南京大学
NANJING UNIVERSITY

研 究 生 毕 业 论 文

（申请硕士专业学位）

论 文 题 目　基于机器学习的 Java 静态漏洞扫描系统的设计与实现

作 者 姓 名　徐 文 远

专 业 名 称　软 件 工 程

研 究 方 向　软 件 工 程

指 导 教 师　陈 振 宇　教 授

房 春 荣　助 理 研 究 员

2020 年 5 月 22 日

学　　号：　MF1832200

论文答辩日期：　2020 年 5 月 23 日

指 导 教 师：　 （签字）

The Design and Implementation of Static Code Analysis System based on Machine Learning for Java

By

Wenyuan Xu

Supervised by

Professor **Zhenyu Chen**

Research Assistant **Chunrong Fang**

A Thesis
Submitted to the Software Institute
and the Graduate School
of Nanjing University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Engineering

Software Institute

May 2020

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：基于机器学习的Java静态漏洞扫描系统的设计与实现

工程硕士（软件工程领域）专业 2018 级硕士生姓名：徐文远

指导教师（姓名、职称）：陈振宇 教授 房春荣 助理研究员

摘 要

随着软件系统应用领域不断扩大，对于软件和信息系统的攻击日益增多，人们对软件安全的要求不断提升。为此，在软件开发和测试过程中，工程师使用静态代码分析系统对软件代码进行安全性扫描，并评估分析报告，只有通过安全性评估的软件才可以部署至线上运行。然而，传统代码分析是保守的，为不遗漏安全风险，系统往往会抛出大量误报，这些误报增加了安全工程师工作量，进而对软件开发进度产生影响。随着机器学习领域的发展，学术界已将机器学习算法运用于代码分析以发现漏洞或降低误报，然而这些工作只适用于小规模程序，对于在大规模应用还存在着种种问题。

本系统旨在将学术研究成果应用于工业界实际项目中，面向 Web 开发常用的 Java 语言，利用污点分析、程序切片和 BLSTM 为开发或安全工程师提供更准确的代码扫描服务，进而减轻用户工作量，保证软件开发进度和质量。在污点分析方面，本系统利用 Find Security Bugs 的大量规则，保证低漏报的同时，对该工具的输出进行改进，使之反馈更详细的污点传播路径，增强报告可解释性；接着，本系统利用程序切片技术，对每一个漏洞实例进行代码切片，为了保证切片效率和稳定性，本文针对实际 Jar 包对切片器进行优化并提出分段切片思想，对于一个漏洞报告，将其对应的污点传播路径分解为小的污染流片段集合，再对每一个片段进行后向程序切片；最后，系统通过 BLSTM 模型，对预处理后的漏洞切片集合做预测，根据切片的预测结果推导漏洞实例本身是否为误报。

本系统目前已代替传统污点传播分析引擎在线上部署，同时实验结果表明，本系统可以在可接受的扫描时间下，取得更加准确的扫描结果。在效率上，本系统优化传统切片，每个项目的整体扫描时间不超过 1 小时，在准确性上，本系统误报预测精确率达 90.53 %，即相对于单纯的污点传播，本系统在遗漏少量真实漏洞的前提下，排除 25.44 % 误报，大大减轻安全运营人员的代码审计工作量，从而在整体上加快软件开发过程。

关键词：静态代码分析，污点分析，程序切片，BLSTM

南京大学研究生毕业论文英文摘要首页用纸

THESIS: The Design and Implementation of Static Code Analysis System based on Machine Learning for Java

SPECIALIZATION: Software Engineering

POSTGRADUATE: Wenyuan Xu

MENTOR: Professor Zhenyu Chen, Research Assistant Chunrong Fang

Abstract

As the application area of software systems continues to expand, attacks on software and information systems are increasing. So nowadays people pay more and more attention to software security. During software development and testing, engineers often scan software codes with static code analysis systems. Only software that has passed the security testing can be deployed online. While traditional static code analysis is often conservative. In order not to miss any security risks, they often give a large number of false positives. These false positives not only increase the workload of security engineers, but also delay the progress of software development. With the development of machine learning, researchers have applied machine learning on code analysis to discover vulnerabilities or reduce false positives. However, their works are only applicable to small-scale programs.

This system aims to apply academic research to the real world. For one of the most common languages in Web development, Java, this system utilizes taint analysis, program slicing, and BLSTM to provide more accurate code analysis services for development or security engineers. In terms of taint analysis, the system uses a large number of rules in Find Security Bugs to ensure low false negatives. In addition, it can give taint propagation paths to make the report more readable. Then, the system slices each vulnerability instance. In order to ensure slicing efficiency and stability, this system optimizes the slicer for the actual Jar package and proposes an idea called segmented slicing. For a taint vulnerability, its taint paths are divided into small set of taint flow fragments, then the system uses backward program slicing to slice each taint flow. Finally, the system uses the BLSTM model to predict vulnerability's slicing

set which has been pre-processed, and infer whether the vulnerability instance is false positive based on the slice prediction and taint flow logic.

This system has replaced the traditional taint analysis engine online. The experimental results show that the system can obtain more accurate scanning results within an acceptable scanning time. In terms of efficiency, the system optimizes traditional slicing to ensure that the scan time of each project does not exceed 1 hour. For accuracy, the system's precision rate reaches 90.53%. In other words, compared to Find Security Bugs, the system has eliminated 25.44% false positives, which greatly reduced the code audit work.

Keywords: SCA, Taint Analysis, Program Slicing，BLSTM

目录

表 目 录 ····· **x**

图 目 录 ····· **xii**

第一章 引言 ····· **1**

1.1 项目背景和意义 ····· 1

1.2 研究现状 ····· 2

1.2.1 传统静态代码分析系统应用现状 ····· 2

1.2.2 基于机器学习的静态安全扫描研究现状 ····· 3

1.3 本文主要研究工作 ····· 5

1.4 本文组织结构 ····· 6

第二章 相关技术综述 ····· **7**

2.1 漏洞挖掘技术 ····· 7

2.1.1 基于代码分析的漏洞挖掘技术 ····· 8

2.1.1.1 词法分析技术 ····· 8

2.1.1.2 数据流和控制流分析技术 ····· 8

2.1.1.3 形式化方法分析技术 ····· 9

2.1.1.4 符号执行技术 ····· 9

2.1.2 基于模糊测试的漏洞挖掘技术 ····· 10

2.2 污点分析 ····· 10

2.2.1 污点分析原理 ····· 11

2.2.1.1 污点分析三要素 ····· 11

2.2.1.2 污点分析过程 ····· 11

2.2.2 污点分析的优势和不足 ····· 13

2.2.3 Java 污点分析工具选型 ····· 14

2.3 程序切片技术 ····· 14

2.3.1 程序切片定义 ····· 14

2.3.2 程序切片技术 ····· 15
2.3.3 后向程序切片的优势与不足 ····· 16
2.4 BLSTM 算法 ····· 17
2.4.1 LSTM 原理介绍 ····· 17
2.4.2 双向读取——BLSTM ····· 18
2.4.3 BLSTM 的优势 ····· 19
2.5 Django 框架 ····· 19
2.5.1 Django 框架简介 ····· 19
2.5.2 Django 框架优势 ····· 20
2.6 本章小结 ····· 20

第三章 Java 静态安全扫描系统需求分析与设计 ····· 21
3.1 系统整体概述 ····· 21
3.2 系统需求分析 ····· 21
3.2.1 功能性需求 ····· 21
3.2.2 非功能性需求 ····· 24
3.2.3 系统用例描述 ····· 25
3.3 系统总体设计 ····· 29
3.4 污点分析模块设计 ····· 34
3.4.1 流程设计 ····· 34
3.4.2 污点传播图类图设计 ····· 34
3.4.3 污点传播树和漏洞报告类图设计 ····· 36
3.4.4 污点分析器类图设计 ····· 37
3.5 程序切片模块设计 ····· 39
3.5.1 流程设计 ····· 39
3.5.2 类图设计 ····· 40
3.6 数据预处理模块设计 ····· 42
3.6.1 流程设计 ····· 42
3.6.2 类图设计 ····· 43
3.7 误报预测模块设计 ····· 44
3.7.1 架构设计 ····· 44

3.7.2 类图设计 ····· 45
3.7.3 流程设计 ····· 47
3.8 数据库设计 ····· 49
3.9 本章小结 ····· 54

第四章 Java 静态安全扫描系统实现和测试 ····· 55
4.1 一个 XSS 漏洞实例 ····· 55
4.2 污点分析模块的实现 ····· 56
4.2.1 记录污点传播信息实现 ····· 56
4.2.2 构造污点传播图实现 ····· 57
4.2.3 构造污点传播树并生成注解的实现 ····· 58
4.3 程序切片模块的实现 ····· 61
4.3.1 切片控制模块实现 ····· 61
4.3.1.1 切片控制代码 ····· 61
4.3.1.2 过滤漏洞实例 ····· 61
4.3.1.3 分解污点传播树 ····· 63
4.3.2 后向切片的实现 ····· 64
4.3.2.1 基于 Joana 的后向切片 ····· 64
4.3.2.2 SDG 的生成配置类 ····· 65
4.4 数据处理模块的实现 ····· 70
4.4.1 泛化处理 ····· 70
4.4.2 建立单词表与向量化 ····· 71
4.5 误报预测模块的实现 ····· 72
4.5.1 误报预测控制 ····· 72
4.5.2 误报预测时序图 ····· 72
4.5.3 漏洞标记时序图 ····· 74
4.5.4 批训练过程 ····· 74
4.6 系统测试与运行展示 ····· 76
4.6.1 测试目标 ····· 76
4.6.2 功能测试 ····· 78
4.6.3 健壮性和性能测试 ····· 80

4.6.4 系统效果评估 ····· 81
4.6.4.1 参数设置 ····· 81
4.6.4.2 评估方法和度量 ····· 81
4.6.4.3 评估数据集 ····· 82
4.6.4.4 实验结果 ····· 82
4.6.5 系统运行展示 ····· 83
4.7 本章小结 ····· 86

第五章 总结与展望 ····· 87
5.1 总结 ····· 87
5.2 展望 ····· 88

参考文献 ····· 89

简历与科研成果 ····· 95

致谢 ····· 97

版权与原创性说明 ····· 99

表 目 录

3.1 污点分析模块功能性需求列表 ····· 22
3.2 程序切片模块功能性需求列表 ····· 23
3.3 预处理模块功能性需求列表 ····· 23
3.4 误报预测模块功能性需求列表 ····· 24
3.5 基于污点分析的静态扫描用例描述 ····· 26
3.6 保存污点分析结果用例描述 ····· 26
3.7 设置预测服务器用例描述 ····· 27
3.8 客户端令牌管理用例描述 ····· 27
3.9 扫描结果切片和预测用例描述 ····· 28
3.10 标记正报结果用例描述 ····· 28
3.11 标记误报结果用例描述 ····· 29
3.12 清空切片、标记和预测结果用例描述 ····· 29
3.13 模型训练用例描述 ····· 30
3.14 Client_Token 表 ····· 50
3.15 Project 表 ····· 50
3.16 Method_Description 表 ····· 51
3.17 Location 表 ····· 51
3.18 Taint_Flow 表 ····· 52
3.19 Label 表 ····· 52
3.20 Model_Config 表 ····· 53
3.21 BLSTM_Model 表 ····· 54

4.1 系统测试软硬件环境 ····· 77
4.2 系统测试用例套件 ····· 78
4.3 基于污点分析的静态扫描测试用例 ····· 78
4.4 程序切片测试用例 ····· 79
4.5 数据预处理模块测试用例 ····· 79

4.6 误报预测模块测试用例 ····· 80
4.7 健壮性和性能测试结果 ····· 80
4.8 漏洞扫描报告混淆矩阵 ····· 81
4.9 效果评估数据集 ····· 82
4.10 本系统与 Find Security Bugs 对比结果 ····· 82

图 目 录

2.1 过程内污点分析 ····· 12
2.2 过程间污点分析 ····· 12
2.3 污点传播的不足 ····· 13
2.4 Joana 和 WALA 程序切片示例 ····· 16
2.5 LSTM 的神经元 ····· 17
2.6 LSTM 的处理时序数据示意图 ····· 18
2.7 BLSTM 处理时序数据示意图 ····· 19

3.1 系统总体架构图 ····· 22
3.2 系统用例图 ····· 25
3.3 系统逻辑视图 ····· 30
3.4 系统开发视图 ····· 32
3.5 系统进程视图 ····· 33
3.6 系统物理视图 ····· 33
3.7 污点分析模块流程图 ····· 35
3.8 污点传播图设计 ····· 36
3.9 污点传播树和漏洞报告类图设计 ····· 38
3.10 污点分析器类图设计 ····· 38
3.11 程序切片模块流程图 ····· 40
3.12 程序切片模块类图 ····· 41
3.13 数据预处理模块类图 ····· 43
3.14 数据预处理模块类图 ····· 44
3.15 预测模块架构图 ····· 45
3.16 预测模块类图 ····· 46
3.17 预测模块流程 ····· 48
3.18 与鉴权相关表的实体关系图 ····· 49
3.19 与程序分析相关表的实体关系图 ····· 50

3.20 与模型相关表的实体关系图 ····· 53

4.1 一段含 XSS 漏洞的 Java 代码 ····· 55
4.2 构造污点传播图的代码片段 ····· 58
4.3 *doGet()* 函数的污点传播树 ····· 60
4.4 切片控制模块实现代码片段 ····· 62
4.5 漏洞过滤代码片段 ····· 62
4.6 污点传播类漏洞 ····· 63
4.7 污点传播树切片代码片段 ····· 63
4.8 基于 Joana 后向切片的代码片段 ····· 65
4.9 判断 SDGNode 是否需要移除的代码片段 ····· 66
4.10 生成 SDG 的配置类图 ····· 66
4.11 SliceWalaClassLoader 的代码实现 ····· 67
4.12 自定义 JavaClassLoader 的代码实现 ····· 68
4.13 NodeLimitPruner 的代码实现 ····· 69
4.14 限制调用图示意图 ····· 70
4.15 *unsafe()* 函数的代码切片 ····· 70
4.16 建立单词表和向量化的代码片段 ····· 72
4.17 误报预测控制逻辑的实现 ····· 73
4.18 误报预测时序图 ····· 73
4.19 漏洞标记时序图 ····· 75
4.20 DataLoader 的代码片段 ····· 76
4.21 神经网络后向传播的代码实现 ····· 76
4.22 批处理训练示意图 ····· 77
4.23 污点分析结果界面 ····· 83
4.24 设置预测服务器界面 ····· 84
4.25 预测结果展示 ····· 85
4.26 标记正报界面 ····· 85
4.27 标记误报界面 ····· 86

第一章　引言

1.1　项目背景和意义

随着网络技术和应用飞速发展，信息系统安全正面临着前所未有的挑战。网络化和互联互通性已经成为当前软件和信息系统发展的大势所趋，互联网导致系统攻击面增大，使系统面临的安全威胁空前增加。另一方面，随着软件和信息系统的业务不断丰富，复杂性也不断提高，这不仅使漏洞利用方式不断增加，而且使漏洞检测难度不断提高，尽管开发者已经投入了大量精力进行安全编程，但是软件漏洞仍然存在，并将继续成为一个重大问题 [1]。根据国际通用漏洞发布组织 MITRE 统计，在 1999 年，仅有 1600 个通用漏洞（CVE，Common Vulnerabilities and Exposures）被发现；到了 2014 年，新发现的 CVE 数量已接近 10000 个 [2]；而至今（2020 年 1 月 26 日），该数字已经上升到了 129695 个[1]。同时，根据国家互联网应急中心（CNCERT/CC，National Computer Network Emergency Response Technical Team/Coordination Center of China）的报告[2]显示，仅 2019 年上半年，国家信息安全漏洞共享平台（CNVD）就收录了通用型安全漏洞 5859 个，其中 Web 漏洞占比 24.9%，位居第二。

从以往发生的安全事件来看，针对 Web 漏洞的攻击可导致的后果极为严重，一个网站系统只要有一处脆弱点，攻击者将会使用一切手段对该网站发起攻击，篡改网页内容，窃取服务器数据甚至在服务器中执行恶意代码，造成更严重的后果。

源代码安全是保证软件安全的基础。对于源代码进行安全扫描，可以在源头处杜绝安全问题，根据 2019 年中国软件评测中心的《2019 大中型政企机构网络安全建设发展趋势研究报告》显示，进行源码扫描能有效减少 10%-50% 安全漏洞，降低软件安全建设和运维成本。因此，源码安全日益受到中大型政企机构的重视，在此方面投入也不断提高，其中，构建一款具有高准确率的静态代码分析（SCA，static code analysis）系统显得尤为重要 [3]。

由于软件数量和规模扩大，人们总希望能够通过自动化技术发现代码中的漏洞，从开源工具到商业工具，从工程实践到学术领域，目前已有许多关于代码静态扫描系统的应用和研究 [1, 3–15]。然而目前静态代码分析系统仍存在或多或少的问题，在工业界，SCA 使用基于模式匹配、数据流分析等技术，往往希望

[1] https://cve.mitre.org/cve/

[2] https://www.cert.org.cn/publish/main/upload/File/2019Firsthalfyear.pdf

不漏掉真实漏洞——不产生漏报，但是它们产生了大量误报 [3, 16, 17]。过高的误报使人工审核任务加重，让安全工程师将大量精力浪费在无害代码上 [5]。在实际生产中，安全工程师为了权衡误报与漏报，必须不断地设计更加精巧的扫描规则，再一次加重了安全工程师负担 [1, 18]，并对安全工程师个人经验有很高的要求。学术界对提高 SCA 的准确性进行了更深入的研究，引入了诸如符号执行，机器学习等技术，但是受限于计算能力，它们往往只能应对小规模代码项目，如何让学术界成果应用到工业界，使其适应大规模程序，是安全工程师亟待考虑的问题。

本文旨在设计一款基于污点传播和机器学习的 Java 静态代码扫描系统，通过污点传播、程序切片和双向长短记忆网络（BLSTM, Bidirectional Long Short-Term Memory）技术，提供准确的安全扫描服务。具体来说，用户输入一个已编译好的 Jar 包后，该系统以污点分析为基础，先产生一个初步的漏洞结果。由于污点分析存在过污染、无法识别清洁函数问题，该结果存在大量误报，接着系统根据初步结果进行分解，将其划分为多个子污染流，对每个污染流进行后向程序切片，并用 BLSTM 算法模型预测切片是否能够向下传播污点，从而推导每个漏洞实例是否可利用，并将疑似误报的漏洞威胁等级降低，以此让安全工程师或软件开发人员将更多注意力放在更可能为漏洞的结果上。同时，用户可以对漏洞进行标注，通过模型周期性的迭代学习，预测准确率将不断提高。系统对于污点分析得到的传播路径进行拆解，通过限制调用图和去依赖切片技术，解决了学术工具只能应对小规模程序的问题，填补了学术界和工业界的鸿沟，而基于学术界前沿工作，本系统相对于传统扫描工具结果更准确。

1.2　研究现状

本系统旨在利用传统方法配合机器学习对 Java 软件源代码进行静态安全扫描，因此本节将详细介绍工业界传统静态代码分析系统应用现状和学术界基于机器学习的静态安全扫描研究现状。

1.2.1　传统静态代码分析系统应用现状

在项目上线前对项目代码进行静态扫描已经成为软件开发中的最佳实践，在实际操作中，各个企业会根据实际情况使用第三方或自研静态扫描产品。本小节就目前常用扫描产品和静态扫描技术进行介绍。

目前静态安全扫描产品使用的技术主要有词法分析技术、数据流分析技术和形式化分析技术。词法分析技术主要是对代码中敏感函数进行匹配，从而发

现安全性漏洞，这一类分析技术产生时间较早，实现也较为简单，但是由于其不考虑程序上下文，存在大量误报，在实际情况中使用较少。

数据流分析技术模拟程序中敏感数据流动从而发现安全漏洞，其衍生出的污点分析方法是目前实际使用的主要检测方法，这一类分析方法适合于数组越界，注入类漏洞，因为这些漏洞特征表现为特定变量在某一程序点上状态满足或不满足某一规则，通过对数据的跟踪恰好可以识别这些特征。由于其能覆盖的漏洞种类较多，本系统选择以该技术为预测的基础，然而随着目前程序规模不断扩大，新型程序设计思想和语法不断出现，加之漏洞成因愈发复杂，即使是这类分析技术目前也存在大量误报，例如污点分析无法很好的处理容器变量、控制流、清洁函数等，企业中的安全运营人员必须通过手工设计精巧的污点传播规则，限制开发者所用的清洁函数，或是以手工添加函数摘要的方式添加清洁函数等方式降低误报，然而这些方式并不能完全解决污点分析的弊端，仍有误报不断产生，安全工程师需要花大量时间处理误报。

形式化分析技术往往用于一些可以用数学逻辑表达的安全漏洞，如 C/C++ 语言中指针变量二次释放，缓冲区溢出等问题，这一类技术将漏洞存在性问题变为数学逻辑上的约束求解和逻辑推导问题，由于目前程序规模较大，该技术面临着求解规模空间大，无法推导等问题，此外该技术针对的漏洞类型较少，将问题转变为数学模型需要大量人力成本。

目前商业 SCA 主要有 Fortify SCA [3]、LGTM [4]，Infer [5]等。Fortify 为老牌安全扫描工具，其主要结合词法分析，数据流分析、配置分析等扫描技术，但就像前文所说，其默认规则存在大量误报；LGTM 为较为新型的安全扫描器，是 Google、微软所使用的安全扫描器之一，其创新点在于其通过定义 QL 语法编写扫描器规则，将扫描逻辑和规则逻辑分离，使安全工程师能够较快地编写规则；Infer 为 Facebook 所使用的安全扫描器之一，也是少数使用形式化分析技术实现的扫描器，其使用分离逻辑和双向假说推理的思想，解决了传统形式化分析求解规模过大的问题，但其检测能力目前只限于资源泄露，空指针漏洞，不能用于其他类型（如常见的 OWASP TOP 10）的漏洞检测。

1.2.2　基于机器学习的静态安全扫描研究现状

为解决传统扫描器高误报、需要花大量人力定义规则等问题，随着机器学习领域的不断发展，学术界也一直在探索机器学习在静态安全扫描上的应用。

[3] https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

[4] https://lgtm.com/

[5] https://infer.liaohuqiu.net/

研究初期，研究者们提出了一些半智能化的扫描方案 [3, 19, 20]，这些方案主要由人工定义代码特征（如：语句长度，关键 API 调用位置等），并使用传统的机器学习算法进行分类。这一类工作的人工标记思路与本系统类似，然而由于其仍需要手工设计特征、泛化能力较弱等原因，并没有得到大面积推广。

随着研究工作不断深入，部分研究者提出通过相似度识别漏洞 [4, 21]，它的主要思想是如果一个待测代码片段 A 与已有代码片段 B 在某漏洞特征上具有高相似度，并且已知代码 B 为漏洞代码，那么代码 A 就很可能也为漏洞代码。在特征工程上，研究者们分别将代码特征抽象为词向量 [21]、树和图 [22]，以此比较相似性。李珍等学者提出 VulPecker [4] 从六个维度定义多个代码特征并且设计多种相似度算法，他们认为不同类型漏洞适用于不同特征和相似算法，并且一个优秀的相似度算法可以将含漏洞和安全代码进行区分，因此设计相似度算法选择器，通过已有漏洞代码库（NVD）和补丁信息训练相似度算法选择器，得到 < 漏洞类型, 算法 > 对应关系，当有新的被测代码输入时，通过各类漏洞的相似度算法识别代码是否有这些漏洞。这类工作的优势在于不需要大量标记数据集即可进行扫描，但是大多数工作仍需要手工设计特征，并且代码识别精度较低，因此并没有得到大规模使用。

还有部分研究者使用深度学习直接提取代码特征，将识别漏洞代码问题转化为分类问题 [1, 9, 14]。李珍等学者提出的 VulDeePecker [1] 是这一方向的代表工作之一，该工具首先抽取与安全 API 相关的程序代码（文中称其为 gadgets，它们类似于对调用安全 API 位置的后向切片），再将其代码转化为向量，结合标记后使用 BLSTM 学习，当输入待测代码后，将其做相同的向量化操作，再由 BLSTM 预测其是否为漏洞，经过试验，该工具在数据集上具有较高准确率，并且在 3 款实际软件中发现了 4 个未知漏洞，但是该工具只能预测 C/C++ 程序的漏洞，并且在切片时没有考虑控制流信息；Koc 等人提出使用机器学习降低污点传播误报率的方法 [14]，他们在文中总结了污点传播的误报模式，并且提出使用对方法体进行程序切片和 LSTM 预测降低误报率的想法，经过试验，他们认为该方法能够有效降低误报；在 2019 年，Koc 等人对先前工作展开进一步实证研究 [9]，对比手工设计特征、词袋结合随机森林、程序切片结合 BLSTM 和程序切片结合图神经网络（GNN）的预测效果，实验证明，BLSTM 组预测效果普遍优于其他组，并且，在切片后对切片进行泛化处理可以大大提高预测效果。Koc 等人工作是本系统的主要参考，对于函数体，本系统也是用类似的切片技术和预测技术，然而，他们工作中提到程序切片只适用于小规模程序的问题，因此他们工具只能对于漏洞的最后一个函数体（污点传播汇聚点）进行切片，由于污点是否能够传播并不等价于污点是否能够在汇聚点传播，因此无法用于实际使用，

本系统创新性的提了分解污点传播过程和限制调用图切片方法，解除这一限制，即本系统可以对污染传播全流程进行切片和预测，保证预测结果准确性。

1.3　本文主要研究工作

为解决传统扫描工具检测误报率高的问题，本系统实现一款基于污点分析、程序切片和 BLSTM 的静态代码扫描系统。该系统面向 Java 语言，以开发者编译后得到的 Jar 包为输入，提供准确的漏洞扫描报告，从而减轻安全工程师工作量，加快软件开发效率。本文主要研究工作如下。

本文首先分析目前常用程序安全分析技术的原理和优缺点，选择静态污点分析作为系统的基础分析方法，然后根据学术界前沿研究成果，选择基于 Joana 的后向程序切片和 BLSTM 作为误报预测方法，最后分析用户需求设计系统架构，将系统分为污点分析模块，程序切片模块，数据预处理模块和误报预测模块，以 C/S 架构向用户提供服务。

在污点分析模块，本系统以 Find Security Bugs 工具[6]的污点分析为基础，改进其分析流程，对用户提交的 Jar 包集合进行污点分析，输出污点分析报告，与原始 Find Security Bugs 不同，本系统在报告中不仅报告代码中潜在漏洞项、代码位置之外，还额外向用户展示可能被利用的污点传播路径，路径由污点传播树表示，一方面让用户更清晰地了解漏洞发生原理，快速判断误报，另一方面为程序切片提供依据。

在程序切片模块，系统基于 Joana 实现了一套适用于实际应用程序的切片器，完成无依赖切片、限制调用图规模切片和异构 Jar 包的切片。具体来说，对于上一模块报告中的污点传播树，系统对其相关函数拆分，得到污染流，对污染流进行限制规模切片，进一步缩小切片规模，为系统稳定运行和准确预测提供保障。

定义 1.1 (污点传播树). 污点传播树是反映污点传播的树形数据结构，树中根节点为调用入口函数的调用者函数，树中其他节点为函数调用信息或返回语句信息，由于同一层节点存在顺序，由孩子兄弟表示法表示。

定义 1.2 (污点传播流). 污点传播流用于指导切片的一种数据结构，实际为函数摘要至关键点行号的二元组。

[6] https://find-sec-bugs.github.io/，最初为 FindBugs [7]，如今被拆分成 Spotbugs 和 Find Security Bugs 插件，Spotbugs 提供 UI 和数据流基本分析框架，Find Security Bugs 提供污点分析框架和大量扫描规则，下文中提到 Find Security Bugs 实际指 Spotbugs 和 Find Security Bugs。

在预处理模块，系统参考 Koc 等学者工作 [9] 和实际应用的切片内容，提出一套适用于实际应用代码的泛化和向量化方案，保证预测模型快速收敛和用户对准确性的要求。

在误报预测模块，系统基于 Pytorch 实现经典的 BLSTM 神经网络三层架构，通过该模型预测污染流是否能够传播污点，若一棵传播树中存在一处传播流无法传播污点，那么该传播树表示的传播路径被标记为不可利用，当一个漏洞实例的所有传播树不可利用时，系统推导得到该漏洞实例为误报，经过试验，该模型可以准确预测误报。同时系统基于 Django、Celery 和 MySQL 等技术实现后台服务，用户可以根据需要定制模型各项参数并定时触发模型训练流程。

综上，本系统将学术界前沿技术成功运用到实际生产中，首次提出对污点分析结果拆分，将漏洞预测任务转化为污点传播树中子污染流是否安全的预测任务；改进后向切片器，通过限制调用图的方式保证切片过程在绝大多数污染流上能够顺利进行，从而保证系统在各类规模程序上都能提供准确的漏洞扫描报告。

1.4　本文组织结构

本文共分为五个章节，组织结构如下：

第一章，引言部分，介绍本系统背景和意义，分析当前关于静态安全扫描的研究现状以及介绍本文主要研究工作。

第二章，介绍相关技术，介绍目前较常见的漏洞挖掘技术，对比相关技术优缺点，说明本系统的技术方案，并对系统中使用的核心技术如污点分析，程序切片和 BLSTM 算法进行介绍。

第三章，Java 静态安全扫描系统的需求分析和设计，首先介绍系统整体概述，分析系统功能性和非功能性的需求，接着说明系统总体设计，再分模块从类图和流程两方面分别说明模块设计。

第四章，Java 静态安全扫描系统实现和测试，对于各个模块的实现细节结合漏洞实例代码进行详细说明，接着介绍系统的功能测试、健壮性和性能测试，对系统预测效果进行评估，最后展现系统运行效果。

第五章，总结和展望，对本系统和论文的工作进行总结，并提出本系统的不足和未来工作方向。

第二章　相关技术综述

本章主要介绍时下常用的漏洞挖掘技术，以及本系统将要用到的技术、工具和开发框架。

2.1　漏洞挖掘技术

漏洞挖掘指用自动化或半自动化技术对软件进行本身进行静态、动态分析，检测其是否存在安全漏洞的过程 [2]。随着软件规模扩大，软件功能种类多样化，安全漏洞其种类也在不断增多，不同漏洞产生原因不同，利用方式也不同，一种漏洞挖掘技术不能适用于所有漏洞，因此在实践中，人们会在软件开发各个阶段应用不同类型的技术，本文将这些技术的应用场景主要分为三类：

- 白盒测试：也称为静态测试，通常发生在软件编码阶段，对应用程序源代码或对源码编译产生的二进制文件进行安全性审计，从而发现漏洞。该场景下的技术由于获取了源代码信息，也被称为源码扫描或静态扫描，其可以获得较高的代码覆盖率，发现更多漏洞，但由于无法运行而产生大量误报，它也是本文系统的应用场景。

- 黑盒测试：通常发生在软件测试和运行阶段，对应用程序动态运行，并输入数据，分析程序反馈从而发现漏洞。适用于该场景的技术可以在程序只暴露接口的情况下展开测试，因此应用广泛，同时动态运行程序使其具有更低的误报，甚至产生漏洞利用报告，但由于如今应用程序复杂，自动化测试技术往往无法覆盖所有代码逻辑。

- 灰盒测试：介于白盒和黑盒之间的一种测试，这一场景下的技术往往通过软件插桩或是逆向工程，不但关注程序输入输出信息，还可以了解部分程序内部逻辑。因此具备上述两种测试的优点，许多黑白盒测试技术稍加改造也可以变为灰盒测试技术，但由于其终究还是需要获取程序本身，以及需要动态运行，使用比纯黑白盒测试更为复杂。

在漏洞挖掘技术发展早期，每一种技术往往只能应用于一种特定场景，但随着研究者不断完善和改进，如今一种技术也可以适用于多种场景，并且技术本身也产生了相互组合，对其正交分类较为困难，本章就目前常用的漏洞挖掘技术进行粗略分类并分别进行简要介绍 [2, 23]。

