西安科技大学

学 位 论 文 诚 信 声 明 书

本人郑重声明：所呈交的学位论文（设计）是我个人在导师指导下进行的研究（设计）工作及取得的研究（设计）成果。除了文中加以标注和致谢的地方外，论文（设计）中不包含其他人或集体已经公开发表或撰写过的研究（设计）成果，也不包含本人或其他人在其它单位已申请学位或为其他用途使用过的成果。与我一同工作的同志对本研究（设计）所做的任何贡献均已在论文中做了明确的说明并表示了致谢。

申请学位论文（设计）与资料若有不实之处，本人愿承担一切相关责任。

学位论文（设计）作者签名：　　日期：

学 位 论 文 知 识 产 权 声 明 书

本人完全了解学校有关保护知识产权的规定，即：在校期间所做论文（设计）工作的知识产权属西安科技大学所有。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。本人允许论文（设计）被查阅和借阅；学校可以公布本学位论文（设计）的全部或部分内容并将有关内容编入有关数据库进行检索，可以采用影印、缩印或其它复制手段保存和汇编本学位论文。

保密论文待解密后适用本声明。

学位论文（设计）作者签名：　　指导教师签名：

年　　月　　日

分 类 号　　密 级

学校代码 10704　　学 号 140810224

西安科技大学

学 士 学 位 论 文

题目：基于 JSA 的 JAVA 程序字符串变量静态分析

作者：宋 磊

指导教师：刘晓建　　专业技术职称：副教授

学科专业：软件工程　　申请学位日期：2018 年 6 月

摘　要

随着软件广泛地应用在各行各业，软件的安全问题也随之而来，例如数据库安全，恶意程序调用敏感 API 进行攻击等。字符串变量是程序中普遍使用的变量之一，很多安全问题和恶意攻击都与字符串变量值有关。由于字符串变量取值随着程序运行不断变化，简单的搜索程序中的字符串常量并不能有效的获取变量值，因此必须通过程序分析的方法分析字符串变量的可能取值范围，进而对程序的安全性进行分析。

JSA（Java String Analyze）是目前学术界广泛使用的一种 Java 程序字符串分析的开源程序包，但是由于其开源特性，它几乎没有相应的技术支持，文档、实施案例等也很不完善，因此在安全性分析中使用 JSA 较为困难。

本文主要完成四方面工作：一是，下载并分析了 JSA 的源码、技术文档，并对主要案例进行了分析和调试，并最终运行成功；二是，针对 SQL 注入漏洞，基于 JSA 分析包，编写了漏洞检测和分析程序；三是，采用 JSA 对 Java 程序中的反射调用接口参数进行了分析；四是，完成了 JSA 相关技术资料的分析和整理，形成了相关技术文档，为后续学习者提供了学习资料。

关键词：安全性；字符串分析；SQL 注入；反射

ABSTRACT

With the wide application of software in all walks of life, the security problems of software come with it, such as database security, malicious programs calling sensitive APIs to attack,etc. String variables are one of the most commonly used variables in programs, and many security problems and malicious attacks are associated with string variable values. Since the value of the string variable constantly changes with the running of the program, the string constants in simple search program not effectively obtain the value of the variable. Therefore, the ranger of possible values of the string variable must be analyzed through program analysis, and then analyze the safety of the procedure.

JSA(Java String Analyze) is presently widely used a Java program String analysis of open source packages. However, because of the open source characteristics, it has almost no corresponding technical support, documentation, implementation of the case and so on also is not very perfect. It is more difficult to use JSA in security analysis.

The paper mainly completes four aspects of work: First, download and analyze the source code and technical documents of JSA, analyze and debug the main cases, and finally run successfully. Secondly, SQL injection vulnerability, vulnerability detection and analysis program was written based on JSA analysis package. Thirdly, JSA is adopted to analyze the interface parameters of reflection calls in Java programs. Fourth, the competition of the analysis and collation of JSA related technical data. Formed the relevant technical documents, and provided learning materials for subsequent learners.

Key Words: Security; String Analysis; SQL injection; Reflection

目　录

1 绪 论

1.1 选题背景及研究意义

程序分析是软件工程的基本技术之一，并在软件工程的很多方面得到了应用。早期的软件主要用于计算，所以传统的程序分析主要针对程序中的数值和运算。对于字符串变量和字符串操作，传统的程序分析仅将其视为不可确定的数值和运算。例如，对于字符串变量的分支条件，传统程序分析只能假定字符串变量的值是一个任意值，从而简单地判定任何分支都是可达的；对于字符替换操作，传统的程序分析也只能简单判断这一操作的所有操作数（原始字符串、操作字符、被操作字符）都是操作输出的可能数据来源。实际上，被操作字符串不可能是操作输出的数据源，对于其他操作数，能否作为输出的数据源，还需要具体分析。

随着软件在各个方面的广泛应用，字符串在程序中的地位日益增加。例如，在访问数据库的程序中，SQL 语句一般都是通过字符串拼接构造而成的。图形软件的界面中也包含大量的字符串，并通过字符串操作动态地生成很多需要显示的字符串。而 Internet 时代主要软件类型的 Web 应用软件中，字符串更是其界面的基本构成元素，Web 应用软件的整个用户界面、以及界面的相关源代码都由字符串操作产生。在当前被广泛采用的多数主流语言（例如 Java、PHP、C#等）中，字符串都是基本类型。而且这些语言的基本库函数中都包含一些字符串操作函数，例如字符串比较、字符替换、字符串切分等。因此，软件工程中涉及字符串的任务有很多，为了支持这些任务，需要一些技术来分析程序中的字符串。

1.2 相关工作

目前的字符串分析分为两个阶段：字符串值分析和字符串约束求解。其中字符串值分析主要针对程序中的某个字符串变量（称为“热点”），通过分析程序的控制流和数据流得出该变量的一系列信息，如该热点的可能取值、来源是否可靠等。字符串约束求解是指，在字符串值分析结果的基础上，判断该结果是否满足事先给定的约束，进而开展特定分析任务，如静态分析、漏洞检查、测试用例生成等。

下面分别介绍字符串值分析及字符串约束求解。

1.2.1 字符串值分析技术

字符串值分析是一种通过分析程序，判断热点变量的值域的一种程序分析方法，输

入是源程序或者编译之后的目标程序，输出即是描述热点变量的所有可能取值的一个符号系统。通常要求字符串值分析输出的符号系统能够描述热点的所有可能取值，但该符号系统也可能引入热点取不到的一些值。

根据使用的符号系统的不同，可将现有字符串值分析分为基于正则文法的字符串值分析、基于上下文无关文法的字符串值和基于一元谓词的二阶逻辑的字符串值分析。

1． 基于正则文法的字符串值分析

基于正则文法的字符串值分析是最早提出的一种针对字符串的程序分析方法，因此在文献中也直接称为字符串分析（string analysis），它是 2003 年由 Christensen 等人[1]提出来的。

Christensen 等人的方法是使用正则文法作为表示字符串变量值域的符号系统，并使用预定义的自动机映射模拟字符串操作，包括以下四个步骤：

1）将软件源代码转化为静态单赋值形式

首先，将待分析程序的源代码转化为静态单赋值形式（static single assignment 简称 SSA）[2]，静态单赋值形式是 1991 年提出的一种源代码形式。在将源代码转化为静态单赋值形式的过程中，若某个变量被多次赋值，通过别名分析会将该变量拆分成多个变量。

2）提取字符串操作文法

使用程序数据依赖分析对静态单赋值形式的代码进行分析，结果是包含程序中所有变量、常量、表达式之间数据依赖关系的一个程序数据依赖图。热点变量是这个字符串依赖图中的一个结点。字符串分析通过在程序数据依赖图上进行可达分析，删去热点变量不可达的结点，并删去那些不对应字符串类型的变量、常量、表达式的结点，从而得到一个字符串依赖图。

字符串操作文法是一个扩展的上下文无关文法。字符串操作文法定义为一个六元组(N, T, S, P, OP_1, OP_2)。其中，N 为非终结符的集合，T 为终结符的集合，S 为起始非终结符，P 为产生式集合，OP_1为一元字符串操作集合（例如 java 语言中的 reverse 操作），OP_2为二元字符串操作的集合（例如 Java 语言中的对字符串进行 replace 的操作）。

3）对字符串操作文法进行线性近似，得到一个正则操作文法

正则近似的目标是求出一个正则操作文法 N，这个正则操作文法的语言 L(N)是字符串操作文法 G 的语言 L(G)的尽可能小的超集。由于字符串操作文法是扩展的上下文无关文法，因此可以使用对上下文无关文法进行线性近似的 Mohri-Nederhof 算法[3]对字符串操作文法进行近似。对上下文无关文法进行正则近似的基本思路是，通过使用$\sum *$近似去掉上下文无关文法的左递归圈和右递归圈，从而得到一个包含字符串操作的正则操作文法。

4）消除字符串操作

字符串分析的最后一个步骤是消除正则操作文法中的字符串操作，从而得到一个真正的正则文法来估计热点变量的所有可能取值。

基于正则文法的字符串值分析是最早被提出来的字符串值分析方法，其优点是分析速度较快。缺点是由于正则语法的表达能力有限，得出的正则文法通常不是很准确。另外，由于这一方法未能使用统一的模型描述字符串操作，因此可扩展性较差，Yu 等人[4][5]对基于正则文法的字符串值分析加以改进，引入自动机变化模拟字符串操作，从而避免了对每个字符串操作定义相应的映射。

