语义图以便后续进行漏洞检测，图嵌入过程中应当考虑节点特征向量、边特征向量和整个图向量的生成。

在抽象语义图中，每个节点表征的是一条源代码语句，边属性也是一个语句，因为图神经网络难以处理文本数据，故而要将以文本形式表征的语句进行向量化处理，以此来表征节点和边的特征。本文采用 Word2vec 模型来对源代码生成的抽象语义图节点属性和边属性作词嵌入，从而将语句转换为稠密的向量表示形式。首先，遍历整个抽象语法树所生成节点的所有文本，将这些文本作为 Word2vec 的语料库；然后，将每个节点

的文本采用 one-hot 编码为稀疏向量，接着采用如图 3-7 所示的 CBOW 模型和基于层次 softmax 多类别分类器来对模型进行训练。

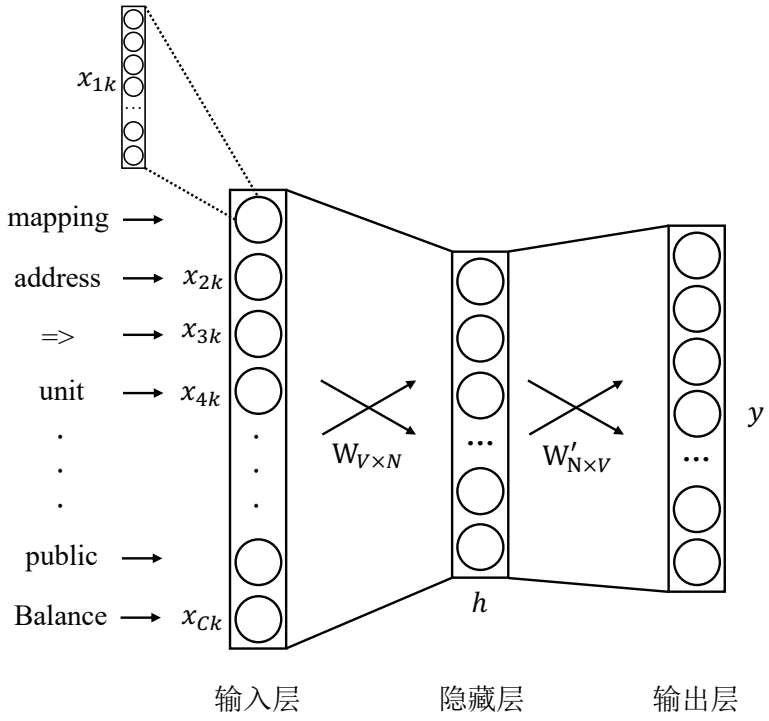


图 3-7 CBOW 模型

这个模型主要采用了上下文预测当前词的向量稠密表示，分为输入层、隐藏层和输出层。对于输入层，通过设置固定大小的窗口，然后将向量化的源代码语句作为输入，把源代码抽象语义图中的节点作为一个单元，将其中的每个词转换为 one-hot 编码，于是就可以把一个节点编码为 $X=(x_{1k},x_{2k},...,x_{ck})$，其中 c 表示词的总数，x_i 为其中一个词的 one-hot 编码，k 表示整个语料库的长度，每一个 x_i 的维度都是 V。隐藏层是对所有稀疏向量做求和平均，由此获得需要的隐藏向量，如式 (3.1) 中所示，N 表示隐藏层的维度，$W\in R^{V\times N}$ 表示隐藏层的权重参数矩阵。

$$h=W^{T}X \tag{3.1}$$

从隐藏层到输出层的计算如式 (3.2) 所示，将 h 与 W' 进行左乘得到 $\mu=(\mu_1,\mu_2,\ldots,\mu_V)$，其中 μ_j 是由权重参数矩阵 $W'\in R^{N\times V}$ 的第 j 列 μ_{w_i} 与 h 作内积得到。

$$\mu=W'^{T}h \tag{3.2}$$

如式 (3.3) 所示利用 softmax 函数将 μ 进行归一化处理，其中 $v_{w_k}^{T}$ 表示输入词向量，μ_{w_i} 表示输出词向量。

$$y = \frac{\exp(\mu_j)}{\sum_{k \in V} \exp(\mu_k)} = \frac{\exp(v'^{T}_{w_j} \cdot v_{w_i})}{\sum_{k \in V} \exp(v'^{T}_{w_k} \cdot v_{w_k})} \tag{3.3}$$

通过反向传播误差算法来逐步更新权重参数矩阵 W'^{T} ，使得损失函数的值达到最小，从而达到最优的词嵌入效果。损失函数如式 (3.4) 所示，其中 j^* 为词 w_j 在语料库中的编号。

$$E = -\mu_{j^*} + log \sum_{k \in V} \exp(\mu_k) \tag{3.4}$$

Word2vec 模型对语料库学习之后，就可以构建出语句所有文本的向量映射表，进而能够对语料库中每个节点和边进行向量化表示。利用 Word2vec 模型来向量化从源代码中生成的抽象语义图，从而生成源代码的节点属性向量和边属性向量。整个图嵌入过程如算法 3.2 所示，第 1 - 5 行是利用训练好的 Word2vec 模型将输入的文本 X 转化为对应的向量，第 6 行是对要生成的图数据结构进行初始化，第 7 - 9 行将抽象语义图的节点进行向量化，第 10 - 12 行是对抽象语义图的边进行向量化。

算法 3.2 源代码图嵌入过程算法

输入: 源代码抽象语义图 ASG

输出: 节点嵌入向量 V 和边嵌入向量 E

1: **function** Word2vec(X)
2: $h = W^T X$
3: $\mu = W'^T h$
4: $y = \frac{\exp(\mu_j)}{\sum_{k \in V} \exp(\mu_k)} = \frac{\exp(v'^{T}_{w_j} \cdot v_{w_i})}{\sum_{k \in V} \exp(v'^{T}_{w_k} \cdot v_{w_k})}$
5: **end function**
6: V, E ← init(ASG) ▷ 将 ASG 的节点和边进行初始编码
7: **for** X_{Node} **in** V **do** ▷ 对节点进行词嵌入
8: V.X ← Word2vec(X_{Node})
9: **end for**
10: **for** X_{Edge} **in** Edge **do** ▷ 对边进行词嵌入
11: Edge.X ← Word2vec(X_{Edege})
12: **end for**

3.3 基于图神经网络的漏洞检测

本文采用的图匹配神经网络技术不仅能够利用节点和边属性向量信息，还可以利用智能合约源代码中隐含的数据依赖和控制依赖的这些结构信息，并依据这些信息更好地学习智能合约源代码的漏洞特征，从而检测出智能合约源代码中所包含的具体漏洞。

3.3.1　图匹配神经网络架构

图神经网络是学习图结构数据信息的关键技术，它将图结构数据作为输入。输入的图经过几层图网络的更新之后，每个节点的特征向量都会得到更新，使用图聚合的方法将图中所有节点和边的特征向量进行计算，从而得到整个图的特征向量。本文所采用的具有跨图注意力机制的图匹配神经网络（Graph Matching Network，GMN）[64]，其架构如图 3-8 所示。图匹配神经网络需要输入两个图，并且每个图都需要有一定的结构，比如节点、边以及节点向量和边向量等。一开始需要对两个图中的节点与边做初始化编码，利用跨图的注意力机制将两个图进行关联融合以获取更加丰富的图信息；接着利用聚合函数来将图中的所有向量进行聚合，从而分别获得两个图的全局图向量；最后利用预测函数来对两个图的图向量相似度进行计算，从而判断输入的两个图是否相似。

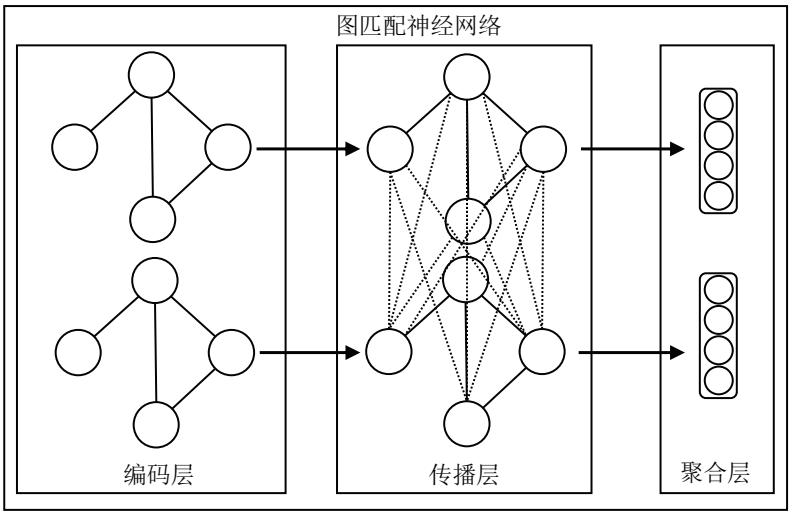


图 3-8 图匹配神经网络

如图 3-8 所示，图匹配神经网络分为三个模块：编码层、传播层和聚合层。其中编码层主要是利用多层感知器（Multiple Layers of Propagation，MLP）[65] 来对抽象语义图数据中的节点和边向量进行编码。多层感知器包含输入层、多个隐藏层和输出层，用于将原始的向量映射到图匹配神经网络的语义空间。传播层是图匹配神经网络的核心层，采用跨图注意力机制来更新图中的节点和边信息。聚合层对节点向量与边向量进行聚合，从而得到包含智能合约完整语义和语法信息的全图向量。

3.3.2　图网络更新

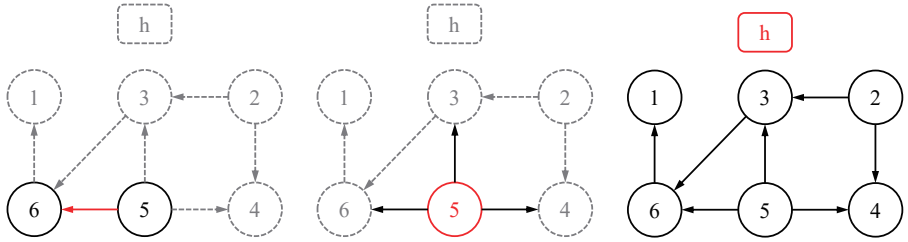
将抽象语义图中的节点和边属性映射到相应的节点向量和边向量，初始状态下源代码的抽象语义图并不包含整个图的全局属性，而图向量是通过对图网络中的节点向量和边向量逐步更新获得。在对抽象语义图进行向量化之后，可以把它表示为 $G = (V, E)$。图（Graph）是计算机中最基础的几种数据结构之一，由若干个节点（Vertex）组成，节点与节点相互连接便组成了边（Edge），其中 $V = \{v_1, \ldots, v_n\}$ 为节点的集合，每一个节

点都有特征向量 x_i。$E=\{e_{ij}\}_{i,j=1}^{n}$ 为图中所有边的集合，每一条边 $e=<v_i,v_{i+1}>$ 包含两个端点，这两个端点互为邻居节点（Neighbor）。$N(v_i)$ 包含节点 v_i 的所有邻居节点，$A=\{x_{ij}\mid x_{ij}\geq 0\}_{i,j=1}^{n}\in\mathbb{R}^{n\times n}$ 是邻接矩阵（Adjacency Matrix），连接节点 v_i 和 v_j 的边的特征向量为 x_{ij}。如式 (3.5) 和 (3.6) 中所示，图网络的编码层对节点向量和边向量进行初始编码，其中 Linear 指代线性函数，ReLU 指代整流线性单位函数，x_i 表示节点的单位向量，x_{ij} 表示边的单位向量，$h_i^{(0)}$ 表示节点在图网络隐藏层中的初始向量。

$$\mathbf{h}_i^{(0)}=\mathrm{MLP}_{\mathrm{node}}(x_i)=\mathrm{Linear}(\mathrm{ReLU}(\mathrm{Linear}(x_i))),\quad \forall i\in V \tag{3.5}$$

$$\mathbf{e}_{ij}=\mathrm{MLP}_{\mathrm{edge}}(x_{ij})=\mathrm{Linear}(\mathrm{ReLU}(\mathrm{Linear}(x_{ij}))),\quad \forall (i,j)\in E \tag{3.6}$$

图网络中的传播层采用三个步骤来逐次更新图网络中边向量、节点向量和全图向量。图 3-9 所示的是图中节点和边的更新情况，边属性更新时，将要更新的边初始向量属性与其邻接的所有节点向量进行聚合计算，然后更新到原有边的向量。如图 3-9 (a) 中所示，更新节点 5 与节点 6 之间的边向量，要同时使用到该边初始的向量和节点 5 、节点 6 的向量。对节点属性进行更新时，需要将节点的初始属性向量、该节点邻接的所有出度的边属性向量和所有入度的边属性向量进行聚合，以作为更新该节点函数的输入，从而得到新的节点属性。如图 3-9 (b) 中所示，当对节点 5 属性进行更新时，不但要使用节点 5 的初始属性，还同时要用到与节点 5 邻接的一条入度边和两条出度边的属性向量。最后进行全图属性的更新，这需要将图初始属性、图中所有的节点属性向量和所有边属性向量作为全图属性向量更新函数的输入，通过计算之后得到更新之后的全图属性向量。如图 3-9 (c) 中所示，要更新抽象语义图的全图向量，需要将图中所有的节点属性向量和边属性向量同时进行聚合运算。



(a) 边属性更新　(b) 节点属性更新　(c) 全图向量更新

图 3-9 图网络属性更新

传播层利用了跨图注意力机制来更新图中的节点和边信息，它将一组节点隐藏向量 $h_i^{(t)}$ 更新到新的节点隐藏向量 $h_i^{(t+1)}$ 中，即每个图的信息不仅会在自身进行传播，也通过跨图机制传播到另一个图中，公式如下：

$$\mathbf{m}_{j\to i} = f_{\text{message}}\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}, \mathbf{e}_{ij}\right), \forall (i,j) \in E_1 \cup E_2 \tag{3.7}$$

$$\boldsymbol{\mu}_{j\to i} = f_{\text{match}}\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}\right), \forall i \in V_1, j \in V_2, \text{ or } i \in V_2, j \in V_1 \tag{3.8}$$

$$\mathbf{h}_i^{(t+1)} = f_{\text{node}}\left(\mathbf{h}_i^{(t)}, \sum_j \mathbf{m}_{j\to i}, \sum_{j'} \boldsymbol{\mu}'_{j\to i}\right) \tag{3.9}$$

其中用于聚合边 e_{ij} 的 $f_{message}$ 是一个多层感知器，节点 j 和节点 i 的信息生成传播信息 $\mathbf{m}_{j\to i}$，f_{match} 是利用跨图注意力机制来传播两个图之间的信息，f_{node} 是采用门控循环神经单元（Gate Recurrent Unit，GRU）来将图中与该节点的相关邻域信息进行聚合，聚合的不单单是本图中其他节点与边的信息，还有另一个图中的信息，聚合主要包含两个步骤：计算注意力权重和上下文表示。

如图 3-10 所示对图 G1 中节点 i 与跨图 G2 任意节点 j，计算它们之间的注意力权重。

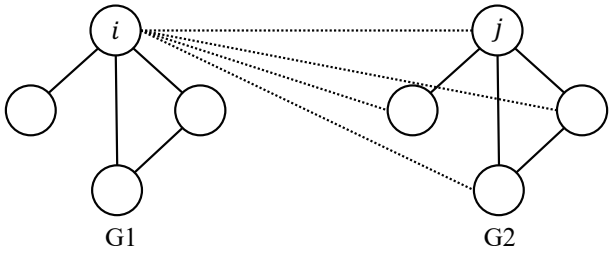


图 3-10 跨图注意力机制

式 (3.10) 通过加权求和的方式来计算图 G2 对 G1 节点 i 的上下文表示 $a_{j\to i}$，其中 s_h 为向量相似函数，本文中所采用的是余弦函数。

$$a_{j\to i} = \frac{\exp\left(s_h\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}\right)\right)}{\sum_{j'} \exp\left(s_h\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}\right)\right)} \tag{3.10}$$

式 (3.11) 中的 $\sum_j \boldsymbol{\mu}_{j\to i}$ 是用于衡量 $\mathbf{h}_i^{(t)}$ 与另一个图上与其最近临界点 $\mathbf{h}_j^{(t)}$ 的差异值。

$$\sum_j \boldsymbol{\mu}_{j\to i} = \sum_j a_{j\to i}\left(\mathbf{h}_i^{(t)} - \mathbf{h}_j^{(t)}\right) = \mathbf{h}_i^{(t)} - \sum a_{j\to i}\mathbf{h}_j^{(t)} \tag{3.11}$$