2.1.1　基于代码分析的漏洞挖掘技术

这一类漏洞挖掘技术侧重于对程序代码本身进行分析，同时对漏洞产生原理进行建模，将程序分析结果结合漏洞模型发掘漏洞，主要用于白盒测试场景。主要有词法分析技术、数据流分析技术、形式化分析技术和符号执行技术。

2.1.1.1　词法分析技术

词法分析技术是最简单的一类漏洞挖掘技术，其主要思想是将代码文本与归纳好的缺陷模式进行匹配，以此发现漏洞。由于其不深入分析程序结构和语义，往往只能挖掘较为简单的一类漏洞，并且存在相当高的误报率，在实际场景下应用较少，但由于其思想简单，适用性很广，目前也还存在类似工具，如：MobSF [1]，Cobra [2]。

2.1.1.2　数据流和控制流分析技术

数据流分析是一种按程序执行路径模拟数据流动的一种分析技术，其原本用于进行程序优化 [24]，安全研究者们发现后将其运用于漏洞挖掘中，如今该技术在白盒，灰盒和黑盒测试都有应用 [25]。

在数据流分析过程中，存在过程内分析和过程间分析，过程内分析主要对函数内分析，而过程间分析主要处理跨函数分析。对于过程内分析，根据其对程序路径的分析精度，可分为流不敏感分析，流敏感分析和路径敏感分析。流不敏感的数据流分析只是按代码行号从上而下进行分析；流敏感分析会首先产生程序控制流图（CFG, Control Flow Graph），再按照 CFG 的拓扑排序正向或逆向分析；路径敏感信息不仅考虑到语句先后顺序，还会考虑语句可达性，即会沿实际可执行到路径进行分析。过程间分析首先构造程序的调用图（CG, Call Graph），接着遍历图中函数进行过程内分析，当遇到其他函数时，若已分析过，则直接使用分析结果向下分析，若未分析过，则跟进该函数，再次进行过程内分析，并且将分析结果保存。

数据流分析能够一定程度上理解程序语义，是一种比词法分析技术更为精确的一类分析技术，其关键在于准确计算程序的数据流，此外，污点分析技术作为数据流分析的一种特例，是本系统所使用的主要技术之一，将在下文单独一章进行介绍。

[1] https://github.com/MobSF/Mobile-Security-Framework-MobSF

[2] https://github.com/WhaleShark-Team/cobra

2.1.1.3　形式化方法分析技术

形式化方法分析主要思想是将软件代码性质进行形式化描述，再判断该描述是否满足漏洞特征的一类分析方法 [26]，其中定理证明技术是形式化代码分析技术的主要代表。

定理证明技术将漏洞存在（或不存在）定义为一定理，再将源程序代码特征转化为数学表达形式，最后对数学表达进行逻辑推理，若定理存在性得以证明，则漏洞存在（或不存在），即漏洞挖掘过程类似于数学上的定理证明过程。主要代表性工具有 infer [3] [27]、ESC/Java [13] 和 saturn [11]。

该技术作为一种使用严格数理逻辑推理作为检测手段的技术，误报率较低，但由于其需要针对特定漏洞构建数学条件，需要大量人工参与，有的漏洞甚至难以用数学结构表达，目前只适用于死循环、资源泄露和空指针等问题，对新漏洞的扩展性不高，同时，如何将大规模程序应用于形式化方法分析也成为工业界亟待解决的问题。

2.1.1.4　符号执行技术

符号执行技术是一种将程序执行可达性问题转化为约束求解问题，并以此进行漏洞挖掘的技术 [28]，代表性工具有 angr [4]，DART [29], CUTE [10], EXE [6] 和 KLEE [12]。

具体来说，符号执行包含一个符号状态表 σ 和一个符号路径约束 PC，开始时，$\sigma = \varnothing, PC = true$，每读取一条语句，就将变量抽象为约束求解中的变量、常量或他们的表达式放入 σ 中，特别的，当遇到条件判断 $if(e)$ 时，将 if 分支的 PC 更新为 $PC \wedge \sigma(e)$，将 else 分支的 PC' 更新为 $PC \wedge \neg\sigma(e)$，随后使用约束求解器求解 PC 和 PC'，如果约束不满足，则停止对该分支的解析（因为该分支不可达）。当符号执行遇到程序崩溃、预先定义的漏洞语句、或是程序正常退出时，整个分析停止，同时可以计算可以到达停止点的输入。

符号执行可以分析程序中的控制流、覆盖更多代码，同时也有效降低了误报率，但传统符号执行严重依赖于约束求解器的能力，例如，若约束求解器不能处理非线性计算，或是整个程序中存在无法分析的第三方库，那么整个分析将无法继续。为解决这些问题，研究者们提出了动态符号执行的想法 [6, 10, 12, 29]，但其在实际应用中仍不是很广泛，主要原因在于其需要大量计算资源，甚至在处理大规模程序时，出现的路径爆炸问题会导致约束求解无法产生结果。

[3] https://fbinfer.com/

[4] http://angr.io/

2.1.2　基于模糊测试的漏洞挖掘技术

模糊测试是一种通过构造大量非预期输入，同时观察软件运行反馈来发现软件漏洞的方法 [30]。

由于其不需要了解程序内部具体实现，不论是 Web 应用还是二进制程序，其都是一种非常受欢迎的技术。该技术的关键在于如何构造能够引发软件漏洞的输入，对于 Web 应用来说，扫描器会针对每个漏洞（如 SQL 注入，XSS 等）准备若干个（或若干组）可能会引发漏洞的输入模式，接着爬虫程序会爬取网站所有 URL（或是将 URL 也作为模糊输入的一部分），将输入模式整合进 HTTP 报文中并发送给服务器，若服务器返回符合漏洞特征（也被称为测试断言，Test Oracle），则报告程序存在漏洞，主要工具有 AWVS [5]，Netsparker [6]和 ZAP [7]。

学术界更热衷于对二进制程序的模糊测试技术进行改进 [31]，为了构造能够到达更深层代码的输入，研究者们提出了基于变异的模糊测试和基于生成的模糊测试 [32]，基于变异的模糊测试通过当前模糊测试结果反馈和结合程序特征，对输入进行各类变异，以此指导测试方向，该方向主要有：代码覆盖率制导的模糊测试技术，如 AFL [8]和 libFuzzer [9]；由符号执行制导的模糊测试技术，如 Driller [33] 和由信息流制导的模糊测试技术，如 VUzzer [34]。基于生成的模糊测试技术适用于输入具有一定模式的场景，例如 PDF 阅读器，程序编译器或解析器等，例如 CodeAlchemist [35] 即设计了一套 JavaScript 代码生成工具，以此发现 JavaScript 解释器的漏洞。

模糊测试作为一种能够得到漏洞利用输入的漏洞挖掘技术，在黑盒和灰盒场景下应用广泛，但由于目前程序日益复杂，该漏洞挖掘技术很难测试隐藏在复杂状态已经条件分支下的代码块，导致程序覆盖率不高，即其具有低误报，高漏报的特点。

2.2　污点分析

污点分析属于数据流分析的变种，通过判断关键操作的数据（如调用危险函数的参数）是否可被用户操控，推测程序是否存在安全性漏洞 [36]。由于其了解程序上下文，并且有较强的可解释性——安全工程师可以通过跟踪污点传播过程判断是否存在安全问题，因此其也成为了挖掘 Web 或 Android 漏洞较为常

[5] https://www.acunetix.com/vulnerability-scanner/

[6] https://www.netsparker.com/

[7] https://www.zaproxy.org/

[8] http://lcamtuf.coredump.cx/afl/

[9] https://llvm.org/docs/LibFuzzer.html

用的技术，也被很多开源或商用白盒扫描器使用，如：Pixy [15]、Find Security Bugs，Fortify [10]和 LGTM [11]。

2.2.1　污点分析原理

2.2.1.1　污点分析三要素

污点分析主要有三个组成要素：污点信息产生点（source）、污点信息汇聚点（sink）和污点信息清洁点（sanitizer），它们通常需要富有经验的安全工程师手动设置。

- 产生点（source）：污点产生点往往是用户输入的数据，比如 Web 应用中读取 URL 参数的函数，顾名思义，这些函数调用后的返回值被标记为污点——攻击者可以操控的数据点。
- 汇聚点（sink）：检查点是程序的一些敏感操作，如调用数据库查询语句，或是将数据返回到网页，如果这些操作的数据是污点，那么意味着操作可被攻击者利用，即程序存在漏洞。
- 清洁点（sanitizer）：清洁点通常是对污点进行消除的一类操作，如 SQL 注入、XSS 中的过滤函数。清洁点是污点传播准确性的重要保证，不能识别清洁点即会引发污点过污染问题。

2.2.1.2　污点分析过程

污点分析分为静态污点分析和动态污点分析，两者区别在于静态污点分析只使用程序代码模拟污点传播过程，而动态污点传播则通过程序的实际运行进行传播，由于本文关注于白盒测试情景，故只介绍静态污点传播方法，而在下一子章节会介绍动态传播的优劣势。

在定义好三要素之后，污点分析法会与数据流分析一样，对程序进行过程内分析和过程间分析。

过程内分析包括了显式流分析和隐式流分析，显示流分析即通过分析变量的数据依赖关系进行污点传播，而隐式分析则是指考虑控制依赖进行污点传播。

如图 2.1所示，首先假设变量 a 为污点变量，实线箭头表示了显示污点传播路径，而虚线箭头表示了隐式污点传播路径，同时该图也说明了过程内污点传

[10] https://www.microfocus.com/en-us/solutions/application-security

[11] https://lgtm.com/

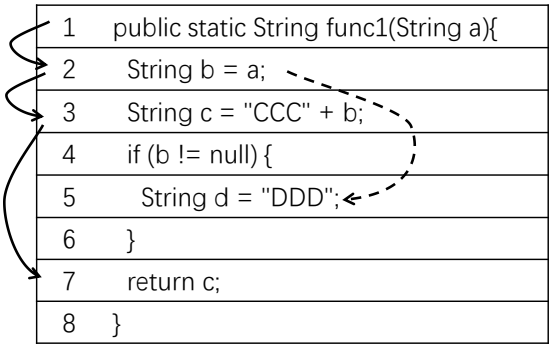

图 2.1: 过程内污点分析

播基本思想，即从上至下遍历数据流图，若未标记的变量依赖于污点变量，则新变量也被标记为污点变量。虽然攻击者确实可以利用控制依赖操作数据进行攻击，但由于其分析复杂且会产生大量误报，在工程领域常常只做数据流依赖的显示分析，因此本文主要讨论显式流分析。

现代程序存在着复杂的函数调用，除了进行过程内分析，还需要进行过程间分析。其分析首先构造函数调用图（Call Graph），接着搜索存在产生点的函数，对于每一个存在产生点的函数，自顶向下分析（也可以自底向上分析）。遇到函数调用时，跟进被调函数，进行过程内污点分析，将分析结果表达为 $\langle f, S, r\rangle$ 的函数摘要，其中 f 包含函数本身摘要信息（类名方法名和函数签名），S 指调用过该函数后被污染的变量集合，r 取值 0 或 1，标记函数返回值是否被污染；接着根据函数摘要，再进行过程内分析，如此往复直至分析完函数所有代码块或是污点传播至汇聚点，报告漏洞。

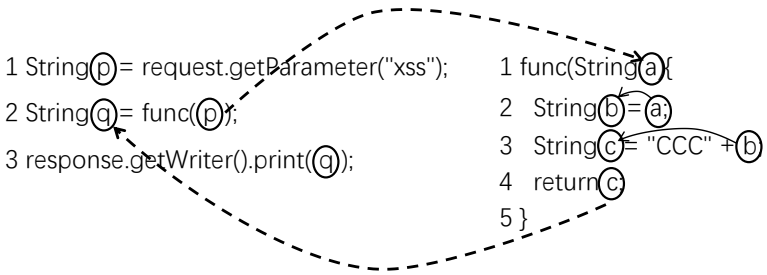

图 2.2: 过程间污点分析

如图 2.2所示，分析过程从左侧函数开始，因为其找到了一处产生点——*request.getParameter("xss")*，于是将污点传递到变量 *p*，接着调用函数 *func(p)*，于是对函数 *func()* 做过程内分析，得到其函数摘要，$\langle func, \{a, b, c\}, 1\rangle$，于是回到调用者的函数内，变量 *q* 被标记为污点，又因为第三行存在一处汇聚点——

response.getpriter.print()，并且参数为污点，于是报告此处有漏洞，并且根据汇聚点可以判断该漏洞是一个 XSS 漏洞。

2.2.2　污点分析的优势和不足

污点分析能够对程序上下文有一定理解，往往能产生误报率相对较低以及可解释的漏洞报告，其方法对 Web 类型的安全漏洞覆盖率较高，而污点类型的漏洞普遍具有较高危害性 [3, 18]，因此该方法已被很多工业界、学术界的安全静态扫描工具所使用 [8, 15, 18]，本文也选择该技术产生初步漏洞扫描结果。

然而，污点传播仍有可能发生误报，以下通过简单示例来说明。

```
1 String p = request.getParamter("p");
2 Map<String, String> map =
3     new HashMap<>();
4 map.put("p",p);
5 map.put("b","clean");
6 String q=map.get("b");
7 response.getWriter().print(q);
```

(a) 污点传播难以处理容器类型

```
1 String sqlFilter(String s){
2     String[] illegals = new String[]{
3         "\"", "\\", /*...*/, "(", ")"};
4     for(String e: illegals)
5         s=s.replace(e,"");
6     return s;
7 }
```

(b) 污点传播难以分析清洁函数

```
1 String p = request.getParameter("p");
2 URL u = new URL(p);
3 Boolean b = isInnerIP(u);
4 if (b) return;
5 u.openConnection();
```

(c) 污点传播难以分析控制流

```
1 String p = request.getParameter("p");
2 String q =
3     "http://www.baidu.com/s/"+p;
4 URL u = new URL(q);
5 u.openConnection();
```

(d) 污点传播难以处理特殊污染条件

图 2.3: 污点传播的不足

首先，污点传播对容器类型无法做很好处理，如图 2.3(a)所示，当污点传入容器类型时（在此例子中为 *map*），静态污点传播只能将这类变量的传播规则设为传播/不传播污点，从而造成过污染/欠污染，就如图所示，若设为 *map* 传播污点，由于案例实际从容器中取出的是没有污点的变量，即过污染，而若设为不传播，若 q 取出参数 p，那么又导致欠污染。动态污点传播虽然解决了这一问题，但是由于其使用条件复杂，且无法用于静态分析，本文暂不讨论。

此外，不论是动态污点传播还是静态污点传播，其对污点清洁点的识别能力几乎为零，如图 2.3(b)所示，该函数是一个典型的防御 SQL 注入的污点清洁函数，即在第 2∼5 行对待拼接 SQL 字符串存在的特殊字符进行替换和过滤，但是污点传播并不能识别这些清洁函数，导致误报。

再者，静态污点传播对控制流没法做很好的处理，如图 2.3(c)所示，在第三行，程序已经对可能产生的 SSRF 漏洞进行了处理，即如果是内部地址则直接返回，但是不论是考虑显式流还是隐式流，污点传播都不能避免这一类误报。

最后，对于特殊触发条件的漏洞，污点传播无法很好处理，如图 2.3(d)所示，在第二行，因为 SSRF 要求攻击者能够操控主机名，所以即使用户输入的污点变量拼接在一个正常主机名之后，程序也不会出现 SSRF ，而按照污点传播分析法，毫无疑问它会报告这段程序存在 SSRF 漏洞。

正是因为存在这些不足，本文将在下文引入程序切片技术和 BLSTM 来降低误报率。

2.2.3 Java 污点分析工具选型

Java 上的污点分析工具有很多，如 Find Security Bugs，TAJ [8]。但对于安全扫描工具来说，除了污点传播引擎，针对漏洞定义好大量的入口点和汇聚点是扫描低漏报的重要保证。Find Security Bugs 作为较为经典的扫描工具，前人已经为预先定义了大量的污点传播规则，并且支持自定义新的入口点和汇聚点，非常适合二次开发。

本系统保证准确性的手段在于利用机器学习对污点传播的误报进行改进，而不是定义大量污点传播规则，因此本文使用 Find Security Bugs 进行污点传播分析。此外，本系统解决了原版 Find Security Bugs 不能展示传播路径的问题，使其更加易用。

2.3 程序切片技术

2.3.1 程序切片定义

程序切片技术是一种通过对程序分析，抽取程序中与关注点相关的一组语句集合的技术，目前广泛运用于程序调试，程序测试，优化，安全分析等领域。

该技术首次由 Mark Weiser 在其博士论文中提出 [37]，他将程序切片做出如下定义：

将程序抽象为图 $G\langle N, E\rangle$，N 为程序中的语句集合，E 为 $\langle n, m\rangle$ 的集合，其中 n 为数据流的上一条语句，m 为数据流的下一条语句。

定义 2.1 (程序状态序列（state trajectory）). *若程序 P 中有长度为 k 的程序状态序列 T，则：*

$$T = \langle \langle n_1, s_1 \rangle, \langle n_2, s_2 \rangle, \cdots, \langle n_k, s_k \rangle \rangle$$

其中 $n_i \in N$，s_i 为一个单射函数，记录所有变量到具体值的映射。

定义 2.2 (切片准则（Slicing citerion）). *若对于程序 P 有切片准则 C，则 $C = \langle i, V \rangle$，其中 i 指关注点，通常是指一条程序语句，V 表示程序 P 中的变量子集（通常为 i 上的变量集合）。*

该切片准则决定了一个投影函数 $Proj_C$：

$$\mathrm{Proj}_{\langle i,V\rangle}(T)=\langle \mathrm{Proj}'_{\langle i,V\rangle}(t_1),\cdots,\mathrm{Proj}'_{\langle i,V\rangle}(t_n)\rangle$$

其中：

$$\mathrm{Proj}'_{\langle i,V\rangle}(\langle n,s\rangle)=\begin{cases}\lambda & \text{if } n\neq i\\ \langle n,s|V\rangle & \text{if } n=i\end{cases}$$

其中 λ 指空字符串，$s|V$ 指 V 中变量的单射函数。上式的含义是指，切片准则让我们只考虑 V 中变量的状态序列，并且只有当语句是关注点时，函数才返回状态序列，否则返回空字符。

定义 2.3 (程序切片). *切片 S 为一组源程序语句的子集，它由不停地删除零条或多条原程序语句得到，同时保证 $\mathrm{Proj}_C(T)=\mathrm{Proj}_{C'}(T')$ 其中 T' 指切片中的状态序列，而 $C'=\langle succ(i),V\rangle$，$succ(i)$ 指最靠近关注点 i 的语句（如果 V 不是 i 上的变量集合）或者就是 i 本身（如果 V 是 i 上的变量集合）。*

按照定义，可以知道 Mark Weiser 实际上定义的是后向切片（backward slices），即切出对关注点造成影响的所有语句和谓词集合。对其改进后也出现了前向切片（forward slices），即切出被关注点影响的其他语句和谓词集合。

2.3.2　程序切片技术

程序切片技术自诞生以来，也经历了多个阶段 [38]，起初为 Mark Weiser 的概念阶段，主要通过控制流来进行程序切片；随后发展为基于依赖图的切片阶段，Ottenstein 等人 [39] 首先提出运用程序依赖图（PDG，Program Dependence Graph）进行切片，主要思想是首先对程序建立程序依赖图，再通过对程序依赖图的可达性计算进行后向切片，程序依赖图是一种反应语句数据依赖和控制依赖的一种程序中间表达形式，图中的点为单条程序语句，而图中的边则表示语句之间存在数据依赖关系，例如图 2.1的代码中，第三行与第二行就存在数据依赖关系，第七行与第四行则存在控制依赖关系。PDG 只适用于单个函数的切片，horwitz 等人 [40] 提出系统依赖图（SDG, System Dependence Graph）的概念，在 PDG 基础上，额外增加两种边，一种边表示调用函数和被调用的直接依赖关系，另一种边描述由于边的依赖而引发的间接依赖关系，同时，前向切片也在此阶段产生；第三阶段为面向对象程序切片阶段，研究者提出将接受对象作为一个隐藏参数，结合多态解析手段，将面向对象切片问题转换为一般切片问题，使程序切片可以运用于面向对象语言。

目前已有很多工具可以对 java 程序进行切片，例如 WALA（T.J. Watson Libraries for Analysis）[12]和 Joana [13]。WALA 原本由 IBM T.J. Watson 研究中心作为 DOMO 项目的一部分进行开发，在 2006 年，IBM 将该软件转赠给社区。其主要功能包括 Java 类型系统和类层次分析，过程间数据流分析、指针分析和调用图构造以及上下文相关的切片等。Joana 是由卡尔斯鲁厄理工学院推出，基于 WALA 一些分析实现构造的另一款 Java 切片工具，其包含指针分析，异常分析和 PDG 构图等复杂功能，并且其切片结果中能够显示函数调用时实参的值，而对于污点传播类型中的自定义过滤函数，实参值是很重要的特征（见图 2.4）之一，因此本文选择 Joana 作为程序的切片器。

```
1  String arg=args[0];
2  arg=arg.replace("<","&lt")
3  arg=arg.replace(">","&gt");
4  System.out.println(arg);
```

(a) 目标程序

```
Ljava/lang/Object :: v4 = p1 $args [#(0)]::
Ljava/lang/String :: v8 = v4.replace(#(<), #(&lt))::
Ljava/lang/String :: v12 = v8.replace(#(>), #(&gt))::
Ljava/io/PrintStream :: v13 = java.lang.System.out::
V :: v13.println(v12)::
```

(b) Joana 程序切片结果

```
arrayload 1[3]
inst:8 = invokevirtual < String, replace(CharSequence; CharSequence;) String;
inst:12 = invokevirtual < String, replace(CharSequence; CharSequence;) String;
inst:13 = getstatic < System, out, <PrintStream> >
inst:invokevirtual < Application, PrintStream, println(String;)V >
```

(c) WALA 程序切片结果

图 2.4: Joana 和 WALA 程序切片示例

2.3.3　后向程序切片的优势与不足

程序切片能够得到影响关注点的程序语句，在有效缩短程序上下文、降低被分析代码量的同时，还能有效保留控制流信息，在程序调试，软件度量或是软件安全中已经得到了很广泛的应用。

由于污点传播已经对程序由上而下进行了分析，通过对汇聚点和污点值进行后向程序切片，可以从反方向分析漏洞代码，暴露先前没有考虑的控制语句，同时通过进一步特征处理，可以作为神经网络输入，弥补污点传播对控制流分析不足的缺陷，因此本文采用该技术作为特征处理的基础工作，另外由于程序

[12] http://wala.sourceforge.net/wiki/index.php/Main_Page

[13] https://pp.ipd.kit.edu/projects/joana/

切片属于较为传统的技术，在 Java 语言上已有较为成熟的工具，考虑到 Java 漏洞的特征与函数参数强相关这一特性，本文选择 Joana 进行程序切片。

然而，传统后向程序切片也存在不足，目前最大的挑战在于当程序规模变大时，后向程序切片效率低下；此外原生 Joana 并不支持在缺失程序依赖下对其程序进行切片，而在 SCA 的输入中，缺失依赖的程序又是非常常见的，针对以上问题本文做出分段切片、限制调用图切片和无依赖切片的技术调整，具体细节将在第三四章节详细说明。

2.4　BLSTM 算法

双向长短期记忆网络（BLSTM，Bidirectional Long Short-Term Memory）是一种特殊的循环神经网络（RNN，Recurrent neural network），主要改进有双向读取和增加记忆门两方面，是一种在自然语言处理上应用广泛的神经网络 [41]。

2.4.1　LSTM 原理介绍

为了保证在不同的序列数据上，相同数据点有不同输出结果，人们将其记忆体加入神经网络之中，从而产生了 RNN，然而 RNN 在处理较长文本时，存在着梯度爆炸或梯度消失的问题 [42]，Hochreiter 和 Schmidhuber 于 1997 年在记忆体中加入写入、输出和遗忘门，而改进后的神经网络也被称为长短期记忆网络（LSTM，Long Short-Term Memory） [43]。

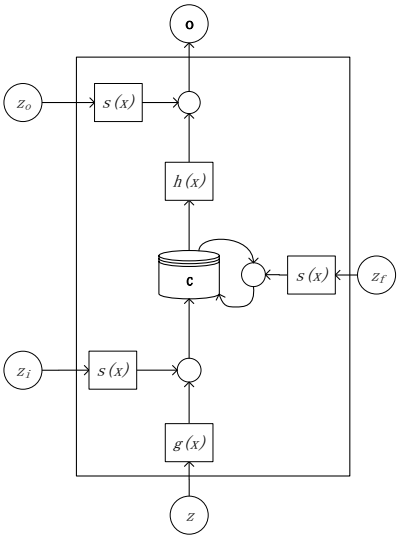

图 2.5: LSTM 的神经元

如图 2.5所示，一个 LSTM 的神经元有四个输入：z 表示原输入，z_i 表示输入门，z_f 表示遗忘门，z_o 表示输出门，以及有一个输出 o，神经元内有一记忆体 C，在下一时刻输入进入神经元时，记忆体下一时刻的值 $C' = g(z) * s(z_i) + C * f(z_f)$，$g(z)$ 为输入值激活函数，$s(z)$ 为门限激活函数，通常取 sigmoid 函数，从公式可以看出，首先输入门控制输入数据本身是否可以输入进神经元，接着遗忘门控制记忆体中旧数据是否会保留，最后输出 $a = tanh(c') * f(z_o)$，显然，z_o 控制记忆体是否向外输出。

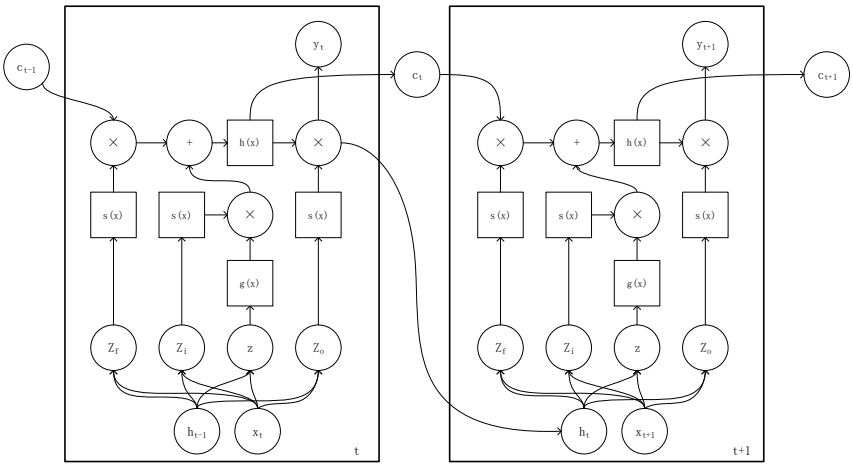

图 2.6: LSTM 的处理时序数据示意图

利用 LSTM 处理时序数据情况如图 2.6所示，对于任意时刻 t，有 h_{t-1} 和 c_{t-1}（t_0 时为随机值），h_{t-1} 和 x_t 通过线性变换得到 z_f、z_i、z 和 z_o，例如，$z_f = W_f \times [h, x] + b_f$。$z_f$、$z_i$、$z$ 和 z_o 均为一维向量，其长度等于该层 LSTM 神经元个数，按上文所述，这 4 组向量输入神经元，结合神经元内记忆体 c_{t-1} 计算得到输出 y_t，同时 y_t 也作为下一轮输入的一部分，即 h_t。h_t 和 $x_t + 1$ 作为下一时刻输入，输入下一时刻神经元。在自然语言处理领域，人们通常利用最后一个时刻的输出对自然语言分类。

2.4.2　双向读取——BLSTM

早在 RNN 时期，就有研究者提出双向 RNN 提升拟合或预测准确性的思想，而 LSTM 作为 RNN 的一种，自然也可以运用该思想对其进行优化。

如图 2.7所示，对于一组时序数据 $X = \{x_0, \ldots, x_t, x_{t+1}, \ldots, x_k\}$，训练两个 LSTM 网络，第一个网络正向读取数据（如图上方所示），第二个网络逆向读取数据（如图下方所示），在同一时刻，结合两个方向的网络输出值作为该时刻的输出（如果用于分类问题，通常结合正向 t_k 的输出和逆向 t_0 的输出）。

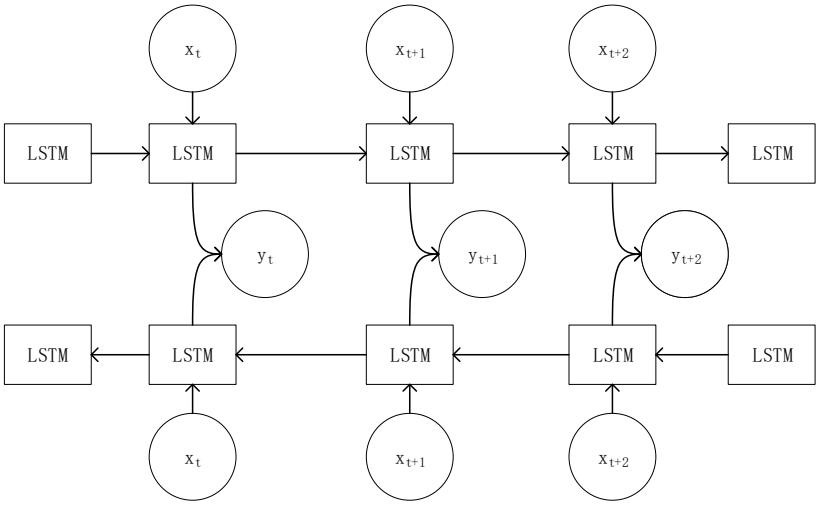

图 2.7: BLSTM 处理时序数据示意图

2.4.3　BLSTM 的优势

BLSTM 的优势在于，单向 LSTM 在 x_t 时刻的输出实际上只被 $x_0, x_1, \ldots, x_t$ 影响，即其只能考虑截止到当前时刻的数据，而 BLSTM 在某一时刻的输出值不仅受到该时刻正向数据影响，还能受到该时刻后向数据影响，因此有纵观全局的能力。

LSTM 增加门限机制，解决了 RNN 梯度爆炸或梯度消失的问题，而 BLSTM 通过双向传播机制，能够综合正向时序和逆向时序信息，是时序数据的分类和拟合问题（如自然语言处理问题）较为合适的算法。而目前已有很多利用自然语言处理和该模型进行程序分析的研究 [1, 9, 14, 44–46]。所以本系统采用 BLSTM 作为污点传播预测算法，将污点传播的切片作为上下文敏感的自然语言数据，通过训练集训练 BLSTM 参数将其保存，在给定污点传播的切片输入时，系统调用训练好的模型对其进行预测。

2.5　Django 框架

2.5.1　Django 框架简介

Django 是一个高级的 Python Web 开发框架 [14]，它使开发者能够对 Web 应用进行简洁实用的设计并对其进行快速开发。Django 由一群有经验的开发者开发，

[14] https://www.djangoproject.com/

其框架和内部设计避免了很多 Web 开发问题，帮助开发者只关心于他们 Web 应用本身逻辑而不需要重新开发通用组件。

Django 采用模型视图模板（MVT，Model-View-Template）模式，在模型层，其有一套高效的对象关系映射（ORM，Object Relational Mapping）框架和一些常用对象，比如用户对象，管理员对象等；在视图层，Django 借鉴了基于正则表达式的 URL 分派系统，能够让用户灵活的处理 URL 路由，并且内置的 Request，Response 对象能够让开发者快速的获取和返回数据；在模板层，Django 有自己的原生模板引擎，类似于 Jinja2，同时支持用户更换第三方引擎，保证用户充分自由的展现其数据。除此之外，Django 还内置一个用户管理后台和各类安全保障模块，大大加速开发者的开发过程。