2. 基于上下文无关文法的字符串值分析

2005 年，Minamide[6]在基于正则文法的字符串值分析的基础上，提出了基于上下文无关文法的字符串值分析。该方法使用上下文无关文法作为描述字符串变量值域的符号系统，使用有限状态转换机（finite state transducer）[7]模拟字符串操作。

基于上下文无关文法的字符串分析主要分为三个步骤，其中前两步与基于正则文法的字符串分析相同。在第三步中，Minamide 提出使用一个有限状态转换机模拟每一个字符串操作，并且对于每一个字符串操作OP，使用有限状态转换机的转换算法构造一个文法 G，通过有限状态转换机之后的文法G′，将G′加入原文法 OP 的位置即可消除 OP，通过不断地消解字符串操作，即可将包含字符串操作的文法直接转换为一个普通的上下文无关文法。例如：对于字符串操作 str_replace(“00”,”0”,$x)，可以通过如图 1-1 所示的有限状态转换机模拟。在图 1-1 中，a 表示除 0 以外的字符串。转换机共有 0，1，2 这 3 个状态，0 状态是初始状态，1 和 2 是终结状态。转换机的状态转移上形如 X/Y 的规则表示当转换机接受输入 X 时会输出 Y。例如，0/0a 表示接受输入 0 并输出 0a。一个输入 00abc11 经过该有限状态转换机后，输出为 0abc11。

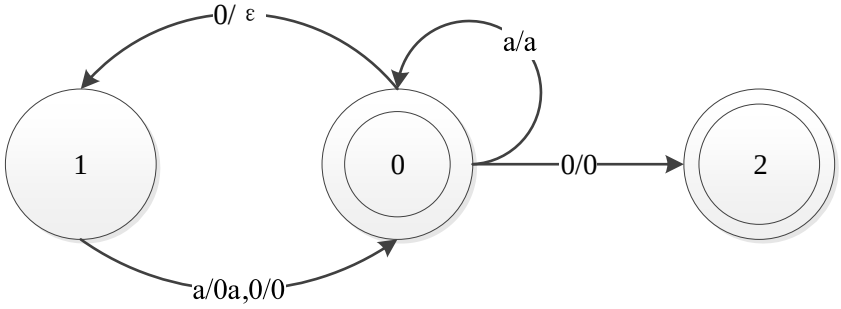

图 1-1 一个有限状态转换机

由于 Minamide 提出的方法使用一个上下文无关文法来近似表示热点字符串变量或表达式的取值，而上下文无关文法较正则文法表达能力更强，因此能够更准确地刻画热

点变量的取值。

3. 基于一元谓词的二阶逻辑的字符串值分析

2011 年，Tateishi 等人[8]提出了一种基于带一元谓词的二阶逻辑（monadic second-order logic，简称 MSOL）的字符串分析方法。该方法的核心思想是，使用在字符串上定义的带一元谓词的二阶逻辑表达式M2L(str)作为符号系统描述热点变量的值域，并使用逻辑推导公式模拟字符串操作。M2L(str)的基本组成部分是逻辑表达‘a’(t)，当热点变量的位置 t 上的字符串是‘a’时，这一表达式的值为真，根据程序中生成热点变量值的过程对这一基本逻辑表达式进行组合，可以生成热点变量对应的逻辑表达式Expr，使得 Expr 为真的所有字符串即为热点变量的所有可能取值。例如，当$\mathrm{Expr} = 'a'(1)\wedge['b'(2)\vee'c'(2)]$时，即表达该分析方法求出的热点变量的取值可能为 ab 或 ac。

由于这一字符串值分析方法能够处理与字符串中的字符下标相关的逻辑表达式，因此这种方法能够处理与字符串下标相关的字符串操作，例如 IndexOf 操作等。目前，这一方法也是唯一能够处理与字符下标相关的字符串操作的字符串值分析方法。

4. 字符串值分析比较

表 1-2 给出了现有字符串值分析方法之间的比较，在表 1-2 中，分别比较了现有字符串值分析的技术特点。从表 1-2 中可以看出，在现有的四种字符串值分析方法中，大部分方法使用形式语言（正则语言或上下文无关语言）描述字符串变量的取值，Tateishi 等人的方法使用带一元谓词的二阶逻辑描述字符串变量的取值，四种方法分别使用不同的字符串操作处理方式，但只有 Tateishi 等人的方法是路径敏感的，且可以处理字符下标。基于正则语言的两种字符串值分析方法的主要优点是计算效率高，缺点是正则语言的表达能力较差，结果不够准确。基于上下文无关文法的分析方法计算效率有所降低，但是表达能力有所增强。Tateishi 等人的方法准确度最高，主要的问题是缺乏相应的字符串约束求解方法。

表 1-2 字符串值分析方法比较

字符串值分析方法		Christensen等人(2003)	Minamide等人(2005)	Yu等人(2008)	Tateishi等人（2011）
技术特点	表示字符串变量值域的符号系统	正则语言	上下文无关文法	正则语言	MSOL
	字符串操作处理方式	映射规则	有限状态转换机	自动机变换	逻辑推导公式
分析准确度	是否路径敏感	否	否	否	是
	能否处理字符下标	否	否	否	是
主要优点		效率高	效率高，易拓展	表达能力强，易拓展	表达能力强，准确度高，易扩展
主要缺点		不易扩展，准确度低，表达能力差	准确度低，表达能力较差	准确度较低，效率较低	效率低，缺少复杂约束求解的算法

1.2.2 字符串约束求解技术

字符串约束求解的目的是对于给定的字符串约束，判断是否存在字符串值满足这一约束。在使用字符串分析求出字符串变量的可能取值后，经常需要判断这些值是否满足给定的约束，从而达到正确性验证、安全性检测等目的。字符串约束求解的方法很多，主要分为下面两类。

1. 基于文法交集求解的字符串约束求解

正则文法求交集的算法以及正则文法与上下文无关文法之间求交集的算法是计算机语言理论方面的基本算法[9]。在实际的应用中，存在很多与文法包含相关的约束。例如要判断一个表示 SQL 语句的字符串变量是否会取到非法的 SQL 语句，可以约束该变量的取值包含于 SQL 文法推出的语言。由于文法包含关系可以通过求交集判断($A \cap \overline{B} = \emptyset \rightarrow A \subseteq B$)，因此，文法交集求解算法自然就成为一种字符串约束求解算法。这种算法可以求解文法包含方面的约束。Klarlund 等人根据字符串约束求解的特点实现了这一算法，并开发了学术和产业界广泛使用的工具 MONA。

2. 基于布尔向量约束求解的字符串约束求解

基于文法交集的字符串约束求解的最大缺点就是无法求解与值相关的判断约束，例如字符串相等、字符串长度比较等。针对这一问题，很多学者开始研究基于布尔向量约束求解的字符串约束求解。这一方法的基本思想就是将字符串解码为布尔向量，然后使用布尔向量约束求解算法求解。

2 JSA 介绍

JSA 全称为 Java String Analyze, 是 Aarhus 大学计算机科学系的 Christensen，Moller 以及 Michael 合作开发的工具，这个工具可以分析 Java 程序中的字符操作和字符串。JSA 分为前端和后端，前端的主要任务是将 Java 程序转化成流图。后端的主要任务是分析流图并生成有限状态自动机。整个研究流程如图 2-1 所示：

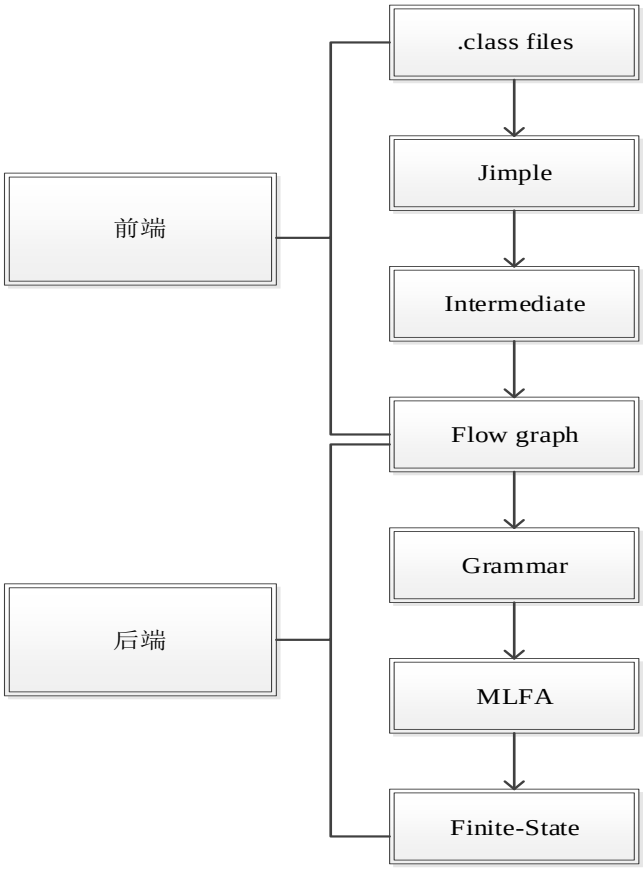

图 2-1 JSA 研究流程图

2.1 JSA 前端

JSA 前端把 Java 中.class 文件当作输入来分析，最终的输出结果是一个流图。我们一般称.class 文件为应用类，在一个应用类中，存在着可以作为热点的字符串表达式（可以有多个热点）。对于每一个热点，JSA 会在运行时系统中计算它，然后输出一个有限状态自动机。这个自动机是一个合理的近似值，因为输出的自动机会接受比实际更多的字符串。

