中国科学技术大学

专业硕士学位论文

（专业学位类型）

基于软件分析的智能合约漏洞检测系统

作者姓名： 叶家鸣

专业领域： 软件工程

导师姓名： 薛吟兴 研究员

完成时间： 二〇一九年十一月十三日

University of Science and Technology of China

A dissertation for master's degree

(Professional degree type)

A Software Analysis Based Vulnerability Detection System For Smart Contracts

Author: Jiaming Ye

Speciality: Software Engineering

Supervisor: Prof. Yinxing Xue

Finished time: November 13, 2019

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与本文一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☑ 公开 ☐ 保密（____ 年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘　要

区块链，包括以太坊、智能合约等等，从方方面面改变着人们的生活。这项技术给各种行业带来了革新，如分布式银行，分布式能源管理系统等等。它给本文们带来了巨大的技术便利，同时也可能带来巨大的经济损失。在 2016 年，由黑客发起的针对 DAO 合约的攻击共造成了数百万美元的损失。一方面，区块链技术带来的技术革命激励着人们不断创新；另一方面，区块链应用，作为典型的软件，理应接受软件分析技术的检验。

软件分析技术，包括软件静态分析技术、动态分析技术、形式验证技术等等，有着多种不同的形式，各种分析技术有不同的特点，在处理不同类型的问题时，应该采用不同的分析技术。在这篇工作中，本文使用静态分析技术，辅以克隆分析技术，来构建漏洞检测系统。目前在智能合约软件上的分析工作并不多，智能合约软件目前还属于一个新领域。这篇工作将改善现有系统的检测效果，调研字节码的相关工作，并提出标准漏洞库，这几项目标将改善现有智能合约领域缺乏的开发生态，并为之后的工作打下坚实的基础。

本文调研了著名的 4 个智能合约的漏洞，对它们的基本形式，主要特点，危害性都在文中做出了描述。本文也调研了现有工具在这几个主要漏洞上的检测能力，并对工具的检测能力的原因做了简单分析，本文发现，极少有工具能覆盖所有类型的漏洞。除此之外，本文调研了以太坊虚拟机对生成字节码的影响，探寻了字节码背后的奥秘。

为了有效改善现有的软件分析技术，在这篇工作中，本文提出结合传统静态分析技术与克隆分析技术。通过静态分析技术本文归纳总结了四种漏洞签名，再根据克隆分析技术去寻找漏洞，本文还在后面加入了九种 CPT 技术以降低系统的误报数量。在整个系统中，由于标准漏洞库的缺乏，本文在部分环节加入了领域专家的审计以保证本文提出的漏洞库的公正性。最后，本文提出了基于软件分析的漏洞检测系统，Athena。

经过本文的大量实验，能明显地看出，Athena 在这四种漏洞上有着优秀的表现，整体准确率和整体覆盖率都是最高的。至于系统的运行效率，Athena 的运行速度和最快的工具仍有差距，但也在可接受的范围内。因此，本文提出的 Athena 系统，改进了现有的检测技术，达到了更好的检测效果，同时也拥有着良好的运行效率。

关键词：以太坊；智能合约；软件分析技术；软件漏洞

ABSTRACT

Blockchain, including Ethereum and smart contracts, are changing the daily life of people. This technique brings innovation to industries, such as decentralized banks, decentralized energy management systems etc. It brings great convenience in techniques, meanwhile it can cause devastating financial damages. In 2016, hackers launched a attack aiming at the DAO contract and cause millions of dollars losses. On the one hand, the blockchain technique inspires people to continuously innovate; on the other hand, the blockchain softwares, as an ordinary software, should be verified by software analysis techniques.

Software analysis techniques, including static analysis, dynamic analysis and verifications, have various features. When facing problems, we must choose appropriate one. In this work, we use static analysis, with clone analysis to build a vulnerability detection system. Our work will improve the existing detection results and propose a benchmark for Solidity vulnerabilities, which will make up the need of the vulnerability database, and ease future works on Solidity softwares.

We surveyed 4 smart contracts vulnerabilities, and introduces their features and their harm for softwares. We also surveyed the capability of cutting-edge tools, then we found that few tools can detect all types of vulnerabilities. What's more, we introduced the bytecode generation process.

To effectively improve the existing software analysis technique, in this work, we propose combining static analysis with clone analysis. We summarized 4 types of vulnerability signatures, and match suspicious code by clone analysis. We appended our system with 9 Security Shield techniques to reduce false positive reports. As the fairness of our benchmark, our domain experts will audit the suspicious code twice. Finally, we propose our software analysis based detection, Athena.

In our experiment, we find Athena has outstanding performance in both precision rate and recall rate. Considering the system efficiency, Athena still has gap with the fastest tool. However, it is proved Athena has better overall performance.

Key Words: Ethereum;Smart Contract;Software Analysis;Vulnerability

目　录

第 1 章　序　　言

1.1　课题研究意义及国内外现状分析

1.1.1　研究意义

自 2009 年“中本聪”发布了比特币的第一篇论文以来，区块链技术受到了越来越多的关注。虽然相比传统的计算机技术，区块链技术还比较年轻，但是区块链技术本身带来的新型货币体系，包括其在各行各业的应用：去中心化的新型银行[1]，基于去中心化智能合约的新型能源管理系统[2] 等等，无不凸显这一崭新技术在各行各业带来的革命性意义。

相比传统的中心化服务器，区块链技术的核心为去中心化的服务器系统，即网络中的每个运行区块链软件的设备都是一个独立的对等节点[3]。相比传统的中心化服务器系统，区块链的网络中的所有节点在进行交易时不依赖于中间人，所有人都维护着同一个“账本”，并通过密码学算法对自己的交易信息进行签名加密，最后通过计算量证明来保证账本的有效和可靠性。区块链技术中体现出的去中心化，独立平等和无法随意篡改等等特性吸引了大量学者、企业、研究单位的关注。

在区块链的所有应用中，以太坊（Ethereum）是推行加密货币的先驱之一。加密货币是一种经过密码学技术加密的数字货币。和其他货币如美元不同的是，加密货币没有实物，而且他们并不会受到中央银行或者政府部门的管理。加密货币独特的地方在于任何一个人都能使查看过往合约的所有交易记录。以太坊是一个基于区块链技术的平台，发起的初衷为“建立无法被停止的应用”[4]。以太坊使用了和比特币不同的技术路线，它提供了一个交易平台，在这个平台上用户解决标记（Tokens）去创建和运行各种应用。以太坊使用以太币（Ether）作为平台的主要货币，用密码学的算法来保证交易的正常进行。以太坊使用智能合约作为交易的主要媒介，网络中的各个节点通过智能合约完成交易、互动等功能。从 2017 年开始，以太坊的交易量开始高速增长[5]，越来越多的用户开始使用智能合约完成以太币的交易，区块链游戏的交互等等任务。

智能合约是由一种图灵完整的编程语言，Solidity，编写而成的。Solidity 是一种面向对象、高层次的编程语言。Solidity 受到了 C++、Python 和 JavaScript 的影响，使用了以太坊虚拟机（Ethereum Virtual Machine）进行编译运行。Solidity 是一门静态编程语言，支持继承关系，库调用和完整的用户自定义类型。使用 Solidity 能够创建投票、募集资金、盲选等等智能合约应用。随着使用智能合约的人越来越多，保证智能合约使用安全的呼声也越来越高，很多学者和研究机构

开始注意这门新语言，并用软件分析的方法（静态、动态分析等等）去研究这门新语言。2016 年 9 月，一场被称为“DAO 攻击”的事件窃取了价值数百万美元的以太币[6]。保证以太坊软件的安全迫在眉睫。

软件分析方法，包括静态分析、动态分析、形式验证、克隆分析等等方法，是保证软件安全、分析软件特性的必经之路。以上提到的几种分析方法，各有特点，应根据使用场景进行考虑，选取合适的分析方法。其中，静态分析技术不直接运行软件，主要的分析对象为软件的源代码，分析代码的抽象语法树（AST），控制流图（CFG），辅以特征提取[7]、图匹配等方法，分析软件的特点，静态分析技术也使用符号执行技术，将程序控制流图上的变量使用符号进行替代，使用各种约束求解器进行求解；动态分析技术是运行程序或使用虚拟机，并使用大量的测试用例做输入，以期程序产生异常的分析技术，它使用一些软件测试技术，如代码覆盖率（Code Coverage Rate）来保证测试用例能够覆盖到程序的所有运行路径，动态分析相比静态分析耗时更长，但准确度更高；形式验证分析的对象不是软件的实际代码，而是将软件的实现以另一种更形式化的方式描述，最后通过数学计算验证软件的安全是合理的，形式验证分析技术有很强的理论性，并且能保证较高的软件安全等级；代码克隆借助各种相似度计算算法，从漏洞代码入手，寻找在语义上或者程序结构上与漏洞代码相似的代码，常常辅以其他的技术来提高准确度。

1.1.2　国内外现状分析

1. 智能合约漏洞国内外研究现状

从 2016 年开始，越来越多的学者加入对智能合约的研究。出现了很多优秀的整理智能合约上容易出现的漏洞的工作，比如[8]，不仅收集了很有名的可重入漏洞，也有诸如错误乱序这种较难发现的漏洞，文中对各个漏洞也附上了代码示例。这些示例显然不是真实的例子，是作者自己根据漏洞的原理自己构造的，但浅显易懂，为刚开始接触智能合约的研究者提供了详实的资料。也有的团队[9]，从分析以太坊的使用记录入手，根据他们的数据，目前大部分的智能合约都来交易以太币或者管理钱包，极少部分智能合约被用来进行游戏的交互和其他应用，这也说明大部分的智能合约是和经济安全紧密联系起来的。如果智能合约的安全受到侵害，会造成大量的经济损失，这更说明本文们在智能合约的基础上研究软件安全是十分有必要的。

2. 软件分析方法国内外现状

目前，国内外已有大量的软件分析方法的工作。静态的分析方法通过抽象语法树（AST），控制流图（CFG），程序依赖图（PDG）等分析程序的特点。动态分析方法比如[10] 就总结了目前流行的动态分析方法，动态分析方法具有精确度

高的优点，但是覆盖率和耗时长也是在使用动态方法的时候不得不考虑的问题。但更多的工作是把静态分析和动态分析结合起来，既弥补了动态分析在覆盖率上的不足，也部分解决了静态分析准确度不高的问题。

由于智能合约还属于新事物，在智能合约上的软件分析方法还以静态分析和动态分析为主，当然也有使用形式验证、模糊测试等方法分析智能合约的。静态方法包括 Slither[11]，是通过分析 Solidity 编译器生成 AST，再将 AST 转换成 IR（中间语言）和 CFG（控制流图）；动态方法包括 Oyente[12]、Securify[13]、Manticore[14] 都是使用了约束求解的方法分析程序的 CFG，然后根据约束求解的结果来分析软件的特点，其中 Oyente 和 Manticore 均使用了在软件分析领域小有名气的约束求解器 Z3[15]，而 Securify 则使用的是 Souffle[16] 进行约束求解。同样的，IBM 的以太坊研究团队提出了 Zeus[17]，使用了形式验证的方法实现了对 Solidity 软件的分析。而 Echidna[18]，则是目前智能合约软件唯一的自动化模糊测试工具。智能合约虽然出生不久，但是分析智能合约的软件已经百花齐放。

3. 软件克隆分析方法国内外发展现状

提到软件克隆分析就不得不提到 ChanChal 等人的工作[19]，这篇文章分析了软件克隆分析法的优势和局限性，指出了克隆的四种级别，也推荐了一些克隆的分析方法。虽然这篇工作推荐的分析方法有部分已经落后于时代，但其中对克隆级别和克隆关键问题的讨论令人印象深刻。大体来说，各家对于什么是克隆、克隆的准确定义争论已久。同时，文中也支出了克隆的四种级别，第一类克隆、第二类克隆、第三类克隆和第四类克隆。其中，第一类克隆也叫完全克隆，两段代码只有空格和注释的不同。第二类克隆包括重命名克隆，即两段代码对变量的命名有差异。第三类克隆包括代码执行顺序，代码间隔的不同。而第四类克隆是语义上的相似，两段代码可以结构完全不同，但是功能是完全相同的。在这四种克隆类型中，第一类到第二类克隆因为比较简单，可以借助字符串匹配完美的实现，第三类克隆需要借助一些克隆分析手段，如最长公共子串（LCS）来实现，而第四类克隆需要加入更多的语义分析才能实现，是目前克隆的难点之一。

在针对二进制程序的克隆检测中，有工作比如 DECKARD[7]，通过分析编译器生成的 AST，从 AST 中的关键节点提取关键信息，并将提取的信息扁平化，再比较特征向量在高维空间中的距离来分析两个软件代码的相似度；有工作 BinGo[20] 采用了多样的分析方法，通过仿真，函数的执行路径，控制流图等方式来分析两个软件代码的相似度。

软件克隆有不得不面临的困难，可以也有得天独厚的优势。软件克隆方法在保证了一定的准确性的前提下，有着较好的可扩展性，在面对大量数据集的时候也能保证一定的的分析速度。本文的系统将把静态分析同克隆的方法结合起来，实现在字节码上的的软件分析。

1.2　选题的创新性和先进性

本系统是针对智能合约的字节码开发的软件分析系统。目前，世界上还没有一个完备的系统能够用软件克隆的方法实现对智能合约的软件分析工作；而智能合约软件内存在着严重的代码克隆现象，漏洞随着代码克隆进行广泛传播。使用克隆分析技术，能有效改善当前工具的检测效率。因此，这不仅是创新的工作，也是具有有很大贡献的工作。同时，这些智能合约软件和以太坊的经济安全息息相关，保证这部分智能合约的软件安全是很有价值的工作。

1.3　主要研究内容

该选题技术难度偏难，主要的难点在于以下几点：如何总结现有的漏洞主要攻击模式、如何构建标准漏洞库、如何将克隆的方法同静态分析的方法结合起来，选题能满足研究生论文的研究要求；工作量主要体现在漏洞的主要攻击模式上，需要阅读大量的漏洞代码，并且克隆方法的开发也需要大量的工作，但能够在预定的时间内完成课题研究的内容。针对以上难点，本文提出以下主要研究内容。

1.3.1　调研主要的几种 Solidity 漏洞

智能合约和以太坊的经济安全息息相关，因此有不少不法分子妄图借助智能合约的软件漏洞，窃取钱财。为了更好的完成软件分析技术的开发工作，首先要做的就是充分调研目前现有的智能合约漏洞，这样不仅能更清楚地分析出黑客利用漏洞的攻击模式，也能细微地调整软件分析技术在不同漏洞上的表现。