2.5.2　Django 框架优势

本系统后台主要对机器学习模型操作，因此自然选择 Python 开发语言，而 Django 作为 Python 语言上应用最为广泛的 Web 开发框架之一，自然也成为本系统后台的首选技术。利用 Django 的灵活性，本系统利用其 URL 分派系统接受客户端传来的预测、训练、标记等请求并对其处理，通过 ORM 框架将程序切片、模型训练数据快速本地化，以完成对系统后台的快速开发。

2.6　本章小结

本章首先介绍目前流行的漏洞挖掘技术，并对其优劣势进行说明，接着介绍本系统使用的技术、工具和框架，并分别说明使用这些技术、工具和框架的原因。在技术角度，本章依次介绍污点分析技术、程序切片技术和 BLSTM 神经网络并对其优缺点进行总结，分析使用后向程序切片和 BLSTM 对污点传播会使污点分析更精确的原因；在工具角度，本章指出使用 Find Security Bugs 进行污点分析，使用 Joana 进行程序切片的原因及本系统对其进行的优化改进；最后简单介绍 Django 框架以及其优点。

第三章　Java 静态安全扫描系统需求分析与设计

3.1　系统整体概述

Java 语言通常用于 Web 服务和 Android 程序开发，其中大量安全漏洞类型可以通过污点分析法进行扫描，然而随着漏洞种类和开发方式不断丰富，污点分析存在大量误报，为此安全工程师手动设计更精巧的污点传播规则，或以增加漏报为代价换取低误报，或人工对大量误报漏洞判断，这两种做法无疑会增加潜在安全风险。

本系统旨在提供对于 Java 代码的精确漏洞静态扫描功能。相对于传统污点传播扫描工具，本系统结合了程序切片和 BLSTM，从已有误报漏洞片段进行分析，降低污点传播误报率。系统总体架构如图 3.1所示，系统采用 C/S 架构，主要分为客户端和服务端两大部分，客户端包含污点分析模块、程序切片模块、部分预测模块，而服务端包含预处理模块，另一部分预测模块和数据库。

污点分析模块基于 Find Security Bugs 进行改进，使之能够输出以污点传播树表示的污点传播路径，为误报预测提供基础数据；程序切片模块分析污点传播树，将其拆分为子污染流，利用 Joana 生成 SDG 并进行切片；数据预处理模块将切片模块输出数据进行泛化和向量化处理，使之可以输出预测模块；误报预测模块利用切片模块的子污染流和其对应切片，通过 BLSTM 进行切片预测，从而对漏洞进行预测。本系统客户端采用 Java 进行开发，管理后台和程序预测服务基于 Python、Django 和 Pytorch 开发。

3.2　系统需求分析

3.2.1　功能性需求

本系统功能性需求分析按污点分析模块，程序切片模块，预处理模块和误报预测模块四个方面进行。

污点分析模块由污点分析引擎对用户提交的 Jar 文件进行扫描，产生初步结果，该模块所涉及的功能性需求如表 3.1 所示，其主要功能为污点分析，而对于用户来说，系统需要友好的展现污点分析结果，即查看分析结果的需求，分析结果的主要内容为污点传播路径，污点传播路径由污点传播树表示，因此存在构造污点传播树和查看分析结果的需求，污点传播树由污点传播图构造而来，因此存在构造污点传播图需求，此外，污点分析是一项耗时操作，开发工程师可能

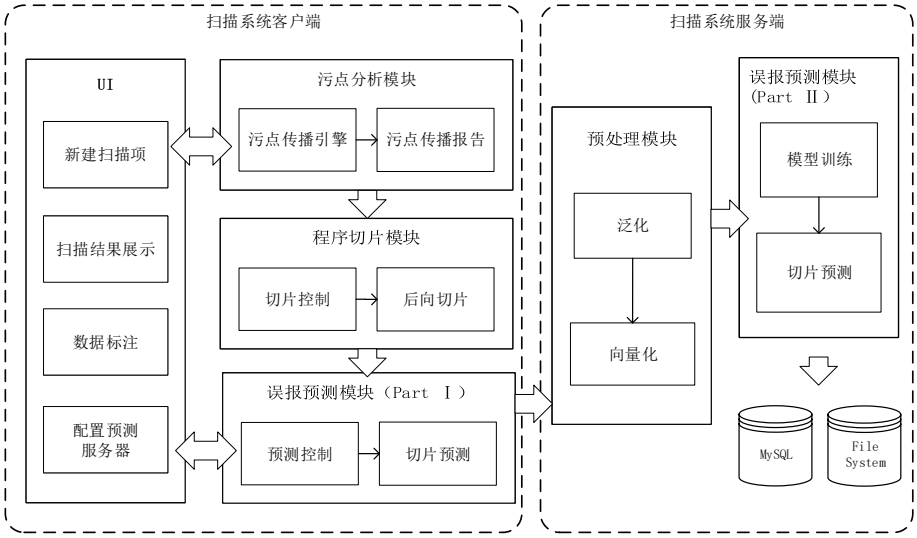

图 3.1: 系统总体架构图

会对一次污点分析结果进行保存，交给安全工程师进行漏洞验证，因此系统最好还需有保存分析结果和读取分析结果功能。

定义 3.1 (污点传播图)**.** 污点传播图为辅助生成污点传播树的一种图数据结构，其类似于函数调用图，图中顶点为与污点传播相关的函数摘要或返回语句，边为调用函数语句或执行返回语句的发生位置。

表 3.1: 污点分析模块功能性需求列表

需求编号	需求名称	需求内容	优先级
R1	污点分析	对于用户上传的代码进行污点分析，包括过程内分析和过程间分析	高
R2	构造污点传播图	对于代码中存在的汇聚点，构造其污点传播图	高
R3	构造污点传播树	对于一个污点传播图，构造多棵污点传播树，一个传播树上清晰标注有一个产生点和一个汇聚点	高
R4	查看分析结果	对于一个污点传播类型的分析结果，用户最终看到一棵或多课污点传播树，对于叶子节点，用户能够知晓其代码位置	高
R5	保存分析结果	用户可以对当前代码的漏洞分析详情进行保存，其中关键点在于对污点传播树进行序列化	中
R6	读取分析结果	用户可以对保存后的分析结果文件进行读取，二次查看漏洞详情，其中关键点在于对污点传播树结构的反序列化	中

程序切片模块对污点分析结果进行切片处理，该模块所涉及的功能性需求如表 3.2所示，该模块的主要功能是根据污点分析结果进行后向程序切片，将其拆解，得到三个子需求，首先其需要污点分析结果进行处理，包括对污点传播树进行拆分；对于每一个污染流，其包含有若干个子污染流，模块对这些污染流进行后向切片；最后，模块最好有格式化输出切片的功能，方便用户检查切片内容。

表 3.2: 程序切片模块功能性需求列表

需求编号	需求名称	需求内容	优先级
R7	处理污点分析结果	对污点分析的结果进行处理，对于每一个漏洞实例，反序列化污点传播树，并将每一个传播树拆分为多个子污染流	高
R8	后向程序切片	对于每一个 < 函数入口点，关注点 >，结合用户上传的 Jar 包，进行后向程序切片	高
R9	输出切片结果	对于切片，程序能将其表示为文本，即对切片结构体序列化	中

预处理模块对切片结果进行预处理，该模块所涉及的功能性需求如表 3.3所示，该模块的主要功能是对切片文本进行泛化和向量化，其中泛化需求主要保证一个切片能抽象代表一类代码片段；向量化需求指对于泛化后的单词序列进行 token 标记，产生字典和切片向量，以便作为 BLSTM 模型的输入。此外，一个较低优先级的需求为根据字典，将切片向量还原为切片的单词序列。

表 3.3: 预处理模块功能性需求列表

需求编号	需求名称	需求内容	优先级
R10	泛化	对于每一个切片进行泛化处理，包括抽象数值、字符串、类名和方法名等	高
R11	向量化	对于一个切片的单词序列，生成单词与整形值的字典并按字典对其向量化	高
R12	反向量化	对于一个单词序列的向量，根据字典将其还原为切片的单词序列	低

误报预测模块是保障漏洞报告准确性的核心模块，包括客户端、后端和 Web 前端部分，该模块所涉及的功能性需求如表 3.4所示，该模块的需求主要为对给定漏洞实例进行预测，标记以及模型训练。预测依赖后端服务，因此有配置预测服务器的需求，配置需要用户提供服务器地址和身份令牌（token）；对应的，管

理员有管理用户令牌的需求；预测分为预测切片安全性需求和预测漏洞真实性需求，分别在服务端和客户端实现；标记分为对切片安全性标记和漏洞真实性标记，当漏洞标记为正报时，用户需要指定一棵可利用的传播树，树中存在的所有污染流均标记为不安全，漏洞标记为误报时，用户需至少指定一处安全污染流（直到所有传播树都可被推导为不可利用）；同时，管理员有训练预测模型的需求；此外，系统有保存预测结果的需求，但由于预测过程并不是很占用资源，对该需求程度较低；最后用户可以清空标记和预测结果。

表 3.4: 误报预测模块功能性需求列表

需求编号	需求名称	需求内容	优先级
R13	预测切片安全性	对于一个切片向量，预测该切片是否安全（污染流无法传播）	高
R14	标记切片安全性	对切片向量，标记其是否安全	高
R15	预测漏洞真实性	用户在客户端可以查看预测结果，即该漏洞是否真实存在，需要结合切片预测需求	高
R16	标记漏洞真实性	用户在客户端可以对于一个漏洞，标记漏洞是否存在，不存在时用户需提供理由（指出安全污染流）	高
R17	训练预测模型	管理员可以在 Web 页面发起训练模型任务，或是安排定时器定期更新模型	高
R18	保存预测结果	用户在客户端可以保存漏洞预测结果	低
R19	清空标记和预测结果	用户在客户端可以清空标记和预测结果，以便重新预测	低
R20	配置预测服务器	用户在客户端配置远程服务器	中
R21	客户端令牌管理	管理员在 Web 控制台中发放或吊销令牌	中

3.2.2　非功能性需求

本系统旨在为开发者和安全工程师提供 Java 代码安全扫描功能，考虑到系统功能性质、使用场景和所属领域，系统非功能性需求主要有高效率，健壮性，保密性，安全性和扩展性五点。

首先，本系统功能点在于对 Java 代码进行安全扫描，其使用场景位于软件开发至测试阶段，因此扫描过程要保证一定的效率和健壮性，如果一次扫描等待时间过长，或是扫描过程中途崩溃，那么开发过程就会受到影响，甚至影响整个扫描服务的可用性，因此本系统需要是高效且健壮的。

其次，本系统涉及软件安全领域，系统本身很可能是恶意攻击者的首要攻击对象，因此需要具备保密性和安全性，这里的保密性是指保证用户提交的 Java 代码和 Jar 包数据安全，安全性是指扫描服务本身通过充分安全性测试。

最后，很多企业目前已经自研适用于自身特点的扫描器，为此，系统需要保证一定的扩展性，如将污点传播的解析抽象为接口，保证其后期可以自由更换污点分析扫描器。

3.2.3　系统用例描述

本系统主要用户为软件开发工程师和软件安全运营人员，根据对系统需求的分析，得到系统用例如图 3.2 所示，软件开发工程师和运营人员均有基于污点分析的静态扫描、扫描结果切片和预测、漏洞实例标记这三个用例，其中基于污点分析的静态扫描存在两例扩展用例，即保存污点分析结果和读取分析结果，漏洞实例标记实际包含标记正报结果和标记误报结果用例，安全运营人员还多出模型训练和客户端令牌管理的用例。

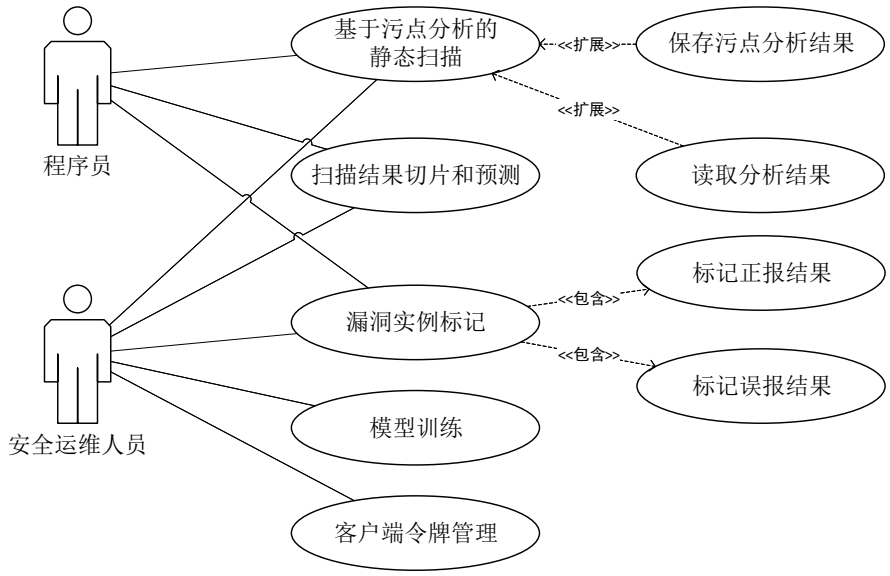

图 3.2: 系统用例图

基于污点分析进行静态扫描是本系统的基本功能，同样也是漏洞预测的基础。其用例描述如表 3.5 所示，安全工程师或软件工程师（下面统称“用户”）对将开发好的项目进行构建，新建项目时将源代码和构建得到的 Jar/War 包添加到项目中，随后系统对其进行污点分析，并返回给用户潜在的安全漏洞。

污点分析后，用户可以对其分析结果进行保存和读取，其用例描述见表 3.6，用户点击保存按钮后，系统将污点传播树的每一节点序列化为漏洞注解，并将含有一组注解的一系列漏洞保存为本地文件。用户可以在启动客户端后读取本地文件，加载文件的扫描结果。

表 3.5: 基于污点分析的静态扫描用例描述

描述项	说明
用例编号	C1
用例名称	基于污点分析的静态扫描用例
对应功能需求编号	R1、R2、R3、R4
参与者	用户
前置条件	用户已开发好项目并已构建的 Jar/War 包
后置条件	无
正常流程	1. 用户选择“文件”，“新建项目”，打开新建项目会话框 2. 用户输入项目名称，选择需要分析的包文件以及源文件所在目录 3. 用户点击“分析”按钮，程序开始对其进行基于污点分析的静态扫描 4. 用户在界面上看到漏洞列表，点击一个污点传播类型的漏洞 5. 用户在界面上看到该漏洞的若干污点传播树 6. 用户点击任意污点传播树的叶子节点，在界面上看到对应的源代码
异常流程	2. 若出现异常，则弹出异常会话框并显示异常信息

表 3.6: 保存污点分析结果用例描述

描述项	说明
用例编号	C2
用例名称	保存污点分析结果用例
对应功能需求编号	R5、R6
参与者	用户
前置条件	用户已创建项目且进行基于污点分析的静态扫描
后置条件	用户指定位置出现保存好的分析报告
正常流程	1. 用户选择“文件”，“另存为”，打开另存为对话框 2. 用户输入保存文件名、文件类型和保存位置，点击保存按钮，分析结果被保存 3. 用户选择“文件”，“打开”，显示打开文件对话框 4. 用户在打开文件对话框中选择报告文件，点击“打开”按钮，程序加载并在界面上显示污点分析报告中的内容
异常流程	2. 若出现异常，则弹出异常会话框并显示异常信息

对于客户端来说，对漏洞进行标记或者预测前提需要设置远程服务器，设置预测服务器的用例描述如表 3.7 所示，用户在客户端的菜单栏选择配置服务器功能，在对话框中输入管理员发放的令牌和服务器地址，完成远端服务器的配置。

客户端令牌管理的用例描述如表 3.8 所示，管理员通过在 Web 后端对客户端令牌进行管理，即新增令牌或吊销令牌等操作。

扫描结果切片和预测功能是本系统的最核心功能，程序首先对污点传播结

表 3.7: 设置预测服务器用例描述

描述项	说明
用例编号	C3
用例名称	设置预测服务器用例
对应功能需求编号	R21
参与者	用户
前置条件	管理员已向用户发放令牌
后置条件	无
正常流程	1. 用户选择“AI”，“Set Server”，显示服务器设置对话框 2. 用户服务器对话框中设置服务器地址以及令牌信息 3. 用户点击“验证”按钮，检查配置是否正确 4. 用户点击“应用”按钮，完成服务器配置
异常流程	2. 若出现异常，则弹出异常会话框并显示异常信息

表 3.8: 客户端令牌管理用例描述

描述项	说明
用例编号	C4
用例名称	客户端令牌管理用例
对应功能需求编号	R21
参与者	安全运营人员
前置条件	无
后置条件	无
正常流程	1. 用户登录 Web 控制台 2. 用户在控制台中点击“Client Token”进入客户端令牌管理页面，对客户端令牌进行增加，修改，删除等操作
异常流程	无

果分析，拆解为污点传播片段并分别切片，再由服务端预测每一个切片是否为清洁片段，最后得出漏洞实例是否为真实漏洞。涉及的用例描述如表 3.9所示。

预测模块的学习模型为监督式学习模型，因此对漏洞报告结果进行标记是预测准确性提升的关键，在实际应用时，即使没有学习模块，开发者或是安全工程师也会对漏洞报告进行是否为误报的判断，因此本系统只是利用用户的反馈使其构成回路，而并没有加重用户的使用负担。

标记正报结果用例描述如表 3.10所示，若用户认为漏洞真实存在的，用户可将其标记为正报，此时用户，此时所有污染流片段均自动标记为不安全，并且发送给服务端。

标记误报结果用例描述如表 3.11所示。若漏洞实例被用户判断为不存在，用

表 3.9: 扫描结果切片和预测用例描述

描述项	说明
用例编号	C5
用例名称	扫描结果切片和预测用例
对应功能需求编号	R7、R8、R9, R10、R14、R16
参与者	用户
前置条件	用户已完成远程服务器配置并且已有污点分析结果
后置条件	预测所需所有切片信息发送至服务器
正常流程	1. 用户选择“AI”点击“Slice and Predict”，程序开始对污点分析结果进行切片和预测 2. 用户等待切片和预测完成时，可以在弹出的会话框中观察预测进度，并随时取消预测 3. 用户在程序主界面看到每一漏洞预测结果，预测为误报的漏洞由灰色图标标识且在漏洞列表标注为“（P: FP）”，预测结果为正报的漏洞图标不变，文字标注为“（P: TP）” 4. 用户在点击一个漏洞，在污点传播图上可以看到预测解释，当预测结果为误报时，所有与预测为安全的污染流相关叶子节点由“（*）”标注
异常流程	2. 若出现异常，则弹出异常会话框并显示异常信息

表 3.10: 标记正报结果用例描述

描述项	说明
用例编号	C6
用例名称	标记正报结果用例
对应功能需求编号	R15、R17
参与者	用户
前置条件	用户已经对污点分析结果进行切片和预测
后置条件	用户指定漏洞的所有污染流片段均标记为不安全且发送至服务器
正常流程	1. 用户右键点击需要标记的正报漏洞实例，弹出标记菜单 2. 用户点击“Labeled as True Positive”，该漏洞被标记为正报，漏洞说明内容被显示为“[L: TP]”
异常流程	2. 若出现异常，则弹出异常会话框并显示异常信息

户可以对该漏洞标记为误报，此时用户需要指定若干条最短的污染流，直到该漏洞的所有污点传播树都无法被利用，完成后用户可以看到漏洞被标注为“[L: FP]”，同时，这些“安全污染流”被发送至服务端，用于下一轮的模型训练。

由于切片、预测是复杂操作，系统会将其放入缓存，因此本系统提供一键清除切片、标记和预测结果的功能，当用户有该需求时，可以点击“AI”菜单下的“Clean DB”按钮，将当前数据清空。其用例描述如表 3.12所示。

表 3.11: 标记误报结果用例描述

描述项	说明
用例编号	C7
用例名称	标记误报结果用例
对应功能需求编号	R15, R17
参与者	用户
前置条件	用户已经对污点分析结果进行切片、预测或标记
后置条件	无
正常流程	1. 用户右键点击需要标记的误报漏洞实例，弹出标记菜单 2. 用户点击“Labeled Safe Flow”，弹出误报标记会话框 3. 用户选择若干条最短安全污染流，在选择时用户看到该污染流的哈希和切片信息，点击确定后完成标记 4. 用户看到其所标记的漏洞被标注为“[L: FP]”，并且在污染传播树中，所有与被用户标记的那段污染流相关的叶子节点由“[*]”标记。
异常流程	2. 若出现异常，则弹出异常会话框并显示异常信息

表 3.12: 清空切片、标记和预测结果用例描述

描述项	说明
用例编号	C8
用例名称	清空切片、标记和预测结果用例
对应功能需求编号	R20
参与者	用户
前置条件	用户已经对污点分析结果进行切片和预测
后置条件	用户指定漏洞的最短安全污染流被标记为安全且发送至服务器
正常流程	1. 用户选择“AI”点击“Clean DB”，程序清空切片、预测和标记数据，并在漏洞的预测标签上显示“(P: UNK)”，在标记标签上显示“[L: UNK]”
异常流程	无

模型训练的用例描述如表 3.13所示，管理员通过在 Web 后端发起模型训练请求，系统会进行异步调用，通过消息队列，训练程序会根据选择的训练配置进行模型训练，并将训练模型保存，供预测时调用，用户也可以设置定时任务，系统将按时自动训练模型。

3.3　系统总体设计

本系统的框架图已在本节第一章图 3.1 展示，下面将对系统设计的模块从逻辑视图、开发视图、进程视图、物理视图和场景视图五个方面 [47] 详细介绍系统总体设计。

本系统的逻辑视图如图 3.3所示，该视图主要反映了本系统的对象设计。

表 3.13: 模型训练用例描述

描述项	说明
用例编号	C9
用例名称	模型训练用例
对应功能需求编号	R11, R12, R18
参与者	用户（这里特指安全运营人员）
前置条件	数据库中已存在用于训练的切片和标记数据
后置条件	数据库中记录已经训练好的模型
正常流程	1. 用户登录 Web 控制台 2. 用户在控制台中点击“Model Config”进入模型配置页面，查看或新建模型配置 3. 用户在控制台中点击“Periodic Tasks”，进入任务页面，在任务页面中选择模型训练任务，并设置任务类型（定时任务或单次执行任务）并且指定时间，以及训练指定的模型配置，系统到规定时间后自动训练模型
异常流程	4. 若模型训练中遇到错误，将错误信息返回到任务结果表中，供用户查看

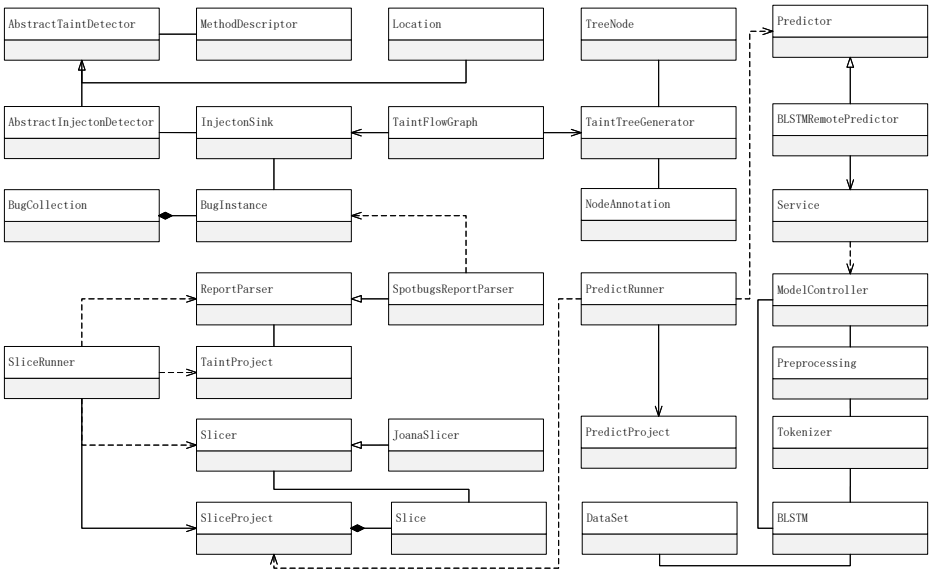

图 3.3: 系统逻辑视图

图中 AbstractTaintDetector、MethodDescriptor、Location、AbstractIncjectionDetector、InjectionSink、TaintFlowGraph、TaintTreeGenerator、TreeNode、NodeAnnotation、BugInstance、BugCollection 类主要与污点分析相关（部分类在 Find Security bugs 中已有定义，本系统对这些类存在重用或改造）。AbstractTaintDetector 是污

点传播分析类的抽象类，与之相关的类有 MethodDescriptor 和 Location ，其分别表示函数摘要和代码位置；AbstractIncjectionDetector 是注入类型漏洞分析器的抽象类，继承自 AbstractTaintDetector，相对于 AbstractTaintDetector 其主要增加输出漏洞报告的功能，目前几乎所有污点分析的漏洞类都继承该类；与 AbstractIncjectionDetector 相关的有 InjectionSink 类，该类表示污点汇聚点，其记录汇聚点可产生污点传播图，即 TaintFlowGraph 类；与 TaintFlowGraph 相关的有 TaintTreeGenerator ，即污点传播树生成类，其用于生成污点传播树，树上节点为 TreeNode 类；NodeAnnotation 是树节点注解类，用于在用户界面上显示一个污点传播树的叶子节点；若干棵污点传播树共同构成一个漏洞实例 BugInstance 类；若干个 BugInstance 类组合产生 BugCollection，一个项目的漏洞集合类。

图中 SliceRunner、TaintProject、SliceProject、Parser、Slicer、SpotbugsReportParser、JoanaSlicer 和 Slice 类主要与程序切片相关。SliceRunner 为切片控制类，其首先调用报告翻译器对漏洞报告进行翻译，产生污点项目 TaintProject 类，接着调用切片器对程序进行切片，最终产生切片项目 SliceProject 类；Parser 和 Slicer 分别为报告翻译器和切片器接口；SpotbugsReportParser 为 Spotbugs 的报告翻译器，实现翻译器接口，负责筛选出 Find Security Bugs 的污点传播类型漏洞并进行翻译；JoanaSlicer 为基于 Joana 的切片器，实现切片器接口；切片项目中包含切片集合，切片类为 Slice，其表示与服务器通讯时切片的数据结构。

图中 PredictRunner、Predictor、PredictProject、BLSTMRemotePredictor、Predictor、Service、ModelController、Preprocessing、Tokenizer、BLSTM 和 Dataset 类主要与预处理和预测相关。PredictRunner 为预测控制类，其依赖预测器接口 Predictor，该结构对漏洞实例进行预测，返回为预测项目类 PredictProject 的实例；BLSTMRemotePredictor 为远程调用服务端预测接口的预测器，实现 Predictor 接口；后端服务器所有服务表示为 Service 类，其中包含对 BLSTM 模型的控制类 ModelController；表示预处理的类有 Preprocessing 类，该类中有对切片的各类泛化操作，以及 Tokenizer 类，其中包含对切片的向量化和反向量化操作；最后模型类表示为 BLSTM 类；其通过数据集 Dataset 类进行训练。

本系统的开发视图如图 3.4所示，该视图以开发者角度描述模块的静态组织结构，具体来说本系统分为用户界面（UI），服务（Service）和第三方依赖包（Third-part Dependency）组成。

UI 主要包括客户端包和管理员 Web 端包；Service 中存在污点分析（TaintAnalysis）包，切片（Slice）包，Joana 切片器（Joana）包，预处理（Preprocessing）包，预测模块客户端包（PredictCli）和服务端（PredictSrv）包，以及用于表示各种数据的 Data 包；第三方依赖包中主要有 Spotbugs 和 Find Security Bugs，Spotbugs

包含基础 GUI 界面和数据流分析框架，Find Security Bugs 包括基本污点分析引擎，Joana 为本系统选用的切片器，Pytorch 用于构建 BLSTM 模型，Django 框架用于构建后端服务，Celery 和 Redis 用于训练请求的异步调用，EhCache 用于客户端的数据缓存。

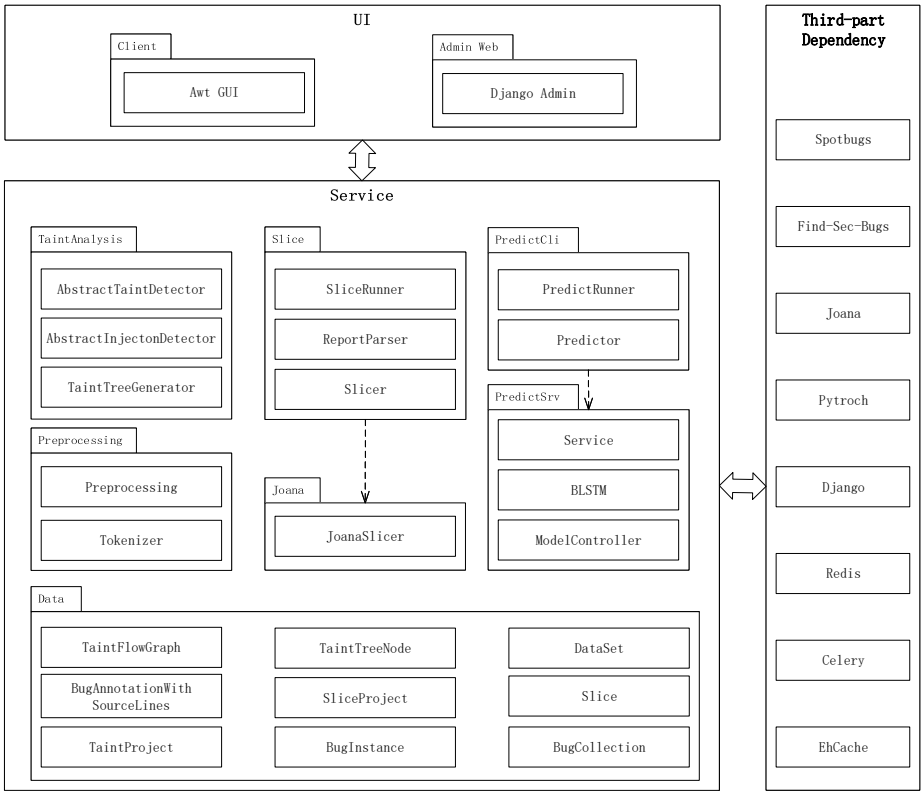

图 3.4: 系统开发视图

本系统的进程视图如图 3.5所示，用户端有管理端浏览器进程和代码扫描客户端进程，这两个进程通过 HTTP 协议与后端服务进程通信，服务进程通过 TCP 与模型训练任务进程和数据库进程通信，模型训练进程监听训练任务队列消息，收到训练消息后则开始模型训练，期间与数据库进程通信，获取训练配置信息。

本系统的物理视图如图 3.6所示，该视图在部署方面描述了本系统的架构。

在用户端部分，主要功能集中于扫描客户端，其包括 GUI 界面和核心 Jar 包（Core.jar），核心 Jar 包中含污点分析、切片和一部分预测模块。管理员通过浏览器使用本系统，主要完成发布模型训练任务、发放客户端 Token 或是管理数据对象的操作。