如图 3-11 所示，图中信息经过 T 轮更新之后，需要将图中所有节点的信息通过聚合层生成能够代表全局图结构信息的向量表示，如式 (3.12) 和 (3.13) 所示，其中 f_G 为全图聚合函数。

$$\mathbf{h}_{G_1} = f_G\left(\left\{\mathbf{h}_i^{(T)}\right\}_{i\in V_1}\right) \tag{3.12}$$

$$\mathbf{h}_{G_2} = f_G\left(\left\{\mathbf{h}_i^{(T)}\right\}_{i\in V_2}\right) \tag{3.13}$$

如式 (3.14) 所示，经过 T 轮更新之后图节点的隐藏向量可以用来生成全图向量。

$$\mathbf{h}_G = \mathrm{MLP}_G\left(\sum_{i\in V}\sigma\left(\mathrm{MLP}_{\mathrm{gate}}\left(\mathbf{h}_i^{(T)}\right)\right)\odot\mathrm{MLP}\left(\mathbf{h}_i^{(T)}\right)\right) \tag{3.14}$$

当图网络收敛之后，图向量就会包含节点和边的属性，也就是说它同时拥有了智能合约源代码的语法信息、语义信息和不同语句之间相互作用的数据依赖关系与控制依赖关系。此时整个图的图向量就可以表征源代码，从而作为检测智能合约是否包含漏洞的依据。

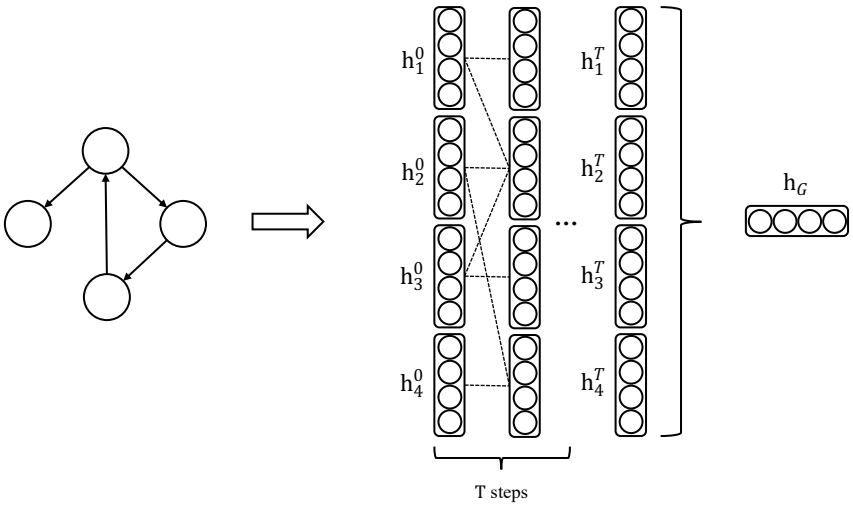


图 3-11 图节点属性向量多轮迭代

3.3.3 图相似度计算

两个抽象语义图通过图匹配神经网络生成图向量之后，就可以采用如式 (3.15) 所示的空间向量相似度计算函数来衡量两个图向量之间的相似度，进而判断是否存在漏洞，本文采用的 f_s 是余弦相似度计算函数。

$$s = f_s\left(\mathbf{h}_{G_1}, \mathbf{h}_{G_2}\right) \tag{3.15}$$

余弦相似度的计算结果在 [-1,1] 的范围之内，计算结果越接近 -1 ，两图越不相似；越接近 1 ，说明两图越相似。图匹配神经网络的整体流程如算法 3.3 所示，第 4 - 5 行是

利用多层感知器来初始化图节点和边向量，第 6 - 8 行是利用注意力机制对图中的节点和边向量进行更新，第 9 - 10 行是通过聚合函数来计算两个合约的图向量，第 14 行是通过相似度函数计算两个图向量之间的相似度。

算法 3.3 源代码图匹配神经网络算法

输入： ASG1 的词嵌入向量 V1 和 E1，ASG2 的词嵌入向量 V2 和 E2

输出： 源代码图向量 h_{G1} 和 h_{G2} 的相似度 s

1: **for** $t \leftarrow 1$ **to** T **do** ▷ 生成抽象语义图的图向量

2: **for** $i \leftarrow 1$ **to** len(V1) **do**

3: **for** $j \leftarrow 1$ **to** len(V2) **do**

4: $\mathbf{h}_i^{(0)} = \text{MLP}_{\text{node}}(x_i) = \text{Linear}(\text{ReLU}(\text{Linear}(x_i)))$

5: $\mathbf{e}_{ij} = \text{MLP}_{\text{edge}}(x_{ij}) = \text{Linear}(\text{ReLU}(\text{Linear}(x_{ij})))$

6: $\mathbf{m}_{j\to i} = f_{\text{message}}\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}, \mathbf{e}_{ij}\right)$

7: $\boldsymbol{\mu}_{j\to i} = f_{\text{match}}\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}\right)$

8: $\mathbf{h}_i^{(t+1)} = f_{\text{node}}\left(\mathbf{h}_i^{(t)}, \sum_j \mathbf{m}_{j\to i}, \sum_{j'} \boldsymbol{\mu}'_{j\to i}\right)$

9: $\mathbf{h}_{G_1} = f_G\left(\left\{\mathbf{h}_i^{(T)}\right\}_{i\in V_1}\right)$

10: $\mathbf{h}_{G_2} = f_G\left(\left\{\mathbf{h}_i^{(T)}\right\}_{i\in V_2}\right)$

11: **end for**

12: **end for**

13: **end for**

14: $s = f_s\left(\mathbf{h}_{G_1}, \mathbf{h}_{G_2}\right)$ ▷ 计算两个图向量之间的相似度

3.3.4　模型训练与检测

完成对智能合约源代码图嵌入流程和图神经网络检测模型架构的基本介绍之后，本节主要说明如何利用智能合约漏洞数据集来对图匹配神经网络进行训练，并使用训练好的图神经网络模型来检测不同类型的漏洞。本文选取开源的智能合约漏洞数据集作为图匹配神经网络的训练集。首先，对数据集中每一个智能合约进行标记，标记完的集合如图 3-12 所示，分为包含漏洞和不包含漏洞两部分，其中 label:0 指代无漏洞的集合，label:1 指代包含漏洞的集合。图匹配神经网络将一对智能合约生成的图结构数据作为输入，用 <g1,g2> 当作输入图匹配神经网络的一对合约。如果 g1 和 g2 来自智能合约漏洞数据集中的同一个集合，则有 <g1,g2> = 1，否则 <g1,g2> = -1。对数据集中所有智能合约进行合约对匹配，从而将训练集映射到包含标签为 -1 和 1 的两个合约对集合。将包含标签的合约对输入到图匹配神经网络生成两个图向量，将两个图向量的余弦相似度与合约对的标签值进行比对，利用反向传播算法和误差对图匹配神经网络中的参数进行更新，从而让图神经网络学习智能合约的漏洞特征。

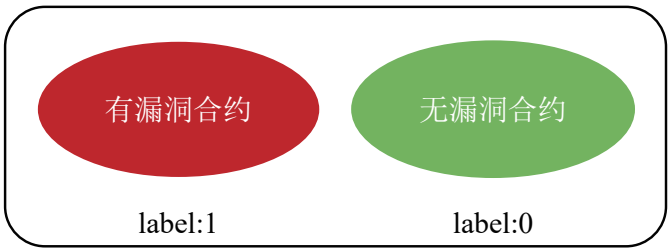


图 3-12 数据集划分

本文研究的智能合约漏洞类型有四种，每一种智能合约漏洞的特征都有所不同，所以对图匹配神经网络进行训练时会分别训练针对四种不同漏洞类型的模型。每种图匹配神经网络模型都采用相应智能合约漏洞类型的数据集进行训练，在训练好针对不同漏洞类型的图网络模型之后，就可以对智能合约进行相应的漏洞检测。首先，将待检测的智能合约转化为图结构数据，也即利用智能合约源代码生成抽象语法树、控制流图和数据流图，构建抽象语义图。然后，利用词嵌入模型对抽象语义图中的边和节点进行特征向量嵌入，从而生成图结构数据。从智能合约漏洞数据集中取出一个包含漏洞的合约，与待检测合约生成的图结构数据组成数据对，输入到检测相应漏洞类型的图匹配网络模型中，通过模型计算出它们的相似度是否超过阈值，从而判断待检测漏洞是否包含该漏洞类型。

3.4 实验评估与分析

上一小节中介绍了如何将智能合约源代码转化为适应图匹配神经网络的图结构数据，本节将通过实验的方式来论证本文提出的方法在面向智能合约源代码进行漏洞检测的有效性。首先介绍实验所用的数据集、实验平台和现有面向源代码漏洞检测的工具，然后通过常用的评估指标来比较各个检测工具的效果，最后分析了图匹配神经网络参数对检测效果的影响。

3.4.1 实验环境设置

本文面向智能合约源代码漏洞检测的实验环境为 Ubuntu 20.04 操作系统，3.7GHz 的 Intel Core i7 CPU 处理器、16 GB 内存、8 GB 显存的 GeForce Tesla T4 显卡，模型使用了 CUDA 10.2 计算库加速模型训练。实验代码采用 python 3.7 编写，基于 Pytorch 1.6 和 PyTorch Geometric 1.6.1 实现，相似度的阈值为 85%。通过 solc-ast 工具[66] 来从智能合约源代码中生成抽象语法树，进而提取源代码中控制流图、数据流图。具体实验步骤设计如下：

（1）源代码漏洞检测有效性评估：从开源数据集中收集包含四种不同漏洞类型的智能合约来组成基础数据集，并划分为对应的训练集和测试集，然后将本文提出的方法与现有面向智能合约源代码的漏洞检测工具进行比较，以论证本文方法的有效性。

（2）图网络参数选取：在本文提出方法的有效性得到验证之后，就需要对图网络模型本身参数的选取对漏洞检测效果的影响进行评估。通过探究图匹配神经网络输入的图数据嵌入向量维度、图匹配神经网络迭代次数以及图网络中隐藏层数对于漏洞检测效果的影响，进而选取出最适合面向智能合约源代码漏洞检测的图网络参数。

3.4.2　数据集与评估指标

为了测试面向智能合约源代码漏洞检测的图网络模型的有效性，本文选取了来自 Smartbug[67]、SolidiFI[68] 和其他开源数据集中带标签的智能合约。本文准备的数据集中不同智能合约漏洞类型和数量分布情况如表 3.1 所示，数据集划分为训练集和测试集，比例为 7:3。训练集用来让图匹配神经网络模型学习智能合约不同类型的漏洞特征，测试集用于评估图匹配神经网络模型对漏洞检测的泛化能力。

表 3.1 智能合约源代码数据集

	重入漏洞	时间戳依赖	区块信息依赖	Tx.Origin 漏洞
漏洞数量	1589	1356	1467	1323

在准备好数据集之后，本文选取了三种面向源代码智能合约漏洞检测工具（Slither[36]、SmartCheck[37]、DR-GCN[38]）来进行实验比较。现有漏洞检测工具的选取基于以下几个原则：（1）检测工具的源代码是能够获取的；（2）该检测工具在智能合约漏洞检测领域是具有一定知名度的，即较多漏洞检测工具在测试自身有效性时选取了该工具作为测试基准；（3）该检测工具将智能合约源代码作为漏洞检测的输入；（4）能够检测到本文提出的漏洞类型的一半及以上。

通过上述四种漏洞所构建的数据集来使用这三种已有工具分别对每种漏洞的数据集进行检测，并且对检测结果进行相应的分析、评估以及对比，从而证明本文提出漏洞检测方法的有效性。为了对漏洞检测效果进行评估，本文采用了广泛应用的指标，包括准确率、召回率、精度和 F1 分数。在对各个工具进行效果测试时，测试结果得出的混淆矩阵如表 3.2 所示。

表 3.2 混淆矩阵

实际合约标签	测试结果	
	包含漏洞	不包含漏洞
漏洞合约	TP	FN
无漏洞合约	FP	TN

表中的指标为 TP（True Positive）为真阳性，意为工具检测到的真正包含漏洞的合

约数量。FP（False Positive) 为假阳性，意为将不包含漏洞的合约错误地识别为包含漏洞的合约数量。FN（False Negative）为假阴性，意为将真正包含漏洞的合约正确识别为不包含漏洞的合约数量。TN（True Negative）为真阴性，意为将不包含漏洞的合约正确地识别不包含漏洞的合约数量。根据上面这些得到的检测数据，可以采取以下的指标对不同工具得到的检测结果进行评估：

（1）准确率（Accuracy, ACC）表示漏洞检测工具从数据集中正确检测出包含和不含漏洞的合约数占数据集中所有合约数的比例，准确率越高越好。

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}} \tag{3.16}$$

（2）召回率（True Positive Rate, TPR）意为漏洞检测工具正确识别出的包含漏洞的合约数占所有包含漏洞的合约数的比例，TPR 取值范围为 [0,1]，值越高说明该工具性能越好。

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{FN} + \mathrm{TP}} \tag{3.17}$$

（3）精确率（Precision, PRE）意为漏洞检测工具正确识别出包含漏洞的合约数与该工具报告含漏洞的合约数的比例，取值范围为 [0,1], 值越高说明该工具性能越好。

$$\mathrm{PRE} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{3.18}$$

（4）F1 分数（F1 Score）是精确率和召回率的调和平均值，能够反映漏洞检测工具的整体智能合约漏洞检测能力。

$$\mathrm{F1} \quad = \quad \frac{2 \times \mathrm{RPE} \times \mathrm{TRP}}{\mathrm{PRE} + \mathrm{TPR}} \tag{3.19}$$

3.4.3 检测效果评估

为了对本文提出的面向源代码的图匹配神经网络漏洞检测方法的有效性进行评估，将与现有的智能合约漏洞检测工具在不同类型漏洞的数据集上进行比对测试。本文提出的方法需要在训练集进行训练，所以评估工作是在已经做好标签的测试集上进行。首先，将测试集中不同类型的漏洞放到不同的文件夹中；然后，分别使用 Slither、SmartCheck、DR-GCN 和本文提出方法对各个漏洞测试集进行检测；最后，将检测工具的结果与原漏洞合约的标签进行对比，从而得到各个检测工具初始指标，即各个检测工具的 TP、FP、NP、TN 合约数目。对评估指标进行计算，分别得到各个检测工具的准确率、召回率、精确率和 F1 分数。各个工具的漏洞检测结果如表 3.3 所示，从表中数据可知本文提出的方法在面向智能合约源代码漏洞检测方面要优于对比的其他几个漏洞检测工具，这证明了本文方法在智能合约源代码漏洞检测方面的有效性。

SmartCheck 对所有能够检测到的智能合约漏洞类型都有着较低准确率和 F1 分数，这与 SmartCheck 的检测机制有很大的关系，它是先从智能合约源代码中构建中间表示形式，然后通过 XPath 模式在中间表示形式中进行搜索。SmartCheck 的这种检测机制有着很大的缺陷，一方面，这种方法缺乏准确性，因为有些漏洞类型很难通过 XPath 表达式来定义，比如重入漏洞就有很多语法难以通过 XPath 模式来表达。另一方面，由于 XPath 模式只是用来检测漏洞的特定语法格式，那么就算是包含漏洞的智能合约，只要它的语法发生变化，那么就很有可能逃脱 SmartCheck 的检测。

表 3.3 面向智能合约源代码的检测结果

漏洞类型	检测工具	ACC(%)	TPR(%)	PRE(%)	F1(%)
重入漏洞	Slither	70.51	85.93	75.83	80.57
	SmartCheck	55.70	74.95	62.08	67.91
	DR-GCN	66.21	81.78	71.08	76.06
	ASGVulDetector	**84.96**	**95.37**	**84.17**	**89.42**
时间戳依赖	Slither	74.19	89.47	77.27	82.93
	SmartCheck	40.41	77.44	37.45	50.49
	DR-GCN	50.59	84.22	47.55	60.78
	ASGVulDetector	**87.02**	**94.59**	**89.09**	**91.76**
区块信息依赖	Slither	67.77	84.46	70.43	76.81
	SmartCheck	-	-	-	-
	DR-GCN	-	-	-	-
	ASGVulDetector	**88.99**	**94.01**	**91.30**	**92.63**
Tx.Origin 漏洞	Slither	60.96	82.09	62.97	71.27
	SmartCheck	41.12	69.34	43.69	53.60
	DR-GCN	-	-	-	-
	ASGVulDetector	**81.18**	**87.58**	**88.35**	**87.97**

