分类号 TP957 学号 13049023

UDC 密级 公开

工学博士学位论文

源代码软件漏洞自动挖掘关键技术研究

博士生姓名 孟庆坤

学科专业 信息与通信工程

研究方向 现代通信技术

指导教师 沈荣骏 研究员

唐朝京 教授

国防科技大学研究生院

二〇一七年十一月

Study on Key Technology for Vulnerability Detection of Souce Code Software

Candidate: Meng qingkun
Supervisor: Prof. Shen Rongjun
Prof. Tang Chaojing

A dissertation

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Engineering

in Information and Communication Engineering

Graduate School of National University of Defense Technology

Changsha, Hunan, P. R. China

January 10, 2018

独 创 性 声 明

本人声明所呈交的学位论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表和撰写过的研究成果，也不包含为获得国防科技大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目：源代码软件漏洞自动挖掘关键技术研究

学位论文作者签名：_____ 日期： 年 月 日

学位论文版权使用授权书

本人完全了解国防科技大学有关保留、使用学位论文的规定。本人授权国防科技大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入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。

（保密学位论文在解密后适用本授权书。）

学位论文题目：源代码软件漏洞自动挖掘关键技术研究

学位论文作者签名：_____ 日期： 年 月 日

作者指导教师签名：_____ 日期： 年 月 日

目　录

表 目 录

图 目 录

摘 要

随着信息技术的发展，软件已成为与世界经济、文化、科技、教育和军事发展息息相关的重要元素，软件作为信息系统中的核心基础设施之一，广泛应用于通信、金融、医疗等众多领域。无论是商业软件还是程序员自行开发的小程序，开源代码 / 组件的使用已经变得越来越普遍，开源已经成为了一种趋势。源代码软件漏洞的影响越来越大，基于开源软件漏洞的网络攻击活动数量在逐年增长。源代码软件漏洞挖掘技术能够针对性的对开源软件进行挖掘，掌握开源软件的漏洞挖掘技术对我国、我军的信息安全具有重大战略意义。

论文围绕源代码软件漏洞挖掘中的关键技术展开研究。通过梳理发现，现有的源代码漏洞挖掘方法还不够完善。在静态分析方面，现有方法存在着支持的漏洞类型少、挖掘精度低的问题。在动态测试方面，符号执行和模糊测试技术虽然都能挖掘漏洞，但是符号执行存在路径爆炸问题，模糊测试存在覆盖率低、不具备导向性等问题。据此，结合源代码的直接或者间接信息、形式化方法，论文在漏洞静态分析、符号执行以及模糊测试方面展开了研究，主要工作和创新如下:

论文提出了一种基于程序性质图的源代码软件漏洞挖掘方法。首先利用语法解析器解析源代码，依次生成语法分析树、抽象语法树、控制流图、数据流图；然后聚合抽象语法树、控制流图以及数据流图形成程序性质图，并定义程序性质图的基本遍历方式；最后，根据多种源代码漏洞的描述，在组合程序性质图遍历方式的基础上挖掘漏洞。实验结果表明，该方法能有效的检测各种类型的源代码漏洞。

论文提出了一种基于机器学习的缓冲区溢出漏洞挖掘方法。该方法首先总结了 7 类缓冲区溢出漏洞静态特征，分别为 sink 类型、缓冲区位置、容器、索引 / 地址 / 长度复杂度、边界检测、循环 / 条件 / 函数调用深度以及是否输入可控；然后，通过扩展的程序性质图检测缓冲区溢出漏洞的各类性质并将其向量化；然后利用有监督机器学习算法在已标记的训练集上训练分类器；最后利用此分类器在新的源代码程序中挖掘缓冲区溢出漏洞。实验结果表明，相对于其他静态分析工具，该方法能在较低误报的情况下挖掘漏洞。

论文提出了一种结合静态程序分析的高效符号执行技术。首先，该技术通过静态程序分析方法，从程序的控制流图中计算出循环程序的不变式；然后，对程序进行插桩，用循环不变式代替循环形成新的控制流图；最后，在新的控制流图

上进行符号执行。对比实验表明，该方法比纯粹的符号执行以及对循环进行定长展开的符号执行发现更多的漏洞，并且耗费的时间更少。

论文提出了一种细粒度变异的导向模糊测试方法。该方法首先利用导向模糊测试收集测试用例；然后利用时间递归神经网络训练出一个模型，用于判断对靠近目标区域起关键作用的字段，同时收集每个字段的权重；最后，通过上述模型判断当前测试用例的关键字段，并利用关键字段权重进行细粒度的变异生成测试用例。实验结果表明，相对于导向模糊测试以及普通的模糊测试，该方法能更有效的导向目标区域并发现漏洞。

关键词: **漏洞挖掘；软件安全；符号执行；导向模糊测试；测试用例生成**

ABSTRACT

With the development of information technology, software has become an important element of the world economy, culture, science &technology, education and military development. As one of the core infrastructures of information system, software is widely used in communication, finance, medicine, etc. However, with the increasing software function and the increasing scale, the software defects and security vulnerabilities are unavoidable, which brings serious threat to the reliability and security of the information system. In the military field, the information and communication system represented by C 4 ISR is the nerve center of the modern army, which plays a vital role in winning the information war under modern conditions. The vulnerabilities, if mastered by the enemy, will cause incalculable damage in wartime. At present, many countries are competing to develop the field of cyberspace, the United States, Russia, Japan and other countries treat the cyberspace as the fifth dimension of war space after the land, sea, air and space. One of the key elements of cyberspace attack and defense is vulnerability, vulnerability detection and exploitation for both sides of offense and defense has important value, the appearance of the Stuxnet virus indicates that the software vulnerabilities have been applied to cyberspace operations. It is of great practical significance to study the techniques of vulnerability detection and discover the vulnerabilities in software.

After many years of research, software vulnerability detection techniques have been put forward, such as static analysis, fuzzing test, symbolic execution and so on. But each technique has different advantages and disadvantages. The static analysis has high coverage and can detect many kinds of vulnerabilities, but its false positive rate is high. Fuzzing has low false positive rate but low code coverage and low efficiency, a large number of test cases may repeat the same program path. Symbolic execution is considered to be the most promising technique in vulnerability detection. The symbolic execution is path sensitive, so the false positive rate is low, and it can theoretically traverse all the paths of the program with constraint solver and once each path, the code coverage and execution efficiency is high, but the biggest problem that plagued symbolic execution is the path explosion problem.

This thesis analyzes the main problems of current software vulnerability detection,and proposes a new framework of vulnerability detection technique targeting binary pro- gram,

which uses the static analysis to locate suspected vulnerabilities and post-dynamic testing to verify the suspected vulnerabilities. The combination of program analysis and testing techniques can learn from each other and improve the efficiency of vulnerabil- ity detection. Based on the translation of the binary code to the intermediate code, the thesis locates the vulnerabilities on the intermediate code through the dataflow analysis and abstract interpretation, and then verifies the suspected vulnerabilities through program slicing, backward symbolic execution and guided fuzzing test.

The main work and innovations are as follows: This thesis presents a method to exploit data flow analysis and abstract interpretation to locate the vulnerabilities in intermediate code. Based on LLVM IR, this thesis presents a method based on boundary-based vulnerability model, which exploits data flow analysis and abstraction to perform vulnerabilities location and screening, and the located vulner- abilities can direct the program slicing, symbolic execution and fuzzing test. With the translation from binary code to intermediate code, on the hand it avoids the direct analysis of the complexity of the binary assembly code, on the other hand it ensures the universality of the static analysis algorithm, that is, any platform code can be converted to intermedi- ate code and then use existing algorithms to analyze, to avoid re-design algorithm for each platform. Based on the theory of data flow analysis, this thesis studies the algorithm of detecting UAF and double free vulnerabilities. Based on the theory of abstract inter- pretation, this thesis studies the method of interval analysis of key variables of memory vulnerabilities.

Key Words: Vulnerability Detection, Software Security, Symbolic Execution, Guided Fuzzing, Testcase Generation

第一章　绪论

1.1　研究背景与意义

随着信息技术的发展，软件日渐成为人民生活、经济发展过程中不可缺失的基础设施，软件安全对国家的经济、政治、国防以及个人隐私有着不可估量的重要影响。互联网技术将国家的文化、科技、艺术、宗教等意识形态领域的边界直接延展到人们的日常生活当中，传统的国土资源边界已被扩展至整个网络资[1]源空间，无论是对在政府、金融、通信、交通、贸易、物流、能源等国家核心基础设施的保障还是对涉及个人隐私的信息资料的保护无不依赖于软件的可靠性和安全性。然而，由于软件安全漏洞的广泛存在，给网络犯罪分子留下可乘之机，制造出的网络犯罪案件层出不穷，导致个人隐私曝光、商业机密泄露、重大经济损失或关键系统被恶意操纵。

由于软件漏洞的普遍性和高危害性，各国政府都十分重视软件漏洞的相关研究。2003 年，美国政府发布的国家安全战略报告《The National Strategy to Secure Cyber space》指出了网络安全防御面临的诸多问题，重点指出由软件安全漏洞问题引发的严重后果，特别强调了网络和软件安全漏洞问题的重要性。2013 年，美国国防高级研究计划局 (DARPA) 启动了一项预算达 5500 万美元的 Cyber Grand Challenge(CGC) 项目，旨在构造一个自动推理系统，自动发现软件中存在的漏洞并修补它们。我国在 2006 年发布的《2006-2020 年国家信息化发展战略》第四章“我国信息化发展的战略重点”中明确指出:“积极跟踪、研究和掌握国际信息安全领域的先进理论、前沿技术和发展动态，抓紧开展对信息技术产品漏洞、后门的发现研究，掌握核心安全技术，提高关键设备装备能力，促进我国信息安全技术和产业的自主发展”。

尽管软件安全问题得到极大重视，相关软件开发企业和研究机构等都投入了大量人力、物力去保证软件产品的安全性，但因漏洞引起的严重的安全问题还时有发生。例如，2014 年 OpenSSL 加密库被曝出的“心脏出血”（Heartbleed）漏洞影响了全球近 1/3 的主要网站；同年，与“心脏出血”同级别的破壳漏洞（ShellShock）被曝出，大约有 5 亿联网设备以及网络服务器受其影响；2015 年 glibc 库的 DNS 客户端被曝出存在缓冲区溢出漏洞，全球数以万计的 app、系统以及嵌入式设备收到影响。

随着各种开源社区的兴起，源代码组件在开源和商业软件中得到了广泛的应用，源代码软件漏洞的影响也越来越大。有研究表明[2]，超过半数的全球 500 强企业使用有漏洞的开源组件或者对开源库进行重新封装。另外一份来自 Aspect

Security 的调查报告[3]，审查了来自 60000 个企业、政府以及非盈利机构企业使用的网站和业务软件，发现超过半数的机构下载和使用了有漏洞的开源组件或者开源库，同时发现在调查之前绝大部分漏洞未得到修补。所以无论是商业软件还是程序员自行开发的小程序，开源代码已经变得越来越普遍，而且开源也已经成为了一种趋势。商业软件项目的免费版本数量已经超过了 50% 甚至更多，而开源软件产品所占比重从 2011 年的 3% 增长到了现在的 33%。平均每一款商业软件都会使用超过 100 个开源组件，而且三分之二的商业应用代码存在已知安全漏洞。Black Duck 软件公司的研究人员根据他们对开源项目所收集到的统计数据预测到，基于开源软件漏洞的网络攻击活动数量在 2017 年将增长 20%。

随着软件功能的日益增多和规模的日益庞大，软件存在故障缺陷和安全漏洞不可避免。源代码软件的漏洞挖掘技术能够针对性的对开源软件 / 组件进行挖掘，其研究一方面能够提升软件的安全性，对我国自主研发软件质量的提高具有重大意义；另一方面，随着开源软件 / 组件被国外军、政、企部门广泛的使用，对我国的漏洞资源储备以及网络空间博弈也具有重大战略意义。

1.2 源代码软件漏洞挖掘研究现状

1.2.1 软件漏洞挖掘方法分类

近年来，漏洞挖掘研究在理论方法层面不断更新，高效的挖掘工具层出不穷，软件漏洞挖掘技术的划分方式也存在不同的维度。这些挖掘方法按照是否使用运行时信息有静态、动态之分；按照自动化程度不同有手工、自动之分；按照挖掘对象不同有面向源码和面向二进制之分。图1.1中可以看到：人工审计、崩溃分析、程序调试、逆向工程是典型的手工分析方法；补丁对比、语法 / 语义分析、模型检测、抽象解释是典型的自动化静态分析方法；错误注入、模糊测试、二进制插桩是典型的自动化动态分析方法；污点分析和符号执行也是自动化分析方法，在动态分析和静态分析中都有应用。图1.2中可以看到：语法 / 语义分析、模型检测主要用于面向源码分析；程序调试、崩溃分析、逆向工程、二进制插桩则主要用于面向二进制分析；人工审计、补丁对比、抽象解释、污点分析、符号执行、模糊测试、错误注入则在源码和二进制分析中都有应用。本文重点关注面向源代码的动态和静态自动化分析方法。

1.2.2 源代码软件静态挖掘技术

静态挖掘技术是在不运行程序的情况下，自动的发现和报告潜在安全漏洞的技术的统称。静态分析能够全局的分析程序，不需要遍历每一个程序执行状态，从而扩大程序分析的规模。总体上，静态分析具有以下优点。(1) 能够在软件生

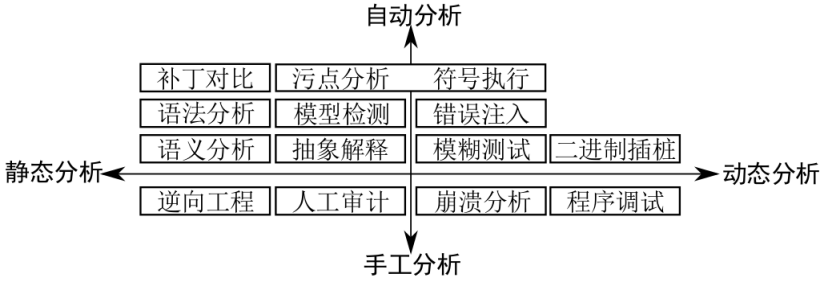


图 1.1　根据挖掘方式分类

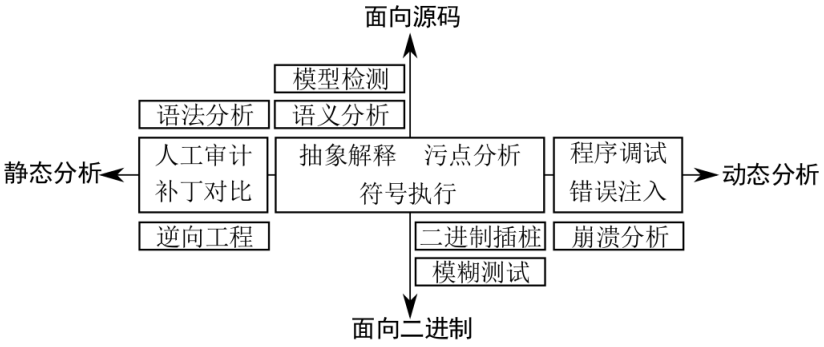


图 1.2　根据挖掘软件类型分类

命周期的早期发现漏洞。静态分析可以在函数单元上进行单元测试，这使得错误可以在集成到大的代码模块之前被发现。进行单元测试之后，可以将精力集中在函数 / 单元之间的交互漏洞检测上，实现对源代码的分层检测。(2) 能够检测大规模的软件。相对于人工审计，静态分析虽然压制了其报告解析能力，但是提高了检测特定漏洞的能力。相对于动态测试，不需要动态的遍历程序的每一个状态或者仅仅在源程序、中间表示上进行分析，能大大减少分析所需要的时间。同时静态分析也有以下不足。(1) 在没有源代码的情况下，例如第三方库、操作系统等，静态分析需要猜测这些缺失的程序行为。(2) 不能处理未域定义的程序错误行为。(3) 不能检测运行时的漏洞、设计漏洞，系统管理以及由于用户的原因产生的漏洞。

程序静态分析技术经过几十年的发展，形成了丰富的理论方法和技术手段，以程序静态分析为基础的静态漏洞挖掘方法的研究也极为深入。在词 / 语法分析、定理证明、抽象解释、符号执行、模型检验和机器学习等理论的支持下，静态漏洞挖掘取得了显著进展。

1.2.2.1　词 / 语法分析技术

词法分析通过对以往漏洞的分析，例如，人们发现缓冲区溢出常与某些语句或者语句组合有关（例如 strcpy，strcat），于是开发了一些通过词法分析发现缓冲

区溢出漏洞的检测工具，例如 RATS[4]、ITS4[5] 和 Flawfinder[6] 等。此类工具对程序的语句进行遍历，并与预先建立的漏洞模式进行匹配，最后根据匹配结果报告可能的缓冲区溢出漏洞。由于这种方法不理解程序语义，所以容易产生漏报或误报。此技术一般只作为手工分析的起点。

1.2.2.2 定理证明方法

定理证明的基本思想是将源程序的语义和安全规范转换为逻辑公式，再将这些逻辑公式输入到一个定理证明器进行验证，从而将软件错误的发现过程转换为逻辑公式的证明过程。这种方法的优点是严格，能够保证通过验证的程序不存在漏洞，但它通常要求用户（用形式逻辑语法）提供循环不变式、过程调用的前置条件和后置条件等信息，以帮助定理证明器完成推导，因而难以实现自动化。Compaq 公司的 ESC[7]、以色列 Tel-Aviv 大学的 CSSV[8] 都采用定理证明方法检测程序中的缓冲区溢出漏洞。

1.2.2.3 抽象解释技术

抽象解释把程序的执行过程看成抽象状态的迁移过程，并通过抽象状态的分析确定程序的性质。为了保证分析结果的正确性，抽象解释要求初始抽象状态是初始实际状态的安全近似，且每次状态迁移都能保持正确关系。正确的抽象解释能够发现程序中的漏洞，但存在较多的误报，所以在实际使用时常常要求用户根据被测试程序的特性，选择合适的抽象域、加宽操作和解释函数等，以降低误报率。例如，Patrick Cousot 领导的项目 ASTREE[9] 采用基于抽象解释理论的程序静态分析器。ASTREE 可以检测数组越界访问、除零异常、浮点运算溢出和整数运算溢出等问题。ASTREE 曾用于检验法国空中客车公司的空中巴士 A340 和 A380 系列飞机飞行控制软件，受到工业界的认可。除了 ASTREE，FLUCTUAT[10]、Coverity[11] 等都是采用抽象解释理论进行静态漏洞检测的典型工具。抽象表达提供了对变量边界值的估计，但无法有效追踪变量之间的约束关系，在路径可行性判定上依然存在不足。

1.2.2.4 静态符号执行技术

静态符号执行是 King 等人 1976 年在文献[12] 一文中首先提出。符号执行的基本思想是，用抽象的符号表示程序中变量的值，根据程序的语义，遍历代码执行空间，来模拟程序的执行。符号执行的主要优势在于能够发现变量之间运算关系，便于理解程序的内在逻辑；在漏洞挖掘时，有利于在复杂的数据依赖关系中发现数据之间本质的约束关系，而且符号执行精确记录了路径的约束条件，可以进一步用于判断路径可行性和路径约束的完备性。

符号执行技术通过将程序的输入由具体值替换为符号变元，将程序中每条指令本身的具体计算语义映射为相应的符号计算语义，在程序控制流指导下，关注

路径的分支谓词，为路径状态树中的每个叶子节点对应的程序路径推导出相应的能够到达该位置的敏感输入满足的最弱前条件。符号执行的核心，在于以等价类的形式对程序的执行路径进行标注，进而高效率地实现路径覆盖。

符号执行可以分析代码的所有语义信息，也可以只分析部分语义信息。符号执行分为过程内分析和过程间分析（又称全局分析）。过程内分析是指只对单个函数的代码进行分析，全局分析指对整个软件代码进行上下文敏感的分析。所谓上下文敏感分析是指在当前函数入口点要考虑当前的函数间调用信息和环境信息等。程序的全局分析是在过程内分析的基础上进行的，但过程内分析中包含了函数调用时就也引入了过程间分析，因此两者之间是相对独立又相互依赖的关系。

符号执行常常在对路径敏感的程序分析中使用。符号执行可以看作是程序测试与程序验证的折中方法，其优点在于它可以精确地静态模拟程序的执行。由于它跟踪了变量的所有可能取值，因此能够发现程序中细微的逻辑错误。但是在处理大程序时，程序执行的可能的路径数目随着程序尺寸的增大而呈指数级增长。在这种情况下需要对路径进行选择，选取定数量的路径进行分析。

早期静态符号执行典型系统包括 Prefix[13] 和 ESP[14]。W.Bush 等人在 1998 年提出了 Prefix，并在美国申请了若干专利。微软公司于 1999 年收购了 Prefix，在 Prefix 基础上实现了 Prefast。Prefix/Prefast 已经成为微软内部标准源代码静态检验工具之一。ESP 借鉴了 MC 由用户制定检查规则的思想，基于符号执行技术识别和保留路径上安全属性状态相关的约束条件，能够有效排除很多不可行路径。

1.2.2.5 模型检测技术

模型检验是一种针对有限状态并发系统的自动分析与验证技术，其基本思想是状态搜索。对状态空间的穷尽搜索有赖于对系统建立的有穷状态模型，以保证搜索过程的终止。模型检测的优点是能够做到完全自动化，在搜索终止时，如果性质没有满足，能够给出反例，这有利于用户对系统进行改进；缺点是建模比较困难，且面临状态空间爆炸的问题。一般情况下，模型检查只适合用于对临时属性（如内存是否被释放、指针是否为空等）的检测。

模型检验技术最早应用于时序电路和通信协议设计的自动化检验，其基本思想是用状态迁移系统描述程序的行为，用时序逻辑、计算树逻辑或 π 演算公式表示系统的性质，从而将系统属性的检验问题转换为搜索不满足逻辑公式的状态系统的问题。模型检验需要遍历系统的整个状态空间，所以只能分析有限状态系统。例如，Berkeley 大学的 MOPS[15]、微软研究所的 SLAM 以及 Bell 实验室的 UNO[16] 都是基于模型检验的检测工具，它们能够发现程序中的一些字符串库函数的滥用。MOPS 系统能够做到检测结果无漏报（完备性），但是 MOPS 是数据流不敏感的分析，无法跟踪数据依赖关系，导致 MOPS 误报率很高。SLAM[17] 是一种基于谓词

抽象技术的软件模型检验器，主要用于检测 Windows 操作系统驱动程序是否满足用户定义的属性约束。SLAM 的核心技术是谓词抽象，将每个变量都抽象为只有 0/1 两种取值。这种过近似（Over Approximations）抽象方法导致 SLAM 误报较为严重。由于模型检验技术面临着状态空间爆炸的问题，在大型复杂软件的漏洞挖掘工作中仍处于探索阶段。

有些模型检验工具，例如 Spin[18] 与 CBMC[19]，能够直接应用于源代码，准确地捕捉程序的真实行为，精确度比较高。但是，它们要么可扩展性不好，不能应用于大规模的程序（如 Spin）；要么只能检查大型程序的部分状态，用作一种查错工具，不能覆盖所有的状态，不能保证找到所有的错误。另外一些模型检验工具基于谓词抽象的思想自动化抽取系统的有穷状态模型，如 BLAST[20] 和 SATABS 等，它们的基本思路都是谓词抽象与 CEGAR（Counter Example Guided Abstract Refinement，反例制导抽象精化）[21]，这也是目前主流的软件模型检验实现思路，但谓词发现以及模型精化仍是此类技术中的难点问题。

1.2.2.6 基于机器学习的软件漏洞静态挖掘技术

随着机器学习算法的成熟，一些研究者将机器学习的有监督、无监督算法以及神经网络算法和程序分析相结合用于挖掘软件漏洞。Padmanabhuni[22, 23] 利用有监督机器学习算法建立分类器用以检测缓冲区溢出漏洞。Neuhaus[24] 根据 Mozilla 的漏洞历史，从中提取向量，并利用机器学习算法预测哪些组件有可能出现漏洞。Zimmermann[25] 以代码复杂度、代码混淆度等为特征预测 Windows Vista 中可能出现漏洞的组件。Perl[26] 利用 github 提交代码的语言类型、贡献者的 fork 数量等特征预测哪些提交可能出现漏洞。Grieco[27] 以 C 语言程序中的函数调用轨迹为特征预测大规模程序中的漏洞。Yamaguchi[28, 29] 利用异常检测检测源代码中缺少边界检测的情况以及使用聚类算法检测 Taint-Style 类型的漏洞。Rajpal[30] 提出了一种基于时间递归神经网络的提高模糊测试代码覆盖率的方法。

1.2.3 源代码软件动态挖掘技术

动态测试是对程序运行时特性的测试分析，有别于静态分析检查源程序的手段，动态分析通过检查运行时的程序来获取程序特性。静态分析的结果通常对于每次程序的执行都成立，而动态分析的结果可能只对某一次或某几次运行成立。动态分析正在被广泛应用于包括系统安全分析在内的多个领域，软件工程国际会议（International Conference on Software Engineering/ICSE）专门设立了动态分析研讨会（Workshop on Dynamic Analysis/WODA），会议的主题包括动态分析的各种研究和应用。

1.2.3.1 运行时监测技术

运行时监测 (Runtime monitoring) 是根据插桩的安全规范代码和检测代码在程序运行时检测是否违反安全规范。如斯坦福的 Michael Martin 等人的 PQL（program query language）[31]，首先使用上下文敏感、流不敏感、基于包含的指针别名分析，找出所有可能的匹配点，然后对这些匹配点进行插桩后在运行时监测。另外像 VS.Net 中使用的放在调用栈中的 canary，canary 值存放在用户数据和返回地址之间，一旦发现 canary 值被改变，则表明发生了缓冲区溢出。运行时监测首先要利用各种静态分析技术（如指针分析、别名分析、类型推断等），结合给出的安全规范，找出程序中所有可能违反安全规范的地方；然后利用代码插桩技术（包括静态插桩和动态插桩）对可能出现安全漏洞的地方插桩检测代码，同时插桩安全规范；最后利用动态检测算法在运行时检测程序是否违反安全规范。

1.2.3.2 动态符号执行技术

基于动态符号执行的漏洞挖掘，一般将关注的程序漏洞以符号断言的形式进行描述。对每条路径的符号分析过程中，在敏感的程序点处对漏洞断言的可满足性进行判定。如确定断言为可满足，即断定当前分析路径中存在该类型漏洞。近年来，该型技术得到了较为广泛的应用。

2005 年，Godefroid 等学者提出的 DART[32] 系统是最早的动态测试用例生成系统。DART 首先生成一个随机值作为外部变量的初始化输入，且对每个外部函数都返回一个随机值，由于随机值不能确保对程序中的每条分支都能覆盖到，因此随机测试的覆盖率通常很低。然而，DART 对输入的随机值进行符号化处理，即在随机化之后，这些变量被标记为符号变量。在遇到新分支时，一条路径约束通过不断收集执行过程中的约束来生成，而下一条路径则是通过对另一分支的约束取反来生成。如果一条路径不可求解，这个约束将用具体值代入，进行化简。DART 的缺点时不能处理指针语义，无法分析 C 语言中常见的指针操作。

CUTE[33] 被定义为一个混合符号执行工具，即联合了符号执行与具体执行。与 DART 系统类似，CUTE 允许用户通过指定代码来决定哪些输入可以被符号化，这使得任意外部用户输入都可以被替代。CUTE 的路径遍历是通过插桩来实现的。与 DRAT 相比，CUTE 最大的改进是能够处理指针操作和数据结构。

斯坦福大学的 Engle 等学者提出的 EXE[34] 系统，用于组件测试，在动态收集路径可达约束信息方面和约束路径精确求解方面的贡献较为突出。EXE 系统是一个源到源的编译器。该编译器将目标应用源码中的每一个赋值和运算前都插入对 EXE 系统符号化执行组件的函数调用，并在程序的分支语句前插入一段调用求解器代码，对当前分支条件进行求解并产生测试用例。这样，当被 EXE 编译后的程序真正执行时，真实的程序执行和符号化执行交替进行，并且符号化执行从程序

的具体执行中获取所需运行时信息，以收集路径可达约束。此外，EXE 系统实现了一个 SMT 求解器 STP，该求解器包含了 BV 理论、ARRAY 理论和整数理论。由于路径可达约束的收集和约束求解都较为精确，EXE 系统生成的测试用例软件缺陷的发掘效率较高。

KLEE[35] 系统是基于 EXE 系统的升级，该系统将 EXE 系统的源到源编译器变为 LLVM 编译器[36]。KLEE 系统纯符号化执行 LLVM 编译出的低级指令。这些改进使得 KLEE 系统可以更为普适地应用于各类 LLVM 支持的语言程序。KLEE 同样需要完整源码支持。并且由于 KLEE 系统采用完全符号化解释执行每一条 LLVM 指令，测试性能低。

SAGE[37] 系统是微软研究院研发的动态测试用例生成系统。与之前的系统不同，SAGE 系统直接对二进制程序进行动态追踪并生成测试用例。该系统采用动态二进制指令追踪工具 Nirvana，监控程序执行，追踪指令执行获得程序的执行流日志。然后再离线地对日志文件进行符号化分析执行，构建路径可达约束，以便为输入可控分支生成测试用例。与 EXE 系统不同，SAGE 系统为了能够对大规模程序进行分析，采取了一系列的简化措施:（1）SAGE 系统为了简化约束收集和求解的复杂性，忽略对非线性约束的收集与求解。（2）SAGE 系统不对程序中的指针进行分析。相对于 EXE 系统，以上简化措施一定程度上加快了 SAGE 系统对大型目标软件的测试用例生成速度，然而这却以损失测试用例对路径覆盖指导的精度为代价。SAGE 系统生成的测试用例中有接近 60% 的测试用例无法准确指导程序覆盖目标路径。此外，SAGE 系统对日志中的所有指令流都采用完全符号化解释执行，导致符号化执行时间占全系统工作时间的 25%。这就抵消了部分由前面所述简化操作带来的性能提升。

伯克利大学的 SmartFuzz[38] 系统与 SAGE 系统类似，采用了基于 Linux 下的二进制指令追踪工具 Valgrind 追踪程序执行和在线约束收集方式。SmartFuzz 系统仅支持整数溢出缺陷的发掘。

李根设计和实现了 Hunter[39] 系统，提出了基于路径完备可达理论以及污点可控指针分析的算法，能有效覆盖基于攻击面污点输入可达的测试路径，由于采用了基于虚拟机平台的符号化执行及线程监控技术，具有良好的跨平台兼容性。

在符号执行的实际应用中，还面临这许多问题，比如符号执行的精度问题[40]，复杂的输入格式问题[41, 42]，路径爆炸问题[43, 44]，符号指针问题[45] 等。其中路径爆炸问题是一个符号执行研究的一个重要分支。其主要形成原因在于，每一个分支条件语句都可能导致当前路径再分支出一条新的路径进行分析计算，而这是以指数的形式增长的。尤其体现在程序中存在跳转谓词与输入变元相关的循环结构时。

当前的解决办法包括：为循环引入“执行计数变量”，通过进行“执行计数变量”与输入的关联和程序中的变量与“执行计数变量”的关联，实现将程序中

“循环次数相关”的变量的值完全通过输入相关属性表达的抽象计算。作为一种摘要计算技术，约减了循环处理中的分析计算量；在对大量实际应用程序分析的基础上，仅仅为输入的“长度”和“部分输入前缀字符”引入符号变元，进行符号计算。该方法理论上并不是完备的，但实际应用中能覆盖较多的缓冲区溢出情形；有些相关研究提出综合使用程序切片、抽象解释、循环不变量计算和循环单值识别等技术对循环执行次数的上界进行估算：利用程序切片获取与循环结束判断条件相关的变量和语句，进而在此基础上进行抽象解释，对相关变量的取值范围近似获取。在经过循环不变量分析和循环单值识别后，进一步剔出变量值域，最终获取循环次数的估计值。

Godefroid 等在文献中通过为包含归约变量的输入相关的循环计算循环摘要[46]，避免了对每条循环相关分支的分析。但仅仅适用于循环中变量与归约变量存在线性关系的情况下；瑞士的开源项目 S2E[47] 提出了选择式符号执行的思路：仅仅对关注的程序部分（某一特定代码模块或访问某一特定数据的代码部分）进行符号执行，在关注的程序部分和不关注的程序部分之间，维护确保分析状态一致性的转换机制。

导向符号执行是符号执行中利用程序分析以及约束求解技术有效的搜索可能目标路径空间的一个方向[48]。Haller[49] 提出了一种导向危险库函数以及危险操作的区域的符号执行方法。另外一些研究者将导向符号执行应用到程序补丁验证[50–52]、增加符号执行代码覆盖率[53]、减少静态分析误报[54] 以及崩溃重现之上[55, 56]。但是导向符号执行是重量级的，需要大量的程序分析以及约束求解。

1.2.3.3 模糊测试技术

Miller 等[52] 在 1991 年首次提出了模糊测试的概念，目的是检测 UNIX 命令行工具程序的可靠性。因为事先不知道程序、输入的结构信息，最先的模糊测试被称为黑盒模糊测试。模糊测试通过不同的方式（例如比特反转、边界值替换、输入块删除和复制）变异程序输入，从而产生大量的畸形输入；然后将畸形输入反馈给程序执行，观察程序是否异常。这个简单且有效的方法奠定了模糊测试的基础。下面从三个方面概述模糊测试技术。

（1）基于模型的黑盒模糊测试技术

传统的黑盒测试[57] 没有考虑输入的结构信息，以至于变异的绝大部分输入因为不符合正确的格式标准而只能检测程序的一小部分路径。这种情况导致了传统的黑盒模糊测试不能挖掘程序的深层次信息。为了解决这个问题，研究者提出了基于模型的黑盒模糊测试框架例如 Peach[58] 和 Spike[59]。本质上，当输入是有格式的文件数据或者协议数据时，测试用例变异需要保证程序输入能够满足格式规范，从而通过程序的格式检测。这种方法增加了生成的测试输入的有效性，从而

能够测试更深更关键的程序状态。但是，有的文件或报文格式并未公开，如何获得准确的输入格式或者绕过对输入格式的校验成为研究的重点问题[60–64]。另外还有一些研究者将遗传算法应用到基于模型的黑盒测试技术当中[65–68]。

（2）覆盖率导向的模糊测试技术

覆盖率导向的模糊测试是通过一系列方法增加种子语料库代码覆盖率的技术。其出发点是基于这样一个判断“如果要检测某个程序元素 e 中的漏洞，种子语料库中就必须要包含一个种子在程序执行时能够到达 e”，这里 e 可以是一个语句、一个基本块或者其他的区域。在执行一个测试用例时，覆盖率导向的模糊测试工具[69–73] 使用轻量级的代码插桩在运行时收集代码覆盖信息。AFLFast[69] 设计了一种路径概率计算方式，变异概率较低的路径对应的测试用例能触发更多的路径，从而增加代码覆盖率。同时，设计一种测试用例变异能量分配方式，驱动模糊测试多变异遍历概率低的路径。Vuzzer[71] 为每一个基本块设置了一个权重，并且计算每条路径的权重，在模糊测试时着重测试更可能触发漏洞的路径。Sparks[72] 使用了遗传算法的思想变异定长的种子语料库并根据输入数据格式以遍历更深的路径。

（3）目标区域导向的模糊测试技术

目标区域导向的模糊测试是指在指定目标区域的情况下，通过设定一系列标准，驱动模糊测试执行到目标区域的技术。Marcel Böhme 在 AFL 的基础上开发了 AFL-go 工具[74]。该工具定义了每个测试用例和目标区域的距离，并设计一种基于模拟退火的能量分配策略，使得随着时间的增加到目标区域距离较小的测试用例变异次数增加，而到目标区域距离较大的测试用例变异次数减少，以此增加测试用例经过目标区域的概率。虽然 AFL-go 在一定程度上实现了导向性，但是其变异的方式依然是无序的。基于此分析，本文第五章提出了一种细粒度变异的导向模糊测试方法，根据关键字段权重变异测试用例以达到快速导向并发现更多漏洞的目的。

还有另外的研究者提出了基于污点分析的导向模糊测试技术[75, 76]。该技术利用污点分析方法判断哪些字段对导向目标区域具有关键作用，变异这些字段产生的测试用例能够以更大的可能性遍历到目标区域。通过这种方式能够极大的减少需要遍历的程序状态空间。