用户端通过 HTTP 与应用服务器通信，应用服务器上部署了基于 BLSTM 预测的 Web 应用，以及配置好的 Django 管理员后台应用，服务器上还部署了

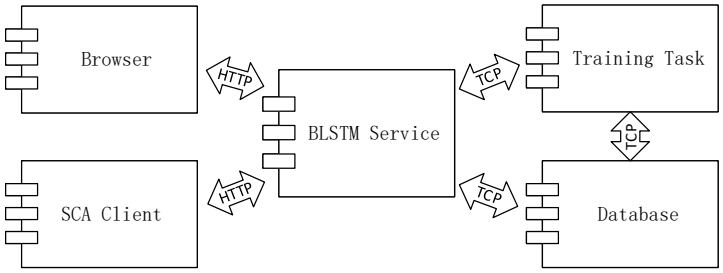

图 3.5: 系统进程视图

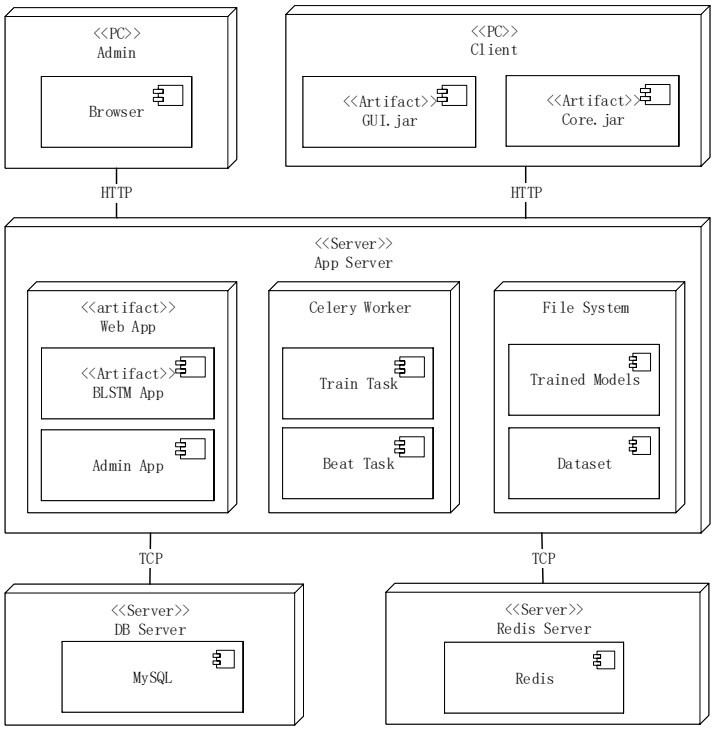

图 3.6: 系统物理视图

Celery Work，以便管理员发布关于模型训练的异步任务和定时任务，训练数据和已经训练好的模型直接存放在服务器文件系统上。

服务端通过 TCP 与数据库连接，MySQL 数据库主要存放本系统的各种实体数据，Redis 数据库供 Celery 使用。

本系统的场景视图即用例图（图 3.2），在 3.2.3 章已经详细说明。

3.4　污点分析模块设计

污点分析模块是系统的基础模块，以用户提交的 Jar 包为输入，输出污点分析报告，本节从流程和类图两方面，说明该模块设计。由于本模块主要是对 Find Security Bugs 进行改进，使之在报告中展示污点传播路径，本节只说明改进部分和与改进部分相关的类和流程，省略或简化其他部分。

3.4.1　流程设计

本系统的污点分析模块流程如图 3.7 所示，首先用户输入被扫描的 Jar 包对象，模块初始化漏洞报告，即产生一个空漏洞实例集合；接着其调用分析器对每一个类进行分析，对于第 i 个类，模块会对类函数的 CFG 进行拓扑排序，再依次对 i 类中的每个函数 f 进行污点分析，在污点分析时，本模块需要记录关于污点传播的有关信息，包括函数 f 内调用指令（如 *invokestatic*、*invokedynamic*、*invokeinterface* 等）位置、函数返回指令位置（如 *ireturn*、*areturn*、*dreturn* 等）和与污点汇聚点相关的代码语句位置信息，以便之后通过污点传播图和传播树产生漏洞报告。

污点传播最终输出为若干棵污点传播树，但是污点实际上是以图的形式进行传播的，因此本模块设计在漏洞报告阶段先生成污点传播图，在分析应用中类和函数时，模块会遍历所有污点汇聚点，记录汇聚点本身以及与污点传播到汇聚点的所有函数调用信息，对于每个汇聚点，利用汇聚点函数调用信息、分析时记录的函数调用语句位置和函数返回语句位置产生污点传播图。

最后，根据污点传播图构造污点传播树，并将传播树表示为能由 GUI 展示的漏洞实例对象，放入漏洞集合中，当所有汇聚点遍历完成后，模块最终输出漏洞实例的集合。

图中记录污点传播信息、构造污点传播图和构造污点传播树的流程将在第 4.2 节中详细说明。

3.4.2　污点传播图类图设计

为方便图的遍历，污染传播图的数据结构采用邻接链表的形式，其类图设计如图 3.8 所示。

类 TaintFlowGraph 表示污点传播图，图上的点由 Vertex 类表示，边由 Edge 类表示。TaintFlowGraph 中 entries 属性表示这张图入口函数的集合，这些函数只有出度没有入度，换句话说，污点在这些函数的上下文中传入，并传播至图中其他点，通常这些函数为 Web 应用的视图层函数（如继承 HttpServlet 子类中

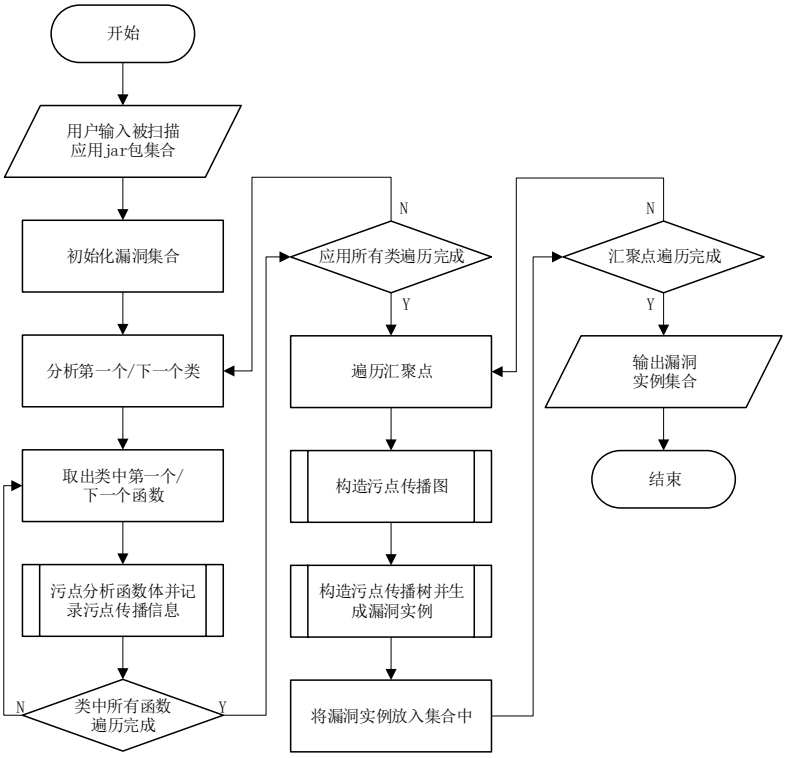

图 3.7: 污点分析模块流程图

doGet() 或 *doPost()* 方法）或是客户端程序的入口函数（如 *main()* 函数），该类中有向图中增加点的方法 *addVertex()* 和向图中增加边的方法 *addEdges()*，还有一个检查函数摘要是否存在于图中的方法 *contains()*。函数摘要类 MethodDesc 在系统逻辑视图中已出现过，该类主要用来唯一标识类中的函数，含有属性 className、methodName、methodSig 和 isStatic，分别表示类名，方法名，函数签名和是否为静态类。

顶点类 Vertex 是一个抽象类，具体派生出表示函数的函数顶点类 MethodVertex 和返回指令顶点类 RetVertex，MethodVertex 包含一个函数摘要 caller，以及该函数内连向其他顶点的边 edges，并且包含添加边的方法 *addEdge()*。返回指令顶点类表示函数体中返回污点变量的返回值语句，其属性只有该语句类型 retType——通常来说 Java 字节码返回语句有 *ireturn*，*dreturn*，*areturn* 等。不难发现，污点传播图上的顶点为函数或返回语句，并且只有 MethodVertex 才可以有出度。

最后介绍与边相关的类，Edge 类，该类表示一条边和连接边的对端顶点，peer

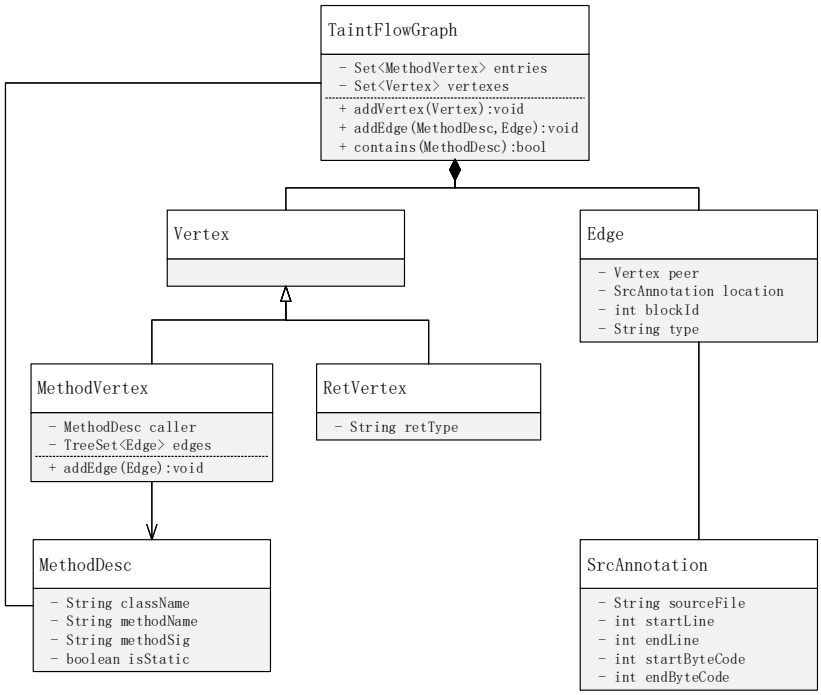

图 3.8: 污点传播图设计

属性表示对端顶点，location 代表两点之间通过哪一个代码位置与之相连，type 指这条边的类型，模块中主要有三种类型（“CALL”，代表函数调用；“SINK”，代表调用到汇聚点，“RET”，代表函数返回），blockId 属性表示这条边所在的代码块，该属性的目的是为了在结合 CFG 和污点传播图生成传播树时，保证边的顺序。代码位置由 SrcAnnotation 表示，该类将在下一章详细说明。

3.4.3　污点传播树和漏洞报告类图设计

污点传播树是展示污点传播路径的重要数据结构，其中叶子结点中有一污点入口点和一污点汇聚点，其他叶子节点为函数或返回语句节点，树的层级关系反映出函数调用关系，一棵传播树代表一个唯一的污点传播全流程，其结构体和漏洞报告类图设计如图 3.9 所示。

对于污点传播树，有一工具类 TaintTreeGenerator，其有两个方法：*makeTree()* 和 *makeAnnotation()*，*makeTree()* 方法输入为污点传播图的一个入口点，根据这一入口点构造若干棵传播树；*makeAnnotation()* 根据污点传播树构造出一注解列表，这些注解列表最终在 GUI 界面上展现污点传播树。构造污点传播树首先需要根据入口点计算产生路径，因此有 PathGenerator 类，该类包含函数内与污点传播相关的所有边 edges，以及函数的 CFG，*getPaths()* 用于获取一个函数内部

关于边集合的所有路径，*getOnePath()* 用于计算一个函数的单条路径，*dfs()* 用于计算路径，并将其放在 paths 属性中。

考虑到污点传播树为树形结构，并且节点存在运行顺序特征，本模块设计使用树的孩子兄弟结构来表示污点传播树。TreeNode 表示树上节点对象，其 id 表示该节点的序号，其主要用来在反序列化时重构树；location 属性表明该节点在程序中的位置，type 表示节点类型，在本系统中，该类型种类和传播图的 Edge 类型一致；firstChild 为 TreeNode 对象，指向其第一个子节点，具体来说，父节点是一个函数节点，而子节点可以为函数节点或返回节点；nextSib 为 TreeNode 对象，指向同一深度的下一个叶子对象，具体地说，是同一函数上下文下的下一对象。TreeNode 派生出两类节点，一个是表示函数的节点 MethodTreeNode，其有一类型为 MethodDesc 的属性 caller，另一个是表示返回语句的节点 RetTreeNode，其有一属性 retType 表示返回语句类型，与上文 RetVertex 类型一致。

漏洞注解类 BugAnnotation 、漏洞实例 BugInstance 类和漏洞实例 BugCollection 类都是 Spotbugs 定义的数据类型，漏洞注解类 BugAnnotation 为抽象类，表示漏洞的单行解释性文字。BugInstance 是应用中一个漏洞的基本单位，其由一个漏洞类型（type 属性）和多行漏洞注解（list 属性）组成。在 BugAnnotation 中，*format()* 是该条注解在 GUI 界面上的显示文字时被调用的方法，*writeXML()* 用于将注解保存为 XML 格式，以上为原生 Spotbugs 定义的方法，除此之外，为了方便反序列化，本模块还为其增加 *fromXML()* 方法。NoteAnnotation、RetNodeAnnotation 和 MethodNodeAnnotation 对应于污点传播树的 TreeNode、RetTreeNode 和 MethodTreeNode 节点，对于抽象的节点注解类 NoteAnnotation 而言，其 id、firstId，nextId 分别指向 TreeNode 的 id，第一个孩子节点 id 和第一个兄弟节点 id，depth 表示该节点距离树根的深度，具体类 RetNodeAnnotation 和 MethodNodeAnnotation 将实现 BugAnnotation 类的三个方法，特别的，RetNodeAnnotation 的 firstId 恒为 -1，因为返回语句的节点不会存在子节点。BugCollection 类主要有漏洞实例组成，除此之外，其还记录有项目名称（project），用户提交的应用 jar 包（appJars）、依赖包（libJars）和源码包（srcs）。

3.4.4　污点分析器类图设计

污点分析器类图设计如图 3.10 所示，Detector 类是所有分析器的基类，对于每一个 Detector 子类，Spotbugs 会按流程图 3.7 所示使其分析目标 Jar 中的每个类，其中调用的就是类中 *visitClassContext()* 方法，类遍历完成后，又会调用 *report()* 方法产生报告。

AbstractTaintDetector 继承 Detector 类，该类也是记录污点传播信息的重要

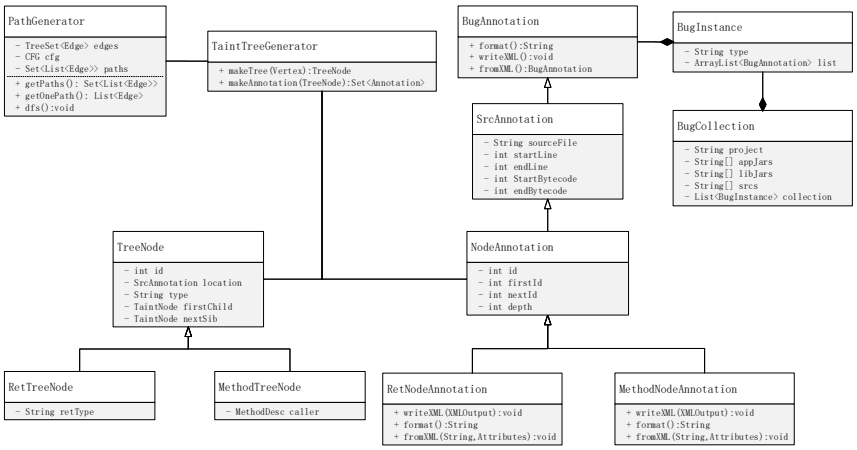

图 3.9: 污点传播树和漏洞报告类图设计

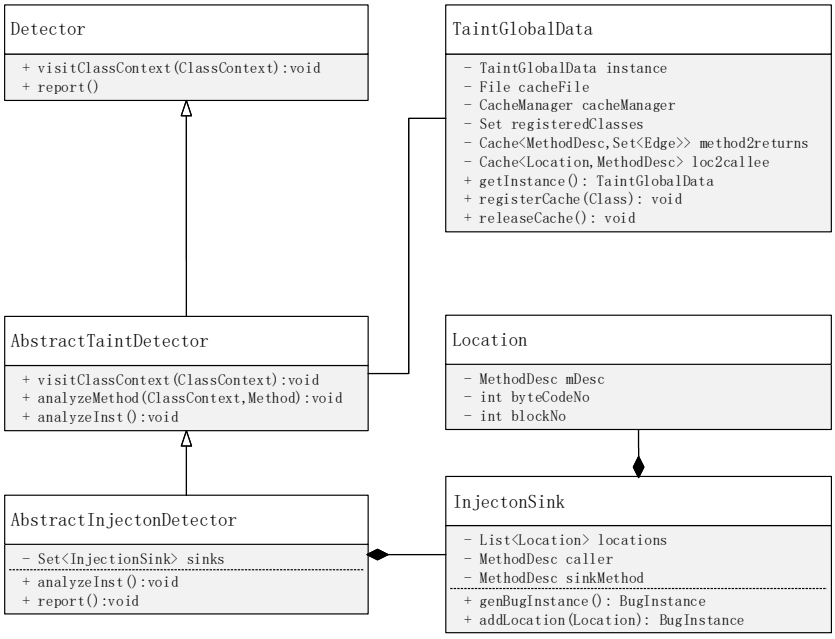

图 3.10: 污点分析器类图设计

基类，其实现 Detector 的 *visitClassContext()* 方法，并在其中调用 *analyzeMethod()* 方法分析每一个目标函数，本文优化了该方法，在方法中增加记录函数调用关系和函数返回语句位置的逻辑。对于特殊指令，如 *return* 和 *invoke*，*analyzeMethod()* 会调用 *analyzeInst()* 对指令进行分析。

TaintGlobalData 类是一个单例类，使用 EhCache 保存被测程序中的函数调用关系（loc2callee 属性）和函数内部返回语句位置（method2returns 属性），loc2callee

实际上记录函数调用发生时 invoke 指令所在函数位置与被调函数摘要的映射，method2returns 实际上记录函数摘要和以返回语句集合的映射，这些信息将在生成污染传播树时使用。其 *getInstance()* 用于获取唯一实例，*registerCache()* 和 *releaseCache()* 用于一个 Detector 注册或释放缓存，当所有 Detector 释放缓存后，释放缓存方法会彻底清除缓存资源。

AbstractInjectionDetector 继承 AbstractTaintDetector，进一步实现污点分析过程，即实现 *analyzeInst()* 和 *report()* 方法，其属性 sinks 记录目标应用的所有汇聚点，在所有类分析完成后，依次调用 *report()* 生成报告。其构造方法会调用，在 *report()* 方法结束后，其会调用 TaintGlobalData 的 *registerCache()* 方法，在 *report()* 方法中，其会调用 *releaseCache()* 方法。

污点汇聚点的类名为 InjectionSink，其属性 caller 和 sinkMethod 为调用敏感函数的最后一个函数，以及敏感函数的函数摘要，除此之外，所有将污点汇聚到该点的调用语句有被记录在其 locations 列表上，*addLocation()* 就是记录调用语句发生位置的方法，*geit adtBugInstance()* 方法为产生报告的具体方法，被 *AbstractInjectionDetector.report()* 方法调用，本系统对该方法进行改进，方法中将调用构造污点传播图、传播树和生成漏洞实例方法。

上面提到的位置信息由 Location 类表示，其包含的属性有语句所在函数的函数摘要（mDesc），语句所在的字节码位置（byteCodeNo）和所在 CFG 代码块的序号（blockNo）。

3.5　程序切片模块设计

程序切片模块是误报预测的基础，本系统以前人工作为基础，根据污点传播类型漏洞提出新的切片方式，能够对一个漏洞进行完整切片，本节从流程设计类图设计两方面介绍该模块的设计。

3.5.1　流程设计

本模块的流程设计如图 3.11 所示，该模块的输入是用户提交的项目 Jar 包和在上一模块已经产生的漏洞报告中的漏洞实例集合，其输出是切片项目，项目中含有所有漏洞及对应的所有切片集合。

因为 Spotbugs 中除了上文提到的污点传播检测器还存在其他漏洞检测器，其他检测器产生漏洞不属于本系统处理范围，在模块运行前，先要对漏洞报告进行过滤，提取出污点传播类漏洞（如 XSS、SQL 注入等），反序列化这些漏洞的注解，还原其漏洞实例，接着模块初始化一个具体的程序切片器，其中包括对切片器的各种配置，具体会在实现章节介绍，接着初始化一个空的切片项目；遍

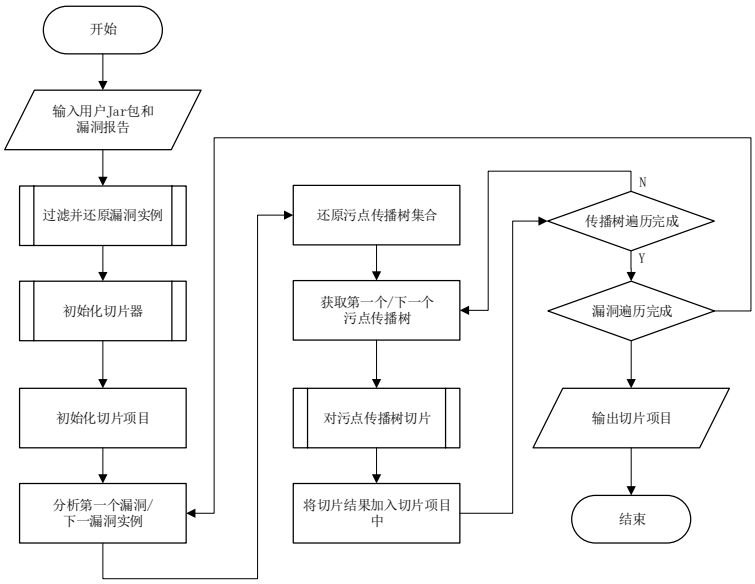

图 3.11: 程序切片模块流程图

历每一个污点传播漏洞，对于一个漏洞，根据其漏洞实例注解集，还原出若干棵污点传播树；对每一污点传播树进行拆解并切片，该步骤是本系统的核心步骤之一，包括对污点传播树进行拆解，将其分为污染流作为更小的切片任务单位，其具体实现也将在下一实现章节介绍；对污点传播树切片结束后，将传播树信息和对应切片信息加入切片项目中，遍历完所有漏洞的污点传播树后，模块输出切片项目，项目内包含与漏洞相关的所有切片。

3.5.2　类图设计

本模块的类图设计如图 3.12 所示。首先介绍模块的核心控制类 SliceRunner，该类实现完整的切片控制流程，即读取污点传播报告，至最终产生每一漏洞实例的切片集合，其有三个属性成员：slicer，表示用于程序切片器；parser，污点传播报告翻译器；project，最终输出切片项目。切片器和翻译器分别由 *setSlicer()* 和 *setParser()* 设置，*run()* 方法是执行对漏洞报告切片的方法，其参数是一个监视器，上一节所说的模块流程将在该方法内完成，*sliceTaintTree()* 是对一个污点传播树切片的方法，可以看到其参数是一个树节点，其表示该传播树的树根，*sliceFlow()* 为对污染流切片的方法。

Monitor 类为监视器接口，用于各种耗时较长的函数返回中间信息，具体地说，该接口的 *init()* 方法用于通知开始某一流程，第一个参数为过程名，第二个

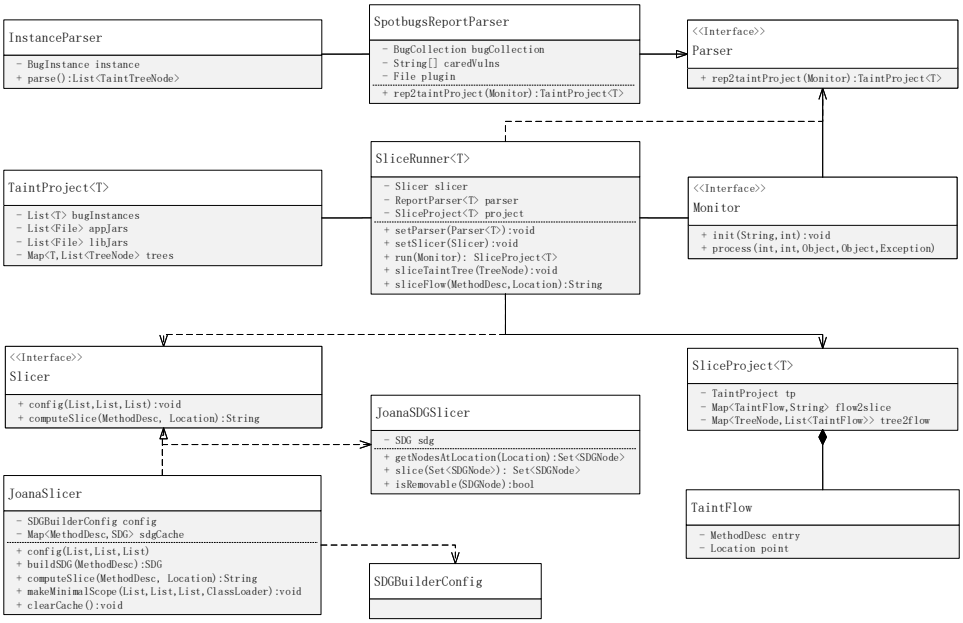

图 3.12: 程序切片模块类图

参数表示过程共几个子过程；*process()* 方法用于返回子过程信息，在每一个子过程完成后调用，其参数依次向监视器传递当前子过程序号、过程总数、子过程输入、子过程输出和遇到的异常。该接口规范系统内的任务进度报告模式，使外部能够自由地处理每一子过程日志并及时通知用户（例如，在每一子过程完成时向屏幕输出进度或是处理异常），除切片控制类会使用该类外，下文中预测控制类也会使用。

接下来介绍与污点传播翻译器相关的类，Parser 是翻译器接口，用于将漏洞报告翻译为一个污点传播项目（*rep2taintProject()* 方法），同样，该方法也有一个监视器作为参数。SpotbugReportParser 实现该接口，其属性有 bugCollection，该属性在类初始化时传入，表示污点传播报告，caredVulns 表示污点传播漏洞类型白名单，该属性为静态属性，plugin 为本系统优化后的 Find Security Bugs 插件文件地址。SpotbugReportParser 的 parse() 方法将 Spotbugs 项目的漏洞报告转化为污点传播项目，而真正将每一漏洞实例还原为若干棵污点传播树的是 InstanceParser 类中的 *parse()* 方法，该类成员变量 instance 为一漏洞实例，在类构造时传入。翻译的最终产出是 TaintProject 对象实例，该类中 bugInstances 表示所有的污点传播漏洞集合，appJars 和 libJars 表示用户提交的应用 Jar 包集合和第三方依赖 Jar 包集合，trees 为一组键值对，记录一个 bugInstance 的所有污点传播树的树根节点。

Slicer 是切片器的接口，用于对污染流进行切片，*config()* 方法用来配置一个切片器，其中三个列表参数分别代表引用 Jar 包，第三方依赖 Jar 包和黑名单类，当切片内容在黑名单类的上下文时，该条切片被舍弃，这主要用来过滤 Java Runtime 类中的切片，或者排除其他第三方包的切片。*computeSlice()* 方法对一个污染流切片，其参数是指以 MethodDesc 的函数为入口对关注点 Location 的切片，切片返回 String 类型，每条切片用“\n”分割。JoanaSlider 类是 Slicer 的一个具体实现，其 config 变量表示生成 SDG 的配置信息，配置类名为 SDGBuilderConfig，由于该类涉及到实现细节，将在下一章进行讨论，sdgCache 用于缓存 SDG，类中除了实现 *config()* 和 *computeSlice()* 方法，还有构造 SDG 的方法 *buildSDG()*，其参数为 SDG 入口函数的函数摘要，*makeMinimalScope()* 方法为构造最小搜索范围方法，被构造 SDG 时使用，clearCache() 方法用于清除各类缓存。在构造 SDG 后，后向切片逻辑在 JoanaSDGSlicer 类中实现，该类在创建时传入 SDG 作为属性，先通过 *getNodesAtLocation()* 获取关注点的 SDGNode，再通过 *slice()* 方法切出所有相关的 SDGNode，并且通过 *isRemovable()* 函数将一些无效节点删除（如非函数语句节点，抽象类节点等）。

切片模块最终返回切片项目类 SliceProject 对象，该对象内部持有一污点传播项目对象、从传播树到污染流的映射（tree2flow 属性）和从污染流到切片的映射（flow2slice 属性）。TaintFlow 对象代表污染流，可以看到，其属性 entry 和 point 正好对应切片器接口切片函数的参数，其实际上代表污点在一个函数内传播的过程。

从本模块的类图设计上可以看出，本模块基本实现对切片器和污点分析器解耦，在某些类上的泛型 T 表示漏洞类型类，考虑到今后不同污点分析工具的返回漏洞类型不同这一情况，即使今后需要换用其他污点分析引擎或是切片器，只需要重新实现 Parser 和 Slicer 就可重用本模块。

3.6　数据预处理模块设计

数据预处理模块是预测过程中承上启下的模块，其接受切片数据，并将其泛化和向量化，使之可以输入 BLSTM 模型。本节从模块流程设计和类图设计两方面介绍模块的设计思路。

3.6.1　流程设计

图 3.13 展示了本模块的流程图，首先模块输入上一模块产生的切片字符串，接着字符串经过各类泛化处理，变为由空格分隔单词序列，再根据该序列构建单词表，最后通过查表完成字符串的向量化，最终模块输出切片向量。

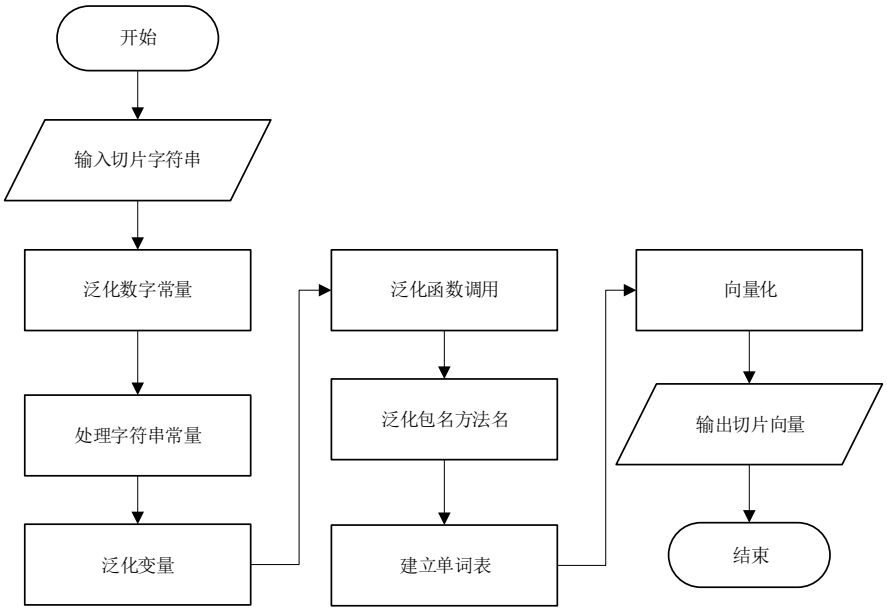

图 3.13: 数据预处理模块类图

泛化处理是模型准确性的重要保障，针对本系统应用场景，模块依次完成五类泛化处理，即泛化数字常量，泛化字符串常量，泛化变量，泛化函数调用和泛化包名及方法名，其具体做法将在实现章节说明。

3.6.2　类图设计

图 3.14 展示了本模块的类图，Preprocessing 类用于对切片数据进行泛化处理，其 rawSlice 属性为原切片字符串，strList 为字符串常量表，varList 为变量常量表，其 *preprocess()* 方法为主要的泛化方法，输出以空格分隔的单词（token）序列。*processNumber()* 、*processStr()* 和 *processVar()* 分别是泛化数字常量、字符串常量和变量的方法，它们的输入为一行切片，输出为泛化后的切片字符串。*processType()* 为泛化返回值类型的方法，其输入为表示返回值类型的字符串，输出为泛化后字符串。

Tokenizer 类用于将泛化后字符串转换为一维向量（向量化），其属性 freqGt 指只有当单词在数据集中出现频率大于该值时单词才有效否则表示为 UNK，其还包含一单词表对象 tokenDict，*buildDict()* 用于构造一个单词表，其输入为 Pytorch 中定义的 TextDataset，通过该对象可以遍历数据集中每一条数据，其输出为单词表。*encode()* 方法用于将泛化字符串编码为向量，*decode()* 是其反函数，用