Slither 是采用静态分析的方式来对源代码进行检测，它通过生成中间表示形式的方式来表征智能合约语义信息，对四种不同的漏洞类型都有着较高的 F1 分数和准确度。DR-GCN 是通过将单一的智能合约生成图的表示形式，然后利用图神经网络进行训练和检测，由于其只保留源代码中很少一部分的节点和边，导致该工具在语法和语义上的表征不够充分，使得该方法虽然在召回率上有着不错的表现，但会有较高的假阳性，从而削弱了其整体漏洞检测性能。

3.4.4 检测效率评估

本节针对面向智能合约源代码的漏洞检测工具的各个漏洞类型的平均检测时间进行了评估，结果如图 3-13 所示。从图中可以看出 SmartCheck 在每种漏洞检测时间上都耗费了最长的时间，因为其 XPath 模式匹配耗时较长。Slither 的平均检测时间低于 SmartChecker，这是由于其方法生成的中间表示形式可以有效缩短检测时间。DR-GCN 的平均检测时间最短，这与其生成的图节点数量与边数量较少有关，使得它进行漏洞特征匹配时花费时间较少。本文方法需要生成将源代码转化为抽象语义图并进行图特征向量化，所耗费的时间要长于 DR-GCN，而采用的图神经网络可以为漏洞检测提升效率。

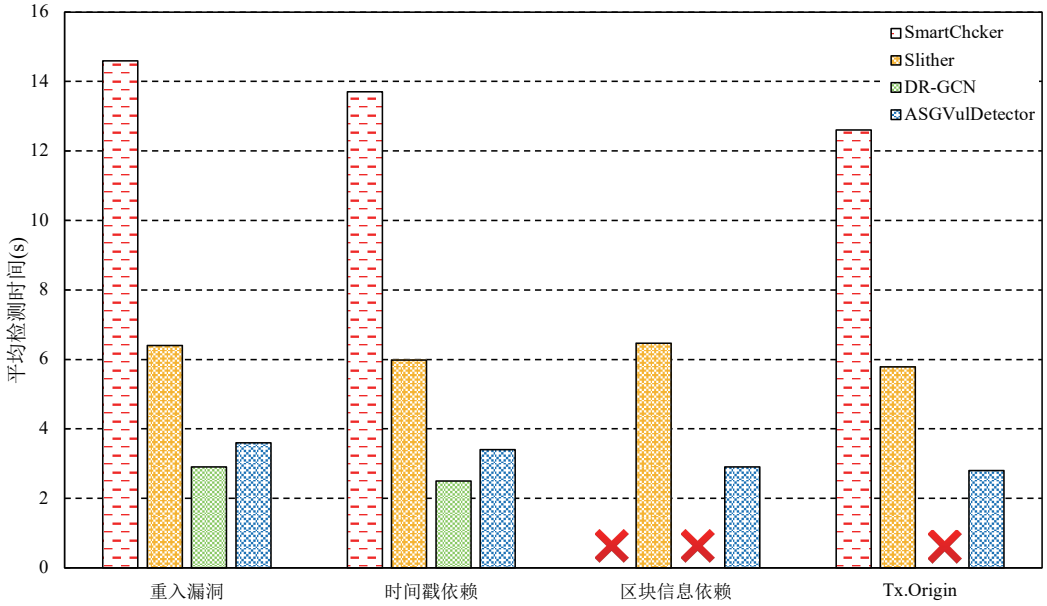


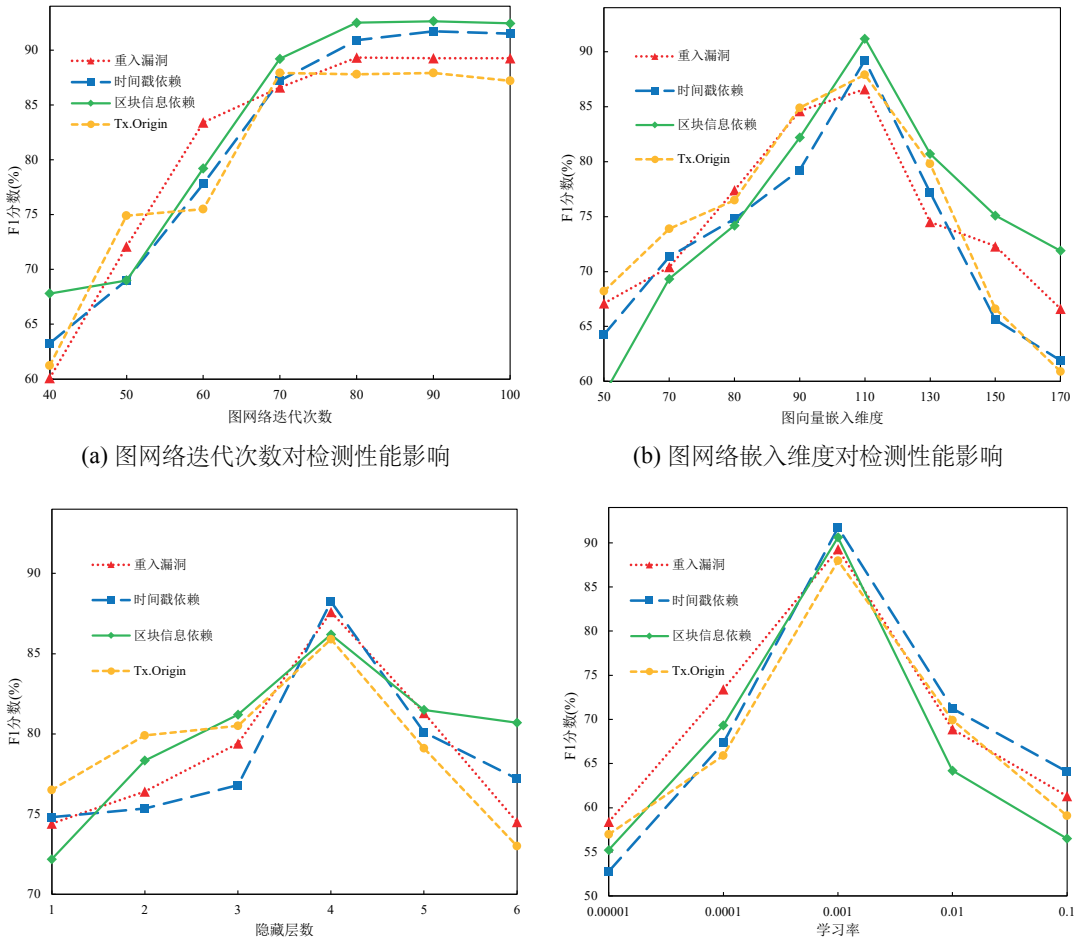
图 3-13 检测工具的平均检测时间

3.4.5 模型参数评估

图匹配神经网络模型自身的参数也会对智能合约漏洞检测的效果产生影响，因此本节将通过实验的方式来探究图网络中不同参数，如图网络的迭代次数、图向量的嵌入维度、隐藏层的层数、学习率和注意力机制对模型漏洞检测效果的影响。经过实验比对，选取了最适合面向智能合约源代码检测的模型参数，从而使得本文提出的方法在漏洞检测上能够达到最佳的性能。

为了探究图网络迭代次数对于模型检测效率的影响，本文使用了不同的网络迭代次数对图匹配神经网络模型进行训练，比较了在不同的网络迭代次数下模型对于四种智能合约漏洞的检测效果。对比结果如图 3-14 (a) 所示，利用 F1 分数来衡量模型对不同漏洞的检测效果，其中四条折线对应了四类漏洞类型。从图中可以看出，随着网络的迭代次数增加，四类漏洞的检测效果都随之增加，当迭代次数较小时，检测效果上升地尤为明显。当网络的迭代次数超过 80 之后，模型针对四类漏洞的检测效果趋于平缓。图网

络的每一次迭代就意味着将图中所有节点与边向量进行一次更新，无节制地增加图网络的迭代次数将会给模型学习增加不必要的成本，并且在迭代次数超过 80 之后检测效果没有明显提升，因此最终选取 80 作为图匹配神经网络的图网络迭代次数的参数。



(a) 图网络迭代次数对检测性能影响　(b) 图网络嵌入维度对检测性能影响

(c) 图网络隐藏层数对检测性能影响　(d) 图网络学习率对检测性能影响

图 3-14 模型参数对漏洞检测效果影响

图向量的嵌入维度对于图网络模型的检测效率是一个重要的参数，图 3-14 (b) 是图向量嵌入维度对模型检测效果的对比结果。在图向量嵌入维度较低时，检测效果随着嵌入的维度增加而上升，直到嵌入维度到 110 时，四种漏洞的检测效果基本达到最大；随着图向量嵌入维度的继续增加，检测效果却逐渐下降。当图向量嵌入维度较低时，图向量无法对漏洞特征进行有效的表征，导致图网络检测结果出现欠拟合的现象；图向量嵌入维度超过某个临界点之后，随着嵌入维度的增加，模型出现了过拟合的现象，从而导致检测效果的下降。图嵌入维度越多，模型的学习成本越高。综合比较，最终选取 110 作为图匹配神经网络的图向量嵌入维度的参数。

图匹配神经网络的隐藏层可以在线性上对不同种类的数据进行分类，所以可以通过改变图网络中隐藏层的层数来使得模型检测效果达到最佳。不同的隐藏层层数对模型漏洞检测效果的影响如图 3-14 (c) 所示，可以看出当隐藏层层数为 4 时，图网络模型对于

四种漏洞类型的检测效果达到最佳；超过 4 层之后，随着层数的增加反而检测效果出现下降。隐藏层层数过大之后，会使得模型对特征分类的能力下降，从而模型对漏洞特征的敏感度降低，造成检测效果的下降。

学习率对于任何一种深度学习模型都是相当关键的参数。不同的模型学习率对智能合约四种漏洞类型的检测效果如图 3-14 (d) 所示。当学习率太低时，图网络模型的收敛速度过慢，特征参数更新步长过短，导致模型困于局部最优解而出现过拟合；而当学习率过高，特征参数更新步长过长，模型将会在最小值附近震荡，难以达到最优解。从图中可以看出，随着模型学习率的增加，漏洞检测性能呈现先增加后降低的趋势，并在学习率为 0.001 时模型检测效果达到最佳。

图网络的注意力机制对漏洞检测也有着重要的作用，为了探究注意力机制对本文方法的影响，本文使用门控图神经网络（Gated Graph Neural Network，GGNN）模型来学习智能合约源代码中的漏洞并进行漏洞检测，其中 GGNN 与 GMN 主要区别就在于是否采用了注意力机制。

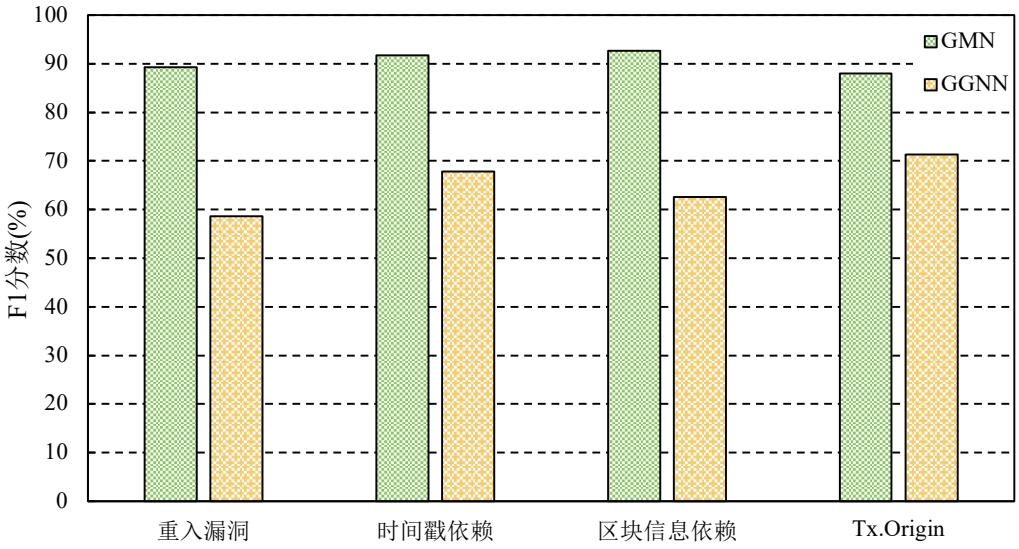


图 3-15 图网络注意力机制对检测效果影响

注意力机制的对比结果如图 3-15 所示，可以看出采用了注意力机制的 GMN 模型在每种漏洞的检测上都要优于不采用注意力机制的 GGNN 模型，这也说明注意力机制的确能够让图网络模型更加有效地学习智能合约漏洞特征。

3.5 本章小结

本章介绍了如何将智能合约源代码转化为包含语法和语义信息的抽象语义图。首先，从智能合约源代码中生成抽象语法树，在抽象语法树的基础上生成智能合约的控制流图，并从智能合约中的数据依赖关系上生成数据流图，最后在抽象语法树上添加数据流图和控制流图从而生成抽象语义图。本章同时说明了如何利用图嵌入技术将抽象语义

图转化为适合图网络分析的图结构数据，使得图神经网络能够通过抽象语义图学习智能合约中漏洞特征。

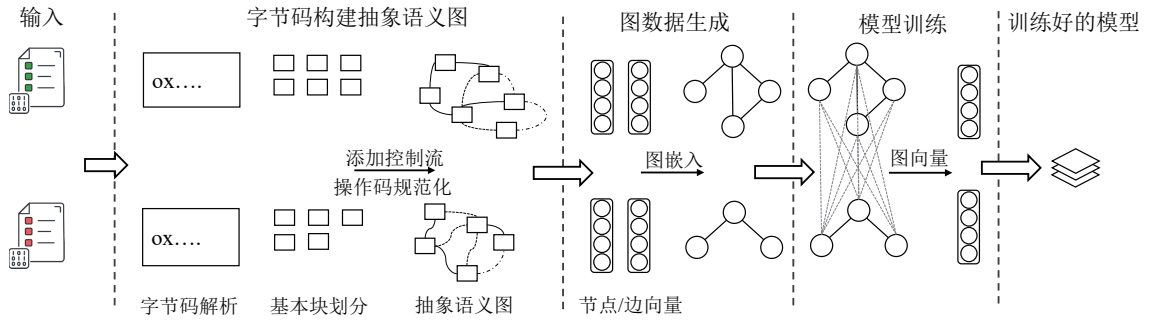
此外，本章还介绍了图匹配神经网络的整体构架，说明了图结构数据的节点和边属性向量如何在图网络中一步步传播，并且利用图网络的跨图注意力机制从初始的图数据生成能够反映全图信息的图向量。本章还阐述了图匹配神经网络的训练和模型检测的原理与过程。为了说明本文提出的检测方法的有效性，本章阐述了使用图神经网络对智能合约源代码进行漏洞检测的整体思路和实验。首先，介绍了面向智能合约源代码漏洞检测的实验环境、实验步骤、智能合约数据集划分和评估指标。之后，在已经标签过的智能合约数据集上与其他三个检测工具进行对比，并对实验结果进行分析，证明了本文所提出的方法在面向智能合约源代码漏洞检测上有着较好的时间效率和漏洞检测性能。

第四章　面向字节码的检测方法

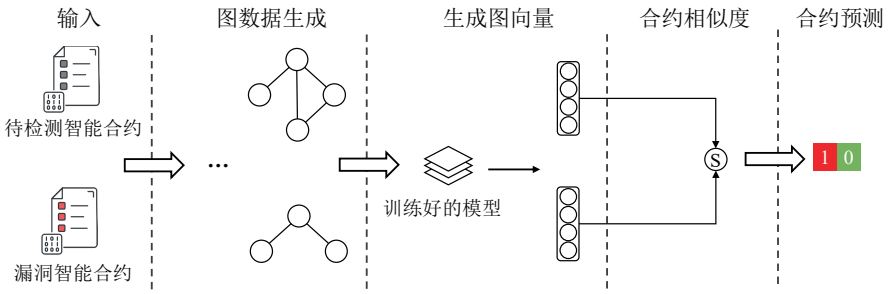
本章面向仅可获得字节码的场景介绍检测智能合约漏洞的方法，与上一章类似，在面向字节码进行漏洞检测时，首先将智能合约字节码在语法和语义层面上进行表征，从而将字节码转化为抽象语义图。随后将抽象语义图进行图嵌入生成图结构数据，再利用图匹配神经网络对字节码的漏洞特征进行学习。最后通过实验比对，论证本文提出方法的有效性。