1.3.2　调研现有智能合约分析工具

随着以太坊和智能合约的逐渐火热，有越来越多的团队加入智能合约的分析工作之中，有的团队提出分析技术，而更多的团队则是在原来的分析方法上做出针对智能合约改进。其中，有的团队使用了静态分析技术，如 Slither[11]、SmartCheck[26] 和 Securify[13] 等等；有的团队使用了动态分析技术，如 Oyente[12]、Mythril[24]、MythX[25]、Manticore[14] 等等；有的团队使用了形式验证的技术，如 Zeus[17]；也有的团队使用了模糊测试的技术，如 Echidna[18] 等等。为了提出一个经得起考验的标准数据集，广泛调研目前各家的技术是非常有必要的。

1.3.3　软件克隆分析技术在漏洞检测上的应用

软件克隆分析技术并不是第一次被使用于漏洞检测系统上，通过调研，本文发现之前的工作如 VUDDY[21] 等使用基于克隆分析技术的漏洞检测系统在大规

模数据集上取得了良好的检测效果。但目前还未有在智能合约，Solidity 软件上使用克隆分析技术进行漏洞检测的工作。本文需要充分调用现有的和之前的相关工作，克服智能合约软件平台特性带来的困难，设计一套漏洞检测系统。

1.4 拟解决的关键问题

1.4.1 如何保证标准漏洞库的标准性和代表性

智能合约发展至今，还没有一个团队或者企业提出这样的一个标准数据集。本文从互联网上搜集合约代码数据，根据之前出现过的合约漏洞进行人工统计和分析。但即使是人工查验，也有误判和漏判的情况出现，这样的标准数据集是否足够标准将会成为一个值得讨论的问题。

1.4.2 现有漏洞检测工具的真实表现如何

智能合约上的漏洞检测工具经过近几年的发展，已经可以使用多种漏洞检测方法对漏洞进行定位、检测。这些工具有的选择投稿会议，有的选择进行开源以扩大社区影响力。这些工具各有优缺点，在不同检测方法上各展身手，但是本文仍需要一个囊括主流检测工具的横向比较，只有这样，才能对工具的真实性能做出准确评估，更进一步，这对比较各检测方法在智能合约软件上的适用性也有很大的帮助。

1.4.3 使用克隆技术进行漏洞检测的效果是否符合预期

在完成调研工作之后，本文需要实现将软件克隆技术应用于漏洞检测系统。以往的软件克隆分析技术，包括提取操作序列、提取函数特征等等，在智能合约的这门新语言上是否有效，要打一个问号。如果单一的技术行不通，那可以考虑加入其他静态分析技术或者动态分析技术，不断调整和优化软件分析系统。

第 2 章　智能合约相关背景知识

2.1　智能合约中的常见漏洞

智能合约有数十种漏洞，这些漏洞可以按漏洞危害，分为不同的等级。具有较高危害的几种漏洞，应该引起足够的重视。在本文中，本文参考了、、等文献，提出了以下四种高危漏洞：可重入漏洞、意外异常漏洞、低级调用漏洞和自毁漏洞。本文选择这几种漏洞作为讨论的对象是因为，这几种漏洞在智能合约软件开发中较为常见，容易触发，且造成巨大伤害，在、、中都对这几种漏洞提出了分析。因此，本文也必须在本文中详细研究这几种漏洞。

2.1.1　可重入 (Reentrancy) 漏洞

可重入漏洞是智能合约中最负盛名的漏洞。这个漏洞源于 Solidity 语言上的一个特殊机制，像 JavaScript 等语言一样，Solidity 语言有回调 `fallback` 函数。每个合约仅能有一个回调函数，并且这种函数没有参数，没有函数名，也不会返回任何的返回值。根据 Solidity 的开发文档，这种函数只会在以下两种情况被触发：当合约收到函数调用但却没有符合的函数签名（函数的唯一标识符）时；当合约接收到任何数量的以太币，即发生交易时。前一种情况极少出现，而可重入攻击抓住的就是后面的那种情况。如果一个黑客妄图发动可重入攻击，他需要做的就是在自己的合约中编写一段恶意的回调函数，从而利用回调函数的特性反复记性转账操作，达到窃取以太币的目的。另一方面来说，如果被回调函数调用的函数中不包含转账语句，可重入漏洞也可利用重复调用，来造成拒绝服务（Denial of Service）攻击。因为在以太坊的体系中，每个合约都有一定的 gas 限制，gas 会在合约的每一次调用、任何操作中消耗掉。如果重复调用一个函数，会不断消耗被调用函数所在合约的 gas，待 gas 耗尽，合约不能提供任何的服务，也就达到了拒绝服务攻击的目的。在 2016 年，黑客借助可重入漏洞对 DAO 合约发动攻击，共盗取了价值数百万美元的以太币。

可重入漏洞在智能合约的漏洞中算是比较灵活的漏洞，其发动攻击的方式灵活多变，目前主要的防范方式是对函数调用者的身份进行检验，新版的 Solidity 编译器也对有可重入风险的函数做出了提醒，但是仍有很多已经部署在以太坊平台上的合约存在可重入的风险，因此本文很重视可重入漏洞攻击模式的的分析，并会在之后的技术开发阶段对这种漏洞进行针对性的改进。

2.1.2　意外异常 (Unexpected Revert) 漏洞

意外异常漏洞是由于 EVM 的异常处理机制引起的一种拒绝服务 (Denial of Service) 攻击。这种漏洞多出现于在 for 循环中进行的批量转账操作中。产生该漏洞的主要原因是，开发者使用 `require` 或者 `assert` 去检查写于 `for` 循环内的转账外部调用。如果其中一次转账发生错误，`require` 函数将抛出异常，并终止整个交易过程，这将导致后面的地址无法正常进行转账业务。更进一步说，如果这样的转账错误是攻击者蓄意为之，那么这个意外异常漏洞就给了攻击者一个停止当前服务的机会。Solidity 研发团队不推荐也不建议开发者们在循环中进行任何的转账业务。

如下图2.1代码所示，在第 5 行，函数试图向 `refundAddress` 中存放的地址发送 `ether`。该函数使用 `send` 函数，通过不断循环进行转账。同时，该函数也针对 `send` 函数用 `require` 进行了返回值检查。如果 `refundAddress` 中的任何一个地址发生转账错误，则该函数将停止运行，剩下的合约地址也没法正常进行转账业务。

```
1    address[] private refundAddresses;
2    mapping (address => uint) public refunds;
3    function refundAll() public {
4            for(uint x; x < refundAddresses.length; x++) {
5                require(refundAddresses[x].send(refunds[refundAddresses[x]]));
6            }
7    }
```

图 2.1　意外异常调用漏洞示例代码

2.1.3　低级调用 (Unchecked Low-Level-Call) 漏洞

根据 Solidity 的官方文档，Solidity 语言中的低级调用 (call, delegatecall, etc.) 在调用失败时会返回失败布尔值，这些布尔值如果没有被用户捕捉，用户就会错过调用的失败信息，从而造成智能合约功能的混乱。Solidity 语言经过数个版本的演化，推出了相对高级的内建外部调用 (transfer, send, etc.)，这些高级调用会检查调用的返回值，并在发现调用失败时抛出异常。新部署的合约使用高级的外部调用，能避免使用低级外部调用带来的低级调用漏洞。然而，由于区块链语言的特性，智能合约一经部署，无法撤除。根据本文的统计和观察，在以太坊平台上，仍有大量仍在使用的合约在使用低级调用时，对外部调用的调用结果没有任何的检查措施，这些合约有受到低级调用漏洞攻击的危险。

2.1.4　自毁 (Suicidal) 漏洞

Solidity 提供内建的自毁函数 `selfdestruct()`，这个函数需要输入一个地址类型的参数，任何调用这个函数的合约将停止当前合约功能，并把当前合约所

有的余额转移到这个参数的地址。相比直接使用 `send` 函数进行余额转移，使用 `selfdestruct` 消耗的 gas[①]更少。

自毁函数给智能合约提供了方便的功能，但在实际的使用中，如果自毁函数没有被很好的限制，或对调用者的身份进行严格检查，就能造成漏洞攻击。如下面2.2代码所示，`suicidal` 函数实现了简单的自毁函数功能，并通过传参来确定转移余额的目标地址。可是，函数的访问是公开的，任何一个访问该合约的用户都能调用这个自毁函数，并把自己的攻击地址作为余额的转账地址，造成自毁漏洞攻击。Solidity 研发团队在开发文档中提醒开发者要限制自毁函数的调用，并对调用者进行身份检查。

```
1  contract gamble{
2      ...
3      public suicidal(address target) public {
4          selfdestruct(target);
5      }
6  }
```

图 2.2　自毁漏洞示例代码

2.2　前沿智能合约分析工具

2.2.1　静态分析工具

1. 传统静态分析工具

传统静态分析是借助程序的中间语言（IR）、抽象语法树（AST）、控制流图（CFG）等等工具去分析程序特点的分析方法。静态分析方法的主要特点是覆盖率高（即能保证覆盖程序的尽可能多的执行路径），但准确率低。现有的传统静态分析工具代表如 Slither，他的分析路径如下图2.3所示：

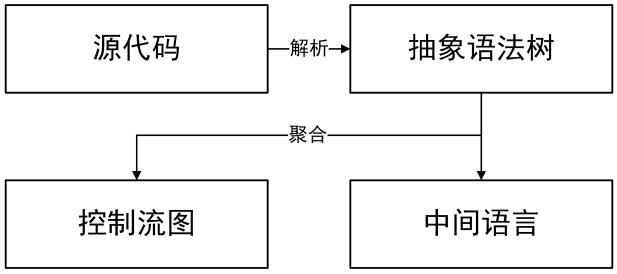

图 2.3　Slither 系统流程图

以源代码作为输入，Slither 首先借助 Solidity 编译器将源代码经过编译，获取了程序的抽象语法树，再在 AST 上通过聚合、规则匹配的方式生成 Slith 中间语言（Intermediate Representation）和控制流图（Control Flow Graph）。由于 Solidity 编译器生成了结构性强的抽象语法树（Abstract Syntax Tree），为分析

[①]以太坊上的每条指令都有的特定属性，表示这条指令消耗的资源多少，gas 为 0 则停止任何服务

提供了方便，故 Slither 能够在 AST 上如此顺利地实现 IR 和 CFG 的转换。其中，Slith IR 是 Slither 的开发团队设计的一款中间语言，对源代码的常见操作做了不同程度的抽象。在以上所有步骤都完成之后，Slither 再借助不同的探测器对生成的 CFG 进行探测。探测器中是 Slither 预先设定好的探测规则，如果在分析过程中发现和探测规则吻合的程序片段，探测器就会报告这个程序片段存在漏洞。故严格意义上来说，Slither 是一款基于规则的漏洞检测软件。一方面来说，基于规则的设定提升了 Slither 的检测速度，也保证了它的可扩展性；另一方面，如果规则的制订不够合理，或者规则的分析程度不能做到抽象性和准确性的平衡，基于规则的检测方法就会带来很高的假正确率（False Positive Rate）。例如，Slither 中针对可重入漏洞的检测规则如下图2.4所示：

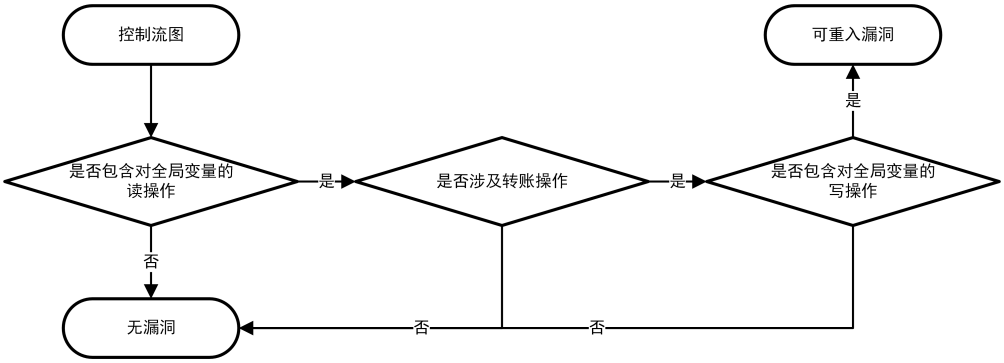

图 2.4　Slihter **对可重入漏洞的判断逻辑**

在这个检测规则中，检测器逐一扫描 CFG 中的各个代码块，如果规则符合，就会报告漏洞。可重入漏洞的规则主要由三部分组成，其中任何一个条件不满足，都会被判断为清白的代码块，这三个条件包括：是否有对全局变量的读取操作、是否涉及交易过程、是否有对全局变量的写入操作。这个规则对可重入漏洞中的某一类型做出了准确地判断，但对其他新型变种可重入攻击，这个规则显得不够灵活。规则的修订过程是无止境的，基于规则的探测器不能保证漏洞检测的鲁棒性。如图2.5所示，在这种潜在的可重入漏洞中，在安全的交易操作后面紧跟了一个危险的外部调用，这个危险的外部函数调用能触发对当前函数的二次访问。在本文看来，即使只重入了一次，只要造成经济损失，就要算作可重入漏洞。在本文观察 Slither 的内部原理后，本文发现 Slither 会漏掉类似的漏洞代码，它在安全的交易操作之后，没有发现对全局变量的写操作，这和 Slither 的检测规则相悖，故停止了检查，从而漏洞了紧随其后的危险外部调用。类似的例子还有很多，对于基于规则匹配的静态分析技术而言，不断修缮规则是一项长期的工作。

2. 符号执行分析工具

符号执行分析工具主要将变量用符号进行标识，并利用约束求解器进行求解，来达到测试软件的墓地。符号执行工具没有像动态分析工具那样使用真实的

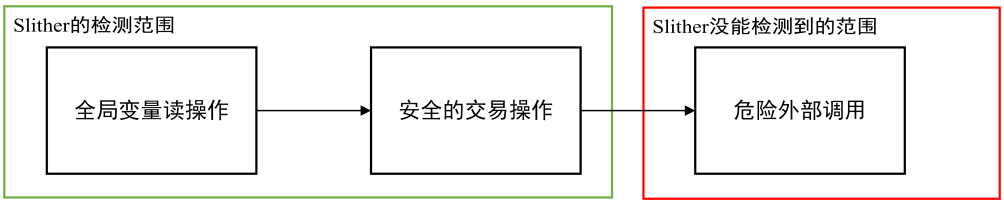

图 2.5　Slither 漏掉的潜在可重入漏洞

用例观察软件的运行情况，而是利用约束求解技术或者其他技术代替真实值执行。在智能合约领域，代表的符号执行分析工具有 Oyente，Oyente 的分析过程如下图2.6所示：