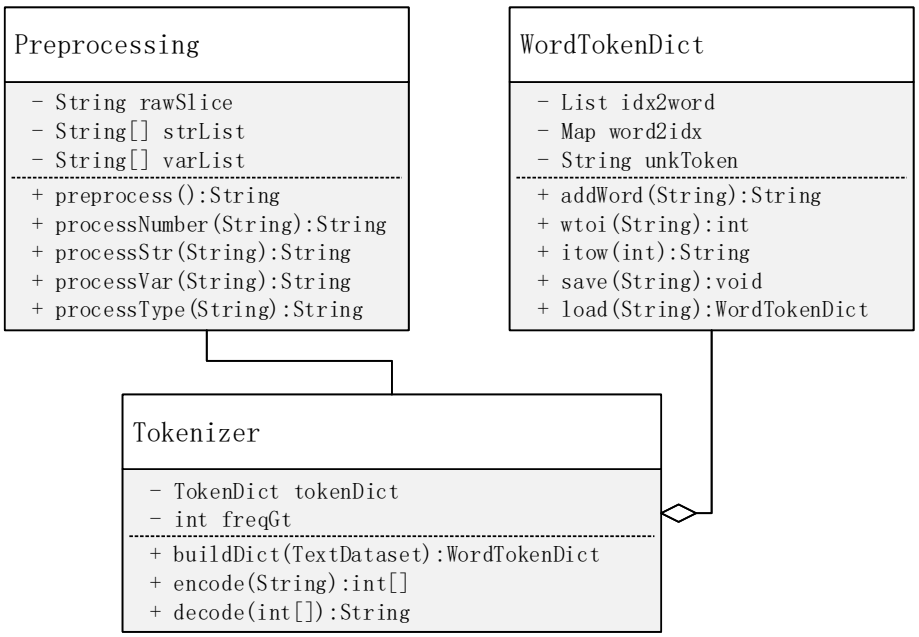

图 3.14: 数据预处理模块类图

于将向量还原成字符串。

TokenDict 为单词表类，用于记录单词和其整形 id 的对应关系，idx2word 和 word2idx 属性分别记录 id 至单词和单词至 id 的对照表，unkToken 属性记录未知单词的值（默认为“UNK ”），*addWord()* 方法实现将一单词放入单词表的逻辑，*wtoi()* 和 *itow()* 是一对互逆操作，用于将单词转换为数字 id，以及将 id 还原为单词，*save()* 和 *load()* 分别实现将单词表保存为文件，以及从文件中反序列化为单词表的操作。

3.7　误报预测模块设计

预测模块是系统最后也是最重要的模块之一，通过机器学习预测漏洞是否为误报，从而提高扫描结果的准确性。本节将通过架构设计、类图设计、流程设计三个方面介绍该模块的设计。

3.7.1　架构设计

相较于先前模块，预测模块的特殊之处在于其不只存在于一处硬件资源上（其跨越客户端和服务器），由于其结构较为复杂，因此本节首先介绍模块的架构设计，其架构图如图 3.15所示。

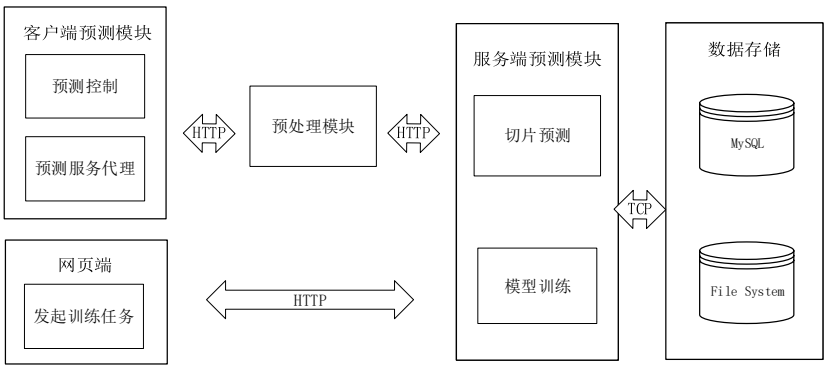

图 3.15: 预测模块架构图

预测模块的一部分存在于客户端中，其作用在于代理远程服务器，以及对于一个漏洞实例中的每一个传播树中每一个污染流切片进行预测，再根据预测结果判断一个实例是否是误报——预测控制。预测服务代理会将原始切片内容发送给服务端预处理模块，预处理模块转化后将向量发送给服务端预测模块。

服务端预测模块主要功能有基于 BLSTM 模型对于单个污染流切片进行预测、切片标记以及将标记数据和切片存储于数据库和本地系统中。同时用户可以直接通过网页端，发起训练任务，模型训练任务同样也在服务端预测模块中完成。

3.7.2　类图设计

本模块类图如图 3.16 所示，在客户端，有预测控制类 PredictRunner、预测器类 Predictor、切片类 Slice、标记类 Label、远程预测器代理类 BLSTMRemotePredictor、预测项目类 PredictProject 和预测器异常类 PredictorException；在服务端，有预测服务视图类 Views，模型控制类 ModelController，模型效果度量类 MetricCalculator 和神经网络模型类 BLSTM。

预测器控制类用于对一个漏洞报告项目进行预测，其 predictor 属性是其预测的预测器实例，sliceRunner 是生成切片项目的控制器，sliceProject 保存切片模块的切片项目，predictProject 保存最终预测项目，其有设置切片器的方法 *setSlicer()* 和设置预测器的方法 *setPredictor()*，进行预测的方法为 *run()*，预测进行前，若类中无切片项目，则会自动调用切片模块对其切片，接着再进行预测，最终方法返回预测结果。

PredictProject 类为预测结果项目类，包含最终预测结果，其属性 bugIsSafe 为漏洞实例至漏洞是否为误报的映射表（若漏洞安全则说明误报），bug2safeFlow

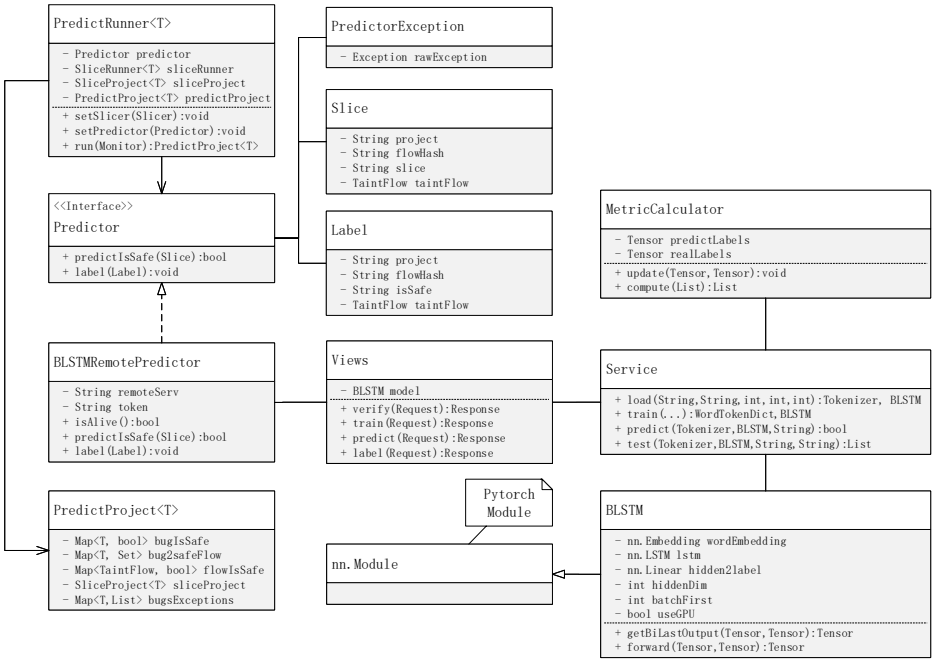

图 3.16: 预测模块类图

记录漏洞至安全污染流的映射，用于向开发者解释判断误报的原因（因为污点消失于那些污染流），flowIsSafe 记录单条污染流是否为安全（即污点无法在该污染流中传播），sliceProject 为切片模块的项目，bugsExceptions 记录预测运行时发生的异常。

预测器接口 Predictor 用于实现漏洞预测控制和切片预测本身逻辑的解耦，其有两个方法：*predictIsSafe()* 用于对一个切片进行预测，返回该切片是否安全，若切片安全，则说明污点在该污点流上无法传播，*label()* 用于对切片进行标记。Slice 类和 Label 类分别对应以上两种方法的输入对象，Slice 类除了包含切片内容以外，还包括污染流对象，污染流哈希和项目名，Label 类包括污染流对象、污染流哈希、项目名以及该污染流是否安全的标记。预测器在预测时允许抛出 PredictorException 异常，该异常类中含原始异常信息，以便错误跟踪。

BLSTMRemotePredictor 实现预测器接口，其属性 remoteServ 记录服务器地址，token 记录身份令牌，除了实现接口方法外，其还有 *isAlive()* 方法用于判断到远程服务器的连接是否有效。

Views 是远程服务器的对外服务类，其包含验证口令是否合法的方法 *verify()*（对应于 BLSTMRemotePredictor.isAlive()，处理训练请求的方法 *train()*，处理预测请求的方法 *predict()*，以及处理标记请求的方法 *label()*，其 model 属性代表当

前最新的模型实例。

Service 是模型的服务类，视图层接收到请求后，会将请求参数取出交给该类，再由该类调用 BLSTM 模型提供相应服务。这些服务主要有：*load()* 用于加载一模型，其参数分别为字典序列化文件地址、BLSTM 序列化文件地址、最小词频、词嵌入向量维度和隐藏层神经元个数；*train()* 用于训练一个 BLSTM 模型，由于其参数较多这里省略，详细可以参照下一节中表 3.20；*predict()* 对应预测请求，其参数分别为序列化器、BLSTM 模型，以及一个切片字符串，返回该切片是否安全。*test()* 用于测试一个神经网络模型准确率，前两个参数与 *predict()* 方法相同，后面两个 String 类型参数分别为切片数据文件夹和标记数据文件夹。

MetricCalculator 为度量计算器类，其通过一系列度量指标用于评估预测效果，其属性 predictLabels 代表预测标签，realLabels 为真实标签，*update()* 方法用于增加预测和真实标签数据，*compute()* 方法用于产生一系列度量值，其输入为一系列度量方法的名称，输出为这些度量的具体值。

BLSTM 类是 BLSTM 神经网络类，其继承于 Pytorch 模型基类 nn.Module，本模块的神经网络有三层组成，首先是词嵌入层，对应属性为 wordEmbedding，其作用是将一个单词序号转换为一维向量，这样一组单词序列就可以转化为二维张量数据，接着是 BLSTM 层，对应属性为 lstm，nn.LSTM 为 Pytorch 实现的 LSTM 神经网络对象，模块只需将其指定属性指定为双向即可构建 BLSTM 模型，最后为一个线性层 hidden2Label，该层用于将 BLSTM 输出映射到 0～1 范围上。hiddenDim 记录隐藏层神经元数目，batchFirst 记录批量训练时的二维向量中，单条数据是以行向量形式给出（$batchFirst = true$）还是列向量形式给出（$batchFirst = false$），useGPU 指模型计算时是否需要用 GPU，这三个属性均需要在建立模型时给出。*forward()* 方法为神经网络后向传播方法，其参数依次表示批量数据的二维向量，以及数据长度向量，返回预测结果，其会调用 *getBiLastOutput()* 方法，该方法获取两个方向上的 LSTM 最后一个时刻的输出。

3.7.3　流程设计

本模块流程图如图 3.17 所示，本模块以切片模块的报告为输入，输出漏洞预测项目，其中记录每个漏洞是否为误报的预测结果。

模块首先初始化并设置一个切片器实例，随即处理每一个漏洞实例，对于单个漏洞实例，漏洞传播模块已经将其分解为若干棵污点传播树，于是模块开始遍历这些传播树，对于每一个传播树，程序切片模块又已将其分解为若干条程序切片，因此模块将依次预测这些切片，并记录切片安全性，当切片遍历完成后，根据切片结果评估传播树的安全性，根据污点传播规则，只要有传播树上任

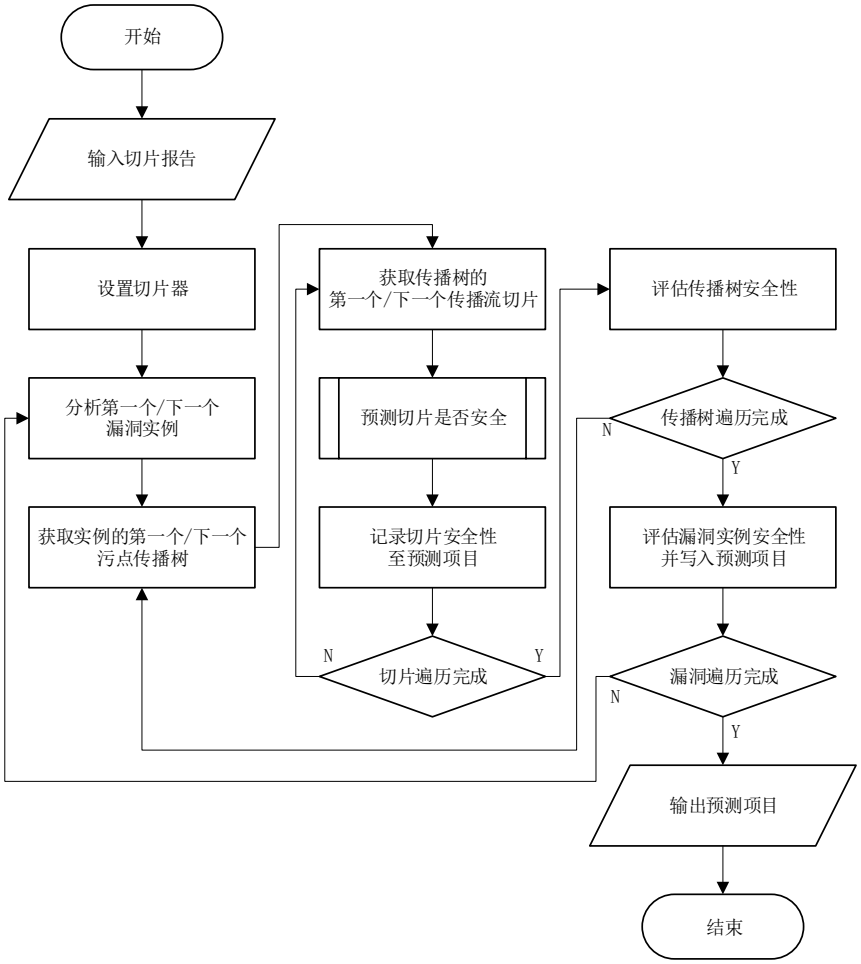

图 3.17: 预测模块流程

意一条污染流是安全的，那么整个传播树即为安全，接着当一漏洞的传播树遍历完成后，可以根据传播树的安全性评估漏洞是否为误报，根据漏洞触发规则，只要有一棵传播树是不安全的，那么漏洞即是真实的，否则为误报，将每个漏洞实例预测完后，模块输出预测项目结果。

本模块是系统中最后一个模块，也就是说，预测项目是本系统的最终输出，用户可以优先关注真实漏洞并展开修复工作，接着再观察误报漏洞是否为漏报，并对它们进行标记。

3.8　数据库设计

本系统的数据库主要用来存储用户身份信息、污点分析得到的污染流数据和标记、用于模型训练的参数配置信息以及模型本身信息。经过分类，可以总结为三类数据表：

第一类是与鉴权有关的数据表，由于本系统主要使用 C/S 架构，为防止恶意攻击者向服务器注入垃圾数据影响预测，需要对客户端身份进行鉴定，为此设立客户端口令 Client_Token 表，由安全运营人员进行维护。

第二类是与程序分析相关的数据表，由于本系统后台主要是对子污染流的切片进行有监督预测，因此自然需要污染流 Taint_Flow 表和对污染流的标记 Label 表，对一个污染流而言，需要对其函数入口和关注点（即语句位置）进行切片，因此存在函数摘要 Method_Description 表和语句位置 Location 表，不同项目可能具有名称相同的类、函数或文件，为了在逻辑上唯一标识一个函数摘要和语句位置，因此建立项目 Project 表。

第三类是与模型本身相关的数据表，本系统使用 BLSTM 进行切片预测，对 BLSTM 训练和 BLSTM 结构本身涉及到一系列参数，为此将其保存在模型配置 Model_Config 表中，供安全运营人员调整，对于已经训练好的模型，将其保存在 BLSTM 模型 LSTM_Model 表中。

与鉴权有关的数据表实体关系图如图 3.18 所示，其中只有一张表，即客户端口令表，其中字段如表 3.14 所示，id 为一个口令的唯一标识；token 为一字符串，由管理员设置和发放，客户端连接时需要有一合法 token 才可与服务器通信；description 用于保存该口令的一些描述信息；create_time 为该口令的创建时间。

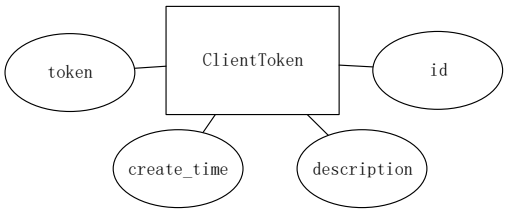

图 3.18: 与鉴权相关表的实体关系图

与程序分析相关的实体关系图如图 3.19，其中，项目表 Project 保存项目名称，函数摘要表 Method_Description 保存一个项目中的函数摘要信息，语句位置表 Location 记录关注点所在的代码位置，污染流表 Taint_Flow 记录污染流和其中的切片信息，标记表 Label 记录一个污染流记录是否安全。一个项目可以有多

表 3.14: Client_Token 表

字段名	数据类型	属性	说明
id	INT	PK	主键，token 的唯一标识
token	VARCHAR(255)	NN, UNIQUE	表示管理员向用户发放的 token
description	VARCHAR(255)	N	用于保存管理员对此 token 的备注信息
create_time	DATETIME	NN	表示 token 的创建时间

个函数摘要、位置和污染流，即有一对多的关系，一个函数摘要或语句位置可以有多个污染流记录，而标记和污染流记录呈一对一关系。

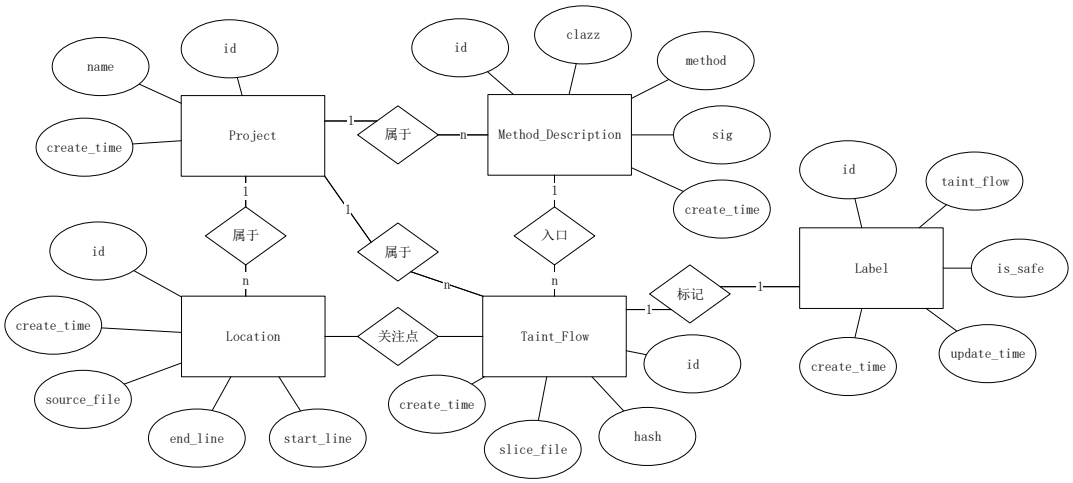

图 3.19: 与程序分析相关表的实体关系图

表 3.15是对项目表的字段说明，其中，id 是一个项目的唯一标识；name 标识项目名称，存在唯一性约束，由客户端建立扫描项目时指定；create_time 记录该项目的创建时间。

表 3.15: Project 表

字段名	数据类型	属性	说明
id	INT	PK	主键，项目的唯一标识
name	VARCHAR(255)	NN,UNIQUE	表示项目的名称
create_time	DATETIME	NN	表示项目创建时间

表 3.16是对函数摘要表的字段说明，其中，id 是一个函数摘要的唯一标识；clazz 记录函数的类名；method 记录函数的方法名；sig 记录函数签名，其包括

参数类型和返回值类型；project 作为外键，标识该函数摘要位于哪个项目中；create_time 记录该函数摘要的创建时间。此外，clazz、method、sig 与 project 能够唯一标识一个函数摘要。

表 3.16: Method_Description 表

字段名	数据类型	属性	说明
id	INT	PK	主键，函数摘要的唯一标识
clazz	VARCHAR(255)	NN	表示该函数的类名
method	VARCHAR(255)	NN	表示该函数的方法名
sig	VARCHAR(255)	NN	表示该函数的签名，即函数参数和返回类型
project	INT	FK, NN	外键，指向 Project，表示该函数出现在此项目中
create_time	DATETIME	NN	表示标记创建时间

表 3.17是对语句位置表的字段说明，其中，id 是一个语句位置的唯一标识；source_file 记录该语句所在的文件名；start_line 记录该位置开始的代码行号；end_line 记录该位置结束的代码行号；project 作为外键，记录该位置位于哪个项目中；create_time 记录该项目的创建时间。此外，source_file、start_line、end_line 与 project 联合可以唯一标识一个语句位置。

表 3.17: Location 表

字段名	数据类型	属性	说明
id	INT	PK	主键，语句位置的唯一标识
source_file	VARCHAR(255)	NN	表示语句所在文件名
start_line	INT	NN, UNSIGNED	表示语句所在的开始行号
end_line	INT	NN, UNSIGNED	表示语句所在的结束行号
project	INT	FK, NN	外键，指向 Project，表示该位置出现在此项目中
create_time	DATETIME	NN	表示标记创建时间

表 3.18 是对污染流表的字段说明，其中，id 是一个污染流的唯一标识；hash 是这个污染流的哈希，通过切片文本的 SHA1 得到；slice_file 切片文件在硬盘上的保存位置，当数据库中记录删除时，切片文件也应被删除；entry 和 point 分别为切片的入口点和关注点；project 作为外键，记录污染流存在于哪个项目中；create_time 记录该污染流的创建时间。此外，entry，point 和 project 可以联合唯一标识一个污染流记录。

表 3.18: Taint_Flow 表

字段名	数据类型	属性	说明
id	INT	PK	主键，污染流的唯一标识
hash	VARCHAR(255)	NN	表示污染流的哈希
slice_file	VARCHAR(255)	NN	表示污染流切片的文件名
entry	INT	FK	外键，指向 Method_Description，表示污染流切片的入口
point	INT	FK	外键，指向 Location，表示污染流切片的结束位置
project	INT	FK, NN	外键，指向 Project，表示污染流所在的项目
create_time	DATETIME	NN	表示污染流的创建时间

表 3.19是对标记表的字段说明，其中，id 是一个标记的唯一标识；taint_flow 作为外键，指向被标记的污染流对象，同时一个污染流同时只能有一个标记；entry 和 point 分别为切片的入口点和关注点；is_safe 标记该污染流是安全/不安全的；create_time 记录该标记的创建时间；update_time 记录该标记的更新时间。

表 3.19: Label 表

字段名	数据类型	属性	说明
id	INT	PK	主键，标记的唯一标识
taint_flow	INT	FK, NN	外键，指向 Taint_Flow，表示本标记的污染流对象
is_safe	BOOLEAN	NN	表示这个污染流是安全的
create_time	DATETIME	NN	表示标记创建时间
update_time	DATETIME	NN	表示标记更新时间

与模型相关的实体关系图如图 3.20 所示，其中，模型配置 Model_Config 表记录模型配置信息，主要包括配置名，模型训练相关信息和 LSTM 模型的参数等，LSTM 模型表 LSTM_Model 记录已训练好的模型信息，主要信息有训练度量，模型保存位置等，一个配置可以训练多个模型。

表 3.20 是对模型配置表的字段说明，其中 name 作为主键，用于唯一标识一个配置，训练时通过指定该值选择训练配置；slice_dir 和 label_dir 记录学习用的数据集位置，分别表示切片所在文件夹和标记所在文件夹（模型直接通过文件进行读取，而不从数据库读取，减少 IO 消耗）；embedding_dim、hidden_dim 为模型自身参数，分别代表词嵌入的维度和隐藏层神经元个数（模型的每层神经元个数相等）；word_freq_gt 表示将出现次数小于该值的单词表示为“UNK”（Unknown）；

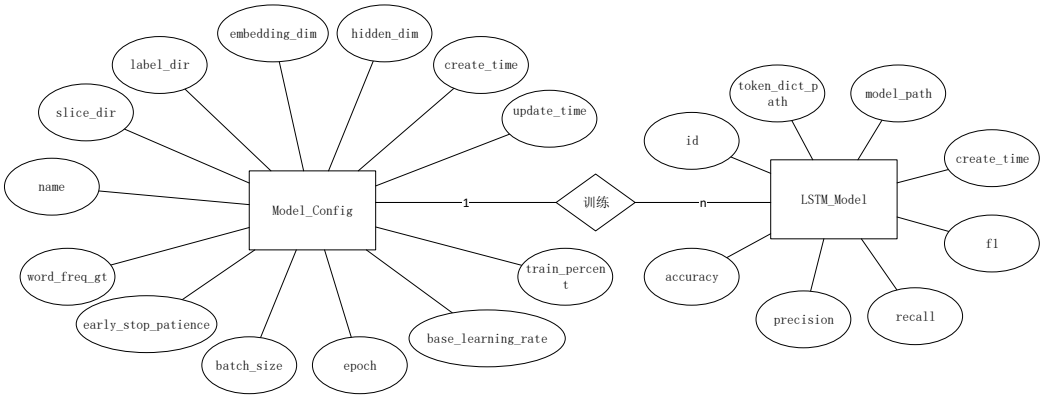

图 3.20: 与模型相关表的实体关系图

early_stop_patience、base_learning_rate、batch_size、epoch 和 train_percent 记录模型训练的配置，early_stop_patience 指训练轮数超过该值后，若模型效果（loss）仍比之前最优效果差，则提前停止训练，base_learning_rate 指初始学习率，batch_size 指批量传入模型的数据量，epoch 指最大训练轮数，train_percent 指训练集占总数据集的百分比，剩下数据作为训练集，用于评估训练效果；最后，create_time 和 update_time 分别表示配置创建时间和配置更新时间。

表 3.20: Model_Config 表

字段名	数据类型	属性	说明
name	VARCHAR(255)	PK	主键，唯一标识一个配置
slice_dir	VARCHAR(255)	NN	模型使用切片数据所在的文件夹
label_dir	VARCHAR(255)	NN	模型使用标记数据所在的文件夹
embedding_dim	INT	NN, UNSIGNED	词嵌入时的维度，值为正整数
hidden_dim	INT	NN, UNSIGNED	隐藏层神经元的个数，值为正整数
word_freq_gt	INT	NN, UNSIGNED	最小词频，小于该词频则表示为 UNK，值为正整数
early_stop_patience	INT	NN, UNSIGNED	提前停止忍耐度，如果学习轮数大于此值，效果仍比以前差则学习停止，值为正整数
base_learning_rate	DOUBLE	NN, UNSIGNED	初始学习率
batch_size	INT	NN, UNSIGNED	批量数据记录数，每次向神经网络输入 batch_size 条记录，再进行反向传播
epoch	INT	NN, UNSIGNED	训练最大迭代次数，超过该次数则停止训练
train_percent	DOUBLE	NN, UNSIGNED	表示训练集占整个数据的百分比，默认为 1，即将所有数据代入训练
create_time	INT	NN, UNSIGNED	配置创建时间
update_time	INT	NN, UNSIGNED	配置更新时间

表 3.21 是对 BLSTM 模型表的字段说明，其中 id 作为主键唯一标识一个模型；token_dict_path 和 model_path 分别表示单词到整形值的映射字典文件位置和 BLSTM 模型位置，这两个字段用于加载模型实现对切片的预测；config 表示模型是由何配置训练得到的；accuracy、precision、recall 和 F1 表示模型效果的准确率、精确率、召回率和 F1 度量，当配置划分没有训练集时，这些值为-1；create_time 记录模型被训练出的时间。

表 3.21: BLSTM_Model 表

字段名	数据类型	属性	说明
id	INTEGER	PK	主键，模型的唯一标识
token_dict_path	VARCHAR(255)	NN	保存单词到 tokenid 的字典文件位置
model_path	VARCHAR(255)	NN	保存 BLSTM 模型文件的位置
config	VARCHAR(255)	FK,NN	指向 Model_Config 表，表示模型对应的配置信息
accuracy	DOUBLE	NN	表示学习后训练集的准确率，若无训练集则为-1
precision	DOUBLE	NN	表示学习后训练集的精确率，若无训练集则为-1
recall	DOUBLE	NN	表示学习后训练集的召回率，若无训练集则为-1
F1	DOUBLE	NN	表示学习后训练集的 F1，若无训练集则为-1
create_time	DATETIME	NN	记录模型创建时间

3.9　本章小结

本章主要介绍了基于污点分析、程序切片和 BLSTM 的 Java 静态代码扫描系统的需求分析和系统设计。首先本章分析系统的功能性需求和非功能性需求，并为此设计系统用例，随后结合 4+1 视图说明系统的总体设计，接着按污点分析模块、程序切片模块、数据预处理模块和预测模块的顺序介绍系统的各个模块设计；最后介绍本系统的数据库设计。

第四章　Java 静态安全扫描系统实现和测试

4.1　一个 XSS 漏洞实例

为了说明扫描系统实现，本小节将首先介绍一段简单的 XSS 漏洞 Java Web 代码，在接下来几个小节中都将使用该代码辅助说明模块实现。

```
1 public class XSS extends HttpServlet {
2     @Override
3     public void doGet(HttpServletRequest request, HttpServletResponse
          response)
4         throws IOException {
5         String p = request.getParameter("p");
6         String q;
7         if (p.startsWith("safe")) {
8             q=safe(p);
9             sink(response, q);
10        } else {
11            q=unsafe(p);
12            sink(response, q);
13        }
14    }
15    @Override
16    public void doPost(HttpServletRequest request, HttpServletResponse
          response)
17        throws IOException {
18        String p = request.getParameter("p");
19        sink(response, safe(p));
20    }
21    public String safe(String s) {
22        String r = s.replace("&", "&amp;").replace("<", "&lt;")
23            .replace(">", "&gt;").replace(" ", " ")
24            .replace("\"", "&#34;").replace("'", "&#39;");
25        return r;
26    }
27    public String unsafe(String s) {
28        if (s.equals("clean")){
29            return "clean";
30        }  else {
31            return s;
32        }
33    }
34    public static void sink(HttpServletResponse response, String s)
35            throws IOException {
36            response.getWriter().write(s);
37    }
38 }
```

图 4.1: 一段含 XSS 漏洞的 Java 代码

代码如图 4.1 所示，类 XSS 继承 HttpServlet，因此其可以通过重写 *doGet()* 和 *doPost()* 方法对外提供 HTTP 服务，在 *doGET()* 方法中，其从用户输出参数 p 中获取值，接着判断值，若值以“safe”开头，则 q 被赋值为 safe(p)，否则 q 被赋值为 unsafe(p)，最终通过 *sink()* 方法向用户返回 *q*，在 *doPOST()* 方法中，其直接调用 *safe()* 方法，并将其返回。分析 *safe()* 和 *unsafe()* 方法不难发现，*safe()* 方法

是一个典型的过滤方法，而 *unsafe()* 方法则存在安全性问题，这也造成 *doGet()* 方法存在安全性问题，而 *doPost()* 不存在安全问题。