1.3 源代码软件漏洞挖掘的关键问题

通过对比分析现有的源代码软件漏洞挖掘方法及工具，可发现源代码软件漏洞挖掘存在以下发展趋势[77]。

(1) 从内置安全规则到可扩展安全规则。现在的源代码安全分析工具都会提供多种配置机制和接口，供分析人员根据实际需要编写程序实现自定义的安全分析[78, 79]。

(2) 从依靠抽象语法树的词法匹配的方式到到多中间表示混合匹配的方式。抽象语法树只能识别出程序中的敏感操作，且会产生大量的误报，将其他的中间表示形式，如控制流图和数据流图综合考虑能大大减少误报。

(3) 从单纯的程序员指导程序分析到以程序员为辅助的自动程序分析。程序员在安全分析中的参与比重越来越小。现在许多模型检测、抽象解释工具的使用，只需要程序员在可能发生漏洞的区域添加注释就可以完成。而另外一些分析工具[80, 81]，因为使用了更为复杂的语义提取方式，程序员的参与度近一步减少。

(4) 使用形式化方法辅助软件测试。软件测试方法本质上是遍历程序状态空间，容易引起状态空间爆炸。形式化方法通过提供语义等价的程序转换，能够减少状态空间。将二者结合，能够发现软件测试检测不了的漏洞。

(5) 使用源代码的直接或者间接信息辅助软件测试。越来越多的从源代码中获取的信息用于帮助符号执行、模糊测试等软件测试方法，例如程序的执行路径[51]、以及程序基本块的距离[74]。

通过多年的研究，源代码软件漏洞挖掘在上述各方面取得了不少进展，但仍然有很多问题有待解决。

（1）多种类源代码软件漏洞精确静态挖掘问题

繁多的源代码漏洞种类给漏洞挖掘带来了困难。由于不同种类的漏洞具有不同的特征，现存的基于模式匹配的漏洞挖掘工具，一方面不能检测多种源代码软件漏洞，另外一方面则会造成很高的误报。例如对于基于抽象语法树匹配的缓冲区溢出漏洞检测，通过遍历所有程序语句的抽象语法树虽然可以找到所有的内存拷贝函数，但是无法探知程序中是否对内存拷贝操作进行了边界检测；另外，抽象语法树对于缓冲区循环写造成的缓冲区溢出漏洞则完全没有检测能力。而上述问题可以通过将抽象语法树、控制流图、数据流图结合来解决。所以，研究多种中间表示的聚合方式并在此之上研究各类源代码漏洞的挖掘方法是一个关键问题。

（2）机器学习算法和源代码软件漏洞挖掘的结合问题

一般的漏洞静态挖掘方法则局限于特定的漏洞模式，若模式限制条件过强，则漏报率较高；若限制条件过弱，则误报率较高。对于缓冲区溢出漏洞，影响漏洞的静态特征有很多种，漏洞的形成是多个特征综合作用的结果。在源代码中这些静态特征可以通过各种中间表示获取，若将漏洞挖掘转换成机器学习中的分类问题，则能从以往的漏洞数据中学习出规律，解决上述问题。因此如何将机器学习算法和漏洞挖掘结合也是一个关键问题。

（3）结合形式化方法缓解符号执行的路径爆炸问题

动态符号执行作为典型的动态测试方法，在理论上能够有效遍历目标软件的状态空间，从而深度挖掘软件中存在的漏洞。但实际检测中，随着程序规模增大，符号执行必然会引起路径爆炸。导致符号执行路径空间爆炸的一个非常关键的因素是程序中的循环语句。在分析循环程序时，符号执行每一次进入循环体内部，都需要构造两条分支路径，分别对应了循环条件为真或假的情况，导致分支路径总数目随着循环次数的增加呈指数级增加。在源代码基础上的形式化方法能够将循环语义等价的转化成一个顺序语句，以缓解循环引起的路径爆炸问题。所以如何将形式化方法和符号执行结合缓解路径爆炸也是一个关键问题。

（4）源代码静态信息辅助的模糊测试导向问题

模糊测试已被证明是一种非常强大的软件漏洞测试方法，是工业界和学术界研究的热点。但是模糊测试一样有它的局限性，盲目的随机的变异测试用例很难到达测试程序的特定路径，导致一些漏洞很难被触发。而在源代码中，漏洞可疑区域及其与其他代码的距离信息可以通过静态分析获取。如何利用这些信息指导模糊测试动态的测试目标区域也是一个关键问题。

1.4 论文的研究思路与主要研究内容

源代码漏洞挖掘是一项系统性的工作，只有融合多种理论、方法和技术才能挖出潜在漏洞。软件漏洞挖掘不只是经验性的工作，而应该作为系统性的工程来考虑。源代码软件漏洞挖掘起源于程序分析与测试领域，虽然有许多理论方法支撑，但是各种方法都有其局限性。使用单一挖掘方法来发现软件漏洞越来越难，只有系统性地看待软件漏洞挖掘问题，才能有所突破。

随着软件安全工程的引入，很多曾经挖掘出大量有价值漏洞的方法与工具在挖掘漏洞方面的优势越来越不明显。需要借鉴这些方法以前取得成功的经验，研究新的挖掘思路。论文系统的研究了现有各种源代码软件漏洞挖掘基础理论和方法，理清这些理论和方法的脉络关系，明确这些方法的优点和不足，并在此基础上提出论文的漏洞挖掘思路和方法。论文的基本思路是：首先针对现有的基于模式匹配的挖掘方法不能精确挖掘多种类源代码软件漏洞的问题，研究将抽象语法树、控制流图和数据流图三种中间表示聚合的方法，并利用遍历聚合后的表示形式——程序性质图，挖掘各类源代码漏洞；其次，针对缓冲区溢出漏洞静态挖掘方法精确度不高的问题，利用缓冲区溢出漏洞的程序静态特征，研究基于有机器学习的缓冲区溢出漏洞挖掘方法，提高挖掘精度；随后，针对符号执行的路径爆炸问题，研究结合静态程序分析的高效符号执行算法，以控制符号执行规模；最后，考虑到现有导向模糊测试方法不能细粒度导向的问题，在利用时间递归神经

网络训练测试用例的关键字段、累积关键字段权重的基础上，研究细粒度变异的导向模糊测试技术，增强导向模糊测试的效率以及漏洞挖掘的能力。

主要研究内容包括：

（1）基于程序性质图的源代码软件漏洞挖掘方法研究

基于模式匹配挖掘多种类源代码漏洞必须结合多种中间表示形式。论文研究了多种中间表示聚合的方法。该方法利用语法解析器解析源代码，依次生成语法分析树、抽象语法树、控制流图、数据流图。在将抽象语法树、控制流图以及数据流图聚合成程序性质图的基础上，定义并利用程序性质图的各种遍历方法，挖掘多种源代码漏洞。

（2）基于有机器学习的缓冲区溢出漏洞挖掘方法研究

针对缓冲区溢出漏洞挖掘精确度不高的问题，利用缓冲区溢出漏洞的程序静态特征，研究基于有机器学习的缓冲区溢出漏洞挖掘方法。该方法将 22 种程序静态特征约简成 7 类，分别是 sink 类型、缓冲区位置、容器、索引 / 地址 / 长度复杂度、边界检测、循环 / 条件 / 函数调用深度以及是否输入可控。通过扩展的程序性质图检测缓冲区溢出漏洞的各种静态特征并将其向量化，利用有监督机器学习算法在已标签的训练集上训练分类器。此分类器可用于在新的源代码中挖掘缓冲区溢出漏洞。

（3）结合静态分析的高效符号执行技术研究

导致符号执行路径空间爆炸的一个非常关键的因素是程序中的循环语句。在分析循环程序时，符号执行每一次进入循环体内部，都需要构造两条分支路径，分别对应了循环条件为真或假的情况。分支路径数目随着循环次数的增加呈指数级增加。针对上述问题，研究了一种基于抽象解释的高效符号执行技术。给定待分析的源程序，首先解析出程序的控制流图。区别于其他符号执行技术，新技术通过静态程序分析方法，从程序的控制流图中计算出循环程序的不变式，然后通过用循环不变式代替循环，形成新的控制流图。在新的控制流图上进行符号执行，将会大幅减少符号执行的路径数量。

（4）基于细粒度变异的导向模糊测试技术研究

针对导向模糊测试的测试用例变异具有盲目性的问题，研究了一种细粒度变异的导向模糊测试技术。该技术首先利用导向模糊测试收集测试用例；然后利用时间递归神经网络训练出一个模型，用于判断哪些字段对靠近目标区域其关键作用，同时收集每个字段的权重；在动态执行测试用例之前，利用上述模型判断当前测试用例中哪些属于关键字段并根据字段权重进行定向的变异。本文方法能够更细粒度的变异指定字段，很大程度上消除模糊测试变异的盲目性，从而提高导向模糊测试的效率。

1.5 论文的组织结构

论文的结构排如图1.3所示。

第一章为绪论，介绍课题的研究背景与意义，概述软件漏洞挖掘领域的国内外研究现状与进展，阐述本文的研究思路和主要研究内容。

第二章介绍源代码软件漏洞挖掘的基础知识和动态符号执行测试方法。介绍的基础知识包括：源代码的四种中间表示形式、源代码漏洞分类以及静态程序分析的理论基础。

第三章介绍基于程序性质图的源代码软件漏洞挖掘方法。介绍将抽象语法树、控制流图和数据流图三种中间表示用性质图进行聚合的方法，并利用遍历聚合后的表示形式——程序性质图，挖掘各类源代码漏洞的方法。介绍了缓冲区溢出漏洞的程序静态特征以及基于机器学习的缓冲区溢出漏洞挖掘方法。此章节属于纯静态分析，并不能产生导致程序崩溃的测试用例，旨在为动态测试标定目标区域，缩小测试范围。

第四章介绍结合静态程序分析的高效符号执行技术。介绍了基于抽象解释的静态程序分析技术，设计了基于静态程序分析的高效符号执行方法。

第五章介绍基于细粒度变异的导向模糊测试技术。介绍如何利用时间递归神经网络获取关键字段以及如何计算关键字段权重；介绍在动态测试时，如何利用关键字段以及关键字段权重进行细粒度变异。

第六章总结全文，并展望继续研究的方向。

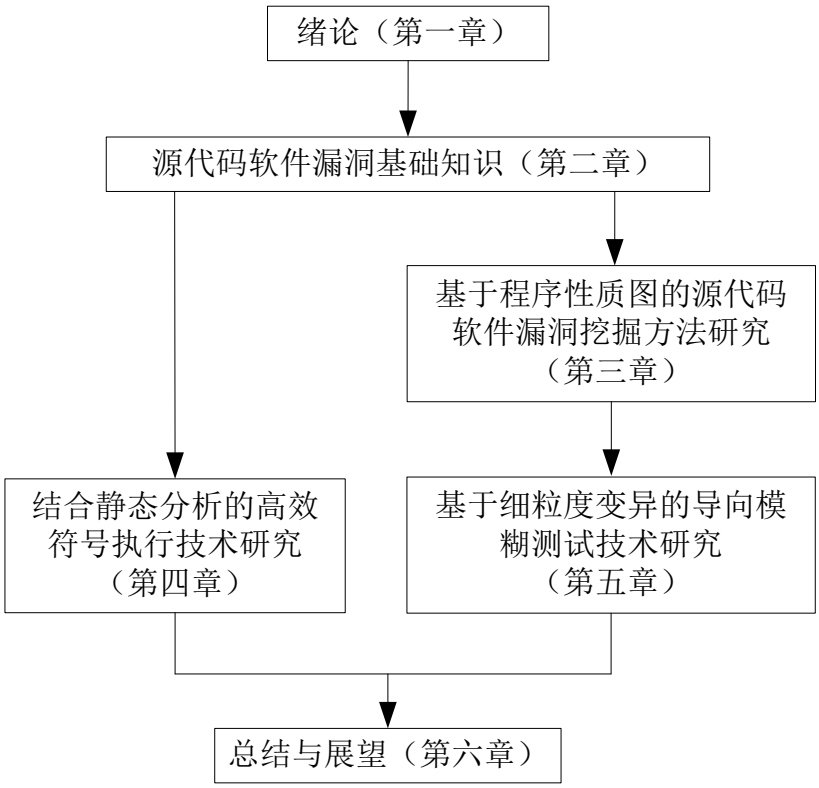

绪论（第一章）
源代码软件漏洞基础知识（第二章）
基于程序性质图的源代码软件漏洞挖掘方法研究（第三章）
结合静态分析的高效符号执行技术研究（第四章）
基于细粒度变异的导向模糊测试技术研究（第五章）
总结与展望（第六章）

图 1.3　论文的组织结构

第二章　源代码软件漏洞基础知识

为了更好的介绍源代码漏洞挖掘后续章节研究工作，本章首先介绍了源代码的各种中间表示形式；其次，根据不同的中间表示表达能力的不同，对源代码漏洞进行了分类；然后介绍了程序分析理论基础；最后介绍了一种常规源代码软件测试技术——符号执行技术。

2.1　源代码中间表示形式

因为源代码呈现出来的复杂性，在程序分析以及编译器设计技术里，研究人员提出了很多很多的中间表示用于表达程序。其中，语法分析树（Parsing Tree，PT）是语法解析器的直接产出结果，也是生成另外三个基本中间表示即抽象语法树（Abstract Syntax Tree，AST）、控制流图（Control Flow Graph，CFG）以及数据流图（Data Flow Graph，DFG）的基础。

2.1.1　语法分析树

语法分析树能够完全的反映程序语句的结构，以及如何将语句串联成一个完整的程序。输入一个程序，解析器根据语法文件辨别每一个终结符和非终结符，将这些标识符连接之后会得到语法分析树。语法分析树能够展示编程语言元素的类别、结构以及相互之间的关系，表现形式冗余性以及对微小改动非常敏感。Listing 2.1是一个简单的示意程序，变量 x 接受由外部输入函数 fun 得到的数据，y 是 x 的乘法运算的结果，vul_fun 函数是一个漏洞函数。图2.1是 Listing 2.1的语法分析树示意图，其根节点是一个函数（FUNC），FUNC 只有一个复合语句子节点（CMPD），复合语句包含两个语句（STMT）分别为赋值语句（ASSIGN）和跳转语句（IF）。

Listing 2.1　一个简单的实例程序

```
1 void fun()
2 {
3       int x = input();
4       if(x < MAX)
5       {
6             int y = 2 * x;
7             vul_fun(y);
8       }
9 }
```

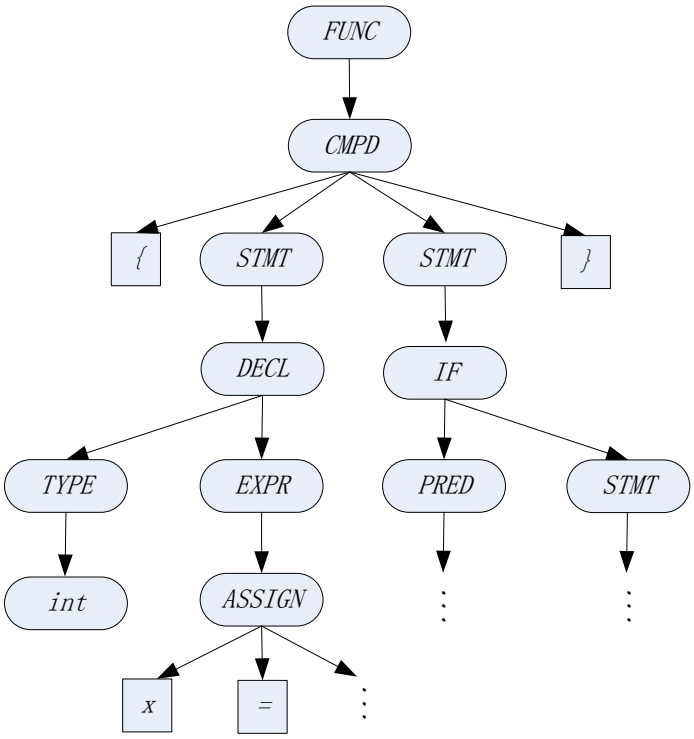


图 2.1　Listing 2.1语法分析树

2.1.2　抽象语法树

抽象语法树是语法分析树生成的最基本的中间表示，也是生成其他中间表示的基础。抽象语法树详尽的展示了操作数和操作符如何组成程序表达式以及语句，进而展示程序的整体形式。不同于语法分析树，抽象语法树不讲究和程序的每个token 表示完全一致，其更偏向于语义的相似性。例如，两个用逗号隔开的变量声明将会产生两个连续的变量声明语法树。

抽象语法树是有顺序的树结构，内部节点是操作符（例如“+”和“=”）而叶子节点是操作数（例如常量和标识符）。Listing 2.1的抽象语法树如图2.2所示。可以看出抽象语法树中删除了大括号和分号，另外函数调用节点（CALL）直接和赋值表达式相连，而语法分析树中在发现函数调用节点之前则需要遍历中间的所有节点。抽象语法树非常适合于简单的代码转换，经常被用来检验源代码的相似性[82, 83]。但是抽象语法树不适用于更复杂的代码分析，例如死代码和以及未初始化的变量的检测，其原因是抽象语法树不能提供明显的控制流和数据流信息。

2.1.3　控制流图

控制流图能够描述程序语句的执行顺序以及为了执行某个语句哪些条件必须满足。无论是一般程序语句还是控制语句都用节点表示，节点之间用有向边连接以传递控制关系。相对于抽象语法树，控制流图的每条边都有一个标签 *true*、*false*

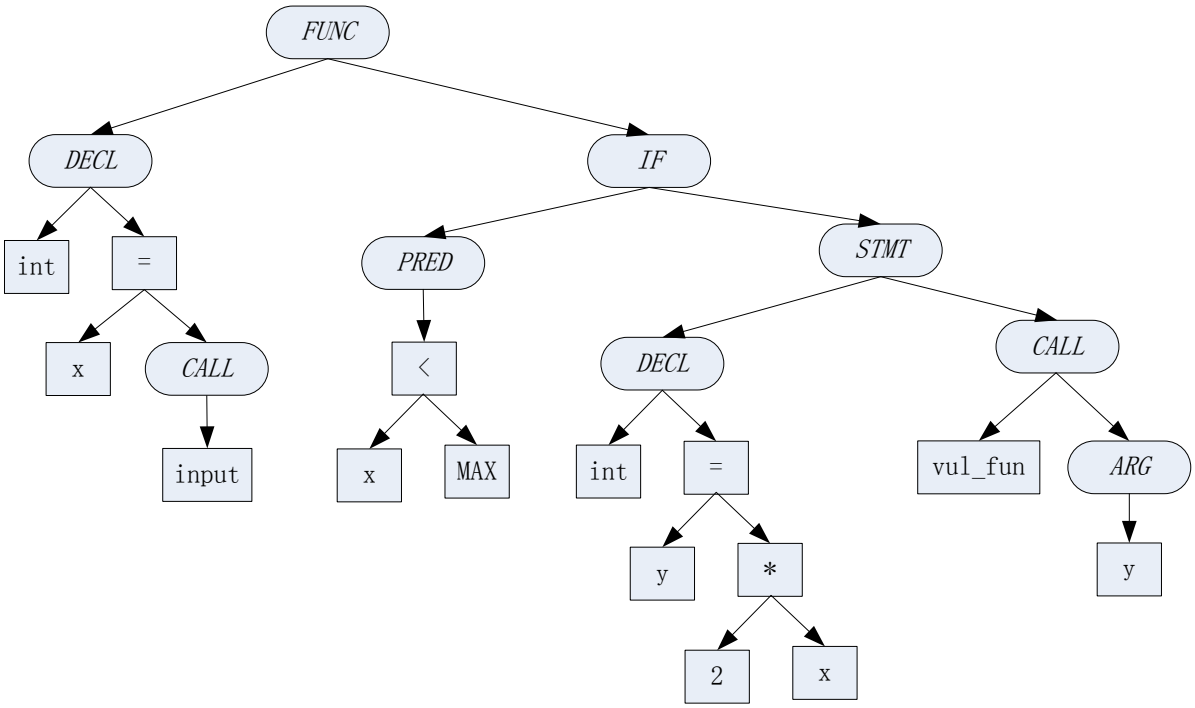


图 2.2　Listing 2.1抽象语法树示意图

以及 *true* 分别表示控制语句的二类出边以及连接其他语句的边。Listing 2.1的控制流图如图2.2所示。

控制流图在程序安全性分析中应用非常广泛，例如已知危险操作的变种[84] 以及引导符号执行[72] 等。另外，控制流图也已经被用在逆向工程中以帮助程序理解。虽然控制流图能够展示语句的执行顺序，但其不能提供数据流信息。在分析可疑漏洞时，不能判定某个语句是否可以被攻击者控制。

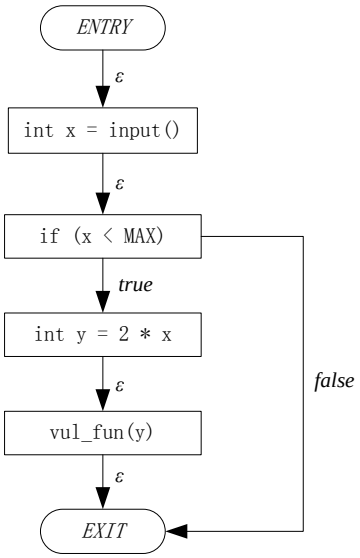


图 2.3　Listing 2.1控制流图

2.1.4 数据流图

数据流图能够表示程序的语句之间数据依赖关系，数据流边连接的节点之间语句与语句之间的执行顺序。在数据流图上，可以不实际运行程序，直接考察数据的转移情况，从而获取数据流属性信息。Listing 2.1的数据流图如图2.4所示。变量 x 从声明语句传递到两个语句 if(x<MAX) 和 int y = 2*x，变量 y 从声明语句传递到 vul_fun(y)。。

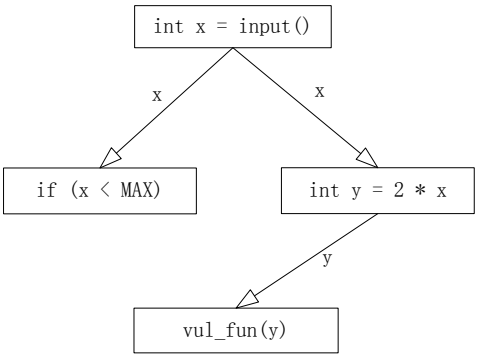


图 2.4 Listing 2.1程序数据流图

2.2 源代码漏洞分类

源代码漏洞挖掘方法都是建立在不同的中间表示之上的，根据表达源代码漏洞模型所需的中间表示的不同，本节将源代码漏洞为三类：（1）基于抽象语法树的源代码漏洞；（2）基于控制流的源代码漏洞；（3）基于污染传播的源代码漏洞。

2.2.1 基于抽象语法树的源代码漏洞

抽象语法树中可以获取源代码的每一个函数调用、语句甚至每一个操作数和操作符。所以利用抽象语法树可以获取攻击者可控的输入语句，敏感操作语句以及边界检测，但是不能检测出语句之间的关系。通过抽象语法树可以获取以下三类源代码漏洞的粗定位：

(1) 不安全的参数使用。不安全的参数是引起源代码漏洞的一个重要原因。例如，格式化字符串漏洞的一个重要重要的一个特征是传递给 *printf* 和 *sprintf* 的格式化字符串攻击者可控。所以在仅仅使用抽象语法树描述格格式化字符串漏洞时，可以将其描述为“*printf*，*sprintf*，*fprintf* 的格式化字符串不是常量”；

(2) 整数溢出。当内存分配函数（alloc*）的表示内存分配大小的参数存在复杂算术运算时，整数溢出漏洞就有可能发生；

(3) 整数数据类型错误使用。很多漏洞的产生是由于数据类型不正确的转换产生的，例如造成缓冲区溢出漏洞的一个重要原因就是缓冲区在初始化时内存大小的运算未进行正确的转换和验证。在进行赋值操作时，如果右操作数的位宽比左操作数大则会发生整数截断。

综上，下面给出基于抽象语法树的源代码漏洞描述。

定义 2.1: 基于抽象语法树的源代码漏洞可以用一个二元组 (M_0, M_1) 表示，表示满足 M_0 而不满足 M_1 的所有的语句。其中，M_0 和 M_1 表示两个谓词描述。

利用抽象语法树挖掘源代码漏洞是非常有效的，但是其既不能完全表达攻击者可控的变量和敏感区域的关系。所以仅仅使用抽象语法树会产生非常多的误报。下一小节将会阐述控制流图是如何帮助部分的解决这个问题。

2.2.2 基于控制流的源代码漏洞

因为在控制流图中可以获取程序的执行顺序，很多源代码漏洞不但需要抽象语法树，还需要控制流图才能检测，具体漏洞类型如下所示:

(1) 内存泄露。很多漏洞的产生是因为分配的内存没有正确的释放。内存泄露会导致程序崩溃，另外也可以导致其他的漏洞;

(2) Use After Free 漏洞。若一个内存区域在释放后继续使用则会导致程序崩溃或者任意代码执行。此漏洞触发的主要原因是看起来无关的函数调用语句之间复杂的控制流交互关系。使用控制流分析可以轻易的得出内存释放和内存使用之间是否有控制依赖关系;

上面两种情况，产生漏洞的原因都和控制流图上的某一条路径有关。例如，内存泄露是在一条控制流路径上内存被分配给一个变量，但在路径结束之前没有被释放而造成的。下面给出了基于控制流的源代码漏洞描述。

定义 2.2: 基于控制流图的漏洞可以用一个四元组 $(S_{src}, S_{end}, S_{dst}, \{(S^i_{cnd}, t_i)\}_{i=1...N})$，其中 S_{src} 是是一个基于抽象语法树描述的源语句，S_{dst} 是目的语句，$\{(S^i_{cnd}), t_i\}$ 是过程中条件的语法描述和对应的结果，$t_i \in \{true, false\}$。一个语句 v 如果满足了以下三个条件：(1) v 的语法树子节点中存在 v_{src} 与 S_{src} 想匹配；(2) 在 v_{src} 和 v_{end} 之间存在一条控制流路径，且路径中不包含一个节点和 S_{dst} 相匹配，其中 v_{end} 是和 S_{end} 相匹配的语法树节点；(3) 对于所有的 $1 < i \leq N$，若存在一个节点匹配 S^i_{end}，则从此节点的所有出边的标记必须是 t_i。 ■

控制流漏洞挖掘可以通过深度遍历从源节点到结束节点的所有路径，且结束节点不满足目的节点的描述，且中间节点必须满足相应的条件约束。但是仅仅使用控制流和语法树信息不能确定跟踪攻击者信息流，所以本文将在下一小节中介绍基于污染传播的源代码漏洞描述。

2.2.3 基于污染传播的源代码漏洞

基于污染传播的漏洞的检测需要抽象语法树、控制流图以及数据流图。符合此标准的源代码漏洞包括：缓冲区溢出、缺乏权限检测以及代码注入等。例如，很多的缓冲区溢出漏洞产生的原因是内存拷贝函数中的长度参数没有被有效的验证；检测此类型漏洞就必须首先要获取内存拷贝函数调用的长度参数，然后利用控制流分析每一个控制指令是否对其做了有效的验证，同时还需要使用数据依赖分析判断长度变量是否和输入有数据依赖关系。下面给出了基于污染传播的源代码漏洞的形式化描述。

定义 2.3：基于污染传播的漏洞可以表示为一个三元组 $(S_{src}, S_{dst}, S^{s}_{san})$，其中，$S_{src}$ 表示攻击者能够控制的输入语句，S_{dst} 是导致漏洞的敏感操作，S^{s}_{san} 是对应的边界检测。对于一个节点 v，若 v_{source} 是满足 S_{src} 的一个语法树子节点，v_{sink} 是满足 S_{dst} 的一个语法树子节点，且 v_{source} 和 S_{src} 满足以下个条件：（1）v_{source} 和 S_{src} 之间存在一条数据依赖路径，即二者有数据依赖关系；（2）对于每一条数据依赖边 $e_i = (v_i, v_{i+1})$，存在一条路径 $(v_0, ..., v_m)$ 且对于任意 v_k 不满足边界描述 S^{s}_{san}，$0 \le k \le m$。 ■

2.3 静态程序分析理论基础

直观上，静态程序分析是指在不实际运行程序的前提下，对程序进行语义分析。本节介绍程序的可达状态空间语义和不动点语义，二者是等价的。

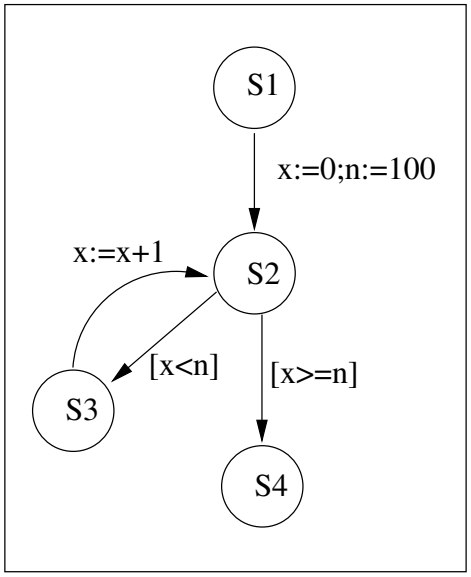
2.3.1 程序可达状态空间语义

本文用符号 Var 指代程序变量的集合，符号 $Func$ 指代函数的集合，符号 $Expr(Var, Func)$ 指代所有基于变量 Var 和函数 $Func$ 构造的表达式，如赋值表达式 $x = y + f(z)$。给定变量集合 Var 和函数集合 $Func$, 用符号 $Prog(Var, Func)$ 指代一段程序源代码。在计算机中，程序源代码的的一般表示形式为控制流图 (Control Flow Graph)。

定义 2.4：给定程序 $Prog(Var, Func)$, 其控制流图定义为一个三元组 $CFG_{Prog} = (L, E, l_0)$，其中，集合 L 表示是程序的所有控制节点，即源代码中程序指令位置；

集合 $E \subseteq L \times Expr(Var, Func) \times L$ 中的元素连接程序的控制节点，表示程序的控制流;l_0 表示程序的初始节点。

直观上，程序的控制流图是一个有向图，图的节点表示指令在源代码中的位置，图的边表示程序的控制流，同时边上含有一个指令或者一个基本指令块。一个简单的程序及其控制流图如图2.5所示。节点 $S1$ 表示程序的初始入口，边 $(S1, x = 0; n = 100, S2)$ 表示程序文本中顺序执行的赋值指令序列。




```
1    int x,n;
2    x=0;
3    n=100;
4    while(x<n)
5          x=x+1;
```

图 2.5　示例程序及其控制流图

控制流图存在其他的变种，例如编译器 LLVM/Clang 定义的控制流图将指令放在节点中。但是，它们所定义的程序语义是等价的。程序的语义通常由状态迁移系统定义。

定义 2.5: 一个状态迁移系统可以定义为一个三元组 $T = (S, R, S_0)$，其中，S 是所有可能状态的集合 (也称为状态空间);$R \subseteq S \times S$ 是迁移关系集合;S_0 是初始状态集合。

程序的状态空间由两部分构成：程序指令的位置和程序变量的取值。给定程序变量集合 Var，用符号 **Var** 表示变量所有可能取值集合，那么，程序的状态空间可以表示为 $S = L \times \mathbf{Var}$，其中，$L$ 是程序控制流图中的控制节点集合。程序的状态空间可能是有穷集合，也可能是一个无穷集合，如程序中含有整型变量时，其可能的取值范围为所有整数集合。同理，程序的初始状态可能有无穷多个，如变量未初始化时。

迁移关系 R 定义了程序的进行一步操作时的状态变化过程，例如，在图2.5所示程序中，迁移 $((S1, x = *, n = *), (S2, x = 0, n = 100))$ 表示了程序在 $S1$ 位置处执行赋值操作后，变量赋予相应的取值，其中，符号 $*$ 表示变量的取值为任意值。

一般地，程序的执行可以用一个迁移序列表示，例如，序列 $((S1, x = *, n = *), (S2, x = 0, n = 100)(S3, x = 0, n = 100), (S2, x = 1, n = 100), (S3, x = 1, n = 100))$ 表示图2.5 中的程序从初始状态执行，并进入一次循环的状态变化过程。为表述方便，本文用符号 $r \in R$ 表示一个迁移，序列 $r_0 r_1 \cdots r_n$ 表示一条迁移序列。

一个状态是可达的，当且仅当存在一条执行路径，使得程序能够从初始状态运行到达该状态。形式化地，状态 $s \in S$ 是可达的，当且仅当存在迁移序列 $r_0 r_1 \cdots r_n$，其中 $r_n = (s_n, s_{n+1})$，使得 s_0 为初始状态，且 $s = s_{n+1}$。

给定一些错误状态，或者不安全的状态，检测程序是否安全的问题可以归结为检测这些错误状态是否可达的问题，即是否存在一条执行路径，使得程序能够从初始状态执行到某些错误状态。因此，本质上，程序安全性分析问题可以归结为程序的可达状态空间遍历问题，即通过遍历程序的所有可能状态空间，检测是否存在不安全的状态。

2.3.2 程序不动点语义

程序的可达状态集合通常是通过计算程序语义泛函的最小不动点得到。该语义泛函是定义在程序状态迁移系统上的 *Post* 后继算子。

定义 2.6: 给定状态迁移系统 $T = (S, R, S_0)$，后继算子 $Post : S \times R \to S$ 是一个从状态空间和迁移关系到状态空间的映射。给定状态集合 $S_1 \subseteq S$，$Post(S_1, R) = \{s \in S | \exists s_1 \in S_1 \wedge (s_1, s) \in R\}$。 ■

与控制流图类似，后继算子也存在其他变种定义。在程序分析中，后继算子通常也定义为从状态空间和程序指令到状态空间的映射，即 $Post : S \times Expr(Var, Func) \to S$。通过将程序指令进行符号化编码 (symbolic encoding)，上述两个定义是等价的。为表述方便，本文对上述两种定义不做区分。

直观上，给定程序的当前状态和迁移关系，*Post* 算子定义了在下一时刻，程序从当前状态执行一步迁移可能到达的状态集合。

Post 后继算子作用在状态集合上。一个状态集合称为程序语义的具体域中的一个元素。因此，*Post* 算子也可以看做是程序语义的具体域上的函数。给定程序变量集合 Var，以及变量可能取值集合 $\mathbf{Var}$，程序的具体域为幂集 $2^{\mathbf{Var}}$。可证后继算子是该具体域上的单调递增函数。

引理 2.1: *Post* 后继算子是单调递增函数。

证明 2.1: 假设状态集合 $S_1 \subseteq S_2 \subseteq S$，证明义务为 $Post(S_1 R) \subseteq Post(S_2, R)$。

依据 *Post* 后继算子的定义，$Post(S_1, R) = \{s \in S | \exists s_1 \in S_1 \wedge (s_1, s) \in R\}$，那么对任意 $s \in Post(S_1, R)$, 存在 $s_1 \in S_1$，使得 $(s_1, s) \in R$。依据假设 $S_1 \subseteq S_2$，得知 $s_1 \in S_2$，故 $s \in Post(S_2, R)$。 ■

一般地，从程序的初始状态，执行 n 次 *Post* 后继算子，将得到程序执行 n 步可能到达的状态空间。理论上，程序的可达状态空间可以通过 *Post* 后继算子进行数学刻画: $Reach = \bigcup_{i=0}^{\infty}\{Post^i(S_0, R)\}$，其中，$S_0$ 是程序的初始状态集合。程序的可达状态空间可以通过不断迭代运用 *Post* 后继算子，直至得到所有可能状态。然而实际情况下，该迭代计算是不收敛的。假设上述计算在迭代 n 次后终止，表示程序的所有可达状态可以在 n 步计算内穷举，第 $n+1$ 计算不会产生新的状态，即 *Post* 算子存在不动点。

引理 2.2: 程序的可达状态空间 *Reach* 等价于程序语义泛函 *Post* 算子的最小不动点，即 $Reach = LFP(Post)$。

由 Knaster-Tarski 不动点定理可知，完备格上的单调函数存在最小不动点。因此，如果程序语义的对象域构成一个完备格，并且程序的语义泛函是一个单调函数，那么程序语义的最小不动点一定存在。然而，即便理论上存在最小不动点，该迭代过程的计算复杂度也随着程序规模的增长而呈指数级增长。这就是通常所说的状态空间爆炸问题。