对于如何把.class 文件转化成 Jimple 代码，可以参考 Soot 手册[10]。

2.1.1 Jimple 转换到 Intermediate

图 2-2 展示整个转化的流图。为了使依赖关系非常明确，所有的方法调用都在没有 Impl 后缀的接口中声明。

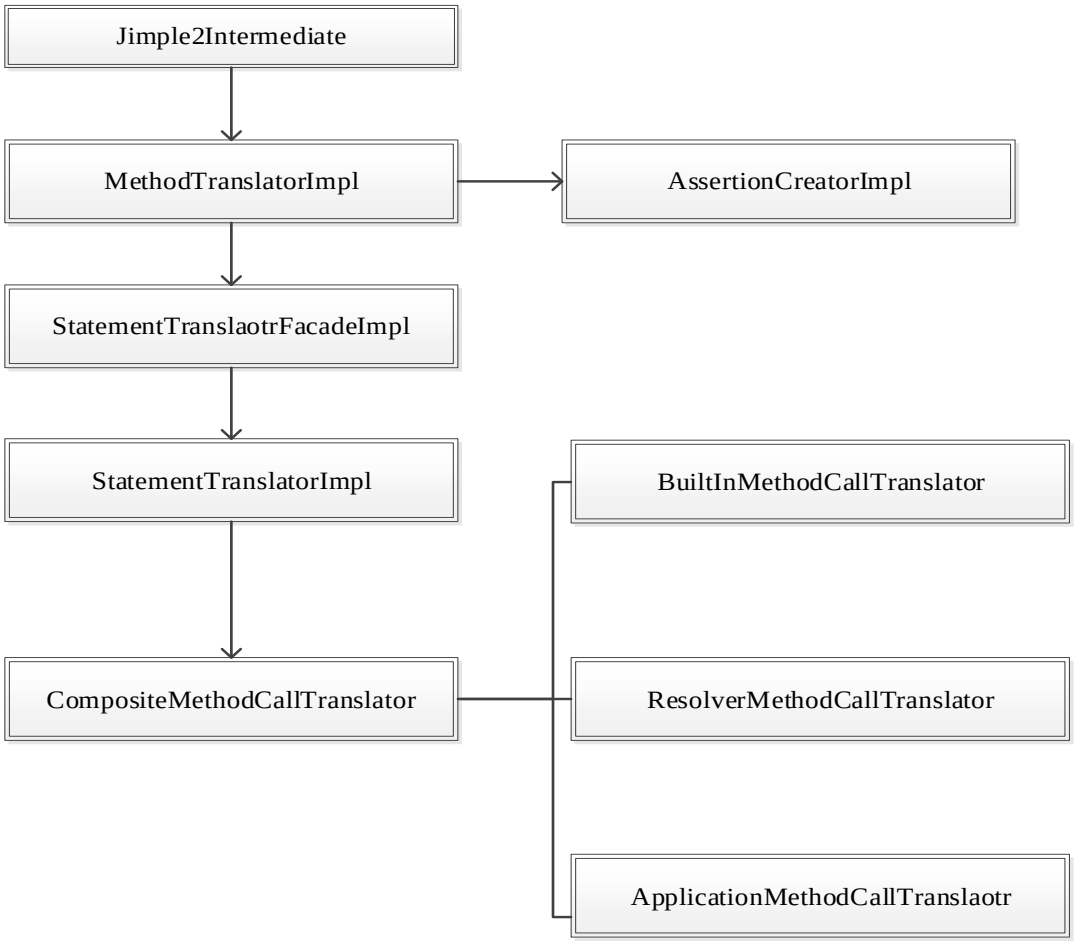

图 2-2　Jimple 到 Intermediate 转化过程

2.1.2 Intermediate 转换到流图

从 Intermediate 程序到流图需要进行下面的分析，执行的顺序如下：

1）字段使用分析

2）活性分析

3）检测无效别名断言

4）别名分析

5）可达性分析

6）检测无效操作断言

2.2 JSA 的流图

JSA 的流图是 JSA 前端和后端的一个转折点，流图之前是和 java 相关的工作，替换掉前端，就可以将字符串分析应用到除了 java 以外的其他语言。。流图会对一个执行字符操作的程序进行抽象的描述。因为流图中只有灵活使用的边，所以控制流被抽象出来。

2.2.1 流图的定义

流图会捕捉在程序中的字符流和字符串操作，其他东西抽象处理。流图中的结点代表一个变量或者表达式，流图中的边代表可能的数据流。更准确的说，一个流图由以下类型的一个有限结点集合构成：

1. 初始点：构造字符串值，例如常量或者 Inter.toString 函数，并且关联一个符合 *reg* 表示一个正则的语言$[\![reg]\!]$代表可能的字符串。
2. 加入点：一个分配的结点或者其他结点的位置
3. 合并：一个字符连接
4. 一元操作：一个一元字符串操作，例如 setCharAt 或者 reverse，关联一个符合 *op1* 表示函数$[\![op1]\!]$：$\Sigma^* \to \Sigma^*$。字符串操作的非字符串参数被认为是函数符号的一部分。
5. 二元操作：一个二元字符串操作，例如 Insert，关联一个符号 *op2*，代表函数$[\![op2]\!]$：$\Sigma^* \times \Sigma^* \to \Sigma^*$。

初始结点没有输入边，加入点也许有任意数量的输入边，每一个一元操作都有一个明确的输入边，每一个合并和二元操作结点都有一个有序的输入边，它们代表了各自的参数。注意到我们流图的概念本质上是一种静态单赋值的形式加入结点对应的 Φ 功能。

2.2.2 流图的构造

当具备可达性的时候，流图的结点已经创建，并且边也会随之创建。对于每一个已经定义的本地变量，每个语句都会为之精确构造一个结点。根据可达性，这些结点之间会产生边。简单的说，如果从 A 到 B 有一个定义，那么我们可以创建一条从 A 结点到 B 结点的边。

因为每一个字段变量都有一个带着使用点的结点，所有字段变量的处理是比较困难的。为此，定义字段变量的语句将在自己结点和字段变量结点之间添加一条边，但使用

字段变量的语句将字段变量的结点添加到自己的结点中。这意味着在字段上可能只有弱更新，但是对于多线程应用来说，分析是合理的。

1. 通过 Tricky 例子构造流图

Tricky 例子如下所示：

```
public class Tricky {
    String bar(int n, int k, String op) {
        if (k == 0) return "";
        return op + n + "]" + bar(n - 1, k - 1, op) + " ";
    }
    String foo(int n) {
        StringBuffer b = new StringBuffer();
        if (n < 2) b.append("(");
        for (int i = 0; i < n; i++) b.append("(");
        String s = bar(n - 1, n / 2 - 1, "*").trim();
        String t = bar(n - n / 2, n - (n / 2 - 1), "+").trim();
        return b.toString() + n + (s + t).replace(']', ')');
}
```

通过 Tricky 程序产生的流图如图 2-3 所示：

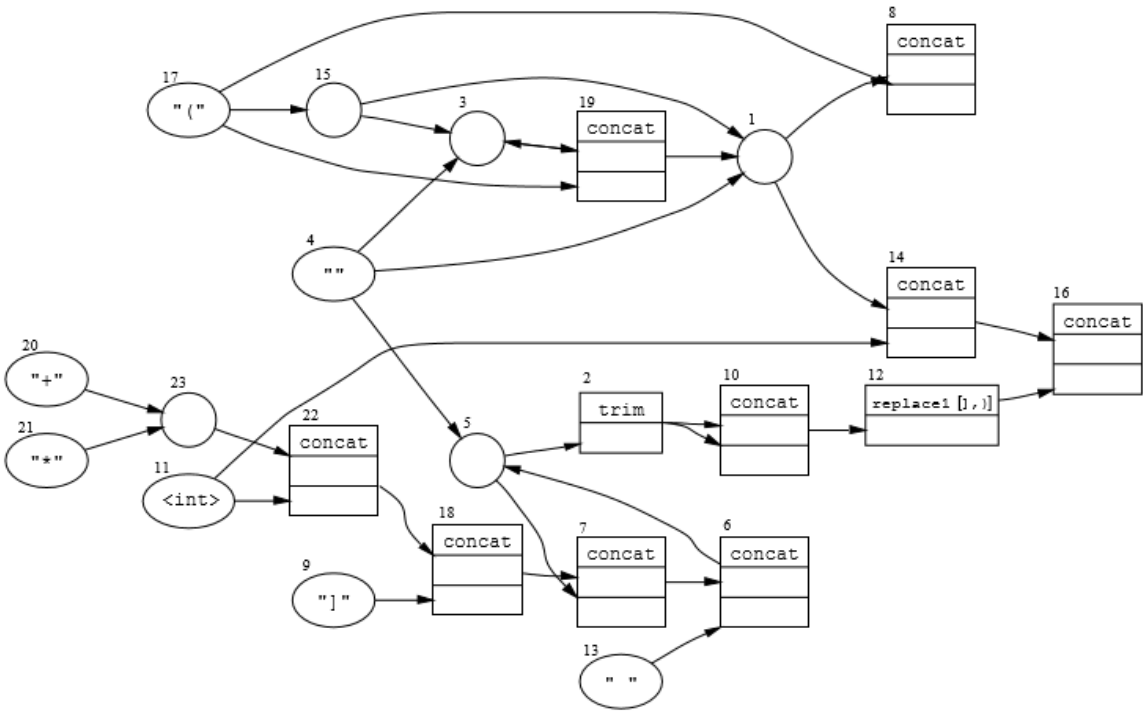

图 2-3 从 Tricky 程序产生的流图

最右边的结点与输出的单一热点所对应。

流图的语义被定义为约束系统的最小解。这样 $\mathrm{F}(n)$为每个包含源程序的所有可能值的结点 n 或对应于 n，生成一个映射$\mathrm{F}: \mathrm{N} \rightarrow \Sigma^{*}$。根据下面的规则生成约束：