4.1　基本框架

本文面向智能合约字节码漏洞检测思路与上一章中面向源代码漏洞检测的思路基本一致，主要区别在将字节码的图向量嵌入技术与图匹配神经网络相结合，整体检测架构如图 4-1 所示。



(a) 模型训练



(b) 模型检测

图 4-1 面向字节码漏洞检测模型框架

字节码漏洞检测系统也分为训练与检测两个阶段：训练阶段从输入的智能合约源代码中提取字节码，也可以直接将字节码作为输入，首先对字节码进行反汇编生成操作码，接着模拟以太坊虚拟机执行操作码来划分出操作码基本块并生成基本块之间的跳转

关系，接着对操作码进行规范化从而构建出字节码的抽象语义图；对字节码抽象语义图进行词嵌入生成节点向量和边向量，然后用生成的图数据对图神经网络进行训练。检测阶段与面向源代码的漏洞检测流程一样，将待检测的智能合约输入到训练好的图神经网络中与其他包含标签的智能合约进行匹配，如果相似度超过阈值则认为该合约包含相应类型的漏洞。

4.2 基于字节码的图数据结构生成

不同于源代码，字节码无法直接提取出代码的抽象语法树信息，但从字节码中获取控制结构与语义信息是实现漏洞检测的关键。本文提出一种从字节码尽可能还原智能合约语法和语义信息的图结构的方法，主要有以下几个步骤：（1）解析字节码，对智能合约字节码二进制表示中的代码与数据进行切分，从而将字节码转化为操作码；（2）构建基本块，将表示同一组功能的操作码归类到一个基本块中，从而实现语义上的连续，并且构建基本的顺序流关系；（3）通过对字节码进行符号执行操作，构建基本块之间更加复杂的控制流关系；（4）将生成的控制流与基本块进行结合，从而构建字节码的抽象语义图。

4.2.1 字节码解析

在以太坊中运行智能合约，首先需要将智能合约源代码通过编译器编译为以太坊字节码，然后通过以太坊虚拟机（Ethereum Virtual Machine，EVM）执行该字节码。以太坊的黄皮书[3] 中定义了操作码及其语义的完整列表，截止到目前共有 140 多个唯一的操作码，不同的虚拟机版本之间几乎没有变化，以太坊虚拟机使用这些操作码来执行具体的操作。如表 4.1 所示，依据操作码实现的功能不同，可以将操作码分为停止与算术运算、逻辑判断与位运算、加密操作、合约环境信息、栈操作、内存与存储操作、控制流操作、入栈操作、栈数据交换、日志操作和系统操作等。

以太坊虚拟机是一种基于栈的虚拟机，它维护着运行时栈（Stack）、临时内存（Memory）和持久存储器（Storage）。栈主要是用于处理大小为 32 字节的数据，以太坊虚拟机的栈最大高度为 1024，绝大部分操作码指令只能够操作栈中的数据，比如 PUSH 将数据推入栈中，ADD 从栈顶取出两个数据进行相加后将结果推入栈中；内存用来处理超过 32 字节的数据，其功能相当于传统系统中的堆，一般采用 MLOAD 和 MSTORE 来操作内存中的数据；持久存储器作为以太坊区块链的组成部分，栈和内存生命周期与智能合约相同，而存储器在智能合约执行完毕并销毁之后仍然维护着智能合约的状态，常用 SLOAD 与 SSTORE 来操作持久存储器中的数据。

表 4.1 以太坊虚拟机操作码分类

操作码十六进制值	操作码功能类别	具体操作码
0x00	停止操作	STOP
0x01-0x0B	算术操作	ADD,MUL,SUB,DIV,SDIV MOD,SMOD,ADDMOD,MULMOD EXP,SIGNEXTEND
0x10-0x19	逻辑判断	LT,GT,SLT,SGT,EQ,ISZERO
0x1A-0x1D	位操作	BYTE,SHL,SHR,SAR AND,OR,XOR,NOT
0x20	加密操作	SHA3
0x30-0x3E	环境信息	ADDRESS,BALANCE,ORIGIN,CALLER CALLVALUE,CALLDATALOAD,CALLDATASIZE CALLDATACOPY,CODESIZE,EXTCODESIZE
0x40-0x45	区块信息	BLOCKHASH,COINBASE,TIMESTAMP NUMBER,DIFFICULTY,GASLIMIT
0x50-0x55	堆栈、内存、存储器操作	POP,MLOAD,MSTORE,MSTORE8 SLOAD,STORE
0x56-0x5B	控制跳转操作	JUMP,JUMPI,PC,MSIZE,GAS,JUMPDEST
0x60-0x9F	栈数据操作	PUSH1-PUSH32,DUP1-DUP16,SWAP1-SWAP16
0xA0-0xA4	内存日志操作	LOG0-LOG4
0xF0-0xFF	系统调用操作	CREAT,CALL,CALLCODE,RETURN DELEGATECALL,CREATE2,STATICCALL REVERT,INVALID,SELFDESTRUCT

以太坊字节码是由十六进制的数字组成的一组字符串，主要是由 Creation Code 和 Runtime Code 组成。Creation Code 是智能合约编译器生成，当合约部署到区块链上时自动生成，以太坊虚拟机只执行一次，主要用于确定部署的智能合约初始状态。Runtime Code 可以分为三个部分，最重要的一段是包含了以太坊虚拟机执行的操作码；第二段是可选，包括字符串或常量数组等静态数据；最后一段是包含元数据，比如编译器的版本和用于保证智能合约一致性的哈希值，元数据随着智能合约编译器的版本在不断地变化。如图 4-2 所示，将 Basic 合约通过智能合约编译器 solc 编译获取字节码，将其中的 Runtime Code 转换为操作码即可获取智能合约字节码中隐含的最基础语义信息。

当执行具体事务时，以太坊虚拟机首先将字节码拆分为字节，每个字节代表一个操作码或者一个操作数。如图 4-3 所示，当处理字节码 0x6070604001 时，以太坊虚拟机首先将字节码拆分为单个字节（0x60、0x70、0x60、0x40、0x01），按照顺序依次执行第一个字节 0x60，即操作码 PUSH1，PUSH1 功能为将一字节数据推送到 EVM 栈顶，这意味着后续一个字节 0x70 将作为数据被推入栈顶。以太坊虚拟机读取下一个 0x60 并将

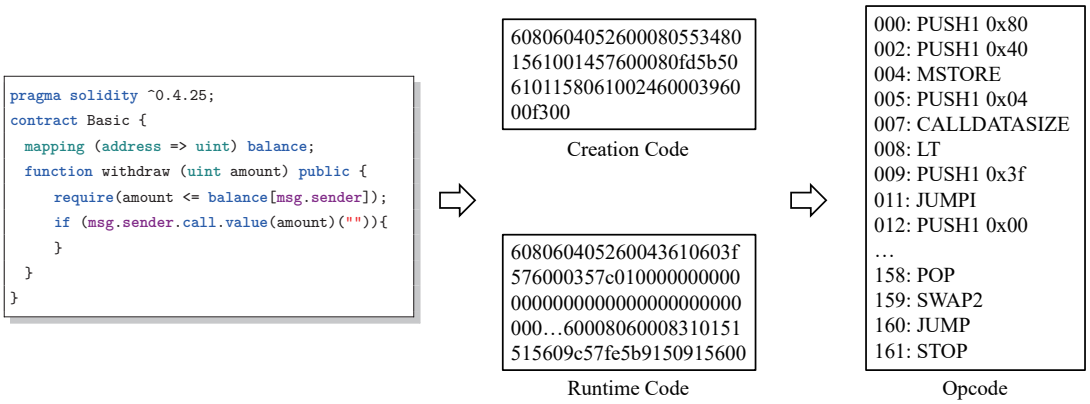


图 4-2 智能合约源代码生成操作码

0x40 推入栈。最后执行 0x01，即操作码 ADD，ADD 从栈顶获取两个值，并将它们的运算结果 0xB0 放入栈顶。

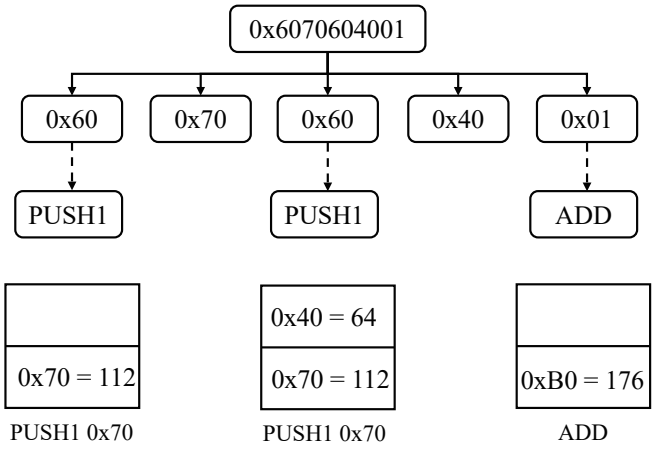


图 4-3 字节码拆分与栈数据状态

4.2.2 控制流图

智能合约字节码转换成适合图神经网络输入数据类型，需要进一步提取智能合约字节码中的控制流信息，不同于从源代码中构建控制流图，智能合约字节码控制流图的生成依赖于操作码。首先，将所有的操作码划分成一个个基本块，这些基本块就是控制流图的节点，然后通过操作码之间的跳转关系在不同基本块之间生成不同类型的控制流边。基本块是由一系列连续的操作码组成，每个基本块都包含一个入口和一个出口，入口就是第一个操作码，出口就是最后一个操作码。操作码 JUMP、JUMPI、STOP、REVENT、RETURN、INVALID、SELFDESTRUCT 用来标记一个基本块的结尾，而 JUMPDEST 一般出现在新基本块的开头。按照基本规则将图 4-2 中的操作码进行基本块划分，可以得到图 4-4 所示的基本块。

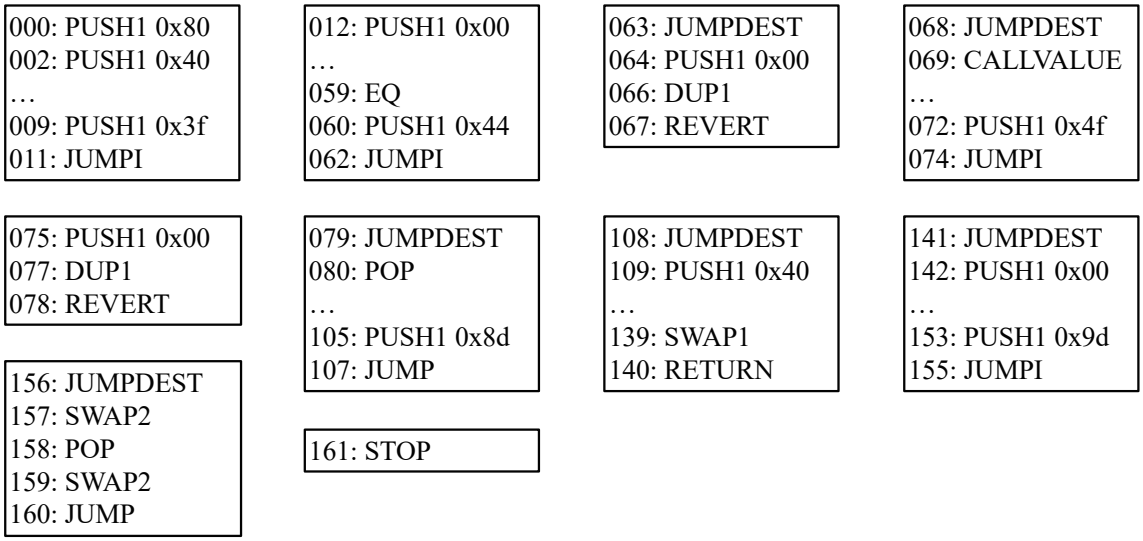


图 4-4 字节码基本块划分

操作码被划分出基本块之后，接下来要在基本块之间生成边。以太坊虚拟机通过栈来执行操作码，通过分析基本块中的操作码来确定两种基本跳转：无条件跳转和有条件跳转。当 JUMP 操作码紧跟在 PUSH 操作码之后就是无条件跳转，跳转的偏移地址就是 PUSH 操作码后面的内存地址，如图 4-4 中第 107 条指令跳转位置就是指令 105 中指明的 0x8d。当 JUMPI 操作码紧跟在 PUSH 操作码之后就是条件跳转，此跳转的目标取决于 EQ 的结果，如果 EQ 为真，则跳转入 PUSH 操作码指定的目标；如果 EQ 为假，则直接跳转到紧邻的基本块；如图 4-4 中第 62 条指令为 JUMPI，如果前面 EQ 为真，则跳转到 60 指令中指定的 0x44，否则跳转到 63 指令。

当基本块的最后一条操作码为 REVERT、SELFDESTRUCT、RETURN、INVALID, STOP 时，则说明基本块没有后续的跳转位置，故而直接退出。为了使得基本块在语义上保持连贯，在基本块之间依据操作码的顺序添加顺序流边，从而让整个控制流图保持连通。当 JUMP 操作码前不存在 PUSH 操作时，就采用模拟符号执行的方法来确定该基本块的跳转目标，最后生成的字节码控制流图如图 4-5 所示。

4.2.3　抽象语义图生成

为了让字节码控制流图能更好地表达智能合约语义信息，需要对操作码做进一步规范化操作，使得不同智能合约字节码生成的控制流图有一定的通用性，并为后续抽象语义图的图嵌入打下基础。

在生成的操作码中有很多表示跳转位置和操作数的常数，比如 PUSH 操作码后就跟着以太坊内存地址的偏移量，不同智能合约编译生成的操作码中这些偏移量的值并不相同，直接进行图嵌入将会减弱特征表达的有效性，进而降低模型的检测效率。因此需要为不同操作码后的数据添加统一的标签，从而实现数据的规范化，每个数据的具体标签取决于它对应的操作码。如表 4.2 所示，BLOCKHASH 和 COINBASH 等操作码

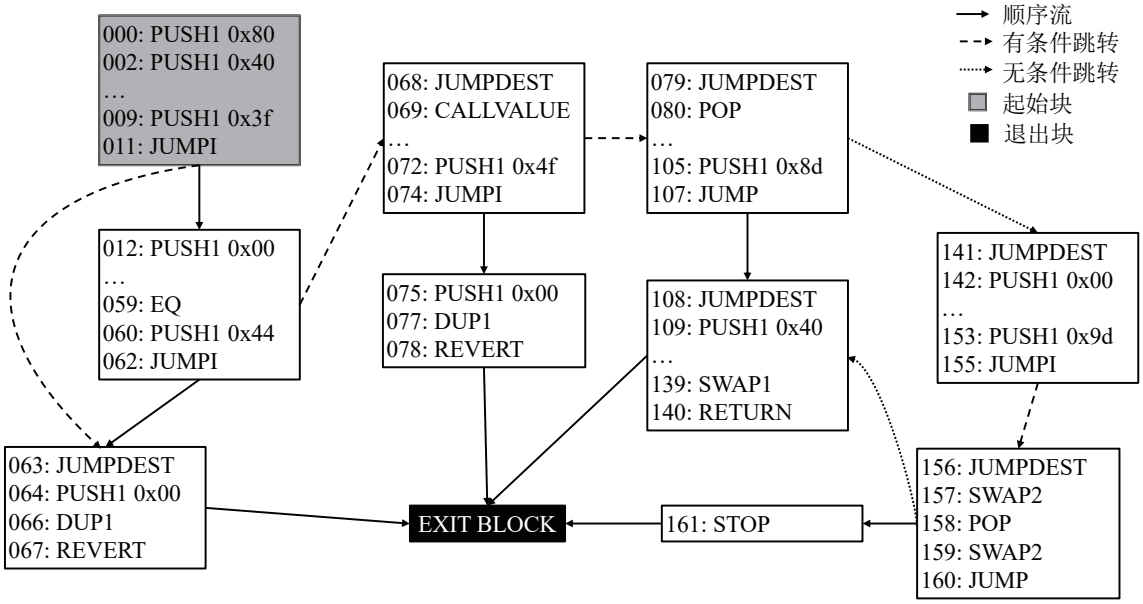


图 4-5 字节码的控制流图

之后的数据添加的标签为 BlockData，LT 和 GT 等逻辑判断操作码生成的数据的标签为 LogicData，ADD 和 MUL 等算术运算操作码生成的数据标签为 ArithData。

表 4.2 操作码与其对应的数据标签

操作码	数据标签
ADD,MUL,SUB,EXP,SIGNEXTEND	ArithData
BLOCKHASH,COINBASE,TIMESTAMP, NUMBER,DIFFICULTY,GASLIMIT	BlockData
LT,GT,SLT,SGT,EQ,ISZERO	LogicData
MLOAD	MemData
SLOAD	StorData
BYTE,SHL,SHR,SAR AND,OR,XOR,NOT	BitData

