分 类 号 _____ 学号 M201272707

学校代码 10487 密级 公开

華中科技大學

硕士学位论文

基于漏洞约束求解的测试用例生成方法研究

学位申请人： 王辉

学 科 专 业： 计算机软件与理论

指 导 教 师： 江胜 讲师

答 辩 日 期： 2015 年 5 月 30 日

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Engineering

Research on Test Case Generation based on Vulnerability Constraint Solving

Candidate : Wang Hui

Major : Computer Software and Theory

Supervisor : Lecturer Jiang Sheng

Huazhong University of Science and Technology

Wuhan 430074, P. R. China

May, 2015

独创性声明

本人声明所呈交的学位论文是我个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，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　月　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本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和汇编本学位论文。

保密□，在_____年解密后适用本授权书。

本论文属于

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　月　日　　日期：　　年　月　日

摘 要

软件测试能够检测软件中的错误并保障软件质量，是软件开发周期中最重要的环节之一。随着软件规模的不断扩大，软件测试中的各项费用也不断增加。软件测试自动化是减少测试时间与费用的重要途径，而自动化软件测试的质量高度依赖于测试用例的自动生成。因此，测试用例生成在软件测试过程中扮演着至关重要的角色，是软件测试领域一个重要的研究方向。

基于漏洞约束求解的测试用例生成方法是一种面向路径的测试用例生成方法。该方法建立在符号执行技术的基础上，主要分为两个部分，一个是基于程序中间表示的符号执行系统的建立，另一个是基于漏洞类型的漏洞约束构建。具体来说，首先对源代码进行静态分析依次得到程序的中间表示、依赖关系图以及漏洞报告等相关信息；接着符号执行系统在程序中间表示的基础上对相关路径上的语句进行符号替换和路径约束收集，同时，对漏洞报告进行分析得出漏洞类型进而构建漏洞约束；最后对收集到的路径约束集合以及漏洞约束进行合并求解继而生成测试用例。基于漏洞约束求解的测试用例生成方法生成的测试用例不仅能够满足覆盖相关路径的要求，同时，该方法生成的测试用例具有针对漏洞特性的较强检错能力。

实验结果表明，基于漏洞约束求解的测试用例生成方法在实际测试中能有效的生成测试用例，并且生成的测试用例具有较强的检错能力。

关键字：中间表示，符号执行，漏洞约束，测试用例生成

Abstract

Software testing detects faults in software and ensures software quality,it is one of the most important phases in development of software. With the continuous expansion of software size, the cost is also increasing. Automation Software testing is an important way to reduce testing time and costs. Meanwhile, software testing is highly dependent upon faults detection ability of test cases.Therefore,test case generation plays a vital role in testing process and is main area of research in the field of software testing.

Test case generation based on vulnerability constraint solve is one of Path-oriented method for test case generation.The method based on symbolic execution and divided into two parts,one is the establishment of symbolic execution system based on intermediate representation,and the other is adaptive vulnerability constraint build based on vulnerability type. Specifically, first of all static analysis of source code in order to obtain an intermediate representation of the program, the dependency graph and bug report and other related information; Then the system execute symbol substitution and path constraint collection on the related path statements on the basis of the program intermediate representation, at the same time,analyze the bug report concluded that vulnerability type and then build vulnerability constraints; Finally,merge path constraints and vulnerability constraints together and solve it, then test cases generated . The method of test case generation based on vulnerability constraint solve not only able to meet the requirements of the related path coverage, while the method generates test cases with strong error detection capability for vulnerability characteristics.

Experimental results show that the test case generation method based on vulnerability constraint solve in the actual test can effectively generate test cases and generates test

cases with strong error detection capabilities.

Keywords: Intermediate Representation,Symbolic Execution,Vulnerability Constraint,Test Case Generation

目 录

1 绪论

1.1 研究背景及意义

随着信息技术的快速发展以及各种软件漏洞攻击手段增多，软件安全问题[1]也变得日益严重。软件存在安全问题主要是因为软件开发过程中开发人员对程序细节考虑不够周全，导致软件中存在一些漏洞。当其中的某一个漏洞被攻击者发现并利用将导致一系列严重的后果，包括用户信息的泄露、服务器瘫痪等等。

缓冲区溢出漏洞是软件中最常见的漏洞之一，2014 年 4 月 9 日，一个名为“心脏出血”（Heartbleed）的重大安全漏洞被曝光[2]，攻击者利用该漏洞可以从网站中窃取用户的账户密码。该漏洞是 OpenSSL 中的一个由缓冲区溢出而导致的内存信息泄露高危漏洞，由于 OpenSSL 应用极为广泛，包括政府、高校网站以及电子商务、邮件系统、网上支付、金融证券等诸多服务提供商均受到该漏洞的直接影响。据奇虎 360 公司提供的抽样检测数据结果显示[3]，国内网站中约有 1.1 万个（占其抽样的 1.0%）服务器存在该漏洞。而引起该漏洞的原因则是因为程序中用到了 memcpy 函数却没有对传递的参数长度仔细检查，这属于典型的缓冲区溢出漏洞。

2014 年 11 月 11 日，微软公布了一个存在于 Windows 操作系统的高危漏洞[4]。据微软的公告称，该高危漏洞存在于微软的安全套接层协议和安全传输层协议中。因为它对于某些特殊形式的数据包不能正确的过滤，这就使得成功利用此漏洞的攻击者可以在目标服务器上运行任意代码。由于受影响的包含了几乎所有的 Windows 版本，一旦漏洞被利用后果将不堪设想。

面对日益突出的软件安全问题，软件测试作为发现和保障软件安全问题的首要方法，在整个软件生命周期所占的比重也越来越大[5]。在计算机软件技术发展初期，软件测试的作用等同于程序调试，常常由程序开发人员自己完成，整个软件开发周

期对测试的投入极少。直到20世纪70年代初，软件测试理论首次被提出[6]，软件测试才被提高到理论的高度。对于软件测试的定义有两种思维，以Hetzel为代表的认为测试就是让程序在已知的环境下根据特定输入的结果推测运行特性的过程[7]，针对软件系统中的功能来验证其准确性，这是一种正向思维；以Myers为代表的认为测试就是运行程序发现错误的过程[8]。现实中对软件进行测试的时候大多数也是以发现程序中的漏洞为目的[9]。

在以发现程序错误为目标的软件测试中，测试的质量高度依赖于测试用例的检错能力。而随着软件规模的变得越来越大，手动测试已经不能满足复杂的测试需求，这就对软件测试提出了更高的自动化测试需求，而自动化的软件测试关键之一就是测试用例的自动生成。因此，自动化测试用例生成在软件测试过程中扮演着至关重要的角色。其中，面向路径的测试用例生成方法[10]对自动化测试用例生成有着重要的意义，是软件测试领域一个重要的研究方向。在面向路径的方法中，基于符号执行的方法有着更加广泛的应用，但大多数方法支持的编程语言单一、生成的测试用例不具有检错能力。因此，研究一种面向多语言的符号执行框架，并且能生成具有一定检测能力的测试用例具有重大的意义。

1.2 国内外研究现状

1.2.1 测试用例生成方法现状概述

软件测试是软件开发中不可缺少的环节，在整个软件开发周期中，软件测试花费的费用占了大约50%[11,12]，甚至更高。通过提高软件测试的自动化程度能够大幅度减少这项花费，其中，测试用例的自动生成是提高软件测试自动化程度的一种重要途径[13]。

作为一个重要的软件组成，测试用例生成所需的信息也必然是从软件的其他构件信息所得出，包括程序的结构、软件的规则说明、程序的输入或者输出空间以及动态执行程序时获得的信息等等[9]。因此，本节将主要从以下几个方法来介绍测试用

例自动生成方法的研究现状。

1：基于符号执行的测试用例生成

符号执行（Symbolic Execution）是由 James C.King[14]于 1975 年提出的一种程序分析技术，该方法被提出后便受到了广泛的关注。符号执行在分析程序过程中将代码中的变量值替换为符号表达式，然后模拟程序的执行流程，遇到分支语句时选择其中的一条分支语句，同时收集条件分支中满足该分支的约束。

作为程序分析技术的一种，符号执行技术以分析程序源代码从而为待测程序自动生成测试用例。现存大量的工作展示了符号执行能有效的处理众多软件工程难题，包括测试用例生成。文献[15]、[16]中提出利用符号执行方法去改善测试用例的路径的覆盖率以及挖掘软件漏洞。文献[17]利用符号执行生成测试用例，在不需要用户隐私信息的情况下，从开发者的角度重现用户在使用时发生的软件错误。文献[18]将符号执行用在压力测试上，该方法显著增长了程序的响应时间以及资源使用率。

2：基于模型的测试用例生成

基于模型的软件测试（Model-Based Testing）是利用软件系统模型来驱动产生测试用例的形式化方法[19]。根据选择模型的不同，具体的实现方法不同。目前，典型的软件系统模型有：有限状态机(FSM)、UML 和马尔可夫链等模型[20]。下面对基于 FSM 的方法进行简单的介绍。

FSM 是表示程序中有限个状态以及反应这些状态之间的关系的数学模型，它是对源程序执行过程的一种抽象。FSM 一般采用状态迁移图表示，根据所使用的状态覆盖从而产生测试用例。文献[21]提出了一个基于 FSM 的自动化测试方案，实现了基于 FSM 的测试用例生成，并将有限状态机模型生成测试序列这个过程自动化。在实际系统的测试中发现，该方案提高了测试发现错误的能力。文献[22]中钱忠胜利用 FSM 模型成熟的理论基础，开发了一个 UML 转 FSM 的模型转换器，结合 UML 与 FSM 的优点对 WEB 进行测试。

3：基于组合测试的测试用例生成

组合测试（Combinatorial Testing）是指通过构造软件系统参数的所有取值组合对系统进行测试[23]，该特性导致测试用例的生成具有 NP 难度性质[24]。文献[25]中提出了一种基于组合空间特性的自适应随机测试用例生成方法，该方法利用将 ART 算法运用到组合空间生成自适应随机测试用例，并在实验中证明了算法的有效性。文献[26]中提出结合遗传算法和蚁群算法，使用 one-test-at-a-time 生成策略，该策略从每次搜索中得到一条较好的测试用例加入测试用例集，直至所有的组合均被覆盖。

4：基于随机测试的测试用例生成

作为一种黑盒测试方法，随机测试(Random Testing)是通过随机地从输入域中选择测试用例来进行测试[27]。随机测试最大的缺点是其产生的测试用例的测试有效性低，只有很少的一部分测试用例能够发现程序中的错误。文献[28]中首次提出了采用随机法来自动生成测试用例。为了改善随机测试的效率，文献[29]中首次提出了自适应随机测试用例生成方法（Adaptive random testing），该方法是对随机测试方法的改进，不仅随机测试的随机性特点，同时生成的测试用例可以均匀的分布在整个输入域中，提高了随机测试的检测效果。

5：基于搜索的测试用例生成

基于搜索的软件测试（Search-Based Software Testing）方法就是在测试用例生成时，将测试准则（如分支覆盖准则）转换为一个目标函数，从而将用例生成的问题转换成搜索问题。文献[30]提出结合分支限界和爬山法，将分支限界法和爬山法分别用于全局搜索和局部搜索中，发挥两者的优势搜索测试用例的解空间。文献[31]提出了一种新的基于进化测试的测试用例生成方法，该方法在进化搜索过程中以控制依赖分析取代路径选择，减少了进化搜索次数。

1.2.2 现有测试用例生成工具概述

基于漏洞约束求解的测试用例生成方法是建立在符号执行技术的基础上，虽然符号执行技术在上世纪七十年代中期就已经提出，但是直到近些年才得以真正的应

用到程序分析中。主要的原因就是符号执行要应用在大型真实的程序中需要解决大量复杂的约束，而近十年来许多强大的约束求解器被开发出来，例如 Z3[32]，Yices[33]，STP[34]。利用这些强大的约束求解器可以为符号执行求解大量而且复杂的约束，使得符号执行的应用更加广泛。近年来，大量的研究工作已开发了多种利用符号执行和约束求解技术实现的软件工具，并在实际中取得了良好的效果。

由斯坦福大学的 Cristian Cadar 等人开发的 KLEE[35]及其前身 EXE[36]是针对于提高代码覆盖率的自动化软件测试工具。EXE 使用 STP 约束求解器，利用符号执行技术生成测试用例。KLEE 在 EXE 的基础上对约束系统进行了优化，简化了约束集，提高了约束求解能力，并增强了程序的交互能力。KLEE 以及 EXE 均是面向 C 语言的分析工具。

JPF-SE[37]是建立在 JPF（Java PathFinder）[38]基础上的符号执行工具。JPF 是面向 Java 字节码的模型检测工具，构建在定制的 Java 虚拟机上。JPF-SE 结合了符号执行与模型检测技术用来解决测试用例生成问题，它可以解决较复杂的数学约束、数组问题、多线程问题以及部分字符串类型的约束。

Pex[39]是 Tillmann 和 Halleux 等人研发的针对.NET 的白盒测试工具，面向单元测试。Pex 采用 Z3 作为约束求解器求解约束，并利用符号执行技术生成测试用例。Pex 针对的是.NET 平台多种编程语言。

其他的利用符号执行技术和约束求解技术生成测试用例的工具还有很多，诸如 DART[40]、SAGE[41]、CUTE[42]等等，本文不作逐一介绍。

综上所述，虽然目前存在许多利用符号执行和约束求解技术的测试用例生成工具。但是大多数都存在其针对的语言类型单一、求解的约束类型不够丰富等缺点，同时，这些工具生成的测试用例检测漏洞能力弱[35-39]。因此，构建一个语言无关并且能够生成具有一定检错能力的测试用例的符号执行检测系统具有重要的研究意义。

1.3 论文主要研究工作

本文的工作建立在软件安全检测系统（Software Security Detect System，简称SSDS）研究课题上，该系统主要分为静态检测和动态验证两个部分。静态检测负责对待测源程序的静态分析并给出漏洞检测报告，动态验证负责对静态分析的结果进行验证，进一步提高整个系统的准确度。本文的主要工作是结合静态分析基础以及检测出的漏洞报告，为动态验证阶段提供测试用例。

本文介绍了国内外关于测试用例生成技术以及约束求解相关的测试工具，结合本课题中已有的静态检测的基础，研究一种基于漏洞约束求解的测试用例生成方法，使得生成的测试用例具有较强的检错能力。本文的主要工作如下。

1.结合程序中间表示设计一个面向多语言的符号执行框架，解决目前符号执行工具面临的支持语言单一性问题。

2.提出一种新的测试用例生成算法，提高测试用例的检错能力。在符号执行技术的基础上，结合漏洞约束的构造，提升测试用例的有效性。

3.实现一个字符串约束求解器，对生成含有字符串的测试用例有很大的意义。

4.设计一个通用约束求解模型，可以调度多种约束求解器。

5.设计和初步实现基于漏洞约束求解的测试用例生成系统。

1.4 论文组织结构

本文的组织结构以及各章的主要内容如下。

第 1 章：绪论，主要介绍本文的研究背景和意义，同时对国内外在测试用例自动生成方法领域的研究现状进行了综述，重点介绍了基于符号执行的测试用例方法研究现状，并简要的阐述本文的研究工作。

第 2 章：基于程序中间表示的符号执行方法，主要介绍了基于中间表示的符号执行系统的设计方法，并给出了整体设计框架以及具体的核心算法。

第 3 章：基于漏洞约束求解的测试用例生成方法，详细阐述了漏洞约束的概念以及漏洞约束构建过程，给出了基于漏洞约束求解的测试用例生成的算法。

第 4 章：基于漏洞约束求解的测试用例生成系统的设计与实现，给出了系统的总体设计框架，并且详细阐述了各个核心模块的实现方案以及关键技术的实现方法。

第 5 章：实验，给出本系统在实际测试中的测试用例生成效果，并对实验结果进行了分析。

第 6 章：总结与展望，对本文已完成的工作进行了总结，并展望了下一步需要继续进行研究的工作。

2 基于程序中间表示的符号执行方法

作为静态程序分析技术的一种，符号执行方法利用源代码来进行程序分析。由于不同编程语言的源代码语法各不相同，目前流行的大多数符号执行系统并不能同时面向多种编程语言，这就导致了这些符号执行工具存在一定的局限性。因此，构建一个语言无关的符号执行检测系统具有一定的研究意义。