2.4 动态符号执行

动态测试用例生成技术从 2005 年开始兴起，又被称为白盒 Fuzzer。该技术以一组随机生成的数据作为输入，通过分析程序动态执行信息来获取可达路径对输入数据的约束，并在遇到受输入数据控制的分支跳转时将收集到的约束，输出至可满足模求解器（SMT Solver），用以判定另一分支路径是否可达。若可达，求解器则给出覆盖目标分支路径的测试用例，并利用生成的测试用例发掘目标分支路径中的软件缺陷。典型系统包括微软研究院的 DART[32]、SAGE[37]、SmartFuzz[38]、Hunter[39] 等。

动态测试用例生成技术主要目标有两点：（1）为待测程序单元自动的生成可达到较高覆盖率的测试数据；（2）在动态过程中寻找待测程序的缺陷。

作为动态测试用例生成技术的前身，传统的符号执行技术使用符号变量代替待测程序单元的实际输入值，之后使用替代的程序执行语义模拟执行待测程序单元。在执行过程中遇到的包含符号变量的条件表达式即为可决定程序执行路径的路径约束。待测单元的所有路径可通过对路径约束的真值赋值表示为一棵二叉执

行树，其中每一个内部节点代表了一个分支路径选择。通过在树上深度优先的回溯搜索，寻找、判定预期路径约束的可达性并寻找对应的数据样本，满足预期执行路径的实际输入值即可被产生。但是，这种传统方法在处理大规模的或者较为复杂的待测单元时存在问题。例如，如果某一个路径约束的可满足性是无法判定的，那么传统的符号执行技术就无法继续进行搜索，从而导致了不佳的覆盖率；对于包含复杂数据类型（例如指针、数组）的程序，依靠静态的模拟执行无法精确的求解路径约束，同样也会在真实测试中导致低覆盖率。

动态测试用例生成技术的提出正是为了解决传统方法的不足，其基本想法在于将传统符号执行中的符号输入与具体值输入相结合。动态符号执行采用实际输入值真实地执行待测程序单元，在程序执行的过程中进行符号执行过程构建路径约束，同时记录通过符号变量表达的抽象程序状态，以及用实际数值表达的真实程序状态。由于待测程序被真实的执行了，所以所收集到的路径约束是遵循程序本身的语义的，且是精确的。因而在动态方法中不会出现如静态方法的虚假警报，并且也有助于对复杂数据结构进行分析处理。另外，由于程序的真实状态被记录，一旦遇到不可判定的路径约束，可使用变量的实际值替代符号值，使后续的路径探索过程可以继续进行。因为相比传统方法，可达到更高的覆盖率，所以其被广泛的应用在了程序测试与验证中，并且有相当多的工作扩展了动态符号执行的适用范围，如通过引入额外的内存模型以更精确的处理数组；通过引入额外的分析方法提高对循环的覆盖率等。

动态测试用例生成技术的一般流程如图2.6所示。其处理流程如下：

(1) 以某一具体输入启动目标程序，该输入可随机生成；

(2) 在目标程序的地址空间中，将输入对应的机器位置标记为符号变元；

(3) 对目标程序的每条指令，将其计算语义提升到基于符号变元的符号表达式域下，计算其抽象语义，更新对应的符号机器状态。对于符号计算相关的流程转移类的指令，计算其对应于当前具体执行路径的分支谓词，累积至当前路径对应的全局约束条件，同时计算另外一条分支对应的符号约束，通过约束求解进行可满足性判断，确定该路径的可行性。在可行的情况下，将分支对应的路径约束保存入调度队列；

(4) 若调度队列非空，则从调度队列中抽取出一个分支约束，计算出满足约束条件的输入的具体值，回到步骤 1 对该路径进行符号分析；否则，即结束对目标程序的符号分析。

论文第四章使用的符号执行工具是 KLEE[35]。KLEE 是由 Cadar 与 2008 年开发的动态符号执行工具。KLEE 依托于 LLVM 编译器，支持的最高的 LLVM 版本

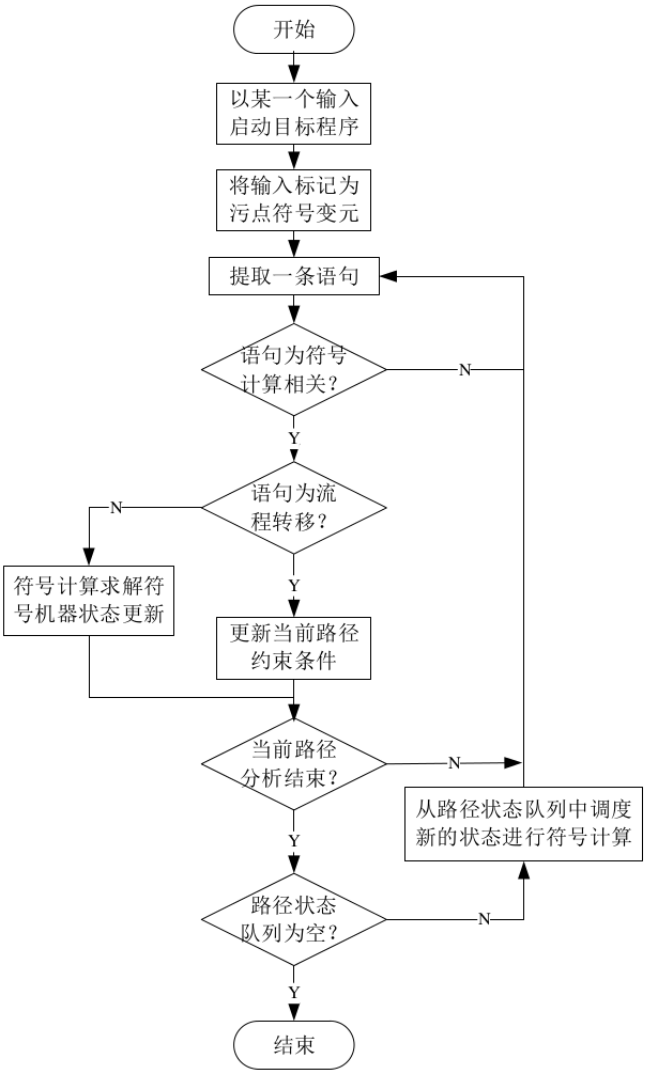


图 2.6　动态符号执行流程

是 3.4。KLEE 执行 LLVM 字节码，记录程序中不同的路径，使用运行时检查器（Runtime Checker）检测程序错误，并利用约束求解器求解路径约束产生测试用例。KLEE 能够获取很高的代码覆盖率并且发现深层次漏洞。

KLEE 的架构如图2.7所示。和很多其他的符号执行工具一样，KLEE 包含的一个主要的组件是 metaSMT 接口，支持多种约束求解器，例如 Z3，STP，Boolector。KLEE 还包含多种搜索策略即深度优先、宽度优先、随机搜索、最大覆代码覆盖率以及混合搜索策略，且提供了基本的搜索策略接口，可以方便的自定义搜索接口。此外，KLEE 还实现了环境交互模型，即利用自己编写的数据读写函数代替程序原始调用的函数，例如 open，read 等。此做法有三个好处：（1）简化版的函数能够减轻原始复杂程序造成的路径爆炸；（2）能够根据代码规范模拟闭源的代码库，使得符号执行能够顺利执行；（3）通过模拟网络通信可以符号执行网络程序。尽

管环境建模有这些优点，但是程序的模拟代码都是使用者手动编写的，每次遇到不同的程序需要重新编写。

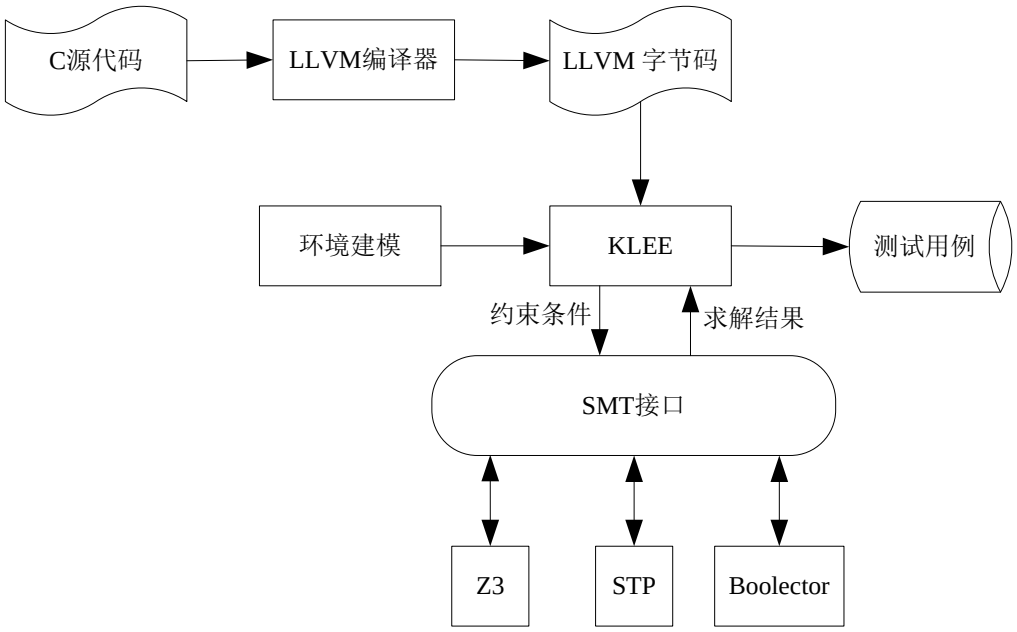


图 2.7　KLEE 架构

2.5　本章小结

本章介绍了源代码漏洞挖掘中所使用的四种中间表示：语法分析树、抽象语法树、控制流图以及数据流图。在分析了各类中间表示的表现能力的基础上，根据描述各类漏洞需要的中间表示的不同将源代码漏洞分成了三类：基于抽象语法树的源代码漏洞、基于控制流的源代码漏洞以及基于污染传播的源代码漏洞。然后，介绍了第四章所使用的程序静态程序分析基础以及动态符号执行技术。

第三章　基于程序性质图的源代码软件漏洞挖掘方法研究

根据 Rice 定理[98]，通过一个程序完全的自动检测其他程序的复杂属性是不可行的。一般的源代码静态分析工具都是基于字符串匹配或者抽象语法树匹配的，所以现行的工具要么局限于特定类型的漏洞，要么需要大量的人工审计工作[99]用以消除误报。例如 PREfast[13]、RATS[4] 以及 PScan[100] 虽然能够检测一些程序开发过程中产生的漏洞，但是很难检测成因复杂的漏洞。而另外一些工具，例如 Splint[81] 只能检测特定的漏洞。

本章提出了一种基于程序性质图的源代码漏洞挖掘方法。该方法利用程序性质图聚合抽象语法树、控制流图、数据流图三种中间表示，通过定义以及组合不同的程序性质图遍历方式能够检测多种源代码软件漏洞。针对缓冲区溢出漏洞挖掘精确度不高的问题，本章提出了一种基于机器学习的缓冲区溢出漏洞挖掘方法。该方法首先将缓冲区溢出静态特征分成 7 类；然后，在已标记的漏洞中，利用扩展的程序性质图获取静态特征并将其映射到向量空间，作为机器学习算法的训练集；最后，选取 5 种有监督机器学习算法在训练集中训练分类器，并通过分类器在新的源代码中挖掘缓冲区溢出漏洞。

3.1　程序性质图生成

源代码软件漏洞分析依赖于多种代码的中间表示例如语法分析树、抽象语法树、数据流图、控制流图等，对于 C/C++，很多编译器的前端能够很容易的通过插桩的方式生成中间表示。但是当分析涉及到复杂的程序或者跨越多个程序搜集中间表示时，编译程序所需要的环境就很难得到保证，特别的是，一般的编译器前端无法处理代码有头文件缺失的情况。与源代码相关的中间表示都直接或者间接的建立在语法分析树的基础上，所以程序性质图的生成首先需要将源代码转换成语法分析树。在语法分析树的基础上生成抽象语法树、控制流图、数据流图，然后将各种中间表示合并为一个统一的表示形式——程序性质图，并存入图数据库中以备查询。程序性质图的生成过程如图3.1所示。

3.1.1　鲁棒的源代码中间表示生成

鲁棒的源代码中间表示生成顺序依次是语法分析树、抽象语法树、控制流图、数据流图。语法分析树建立在 ANTLR[101](ANother Tool for Language Recognition) 的基础上。ANTLRv4 是一种非常强大的用于读取、执行或者翻译结构性的文本以及二进制文件的解析器生成器，广泛的应用于建立编程语言、工具和框架上。输入一个语法和源代码，ANLTR 能够生成一个用于构建遍历解析树的解析器。对于

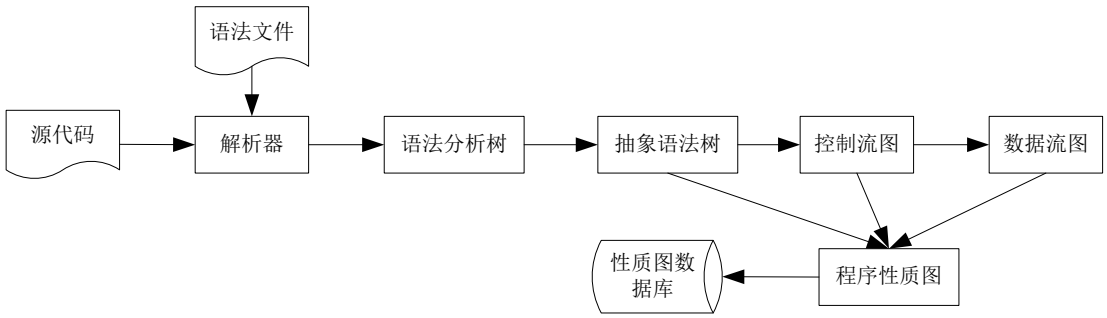


图 3.1　程序性质图生成过程

C/C++, 鲁棒的语法分析基于岛屿语法（Island Grammar）实现了三个层次上的语法分析器：分别是模块解析器、函数解析器以及语句解析器。模块解析器解析粗略的结构信息，例如函数、命名空间以及全局的变量声明。函数解析器解析辨别在一个函数内部能够影响到程序控制流的程序结构，例如跳转语句 goto、continue 和 break；选择语句 if 语句、switch 语句；循环语句 for 循环、while 循环和 do-while 循环。语句解析器主要的作用是将任意一个语句分解为表达式、操作符以及标识符等。岛屿语法[102] 是一种定义文本结构的语言，通过此语法可以定制解析器的粒度。

Listing 3.1是模块解析器的岛屿语法片段，第一行中的 module 表示一个模块，function_def 表示一个函数，simple_decl 表示变量或者类的声明，water 表示其他的任意未定义字符，Listing 3.1表达的意思是一个 module 可以由一个或者多个 functionDef、simple_decl、using_directive 或者 water 组成。Listing 3.2是函数解析器的岛屿语法片段，表示函数有哪些元素组成，其中 template_declStart，return_type，function_name，compound_statement 分别表示函数模板、函数返回类型、函数名以及函数内容。Listing 3.3是语句的岛屿语法片段，第一行表示 statements 可以由条件编译标识符（pre_opener，pre_closer，pre_else 分别表示 #if，#endif 和 #else）和语句 statement 构成。statement 的组成当中最主要的是 simple_decl 和 expr_statement。expr_statement 是 C /C++ 作为程序主体的一行行代码，在岛屿语法中需要按照运算符的优先级、结合进行设计（14-17 行）。解析器按照语法文件上规定的顺序解析程序中的非终结符，其他的未被制定的终结符和非终结符用 water 表示。

Listing 3.1　模块解析器的岛屿语法片段

```
1 module : (function_def | simple_decl | using_directive | water)
      *;
2 using_directive: USING NAMESPACE identifier ';';
3 simple_decl : (TYPEDEF? template_decl_start?) var_decl;
4 ...
5 water : any_token
```

Listing 3.2　函数解析器的岛屿语法片段

```
function_def : template_declStart? return_type? function_name
   function_paramList ctorList? compound_statement;
template_decl_start : TEMPLATE '<' template_param_list '>';
...
returnType : (functionDeclSpecifiers* typeName) ptrOperator*;
...
function_param_list : '(' parameter_decl_clause? ')'
   CV_QUALIFIER* exception_specification?;
parameter_decl_clause: (parameter_decl (',' parameter_decl)*) (
   ',' '...')? | VOID;
parameter_decl : param_decl_specifiers parameter_id;
...
function_name: '(' function_name ')' | identifier | OPERATOR
   operator;
...
compound_statement: opening_curly (statements) closing_curly;
...
```

Listing 3.3　”语句解析器的岛屿语法片段”

```
statements: (pre_opener | pre_closer | pre_else | statement)*;
statement: opening_curly
       | closing_curly
       | block_starter
       | jump_statement
       | label
       | simple_decl
       | expr_statement
       | water;
...
expr_statement: expr? ';';
expr: assign_expr (',' expr)?;
assign_expr: conditional_expression (assignment_operator
   assign_expr)?;
conditional_expression: or_expression
                 | or_expression ('?' expr ':'
                    conditional_expression)
or_expression : and_expression ('||' or_expression)?;
and_expression : inclusive_or_expression ('&&' and_expression)
   ?;
...
```

输入一个程序，解析器根据语法文件辨别每一个终结符和非终结符，将这些标识符连接之后会得到语法分析树。语法分析树能够展示编程语言元素的类别、结构以及相互之间的关系. 语法分析树是上述解析器输出的唯一表示形式，它忠实的按照语法文件解释源代码的结构细节，其表现形式较为冗余且对微小改动非常敏感。ANTLRv4 内置了生成语法分析树的功能。语法分析树如图2.2所示。

抽象语法树能够直接从语法分析树生成，与语法分析树相比，抽象语法树删除了对程序没有影响的实现细节，更加注重代码的结构，是一种更为紧致简明的中间表示形式也更加的适合于源代码的分析。图2.2是 Listing 2.1的抽象语法树的

示意图。抽象语法树能够松的定位和检测程序中的每一行代码以及每行代码中的变量、变量类型操作符等信息，但却不适用于检测代码的执行顺序，所以需要程序的控制流图来展示程序语句的执行顺序以及控制条件。

在辨别程序跳转关键词的基础上，程序的控制流图可以从抽象语法树直接生成，常见的关键词有 *if*，*for* 以及 *goto*。根据这些信息，可以使用以下两步将抽象语法树转换成控制流图。首先，在过程内顺序连接抽象语法树中的语句节点，定义 *if*，*for* 以及 *while* 语句转换为控制流图的方式并在所有的抽象语法树上使用此规则；然后处理 *break*，*continue* 以及 *goto* 等跳转语句矫正控制流图。实际上，第二阶段仅仅在控制流图上加入了由跳转语句引入的额外的边。图2.3是 Listing 2.1的控制流图。

每一个程序语句可以使用（use）或者使用（define）变量。如果语句 $s2$ 使用了 $s1$ 定义的变量 v，且从 $s1$ 到 $s2$ 存在一条路径，且 v 在这条路径中未被重定义，则称 $s2$ 数据依赖于 $s1$。将具有数据依赖关系的程序语句使用数据流边连接起来则形成程序的数据流图。

3.1.2 程序性质图

抽象语法树、控制流图和数据流图中的任何一种都不能单独的提取完全的缓冲区信息，所以需要一种综合的表示形式——程序性质图（Code Property Graph, CPG）程序性质图可以通过存入专门的图数据库例如 Neo4j，Titan 和 OrientDB，然后利用专门的数据库查询语句对程序语句节点以及语句节点之间的关系进行查询。本节首先介绍性质图（Property Graph，PG）的基本概念；然后介绍如何将抽象语法树、控制流图和数据流图合成程序性质图，最后介绍了如何将程序性质图如何扩展以完成过程间的程序分析。

3.1.2.1 性质图

在图理论当中，有向图可以用一个关系对表示，$G = (V, E)$，V 是节点集合，$E \subseteq (V \times V)$ 是节点之间边的集合。性质图的节点具有以下特性：（1）每个节点都对应一个唯一标识符；（2）每个节点都包含一个入边和出边集；（3）每个节点都包含键 -值集。性质图的边具有相似的性质：（1）每条边对应一个唯一标识符；(2）每条边都包含一个标签表示边的类型；（3）每条边都有一个键 -值集；（4）每条边都有一个源节点和目的节点。图3.2展示了上述特性。

定义 3.1: 性质图 $G = (V, E, \lambda, \mu, s, d)$ 是一个有向的、边标记的、属性化的多重图，V 是节点的集合，E 是有向边的集合，$s : E \to V$ 和 $d : E \to V$ 分别是有向边到源节点和目的节点的映射，$\lambda : E \to \sum$ 是边标记函数，将一条边映射到

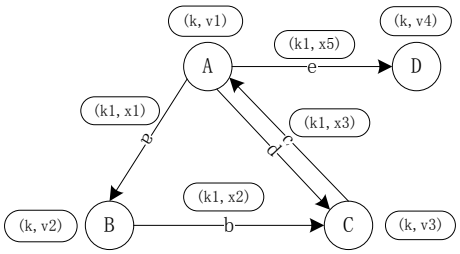


图 3.2　性质图示意图

字符集 $\sum$ 中的一个字符。节点和边都用键—值对表示性质，性质的赋值函数为 $\mu : (V \cup E) \times K \rightarrow S$，$K$ 和 S 分别是性质的键和值。

3.1.2.2　抽象语法树性质图

程序性质图是一个抽象语法树、控制流图和数据流图的综合表示。在本小节分别定义三类中间表示的性质图表现形式，然后再定义程序性质图。

定义 3.2: 抽象语法树性质图 $G_A = (V_A, E_A, \lambda_A, \mu_A, s_A, d_A)$，$V_A$ 是抽象语法树节点的集合，E_A 是边集合，λ_A 是边的标签函数，用任意字符标记每一条边，s_A 和 d_A 是边到节点的映射，分别将边映射到源节点和目的节点。

抽象语法树节点的性质包括：节点类型 (*type*)、代码（*code*）、节点序号（*childnum*）、操作符（*operator*）、位置 *location)*、所属文件 *file* 是否控制流节点（*isCFGNode*）。图3.3为 Listing 2.1的生成的部分抽象语法树。

定义 3.3: 控制流性质图可以用 $G_C = (V_C, E_C, \lambda_C, \mu_C, s_C, d_C)$ 表示，其中 $V_C = \{v | v \in V_A \cup \{ENTRY, EXIT\}$, 且$\mu_C(v, type) = Stmt$或者$C\}$，其中 $ENTRY$ 和 $EXIT$ 是控制流性质图的入口节点和结束节点，E_C 是边的集合，标签函数 λ_C 将边映射到字符集 $\sum_C = \{true, false, \epsilon\}$，$true$ 和 $false$ 用于标记 *Condition* 语句的两种出边，ϵ 用于表示其他没有控制流跳转的边。

图3.4是 Listing 3.4的控制依赖性质图的一部分，在此图中删除了 $ENTRY$ 节点且节点的属性 *isCFGNode=True*。

定义 3.4: 数据依赖性质图可以用 $G_D = (V_D, E_D, \lambda_D, \mu_D, s_D, d_D)$ 表示，其中 $V_D \subseteq V_C \subseteq V_A$，$E_D$ 是边的集合，标签函数 $\lambda_D : E_D \rightarrow \sum_D$，$\sum_D = \{D\}$，其中 D 表示数据依赖，同时在表示数据依赖关系的边上增加一个属性 *symbol* 表示传递的标识符。

图3.5是 Listing 2.1的数据依赖图的一部分，此图的节点是是控制流依赖图节点的子集，每条边都有 *symbol* 属性表示传递的数据。

通过聚合上述三种性质图的定义，可以获得程序性质图的定义如3.5所示。

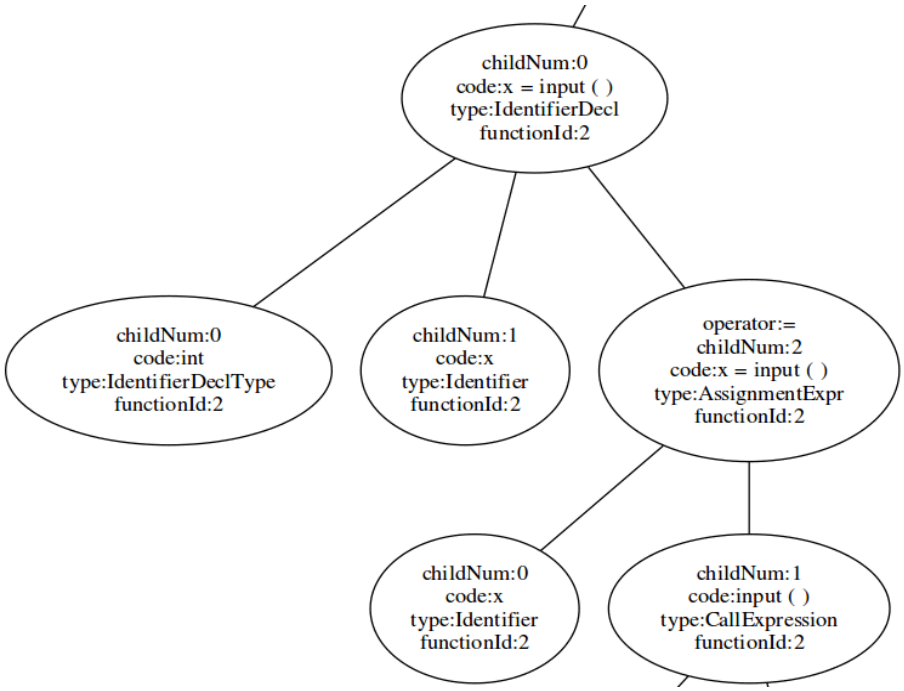


图 3.3　抽象语法树实例

定义 3.5: 程序性质图可以用 $G = (V, E, \lambda, \mu, s, d)$ 表示，其中 $V = V_A \cup \{ENTRY, EXIT\}$, $E = E_A \cup E_C \cup E_D$, $\lambda = \lambda_A \cup \lambda_C \cup \lambda$, $\mu = \mu_A \cup \mu_C \cup \mu_D$, $s = s_A \cup s_C \cup s_D$, $d = d_A \cup d_C \cup d_D$。■

3.1.3　程序性质图的基本遍历方式

生成程序性质图之后，需要有效的数据检索方法用以搜索源代码中特定的代码片段。图数据库提供了专门的查询语言 Gremlin[104]，给定节点或边，根据程序性质图边和节点的性质寻找相关节点和边，*traversals* 表示其形式化的表现形式。

定义 3.6: *traversal* 可以表示为 $\tau : P(V \cup E) \rightarrow P(V \cup E)$，其中 V 和 E 分别表示程序性质图的节点和边的集合，$P(V \cup E)$ 表示 $(V \cup E)$ 的子集。

最基本的 *traversal* 是在性质图数据库中搜索符合条件的节点，称之为过滤器，用 $Filter_p$ 表示，如下式所示，其中 p 表示过滤条件的谓词形式。p 可以任意指定，例如在查找所有 $type = Array$ 的抽象语法树。

$$Filter_p(X) = \{x \in X : p(x)\}$$

给定节点搜索查找相应边的 *traversal* 可以用下面的公式表示。X 是所有节点的集合，第一行公式表示在 X 中搜索所有的出边被标记为 l，且边的性质 k 等于 a 的所有节点。相应的，第二个公式表示在 X 中搜索所有的入边被标记为 l，且边的性质 k 等于 a 的所有节点。

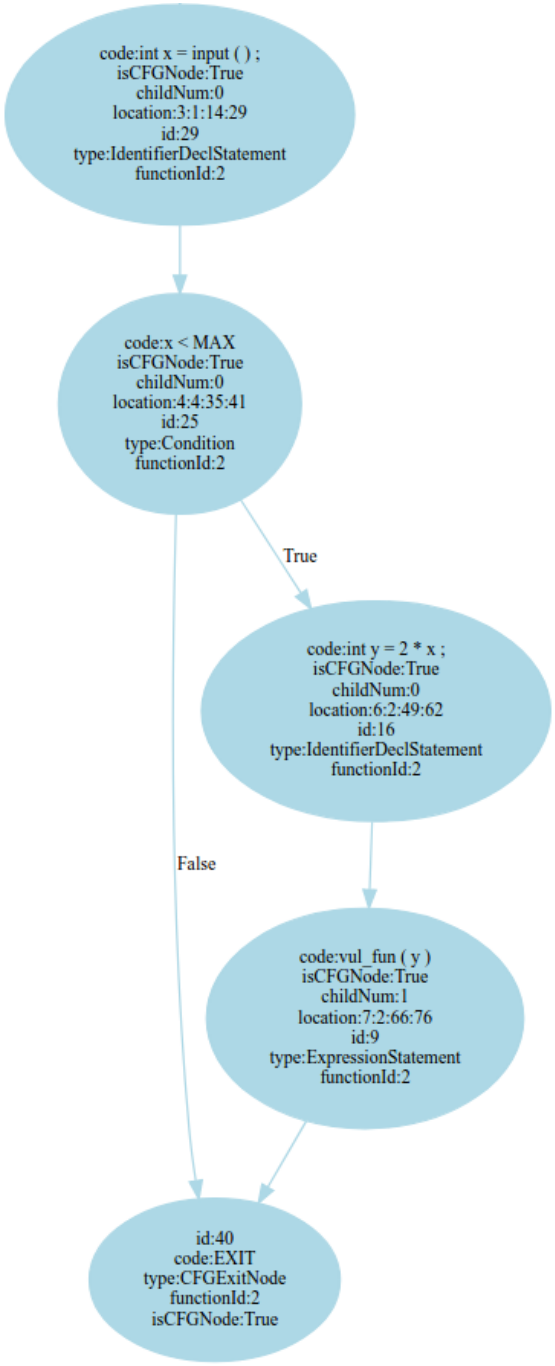


图 3.4　控制依赖性质图实例

$$OutE_l^{k,a}(X) = \bigcup_{x \in X} \{e : e \in E \text{ and } s(e) = x \text{ and } \lambda(e) = l \text{ and } \mu(e,k) = a\}$$

$$InE_l^{k,a}(X) = \bigcup_{x \in X} \{e : e \in E \text{ and } s(e) = x \text{ and } \lambda(e) = l \text{ and } \mu(e,k) = a\}$$

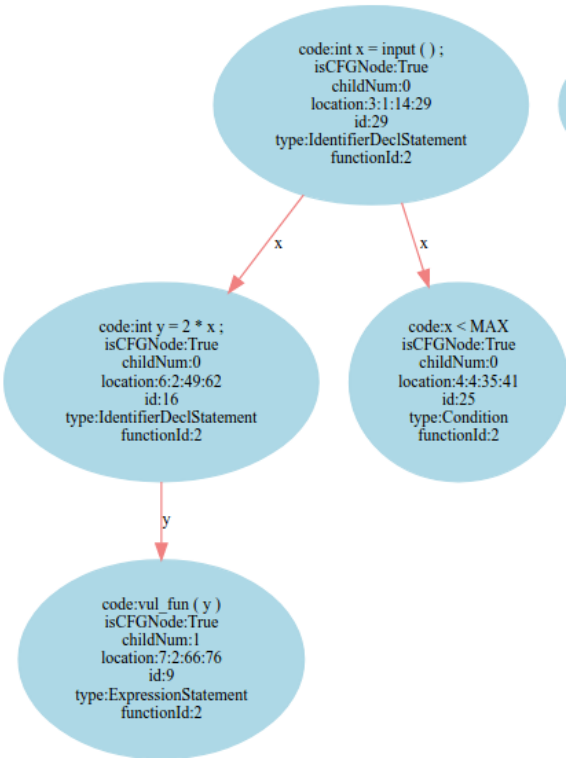


图 3.5　数据依赖性质图实例

除了根据节点搜索边，根据边查找源节点和目的节点是另外一个基本的 *traversal*，如下式所示。

$$Src(X) = \{s(e) : e \in X\}$$

$$Dst(X) = \{d(e) : e \in X\}$$

在搜索具体的代码元素时，需要一些定义一些基本的 *traversal*。给定一个抽象语法树节点，搜索此节点的所有抽象语法子树节点, 本章将其定义为 *TNode*, 如下式所示，其中 $Out_l(\{v\})$ 表示查找 υ 的子节点。

$$TNodes(X) = \bigcup_{\upsilon \in X} (\upsilon \cup (\bigcup_{\upsilon_c \in Out_l(\{v\})} TNodes(\{\upsilon_c\})))$$

给定一个抽象语法树节点，搜索包裹的语句节点也是一个基本需求。例如已知一个 $type = Array$ 的抽象语法树节点，找到包裹这个节点的语句，然后才能进一步根据语句节点进行控制流分析，其 *traversal* 如下式所示。

$$Statement(X) = \bigcup s(\{\upsilon\})$$

$$\text{其中，} s(X)=\begin{cases} X_1, & \text{如果}\mu(X_1, type)=Stmt \\ s(In_l(X)), & \text{其他} \end{cases}$$

traversal 可以使用复合函数符号 ∘ 组合使用。如首先搜索所有的数组节点（*type* = *Array* 的节点），然后获取这个包裹此节点的语句节点，这个过程可以表示为 $Statement \circ Filter_{\mu(v,type)=Array}(X)$。

3.2 基于程序性质图的源代码漏洞挖掘方法研究

程序性质图里包含了源代码静态分析的所有信息，本小节主要阐述程序性质图的遍历方式（*traversal*），以及如何定制和组合程序性质图 *traversal* 粗定位三种源代码软件漏洞：缓冲区溢出漏洞、格式化字符串漏洞、Use After Free（UAF）漏洞。

3.2.1 缓冲区溢出漏洞挖掘方法

在程序开发中缓冲区是指在栈或者堆上的一段内存区域，在源代码上表示为一个指向内存的指针或者一个数组变量。在源代码中，缓冲区的写操作有两种形式：（1）内存拷贝函数的调用，如 *strcpy*，*strncpy*，*memcpy*，*strcat* 等；（2）数组赋值或者指针解引用赋值；对于内存拷贝函数，如果拷贝的内存大小受输入控制且超过了分配的内存就会造成缓冲区溢出。对于数组赋值或者指针解引用赋值，如果出现在循环内部，在循环控制变量很大或者受输入控制也很容易造成缓冲区溢出。非循环的写操作也可以造成缓冲区溢出，但相对来说溢出的可能性较小。综上，缓冲区溢出漏洞的判定可总结为三个条件：（1）缓冲区的循环写或内存拷贝操作；（2）循环写或内存拷贝操作输入可控；（3）没有边界检测。

内存拷贝函数用 MCF（Memory Copy Functions）表示，可以分成分两类：第一种包含三个参数，第一个参数是拷贝的目的内存地址，第二个参数是被拷贝的源数据地址，第三个参数是拷贝内存的数量，如 *strncpy*，*memcpy* 等；另外一种包含前两个参数，不限制拷贝的内存数量，如 *strcpy* 等。对于缓冲区循环写 BLW（Buffer Loop Write），在一个函数中判断其是否存在需要两个条件：（1）赋值语句的左半部分是数组索引或者是指针的解引用；（2）赋值语句必须要包含在循环里。对于缓冲区写操作本章统称为 BOSink（Buffer Overflow Sinks），所以有 $BOSink = MCF \cup BLW$。另外，编程人员自己定义的和内存拷贝函数也需要考虑在内。例如，CVE-2016-9537 [105] 是一个缓冲区溢出漏洞，溢出的原因是自定义的内存拷贝函数 _TIFFmemcpy 没有进行正确的边界检测。此类函数其参数的含义和数量和传统的内存拷贝函数相似，所以这种自定义函数可以和传统的统一进行处理。

输入可控 IC（Input Control）是指 BOSink 中的关键变量和输入变量有数据依赖关系。输入变量可以是函数的参数，也可以通过命令行、环境变量、文件输入或者网络传入的变量。对于包含三个参数的内存拷贝函数如 *strcnpy*，需要第二和第三个参数输入可控；对于两个参数的内存拷贝函数如 *strcpy*，只需要第二个参数输入可控。内存拷贝函数的输入可控表示为 ICMCF（Input Control of Memory Copy Functions）。对于缓冲区的循环写入则要求赋值语句的右操作数、数组索引变量或者循环控制变量输入可控，表示为 ICBLW（Input Control of Buffer Loop Write）。所以输入可控可以表示为 $IC = ICMCF \cup ICBLW$