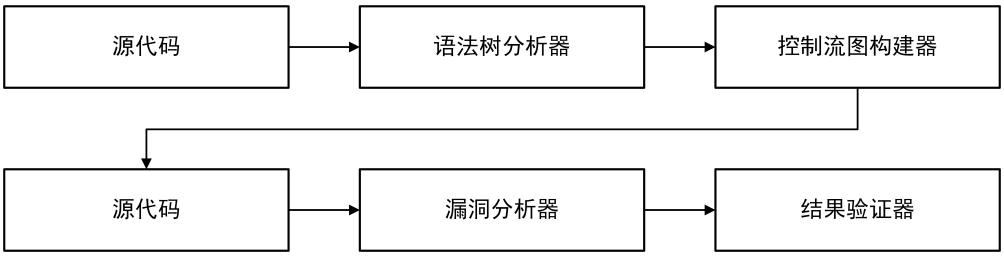

图 2.6　Oyente 系统流程图

Oyente 的输入为源代码，再借助 Solidity 编译器生成 AST，进而生成 CFG，再在 CFG 的基础上用 Z3 约束求解器进行符号执行，最后分析约束求解器的返回值并根据输出报告。Oyente 的探测器在分析约束求解器的结果时，也用了基于规则的方法，如果约束求解器的结果和规则相符，则证明这个程序片段是有可能存在漏洞的。在执行流程的最后还有个验证器负责减少程序的误判。

作为典型的符号执行分析工具，Oyente 充分体现了他的特点，它在同样的数据集上花费了近十倍于 Slither 的分析时间，耗时较长。遗憾的是，Oyente 的结果并不是很好，输出的结果有很高的假正确率，本文推测是由于规则设置的不合理导致的。但不可否认的是，Oyente 作为静态分析工具，能够分析一些传统静态分析工具无法分析的漏洞，比如整数的上溢、下溢，栈溢出等等，这些漏洞无法通过传统分析方法得出。

其他的分析工具如 Manticore，也采用了符号执行分析技术，这种技术带来了更高的精确度的同时也容易引入更差的性能表现。

2.2.2　动态分析工具

动态分析工具主要依赖于动态分析技术进行分析，这种技术使用测试用例作为输入，观察软件运行的状态（内存，耗时等等）和输出结果，进行分析并寻找程序漏洞。早期的动态分析技术多为黑盒测试，使用测试用例生成器不断产生测试用例，输入到软件中并观察结果。黑盒测试不依赖于软件本身的结构信息，只分析软件的运行状态及输入和输出之间的关系。早期的测试用例也是根据程序员本身的领域知识随机产生，由此也带来了测试用例测试效率低的问题

——产生了大量的测试用例，却无法有效地测试到漏洞。为了解决以上问题，灰盒测试以及模糊测试诞生了。灰盒测试借助了静态分析技术来提高性能和表现，而模糊测试通过不同的标准来限制测试用例的数量，追求用最少的测试用例达到最高的测试效果。传统的模糊测试主要以程序的覆盖率作为标准，来引导整个测试过程，并采用了基于模型的或基于突变的测试用例生成器，解决了黑盒测试的主要问题，提高了测试效率。

在 Solidity 这门语言上，静态分析工具有一定的局限性。例如，对于复杂的控制流关系，静态分析技术无法保证漏洞的触发问题，即虽然静态技术能保证很高的路径覆盖率，却不能保证检测到的漏洞能触发。Solidity 语言实现的智能合约，多数功能简单，控制流图相比传统的 Linux 软件要小很多，但即使在这样的软件上，静态分析也不能打到完美的效果。为此，在某些场景下，需要结合动态分析技术进行分析。当下前沿的动态分析工具有 ContractFuzzer，ContractFuzzer 对 Solidity 字节码进行测试，以函数的覆盖率为引导生成测试用例，并修改了底层 EVM 虚拟机以提供更多信息，综合以上信息进行漏洞的分析。虽然以函数的覆盖率为引导的测试在智能合约软件上可能效果不太好，但是不可否认 ContractFuzzer 迈出了重要的一步。同样的模糊测试工具还有 Echidna，它是基于 Haskell 进行开发的，提供对 Solidity 函数的测试功能。但 Echidna 需要手工对 Solidity 的函数进行打桩，这意味着它的扩展性有限，在面对大数据集时可能会耗费大量时间。

2.2.3　其他分析工具

还有其他类型的程序分析工具比如基于形式验证的工具。形式验证是一种高级的软件安全分析技术，它直接分析软件的形式化描述，而不是软件的源代码，并使用数学工具证明软件的安全。形式验证往往能保证较高的安全等级，在一些有很高安全等级需求的软件，如果空客公司的飞行管理软件，需要形式验证来保证软件的安全。

在 Solidity 上，有形式化验证的分析工具 Zeus，但是 Zeus 的团队在投递论文之后并没有公开程序源代码，也没有试图将工具商业化，因此本文没办法观察 Zeus 的运行原理。

2.2.4　现有智能合约分析工具总结

本文收集了当前公开了信息的 Solidity 分析工具的资料，并在表2.1中做出总结：

从表中可以看出，大部分工具现有的检测工具都选择了开源。MythX 选择了商业化，对软件后续的更新进行了闭源处理；Zeus 在 2018 发表于 NDSS，文

表 2.1　Solidity 前沿检测分析工具汇总

Tool Name	Github Stars	Open Sourced	Method	Technique
Mythril	1177	✓	Dynamic	Constraint Solving
MythX	n.a.	×	Dynamic	Constraint Solving
Slither	247	✓	Static	CFG Analysis
Echidna	306	✓	Dynamic	Fuzzy Testing
Manticore	1546	✓	Dynamic	Testing
Oyente	663	✓	Static	Symbolic Analysis
Securify	129	✓	Static	Datalog Analysis
SmartCheck	47	✓	Static	AST Analysis
Octopus	153	✓	Static	Reverse Analysis
Zeus	n.a.	×	Static	Formal Verification
ContractFuzzer	34	✓	Dynamic	Fuzzy Testing

章中 Zeus 具有着卓越的效果，但在发表文章之后，Zeus 的开发团队并没有把 Zeus 开源的想法。在现在的这篇工作中，本文无法接触到这两种工具，无法从内部剖析他们检测漏洞的原理和规则，因此本文只能讨论现有的开源工具。有趣的是，在现有的检测工具中，采用静态检测方法的工具和采用动态检测方法的工具各占一半，静态工具速度快，但动态工具的检测准确度高，两个类型的工具互相结合，才能有更好的检测效果。

2.3　智能合约字节码相关知识

字节码是一种经过 Solidity 编译后生成的十六进制代码串，能直接被 EVM 运行。字节码中的那部分字节都可以和特定的操作对应起来。如表2.2所示：

表 2.2　操作码 gas 消耗对应表

Opcode	Name	Description	Gas
0x00	STOP	Halts execution	0
0x01	ADD	Addition operation	3
0x02	MUL	Multiplication operation	5
0x03	SUB	Subtraction operation	3
...	...	...	...

操作名和字节码中的十六进制码一一对应，每个操作都有特定的意义，并且有着不同的 gas 消耗量。因此，本文分析字节码的第一步应该从对应表着手，先将十六进制码转换成对应的操作名，区分出操作码和操作数，并将字节码翻译为汇编码。

仅仅将十六进制码转换为汇编码是远远不够的。在学习相关文献和黄皮书后，本文对 Solidity 字节码的结构有了更深刻的认识。Solidity 将源代码编译后，会生成构造函数（Creation Function）和运行时函数（Runtime Function）两部分。这两部分中，有大量代码和源代码没有关系，是 EVM 为了更好的运行在编译时加上的。

一个完整的构造函数，除了构造器本体，还包括重置内存指针（Free Memory Pointer）、支付检查（Non-payable Check）、构造器参数获取（Retrieve Constructor Parameters）、拷贝运行时函数地址（Copy Runtime Function Address）等等部分。这些代码除了构造器本体外，其余的部分都是和源代码无关的代码，因此本文在反编译字节码时，可以考虑去掉相关代码，考虑到这些代码在大多数情况下不会有太多变化，这个环节的实现难度适中。运行时函数包括了源代码的主体部分，体量也比构造函数大不少。在运行时函数的开始部分，会先重置内存指针、支付检查，随后会将调用的函数名称传入函数选择器。函数选择器中存放了源代码中各个函数的函数签名，将传入的函数名称经过 hash 和函数签名进行比对，就可以得到各个函数在内存中的位置。在获取函数位置之后，并不能马上直接运行函数主体，EVM 会先运行函数包装器（Function Wrapper）中的内容。包装器的主要功能包括保管函数入口地址、清理内存、接受函数返回值等等，它的存在保证了函数的正常稳定运行。在包装器准备好之后，才会进入函数的主体部分。主体部分包括了源代码中的绝大部分内容，故在反编译过程中，如果能跳过函数选择器和函数包装器，直接找到函数的主体部分，将会大大减少本文的分析代码量。

此时汇编码还没有结束，在汇编码的最后部分是大量的无效代码，这部分的十六进制代码无法和对应表中的任何操作联系起来。事实上，这部分的代码并不是真的无效代码，这部分的代码称为元数据 hash（Metadata Hash）。根据黄皮书介绍，每个合约部署在区块上时，会根据合约的信息（函数签名、构造器参数、函数数量等等）生成一个 hash 值，每个合约的 hash 值都是唯一的，这个 hash 值称之为元数据 hash，放在运行时函数的最后部分，作为合约的指纹小心存放。在构造函数和运行时函数中并不会有任何的函数调用会涉及这一部分的代码，它们仅作为合约的唯一代码在部署时被使用。

第 3 章　主要研究方法及技术路线

3.1　现有工具漏洞检测能力

在开始进行实验之前，本文需要对现有工具的检测能力进行充分地调研。只有在了解现有工具的检测能力、检测特点的情况下，才能开始进行漏洞标准库的构建和新工具的研发工作。为此，本文调研了当下对 Solidity 的研究工作，有的工作来自于商业团队，有的来自于学术团队；有的工具已经开源，并具备一定的社区影响力，有的工具发表于计算机顶级国际会议，带来了巨大的科研价值。发表于国际会议的工作，有的没有开源，对于这些没有开源的工作，虽然有论文做详细的说明，但由于不能获取到源代码，本文没办法对系统的内核做更深一步的分析，所以这些工具尽管有一定的学术影响力，也只能放弃。对于已经开源的工作，有些工具的开发逻辑不够严谨，或者相关文档不够完备，对于这些工具，本文虽然能取得它们的源代码，但由于无法清晰地分析系统实现，故这些工具本文也无法很好地去分析他们的检测原理和检测能力。综上，在经过本文的筛选后，对如下工具在主要漏洞上的检测能力做出了总结，并做比较列于表3.1。

表 3.1　现有工具对于主要漏洞的检测能力总结

	Slither	Oyente	Smartcheck	Securify
可重入漏洞	✓	✓	×	✓
意外异常漏洞	✓	×	✓	×
低级调用漏洞	✓	×	✓	×
自毁漏洞	✓	×	×	×

上表所列的工具皆为静态分析工具，其中 Oyente 主要使用符号执行分析技术；Securify 主要把代码转换成 Datalog 语言，并使用 Souffle 进行分析。Slither 和 Smartcheck 采用的是传统的静态分析技术，即通过分析源代码得到程序的控制流图，并在控制流图上用预先设定好的匹配规则去寻找漏洞。从表中不难看出，使用传统静态分析技术的工具分析能力都比较不错，Slither 支持本文提及的所有漏洞的检测，Smartcheck 不支持两个漏洞的检测；而使用符号执行分析技术，包括使用其他静态分析技术的工具，对主要漏洞的支持都不太好，甚至只支持一个漏洞的检测。值得一提的是，这两个工具 Oyente 和 Securify 皆是在源代码编译之后产生的字节码上进行软件分析的，字节码的分析提供的信息更少，相比之下 Slither 和 Smartcheck 都是对源代码或者中间语言进行分析，故本文推测是由于技术路线的差异造成它们在不同漏洞上的分析难度不同，也就没办法支持所有漏洞的分析任务。

3.2　使用克隆分析技术寻找漏洞代码

在克隆代码分析技术中，按照代码相似的不同程度，本文可以把代码划分为以下几种克隆层次：

- **第一类克隆：完全克隆。**这种克隆下层次下的相似代码之间完全相似，没有任何差异。
- **第二类克隆：重命名克隆。**这种克隆层次下的代码之间绝大部分相似，在类型、标识符、注释、空格之间有些许差异。
- **第三类克隆：重构克隆。**这种克隆层次下的相似代码之间具有结构层次的不同，例如缺少部分语句，多出部分语句，语句顺序不同等等。
- **第四类克隆：语义克隆。**这种克隆层次下的相似代码之间可能完全不同，但他们具有相同的语义，实现了相同的功能或流程。

从克隆层次的分类可以看出，第一类克隆和第二类克隆不涉及代码结构上的变化，因而能用比较简单的技术进行分析。在 Kamiya 等人的工作中[28]，使用了基于标记的克隆分析技术来寻找克隆代码，将代码的关键部分转换为标记，再在标记上进行分析寻找。由于前两类克隆代码的分析并没有什么挑战，因此现在已经有很多这方面的工作。对于第三类克隆，因为代码之间出现了语句结构的差异，例如多的语句，少的语句等等，直接借助基于标记的克隆分析技术来寻找克隆代码可能会遇上很多困难。为了解决这一问题，有人提出了提取代码特征转换成特征向量，并在高维空间进行比较的办法[7]，也有的工作提出使用软件的控制流图进行语句结构的比较[29]。而对于第四类克隆的分析，仍然是当前软件工程学界的一个有挑战的课题。有的工作提出使用深度学习算法进行代码语义的提取[30]，但仍有很大的局限性如学习算法的数据集匮乏，很难找到数量充足且质量上乘的训练材料。因此，在讨论克隆分析技术时，本文主要解决的是寻找前三类克隆的相似代码的问题。

针对以上三种代码克隆等级，之前的工作提出了很多不同粒度的解决方案：

1. **基于标记粒度的克隆分析技术：**使用基于标记力度的克隆分析技术试图使用将代码语句转换成标记序列，然后再在标记序列上比较相似度。这其中最出名的工作有 CP-Miner[31]。CP-Miner 解析了程序代码，并使用了“最频繁子序列挖掘”算法对代码生成的标记序列进行比较。这个算法由 CloSpan[32] 这篇工作提出。多亏了 CloSpan 这篇工作在改进算法运行效率方面的贡献，CP-Miner 可以在大规模代码，如 Linux 内核代码下仍保持了较低的内存占用。但是，CP-Miner 的运行时间复杂度在最坏情况下为 $\mathcal{O}(n^2)$，n 为代码行数，运行耗时较长。除开在大规模代码下的效率问题，CP-Miner 也容易产生很多误报，这个是由于他们激进的代码抽象策略及

有筛选的遗产算法导致的。虽然 CP-Miner 的开发者认为 CP-Miner 在数据集上的表现不错，但很明显 CP-Miner 并没有在漏洞代码检测这项任务上有足够的可靠行。