$\mathrm{F}(n) \supseteq [\![reg]\!]$	对于每一个初始结点 n
$\mathrm{F}(n) \supseteq \mathrm{F}(m)$	对于每一个从结点 n 到结点 m 的边
$\mathrm{F}(n) \supseteq \mathrm{F}(m)\mathrm{F}(p)$	对于每一个带有边（m,p）的合并结点 n
$\mathrm{F}(n) \supseteq [\![op1]\!](\mathrm{F}(m))$	对于每一个带着来自 m 的边的一元结点 n
$\mathrm{F}(n) \supseteq [\![op2]\!](\mathrm{F}(m)，\ \mathrm{F}(p))$	对于每一个带着边（m,p）的二元结点 n

2.3 JSA 后端

JSA 的后端是整个 JSA 的核心，主要任务是分析流图并生成有限状态自动机。流程分为下面几个：

1. 将流图转换成一个上下文无关文法。
2. 使用 Mohri-Nederhof 算法对流图转换出来的上下文无关文法进行处理产生强正则文法。
3. 通过强正则文法构造多层次有限状态自动机。

2.3.1 构造上下文无关文法

从流图中，我们将构造一个特殊上下文无关文法，对于每一个流图结点$\mathrm{n} \in \mathrm{N}$, 都关联一个非终结符$A_n$。这个语法有下列的特性：对于每一个结点 n，语言$\mathcal{L}(A_n)$（也就是说，带有A_n语法的语言是作为开始的非终结符）是和$F(n)$一样的。

首先，我们定义可以产生操作的一个上下文无关文法，并将操作结果定义为一个语法，其中的例子如下所示：

$X \rightarrow Y$	[单个]
$X \rightarrow Y\ Z$	[一对]
$X \rightarrow reg$	[正则]
$\mathrm{X} \rightarrow \mathrm{op}_1(Y)$	[一元操作]
$\mathrm{X} \rightarrow \mathrm{op}_2(Y,Z)$	[二元操作]

X,Y,Z 都是非终结符，这个语法的语言被我们所期望的那样定义：对于产生式$X \rightarrow reg$，X 可以派生出在$[\![reg]\!]$的所有字符串。因为产生操作的原因，文法不一定是上下文无关的。

从流图转换到文法是非常简单的：对每一个结点 n，我们添加一个非终结符A_n和一组对应于 n 的输入边的结果。

1. 对于一个带有文法 reg 的初始结点，增加$A_n \to reg$
2. 对于一个结合点，为每个包含到结点 n 的边的结点 m 增加$A_n \to A_m$ 。
3. 对于一个合并结点，增加$A_n \to A_m A_p$，m 和 p 都是对应于 n 的输入边的结点。
4. 对于一个带有操作op_1 的一元操作结点，增加$A_n \to \mathrm{op}_1\,(A_m)$，其中 m 是有一条到结点 n 的边的结点。
5. 对于一个带有操作op_2的二院操作结点，增加$A_n \to \mathrm{op}_2(A_m, A_p)$，$m$ 和 p 都是对应于 n 的输入边的结点。

生成的文法的大小在流图中是线性的。Tricky 的文法如图 2-4 所示：

$X_1 \to X_4$ $X_1 \to X_{15}$ $X_1 \to X_{19}$ $X_2 \to \texttt{trim}(X_5)$
$X_3 \to X_{19}$ $X_3 \to X_{15}$ $X_3 \to X_4$ $X_4 \to \texttt{""}$
$X_5 \to X_4$ $X_5 \to X_6$ $X_6 \to X_7\ X_{13}$ $X_7 \to X_{18}\ X_5$
$X_8 \to X_1\ X_{17}$ $X_9 \to \texttt{"]"}$ $X_{10} \to X_2\ X_2$ $X_{11} \to \texttt{<int>}$
$X_{12} \to \texttt{replace[],)}](X_{10})$ $X_{13} \to \texttt{" "}$ $X_{14} \to X_1\ X_{11}$ $X_{15} \to X_{17}$
$X_{17} \to \texttt{"("}$ $X_{16} \to X_{14}\ X_{12}$ $X_{18} \to X_{22}\ X_9$ $X_{19} \to X_3\ X_{17}$
$X_{20} \to \texttt{"+"}$ $X_{21} \to \texttt{"*"}$ $X_{22} \to X_{23}\ X_{11}$ $X_{23} \to X_{20}$
$X_{23} \to X_{21}$

图 2-4 Tricky 的文法

2.3.2 正则近似

我们希望化简先前章节中产生的语法，使得新的语言包含了原来的语法。由于左线性和右线性上下文无关文法有效地定义了规则语言，因此我们可以借鉴这个思想。将最终结果拓展为强正则表达式，如下面所解释的。

在 Mohri-Nederhof 算法中[3]，我们通过把非终端作为结点，并将其从左侧的非终端到右侧的一条边，来找出语法中的强连接组件。可以通过 Tricky 例子获得图 2-5。

Mohti-Nederhof 近似算法要求在所有操作过程中，右侧出现的非终端与左侧非终端属于不同的组件。基于这个原因，我们首先消除包含操作过程的循环：对于每一个使用的一元操作符，我们要求一个字符设置近似值$[\![op_1]\!]C: 2^{\Sigma} \to 2^{\Sigma}$，对于属于$S^*$集合的字符串 x，$[\![op_1]\!]C(S)$可能包含一系列字符，这些字符可能存在$[\![op_1]\!](x)$中，二元操作与之类似。将这些近似算法应用在语法上的一个简单的定点，对于每一个非终结符 X，寻找$C(X) \subseteq \Sigma$，其包含所有可能在 X 中出现的字符。对于每一个循环，我们用一个带有$X \to r$的操作代替之，r 证明了正则语言$C(X)^*$。经过转换后，强连接组件将会重新计算。对于 Tricky 例子，在循环中既没有修剪也没有代替操作发生。

注意到与第三节的流程图相似，两个标记的节点组对应于重要的强连接组件。

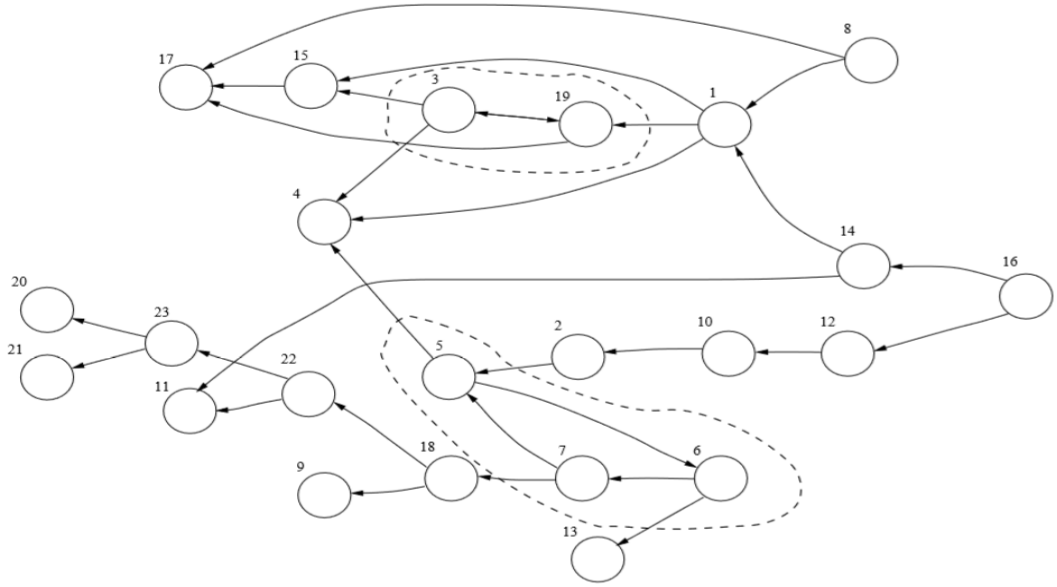

图 2-5 对 Tricky 使用 Mohri-Nederhof 算法之后获得的流图

我们把每一个非强正则规则组件 M 转换成右线性，这是 Mohri-Nederhof 算法的应用，该转换遵循下列规则：对于每一个 M 中的非终结符 A，增加新的非终结符 A'。如果 A 对应一个热点或者不仅仅在组件 M 中有作用，则增加一个产生式 $A' \to e$，e 代表 $\{\varepsilon\}$。然后将所有含有 A 的产生式根据图 2-6 中的产生式进行替换：

$$
\begin{array}{lcl}
A \to X & \rightsquigarrow & A \to X\ A' \\
A \to B & \rightsquigarrow & A \to B,\quad B' \to A' \\
A \to X\ Y & \rightsquigarrow & A \to R\ A',\quad R \to X\ Y \\
A \to X\ B & \rightsquigarrow & A \to X\ B,\quad B' \to A' \\
A \to B\ X & \rightsquigarrow & A \to B,\quad B' \to X\ A' \\
A \to B\ C & \rightsquigarrow & A \to B,\quad B' \to C,\quad C' \to A' \\
A \to reg & \rightsquigarrow & A \to R\ A',\quad R \to reg \\
A \to op_1(X) & \rightsquigarrow & A \to R\ A',\quad R \to op_1(X) \\
A \to op_2(X,Y) & \rightsquigarrow & A \to R\ A',\quad R \to op_2(X,Y)
\end{array}
$$

