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	C++源 代 码 漏 洞 静 态 扫 描 系 统 的 设 计 与 实 现
作 者 姓 名	史 洋 洋
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　28 日

学　　号：MF1832139

论文答辩日期：2020 年　5 月　23 日

指 导 教 师：　  　  　**（签字）**

The Design and Implementation of C++ Source Code Vulnerability Static Scanning System

By

Yangyang, Shi

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： C++ 源代码漏洞静态扫描系统的设计与实现

工程硕士（软件工程领域） 专业 2020 级硕士生姓名： 史洋洋

指导教师（姓名、职称）： 陈振宇 教授 房春荣 助理研究员

摘 要

C++ 源代码漏洞静态扫描是指在不运行程序的情况下，使用污点分析、数据流分析技术挖掘潜在的漏洞。C++ 是最流行的语言之一，但其内存模型决定了 C++ 相比于 Java 等语言更容易出现内存损坏等漏洞。静态扫描技术成本低、速度快，得到了开发者的广泛使用。随着软件的规模变大、复杂度变高，静态扫描工具会忽略控制流、上下文来提升扫描效率，但会导致误报数量的增多。为了改善开发者漏洞审核流程，降低漏洞审核难度，C++ 源代码漏洞静态扫描系统亟需降低误报率来协助开发者交付更健壮的代码。

本系统创新地引入基于机器学习的迭代反馈式误报过滤机制，来解决 C++ 源代码漏洞扫描中误报率高的问题。首先，系统融合多个开源漏洞扫描工具对程序进行扫描，获取丰富的原始漏洞报告。其次，系统使用漏洞扫描工具集对带漏洞标签的源码数据集进行扫描来获取误报数据，并使用该数据结合机器学习算法训练误报过滤器，获取过滤后的漏洞报告。接着，系统将漏洞交给漏洞专家审核，获取误报漏洞。最后，系统使用相似度算法寻找与误报漏洞代码相似的代码，并用这些数据再次训练误报过滤器。通过将漏洞列表中的误报项过滤掉，漏洞报告的有效性和可用性得到了提升，同时开发者参考过滤后报告修复漏洞，可以生产更高质量代码。本系统划分为 C++ 源代码漏洞静态扫描模块、C++ 源代码特征提取模块、漏洞静态扫描误报过滤模块、漏洞审核反馈模块。为实现服务间松耦合，本系统使用 Docker 容器技术对扫描服务、误报过滤服务进行封装。为保证扫描服务的高性能，本系统使用异步队列中间件 RabbitMQ 进行服务间消息传递。为保证扫描服务的持续优化，本系统使用 Jenkins 持续集成工具实现误报过滤模型的自动更新。

本系统相比于目前使用最广泛的开源工具 TscanCode 和 Cppcheck，F1 值分别提高了 30% 和 22%，有效地降低了 C++ 源代码漏洞静态扫描的误报。本系统提高了 C++ 源代码漏洞静态扫描器的可用性，减少了误报漏洞数量，减轻了开发者审核漏洞负担，为交付高可靠的代码提供保障。

关键词： C++ 源代码，漏洞静态扫描，误报过滤，机器学习，代码相似度检测

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 C++ Source Code Vulnerability Static Scanning System

SPECIALIZATION: Software Engineering

POSTGRADUATE: Yangyang, Shi

MENTOR: Professor Zhenyu, Chen, Research Assistant Chunrong, Fang

Abstract

C++ source code vulnerability static scanning is digging the potential vulnerability of C++ source code using taint analysis and data flow analysis without running the program. C++ is one of the most popular languages, but its memory model determines that C++ has more memory vulnerabilities than other languages, such as Java.

Because static scanning technology has low resource cost and high efficiency, it has been widely used by developers. As the size and complexity of the software become larger, the static scanning system will ignore control flow and context to improve the scanning efficiency. However, the number of vulnerabilities and false positives has increased along with it. In order to improve the developer vulnerability review process and reduce the difficulty of vulnerability review, the C++ source code vulnerability static scanning system urgently needs to reduce the false positive ratio of false negatives to assist developers in delivering more robust code.