2. **基于代码行粒度的克隆分析技术：**在 ReDeBug[33] 中，分析系统将代码行的集合作为处理单元。系统驱使一个 n 行（n 默认为 4）的窗口在源代码中滑动，并在每个窗口上使用三种不同哈希函数。该系统通过对比两文件各窗口的哈希值来计算文件之间的相似程度。虽然 ReDeBug 的这个特性使它能够检查一些第三类克隆的克隆代码，但它却无法检查那些第二类克隆，即变量名或者数据类型有变化的克隆代码。因此，ReDeBug 会漏掉很多有漏洞但差异很小的克隆代码。更进一步，使用基于代码行粒度的克隆分析技术会导致上下文信息被局限一个很小的范围内，并最终导致引入了很多的误报。同时，ReDeBug 需要花费大量的时间去处理源代码文件并建立哈希库，性能表现欠佳。
3. **基于函数粒度的克隆分析技术：**SourcererCC[34] 使用了基于函数粒度的克隆分析技术，试图来检测第三类克隆的克隆代码。它使用了标记集的检测技术解析了所有的函数，并针对每个函数的标记集建立了检索目录；然后，它寻找两个函数间相同的标记，并使用了 Overlap 函数计算这两个函数之间的相似度。如果这个相似度超过了人为预先确定的一个阈值，则判断在这两个函数之间存在代码克隆现象。该系统在实现时，为了减少相似度的计算次数，对标记按出现的频数进行权重计算，出现频数高的标记获得较高权重，对持有较高权重的标记进行计算。但是，在权衡检测第三类克隆代码的能力与检查漏洞的能力时，SourcererCC 对与漏洞代码的检测能力受到了很大的限制。在很多情况下，打过补丁的代码（安全代码）与未打补丁的代码（漏洞代码）之间的差距非常小，甚至只有一个 `if` 语句的差距，SourcererCC 也无法检测这些漏洞代码。

   Yamaguchi 等人提出使用漏洞推导的方法[35] 来分析第四类克隆，即语义克隆的代码。他们分析函数的抽象语法树，并提取语法树的特征并将其嵌入向量空间中；在完成提取工作后，他们对高维向量使用奇异值分解来获取函数的结构信息。虽然他们的方法具备了检测一定程度的第四类克隆的能力，但是他们的系统运行流程有太多的时间和空间消耗，并且在论文中他们也没有明确给出这种技术的准确程度。

   本文必须指出，在使用高级别的代码抽象技术（标记序列，语法树）来分析克隆代码可能对分析代码克隆是有帮助的，但他们不足以准确地分析相似的漏洞代码，因为这些漏洞的语境通常会非常复杂。
4. **基于文件粒度的克隆分析技术：**DECKARD[7] 对每个源代码文件都分别构

建了抽象语法树，并从文件的抽象语法树上提取了特征向量，再在特征向量上使用欧几里得距离来进行聚类，经过聚类，欧氏距离较近的文件则被判断为克隆代码文件。这种基于语法树的方法需要大量的执行时间，因为子图同构是一个著名的 NP 完全问题。更进一步说，DECKARD 并没有保证足够的扩展性，在面对大数据集时表现差强人意，同时，DECKARD 也带来了很高的误报率，这也说明具有相同语法树结构的代码片段可能不是克隆代码。

FCFinder[36] 去除了代码的注释、重复空格、换行，再对代码用 MD5 算法计算哈希值。它建立了一个哈希表，表的键值为文件名，数值为对应文件的哈希值。如果发现有文件的哈希值重复，则判断这几个文件为克隆代码文件。相比于 DECKARD，FCFinder 具有了良好的可扩展性。再 FreeBSD 软件上的克隆检测上，耗时更少，同时保持了一定的准确率。可是，和 DECKARD 一样，它也不能很好地检测高度相似但具有微笑不同的克隆代码。

5. **混合粒度的克隆分析技术：**有一些工作使用了几种不同粒度的克隆分析技术。VulPecker[37] 是一个能自动检查漏洞的软件分析系统。它给漏洞加上了事先定义好的特征，再根据代码的实际情况算则合适的代码相似度算法（如最长公共字串算法）计算代码相似度。在这个分析技术的帮助下，它找到了 40 个未被 NVD（National Vulnerability Database）记录在案的漏洞。可是，这个系统在大量代码的情况下耗时过长，无法应对大量代码的检测任务。

综上所述，对于不同粒度不同情况下的克隆代码分析，之前的工作做出了相当的努力。同时，也不难看出，要达成使用克隆分析技术来寻找代码漏洞的目标，不仅要保证本文设计的检测器具备一定的扩展性，以应对检测大量的代码的情况；其次也应该选用合理的代码抽象代码，来提取不同漏洞的特征，借助提取的特征来匹配相似的代码片段；最后，代码相似度计算算法的选择对分析克隆代码的能力至关重要，本文应该合理地选择代码相似度的计算算法。

3.3　在智能合约软件上使用克隆分析技术的可行性

在上一个部分本文调研了现有的软件克隆分析技术在不同的软件克隆等级上的效果和优缺点，本文也提出了要实现使用克隆分析技术去寻找漏洞代码，不仅要谨慎选择相似度算法，也必须保证系统即使在面对大量代码时也能维持较快的检测速度。但是，目前还没有工作在智能合约上使用克隆分析技术去寻找漏洞代码，因此，在这一部分，本文将证明在智能合约软件上使用克隆分析技术是可行的。

3.3.1　智能合约代码相似度观察

在本文观察了大量的智能合约软件代码过后，本文发现，在以太坊平台上存在着大量的代码克隆现象。在本文从 Etherscan[①] 上爬取了接近十万的智能合约代码，在使用最长公共字串来计算代码间相似度之后本文发现，统计代码相似度层级如下图3.1所示。可以明显看出，大部分智能合约软件内部存在着代码克隆

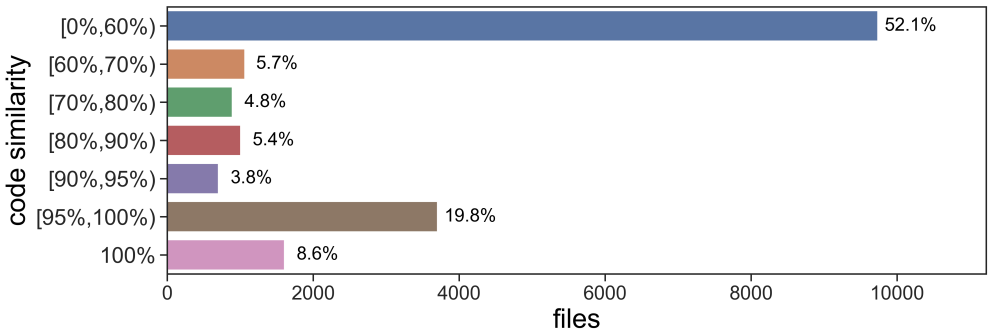

图 3.1　智能合约代码相似度统计

现象，本文推测这是由于 Solidity 代码无法引用代码引起的。很多智能合约软件直接复制已经存在的软件代码，稍加改动，如修改了交易地址，甚至不改动就添加到自己的代码中，参与自己软件的运行过程，如图3.2所示。很明显，这虽然方便了开发者，减轻了开发任务，但是直接拷贝代码的行为容易在无意间引入漏洞。在一个存在很多克隆代码的平台，要保证软件的平均安全等级是很困难的。Solidity 研究团队推荐开发者拷贝或使用经过官方团队审计的接口代码，但是不推荐开发者拷贝其他任何代码。

```
contract BanyanIncomeLockPosition is Ownable {
uint64 public unlockBlock = 8372051;
address public tokenAddress = 0x35a69642857083BA2F30bfaB735dacC7F0bac969;
...
}
```

```
contract BanyanIncomeLockPosition is Ownable {
uint64 public unlockBlock = **6269625**;
address public tokenAddress = **0x0a3f9678d6b631386c2dd3de8809b48b0d1bbd56**;
...
}
```

图 3.2　相似代码示例，两个地址不同的智能合约软件仅有微小不同，不同之处已用 ** 标出

3.3.2　智能合约克隆分析技术探究

针对以太坊平台如此严重的代码克隆现象，本文提出使用代码克隆分析技术来寻找漏洞的方法。在 VUDDY[21] 这篇工作中，研发团队使用了根据漏洞代码的语法特征提取了漏洞的抽象表示，再通过相似度计算算法来寻找漏洞代码。

[①] 一个提供智能合约数据及分析服务的网站。

VUDDY 的这种技术，是从漏洞本身出发，寻找与漏洞相似代码的一种技术。这种方法具备良好的可扩展性，在寻找大量相似代码时也能保证令人满意的速度，在 VUDDY 的论文中，开发团队使用 VUDDY 完成了数以亿计行的代码的漏洞检测工作，并成功申请了数十个 CVE[①]。VUDDY 的工作给了本文启发，在智能合约上，本文需要使用一种足够巧妙的方法去提取漏洞的抽象表现形式，抽象程度不能太高或者太低，太高会误报很多漏洞代码，太低则会漏掉很多真实的漏洞代码；本文还需要使用足够巧妙的代码相似度算法去计算代码之间的相似度，这种相似度计算方法必须和代码的抽象表现形式结合，不能有太高的时间复杂度；最后，本文需要对系统报告的漏洞做一定程度的人工分析，虽然在严谨的实验中，人工的行为容易带来实验结果的偏差，但是本文参与实验的检查人员都是智能合约、区块链及软件工程方面的领域专家，可以尽可能减少人工介入带来的偏差。那些经过人工检查，或者被系统报告有漏洞的代码，本文会用来不断地改进系统的检测能力，并建立漏洞标准库。

3.3.3　结合克隆分析技术与字节码分析技术

本文的系统不仅提供对智能合约源代码的漏洞检测功能，也能完成对字节码的反汇编任务以及控制流图分析。Oyente[12] 在字节码的基础上，使用了反汇编技术结合控制流图分析，并加上了符号执行分析技术，实现了漏洞的检测。无独有偶，之后的符号执行技术如 SCompile[38]，Manticore[14] 也是在字节码上使用了符号分析技术，并结合自己确定的漏洞规则来进行检测。在本文的系统设计中，本文并不打算在字节码上加入符号分析技术并对字节码进行漏洞检测，本文只提供对字节码的代码结构静态分析，这是因为经过本文的广泛调研，如今字节码的反编译技术及语义解析技术并不是很成熟，本文觉得不足以支撑实现一个具有一定准确率的漏洞检测系统；其次，加入符号分析技术的工作量过大，超出了这篇论文的研究范畴，本文也希望相关同僚在字节码的漏洞分析检测技术上取得进一步的突破。

3.3.4　漏洞检测系统结构设计

本文提出了智能合约漏洞检测系统 Athena，整体的系统结构如图3.3所示。系统使用智能合约软件作为输入，分析结果及标准数据库作为输出。在系统的开始，会对输入的智能合约软件做相关检测，如果输入的智能合约软件为源代码形式，则进行源代码分析及进一步的漏洞检测；如果输入的智能合约软件为字节码形式，则对字节码进行反汇编，并将得到的汇编码作为输入进行控制流图分析。在结束控制流图分析之后，本文将对软件进行漏洞规则匹配。漏洞规则是经过本

[①]Common Vulnerabilities and Exposures，著名软件漏洞库和漏洞审计平台

文智能合约领域专家总结得出的一套漏洞模板，并进行一定程度的抽象。再之后，本文用相似度算法计算漏洞规则与输入的控制流图进行相似度计算，如果得到的相似度高于系统设定好的阈值，就判断这是一个可疑的漏洞代码，将把这段代码保存并交由领域专家进行进一步检查。加入人工检查的目的是为了漏洞检测系统的公平公正，当下的对于智能合约漏洞检测的论文中对于漏洞的定义并不一致，且尚没有一个高度完备的工作能囊括所有的漏洞形式，因此，为了方便标准库构建和系统的优化，本文需要专家进行一定程度的人工干预，对系统的检测结果进行二次检测。如果专家检查到了系统的误报，说明本文总结的漏洞检测规则仍有不完备的地方，同时本文将根据误报的漏洞形式进一步完善系统的漏洞检测规则。本文将根据漏洞的误报形式来决定是否改动漏洞检测规则，并不是所有的误报都要进行修改。这也是进行人工干预的优点，可以选择需要改进的地方进行恰当的改进。最后，对于那些系统正确判断的软件漏洞，本文将把漏洞代码加入漏洞库。漏洞库是本文标准库构建工作的结晶，经过不断修缮也能给后续的智能合约软件分析工作带来长远的便利。出于人道主义和安全考虑，本文并不会公开漏洞库的所有漏洞，但是本文会匿名掉关键信息并展示部分漏洞代码作为案例分析的一部分。随着和漏洞代码的作者联系不断进行，如果软件作者已经做出相关回应，修复了漏洞，本文也会在将来公开更多的漏洞代码用作学习和警示。

3.4　基于规则的漏洞检测技术

对于本文需要研究的主要漏洞，本文需要从现有的漏洞示例和现有工具的检测策略中不断学习，加上领域专家对以太坊平台及智能合约的认识，总结归纳出漏洞的检测规则，以便于之后使用克隆分析技术在漏洞检测规则的基础上寻找漏洞代码。对于现有的几个开源工具，本文可以通过阅读源代码的方式去研究他们的实现原理，分析他们的制定的漏洞规则的优缺点，不断完善系统的漏洞检测规则。在本节，本文将针对不同的漏洞进行分析，并得到适合用作克隆分析技术的漏洞检测规则。

3.4.1　可重入漏洞检测规则

对于影响力巨大的可重入漏洞，Slither 在实现中使用的检测器是以式3.1为规则进行检测的。

$$r(var_{global}) \wedge (operation_{sendeth}) \wedge w(var_{global}) \Rightarrow \text{Reentrancy} \tag{3.1}$$

Slither 在检测可重入漏洞时，会首先检查控制流图中的全局变量读操作，即 $r(var_{global})$，这一步是因为全局变量在以太坊中有着重要的作用。由于以太坊的