4.2　污点分析模块的实现

4.2.1　记录污点传播信息实现

在污点分析模块的设计章节中，提到若要输出带有污点传播树信息的漏洞实例报告，必须在污点分析函数时记录函数内部信息，具体来说，不仅要记录与污点传播相关函数调用的指令位置，还需要记录函数内返回污点的返回语句位置，以及函数调用语句位置。本小节将结合伪代码和具体示例，说明这一部分实现。

Algorithm 1: 记录污点传播信息算法实现

Input: $method$，函数上下文信息
Output: R, 返回语句位置集合；I, 函数调用指令位置集合；T, 与污点传播相关函数调用指令集合

```
1  mDesc ← getDesc(method)
2  R ← ∅
3  I ← ∅
4  T ← ∅
5  initTaint ← −1
6  dataflow ← getDataflow(method);
7  foreach block in dataflow do
8      foreach location in block do
9          inst ← location.getInst()
10         if initTaint = -1 then
11             initTaint ← dataflow.getFact(location).getNum()
12         if inst is ReturnInst then
13             currTaint ← dataflow.getFact(location).getNum();
14             if currTaint >initTaint then
15                 edge ← buildEdge(location, inst)
16                 R ← R ∪ {⟨mDesc, edge⟩}
17         else if inst is InvokeInst then
18             callee ← inst.getCallee();
19             I ← R ∪ {⟨location, callee⟩}
20             analyzeLocation(method, location, inst)
21             if sink influenced by invoke then
22                 T ← T ∪ {location}
```

伪代码 1 说明了函数内记录传播信息的实现，该算法以一个函数上下文信

息 method 为输入，输出函数内返回指令位置集合 R、调用指令位置集合 I 和与污点传播相关语句位置集合 T 。显然，实际该算法实现在 *AbstractTaintDetector.analyzeMethod()* 方法中，并且 R 和 I 实际上可以用 Map 类型记录，在该函数内 R、T 和 S 会加入到类中对应全局集合中。

1∼6 行为算法初始化阶段，包括获取函数的函数摘要，初始化算法输出集合 R、I 和 T，将函数初始被污染的参数个数 $initTaint$ 设为 -1（表示初始值还未设置），以及获取当前函数的数据流图。随后算法通过两层遍历，按数据流图顺序遍历每一个代码块中的每一条语句位置，在循环体中，首先根据语句位置获取语句内容，当 $initTaint$ 未设置值时，将其赋值为获取第一条语句时的污染个数，通常该值小于等于该函数的参数个数（污点传播到一个或多个函数实参上）。

若当前语句为返回值调用语句，由于一个函数内可能有若干条返回语句，并且返回语句未必会返回可能携带污点的变量，不传播污点的返回语句不需要记录在 R 中，例如在代码片段 4.1 中，函数 *unsafe()* 就存在两条返回语句，然而，第 29 行的返回语句并不可能传播污点，因此 29 行并不需要记录。判断语句是否传播污点的方法是判断当前语句的传播的污点数量是否比刚进入时的污点数量多（根据 Find Security Bugs 污点分析方法，即使函数原样返回实参，污点个数也会比初始值多 1）。

若语句为调用语句，首先通过该语句获取被调用的函数摘要，将当前位置和函数摘要一并放入 I 中，不难看出 I 实际上记录了该类的函数调用图，再做函数调用间分析，该分析已由原生 Find Security Bugs 实现，这里不再赘述，根据分析结果判断汇聚点是否可以被当前函数调用影响，若影响则向 T 中加入这一位置，以代码片段为例，当分析到第 8 行时，因为先前分析过 *safe()* 可以传播污点（函数分析按调用图的拓扑排序进行），即 p 污点传播到了 q，又因为污点 *q* 作为参数流入了汇聚点 *sink()*，即根据污点传播结果发现汇聚点受到这次调用的影响，便将第 8 行加入 T 中，同理，被加入 T 的还有第 5 行（污点入口点）、第 9 行、第 11 行和第 12 行。

4.2.2　构造污点传播图实现

在得到返回语句位置集合、调用语句位置集合和污点传播相关语句位置集合后，可以构造污点传播图，其构造方式如图 4.2 所示，实际存在于 *InjectionSink.genBugInstance()* 方法中。

首先，代码构造了一个空污点传播图，将直接调用敏感函数的边加入图中，以图 4.1 中的代码为例，将 $sink() \rightarrow Writer.write()$ 加入图中；接着向图中添

```
1 TaintFlowGraph taintFlowGraph = new TaintFlowGraph();
2 // 加入调用敏感函数的边
3 AbstractInjectionDetector injDetector = (AbstractInjectionDetector)
      detector;
4 taintFlowGraph.addEdge(sinkCaller,
5        new Edge(sinkMethod,
6            SourceLineAnnotation.fromVisitedInstruction(classContext,
7                method,
8                instructionHandle),
9            Edge.SINK_EDGE,
10           0));
11 // 增加函数调用位置
12 for (Location location : locations) {
13    MethodDescriptor caller = location.getMethodDescriptor();
14    MethodDescriptor callee = injDetector.loc2callee.get(location);
15    SourceLineAnnotation src =
16       SourceLineAnnotation.fromVisitedInstruction(
17              location.getMethodDescriptor(), location.getPosition());
18    taintFlowGraph.addEdge(caller, callee, src, key.getOrderedId());
19 }
20 // 附加返回语句位置
21 for(MethodDescriptor method : injDetector.returns.keySet()){
22    for(Edge edge: locations.get(method)){
23       taintFlowGraph.addEdge(method, edge);
24    }
25 }
```

图 4.2: 构造污点传播图的代码片段

加相关函数调用的边，具体地说，遍历污点传播相关语句位置集合，对于每一个 location，获取其调用函数（location 记录了调用函数的函数摘要），再获取其被调用函数（只需查找调用语句位置集合 loc2callee 即可），随后在将 location 转化为 SrcAnnotation，生成一条边并加入图中；最后，只遍历先前得到的返回语句位置集合（injDetector.returns），向图中加入返回语句的点和边（代码 21∼27 行）。

4.2.3　构造污点传播树并生成注解的实现

在得到污点传播图后，对图的每一入口进行深度优先搜索，得到污点传播树集合，搜索算法 2所示，算法实现于 *TaintTreeGenerator.makeTree()* 函数中。

搜索函数即 $dfs()$，其参数有函数 u 、调用 u 的污点传播边（若入口函数则置空）、获取 CFG 的函数以及是否为入口函数，输出该函数 u 的污点传播树集合。

函数初始化返回的树集合 trees 为空，由于传播树的代表了污点传播的一条路径，因此根据函数 u 的 CFG 遍历出所有程序执行路径，即 *getCFG()* 函数，在该函数中，还需根据函数性质去除不符合要求的路径，如在污点入口函数处，路径上的第一个节点必须为污点入口点。

遍历每一条执行路径，接着建立当前路径上的传播树集合 pathTrees，初始化为空，并且将当前函数入口 u 所在节点作为树根加入集合中，用 i 表示路径序号。遍历路径上出现的污点传播边，若边的对端顶点是一个函数类型点，则

Algorithm 2: 构造污点传播树伪代码实现

```
Input: v，污点传播图的一个入口，getCFG()，获取当前函数的 CFG
Output: trees, 污点传播树根的集合
1  Function dfs(u: MethodVertex, e: Edge, isSource: bool): Set<TreeNode>
2      trees ← ∅
3      paths ← getPaths(u, u.edge, getCFG(u.caller), isSource)
4      foreach path in paths do
5          pathTrees ← ∅
6          pathTrees ← pathTrees ∪ {node(u, e))}
7          i ← 0
8          foreach edge in path do
9              if edge.peer is MethodVertex then
10                 subTrees ← dfs(edge.peer, edge, false)
11                 newPathTrees ← ∅
12                 for tree in pathTrees do
13                     for subTree in subTrees do
14                         newTree ← tree
15                         addNode(newTree, subtrees, i)
16                         newPathTrees ← newPathTrees ∪ {newTree}
17                 pathTrees ← newPathTrees
18             else
19                 for tree in pathTrees do
20                     leaf ← node(edge.peer, edge)
21                     addNode(tree, leaf, i)
22             i ← i + 1
23         trees ← trees ∪ pathTrees
24     return trees
25 trees ← dfs(v, null, true)
```

新建一个传播树集合 newPathTrees 并递归获取其子传播树集合，下面两层循环遍历当前传播树集合和被调函数的子传播树集合，产生新的传播树，将其加入 newPathTrees 中，最后用 newPathTrees 集合代替原先集合（第 12-20 行）。这一列操作本质在于，pathTrees 中树个数代表了当前路径的路径数 n，而 subTrees 中树的个数代表了被调函数内有的路径树 m，那么执行完被调函数后，总路径数应该是 $n \times m$，即 newPathTrees 中树的个数。若边的对端顶点是一个返回值顶点，则直接生成叶子节点并将其加入树中。$addNode(tree, subtree, i)$ 函数用于将子树加入主树的第一层树干中，若 $i = 0$ 将其加入至孩子节点，否则将其加入至原最后一个兄弟节点后。

对于入口函数 v（注意，该入口函数不是第二章介绍的污点入口点，而是

图上没有入度的点，实际上的污点入口点发生在 v 节点代表的函数中），调用 $dfs(v, null, true)$ 即可返回函数入口的多个污点传播树。

由上述算法可知，因为一个入口点到达汇聚点的路径不同，一个入口可以有多个污点传播树，例如代码 4.1 中 *doGet()* 函数，搜索后，可以得到如图 4.3 所示的两个污点传播树。

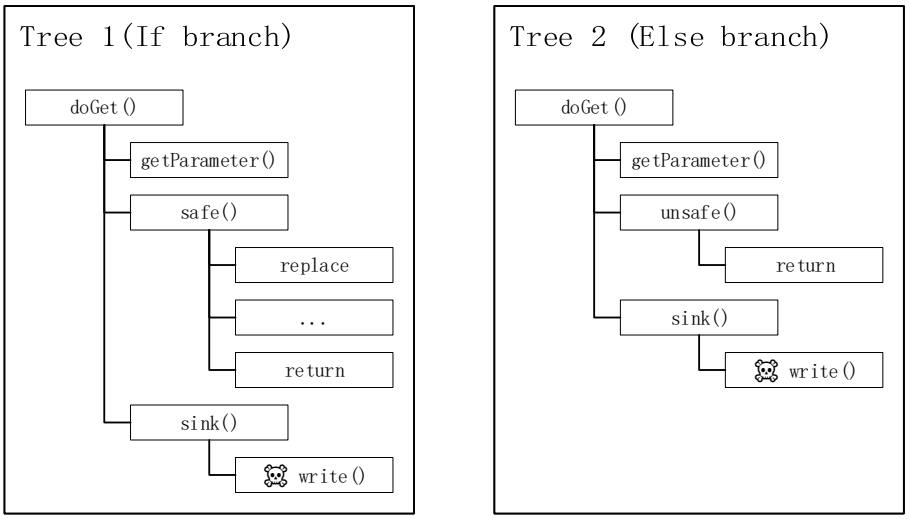

图 4.3: *doGet()* 函数的污点传播树

真实情况下，一个程序中可能存在大量 if-else 语句，可能会导致路径爆炸的问题，因此系统在这里对算法进行了限制，当入口函数的传播树个数大于 300 时，将不再获取子传播树集合，而是直接获取一棵可行的传播树。

得到污点传播树后，对传播树再做一次深度搜索可以将树上的所有节点转化为可以显示在界面上的注解，其算法表示如算法 3 所示。

对于污点传播树的搜索比传播图要简单的多，只要先将当前节点构造成一条注解，再搜索其孩子节点和兄弟节点即可，注意兄弟孩子节点可能为空，因此在搜索前需要判断，若为空值则返回空集，此外，通过算法 2 构造的树节点实际上是存在回路的，在打印时需要考虑到这点，因此增加了 book 参数记录，限制当前节点不能循环搜索。在一开始调用时，只需要将 depth 设为 0，book 和 A 设为空集即可。

对于污点传播图的每一个入口函数，可以构造一至多棵污点传播树，对应的就有一至多组注解集合。

Algorithm 3: 构造污点传播树注解伪代码实现

Input: n，污点传播树树根；$depth$，树节点深度，初始化为 0；$book$ 以处理的位置，初始化为 ∅；$annotations$，当前注释集合，初始化为 ∅

Output: R, 污点传播树的注解集合

```
1 Function dfs(n:TreeNode, depth: int, book: Set<SrcAnnotation>, annotations: Set<SrcAnnotation>) :
    Set<SrcAnnotation>
2     if n is null then
3         return ∅
4     if n in book then
5         return ∅
6     R ← annotations ∪ {getAnnotation(n)}
7     R ← R ∪ dfs(n.firstChild, depth + 1, book ∪ {n.location}, annotations)
8     R ← R ∪ dfs(n.nextSib, depth, book ∪ {n.location}, A)
9     return R
```

4.3　程序切片模块的实现

在设计章节，本文已经对该模块的功能、主要流程和重要类图进行了介绍，在本节中，将从整个流程的控制模块和后向切片两方面介绍该模块的具体实现。

4.3.1　切片控制模块实现

4.3.1.1　切片控制代码

切片控制模块的实现代码片段如图 4.4 所示，该代码出现在 SliceRunner 的 *run()* 方法中，主要功能是调用翻译器和切片器对一个目标项目中所有漏洞切片，即执行流程图 3.11 上的流程。

代码第 1 行首先调用 reportParesr 将项目进行过滤和翻译，得到 taintProject，接着第 3 行通过项目中应用 Jar 包和依赖 Jar 包，对 slicer 进行配置。接着遍历污点传播类型的漏洞实例，对于每一个实例，获取其污点传播树集合（第 11 行），接着遍历污点传播树，对每一个污点传播树进行切片（第 19 行），每完成一个漏洞实例后向监视器汇报进度（第 24-25 行），并且若是 Joana 类型的切片器，对切片器进行清除缓存操作，防止缓存过多造成 OOM，最后返回切片项目。

4.3.1.2　过滤漏洞实例

本系统为实现切片预测逻辑和污点传播逻辑的解耦，产生了翻译漏洞报告这一过程。在 Spotbugs 报告类型的翻译器中，主要逻辑在于对漏洞过滤并还原污点传播树。对于污点传播树的还原可以通过对漏洞注解二次遍历实现，这里不再赘述，本节主要通过代码 4.5，介绍过滤漏洞实例实现。

```
taintProject = reportParser.rep2taintProject(monitor);
try {
    slicer.config(taintProject.getAppJars(), taintProject.getLibJars(), null
        );
    monitor.process(1, 1, null, null, null);
} catch (ClassHierarchyException | IOException e) {
    monitor.process(1, 1, null, null, e);
}
monitor.init("Slice", taintProject.getTaintTreeMap().size());
project = new SliceProject<>(taintProject);
for (int i = 0; i < taintProject.getBugInstances().size(); i++) {
    List<TreeNode> taintTrees = taintProject.getTaintTrees(
           taintProject.getBugInstances().get(i));
    if (taintTrees == null) {
        continue;
    }
    String slice = null;
    for (TreeNode node : taintTrees) {
        try {
            sliceTaintTree(node);
        } catch (SlicerException e) {
            LOGGER.error(ExceptionUtil.getStackTrace(e.getRawException()));
        }
    }
    monitor.process(i + 1, taintProject.getBugInstances().size(),
           taintTrees, slice, exception);
    if (slicer instanceof JoanaSlicer){
        ((JoanaSlicer) slicer).clearCache();
    }
}
return project;
```

图 4.4: 切片控制模块实现代码片段

```
Collection<BugInstance> c = bugCollection.getCollection();
List<BugInstance> bugInstances = c.stream()
    .filter(e -> caredVulns.contains(e.getType()) && (!e.isDead())
           && e.getPriority() != Priorities.LOW_PRIORITY)
    .filter(e -> {
        for (BugAnnotation annotation : e.getAnnotations()) {
            if (annotation.toString().equals("Method usage not detected")) {
                return false;
            }
        }
        return true;
    })
.collect(Collectors.toList());
return bugInstances;
```

图 4.5: 漏洞过滤代码片段

从第 3∼4 行可以看出，模块按漏洞类型过滤掉了非污点传播类型漏洞、已经修复的漏洞（在 Spotbugs 中，已经修复的漏洞 *isDead()* 函数返回 True），以及漏洞危害等级为低级的漏洞，在 Spotbugs 中，污点传播类漏洞实例的评级往往是根据污点传播情况制定的，而低级漏洞往往不存在污点入口点而空有污点汇聚点，这种情况按照当前漏洞观念已经可以判断为误报，不需要本系统处理，在第 6∼11 行过滤的漏洞其污点汇聚点没有被其他函数调用（相当于代码 4.1 中 *sink()* 函数没有被调用），这种漏洞实例也可以直接判断为误报。

在按类型过滤漏洞时，使用了白名单过滤方式，目前模块内的白名单如代

```
1 public static List<String> caredVulns = Arrays.asList(
2        "COMMAND_INJECTION",
3        "XSS_JSP_PRINT", "XSS_REQUEST_WRAPPER", "XSS_SERVLET",
4        "URLCONNECTION_SSRF_FD",
5        "SQL_INJECTION_HIBERNATE", "SQL_INJECTION_JDO", "SQL_INJECTION_JPA",
6        "SQL_INJECTION_JDBC", "SQL_INJECTION_SPRING_JDBC",
7        "SQL_INJECTION_TURBINE",
8        "UNVALIDATED_REDIRECT",
9        "LDAP_INJECTION",
10       "XPATH_INJECTION");
```

图 4.6: 污点传播类漏洞

码 4.6所示，列表中覆盖绝大多数 Web 安全问题，如 XSS、SQL 注入和 SSRF 等漏洞。

4.3.1.3　分解污点传播树

在代码 4.4 中，其使用 sliceTaintTree(node) 对一个污点传播树进行切片，该函数具体实现在代码 4.7 中。

```
1 public void sliceTaintTree(TaintTreeNode source) throws SlicerException {
2     Set<TaintFlow> flows = new DFSTaintTree().getTaintFlows(source);
3     project.source2taintFlow.put(source, flows);
4     for (TaintFlow edge : flows) {
5         sliceFlow(edge);
6     }
7 }
8 public String sliceFlow(Func func, Location location) throws
       SlicerException {
9     String slice = project.getSlice(func, location);
10    if (slice != null) {
11        return slice;
12    } else {
13        slice = slicer.computeSlice(func, location);
14        project.putSlice(func, location, slice);
15        return slice;
16    }
17 }
```

图 4.7: 污点传播树切片代码片段

在代码中可以看出，对于待切片的污点传播树，首先将其拆解为污染流集合（第 2 行），再对每一污染流进行切片，将传播树与污染流的映射关系，以及污染流和切片的对应关系放入切片项目中。其中关键点是将污点传播树分解为污染流。

算法 4 描述了分解过程，算法输入为一个传播树的树根，输出为污染流的集合，首先将初始化一个空污染流集合，接着通过深度优先搜索得到所有污染流。在搜索过程中，遍历当前树节点下的所有孩子节点，如果节点是函数节点，首先检查其是否为汇聚点函数，如果是，则将树根到当前位置作为一个污染流放入集合中，同时返回 $true$ 表示函数上下文中有汇聚点，如果不是，则递归向

Algorithm 4: 从污点传播树中分解污染流集合

```
Input: r: 污点传播树树根
Output: flows, 污染流的集合
1  flows ← ∅
2  dfs(r)
3  Function dfs(s: TreeNode): bool
4      nextNode ← s.firstChild
5      while nextNode != null do
6          if nextNode is MethodTreeNode then
7              if nextNode.type is Edge.SINK then
8                  flows ← flows ∪ {Flow(s.caller, n.location)}
9                  return true
10             hasSink ← hasSink ∨ dfs(nextNode)
11             if hasSink then
12                 flows ← flows ∪ {Flow(s.caller, n.location)}
13                 return hasSink
14         else
15             flows ← flows ∪ {Flow(s.caller, n.location)}
16             return hasSink
17         nextNode ← nextNode.nextSib
```

下搜索，若搜索中递归函数返回 $true$，将树根到当前节点位置放入集合中。若节点是返回值节点，则直接将从树根到当前节点的作为污染流放入集合中。

从算法可以看出，污染流实际上是传播树上涉及的每一个函数和其对应的关注点二元组，这里的关注点可能是返回语句，或是调用汇聚点函数的语句，例如，对于图 4.3 中的 Tree 1，可以得到 $doGet() \rightarrow sink()$、$safe() \rightarrow return$ 和 $sink() \rightarrow write()$ 污染流。

4.3.2　后向切片的实现

4.3.2.1　基于 Joana 的后向切片

对于得到的污染流对象，需要由切片器对其后向切片，本系统默认使用 Joana 对其切片，其主要过程如图 4.8 所示，该代码实现于 *JoanaSlicer.computeSlice()* 方法中。

代码首先要获取函数 func 的 SDG，若该函数在缓存中存在，则直接拿出（第 3 行），若不存在，则需要根据配置和入口生成 SDG（第 5~16 行）。接着使用 SDG 初始化一个 JoanaSDGSlicer，获取关注点行号上的所有 SDGNode 并且

```
SDG sdg = null;
if (sdgCache.containsKey(func)) {
    sdg = sdgCache.get(func);
} else {
    try {
        String entryClass = "L" + func.className.replace('.', '/');
        String entryMethod = func.methodName;
        String entrySig = func.methodSig;
        config.entry = findMethod(config, entryClass, entryMethod, entrySig)
            ;
        sdg = SDGBuilder.build(config, new SliceMonitor());
        CSDGPreprocessor.preprocessSDG(sdg);
        sdgCache.put(func, sdg);
        LOGGER.info("Computing Slice...");
    } catch (Exception e) {
        throw new SlicerException(e.getMessage(), e);
    }
}
JoanaSDGSlicer jSlicer = new JoanaSDGSlicer(sdg);
HashSet<SDGNode> sinkNodes = jSlicer.getNodesAtLocation(line);
Collection<SDGNode> slice = jSlicer.slice(sinkNodes);
List<SDGNode> sortedSlice = new ArrayList<>(slice);
sortedSlice.sort(Comparator.comparingInt(SDGNode::getId));
StringBuilder result = new StringBuilder();
for (SDGNode node : sortedSlice) {
    result.append(node+" :: "+node.getKind()+" :: "+node.getOperation()
           +" :: "+node.getType()+" :: "+node.getLabel()+"\n");
}
String sliceStr = result.toString();
```

图 4.8: 基于 Joana 后向切片的代码片段

对其切片。切片完成后，对其按行号进行排序——这主要保证了每一次对相同污染流的切片输出一致性。最后将其格式化为字符串。

在 *JoanaSDGSlicer.slice()* 函数主要实现利用 Joana 包中的 SummarySlicer-Backward 类对程序进行切片，并对切片的 SDGNode 集合进行过滤。对于需要移除的 SDNode 类型由 isRemovable() 实现，如代码 4.9 所示。从代码中可以看到，如果 SDGNode 没有对应源代码、不存在于用户 Jar 包中、源代码行号小于 0、代表异常节点、代表虚节点或抽象节点则将其移除。

4.3.2.2　SDG 的生成配置类

切片需要通过配置生成 SDG，然而默认的 Joana 配置并不具备缺失依赖下生成 SDG 的功能，为此本系统基于最新版本 WALA，重新编译了 Joana，添加自定义配置，缺失依赖的 SDG 生成等功能；当程序规模过大时，生成 SDG 时会消耗过多资源，针对本系统的切片使用场景，本模块切片器实现了限制调用图的切片，这同样是依赖配置实现的，另外，对于异形目录结构的 Jar 包（如 Spring Web 项目 Jar 包）默认配置会存在找不到类的问题，这依旧依赖配置实现，因此本节将详细介绍这些配置类，这些类的类图如图 4.10 所示。

JoanaOutputStream 类用于打印日志，其继承 Java 原生的 OutputStream 类，*write()* 函数用来向控制台打印日志信息，其实例保存于配置的 out 属性中。

```
boolean isRemoveNode(final SDGNode node) {
    return node.getSource() == null || (node.getClassLoader() != null &&
        node.getClassLoader().equals("Primordial"))
           || node.getSr() < 0 || isExcluded(node.getSource()) || node.
               getLabel().contains("_exception_")
           || node.getLabel().contains("fake")
           || (node.getBytecodeIndex() < -2 && node.getOperation() !=
               SDGNode.Operation.ASSIGN) || isAbstractNode(node) ;
}

boolean isExcluded(final String str) {
    return str.contains("java/lang") || str.contains("java/io") || str.
        contains("java/util")
           || str.contains("com/ibm/wala") || str.contains("sun/reflect") ||
                str.contains("java/security")
           || str.contains("sun/") || str.contains("javax/servlet");
}

boolean isAbstractNode(SDGNode sdgNode) {
    return sdgNode.getKind().equals(Kind.FORMAL_IN) || sdgNode.getKind().
        equals(Kind.FORMAL_OUT)
           || sdgNode.getKind().equals(Kind.ACTUAL_IN) || sdgNode.getKind().
               equals(Kind.ACTUAL_OUT)
           || sdgNode.getLabel().equals("many2many") || sdgNode.getLabel().
               contains("UNIQ(")
           || sdgNode.getLabel().contains("<init>") || sdgNode.getLabel().
               equals("immutable");
}
```

图 4.9: 判断 SDGNode 是否需要移除的代码片段

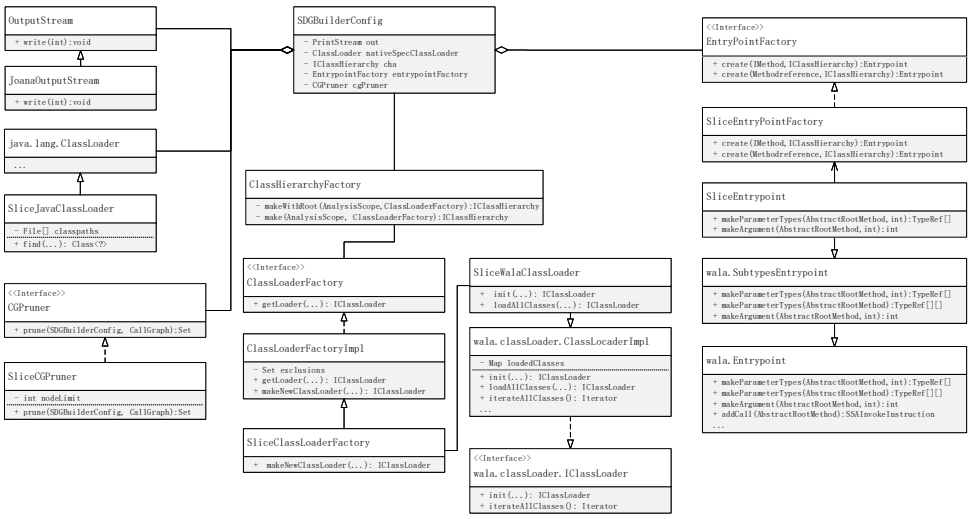

图 4.10: 生成 SDG 的配置类图

SliceJavaClassLoader、SliceClassLoaderFactory 和 SliceWalaClassLoader 用于解决异形 Jar 包找不到类的问题，例如，Springboot 的项目 Jar 包中，类文件的地址前缀为“BOOT-INF/classes/⟨package⟩.⟨class⟩.class”。对于 Joana 切片来说，其首先通过 ClassHierarchyFactory 工厂类获取类层次图（IClassHierarchy），在之后分析中，只分析类层次图中存在的类。工厂类（ClassHierarchyFactory）中函数

makeWithRoot() 和 *make()* 为构建类层次图的函数（它们区别在于 *makeWithRoot()* 支持缺失依赖的层次图生成，这将在后文详细介绍），函数第一个参数是分析范围，该对象可以通过传入的 Jar 包路径创建，函数执行后，其会扫描 Jar 包中所有.class 文件作为类文件，并将其路径转化为包名和类名，如：从“a/b/XSS.class”得到类“a.b.XSS”，而第二个参数为 Jar 包的类加载器工厂类（ClassLoaderFactory），ClassLoaderFactory 实际上为一接口，其 *getLoader()* 方法用于返回一个类加载器（IClassLoader）实例，注意这里的类加载器不等同于 Java 原生的 Classloader。

类加载器 IClassLoader 为一接口，其中定义了 *iterateAllClasses()* 方法用于遍历分析范围类的所有类，为了解决类查找问题，首先需要保证该方法能够正确返回 Jar 包中所有类名。Joana 默认的（准确的说是 Joana 借助 WALA 的）类加载器为 ClassLoaderImpl，其 *init()* 方法通过 *loadAllClasses()* 将 Jar 中的所有类读取至缓存，当调用 *iterateAllClasses()* 时，只需遍历缓存中的类即可。

```
1  private void loadAllClasses(
2          Collection<ModuleEntry> moduleEntries,
3          Map<String, Object> fileContents,
4          boolean isJMODType) {
5      for (ModuleEntry entry : moduleEntries) {
6          /*...*/
7          String className = entry.getClassName().replace('.', '/');
8          /*...*/
9          if (className.contains("/classes/")) {
10             className = className.replaceFirst(".*?/classes/", "");
11         }
12         if (myExclusions != null && myExclusions.contains(className)) {
13             continue;
14         }
15         ShrikeClassReaderHandle entryReader = new ShrikeClassReaderHandle(
               entry);
16         className = 'L' + className;
17         try {
18             // 使用classloader装载该类
19             TypeName T = TypeName.string2TypeName(className);
20             if (loadedClasses.get(T) != null) {
21                 Warnings.add(MultipleImplementationsWarning.create(className)
                       );
22             } else if (super.getParent() != null && super.getParent().
                   lookupClass(T) != null) {
23                 Warnings.add(MultipleImplementationsWarning.create(className)
                       );
24             } else {
25                 ShrikeClass tmpKlass = new ShrikeClass(reader, this, cha);
26                 loadedClasses.put(T, new ShrikeClass(entryReader, this, cha))
                       ;
27             }
28         } catch (InvalidClassFileException e) { /*...*/ }
29     }
30 }
```

图 4.11: SliceWalaClassLoader 的代码实现

根据以上分析，本模块继承 ClassLoaderImpl 并重写 *loadAllClasses()* 方法，得到 SliceWalaClassLoader 类，并且构造对应的工厂类 SliceClassLoaderFactory。SliceWalaClassLoader 的部分实现如代码 4.11 所示，可以看到代码第 9∼10 行，当

包名中存在“/classes/”目录时，去除“/classes/”及前部分字符，只留下包名和类名。

接着模块需要解决类实际加载的问题，模块通过自定义 Java 原生 ClassLoader（SliceJavaClassLoader）解决该问题，该类实现代码如代码 4.12 所示，其关键在于代码第 9 行，即通过字符串尾部匹配来判断 classpath 下是否包含文件，如果包含则读取该类的内容至比特数组 classByte 中。

```
1  protected Class<?> findClass(final String name) {
2      String targetFile = name.replace(".", "/") + ".class";
3      byte[] classByte = null;
4      for (File file : classpaths) {
5          ZipFile zipFile = new ZipFile(file);
6          Enumeration<?> entries = zipFile.getEntries();
7          while (entries.hasMoreElements()) {
8              ZipEntry entry = (ZipEntry) entries.nextElement();
9              if (entry.getName().endsWith(".class") && entry.getName().
                   endsWith(targetFile)) {
10                 try {
11                     /* read zip entry to classByte */
12                 } catch (IOException e) {
13                     e.printStackTrace();
14                 }
15                 break;
16             }
17         }
18     }
19     if (classByte == null) { throw new ClassNotFoundException(name); }
20     return defineClass(name, classByte, 0, classByte.length);
21 }
```

图 4.12: 自定义 JavaClassLoader 的代码实现

最后将 ClassHierarchyFactory 产生的类层次图配置在 SDGBuilderConfig 实例 cha 属性中。

很多用户提交的 Jar 包并不包含第三方包依赖信息，默认 Joana 无法完成切片，其关键问题在于以下两点：（1）当类继承于其他第三方包中类时，Joana 由于无法找到第三方类而无法构造该类；（2）当函数参数类型为第三方包中类时，Joana 由于无法找到类原型无法分析。