缓冲区溢出的边界检测 San（Sanitization）包含四种表现形式。

（1）关系表达式中包含内存拷贝函数的长度变量或者表达式，对于 Listing 3.4中第 5 行的 *memcpy* 函数，第 4 行的边界检测就属于此类，表示为 LS（Length Sanitization）。

（2）关系表达式中包含内存拷贝函数长度变量所依赖的变量，对于 Listing 3.4中第 6 行的 *strncpy*，表示为 LDS（Length Dependency Sanitization）。

（3）关系表达式中包含目的缓冲区变量，对于 Listing 3.4中第 9 行的 *memcpy* 函数，第 8 行的边界检测属于此类，表示为 BVS（Buffer Variable Sanitization）。

(4) 对于缓冲区循环写，关系表达式中包含循环控制变量，如 Listing 3.4中第 13 行的数组写，第 11 行的边界检测属于此类，表示为 LCVS（Loop Control Variable Sanitization）。

综合以上四种情况，缓冲区溢出边界检测可表示为 $San = LS \cup LDS \cup BVS \cup LCVS$。以上四种边界检测判断条件是非常严格的，能够滤除大部分的误报，但同是也一定会引起漏报。

Listing 3.4　缓冲区溢出边界检测示例代码

```
1  void foo(char *p, int length, int padding)
2  {
3        char dst[255], char dst1[255];
4        if(length + padding < 255){
5              memcpy(p, dst, length + padding);
6              strncpy(dst, p, length);
7        }
8        if(sizeof(dst) < 255){
9              memcpy(p, dst, length);
10       }
11       if(length<255){
12             for(int i=0; i<length; i++){
13                   dst[i] = 1
14             }
15       }
16 }
```

下面就基于上述的三个条件给出缓冲区溢出基于程序性质图特性的形式化定义。

定义 3.7: 缓冲区溢出漏洞可以用一个三元组表示 $(IC, BOSink, San)$，其中 $IC = ICMCF \cup ICBLW$ 表示程序的外部输入，$BOSink = MCF \cup BLW$ 表示缓冲区溢出在源代码上的表示形式，$San = LS \cup LDS \cup BVS \cup LCVS$ 表示边界检测的四种语法描述。

程序性质图中承载了所有的检测缓冲区溢出漏洞需要的信息，通过组合不同的 traversal 可以实现在性质图上检测缓冲区溢出漏洞。基于程序性质图检测由内存拷贝函数引起的缓冲区溢出漏洞的过程如算法3.1所示。在检测缓冲器溢出漏洞之前，源代码程序首先被解析并生成程序性质图。算法的输入是一个配置文件，用来记录所要检测的内存拷贝函数名字以及关键的参数次序，返回的是可疑的缓冲区溢出函数已经相应的缓冲区变量，用 BOs 表示。算法的具体执行过程如下:

(1) 从配置文件中导入所要检测的内存拷贝函数名字以及关键的参数次序（第 2 行）;

(2) 算法设定两个标识变量 flag1 和 flag2，分别用来判断内存拷贝函数的参数输入可控以及缓冲区无边界检测 (第 3 行);

(3) 对于每一个内存拷贝函数获取相应的函数调用 AST 节点 callExpr 和对应的参数 AST 节点 args（第 5、6 行），并且根据 args 获取语句节点 statement，此 statement 和一样源代码对应，此节点同时也是 CFG 节点（isCFGNode 属性为 true）;

(4) 对于每一个参数获取参数表达式的符号节点，例如参数是一个表达式 a+b，则获取的符号是 a 和 b（第 10 行）;

(5) 判断是否所有的符号都不包含在从输入函数到当前语句的数据流上（第 11 行），如果判断成立则说明此参数非输入可控，此时 flag1 设为 false（第 11-12 行）;

(6) 根据当前语句节点，沿着控制流图逆向搜索所有的 condition 节点，对于每一个 condition 节点，判断是否有 symbol 是 condition 的子表达式，如果有说明存在边界检测（第 14-17 行）;

(7) 根据 flag1 和 flag2 判断内存拷贝函数是否是为缓冲区溢出漏洞，并将当前函数的 functionId 和 location 加入到 BOs（第 21-22 行）。

算法 3.1 内存拷贝函数缓冲区溢出漏洞检测算法

Input: 内存拷贝函数与对应的参数的配置文件 conf
Output: 缓冲区溢出漏洞可疑函数及缓冲区变量 BOs

```
1:  BOs ← ∅
2:  MCFAndArgs = loadMCFAndArgs(conf)
3:  flag1 ← true,flag2 ← true
4:  for MCF in MCFAndArgs do
5:      MCFCallExprs = findCallASTNode(MCF)
6:      for callExpr in MCFCallExprs do
7:          statement = findStatementFromChildASTNode(callExpr)
8:          args = findKeyArgsFromCallExpr(callExpr)
9:          for arg in args do
10:             symbols = usedSymbols(arg)
11:             if (not DDGEdgeConoveySymbol(symbols)) or not DDGBegin-
                WithInputFunctions(statement)) then
12:                 flag1 = false
13:             end if
14:             conditions = obtainConditionFromIncomingCFG(statement)
15:             for condition in conditions do
16:                 if isInteract(condition, symbols) then
17:                     flag2 = false
18:                 end if
19:             end for
20:         end for
21:         if (flag1 and flag2) then
22:             BOs.add(statement.functionId, statement.location)
23:         end if
24:         flag1 = true, flag2 = true
25:     end for
26: end for
27: return BOs
```

算法3.2是检测因为缓冲区循环写引起的缓冲区溢出漏洞的过程。此类缓冲区溢出漏洞的检测没有输入，同样输出缓冲区溢出漏洞可疑函数及缓冲区变量，算法的具体执行过程如下：

(1) 对于指针解引用造成的缓冲区循环写，首先获取所有的指针解引用 AST 节点（第 2 行）；

(2) 获取指针解引用节点对应的缓冲区变量 bufferVar 以及此节点对应的语句 AST 节点 statement；

(3) 判断当前语句是否是赋值语句，如果是则通过 CFG 边逆向分析获去控制流节点，判断当前语句节点的控制流是否由循环语句节点流入（第 6 行）；

(4) 根据当前语句节点，沿着控制流图逆向搜索所有的 condition 节点，对于每一个 condition 节点，判断 bufferVar 是否为 condition 的子表达式，如果有说明存在边界检测；

(5) 根据 flag1 和 flag2 判断指针解引用是否是为缓冲区溢出漏洞（第 21-22 行），如果 flag1 和 flag2 都是 true，则将当前的函数和缓冲区变量加入到 BOs 中；

(6) 对于数组赋值形式的缓冲区循环写，首先获取所有的数组索引 AST 节点 arrays（第 23 行），形如”buf[i]“的都在此列；

(7) 对于每一个索引节点，获取当前的语句 AST 节点 statement，继而判断此语句节点是否为赋值语句且是否包含在循环内（第 25-26 行），如果成立，则获取数组的索引表达式节点 indexField（第 27 行），进而获取 indexField 使用的所有的符号；例如对于一个在循环内的语句节点”buf[a+b]=1”，indexFiled 节点为 a+b，symbols 为 [a,b]；

(8) 对于每一个 symbol，判断是否所有的符号都不包含在从输入函数到当前语句的数据流上（第 11 行），如果成立则说明此参数非输入可控，flag1 设为 false（第 30-31 行）；

(9) 获取数组变量，根据当前语句节点，沿着控制流图逆向搜索所有的 condition 节点（第 34-35 行）；

(10) 沿着控制流图逆向搜索所有的 condition 节点，对于每一个 condition 节点，判断 bufferVar 是否为 condition 的子表达式或者 symbol 是否为 condition 的子表达式，如果有说明存在边界检测，此时设定 flag2 为 false（第 36-38 行）；

(11) 根据 flag1 和 flag2 判断内存拷贝函数是否是为缓冲区溢出漏洞，并将当前函数的 functionId 和 location 加入到 BOs（第 41-42 行）。

Algorithm 3.2 缓冲区循环写缓冲区溢出漏洞检测算法

Input: 无

Output: 缓冲区溢出漏洞可疑函数及可疑区域 BOs

```
1: BOs ← ∅
2: dereferences = findDereferenceASTNode()
3: for dereference in dereferences do
4:         bufferVar = obtainBufferVar(dereference)
5:         statement = findStatementFromChildASTNode(dereference)
6:         if (isAssignment(statement) and inLoop(statement)) then
7:                 if (not DDGEdgeConoveySymbol(bufferVar)) or (not DDGBeginWith-
                   InputFunctions(bufferVar)) then
8:                         flag1 = false
9:                 end if
10:                symbols = usedSymbols(dereference)
11:                conditions = obtainConditionFromIncomingCFG(statement)
12:                for condition in conditions do
13:                        if isInteract(condition, symbols) then
14:                                flag2 = false
15:                        end if
16:                end for
17:                if (flag1 and flag2) then
18:                        BOs.add(statement)
19:                end if
20:                flag1 = true, flag2 = true
21:        end if
22: end for
23: arrays = findArrayASTNode()
24: for array in arrays do
25:        statement = findStatementFromChildASTNode(dereference)
26:        if (isAssig(statement) and inLoop(statement)) then
27:                indexField = obtainLength(array)
28:                symbols = usedSymbols(indexField)
29:                for symbol in symbols do
30:                        if (not DDGEdgeConoveySymbol(symbol)) or (not DDGBegin-
                           WithInputFunctions(statement)) then
31:                                flag1 = false
32:                        end if
```

```
33:                 end for
34:                 bufferVar = obtainBufferVar(array)
35:                 conditions = obtainConditionFromIncomingCFG(statement)
36:                 for condition in conditions do
37:                     if isInteract(condition, symbols) or isInteract(condition, bufferVar)
                        then
38:                         flag2 = false
39:                     end if
40:                 end for
41:                 if (flag1 and flag2) then
42:                     BOs.add(statement.functionId, statement.location)
43:                 end if
44:                 flag1 = true, flag2 = true
45:             end if
46:  end for
47:  return BOs
```

3.2.2 格式化字符串漏洞挖掘方法

格式化函数是一系列标准库函数, 比如 *printf 系列函数。这些函数接受可变数量的参数, 其中一个参数作为其他参数的格式化描述, 被称为格式化字符串参数。格式化函数执行时对格式化字符串参数进行解释, 并根据其格式描述使用其它的参数填充对应的描述字符串, 形成最终的输出。软件编写过程中一般会大量使用这类函数, 用于输出、处理字符串等。但是, 当格式化字符串与其他输入参数不相符时, 就会出现意外情况, 比如栈数据泄露, 任意内存写等。格式化字符串漏洞一般是指: 格式化字符串参数受外部输入影响, 从而精心构造的输入可能窃取栈上数据或向内存任意位置写入数据。软件中一般会有很多位置使用格式化函数, 如果将他们都直接标记出来, 则误报率将会变得相当高, 从而导致整个分析过程无效。因此, 需要做进一步的分析, 尽可能地提高分析精度。

当调用格式化函数时, 格式化字符串参数的来源可分为两种类型: 字符串常量类型、变量类型。通过分析漏洞的形成机理和已有的格式化字符串漏洞发现: 参数为字符串常量的调用链一般不会出现问题, 受输入控制的变量类型则都有可能导致格式化字符串漏洞。输入可控的定义和缓冲区溢出输入可控的定义相同，表示输入函数或者当前函数的参数和格式化函数的参数存在数据依赖关系。基于程序性质图的格式化字符串漏洞的挖掘如算法3.3所示，算法的输出是格式化字符串漏洞可疑函数及可疑区域。格式化字符串漏洞可疑区域是指危险函数中 *printf 函数

调用的行号，此信息由节点的 *location* 性质表示。算法3.3和算法3.1具有很大的相似性。不同的是算法3.3的输入配置文件是格式化函数和格式化参数的次序组成，例如对于 printf 函数，其第一个参数是格式化参数，可用一个二元组 <printf,0> 表示。算法3.3的具体执行过程如下：

(1) 从配置文件导出相应的格式化函数和格式化参数的次序（第 2 行）；

(2) 对于每一个格式化函数 fmtFun，在性质图数据库中搜索所有的名为 fmtFun 的函数调用 AST 节点 fmtcallExprs，针对每一个 fmtcallExpr 获取语句节点 statement 并获取参数节点 arg（第 3-7 行）；

(3) 判断 arg 是否包含在从输入函数到当前语句的数据流上，如果判断成立则将当前函数的 functionId 和 location 加入到 BOs；

算法 3.3 格式化字符串漏洞检测算法

Input: 格式化函数与对应的参数的配置文件 conf
Output: 格式化字符串漏洞可疑函数及可疑区域 FSVs

```
1:  FSVs ← ∅
2:  fmtFunAndArgs = loadFmtFunAndArgs(conf)
3:  for fmtFun in FmtFunAndArgs do
4:      fmtCallExprs = findCallASTNode(fmtFun)
5:      for callExpr in fmtCallExprs do
6:          statement = findStatementFromChildASTNode(callExpr)
7:          arg = findFmtArgFromCallExpr(callExpr)
8:          if (DDGEdgeConoveySymbol(arg)) or DDGBeginWithInputFunc-
            tions(statement)) then
9:              FSVs.add(statement.functionId, statement.location)
10:         end if
11:     end for
12: end for
13: return FSVs
```

3.2.3 Use After Free 漏洞挖掘方法

UAF(Use After Free) 漏洞属于指针相关的漏洞, 其主要表现形式是引用已经被释放的内存对象。UAF 漏洞可能导致程序行为异常, 经过精心构造的攻击代码可以利用 UAF 漏洞实现任意代码执行。根据 UAF 漏洞的表现形式, 可以分析在内存释放函数 (如 free、delete 等等) 调用后, 被释放的内存对象指针是否任然“存活”,

即是否存在后续指令继续使用该指针的行为, 如果发现被释放的内存对象指针任然“存活”, 则报告一个可疑 UAF 漏洞。要实现上述过程需要分析每一个语句使用的符号和内存释放是否有控制依赖关系，而每个语句节点使用的符号可以由程序符号图 PSG（Program Symbol Graph）表示。

定义 3.8: 程序符号图 $G_{PS} = (V_{PS}, E_{PS}, \lambda_{PS}, \mu_{PS}, s, d)$，$V_{PS}$ 是所有语句节点和标识符节点 ($\mu(type) = identifier$) 的集合，E_{PS} 连接语句节点和语句使用的符号，$\lambda : E \rightarrow \sum_{PS}$ 是边标记函数，其中 $\sum_{PS} = \{use\}$。 ■

通过程序符号图引入程序性质图，可以得出基于程序性质图的 UAF 检测算法，如算法3.4所示。该算法返回的是 free 函数调用的位置、free 的变量再次使用的语句以及当前函数具体的执行过程如下:

(1) 初始化返回值以及内存释放函数的名称（第 1 行），在性质图库中查询所有的内存释放函数的调用语句（第 2 行）;

(2) 针对内存释放函数调用语句 freeCall，获取所释放内存对应的变量 freeVar（第 4 行），然后在本函数中获取所有控制依赖于 freeCall 的语句 statements（第 5 行）;

(3) 对于每一个语句 statement，根据符号性质图获取 statement 使用的所有符号 symbols（第 7 行），判断 symbols 是否包含 freeVar，如果为 true 说明可能存在一个 UAF 漏洞，则将 freeCall 的位置，当前函数 Id 以及 freeVar 再次使用的语句位置加入到 UAFs 中（第 9 行）。

3.2.4 实验结果与分析

3.2.4.1 实验设计

（1）实验目的

本实验的目的是验证基于程序性质图的源代码软件漏洞挖掘方法在挖掘缓冲区溢出漏洞、格式化字符串漏洞以及 Use After Free 漏洞上的有效性。

（2）实验环境与过程

本节在 poppler0.10.6、a2ps4.14 以及 libxml2-2.9.3 三个软件上分别对缓冲区溢出漏洞检测算法、格式化字符串漏洞检测算法以及 Use After Free 漏洞检测算法进行测试。实验采用的源代码语法解析器 ANTLR[101]，图数据库为 Neo4J[106]，图查询语言为 Gremlin[107]，漏洞挖掘脚本语言为 python。实验的采用的硬件环境是 Inter Xeon CPU E3-1231 v3 @ 3.40GHz，16G RAM。

算法 3.4 UAF 检测算法

Input: 无

Output: 内存释放函数调用语句，被 free 的变量再次使用的语句以及当前函数 UAFs

```
1:  UAFs ← ∅,freeCallStr ←{"free", "delete"}
2:  freeCallExprs = findCallASTNode(freeCallStr)
3:  for freeCall in freeCallExprs do
4:      freeVar = obtainFreeVar(freeCall)
5:      statements = findStatementAlongCFG(freeCall)
6:      for statement in statements do
7:          symbols = statement.use()
8:          if symbols.contains(freeVar) then
9:              UAFS.add(freeCall.location, statement.location, state-
                ment.functionId,)
10:         end if
11:     end for
12: end for
13: return UAFs
```

在漏洞挖掘之前，需要将源代码转化成程序性质图，然后使用算法3.1、算法3.2挖掘缓冲区溢出漏洞、使用算法3.3挖掘格式化字符串漏洞、使用算法3.4挖掘 Use After Free 漏洞。

3.2.4.2 结果分析

（1）缓冲区溢出漏洞实验

本实验测试的软件是 pdf 渲染库 poppler0.10.6。本实验使用了算法3.1与算法3.2，发现了四个缓冲区溢出漏洞，漏洞对应的文件名，漏洞所在的函数名已经测出的漏洞如表3.1所示。除了发现的漏洞，本实验还产生了 3 个误报，误报率为 42.9%。通过调查可知，poppler0.10.6 包含 11 个缓冲区溢出漏洞，漏报率为 63.6%。造成本实验漏报率较高的原因是实际缓冲区溢出漏洞形成原因较为复杂，通过固定的模式很难发现较为复杂的漏洞，因此本章在节3.3中提出了一种基于机器学习的缓冲区溢出漏洞挖掘方法，能够以较低的误报率挖掘漏洞。

（2）格式化字符串挖掘实验

本实验用于评估算法3.3，实验的软件是 GNU a2ps4.14[108]。a2ps4.14 是一个将各种格式文件转化成 PostScript 的转换器，支持各种字体且允许用户自定义输出格式。运行算法3.3输出的结果如表3.2所示。算法输出漏洞函数所在文件、函数名以及格式化字符串函数名称。从表中可以看出，本实验在 a2ps4.14 中发现了一个格式化字符串漏洞。在 lib 目录下 dstring.c 文件的第 326 行，格式化字符串调用

表 3.1　缓冲区溢出漏洞实验

文件名	函数名	是否漏洞	CVE ID
poppler/Function.cc	PostScriptFunction	N	N/A
poppler/Function.cc	ExponentialFunction	Y	CVE-2015-8868
fofi/FoFiTureType.cc	cvtSfnts	N	N/A
splash/Splash.cc	transformDataUnit	Y	CVE-2013-1788
fofi/FoFiType1.cc	parse	Y	CVE-2010-3704
fofi/FoFiType1C.cc	readFDSelect	N	N/A
splash/Splash.cc	drawImage	Y	CVE-2009-3604

语句为 vsprintf (ds->content + ds->len, format, args)；其格式化字符串参数与函数参数有数据依赖关系，通过查询 CVE 库发现，其对应的漏洞是 CVE-2015-8107。本实验的误报率为 87.5%，但是 a2ps4.14 中包含 484 个格式化字符串函数调用，所以总体上本实验大大缩小了需要关注的代码范围。

表 3.2　格式化字符串漏洞实验

文件名	函数名	格式化字符串函数	是否漏洞
/lib/printlen.c	printflen	vprintflen	N
/lib/document.c	documentation_print_plain	fprintf	N
/lib/document.c	documentation_print_texinfo	fprintf	N
/lib/dstring.c	ds_vsprintf	vsprintf	N
/lib/dstring.c	ds_unsafe_sprintf	ds_unsafe_vsprintf	N
/lib/dstring.c	ds_cat_vsprintf	vsprintf	N
/lib/dstring.c	ds_unsafe_vsprintf	vsprintf	N
/lib/dstring.c	ds_unsafe_cat_vsprintf	vsprintf	Y

（3）Use After Free 漏洞挖掘实验

本实验测试的软件是 libxml2-2.9.3，libxml2 是一个 xml c 语言版的解析器，本为 Gnome 项目开发的工具，是一个基于 MIT License 的免费开源软件。它除了支持 c 语言版以外，还支持 c++、PHP、Pascal、Ruby、Tcl 等语言的绑定，能在 Windows、Linux、Solaris、MacOsX 等平台上运行。表3.3是算法3.4运行在 libxml2-2.9.3 上的实验结果。算法共检测了 2 个 Use After Free 漏洞对应的 CVE 编号分别为 CVE-2016-1837 和 CVE-2016-1835。同时也产生了 13 个误报，误报率为 86.7%。虽然误报率较高，但是 libxml2-2.9.3 中包含 245 个内存释放，所以总体上算法3.4大大缩小了需要关注的代码范围。

表 3.3　Use After Free 漏洞挖掘实验

文件名	函数名	是否漏洞	CVE-ID
tree.c	xmlNodeGetBase	N	N/A
xmlschemas.c	xmlSchemaGetCanonValueWhtspExt	N	N/A
HTMLparser.c	htmlParseFile	N	N/A
HTMLparser.c	htmlFreeParserCtxt	N	N/A
HTMLparser.c	htmlParseSystemLiteral	Y	CVE-2016-1837
parse.c	xmlIOParseDTD	N	N/A
parse.c	xmlNewBlanksWrapperInputStream	N	N/A
parse.c	xmlParseStartTag2	Y	CVE-2016-1835
relaxng.c	xmlRelaxNGNewStates	N	N/A
relaxng.c	xmlRelaxNGNewValidCtxt	N	N/A
relaxng.c	xmlRelaxNGValidateDatatype	N	N/A
relaxng.c	xmlRelaxNGFreeDefine	N	N/A
relaxng.c	xmlRelaxNGNewDocParserCtxt	N	N/A
runtest.c	pushParseTest	N	N/A
xzlib.c	xz_head	N	N/A

3.3　基于机器学习的缓冲区溢出漏洞挖掘方法研究

第3.2.1节中仅仅考虑了缓冲区写的表现形式和边界检测两个缓冲区溢出的直接因素，同时对缓冲区溢出检测方法执行了非常严格的边界检测策略，即缓冲区大小控制变量不存在任何的条件语句当中，此策略能够非常精确的检测缓冲区溢出漏洞，但同时会造成很多的漏报。在程序的编写过程中，造成缓冲区溢出的因素不仅仅这些，Kratkiewicz[109] 将缓冲区溢出漏洞根据不同程序静态特征分成 22 类。程序静态特征被广泛的应用在建立程序的抽象模型，本小节在程序静态特征的基础上，提出了一种基于机器学习的缓冲区漏洞挖掘方法方法。

3.3.1　基本框架

基于机器学习的缓冲区漏洞挖掘方法的基本思想是通过已存在的漏洞（本章中如无特别说明，漏洞一词即代表缓冲区溢出漏洞）去辅助挖掘新的漏洞。本节主要介绍了方法的流程以及各个模块的具体组成和作用。

基于机器学习的缓冲区溢出漏洞挖掘方法的框架包括两个流程相似的过程：训练和分类，如图3.6所示。对于训练流程，首先在 CVE 漏洞库中手动的查找缓冲区溢出漏洞并标记，通过鲁棒语法分析器对这些代码进行语法分析，然后基于语法分析结果生成扩展的程序性质图并存入性质图数据库，其中扩展程序性质图

是将过程间边界检测性质图、函数调用性质图并入程序性质图而形成的；然后利用扩展性质图遍历对程序的缓冲区溢出漏洞静态特征进行提取，最后将提取出来的静态特征向量化并利用多种机器学习算法训练分类器。相似地，对于测试的源代码也需要经历这几个过程，不同的是在提取缓冲区漏洞静态特征并向量化之后，将这些向量数据输入到训练产生的分类器中进行分类，最后生成可疑函数和可疑缓冲区。这里的可疑函数指指被分类器判断包含可疑缓冲区的函数，可疑缓冲区是指可疑函数中代表栈或者堆上的一段内存区域的变量。

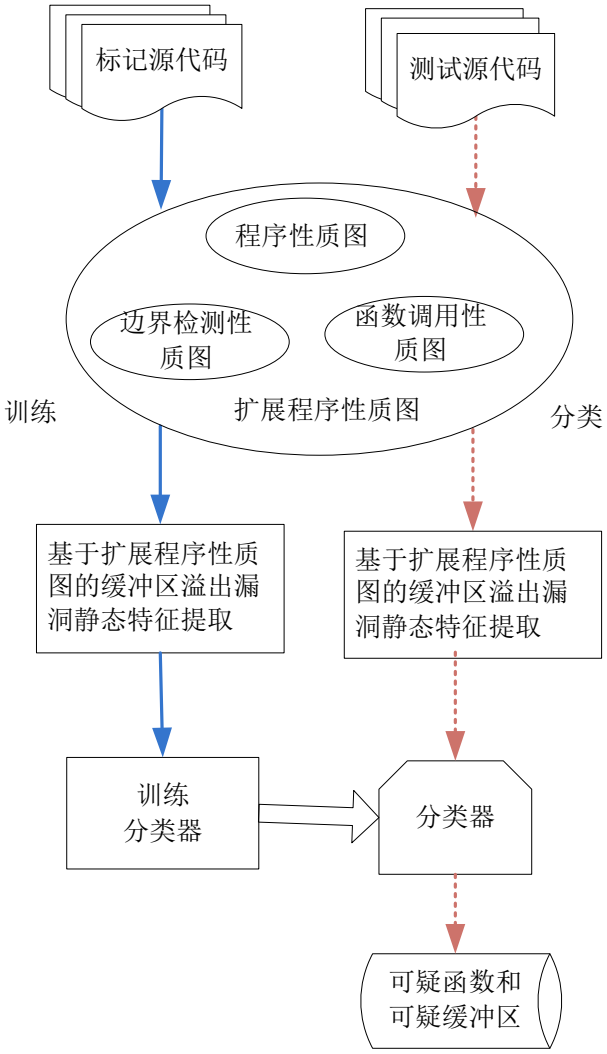


图 3.6　基于机器学习的缓冲区溢出漏洞挖掘框架

3.3.2　程序静态属性与映射规则

通过分析和总结 Kratkiewicz[109] 提出的缓冲区溢出漏洞 22 种分类方法，本小节将缓冲区溢出漏洞的静态属性分为 7 类，分别为 sink 类型、缓冲区位置、索引 /

地址 / 长度复杂度、边界检测、循环 / 条件 / 函数调用深度以及是否输入可控。后续内容详细介绍了各类程序静态属性以及将程序静态属性映射到数字向量空间的规则。

3.3.2.1 sink 类型

缓冲区溢出漏洞包含三种 sink 类型：指针解引用（pointer dereference）、数组写 dangerous function 以及危险函数（dangerous function），如表3.4所示。如果函数中的一条语句符合三种 sink 类型的任意一条而又没有进行缓冲区边界检测，则程序就有可能发生缓冲区溢出。在 C/C++ 中，数组的元素即可以通过指针解引用也可以用数组下标访问，但是对同一个缓冲区变量进行访问时很少混合使用，所以这里将二者分成两种类型。例如 Listing 3.5中的第 24、25 行的变量 src 和 dst 就属于指针解引用这一类。危险函数是指像 memcpy，strncpy 等的内存函数拷贝或者字符串拼接函数，或者用户定义的具有相似功能的函数，例如程序 Listing 3.5中第 15、16、17 行中的用户定义函数“_TIFFmemcpy”，此函数和标准库函数 memcpy 的作用相同。如果一个用户定义的函数和标准库函数的参数个数相同、功能相近，则亦将此函数归为危险函数一类。另外某些格式化字符串函数也可能引起缓冲区溢出漏洞，但在本节中不予考虑。

表 3.4　缓冲区溢出漏洞三种 sink 类型

Sink Type	Example	Mapping Value
pointer dereference	*p++ = 1	1
array write	p[i] = 1	2
dangerous function	strcpy(dst, src), strncpy(dst, src, n) strcat(dst, src), strncat(dst, src, n) memcpy(dst, src, n), memmove(dst, src, n) gets(str), fgets(str, n, fp)	3

Listing 3.5　CVE-2016-9537 示例代码

```
1  static int reverseSamplesBytes (uint16 spp, uint16 bps, uint32
       width, uint8 *src, uint8 *dst)
2  {
3   int i;
4   uint32 col, bytes_per_pixel, col_offset;
5   uint8 bytebuff1;
6   unsigned char swapbuff[32];
7   if ((src == NULL) || (dst == NULL)){
8    TIFFError("reverseSamplesBytes","Invalid input or output
       buffer");
9    return (1);
10  }
11  bytes_per_pixel = ((bps * spp) + 7) / 8;
```

```
12   switch (bps / 8){...
13    case 2: for (col = 0; col < (width / 2); col++){
14      col_offset = col * bytes_per_pixel;
15      _TIFFmemcpy (swapbuff, src + col_offset, bytes_per_pixel);
16      _TIFFmemcpy (src + col_offset, dst - col_offset -
           bytes_per_pixel, bytes_per_pixel);
17      _TIFFmemcpy (dst - col_offset - bytes_per_pixel, swapbuff,
            bytes_per_pixel);
18     }
19    break;
20    case 1: /* Use byte copy only for single byte per sample
         data */
21     for (col = 0; col < (width / 2); col++){
22      for (i = 0; i < spp; i++){
23       bytebuff1 = *src;
24       *src++ = *(dst - spp + i);
25       *(dst - spp + i) = bytebuff1;
26      }
27      dst -= spp;
28     }
29  ...}
```

3.3.2.2 缓冲区位置

缓冲区位置属性（memory location）描述的是缓冲区内存分布的位置，即栈、堆、数据段、BSS 段与共享内存五种位置。从编程角度上，这五种内存位置的初始化方式是不相同的。局部非静态变量定义在栈上，通过 malloc 函数动态分配的缓冲区在堆上，数据段上存储的是全局初始化变量，BSS 段上存储的是未初始化的全局或者静态变量，共享内存是特别分配的映射进或者映射出程序地址空间，并且通过特殊的操作系统函数（如 Linux 中的 shmget,shmat,shmdt 和 shmctl）释放的内存区域。本节只考虑栈、堆和数据段三类缓冲区位置。三种内存位置被映射成一个三维向量表示为 (stack,heap,data segment)，提取静态属性时，根据缓冲区的初始化方式进行判断，如果相应的内存位置出现则将其赋为 1，否则赋为 0。例如 Listing 3.5中的指针变量 *swapbuff*，因为被声明为局部变量，所以对应的变量表示为 $(1,0,0)$。

3.3.2.3 容器

容器属性（container）描述的是缓冲区是否包含或者包含在一个什么样的容器当中。一般的，容器结构越复杂缓冲区的使用越容易出现错误。根据 Zitser[110] 统计，7% 的漏洞缓冲区包含在 *Union* 结构中，而在本节的训练集中，近 30% 的漏洞包含在不同的容器中，所以这里将容器也作为一个属性，容器属性的示例和映射规则规则如表3.5所示。这里将 *Union* 和 *Struct* 二者都映射成 2，是因为他们在编程上的相似性，*others* 表示更复杂的结构如多重 *Union* 或者 Struct 等。

表 3.5　容器属性

Instance	Container Type	Mapping Value
p[256]	none	0
p[256][256]	array	1
struct.p[256]	struct	2
union.p[256]	union	
ohters	others	3

3.3.2.4　索引 / 地址 / 长度复杂度

索引复杂度属性描述的是数组索引的复杂度，其引入是基于这样一个假设：对于索引值的操作越复杂，数组越容易发生溢出。索引复杂度可以分成六种：常量（constant）、加减法（addition）、乘除法（multiplication）、非线性（non-linear）、函数调用（function call）以及数组访问（array access），如表3.6所示。加减法操作包括类似加法和减法的操作，自增和自减操作也归入此类。乘除法操作包含乘法、除法以及逐位左移和右移等操作。非线性操作包含取模以及标准库的其他非线性函数例如 *pow()* 与 *sqrt()* 等。函数调用操作描述的是数组索引操作是否包含函数的返回值（除标准库的非线性函数）。数组访问操作表示数组索引操作是否包含数组访问。尽管很难被利用，常量索引还是有造成缓冲区溢出，所以也被列为索引操作之一。除了常量类型，其他的索引操作类型都有两种不同的呈现方式：p[i-8] 和 (p-8)[i]。

图3.7是获取复杂度的一个简单的示例。*i* 是数组 *p* 的索引变量，沿着数据流获取对 *i* 的操作形成一个六维向量 (0,1,0,0,1,1)。

表 3.6　索引操作类型

Operation Type	Instance
constant	p[256]
addition	p[i+8], p[i-8], (p+8)[i], (p-8)[i]
multiplication	p[i*8], p[i/8], p[i≫8], p[i≪8], (buf+8*n)[i]
non-linear	p[i%8], p[pow(i, j)], p[sqrt(i)]
function call	p[f(i)], (p+f(n))[i], (getAdtres(n))[i]
array access	p[buf[i]], (p+buf[n])[i], (buf[n])[i]

地址和长度复杂度属性和索引复杂度相似，其的示例和映射规则如表3.7和表3.8所示。地址和长度复杂度也被映射成一个六维向量，相应的操作每出现一次对应的值就加 1。对于 Listing 3.5第 24 行、25 行的缓冲区变量 *src* 和 *dst*，地址复

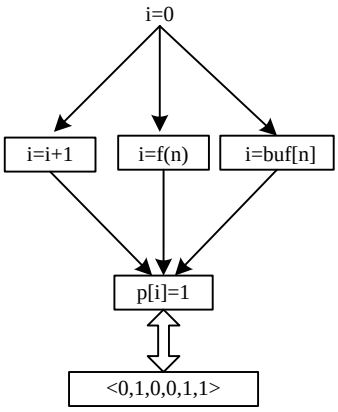


图 3.7　变量 i 复杂度的获取

杂度属性对应的向量为 (0,1,0,0,0,0) 和 (0,3,0,0,0,0)。对于危险函数 sink 类型的缓冲区变量 *swapbuff*，其长度复杂度向量为 (0,1,2,0,0,0)。另外，索引 / 地址 / 长度复杂度属性都需要将缓冲区别名考虑在内，即对缓冲区别名的操作需要类累加到相应的向量中。

表 3.7　地址操作类型

Operation Type	**Instance**
constant	not appliable
addition	*p(i+8), *p(i-8)
multiplication	*(p+i*8), *p(i/8), *p(i>>8), *p(i<< 8)
non-linear	*(p+i%8), (p+pow(i, j)), *(p+sqrt(i))
function call	*(p+f(i)), *(getAddress())
array access	*(p+buf[i])

表 3.8　长度操作类型

Operation Type	**Instance**
constant	memcpy (dest, src, 256)
addition	memcpy (dest, src, i+256)
multiplication	memcpy (dest, src, i*8)
non-linear	memcpy (dest, src, i%8)
function call	memcpy (dest, src, f(n))
array access	memcpy (dest, src, buf[255])

3.3.2.5 边界检测

虽然静态分析不能精确的捕捉缓冲区的边界检测（Sanitization）信息，但依然可以通过边界检测的模式去近似的估计。如果一条语句符合以下的几种模式，就可以近似的认为编程人员已经考虑了边界检测，这些模式出现的次数越多则说明编程人员添加边界检测的概率要更高。边界检测可以分为以下三种类型。

1 直接边界检测：假设 $P =< n_1, n_2, ..., n_N >$ 是 CFG 上的一条路径，$0 < i < N$。如果 n_j 是一个 sink 节点，n_i 是一个条件语句节点，若程序满足以下任一条件，则 n_i 是一个直接边界检测节点。（1）n_j 是一个数组写 sink，iv 是 n_j 的数组索引变量或表达式，如果 n_i 使用 iv；（2）n_j 是一个危险函数 sink，l 是 n_j 索要拷贝的长度变量或者表达式，如果 n_i 使用 l；（3）buf 是 n_j 的缓冲区变量且 n_i 使用了 buf。像 n_i 这样的节点在函数中每出现一次，直接边界检测的值加 1。例如 Listing 3.5的补丁程序 Listing 3.6，第 1 行的条件语句就是一个直接边界检测。