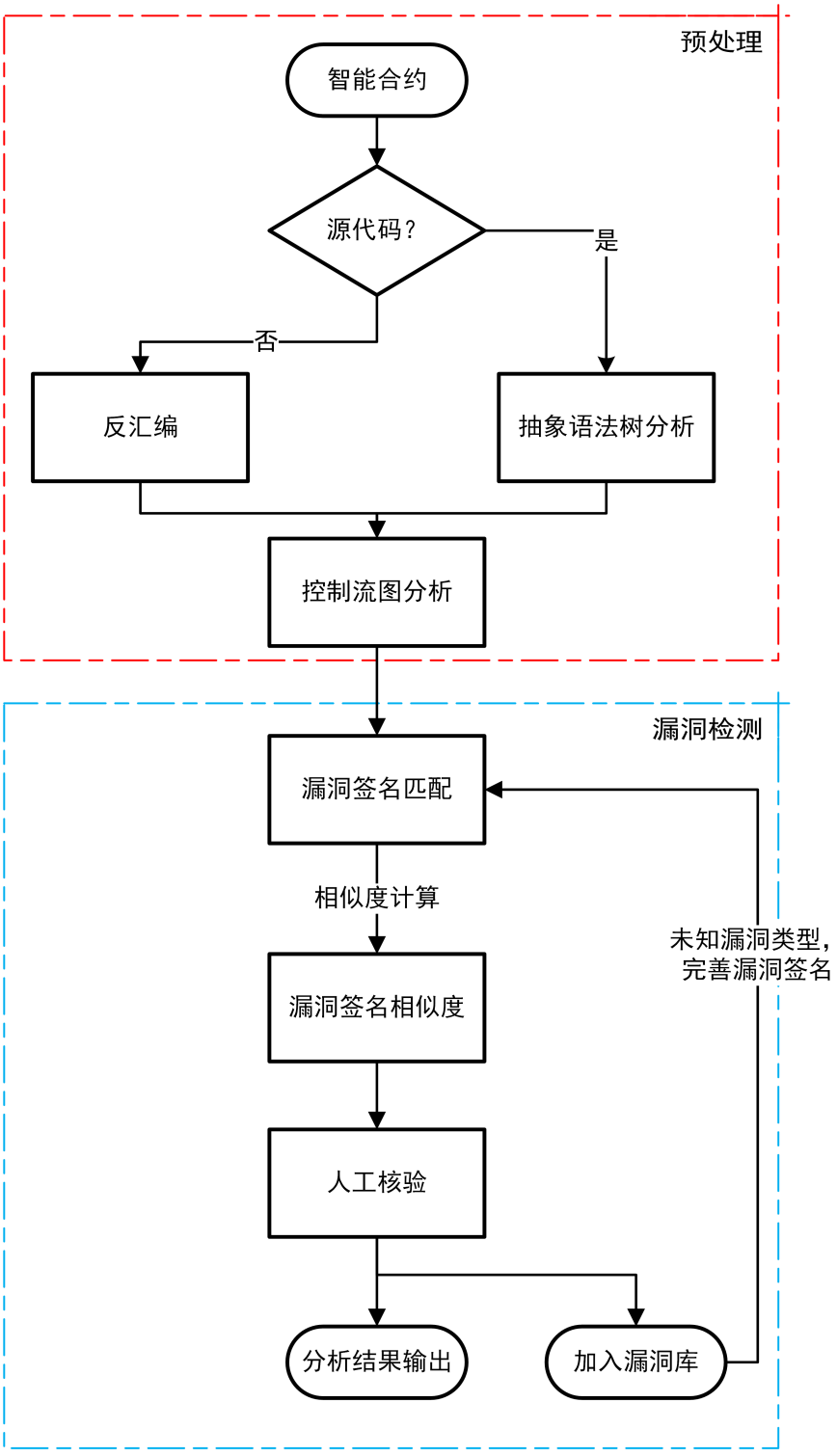

图 3.3　漏洞检测系统结构设计图

底层运行是基于操作栈实现的，全局变量在以太坊通常是以 `public` 类型修饰，作用域包括整个合约。因此，对于全局变量的操作容易带来安全隐患，而对全局变量的读取操作则是对全局变量修改的第一步。Slither 重视对全局变量的敏感操作，因此会检查对全局变量的读取和写入操作。更进一步，为了保证检测规则的准确性，Slither 优先检查涉及转账操作 $operation_{sendeth}$ 的全局变量修改，因为

在转账操作周围的全局变量读写操作通常会带来收到漏洞攻击危险。Slither 在控制流图上进行分析，控制流图的每个块内部都是中间语言形式的代码语句，这种中间语言在源代码的基础上做了一定程度的抽象，有利于 Slither 无视无关代码进行准确的漏洞检测。

Oyente 的检测规则如式3.2所示。

$$r(var_{global}) \wedge (gas_{trans} > 2300) \wedge \left(type(var_{amount}) == type(var_{global})\right) \Rightarrow \text{Reentrancy} \tag{3.2}$$

Oyente 的检测规则和 Slither 有很大的不同，因为 Oyente 是对智能合约的字节码进行检测的工具。对字节码反编译过后的中间语言建立控制流图，并进行分析。因为字节码的操作码和操作所消耗的 `gas` 是一一对应的，如表2.2所示，因此，对字节码进行分析可以分析函数的 `gas` 消耗，这是以源代码为输入的分析方式所做不到的。同时，Oyente 检查了外部调用操作消耗的 `gas` 是否超过 2300，这是因为可重入漏洞的一个最大的特点就是针对重复调用并不断消耗 `gas`。如果发现了外部调用大于 2300 这个阈值，则很有可能发生了重入。最后，Oyente 检查了交易的金额变量是否为全局变量。这是因为全局变量经常涉及敏感操作，而控制转账金额的变量决定了受害者受到的损失的大小。Oyente 的规则要求控制转账金额的变量必须不为全局变量。

Securify 的检测规则如式3.3所示。Securify 的检测规则相比 Slither 和 Oyente 要简单很多。Securify 是基于 Datalog 语言做的符号执行分析，它会在检测漏洞之前将智能合约字节码转换成 Datalog 语言。Securify 在检查可重入漏洞时，会检查语句块内部有没有外部调用和对全局变量的操作，避免发生敏感操作。其次，如果检测器发现对全局变量的操作没有放在外部调用之前，检测器会报告这是一个可重入漏洞。Securify 的检测逻辑简单易懂，它抓住全局变量和外部调用的先后关系进行检测，Datalog 语言也对产出准确的检测结果有一定的帮助。但是，相比其他两个工具，Securify 的检测模式还是过于简单和粗暴，这可能会带来很多的误报。

$$external\_call \wedge w(var_{global}) \wedge (\text{ write must follow the call}) \Rightarrow \text{Reentrancy} \tag{3.3}$$

通过对其他三个工具的观察和对智能合约漏洞的理解，本文认为，一个完备的可重入漏洞规则，应该是如式3.4所示。在本文的检测规则中，并没有采用 Oyente 的检测规则中对操作消耗 `gas` 检查，因为本文的检测工作是面向源代码而不是字节码，在源代码上做 `gas` 消耗分析会产生很多困难；也没有采用 Securify 中对于外部调用和全局变量操作的前后关系进行检测的规则，因为这个规则容易带来误报。本文吸取了 Slither 的检测规则中对交易操作与全局变量写入操作的检查，并加入了没有 `gas` 限制的危险调用的检查。这是因为早期的 `call`、`delegatecall`

等低级调用如果不加限制，很容易带来重入隐患。如果出现没有 gas 限制的危险调用，就可以认为这能够带来一定的漏洞隐患；同时如果出现在转账操作之后有相关的全局变量写入操作，也同样认为会带来漏洞隐患。

$$\big((dangerous\_call) \wedge (no\_gas\_limit)\big) \vee \big((operation_{sendeth}) \wedge w(var_{global})\big) \Rightarrow \text{Reentrancy} \tag{3.4}$$

3.4.2　意外异常漏洞检测规则

意外漏洞多发生于在 for 或者 while 循环中进行交易操作的情况，账户持有者向其他人循环发送金额，如果在循环中的某一次转账操作发生异常导致转账异常，则后面的数次转账操作都将被取消。这个漏洞容易被黑客用于 DoS 攻击，如果黑衣故意在转账过程中加入异常代码，使得整个交易次数取消，就达到了拒绝服务攻击的目的。

对于意外异常漏洞，支持的工具有 Slither 和 Smartcheck，而 Oyente 和 Securify 因为是在字节码的基础上做进一步分析，很难抓取到这个漏洞的特点，因此这两个工具不具备检查这个漏洞的功能。Slither 对于这个漏洞的检测规则如式3.5所示。

$$(has\_loop) \wedge (calls\_inside) \Rightarrow \text{Unexpected Revert} \tag{3.5}$$

在式中，Slither 首先检查控制流图中的循环结构，如 for 或者 while 循环，然后检查在循环结构的内部是否包含外部调用语句。在检查外部调用的时候，Slither 检查高级外部调用（即用户定义的函数的调用）、低级调用以及转账函数 send 和 transfer。如果有其中任何一种调用出现在循环结构内，Slither 则判定这段代码含有意外异常漏洞。

Smartcheck 检测意外异常漏洞的规则如式3.6所示。

$$\big((if) \wedge (revert\_inside)\big) \vee \big((for) \wedge (calls\_inside)\big) \Rightarrow \text{Unexpected Revert} \tag{3.6}$$

Smartcheck 在处理意外异常错误的时候会检查两种情况，一种是 if 语句和函数体内部有没有 revert，如果同时成立则有了抛出异常的可能性，引发意外异常漏洞；一种是在 for 循环语句的内部是否包含外部调用，如果外部调用失败则有可能会造成意外异常错误。Smartcheck 和其他工具不同的地方是把 revert 也当作漏洞检测规则的一部分，在本文看来 revert 多用于需要抛出异常的函数体内，属于正常的业务逻辑，不能被归纳为漏洞的一部分。同时，Smartcheck 对一切的外部调用一视同仁，只要出现在 for 循环内部则判断为有漏洞，这种判断规则有些过于简单和武断。

在观察过其他两个工具对于意外异常漏洞的检测规则后，制定本系统的检

测规则如式3.7所示。

$$(for \vee while) \wedge \big((require \wedge send) \vee transfer\big) \Rightarrow \text{Unexpected Revert} \tag{3.7}$$

在本文的规则中，首先检查函数体是否包含 `for` 或者 `while` 循环语句，再检查函数体内部有没有 `require` 语句和 `send` 语句，如果没有的话再检查函数体内部是否包含 `transfer` 语句。制定这样的检测规则是因为 Solidity 的部分内建语句自带了抛出异常功能。例如，`transfer` 语句有抛出异常的功能，而 `send` 语句没有。所以如果使用 `send` 语句，当调用执行失败，`send` 会返回 `false`，但不会抛出异常。如果 `send` 语句外部有 `require` 函数对交易的执行结果进行检验，`send` 语句就具备了抛出异常的功能，也就有了触发意外异常漏洞的风险。

3.4.3　低级调用漏洞检测规则

低级调用漏洞主要是由低级漏洞自身的安全检查不完备造成的。多数低级调用，如 `call`、`delegatecall` 都是 Solidity 语言诞生初期的产物。这些低级调用实现了最基本的功能，但缺少了一些安全检查，导致使用这些低级调用的行为变成引发漏洞的行为。Solidity 不推荐开发者使用这些低级调用函数，如果实在有使用的需要，要求开发者在低级调用的外部加上 `require`、`assert` 等检查函数。如果低级调用执行失败，这些检查函数会收到低级调用的返回值 `false`，并抛出异常终止整个合约的运行。

Slither 对于这种低级调用漏洞的检测规则如式3.8所示。

$$(call) \vee (delegatecall) \vee (send) \Rightarrow \text{Unchecked Low Level Call} \tag{3.8}$$

Slither 对于这个漏洞的检查规则非常简单粗暴，它检查函数控制流图上所有的低级调用操作，一旦出现低级调用，就判断函数出现低级调用漏洞。毋庸置疑，这会给 Slither 在这个漏洞的检测任务上带来很高的误报率。

Smartcheck 对与这种低级调用漏洞的检测规则如式3.9所示。

$$(if) \wedge \big((call) \vee (delegatecall) \vee (send)\big) \Rightarrow \text{Unchecked Low Level Call} \tag{3.9}$$

Smartcheck 对于这个漏洞会首先检查 `if` 语句的使用，如果出现了这个语句，再检查语句内部是否包含 `call`、`delegatecall` 或者 `send` 这三个低级调用，如果有的低级调用没有被包含在 `if` 语句内部则判断代码包含低级调用漏洞。Smartchecl 认为只有 `if` 语句才能算作检查，而忽略了 `require`、`assert` 这两种检查函数，这会忽略掉很多真正的漏洞。

在观察其他两个工具对低级调用漏洞的检测规则之后，制定了本系统的检测规则如式3.10所示。

$$\big((call) \vee (delegatecall) \vee (send)\big) \wedge without\big((require) \vee (assert) \vee (if)\big) \Rightarrow \text{Unchecked Low Level Call} \tag{3.10}$$

本文的规则覆盖了主要的低级调用如 `call`、`delegatecall` 和 `send`，同时也附加了对几种主要的检查函数 `require`、`assert` 和 `if` 的检查。本文认为 Slither 的规则会产生很多误报，因为它的规则过于武断，没有对检查函数的检查环节；也认为 Smartcheck 的规则也是不完备的，因为它只检查 `if` 这一个检查函数，而没有覆盖其他的两个检查函数，这也会对检查结果产生影响。在分析以上两个工具的规则的优缺点之后，本文设计了自己的检查规则。

3.4.4　自毁漏洞检测规则

自毁漏洞是也是由于函数的正常功能在不正当使用条件下使用产生的。`selfdestruct` 函数实在 Solidity 语言设计之初就加入的函数，它停止当前的任何功能，清空 `gas`，并把当前合约剩余的余额全部转移至其他合约。这个内建函数有一个参数，即转移余额的地址。这个函数本属于智能合约正常交易周期的一部分，负责在合约完成其生命周期之后清理剩余的合约资源，避免合约当中的资源浪费。但是，如果这个函数没有受到严格的调用限制，比如在特定的时间、特定的运行次数或者特定的访问者调用，甚至转账的地址是不受到合约的持有者控制的，这个函数就会引发自毁漏洞。攻击者触发函数，停止合约的当前功能，并向攻击者的账户转移所有财产。

Slither 对于这个漏洞的检查规则如式3.11所示。

$$(selfdestruct) \wedge (func_{public}) \wedge (access\_available) \Rightarrow \text{Suicidal} \tag{3.11}$$

Slither 在检查这个漏洞时候，首先检查这个函数受否包含 `selfdestruct` 自毁函数，其次检查包括这个自毁函数的外部函数修饰符是否是 `public`，最后检查这个自毁函数是否能被轻易到达。Slither 的检测规则目前已经比较完备，但还有一点不足，Slither 这样的检查方式会把正常的业务逻辑也判为漏洞代码。因为正常的业务代码函数体积较大，在函数的最后有自毁函数，这个函数符合 Slither 的检测规则，但是它对自毁函数的使用是正确的。

为了改正 Slither 的一点缺陷，提出本系统的检查规则如式3.12所示。

$$(selfdestruct) \wedge (func_{public}) \wedge (access\_available) \wedge (func_{selfdestruct}) \Rightarrow \text{Suicidal} \tag{3.12}$$

本文的检测规则在 Slither 的基础上加入了新的要求，要求只有当检测的函数为自毁函数体，即只包含自毁功能的函数时，才判断这个函数为漏洞函数。这个规则修正了 Slither 把很多正常的业务逻辑误判为漏洞的缺陷，降低了系统的误报率。