图 2-6 应用 Mohri-Nederhof 算法的产生式

在这里，A，B 和 C 都是 M 中的非终结符，X 和 Y 都是 M 之外的非终结符，每一个 R 都是一个新产生的非终结符。直观上看，当考虑到一个特殊的组件 M，我们也许可以将 M 之外的非终结符作为终结符，因此所有带有操作产生式的循环都被消除，操作变量不再属于 M。

经过这次转换，组件现在是右线性组件。它的大小与原尺寸成正比，它是在线性时间

内构造的，与 Mohri 和 Nederhof 的应用相比，语法总是有一个固定的启动非终结符，我们的应用对于所有对应于热点的非终结符都要求正则近似。通过构造，在原始语法中，非终结符热点的语言始终是近似语法中同一个非终端语言的子集。

我们要求每一个一元操作符op_1使用一个保守的正则近似（例如：以自动化操作的形式）$[\![op_1]\!]R: REG \rightarrow REG$，$REG$ 属于正则语言，二元操作符与之类似。当语法中使用的操作被它们的近似对等物代替时，每个非终结符的语言保证是正则的。

对$A' \rightarrow e$ 产生式增加限制对我们应用时必须的。考虑下面的语法：

$$S \rightarrow T\ S \mid \mathrm{a}$$

$$T \rightarrow S +$$

该语法接受一个 a+a+a…+a 形式的字符串，这种字符串可以从一个简单的 Java 递归函数中构造。没有限制的话，就是增加$T' \rightarrow e$，所以由此产生的语法是可接受的，例如字符串 a+是一个不能接受的粗略近似。相反，目前的算法产生一个对应于正则表达式 a(+a)*的近似值，这是我们所希望的。

Tricky 例子包含一个非强正则组件$\{X_5, X_6, X_7\}$，近似算法将其转换为图 2-7 所示的文法：

$X_1 \rightarrow X_4$ $X_1 \rightarrow X_{15}$ $X_1 \rightarrow X_{19}$ $X_2 \rightarrow \texttt{trim}(X_5)$
$X_3 \rightarrow X_{19}$ $X_3 \rightarrow X_{15}$ $X_3 \rightarrow X_4$ $X_4 \rightarrow \texttt{""}$
$X_5 \rightarrow X_4\ X_5'$ $X_5 \rightarrow X_6$ $X_6' \rightarrow X_5'$ $X_6 \rightarrow X_7$
$X_7' \rightarrow X_{13}\ X_6'$ $X_7 \rightarrow X_{18}\ X_5$ $X_5' \rightarrow X_7'$ $X_8 \rightarrow X_1\ X_{17}$
$X_9 \rightarrow \texttt{"]"}$ $X_{10} \rightarrow X_2\ X_2$ $X_{11} \rightarrow \texttt{<int>}$ $X_{12} \rightarrow \texttt{replace[],)}](X_{10})$
$X_{13} \rightarrow \texttt{" "}$ $X_{14} \rightarrow X_1\ X_{11}$ $X_{15} \rightarrow X_{17}$ $X_{16} \rightarrow X_{14}\ X_{12}$
$X_{17} \rightarrow \texttt{"("}$ $X_{18} \rightarrow X_{22}\ X_9$ $X_{19} \rightarrow X_3\ X_{17}$ $X_{20} \rightarrow \texttt{"+"}$
$X_{21} \rightarrow \texttt{"*"}$ $X_{22} \rightarrow X_{23}\ X_{11}$ $X_{23} \rightarrow X_{20}$ $X_{23} \rightarrow X_{21}$
$X_5' \rightarrow \texttt{""}$

图 2-7 转化之后 Tricky 的文法

注意导致正则近似中不精确的来源：Mohri-Nedefhof 转化处理发生在非强连接正则组件中的连接操作。而其他字符操作是由自动操作所制定的规则近似处理的，所以字符集使用了最粗略的近似来打破操作产生式的循环。

2.3.3 多层次有限状态自动机

我们从强正则语法中提取出自动机[11]。然而，考虑到文法不仅仅只有一个非终结符，并且还会有特殊的操作产生式，因此我们使用一个新颖的形式，即具有两个特性的多层次有限状态自动机（Multi-Level Finite Automata, MLFA：1）强正则语法可以在线性时间中转换成一个同等的多层次有限状态自动机，2）对于每一个热点，我们可以从多层次有

限状态自动机中有效的提出一个最小确定性（正常）自动机。

我们定义一个 MLFA，这个 MLFA 是由一组有限的状态 Q 和一组转换$\delta \subseteq Q \times T \times Q$构成，其中 T 是下列种类的一组标签。

$-reg$

$-\epsilon$

$-(p,q)$

$-0p_1(p,q)$

$-0p_2((p_1,q_1),(p_2,q_2))$

每一个 p 和 q 都是从 Q 中来的状态。所以那里必须存在如下的一个水平映射$l: Q \to N$:

$$-(s,(p,q),t)\epsilon\delta \Rightarrow l(s) = l(t) > l(p) = l(q)$$

$$-(s,0p_1(p,q),t)\epsilon\delta \Rightarrow l(s) = l(t) > l(p) = l(q)$$

$-i = 1,2$ 时$(s,0p_2((p_1,q_1),(p_2,q_2)),t)\epsilon\delta \Rightarrow l(s) = l(t) > l(p_i) = l(q_i)$

也就是说，转换的端点都在同一级别，但是转换中提到的状态总是处于比端点更低的级别。一个转换的语言$\bar{\mathcal{L}}$是根据它的类型来定义的：

$$\bar{\mathcal{L}}(reg) = [\![reg]\!]$$

$$\bar{\mathcal{L}}(\in) = \{\in\}$$

$$\bar{\mathcal{L}}\big((p,q)\big) = \mathcal{L}(p,q)$$

$$\bar{\mathcal{L}}\big(0p_1(p,q)\big) = [\![OP_1]\!]\mathrm{R}(\mathcal{L}(p,q))$$

$$\bar{\mathcal{L}}(0p_2((p_1,q_1),(p_2,q_2))) = [\![OP_2]\!]\mathrm{R}(\mathcal{L}(p_1,q_1),\mathcal{L}(p_2,q_2))$$

对于$q \in Q$和$x \in \Sigma^*$,使得$\bar{\delta}(q,x) = \{p \in Q | (q,t,p) \in T \wedge x \in \bar{\mathcal{L}}(t)\}$，并且使$\hat{\delta}$：$Q \times \Sigma^* \to 2^Q$被下列所定义：

$$\hat{\delta}(q,\epsilon) = \bar{\delta}(q,\epsilon)$$

$x \neq \epsilon$时，$\hat{\delta}(q,\epsilon) = \bar{\delta}(q,\epsilon) = \{r \in Q | r \in \bar{\delta}(p,Z) \wedge p \in \bar{\delta}(q,y) \wedge x = yz \wedge z \neq \epsilon\}$

Q 有一个开始状态 s 和一个终止状态 f, 在这两个状态得语言$\mathcal{L}(s, f)$中，$l(s) = l(f)$a像$\mathcal{L}(s, f) = \{x \in \Sigma^* | f \in \hat{\delta}(s,x)\}$一样被定义。因为处于同一水平，所以这是良好的定义

先前部分的一个强正则语法遵循下列规则被转换成一个多层次有限状态自动机：首先，对于每一个非终结符 A 构造状态q_A，另外，对于每一个强联系组件 M 构造状态q_m。然后，对于每一个组件 M，根据 M 的类型和在 M 中左边的结果来添加转换。对于一个简

单组件或者右线性组件进行下列转化：

$$
\begin{array}{lll}
A \to B & \rightsquigarrow & (q_B, \epsilon, q_A) \\
A \to X & \rightsquigarrow & (q_M, \Psi(X), q_A) \\
A \to B\ X & \rightsquigarrow & (q_B, \Psi(X), q_A) \\
A \to X\ Y & \rightsquigarrow & (q_M, \Psi(X), p),\ (p, \Psi(Y), q_A) \\
A \to reg & \rightsquigarrow & (q_M, reg, q_A) \\
A \to op_1(X) & \rightsquigarrow & (q_M, op_1(\Psi(X)), q_A) \\
A \to op_2(X, Y) & \rightsquigarrow & (q_M, op_2(\Psi(X), \Psi(Y)), q_A)
\end{array}
$$

图 2-8 对简单或右线性组件的转化式

对于一个左线性组件进行下列转化：

$$
\begin{array}{lll}
A \to B & \rightsquigarrow & (q_A, \epsilon, q_B) \\
A \to X & \rightsquigarrow & (q_A, \Psi(X), q_M) \\
A \to X\ B & \rightsquigarrow & (q_A, \Psi(X), q_B) \\
A \to X\ Y & \rightsquigarrow & (q_A, \Psi(X), p),\ (p, \Psi(Y), q_M) \\
A \to reg & \rightsquigarrow & (q_A, reg, q_M) \\
A \to op_1(X) & \rightsquigarrow & (q_A, op_1(\Psi(X)), q_M) \\
A \to op_2(X, Y) & \rightsquigarrow & (q_A, op_2(\Psi(X), \Psi(Y)), q_M)
\end{array}
$$