2 间接边界检测：此类边界检测只适用于数组写 sink 和危险函数 sink；假设 s 是一个语句节点，$Use(s) = \{v|s$ 是一个定义语句且使用 $v\}$，$DDUse(v)$ 表示所有 v 数据依赖的变量，即 $DDUse(v) = \bigcup_{v_c \in Use(v)} v_c \cup DDUse(v_c)$；对于 Listing 3.5第 15 行的长度变量 bytes_per_pixel，$DDUse(bytes_per_pixel) = bps, spp$；假设 $P =< n_1, n_2, ..., n_N >$ 是 CFG 上的一条路径，$0 < i < N$。如果 n_j 是一个 sink 节点，n_i 是一个条件语句节点，iv 是一个长度变量或者数组索引变量，n_i 使用 v 且 $v \in DDUse(iv)$，在 n_i 是一个间接边界检测节点。例如 Listing 3.7的第 1 行就是一个间接边界检测节点。

3 过程间检测：如果函数的参数与长度变量、数组索引或者缓冲区变量有数据依赖关系，且参数出现在上级调用函数的条件语句中，则称此条件语句是一个过程间边界检测；Listing 3.8是一个过程间边界检测的示例。

此外，一些条件语句虽然符合上面的描述，但不能算作为边界检测。例如第 7 行的条件语句不能被当做直接边界检测，因为将一个缓冲区变量和 NULL 做比较的作用是判断一个指针是否为 NULL。边界检测属性被映射成一个三维向量，每一类边界检测出现一次则相应的数值加 1。

Listing 3.6 CVE-2016-9537 补丁程序

```
1 if( bytes_per_pixel > sizeof(swapbuff) ){
2   TIFFError("reverseSamplesBytes","bytes_per_pixel too large");
3   return (1);
4 }
```

Listing 3.7　间接边界检测示例程序

```
if(i<256)
{
  buf[i+j] = 1;
}
```

Listing 3.8　过程间边界检测示例程序

```
void woo(int arg){
  if(arg<256)
  {
    foo(arg) ;
  }
}
void foo(int param)
{
  buf[param]=1;
}
```

3.3.2.6　循环 / 条件 / 函数调用深度

循环 / 条件 / 函数调用深度属性反映了程序的复杂度。其中，循环 / 条件深度表示包裹 sink 语句循环 / 条件层次，函数调用深度描述从入口函数到 sink 函数之间的函数调用数量，函数调用深度可以通过3.3.3引入的函数调用性质图获取。循环 / 条件 / 函数调用深度属性被映射成一个三维向量。对于 Listing 3.5的 sink 变量 *src*，其调用链为 $main \rightarrow createCroppedImage \rightarrow mirrorImage \rightarrow reverseSanplesBytes$，switch 语句在本节中也被当做条件语句，所以其向量为（2,1,4）。

3.3.2.7　输入可控

输入可控用于判断 sink 语句是否和输入有数据依赖关系，此属性的判定和3.2.1节相同。输入数据可以是函数的参数，也可以通过命令行、环境变量、文件输入或者网络传入的变量。如果 sink 语句输入可控，则将其设为 1，否则设为 0。

综上，将七类属性映射的向量串接得到一个 18 维向量，此向量将被用来训练分类器。

3.3.3　扩展的程序性质图

3.1.2节中介绍了由过程内的抽象语法树、控制流图以及数据依赖图融合而成的程序性质图。在程序性质图中，每个节点都拥有多个属性，节点之间通过各类边连接，每条边亦拥有多个属性。通过抽象语法树性质图可以获取源代码的所有操作数和操作符，所以 sink 类型和容器属性可以很容易的获取；获取缓冲区位置和输入可控属性需要抽象语法树和数据依赖图；条件 / 循环深度需要抽象语法树和控制流图；直接 / 间接边界检测以及索引 / 地址 / 长度复杂度需要三类性质图的结合。但是，程序性质图不能处理过程间边界检测和函数调用深度，因为二者的

获取需要过程间的数据依赖和控制依赖关系。为了解决这个问题，将过程间边界检测性质图和函数调用性质图并入到程序性质图形成扩展的程序性质图。

定义 3.9: 过程间边界检测性质图 $G = (V_A, E_{IP}, \lambda_{IP}, \mu_I P, s, d)$，$V_A$ 是抽象语法树节点，E_{IP} 连接条件语句节点和被调用函数的 sink 语句节点，λ_{IP}，$\lambda_{IP} : E_{IP} \rightarrow \sum_{IP}$，其中 $\sum_{IP} = \{IC, ID\}$，IC 表示过程间控制依赖，ID 表示过程间数据依赖，过程间数据依赖边被赋予一个 *symbol* 属性。

Listing 3.9是过程间边界检测和函数调用性质图解释程序。图3.8Listing 3.9的简化程序性质图，删除了部分抽象语法树节点和边，图3.9(a) 是 Listing 3.9的过程间边界检测性质图示意图。通过对比图3.8和图3.9，可以看出为了产生 E_{IP}，条件语句 *if(size<100)* 必须和函数具有数据和控制依赖关系且 *memcpy* 函数的参数 *src*，*count* 必须和 *foo* 函数的参数有数据依赖关系。

定义 3.10: 过程间边界检测性质图 $G = (V_A, E_{CC}, \lambda_{CC}, \mu_C C, s, d)$，$V_A$ 是抽象语法树节点，如果函数 f_1 调用 f_2，则 E_{IP} 连接两个函数定义节点，$\lambda_{IP} : E_{CC} \rightarrow \sum_{CC}$，其中 $\sum_{IP} = \{CC\}$，每条边被赋予一个 *symbol* 属性，用于表示函数之间传递的符号。

图3.9(b) 是 Listing 3.9的函数调用性质图示意图。函数调用性质图能获取函数调用深度，而且能获取影响 sink 语句的输入源。

Listing 3.9 过程间边界检测和函数调用性质图解释程序

```
1  int main(int argc, char **argv){
2      int i = atoi(argv[1]);
3      char *p = argv[2];
4      woo(p, i);
5  }
6  void woo(char* src, int size){
7      if(n<100){
8          foo(src, size);
9      }
10 }
11 void foo(char* src, int n){
12     char dst[200];
13     int count = n+100;
14     memcpy(dst, src, count);
15 }
```

将过程间边界检测性质图，函数调用性质图并入程序性质图形成扩展的程序性质图 $G = (V, E, \lambda, \mu, s, d)$，其中，

$$V = V_A$$

$$E = E_A \cup E_C \cup E_P \cup E_{IP} \cup E_{CC}$$

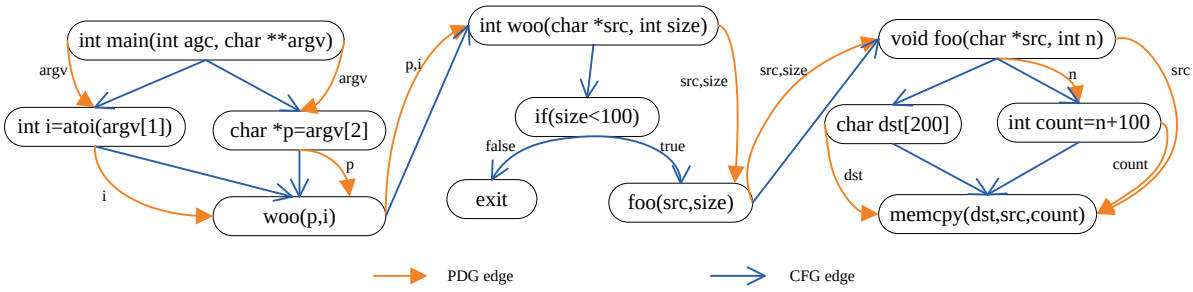


图 3.8　Listing 3.9的简化程序性质图

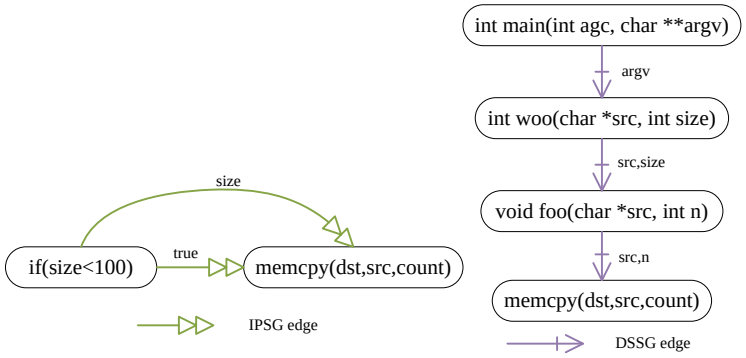


图 3.9　Listing 3.9过程间边界检测性质图和函数调用性质图

$$\lambda = \lambda_A \cup \lambda_C \cup \lambda_P \cup \lambda_{IP} \cup \lambda_{CC}$$

$$\mu = \mu_A \cup \mu_C \cup \mu_P \cup \mu_{IP} \cup \mu_{CC}$$

。通过组合3.1.3的遍历方式，获取缓冲区溢出漏洞静态属性。

3.3.4　实验结果与分析

3.3.4.1　实验设计

（1）实验目的与评估参数

通过实验以及和其他工具对比，验证基于机器学习的缓冲区溢出漏洞方法的有效性。方法的性能通过混淆矩阵展示。在机器学习领域，混淆矩阵[111] 经常被用来评判分类器的性能，如表3.9所示。混淆矩阵描述了实际类别和挖掘类别的交叉组合，即真正类（True Positive，TP），假负类（False Positive，FP），真负类（True Negative，TN）以及假负类（False Negative，FN）。在漏洞挖掘领域，FP 又被称为误报，表示一个非漏洞被分类器归为漏洞；FN 又称为漏报，表示一个漏洞被分类器归为非漏洞；TP 表示一个漏洞被分类器归为漏洞；TN 表示一个非漏洞被分类器归为漏洞。在一个正常的程序中，非漏洞函数要远比漏洞函数多，如此会造成训练和测试集的不均衡，所以再次添加了其他的衍生评估参数，即召回率（recall）同时也被称为真值率（True Positive Rate，TPR），真负率（True Negative

Rate，TNR），精度（precision）也被称为阳性预测值（Positive Predictive Value，PPV）以及 F_1 值，此四参数的计算如式3.1所示。

表 3.9　混淆矩阵

	Predicted Positive	**Actual Positive**
Actual Positive	True positive (TP)	False Negative (FN)
Actual Negative	False Positive(FP)	True Negative (TN)

$$\begin{aligned} recall &= TP/(TP+FN) \\ TNR &= TN/(FP+TN) \\ precision &= TP/(TP+FP) \\ F_1 &= 2\times(recall\times precision)/(recall+precision) \end{aligned} \tag{3.1}$$

（2）实验环境与过程

本方法选取了 8 个开源软件作为训练集，分别为 ffmpeg，HDF5，libtff，mupdf，openssl，qemu，zziplib 以及 blueZ，如表3.10所示。从这些软件中，结合 CVE 漏洞库手动审计了 58 个缓冲区溢出漏洞函数以及另外的 174 个非漏洞函数，对于所有的函数，都选择一个缓冲区代表函数提取静态属性。列 Vul-Num 和 Not-Vul-Num 分别表示漏洞函数和非漏洞函数数量。漏洞函数被标记为 1，非漏洞函数被标记为 0。

哪种分类器的性能更好是不可预知的，所以需要测试不同的分类器算法性能。本实验利用了五个著名的分类器：K 近邻法（K-Nearest Neighbors，KNN），决策树（Decision Tree，DT），朴素贝叶斯（Naive Bayes，NB），AdaBoost 以及支持向量机（Support Vector Machines，SVM）。为了这是的评估分类器性能，本实验采用了 10 重交叉验证。各分类器参数设定如下：KNN 中的参数 k 为 7，DT 算法用的是 C4.5，AdaBoost 使用的弱分类器的数量是 20，SVM 使用的是 RBF 核，参数 C=10，$\gamma = 0.01$。实验的采用的硬件环境是 Inter Xeon CPU E3-1231 v3 @ 3.40GHz，16G RAM。

实验过程如图3.6所示。首先，利用训练集，根据缓冲区溢出漏洞的静态特征训练出分类器；然后，从新的源代码中抽取缓冲区溢出漏洞静态特征并向量化，利用分类器挖掘缓冲区溢出漏洞。

3.3.4.2　结果分析

五个分类器的性能展示如表3.11所示。平均 recall 为 83.5%，即 58 个漏洞平均可以检测出 48 个。平均 TNR 是 87.3%，即 174 个非漏洞 152 个检测正确。平

表 3.10　实验数据列表

Program	CVE-ID	Vul-Num	Not-Vul-Num
ffmpeg	CVE-2016-7562, CVE-2016-6920, CVE-2016-10192, CVE-2016-10191, CVE-2016-10190, CVE-2016-8364, CVE-2014-5271, CVE-2014-3157, CVE-2014-2263, CVE-2013-0894, CVE-2013-0868, CVE-2013-0863, CVE-2012-0947, CVE-2012-0857, CVE-2012-0856, CVE-2012-0855, CVE-2012-0848, CVE-2012-0847	18	54
HDF5	CVE-2016-4333, CVE-2016-4330	2	6
libtiff	CVE-2017-5225, CVE-2016-9540, CVE-2016-9537, CVE-2016-9536, CVE-2016-9535, CVE-2016-9533, CVE-2016-5652, CVE-2016-5319, CVE-2016-5318, CVE-2016-5102, CVE-2016-3991, CVE-2016-3990, CVE-2016-3632, CVE-2016-3624, CVE-2015-8784, CVE-2015-8782, CVE-2013-4244, CVE-2013-4231	18	54
mupdf	CVE-2017-5869, CVE-2016-6525, CVE-2014-2013, CVE-2011-0341	4	12
openssl	CVE-2016-2182, CVE-2015-0235, CVE-2014-3512	3	9
qemu	CVE-2016-7170, CVE-2016-5238, CVE-2016-4439, CVE-2013-4151, CVE-2013-4150	5	15
zziplib	CVE-2017-5976, CVE-2017-5975, CVE-2017-5974, CVE-2017-1614	4	12
BlueZ	CVE-2016-9917, CVE-2016-9804, CVE-2016-9803, CVE-2016-9800	4	12

均 precision 为 68.9%，平均 F_1 为 75.2%，两个指数不高的原因是训练集是不均衡的，非漏洞是漏洞的 3 倍。在漏洞检测领域，尽可能的检测最多的漏洞比减少误报更为重要，所以可以认为 NB 是 5 个分类器当中表现最好，将 96.6% 的漏洞分类正确。

表 3.11　五个分类器的性能

Classifierss	TP	FN	FP	TN	recall(%)	TNR(%)	precision(%)	F_1(%)
KNN	42	16	20	154	72.4	88.5	67.7	70
DT	44	14	31	143	75.9	82.2	58.7	66.2
NB	56	2	27	147	96.6	84.5	67.5	79.5
Adaboosting	46	12	15	159	79.3	91.4	75.4	77.3
SVM	54	4	18	156	93.1	89.7	75	83.1

表3.12所示是本章方法和 BOMiner[22] 的对比。BOMiner 最多能检测出 38 个漏洞，比本章方法最差的分类器 KNN 少 4 个，比最优的分类器 NB 少 18 个。BOMiner 最高的 recall 是 65.5%，比本章训练的所有分类器低。图3.10（a）是本章方法和 BOMiner 在混淆矩阵平均值上的比较。本章方法平均比 BOMiner 多发现 14.8 个漏洞，10.4 个非漏洞。图3.10（b）是本章方法和 BOMiner 在衍生评估参数平均值上的比较。相比与 BOMiner，本章方法在 recall 上增加了 17.5%，F_1 值增加了 21.1%。造成性能差距的主要原因是 BOMiner 并没有更深入的讨论数组写 sink 和危险函数 sink，BOMiner 在处理这两类 sink 时获取的信息太少以至于不能正确的分类漏洞和非漏洞。

表 3.12　BOMiner 性能

Classifierss	TP	FN	FP	TN	TPR(%)	TNR(%)	precision(%)	F_1(%)
KNN	29	29	21	153	50.0	87.9	58	53.7
DT	31	27	35	139	53.4	79.9	47	50
NB	38	20	46	128	65.5	73.6	45.2	53.5
Adaboosting	33	25	28	146	56.9	83.9	54.1	55.5
SVM	37	21	33	141	63.8	81	52.9	57.8

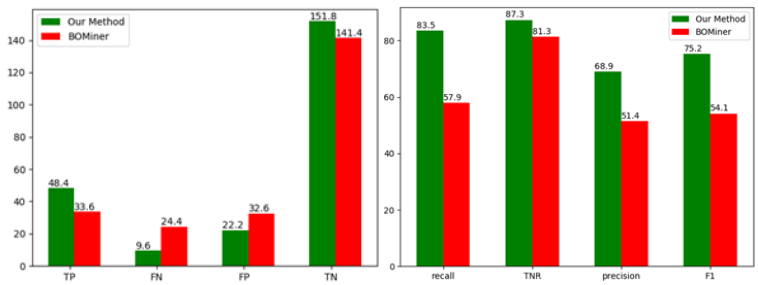


图 3.10　(a) 和 BOMiner 在混淆矩阵平均值的比较, (b) 和 BOMiner 在衍生评估参数上的比较

表3.13是本章方法和 Joern[103] 的性能比较。Joern 能够以很低的误报检测源代码缓冲区溢出漏洞。通过设计特定的搜索模式，Joern 从 linux 内核中的驱动中检测了 6 个缓冲区溢出漏洞。但是，通过研究 CVE 数据库发驱动中已发现的漏洞有 18 个，通过分析驱动源代码发现有 179 个非漏洞 sink 函数。使用本章方法，最少可以检测 12 个缓冲区漏洞，五个分类器当中最小的 recall 是 66.7%，在这两个参数上本章方法要比 Joern 表现好。最高的 TNR 是 93.8%，比 Joern 低 6%。图3.11（a）是本章方法在 linux 驱动上和 Joern 的比较，本章方法平均比 Joern 多 8.2 个漏洞，平均比 Joern 多 11.8 个误报。图3.11（b）是本章方法衍生参数和 Joern 的比较，本章方法的平均 recall 是 78.9%，比 Joern 高了 45.6%，在 TNR 上仅仅低了 6%；平均 precision 比 Joern 低了 24.1%；平均 F_1 值比 Joern 高 15.5%。造成性能差异的原因有两个：（1）Joern 仅仅专注于 memcpy 引起的缓冲区漏洞；（2）Joern 使用的两个边界检测规则，即目的缓冲区的动态分配和关系表达式都可以当做本章的直接边界检测属性，二者虽然可以减少误报但增加了误报。

表 3.13　与 Joern 性能比较

Classifierss	**TP**	**FN**	**FP**	**TN**	**TPR(%)**	**TNR(%)**	**precision(%)**	**F_1(%)**
Joern	6	12	2	177	33.3	98.9	75	46.2
KNN	12	6	11	168	66.7	93.9	52.2	58.6
DT	14	4	14	165	77.8	92.2	50	60.9
NB	15	3	15	164	83.3	91.6	50	62.5
Adaboosting	14	4	12	167	77.8	93.3	53.8	63.6
SVM	16	2	17	162	88.9	90.5	48.5	62.8

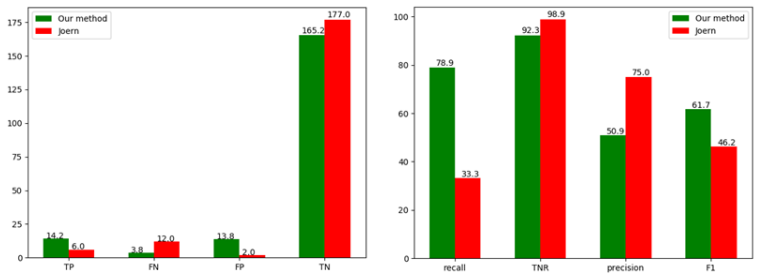


图 3.11　(a) 和 Joern 在混淆矩阵平均值的比较, (b) 和 Joern 在衍生评估参数上的比较

除了在在训练集上检验方法的性能，本节在 poppler 0.10.6 进行了测试，并与 FlawFinder 做了比较，以验证本章方法在减少可以函数数量的作用。Poppler 是一个广泛应用的开源 PDF 库，历史上被发现了很多漏洞。至今 poppler 0.10.6 已经

被发现了 10 个被证实的 CVE：CVE-2015-8868, CVE-2013-1788, CVE-2010-3704, CVE-2009-3938, CVE-2009-3608, CVE-2009-3607, CVE-2009-3606, CVE-2009-3604 以及 CVE-2009-3603。表3.14展示了 5 个分类器在 poppler 0.10.6 的性能，SVFs（Suspect Vulnerable Functions）是 TP 和 FP 之和，表示需要考虑的可以函数；Sink（Sink Functions 表）示所有的满足三种 sink 类型的函数；All （All Functions）表示 poppler 0.10.6 源码中包含的所有函数数量。5 个分类器的平均 TP 为 8.8，表示本方法可以检测 11 个漏洞中的 9 个。9 个 CVE 有 11 个漏洞的原因是 CVE-2013-1788 有三个漏洞函数。平均 recall 是 80%，平均 TNR 为 94%，因为非漏洞函数的数量远小于漏洞函数数所以平均 F_1 较低只有 28.9%。从实验结果可以看出，本方法可以以很低的漏报率发现绝大多数漏洞。

Flawfinder[6] 是一个基于语法分析的静态漏洞扫描器。当在 poppler 0.10.6 运行 Flawfinder 时，其能发现 11 个中的 8 个缓冲区溢出漏洞，稍稍低于本章方法。但是 Flawfinder 同时产生了 500 误报，是本章方法的 12 倍之多。以此可以看出，本章方法在缩小可疑函数上效果非常显著。

表 3.14　五个分类器在 Poppler0.10.6 上的性能测试

	TP	**FN**	**FP**	**TN**	**recall**	**TNR**	**precision**	F_1	**SVFs**	**Sink**	**All**
KNN	7	4	37	648	63.6	94.6	15.9	25.4	44		
DT	8	3	44	641	72.7	93.6	15.4	25.4	52		
NB	10	1	52	633	90.9	92.4	16.1	27.4	62	685	4876
Ada	9	2	39	646	81.8	94.3	18.8	30.6	48		
SVM	10	1	35	650	90.9	94.9	22.2	35.7	45		

3.4　本章小结

本章主要研究了基于程序性质图源代码漏洞挖掘方法与基于机器学习的缓冲区溢出漏洞挖掘方法，具体包括以下两个内容。

（1）提出了基于程序性质图的源代码漏洞挖掘方法，用于标定可疑区域。该方法首先利用鲁棒的中间表示生成语法分析树、抽象语法树、控制流图、数据流图；然后通过性质图将三种中间表示聚合成统一的程序性质图，并定义程序性质图遍历方式；最后根据缓冲区溢出漏洞、格式化字符串漏洞以及 Use After Free 漏洞的特征给出三类漏洞的挖掘算法。通过在 poppler0.10.6、a2ps4.14 以及 libxml2-2.9.3 三个软件上的实验表明，基于程序性质图能够有效的挖掘源代码漏洞。

（2）提出了一种基于机器学习的缓冲区溢出漏洞挖掘方法。该方法首先将 22 种程序静态特征约简成 7 类，分别是 sink 类型、缓冲区位置、容器、索引 / 地址

/ 长度复杂度、边界检测、循环 / 条件 / 函数调用深度以及是否输入可控；其次通过扩展的程序性质图提取静态特征；然后在 CVE 库中查找现存的缓冲区溢出程序和未溢出程序作为训练集，并选取 5 种有监督机器学习算法在训练集中训练分类器；最后利用分类器在新的源代码程序中挖掘漏洞。实验结果表明，5 个分类器的平均召回率（recall 亦称为 TPR,True Positive Rate）是 83.5%, 平均真负率（TNR，True Negative Rate）为 85.9%，最好的召回率达到了 96.6%，最好的真负率达到了 91.4%。因为实验的数据是非均衡数据，所以准确率（Precision）和 F_1 稍低，分别为 68.9% 和 75.2%。将此方法应用到 poppler0.10.6 上，并和经典的静态分析工具 FlawFinder 做对比，本方法能够讲误报率消减到 1/12。实验证明，本方法能够有效的挖掘缓冲区溢出漏洞。

第四章　结合静态程序分析的高效符号执行技术研究

动态符号执行作为典型的动态测试方法，在理论上能够有效遍历目标软件的状态空间，从而深度挖掘软件中存在的漏洞。但对复杂的源代码软件进行完整的符号执行测试是一个 NP 完全问题。因此，现有的测试方法必须在完备性和有效性之间做权衡，引入各自的优化方案。导致符号执行路径空间爆炸的一个非常关键的因素是程序中的循环语句。在分析循环程序时，符号执行每一次进入循环体内部，都需要构造两条分支路径，分别对应了循环条件为真或假的情况。总共的分支路径数目随着循环次数的增加呈指数级增加。特别的，如果循环控制变量输入相关，且没有对循环展开次数进行人为的限定，则循环将进行无限展开，一个简单的循环将直接导致符号执行的路径爆炸问题目前试图缓解路径空间爆炸问题的方式是，在符号执行的具体实现中，设计出好的路径遍历策略或者路径调度算法来提高符号执行的分析性能，以在有限的时间和空间范围内达到最大的代码检测覆盖率。在给定的遍历策略下，通过限定每个过程内的分析路径数目上限的方法来缓解该问题所产生的影响，或者通过设置时间上限或者内存空间上限的方法来缓解路径爆炸问题所可能造成分析工具崩溃的影响。但是这些路径遍历策略或者调度算法都只是局部改进，很难从根本上解决这一问题。

本文提出了一种基于抽象解释静态分析的高效符号执行技术。给定待分析的源程序，首先解析出程序的控制流图。区别于其他符号执行技术，新技术通过静态程序分析方法，从程序的控制流图中计算出循环程序的不变式，然后对程序进行插桩用循环不变式代替循环，形成新的控制流图。在新的控制流图上进行符号执行，符号执行的路径将大大减少。

4.1　问题描述

考虑图4.1中所示 C 程序片段，希望检测程序执行至第九行时，是否存在输入 a 和 b 使得断言 $assert(x \neq y)$ 不被满足。由于变量 a 和 b 各自均有 2^{32} 个可能的取值，通过随机测试的方式检测程序是否存在错误是不可行的。符号执行将输入变量的取值符号化，并在符号值上模拟程序的执行，遍历所有可能的程序路径。

程序路径是程序的一个语句序列，包括一系列程序中顺序执行的代码片段，代码片段之间的连接代表了分支语句导致的控制转移。一个路径是可行的，是指至少存在一组具体的程序输入变量取值，使得程序以该组取值作为输入，将沿着这条路径执行。反之，路径就是不可行的。对于一条程序路径，路径条件是一个关于程序输入变量的符号值的约束。一组输入值使得程序沿着这条路径执行当且仅当这组输入值满足这条路径的路径条件。

```
1  void foobar(int a, int b){
2    int x = 1, y = 0;
3    if (a != 0) {
4      y = 3+x;
5        if (b == 0) {
6          x = 2 * (a+b);
7        }
8    }
9    assert(x != y);
10 }
```

图 4.1　示例程序 1

具体地，符号执行的计算过程通常可表示为三元组的形式：$(stmt, \sigma, \pi)$，其中，*stmt* 表示当前被分析的程序指令，例如赋值语句，条件语句或者函数调用；σ 表示程序路径中不同位置处变量的值，(可以是符号值，也可以是常量，或二者形成的函数)，σ 通常用函数映射表示;π 称为路径约束，是路径在各个分支执行点（位置）的分支条件 (通常是一阶谓词及其布尔合成) 的合取，π 的初始值为布尔常量 *true*。

依据 *stmt* 的不同，符号执行采用不同的计算方式更新 σ 和 π 的取值：

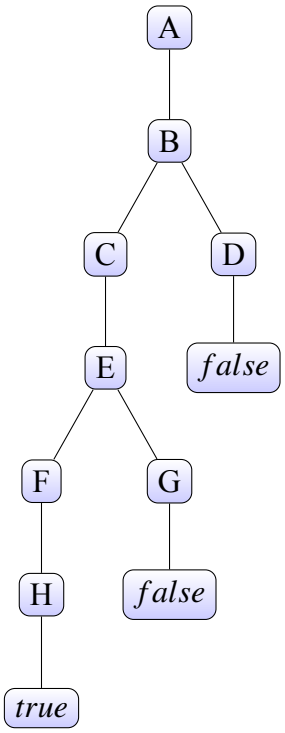
1. 如果 *stmt* 是赋值语句 $x = e$，其中表达式 e 可能是常数，变量或者函数应用，符号执行将更新 σ 中变量 x 的取值为 e 的符号值；

2. 如果 *stmt* 是条件语句 if e then s_1 else s_2，符号执行将构造两条分支路径，分别对应了条件语句取值为真和假的两个情况，并更新两条分支路径的路径约束为：$\pi_{true} = \pi \wedge e$，$\pi_{false} = \pi \wedge \neg e$。

图4.1中 C 程序代码的符号执行过程如下图所示，左边部分表示了符号执行的执行树，右边的表格列出了执行树的每个节点中的数据信息。

从执行树中，可以看出路径 *ABDEFH* 表示了一条从程序初始入口到目标断言的执行，该路径由指令序列 $x = 1, y = 0; if(a \neq 0); y = 3 + x; if(b == 0); x = 2 * (a + b); assert(x \neq y)$ 构成。在断言位置，程序变量的符号取值为 $\sigma = \{a = m, b = n, x = 2(m + n), y = 4\}$，路径约束为 $m \neq 0 \wedge n = 0$, 其中，m, n 为符号值。那么，检测断言是否满足可以转化为检测逻辑公式 $x = 2(m + n) \wedge y = 4 \wedge m \neq 0 \wedge n = 0 \wedge xy$ 是否可满足。一阶逻辑的可满足性可以通过约束求解器 SMT 求解。通过约简，可以容易得到当 $m = 2, n = 0$ 时，断言 $assert(x \neq y)$ 不被满足。

从图4.1中也可以看出，符号执行中，每一个分支条件语句都可能会使当前的路径再分支出一条新的路径。执行树的分支数 (即程序路径数)，随条件语句的数目的增加而增长，且增长关系是指数级的。当分支数目变得巨大时，程序路径数目也将变得庞大而使得计算不可行，该问题称为路径爆炸问题。



节点	σ	π	*stmt*
A	$\{a=m, b=n\}$	*true*	$x=1, y=0$
B	$\{a=m, b=n, x=1, y=0\}$	*true*	$if(a \neq 0)$
C	$\{a=m, b=n, x=1, y=0\}$	$m \neq 0$	$y=3+x$
D	$\{a=m, b=n, x=1, y=0\}$	$m=0$	$assert(x \neq y)$
E	$\{a=m, b=n, x=1, y=4\}$	$m \neq 0$	$if(b=0)$
F	$\{a=m, b=n, x=1, y=4\}$	$m \neq 0 \wedge n=0$	$x=2*(a+b)$
G	$\{a=m, b=n, x=1, y=4\}$	$m \neq 0 \wedge n \neq 0$	$assert(x \neq y)$
H	$\{a=m, b=n, x=2(m+n), y=4\}$	$m \neq 0 \wedge n=0$	$assert(x \neq y)$

图 4.2　示例程序 1 符号执行树及其计算过程

导致符号执行路径空间爆炸的另一个关键因素是程序中的循环语句，例如图4.3中所示的 while 循环。在分析这类循环程序时，符号执行每一次进入循环体内部，都需要构造两条分支路径，分别对应了循环条件为真或假的情况。总共的分支路径数目随着循环次数的增加呈指数级增加。特别的，如果循环控制变量输入相关，且没有对循环展开次数进行人为的限定，则循环将进行无限展开，一个简单的循环将直接导致符号执行的路径爆炸问题。因此本章提出了一种基于抽象解释静态分析的高效符号执行技术用于缓解由循环引起的路径爆炸问题。

```
1  void LoopExample(int x) {
2    int i = 3, p = x;
3    while(p > 0) {
4      i = i + 4;
5      p = p - 1;
6         if (i >= 80)
7           assert(0); //target1
8    }
9    if(i>78)
10        assert(0);//target2
11 }
```

图 4.3　示例循环程序

4.2 技术方案设计

针对程序中循环语句可能导致的路径空间爆炸问题，本章提出了一种基于抽象解释的高效符号执行技术。该技术方案的框架图如图4.4所示。给定待分析的源程序，首先解析出程序的控制流图。区别于其他符号执行技术，新技术通过静态程序分析方法，从程序的控制流图中计算出循环程序的不变式，然后对程序进行插桩用循环不变式代替循环，形成新的控制流图。在新的控制流图上进行符号执行，符号执行的路径将大大减少。图4.4的左下部分为原始程序的符号执行路径，假设循环 $b \to c \to d \to b$ 的循环次数是 n，则在符号执行时此循环将生成 2^n 条路径。经过静态分析后，$b \to c \to d \to b$ 代表的循环由 b' 代替。在对修改之后的控制流图进行符号执行时，原本的 2^n 用一条路径代替。简言之，新技术旨在通过基于抽象解释的静态程序分析技术合成循环的约束，并将该约束提供给符号执行引擎，减少符号执行路径。

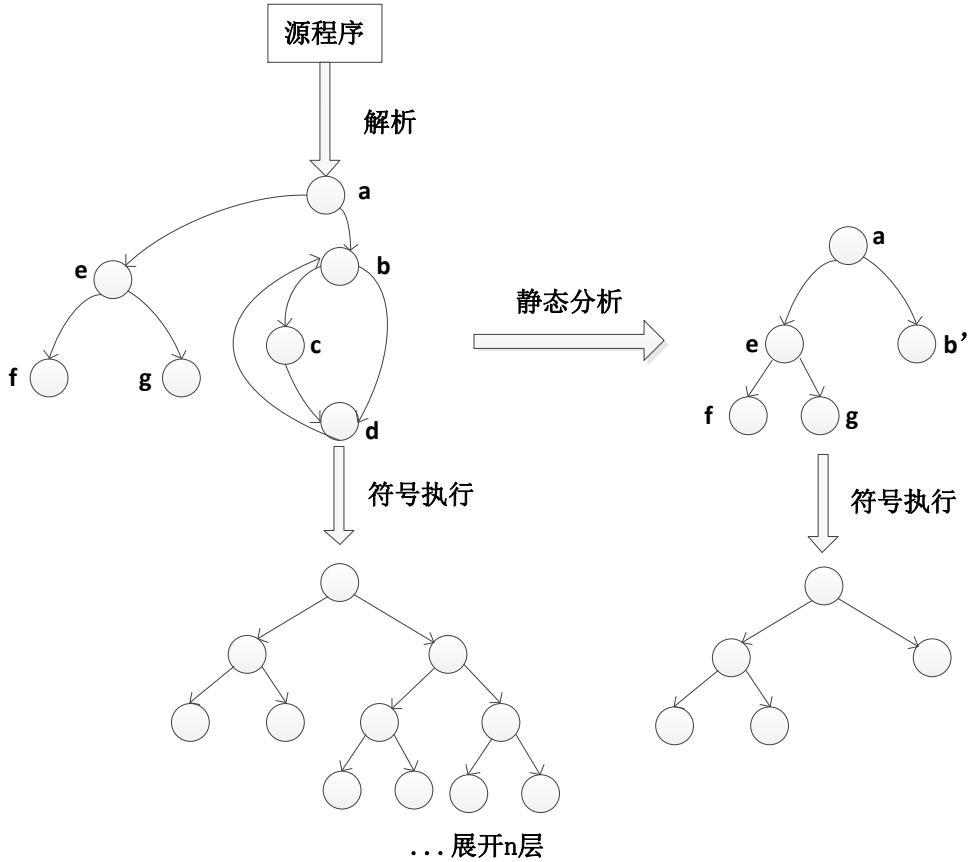


图 4.4　基于抽象解释的符号执行技术框架

4.3 基于抽象解释的静态程序分析技术研究

由于一般情况下，程序具体对象域上的不动点语义计算是不终止的，基于抽象的静态程序分析技术通过构造一个抽象的对象域，将程序语义映射到抽象对象域上，并构造程序的抽象不动点语义，作为对程序具体语义的逼近。有效的抽象方法需要确保：(1) 抽象域上的不动点语义计算是可行的；(2) 抽象语义是对具体语义的正确逼近。抽象解释理论解决了如何快速计算程序不动点语义的问题，在程序不动点语义计算的精确度和效率之间取得均衡，以低精确度的抽象计算换取较高的计算效率。抽象解释为设计实用抽象技术，以及证明其的可靠性和正确性提供了理论工具。