3.5　使用 CPT 降低系统误报数

合约保护技术（Contract Protection Technique）指开发者在开发过程中使用特定的技术保护代码不受漏洞侵害的技术。在确定针对不同漏洞的检测规则之后，可以借助克隆检测技术，在程序的控制流图上寻找符合漏洞检测规则的代码。如果出现了符合规则的代码，则将这段代码报告为漏洞代码。这样做确实能获取大量的疑似漏洞代码，但却不能保证这些疑似漏洞代码真实存在漏洞。很明显，在3.4节中，本文归纳了针对不同漏洞的众多规则，可是这些检测规则缺乏对上下文语境的考虑，也没法保证检测规则是完美无缺的，因此在实际操作中，尽管本文的规则已经针对现有规则做出了相当的改进，但是仍会有很高的误报率。如图3.4中代码所示。代码中的 `owner` 地址在构造器 `constructor` 中被赋值，而

```
1   contract buyOne {
2       constructor{
3           owner = msg.sender
4       }
5       ...
6       function buy(address target, uint amount) public {
7           require(msg.sender == owner);
8           target.call.value(amount);
9           ...
10      }
11  }
```

图 3.4　CPT 实例，开发者借助图中第 7 行的身份验证，有效防止可重入攻击

构造器只有在合约软件初始化时才能被调用，故 `owner` 变量一定是合约创建者的地址。在函数中 `buy` 中，第 7 行检查了合约访问者 `msg.sender` 与 `owner` 地址是否相等。这不起眼的一行，有效地阻止了可重入攻击。

在阅读 Solidity 源代码，分析程序执行情况的过程中，本文发现有大量优秀的开发者，在开发智能合约软件的时候遵守了软件的开发规范，使得智能合约软件具备了防范漏洞攻击的能力。这些防范技术各有差异，虽然可以把它们大致分为几类，但它们个体之间的实现差异较大，这不利于克隆技术的实现，会带来很高的误报率。这是由于克隆分析技术的特性决定的。事实上，在其他软件平台，漏洞代码和安全代码之间的差异可能会变得很小，过于相似的代码导致克隆分析技术无法直接对这些代码做出准确判断。因此，在本检测系统中，将加入 CPT 技术，通过这项技术减少现有克隆分析技术可能带来的高误报数，提高系统检测漏洞的准确率。在接下来的几小节中，本文将针对使用的 CPT 技术进行阐述。其中，对于可重入漏洞，本文总结了 CPT1-CPT5，对于意外异常漏洞，本文总结了 CPT6-CPT7，对于低级调用漏洞，本文总结了 CPT8，对于自毁漏洞，本文总结了 CPT9。

3.5.1　CPT1：在函数体中加入身份检查

CPT1 通过简单的方法就实现了限制函数访问的功能。很多开发者在编写函数的功能的时候，会在实际的功能代码前面加入相关的身份检查，检查合约的访问者是否是合约的持有者或者是否合法。实现身份检查有多种形式，本文举例一种形式如图3.5所示。在图3.5中，代码的第 3-6 行实现了四种检查功能，其中在

```
1    function finalize() public initialized {
2        require(finalizedBlock == 0);
3        require(finalizedTime == 0);
4        require(getBlockTimestamp() >= startTime);
5        require(msg.sender == controller || getBlockTimestamp() > endTime);
6
7        uint256 tokenCap = aix.totalSupply().mul(100).div(51);
8        aix.generateTokens(devHolder, tokenCap.mul(20).div(100));
9        aix.generateTokens(communityHolder, tokenCap.mul(29).div(100));
10       ...
11   }
```

图 3.5　CPT1：在函数体中加入身份检查

第 6 行通过检查 `msg.sender == controller` 来限制对该函数访问。`controller` 是一个全局变量，在合约的构造函数中被赋值，这意味着他只能由创建者进行赋值，无法被其他人更改。在这里这个函数检查了访问这身份是否是 `controller`，这也就意味着这段代码只能被合约的持有者执行。在很多情况下，执行转账操作、自毁操作时，合约的创建者希望函数功能由创建者本人或者创建者信任的访问者执行，他们通过添加身份检查实现了这个功能。

3.5.2　CPT2：限制转账的目标地址

开发者也可以通过限制转账的目标地址来进行可重入攻击的防御。目标的转账地址通常都是由一个全局变量进行维护或者通过函数的传参进行更新，如果不注意转账地址的限制，攻击者就可以采用多种方式修改转账的目标地址，改为自己的账户地址进而通过 `fallback` 函数进行重入攻击。在 Solidity 开发过程中，对与转账地址的管理，一定要慎之又慎。通常，开发者可以通过限制转账地址，即采用硬编码的方式制定转账的目标地址，或者限制转账地址的修改，即只允许部分访问者有权利更改转账的目标地址这两种方式去防御可重入攻击。如图3.6中代码所示。在图中，函数在第 11 行向 `_data` 进行转账操作，而这个全局变量在函数构造器 `constructor` 里面进行初始化，初始化时，这个初始化函数使用了一个 20 位的 16 进制的地址，即合约账户在以太坊平台上的绝对地址进行了赋值，故这个转账的目标地址在一开始是被限制死的。随后，虽然函数中准备了更改目标转账地址的函数 `changeData` 可以进行 `_data` 数据的修改，但是在这个修改函数的一开始就使用了 `require` 函数进行了身份验证，限制了修改函数的操作必须由 `creator` 执行，而和 `_data` 一样，这个 `creator` 的地址也是在构造器中进行的初始化，外来的访问者是无法随意突破这道屏障的。总的来说，这

```
contract Etherama{
    constructor() public {
        require(dataContractAddress != address(0x0));
        _data = EtheramaData(0x00a2409f41fdf485afd23599219c60a77524bba2);
        ...
        creator = msg.sender;
    }
    function changeData(address newAddress) public {
        require(msg.sender == creator);
        _data = EthermaData(newAddress);
    }
    function migrateToNewNewControllerContract() public {
        uint256 remainingTokenAmount = getRemainingTokenAmount();
        ...
        _data.transfer(remainingTokenAmount);
        ...
    }
}
```

图 3.6　CPT2：限制转账的目标地址

个合约的开发者虽然提供了转账操作、修改地址等功能，但是对涉及转账目标地址的操作，每一步都设置了防范措施，在这些防范措施的保护下，合约的转账地址很难被攻击者肆意修改。

3.5.3　CPT3；使用函数修饰器限制访问

CPT3 的原理和 CPT1 很像，相比 CPT1 在函数体中直接进行身份检查，CPT3 把这个实现放到里函数修饰器里。函数修饰器是 Solidity 语言的特点之一。Solidity 语言提供 `internal`、`private`、`public` 等内建修饰符限制对函数的访问，除此之外，Solidity 还允许开发者使用 `modifier` 模块自行设计函数修饰器。自定义函数修饰器中通常都有下划线 `_;`，表明修饰目标函数的原函数体。将自定义修饰符添加至函数原型上，执行时通常将优先执行修饰器内部的功能，然后再执行原函数体。因为自定义修饰符的这个特性，开发者们将身份验证的模块写入函数修饰器内部，保证在执行时，先通过修饰器内部的身份检查函数检查访问者的身份，再执行函数的具体功能。如图3.7中的代码所示。图中的函数第 2-4

```
contract ShortOrder is SafeMath {
    modifier onlyAdmin() {
        require(msg.sender == admin);
        _;
    }
    ...
    function claims(address[2] tokenUser, address usr, uint amount) external onlyAdmin
        {
        bytes32 orderHash = keccak256(tokenUser[0],tokenUser[1]);
        ...
        usr.transfer(safeAdd(orderRecord[orderHash],amount));
        Token(tokenUser[0]).transfer(admin,orderRecord[orderHash]);
        orderRecord[orderHash].balance = uint(0);
        ...
    }
}
```

图 3.7　CPT3：使用函数修饰器限制访问

行实现了修饰器 `onlyAdmin`，在修饰器的开始使用了 `require` 语句检查访问者的身份，限制函数仅允许合约的管理员访问。这是个简单的函数修饰器实例。随后

这个修饰器被用于 `claimDonations` 上。这个函数在第 10 行向由函数参数传入的 `usr` 账户进行转账，本来直接使用函数传参用于转账操作是一种容易引发漏洞的高发行为，但是通过使用 `onlyAdmin` 修饰符，每次在执行这个函数之前都要检查函数的访问者是否为合约的管理者，有效地阻止了函数被攻击者恶意利用。

3.5.4　CPT4：使用程序的执行锁防止重入

使用程序锁防止重入是一种比较巧妙的技术，通常开发者会设置一个互斥变量用作执行锁，在每次执行关键操作之前先检查执行锁的值，并修改执行锁的现有值，再进行正常的业务操作。这种防范方式会在攻击者第二次试图进入函数时挡住攻击者，因为执行锁的值已经在第一次运行时改变了，在第二次运行时无法通过执行锁值的检查，进而达到防止程序被重入的效果。在图3.8中，函数 10 行进行执行锁的检查，在第 11 行更改了锁的值，在更改执行锁的值之后再在第 12 行进行转账操作。如果有攻击者试图使用重入进行函数攻击，那么在第二次重入时，攻击者将无法通过程序执行锁的检查，也无法进行可重入攻击。因此，这个 CPT 技术通过简单的代码即实现了对可重入攻击的防范。

```
1   contract MicroDAO{
2       bool public executed = false;
3       ...
4       function execute() internal {
5           ...
6           executed=false;
7       }
8       function executeSpendingRequests(address _addr, uint amount) public payable {
9           MicroDAO m =MicroDAO(allowances[_addr]);
10          if(!executed) {
11              m.execute();
12              _addr.transfer(amount);
13              ...
14          }
15      }
16  }
```

图 3.8　CPT4：使用程序的执行锁防止重入

3.5.5　CPT5：提前更新状态变量

这种防御手段主要是在转账之前更新状态变量（即转账金额），这样在进行重入时，状态变量已经发生了改变，攻击者无法通过状态变量的检查，进而达到防御可重入攻击的目的。如图3.9所示，图中代码在第 6 行进行转账操作，但在转账操作之前，第 5 行进行余额的检查，而余额时在第 3 行进行赋值的，被赋值为 `balanceOf` 数组中访问者的余额，在赋值完成之后，随即把数组中访问者的余额赋值为 0，并马上进行转账操作。在经过这样的预先处理之后，攻击者在第二次重入时，数组中访问者的余额已经被赋值为 0 了，故攻击者无法通过第 5 行的余额检查。这个函数通过对余额的检查以及在转账之前进行状态变量的更新成功达到了防范可重入漏洞攻击的目的。

```
1   function safeWithdrawlal() public companyCanBeFinished{
2       if (!fundingGoalReached){
3           uint amount = balanceOf[msg.sender];
4           balanceOf[msg.sender] = 0;
5           if (amount > 0){
6               msg.sender.send(amount);
7               FundTransfer(msg.sender, amount, false)
8           }
9           ...
10      }
11  }
```

图 3.9　CPT5：提前更新状态变量

3.5.6　CPT6：不在循环语句中使用检查函数

根据本文观察，在循环语句中对 `send` 使用 `require` 会容易导致意外异常漏洞。这是因为在 Solidity 语言中，`send` 语句转账失败时会返回 `false` 而不抛出异常，而 `transfer` 在语句转账之后会检查返回结果，如果返回结果为 `false` 则自动抛出异常。如果本文使用 `send` 配合 `require` 做检验，那么会导致在转账失败时，合约失去服务能力，进而发生拒绝服务的行为。因此，Solidity 开发者们为了阻止由 `send` 语句引发的漏洞攻击，不会使用 `require` 检查 `send` 转账的结果。

3.5.7　CPT7：在循环语句中向单个地址进行转账

在智能合约软件代码中，有些业务逻辑使用循环语句向单个账户地址发送一定金额的以太币。这种情况下，如果某次转账失败，导致整个循环转账的业务逻辑中断，本文也不会判断这个代码是包含漏洞的代码。因为本文在考虑漏洞代码的特征时，不仅要考虑代码本身的攻防逻辑，也要考虑在实际情况中的应用场景。不符合实际应用场景的攻击逻辑，即便可以轻松达到攻击代码的目的，本文也不认为这是一个漏洞代码。在此处这个向单个用户发送以太币的情况下，如果在循环中有单词转账失败导致整个业务的暂停，业务的暂停也不会影响到别的账户，不符合拒绝服务攻击的应用场景，因此本文把这归纳为 CPT 技术的一种，认为不对他人造成任何侵害的代码不应该属于漏洞代码。

在图3.10中，第 15 和 16 行都有 `transfer` 转账函数，这两个转账函数都在 `for` 循环的内部，但是这两个转账的地址 `_player` 和 `platform` 都是固定的转账地址。其中，`_player` 为函数的参数传入，而 `platform` 是一个全局变量，这两个转账地址都不会随着循环结构的进行而改变，进而不会引发意外异常漏洞。

3.5.8　CPT8：使用多种检查方式检查低级调用

Solidity 语言诞生之初带来的诸多低级调用函数被开发者们诟病久矣。这些低级调用接近底层，有诸多使用限制，对 Solidity 语言不够熟练的开发者很容易在使用低级调用时埋下安全隐患。Solidity 在后面的迭代过程中，推出了很多高

```
function assign2(address _player, uint256 _roundID) public isHuman() isInitialized()
{
    require(rounds[_roundID].state == DataSet.RoundState.DRAWN, "it's not time for
        assigning");

    uint256[] memory numbers = playerNumbers[_roundID][_player];
    require(numbers.length > 0, "player did not involve in");
    uint256 targetNumber = rounds[_roundID].winningNumber;
    for (uint256 i = 0; i < numbers.length; i ++)
    {
        (uint256 start, uint256 end) = NumberCompressor.decode(numbers[i]);
        if (targetNumber >= start && targetNumber <= end)
        {
            // assgin bonus to player, and the rest of the pond to platform
            uint256 fund = rounds[_roundID].pond.sub(bonus);
            _player.transfer(bonus);
            platform.transfer(fund);
            rounds[_roundID].state = DataSet.RoundState.ASSIGNED;
            rounds[_roundID].winner = _player;

            emit onAssign(msg.sender, block.timestamp, _player, _roundID, rounds[
                _roundID].pond, bonus, fund);

            break;
        }
    }
}
```

图 3.10　CPT7：向单个地址转账

级的外部调用来替代低级调用的功能，这些高级调用对低级调用中极少使用但容易埋下漏洞隐患的部分做了封装，让用户能更安全地使用这门语言进行开发。如果开发者一定有使用低级调用的必要，也需要配合 `assert`、`require`、`if` 等检查语句一并使用，保证低级调用的失败得到妥善处理，也保证智能合约软件的业务逻辑能正常进行。

3.5.9　CPT9：在使用自毁函数时增加严格的身份检查

自毁函数 `selfdestruct` 具有很强的攻击性，在使用不当的情况下很容易造成漏洞攻击。本文在观察大量代码之后，认为防范这种漏洞引发的攻击最好的方法便是增加严格的身份检查功能。通过函数修饰符或者直接在函数体内部实现身份检查都能防止自毁函数被任意的访问者调用，增加函数的安全性。在图3.11中，

```
function killContract() returns (uint error){
    if (msg.sender != dev) { return 1; }

    selfdestruct(dev);
    return 0;
}
```