本章首先介绍基于程序中间表示的符号执行方法的系统流程，然后概述了从源代码到中间表示的处理过程以及相关的依赖分析技术，最后详细介绍了基于中间表示的符号执行方法。

2.1 基于中间表示的符号执行系统框架

基于程序中间表示的符号执行主要由两个部分组成：静态分析层以及符号执行层。如图 2.1 所示。

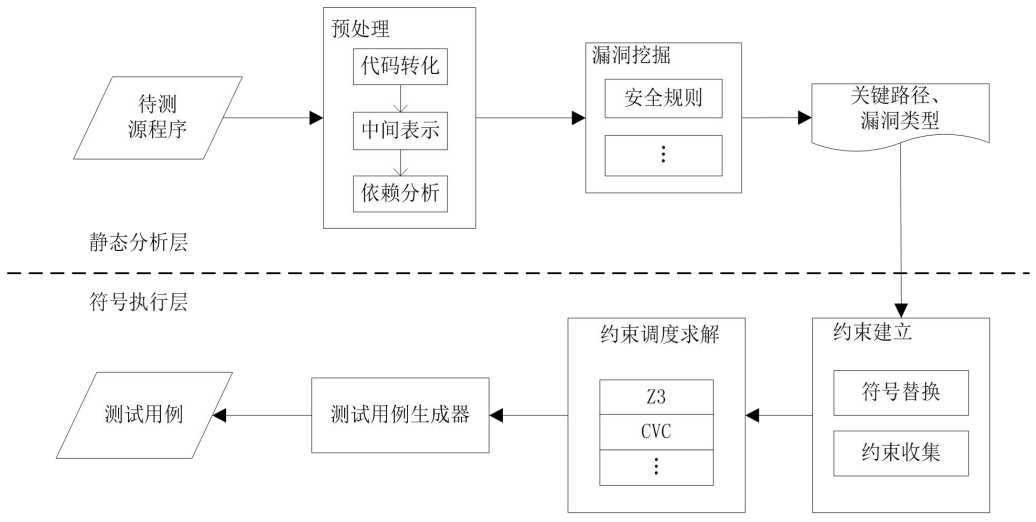

图 2.1 基于中间表示的符号执行系统框架

静态分析层主要利用 ANTLR[43]工具来实现源程序的预处理，生成程序的中间表示，通过对中间表示的依赖分析得到程序的控制依赖、数据依赖等依赖关系，在此

基础上对源程序进行漏洞检测得出漏洞报告同时将一条关键路径推荐给符号执行层；符号执行层主要实现了基于中间表示的符号执行功能，涉及到符号替换、约束收集、约束求解以及测试用例生成等技术。

为了实现符号执行系统的语言无关性，首先要解决源程序分析与符号执行的逻辑分离。虽然由程序源代码直接实现符号执行的功能会更高效，但是通过使用统一的中间表示接口，能够实现面向不同编程语言的符号执行系统。这样，对于一个新语言的待测源程序，利用 ANTLR 工具为其构造相应的词法、语法分析器转换成中间表示的形式，就可以在不改变已有的符号执行系统框架的情况下对其进行检测。例如，C 语言中并没有布尔值类型，在条件语句中只要是非 0 值就代表 true，这与其他语言中的布尔值的表示有很大区别，如果将一个原来面向 java 的符号执行系统的输入改成面向 C 语言，需要更改的细节有很多，十分不便。而通过使用中间表示作为符号执行的输入，就不需要太多关注语言本身。比如上面所说的情况，遇到了条件语句中直接是一个整型的变量的时候，静态分析层在转换源程序到中间表示的时候直接将其表示成 x != 0（假设变量名为 x），这就使得符号执行层在对语句分析的时候无需过多的关注实现语言的具体语法元素。

2.2 源程序预处理

在对源程序静态分析之前，作为系统的输入，源程序本身只是一连串的字符，并没有任何意义。源程序在经过 ANTLR 生成的词法分析器、语法分析器处理后，形成抽象语法树，这个时候系统才识别出源程序所表示的程序结构。通过遍历抽象语法树，我们可以将程序中的语义信息存储到相应的中间表示结构中。通过对程序中间表示的分析，可以得到源程序中的数据依赖、控制依赖、调用依赖等依赖关系。本节将简要介绍程序中间表示的设计结构以及转换过程，具体的分析可参考文献[44]。

2.2.1 程序中间表示

作为源程序在系统中存储的中间形式，IR 必须要能够表达出我们所需要的程序语义信息，因为后续的所有分析都是在 IR 的基础上进行的，与源程序无关。所以一个完善的 IR 结构对于整个系统来说很重要。Jimple[45]是一种面向 Java 字节码的中间表示，它具有结构清晰、访问方便等优点。SOOT[46]利用 Jimple 实现了一个 Java 编译优化框架。SSDS 系统借鉴了 Jimple 的设计，设计了一套面向源代码的中间表示。

SSDS 系统的 IR 主要包含 5 种顶层元素：类型、常量、表达式、语句、实体结构。类型（Type）主要包含了整型，字符型，浮点型等类型；常量（Constant）则包括整型常量，字符型常量等；语句（Stmt）包含程序中的行为性语句，包括赋值语句，循环语句等；表达式（Expr）包含了如数组下标访问表达式、强制转换表达式等等。实体结构（Entity）包括诸如函数、类、结构体这样的。在系统项目包中这些结构都存放在 ir 包中，根据不同的编程语言具体的子类实现有所不同，但都使用统一的顶层接口来表示。表 2.1~2.5 分别列出了 IR 系统中 Type、Constant、Expr、Stmt、Entity 五个顶层元素中包含的基本的类，不同的编程语言的具体实现会有所不同。

表 2.1 IR 中类型的主要类

类名	描述	类名	描述
IType	顶级接口	DoubleType	双精度浮点型
IntType	整型	FloatType	单精度浮点型
ByteType	字节型	LongType	长整型
BooleanType	布尔型	ShortType	短整型
CharType	字符型	VoidType	空类型
ArrayType	数组	OtherType	其他类型

表 2.2 IR 中常量的主要类

类名	描述	类名	描述
Constant	顶级抽象类	BooleanConstant	布尔常量
IntConstant	整型常量	FloatConstant	单精度浮点数常量
StringConstant	字符串常量	DoubleConstant	双精度浮点数常量
CharConstant	字符常量	OtherConstant	其他常量

表 2.3 IR 中表达式的主要类

类名	描述	类名	描述
IExpr	顶级接口	ConditionExpr	关系表达式
UnopExpr	一元操作表达式	BinopExpr	二元操作表达式
ArrayAccessExpr	数组访问操作	CastExpr	强制转换
InstanceFieldAccessExpr	成员访问	OtherExpr	其他表达式

表 2.4 IR 中语句的主要类

类名	描述	类名	描述
IStmt	顶级接口	VarDeclStmt	声明语句
ArrayAssignStmt	数组初始化语句	AssignStmt	赋值语句
InvokeStmt	方法调用语句	IfStmt	条件语句
WhileStmt	循环语句	ReturnStmt	返回语句
SwitchStmt	Switch 语句	OtherStmt	其他语句

表 2.5 IR 中实体结构的主要类

类名	描述	类名	描述
Field	字段	Local	局部变量
Class	类	Body	方法体
Method	函数	Interface	接口

2.2.2 源代码分析与转换

ANTLR[43]（Another Tool for Language Recognition）是一个开源的分析器，它可以接受文法语言描述，并能产生识别这些语言的语句的程序。它能够支持 C、C++、Java、Python 等多种编程语言。ANTLR 一个很大的方便之处在于它可以让用户在语法文件嵌入一段辅助代码，例如 Java。利用这个功能，在遍历抽象语法树的时候我们可以在不同的语法规则处嵌入相应的代码，从而将程序中的语义信息存储到相应的 IR 数据结构中。

如图 2.2 所示，从源代码到 IR 主要经过了词法分析、语法分析、遍历抽象语法树三个过程。对于不同的编程语言，利用 ANTLR 来生成相应的词法、语法分析器。词法分析器用于将程序从源代码转换为字符流；语法分析器以词法分析的输出为输入，将字符流转换为抽象语法树；最后遍历抽象语法树将程序的语义信息存储到相

应的 IR 数据结构中。

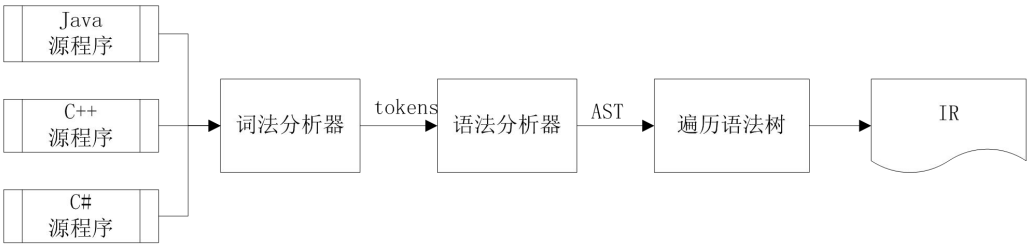

图 2.2 源代码的分析与转换

2.2.3 依赖分析

静态分析技术就是在不运行程序的情况下来分析程序，而要在不运行程序的情况下获取一些程序在运行时可能的流程就必须得到程序的各种依赖信息。程序的依赖关系图反应了程序语句、方法之间的信息流动，它是源程序的一种中间形式，直观的反应了程序的依赖信息。方法体内语句的数据依赖关系反映了程序语句之间的数据关系，而控制依赖关系则反应着语句之间的控制关系。程序的控制依赖、数据依赖可以为静态检测提供程序一些最基本的依赖信息。

面向路径的符号执行方法中，路径中的各个语句之间的控制依赖关系唯一的确定了这条路径。符号替换算法的基础是数据依赖分析，同时，数据依赖关系在后续建立漏洞约束过程中也起着重要的作用。本节主要介绍程序控制依赖以及数据依赖相关概念，并给出形式化的定义。

1：控制依赖

定义 2.1 控制流程图(Control Flow Graph，CFG)

在方法 M 中，给出四元组 $CFG=(N,E,s,q)$，其中 N 代表方法体内所有语句节点的集合；$E\subseteq N\times N$ 表示所有语句节点之间的有向边的集合；$s\in N$ 表示方法的开始节点；$q\in N$ 表示方法的终止节点，那么我们将这样的四元组称之为方法 M 的控制流程图。

定义 2.2 控制依赖(Control Dependence，CD)

在方法 M 中，当控制流程图中存在语句节点n_i和n_j，如果节点n_i所代表的语句S_i的执行结果会决定节点n_j所代表的语句S_j是否执行，这样节点n_j控制依赖于节点n_i（记作$n_i \xrightarrow{CD} n_j$）。

定义 2.3 控制依赖图(Control Dependence Graph，CDG)

在方法 M 中，控制依赖图是一个有向图$G =< N, E >$，其中 N 表示方法 M 内所有语句节点的集合；$E \subseteq N \times N$表示节点之间的有向边的集合，反映语句间控制依赖关系。

2：数据依赖

在给出数据依赖的定义前，先了解一个函数概念，设n_i是 CFG 的结点集合中的一个节点，我们用$Def(n_i)$表示节点n_i中定义的变量，$Use(n_i)$表示节点n_i中引用到的变量集。例如，程序“int var = x + y;”中$Def(n_i)$={var}，$Use(n_i)$={x，y}。

定义 2.4 数据依赖(Data Dependence，DD)

对于控制流程图中的任意两个节点n_i和n_j，如果存在一条可达路径且这条路径中没有对$Def(n_i)$集合中的变量的重定义，那么当 $Def(n_i) \cap Use(n_j) \neq \varnothing$时，我们称$n_j$数据依赖于$n_i$（记作$n_i \xrightarrow{DD} n_j$）。

定义 2.5 数据依赖图(Data Dependence Graph，DDG)

数据依赖图是一个有向图$G =< N, E >$，其中 N 表示一个方法体内所有语句节点的集合；$E \subseteq N \times N$是有向边的集合。

2.3 基于中间表示的符号执行方法

本节将主要叙述基于中间表示的符号执行方法，首先简要介绍符号执行技术，

接着分别介绍了如何在程序中间表示的基础上进行符号替换以及约束的转换和收集。

2.3.1 符号执行过程

与黑盒测试在生成测试用例过程中不考虑测试程序代码相反，白盒测试会分析程序源代码或二进制代码来生成测试数据，使用符号执行方法来进行测试用例生成在近些年引起了很大的关注[47-49]。符号执行，顾名思义在分析程序时使用符号值而不是具体值来作为程序的输入，并且将程序中的变量都使用符号表达式来代替。在程序符号执行过程中的任一点，符号执行的程序状态主要包括：符号表、路径约束以及程序计数器[9]。符号表用于记录程序中出现的变量的符号表达式。路径约束（PathConstraint，简称 PC）是布尔表达式，用于收集程序运行到这一点所必须满足的所有约束。在符号执行遇到分支语句后，路径约束随之更新，如果约束集合无解，则表示该分支不可达，则符号执行就停止在该分支上继续执行；如果求解约束成功，那么更新程序计数器指向该分支的下一条语句。

如图 2.3 所示，图 2.3（a）中的程序是用来交换变量 x 和 y 的值，当且仅当 x 的值大于 y 的时候。图 2.3（b）显示了该程序段符号执行过程中相应的符号执行树。符号执行树显示了符号执行过程中路径约束的收集以及程序变量的符号表达式更新。在符号执行树中，树节点代表了程序的状态，而节点之间的边代表程序状态之间的转换，树节点上的数字代表程序计数器的值，这里就是程序段中的行号。在执行语句 1 前，初始化 PC 为 true，给变量 x、y 分别赋予符号值 X、Y。在执行条件语句 1 和 5 之后，PC 也要相应的更新。图 2.3（c）中的表显示了上述程序代码中对应的三个程序执行路径的符号执行情况，对于不同的路径符号执行后收集的 PC 是不同的。例如，路径（1→2→3→4→5→8）的 PC 为 X > Y&Y - X <= 0。通过约束求解器求解得到一个可行输入 X = 2，Y = 1。因此该路径是可满足的。而对于另一条路径（1→2→3→4→5→6），它对应的 PC：X > Y&Y - X > 0 是无解的，表明不存在执

行该路径的程序输入。

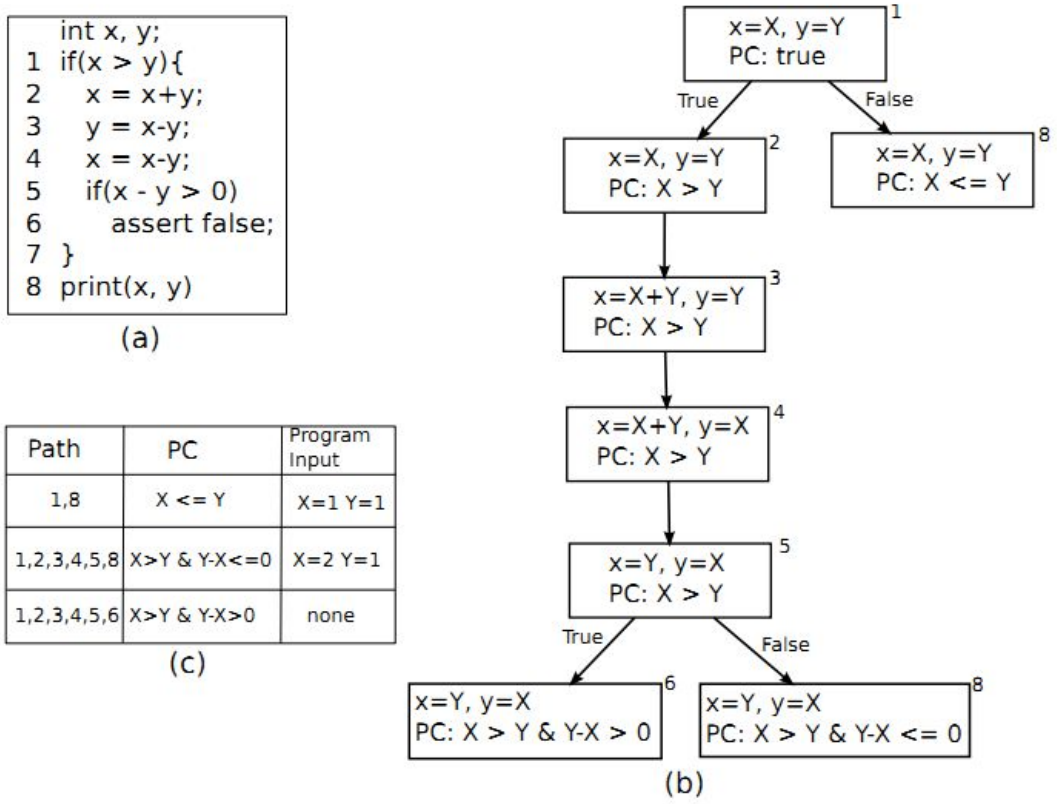

图 2.3 符号执行过程示例

通过上面的例子可以了解到，符号执行过程中主要涉及到程序变量的符号表达式更新以及路径约束的收集。下面将主要介绍这两种算法。

2.3.2 符号替换

符号执行层的输入是由静态分析层分析后给出的一条路径上的所有语句，这是由 IR 语句组成的语句集合。由于符号执行中变量都是符号值代替的，而不是具体值，所以，符号执行首先要解决用符号值替换程序中的变量问题。