This system innovatively introduces an iterative false alarm filtering mechanism based on machine learning to reduce false positive ratio in vulnerability scanning. First, the system integrates multiple open source vulnerability scanning tools to scan the program to obtain the richer original vulnerability reports. Second, the vulnerability scanning tool set is used to scan the source data set with vulnerability tags to obtain false positive data, and the data is used in machine learning processes to train false positive filters and get the filtered vulnerability report. Then, the vulnerability expert conducted a manual review on the filtered vulnerability list to obtain the false positive. Finally, the similarity algorithm is used to find the vulnerable code similar to the code with false positive label vulnerability, and the false positive filter is trained again with this data to

form a closed loop. By filtering out the false positive items in the vulnerability list, the validity and usability of the vulnerability report has been improved. Developers refer to the vulnerability report to fix the vulnerability and produce higher quality code. The system is divided into four modules, including vulnerability static scanning module, C++ source code feature extraction module, false positive filtering module, and false positive feedback module. In order to achieve loose coupling between services, this system uses Docker container technology to encapsulate scanning services. In order to ensure the high performance of the scanning service, the system uses RabbitMQ, an asynchronous queue middleware, for message transferring between services. To ensure the continuous optimization of scanning services, the system uses Jenkins continuous integration tools to automatically update the false positive filtering model.

The F1 value of the system is increased by 30% and 22% compared with Tscan-Code and Cppcheck, and it can effectively reduces false positive in the C++ source code vulnerability static scanning. This system improves the availability of static scanners for C++ source code vulnerabilities, reduces the number of false positive vulnerabilities, reduces the burden on developers to review vulnerabilities, and provides guarantees for delivering highly reliable code.

Keywords: C++ source code vulnerability static scanning, false positive filtering, machine learning, code similarity detection

目录

表 目 录

插图

第一章　引言

1.1　项目背景与意义

源代码漏洞静态扫描是指在不运行源代码的情况下，使用污点分析和数据流分析技术挖掘源代码中潜在的漏洞。源代码漏洞静态扫描一直是一种应用广泛的技术，得到众多商业平台采纳，例如 Coverity[1]、Fortify[2]、360 代码卫士[3]、Checkmarx[4]等。源代码漏洞静态扫描有很多优点。首先，早期发现漏洞，漏洞静态扫描可以挖掘源代码中漏洞确切位置，并且在项目开发早期进行，降低修复成本 [1, 2]。其次，全代码覆盖，可以检查到一些很少被运行到的路径。然后，无编译依赖，可以检测到使用不同于当前编译器版本才可能暴露出来的漏洞。最后，高可扩展性，可以检测大规模项目 [3, 4]。