图 2-9 对左线性组件的转化式

每一个 p 代表一个新状态。函数Ψ的每一个非终结符映射到某个状态：如果 A 属于一个简单或者右线性组件 M，则$\Psi(A) = (q_A, q_M)$，其他情况下，$\Psi(A) = (q_M, q_A)$。这一结构的本质是标准或左线性语法转换为自动机[12]。这个构造是正确的，一个非终结符 A 的语法$\mathcal{L}(A)$等价于$\mathcal{L}(\Psi(A))$。

基于源程序中的热点，我们找到流图结点 n，进而对应于一个语法非终结符An，这个终结符与在多层次有限状态自动机 F 中的状态$(s, f) = \Psi(An)$相关联。使用下面的算法，我们从这一对状态中提取了一个正常的非确定性自动机，其语言为$\mathcal{L}(s,\ f)$。

1. 对于$l(q) = l(s)$的 F 中的状态 q，在 U 中构造一个状态q'，让q'和f'成为各自的开始和最终状态。
2. 对于$l(q_1) = l(q_2) = l(q_3)$的 F 中的每一个转变(q_1, t, q_2)，从q_1'到q_2'中增加一个等价的子自动机：如果 $t = reg$，我们使用一个语法为$[\![reg]\!]$的子自动机，类似的，应用到$t = \epsilon$。如果$t = (p,\ q)$，则子自动机是根据算法$\mathcal{L}(p,\ q)$，递归地应用提取得到的。如果$t = op_1(p,\ q)$，则子自动机的语法为$[\![op_1]\!]R(\mathcal{L}(p, q))$，类似的对于$t = op_2((p_1, q_1), (p_2, q_2))$。

这个理论建设性的展示了多层次有限状态自动机定义正则语言。因为子自动机可能会被复制，因此 U 的大小最坏情况下是 F 的大小。因此我们决定最小化 U，最终 DFA 的

大小最坏情况是双指数。我们的应用使用了备忘，例如自动机对于$(s，f)$的状态只会计算一次。对于有许多热点的程序来说，计算的重用是非常重要的，特别是涉及到普通的子计算。

我们现在可以看到的好处表示在所有的阶段，而不仅仅是在一元和二元的操作。例如，应用字符集近似所有操作处于初级阶段，那些操作流程图不发生在建模于高精度的循环上。例如，插入函数可以用自动机操作来精确建模，但是在流图或语法级别上很难实现。

2.4 本章小结

本章系统的对 JSA 原理进行详细的讲解。通过上文的理论分析，我们可以看出：JSA 前端对 Java 程序进行了保守的建模，JSA 后端对上下文无关文法的正则近似也是保守的。所以从流图到上下文无关文法的转换，从强规则文法到 MLFAs 的转换，以及从 MLFAs 中提取 DFAs 都是准确的。这意味着，如果某个程序点上的 Java 程序在执行过程中可能产生一个特定的字符串，那么该字符串将被保证为程序点所提取的自动机所接受。

3 SQL 注入的检测方法

3.1 SQL 注入攻击简介

3.3.1 概述

结构化查询语言 SQL 是用来和关系数据库进行交互的文本语言。它允许用户对数据进行有效的管理，包括对数据的查询、操作、定义和控制等方面，例如向数据库写入、插入数据、从数据库读取数据等。

关系数据库广泛存在于软件以及网站中，用户一般通过界面和数据库进行交互。安全性考虑不周的数据库软件或者网站使得攻击者能够构造并提交恶意语句，将特殊构造的 SQL 语句插入到提交的参数中，在和关系数据库进行交互时获得私密信息，或者修改数据等。这就是所谓的 SQL 注入攻击。

3.3.2 技术原理

SQL 注入攻击的主要是通过巧妙地构造 SQL 语句，和程序的内容结合起来对数据库进行注入攻击。比较常用的技巧有：使用注释符号、恒等式（如 1=1）;使用 union 语句进行联合查询、使用 insert 或 update 语句插入或修改数据等。

在关系型数据库中，数据一般都是以表的方法存储。假设存在一个名为 user 的表格，包含有 id、username 和 pwd 三个列，分别表示了用户 ID、用户名、密码。表 3-1 中列举了几种简单的构造方法，其中“$username”和“$password”等字符串为变量名称。

表 3-1 SQL 注入语句实例

语句	说明
正常语句：select * from user where username = ‘$username’ AND pwd = ‘$password’; 注入语句：select * from user where username = ‘jack’/*’ and pwd = ‘’;	通过注释符“/*”，将注释符后面的内容“AND pwd = ‘’”注释掉，这样攻击者就可以不用验证密码而直接登陆。
正常语句：select * from user where username = ‘$username’ and pwd = ‘$password’; 注入语句：select * from user where username = ‘jack’ and pwd = ‘’ or ‘2=2’;	这条语句通过 2=2 这个恒等式作为逻辑判断，无论 pwd=”是什么，该语句依然能够得到正确的执行，使得攻击者不使用密码而成功登陆。
正常语句：update user set pwd = ‘$password’ where username = ‘$username’; 注入语句： update user set pwd = 'abcd'or 1=1 /* where username = 'tom';	这条语句利用单引号以及后续的恒等式和注释，将数据库所有的密码都修改为“abcd”。

续表 3-1 SQL 注入语句实例

语句	说明
正常语句：select * from user where username = ‘$username’ 注入语句:：select * from user where username = ‘jack into outfile ‘c://text.txt’/*’ and pwd = ‘’	这条语句到相关路径下去寻找文件，其中路径的内容就是 jack 这个用户的相关信息。
正常语句：insert into user values(‘$id’, ’$username’ ,’$password’,’1’); 注入语句：insert into user values(‘23’,’jack’,’pwd’, ’3’)/*’,’1’);	这条语句新插入了一个 ID 为 10 用户 tjack,并将其权限直接设定为最高的 3，这种攻击方法可利用 SQL 注入注册高权限用户。
正常语句：select * from user where username = ‘$username’; 注入语句：select * from user where username = ‘tom’ and left(pwd,1) = ‘1’;	这条语句利用内置函数来判断字符串中某个位置的字符，如果成功则显示出用户 ID。这样反复的尝试下去，便会得到真正的密码。

3.2 SQL 注入攻击检测

对SQL语句进行分析，分析程序是基于JSA的。分析流程图如图3-2所示。

测试之前，我们利用sqlAutomaton类生成一个SQL语句自动机，这个自动机只接受不存在SQL注入的SQL语句，这个自动机具有577个状态和2312条推导式，这是我们分析程序的关键。

另外，我们还需要选取热点变量，因为程序中的字符串非常多，如果每一个字符串都需要去检测的话，工作量会特别大并且效率低。在本次测试中，我们选择三个函数作为我们的热点变量，分别为：

1. java.sql.Statement: boolean execute(java.lang.String)。
2. java.sql.Statement: java .sql.ResultSet executeQuery(java.lang.String)。
3. java.sql.Statemnt: int executeUpdate(java.lang.String)。

之所以选择这三个函数作为热点，是因为所有的SQL语句都需要通过这三个函数与数据库交互，不能与数据库交互的SQL语句对于我们的分析而言，是毫无意义的。

我们需要分析的SQL语句存在于热点变量的参数中，所以在目标程序发现热点变量时，将该参数传给之前的自动机，若自动机可以接受这个参数，说明该参数是一条不具有SQL注入的SQL语句；若自动机不接受这个参数，说明该参数具有潜在的SQL注入风