符号替换过程中主要涉及到两个数据结构：路径语句列表（PathStmt），符号值映射表（SymbolMap）。路径语句列表主要存放路径推荐模块传递过来的语句集合，这条语句集合是可能引发漏洞的路径上的所有语句。在系统中用 List<IStmt>来存储

语句集合，其中 IStmt 就是 IR 中用来表示语句的接口，所有类型的语句都是该接口的子类。符号值映射表则保存着程序中出现的变量名称与符号表达式的对应关系。系统采用 Map<String,IValue>数据结构来保存每一个映射关系，其中 String 是变量的名称类型，IValue 则是 IR 中所有的语句、变量、常量、表达式的顶层接口。例如图 2.4 中的程序段，假设路径推荐模块推荐的路径是（1→2→3→4→5→6），表 2.6 对应的是该路径的语句列表。

```
public int test(int x) {
1       int a, b;
2       a = x + 1;
3       x++;
4       b = x++;
5       if ((a + b) > 7) {
6               return 1;
7       }
8       return 0;
}
```

图 2.4 示例程序

表 2.6 路径语句列表

行号	PathStmt	IR 类型
1	int a,b	VarDeclStmt
2	a = x + 1	AssignStmt
3	x++	ExprStmt
4	b = x++	AssignStmt
5	if ((a + b) > 7)	IfStmt
6	return 1	ReturnStmt

文献[50]中提到的符号计算在替换过程中仅仅只是在遇到赋值语句才会对符号映射表进行更新，而忽略了自增自减运算对符号映射表的影响。下面以图 2.4 中的程序为例来解释本文中使用的符号替换过程。

SymbolMap 刚开始首先存入对方法参数的符号映射，我们直接以参数的名称作为符号值。初始化 SymbolMap，导入相关信息，此时 SymbolMap 中的状态如表 2.7 所示。

表 2.7 SymbolMap 状态表 1

变量名称	x		
符号表达式	X		

当执行到第二条语句 a = x + 1 后，首先向 SymbolMap 中添加变量名称 a；然后获取该赋值语句的右值，即 x + 1；通过数据依赖关系可以得到该表达式对变量 x 有数据依赖关系，获取 x 在 SymbolMap 中的符号表达式来替换掉该赋值语句中的右值中的变量 x。最后，更新 SymbolMap。更新后的 SymbolMap 如表 2.8 所示。

表 2.8 SymbolMap 状态表 2

变量名称	x	a	
符号表达式	X	X+1	

当执行到第三条语句 x++的时候，该自增表达式只对自身变量产生影响，所以只需要对 SymbolMap 表中的变量 x 进行符号表达式更新。更新后的各变量状态如表 2.9 所示。

表 2.9 SymbolMap 状态表 3

变量名称	x	a	
符号表达式	X+1	X+1	

执行语句 b = x++ 后，首先我们判断出这个赋值语句的右值是个自增操作，同时，该操作结果是将 x 自增前的结果赋予 b。所以首先是要对 x 的符号表达式更新并记录下更新前的符号表达式，然后在 SymbolMap 中更新 b 的符号表达式。由于此时表中并没有变量 b，则向表中添加该项，如表 2.10 所示。遍历示例程序中后续的语句都没有对 SymbolMap 表中的数据产生改变，因此，SymbolMap 表更新完毕。

表 2.10 SymbolMap 状态表 4

变量名称	x	a	b
符号表达式	X+2	X+1	X+1

本节所叙述的符号替换过程都是在对 IR 中的元素进行替换。刚开始的时候符号映射表中只含有参数的符号表达式，遍历语句列表，当遇到赋值语句或者自增自减表达式语句的时候对符号映射表进行更新。具体过程如算法 2.1 所示。

算法 2.1 需要遍历一遍路径语句列表，判断出其中可能对符号映射表有影响的语句。在每一遍遍历过程中，如果遇到的是赋值语句还需要遍历该语句右表达式所使用的变量集合，这个循环的时间复杂度是 O(1)，具体的替换相应为符号表达式的算法 substitute，这个算法的时间复杂度也是 O(1)，限于篇幅，这里并不对 substitute 函数进行详细介绍。所以整个算法的时间复杂度为 O(N)，其中 N 为路径中所含有的语句数量。

算法 2.1 符号替换算法(makeSymbolic)

输入：路径上的语句集合、参数列表、数据依赖图
输出：符号映射表

```
1.  pathStmts:路径上的语句集合
2.  symbolMap：符号映射表
3.  step 1: 遍历 pathStmts 中的每条语句
4.  foreach(stmt ∈ pathStmts)
5.  step 2: 判断语句类型、更新 symbolMap
6.      if(stmt ∈ AssignmentStmt) // 赋值语句
7.          获取赋值的变量名 varName、右表达式 right；
8.          获取 right 中的数据依赖变量 uses；
9.          foreach(use : uses)//遍历使用过的数据集；
10.             从 symbolMap 获取 use 的符号表达式 expr；
11.             substitute(right,use,expr)//替换 right 中的 use 为 expr；
12.         更新 symbolMap 中的 varName，没有则添加；
13.     else if(stmt ∈ UnopExprStmt ) // 自增自减表达式语句
14.         获取变量名 varName；
15.         获取表达式的具体操作类型；
16.         从 symbolMap 获取 varName 的符号表达式 expr；
17.         根据不同类型做不同的符号表达式更新；
18.         更新 symbolMap 中的 varName；
19.     endif
20. endfor
```

2.3.3 路径约束收集

定义 2.4 路径约束（PathConstraint，简称 PC）

在程序中，条件控制语句的值决定了程序的实际执行路径。因而该条件语句及

其对应取值代表了该路径的一条约束，该路径上的所有的约束组成了这条路径的约束集合，简称路径约束。

在基于中间表示的符号执行系统中，程序中的一条约束实际上就是 IR 中 if 语句中的条件表达式。IR 中 if 语句的条件表达式主要有三种形式：一是布尔类型的局部变量(Local)；二是条件表达式(ConditionExpr)；三是函数调用(VirtualInvokeExpr)。ConditionExpr 中包含常见的大于、小于、等于等返回值为布尔类型的表达式，VirtualInvokeExpr 通常则是字符串类的一些返回值为布尔类型的操作函数。不同的 IR 约束表达式在进行约束求解的时候显然要分别转换成求解器理解的约束形式。例如条件表达式 x>2 与 x<2 是两个意思完全相反的条件表达式。为了将约束表达的更加清晰，本系统在 IR 的基础上设计了一个约束类型子系统，对于不同的约束，系统设计了不同的约束类来表示，这样在约束转换求解的时候，可以很方便的识别出约束的类型继而转换成约束求解器可以理解的约束表达式。

约束系统中主要将约束分为两大类，一种是只含有基本类型的约束，主要包括比较运算(例如 >，<，==，<=)、布尔值以及逻辑运算。另一种则是字符串类型的约束，这些约束主要是表达字符串操作函数中返回值为布尔型的函数。这样分类的原因在于分类之后可以方便的判断出约束的类型，同时，判断出约束的类型对后续的约束求解进行优化具有较好的现实意义。

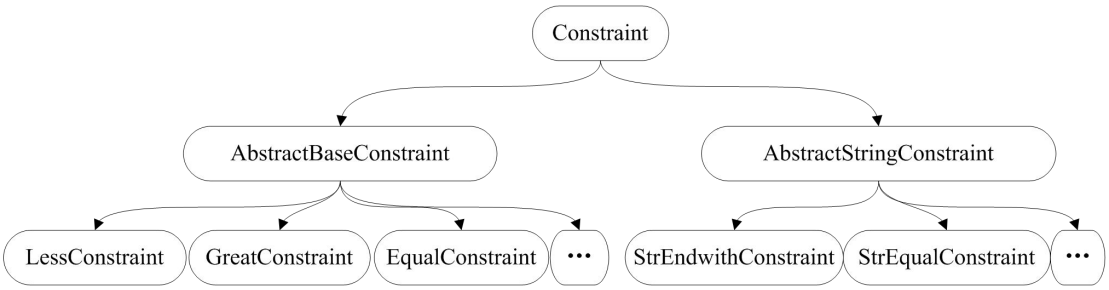

图 2.5 约束系统类型图

约束系统的类图如图 2.5 所示。本文提到的所有约束都是包含在该约束系统下，程序中所有的约束条件都可以转换成该约束系统中的约束。由 2.3.1 节可知，对于路

径约束的收集实际是在符号替换的过程中进行的。如果不是这样，即约束的收集是在符号替换方法之后，在遇到对路径敏感的约束条件时往往会生成错误的约束，导致约束求解失败。例如图 2.6 中的程序(a)中给定的的路径（1→2→3→4→5），在条件约束 $x == 2$ 之后又有了对 x 的一次赋值操作，执行到 $x = x + 1$ 这句程序后，符号表中的 x 的值变为了 $x + 2$。此时符号替换完毕，收集路径约束。约束求解器得到的路径约束就是 $x + 2 = 2$，对此求解得出 $x = 0$。显然，这个输入并不能满足给定的路径（1→2→3→4→5），而是路径（1→2→7）。而在程序(b)中，给定路径（1→2→3→4→5），经过符号替换后约束求解的解是空。虽然对于这两个程序经过符号执行后求解出的数据都满足不了要求，但两个程序事实上都是存在一个输入（x = 1）可以满足给定的路径约束。所以，对于路径约束收集与符号替换必须是同时进行。

```
public int test(int x){
1   x = x + 1;
2   if(x == 2){
3           x = x + 1;
4           // do something
5           return 1;
6   }
7   return 0;
}
```

程序 a

```
public int test(int x){
1   x = x + 1;
2   if(x == 2){
3           x = x + 1;
4           if(x == 3){
5                   return 1;}
6   }
7   return 0;
}
```

程序 b

图 2.6 约束收集示例程序

路径约束收集的过程就是将程序中的条件约束转换成如图 2.5 所示的约束类型系统中的某个约束类并收集到路径约束集合中。下面给出路径约束收集算法的详细实现，如算法 2.2 所示。与符号替换算法相似，该算法需要遍历路径上的语句集合，遍历的过程中会调用符号替换算法，但对于每条语句来说，不管是符号替换还是建立约束处理它只需要遍历一次就行。因此，整个算法的时间复杂度是 O(N)。N 代表语句列表中语句数量。

算法 2.2 路径约束收集算法(getPathConstraint)

输入：路径上的语句集合、数据依赖图、符号映射表
输出：路径约束

```
1.  pathStmts:路径上的语句集合
2.  symbolMap：符号映射表
3.  pathConstraint：路径约束
4.  step 1: 遍历 pathStmts 中的每条语句
5.  foreach(stmt ∈ pathStmts)
6.  step 2: 判断语句类型
7.      if(stmt ∈ IfStmt) // if 语句
8.  step 3: 收集语句中的约束条件
9.          获取 if 语句中的条件 value；
10.         根据数据依赖图获取该 value 上的所有依赖变量 uses；
11.         foreach(use ∈ uses)// 遍历 uses
12.             获取该 use 在 symbolMap 中的符号表达式；
13.             替换 value 中 use 变量为相应的符号表达式；
14.         endfor
15.         switch(value)
16.             case 布尔变量：新建相应约束 Constraint
17.             case 条件表达式：新建相应约束 Constraint
18.             case 函数调用表达式：新建相应约束 Constraint
19.         pathConstriant.add(Constraint)；
20.     elseif  //符号替换
21.         执行符号替换算法，更新 symbolMap；
22.     endif
23. endfor
24. return pathConstraint；
```

2.4 本章小结

本章主要介绍了基于程序中间表示的符号执行方法，首先简要介绍了源程序的预处理过程，接着详细介绍了从源程序的中间表示到符号执行后的约束表达过程。这其中主要涉及到符号表达式的替换以及约束的转换与收集等相关技术。对于约束求解，本文将在后面的章节中给出详细的介绍。

3 基于漏洞约束求解的测试用例生成方法

本文所研究的测试用例生成算法是针对静态分析已经检测出的漏洞点，有针对性的生成测试用例，并不是盲目的生成测试数据来对目标程序进行测试，也不是只针对特定漏洞的生成测试用例。因此，本章提出一个基于漏洞约束求解的测试用例生成方法，该方法建立在符号执行技术的基础上，能显著提高测试用例的检错能力。

本章将从以下几点来介绍，首先重点介绍构建漏洞约束的过程；最后综合第二章和本章的方法给出了测试用例生成算法；然后介绍现阶段符号执行工具的局限性以及基于漏洞约束求解的测试用例生成方法的优点。

3.1 基于漏洞类型的漏洞约束构建

3.1.1 CWE 漏洞类型分类

常见缺陷列表（Common Weakness Enumeration，CWE）[53]是已经在软件中发现缺陷的安全漏洞词典，该词典由麦特公司（MITRE Corporation）维护。对于 CWE 中所列出的每一种缺陷，CWE 都提供了一套可测试源码集帮助软件在发布之前识别、发现并解决其中的 bug、缺陷和易受攻击点。CWE 中对常见的漏洞类型有如表 3.1 中所示分类。本系统中漏洞检测模块对漏洞类别的分类就是基于 CWE 的漏洞分类标准，所以本章接下来所述的漏洞类型都是指在该漏洞分类标准范围内的类型。

3.1.2 漏洞约束

在基于符号执行的测试用例生成方法中，要生成具有针对性的测试数据必然需要有相应的针对性约束。当一个标志着可以触发漏洞的约束被建立后，加入到程序中的路径约束当中一起交由约束求解器求解。求出的解不仅仅可以满足路径的执行要求，同时也满足了具有触发漏洞的可能性要求。

表 3.1 CWE 常见漏洞类型

ID 编号	漏洞类型
CWE-119	Buffer Errors
CWE-134	Format String Vulnerability
CWE-16	Configuration
CWE-189	Numeric Errors
CWE-20	Input validation
CWE-200	Information leak/disclosure
CWE-22	Path traversal
CWE-255	Credentials Management
CWE-264	Permissions, Privileges, and Access Control
CWE-287	Authentication issues
CWE-310	Cryptographic issues
CWE-352	Cross-Site Request Forgery
CWE-79	Cross-Site Scripting
CWE-78	OS Command Injections
CWE-89	SQL Injection
CWE-94	Code Injection
Others	Insufficient information

定义 3.1 漏洞特征值(Vulnerability Eigen-value，VEV)[54]

如果存在一个常量c能触发某个漏洞类型的漏洞v，则c被称为触发漏洞v的一个漏洞特征值。对于已知的漏洞类型，一般来说都有能触发它发生的常量值，例如空指针解引用漏洞类型的漏洞特征值为“NULL”，SQL 注入攻击的漏洞特征值包含“1 OR 1=1”等。

定义 3.2 漏洞约束(Vulnerability Constraint，VC)

漏洞约束是系统根据程序中可能存在漏洞的所属类型，对触发漏洞的参数添加相应的约束限制，产生的约束即为漏洞约束。一个或多个漏洞约束的约束解对应一种可以触发该漏洞的漏洞特征值。

由定义 3.2 可知，漏洞约束是程序本身并不包含的约束条件，它是根据程序本身可能存在的漏洞类型而建立的。对于某个漏洞，触发它的漏洞特征值可以有很多个。为每个漏洞特征值添加相应的约束类型，这样就产生多个漏洞约束。这里所说的约

束类型与 2.3.3 节中所提的约束类型是一致的，漏洞约束也是一种程序约束。

如图 3.1 所示。系统根据漏洞报告获取漏洞类型，再由漏洞类型获得相应的漏洞特征值，结合约束类型建立对应的漏洞约束。下面以 SQL 注入为例来详细介绍漏洞约束是如何建立的。