对于第一点，模块重新编译了 Joana 使其依赖于高版本的 WALA，在生成类层次图时，使用 *makeWithRoot()* 方法生成，该方法在生成类层次结构图时，若该类父类为搜索范围之外的类，则将其继承自 Object 类（在 Java 中 Object 类是所有类的父类）。

对于第二点,为此模块在 SliceEntrypoint 类中重新写了 Entrypoint 类的 *makeParameterTypes()* 和 *makeArgument()* 方法， *makeParameterTypes()* 用在函数调用语句发生时，返回函数调用参数类型，这里当调用参数类型为范围外的类时，函数返回一个伪造的类型， *makeArgument()* 用于当函数调用发生时，为实参创建一个实例，并返回实例节点的序号，这里当实参类型为范围外的类时，伪造一个

虚节点加入序号表中，并将其返回。此外，模块实现 SliceEntrypoint 类的工厂类 SliceEntryPointFactory，并将其配置在 SDGBuilderConfig 实例的 entrypointFactory 属性中。

用户上传的实际程序规模不可控，对其进行完整切片很可能造成物理资源耗尽或是程序崩溃，为此，模块在 *NodeLimitPruner.prune()* 中实现了对调用图剪枝来限制切片规模，代码 4.13 描述了其过程。首先，初始化需要保留的 CGNode 集合 keep，向搜索队列中加入抽象函数节点和真实函数节点，向搜索队列中加入真实入口函数节点，接着通过广度优先搜索，向 keep 中加入不超过 nodeLimit 数量的节点，最后返回 keep 即可。

```
1 public Set<CGNode> prune(final SDGBuilder.SDGBuilderConfig cfg, final
      CallGraph cg) {
2     Set<CGNode> keep = new HashSet<>();
3     Set<CGNode> marked = new HashSet<>();
4     Queue<CGNode> queue = new LinkedList<>();
5     CGNode head = cg.getFakeRootNode();
6     keep.add(head);
7     marked.add(head);
8     marked.addAll(cg.getEntrypointNodes());
9     keep.addAll(cg.getEntrypointNodes());
10    queue.addAll(cg.getEntrypointNodes());
11    int limit = nodeLimit + keep.size();
12    while (!queue.isEmpty()) {
13        if (keep.size() >= limit)
14            break;
15        head = queue.poll();
16        keep.add(head);
17        for (Iterator<CGNode> it = cg.getSuccNodes(head); it.hasNext(); ) {
18            CGNode childNode = it.next();
19            if (!marked.contains(childNode)) {
20                marked.add(childNode);
21                if (cfg.pruningPolicy.check(childNode)) {
22                    queue.add(childNode);
23                }
24            }
25        }
26    }
27    return keep;
28 }
```

图 4.13: NodeLimitPruner 的代码实现

下面以代码 4.1中 *doGet()* 函数为例，说明剪枝过程，首先，WALA 会计算出如图 4.14 所示的函数调用图，*fakeRootMethod()* 是 WALA 的一个抽象函数，它是所有调用图的入口，遍历时通过该函数寻找其他函数，接着该函数会连向 *fakeWorldClinit()*——该函数也是 WALA 的抽象函数，WALA 通过该函数调用其他类的构造方法；该函数还会连向 *XSS.<init>()*，它是 XSS 类的构造方法，最后函数连向真正需要进行切片的入口方法 *doGet()*，在 *doGet()* 中由于调用其他函数，又会出现其他方法节点。在剪枝时，首先需要将 *fakeRootMethod()* 方法加入 keep，否则无法进行遍历找到其他方法；接着加入入口方法 *doGet()*，随后按顺序依次加入 *safe()* 和 *unsafe()* 方法，如果限制调用节点数为 4，那么 *sink()* 节点将被剪枝，

即最后保留的函数依次为 *fakeRootMethod()*、*doGet()*、*safe()* 和 *unsafe()*，可以看到在搜索时，代码还忽略了非应用定义的方法，如 *getParameter()*、*startsWith()* 等，这是为了在有限集合内尽可能多的保留程序特性，而省略其共性（因为不管是安全和非安全代码，这些非应用函数内切片内容不变），同时被删去的 *sink()* 函数也并不是被彻底忽略，因为它会在其他子污染流中被计算。

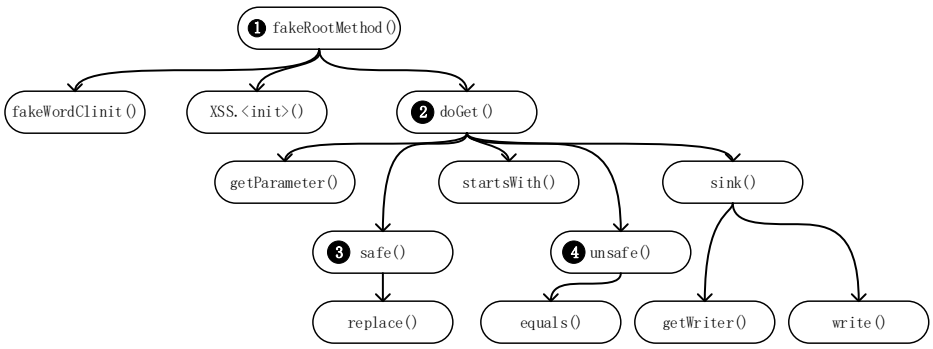

图 4.14: 限制调用图示意图

4.4　数据处理模块的实现

数据处理模块将程序切片进行泛化和向量化处理，以保证神经网络的泛化性能，本节就泛化处理和向量化两方面说明该模块的实现。

4.4.1　泛化处理

在设计阶段，本文已经介绍了模块对程序切片的泛化主要分泛化数字常量、泛化字符串常量、泛化变量、泛化函数调用和泛化包名方法名五个方面，下面以 $unsafe() \rightarrow return(line:31)$ 函数的代码切片 4.15为例，介绍这些泛化的具体含义。

```
1 1 :: ENTR :: entry :: null :: demo.XSS.unsafe(java.lang.String)
2 6 :: CALL :: call :: Z :: v6 = p1 $s .equals(#(clean))
3 11 :: PRED :: IF :: Z :: if (v6 == #(0)) goto 11
4 18 :: NORM :: compound :: Ljava/lang/String :: return v6
```

图 4.15: *unsafe()* 函数的代码切片

首先是泛化数字常量，数字常量在切片中往往带有数据信息，然而程序切片中的数字返回往往变化很大，为了泛化总结出共性，模块将三位数及以上的整数同一替换为“N3P”，将三位数及以上的负整数替换为“NN3P”，将科学计

数法的数字替换为“NSMALL”，剩余三位数及以下数字将其“#()”去除，保留数字字符串，因为系统认为这些数字的具体值有更为重要信息，如在示例切片第 3 行中的“#(0)”，将其替换为“0”。

字符串常量在切片代码中非常常见，其内容也可能是判断一个切片是否是安全的重要标准，如 SSRF 漏洞切片中发现有“http://site”开头的字符串，那么漏洞很可能是误报，或是注入型漏洞出现特殊符号的替换字符串，那么这些漏洞也很可能是误报。因此，对于字符串要做特殊泛化处理，首先，若包含如“HTTP”,“HTTPS”且长度大于其 4 的字符串，将其泛化为“URL”；此外，若字符串长度大于 2，因为这些字符串是清洁字符串的概率较小，将其泛化为“STR S{i}”的形式，i 指该字符串为切片中出现的第 i 个字符串；否则保留其本身，因为其本身很可能是清洁类字符串。例如在第 2 行切片中出现的“#(clean)”，则将其泛化为“STR S1”（其为切片中出现的第一字符串）。

由于切片对象为静态单赋值形式（SSA），以“v{数字}”表示的变量在切片中很常见，但是由于切片原因，这些数字并不是自然增长的，因此在泛化流程中，对其出现顺序进行映射，将其泛化为“VAR V{i}”的形式，其中 i 代表该变量是第 i 个出现在切片中的，例如第 2 行、第 3 行和第 4 行出现的 v6 变量，将其泛化为“VAR V1”，即该变量第 1 个出现在切片中。

函数调用的泛化主要是针对于调用切片和返回，对于调用切片，如第 2 行，将其调用函数前的“.”去除，若有形参（“$”开头的变量）也将其去除，因为形参名称与漏洞本身无关，并在括号前后加空格，当做完经过该步骤后，第 2 行切片的函数调用部分泛化为“VAR V1 = p1 equals (STR S1) ”，若函数调用返回为对象，则将其斜杠形式改为“.”连接的形式，如最后一行中的“Ljava/lang/String”变为“java.lang.String”。

泛化包名方法名，此类泛化主要对于 ENTER 类型的节点，如第 1 行，这里需要将切片拆分为“包名 _ 方法名 _ 函数参数类型”，如将“demo.XSS.unsafe(java.lang.String)”泛化为“demo XSS unsafe (java.lang.String)”。

数字常量、字符串常量和变量的泛化分别在 Preprocessing 类中 *processNumber()*、*processStr()* 和 *processVer()* 中实现，对于函数调用和包名方法名的泛化在 *processType()* 实现。

4.4.2　建立单词表与向量化

在本模块 Tokenizer 类中，泛化后字符串首先生成单词表，再按照单词表可以对句子进行向量化。其实现代码如图 4.16 所示。

```
def build_dict(dataset: TextDataset, freq_gt: int) -> WordTokenDict:
    _dict = WordTokenDict()
    word_freq = {}
    for sentence, label in dataset:
        for word in sentence.split():
            word = word.strip()
            if word in word_freq:
                word_freq[word] += 1
            else:
                word_freq[word] = 1
    for word, freq in word_freq.items():
        if freq > freq_gt:
            _dict.add_word(word)
    return _dict

def encode(string: str, _dict: WordTokenDict) -> [int]:
    words = string.split()
    encoded = map(lambda e: _dict.wtoi(e.strip()), words)
    return list(encoded)
```

图 4.16: 建立单词表和向量化的代码片段

build_dict() 方法输入为切片数据集，方法首先建立一个空的字典 _dict 和词频表 word_freq，接着遍历数据集中的所有切片，按空格拆分，并开始统计词频，遍历结束后，根据词频表，将词频大于 freq_gt 的单词放入单词表，单词进入单词表的顺序将成为其 id，小于等于该词频的单词会被映射为“UNK”，方法最终返回一个单词字典。*encode()* 方法用于将一个切片数据向量化，其只需要将泛化后的单词序列分解为单词，再通过字典将每一个单词映射为 id 即可，方法最终返回为一个句子的向量。

4.5　误报预测模块的实现

4.5.1　误报预测控制

误报预测控制负责执行图 3.17 中的主要流程，其代码如 4.17 所示。代码遍历 sliceProject 中的每一个实例、每个实例中每个污点传播树以及每个传播树中的污染流，从代码 20∼23 行可以看出，只要树中有一条污染流是安全的，那么整个传播树就是安全的，因为污点无法向下传播至汇聚点，对应的安全流会保存到 safeFlows 集合中。从 27 行可以看到，只有当所有的污点传播树都是安全时，一个漏洞实例才可判断为误报，最后，将是否为误报的结果和安全的流放入预测项目中。*flowIsSafe()* 在调用远程预测服务前，会先检查缓存是否已有记录，若存在记录则直接将其返回，否则调用远程服务。

4.5.2　误报预测时序图

误报预测模块存在客户端至服务器调用流程，本文通过时序图 4.18 说明误报预测的调用实现。

```
for (int i = 0; i < sliceProject.getBugInstances().size(); i++) {
    List<Exception> causedExceptions = new LinkedList<>();
    T buginstance = sliceProject.getBugInstance(i);
    List<TaintTreeNode> taintTrees = sliceProject.getTaintProject().
        getTaintTrees(buginstance);
    boolean instanceIsSafe = true;
    Set<TaintFlow> safeFlows=new HashSet<>();
    for (TaintTreeNode source : taintTrees) {
        boolean treeIsSafe = false;
        Set<TaintFlow> flow = sliceProject.getTaintFlows(source);
        if (flow == null) {
            causedExceptions.add(new NotFoundException(source, sliceProject))
                ;
            continue;
        }
        if (flow.isEmpty()){
            treeIsSafe=true;
        } else {
            for (TaintFlow subFlow : flow) {
                boolean bFlowIsSafe = false;
                bFlowIsSafe = flowIsSafe(subFlow);
                if (bFlowIsSafe){
                    treeIsSafe=true;
                    safeFlows.add(subFlow);
                    break;
                }
            }
        }
        instanceIsSafe = instanceIsSafe & treeIsSafe;
    }
    predictProject.putProofs(buginstance, safeFlows);
    predictProject.putPrediction(sliceProject.getBugInstances().get(i),
        instanceIsSafe);
}
```

图 4.17: 误报预测控制逻辑的实现

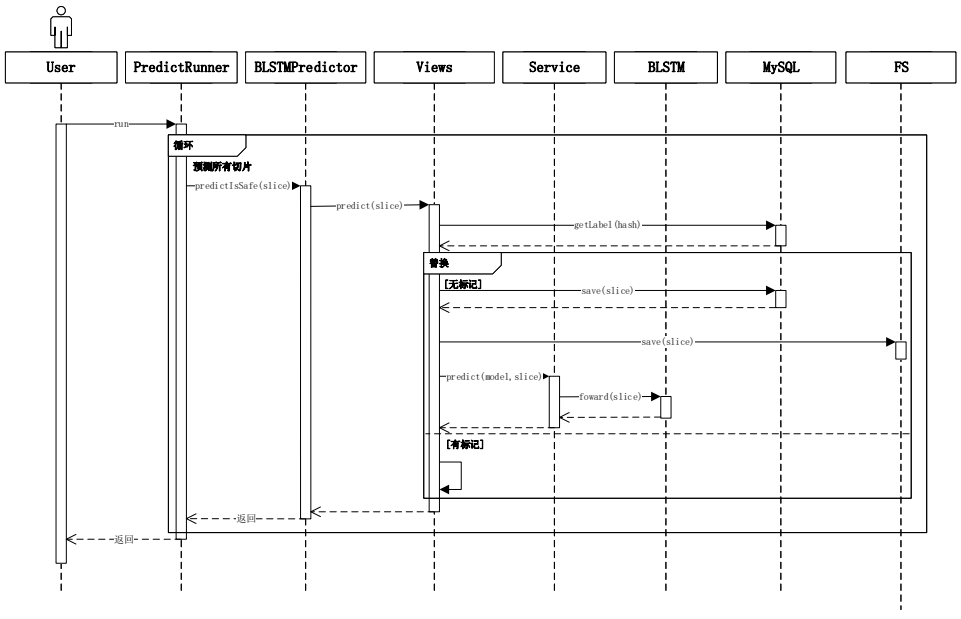

图 4.18: 误报预测时序图

用户首先发起对污点传播报告的预测请求，接着预测控制逻辑进入循环，对于每一个切片调用 BLSTMPredictor 进行预测，BLSTMPredict 为远程服务的本地代理，其会向远程服务器发送预测请求，服务器视图层收到预测请求后，首先从数据库查找该切片是否有标记，如果有标记则优先返回标记数据，否则先将切片依次保存至数据库和本地文件系统中，再向预测服务发起预测请求，预测服务收到请求后发送给 BLSTM 模型，模型将预测结果依次返回至用户。

4.5.3　漏洞标记时序图

本模块的另一重要流程在于对漏洞实例标记，时序图 4.19 展示了漏洞标记的实现过程。

用户可以对一个漏洞进行误报或正报的标记，首先看误报标记流程，对于一个误报漏洞实例，用户需要提供误报原因，即指定污点消失于哪些污染流中，接着 GUI 界面会生成一个 Label 实例并向预测器发送标记为安全污染流的请求，本地预测器会将请求转发到服务端的 View 层，View 层收到请求后将标记数据依次写入数据库和本地文件中，由于污染流哈希的唯一性，当数据库存在已有标签时会自动覆盖。

对于一个正报标记，用户需指定漏洞实例的一个可利用的传播树，因为一个真实的漏洞只要有一棵传播树的污点传播成立，该漏洞就真实存在，GUI 收到请求后会遍历传播树相关的所有切片，并发送标记为不安全的标签。对于每一个不安全标签的标记时序与安全标记时序相同，这里不再赘述。

4.5.4　批训练过程

用户在 Web 端可以发起训练任务，Celery Worker 收到训练任务后，会调用训练服务按指定配置训练 BLSTM 神经网络。

该模型训练时的输入为切片向量集合，对应标记集合，输出为一个训练好参数的 BLSTM 模型。由于程序切片大小有差异，切片向量长度也是不一致的，对于时序型神经网络 BLSTM 来说，本身就具有接受不定长向量的能力，然而在实际训练时，为了保证训练效率和效果，往往会将一批切片向量和对应标记合并为一批数据输入神经网络，再进行权重更新——批训练 [48–50]，这时一组切片向量需要转换为二维向量，即需要对末尾补 0，因此在 BLSTM 模型中，要考虑无效数据。

Pytorch 的 DataLoader 对象可手动指定校对函数 *collate_fn()*，该函数用于从数据集加载数据后，对数据进行最后处理，本模块根据需要构造的校对函数如图 4.20 所示。

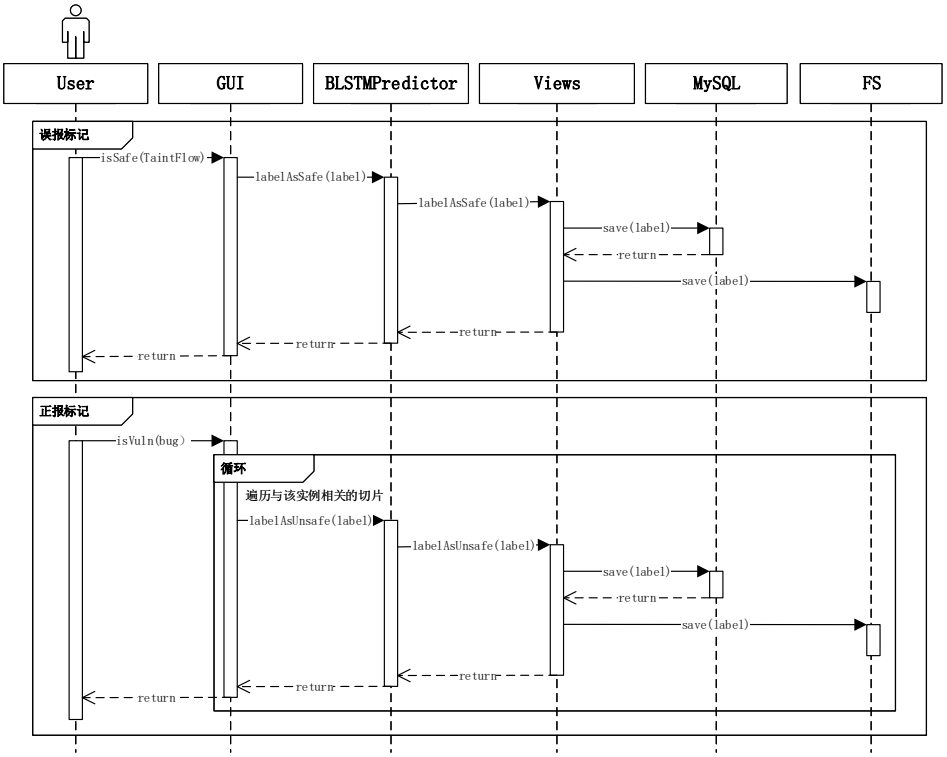

图 4.19: 漏洞标记时序图

可以看出，该函数的输入为一个向量化类和一批数据集合，该集合中存放切片字符串和对应标记对，对于每一个标记对，首先调用预处理模块对其泛化和向量化（第 3 行），接着按切片长度从长到短对一批数据排序，获取每条切片的长度记为 data_length 变量，使用 Pytorch 预定义的函数 *rnn_utils.pad_sequence()* 对已排好序的一批数据进行补 0，此时 data_x 类型已由原先的 list 类型变为二维张量（Tensor）类型，最后将这批数据的标记作为一维向量，最后函数返回切片张量（二维），切片长度向量（一维）和标记向量（一维）。

接着切片张量和切片长度向量将传递至 BLSTM 类的 forward() 方法中，代码 4.21 展示了神经网络后向传播的细节。

首先，对于切片张量进行词嵌入，即使用向量替换每一个词，其输出的 embeds 为三维张量，注意这里词嵌入已经指定了 padding 值，即将补齐用的数字映射为全 0 向量。接着使用 Pytorch 定义的 *rnn_utils.pack_padded_sequence()* 函数将实际上不等长的切片向量打包，打包数据传递至 BLSTM 层，BLSTM 层输出还原至三维张量，注意，此时张量规模为记录数 × 最长语句长度 × BLSTM 神经元个

```
def collate_fn(tokenizer, data: [(str, int)]):
    X, Y = 0, 1
    data = list(map(lambda e: (torch.LongTensor(tokenizer.encode(e[X])), e[Y
        ]), data))
    data.sort(key=lambda x: len(x[X]), reverse=True)
    data_x = list(map(lambda e: e[X], data))
    data_lengths = torch.LongTensor([len(sq) for sq in data_x])
    data_x = rnn_utils.pad_sequence(data_x, batch_first=True, padding_value=
        PADDING_VALUE)
    data_y = torch.LongTensor(list(map(lambda e: e[Y], data)))
    return data_x, data_lengths, data_y
```

图 4.20: DataLoader 的代码片段

```
def forward(self, sentences: tensor, lengths: tensor):
    embeds = self.word_embeddings(sentences)
    x_packed = rnn_utils.pack_padded_sequence(embeds, lengths, batch_first=
        self.batch_first)
    lstm_out, (h_n, h_c) = self.lstm(x_packed, none)
    lstm_out_unpacked, lstm_out_lengths = rnn_utils.pad_packed_sequence(
        lstm_out, batch_first=self.batch_first)
    if self.use_gpu:
        lstm_out_lengths = lstm_out_lengths.cuda()
    last_lstm_out = self.last_output(lstm_out_unpacked, lstm_out_lengths)
    y = self.hidden2label(last_lstm_out)
    return y
```

图 4.21: 神经网络后向传播的代码实现

数。*get_last_output()* 会获取该三维张量上每一记录的双向传播时最后一时刻的神经元输出，并整合为一二维张量，最终输出到线性层，线性层返回预测结果。

图 4.22 展示了批处理训练的具体做法，假设一批数据中存在两条数据 record 1 和 record 2，它们有效的切片向量长度为 3 和 4，将它们的单词做词嵌入后，得到张量 $\{V_1, V_2, V_3\}$ 和 $\{V_1, V_2, V_3, V_4\}$，经过 BLSTM 神经网络后，正反方向分别产生了每个时刻上的输出，此时对于记录 1 来说，需要提取最后一个时刻的神经元输出，即 O_3 向量和 O'_3 向量，并将其合并，对于记录 2 来说，提取并合并为 $[O_4, O'_4]$，该二维张量输入至线性层，得到最终分类结果。也就是说，线性层神经元个数为 BLSTM 神经元个数的两倍。

4.6　系统测试与运行展示

4.6.1　测试目标

本系统测试主要有功能测试、健壮性和性能测试，在功能测试方面主要对系统功能的正确性进行测试；而健壮性和性能测试主要关注于本系统在扫描各类被测应用时的程序不崩溃，并能在合理时间内返回数据。

在功能测试中，主要对第 3.2 节提出的需求和用例进行测试，对污点分析、程序切片、预处理和误报预测各个模块的生成测试用例并对其进行测试。在健

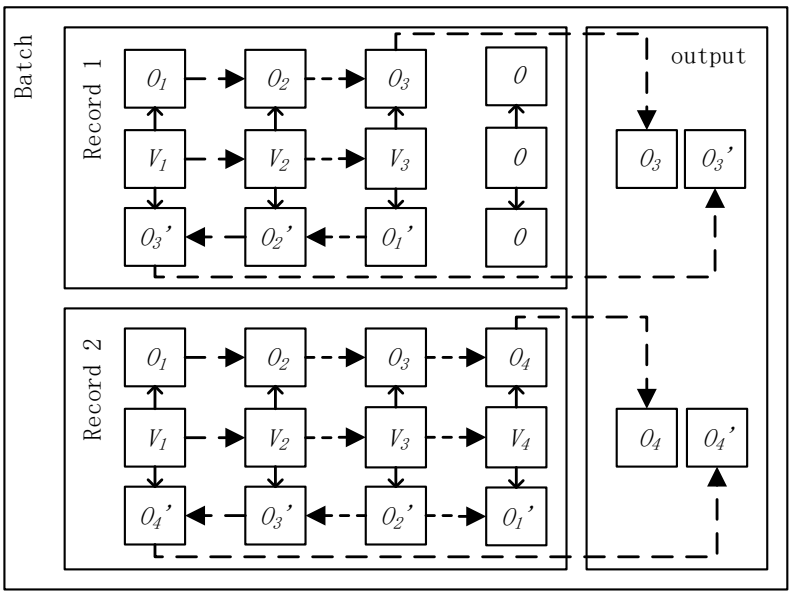

图 4.22: 批处理训练示意图

壮性测试中，主要对系统输入各类 Jar 包，检查系统是否能否在规定时间内产生扫描报告。

表 4.1为测试的软硬件环境，在硬件方面，由于扫描任务是计算密集型任务，所以客户端选择了 CPU 较高配置的 E5-2630 和内存为 32G，并将 JVM 参数调整为“-Xmx8g -Xss500m”，服务端的硬件配置与客户端相同，软件上有 Pytorch、Django、Celery、MySQL、Redis 等。

表 4.1: 系统测试软硬件环境

设备与软件		备注信息
客户端	CPU	型号为 Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
	内存	大小为 32G
	Java	JVM 参数为“-Xmx8g -Xss500m ”
服务端	CPU	型号为 Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
	内存	大小为 32G
	Pytorch	版本为 1.3.1
	Django	版本为 2.2.5
	Celery	版本为 4.4.0
	MySQL	版本为 8.0.12
	Redis	版本为 3.0

4.6.2　功能测试

本章节主要针对各个模块的主要功能设计测试套件，并对每个测试套件进行具体测试。测试套件如表 4.2 所示。

表 4.2: 系统测试用例套件

ID	所在模块	测试功能	测试内容
TS1	污点分析模块	基于污点分析的静态扫描	提交扫描任务，系统能够将污点分析结果显示在用户界面
TS2	污点分析模块	污点分析结果的存取	对于一个污点分析结果，系统可以将其保存，并在打开时导入
TS3	程序切片模块	程序切片	对于一个污点分析项目，对每一个漏洞进行切片，生成程序切片项目
TS4	预处理模块	切片泛化与向量化	对于一个切片文本，对其进行泛化和向量化
TS5	误报预测模块	预测漏洞安全性	对于一个漏洞实例，利用模型预测其是否为误报
TS6	误报预测模块	标记漏洞安全性	对于一个漏洞实例，标记该漏洞是否为误报
TS7	误报预测模块	模型训练	指定模型训练配置，根据配置训练模型

在污点分析模块，主要有测试套件 TS1，对用户提交的 Jar 包集合进行污点分析，并将污点分析结果以漏洞实例和对应若干棵污点传播树的形式返回给用户；测试套件 TS2，用户对于污点分析结果，进行存取操作。其包含的测试用例如表 4.3 所示，TS1 中含一个测试用例 TC1-1；TS2 套件中含两个测试用例，TC2-1 和 TC2-2，分别测试对于结果的保存和读取。经过测试，得出的测试结果与预期结果相符，测试用例全部通过。

表 4.3: 基于污点分析的静态扫描测试用例

ID	说明	输入	预期输出	测试结果
TC1-1	污点分析	用户提交的 Jar 包	污点分析结果（显示到界面）	通过
TC2-1	污点分析结果保存	污点分析结果	记录分析结果的 XML 文件	通过
TC2-2	污点分析结果读取	记录分析结果的 XML 文件	污点分析结果	通过

在程序切片模块，主要有一个测试套件 TS3，即测试系统能否成功将污点分析项目转化为切片项目，在切片项目中，存在污点传播树和其对应的污染流，每个污点传播对应一条切片。该测试套件对应的测试用例如表 4.4 所示，其主要包含有 TC3-1，对上一模块产生的分析结果进行处理，从报告中还原污点传播树；TC3-2，拆解污点传播树，将传播树拆解为子污染流的集合；TC3-3，对一个污染流进行后向切片，由于 Joana 自身存在一定不稳定，当切片成功时，程序返回

切片文本，在切片失败时扫描任务不崩溃，并返回错误原因。经过测试，以上测试用例输出均与预期相符，测试全部通过。

表 4.4: 程序切片测试用例

ID	说明	输入	预期输出	测试结果
TC3-1	处理污点分析结果	污点分析结果	漏洞实例及其对应传播树集合	通过
TC3-2	拆解污点传播树	污点传播树	子污染流集合	通过
TC3-3	污染流后向切片	污染流	切片文本（失败返回错误原因）	通过

数据预处理模块的测试套件为 TS4，其用于测试模块是否能对于一个切片文本进行正确泛化和向量化处理。其包含的测试用例如表 4.5 所示，TC4-1 用于测试预处理模块能否对于一个切片文本进行有效泛化处理，对其输出单词序列；TC4-2 用于测试模块对于单词序列集合，能否将其转化为向量集合和对应的单词表，在输入时，单词表为可选项，若不存在单词表则新建，存在则根据已有单词表生成向量。经过测试，以上测试用例输出均与预期相符，测试全部通过。

表 4.5: 数据预处理模块测试用例

ID	说明	输入	预期输出	测试结果
TC4-1	泛化处理	切片文本	单词序列	通过
TC4-2	向量化处理	单词序列集合和单词表（可选）	单词对照表和向量集合	通过

误报预测模块的测试套件为 TS5；TS6，对漏洞实例安全性进行标记；TS7，对已有新数据进行模型训练。它们对应的测试用例如表 4.6 所示，安全性实例预测套件对应的测试用例有 TC5-1，对切片文本进行安全性预测，由于切片文本与污染流成一一对应关系，切片文本安全即污染流安全，测试检查系统是否能返回预测结果，并同时将污染流保存到文件系统和数据库中；TC5-2，对一个漏洞实例进行误报预测，测试给定任意漏洞实例和对应传播树的集合，系统能否返回漏洞是否为误报。

安全性标记对应的测试套件为 TS6，测试用例有 TC6-1，标记污染流安全性，对于单个污染流标记其是否安全，系统需要将该标记数据存储至数据库和文件系统中，更新预测结果；TC6-2，标记可利用的传播树，系统需要将其对应的所有污染流标记为不安全，当漏洞的所有污染传播树无法传播污点时，对漏洞也应标记为不安全。

模型训练的测试套件为 TS7，对应的测试用例有 TC7-1，用户在后台新建训

练配置，系统将其配置保存至数据库；TC7-2，用户发起训练请求，系统异步完成请求，并将训练好的模型保存至数据库和文件系统。经过测试，以上测试用例输出均与预期相符，测试全部通过。

表 4.6: 误报预测模块测试用例

ID	说明	输入	预期输出	测试结果
TC5-1	切片文本安全性预测	污染流和对应切片	预测结果，同时污染流保存到文件系统和数据库	通过
TC5-2	漏洞误报预测	漏洞实例和对应传播树集合	漏洞是否为误报	通过
TC6-1	标记污染流	污染流切片和标记	更新预测结果，标记存储至数据库和文件系统中	通过
TC6-2	标记可利用的传播树	污点传播树	更新预测结果，所有对应的污染流标记为不安全并存储至数据库和文件系统	通过
TC7-1	新建训练配置	训练配置	生成配置存储至数据库	通过
TC7-2	发布训练任务	训练任务请求	异步进行任务训练，并将训练完成的数据保存到数据库和文件系统	通过