险，此时会返回一个最小反例。

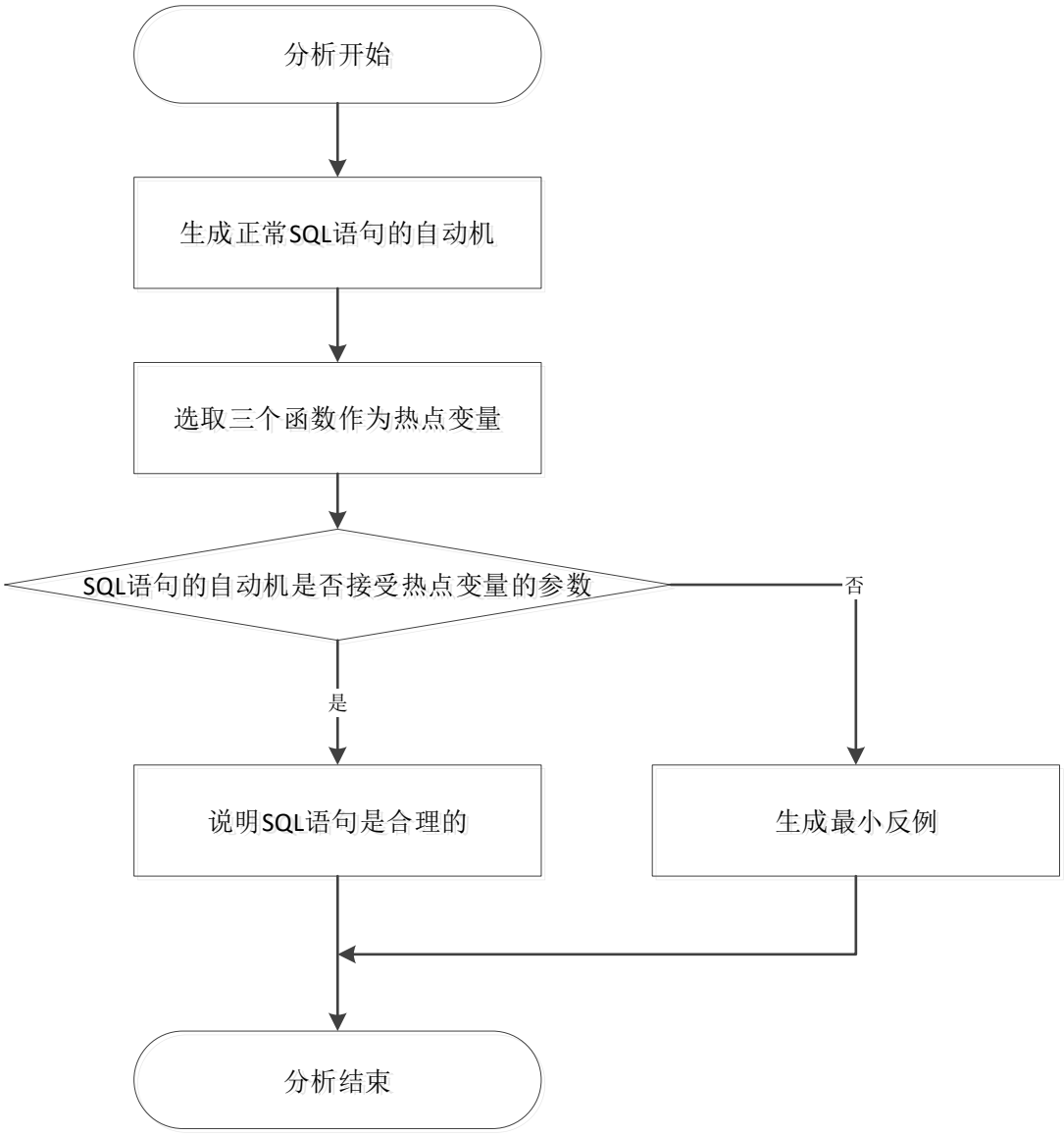

图3-2 SQL注入检测流程图

3.2.1 检测案例 Select 语句

首先，我们先看几行代码：

```
String sql1 = “select * from user where username = ‘tom’ AND pwd = ‘pwd’”;
String sql2 = “select * from user where username = ‘tom’/*’ AND pwd = ‘”;
ResultSet rs1 = state.executeQuery(sql1);
ResultSet rs2 = state.executeQuery(sql2);
```

sql1的语句一般用在登陆时候，可以看出，sql2语句因为注释掉了后面对密码的验

证，所以当’tom’登陆的时候，他不需要验证密码就可以成功登陆。分析结果如图3-3所示：

```
Analyzing...
Checking query at line 22 in SelectSQL.java...
Query might be invalid. Violated by:
"select * from user where username = 'tom'/*' AND pwd = ''"
Checking query at line 21 in SelectSQL.java...
Query is valid!
```

图3-3 分析select语句结果

可以看出，对于代码中执行的 select * from user where username = ‘tom’ AND pwd = ‘pwd’语句，分析程序给出了“Query is valid!”的提示，说明这条语句是一条合理的SQL语句，它的执行不会对数据库产生任何的威胁。但是，当分析到 select * from user where username = ‘tom’/*’ AND pwd = ‘’这条语句，分析给出了一个和这条一样的反例。这是因为这条语句不会被之前产生的SQL自动机所接受，换句话说，这条SQL语句会对数据库产生威胁。

3.2.2 检测案例 Insert 语句

和之前的案例一样，我们先看几行代码：

```
String sql1 = “insert into user values(‘$id’,’username’,’$password’,’1’)”;
String sql2 = “insert into user values(‘10’,’tom’,’password’,’3’)/*’,’1’)”;
ResultSet rs1 = state.executeQuery(sql1);
ResultSet rs2 = state.executeQuery(sql2);
```

sql1的语句是简单的插入语句，不会发生什么意外。但是sql2语句会发生严重的情况，因为它创建了一个不应该存在的高权限用户，这可能会对数据库产生毁灭性的打击。分析结果如图3-4所示：

```
Analyzing...
Checking query at line 22 in InsertSQL.java...
Query might be invalid. Violated by:
"insert into user values('10','tom','password','3')/*','1')"
Checking query at line 21 in InsertSQL.java...
Query is valid!
```

图3-4 分析insert语句结果

从程序给出的结果可以看出，和我们之前的分析结果并无差别。所以程序的分析结果是合理的，可信的。

3.2.2 检测案例 Update 语句

接下来，我们看看Update语句。先看下面的几行代码：

```
String sql1 = “Update user set pwd = ‘password’ where username = ‘tom’”;
String sql2 = “Update user set pwd = ‘abcd’ or 1=1 # where username = ‘tom’”;
ResultSet rs1 = state.executeQuery(sql1);
ResultSet rs2 = state.executeQuery(sql2);
```

从上面的代码，我们可以看出，sql1是正常的修改语句，不会发生什么意外。但是sql2会发生严重的问题，因为sql2中，有一个永真语句‘1=1’，并且用‘#’符号注释掉后面的where语句，所以这条语句会导致数据库所有的密码都被修改为“abcd”，造成严重的后果。分析结果如图3-5所示：

```
Analyzing...
Checking query at line 22 in UpdateSQL.java...
Query might be invalid. Violated by:
"update user set pwd = 'abcd' or 1=1 # where username = 'tom'"
Checking query at line 21 in UpdateSQL.java...
Query is valid!
```

图3-5 分析update语句结果

可以看出，和我们之前的推断没有什么差距。分析程序识别出包含SQL注入的SQL语句，达到了我们的预期。

3.2.4 检测案例 Delete 语句

之前的案例，我们都是用现成的SQL语句去分析，但是实际情况却与大不相同。因为在很多时候，SQL语句都是用字符串拼接而成，所以需要我们的程序对此类问题也要能做出正确的解释。我们先看看下面的代码：

```
String sql = “Delete From users where username = ‘” +uid+ “’”;
int ok = state.executeUpdate(sql);
```

从上面的代码，可以看出，sql是一条包含Delete语句的字符串，并且它是由一条字符串常量加一个字符串变量拼接而成，其中的uid变量就成为了隐形的SQL注入点。例如，当我们要删除“tom”时，我们输入tom，构造的SQL语句为：delete from user where username = ‘tom’，这个时候的SQL语句是没有问题的。但是，如果我们输入“tom’ or 1=1 ”，构造的SQL语句为：delete from user where username = ‘tom’ or 1=1，此时的SQL语句因为永真式 1 =1 的存在，会删除数据库所有的数据，造成无法挽回的情况。分析结果如图3-6所示：

```
Analyzing...
Checking query at line 23 in DeleteSQL.java...
Query might be invalid. Violated by:
"delte users where username = ''"
```

图3-6 分析delete语句结果

可以看出，针对具有潜在漏洞的 SQL 语句，分析程序给出的反例为：DELETE FROM users WHERE user = '\n'。这是因为，我们的分析是静态分析，不能掌握被分析程序真正运行时候的状态，但是由于这样拼接的 SQL 语句具有极大的风险，所以分析程序会生成一个最小的反例来指出这条 SQL 语句可能会在运行时候发生 SQL 注入。这种提醒是非常有用且合理的。

3.3 本章小结

本章首先介绍了 SQL 注入，以及 SQL 注入的原理和常见语句，然后利用分析程序分别对 SELECT、INSERT、UPDATE、DELETE 语句进行了分析。分析结果是有效的，也达到了我们预期的成果。对于具有 SQL 注入的 SQL 语句，分析程序会把这个语句当作反例输出，而对于动态拼接而成的 SQL 语句，分析程序也给出了最小的反例，这是非常有用的。

4 反射 API 的检测方法

4.1 Java 反射技术简介

Smith 早就在 1982 年提出了反射的概念，主要是指程序可以访问、检测和修改它本身状态或行为的一种能力。这一概念的提出很快引发了计算机科学领域关于应用反射性的研究。它首先被程序语言的涉及领域所采用，并在 Lisp 和面向对象方面取得了成绩。近几年，反射技术也应用在视窗系统、操作系统和文件系统中。反射并不是一个新生物，很容易让我们想到光学中的反射概念，虽然计算机科学赋予了反射概念新的内涵，但是，从现象上来说，计算机的反射概念和光学的反射概念的确有相似之处，这些相似之处有助于我们的理解。在计算机科学领域，反射是指一类应用，它们能够自描述和自控制。也换句话说，这类应用采用反射机制来实现对自己状态及行为的概述和检验，并且根据自己的状态，可以随时修改应用所描述行为的相关语句。

同光学的反射概念相比，计算机科学领域的反射不仅仅指反射本身，还包括对反射结果所采取的方法。所有采用反射机制的系统（即反射系统）都希望自己的系统更加开房。也就是说，实现反射机制的系统都具有开放性，开放性是反射系统的必要条件。除此之外，反射系统还必须满足原因连接。所谓原因连接是指对反射系统自描述的改变能够反映到系统底层的实际状态和行为上的情况。开放性和原因连接是反射系统的两大基本要素。