抽象解释理论是 Patrick Cousot 及其夫人于 1977 年提出的静态程序分析理论。基于抽象解释理论的静态程序分析技术的基本思想是，在保证抽象逼近可靠性的前提下，对程序不动点语义计算过程的各个层次进行抽象逼近，使得计算过程收敛或加速其收敛，然后依据逼近计算的不动点语义验证程序需要满足的性质，若验证过程中出现虚假反例，则对抽象过程进行精化。基于抽象解释理论的形式化验证的特点是可靠但不完备的。如果分析结果表明抽象系统满足验证性质，则抽象解释理论保证原程序也满足验证性质; 若验证结果表明抽象系统不满足验证性质，则原系统可能满足验证性质 (表明抽象过程中出现了不满足验证性质的虚假反例)，可能不满足验证性质 (原系统存在不满足验证性质的真实反例)，需要更精细的抽象分析。

4.3.1 抽象解释理论框架

首先介绍抽象解释理论的数学基础。直观上，程序语义的对象域是是程序运算的对象，包括了程序变量所有可能的取值集合。数学上，通过一个偏序集合来刻画。

定义 4.1: 集合 $< L, \subseteq >$ 称为偏序集合，当且仅当，二元序关系 $\subseteq$ 满足以下条件：

1. $\forall l \in L.(l, l) \in \subseteq$;

2. 如果 $(l, l_1) \in \subseteq \wedge (l_1, l') \in \subseteq$, 那么 $(l, l') \in \subseteq$;

3. 如果 $(l, l') \in \subseteq$，$(l', l) \in \subseteq$，那么 $l = l'$; ■

定义 4.2: 假设 $< L, \subseteq >$ 是一个偏序集合，$< L, \subseteq >$ 是一个完备格当且仅当：L 的任何一个子集均在 L 中存在最小上界和最大下界。特别地，用 $\sqcup$ 表示最小上界算子，用 $\sqcap$ 表示最大下界算子，用 $\perp$ 表示最小元素，用 $\top$ 表示最大元素。完备格通常表示为 $< L, \subseteq, \sqcup, \sqcap, \perp, \top >$. ■

具体对象域与抽象对象域之间的映射的关系可以通过一对算子来刻画。数学上，这对算子构成了一个 Galois 连接。

定义 4.3: 设 $< P, \leq >$ 和 $< Q, \subseteq >$ 是两个偏序集合，$\alpha : P \to Q, \gamma : Q \to P$ 是两个映射，序偶 $< \alpha, \gamma >$ 称为一个 Galois 连接当且仅当：$\forall x \in P, \forall y \in Q.\alpha(x) \subseteq y$ 当且仅当 $x \leq \gamma(y)$，其中，映射 α 称为抽象算子，γ 称为具体算子。

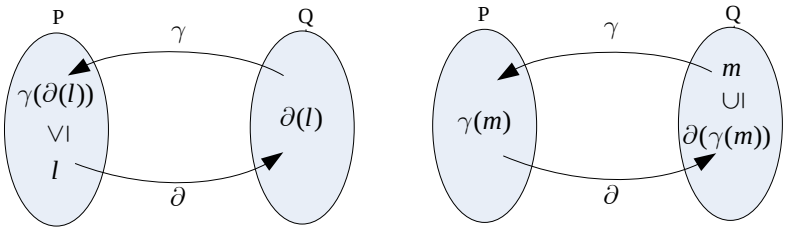


图 4.5 Galois 连接示意图

由 Galois 连接的定义，容易得到如下关于抽象算子和具体算子的引理。

引理 4.1: 序偶 $< \alpha, \gamma >$ 是偏序集 $< P, \leq >$ 和 $< Q, \subseteq >$ 上的一个 Galois 连接，则

1. $\forall x \in P.x \leq \gamma(\alpha(x))$;

2. $\forall y \in Q.\alpha(\gamma(y)) \subseteq y$;

3. α 和 γ 单调递增函数。

证明 4.1: $\forall x \in P.\alpha(x) \in Q$. 由于 $< Q, \subseteq >$ 是一个偏序集合，则 $\alpha(x) \subseteq \alpha(x)$. 依据 Galois 连接的定义，得到 $x \leq \gamma(\alpha(x))$(将右边的 $\alpha(x)$ 视为 y)。同理，$\forall y \in Q.\gamma(y) \in P$. 由于 $< P, \leq >$ 是一个偏序集合，则 $\gamma(y) \leq \gamma(y)$. 依据 Galois 连接的定义，得到 $\alpha(\gamma(y)) \subseteq y$(将右边的 $\gamma(y)$ 视为 x)。

由于 $\forall x \in P.x \leq \gamma(\alpha(x))$，以及偏序关系的传递性，容易得知 $x \leq \gamma(\alpha(\gamma(\alpha(x))))$，依据 Galois 连接的定义，进一步得到 $\alpha(x) \subseteq \alpha(\gamma(\alpha(x)))$，故 α 算子是单调递增的。同理可证 γ 的单调递增性。

由于在程序具体对象域上计算语义最小不动点是不可行的，基于抽象解释理论，可以通过抽象算子，将不动点计算映射到抽象对象域上，并通过具体算子将抽象的计算结果映射到程序具体对象域上逼近实际的语义不动点。假设程序 *Prog* 的最小不动点语义 (即可达状态空间) 为 $LFP(Post_{Prog})$，通过 Galois 连接 $<\alpha,\gamma>$，定义程序的抽象后继算子 $AbsPost_{Prog}=\alpha\bullet Post\bullet\gamma$, 其中，符号 $\bullet$ 表示函数合成, 那么程序的最小不动点语义可以用抽象 $LFP(AbsPost_{Prog})$ 逼近。Galois 连接的性质保证了抽象不动点是对实际语义不动点的正确逼近。

在实际应用中，抽象域上的不动点计算也不一定终止。程序不动点语义的计算过程不收敛是引起程序验证过程中状态空间爆炸的另一个原因。假设 $<L,\subseteq,\sqcup,\sqcap,\bot,\top>$ 是一个完备格，程序的操作语义可以由完备格上的单调语义泛函 *Post* 来刻画，理想情况下，*Post* 算子是完备格 L 上的单调连续函数，并且不动点计算 $\bigcup_{i=0}\{Post^i\}$ 能够在有限次迭代计算后收敛。然而，在实际情况下，不动点计算不一定在有限次迭代过程中收敛，即便收敛，也不一定收敛到最小不动点。在抽象解释的理论框架中，Widening 算子可以给出一个不动点计算的上界逼近，而 Narrowing 算子给出更加精细的上界逼近。

定义 4.4 (Widening **算子**): 假设 $<L,\subseteq,\sqcup,\sqcap,\bot,\top>$ 是一个完备格，算子 $\nabla: L\to L$ 是一个 Widening 算子，当且仅当，

1. ∇ 是一个上界算子;
2. 对于任意的递增链 $(l_n)_n$，递增链 $(l_n^{\nabla})_n$ 收敛，其中，$l_n^{\nabla}=l_0$，当 $n=0$; $l_n^{\nabla}=l_{n-1}^{\nabla}\nabla l_n$，当 $n>0$。 ■

假设程序 *Prog* 的单调语义泛函 *Post* 定义在完备格 $<L,\subseteq,\sqcup,\sqcap,\bot,\top>$ 上，并且算子 ∇ 是完备格上的一个 Widening 算子，定义序列 $Post_n^{\nabla}$ 为:

1. 若 $n=0$，则 $Post_n^{\nabla}=\bot$;
2. 若 $n>0$，并且 $Post(Post_{n-1}^{\nabla})\subseteq Post_n^{\nabla}$，则 $Post_n^{\nabla}=Post_{n-1}^{\nabla}$;
3. 若 $n>0$，并且 $Post_n^{\nabla}\subseteq Post(Post_{n-1}^{\nabla})$，则 $Post_n^{\nabla}=Post(Post_{n-1}^{\nabla})\nabla Post_{n-1}^{\nabla}$。

可以证明序列 $Post_n^{\nabla}$ 是收敛的，并且存在 $m>0$，使得 $Post(Post_m^{\nabla})\subseteq Post_m^{\nabla}$，且 $Post_m^{\nabla}$ 是最小不动点 $LFP(Post)$ 的一个上界逼近。

定义 4.5 (Narrowing **算子**): 假设 $<L,\subseteq,\sqcup,\sqcap,\bot,\top>$ 是一个完备格，算子 $\triangle: L\to L$ 是一个 Narrowing 算子，当且仅当，

1. $\forall l_1, l_2 \in L$，如果 $l_2 \subseteq l_1$，则 $l_2(l_2 \triangle l_1) \subseteq l_1$;

2. 对于任意的递降链 $(l_n)_n$，递降链 $(l_n^{\triangle})_n$ 收敛，其中，$l_n^{\triangle} = l_0$，当 $n = 0$; $l_n^{\triangle} = l_{n-1}^{\triangle} \triangle l_n$，当 $n > 0$。 ■

假设算子 $\triangle$ 是完备格上的一个 Narrowing 算子，定义序列 $Post_n^{\triangle}$ 为:

1. 若 $n = 0$，则 $Post_n^{\triangle} = Post_n^{\nabla}$;

2. 若 $n > 0$，则 $Post_n^{\triangle} = Post_{n-1}^{\triangle} \triangle Post(Post_{n-1}^{\triangle})$。

可以证明 $LFP(Post) \subseteq Post(Post_m^{\nabla}) \subseteq Post_n^{\triangle} \subseteq Post_m^{\nabla}$。$Post_m^{\triangle}$ 同样是 $LFP(Post)$ 的一个上界逼近，并且是一个较 $Post_m^{\nabla}$ 更加精确的上界逼近。

例 4.3.1: 区间抽象域的思想是通过一个区间去逼近变量的取值集合。设 Z 是一个整数集合，如果 $a = min(Z), b = max(Z)$，那么区间 $[a, b]$ 是对 Z 的逼近。区间域的序关系 $\subseteq$ 定义如下：$[a, b] \subseteq [c, d]$, 当且仅当 $a \geq c$，并且 $b \leq d$，既是区间 $[a, b]$ 包含在区间 $[c, d]$ 中。域中的最大元素是 $(-\infty, +\infty)$, 最小元素是空集 $\emptyset$。

区间抽象域含有无穷多个元素，且含有无穷递增序列，因此需要扩展算子 (*widening*) 进行近似计算。区间域的扩展算子定义如下:

1. 如果 $m_2 < m_1$，并且 $n_2 < n_1$, 则 $[m_1, n_1] \nabla [m_2, n_2] = [\perp, n1]$;

2. 如果 $m_1 < m_2$，并且 $n_1 < n_2$, 则 $[m_1, n_1] \nabla [m_2, n_2] = [m_1, \top]$;

3. 如果 $m_2 < m_1$，并且 $n_1 < n_2$, 则 $[m_1, n_1] \nabla [m_2, n_2] = [\perp, \top]$;

4. 如果 $m_1 < m_2$，并且 $n_2 < n_1$, 则 $[m_1, n_1] \nabla [m_2, n_2] = [m_1, n1]$。

除区间域外，常用的抽象域还包括了八边形域、多项式域。区间域的表达能力较弱，只能够刻画单个程序变量的取值约束，但是计算复杂度最低。多项式域的表达能力最强，可以刻画程序变量的相互关系，但是计算复杂度高。

4.3.2 基于抽象解释的静态程序分析

基于抽象解释理论的程序语义不动点计算过程的基本思想在算法4.1所示。算法以待分析的程序 *Prog* 作为输入，并计算程序变量在各个程序位置的不动点语义。形式化地，用函数 Σ 表示程序的语义不动点，该函数是一个从程序控制节点集合到一阶逻辑公式的映射，给定程序控制节点 $l \in L$, $\Sigma(l)$ 表示了在该节点位置，

算法 4.1 基于抽象解释的静态程序分析算法 (AbstractStaticAnalysis)

Input: 待分析程序源码 *Prog*
Output: 程序变量在各个位置的约束映射 Σ

1: 创建程序 *Prog* 的控制流图 $CFG = (L, E, l_0)$
2: 创建初始约束映射 $\Sigma : L/\{l_0\} \rightarrow \{false\} \bigcup \{l_0\} \rightarrow \{true\}$
3: 创建工作列表 *worklist*
4: $worklist \leftarrow \{l_0\}$
5: **while** $worklist \neq \emptyset$ **do**
6: $l \leftarrow \text{pop}(worklist)$
7: $\sigma \leftarrow \Sigma(l)$
8: **if** $l \neq l_0$ **then**
9: $\Gamma_I \leftarrow \text{InEdges}(l)$
10: $\sigma_{new} \leftarrow \sigma$
11: **for** $\gamma \in \Gamma_I$ **do**
12: $\gamma \leftarrow (l', stmt, l)$
13: $\sigma' \leftarrow \Sigma(l')$
14: $\sigma'' \leftarrow post(\sigma', stmt)$
15: $\sigma_{new} \leftarrow \sqcup(\sigma, \sigma'')$
16: **end for**
17: **if** $\sigma_{new} \neq \sigma$ **then**
18: $\Sigma(l) \leftarrow \sigma_{new}$
19: $\Gamma \leftarrow \text{OutEdges}(l)$
20: **for** $\gamma \in \Gamma$ **do**
21: $\gamma \leftarrow (l, stmt, l')$
22: $\text{push}(l', worklist)$
23: **end for**
24: **end if**
25: **else**
26: $\Gamma \leftarrow \text{OutEdges}(l)$
27: **for** $\gamma \in \Gamma$ **do**
28: $\gamma \leftarrow (l, stmt, l')$
29: $\text{push}(l', worklist)$
30: **end for**
31: **end if**
32: **end while**

程序变量所有可能取值的约束条件。该约束条件可能是一个取值区间，或者是一个线性不等式，具体形式取决于不动点计算采用的抽象域。

算法首先构建程序的控制流图 $CFG = (L, E, l_0)$，并对函数 Σ 进行初始化。初始条件下，程序入口位置的约束为布尔常量 $true$，其他位置的约束为常量 $false$。算法采用典型的工作列表设计模式，工作列表中存放待分析的程序节点。算法每一次从工作列表中取出一个节点进行分析，并依据分析结果将该节点的后继节点放入列表中，直至所有节点的分析完成，即工作列表为空。初始条件下，工作列表仅含有程序的初始入口节点 l_0。

对于程序的每个控制节点 l，如果是初始节点，那么算法直接将其后继节点加入工作列表；如果不是初始节点，算法将进行进一步的分析。具体地，算法首先获取该节点 l 的所有前继语句 Γ_l，对每个语句 $(l', stmt, l) \in \Gamma_l$，及其前继节点 l' 的约束 σ'，算法应用 Post 后继算子计算相对于语句 $stmt$ 的后继约束条件 σ''，并应用抽象域中的上界算子 $\sqcup$，得到当前节点 l 处的新的约束值 $\sigma_{new} \sqcup (\sigma, \sigma'')$。上述计算 $post(\sigma', stmt)$，$\sqcup(\sigma, \sigma'')$ 均在给定的抽象域上完成，具体计算规则由抽象解释理论的抽象域给出。

随后，算法判断位置 l 的新约束 σ_{new} 是否与当前约束 σ 等价。如果不等价，则更新该位置的约束为 σ_{new}，并将位置 l 的所有后继节点加入工作列表；如果相同，说明该位置的约束已经包括了程序变量的所有可能取值。当在每个位置的约束均覆盖了程序变量在该位置的所有可能取值时，算法终止。抽象解释理论确保了算法的终止性。

本文采用基于抽象解释理论的静态程序分析技术计算程序变量的约束关系，例约束 $x - p = (i - 3)/4$，并使用该约束辅助符号执行。基于多边形抽象域的静态分析技术可以计算上述形式的约束。

4.4 结合静态程序分析的高效符号执行算法

程序中的循环语句是导致符号执行路径爆炸问题的主要因素之一。缓解该问题的解决方法主要包括了：1: 对循环体做有限次展开，以损失分析的完备性换取计算的可行性；2: 采用特殊的导向策略引导符号执行的路径遍历，避免对循环体做无限次展开；3: 基于循环摘要对循环语句进行抽象，该抽象是对循环体行为的刻画。对循环程序的分析可以转换为对循环摘要的分析。本文提出了一种基于抽象解释的循环程序符号执行技术。为提高对循环程序的分析效率，提出采用抽象解释技术计算出循环程序的语义逼近，并用该抽象语义指导符号执行技术。

4.4.1 核心技术思想

首先通过实例来解释本文提出的技术方案。考虑图4.3所示循环程序片段，为了检测程序是否能够执行至第七行，符号执行进行路径搜索的同时，需要对 while 循环进行展开，展开次数与函数的输入参数 x 以及变量 i 相关。理论上，对于该

程序片段，符号执行需要对循环进行 20 次展开，那么就需要遍历 2^{20} 个程序路径。图4.6和图4.7展示了对循环进行 2 次展开的符号执行过程。

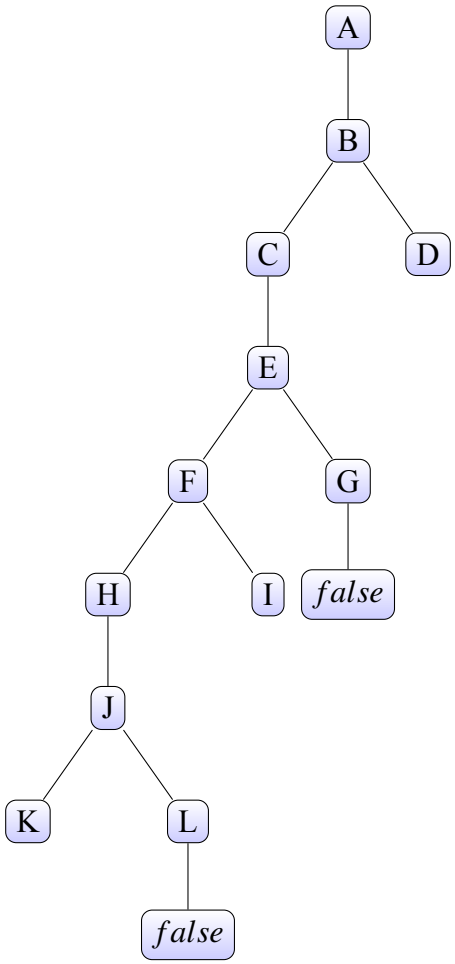


图 4.6　示例程序 2 符号执行树

1. 符号执行树中，程序路径 $ABCEG$ 表示了一条从初始入口到目标指令的执行路径。该路径通过对循环进行一次展开得到，其对应的指令序列为 $i=3, p=x; while(p>0); i=i+4; p=p-1; if(i \geq 80); abort();$，在目标指令位置，程序变量的符号取值为 $x=m; i=7; p=m-1$，路径约束为 $m>0 \wedge i \geq 80$。显然，当前变量 i 的取值不满足路径约束，所以该执行路径是不可行的。

2. 类似地，通过对循环进行两次展开，得到程序路径 $ABCEFHJL$，其对应的指令序列为 $i=3, p=x; while(p>0); i=i+4; p=p-1; if(i<80); while(p>0); i=i+4; p=p-1; if(i \geq 80); abort()$，在目标指令位置，程序变量的符号取值为 $x=m; i=11; p=m-2$，路径约束为 $m>0 \wedge i \geq 80 \wedge m-1>0$。当前变量 i 的取值同样不满足路径约束，所以该执行路径是不可行的。

节点	σ	π	*stmt*
A	$\{x = m\}$	*true*	$i = 3, p = x$
B	$\{x = m, i = 3, p = m\}$	*true*	$while(p > 0)$
C	$\{x = m, i = 3, p = m\}$	$m > 0$	$i = i + 4; p = p - 1$
D	$\{x = m, i = 3, p = m\}$	$m \leq 0$	*skip*
E	$\{x = m, i = 7, p = m - 1\}$	$m > 0$	$if(i \geq 80)$
F	$\{x = m, i = 7, p = m - 1\}$	$m > 0 \wedge i < 80$	$while(p > 0)$
G	$\{x = m, i = 7, p = m - 1\}$	$m > 0 \wedge i \geq 80$	*abort*()
H	$\{x = m, i = 7, p = m - 1\}$	$m > 0 \wedge i < 80 \wedge m - 1 > 0$	$i = i + 4; p = p - 1$
I	$\{x = m, i = 7, p = m - 1\}$	$m > 0 \wedge i < 80 \wedge m - 1 \leq 0$	*skip*
J	$\{x = m, i = 11, p = m - 2\}$	$m > 0 \wedge i < 80 \wedge m - 1 > 0$	$if(i \geq 80)$
K	$\{x = m, i = 11, p = m - 2\}$	$m > 0 \wedge i < 80 \wedge m - 1 > 0$	$while(p > 0)$
L	$\{x = m, i = 11, p = m - 2\}$	$m > 0 \wedge i \geq 80 \wedge m - 1 > 0$	*abort*()

图 4.7　示例程序 2 符号执行计算过程

3. 以此类推，对循环进行第 20 次展示后，得到的程序变量符号约束为 $x = m; i = 83; p = m - 20$，路径约束为 $i \geq 80 \wedge m - 19 > 0$。此时，变量 i 的取值满足路径约束，该执行路径是可行的，相应的输入变量的约束条件为 $x > 19$。当目标指令所在的分支条件，即 $i \geq 80$, 发生改变，如 $i \geq 100$，那么循环的展开次数则变为 100，此时符号执行树的分支数目增长为 2^{100}。由此可知，符号执行在分析此类循环程序时，即便程序规模很小，也难以在合适的时间内给出分析结果。

进一步分析可知，为了快速高效地找到触发程序第一个目标 *assert* 指令的输入值，关键是需要找到变量 p 和 i 的关系：$x - p = (i - 3)/4$。假设我们通过特定的程序分析技术得到了上述关系，那么在运行符号执行时，例如当符号执行遍历路径 *ABCEG* 时，我们不使用当前路径的变量取值 $x = m; i = 7; p = m - 1$，而使用约束 $x - p = (i - 3)/4$ 去判定是否存在输入使得路径约束被满足。由于约束 $x - p = (i - 3)/4 \wedge i \geq 80 \wedge p \geq 0$ 是可满足的，得到输入变量的约束为 $x > 19$。

本质上，约束 $x - p = (i - 3)/4$ 刻画了程序变量在目标指令位置的所有可能取值。例如，当程序通过路径 *ABCEG* 到达目标指令位置时，变量的取值为 $x = m; i = 7; p = m - 1$，显然，$x - p = (i - 3)/4$。当程序通过路径 *ABCEFHJL* 到达目标指令位置时，变量的取值为 $x = m; i = 11; p = m - 2$，同样满足上述约束。在程序分析领域，约束 $x - p = (i - 3)/4$ 通常称为程序的循环不变式。

4.4.2 高效符号执行算法

该技术的核心思想如算法4.2所示。大体上，算法分为两部分：静态分析部分和动态符号执行部分。算法接收待验证程序源码 *Prog* 作为输入，并初始化抽象执行树 tree、工作列表以及程序语义不动点。抽象执行树的节点代表了程序的抽象状态，树的边表示程序的执行路径。每个节点 η 包括了三部分信息：程序的指令位置 *loc*，程序变量的符号取值 σ 和程序路径在当前位置的路径约束 π。算法维护一个工作列表，里面的元素是待遍历的树节点，即程序状态。算法在进行符号执行式，逐次从工作列表中取出一个节点 η 进行分析，直至工作列表为空。结合静态程序分析的高效符号执行算法算法4.2可分为以下过程。

（1）对于程序 *Prog* 中的每个循环运行基于抽象解释的静态程序分析算法生成程序在各个位置的不动点（第 3 行），然后将不动点以注释的形式写入程序中（第 4 行），其中不动点是由各个变量的不变式组成。下面以图4.3中的循环程序为例，阐述生成不变式的过程。首先在循环中加入一个初始值为 0 的计数变量 counter，用于记录循环的展开次数。加入 *counter* 的目的是表达程序变量和循环展开次数的关系。为了在每个程序点产生不变式，本章在多面体抽象域[112] 中运行前向抽象解释算法进行静态分析。多面体抽象域是一个完全的数值关系域，可以用来表达任意的线性等式和不等式。多面体抽象域比数值域[112]、五边形域[113] 以及八边形域[114] 表达能力更强，同时耗费的资源也更多。如图4.8第 4 行所示；此不变式表示要到达当前程序点变量 *x* 需要满足的约束条件。插入不动点的位置有三类：循环的入口处，分支语句处以及循环结束处，分别对应图4.8的第 4-5 行、第 9 行以及第 12-13 行。

（2）在进行符号执行之前，需要通过插桩的方式插入特殊指令（第 6 行），使得在符号执行时能够辨别需要处理的循环对象。在每个循环的开始处插入”LSEntry”表示循环处理开始，如图4.8的第 3 行；在循环的结束处插入“LSEnd”表示循环处理结束，如图4.8的第 15 行。

（3）动态符号执行的过程中会逐条读取指令，根据指令类型的不同执行不同的操作（第 8-21 行）。其中第 9-15 行为正常的符号执行过程。从 *worklist* 中取出一个 η，判断当前位置是否存在错误（第 10 行），如果存在则调用约束求解器求解测试用例（第 13 行），然后返回测试用例。本章对符号执行的修改体现在对新添指令“LSEntry”和“LSEnd”的处理上（第 17 行）。动态符号执行最主要的三个操作是约束累积、更新内存（包括符号内存和具体内存）以及约束求解。图4.9是符号执行变换后程序的示意过程。以图4.8中的程序为例，符号执行时将 x 符号化。当符号执行读取本章自定义的标识指令“LSEntry”时，则会根据抽象解释产生的注释做相应的操作。首先，新建表示循环展开次数的变量 counter。然后，若目标

```
1  void LoopExample(int x) {
2    int i = 3, p = x;
3    LSEntry;
4    while(p > 0) {
5    /* i = 4 * counter + 3, counter >= 0, counter <= 19,
6     x = counter + p, p > 0 */
7      i = i + 4;
8      p = p - 1;
9         if (i >= 80)
10        /* i = 4 * counter + 3, counter = 20, x = counter + p, p
             > 0*/
11          assert(0); //target1
12   }
13   /* i = 4 * counter + 3, x = counter + p, counter >= 0,
14    counter <= 19, p < 0*/
15    LSEnd;
16    if(i>78)
17      assert(0);//target2
18 }
```

图 4.8　注释后的循环实例程序

区域包含在循环内部，则从注释 /* i = 4 * counter + 3, counter = 20, x = counter + p, p > 0*/ 提取等式进行具体和符号内存更新，具体内存的更新为 i=83，counter=20，符号内存的更新为 x=20+p；提取不等式 p>0 更新符号约束，进而约束求解求得满足约束的测试用例，此例中只要 x>=20 都能满足约束条件。若目标区域在循环外部，例如图4.8的第 17 行，则从注释 /* i = 4 * counter + 3, x = counter + p, counter >= 0, counter <= 19, p < 0*/ 提取等式 counter=x-p，i=4*counter+3 更新内存，提取不等式 counter>=0 累积约束，返回到自定义标识指令“LSEnd”继续后面程序的符号执行。例如，在符号执行到图4.9第 16 行时，将 i>78 加入到约束并对约束求解，解得 x=19 就是到达第 17 行 assert 语句的输入。利用抽象解释产生的不变式，将图4.9的 while 循环变换成一个分支语句，一个指向循环中的目标区域，一个指向循环后的程序点；若目标区域不在循环内部，则将 while 循环变换成一个顺序语句。如此则能大大减少符号执行中因循环引起的执行状态。

4.5　实验结果与分析

4.5.1　实验设计

（1）实验目的

本实验的目的是在标准测试程序上验证结合静态程序分析的高效符号执行技术的有效性。验证本章方法相对于其他方法在挖掘漏洞上的效率；验证本章方法在漏洞挖掘数量上的优越性。

（2）实验环境

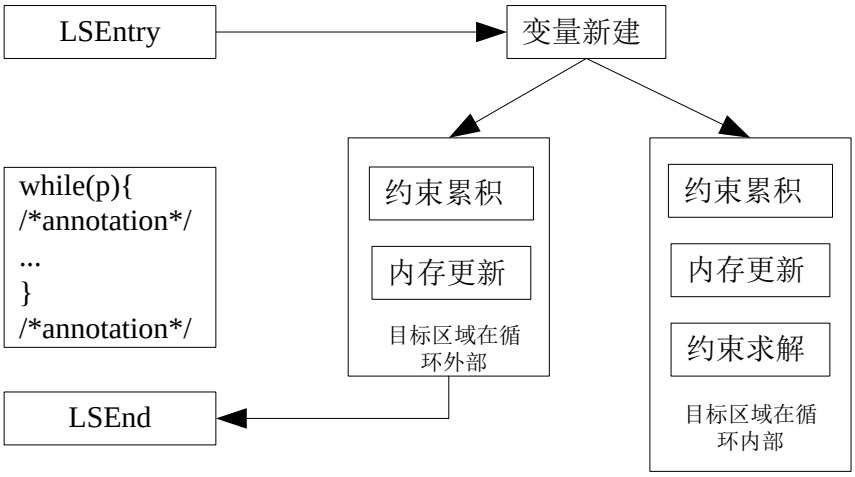


图 4.9　符号执行变换后的程序

算法 4.2 基于静态程序分析的高效符号执行算法

Input: 待分析程序 *prog*

Output: 程序 bug，或者运行超时

```
1: 初始化程序语义不动点 fixpoint
2: for loop in prog do
3:         fixpoint = forwardAnalysis(loop)
4:         annotate(fixpoint)
5: end for
6: prog′ = instrument(prog)
7: worklist = initilize(prog′)
8: while worklist ≠ ∅ do
9:         η ← pop(worklist)
10:        if IsError(η) then
11:                η = (loc, σ, π)
12:                if σ ∧ π is SAT then
13:                        cex ← BuildCEX(η)
14:                        return cex
15:                end if
16:        else
17:                Γ ← ExtActions(η)
18:                Successors(η, Γ)
19:                add successors of η into worklist
20:        end if
21: end while
```

本节基于C语言编译器Clang[115]结合KLEE实现了本文提出的高效符号执行技术（Fast Symbolic Execution，FastSE）。在实验中，本节选取公开的测试程序源码作为测试集与符号执行工具KLEE进行对比。本文的测试集取自2017年软件验证国际竞赛[1]的测试用例库。测试集总共有100个测试程序。每个测试程序中均含有一条assert语句，并且均存在一组输入，使得程序运行至该语句时，该语句的取值为假。100个测试程序中有68个程序包含循环，其中15个包含嵌套循环。实验的采用的硬件环境是Inter Xeon CPU E3-1231 v3 @ 3.40GHz，16G RAM。本实验设置的符号执行停机时间是20分钟，设置的内存限制是12GB。为了保证实验参数的一致性，对每一个测试程序设置的符号执行参数都是相同的。为了进行多方位的对比，除了原始的KLEE符号执行工具，本文还将实施了循环定长展开的KLEE也加入了实验。循环定长展开的KLEE在本实验中称为KLEE-Fix，定长展开的次数设定为10。

（3）实验过程

实验过程如图4.10所示。对与每一个C源程序，首先，将其编译成LLVM IR；其次，在LLVM IR上做静态分析并将结果插桩到程序中；再次，在插桩后的程序上根据设计的特殊指令进行高效符号执行；最后，检查是否在未超时的情况下发现程序错误，从而生成测试用例。

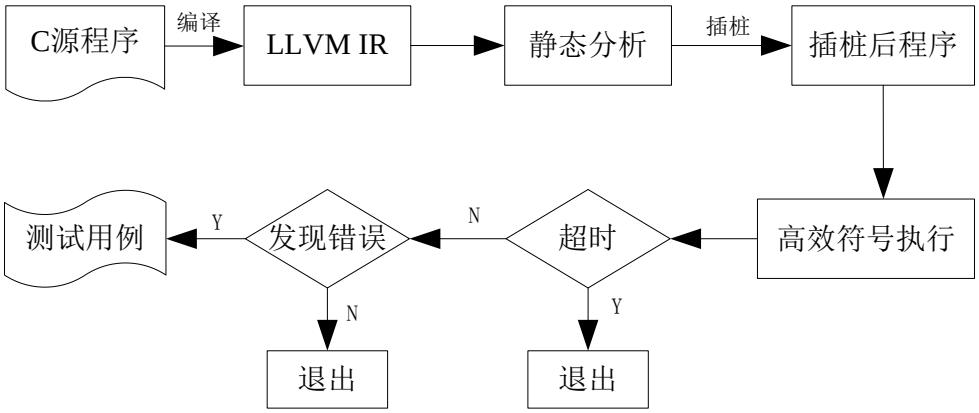


图 4.10　结合静态程序分析的高效符号执行技术实验过程

4.5.2　结果分析

表4.1为FastSE、KLEE-Fix与KLEE在测试实例时的超过时间限制的实例数量、发现错误的实例数量以及总测试时间。从表4.1可以看出，FastSE在三个方面都相对于KLEE和KLEE-Fast都具有优势，分别比KLEE-Fix和KLEE多检测了6和14个错误实例。从超过时间限制的实例数量可以看出，KLEE在符号执行有循

[1] https://sv-comp.sosy-lab.org/2017/

环的传程序时会耗费大量的时间。如果将循环进行定长展开，符号执行时间大大减少的同时也会发现更多的错误。

表 4.1　FastSE、KLEE-Fix 与 KLEE 的错误发现数量以及总时间耗费对比

	Timeout 数量	**检测错误实例数量**	**总测试时间**
KLEE	57	43	22 小时 17 分 12 秒
KLEE-Fix	19	51	14 小时 13 分 34 秒
FastSE	11	57	12 小时 31 分 55 秒

表4.2是 FastSE、KLEE-Fix 与 KLEE 发现错误数量的交叉对比。“>”操作符表示前者发现错误而后者未发现错误的实例数量，例如 FastSE > KLEE-Fix 表示 FastSE 发现了错误而 KLEE-Fix 未发现的实例总数。从表4.2可以看出三种方法发现的错误实例互有交叉而又互有不同。因为 FastSE 虽然能够计算循环的不变式，并将相应的约束插桩进程序中，使后续符号执行的路径数量大大减少，但是不变式计算是在抽象域上进行的，在无法精确表示循环时会将得到一个弱约束。弱约束引起的后果是计算出来的测试用例可能无法触发错误。所以在循环原本展开次数很少的情况下，KLEE 确实能够触发 FastSE 所不能发现的错误。同理，因为循环的定长展开也是不完备的，所以 KLEE 也能发现 KLEE-Fix 所不能发现的错误。在大多数的情况下，只要循环次数增加，KLEE 发现错误的能力就迅速减弱。从表4.2可以看出，KLEE 只能发现 3 个 FastSE 不能发现的错误，发现 5 个 KLEE-Fix 不能发现的错误；而 FastSE 则发现了 17 个 KLEE 不能发现的错误，发现了 9 个 KLEE-Fix 不能发现的错误。

表 4.2　FastSE、KLEE-Fix 与 KLEE 发现错误数量交叉对比

FastSE > KLEE-Fix	**FastSE > KLEE**	**KLEE-Fix > FastSE**
9	17	2
KLEE-Fix > KLEE	**KLEE > FastSE**	**KLEE > KLEE-Fix**
13	3	5

4.6　本章小结

本文研究并提出了一种基于抽象解释的高效符号执行技术。给定待分析的源程序，首先解析出程序的控制流图。区别于其他符号执行技术，新技术通过静态程序分析方法，从程序的控制流图中计算出循环程序的不变式，然后对程序进行插桩用循环不变式代替循环，形成新的控制流图。在新的控制流图上进行符号执

行，符号执行的路径将大大减少。通过对 100 个测试程序进行实验并分析可以得出本文提出的方法 FastSE 在超过时间限制的实例数量、发现错误的实例数量以及总测试时间三个方面均优于原始符号执行工具 KLEE 以及对循环进行定长展开的 KLEE-Fix。实验论证了本文方法在符号执行有循环的程序时的有效性。