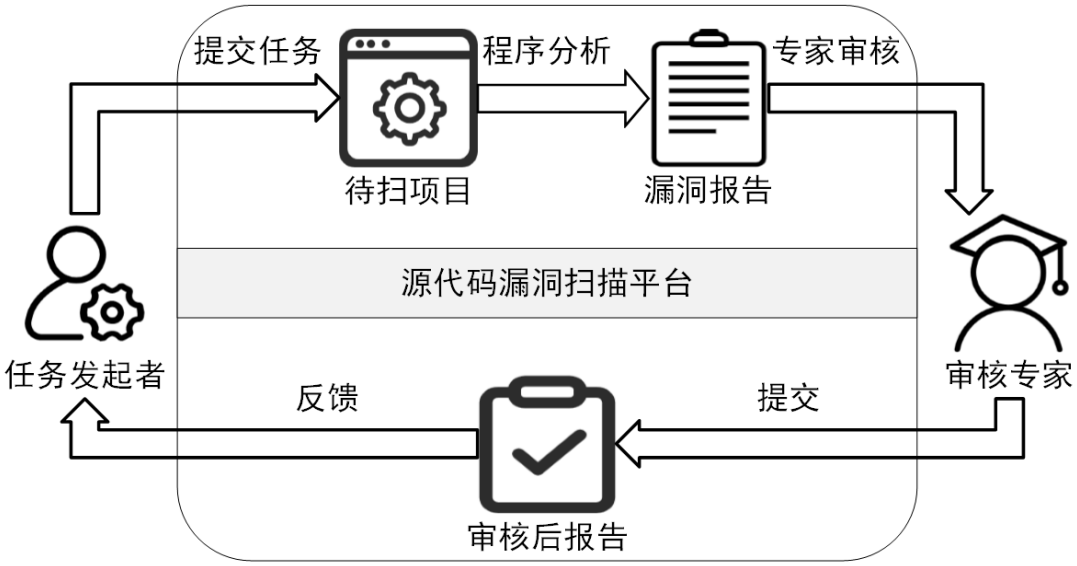


图 1.1: 传统源代码漏洞静态扫描系统流程图

图1.1展示了源代码漏洞静态扫描的主要流程。主要参与者包括任务发起者、漏洞静态扫描平台和审核专家。任务发起者通常是软件开发者，会在漏洞静态

[1]https://scan.coverity.com/

[2]https://www.joinfortify.com/

[3]https://www.codesafe.cn/

[4]https://www.checkmarx.com/

扫描平台上传待审项目源代码。漏洞静态扫描平台使用程序分析技术，包括数据流分析、污点分析和词法分析等技术，扫描源代码中潜在的漏洞，并生成初步漏洞报告。最终，审核专家会对初步漏洞报告进行再次审查，去除漏洞报告中存在的误报。

源代码漏洞扫描技术主要分为漏洞静态扫描和漏洞动态扫描两类 [5, 6]。漏洞静态扫描不涉及源代码的动态执行，并且可以在程序运行的早期阶段就检测出漏洞。漏洞动态扫描一般是基于真实或虚拟环境上运行的程序，一般需要加载外部依赖库以及编译源代码。随着现代软件的体积和复杂度的提升，漏洞动态扫描的时间开销和费用开销越来越高，并且很难覆盖到软件的全部功能点，并且产生的漏洞也很难追溯到源代码中的确切位置 [7]。相比于漏洞动态扫描，漏洞静态扫描不需要实际运行软件，而是使用污点分析和数据流分析等程序分析技术对软件的源代码进行快速、全面的分析。但是为了保证随着软件的体积和复杂度提升，漏洞静态扫描还能够快速的对软件源代码进行扫描并且尽可能不漏掉漏洞，静态扫描工具会使用近似的方法来提高扫描效率，这样做会产生大量的误报。开发者在使用漏洞静态扫描工具时，面对大量的误报，审核难度较大，往往会失去耐心，放弃参考静态扫描漏洞报告。但是软件中还是存在实际的漏洞，这些漏洞虽然可能数量较少，但是危害性高，可能造成很大的经济损失。因此，尽可能不影响扫描性能的前提下，降低漏洞静态扫描的误报率对于提高源代码漏洞静态扫描器的可用性是很有必要的。

本系统面向 C++ 源代码漏洞静态扫描业务，针对现有开源源代码漏洞静态扫描器存在的误报率高的问题，引入程序切片技术来对漏洞相关源代码进行简化，去除无关代码干扰；并且引入机器学习技术来对误报漏洞进行识别和过滤。误报漏洞过滤本质上属于机器学习领域中的二分类问题，因此将机器学习技术应用到误报过滤是可行的。首先，对 C++ 源代码进行漏洞静态扫描获得初步的漏洞报告。其次，结合程序切片技术对漏洞代码进行简化处理，方便提取源代码的特征。然后，使用机器学习技术根据提取到的源代码特征训练误报分类器，对初步的漏洞报告进行过滤，获取低误报率漏洞报告。接着，专家人员对漏洞报告进行误报审核，并对审核为误报的漏洞使用相似度算法，来寻找相似的漏洞进行过滤。漏洞代码依然是源代码，源代码相似度计算技术已经被广泛研究，因此计算漏洞代码相似度在技术上是可行的。最终，本系统为开发者降低时间成本和提供软件健壮性提供帮助。

1.2　国内外研究现状及分析

1.2.1　C++ 源代码漏洞扫描研究现状

为了检测源代码存在的漏洞，目前国内外已有许多针对 C++ 编程语言的静态代码漏洞扫描工具。静态代码漏洞扫描工具无需动态执行程序，而是依赖静态程序分析，检测代码中的潜在威胁。根据对源代码的处理方式，可将静态代码漏洞扫描工具分为两类。