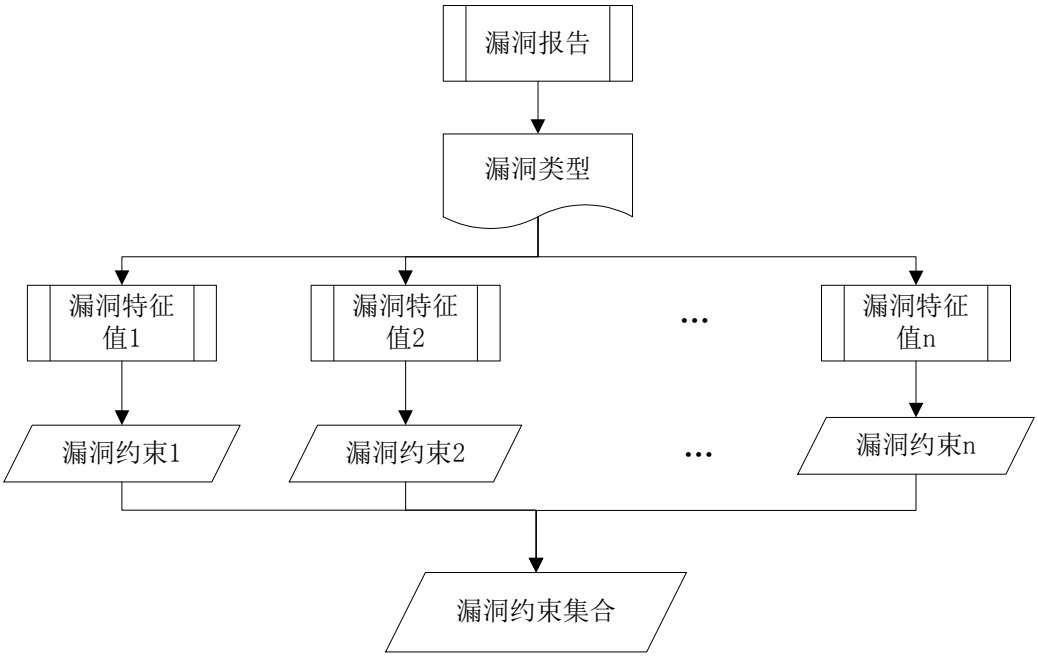

图 3.1 漏洞约束的建立

SQL 注入攻击是网络中一种较为常用的入侵数据库的手段。通过巧妙的构造 SQL 语句来实现代码注入，达到攻击目的。常见的攻击手段有使用注释符号、恒等式、使用 insert 或 update 语句插入或修改数据库中的数据等等。如果忽略了对输入字符串的检查，一段被精心设计的注入语句被数据库认为是正常的 SQL 指令而运行，从而使数据库受到攻击，继而导致重要的数据信息被窃取以及进一步的危险操作导致网站被嵌入恶意代码，这样直接导致了后续的用户访问将受到更多的危害。SQL 注入攻击大体可以分为以下四种类型。

第一种是渗透测试，用于判断服务器是否存在 SQL 注入漏洞。通常的攻击手段是在 URL 后面添加一个分号或者单引号。如果服务器代码中没有对字符串进行过滤，那么数据库在执行带有该参数的指令的时候就会发生异常。

```
http://www.frotest.com/test/?id=1'
```

图 3.2　SQL 注入渗透测试示例

如图 3.2 所示，URL 后面添加了一个单引号，对于没有建立有效验证过程的服务器来说，数据库在运行相应的语句会得到一个参数“1’”（参数没有双引号）。这在运行指令的时候肯定会出错，不同的数据库报出的异常信息各不相同。而一旦数据库报异常，就说明了这里存在 SQL 注入漏洞。

这里，参数“1’”中的单引号就是一个漏洞特征值，且必须是出现在参数的最后一位，对应的约束类型就是 endsWith，即末尾以单引号结束。假设引发该漏洞的参数名为 id，建立的漏洞约束就为 id.endsWith(“1’”)。

SQL 注入攻击的第二种类型是猜测数据库内容。假设渗透测试表明该网页存在 SQL 注入漏洞，当攻击者希望了解数据库中有没有相应的表名的时候可以如图 3.3 这样。

```
http://www.frotest.com/test/?id=1' AND (SELECT COUNT(*) FROM
tablenames)>=0; --
```

图 3.3　SQL 注入猜测数据库内容示例

这里的有个关键的参数是最后的注释“--”，前面的语句可以是其他目的的攻击手段，包含了这个注释，后续的语句都将不需要执行。所以，参数“--”是一个漏洞特征值，相应的约束类型是 contains，即语句中包含了这个符号即可。当然，我们也可以类似的用“and 1=(select count(*) from tablenames); --”来做漏洞特征值，这个特征值生成的测试用例专门查找数据库表中是否含有表名为 tablenames 的表。

第三种 SQL 注入类型是获取系统访问权限。攻击者通常利用“or 1 = 1”来构造一个逻辑永真的语句，或者像前面那样，利用“--”注释掉后续的一些可能的权限验证等。假设数据库要执行一条语句如图 3.4 所示，其中 userName 和 passWord 是参数。如果用户传递过来 userName 参数是“admin’ or 1=1;--”，后续的 passWord 无论是什么值都将会顺利执行而达到攻击的目的。

```
statement=″Select * from users where uname=’ ″+ userName +
″’  and password=’ ″+ passWord + ″’ ;″
```

图 3.4　SQL 注入获取系统访问权限示例

可以说“admin’ or 1=1；--”是一个漏洞特征值，在某些情况下也可以不需要后面的“--”。“admin’ or 1=1；--”对应的约束类型是 startsWith，“admin’ or 1=1；”对应的约束类型是 equals。

```
statement = ″Select * From data Where id = ″ + variable + ″;″
```

图 3.5　SQL 注入获取和破坏数据示例

第四种是获取和破坏数据库中存储的数据。如图 3.5 所示，正常情况下数据库执行这条语句后会返回 data 表中 id 为 variable 的所有信息。如果没有对 variable 进行验证，当传入参数“1;drop table users”后 SQL 语句变成：“select* from data where id = 1;drop table users;”。执行后会将 users 表从数据库中删除。“1;drop table users”也就是引发此漏洞的一个特征值，对应的约束类型是 equals，如果在后面添加分号和注释变为“1;drop table users;--”同样也可以引发漏洞，此时的漏洞特征值是“1;drop table users;--”，对应的约束类型是 startsWith。

3.1.3 漏洞约束库

由上一小节可知一个漏洞类型所对应的漏洞特征值可能有很多个，为了更好的支持程序漏洞约束的建立，本系统采用数据库对漏洞约束进行存储。采用数据库来存放约束信息的优点是当有新的漏洞约束时，可以直接在数据库中添加相应的内容，而不必去修改系统源程序。随着网络的不断发展，必然会出现各种各样新的漏洞信息，用户可以根据实际情况来完善漏洞约束库中的信息。

漏洞约束库中所存放的是可以直接构建漏洞约束的相关信息。构建一个漏洞约束最重要的是要知道该约束所属的约束类型以及对应的漏洞特征值，同一个漏洞类型，不同的漏洞特征值所对应的约束类型可能是不同的。基于此，设计的漏洞约束

库应该包括漏洞类型，漏洞特征值，约束类型。下面对漏洞约束库中的主要字段进行说明。

1. 唯一标识码：唯一标识一条漏洞约束。

2. 漏洞类型：表示该漏洞约束所属的漏洞类型，方便根据漏洞报告信息来查找数据库中该漏洞类型的所有漏洞约束。例如 SQLInjection、XSS。

3. 约束名称：对于一条漏洞约束的命名，用于简要描述该漏洞约束产生的数据会引发什么样的危害。例如上一小节中提到的 SQL 渗透测试等，可以直观的表述该漏洞约束所要表达的漏洞特征。

4. 约束描述：对于约束名称的补充说明，用于详细描述该约束所要产生的测试数据的作用。

5. 约束类型：表示该条漏洞约束信息所建立的约束类型，约束的类型有 less、great、equals、endsWith 等。

6. 漏洞特征值：触发该漏洞类型的特征值，比如漏洞类型为 SQLInjection 时，对应的漏洞特征值是“1 OR 1=1”或者是“1 AND 1=1”等等。

7. 约束入库时间：记录用户录入漏洞约束的时间，系统自动分配当前服务器的时间为入库时间，在用户更新约束的时候无法更改这一字段。

8. 风险等级：描述该测试用例所触发的漏洞对系统的危害程度。

3.1.4 漏洞约束构建算法

从前两节可以知道，由静态检测系统检测出的一个漏洞，在经过了分析后可以得出相应的漏洞类型继而得到一系列的漏洞约束。所以，在本系统完全可以由静态分析层检测出漏洞后交由符号执行层来自动的进行漏洞类型分析后建立漏洞约束，从而达到自动的生成具有针对性的测试数据。因此，本文提出了基于漏洞类型的约束构建算法，结合符号执行技术能够自适应的生成目标明确的测试数据。

如图 3.6 程序所示，该程序是节选于 Juliet 测试集中 SQL 注入漏洞的一段测试代

码。程序中没有对输入数据 data 进行校验，在程序的第 10 行有被 SQL 注入攻击的风险。如果输入数据 data 的值是“admin’ ;drop table users;-- ”，程序在执行到第 10 行后不仅仅是更新了 users 表中的数据，而且还删除了该表。在本系统中，静态分析层在对源程序进行预处理后检测出了程序中可能含有 SQL 注入攻击的漏洞，并给出了相关的漏洞报告和关键路径。符号执行层在得到相关的数据后，首先对程序的关键路径进行路径约束收集。通常这个时候，传统的符号执行方法就会根据路径约束信息进行约束求解，继而生成测试数据。在本例中，由于没有约束条件，求解出 data 的值可以为任意值。本系统根据漏洞报告提供的漏洞信息得出漏洞的类型是 SQL 注入攻击，根据漏洞类型去漏洞约束库中找出该漏洞类型的所有漏洞约束。将得到的漏洞约束添加到约束集合当中交于约束求解器求解。最终，我们将得到一些可能触发 SQL 漏洞的测试数据，data=“admin’ ;drop table users;-- ”就是其中之一。并在实际运行的验证后证明了该测试用例可以引起破坏数据库的后果。

```
1.public void bad(String data) throws Throwable
2.{
3.          Connection dbConnection = null;
4.          Statement sqlStatement = null;
5.          try{
6.           dbConnection = IO.getDBConnection();
7.           sqlStatement = dbConnection.createStatement();
8.          /* POTENTIAL FLAW: data concatenated into SQL statement
9.          used in execute(),which could result in SQL Injection */
10.          Boolean result = sqlStatement.execute("insert into
11.         users (status) values ('updated')  where name='"+data+"'");
12.          if(result) {
13.                    IO.writeLine("Name, " + data + ", updated successfully");
14.          } else{
15.                    IO.writeLine("Unable to update records for user: " + data);
16.          }
17.         }catch (SQLException exceptSql) {
18.          IO.logger.log(Level.WARNING,
19.         "Error getting database connection", exceptSql);
20.         }finally{
21.                    //省略部分代码
22.         }
}
```

图 3.6 存在 SQL 注入漏洞示例程序

依据上述思想，基于漏洞类型的漏洞约束构建算法如算法 3.1 所示。其中在对漏洞类型的判断中，如果该漏洞类型在漏洞约束库中不存在那么就不对该漏洞类型生成特殊的漏洞约束。如果存在，则从漏洞约束库中提取出相关的约束加入到约束集合中。

算法 3.1 构建漏洞约束算法 getVulConstraint

输入：漏洞报告
输出：漏洞约束集合

```
1.  bugReport:漏洞报告
2.  type:漏洞类型
3.  parameter:触发漏洞的参数
4.  vulConstraints:漏洞约束集合
5.  step1: 获取漏洞类型和触发参数
6.      type = bugReport.getType()
7.      parameter = bugReport.getParameter()
8.  step2: 获取漏洞类型为 type 的漏洞约束集合
9.      从数据库中获取类型为 type 的数据项集合 list
10. step3: 建立漏洞约束
11.     foreach(elem : list)
12.         建立约束 con = buildConstraint(parameter ,elem)
13.         vulConstraints.add(con)
14.     endFor
15. return vulConstraints;
```

算法 3.1 主要分为三个阶段，第一阶段从漏洞报告中提取出漏洞类型及触发参数等信息；第二阶段是从数据库中提取出所有关于该漏洞类型的约束项；第三阶段主要是遍历第二阶段所取出的所有约束项，对于每个约束项建立相应的漏洞约束，并将其添加到漏洞约束集合中。最后，返回漏洞约束集合。算法的时间消耗主要集中在第二阶段和第三阶段，第二阶段中从数据库中取数据项，根据数据库表中总的数据量 M 来说，时间复杂度是 O(M)。第三阶段的时间复杂度为 O(N)，N 代表特定漏洞类型在数据库表中的约束数量。显然 M>N，所以该算法的时间复杂度是 O(M)。

3.2 约束求解与测试用例生成

目前存在很多优秀的约束求解器，比如能够处理线性计算的 Z3[32]、Yices[33]以及 STP[34]，还有能够对字符串约束进行求解的 Z3-str[55]、Hampi[56]等等。这些约束求解器都具有强大的约束求解功能，可以分别应对不同的约束求解需求。所以，本文并没有将约束求解作为研究的重点，而是实现了一个通用的约束求解调度模型，可以将系统的约束问题转换成其他约束求解器理解的数据类型。同时我们也基于 Z3 求解器实现了一个字符串约束求解器。本文将在第四章系统实现部分说明该约束求解器的具体实现。

3.2.1 通用约束求解模型

通用约束求解模型的目的就是实现通用的约束求解调度器，通过调度器来调度不同的约束求解器。例如，Z3 以及 Yices 支持对整数相关的约束进行求解，当程序中只含有此类约束的时候，可以通过调用 Z3 或者 Yices 来进行约束求解。而当程序中只包含字符串类型的约束时，Z3 和 Yices 已经无法满足求解需求，这时需要调用 Hampi 等可以求解字符串相关约束的求解器。因此，通用的约束求解调度模型用于系统选择合适的约束求解器，为复杂的程序约束提供了广泛的求解途径。

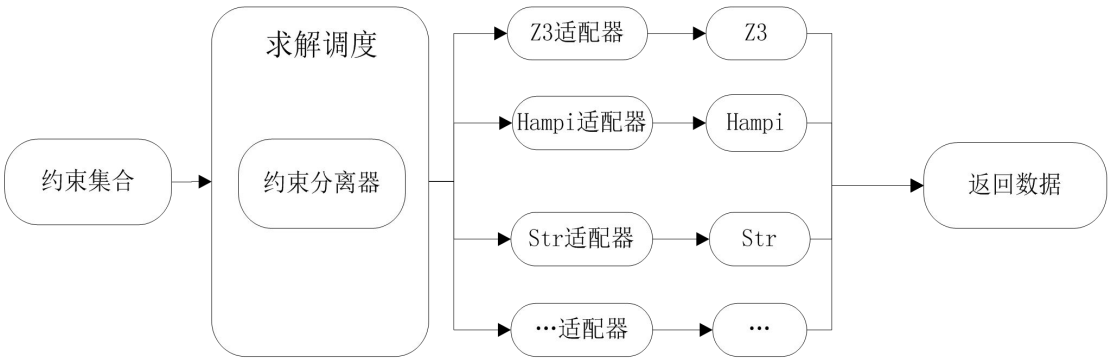

图 3.7 通用约束调度模型

如图 3.7 所示，通用约束调度模型主要由约束分离器和各个约束求解器的适配器所组成。约束分离器的作用是用来判断约束的类型从而选择合适的求解器，如果约

束集合只是单一的字符串类型或者基本类型，那就使用相应的约束求解器来求解约束。如果约束集合同时含有这两种类型的约束，也就是混合约束集合，可以直接利用具有混合约束求解能力的求解器进行求解。约束分离器的核心就是约束分离，约束分离算法如算法 3.2 所示。

约束适配器的作用就是将不同的约束类型转换成该适配器所要适配的约束求解器能够理解的数据类型。第二章中已经提过约束类型，对于具体的适配器接口的设计在第四章中将有更加详细的介绍。

算法 3.2 约束分离算法(solveDispatch)

输入：约束集合，数据依赖图

```
1.  constraints:约束集合
2.  strFlag:约束中是否含有字符串类型的约束
3.  baseFlag:约束中是否含有基础类型约束
4.  baseConstriantSolver:专门求解基本类型的约束求解器
5.  strConstriantSolver:专门求解字符串类型的约束求解器
6.  mixConstriantSolver：混合约束求解器
7.  step 1：判断约束集合中约束类型
8.  foreach(cons : constraints)
9.          集合中含有字符串类型约束
10.             strFlag = true;
11.         集合中含有基本类型约束
12.             baseFlag = true;
13. endFor
14. step 2：选择合适的约束求解方式
15. if(strFlag && baseFlag)// 混合约束
16.     mixConstriantSolver
17. else if(strFlag )// 仅含有字符串类型约束
18.     strConstriantSolver
19. else // 仅含有基本类型约束
20.     baseConstriantSolver
21. endif
```

约束分离算法第一阶段中用一个循环判断约束集合中是否含有各种类型的约束，时间复杂度为 O(N)，N 代表约束的数量。第二阶段主要是判断使用什么样的求解器最合适，最复杂的过程在于判断约束类型之间的数据依赖关系，对于两类约束