第五章　基于细粒度变异的导向模糊测试技术研究

模糊测试已被证明是一种非常强大的软件漏洞测试方法，是工业界和学术界研究的热点。但是模糊测试一样有它的局限性，盲目的随机的变异测试用例很难到达测试程序的特定路径，导致一些漏洞很难被触发。在源代码软件的漏洞挖掘中，一些可疑的漏洞触发点可以通过静态分析获取。所以，如何动态的产生测试用例到达可疑漏洞触发点是一项重要的研究内容。现有的主要的导向测试研究都是基于符号执行的[48, 49, 51, 54, 72, 116–118]，又可称为导向白盒模糊测试。但是导向符号执行在程序分析和约束求解上耗费了大量的时间。在每一轮的测试中，为了到达目标区域，动态符号执行需要利用程序判断哪些分支需要反转，并沿着路径收集约束信息，最后调用约束求解器生成新的输入。而模糊测试不需要约束求解，仅仅通过变异生成测试输入。在相同的时间内，模糊测试产生的输入比动态符号执行高几个数量级。此外，符号执行还有其他的问题如环境交互问题和循环问题[119]。Macel Bohme[74] 提出了一种基于 AFL（American fuzzy lop）[70] 的导向模糊测试工具 AFL-go。此方法设计了一种用于衡量测试用例到目标区域的距离测度以及一种能量分配策略；随着测试时间的增加，AFL-go 通过增加距离近的测试用例的变异次数，减少距离远的测试用例的变异次数，从而完成目标区域导向。但是 AFL-go 的变异还是具有盲目性，导向效率不高的缺点。

本章提出了一种细粒度变异的导向模糊测试方法。该方法首先 AFL-go 收集测试用例；然后利用时间递归神经网络（Long Short-Term Memory，LSTM）训练出一个模型，用于判断哪些字段对靠近目标区域其关键作用，同时收集每个字段的权重；在动态运行测试用例之前，利用上述模型判断当前测试用例的关键字段并根据字段权重进行细粒度变异。本文方法能够更细粒度的变异指定字段，很大程度上消除 AFL 变异的盲目性，从而能够提高导向模糊测试的效率。

5.1　反馈式模糊测试技术介绍

模糊测试是现在最流行的软件测试技术，在很多复杂的程序中发现了很多安全漏洞。模糊测试的主要思想是不间断的产生新的畸形输入让程序执行，以期发现程序的未知行为、崩溃或者异常等行为。通常 fuzzer（模糊测试工具）接受一组初始种子输入，通过随机变异或者其他的人为定义的变异策略产生大量的畸形输入反馈给程序执行，直到满足设定的停机条件。但是若遇到输入的格式特别复杂的情况，有时会需要产生数百万次的畸形输入的输入才能触发程序异常。

AFL（American fuzzy lop）[70] 是目前为止最先进的反馈式 fuzzer，由 Michał Zalewski 于 2015 年开发完成并开源授予大众使用。相对于其他的 fuzzer，AFL 采

用了多种数据变异策略和减少资源耗费的设置，并且不需要复杂的配置，能无缝的处理复杂真实软件，例如图像处理软件和文件压缩库软件。

AFL 的优势主要在于其利用遗传算法对测试用例进行优化选择。AFL 维护一个种子测试用例队列，凡是能够提升代码覆盖率的测试用例都将作为种子测试用例，并且增加其新一轮的变异次数；反之，则丢弃。同时，该队列会通过一系列的筛选策略进行优化，以保证优先测试最优测试用例。AFL 已经成功的在很多开源软件中发现了很多安全漏洞[119]，例如：Mozilla Firefox，ffmpeg，OpenSSL[70] 等。

AFL 进行模糊测试测试的过程如算法5.1所示。算法输入为待测程序 P、初始种子测试用例集以及每个样本的变异次数 limit，第 4 到 12 行描述的是输入文件变异的详细过程，这里只描述了以字节为单位的变异。除了字节变异，AFL 还包含如比特反转、随机替换 / 插入、已知字典、边界值等变异方式。另外，还采用了组合变异方法，将简单变异随机串联成为复杂变异，从而提高了变异样本覆盖新路径的能力。通过执行变异后的输入文件获取程序的结果 result 以及路径覆盖信息 coverage（第 13 行）；如果 result 是崩溃信息，则将其加入到 MaliciousInputs 中（第 14-16 行）；如果路径覆盖 coverage 增加，则将对应输入 input 加入到种子序列中（第 17-19 行）。循环执行上述过程，直到时间耗尽（第 21-23 行）。

AFL 利用 fork server 机制增加执行效率。linux 程序在执行到 main 函数之前会经过三个步骤：内核加载程序文件、调用 libc_start_main 以及初始化必要的数据结构和线程环境。一般的 fuzzer 在每次执行测试用例时都会完整的执行此三个步骤，造成模糊测试速度缓慢。而 AFL 在程序执行到 main 函数时，封存此时的状态，当程序再次执行测试用例时，利用 linux 的 fork 机制复制一个完全相同的进程，从而省略了上述的三个步骤。以执行 binutils 程序为例，AFL 一秒钟能执行 2000 次个测试用例而一般的 fuzzer 只能执行不到 100 次。

5.2 基本框架设计

本文方法的目的是使用细粒度的变异方法尽快的生成能够到达目标区域的测试用例以挖掘漏洞或者验证程序的安全性。方法的前提条件有两个：(1) 指定目标区域，在源代码中以危险操作语句的行数和文件名称表示 (例如 valid.c:6410)；(2) LLVM bitcode 插桩，用于编译时计算每个基本块到目标区域的距离，距离的定义见5.3.1节。

程序执行的路径由输入确定，所以若要控制程序执行到特定区域，相应的输入满足一定条件。图5.1是一个简单的示意图，右半部分是程序的执行路径，左半部分是程序的输入，若想要控制程序执行到关键区域 t，则需要 x_1 和 x_3 满足一定条件；在本章方法中，x_1 和 x_3 被称为关键字段。这里的字段是指测试用例中每个比特的位置。

算法 5.1 AFL 模糊测试算法

Input: seeds，待测程序 P，每个样本变异次数 limit
Output: 畸形测试用例 MaliciousInputs

```
1:  c1 = 0
2:  for seed in seeds do
3:          while c1 < limit  do
4:                  input = seed
5:                  length = len(seed)
6:                  mutations = RandInt(length)
7:                  mut = 0
8:                  while mut < mutations do
9:                          byte = RandInt(length)
10:                         mutate(input,byte)
11:                         mut = mut+1
12:                 end while
13:                 result，coverage = execute(P,input)
14:                 if result is crash then
15:                         MaliciousInputs.add(result)
16:                 end if
17:                 if isIncreased(coverage) then
18:                         seeds.add(input)
19:                 end if
20:         end while
21:         if timeout() then
22:                 break
23:         end if
24: end for
```

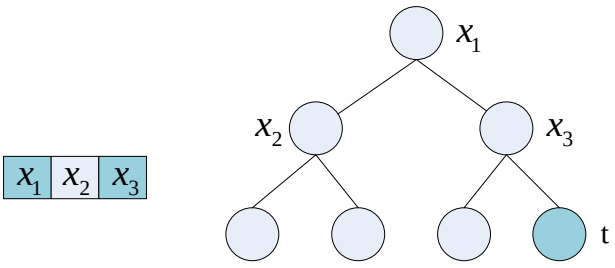


图 5.1　关键字段解释

细粒度导向模糊测试基本框架主要包括训练和测试两部分，如图5.2所示。

训练的目的是产生一个模型；给定一个测试用例，此模型能够输出哪些关键字段能够使测试用例与目标区域的距离减小；同时训练计算每个关键字段的权重。初始测试用例经过变异生成新的不同的测试用例，此处变异为 AFL 的原始变异；测试用例通过执行引擎生成执行迹，计算每个执行迹到目标区域的距离，并与原始测试用例做比较；如果距离减小，则将测试用例改变的位置 $x \oplus x'$ 与相应的改变的距离输入到 LSTM 网络（Long Short Term）以训练模型；在利用 LSTM 网络训练的同时累积各个位置距离的变化值以计算权重。

测试的目的是利用训练的模型和字段权重引导模糊测试执行到目标区域。测试阶段的测试用例是训练阶段中与目标区域距离较小的一部分测试用例；利用基于字段权重的能量调度策略（5.4节）进行变异生成新的测试用例；经过执行引擎生成执行迹并测量距离，若距离减小则保留测试用例以待再次变异，反之丢弃，重复测试过程直到满足设定的停机条件。

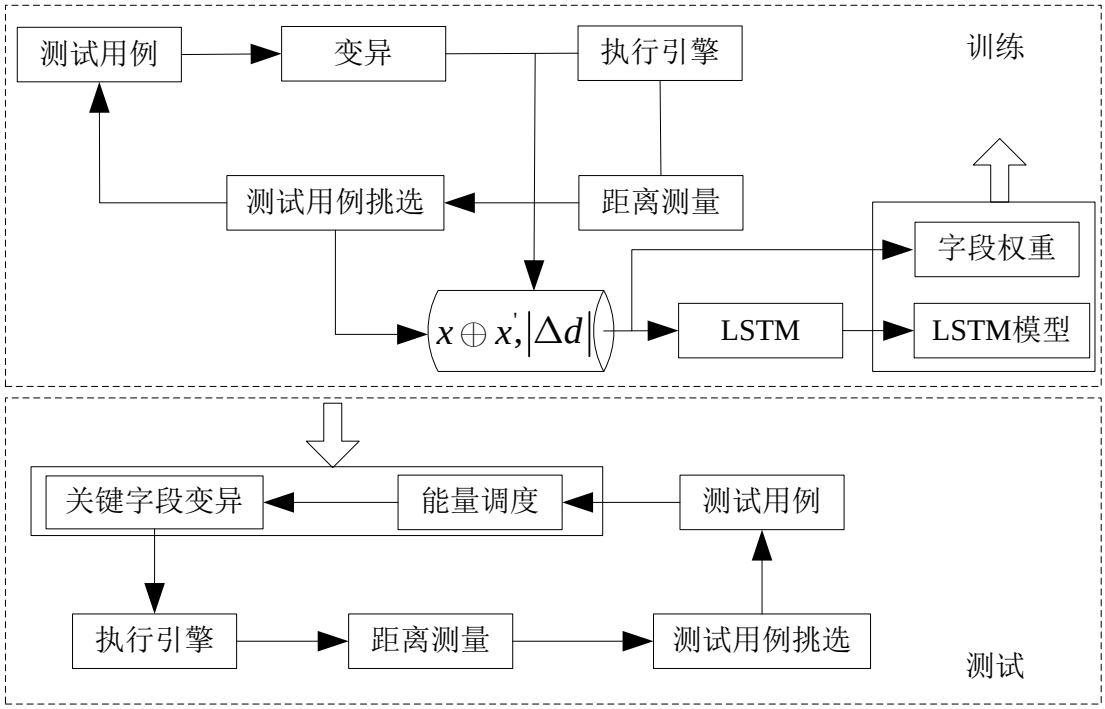


图 5.2　细粒度导向模糊测试基本框架

5.3　基于 LSTM 的关键字段获取与权重计算

本节对应图5.2的训练部分，目的是获取对于距离减少起作用的关键字节区域，以及每个 bit 的权重，主要包括三个部分：衡量测试用例执行迹与目标区域距离的测度、基于 LSTM 的关键字段获取架构以及字段的权重计算。

5.3.1 距离测度

本节采用 AFL-go[74] 的距离测度作为产生测试用例的工具。AFL-go 是一个用于补丁测试和崩溃重现的工具，在静态分析的基础上能有效的调度 AFL 逼近目标区域，本章方法采用了其距离测度表示测试用例到目标区域的距离。其中，每一个目标区域在源代码中表示为一行代码，在软件的自动测试中，程序执行到某行代码和执行到对应的基本块意义相同，所以到某行代码的距离和到代码所在基本块的距离相同。

5.3.1.1 函数距离

函数距离指的是在函数调用图中两个函数的距离。函数 n 和 n' 的距离记为 $d_f(n,n')$，表示在函数调用图中两函数之间边的数量。则函数 n 到目标函数集 T_f 之间的距离 $d_f(n,T_f)$ 可以定义为 n 到 T_f 中所有函数距离的调和平均数，如式 (5.1) 所示。

$$s(X)=\begin{cases} undefined, & 如果R(n,T_f)=\emptyset \\ [\sum\limits_{t_f\in R(n,T_f)} d_f(n,t_f)^{-1}]^{-1} & 其他 \end{cases} \tag{5.1}$$

其中，$R(n,T_f)$ 表示在 T_f 中和 n 具有可达关系的函数。调和平均数是总体各统计变量倒数的算术平均数的倒数。主要是用来解决在无法掌握总体单位数（频数）的情况下，只有每组的变量值和相应的标志总量，而需要求得平均数的情况下使用的一种数据方法。调和平均值的计算公式如式 (5.2) 所示。相对于算术平均数，在多个目标情况下，调和平均能够区分距离一个目标较近离另外一个目标较远的点与多个目标中间点。如图5.3所示，假设每个线段代表距离 1，则中间三个点到 t_1 和 t_2 的算术平均值距离都是 2；以调和平均数计算距离，如果一个点有一端比较靠近的某个目标，其距离比中间点的距离要小。本文方法的目标是要尽可能的到达目标区域，所以调和平均数较为合适。

$$H_n=\frac{n}{\sum_{i=1}^{n}} \tag{5.2}$$

5.3.1.2 基本块距离

虽然在程序的全局控制流图计算基本块的距离非常精确，但是全局的控制流图非常复杂，路径搜索和距离求解较为复杂，所以过程间、全局的基本块距离在这里使用过程内的基本块距离和函数距离去近似。如果函数距离越小，则函数之间的基本块的距离也越小。在一个函数 i 的控制流图 G_i 中，设定 $d(m_1,m_2)$ 为 m_1

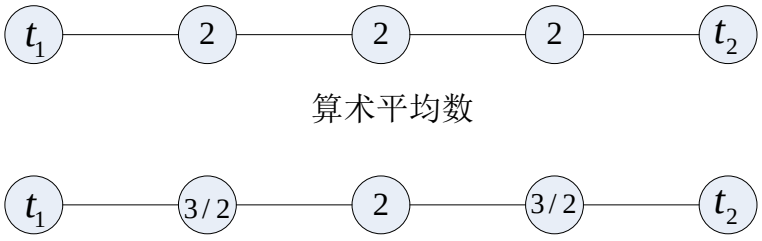


图 5.3　调和平均数和算术平均值区别

和 m_2 之间边的数量；$N(m)$ 为在基本块 m 中调用的函数中，与 T_f 函数具有可达关系的集合，即 $N(m) = \{n|R(n, T_f) \neq \emptyset\}$；$T$ 表示在 G_i 中所有包含函数调用且此函数和 T_f 中函数具有可达关系，即 $\forall m \in T.N(m) \neq \emptyset$。基于以上设定，每个基本块 m 与目标基本块集 T_b 的距离定义如式 (5.3) 所示。式中 $c = 10$ 是一个常量，用于放大函数之间的距离以近似基本块之间的距离，$d_b(m, Tb)$ 中的 m 表示所有的 G_i 中的任意的基本块，G_i 函数调用图 CG 中任意函数的控制流图。

$$d_b(m, Tb) = \begin{cases} 0, & \text{如果} m \in T_b \\ c \cdot \min\limits_{n \in N(m)} (d_f(n, T_f)), & \text{如果} m \in T \\ [\sum\limits_{t \in T} (d_b(m, t) + d_b(t, T_b)^{-1}]^{-1} & \text{其他} \end{cases} \tag{5.3}$$

5.3.1.3　测试用例与目标区域之间的距离

测试用例与目标区域之间的距离可以用程序执行测试用例产生的执行迹与目标基本块集的距离表示，距离越小越可能击中目标基本块。$\varphi(s) = b_1, b_2 ... b_n$ 表示测试用例 s 的执行迹，b_i 为程序执行的基本块。测试用例与目标区域之间的距离如式 (5.4) 所示。

$$d(s, T_b) = \frac{\sum_{m \in \varphi(s)} d_b(m, T_b)}{|\varphi(s)|} \tag{5.4}$$

归一化后的测试用例与目标基本块集的距离如式 (5.5) 所示。

$$\hat{d}(s, T_b) = \frac{d_b(s, T_b) - minD}{maxD - minD} \tag{5.5}$$

其中

$$minD = \min_{s_i \in S}[d(s_i, T_b)]$$
$$maxD = \max_{s_i \in S}[d(s_i, T_b)] \tag{5.6}$$

归一化后，测试用例与目标区域之间的距离 $\hat{d} \in [0,1]$。

5.3.2 距离的获取方式

AFL 是运行非常快速的模糊测试器，如果在运行时动态测量测试用例到目标区域之间的距离会大大降低模糊测试的效率，所以本文采用在编译时将基本块之间的距离插桩到代码的方式来减少 AFL 动态运行负荷[74]。本文采用 LLVM 编译器对源码进行编译和插桩，用于距离测量的的静态插桩如图5.4所示。

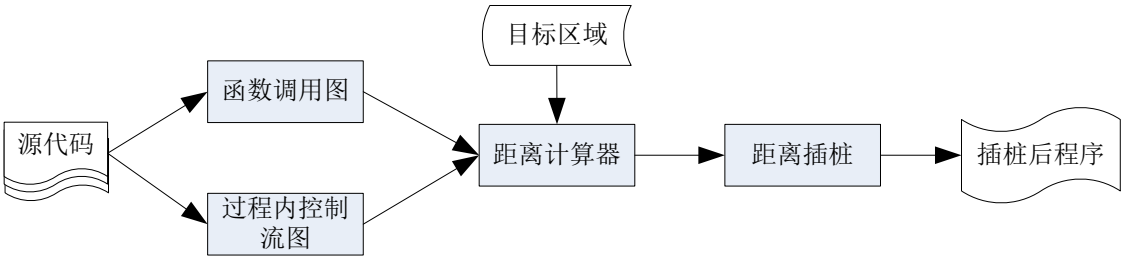


图 5.4 用于距离测量的的静态插桩

静态插桩的具体过程如下：

（1）产生程序的函数调用图和相应的过程内控制流图。函数调用图用函数声明表示；过程内控制流图用进入函数的第一个语句表示。二者通过编写相应的 LLVM Pass 生成。

（2）基本块之间的距离通过函数之间的距离以及过程内基本块的距离一起计算，详细的计算方式见5.3.1节。基本块之间距离的计算是通过调用 Djikstra 算法计算最短距离完成的。

（3）在每个基本块的跳转语句之后插入代码片段用于记录覆盖的控制流边。AFL 使用 64kb 的共享内存存储边在执行过程中的遍历情况，每一条边对应一个字节。在 64 位的架构中，额外增加了另外 16kb 内存：8kb 用于记录距离值另外 8kb 用于记录经过的基本块距离。静态插桩主要实现了两种功能：记录基本块到目标区域的距离以及将遍历的基本块记录到共享内存。

5.3.3 训练过程

5.3.3.1 目标函数

训练的目的是建立一个模型，给定一个测试用例，此模型能够输出一个位图用于表示哪些关键字段的改变能够使测试用例和目标区域的距离减少。因为测试用例是变长的，所以此模型可以被描述为一个函数簇，如式 (5.7) 所示。此函数簇的输入是任意数量的输入位置，输出的是各个位置的变异引起测试用例和目标区域距离减少的概率。

$$\{f_k : \{0x00, 0x01, ..., 0xFF\}^k \rightarrow [0,1]^k | k \in N\} \tag{5.7}$$

式 (5.8) 是以字节为单位判断是否有距离改变，但程序经常会用一个 bit 进行分支判断，所以本文进一步设定了 bit 级的目标函数，如式 (bit 目标模型) 所示。

$$\{f_k^{'} : \{0,1\}^{8k} \rightarrow [0,1]^{8k} | k \in N\} \tag{5.8}$$

在测试阶段，训练出的模型会首先根据当前测试用例相对于变异前的测试用例所改变的位置，去判断是否会引起距离的减少；若减少距离减少则执行此测试用例，反之丢弃，决策函数如式 (5.9) 所示。式中 x 表示测试用例，$x^{'}$ 表示变异后的测试用例，$\oplus$ 表示逐位异或运算，$(x \oplus x^{'})$ 表示测试用例变异的位置，α 是一个阈值，控制改变 bit 的最小数量。式 (5.9) 的主要思想是判断哪些关键 bit 上的改变才能够减小到目标区域的距离。

$$\sum_k (f^{'}(x \oplus x^{'})) > \alpha \tag{5.9}$$

为了训练目标函数 $f^{'}$, 需要测试用例 x，变异后的测试用例 $x^{'}$，x 到目标的距离 $d(x)$ 以及 $x^{'}$ 到目标之间的距离 $d(x^{'})$；结合这四个参数可以设定一个生成一个有监督的数据集，如式 (5.10) 所示。

$$xy = \{(x, x \oplus x^{'}) | \triangle (d(x), d(x^{'})) > 0\} \tag{5.10}$$

5.3.3.2 训练模型选择

因为测试用例的长度是可变的，且 bit 之间存在关联，本文使用循环神经网络（Recurrent Neural Network，RNN）训练决策函数。RNN 非常适用于处理像程序测试用例这样的序列数据[120]。因为 RNN 的记忆功能，所以 RNN 在处理有格式的

文件输入时非常有用。RNN 已经成功的应用于统计机器翻译当中[121, 122]，本章要处理的问题和此相似，因为测试用例也可以当做是一种语言。

但是 RNN 很难处理较长的序列，所以本文采用时间递归神经网络（Long Short-Term Memory，LSTM）。相对于 RNN，LSTM 在最顶层增加了一条名为”cell state“的信息传送带同时，所以能够处理更长的序列。同时，LSTM 丢弃超过生命周期的序列记忆，以增加处理更长序列的能力。LSTM 的一个循环单元（recurrent unit）的状态更新和输出如式 (5.11) 所示。

$$h_t, o_t = f(x_t, h_{t-1}) \tag{5.11}$$

将式 (5.11) 分解，如下式 (5.12) 所示。其中，σ 是 Sigmoid 函数，W_* 为学习的权重向量，b_* 是学习的误差向量。f_t 是忘记门层，i_t 是输入门层二者决定是否保留或者丢弃当前输入。通过三种门交织处理使得 LSTM 能够处理更长的序列。

$$\begin{aligned} f_t &= \sigma(W_f \cdot [x_t, h_{t-1}] + b_f) \\ i_t &= \sigma(W_i \cdot [x_t, h_{t-1}] + b_i) \\ C_t &= f_t \times C_{t-1} + i_t \times tanh(W_C \cdot [x_t, h_{t-1}] + b_C) \\ o_t &= \sigma(W_o \cdot [x_t, h_{t-1}] + b_o) \\ h_t &= o_t \times tanh(C_t) \end{aligned} \tag{5.12}$$

5.3.3.3 关键字段权重计算

本文用一个哈希表存储所有 bit 改变引起的距离的改变，每个 bit 的权重计算如 (5.13) 所示。其中，x 和 x' 表示变异前测试用例和变异后的测试用例，$sum(x \oplus x')$ 表示 x' 相对于 x 变化的字节数量，$\triangle(d(x), d(x')$ 表示 x' 相对于 x 减少的距离。式 (5.13) 表示将 $\triangle(d(x), d(x')$ 平均的分配在改变的每个 bit 上。训练结束后将权重累积到所有 bit 位置上，对权重进行归一化，得到 bit 权重的最终结果。

$$\omega_i^k = \begin{cases} \omega_i = \triangle(d(x), d(x'))/sum(x \oplus x'), & \text{如果} i \in \{x \oplus x'\} \\ 0 & \text{其他} \end{cases} \tag{5.13}$$

$$\omega_i = \sum_{k=1} \omega_i^k \tag{5.14}$$

在实际训练的过程中，并不是所有的 bit 的权重都不相同，本文将具有相同 bit 权重的 bit 集合称之为 bit 集。

5.3.3.4 关键字段权重与程序执行路径的关系

本节利用程序 Listing 5.1论述关键字段权重与程序执行路径之间的关系。图5.5是 Listing 5.1的执行树，每个节点代表一个基本块，第 6 行的代码和图中红色节点相对应，这里将此基本块表示为 t；叶子节点下面的字符 (a,b,c,d,e,f,g,h) 代表的是执行到相应节点的路径。若要执行 h，需要的条件是 $x1>0 \wedge x3>100 \wedge x7>100$。

假设当前的测试用例 $s=(-1,-2,1,-200,1,1,1)$，执行的路径是 a，按照5.3.1节的距离计算方式，$d(s,a)=3$。任意修改 $x2,x4,x5$ 所生成新测试用例 s'，其与 h 的距离 $d(s,s')$ 都不会减少。若要减少到 h 的距离，$x1$ 必须要大于 0，如此 $x1$ 的权重则会增加，$x1$ 的权重一定大于 $x2,x4,x5$。s 经过变异后生成 $s'=(1,-2,1,-200,1,1,1)$，执行的路径是 e。对于 s'，若要减少到 h 的距离就必须要变异 $x3$，从而导致 $x3$ 的权重增加。

在对大量样本进行训练的情况下，关键字段权重能够反应测试用例和输入的关系，即权重大的字段能够控制执行树的节点。对于每一个测试用例，根据其关键字段的权重能够更容易的到达目标区域。

Listing 5.1　权重与程序执行路径的关系实例程序

```
1  void f(int x1, int x2, int x3, int x4, int x5, int x6, int x7)
2  {
3        if(x1 > 0){
4              if(x3 > 10){
5                    if(x7 > 100)
6                          target();
7              }else{
8                    if(x6 > 15){
9                          normal();
10                   }else
11                         normal();
12             }
13
14       }else{
15             if(x2 < -1){
16                   if(x4 < -125){
17                         normal();
18                   }else
19                         normal();
20             }else{
21                   if(x5 < -256){
22                         normal();
23                   }else
24                         normal();
25             }
26       }
27 }
```

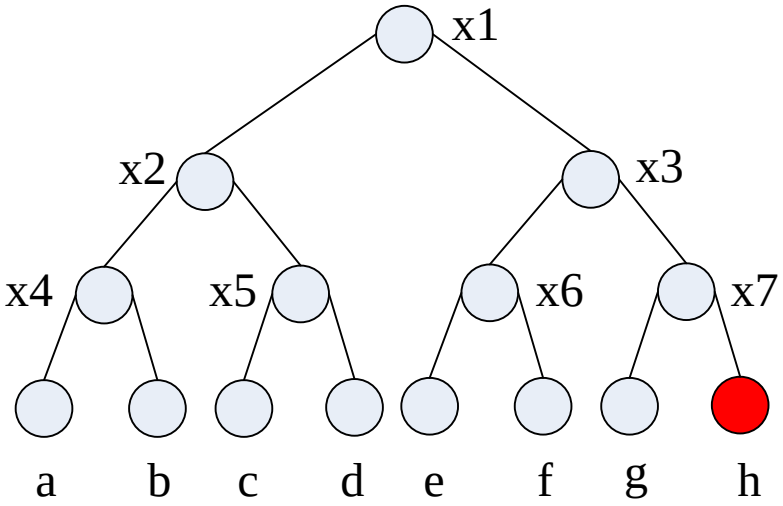


图 5.5　Listing 5.1的执行树

5.4　动态测试

本节重点论述在 AFL 的原有能量调度基础上，基于关键字段和模拟退化算法设计能量调度策略。本章使用的能量的含义是指模糊测试中对一个测试用例的变异次数。

5.4.1　细粒度变异的测试用例生成过程

图5.6是本文采用的通过细粒度变异生成测试用例的过程。对于测试用例 s，LSTM 模型输出使距离减少的 bit 集，对照全局权重表形成归一化后的 bit 集，每个 bit 集的权重用 ω_i' 表示。对于每一个测试用例基于模拟退火框架生成一个能量 p，按照 bit 集的权重重新计算每个 bit 集的权重 p_i，能量越大表明该 bit 集变异的次数越多。通过对靠近目标区域的测试用例分配更多的能量，以及对每个 bit 集根据权值进行细粒度变异能够使测试用例以更大的可能性更快的到距离更近的测试用例，以此增加导向性模糊测试的效率。

5.4.2　细粒度变异能量分配策略

AFL 通过测试用例的执行时间、覆盖的基本块数量以及测试用例的深度（测试用例相对与初始测试用例变异的次数）来决定当前测试用例的能量。AFL 初始的能量分配策略是以代码覆盖率为导向的，并不适合特定目标导向。本节基于字段权重，介绍一种模拟退火框架的能量分配策略。此策略的主要思想是对距离目标区域较近的测试用例变异能更容易击中目标区域。具体实施时，为距离目标区

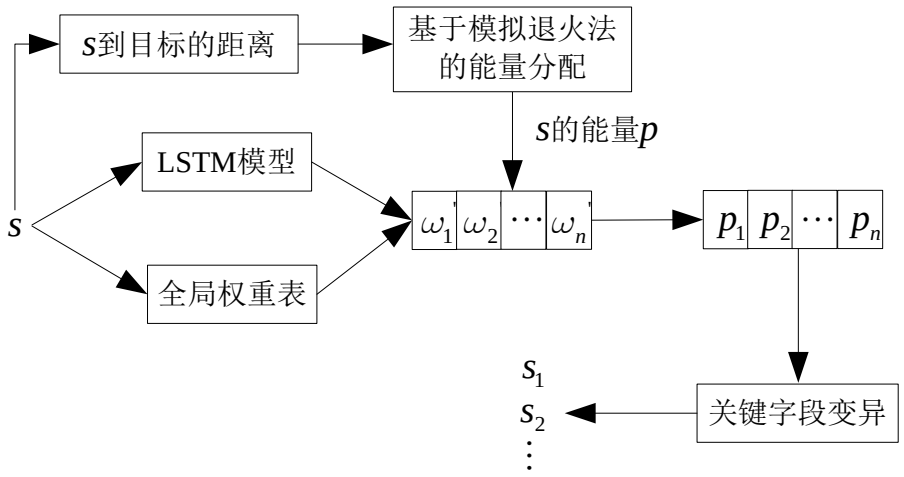


图 5.6　细粒度变异的测试用例生成过程

域较近的测试用例分配更高的能量，从而能以更高的可能性生成到达目标区域的测试用例。

模拟退火法[123] 的出发点是基于物理中固体物质的退火过程与一般组合优化问题之间的相似性。模拟退火算法从某一较高初温出发，伴随温度参数的不断下降，结合概率突跳特性在解空间中随机寻找目标函数的全局最优解，即在局部最优解能概率性地跳出并最终趋于全局最优。模拟退火算法是一种通用的优化算法，理论上算法具有概率的全局优化性能。温度是模拟退火的一个重要参数，随着温度的降低，较差解决方案的接受率也降低。在算法开始运行时，初始温度 $T = T_0 = 1$，此时算法接受较差解决方案的概率较高，当 T 接近于 0 时，算法退化为经典的梯度下降算法。

将模拟退火算法的框架应用到导向模糊测试中想要达到的效果是：测试开始时即温度较高时，允许以较高的概率变异那些距离目标区域较远测试用例，随着时间的推移这种概率越来越低，到时间趋于无穷时只变异距离目标区域最近的测试用例。现在最流行的冷却方案是指数冷却方案[123]，如式 (5.15) 所示。其中，$\alpha < 1$ 是个常量，α 的区间为 $[0.8, 0.99]$。

$$T_{exp} = T_0 \cdot \alpha^k \tag{5.15}$$

在用模糊测试方法挖掘漏洞时，一定会设定一个时间限制 t_x。在使用模拟退火框架设计能量分配策略时，也需要设定一个 t_x。在 t_x 之前是导向模糊测试的搜索路径阶段，在此阶段分配给距离较远的测试用例较大的能量，以搜索更多的路径。在 t_x 之后导向模糊测试进入收敛阶段，在此阶段分配给距离最近的测试用例

的能量几乎达到最大，使产生的测使用例更好的逼近目标区域。在到达 t_x 时，模拟退火算法等价于梯度下降算法，距离目标区域最近的测试用例分配的最高的能量，让 AFL 着重变异和测试最有希望击中目标区域的测试用例。假设设置经过 k 轮的迭代之后 $T_{exp} = 0.1$，则在时间 t 的温度 T_exp 可以通过式 (5.16) 的推导获得。同样，可以将 T_{exp} 设置为其他的值，T_{exp} 越小温度降低的越快。

$$\begin{aligned} &\alpha^{k_l} = 0.1 && \text{当经过}k_l\text{轮迭代之后} \\ &k_l = log(0.1)/log(a) && \text{求解}k_l \\ &T_{exp} = \alpha^{t/t_x \cdot (log(0.1)/log(a))} && \text{将}k_l\text{代入} \\ &T_{exp} = 10^{-t/t_x} && \text{化简} \end{aligned} \tag{5.16}$$

模拟退火的温度参数随着时间的增加而减少的，而基于模拟退火框架的能量分配的目的是距离近的测试用例的能量随着时间的增加越来越高，距离大的测试用例的能量随时间的增加越来越低。所以在式 (5.16) 的基础上可定义模拟退火的能量分配如式 (5.17) 所示。其中 c_1 是一个可调节的常量，控制着搜索阶段距离大的测试用例所分配的能量，c_1 越大距离大的测试用例能量越大。典型的，可以设置 $c_1 = 0.5$；在开始测试时测试距离为 1 的测试用例的能量为 0.5。

$$p(m, T_b) = (1 - \hat{d}(s, T_b)) \cdot (1 - T_{exp}) + c_1 T_{exp} \tag{5.17}$$

在式 (5.17) 的基础上，可定义每个 bit 集的变异能量如式 (5.18) 所示，其中，$\omega_i^{'}$ 测试用例关键字段中每个 bit 的权重。

$$p_i = \frac{\omega^{'}}{\sum_i \omega^{'}} \cdot p(m, T_b) \tag{5.18}$$

将式 (5.18) 应用到 AFL 原有的能量策略上，最终的能量分配如式 (5.19) 所示。其中，c_2 是一个常量，用于控制不同距离的能量分配，c_2 越大，距离大的测试用例分配的能量越小，此数值可以在测试时根据不同的目的具体调整。

$$\hat{p}_i = p_{afl}(s) \cdot 2^{10(p_i - c_2)} \tag{5.19}$$

5.5 实验结果与分析

5.5.1 实验设计

（1）实验目的

为了测试基于细粒度变异的导向模糊测试方法的有效性，本节在测试效率以及已知漏洞的挖掘数量两个方面上进行了实验。

（2）实验环境

本节主要针对 readelf[124] 进行实验。readelf 是 Linux 下 binutils 中用于分析 ELF 文件的命令，本实验测试的 readelf 对应的 binutils 版本是 2.28，测试软件的基本情况如表5.1所示。根据 CVE 数据库的记录 readelf 包含 14 个漏洞。除了将这些漏洞发生的代码行作为导向的目标区域，还在 readelf 代码中随机标记了 36 行代码作为目标区域。本实验将就目标区域的覆盖数量、目标区域的击中次数以及漏洞的发现数量作为标准衡量本文方法的可行性和有效性。

表 5.1　测试软件基本信息

软件版本	CVE-ID	CVE 数量	总标记数量
readelf2.28	CVE-2017-14333，CVE-2017-15996，CVE-2017-16830 CVE-2017-6965，CVE-2017-6966，CVE-2017-6969 CVE-2017-7209，CVE-2017-8398，CVE-2017-9038 CVE-2017-9039，CVE-2017-9041，CVE-2017-9042 CVE-2017-9043 CVE-2017-9044	14	50

本实验使用 Keras[125] 作为前端训练预测模型，选择 Tensorflow[126] 作为 Keras 的底层后端。因为训练测试用例的长度是不同的，而且存在非常大的测试用例一个输入文件可能达到 200KB，所以本实验将大于 10KB 的文件以 10KB 为单位进行分段。本试验采用 Nvidia GTX970 GPUs 训练 12 小时，使用绝对平均误差（Mean Absolute Error，MAE）作为损失函数，使用 Adam 优化器[127] 以 5×10^{-5} 的学习率训练模型。训练模时可以设定不同的输入输出文件大小，本实验采用的是 64bit。

动态导向模糊测试使用的是 Inter Xeon CPU E3-1231 v3 @ 3.40GHz，16G RAM，实验设定的搜索路径时间为 8 小时，8 小时后的时间为目标区域导向时间。

（3）实验过程

本实验的实验过程如图5.2所示，首先训练与距离相关的关键字段以及关键字段的权重，然后进行动态模糊测试。本实验的训练所使用的测试用例是由 AFL-go 产生。刚开始测试时需要搜索更多的路径，所以在运行 AFL-go 时将式 (5.17) 中的 c_1 设为 0.6，给予距离大的测试用例更大的能量，式 (5.19) 中的 c_2 参数设为 0.5。本实验运行了 AFL-go 6 个小时来收集用于训练的样本。将收集的样本按照距离的大小排序，将距离等距的划分为 10 个区间，即 $[0, 0.1), [0.1, 0.2)...[0.9, 1]$。对落入