图 3.11　CPT9：使用自毁函数时增加严格的身份检查

`killContract` 这个函数实现了自毁功能，函数体中的主要成分也是自毁函数。但是在自毁之前，这个函数验证了 `msg.sender` 是否为 `dev`，即验证了这个合约的访问者是否为这个合约的开发者本人。

算法 3.1　基于代码克隆分析的漏洞检测算法

输入:

S, 针对某漏洞的函数签名;

C, 被检测的目标代码;

η, 代码相似度上界;

γ, 代码相似度下界;

itv, 检测窗口变化率;

输出: V, 漏洞函数集;

```
w ← LineofCode(C)
c ← CodeofWindow(w, C)
F ← FunctionList(c)
V ← ∅
for each function f ∈ F do
    if Similarity(S, c) ⩽ γ then
        w ← w × itv
        c ← CodeofWindow(w, C)
    else if Similarity(S, c) ⩾ η then
        w ← w × 1/itv
        c ← CodeofWindow(w, C)
    end
    else if Similarity(S, c) ∈ (γ, η) then
        V ← V ∨ f
    end
end
for each vulnerable candidates v ∈ V do
    for each rule r ∈ CPTs do
        b ← Match(r, v)
        if b == True then
            V ← V − v
    end
end
```

3.6　漏洞检测算法设计

在获取了漏洞的漏洞签名和用于减低误报率的 CPT 技术之后，本文提出漏洞检测算法如算法3.1所示。本文设计的算法总共有五个输入，其中 S 为针对某函数的漏洞签名，使用漏洞签名来匹配疑似漏洞的代码；C 为被检测的代码，即待检测代码；η 和 γ 分别为代码相似度的上界和下界，这两个值是人为预先设定，并根据系统的表现不断调整的，它们决定了寻找克隆代码窗口的调整大小；itv 为每次调整窗口的变化率，变化率越大则窗口大小的改变越激进。输出有一个，即漏洞检测算法找到的漏洞函数集合。

在算法的前面四行，本算法将 w 的初始值设置为整段代码的大小，c 设置为在这段代码上的窗口扫描到的代码，F 为窗口扫描到的代码中的所包含的函数列表，V 为初始为空集。在函数的第一个循环 5-15 行，遍历了函数列表中的每个函数，并对每一个函数和漏洞签名使用了相似度算法——最长公共子串（Longest Common Subsequence）算法计算相似度，如果相似度过高，说明可能窗口值过大，需要按照一定的变化率调整窗口的大小并重新扫描；如果相似度过低，则说明窗口值过小，需要放大窗口的大小并重新扫描；如果相似度在相似度上界和下界之间，则说明窗口值刚好，有很大可能匹配到了漏洞代码，这时把匹配成功的函数加入漏洞函数集。

在算法的第二个循环 16-22 行，本算法对漏洞集 V 中的每个漏洞候选进行遍历，使用在上一节3.5中介绍的 CPT 去一一匹配，如果和 CPT 匹配的相似度高，则说明这个代码对相应的漏洞做出了一定程度的防范，不在本文的考虑范围之列，并在第 20 行将代码从漏洞集中去除。

这个算法最大时间开销来自于算法的第 5-15 行，首先使用循环语句遍历每个函数，并使用 LCS 算法进行相似度计算，时间复杂度为 $\mathcal{O}(n^3)$。而在算法的第二个循环即 17-23 行，本算法虽然使用了两层遍历，但是在计算是否匹配时使用基于规则的匹配技术，综合时间复杂度为 $\mathcal{O}(n^2)$。本文设计的算法并不是最快的，但是得益于智能合约体积软件体积较小，尚没有举行的智能合约软件出现，即便本文使用了如此高时间复杂度的算法，在 Solidiity 软件上检测漏洞的时间开销也不会太高。

第 4 章 实 验 验 证

4.1 实验配置及关键问题

在整个实验验证的过程中，实验使用的机器使用了 Ubuntu 18.04 bionic 系统，配置了 8×2 核 2.10GHz 的英特尔 Xeon E5-2620V4 处理器，32 GB 内存和 4 TB 的机械硬盘。对于在实验过程中使用的分析工具，本文全部使用默认配置，也没有使用多线程技术。

在实验过程中，本文使用几种分析工具的最新版本进行漏洞签名的归纳总结，并在 26354 个合约的学习数据集上完成了对 CPT 技术的总结。为了获取学习数据集，本文实现了一个网络爬虫，这个爬虫可以从 Etherscan，一个著名的第三方区块分析服务提供站上面爬取 Solidity 源代码。在实验的初期，Etherscan 没有限制用户自由下载 Solidity 代码，但是到 2019 年年初，Etherscan 限制每个用户只能查看最新的 1000 个经过验证的合约代码，而不能查看以往的合约代码。最终，本文借助最初爬取的 26354 个智能合约进行标准漏洞库的构建。在爬虫过程中，爬虫使用了随机的搜索策略以保证从 Etherscan 下载的 Solidity 源代码是随机的。

对于用于验证工具的数据集，鉴于本文从 26354 个合约上观察总结了 CPT 技术，如果再把这些数据集用于工具的验证，会显得不够公平。因此，本文从 Google BigQuery Open Dataset 上爬取了一系列的合约地址，再经过从 Etherscan 上获取合约源代码，最终得到了 11516 个部署在以太坊之上的真实 Solidity 代码。在这个数据集上本文能够公正地进行检测系统的验证工作，并和其他工具做比较。至于在验证环节用到的各个分析工具，本文皆采用当前能获取到的最新版本：Slither v0.6.4，Oyente v0.2.7，Smartcheck v2.0 和 Securify v1.0。

在本章，本文希望围绕三个关键问题（Research Questions）来阐述和分析实验：

RQ1 构建的标准漏洞库质量如何？这些提取的签名真的具有代表性吗？发现的漏洞有效吗？

RQ2 构建的检测系统准确率如何？对比其他检测工具是怎样的结果？

RQ3 本系统在检测漏洞的过程中和对比其他工具的过程中的效率如何？

4.2 RQ1：验证漏洞签名以及构建的标准漏洞库

在归纳总结漏洞签名以及 CPT 技术应用于寻找漏洞的过程中，本文通过找到的漏洞不断反馈修缮漏洞签名，使得最终获得的漏洞签名也成为成果的一部

分，进而能借助漏洞签名和 CPT 技术找到更多的漏洞。最终，本文分析了所有工具报告的真实漏洞样本，并在此基础上构建了标准漏洞库。

4.2.1　分析收集具有代表性的漏洞签名

基于各个工具的源代码、论文以及他们所报告的真实漏洞，本文分析总结出了漏洞签名。因为本文主要通过开源的仓库，已经发表的论文等获取分析工具的特点，在一些漏洞签名上无法做到完全理解，由此造成了获取的漏洞签名没有完整体现工具的特点，对于这个缺陷，本文通过观察各工具对漏洞的报告情况不断调整签名。同时，为了保证提取的漏洞签名具有一定程度的代表性，本文将部分包含漏洞的代码使用树的编辑距离算法[39] 计算，并进行聚类，最后再将聚类后得到的代码的共同特征进行抽象，再和漏洞签名进行比对，这样使得本文总结的漏洞签名不仅代表了前沿分析工具的技术结晶，也代表了真实漏洞的代码的核心成分。

4.2.2　不同工具发现的漏洞分布

为了辨认现有工具报告的漏洞，一些人为的审计是无法避免的。但本文采取了一些策略来加速这个人工审计的过程，同时也保证了审计具有一定的准确度。例如，如果一段代码被 Slither、Oyente、Smartcheck 和 Securify 中两个或两个以上的工具报告为漏洞，本文觉得这段代码具有较高的可信度，因此指派一名领域专家对这段代码进行审计；否则如果一段代码只被一个工具报告为漏洞，本文会指定两名领域专家对这段代码进行审计，如果这两名领域专家的意见相反，本文会指定第三名领域专家介入并做最终判断。在获取了以上工具报告的所有真实漏洞后，才能构建一个对漏洞准确率具有较高置信度的标准漏洞库。如图所

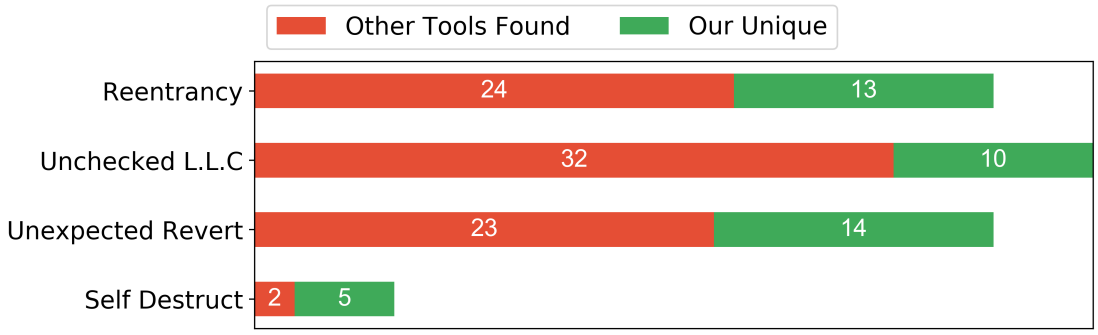

图 4.1　发现的真实漏洞数量统计，“Our Unique”表示只被 Athena 发现的漏洞

示，这个标准漏洞库包含了 123 个漏洞实例。在这之中，可重入漏洞和低级调用漏洞占了所有漏洞数量可观的比重，其他两个漏洞则较少。在实验完成之后并联系上漏洞关联软件的开发者后，本文会将部分漏洞在网站上进行公开。

4.2.3　动态验证漏洞的真实性

为了验证本文在实验中检测到的漏洞的真实性，本文对漏洞实例进行了简单的随机采样，并试图在本文的本地环境进行漏洞的触发。特别地，针对每个漏洞签名采样 2 个漏洞进行测试，只选用两个漏洞是因为使用漏洞签名匹配的可疑代码大多相似，如果测试代码被成功出发，则这些相似的代码也能用类似的测试方式进行触发；还有个原因是每次测试所需要的测试用例都需要本文的领域专家进行独立设计，如果需要进行测试的实例太多，会有大量的时间消耗。测试的环境采用了 Remix 本地环境进行测试。至于如何使用自动化流程进行软件代码测试，或者如何保证测试的高效性和有效性，这些问题超出了本课题研究的范围，不作讨论。

4.3　RQ2：分析各检测系统的检测结果

在上一章第3.4节和第3.5节中，本文讨论了漏洞签名的归纳总结过程和 9 中 CPT 技术的特点。为了分析这些漏洞签名和 CPT 的有效性，本文将 11516 个未使用的智能合约软件用作测试集，在测试集上使用各种前沿工具以及本文的检测系统进行实验，最后比对检测结果。在统计时，本文将每个工具报告的漏洞都经过了人工专家的审计，这使得本文认为的漏洞代码数据是具备一定可信度的。各个工具在各个漏洞上的检测准确率如表4.1、4.2、4.3、4.4所示。

表 4.1　可重入漏洞检测结果

Tools	#N	P%	R%
Slither	79	18.98%	40.54%
Oyente	19	10.53%	5.40%
Smartcheck	×	×	×
Securify	261	4.59%	32.43%
Athena	58	27.59%	43.24%

表 4.2　意外异常漏洞检测结果

Tools	#N	P%	R%
Slither	137	10.95%	51.35%
Oyente	×	×	×
Smartcheck	29	65.52%	40.54%
Securify	×	×	×
Athena	43	79.07%	91.89%

表 4.3　低级调用漏洞检测结果

Tools	#N	P%	R%
Slither	18	100.0%	42.86%
Oyente	×	×	×
Smartcheck	91	45.05%	52.38%
Securify	×	×	×
Athena	38	89.47%	80.95%

表 4.4　自毁漏洞检测结果

Tools	#N	P%	R%
Slither	10	20.00%	28.57%
Oyente	×	×	×
Smartcheck	×	×	×
Securify	×	×	×
Athena	30	23.33%	100.0%

表中的"#N" 表示工具报告漏洞的数量，"P%" 和"R%" 分别表示了精确率和覆盖率，其中精确率的计算方式为 (该工具报告的 TP 数)/#N，覆盖率的计算方式为 (该工具报告的 TP 数)/(所有工具的 TP 集合) 。下面本文将对表格中数据做简单的总结，并对数据背后的原因做简单分析。

4.3.1 Athena 产生误报的原因

在表4.1、4.2、4.3、4.4中，本文列举了 91 个由 Athena 找到的漏洞，不考虑漏洞的类型，Athena 有着 53.84% 的总体精确率。相比之下，Slither 具有着 20.49% 的总体精确率，Oyente 有着 10.53% 的总体精确率，Smartcheck 具有着 40.59% 的总体精确率，Securify 有着 4.59% 的总体精确率。以上的精确率是建立在第3.4节中引入的漏洞签名和在第3.5中引入的 CPT 技术的基础上进行统计和整理的。

1. 可重入漏洞误报的原因

在四个支持检测可重入漏洞的工具中，Athena 有着最低的误报率（准确率 27.53%），这是因为本文采用了 SS1-5 的 CPT 技术，针对可重入漏洞的检测做出改进，有效降低了在可重入漏洞上的误报率。其他工具的误报率都要高于甚至显著高于本系统。例如，Securify 的误报率高达 95.41%，这是因为它的检测规则过于宽泛和简单，而且没有考虑代码中的 CPT 技术。Slither 的检测规则比较严谨和规范，但是它同样不考虑 CPTF 技术，这导致它虽然有着可接受的覆盖率 40.54%，准确率却比较低 18.98%。Oyente 同样没考虑 CPT 技术的存在，由于它的规则过于严格，它的覆盖率 5.40% 较低，但准确率 10.53% 较高。

2. 意外异常漏洞误报的原因

正如在第3.4节归纳的一样，Slither 将在循环结构中的所有调用都判作漏洞，这导致了它 122 个误报。Smartcheck 在检测时考虑了 SS6 但是没有考虑 SS7。相比之下，Athena 考虑了 SS6 与 SS7，这使得本系统产生的误报数量更少。

3. 低级调用漏洞误报的原因

Slither 在这个漏洞上的精确度表现很好，没有一个误报，这是因为它严格地检查了三种低级调用，`call`、`delegatecall`、`send` 的使用。Smartcheck 没有考虑 SS8，因此报告了很多高级调用，导致了 50 个误报。Athena 有 4 个误报，这几个误报的原因是因为在函数中这些低级调用的返回值在其他语句受到了检查，而不是直接低级调用写在一起。因此，这 4 个误报需要借助数据依赖分析去寻找它被检查的位置。