都要遍历，每次遍历一类约束的时候还要遍历另一类约束。所以第二阶段时间复杂度为 $O(N^2)$。综合，该算法的时间复杂度为 $O(N^2)$。

3.2.2 测试用例生成算法

综合前两章中的叙述，基于漏洞约束求解的测试用例的生成算法主要分为以下几个步骤：1，初始化参数列表，获取需要生成测试用例的参数集合；2，在程序中间表示基础上通过符号替换和约束收集，收集程序关键路径的约束信息；3，构建漏洞约束，根据漏洞报告中的漏洞类型提取漏洞约束集合；4，测试用例生成，根据程序中的路径约束结合漏洞约束生成测试用例，由于漏洞约束可能有多个，生成的测试用例也可能有多个。具体算法流程如算法 3.3 所示。

算法 3.3 测试用例生成算法（getTestCase）

输入：路径语句集合，漏洞报告
输出：测试用例

```
1.  pathStmts：路径语句集合
2.  parameters：入口函数的参数列表
3.  testCase：测试用例集合
4.  symbolMap：符号映射表
5.  pathConstraints：路径约束集合
6.  vulConstraints：漏洞约束集合
7.  step1：初始化参数列表
8.      parameters = getParameters();
9.  step2：符号替换、路径约束收集
10.     symbolMap=makeSymbolic();
11.     pathConstraints=getPathConstraint();
12. step3：漏洞约束提取
13.     vulConstraints=getVulConstraint();
14. step4：约束求解
15.     foreach(constraint : vulConstraints )
16.         cons = constraint + pathConstraints
17.         data = solverDispatch(cons)// 返回一个测试用例
18.         testCase.add(data)
19.     endFor
20. return testCase;
```

算法 3.3 首先获取路径的语句集合以及漏洞报告，第一步根据所获得的语句获取入口函数的参数列表，符号映射表也需要根据参数来初始化。这一阶段的时间复杂度为 O(1)。第二步进行符号替换和约束收集，这一步的时间复杂度为 O(N)，N 代表语句列表中的语句个数。第三步是漏洞约束的提取，根据漏洞报告中的漏洞类型信息，从漏洞约束库中提取漏洞约束，这一步的时间复杂度为 O(M)，M 为数据库中所有漏洞约束的个数。第三步是约束求解过程，程序要遍历漏洞约束集合，将每个漏洞约束分别加入到路径约束中进行求解，在有解的情况下将作为一个测试用例添加到测试用例集合中。这一步中，假设漏洞约束的个数为 K，路径约束个数为 L，结合约束分离算法的时间复杂度，这一步的时间复杂度为 $O(K*L^2)$。综合以上，整个测试用例生成算法的时间复杂度为 $Max(O(N),O(M),O(K*L^2))$，在实际情况中，由于漏洞约束个数 K 和路径约束个数 L 都相对较小，并且数据库中的查找时间 O(M)在索引优化后时间也大大减小，因此本算法实际运行时间并不长。

```
1 public ManagerForm login(String account, String password, int id) {
2       ManagerForm manager = null;
3       try {
4               Statement stmt = connection.createStatement();
5               if (id == 0) {
6                       String sql = "select * from tb_manager where account='"
7                                       + account + "' and password = '" + password + "'";
8                       ResultSet rs = stmt.executeQuery(sql);
9                       while (rs.next()) {
10                              manager = new ManagerForm();
11                              manager.setId(Integer.valueOf(rs.getString(1)));
12                              manager.setAccount(rs.getString(2));
13                              manager.setPassword(rs.getString(3));
14                              manager.setName(rs.getString(4));
15                              manager.setSigh(Integer.valueOf(rs.getString(5)));
16                      }
17              }
18      } catch (SQLException ex) {
19              ex.printStackTrace();
20      }
21      return manager;
22}
```

图 3.8 测试用例生成示例程序

以图 3.8 中的程序为例，静态检测后会给出漏洞报告和关键路径等信息。首先获

取该函数的参数 account、password、id；接着对关键路径（2→3→4→5→6→8）进行约束收集，收集到的路径约束集合为{id=0}；然后对漏洞报告分析得到漏洞类型为 SQL 注入，因此，从漏洞约束库中提取约束项结合 account 建立漏洞约束，根据目前漏洞约束库信息，建立的漏洞约束集合如表 3.2 所示；最后，结合漏洞约束集合与路径约束进行求解，求解得出的测试用例集如表 3.3 所示。其中“？”代表该参数可以为任意值。

表 3.2 漏洞约束集合示例

漏洞约束集合
account.endsWith(“1’”)
account.contains(“and 1=(select count(*) from user); -- ”)
account.startsWith(“admin’ or 1=1;-- ”)
account.equals(“admin’ or 1=1;”)
account.startsWith(“1;drop table users;-- ”)

表 3.3 测试用例集结果

编号	account	password	id
1	“1’”	?	0
2	“and 1=(select count(*) from user); -- ”	?	0
3	“admin’ ;-- ”	?	0
4	“admin’ or 1=1;”	?	0
5	“1’;drop table users;-- ”	?	0

将测试用例集中测试用例分别代入程序参数中运行，发现编号 3，5 能够正常运行，并触发了相应的漏洞，运行 1，2，4 号测试用例分别报出 SQL 指令语法错误。

3.3 基于漏洞约束求解的测试用例生成方法的优点

现存大量的工作展示了符号执行能有效的处理众多软件工程难题，包括测试用例生成。尽管现阶段符号执行技术已经被广泛的运用于测试用例生成方法中，但大部分的研究还是集中在提高测试用例基于某种准则的覆盖率[15]。这些工具很难发现软件中的漏洞，而且面临的问题也很多，比如说路径爆炸问题。虽然也有学者利用符号执行来研究面向路径的测试用例生成方法[51,52]，利用定位不安全函数的位置来确

定一条路径，这种方法避免了路径爆炸问题，但仍然没有解决生成的测试数据触发漏洞的概率低的问题。收集符号约束是所有基于符号执行的测试用例生成方法的一个重要环节，收集的约束直接影响后续测试用例的生成效果。如果在这一步添加相关约束，这些约束能够引导并生成具有较高检测能力的测试用例，这样就解决了生成测试用例具有触发漏洞的有效性问题。因此，如何充分发挥符号执行本身的特点是解决测试用例有效性的关键。

基于漏洞约束求解的测试用例生成方法是结合了静态安全性分析与符号执行两者的特点，利用静态安全性分析检测出的漏洞检测报告得出漏洞的类型，再根据不同的漏洞类型生成漏洞约束，结合符号执行中的约束求解系统生成测试用例。例如，静态检测阶段检测出源程序中可能含有缓冲区溢出漏洞，系统将该漏洞信息以及相应的关键路径传递给符号执行模块，这些信息在经过相应的处理后，系统会收集到路径中所包含的约束集合。此时，符号执行模块根据系统所传过来漏洞类型判断出该疑似漏洞可能导致缓冲区溢出，根据这一点，符号执行模块自动在约束集合中添加针对引发该类型漏洞的变量的若干约束。之后求解约束集合，如果有解，当后续执行测试输入该用例的时候将有可能会触发缓冲区溢出。

3.4 本章小结

本章主要介绍了基于漏洞类型的漏洞约束构建技术，在此技术的基础上实现了基于漏洞约束求解的测试用例生成算法。文中详细介绍了漏洞约束的定义以及在程序中漏洞约束的构建过程，并设计了一个漏洞约束库，方便用户对漏洞约束的管理。最后给出了基于漏洞约束求解的测试用例生成算法。

4 系统设计与实现

前面几章重点讨论了基于中间表示的符号执行方法以及基于漏洞约束求解的测试用例生成方法，本章将重点讨论系统的整体框架以及框架中各个组成模块的具体实现。

4.1 系统总体框架

基于漏洞约束求解的测试用例生成系统是一个将待测源程序进行预处理表达成一种程序的中间表示形式，通过符号执行方法并结合漏洞约束构建方法生成测试用例的自动化系统。系统的整体框架如图 4.1 所示，主要包含了源代码分析器、静态安全性分析器、符号执行器、漏洞约束构造器以及通用约束求解调度器。各部分功能如下。

1. 源代码分析器：该模块的主要作用是对待测源程序进行分析和转换，通过调用源码预处理器实现从源码到中间表示 IR 的转换，接着调用依赖分析器对程序进行依赖分析输出各种依赖关系图。

2. 静态安全性分析器：该模块的主要作用是利用静态检测算法检测源程序中存在的漏洞。静态检测算法中主要包含基于安全规则的检测方法[57]等漏洞检测算法。

3. 符号执行器：该模块的主要作用是在程序中间表示的基础之上，对程序中一条路径的所有语句进行符号替换以及约束提取操作，收集该条路径的约束集合。

4. 漏洞约束构造器：该模块的主要作用是通过前期静态安全分析后得出的漏洞类型信息来构造相关的漏洞约束，该约束可以引导约束求解器生成有特定性质的测试用例。

5. 通用约束求解调度器：该模块的主要作用是提供统一的约束接口，通过对不同的约束求解器建立不同的适配类，从而达到对多种约束求解器的灵活调用。同时，对约束集合进行约束求解，生成测试用例。

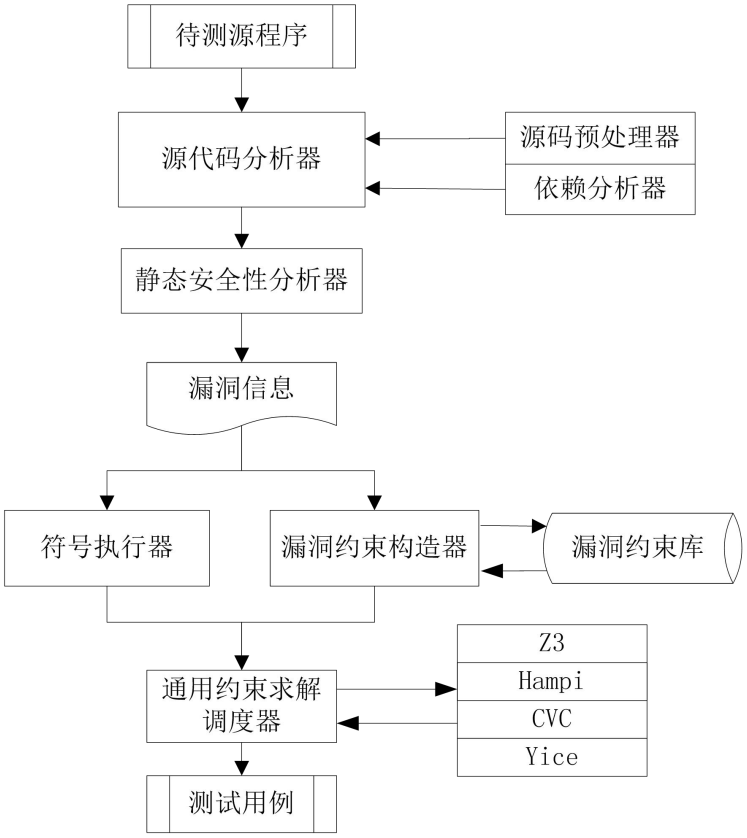

图 4.1 基于漏洞约束求解的测试用例生成系统框架

源代码分析器以及静态安全性分析器不是本文的主要内容，这里不多做讨论。详细的解释可见[54,57]。下面主要介绍符号执行器、漏洞约束构造器以及通用约束求解调度器。

4.2 符号执行模块

4.2.1 约束类型子模块

约束类型是符号执行阶段约束收集所需要用到的类，不同的 IR 约束对应不同的约束类。在源程序的预处理过程中，预处理器对源程序进行了 IR 转换，对于程序中各种约束条件，并没有更加具体的 IR 类来表示，而是统一用了 ConditionExpr、Local、VirtualInvokeExpr 来代表约束条件。符号执行模块实现了将收集到的 IR 约束进行包

装，转换为相应的约束类型。

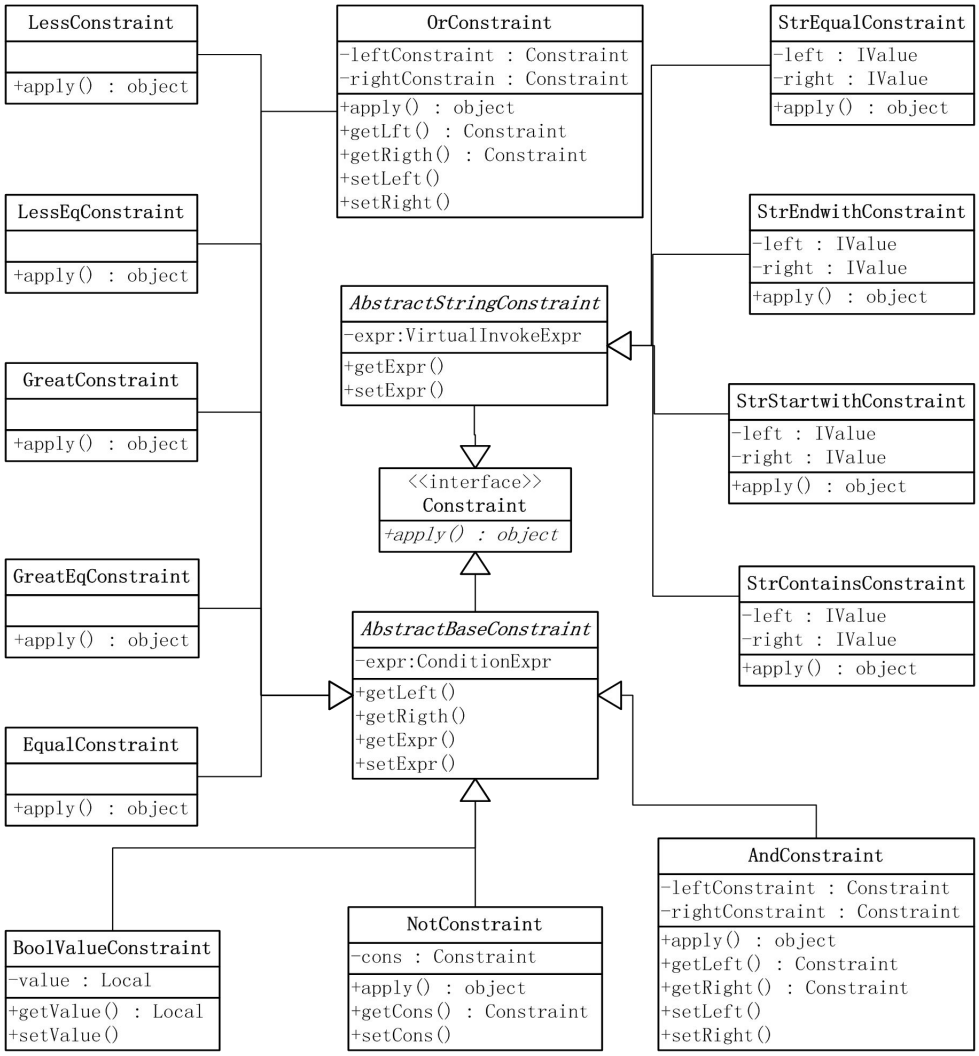

图 4.2 约束类型子模块的类图

约束类型子模块的类图如图 4.2 所示。其中 Constraint 作为约束的顶层接口，所有继承该接口的类都属于约束类。Constraint 接口定义了一个抽象函数 apply，每个继承了 Constraint 的子类都各自实现了 apply 函数，该函数接收一个约束求解器对象，并在函数体内调用该约束求解器相对应的转换函数，该转换函数实现了将该类型的约束转换成约束求解器理解的数据类型。因此，apply 函数的作用就是当约束求解器需要将某个具体约束对象转换为该求解器理解的数据类型时直接调用该约束的 apply