在一些操作码的结尾会有数字来表示当前操作数据的字节长度，但操作码对应数据的具体长度对于字节码的语义解析并没有影响。例如 PUSH32 和 PUSH1 在语义上实现的功能一致，在词嵌入过程中会被当作两个不同操作码，因此需要对此类带有操作数长度的操作码进行精简，保持语义上的一致性。DUP、POP 和 SWAP 操作码能够直接对以太坊虚拟机中的栈数据进行操作，但这些操作码与智能合约的语义并无实际影响，也就是说这些操作码在进行特征匹配时无关紧要，就可以将它们进行移除。表 4.3 中所示就是规范化后的一些操作码。

表 4.3 规范操作码

操作码	规范后的操作码
LOG0-LOG4	LOGX
PUSH1-PUSH32	PUSHX
DUP1-DUP16	-
SWAP1-SWAP16	-
POP	-

从智能合约字节码生成字节码抽象语义图的全部流程如算法 4.1 中所示。第 1- 2 行是从字节码中反汇编生成操作码，第 3 - 10 行是根据操作码指令划分出基本块，第 11 - 13 行是通过模拟以太坊虚拟机执行操作码从而在基本块之间生成各类控制流，第 14 - 18 行是对操作码中的数据字节替换为与操作码有关的标签，并对操作码指令进行规范化，从而生成字节码的抽象语义图。

算法 4.1 字节码构图过程算法

输入: 智能合约字节码 Bytecode
输出: 字节码的抽象语义图 BASG

```
1:  从字节码 Bytecode 拆分出 RuntimeCode
2:  将 RuntimeCode 反汇编为操作码 Opcodes
3:  for opcode in Opcodes do                                  ▷ 划分基本块
4:      if opcode is [JUMP JUMPI STOP REVENT RETURN INVALID SELFDESTRUCT]
    then
5:          BASG.BasicBlocks.add(Block)                       ▷ 添加当前基本块
6:          Block 置空
7:      else
8:          Block.add(opcode)                                 ▷ 当前基本块添加操作码
9:      end if
10: end for
11: BASG.addFallEdges()                                       ▷ 在基本块之间添加边
12: BASG.addConditionsEdges()
13: BASG.addUnconditionsEdges()
14: for block in BASG.BasicBlocks  do                         ▷ 规范化操作码
15:     for opcode in block  do
16:         Normalize(BASG.block[opcode])
17:     end for
18: end for
```

4.2.4 图特征向量化

上一小节中介绍了如何从智能合约字节码中构建出包含智能合约语义信息的字节码抽象语义图，为了接下来便于对智能合约字节码进行图神经网络漏洞检测，需要将字节码抽象语义图进行词嵌入，从而实现抽象语义图的节点特征向量化和边特征向量化。

字节码抽象语义图的一个节点包含多条操作码语句，有一些操作码之后还紧跟着规范后的操作数，为了尽可能地保留整个节点的语义信息，将该节点中所有操作码和操作数作为一个单位进行词嵌入。首先，遍历训练集中所有智能合约生成的抽象语义图，并将其中所有文本作为语料库，来训练 CBOW 模型[54]，然后，利用训练好的 CBOW 模型来字节码抽象语义图的节点和边生成特征向量。字节码抽象语义图的图特征向量的生成过程如算法 4.2 所示，其与面向源代码的抽象语义图的图特征向量算法 3.2 思路基本相同，主要区别在用于训练 Word2vec 模型的语料库不同。

算法 4.2 字节码词嵌入过程算法

输入： 字节码的抽象语义图 BASG

输出： 字节码抽象语义图节点嵌入向量 V 和边嵌入向量 E

```
对 Word2vec 模型进行训练
V, E ← init(BASG)                          ▷ 将 BASG 的字节码的节点和边进行初始编码
for X_Node in V do                         ▷ 对节点进行词嵌入
    V.X ← Word2vec(X_Node)
end for
for X_Edge in Edge do                      ▷ 对边进行词嵌入
    Edge.X ← Word2vec(X_Edege)
end for
```

4.3 实验评估与分析

在对字节码抽象语义图进行词嵌入之后，就可以得到适用于图神经网络的图结构数据, 接下来利用图匹配神经网络来对智能合约字节码进行漏洞检测，主要的漏洞检测流程与小节 3.3 中面向源代码漏洞检测并无太大差别：首先，用训练集对图匹配神经网络进行训练；然后，利用训练好的图匹配神经网络模型对测试集中的智能合约进行检测；最后，将检测得到的结果与其他漏洞检测工具进行比较，以此评估本文提出的面向字节码漏洞检测方法的有效性。

4.3.1 实验环境设置

面向字节码漏洞检测的实验环境与小节 3.4.1 中的实验环境相同，为了与面向源代码漏洞检测保持连贯性，依然沿用了相同源代码数据集，在面向字节码漏洞检测时，需

要将数据集中智能合约源代码编译为字节码，将这些带有标签的字节码作为数据集。数据集的划分同样分为训练集与测试集，划分比例依然保持 7：3。将 Securify[35]、ContractFuzzer[23] 和 Oyente[27] 作为面向字节码漏洞检测方法的对比工作，这些检测工具的输入数据类型都是字节码，并且在智能合约漏洞检测领域有一定知名度。依然选取准确率 ACC、召回率 TRP、精确率 PRE 和 F1 分数作为评估漏洞检测工具有效性的评估指标。

4.3.2　检测效果评估

本文面向字节码的漏洞检测工具 BSGVulDetector 与其他工作对比结果如表 4.4 所示。从表中不难看出，本文提出的面向字节码的漏洞检测方法在准确率、精确率和 F1 分数等指标上都领先于其他检测工具，因此可以说明本文提出的方法在面向字节码的漏洞检测上依然有着良好的性能，同时也证明本文提出的字节码抽象语义图能够有效地表征智能合约字节码的语义特征。

表 4.4 面向智能合约源代码的检测结果

漏洞类型	检测工具	ACC(%)	TPR(%)	PRE(%)	F1(%)
重入漏洞	Oyente	41.64	74.02	42.50	54.00
	Securify	53.62	77.46	54.42	63.93
	ContractFuzzer	37.94	67.12	31.36	42.75
	BASGVulDetector	**80.59**	**92.73**	**80.00**	**85.90**
时间戳依赖	Oyente	43.24	76.62	38.45	52.55
	Securify	52.06	84.62	50.00	62.86
	ContractFuzzer	32.96	82.94	28.36	40.70
	BASGVulDetector	**79.28**	**92.17**	**81.36**	**86.43**
区块信息依赖	Oyente	-	-	-	-
	Securify	49.40	72.15	53.75	61.60
	ContractFuzzer	30.52	58.08	28.75	38.46
	BASGVulDetector	**85.04**	**93.66**	**86.09**	**89.71**
Tx.Origin 漏洞	Oyente	-	-	-	-
	Securify	47.12	71.93	51.00	59.68
	ContractFuzzer	-	-	-	-
	BASGVulDetector	**81.56**	**86.94**	**89.81**	**88.35**

Oyente 是基于符号执行的检测工具，它能够检测到的字节码漏洞较少，检测重入漏洞的效率不高，主要是由于它在检测过程中过于依赖 call.value，从而难以应付其他的

情况。ContractFuzzer 将合约生成的所有调用合并成一个调用池，然后随机地从调用池中选择一个调用来对该合约进行模糊测试，但由于测试用例生成的随机性，导致能够涵盖的系统行为比较有限，从而无法达到理想的路径覆盖率，难以找出合约中潜藏着的漏洞。Securify 在本次评估结果中相较于 Oyente 和 ConractFuzzer 有着较好的性能，但由于 Securify 提供了包含警告的漏洞检测报告，所以有着较高的假阳性。

4.3.3 检测效率评估

将面向智能合约字节码检测工具对各个漏洞类型的平均检测时间进行了记录，结果如图 4-6 所示。从结果来看，本文提出的方法在时间效率上大大领先其他漏洞检测工具。Securify 首先对字节码进行反编译然后分析其语义事实，接着利用 Datalog 语言编写的安全模式来对智能合约代码进行检查漏洞，这使得其在漏洞检测上耗费不少时间，导致其成为耗时最长的工具。ContractFuzzer 基于模糊测试，它在不断随机生成输入种子上花费的时间较长，导致其平均分析一个合约时间为 340s 左右。Oyente 是基于符号执行的，符号执行需要探索程序的可执行路径并不断进行约束求解，这一系列过程花费的时间较多，从而使得它平均分析一个合约时间在 30s 左右。本文提出的面向字节码的漏洞方法依然保持了相对较低的时间开销，平均检测一个智能合约的时间为 4.2s。

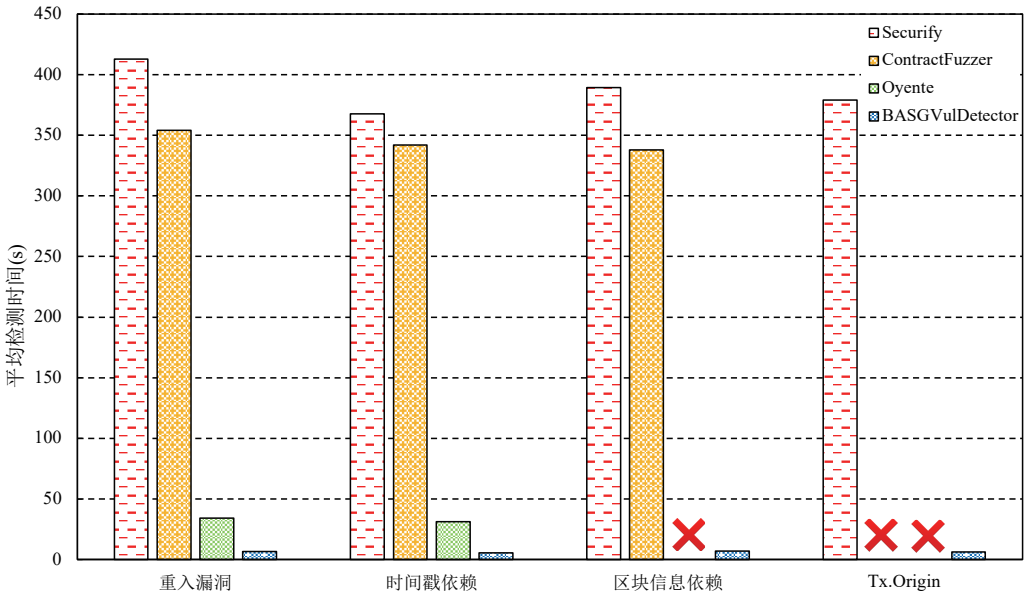
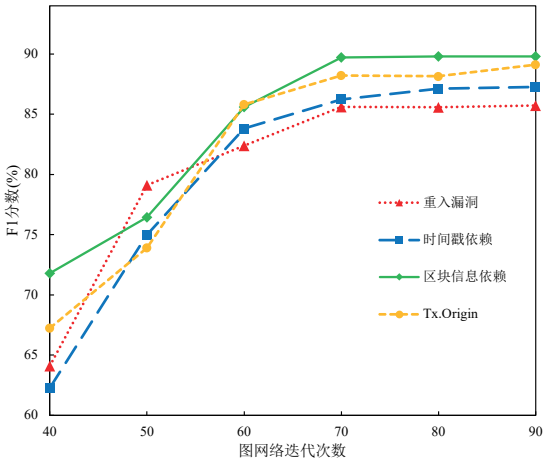


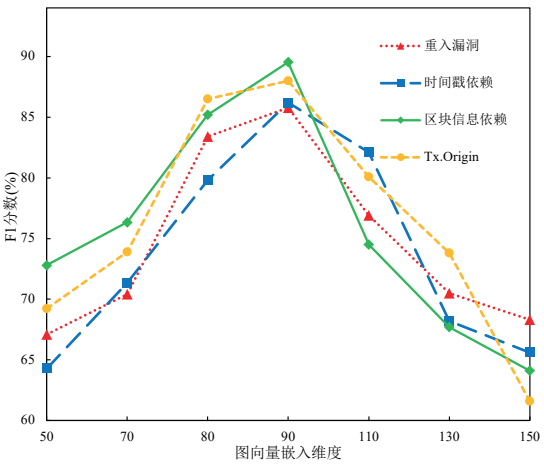
图 4-6 漏洞检测工具的平均检测时间

4.3.4 模型参数评估

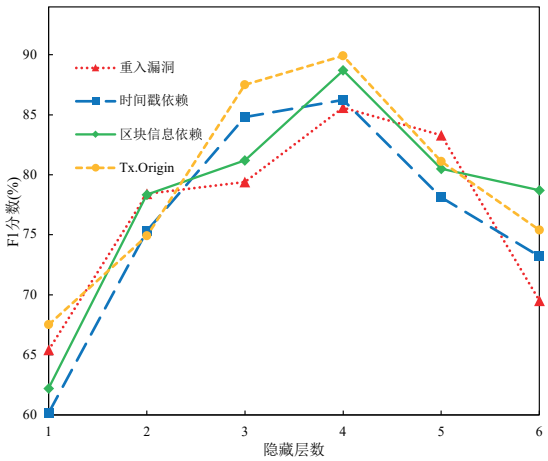
图网络模型不同参数对面向字节码漏洞检测效果的影响如图 4-7 所示。从图中可以看出，当迭代次数为 70、嵌入维度为 90、隐藏层数为 4、学习率为 0.001 时，面向智能合约字节码的图匹配神经网络模型的漏洞检测性能达到最佳。



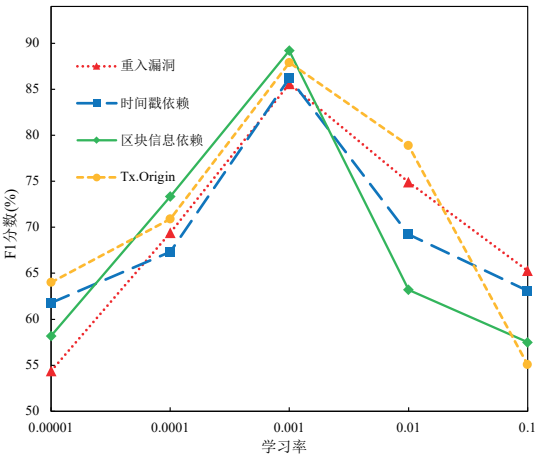
(a) 图网络迭代次数对检测性能影响



(b) 图网络嵌入维度对检测性能影响



(c) 图网络隐藏层数对检测性能影响



(d) 图网络学习率对检测性能影响

图 4-7 模型参数对漏洞检测效果影响

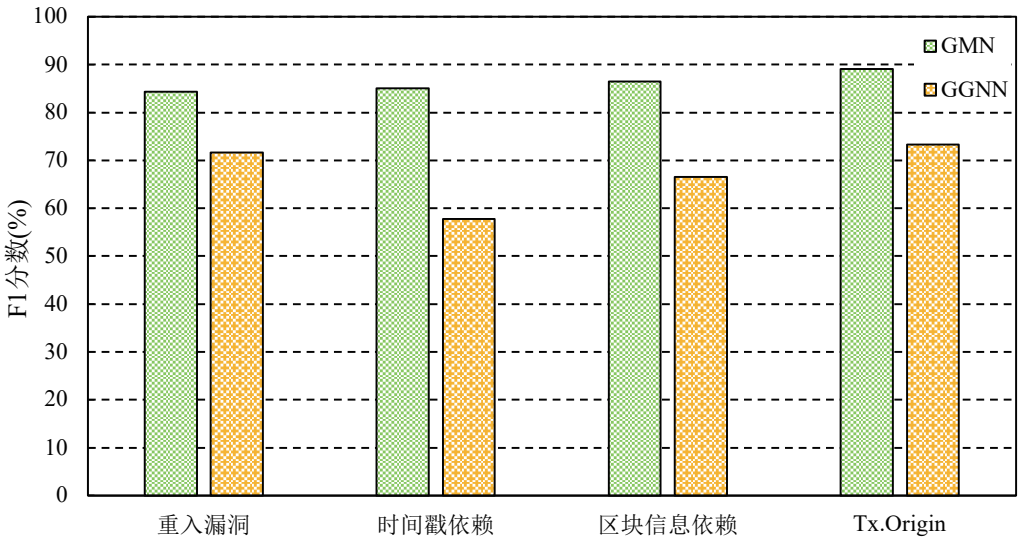


图 4-8 图网络注意力机制对检测效果影响

为了探究注意力机制对字节码的漏洞检测的影响，同样使用 GGNN 模型和 GMN 模型分别来学习智能合约字节码中的漏洞特征并进行漏洞检测，评测的结果如图 4-8 所示，说明在字节码漏洞检测上注意力机制同样能够帮助图网络模型更加有效地学习漏洞特征。