4. 自毁漏洞误报的原因

在这个漏洞上 Athena 有 24 个误报，在观察误报的原因之后，本文发现这些误报的原因是出自于复杂的函数修饰器，这些函数修饰器将身份检查功能隐

藏于调用链中，无法直接准确分析。Slither 有着接近的精确率，并报告了 8 个 FP。

5. 小结

本文总结的 CPT 技术能在多数类型的漏洞中减少误报的数量。可是，CPT3、CPT9 这两种 CPT 的总结还不够令人满意，在遇到复杂的用户自定义函数修饰器或者是许多本地变量的使用时的表现不尽人意。要解决以上问题，必须进一步地采用数据依赖分析技术进行辅助。

4.3.2　Athena 具有高覆盖率的原因

在表4.1、4.2、4.3、4.4中，Athena 具有着最好的覆盖率表现，在四种漏洞上的表现都要好于或者远好于其他工具。

1. 可重入漏洞真实漏洞的分布

Athena 找到了 13 个其他工具都没有发现的漏洞，本文观察这 13 个漏洞之后认为，其他的三个工具都没能考虑用户自定义的 `transfer` 函数，只考虑了 Solidity 内建的 `transfer` 函数。另外，Athena 漏掉了 34 个真实漏洞，很显然这是因为可重入漏洞的复杂超过预计，本文的漏洞签名仍然没能充分包括这部分真实漏洞。

2. 意外异常真实漏洞的分布

对于这个漏洞类型，共有 5 个真实漏洞是只被一种工具唯一发现的，其中有 2 个是被 Athena 发现的。有趣的是，大部分的该类型的真实漏洞是被至少两种工具发现的——这说明其他工具的检测规则和本文的漏洞签名相似。其他 3 个只被一种工具唯一发现的漏洞没有在 Athena 的报告范围内，这是因为这些代码使用 `assert` 检查循环结构中的 `transfer` 语句，这也会造成意外异常漏洞，且不在漏洞签名的考虑范围内。

3. 低级调用真实漏洞的分布

在 18 个只被一种工具唯一发现的真实漏洞中，Athena 发现了 10 个，这 10 个里低级调用被 `if` 的条件语句使用，但是没有在 `if` 语句中进行检查，因此不会被报告为真实漏洞。Athena 漏洞签名去尽可能匹配疑似漏洞的数据集。可是，仍有 8 个漏洞被 Athena 漏掉了。在观察这 8 个漏洞之后，本文发现这 8 个漏洞的低级调用的返回值赋值给了别的变量，对于变量的检查在后面的语句中产生，这是种非直接检查的低级调用。由于本文并没有在系统中加入数据依赖分析，Athena 没能将这类代码报告为漏洞。Slither、Smartcheck 将这 8 个报告为漏洞的原因是因为这些低级调用存在于 `if` 结构的条件语句内。

4. 自毁漏洞真实漏洞的分布

对于这类漏洞，Athena 报告了 5 个唯一发现的真实漏洞，并有着出众的覆盖率表现。本文自毁漏洞的漏洞签名较鲁棒，也考虑了用户自定义的函数修饰器，在这些综合影响之下，系统的检测覆盖率远好于 Slither 的 28.57%。同时，高覆盖率也带来了较低的准确率，Athena 的准确率稍高于 Slither。

5. 小结

基于观察各前沿工具的实现原理和阅读大量代码得到的四种漏洞签名，只被 Athena 检测出的漏洞数要远高于其他工具检测出的数量。在检测低级调用漏洞时，Athena 仍需要再加入准确的数据依赖分析，以提高系统的覆盖率表现。

4.4　RQ3：验证系统的运行效率

在表4.5中，Slither 仅消耗了 52 分钟就完成了检测工作。而 Smartcheck 和 Athena 这两个工具采用了相近的核心技术——基于规则的静态分析，因此本应具有相近的运行时间 (0$\tilde{5}$00 min)。实际上，由于实现的差异，两种工具的运行时间有一定的差距，但这两个工具的检测速度仍然原快于 Oyente 和 Securify，因为他们使用了约束求解技术，增加了时间的消耗。值得一提的是，本文提取漏洞签名的时间，包括 CPT 技术的加入，没有被包含在整个系统的运行时间内。

表 4.5　各工具的检测时间，单位为分 (min)

Contracts	Slither	Oyente	Smartcheck	Securify	Athena
11516	52	1081	112	6201	236

从检测速度上来说，Slither 是效率最高的工具，Oyente 是除 Securify 之外效率较高的工具，Athena 和 Smartcheck 具有较高的效率。如果考虑系统的准确率、覆盖率等综合因素，Athena 是所有工具中最突出的。

第 5 章 总 结

本文总共五个章节，在第一章中，本文介绍了以太坊，智能合约以及工作选题的先进性、创新性，和工作主要的研究内容及拟解决的关键问题；在第二章中，本文介绍了智能合约的相关背景知识，包括智能合约的常见漏洞和前沿的智能合约软件分析工具，除此之外，还简单介绍了 Solidity 字节码的相关知识；在第三章中，本文引入了主要的研究方法及技术路线，首先通过调研现有工具的检测能力和之前软件克隆分析的相关文献，确定了克隆分析技术在智能合约软件漏洞检测上的可行性，接下来，提出了基于漏洞签名的漏洞检测技术和提高检测准确率的 CPT 技术；在第四章中，本文配置了相关实验，并针对三个研究问题，系统是否准确，系统覆盖率如何，系统的运行效率如何做出了回答。

在第一章中，本文列举了国外智能合约研究、软件分析研究方面的进展，进而提出本文选题的先进性和创新性。接下来本文描述了本工作的主要研究内容，内容涉及漏洞、前沿工具、字节码以及克隆分析技术。最后，本文针对这几项研究内容提出了这篇工作要解决的几个关键问题。

在第二章中，本文主要介绍了本课题需要了解的相关背景知识。首先，针对智能合约中常见的四种漏洞做了简单介绍并举例；其次，对现有的分析工具进行了分类并介绍特点；最后，还介绍了智能合约字节码的相关知识。

在第三章中，本文首先介绍了现有工具的漏洞检测能力，并调研了之前的克隆分析技术文献作为本文的方法论指导。然后，本文观察了智能合约代码的特点，分析和讨论了在智能合约软件上使用克隆分析技术的可行性。在这之后，本文提出了基于规则的漏洞检测技术，并观察现有工具运行原理并加入自己的理解，成功提取了各类型的漏洞签名。在经过大量阅读智能合约代码之后，本文也提出使用 CPT 技术来改善检测系统的准确率。针对各种不同的漏洞，本文共提出了九种 CPT 技术。最后，本文提出了漏洞检测系统 Athena 的实现算法。

在第四章中，本文进行了实验。首先对实验的配置进行讲解，并提出实验要解决的三个研究问题。接下来，本文根据实验结果对这三个问题进行了一一解答，这三个问题包括 Athena 系统的检测准确率如何，系统的检测出的真实漏洞覆盖率如何以及系统的运行效率如何。

最后，本文在这里对以上章节做出了总结。本文提出的 Athena 系统在准确率和覆盖率上对比现有前沿工具都有着不小的优势，但是本系统仍有不小的改进空间，如漏洞签名的改进、加入数据依赖分析等等。本文的系统在将来会越来越可靠，也希望能带给智能合约软件的开发者更多的便利。

参 考 文 献

[1] GUO Y, LIANG C. Blockchain application and outlook in the banking industry[J]. Financial Innovation, 2016, 2(1):24.

[2] HUKKINEN T, MATTILA J, ILOMÄKI J, et al. A blockchain application in energy [J]. ETLA Reports, 2017, 71.

[3] 谢辉, 王健. 区块链技术及其应用研究[J]. 信息网络安全, 2016(9):192-195.

[4] RAY J. Ethereum introduction[EB/OL]. https://github.com/ethereum/wiki/wiki/Ethereum-introduction.

[5] ETHERSCAN. Ethereum transaction chart[EB/OL]. https://etherscan.io/chart/tx.

[6] SIEGEL D. Understanding the dao attack[EB/OL]. https://www.coindesk.com/understanding-dao-hack-journalists.

[7] JIANG L, MISHERGHI G, SU Z, et al. Deckard: Scalable and accurate tree-based detection of code clones[C]//Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007: 96-105.

[8] ATZEI N, BARTOLETTI M, CIMOLI T. A survey of attacks on ethereum smart contracts (sok)[C]//International Conference on Principles of Security and Trust. Springer, 2017: 164-186.

[9] BARTOLETTI M, POMPIANU L. An empirical analysis of smart contracts: platforms, applications, and design patterns[C]//International conference on financial cryptography and data security. Springer, 2017: 494-509.

[10] EGELE M, SCHOLTE T, KIRDA E, et al. A survey on automated dynamic malware-analysis techniques and tools[J]. ACM computing surveys (CSUR), 2012, 44(2):6.

[11] CRYTIC. Slither: a static analyzer on smart contracts[EB/OL]. https://github.com/crytic/slither.

[12] 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. ACM, 2016: 254-269.

[13] 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. ACM, 2018: 67-82.

[14] CRYTIC. Manticore: a dynamic analyzer on smart contracts[EB/OL]. https://github.com/trailofbits/manticore.

[15] DE MOURA L, BJØRNER N. Z3: An efficient smt solver[C]//International conference

on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008: 337-340.

[16] SOUFFLE-LANG. Souffle: A translator of horn clauses into parallel c++[EB/OL]. https://souffle-lang.github.io/index.html.

[17] KALRA S, GOEL S, DHAWAN M, et al. Zeus: Analyzing safety of smart contracts. [C]//NDSS. 2018.

[18] CRYTIC. Echidna: a haskell based verification library for smart contract[EB/OL]. https://github.com/crytic/echidna.

[19] ROY C K, CORDY J R. A survey on software clone detection research[J]. Queen' s School of Computing TR, 2007, 541(115):64-68.

[20] CHANDRAMOHAN M, XUE Y, XU Z, et al. Bingo: Cross-architecture cross-os binary search[C]//Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2016: 678-689.

[21] KIM S, WOO S, LEE H, et al. Vuddy: A scalable approach for vulnerable code clone discovery[C]//2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017: 595-614.

[22] ETHEREUM. Solidity: A turing-complete language for smart contracts[EB/OL]. https://solidity.readthedocs.io/en/v0.4.25/.

[23] JIANG B, LIU Y, CHAN W. Contractfuzzer: Fuzzing smart contracts for vulnerability detection[C]//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, 2018: 259-269.

[24] CONSENSYS. Mythril: A security analysis tool for evm bytecode[EB/OL]. https://github.com/ConsenSys/mythril.

[25] CONSENSYS. Mythx: A smart contract security tool for ethereum[EB/OL]. https://mythx.io/.

[26] TIKHOMIROV S, VOSKRESENSKAYA E, IVANITSKIY I, et al. Smartcheck: Static analysis of ethereum smart contracts[C]//2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). IEEE, 2018: 9-16.

[27] QUOSCIENT. Octopus: A security analysis tool for webassembly module and blockchain smart contracts[EB/OL]. https://github.com/quoscient/octopus.

[28] KAMIYA T, KUSUMOTO S, INOUE K. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code[J]. IEEE Transactions on Software Engineering, 2002, 28(7):654-670.

[29] CHAN P P, COLLBERG C. A method to evaluate cfg comparison algorithms[C]//

2014 14th International Conference on Quality Software. IEEE, 2014: 95-104.

[30] GYIMESI P, VANCSICS B, STOCCO A, et al. Bugsjs: A benchmark of javascript bugs[C]//2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). IEEE, 2019: 90-101.

[31] LI Z, LU S, MYAGMAR S, et al. Cp-miner: Finding copy-paste and related bugs in large-scale software code[J]. IEEE Transactions on software Engineering, 2006, 32(3): 176-192.

[32] YAN X, HAN J, AFSHAR R. Clospan: Mining: Closed sequential patterns in large datasets[C]//Proceedings of the 2003 SIAM international conference on data mining. SIAM, 2003: 166-177.

[33] JANG J, AGRAWAL A, BRUMLEY D. Redebug: finding unpatched code clones in entire os distributions[C]//2012 IEEE Symposium on Security and Privacy. IEEE, 2012: 48-62.

[34] SAJNANI H, SAINI V, SVAJLENKO J, et al. Sourcerercc: Scaling code clone detection to big-code[C]//2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016: 1157-1168.

[35] YAMAGUCHI F, LINDNER F, RIECK K. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning[C]//Proceedings of the 5th USENIX conference on Offensive technologies. USENIX Association, 2011: 13-13.

[36] SASAKI Y, YAMAMOTO T, HAYASE Y, et al. Finding file clones in freebsd ports collection[C]//2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, 2010: 102-105.

[37] LI Z, ZOU D, XU S, et al. Vulpecker: an automated vulnerability detection system based on code similarity analysis[C]//Proceedings of the 32nd Annual Conference on Computer Security Applications. ACM, 2016: 201-213.

[38] CHANG J, GAO B, XIAO H, et al. scompile: Critical path identification and analysis for smart contracts[J]. arXiv preprint arXiv:1808.00624, 2018.

[39] SCHWARZ S, PAWLIK M, AUGSTEN N. A new perspective on the tree edit distance [C]//International Conference on Similarity Search and Applications. Springer, 2017: 156-170.

致　谢

在研究学习期间，我得到了各位老师的大力帮助，感谢薛老师、马磊老师和在新加坡、澳洲的各位学界前辈，带着我做了很多富有创新性和挑战性的工作；也感谢同课题的马明亮、彭天勇同学，作为实验的伙伴，也是一起奋斗的战友，在研究学习期间相互提携，彼此都收获了很多；感谢其他同学，帮助我坚定了科研的决心；感谢女朋友周迪、我的父母，给了我稳定的后方支持，是我精神上的鼓励与支柱。

智能合约领域较新，我们在将传统软件分析方法用于这个新领域时遇到了不少挑战。仍记得无数个日夜的挑灯夜战，数万份代码挨个看遍，早已练就了洞察漏洞的火眼晶晶。针对在研究过程中遇到的各个困难，我们不断改进旧方法，使其更贴合智能合约的特点，旧法新用，保证了一定的科研产出和科研的高质量。

回忆起收到科大录取通知书时立下的凌云壮志，今天实现了吗？也许没有全部，但大部分来说，是的。今天脚下走的道路，并不是我两年之前想好的，时代瞬息万变，容不得人装睡。如今人生中另一个短暂的阶段已经接近尾声，在这之后，我将开启下一段学生生涯，祝自己诸事顺利。

在读期间发表的学术论文与取得的研究成果

已发表论文

1. Jiaming Ye, Mingliang Ma, Tianyong Peng et al., Towards Automated Generation of Bug Benchmark for Smart Contracts[C], ICSTW 2019, https://doi.org/10.1109/ICSTW.2019.00049
2. Jiaming Ye, Mingliang Ma, Tianyong Peng et al., A Software Analysis Based Vulnerability Detection System For Smart Contracts[J], Integrating Research and Practice in Software Engineering, 2019, https://doi.org/10.1007/978-3-030-26574-8_6