函数即可，而不必逐一的判断约束的类型然后去找相应的转换函数。

4.2.2 符号执行器

符号执行器主要是符号替换以及约束收集，这两个方法在第二章中已经详细的做了介绍，这里给出整个执行流程。

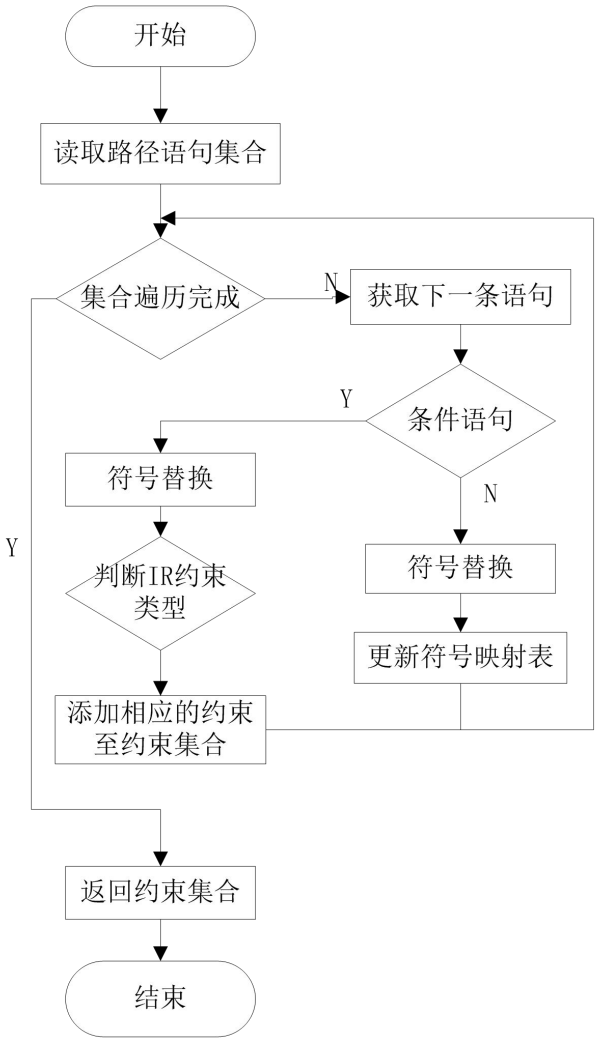

图 4.3 符号执行流程图

如图 4.3 所示，符号执行器对路径语句的集合进行遍历，判断每一条语句的类型，如果是条件语句，则提取条件表达式，根据表达式的类型添加相应的约束类型实例对象到路径约束集合中。遍历完成后返回路径约束集合。

4.3 漏洞约束模块

4.3.1 漏洞约束库

本系统使用 MySQL 数据库来存储漏洞约束，系统采用数据表 VulConstraint 来存储漏洞约束的相关信息，在 3.2.3 节中介绍了漏洞约束库中每个字段的含义和用途，表 4.1 直接给出数据表 VulConstraint 各字段的详细信息。

表 4.1 数据表 VulConstraint 字段信息

字段名	类型	大小	是否允许为空	说明
VC_SRID	bigint	20	否	主键，序列号，自动增长
VC_UID	varchar	30	否	唯一标识码
VC_VType	varchar	30	否	漏洞类型
VC_Name	varchar	100	否	约束名称
VC_Description	varchar	255	否	约束描述
VC_CType	varchar	30	否	约束类型
VC_VEV	varchar	200	否	漏洞特征值
VC_RiskLevel	varchar	30	是	风险等级
VC_AddedDate	date		否	添加日期

4.3.2 漏洞约束构造器

漏洞约束构造器工作流程图如图 4.4 所示，首先从漏洞报告中提取出漏洞类型及触发参数等信息。接着从数据库中提取出所有关于该漏洞类型的约束数据，然后遍历该数据项集合。如果该集合为空，说明漏洞约束库中不存在对该漏洞类型的约束数据，则返回一个空的漏洞约束集合。如果集合非空，遍历该集合，对于每个约束项结合触发参数建立相应的漏洞约束，并将其添加到漏洞约束集合中。遍历完成后，返回漏洞约束集合。

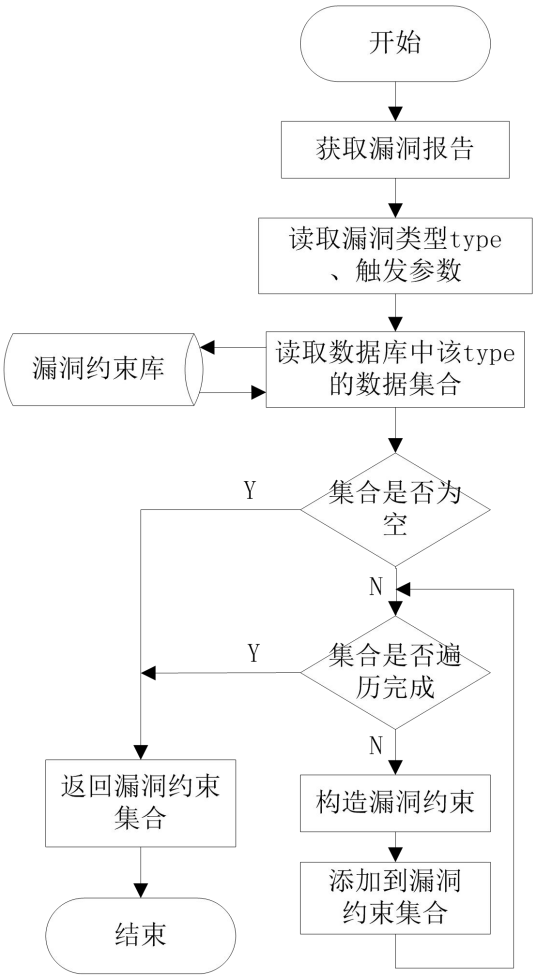

图 4.4 漏洞约束构造器工作流程图

4.4 通用约束求解模块

现阶段许多强大的约束求解器已经在实际中使用，为了方便用户选择合适的约束求解器，通用约束调度模块采用了适配器设计模式。该模块使用一个顶层的接口 ConstraintSolveAdapter，该接口中包含了各种不同的抽象函数，所有实现了该接口的子类必须这些抽象函数。比如转换函数 lessExpr，该转换函数接收一个小于约束类型实例，它的作用就是将该约束实例转换成约束求解器理解的小于表达式。在 Z3 求解器的适配器类中该方法的具体实现如图 4.5 所示。示例代码中的 BoolExpr、ArithExpr 均为 Z3 求解器中的类型，其中 makeExpr 函数实现了将 IR 类型转换成 Z3 的数据类

型。

```
public BoolExpr lessExpr(LessConstraint constraint) {
        // 获取左表达式
        IValue leftExpr = constraint.getLeft();
        // 获取右表达式
        IValue rightExpr = constraint.getRight();
        // 分别转换成Z3的表达式格式
        ArithExpr Left =
                        (ArithExpr)this.makeExpr(leftExpr);
        ArithExpr right =
                        (ArithExpr) this.makeExpr(rightExpr);
        // 合成Z3的小于型布尔表达式
        try {
                return ctx.mkLt(left, right);
        } catch (Z3Exception e) {
                e.printStackTrace();
                return null;
        }
}
```

图 4.5 Z3 适配器中 lessExpr 的代码实现

表 4.1 通用约束求解器接口说明

函数名称	说明
lessExpr()	转换小于约束表达式为求解器理解的约束形式
greatExpr()	转换大于约束表达式为求解器理解的约束形式
greatEqExpr()	转换大于等于约束表达式为求解器理解的约束形式
equalExpr()	转换等于约束表达式为求解器理解的约束形式
notEqExpr()	转换不等于约束表达式为求解器理解的约束形式
lessEqExpr()	转换小于等于约束表达式为求解器理解的约束形式
notExpr()	转换逻辑非约束表达式为求解器理解的约束形式
boolValueExpr()	转换布尔值为求解器理解的约束形式
andExpr()	转换逻辑与约束表达式为求解器理解的约束形式
orExpr()	转换逻辑或约束表达式为求解器理解的约束形式
strEqualExpr()	转换字符串相等比较约束为求解器理解的约束形式
strContainsExpr()	转换字符串包含约束为求解器理解的约束形式
strStartExpr()	转换字符串以特定字符串开头约束为求解器理解的约束形式
strEndExpr()	转换字符串以特定字符串结尾约束为求解器理解的约束形式
isSolvable()	返回约束集合是否有解
solve()	返回约束集合的解

因此，只要相应求解器的适配器类实现了所需的转换函数，就可以将符号执行模块传递过来的约束集合转换成该求解器理解的形式，进而进行约束求解。ConstraintSolveAdapter 中的函数接口说明如表 4.1 所示。

通用的约束求解器类图如图 4.6 所示。其中 AbstractConstraintSolveAdapter 实现顶层接口中的所有函数，这样继承了 AbstractConstraintSolveAdapter 类的子类就不需要去实现全部的接口函数，方便了求解适配器子类的实现，比如一个整型的求解器就不必去实现字符串类的约束转换接口函数。SolverDispatch 用来选择约束器实例，也就是选择具体的适配器子类去解决约束集合。Var 是用来表达传递过来的符号变量，用来包装不同的变量类型。IntVar 代表参数是整形的变量，RealVar 代表参数是浮点型的变量，StringVar 代表参数是字符串型的变量。

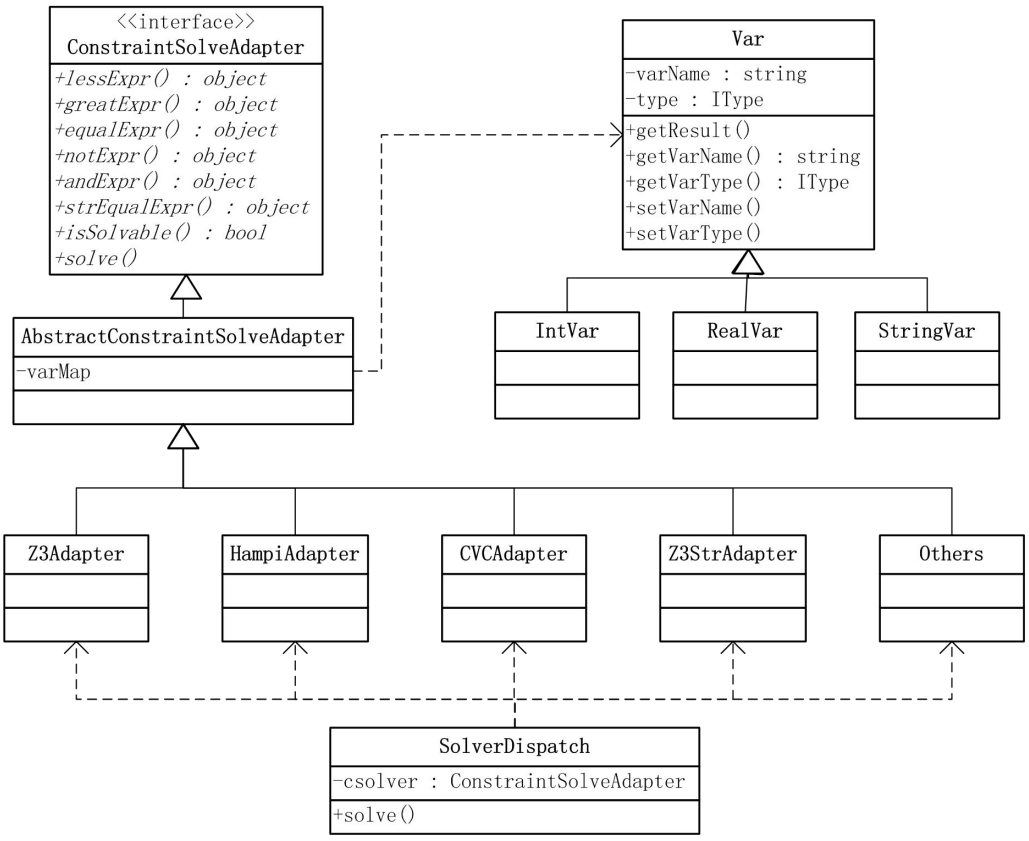

图 4.6 通用约束求解类图

4.5 基于 Z3 的混合约束求解器实现

约束求解是符号执行的一个重要组成部分，原因在于基于约束求解生成的输入数据更加的精确，而且具有一定的目标性。目前大多数的约束求解器能够解决整型等基本类型的约束条件，但是现实程序中有很多漏洞是由字符串引起的，比如 SQL 注入、跨站脚本攻击等，传统的约束求解器已经不能满足这样的求解需求。因此，实现一个能够解决字符串约束的约束求解器具有十分重要的实际意义。

Z3[32]是由微软研究院研发的一款 SMT（Satisfiability Modulo Theories）求解器，它是一个自动化的、可满足性校验的工具，支持对整型、布尔型、实数、数组等类型的约束求解。Z3 是目前支持类型最多的 SMT 求解器，是所有 SMT 求解器当中综合求解能力最强的。当前存在很多种约束求解器可以对程序中含有字符串的约束进行求解，比如 Hampi[56]，Kaluza[58]等等。这些求解器本身也是建立在其他 SMT 求解器基础之上的，它们的缺点仅仅是只能对字符串类型的约束进行约束求解，并不能求解同时含有整型，浮点型的约束。而现实中绝大部分的程序都是字符串与非字符串等各种混合类型组成的，因而这样的求解器很难广泛的适用于现实中的程序分析。因此，本节将实现一个基于 Z3 的字符串约束求解器（Z3-Based String Constraint Solver，简称 Z3-ss），不仅能够对字符串类型的约束进行求解，而且还保持了 Z3 本身先进的求解技术，从而达到对混合的约束进行求解的目的。

4.5.1 基于 Z3 的混合约束求解器系统框架

基于 Z3 的混合约束求解器系统框架如图 4.7 所示。混合约束求解器在接收约束集合后，将该集合分为两部分，一部分是字符串类型约束，另一部分是基本类型约束。基本类型约束直接交于 Z3 求解器进行求解，对于字符串类型的约束需要进行编码后再交于 Z3 进行求解。需要注意的是对基本类型以及编码后的字符串类型进行求解是同时进行而不是分开的。

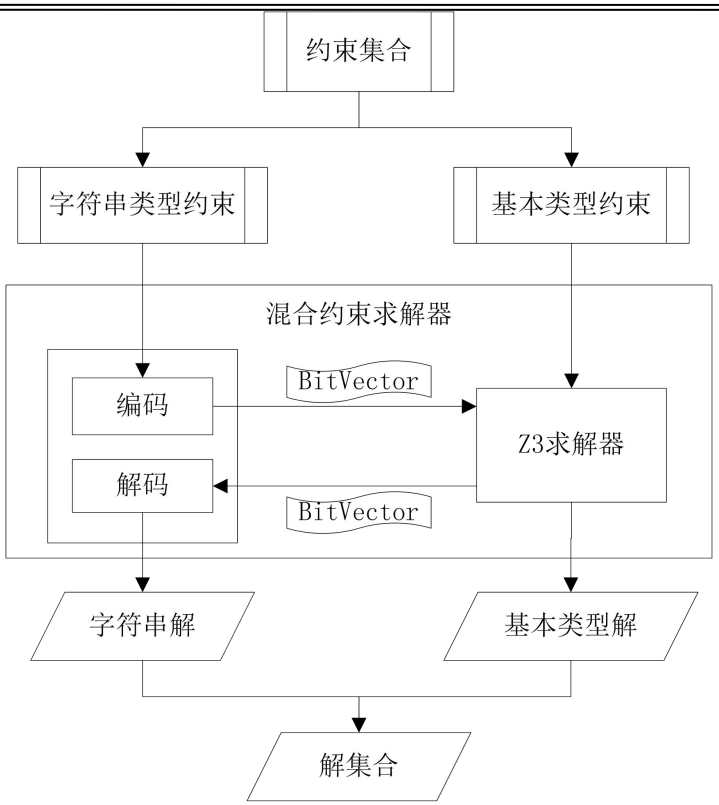

图 4.7 基于 Z3 的混合约束求解器框架

4.5.2 基于位向量的字符串约束表示

同其他 SMT 求解器一样，Z3 本身并不支持字符串类型的约束。所以要解决字符串约束的求解问题，首先要解决的是如何将字符串表达成一个 Z3 可以理解的约束形式。Z3 支持的求解类型有整型、布尔型、数组和位向量等。具体应该将字符串转换成哪一种类型则应该根据字符串本身的组成元素进行合理的分析。

位向量(BitVector)是由若干个位（*bit* ）所组成，每个bit都由 0 或 1 来表示，分别代表 false 和 true。位向量的大小是个常量，在使用位向量之前有多少个*bit* 数量是确定的。一个字符串由若干个字符组成，每个字符我们使用一个 8 位的二进制来表示。例如，字符”a”在 ASCII 表中对应的数字是 97，使用 8 位的二进制表示就是 01100001。因此，在系统中我们可以使用一个 8 位的 BitVector 来对应 ASCII 码表中的一个字符，使用一个 BitVector 数组来表示一个字符串变量或者常量。如此，一个

字符串就有若干个 8 位的 BitVector 组成。假设一个字符串变量长度是 10，相应的申请一个数组长度为 10 的 BitVector 数组。BitVector[0]~BitVector[9]分别代表字符串变量中的第 1~10 字符。