4.4 本章小结

本章介绍了从智能合约字节码中生成包含语法和语义特征的字节码抽象语义图的全部流程。首先，说明了字节码如何反汇编为操作码，依据不同操作码指令来划分出不同的基本块，并按照不同种类的基本块生成不同类型的跳转边，从而构建出字节码的控制流图。接着，对字节码中的指令进行规范化，将并无实际语义的操作码删除和同种语义的操作进行替换，从而构建出基于字节码的抽象语义图。最后，利用词嵌入技术将字节码的抽象语义图转换为图数据结构。

本章还介绍了面向智能合约字节码进行漏洞检测的方法，主要的检测流程与面向源代码的漏洞检测方法相似，并与其他漏洞检测工具进行了比较和评估，论证了本文提出的抽象语义图结构能够有效地反映智能合约字节码的语义特征，同时也说明了注意力机制在字节码漏洞检测中也发挥着重要作用，进而证明了本文提出的方法在漏洞检测上有着较好的性能和较低的时间开销。

第五章　系统设计与实现

本章在面向智能合约源代码和字节码漏洞检测方法的基础上设计并实现了一个智能合约漏洞检测系统，此系统旨在将两个检测方法结合，从而提高智能合约漏洞检测的准确率。此外，收集了真实世界中智能合约来构建数据集，使用训练好的两种图匹配神经网络模型和搭建的系统分别对真实智能合约进行漏洞检测，以此来评估本文提出的方法和系统在真实场景下的有效性。

5.1　系统架构

智能合约漏洞检测系统的核心思想是从智能合约源代码和字节码两个角度分别对智能合约进行检测，并将两种方法检测到的结果进行交叉验证，如果两种检测方法都判定该合约包含特定类型的漏洞，则为该合约标记漏洞类型的确定标签；若只有一种检测方法判定合约包含漏洞，则为该合约标记上该漏洞类型的警告标签。图 5-1 为本文设计的智能合约漏洞检测系统架构。

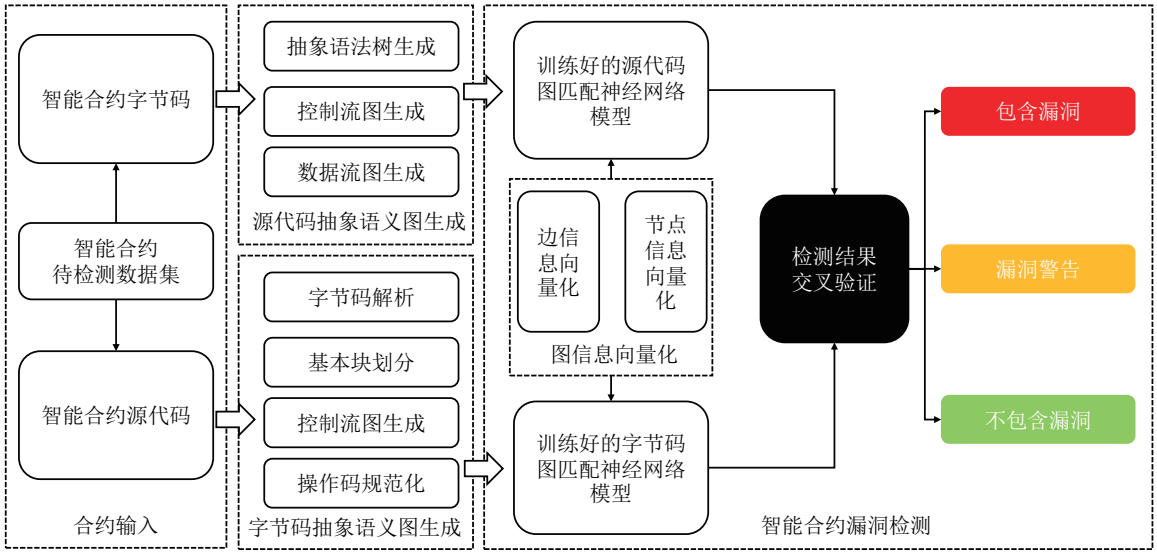


图 5-1 智能合约漏洞检测系统架构

本文实现的智能合约漏洞检测系统首先从待检测的智能合约数据集中取出一个合约，依据该合约的承载形式，采用不同的抽象语义图生成模块来将字节码或源代码转化为抽象语义图。接着，对抽象语义图进行相应的图表征信息向量化，从而将抽象语义图节点和边编码为特征向量。然后，将编码好的图数据结构输入到相对应的图匹配神经网络模块中，得到该智能合约的漏洞检测结果。最后，利用交叉验证比对两种方法的检测结果是否一致，并据此为该合约打上不同的漏洞类型标签。

5.2 系统模块设计

智能合约漏洞检测系统按照功能的不同，分为五个主要的模块：用户界面模块、图信息向量化模块、源代码构图模块、字节码构图模块和图匹配神经网络模块，图 5-2 为智能合约漏洞检测系统的模块划分示意。

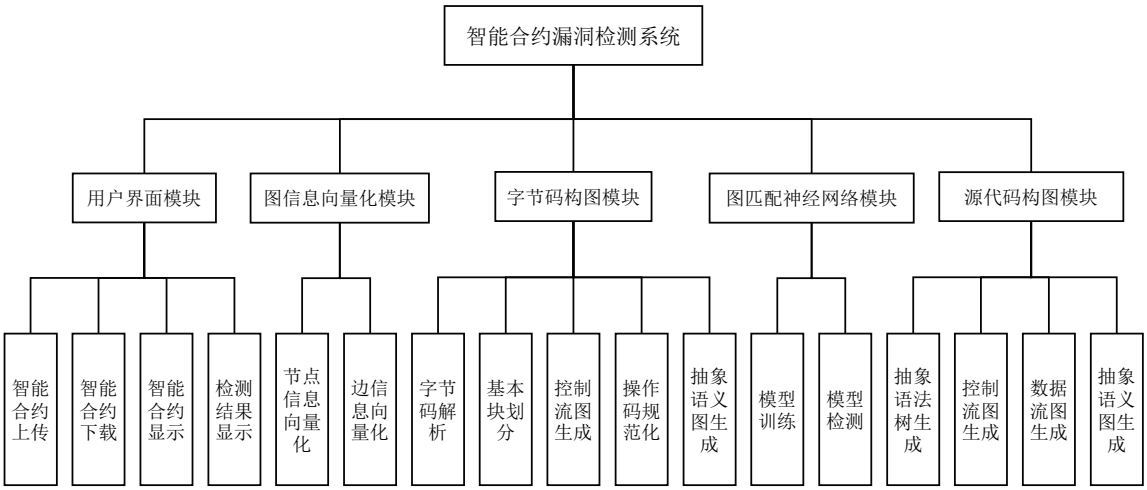


图 5-2 智能合约漏洞检测系统模块划分

其中，用户界面模块与用户进行直接交互，主要功能包括智能合约的下载与上传，同时可以展示智能合约的源代码与字节码，并且将漏洞检测的结果显示出来。系统包含两个构图模块，分别是字节码构图模块和源代码构图模块，能够根据智能合约的承载形式，将智能合约转化为相应的抽象语义图。图信息化模块用于将生成的抽象语义图进行向量化表征，也就是对抽象语义图的节点与边进行词嵌入向量编码。图匹配神经网络模块包含训练和检测两个功能，可以利用标记好的数据集对模型进行训练；也可以用训练好的图网络模型对智能合约生成的图结构数据进行漏洞检测，并输出智能合约漏洞检测结果。

5.2.1 系统用户界面模块

用户界面模块是为用户提供一个直观的交互 UI 界面，从而使得用户能够直接使用本文提出的方法来对智能合约漏洞进行检测，系统的用户界面如图 5-3 所示。智能合约漏洞检测系统用户界面可以按照功能划分为四个部分，智能合约输入模块、智能合约展示模块、检测漏洞类型选择模块和漏洞检测结果展示模块。用户能够选择多种智能合约输入方式，可以直接在用户界面上输入智能合约源代码或字节码，也可以通过提交文件的方式上传智能合约，还可以输入智能合约在 Etherscan[69] 上的在线地址来导入智能合约。

系统后台将会依据输入的文件类型采用不同的模型对智能合约进行检测，前端输入智能合约的分析流程如图 5-4 所示。用户通过前端界面直接输入，如果输入的是源代码，

图 5-3 智能合约漏洞检测系统 UI 界面

首先分析源代码是否为合法源代码，如果为合法则继续编译为字节码，将源代码和编译后的字节码显示到前端界面上，否则报错。

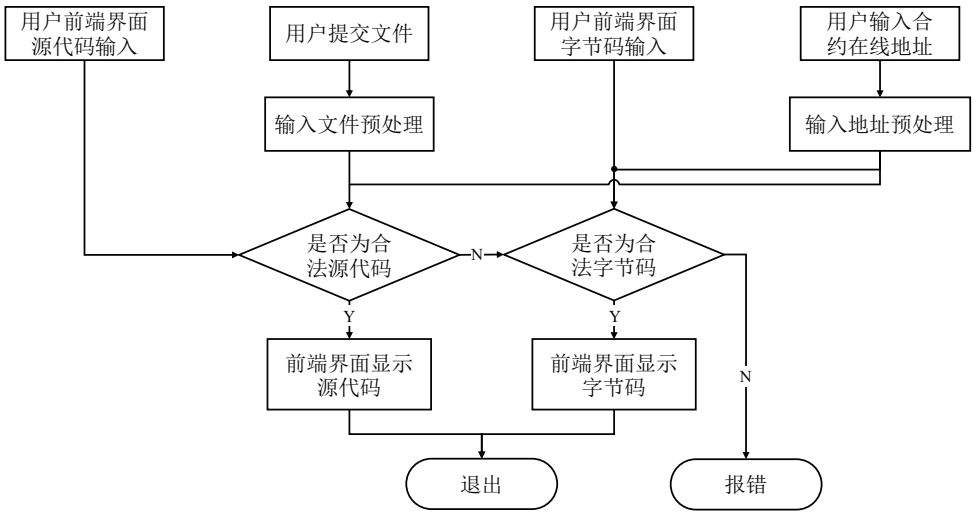


图 5-4 系统分析用户输入流程

如果前端界面输入的是字节码，则首先验证是否为合法的字节码，如果合法则将字节码显示到前端界面上，否则报错。若用户直接提交文件，后台忽视文件的后缀名，根据文件开头的字符简单判断是否为字节码，如果不为字节码则验证是否为合法的源代码，进而验证是否为合法的字节码，如果都不合法则报错。当用户通过输入智能合约在

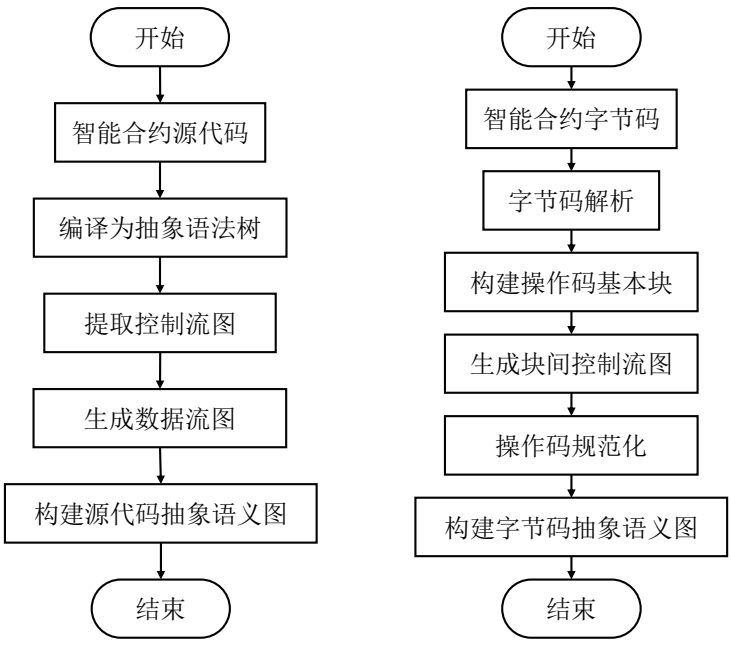
线地址时，后台分析该地址是否合法，然后从 Etherscan 上抓取相对应的字节码和源代码，如果无法获取则报错。

5.2.2 智能合约构图模块

智能合约构图模块分为面向源代码构图模块和面向字节码构图模块，两个构图模块的目标都是将智能合约转化为能够表征语法和语义信息的抽象语义图。依据源代码和字节码的形式不同，两种构图模块中具体流程也不同。

其中面向源代码构图模块的流程如图 5-5(a) 所示，首先从系统前端页面中获取要检测的智能合约源代码，接着将源代码编译为抽象语法树，深度遍历抽象语法树，记录抽象语法树中的关键节点，记录语句之间的控制流边，进而从抽象语法树中构建智能合约的控制流图；再次遍历智能合约的数据流图，记录程序中的数据变量，在数据变量定义与使用之间添加数据流边，从而构建出智能合约的数据流图。最后，将抽象语法树、控制流图和数据流图进行结合构建出智能合约源代码的抽象语义图。

面向字节码构图模块的流程如图 5-5(b) 所示，与获取前端页面的源代码一样：首先，从前端中获取待解析的字节码。首先，将字节码反汇编为操作码和数据字节，将操作码指令依据功能划分成一个个基本块；接着，模拟以太坊虚拟机执行操作码，在基本块之间添加不同类型的控制流边，进而生成控制流图。然后，将操作码中无实际语义的操作码删除并将语义相同的操作码进行合并，根据其对应的操作码为数据字节添加不同类型的数据标签，从而实现操作码的规范化；最后，将控制流图与规范化后的基本块结合生成字节码的抽象语义图。



(a) 源代码构图流程　　(b) 字节码构图流程

图 5-5 智能合约构图模块

5.2.3　图信息向量化模块

图信息向量化模块主要功能是对从智能合约构图模块中生成的抽象语义图进行向量化的表征，以便其适用于图匹配神经网络。具体流程如图 5-6 所示。首先，遍历智能合约的数据集中源代码生成的抽象语义图和字节码生成的抽象语义图，将这些文本用于训练 Word2vec 模型的基础语料库；然后，将抽象语义图的每个节点和边的词汇文本采用 one-hot 的稀疏向量表示，利用 CBOW 模型和基于层次 softmax 多类别分类器来对词嵌入模型进行训练；最后，分别将抽象语义图的节点和边分别编码为节点特征向量和边特征向量。

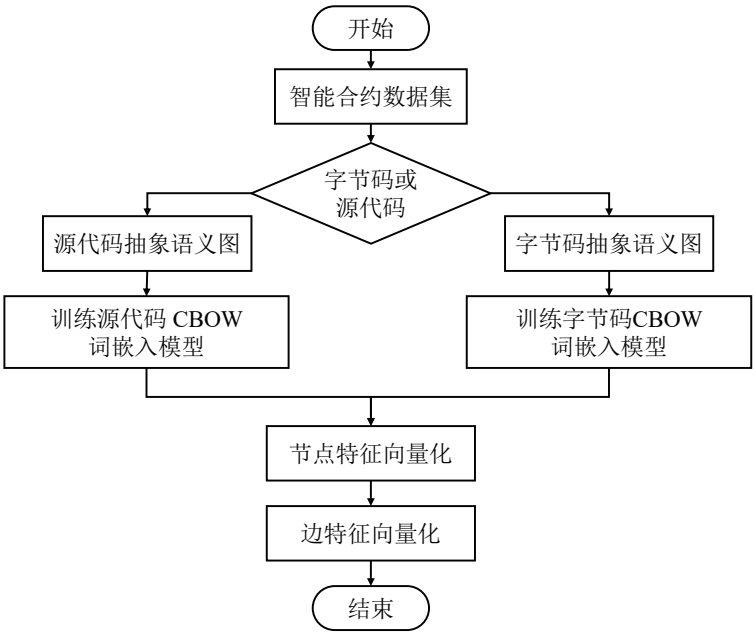


图 5-6 抽象语义图词嵌入流程

5.2.4　图匹配神经网络模块

图匹配神经网络模块是用来衡量两个合约生成的抽象语义图之间的相似性，从而根据相似性来判断该合约是否包含某种特定类型的漏洞。图神经网络主要分为训练模型和检测漏洞两个阶段。