在 Java 中，反射机制是指在程序运行过程中：对于某一个类，程序能够知道这个类的所有属性和方法；对于任意一个对象，程序能够调用这个对象的任意方法和属性。从上面 Java 反射的定义就可以看出，Java 的反射机制破坏了类的封装性，使得类的所有属性和方法暴露出来，包括私有的属性和方法。其次，当恶意程序使用反射来调用一些敏感 API 的时候，特别是通过字符串的形式调用 API 时，付出的代价特别小，但是却对软件安全造成了巨大的威胁。

4.2 反射敏感 API 检测

接下来，我们会用一个案例来说明如何检测敏感的API。首先是选取热点，因为是对API进行检测，所以我们选取下列两个个函数作为热点：

1. public Method getMethod(String name, Class <?>…parameterTypes)\
2. public Method getDeclaredMethod(String name, Class<?>.. parameterTypes)

这四个函数都可以返回任意一个类中的方法，这是调用敏感 API 的必经之路，所以我们在这里设置热点。然后看下面的程序：

```
String string1 = “send”;
String string2 = “Mess”;
String string3 = “age”;
Strin string = string1 + string2 + string3;
Method method1= Class.forName(“com.android.mms.transacti
        on.SmsMessageSender”).getMethod(string,null);
Class cl = Class.forName(“com.android.mms.transaction.SmsMessageSender”);
Method method2 = cl.getDeclaredMethod(“sendMessage”,null);
```

在调用getMethod函数时，我们首先使用forName加载了SmsMessageSender类，但是并没有实例化这个类，之后使用getMethod函数返回sendMessage这个敏感的API，但是我们向getMethod函数传入的是一个组合的字符串。

之后的代码中，我们首先对SmsMessageSender类进行实例化，然后将，将字符串常量“sendMessage”传入getDeclareMethod这个函数中。分析结果如图4-1所示：

```
Analyzing...
Analysis time: 0.202
Checking call at line 12 in example.java...
A finite number of strings:
"sendMessage"
Checking call at line 14 in example.java...
A finite number of strings:
"sendMessage"
```

图4-1 对反射敏感API的分析结果

分析程序准确的识别到了热点所在的行数，并且对于传入热点变量的参数进行了分析，无论传入的是拼接的字符串，还是字符串常量，分析程序都对这些数据进行了准确的输出，达到了我们对于敏感API的检测的预期。

4.3 本章小结

本章通过JSA的原理，对利用Java反射机制调用敏感API的案例做出了，分析结果是合理的。由于时间关系，本次案例分析不够全面，尤其是在热点设置上面：Java反射中可以调用任意类的方法的函数总共有四种，但是我们只设置了两个函数，对于设置其他两个函数为热点，还存在一些问题，这将是今后工作的一个方向。

5 结束语

本文首先介绍了字符串分析分析。由于字符串在程序中的角色日益重要，所以字符串分析是掌握程序状态的一项重要技术。但是目前的字符串分析技术存在一些不足，如基于正则表达式的字符串分析不易扩展，表达能力差；基于带一元谓词的二阶逻辑的字符串分析效率较低。

其次介绍了 JSA，JSA 是 Aarhus 大学计算机科学系的 Christensen，Moller 以及 Michael 合作开发的字符串分析工具，Christensen 教授是最近提出字符串分析理论的，所以学习 JSA 的理论意义重大，并且 JSA 在本文中起到基石的作用。

之后本文利用 JSA 的技术，对一些程序进行静态分析，例如 SQL 注入程序，Java 反射程序。对于 SQL 注入程序，分析程序不仅可以准确的识别了包含 SQL 注入的语句，而且对于拼接的 SQL 语句，也能做出合理的反应。对于包含 SQL 注入的语句以及拼接而成的 SQL 语句，分析程序都会给出一个反例来警示。对于 Java 反射程序，我们设置了两个可以调用任意类的方法的函数作为热点，分析程序对这两个热点的参数进行了分析，并输出所有可能的值，这样我们可以查看被分析程序是否调用了敏感的 API。

当然，对于分析 SQL 注入程序和 Java 反射程序，都存在一些不足。例如，在分析 SQL 注入程序，对于拼接的 SQL 语句，一般是由用户输入。但是分析程序是静态分析，没有办法掌握程序真正运行时用户输入的情况，所以对于这种可能存在 SQL 注入的语句，只能给出反例来提醒，不能做到实时检测。

在 Java 反射程序中，我们设置的热点变量都是函数，然后分析传入热点变量的参数。但是当我们想要设置的热点变量的函数没有参数，我们无法进行分析。这时候需要我们去分析函数的返回值，由于时间关系，我们没有实现这个功能，这将是我们下一步的工作。

致　谢

光阴似箭、日月如梭，四年的本科学习很快就要过去了，在这里我要表达我最诚挚的感谢。

首先感谢我的指导教师刘晓建教授。本文的研究工作是在刘老师的悉心指导和严格要求下完成的。由于研究的课题可供参考的资料不多，刘老师经常和我对于某一个问题讨论一整天，他渊博的知识，严谨的治学态度，敬业的精神和平易近人的态度，都给我留下了深刻的印象，这些品质让我受用一生。在此，请允许我表示深深的敬意和衷心的感谢。

其次，我要感谢 Anders Moller 教授，他是 JSA 工具的开发者，在研究 JSA 的过程中，一度陷于僵局，是他的指点得以让此次毕业设计顺利完成。非常感谢！

感谢舍友和同学四年来的帮助，感谢唐富贵同学给我生活上的帮助和支持，感谢张琳师姐对我的鼓励，在他们身上我看到了对生活的热情和梦想的坚持。

衷心地感谢西安科技大学计算机科学与技术学院的各位老师给予我学业上的帮助和生活上的照顾。

我要把最真诚的感谢献给我的父母。感谢你们 22 年来对我的养育、感谢你们教会我对人真诚相待，对事认真负责。每当我遇到困难和挫折时，你们总是给我极大的信心和鼓励。感谢你们为我做的一切，正是这些激励我进步不断追求自己的梦想，顺利从大学毕业。再一次感谢我的父母！

最后，我要感谢所有在我论文完成过程中给予帮助和关心的老师和朋友。

参考文献

[1] Christensen A, Moller A, Schwartzbach M. Precise analysis of string expression. In: Custor R, ed. Proc. of the Static Analysis Symp. Heidelberg: Soringer-Verlag. 2003.1-18.

[2] Cytron R, Ferrante J, Rosen B, Wegman M, Zadek K. Efficiently computing static single assignment from and the control dependence graph. ACM Trans. On Programming Languages and Systems, 1991,13(4):451-490.[doi:10.1145/115372.115320].

[3] Mohri M, Nederhof M, Robustness in Language and Speech Technology. Dordrecht: Klywen Academic Publishers,2001.1-268.

[4] Yu F, Bultan T,Cova M, Ibarra O. Symbolic string verification: An automata-based approach. In: Havelund K, Majumdar R, Palsberg J,eds. Proc. of the 15th Intl Workshop on Model Checking Software. Heideberg:Springer-Verlag. 2008.306-324.[doi:10.1007/978-3-642-12002_13].

[5] Yu F, Alkhalaf M, Bultan T. Stranger: An automata-based string analysis tool for PHP. In: Esparza J, Majumdar R,eds. Proc. of the 16th Intl Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Heideberg: Springer-Verlag.2010.154-157.[doi:10.1007/978-3-642-12002-2_13]

[6] Minamide Y. Static approximation of dynamically generated Web pages. In: Ellis A, Hagino T,eds. Proc. of the 16th Intl Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Heideberg:Soringer-Verlag, 2010.154-157.[doi.10.1007/978-3-642-12002-2_13]

[7] Berstel J. Transductions and Context-Free Languages. Stuttgart: Teubner Studienbucher,1979.1278.

[8] Tateishi T, Pistoia M, Tripp O. Path- and index-sensitive string analysis based on monadic second-order logic. In: Dwyer M, Tip F, eds. Proc. of the Intl Symp. On Software Testing and Analysis. New York: ACM Press, 2011.166-176.[doi:10.1145/2001420.2001441].

[9] Hoperoft J, Motwani R, Ullman J. Introduction to Automata Theory, Languages, and Computation, Addison-Wesley,1990.

[10] Arni Einarsson andJanus Dam Niclson. A survivor's guide to Java program analysis with Soot,2008.

[11]Mehryar Mohri and Mark-Jan Nederhof. Robustness in Language and Speech Trchnology, chapter 9: Regular Approximation of Context-Free Grammars through Transformation. Kluwer Academic Publishers,2001.
[12]梅宏，王啸吟，张路。字符串分析研究进展[J]。软件学报, 2013,24(01):37-49.
[13]王啸吟.支持源代码回溯定位的字符串分析及其应用研究[D].北京大学, 2011.
[14]石云峰. 恶意代码检测方法及其在安全评估中的应用[D]. 中原工学院, 2012.
[15]刘文生, 乐德广, 刘伟. SQL 注入攻击与防御技术研究[J]。信息忘了安全，2015(09):129-134.
[16]陈小兵, 张汉煜, 骆力明, 黄河. SQL 注入攻击及其防范检测技术研究[J]. 计算机工程与应用, 2007(11):150-152.
[17]尹松强, 傅鹂. Java 反射机制研究[J]。软件导刊, 2008(11):85-87.
[18]王善发，吴道荣. Java 语言的反射机制[J]. 保山学院学报, 2011,30(05):32-36.