假设一个字符串变量 strVar 被赋值“ababab”，其对应的位向量表示就是一个数组大小为 6 的 BitVector 数组。同时，在数组中的每一个元素都有相应的字符约束，第一位必须是“a”、第二位必须是“b”……。通过查找 ASCII 表中对应字符的二进制表示，转换成对应的位向量的约束则是：

$$
\begin{aligned}
&strVarB = new\ BitVe\ ctor[6]\\
&strVarB[0] = 01100001 \wedge\\
&strVarB[1] = 01100010 \wedge\\
&strVarB[2] = 01100001 \wedge\\
&strVarB[3] = 01100010 \wedge\\
&strVarB[4] = 01100001 \wedge\\
&strVarB[5] = 01100010
\end{aligned}
$$

如果一个长度为 10 的字符串变量 strVar，已知该字符串是以”ab”开头，转换成对应的位向量约束就是：

$$
\begin{aligned}
&strVarB = new\ BitVec\ tor[10]\\
&strVarB[0] = 01100001 \wedge\\
&strVarB[1] = 01100010
\end{aligned}
$$

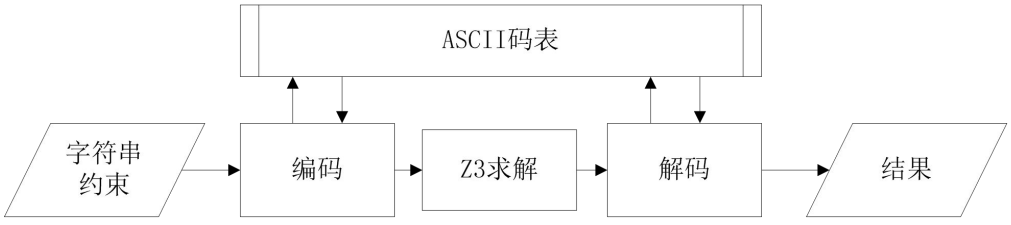

图 4.8 字符串转换过程

其中 strVar[2~9]没有约束，可以为任意值。当约束转换完成后，约束集合将传递给 SMT 求解器进行约束求解。如果约束有解，那么就会传回相应的位向量数组结果。每个位向量都可以转换成一个 0~255 之间的整数，再通过对照 ASCII 码表转换成相应的字符，这样一个位向量数组的解就转换成了一个字符串。整个字符串转换成位

向量的流程如图 4.8 所示。

4.5.3 字符串操作的约束转化

程序中对于字符串的操作函数多种多样，不同语言的操作函数虽然用法不同，但是操作的含义大体相同。以 java 语言为例，常见的字符串操作如表 4.2 所示。

表 4.2 常见的字符串操作

操作	返回值类型	操作含义
s_0.equals(s_1)	布尔型	判断 s_0 与 s_1 是否相等
s_0.contains(s_1)	布尔型	判断 s_0 中是否包含 s_1
s_0.endsWith(s_1)	布尔型	判断 s_0 是否以 s_1 结尾
s_0.startsWith(s_1)	布尔型	判断 s_0 是否以 s_1 开头
s_0.substring(from,to)	字符串型	取 s_0 中的一段字符串
s_0.charAt(index)	字符型	返回 s_0 中 index 位置处字符
s_0.indexOf(s_1)	整型	返回是 s_1 在 s_0 中的位置，没有则返回-1
s_0.concat(s_1)	字符串型	返回合并 s_0 和 s_1 后的字符串

从上一节中得知，一个字符串可以转换成位向量来进行约束求解。如果能将程序中出现的字符串操作函数的语义也转换成对该字符串位向量的约束，那么就可以利用约束求解器求解出满足这些函数约束的字符串的值。

以 s_0.contains(s_1)=true 为例，假设 s_0 和 s_1 的长度分别是 5 和 3。考虑该操作的含义：s_0 中包含 s_1。对应的情况有三种：1，s_0 的前三个字符与 s_1 相等；2，s_0 的中间三个字符与 s_1 相等；3，s_0 的后面三个字符与 s_1 相等。这三种情况之间的关系是逻辑或关系。那么对于 contains 这个函数操作的语义就有对字符串的约束如下：

$$\begin{aligned}&\left(s_0[0]=s_1[0]\wedge s_0[1]=s_1[1]\wedge s_0[2]=s_1[2]\right)\\\vee&\left(s_0[1]=s_1[0]\wedge s_0[2]=s_1[1]\wedge s_0[3]=s_1[2]\right)\\\vee&\left(s_0[2]=s_1[0]\wedge s_0[3]=s_1[1]\wedge s_0[4]=s_1[2]\right)\end{aligned}$$

其中 $s_0[0]$、$s_1[1]$ 等分别代表该字符串中相应位置的字符，紧接着将所有的字符转换为位向量形式的约束就可以传递给约束求解器进行求解。这样，程序中的一个字符串操作函数就转换成了对应的位向量形式的约束表示。对于其他常见的字符串

操作函数，表 4.3 给出了相应的约束形式。

表 4.3 字符串操作对应约束

操作	约束
s_0.equals(s_1)	$s_0[0] = s_1[0] \wedge \ldots \wedge s_0[len] = s_1[len]$
s_0.contains(s_1)	$\forall_{0 <= i < s0len - s1len}\ s_0[i] = s_1[i + s_1len] \wedge \ldots \wedge s_0[i + s_0len] = s_1[s_1len]$
s_0.endsWith(s_1)	$s_0[s_0len - s_1len] = s_1[0] \wedge \ldots \wedge s_0[s_0len] = s_1[s_1len]$
s_0.startsWith(s_1)	$s_0[0] = s_1[0] \wedge \ldots \wedge s_0[s_1len] = s_1[s_1len]$
$s_1 = s_0$.substring(from,to)	$s_0[from] = s_1[0] \wedge \ldots \wedge s_0[to - 1] = s_1[to - 1]$
$s_1 = s_0$.substring(from)	$s_0[from] = s_1[0] \wedge \ldots \wedge s_0[s_0len] = s_1[s_0len - from]$
$s_1 = s_0$.charAt(index)	$s_0[index] = s_1$
i = s_0.indexOf(s_1)	$(s_0[i] = s_1[0] \wedge \ldots \wedge s_0[i + s_1len] = s_1[s_1len]) \wedge (\forall_{0 <= x < i}\ \neg(s_0[x] = s_1[0] \wedge \ldots \wedge s_0[x + s_1len] = s_1[s_1len]))$

假设给定的程序路径中有语句 s_0.equals(s_1)、s_0.endsWith(“end”)、s_1.startsWith(“start”)，且路径的执行要求这三条语句的返回值均为 true。由于变量 s_0、s_1 的长度是未知的，程序的一般做法是从字符串长度为 1 开始，依次计算该长度下对应的约束是否有解。如果计算到了长度上限还没有得出对应解，那么可以认为该路径可能是不可达的。

从上面的三条语句的语义可以看出在字符串长度增长到 8 的时候存在一个可以满足约束的解。此时依据表 4.3 中的约束表达分别将三条语句转换到对应的位向量约束如下：

$$s_0B = new\ BitVector\ [8]\ ;\ s_1B = new\ BitVector\ [8]$$
$$1 : s_0B[0] = s_1B[0] \wedge \ldots \wedge s_0B[7] = s_1B[7] \wedge$$
$$2 : s_0B[5] = 01100101\ \wedge s_0B[6] = 01101110\ \wedge s_0B[7] = 01100100\ \wedge$$
$$3 : s_1B[0] = 01110011\ \wedge s_1B[1] = 01110100\ \wedge s_1B[2] = 01100001\ \wedge$$
$$s_1B[3] = 01110010\ \wedge s_1B[4] = 01110100$$

其中 s_0B、s_1B 分别是是字符串 s_0、s_1 对应的位向量数组，第 1 条约束对应 s_0.equals(s_1)，第 2 条约束对应 s_0.endsWith(“end”)；第 3 条约束对应 s_1.startsWith(“start”)。程序的执行路径中所包含的条件语句必须是同时成立的，这样才能保证程序的执行是按照该条路径唯一确定的。由于这三条语句是包含在程序的一条路径中，也就意味着路径的执行要求这三条约束同时成立，因此这三条约束

之间是逻辑与的关系。

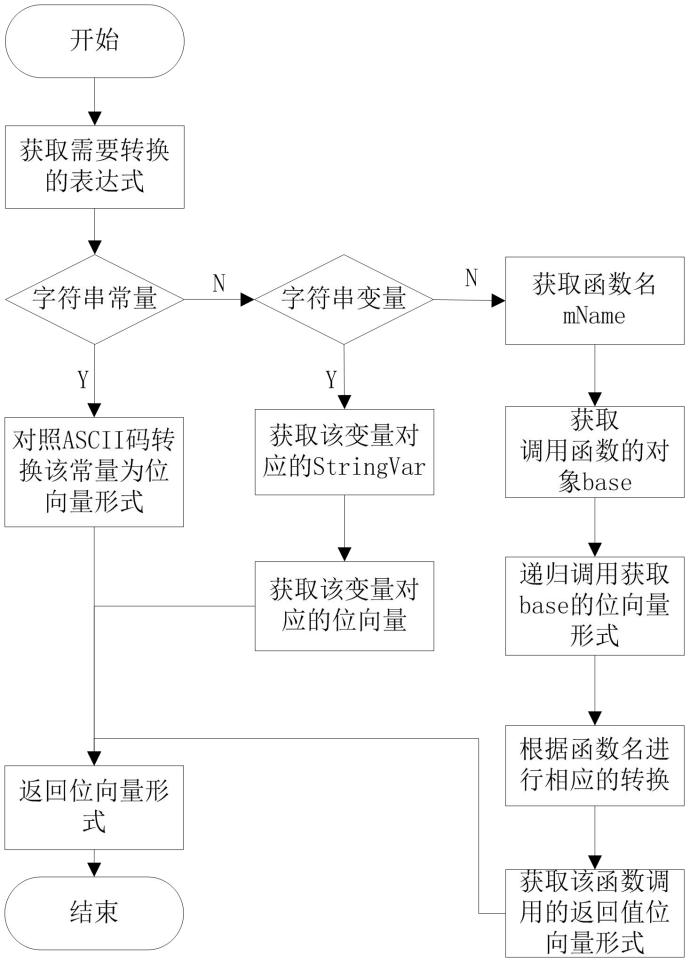

图 4.9 字符串表达式转换流程图

结合 4.5.2 小节内容，这里给出将字符串表达式转换成位向量形式的转换流程图。如图 4.9 所示，首先获取需要转换的表达式，然后判断是否是字符串常量，如果是则对照 ASCII 码表转换该字符串常量为相应的位向量，否则接着判断是否是字符串变量。如果是，那么找出该变量对应的 StringVar，获取该变量中已经初始化过的位向量。如果该表达式既不是常量也不是变量，那么就是含有字符串操作函数的函数调用。获取函数名 mName 以及调用该函数的对象 base，利用递归调用获取 base 对象的位向量形式，接着根据函数名执行后续的转换逻辑，这里的函数名包含 substring、

contact、indexOf 等等。

4.6 本章小结

本章着重讨论了基于漏洞约束求解的测试用例生成系统的设计与实现，给出了系统的框架和处理过程，详细阐述了符号执行器、漏洞约束构造器以及通用约束求解调度器的实现方案。并且给出了在 Z3 约束求解器的基础上实现一个可以解决字符串约束的混合约束求解器实现方案。

5 实验及结果分析

5.1 实验目标

本文提出了基于漏洞约束求解的测试用例生成方法，在符号执行的基础上，利用静态分析的漏洞类型构造漏洞约束，通过约束求解生成有检错能力的测试用例。相比于基于符号执行的测试用例生成工具，本系统的特点是生成的测试用例具有较强的检错性。因此，本次实验的主要目的是验证系统在实际环境中的表现。

5.2 实验环境

1．硬件：

处理器：Intel(R) i3-2100 CPU @ 3.10GHZ

内存：2.92GB

2．软件：

操作系统：Windows 7 (32 位)

开发平台：MyEclipse 10.0、JDK1.6.0_13

其他：Z3：4.3.2

MySQL：5.0

ANTLR：1.4.2

5.3 实验方案和结果分析

鉴于实验目标是验证系统在实际环境中的表现，本次实验方案主要分为两个，实验一用来验证系统在实际程序中的测试用例生成效果；实验二是测试系统的测试用例生成效率，包括生成测试用例所需时间以及生成的测试用例有效性两方面。

5.3.1 漏洞约束效果实验

实验一是验证测试用例生成系统在实际程序中的测试用例生成效果，本次实验采用一个小型商城市场开源程序——Market，具有会员、商品、类别、购物车、支付、配送、促销、排行、公告等基本功能。Market 源码的详细信息如表 5.1 所示。

表 5.1 Market 源码信息统计

测试集	文件个数	代码行数	类个数	方法个数
Market	32	3338	32	275

经过 SSDS 系统中基于安全规则的静态检测框架检测后，总共发现潜在漏洞 5 个，其中 4 个 SQL 注入漏洞，1 个空指针漏洞，分布在 4 个不同的类。图 5.1 是系统检测结果界面。

控制台 漏洞报告

编号	漏洞名称	文件	类	方法	行号
1	SQL注入	Market\src\com\wy\dao\AffficheDao.java	com.wy.dao.AffficheDao	updateAffiche(com.wy.domain.AffficheForm)	112
2	SQL注入	Market\src\com\wy\dao\ManagerDao.java	com.wy.dao.ManagerDao	login(java.lang.String,java.lang.String)	85
3	SQL注入	Market\src\com\wy\dao\ManagerDao.java	com.wy.dao.ManagerDao	selectOne(java.lang.String)	110
4	SQL注入	Market\src\com\wy\dao\MemberDao.java	com.wy.dao.MemberDao	insertMember(com.wy.domain.MemberForm)	55
5	NullPointer	Market\src\com\wy\tool\Logger.java	com.wy.tool.logger	w(java.lang.String)	23

图 5.1 Market 静态漏洞检测结果

如图 5.1 所示，系统检测出潜在的 5 个漏洞后需要对这些漏洞逐一进行动态验证，此时，系统调用测试用例生成模块接口函数，并传递关键路径参数，该关键路径是由路径推荐模块分析后给出。经过测试用例生成子系统处理后，编号 2，3，5 漏洞成功生成测试用例。由于漏洞编号 1，4 所代表的漏洞的参数含有用户自定义的数据结构，系统目前还不支持，因此生成测试用例失败。以编号 2 的漏洞为例，表 5.2 中同时给出了本方法以及在不添加漏洞约束时，仅仅依靠符号执行处理而生成的测试用例，从表中可以看出，在无漏洞约束的情况下，生成的测试用例并没有检测漏洞的特性。值得注意的是，不论是有漏洞约束还是无漏洞约束参与的基于符号执行的测试用例生成方法与随机测试用例生成方法有本质上的差别，前者所生成的每一个

测试用例都能准确覆盖关键路径上的所有语句，后者则依靠大量的随机生成数据才得到少数测试用例来覆盖路径。本次实验充分说明了系统在实际环境中能够准确的生成测试用例从而满足关键路径覆盖以及漏洞检测的需求。

表 5.2 漏洞编号 2 测试用例集

编号	测试用例		无漏洞约束对比	
	参数 1	参数 2	参数 1	参数 2
1	“1’”	?	?	?
2	“and 1=(select count(*) from user); -- ”	?	?	?
3	“admin’ ;-- ”	?	?	?
4	“admin’ or 1=1;”	?	?	?
5	“1’;drop table users;-- ”	?	?	?