图匹配神经网络模型训练阶段示意如图 5-7 所示。首先，将有具体漏洞类型标签的智能合约数据集进行拆分，保证每种漏洞类型都有数据集；然后，依据特定漏洞数据集是否包含该漏洞，将包含该漏洞类型的合约标记为 1，不包含该漏洞类型的合约标记为 0。训练检测该漏洞类型的图匹配神经网络模型的时候，需要同时从数据集中取出一对合约，经过构图模块来生成该合约的抽象语义图，构图模块也分为面向字节码和面向源代码两种；接着，将生成的抽象语义图通过图向量化模块构建节点信息向量和边信息向量。最后，将这一对合约生成的节点和边信息向量输入到相应的模型中，通过交叉注意

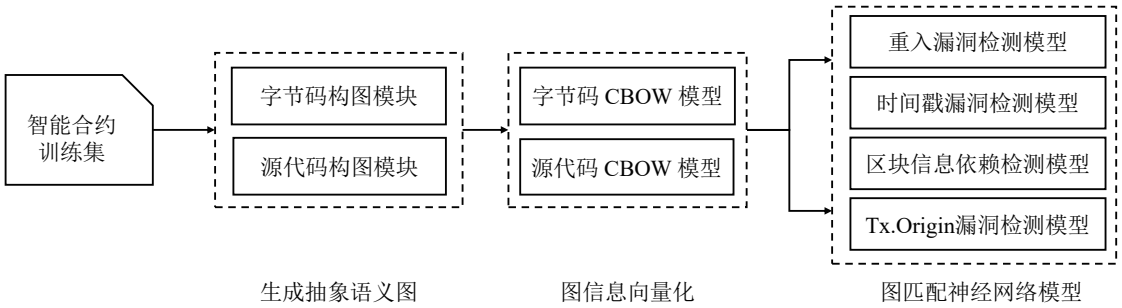


图 5-7 图匹配神经网络训练阶段

力机制来学习智能合约中的漏洞特征，根据模型输出的结果与两个合约的标签来调整图匹配神经网络的模型参数。如此不断地输入合约对，直到完成图匹配神经网络模型的训练。四个不同漏洞类型数据集，依据源代码和字节码的不同输入数据类型，一共可以训练出八种类型的漏洞检测模型。

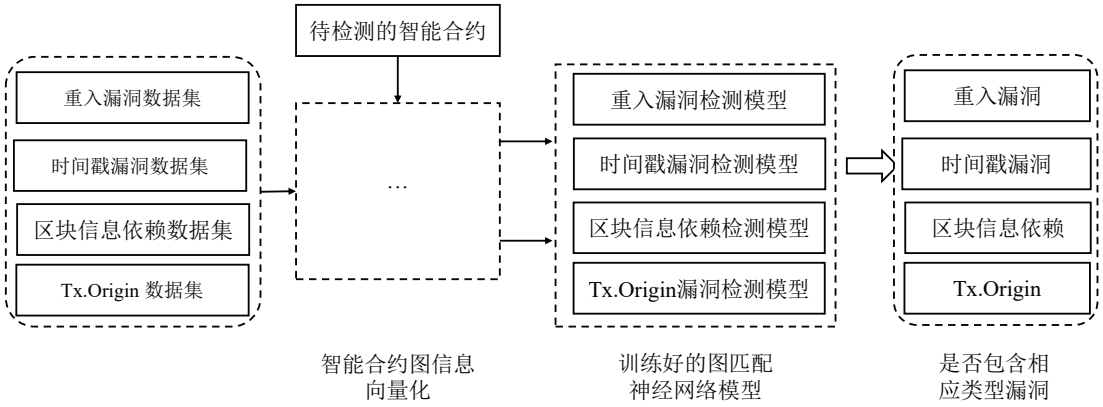


图 5-8 图匹配神经网络漏洞检测阶段

漏洞检测阶段示意如图 5-8 所示，首先要确定待检测智能合约的输入类型，根据输入类型的需要采用合适的智能合约构图模块和图信息向量化模块；此外还需要确定具体要检测的漏洞种类，采用不同的漏洞数据集与待检测智能合约进行匹配，具体判断流程如图 5-9 所示。

确定合约的承载类型和检测漏洞目标之后，从相应的漏洞数据集中取出一个包含漏洞的合约与待检测的合约一同输入到构图模块中，然后输入到训练好的相应漏洞类型的图匹配神经网络，将图匹配神经网络输出的两个合约的图空间向量进行相似度计算，如果相似度达到阈值，则认为该待检测合约包含相应的漏洞类型，否则继续从漏洞数据集中取出一个标签合约进行匹配，直到漏洞数据集用完或匹配成功为止。

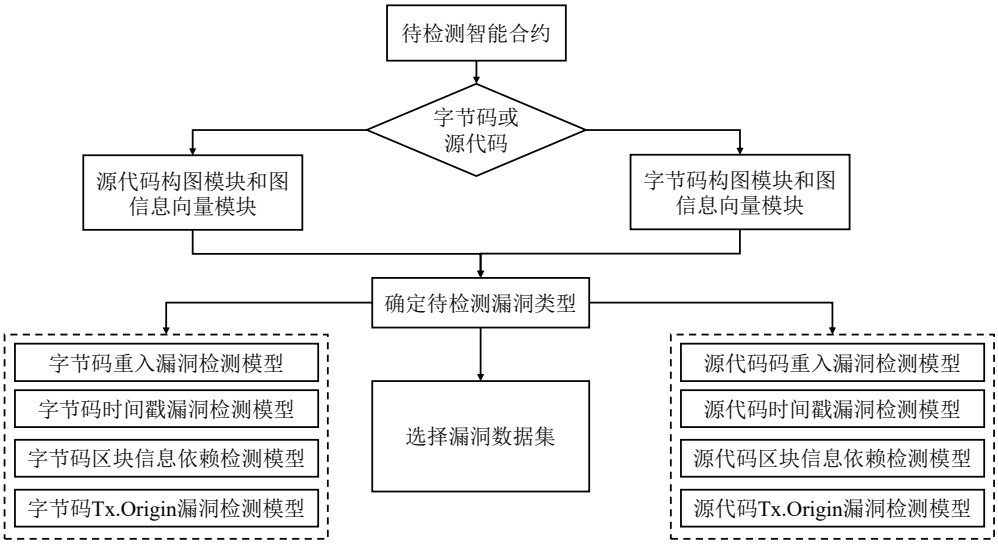


图 5-9 合约漏洞检测模型和数据集选择

5.3 面向真实世界合约实验

进一步验证本文提出的智能合约漏洞检测方法在真实场景的效果，将使用本文提出的方法来检测真实智能合约，并对检测得到的结果进行分析。首先，利用之前标记好的数据集作为训练集让本文的图神经网络模型学习智能合约漏洞特征，然后从以太坊上抓取最新的真实世界智能合约作为测试集，对检测所到的结果通过进一步人工分析与验证，以此来评估本文方法在检测真实智能合约漏洞的有效性。

5.3.1 真实世界数据集

为了验证本文方法在真实场景下的有效性，需要构建一个能够反映当前智能合约安全现状的数据集。一般来说，只要知道智能合约部署在以太坊区块链上的地址，就可以通过 Etherscan 下载到相应合约的源代码和字节码。但近年来以太坊上包含源代码的智能合约数量较少，每天能够抓取的合约数量受到了限制，于是本文从开源数据集 EtherSolve[70] 中收集了 5907 个包含源代码的智能合约，这些都是未被标签过的智能合约。

5.3.2 实验结果与分析

利用之前带漏洞标签的智能合约训练集分别对面向源代码和面向字节码的图神经网络模型进行训练，当模型训练结束之后，就对收集的真实世界智能合约数据集进行面向四种漏洞类型的检测。

本文提出的面向源代码的检测方法输入的是智能合约源代码，面向字节码的检测工具输入的是智能合约的字节码，得到两种检测结果之后，然后通过人工一个个去分析这些智能合约相对应的源代码是否包含相应的漏洞。

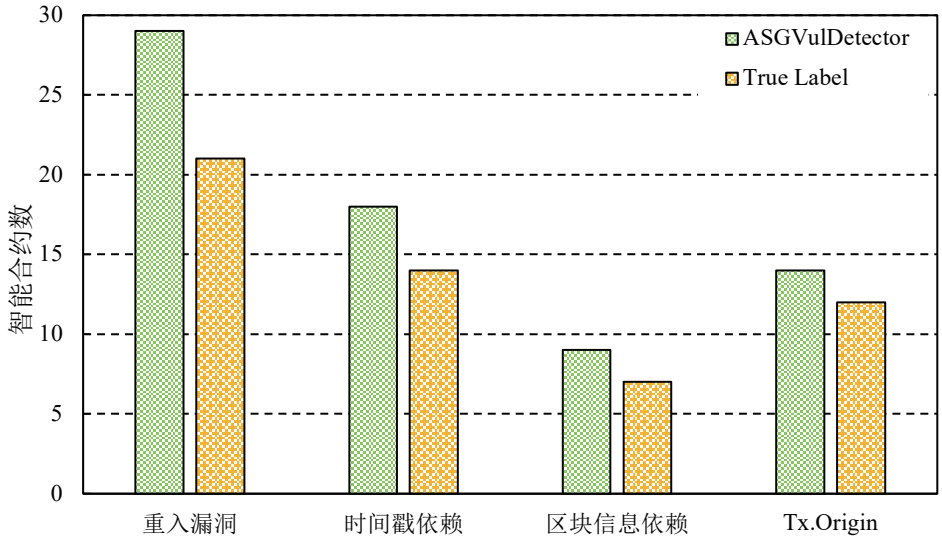


图 5-10 面向源代码漏洞检测结果

面向源代码的检测方法针对真实世界合约的具体检测结果如图 5-10 所示，其中 ASGVulDetector 是本文提出的面向源代码的图神经网络的漏洞检测方法报告出的漏洞合约，而 True Label 对 ASGVulDetector 检测出的智能合约经过进一步的人工分析与验证之后确定为包含相应漏洞的智能合约数。最后，经过评估发现本文提出面向源代码的智能合约漏洞检测方法在真实合约环境下针对重入漏洞、时间戳漏洞、区块信息依赖和 Tx.Origin 的检测准确率为 72.41%、77.78%、77.78% 和 85.71%。

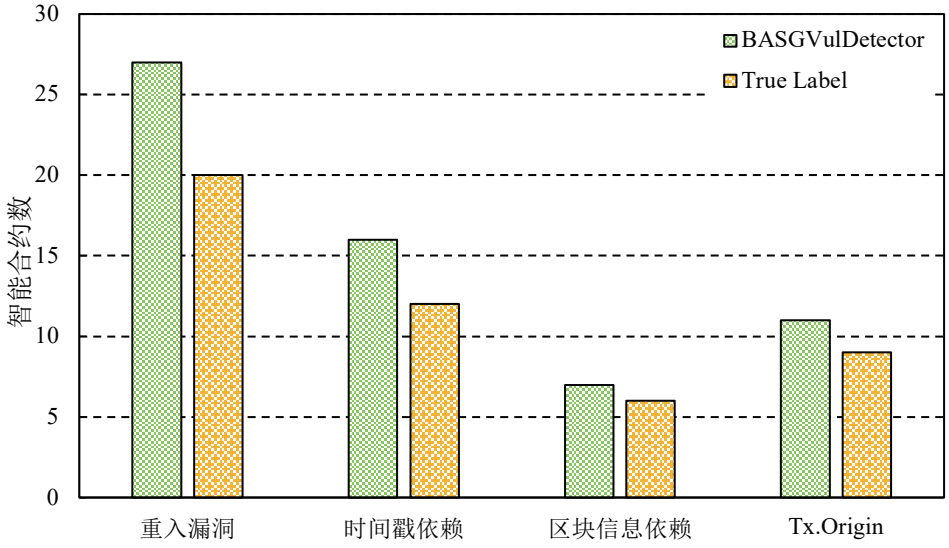


图 5-11 面向字节码漏洞检测结果

图 5-11 为本文提出的面向字节码的图神经网络检测方法 BASGVulDetector 对真实数据集检测的结果，各个漏洞的检测结果的准确率分别为 74.07%、75%、85.71% 和 81.81%。从检测结果上来看，本文提出的方法无论是面向字节码还是面向源代码，都对区块信息依赖、Tx.Origin 和时间戳漏洞有着较高的准确率，说明这些漏洞的特征更加

容易地被模型学习到，而面向重入漏洞的结果准确率相对较低，这可能是随着时间的推移，最新部署的智能合约能够有效地遵守正确编码规范，从而避免重入漏洞。而本文提出模型没能适应这些最新的编码规范，导致准确率的下降。

为了验证本文提出的面向源代码和面向字节码的检测方法进行结合之后的性能，本文采用了一种系统性检测方法 MixVulDetector，即将两个检测方法所得到的结果进行交叉验证，如果两个检测方法都判定同一个智能合约包含漏洞，则认定该合约包含此类型的漏洞，否则为不包含。具体的检测结果如图5-12所示，其中重入漏洞、时间戳漏洞、区块信息依赖和 Tx.Origin 的检测准确率到达了 83.33%、83.61%、100% 和 100%。

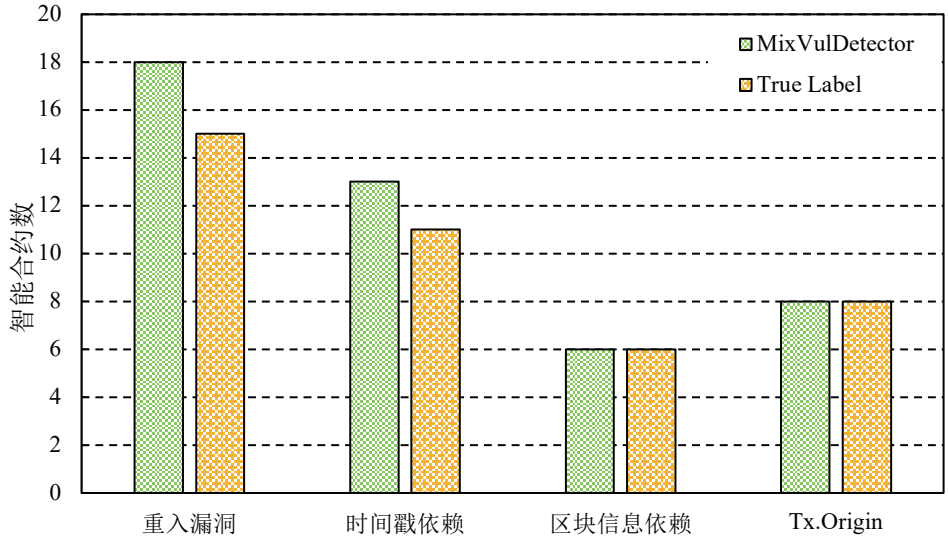


图 5-12 交叉集检测方法结果

从上面的评估结果上可以推断出将本文提出的面向字节码和源代码的漏洞检测方法结合起来能够有效地提升智能合约漏洞检测的准确率，尤其是在检测区块信息依赖和 Tx.Origin 两种漏洞，没有一个误报的检测。但是两种检测方法的交叉会使得整体的漏洞检测合约数下降，也即使得智能合约检测漏报率增加。在实际使用中，可以将两个检测工具检测到的漏洞集进行合并，两个检测工具交叉的智能合约可以标记为警告，而不重叠的部分标记为提醒，从而提升智能合约的安全审计水平。

5.4　本章小结

本章介绍了本文提出的系统设计与实现，整个系统包含用户界面模块、图信息向量化模块、字节码构图模块、源代码构图模块和图匹配神经网络模块等共六个模块，并对其中每个模块都进行了详细的介绍与分析。其中用户界面模块主要是提供一个可交互的前端界面，实现可视化的智能合约检测操作；构图模块是将智能合约的字节码或源代码转化为抽象语义图的模块；图信息向量化模块是将字节码或源代码生成的抽象语义图进行信息特征向量化的模块，使得智能合约能够输入到图匹配神经网络；图匹配神经网络

模块是对智能合约进行漏洞检测的主要模块，主要分为训练模型和漏洞检测两个阶段，可基于字节码和源代码生成多种不同类型的图匹配神经网络模型。

本章介绍了将本文提出的漏洞检测方法运用到面向真实世界的智能合约漏洞检测实验，以评估本文方法在真实场景下的有效性。首先，从以太坊区块链上抓取最新的真实智能合约；然后，分别用面向字节码和源代码的漏洞检测方法对真实世界的智能合约进行检测，将检测的结果通过人工分析和验证，从而证明本文提出的方法在真实场景下的有效性。