4.6.3　健壮性和性能测试

为测试本系统的健壮性和性能，本文选择 Maven 中央仓库流行度 Top 100 的项目，在每个项目选择使用最多的构建版本作为测试对象 [1] 批量建立扫描任务进行测试，对于一个项目，本文将系统的最大扫描时间设为 4 小时，如果超过该时间系统仍未返回结果，则认为系统扫描失败，表 4.7 列出了测试结果。

表 4.7: 健壮性和性能测试结果

测试 Jar 包	合法 Jar 包	扫描成功率	最短扫描时间 (s)	平均扫描时间 (s)	最长扫描时间 (s)
96	82	100%	7.21	67.22	1815.24

其中有 4 个构建为 aar 格式，由于系统只能输入 Jar 格式，因此它们无法参与测试，即实际参与测试的文件为 96 个 Jar 包文件，在 96 个 Jar 文件中，有 14 个 Jar 文件内无“.class”文件，系统均能在启动扫描时检测并提示用户 Jar 包不合法，并正常退出。对于剩下的 82 个文件，系统在限定时间内返回扫描结果，即扫描成功率为 100%，至此本文认为该系统具有较高的健壮性。

[1] https://mvnrepository.com/popular?p=1，抓取时间为 2019 年 10 月 24 日，当前流行度榜单可能已经发生变化

在系统性能方面，可以看出对于测试的 Jar 包，系统最短扫描时间仅有 7.21s，对应的 Jar 包为 support-annotations-27.1.1.jar [2]，这一类包的特点在于其本身项目较小，项目中与安全有关的函数（Sink 点）调用也相对较少，因此其分析时间也较短；平均扫描时间为 67.22s，这也意味着对于绝大多数的 Jar 包，开发者在一分钟左右即可收到扫描结果；系统最长扫描时间为 1815.24s，对应的 Jar 包为 groovy-all-2.4.7.jar [3]，该包为 Groovy 语言的解释器，因此代码中会出现大量的安全相关函数和入口点，因此分析时间较长，但是系统仍能在一个小时之内给出扫描报告，对于大型项目而言，本文认为这是可以被开发者接受的。

综上，本系统对于各类 Jar 包有较高的健壮性，并且能较为及时的完成扫描任务，在保证代码安全同时不会影响代码开发进度。

4.6.4　系统效果评估

相对于传统污点分析系统，本系统主要采取神经网络预测降低污点分析报告误报来提升准确性，在本小节，将通过实验证明本系统相较于传统安全分析工具的优势。

4.6.4.1　参数设置

本系统的 BLSTM 预测模型在训练和构造时存在一系列参数，系统上线前本文对已有数据集进行测试，在权衡准确性和效率后，系统选择的参数配置如下：词嵌入维度（embedding_dim）为 16；隐藏层神经元个数为 32；最小词频为 4；批处理记录数为 32；基础学习率为 0.01；提前停止忍耐度为 3；最大迭代次数为 20。

4.6.4.2　评估方法和度量

表 4.8: 漏洞扫描报告混淆矩阵

		扫描结果	
		存在漏洞	无漏洞
真实结果	存在漏洞	TP	FN
	无漏洞	FP	TN

本文将本系统与目前流行的 Find Security Bugs 扫描报告做对比，将数据集的 90% 作为训练集，10% 作为测试集（对于 Find Security Bugs 而言，其没有训练过

[2] https://mvnrepository.com/artifact/com.android.support/support-annotations/27.1.1

[3] https://mvnrepository.com/artifact/org.codehaus.groovy/groovy-all/2.4.7

程，只对测试集用例进行扫描）再根据两系统的扫描报告，以准确率（Accuracy）、精确率（Precision）、召回率（Recall）和 F_1 作为度量比较系统准确性，为了解释这些度量计算方法，首先介绍对于扫描报告的混淆矩阵，如表 4.8 所示，对于每一个测试用例，扫描结果会对其报告为存在漏洞和不存在漏洞，因此产生实际为漏洞且扫描结果为漏洞的用例数 TP，实际为漏洞而扫描结果为无漏洞的用例数 FN （漏报数），实际无漏洞而扫描结果为有漏洞的用例数 FP（误报数）和实际无漏洞且扫描结果为无漏洞的用例数 TN，在此基础上，报告的准确率计算方式为 $Accuracy = \frac{TP+TN}{TP+FN+FP+FN}$、漏洞精确率为 $Precision = \frac{TP}{TP+FP}$、漏洞召回率为 $Recall = \frac{TP}{TP+FN}$ 以及 F_1 计算方式为 $F_1 = \frac{2 \cdot Precision \cdot Recall}{Precision+Recall}$，每次实验重复三次取平均值作为最终结果。

4.6.4.3　评估数据集

本文使用 OWASP Benchmark v1.1 [4] （简称 OWASP）作为数据集，表 4.9 中显示了各个数据集中的数据量大小。

表 4.9: 效果评估数据集

	测试用例总数	真实漏洞用例数	无漏洞用例数	安全污染流	不安全污染流	污染流总数
OWASP	10769	6377	4392	3501	6377	9878

由于本系统只能处理部分类型，因此本文对该数据集进行筛选，去掉 Find Security Bugs 中无法用污点分析检测的漏洞类型，因此表中测试用例总数会比数据集公布的数目少。

4.6.4.4　实验结果

表 4.10: 本系统与 Find Security Bugs 对比结果

	准确率	精确率	召回率	F1
Find Security Bugs	68.40 %	65.09 %	**100.00 %**	78.51 %
本系统	**87.84 %**	**90.53 %**	88.65 %	**89.58 %**

结果如表 4.10 所示。可以看出，为了不发生漏报，Find Security Bugs 产生非常高的误报率（误报率为 1-65.09%=34.91%）从而导致准确率和 F1 值水平也

[4] https://www.owasp.org/index.php/Benchmark

较低，不准确的漏洞报告不仅会给安全工程师造成巨大压力，更可能造成用户对系统的不信任，而本系统结合污点分析和机器学习的优势，仅牺牲 11.35% 的召回率，将精确率提高到 90% 以上，即平均系统中报告的 10 个漏洞中，只有一例可能为误报。在准确率和 F1 指标上也远高于 Find Security Bugs，这说明牺牲召回率具有较高的收益比，对于新产生的漏报，安全工程师仍可以通过其他检测手段（如黑盒测试、灰盒测试）加以弥补，配合白盒扫描系统共同保障应用安全。

综上，相较于传统污点分析类扫描器，本系统能够更为准确地发现漏洞，并且提升准确性的代价较小。

4.6.5　系统运行展示

本系统为 C/S 架构的安全扫描系统，这里主要展示污点分析结果界面，设置预测服务器界面，预测结果界面、标记正报界面（漏洞可利用）和标记误报界面（漏洞不可利用）任务界面。为了真实展示程序运行结果，本小节系统除了使用前文的 OWASP benchmark v1.1 数据外，还使用部分 Juliet Test Suite for Java v1.3 [5] 的数据进行训练得到预测模型，并使用 Java Sec Code 项目 [6] 作为被测项目，该项目是一个模拟真实 Java 漏洞和修复方案的代码合集，目前已有 500 多个 Star。

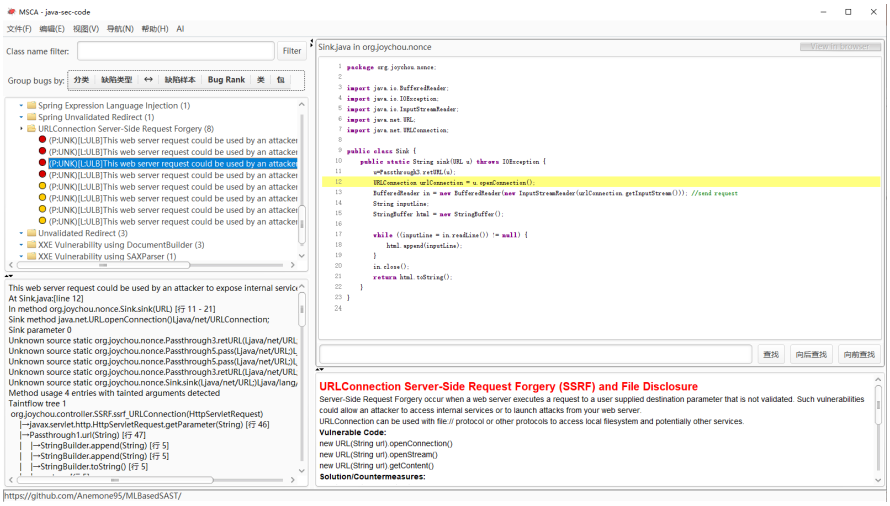

图 4.23: 污点分析结果界面

[5] https://samate.nist.gov/SARD/testsuite.php，由于该数据集中的每个漏洞用例同时存在可利用和不可利用的污点传播树，即用例最终总是成立的，因此无法在效果评估环节使用。

[6] https://github.com/JoyChou93/java-sec-code

图 4.23 展示了系统污点分析结果界面，该界面主要沿用 Spotbugs 界面，在菜单栏可以新建，读取，保存扫描项目，对 UI 进行调整以及进行预测。左上方窗口显示按类型整理的漏洞实例，由于用户还未对漏洞进行预测和标记，因此每个叶子节点的开头显示为“[P:UNK][L:ULB]”，左下方显示一个漏洞实例的污点传播树——这是本系统的主要改进之处，右上方为对应的程序代码，右下方为关于该漏洞的解释信息。

图 4.24 展示了设置预测服务器的界面，点击上方菜单栏“AI”中的“Setting Remote Server”将会出现该界面，设置预测服务器是进行误报预测的首要步骤，在设置服务器窗口中，用户需要指定服务器的 URL 地址和管理员发放的 token。

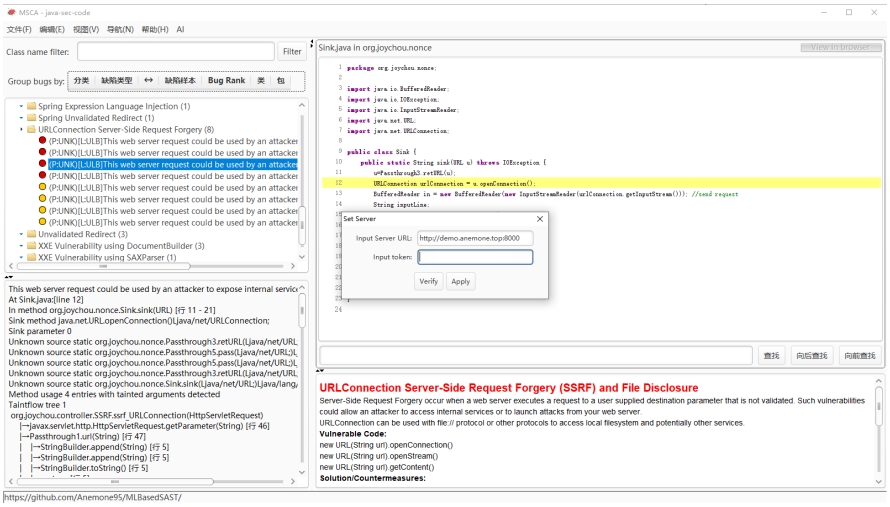

图 4.24: 设置预测服务器界面

点击上方菜单栏“AI”中的“Slice and Predict”按钮，系统将对污点分析结果进行误报预测，图 4.25 展示了预测后一个被预测为误报的漏洞实例，在左上方，可以看到该漏洞实例的叶子节点已经变灰，且预测显示为“[P:FP]”（误报），在左下方，可以看到被预测为清洁函数的污染流已经由有清洁标记标注，这些标记用于向用户解释系统为何将其预测为误报。

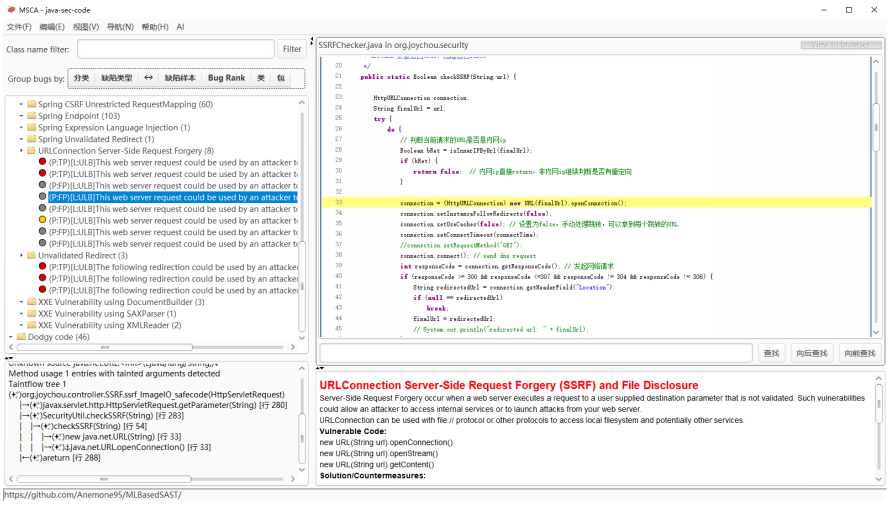

图 4.25: 预测结果展示

另外，可见图中的代码正是第二章中图 2.3(c) 中的代码，根据第二章的分析，该代码是为有效的 SSRF 修复代码，而本系统已经能够做出正确的预测，这证明本系统确实可以根据先前的学习，排除误报。

对于报告中真实存在的漏洞，用户可以对其标记为正报，右击漏洞实例选择“Label as True Positive”即可弹出标记正报的界面，如图 4.26 所示，标记正报时用户必须在下拉菜单中选择一棵真实可以利用的污点传播树，点击提交后标记完成，完成后系统会更新左侧漏洞树，将“[L:ULB]”（未标记）转变为“[L:TP]”（标记为正报）。

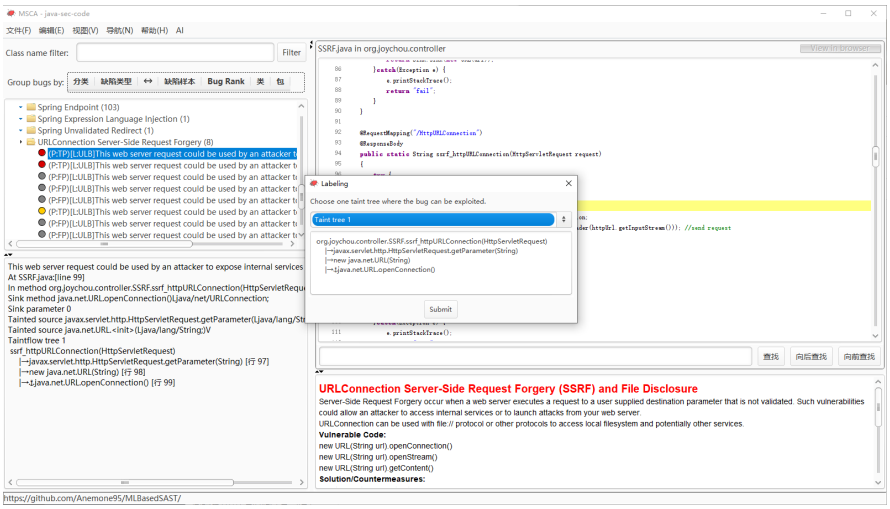

图 4.26: 标记正报界面

对于报告中不可利用的漏洞，用户可以将其标记为误报，右击漏洞实例选

择“Label as False Positive”即可弹出标记误报的界面，如图 4.27 所示。标记误报时用户需要在下拉菜单中指定一条污染流，同时用户可以在下方看到关于此污染流的切片内容，对误报的标记实际是对安全污染流标记，标记完成后系统会重新计算并更新漏洞树，用户可以多次点击标记，直至该漏洞的标记变为“[L:FP]”（标记为误报）。

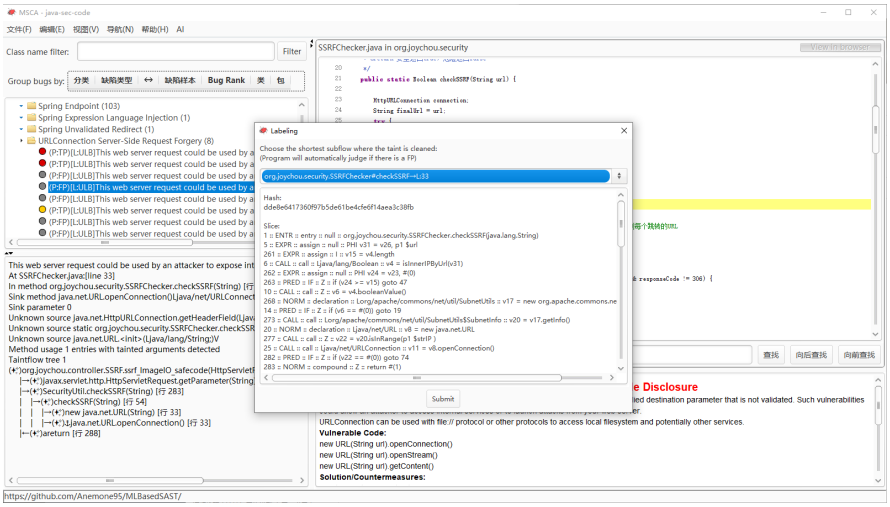

图 4.27: 标记误报界面

4.7　本章小结

本章结合实例，阐述基于机器学习的 Java 静态漏洞扫描系统中各个模块的具体实现，在污点分析模块中，主要介绍在分析过程中记录污点传播信息、构造污点传播图以及构造污点传播树的实现细节，接着结合代码，介绍程序切片模块中切片控制流程和具体的后向切片器实现，在数据处理模块中，阐述泛化和向量化的具体方法，最后介绍预测模块通过预测切片进而推导漏洞是否为误报的过程。本章结尾对系统进行各个方面的测试以及运行结果展示，通过展示说明本系统足够健壮，并且有真实的误报排除能力。

第五章　总结与展望

5.1　总结

为了解决应用安全问题，本文设计了一套基于污点分析、程序切片和BLSTM的静态安全扫描系统，旨在从源头处遏制程序漏洞。相较于传统的污点分析类代码扫描系统，本系统对污点传播树进一步分析，通过程序切片和先前标记进行学习，能够有效地排除误报，保证扫描结果准确性，大大降低了系统使用时的人力成本。本文主要工作如下：

1. 本文对开源污点传播工具 Find Security Bugs 进行改造，使之能在报告中展示污点传播树，友好地向用户展示可能的漏洞利用过程。

2. 系统对污点传播树进行拆分，将污点传播流和子传播流作为切片单位进行程序切片，系统优化了 Joana 切片过程，使之能够在缺失依赖、异构 Jar 包的情况下切片，并通过限制调用图限制程序切片范围，结合污点传播树的拆分，解决了目前切片资源消耗大导致扫描任务无法及时结束的问题。

3. 系统根据实际情况对切片后的 SSA 进行泛化处理，进一步保证预测模型的泛化能力。

4. 系统通过对漏洞的各个污点传播树的传播流及其子污染流进行预测，从而预测漏洞是否为误报。系统采用 BLSTM 模型，该模型在学术界已被多项工作证明在漏洞预测领域有较为显著的效果。用户在系统界面中能够得知漏洞预测结果，若为误报，系统能够清晰给出判断依据，即污点在哪一段传播流中无法继续传播，一定程度上弥补了深度学习可解释性差的问题。安全工程师能够轻松地对漏洞、传播树和传播流进行标记，并发起新的学习任务，不断提高预测模型的准确率。

系统被设计为有良好的可拓展性，对于污点分析结果翻译、切片器和预测器，系统都对其高度抽象为接口，方便后期根据实际需要对系统中各个模块所用技术进行更换。

经过测试和实验，结果表明本系统具有较高的健壮性，在 Maven 仓库中前 100 流行度的项目中系统扫描成功率为 100 %，此外，本系统具有较高准确性，在 OWASP 数据集中，本系统预测模块能够准确预测安全污点传播流，相较于

Find Security Bugs，本系统在 OWASP 数据集上提高了约 20 % 的准确率，减少了 25.44 % 的误报。

5.2　展望

本系统是将深度学习应用于漏洞扫描领域的一次成功探索，实现了传统检测技术与深度学习相结合，对 Java 代码进行更准确的安全扫描任务，然而在未来，系统在以下方面还有较大的改进空间：

1. 系统目前只针对 Java 语言，在未来可以将其方法推广到其他语言的项目。
2. 系统只能适用于能用污点分析方法分析的漏洞类型，在未来可以将切片和预测方法推广到其他基础分析的误报排除中。
3. 系统并不能发现更多漏报，在未来可以使用类似的切片技术和预测方法，结合其他经典漏洞挖掘技术，在消除漏报方面做进一步提升。
4. 系统在特征表示时，目前实际上是将切片转化为单词序列，再将单词序列进行向量化处理，在未来可以参考程序图特征表示的前沿工作，将切片表示成信息更丰富的特征，进一步提高预测准确性。
5. 目前由于数据量有限，对于所有污点漏洞均使用一个模型进行预测，待数据量进一步扩大时，可以将针对每一种漏洞单独训练模型。

参考文献

[1] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, VulDeePecker: A Deep Learning-Based System for Vulnerability Detection, in: Proceedings of the 2018 Network and Distributed System Security Symposium, Internet Society, Reston, VA, 2018.

[2] 刘剑, 苏璞睿, 杨珉, 和亮, 张源, 朱雪阳, 林惠民, 软件与网络安全研究综述, 软件学报 29 (1) (2018) 42–68.

[3] O. Tripp, S. Guarnieri, M. Pistoia, A. Aravkin, ALETHEIA:Improving the Usability of Static Security Analysis, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM Press, New York, New York, USA, 2014, pp. 762–774.

[4] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, J. Hu, VulPecker: An automated vulnerability detection system based on code similarity analysis, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016, pp. 201–213.

[5] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, Y. Jiang, LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment Through Program Metrics, in: Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering, IEEE, 2019, pp. 60–71.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler, EXE: Automatically Generating Inputs of Death, ACM Transactions on Information and System Security 12 (2) (2008) 1–38.

[7] D. Hovemeyer, W. Pugh, Finding bugs is easy, Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications 39 (12) (2004) 92–106.

[8] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, O. Weisman, TAJ: effective taint analysis of web applications, Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation 44 (6) (2009) 87–97.

[9] U. Koc, S. Wei, J. S. Foster, M. Carpuat, A. A. Porter, An Empirical Assessment of Machine Learning Approaches for Triaging Reports of a Java Static Analysis Tool, Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (2019) 288–299.

[10] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering 30 (5) (2005) 263.

[11] Y. Xie, A. Aiken, Saturn: A scalable framework for error detection using Boolean satisfiability, ACM Transactions on Programming Languages and Systems 29 (3) (2007) 16.

[12] C. Cadar, D. Dunbar, D. Engler, Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, Vol. 8, 2019, pp. 209–224.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended static checking for Java, in: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, Vol. 48, ACM Press, New York, New York, USA, 2002, pp. 234–245.

[14] U. Koc, P. Saadatpanah, J. S. Foster, A. A. Porter, Learning a classifier for false positive error reports emitted by static code analysis tools, Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, co-located with PLDI 2017 (2017) 35–42.

[15] N. Jovanovic, C. Kruegel, E. Kirda, Pixy: a static analysis tool for detecting Web application vulnerabilities, in: Proceedings of the 2006 IEEE Symposium on Security and Privacy, IEEE, 2006, pp. 258–263.

[16] Y. Kim, J. Lee, H. Han, K.-M. Choe, Filtering false alarms of buffer overflow analysis using SMT solvers, Information and Software Technology 52 (2) (2010) 210–219.

[17] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: Proceedings of the 35th International Conference on Software Engineering, IEEE, 2013, pp. 672–681.

[18] F. Yamaguchi, A. Maier, H. Gascon, K. Rieck, Automatic Inference of Search Patterns for Taint-Style Vulnerabilities, in: Proceedings of the 2015 IEEE Symposium on Security and Privacy, IEEE, 2015, pp. 797–812.

[19] T. Kremenek, D. Engler, Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations, in: Proceedings of the 10th International Conference on Static Analysis, San Diego, CA, USA, 2003, pp. 295–315.

[20] U. Yuksel, H. Sozer, Automated Classification of Static Code Analysis Alerts: A Case Study, in: Proceedings of the 2013 IEEE International Conference on Software Maintenance, IEEE, 2013, pp. 532–535.

[21] S. Kim, S. Woo, H. Lee, H. Oh, VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery, in: Proceedings of the 2017 IEEE Symposium on Security and Privacy, IEEE, 2017, pp. 595–614.

[22] N. H. Pham, T. T. Nguyen, H. A. Nguyen, T. N. Nguyen, Detection of recurring software vulnerabilities, in: Proceedings of the 2010 IEEE/ACM international conference on Automated software engineering, ACM Press, New York, New York, USA, 2010, pp. 447–456.

[23] 梅宏, 王千祥, 张路, 王戟, 软件分析技术进展, 计算机学报 32 (9) (2009) 1697–1710.

[24] G. A. Kildall, A unified approach to global program optimization, in: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press, New York, New York, USA, 1973, pp. 194–206.

[25] B. Shastry, F. Yamaguchi, K. Rieck, J.-P. Seifert, Towards Vulnerability Discovery Using Staged Program Analysis, in: Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, Cham, 2016, pp. 78–97.

[26] J. M. Schumann, Automated Theorem Proving in Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[27] C. Calcagno, D. Distefano, P. O'Hearn, H. Yang, Compositional shape analysis by means of bi-abduction, in: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Vol. 58, ACM Press, Savannah, GA, USA, 2009, pp. 289–300.

[28] C. Cadar, K. Sen, Symbolic Execution for Software Testing: Three Decades Later, Communications of the ACM 56 (2) (2013) 82–90.

[29] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Vol. 40, ACM Press, New York, New York, USA, 2005, pp. 213–223.

[30] H. Liang, X. Pei, X. Jia, W. Shen, J. Zhang, Fuzzing: State of the Art, IEEE Transactions on Reliability 67 (3) (2018) 1199–1218.

[31] V. J. M. Manes, H. S. Han, C. Han, sang kil Cha, M. Egele, E. J. Schwartz, M. Woo, The Art, Science, and Engineering of Fuzzing: A Survey, IEEE Transactions on Software Engineering (2019) 1–1.

[32] 邹权臣, 张涛, 吴润浦, 马金鑫, 李美聪, 陈晨, 侯长玉, 从自动化到智能化: 软件漏洞挖掘技术进展, 清华大学学报 (自然科学版) 58 (12) (2018) 1079–1094.

[33] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna, Driller: Augmenting Fuzzing Through Selective Symbolic Execution, in: Proceedings of the 2016 Network and Distributed System Security Symposium, Internet Society, Reston, VA, 2016.

[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, H. Bos, VUzzer: Application-aware Evolutionary Fuzzing, in: Proceedings of the 2017 Network and Distributed System Security Symposium, Internet Society, Reston, VA, 2017.

[35] H. Han, D. Oh, S. K. Cha, CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines, in: Proceedings of the 2019 Network and Distributed System Security Symposium, No. February, Internet Society, Reston, VA, 2019.

[36] 王蕾, 李丰, 李炼, 冯晓兵, 污点分析技术的原理和实践应用, 软件学报 28 (4) (2017) 860–882.

[37] M. Weiser, Program Slicing, IEEE Transactions on Software Engineering SE-10 (4) (1984) 352–357.

[38] B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, A brief survey of program slicing, ACM SIGSOFT Software Engineering Notes 30 (2) (2005) 1–36.

[39] K. J. Ottenstein, L. M. Ottenstein, The program dependence graph in a software development environment, Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments 19 (5) (1984) 177–184.

[40] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems (TOPLAS) 12 (1) (1990) 26–60.

[41] J. Su, Z. Tan, D. Xiong, R. Ji, X. Shi, Y. Liu, Lattice-Based Recurrent Neural Network Encoders for Neural Machine Translation, in: Proceedings of the 31st AAAI Conference on Artificial Intelligence, AAAI'17, AAAI Press, 2017, pp. 3302–3308.

[42] B. Hanin, Which Neural Net Architectures Give Rise to Exploding and Vanishing Gradients?, in: Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, Curran Associates Inc., Red Hook, NY, USA, 2018, pp. 580–589.

[43] S. Hochreiter, J. Schmidhuber, Long Short-Term Memory, Neural Computation 9 (8) (1997) 1735–1780.

[44] A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in: Proceedings of the 34th International Conference on Software Engineering, IEEE, 2012, pp. 837–847.

[45] E. C. R. Shin, D. Song, R. Moazzezi, Recognizing Functions in Binaries with Neural Networks, in: Proceedings of the 24th USENIX Conference on Security Symposium, SEC'15, USENIX Association, USA, 2015, pp. 611–626.

[46] M. White, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, Toward Deep Learning Software Repositories, in: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE, 2015, pp. 334–345.

[47] P. Kruchten, The 4+1 View Model of architecture, IEEE Software 12 (6) (1995) 42–50.

[48] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, K. Keutzer, ImageNet Training in Minutes, in: Proceedings of the 47th International Conference on Parallel Processing, ACM Press, New York, New York, USA, 2018, pp. 1–10.

[49] E. Hoffer, I. Hubara, D. Soudry, Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook, NY, USA, 2017, pp. 1729–1739.

[50] Y. You, J. Hseu, C. Ying, J. Demmel, K. Keutzer, C.-J. Hsieh, Large-batch training for LSTM and beyond, in: Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, New York, NY, USA, 2019, pp. 1–16.

简历与科研成果

基本情况 徐文远，男，汉族，1995 年 8 月出生，江苏省南京市人。

教育背景

2018.9～2020.6 南京大学软件学院 硕士

2014.9～2018.6 扬州大学信息工程学院 本科

科研项目

1. 南京安瑞信息通信科技有限公司项目：调度自动化软件安全漏洞与风险实验能力提升关键技术研究（4501084141），2019-2020
2. 国家自然科学基金项目：基于可理解信息融合的人机协同移动应用测试研究（61802171），2019-2021
3. 中央高校基本科研业务费专项资金资助项目：基于群智协同的众包测试技术（14380021），2020-2020
4. 南京大学技术创新基金项目：基于 AI 中台的移动应用测试技术研究与实现（14913413），2020-2020

科研成果

1. 房春荣、龚爱、**徐文远**、蒋燕、李玉莹、陈振宇，“一种基于 Android 源代码的风险等级预测方法”，申请号：201810092876.1，已受理。

致　　谢

首先感谢陈振宇老师，感谢您两年以来对我的谆谆指导，不仅在学术上给我创造了很多机会，更重要的是教会我如何做人，让我如今能以一个研究生的身份走入社会。即使是在疫情严重时期，您也不断地牺牲个人时间通过线上会议和微信了解我们的毕设进展，提出宝贵意见，保证了本文的专业性和技术深度。感谢房春荣老师，在本项目的设计、实验安排等方面提出的宝贵建议和意见。同时感谢实验室黄勇老师在技术上的帮助和指导，是您的严格要求保证了本系统的健壮性。

感谢张双江、李灏宇等师兄为本文的行文安排和格式上提供的帮助。感谢同为安全组的蒋燕和史洋洋同学，让我在科研的道路上不再孤单，感谢你们在平时研究和毕业设计过程中对我的帮助，在进展不顺利的时候给我安慰并与我一起出谋划策。

感谢 Koc 教授，是你们团队在学术上的工作为本文提供了理论支持，感谢 WALA 开发者 Manu，Joana 开发者 Simon，感谢你们的建议和意见让本文无依赖切片成为现实。

感谢暑期实习时与我一起工作的同事们，是你们让我了解到了真实安全运营时面临的问题，启发了本文的工作。

感谢父母，谢谢你们在我完成毕设期间对我的理解和支持，给我了一个舒适的工作环境，保证了毕设的顺利完成。

版权与原创性说明

任何收存和保管本论文的单位和个人，未经作者本人授权，不得将本论文转借他人并复印、抄录、拍照或以任何方式传播，否则，引起有碍作者著作权益的问题，将可能承担法律责任。

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

作者签名：

日期: 2020 年 5 月 29 日