5.3.2 系统效率实验

实验二的目的是为了验证测试用例生成系统的执行效率，主要从生成测试用例所需时间以及生成的测试用例有效数两方面考虑。本次实验采用 Juliet Test Suite 测试集，该测试集是由美国国家标准技术研究所（National Institute of Standards and Technology）针对 CWE 中的漏洞分类而创建。针对 CWE 中的每种漏洞，Juliet Test Suite 都有相应的一套测试集。由于目前系统尚未完善，仅完成了针对 Java 的中间表示分析，因此选用针对 Java 的 Juliet Test Suite v1.2 版本，并选取其中的 CWE80_XSS 和 CWE89_SQLInjection 测试集。实验二分为两个部分，一个用于验证系统的自适应测试用例生成特性，另一个用于验证系统的测试用例的生成效果。

本系统具有无需手动干预而自适应的生成漏洞相关的测试用例的特点，为了验证系该特点，实验中分别从 CWE80 和 CWE89 测试集中抽取 10 个测试程序，将这 20 个测试程序一起交由系统进行测试用例生成，具体的实验结果如表 5.3 所示。实验中对 20 个文件分别生成了 20 个测试用例集，每个测试用例集中包含若干测试用例。经过验证，这 20 个测试集对应的测试用例集均具有有效的检测漏洞效果，即系统对于不同的漏洞程序自适应的生成了相应的测试用例集。

表 5.3 系统自适应验证实验结果

测试集	测试集信息		测试用例信息统计	
	文件数量	代码行数	测试用例集个数	有效数
CWE80、CWE89	20	5632	20	20

为了验证系统在大规模程序中的测试用例的生成效果，实验中分别对 CWE80 和 CWE89 中的 1000 个测试程序进行检测。具体的实验数据见表 5.4。

表 5.4 系统效率实验结果

测试集	代码行数	运行时间				生成测试用例集个数
		静态检测（单位：s）	用例生成（单位：s）		总耗时（单位：s）	
			符号执行	约束求解		
CWE80	68001	116.9	5.7	17.0	139.6	783
CWE89	240905	385.5	13.6	31.1	430.1	695

如表 5.4 所示，表中分别是对 CWE80 和 CWE89 中的 1000 个测试程序进行的测试用例生成实验结果数据。从数据中可以看出，系统的整体时间消耗在可接受的范围内，其中大部分的时间消耗在静态检测阶段，这个阶段主要是 IR 的转换耗时过多。在测试用例生成阶段，约束求解的耗时较多，原因之一是测试集中包含大量字符串类型的约束。同时，数据表明有效测试用例集的生成个数分别是 783 和 695，这表明对于测试集中的部分测试程序，系统并没有生成相应的测试用例。下面主要分析此问题的原因。

造成测试用例生成失败的原因之一是 Z3-ss 约束求解器所支持的字符串函数操作有限，而测试程序中可能包含诸如 split 函数以及正则表达式。如图 5.2 所示，该程序片段是节选于 CWE89_SQL_Injection__Environment_executeBatch_81_bad.java 中的一段程序。程序的 34 行中出现了 split 函数，由于系统目前并不支持对该函数的语义转换，所以在约束求解阶段并不能求解出真正满足该程序的测试用例。该原因在进一步完善系统后可以得到有效的解决。导致测试用例生成无效的另一个原因是在程序中调用了 jar 包中的函数过滤输入参数，由于系统无法取得 jar 包中的程序代码，所以无法添加相应的程序约束，因此生成测试用例所用的约束集合是不完整的，导致不能正确的生成测试用例。

```
29      public void action(String data ) throws Throwable
30      {
31
32          if (data != null)
33          {
34              String names[] = data.split("-");
35              int successCount = 0;
36              Connection dbConnection = null;
37              Statement sqlStatement = null;
38              try
39              {
40                  dbConnection = IO.getDBConnection();
41                  sqlStatement = dbConnection.createStatement();
42                  for (int i = 0; i < names.length; i++)
43                  {
44                      /* POTENTIAL FLAW: data concatenated into SQL statement used in executeBatch(), whic
45                      sqlStatement.addBatch("update users set hitcount=hitcount+1 where name='" + names[i]
46                  }
47                  int resultsArray[] = sqlStatement.executeBatch();
48                  for (int i = 0; i < names.length; i++)
49                  {
50                      if (resultsArray[i] > 0)
51                      {
52                          successCount++;
53                      }
54                  }
55                  IO.writeLine("Succeeded in " + successCount + " out of " + names.length + " queries.");
56              }
57              catch (SQLException exceptSql)
58              {
59                  IO.logger.log(Level.WARNING, "Error getting database connection", exceptSql);
60              }
```

图 5.2　CWE89 示例程序段

5.4 本章小结

本章主要叙述了系统的实验部分，通过实验，验证了基于漏洞约束求解的测试用例生成系统的有效性。同时也发现了系统的不足，为以后对系统的改进提供方向。

6 总结与展望

6.1 工作总结

随着软件安全问题得到越来越多的重视，软件测试作为保障软件安全质量的第一道措施也会变得愈加重要。软件测试的质量高度依赖于测试用例的检错能力，同时，随着软件规模的不断扩大，手动的生成测试用例也变得更加困难。因此，研究测试用例自动生成具有重大的理论和实用价值。

本文的主要目的是在已有的静态检测的基础上，研究一种基于漏洞约束求解的测试用例生成方法，使得生成的测试用例能更加具有一定的检错能力。本文的主要工作有如下几点。

1.实现一个基于程序中间表示的符号执行系统。该系统可以接收程序中间表示的输入而不依赖具体的编程语言，使得该系统具有更广泛的应用价值。

2.结合静态分析与符号执行技术，提出基于漏洞约束求解的测试用例生成算法，根据静态分析中的漏洞检测结果生成漏洞约束，在符号执行的基础上添加漏洞约束求解生成具有一定检测能力的测试用例。

3.实现了一个通用约束求解调度框架，并在实验中成功调度了多个约束求解器。同时，实现了一个基于 Z3 的字符串约束求解器，对生成含有字符串的测试用例有很大的意义。

4.设计和初步实现了基于漏洞约束求解的测试用例生成系统，详细描述了各模块的具体设计和实现。

6.2 下一步工作的展望

纵观整个实验过程，系统还存在一定的不足，还有很多需要完善的地方，因此可以将以下方面作为下一阶段研究工作的重点。

1.目前基于中间表示的符号执行系统还处于初步的探索阶段，很多的细节还没有处理好。比如系统目前对程序中出现的数学表达式能够转换成相应约束，但仍需要完善对程序中数学公式函数的理解。同时可以完善对复杂类型参数包括用户自定义类型参数的求解能力。

2.从实验结果来看，对含有大规模的约束求解时，时间消耗较大，这主要集中消耗在约束求解阶段。所以未来可以对约束集合进行一些优化，精简约束集。同时，完善基于 Z3 的混合约束求解器，使其可以应用到更复杂的字符串约束求解中去。

3.研究精简测试用例集方法。实验中大多数测试用例的生成是不必要的，例如 SQL 注入中，很多漏洞约束即使约束求解后存在可行解，但是实际运行中加入该解的 SQL 指令在语法上就是非法、不可运行的。后续的工作可以针对测试用例精简算法进一步的研究。

致 谢

离期将至，望着窗外的梧桐树，心中不禁感慨万千。从金秋九月的叶叶梧桐坠，到初夏的漫天飘絮，一晃便是三年。伴随着梧桐叶的秋落春生，我也慢慢地从对软件测试的一窍不通成长到拥有一定的软件测试知识。感谢华科，同时，也要感谢我的良师益友。

首先，我要感谢我的导师江胜老师。江老师不仅学识丰富，而且为人随和，不管是在工作中还是生活中，与他的讨论常常是欢声笑语一片。在此，感谢江老师在科研中给我的帮助，也感谢在我选择工作时给我的由衷建议。同时，感谢瞿彬彬老师对我生活上的关怀和科研上的指导。瞿老师为人真诚，治学严谨，从瞿老师那里我学到了很多关于软件测试的知识。

再次感谢两位老师在读研期间对我的谆谆教诲，能在华科遇到二位老师是我的荣幸。同时也感谢卢炎生老师在科研工作上的指导和生活上的关怀。

当然，也要感谢实验室的师兄师姐们，虽然他们大多已经毕业，还是要感谢他们在我刚进入华科时对我生活和科研上的指引。感谢同一届的李花、舒泽林、王烨、曹星辰、王广龙、赵旭、陈敏、胡太祥、朱金华等，非常享受与大家在一起的快乐时光，希望以后能有更多的机会相聚一起。

最后，对我的家人说一句谢谢，感谢家人这么多年对我的支持。我会以我的实际行动来表达对家人由衷的感谢。

参考文献

[1] Banerjee C, Pandey S K. Research on software security awareness: problems and prospects. ACM SIGSOFT Software Engineering Notes, 2010, 35(5): 1~5

[2] Hao Y, Jia Y, Cui B, et al. OpenSSL HeartBleed: Security Management of Implements of Basic Protocols. In: Proceedings of P2P, Parallel, Grid, Cloud and Internet Computing. Guangzhou, China:IEEE, 2014. 520~524

[3] CNVD.http://www.cnvd.org.cn/webinfo/show/3399

[4] CNVD.http://www.cnvd.org.cn/webinfo/show/3536

[5] Naik S, Tripathy P. Software testing and quality assurance: theory and practice. John Wiley & Sons,2011.10~11

[6] Goodenough J B,Gerhart S L. Toward a theory of test data selection. Software Engineering, IEEE Transactions,1975 (2): 156~173

[7] Hetzel W C, Hetzel B. The complete guide to software testing. John Wiley & Sons, Inc., 1991.12~14

[8] Myers G J . The art of software testing. Wiley, 2004.23~24

[9] Anand S, Burke E K, Chen T Y, et al. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 2013, 86(8): 1978~2001

[10] Ferguson R, Korel B. The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology (TOSEM), 1996, 5(1): 63~86

[11] Beizer B. Software Testing Techniques.Dreamtech Press, 2002.5~11

[12] Kaner C. Improving the maintainability of automated test suites. Software QA, 1997, 4(4):233~238

[13] Edvardsson J. A survey on automatic test data generation.In: Proceedings of the 2nd Conference on Computer Science and Engineering. Linkping,Sweden: ECSEL,1999. 21~28

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

[15] Person S, Yang G, Rungta N, et al. Directed incremental symbolic execution.ACM SIGPLAN Notices,2011, 46(6): 504~515

[16] Khurshid S,Pasareanu C, Visser W. Generalized symbolic execution for model checking and testing. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2003.553~568

[17] Castro M,Costa M,Martin J P. Better bug reporting with better privacy.In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems.New York, USA : ACM, 2008. 319~328

[18] Zhang P, Elbaum S G,Dwyer M B.Automatic generation of load tests.In: Proceedings of the 26th IEEE on ACM International Conference on Automated Software Engineering. Washington, DC, USA:IEEE,2011.43~52

[19] Hassan Reza, Sandeep Endapally, Emanuel Grant. A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets. In: Proceeding of International Conference on Information Technology. Washington,DC, USA : IEEE, 2007.278~289

[20] 陈志德. 基于模型的面向对象测试用例生成研究:[硕士学位论文]. 合肥: 中国科学技术大学, 2010.

[21] 黄晓玲，袁兆山，黄超男.一个基于 FSM 测试自动化方案与实现.合肥工业大学

学报:自然科学版,2008,31(1):32~47

[22] 钱忠胜. 基于模型的 Web 应用测试用例生成方法:[博士学位论文]. 上海: 上海大学, 2008.

[23] Nie C, Leung H. A survey of combinatorial testing. ACM Computing Surveys (CSUR), 2011, 43(2): 11

[24] Seroussi G, Bshouty N H. Vector Sets for Exhaustive Testing of Logic Circuits. IEEE Transactionson Information Theory, 1988, 34(3):513~522

[25] 黄如兵. 组合测试用例的自适应随机生成与优先级排序方法研究:[博士学位论文]. 武汉: 华中科技大学, 2013.

[26] Shiba T, Tsuchiya T, Kikuno T. Using artificial life techniques to generate test cases for combinatorial testing.In: Proceedings of the 28th Annual International IEEE on Computer Software and Applications Conference. Washington, DC, USA: IEEE , 2004.72~77

[27] Duran J W, Ntafos S C. An evaluation of random testing. Software Engineering, IEEE Transactions on, 1984, 10(4): 438~444

[28] Bird D L, Munoz C U. Automatic generation of random self-checking testcases. IBM systems journal, 1983, 22(3): 229~245

[29] Chen T Y, Eddy G R, Merkel G, Wong P K. Adaptive random testing through dynamic partitioning. In: Proceedings of the 4th International Conference on Quality Software.Washington,DC, USA:IEEE,2004. 79~86

[30] 邢颖, 宫云战, 王雅文等. 基于分支限界搜索框架的测试用例自动生成. 中国科学: 信息科学, 2014, 44(10): 1345~1360

[31] 江胜. 基于进化测试的用例生成方法研究:[博士学位论文]. 武汉: 华中科技大学,2009.

[32] De Moura L, Bjørner N. Z3: An efficient SMT solver.In:Tools and Algorithms for the

Construction and Analysis of Systems. Berlin Heidelberg:Springer, 2008.337~340.

[33] Dutertre B, de Moura L. A fast linear-arithmetic solver for DPLL(T). In:Proceedings of the 18th International Conference on Computer Aided Verification.Berlin Heidelberg:Springer,2006. 81~94

[34] Ganesh V, Dill D L. A decision procedure for bit-vectors and arrays. In:Proceedings of the 19th International Conference on Computer Aided Verification. Berlin Heidelberg:Springer, 2007.519~531

[35] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. Washington, DC, USA: IEEE, 2008.209~224

[36] Cadar C,Ganesh V,Pawlowski P M,et al. EXE: automatically generating inputs of death.ACM Transactions on Information and System Security, 2008,12(2): 10~25

[37] Anand S, Pasareanu C S, Visser W. JPF－SE: A Symbolic Execution Extension to Java PathFinder. In:Tools and Algorithms for the Construction and Analysis of Systems.Berlin Heidelberg: Springer , 2007.134~138.

[38] Havelund K, Pressburger T. Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer, 2000, 2(4): 366~381

[39] Tillmann N, de Halleux J. Pex-White Box Test Generation for .NET.In: Tests and Proofs. Berlin Heidelberg: Springer , 2008.134~153

[40] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing.ACM Sigplan Notices, 2005, 40(6): 213~223

[41] Godefroid P,Levin M Y,Molnar D A. Automated Whitebox FuzzTesting.NDSS, 2008, 8: 151~166

[42] Sen K, Marinov D, Aghag. CUTE: A concolic unit testing engine for C. Symposium on the foundations of software engineering, 2005,30(5): 263~272

[43] S. Liu, R. Zhang, D. Wang,et al. Implementing of Gaussian Syntax-Analyzer Using ANTLR.In: Proceedings of the 2008 International Conference on Cyberworlds. Piscataway, NJ, USA: IEEE, 2008. 613～618

[44] 陈凌明. Java 程序的安全性分析方法研究:[硕士学位论文]. 武汉: 华中科技大学, 2012.

[45] Vallee-Rai R, Hendren L J. Jimple: Simplifying Java Bytecode for Analysis and Transformations. Sable Technical Report, School of Computer Science, McGill University, Montreal, Canada, 1998

[46] Vallee-Rai R, Co P, Gagnon E, et al. Soot-A Java bytecode optimization framework. In: Proceedings of the 20th Annual CASCON Conference. New York, USA : ACM, 2010. 214～224

[47] Galeotti J P, Fraser G, Arcuri A. Extending a search-based test generator with adaptive dynamic symbolic execution.In:Proceedings of the 2014 International Symposium on Software Testing and Analysis. New York ,USA: ACM, 2014.421~424

[48] Cadar C, Godefroid P, Khurshid S, et al. Symbolic execution for software testing in practice: preliminary assessment.In: Proceedings of the 33rd International Conference on Software Engineering. New York, USA : ACM, 2011: 1066~1071

[49] Bucur S, Ureche V, Zamfir C et al. Parallel symbolic execution for automated real-world software testing.In: Proceedings of the sixth conference on Computer systems. New York, USA : ACM, 2011: 183~198

[50] 万成铖.一种基于符号执行的 Java 缺陷检测工具的设计与实现: [硕士学位论文]. 北京: 北京大学,2012.

[51] Zhang J, Chen X, Wang X. Path-oriented test data generation using symbolic execution and constraint solving techniques. International Conference on Software Engineering and Formal Methods. IEEE, 2004:242~ 250

[52] Luckow K S, Pasareanu C S, Luckow K S, et al. Symbolic PathFinder v7. ACM Sigsoft Software Engineering Notes, 2014, 39(1):1~5

[53] MITRE Corporation.http://cwe.mitre.org/

[54] 王欢. 静动态结合的安全漏洞检测方法研究:[硕士学位论文]. 武汉: 华中科技大学,2014.

[55] Zheng Y, Zhang X, Ganesh V. Z3-str: A z3-based string solver for web application analysis.In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. New York, USA:ACM, 2013: 114~124

[56] Kiezun A, Ganesh V, Artzi S, et al. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. ACM Transactions on Software Engineering and Methodology , 2012, 21(4): 25:1~25:28

[57] 叶亮.基于安全规则的源代码分析方法研究: [硕士学位论文].武汉: 华中科技大学,2013.

[58] Saxena P, Akhawe D, Hanna S, et al. A symbolic execution framework for javascript. In:Security and Privacy (SP), 2010 IEEE Symposium on.Oakland, CA, USA: IEEE,2010.513~528

附录 攻读学位期间参与的科研项目

[1] 国家部委“十二五”预研项目《软件安全分析技术》

[2] 横向项目《舰艇指挥控制软件的安全测试》

[3] 湖北省自然科学基金项目编号：2014CFB1006　基于安全规则的软件安全漏洞检测方法研究