每个区间的测试用例随机的抽取 20%。训练时，每次不放回式抽取两个测试用例，提取每个测试用例与目标区域的距离，按照式 (5.20) 构造样本 *xy*。

$$xy = \{(x, x \oplus x^{'}) || d(x, x^{'})| > 0\} \tag{5.20}$$

在动态模糊测试阶段，针对每个测试用例，LSTM 模型会输出需要变异的关键字段。关键字段结合全局字段权重表，用于指导字段变异和能量调度，从而近一步指导动态模糊测试的导向性。

5.5.2 结果分析

AFL 在判断测试用例是否重复时使用两个标准：（1）程序在执行不同测试用例经过的基本块不相同；（2）经过的基本块相同，但遍历基本块的次数不相同。对于一些有漏洞的危险区域，即使程序执行到目的基本块也不能保证能触发漏洞，例如由循环写造成的缓冲区溢出漏洞，触发此类型的漏洞就需要多次遍历循环内部的基本块。所以可以认为，在测试用例经过的基本块不变的情况下，若能够增加目的区域的遍历次数就更可能触发漏洞。本节将这种能够到达目标区域，经过的基本块相同，但是执行的次数不同的测试用例称为非重复击中目标区域测使用用例。图5.7是本章方法、AFL-go 以及 AFL 的非重复击中目标区域测使用用例数量的对比。

从图5.7上可以看出，在 8 小时之后本章方法和 AFL-go 非重复击中目标区域测试用例数量增加的速率明显增加，而原始的 AFL 则没有显著提高。经过 24 小时本章方法的非重复击中目标区域测试用例数量达到了 213 个，而 AFL-go 只达到了 127 个，AFL 只有 28 个。从非重复击中目标区域测试用例的增加速率以及总数量来看，本章方法要优于 AFL-go 和 AFL。

图5.8是本章方法、AFL-go 以及 AFL 工具在测试 readelf 时，对标记的 50 个区域的覆盖情况。从图5.8可以看出本章方法和 AFL-go 在引导模糊测试到达目标区域时具有显著的优势。在 8 个小时之后，导向目标区域的速率明显增加。本章方法在运行了 17 小时 36 分钟之后能够覆盖 35 个目标区域，AFL-go 在执行了 22 小时 21 分钟之后覆盖了 24 个目标区域，而原始的 AFL 在 24 个小时之后只能覆盖 4 个目标区域。实验表明，本章方法在导向目标区域的速率以及覆盖目标区域的数量比 AFL-go 有明显的提高。

图5.9是本章方法、AFL-go 以及 AFL 工具在测试 readelf 测试漏洞数量的对比情况。从图5.9可以看出，AFL 在 24 个小时之内没有发现任何漏洞，而在指定目标区域后，本章方法和 AFL-go 发现的漏洞数量显著增加。本章方法在运行 16 小时 38 分钟后发现了 11 个漏洞，AFL-go 在运行了 19 个小时 25 分钟后发现了 6 个

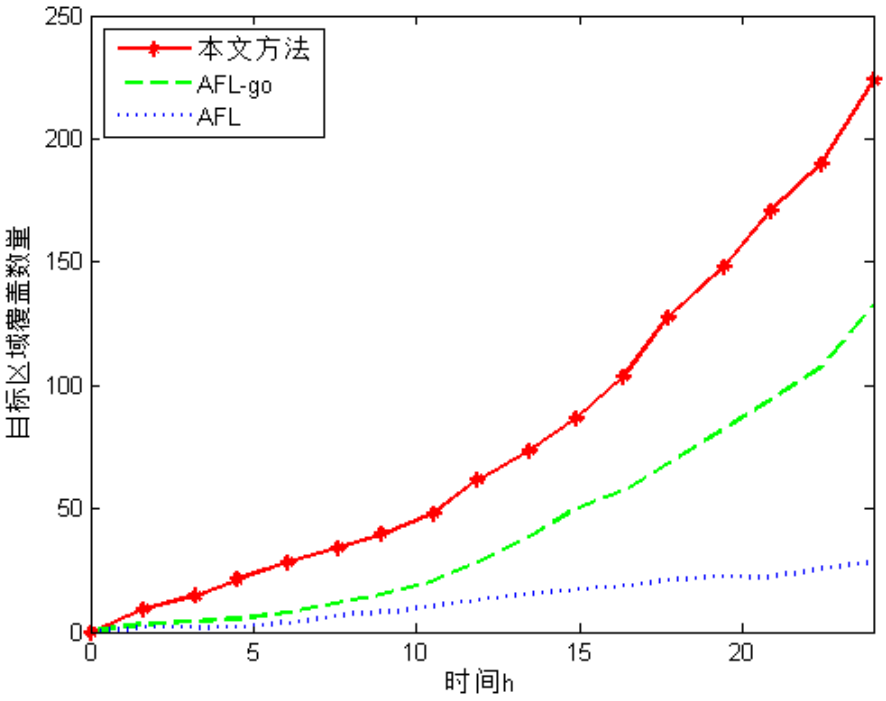


图 5.7　非重复击中目标区域测试用例数量对比

漏洞；本章方法在 5 小时 31 分钟时发现了第一个漏洞，而 AFL-go 在 8 小时 13 分钟发现了第一个漏洞；说明经过一段时间的路径探索之后本章方法能更早更快的发现漏洞，且最终发现的漏洞比 AFL-go 多出 3 个。

5.6　本章小节

本节提出了一种基于细粒度变异的导向模糊测试方法。该方法首先 AFL-go 收集测试用例；然后利用时间递归神经网络（Long Short-Term Memory，LSTM）训练出一个模型，用于判断哪些字段对靠近目标区域其关键作用，同时收集每个字段的权重；在动态测试测试用例之前，利用上述模型判断对于当前测试用例哪些是关键字段并根据字段权重定向的变异。本章方法能够更细粒度的变异制定的字段，很大程度上消除 AFL 变异的盲目性，从而能够提高导向模糊测试的效率。

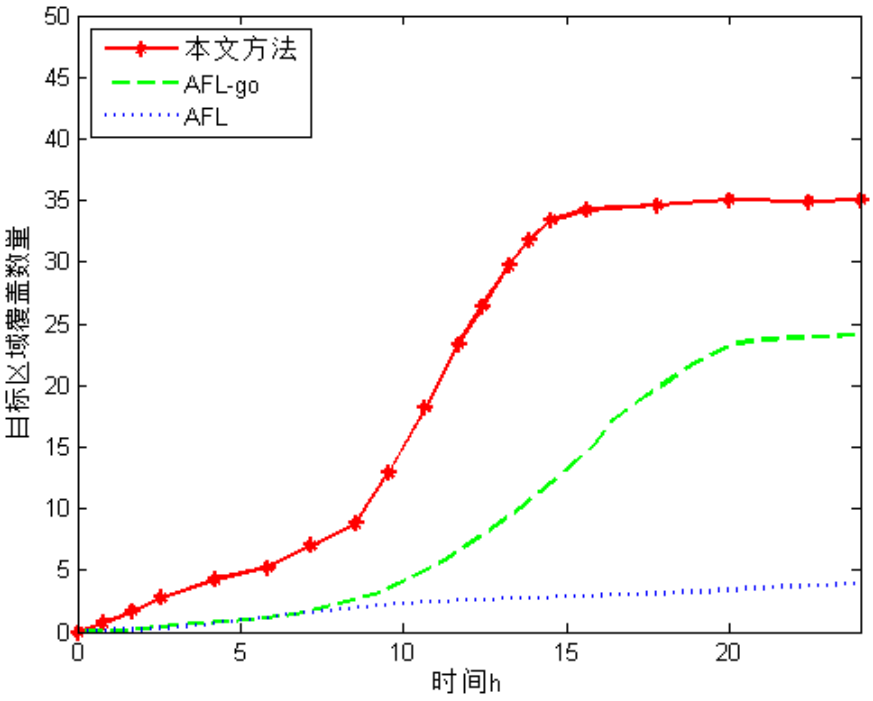


图 5.8　目标区域覆盖数量对比

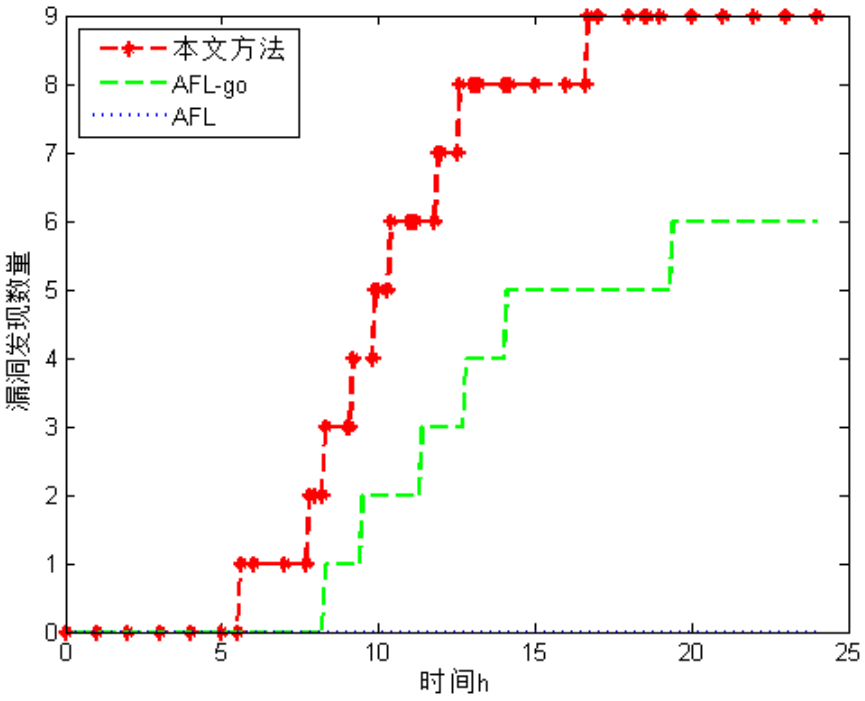


图 5.9　发现漏洞数量对比

第六章　总结与展望

6.1　论文主要工作总结

论文通过对比现有软件源代码漏洞挖掘方法及工具，分析源代码软件漏洞挖掘的发展趋势，总结了源代码软件漏洞挖掘存在四个关键问题：（1）多种类源代码软件漏洞精确静态挖掘问题；（2）机器学习算法和源代码软件漏洞挖掘的结合问题；（3）结合形式化方法缓解符号执行的路径爆炸问题；（4）源代码静态信息辅助的模糊测试导向问题。针对以上四个问题，论文研究了多种源代码动、静态分析技术，其主要工作和创新如下:

（1）提出了一种基于程序性质图的源代码软件漏洞挖掘方法。首先，利用语法解析器解析源代码，依次生成语法分析树、抽象语法树、控制流图、数据流图；然后，聚合抽象语法树、控制流图以及数据流图形成程序性质图，并且定义程序性质图的基本遍历方法；最后，根据漏洞的形成模式，组合程序性质图遍历方法检测缓冲区溢出漏洞、格式化字符串漏洞以及 Use After Free 漏洞。该方法能有效的检测此三类源代码漏洞，并具备扩展的检测其他类型源代码漏洞的能力。

（2）提出了一种基于机器学习的缓冲区溢出漏洞挖掘方法。该方法首先总结了 7 类缓冲区溢出漏洞静态特征，分别为 sink 类型、缓冲区位置、容器、索引 / 地址 / 长度复杂度、边界检测、循环 / 条件 / 函数调用深度以及是否输入可控；其次，通过扩展的程序性质图检测缓冲区溢出漏洞的各类性质并将其向量化；然后，利用有监督机器学习算法在已标记的训练集上训练分类器；最后，利用此分类器在新的源代码程序中挖掘缓冲区溢出漏洞。通过在测试集的对比实验表明，该方法比 BOMiner 多检测了 14.8 个的缓冲区溢出漏洞；通过在 Linux 的驱动程序上的对比实验表明，该方法平均比 Joern 多发现了 8.2 个缓冲区溢出漏洞。

（3）提出了一种结合静态程序分析的高效符号执行技术，用于缓解循环程序导致的符号执行路径指数级爆炸问题。该技术首先通过静态程序分析方法，从程序的控制流图中计算出循环程序的不变式；然后，对程序进行插桩用循环不变式代替循环，形成新的控制流图；最后在新的控制流图上进行符号执行。通过 100 个基准程序上进行的对比实验，本文提出的方法 FastSE 在发现错误的实例数量上，比对循环进行定长展开的 KLEE-Fix 提高了 11.8%，比原始的 KLEE 符号执行工具提高了 32.6%；在时间消耗上，分别减少了 11.9% 和 43.8%。

（4）提出了一种细粒度变异的导向模糊测试方法。该方法首先利用导向模糊测试收集测试用例；然后利用时间递归神经网络训练出一个模型，用于判断对靠近目标区域起关键作用的字段，同时收集每个字段的权重；最后，通过上述模型

判断当前测试用例的关键字段，并利用关键字段权重进行细粒度的变异生成测试用例。通过和导向模糊测试工具 AFL-go 在测试程序上的对比实验，在 24 个小时内，该方法比 AFL-go 多击中 87 次目标区域；多覆盖 11 个非重复目标区域；多检测 3 个漏洞。

6.2 下一步工作展望

由于学术水平所限，本文的研究成果还有许多进一步拓展的空间。另一方面，理论的发展技术的进步不会停止，这也会引导研究者去寻找新的更好的漏洞挖掘方法。因此，未来一段时期内需要进一步研究的主要内容包括:

（1）论文第三章中研究了基于有监督学习的缓冲区溢出漏洞挖掘方法，其静态特征需要人为指定，训练数据需要人为标定。特征的指定需要人的经验，训练数据的标定也是一件非常耗时的工作。所以探索利用深度学习的方法去自动学习漏洞特征是下一步研究方向。

（2）论文第四章中研究了结合静态程序分析的高效符号执行技术，运用抽象解释的方法求解循环程序的不变式。虽然能大大增加符号执行的效率，但同时降低了符号执行的精度，会导致产生虚假的测试用例。如何深层次的结合抽象解释和符号执行技术，在精度和效率之间达到一个更好的平衡是下一步研究的方向。此外，在已知目标区域的情况下如何进行导向符号执行是另外一个需要研究的工作。

（3）论文第五章研究了基于细粒度变异的导向模糊测试技术，但是其训练和动态测试是分离的两部分，动态测试的结果不能及时的反馈给训练过程做实时修正，如何将训练过程与动态测试过程联动起来增加测试的效率和效果是下一步的研究方向。

致　谢

行文至此，心中万分感慨。回顾博士阶段的五年学习生活，经历过认可时的激动和喜悦，也经历过失意时的苦涩和迷茫，少了一份功利，多了一份专注，这些带给我的不仅是科学思维上的磨练，更是全面素质的提升。在此，谨向多年来一直给予我知道、支持帮助的老师、同学和亲人们表示最衷心的感谢，是你们的陪伴和无私帮助激励我不断前行!

衷心感谢我的导师沈荣骏院士。沈院士虽然不在长沙亲身知道，但定期他总要从北京赶来听取学生在课题方面的进展汇报，并给出很多建设性的意见和建议，来长沙出差之际，总是挤出时间和学生交流沟通。

衷心感谢我的博士导师唐朝京教授! 我从 2013 年受教于唐老师就读博士研究生至今, 他严肃认真的治学态度、深厚的学术功底、严密的思维、渊博的知识和分析问题、提炼问题的能力, 使我在学业上受益匪浅。我在博士论文选题、研究和论文写作的整个过程, 都是在唐老师的悉心指导和严格要求下完成的。在几年的学习过程中, 不仅使我的学术水平得到了很大的提高, 在学术道德方面, 我也有了更加深刻的认识, 对自己的要求也更为严格。唐老师极富责任感、对待科研非常严谨, 这种品质深深地影响着我, 促使我形成严谨求实的学术作风, 这将成为我人生的一大笔财富。唐老师还对我的论文写作方面进行了细心的指导, 无私地传授给我论文写作的大量规律和经验。唐老师对工作认真勤恳的态度以及在学术上不断进取的精神永远是我今后工作学习的榜样! 在此谨向他表示最衷心的感谢和最诚挚的敬意!

衷心感谢张权副教授! 是您带领我走进了信息安全研究的科研殿堂。感谢王剑教授、刘俭老师、张琛老师、张磊副教授、冯超老师、李瑞林老师等诸位老师对我的指导和关心。我在教研室学习期间, 他们为我创造了良好的工作环境, 他们在学术上的执着追求以及一丝不苟的工作作风也使我受

感谢李孟君师兄、吴波师兄、解纬师兄、张博师兄、王少磊师兄、帅博师兄，王强师兄，感谢毕兴、刘毅、张斌、陈夏阳、叶嘉曦、张兴、冯冈夫、黄安琪、在同一个实验室中工作使得我们有了更多的讨论, 在相互的交流和学习中我受益良多。

深深的感谢我的家人，没有你们的支持，就没有今天的我!

谨以此文献给所有关心我支持我的亲人和朋友们!

参考文献

[1] Autodafe. http://autodafe.sourceforge.net/.

[2] 78% of Companies Run on Open Source Yet Lack Formal Policies. https://www.blackducksoftware.com/de/node/1011.

[3] Williams J, Dabirsiaghi A. The unfortunate reality of insecure libraries [J]. Asp. Secur. Inc. 2012: 1–26.

[4] CERN Computer Security Information. https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml.

[5] ITS4. http://seclab.cs.ucdavis.edu/projects/testing/tools/its4.html.

[6] Flawfinder Home Page. https://www.dwheeler.com/flawfinder/.

[7] Leino K R M, Nelson G, Saxe J B. ESC/Java user's manual [J]. ESC. 2000, 2000: 002.

[8] Dor N, Rodeh M, Sagiv M. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C [C]. In ACM Sigplan Notices. 2003: 155–167.

[9] Cousot P. The astrée static analysis tool [C]. In ES PASS Workshop, Berlin, Germany. 2007: 16–17.

[10] Delmas D, Goubault E, Putot S, et al. Towards an industrial use of FLUCTUAT on safety-critical avionics software [C]. In International Workshop on Formal Methods for Industrial Critical Systems. 2009: 53–69.

[11] Almossawi A, Lim K, Sinha T. Analysis tool evaluation: Coverity prevent [J]. Pittsburgh, PA: Carnegie Mellon University. 2006.

[12] King J C. Symbolic execution and program testing [J]. Communications of the ACM. 1976, 19 (7): 385–394.

[13] Bush W R, Pincus J D, Sielaff D J. A static analyzer for finding dynamic programming errors [J]. Software-practice and Experience. 2000, 30 (7): 775–802.

[14] Das M, Lerner S, Seigle M. ESP: Path-sensitive program verification in polynomial time [C]. In ACM Sigplan Notices. 2002: 57–68.

[15] Chen H, Wagner D. MOPS: an infrastructure for examining security properties of software [C]. In Proceedings of the 9th ACM conference on Computer and communications security. 2002: 235–244.

[16] Holzmann G J. Static source code checking for user-defined properties [C]. In Proc. IDPT. 2002.

[17] SLAM. https://www.microsoft.com/en-us/research/project/slam/.

[18] Spin - Formal Verification. http://spinroot.com/spin/whatispin.html.

[19] The CBMC Homepage. http://www.cprover.org/cbmc/.

[20] Beyer D, Henzinger T A, Jhala R, et al. The software model checker Blast [J]. International Journal on Software Tools for Technology Transfer. 2007, 9 (5-6): 505–525.

[21] Andraus Z S, Liffiton M H, Sakallah K A. CEGAR-based formal hardware verification: A case study [J]. Ann Arbor. 2007, 1001: 48109–2122.

[22] Padmanabhuni B M, Tan H B K. Predicting Buffer Overflow Vulnerabilities through Mining Light-Weight Static Code Attributes [C]. In Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on. 2014: 317–322.

[23] Padmanabhuni B M, Tan H B K. Auditing buffer overflow vulnerabilities using hybrid static–dynamic analysis [J]. IET Software. 2016, 10 (2): 54–61.

[24] Neuhaus S, Zimmermann T, Holler C, et al. Predicting vulnerable software components [C]. In Proceedings of the 14th ACM conference on Computer and communications security. 2007: 529–540.

[25] Zimmermann T, Nagappan N, Williams L. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista [C]. In Software Testing, Verification and Validation (ICST), 2010 Third International Conference on. 2010: 421–428.

[26] Perl H, Dechand S, Smith M, et al. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits [C]. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015: 426–437.

[27] Grieco G, Grinblat G L, Uzal L, et al. Toward large-scale vulnerability discovery using Machine Learning [C]. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. 2016: 85–96.

[28] Yamaguchi F, Wressnegger C, Gascon H, et al. Chucky: Exposing missing checks in source code for vulnerability discovery [C]. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013: 499–510.

[29] Yamaguchi F, Maier A, Gascon H, et al. Automatic inference of search patterns for taint-style vulnerabilities [C]. In Security and Privacy (SP), 2015 IEEE Symposium on. 2015: 797–812.

[30] Rajpal M, Blum W, Singh R. Not all bytes are equal: Neural byte sieve for fuzzing [J]. arXiv preprint arXiv:1711.04596. 2017.

[31] Martin M, Livshits B, Lam M S. Finding application errors and security flaws using PQL: a program query language [C]. In ACM SIGPLAN Notices. 2005: 365–383.

[32] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing [C]. In ACM Sigplan Notices. 2005: 213–223.

[33] Sen K, Agha G. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools [C]. In CAV. 2006: 419–423.

[34] Cadar C, Ganesh V, Pawlowski P M, et al. EXE: automatically generating inputs of death [J]. ACM Transactions on Information and System Security (TISSEC). 2008, 12 (2): 10.

[35] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. [C]. In OSDI. 2008: 209–224.

[36] The LLVM Compiler Infrastructure Project. https://llvm.org/.

[37] Godefroid P, Levin M Y, Molnar D. SAGE: whitebox fuzzing for security testing [J]. Queue. 2012, 10: 20.

[38] Molnar D, Li X C, Wagner D. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. [C]. In USENIX Security Symposium. 2009: 67–82.

[39] 李根. 基于动态测试用例生成的二进制软件缺陷自动发掘技术研究 [D] [D]. [S. l.]: 长沙: 国防科学技术大学, 2010.

[40] Godefroid P. Higher-order test generation [C]. In ACM SIGPLAN Notices. 2011: 258–269.

[41] Majumdar R, Xu R-G. Directed test generation using symbolic grammars [C]. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. 2007: 134–143.

[42] Godefroid P, Kiezun A, Levin M Y. Grammar-based whitebox fuzzing [C]. In ACM Sigplan Notices. 2008: 206–215.

[43] Boonstoppel P, Cadar C, Engler D. RWset: Attacking path explosion in constraint-based test generation [J]. Tools and Algorithms for the Construction and Analysis of Systems. 2008: 351–366.

[44] Godefroid P, Nori A V, Rajamani S K, et al. Compositional may-must program analysis: unleashing the power of alternation [C]. In ACM Sigplan Notices. 2010: 43–56.

[45] Trtík M. Symbolic execution and program loops [D]. [S. l.]: Masarykova univerzita, Fakulta informatiky, 2014.

[46] Godefroid P, Luchaup D. Automatic partial loop summarization in dynamic test geneation [C]. In Proceedings of the 2011 International Symposium on Software Testing and Analysis. 2011: 23–33.

[47] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems [J]. ACM SIGPLAN Notices. 2011, 46 (3): 265–278.

[48] Ma K-K, Yit Phang K, Foster J, et al. Directed symbolic execution [J]. Static Analysis. 2011: 95–111.

[49] Haller I, Slowinska A, Neugschwandtner M, et al. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. [C]. In USENIX Security Symposium. 2013: 49–64.

[50] Böhme M, Oliveira B C d S, Roychoudhury A. Partition-based regression verification [C]. In Proceedings of the 2013 International Conference on Software Engineering. 2013: 302–311.

[51] Marinescu P D, Cadar C. KATCH: high-coverage testing of software patches [C]. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. 2013: 235–245.

[52] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities [J]. Communications of the ACM. 1990, 33 (12): 32–44.

[53] Xu Z, Kim Y, Kim M, et al. Directed test suite augmentation: techniques and tradeoffs [C]. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. 2010: 257–266.

[54] Christakis M, Müller P, Wüstholz V. Guiding dynamic symbolic execution toward unverified program executions [C]. In Proceedings of the 38th International Conference on Software Engineering. 2016: 144–155.

[55] Jin W, Orso A. BugRedux: reproducing field failures for in-house debugging [C]. In Software Engineering (ICSE), 2012 34th International Conference on. 2012: 474–484.

[56] Rö\s sler J, Zeller A, Fraser G, et al. Reconstructing core dumps [C]. In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. 2013: 114–123.

[57] zzuf – Caca Labs. http://caca.zoy.org/wiki/zzuf.

[58] Peach Platform. https://www.peach.tech/products/peach-fuzzer/peach-platform/.

[59] Immunity Inc. | Knowing You're Secure. http://www.immunitysec.com/.

[60] Li W-M, Zhang A-F, Liu J-C, et al. An automatic network protocol fuzz testing and vulnerability discovering method [J]. Jisuanji Xuebao(Chinese Journal of Computers). 2011, 34 (2): 242–255.

[61] Kim H C, Choi Y H, Lee D H. Efficient file fuzz testing using automated analysis of binary file format [J]. Journal of Systems Architecture. 2011, 57 (3): 259–268.

[62] Gorbunov S, Rosenbloom A. Autofuzz: Automated network protocol fuzzing framework [J]. IJCSNS. 2010, 10 (8): 239.

[63] Lin Z, Zhang X, Xu D. Automatic reverse engineering of data structures from binary execution [C]. In Proceedings of the 11th Annual Information Security Symposium. 2010: 5.

[64] Wang T, Wei T, Gu G, et al. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection [C]. In Security and privacy (SP), 2010 IEEE symposium on. 2010: 497–512.

[65] Sharma C, Sabharwal S, Sibal R. Applying genetic algorithm for prioritization of test case scenarios derived from UML diagrams [J]. arXiv preprint arXiv:1410.4838. 2014.

[66] Sabharwal S, Sibal R, Sharma C. Prioritization of test case scenarios derived from activity diagram using genetic algorithm [C]. In Computer and Communication Technology (ICCCT), 2010 International Conference on. 2010: 481–485.

[67] You L, Lu Y. A genetic algorithm for the time-aware regression testing reduction problem [C]. In Natural Computation (ICNC), 2012 Eighth International Conference on. 2012: 596–599.

[68] Patton R M, Wu A S, Walton G H. A genetic algorithm approach to focused software usage testing [J]. Software Engineering with Computational Intelligence. 2003: 259–286.

[69] Böhme M, Pham V-T, Roychoudhury A. Coverage-based greybox fuzzing as markov chain [C]. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016: 1032–1043.

[70] american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[71] Rawat S, Jain V, Kumar A, et al. Vuzzer: Application-aware evolutionary fuzzing [C]. In Proceedings of the Network and Distributed System Security Symposium (NDSS). 2017.

[72] Sparks S, Embleton S, Cunningham R, et al. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting [C]. In Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual. 2007: 477–486.

[73] libFuzzer – a library for coverage-guided fuzz testing. —LLVM 6 documentation. https://llvm.org/docs/LibFuzzer.html.

[74] Böhme M, Pham V-T, Nguyen M-D, et al. Directed greybox fuzzing [C]. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS’17). 2017.

[75] Henderson A, Yan L K, Hu X, et al. DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform [J]. IEEE Transactions on Software Engineering. 2017, 43 (2): 164–184.

[76] Newsome J, Song D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software [C]. In In In Proceedings of the 12th Network and Distributed Systems Security Symposium. 2005.

[77] 张立勇. 软件源代码安全分析研究 [D]. [S. l.]: 西安: 西安电子科技大学, 2011.

[78] Code Check - Help with Building Codes. http://www.codecheck.com/cc/index.html.

[79] Ashcraft K, Engler D. Using programmer-written compiler extensions to catch security holes [C]. In Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on. 2002: 143–159.

[80] Gregor D, Schupp S. STLlint: lifting static checking from languages to libraries [J]. Software: Practice and Experience. 2006, 36 (3): 225–254.

[81] Splint Home Page. http://lclint.cs.virginia.edu/.

[82] Baxter I D, Yahin A, Moura L, et al. Clone detection using abstract syntax trees [C]. In Software Maintenance, 1998. Proceedings., International Conference on. 1998: 368–377.

[83] Yamaguchi F, Lottmann M, Rieck K. Generalized vulnerability extrapolation using abstract syntax trees [C]. In Proceedings of the 28th Annual Computer Security Applications Conference. 2012: 359–368.

[84] Gascon H, Yamaguchi F, Arp D, et al. Structural detection of android malware using embedded call graphs [C]. In Proceedings of the 2013 ACM workshop on Artificial intelligence and security. 2013: 45–54.

[85] Sutton M, Greene A, Amini P. Fuzzing: brute force vulnerability discovery [M]. Pearson Education, 2007.

[86] Robertson W K, Kruegel C, Mutz D, et al. Run-time Detection of Heap-based Overflows. [C]. In LISA. 2003: 51–60.

[87] Wu Z, Atwood J W, Zhu X. A new fuzzing technique for software vulnerability mining [C]. In International Conference on Software Engineering. 2009.

[88] Lanzi A, Martignoni L, Monga M, et al. A smart fuzzer for x86 executables [C]. In Software Engineering for Secure Systems, 2007. SESS'07: ICSE Workshops 2007. Third International Workshop on. 2007: 7–7.

[89] Rawat S, Mounier L. Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results [C]. In Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on. 2011: 531–533.

[90] Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing [C]. In Proceedings of the 31st International Conference on Software Engineering. 2009: 474–484.

[91] Bekrar S, Bekrar C, Groz R, et al. Finding software vulnerabilities by smart fuzzing [C]. In Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on. 2011: 427–430.

[92] Rawat S, Mounier L. An evolutionary computing approach for hunting buffer overflow vulnerabilities: A case of aiming in dim light [C]. In Computer Network Defense (EC2ND), 2010 European Conference on. 2010: 37–45.

[93] Liu G-H, Wu G, Tao Z, et al. Vulnerability analysis for x86 executables using genetic algorithm and fuzzing [C]. In Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on. 2008: 491–497.

[94] Cordón O, Gomide F, Herrera F, et al. Ten years of genetic fuzzy systems: current framework and new trends [J]. Fuzzy sets and systems. 2004, 141 (1): 5–31.

[95] 刘彪. 基于模糊测试的软件安全漏洞发掘技术研究 [J]. 网络安全技术与应用. 2014 (3): 37–37.

[96] antiparser. http://antiparser.sourceforge.net/.

[97] sulley: A pure-python fully automated and unattended fuzzing framework. December 2017. https://github.com/OpenRCE/sulley. original-date: 2012-04-03T15:28:57Z.

[98] Rice H G. Classes of recursively enumerable sets and their decision problems [J]. Transactions of the American Mathematical Society. 1953, 74 (2): 358–366.

[99] Heelan S. Vulnerability detection systems: Think cyborg, not robot [J]. IEEE Security & Privacy. 2011, 9 (3): 74–77.

[100] DeKok A. PScan: A limited problem scanner for C source files [M]. 2000.

[101] Parr T. The definitive ANTLR 4 reference [M]. Pragmatic Bookshelf, 2013.

[102] Moonen L. Generating robust parsers using island grammars [C]. In Reverse Engineering, 2001. Proceedings. Eighth Working Conference on. 2001: 13–22.

[103] Yamaguchi F, Golde N, Arp D, et al. Modeling and discovering vulnerabilities with code property graphs [C]. In Security and Privacy (SP), 2014 IEEE Symposium on. 2014: 590–604.

[104] gremlin. http://github.com/tinkerpop/gremlin/.

[105] CVE-2016-9537. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-953.

[106] The Neo4j Graph Platform – The #1 Platform for Connected Data. https://neo4j.com/.

[107] Rodriguez M A. The gremlin graph traversal machine and language (invited talk) [C]. In Proceedings of the 15th Symposium on Database Programming Languages. 2015: 1–10.

[108] a2ps - GNU Project - Free Software Foundation (FSF). https://www.gnu.org/software/a2ps/#TOCdownloading.

[109] Kratkiewicz K J. Evaluating static analysis tools for detecting buffer overflows in c code [R]. 2005.

[110] Zitser M, Lippmann R, Leek T. Testing static analysis tools using exploitable buffer overflows from open source code [C]. In ACM SIGSOFT Software Engineering Notes. 2004: 97–106.

[111] Stehman S V. Selecting and interpreting measures of thematic classification accuracy [J]. Remote sensing of Environment. 1997, 62 (1): 77–89.

[112] Chen L, Miné A, Wang J, et al. Interval polyhedra: An abstract domain to infer interval linear relationships [J]. Static Analysis. 2009: 309–325.

[113] Logozzo F, Fähndrich M. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses [C]. In Proceedings of the 2008 ACM symposium on Applied computing. 2008: 184–188.

[114] Miné A. The octagon abstract domain [J]. Higher-order and symbolic computation. 2006, 19 (1): 31–100.

[115] ”clang” C Language Family Frontend for LLVM. https://clang.llvm.org/.

[116] Santelices R, Chittimalli P K, Apiwattanapong T, et al. Test-suite augmentation for evolving software [C]. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. 2008: 218–227.

[117] Person S, Yang G, Rungta N, et al. Directed incremental symbolic execution [C]. In ACM SIGPLAN Notices. 2011: 504–515.

[118] Böhme M, Oliveira B C d S, Roychoudhury A. Regression tests to expose change interaction errors [C]. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. 2013: 334–344.

[119] Baldoni R, Coppa E, D’Elia D C, et al. A survey of symbolic execution techniques [J]. arXiv preprint arXiv:1610.00502. 2016.

[120] Rodriguez P, Wiles J, Elman J L. A recurrent neural network that learns to count [J]. Connection Science. 1999, 11 (1): 5–40.

[121] Cho K, Van Merriënboer B, Gulcehre C, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation [J]. arXiv preprint arXiv:1406.1078. 2014.

[122] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate [J]. arXiv preprint arXiv:1409.0473. 2014.

[123] Kirkpatrick S, Gelatt C D, Vecchi M P. Optimization by simulated annealing [J]. science. 1983, 220 (4598): 671–680.

[124] gnu.org. https://www.gnu.org/software/binutils/.

[125] Chollet F. keras: Deep Learning library for Python. Runs on TensorFlow, Theano, or CNTK. December 2017. https://github.com/fchollet/keras. original-date: 2015-03-28T00:35:42Z.

[126] Abadi M, Agarwal A, Barham P, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems [J]. arXiv preprint arXiv:1603.04467. 2016.

[127] Kingma D, Ba J. Adam: A method for stochastic optimization [J]. arXiv preprint arXiv:1412.6980. 2014.

作者在学期间取得的学术成果

发表的学术论文

[1] Meng Q K, Chao F, Zhang B, Tang C J. Assisting in Auditing of Buffer Overflow Vulnerabilities via Machine Learning. In press.（已被 Mathematical Problems in Engineering 出版，SCI 源刊.）

[2] Meng Q K, Wen S M, Zhang B, Tang C J. Automatically Discover Program Vulnerability Through Similar Functions. 2016 37th Progress In Electromagnetics Research Symposium.（EI 收录，检索号 20165203169818）

[3] Meng Q K, Zhang B, Feng C, Tang C J. Detecting Buffer Boundary Violations based on SVM. 2016 3rd International Conference on Information Science and Control Engineering. （EI 收录，检索号：20165003106907）

[4] Meng Q K, Wen S M, Feng C, Tang C J. Predicting buffer overflow using semi-supervised learning. 2016 9th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics.（EI 检索，检索号：20171303496365）

[5] Meng Q K, Wen S M, Feng C, Tang C J. Predicting Integer Overflow through Static Integer Operation Attributes. 2016 5th International Conference on Computer Science and Network Technology. （IEEE 国际会议，已入 ieee Xplore）

参与的主要科研项目

[1] XXXX 信息系统，海军项目，项目主要负责人.

[2] 全军 XXXX 装备业务系统，总参项目，项目主要负责人.

[3] XXXX 挖掘验证测试技术，国家 863 项目，项目主要参与者.

[4] 大规模 XXXX，2016 国家重点研发计划，项目主要参与者.