第六章　总结与展望

6.1　总结

以太坊作为最流行的区块链平台之一，上面部署的智能合约数量越来越多，其承载的金融价值也水涨船高，一旦智能合约漏洞被别有用心者利用，将极大地损害以太坊公信力并将给投资者造成巨大的经济损失。智能合约一经部署就无法修改的特性，使得事先对智能合约进行漏洞检测显得尤为重要。本文通过对现有的智能合约漏洞检测方法调研，发现传统的漏洞检测方法依赖专家规则、自动化程度低、检测时间长且难以应用到较大规模的检测当中。新兴的基于机器学习的智能合约漏洞检测方法没有充分利用智能合约中丰富的结构信息，在面对不断涌现的各种各样的漏洞检测需求时显得捉襟见肘。为解决现有智能合约漏洞检测方法中的局限性，本文构建了一种新的表征智能合约结构信息的图中间表示形式，并与图神经网络技术结合，从而实现对智能合约的漏洞检测。此外，在提出的漏洞检测方法之上，设计相关比较实验以评估本文所提出的合约漏洞检测效果，并实现了一个检测原型系统。本文主要研究工作如下：

（1）现有智能合约漏洞检测方法现状分析。对智能合约中常见四种漏洞类型进行了研究与分析，阐述它们漏洞触发机制并给出开发人员在实践中避免这些漏洞的建议；对目前主流的智能合约漏洞检测方法开展调研，从技术角度分析了当前的漏洞检测方案的基本原理和在机制上存在的局限性。

（2）构建了能够表征智能合约的语法和语义信息的抽象语义图结构。针对智能合约两种不同类型的承载形式，分别提出了面向源代码和面向字节码的抽象语义图构建方法。面向源代码的抽象语义图构建方式是首先从源代码中生成抽象语法树，然后在抽象语法树基础上构建控制流图和数据流图，接着将控制流、数据流和抽象语法树相结合来构建面向源代码的抽象语义图。面向字节码的抽象语义图的构建方式是首先将字节码进行反编译生成操作码，接着按照一定的规则将操作码划分成基本块，然后模拟以太坊虚拟机执行操作码来生成基本块之间的控制流，最后对操作码进行规范化，从而构造出面向字节码的抽象语义图。

（3）基于抽象语义图的图匹配神经网络检测方法研究。本文通过结合图表示形式与图神经网络检测技术的思想，提出了基于图匹配神经网络的智能合约漏洞检测方法。首先，由两种不同类型数据生成的抽象语义图通过自然语言处理的词嵌入技术，分别生成结构类似的节点特征向量和边特征向量；然后，将生成的图表示形式的数据输入到图匹配神经网络模型中，让模型学习智能合约的漏洞特征；最后，利用训练好的图神经网络模型对智能合约进行漏洞检测。

（4）设计并实现了一个智能合约漏洞检测系统。在本文提出的基于抽象语义图的图

匹配神经网络漏洞检测方法的基础上，搭建了一个智能合约漏洞检测系统；详细介绍了检测系统的各个模块，包括用户界面模块、图信息向量化模块、字节码构图模块、源代码构图模块和图匹配神经网络模块；此外，还收集了开源数据集中智能合约来作为本文检测方法的数据集，并对智能合约漏洞检测系统进行了测试，证明了本文方法在真实的智能合约检测场景下的有效性。

6.2 展望

本文提出的抽象语义图在一定程度能够有效地表征智能合约的语法和语义信息，并且与图匹配神经网络进行结合有效地提高了漏洞检测效率。但是本文提出的方法依然存在着可以改进的地方：

（1）抽象语义图缺少对智能合约的函数间调用关系的表征，智能合约中可以通过调用函数的方式来实现某些功能，因此如果能够在抽象语义图中添加函数调用关系能够进一步提高抽象语义图对复杂智能合约的表征能力。

（2）对抽象语义图进行向量化使用的是基本的 Word2vec 模型，而当前自然语言处理领域有着全新的进展，应该有比 Word2vec 更好的模型来对本文提出抽象语义图的节点和边进行信息向量化，故而探究新兴的词嵌入模型进而提升漏洞检测效果可成为下一阶段的研究工作。

（3）通过实验结果分析，本文提出的漏洞检测方法能够对单一智能合约引入的漏洞类型，如重入漏洞、时间戳依赖和区块信息依赖等；然而，对于由多个智能合约一起引发的漏洞类型无能为力，因此未来的一大工作方向是对跨合约的漏洞类型进行研究。

（4）本文提出的智能合约漏洞检测方法是基于现有已知的漏洞，需要为图神经网络训练提供已标记过的智能合约漏洞类型的数据集，才能进行相关类型漏洞的检测。因此本文提出的方法适用于智能合约中已知的漏洞检测，无法发现新的漏洞类型。在未来的工作中，可以为已知智能合约漏洞类型建立一套相关的漏洞模版，并且采用漏洞注入的方式来生成尽可能多的漏洞合约，从而使得图神经网络充分学习到相关漏洞的特征。

参考文献

[1] SZABO N. Formalizing and Securing Relationships on Public Networks[J]. First Monday, 1997.

[2] NAKAMOTO S. Bitcoin: A peer-to-peer electronic cash system[J]. Decentralized Business Review, 2008:21260.

[3] BUTERIN V, et al. A next-generation smart contract and decentralized application platform[J]. white paper, 2014, 3(37).

[4] WANG S, YUAN Y, WANG X, et al. An overview of smart contract: architecture, applications, and future trends[C]. Proceedings of the 2018 IEEE Intelligent Vehicles Symposium (IV). IEEE, 2018. 108-113.

[5] XING C, CHEN Z, CHEN L, et al. A new scheme of vulnerability analysis in smart contract with machine learning[J]. Wireless Networks, 2020:1-10.

[6] SCHÄR F. Decentralized finance: On blockchain-and smart contract-based financial markets[J]. FRB of St. Louis Review, 2021.

[7] ZOU W, LO D, KOCHHAR P S, et al. Smart contract development: Challenges and opportunities[J]. IEEE Transactions on Software Engineering, 2019, 47(10):2084-2106.

[8] KHATOON A. A blockchain-based smart contract system for healthcare management[J]. Electronics, 2020, 9(1):94.

[9] ZHANG Y, KASAHARA S, SHEN Y, et al. Smart contract-based access control for the internet of things[J]. IEEE Internet of Things Journal, 2018, 6(2):1594-1605.

[10] HE D, DENG Z, ZHANG Y, et al. Smart contract vulnerability analysis and security audit [J]. IEEE Network, 2020, 34(5):276-282.

[11] MEHAR M I, SHIER C L, GIAMBATTISTA A, et al. Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack[J]. Journal of Cases on Information Technology (JCIT), 2019, 21(1):19-32.

[12] DESTEFANIS G, MARCHESI M, ORTU M, et al. Smart contracts vulnerabilities: a call for blockchain software engineering?[C]. Proceedings of the 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, 2018. 19-25.

[13] LAI E, LUO W. Static analysis of integer overflow of smart contracts in ethereum[C]. Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy. 2020. 110-115.

[14] WANG B, LIU H, LIU C, et al. Blockeye: Hunting for defi attacks on blockchain[C]. Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, 2021. 17-20.

[15] WAN Z, XIA X, LO D, et al. Smart contract security: a practitioners'perspective[C]. Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021. 1410-1422.

[16] RESEARCH V M. Smart Contracts Market size worth $ 770.52 Million by 2028 at 24.55% [J]. GlobeNewswire News Room, 2021.

[17] DURIEUX T, FERREIRA J F, ABREU R, et al. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts[C]. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering(ICSE). Association for Computing Machinery, 2020. 530-541.

[18] WU F, WANG J, LIU J, et al. Vulnerability detection with deep learning[C]. Proceedings of the 2017 3rd IEEE International Conference on Computer and Communications (ICCC). IEEE, 2017. 1298-1302.

[19] LI Z, ZOU D, TANG J, et al. A comparative study of deep learning-based vulnerability detection system[J]. IEEE Access, 2019, 7:103184-103197.

[20] NGUYEN A T, NGUYEN T N. Graph-based statistical language model for code[C]. Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, 2015. 858-868.

[21] LOU B, SONG J. A study on using code coverage information extracted from binary to guide fuzzing.[J]. International Journal of Computer Science and Security (IJCSS), 2020, 14(5):200-210.

[22] WISBEY B, MONTGOMERY P G, PYNE D B, et al. Quantifying movement demands of afl football using gps tracking[J]. Journal of science and Medicine in Sport, 2010, 13 (5):531-536.

[23] JIANG B, LIU Y, CHAN W K. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection[J]. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018:259-269.

[24] ZHANG Q, WANG Y, LI J, et al. EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts[C]. Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2020. 116-126.

[25] HE J, BALUNOVIĆ M, AMBROLADZE N, et al. Learning to fuzz from symbolic execution with application to smart contracts[C]. Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019. 531-548.

[26] BALDONI R, COPPA E, D'ELIA D C, et al. A survey of symbolic execution techniques [J]. ACM Computing Surveys (CSUR), 2018, 51(3):1-39.

[27] LUU L, CHU D H, OLICKEL H, et al. Making Smart Contracts Smarter[C]. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security(CCS). Association for Computing Machinery, 2016. 254-269.

[28] MUELLER B. A framework for bug hunting on the ethereum blockchain[M]. Webiste, 2017.

[29] MOSSBERG M, MANZANO F, HENNENFENT E, et al. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts[C]. Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019. 1186-1189.

[30] TORRES C F, SCHüTTE J, STATE R. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts[C]. Proceedings of the 34th Annual Computer Security Applications Conference(ACSAC). Association for Computing Machinery, 2018. 664-676.

[31] GARFATTA I, KLAI K, GAALOUL W, et al. A survey on for solidity smart contracts[C]. Proceedings of the 2021 Australasian Computer Science Week Multiconference. 2021. 1-10.

[32] 倪远东, 张超, 殷婷婷. 智能合约安全漏洞研究综述[J]. 信息安全学报, 2020, 5(3): 78-99.

[33] BHARGAVAN K, Delignat-Lavaud A, FOURNET C, et al. Formal Verification of Smart Contracts: Short Paper[C]. Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security(PLAS). Association for Computing Machinery, 2016. 91-96.

[34] HILDENBRANDT E, SAXENA M, RODRIGUES N, et al. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine[C]. Proceedings of the 2018 IEEE 31st Computer Security Foundations Symposium (CSF). 2018. 204-217.

[35] TSANKOV P, DAN A, Drachsler-Cohen D, et al. Securify: Practical Security Analysis of Smart Contracts[C]. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security(CCS). Association for Computing Machinery, 2018. 67-82.

[36] FEIST J, GRIECO G, GROCE A. Slither: A Static Analysis Framework for Smart Contracts[C]. Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). 2019. 8-15.

[37] TIKHOMIROV S, VOSKRESENSKAYA E, IVANITSKIY I, et al. SmartCheck: Static analysis of ethereum smart contracts[C]. Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. ACM, 2018. 9-16.

[38] ZHUANG Y, LIU Z, QIAN P, et al. Smart contract vulnerability detection using graph neural network.[C]. Proceedings of the Twenty-Ninth Int'l Joint Conf. on Artificial Intelligence. IJCAI, 2020. 3283-3290.

[39] SAMREEN N F, ALALFI M H. Reentrancy vulnerability identification in ethereum smart contracts[C]. Proceedings of the 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, 2020. 22-29.

[40] The DAO Attrack[J]. Wikipedia, 2021.

[41] ANDESTA E, FAGHIH F, FOOLADGAR M. Testing smart contracts gets smarter[C]. Proceedings of the 2020 10th International Conference on Computer and Knowledge Engineering (ICCKE). IEEE, 2020. 405-412.

[42] CHEN J, XIA X, LO D, et al. Defining smart contract defects on ethereum[J]. IEEE Transactions on Software Engineering, 2020.

[43] LI X, JIANG P, CHEN T, et al. A survey on the security of blockchain systems[J]. Future Generation Computer Systems, 2020, 107:841-853.

[44] ATZEI N, BARTOLETTI M, CIMOLI T. A survey of attacks on ethereum smart contracts (sok)[C]. Proceedings of the International conference on principles of security and trust. Springer, 2017. 164-186.

[45] DIKA A. Ethereum smart contracts: Security vulnerabilities and security tools[D]. NTNU, 2017.

[46] GROUP N, et al. Decentralized application security project (dasp) top 10[M]. November, 2018.

[47] REGISTRY S. Smart contract weakness classification and test cases[EB/OL]. https://swcregistry.io/, 2018.

[48] RAMEDER H. Systematic review of ethereum smart contract security vulnerabilities, analysis methods and tools[J]. 2021.

[49] ZHANG J, WANG X, ZHANG H, et al. A novel neural source code representation based on abstract syntax tree[C]. Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019. 783-794.

[50] LI Y, WANG S, NGUYEN T N, et al. Improving bug detection via context-based code representation learning and attention-based neural networks[J]. Proceedings of the ACM on Programming Languages, 2019, 3:1-30.

[51] CHURCH K W. Word2vec[J]. Natural Language Engineering, 2017, 23(1):155-162.

[52] BUCKMAN J, ROY A, RAFFEL C, et al. Thermometer encoding: One hot way to resist adversarial examples[C]. Proceedings of the International Conference on Learning Representations. 2018.

[53] KONDRAK G. N-gram similarity and distance[C]. Proceedings of the International symposium on string processing and information retrieval. Springer, 2005. 115-126.

[54] MIKOLOV T, CHEN K, CORRADO G, et al. Efficient estimation of word representations in vector space[J]. arXiv preprint arXiv:1301.3781, 2013.

[55] YIN Z, SHEN Y. On the dimensionality of word embedding[J]. Advances in neural information processing systems, 2018, 31.

[56] 张峰逸, 彭鑫, 陈驰, 等. 基于深度学习的代码分析研究综述[J]. 计算机应用与软件, 2018, 35(6):9.

[57] MIKOLOV T, KARAFIÁT M, BURGET L, et al. Recurrent neural network based language model.[C]. Proceedings of the Interspeech. Makuhari, 2010. 1045-1048.

[58] KIM P. Convolutional neural network[M]. MATLAB deep learning. Springer, 2017. 121-147.

[59] KARATZOGLOU A, JABLONSKI A, BEIGL M. A seq2seq learning approach for modeling semantic trajectories and predicting the next location[C]. Proceedings of the 26th acm sigspatial international conference on advances in geographic information systems. 2018. 528-531.

[60] GORI M, MONFARDINI G, SCARSELLI F. A new model for learning in graph domains [C]. Proceedings of the IEEE International Joint Conference on Neural Networks. IEEE, 2005. 729-734.

[61] SCARSELLI F, GORI M, TSOI A C, et al. The Graph Neural Network Model[J]. IEEE Transactions on Neural Networks, 2009, 20(1):61-80.

[62] GILMER J, SCHOENHOLZ S S, RILEY P F, et al. Neural Message Passing for Quantum Chemistry[C]. Proceedings of the 34th International Conference on Machine Learning. PMLR, 2017. 1263-1272.

[63] LI Y, ZEMEL R, BROCKSCHMIDT M, et al. Gated graph sequence neural networks[C]. Proceedings of the ICLR'16. 2016.

[64] LI Y, GU C, DULLIEN T, et al. Graph matching networks for learning the similarity of graph structured objects[C]. Proceedings of the International conference on machine learning. PMLR, 2019. 3835-3845.

[65] MEULEMANS A, CARZANIGA F, SUYKENS J, et al. A theoretical framework for target propagation[J]. Advances in Neural Information Processing Systems, 2020, 33: 20024-20036.

[66] solc-ast[EB/OL]. https://github.com/iamdefinitelyahuman/py-solc-ast.

[67] FERREIRA J F, CRUZ P, DURIEUX T, et al. SmartBugs: A Framework to Analyze Solidity Smart Contracts[C]. Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2020. 1349-1352.

[68] GHALEB A, PATTABIRAMAN K. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection[C]. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020. 415-427.

[69] Etherscan[EB/OL]. http://etherscan.io/.

[70] CONTRO F, CROSARA M, CECCATO M, et al. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode[C]. Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). IEEE, 2021. 127-137.

致　谢

行文至此，意味着论文即将完结，同时也表示着自己三年的硕士学习生涯也即将画上句号，不由得感慨万千。研究生期间发生的种种一一回现，三年来自己也在不断进步，借此机会由衷地向所有帮助和关心过我的人表达感谢。

盾，也是激励我前行的动力。

最后，感谢在研究生三年期间的自己，希望在以后的人生道路上，自己也能继续保持学习，不断进步。

感谢每一个关心和帮助过我的人，祝福你们万事顺遂！