第一类是基于模式的方法 [8]，其中的模式主要有三种生成方式，第一种是根据安全专家手动生成，比如开源工具 Flawfinder[5]和 ITS4[6]、商业工具 Checkmarx、Fortify 和 Coverity，这类工具通常具有较高的误报率或漏报率；第二种是根据已有的漏洞类型，如缺失检查漏洞、污点式漏洞和信息泄露漏洞，进行半自动生成模式；第三种模式是从类型无关的漏洞中半自动生成的，这些方法使用机器学习技术，依靠安全专家来定义特征，描述漏洞。

第二类是基于代码相似度的方法 [9, 10]。这类方法首先将程序划分为一些代码片段，然后以抽象的方式表示每个代码片段，包括标记、树、和图。最后通过在上一步中获得的抽象表示来计算代码片段之间的相似性。基于代码相似性的方法进行漏洞检测的优点是，单个漏洞代码实例足以检测目标程序中的相同漏洞。为了实现更好的漏洞检测效果，安全专家需要定义特征，以便自动为不同类型选择合适的代码相似性算法。然而，即使是具有专家定义特征的方法也无法检测到不是由代码克隆引起的漏洞。

上述静态代码漏洞检测工具各有优势，但仍然存在一些缺陷，如误报率高、可检测出的漏洞类型有限、准确率低等 [11, 12]。开发人员在查看漏洞报告时，需逐个验证漏洞是否真是存在，导致程序修复的成本高昂，因此本系统结合多种开源 C++ 代码漏洞检测工具，并针对这些工具的固有缺陷，结合机器学习的技术，进行静态代码检测工具的误报漏洞自动识别与过滤。

1.2.2　降低扫描器误报研究现状

目前国内外对于静态代码漏洞检测工具的误报报告问题已有一些研究。降低误报的方法可分为两类，一类是根据源代码的特征属性，进行统计分析，并预测漏洞报告为误报的概率；另一类是借助机器学习的技术，在静态代码漏洞扫描工具的漏洞报告基础上，进行误报的识别和过滤 [13]。

基于源代码特征属性的统计方法较为传统，Kremenek 和 Engler [14] 提出了 z 排序技术，针对静态代码漏洞检测工具生成的漏洞报告，根据每个漏洞报告被

[5]https://dwheeler.com/flawfinder/

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

判断为真实漏洞的准确性进行排序。Yüksel 和 Sözer [15] 使用 34 种不同的机器学习算法、10 种代码特征进行实验，误报报告分类的准确性达到 86%。

基于机器学习技术的误报检测方法的研究起步较晚，Yoon 等人应用支持向量机（SVMs）过滤掉误报，这是由一个名为 Sparrow 的商业静态分析工具生成的 [16]。ALETHEIA 提出一种学习误报漏洞报告特征的算法，使用聚类、基于规则的方法和贝叶斯等方法进行实验，改进静态代码漏洞检测工具的输出 [11]，以解决现有漏洞扫描工具误报过多的问题。

1.2.3　漏洞代码相似性检测研究现状

基于代码相似度的漏洞检测方法，针对每种漏洞类型，只需给定一个漏洞代码的实例，由于类似的程序可能包含相同的漏洞，可通过一系列代码相似度检测的技术来检测漏洞代码 [9]。此类方法通常包含下列三个基本步骤。

选择代码粒度。对于代码相似性的比较，需要在一定粒度级别上抽象代码，即需要合理提取代码，形成漏洞代码片段。代码粒度可按从细到粗划分为五个等级：上下文无关的补丁级别、程序切片级别、上下文相关的补丁级别、函数级别和文件级别 [17]。其中，上下文无关的补丁是在对比文件中，通过提取以“-”符号（指应该修补的行）为前缀的连续行，得到的代码片段。程序切片是通过程序依赖关系图，提取出存在控制依赖和数据依赖的一段不连续代码，程序依赖关系图也可以反应不同程序之间的关系。上下文相关的补丁代码，即是将以“-”符号（指应该修补的行）为前缀的连续行和无前缀的代码行合并在一起形成的代码片段。函数级别是指将一个单独的函数视为代码片段，文件级别是指整个代码文件不经过任何提取，直接视为整体的代码片段。

表示代码特征。每个代码片段可以通过文本、标识符、树和图来表示。文本的表示方法无法体现代码的语法和语义特征，不适用于漏洞代码识别。在基于标识符的表示中，通过编译器样式的词汇分析将源代码转换为“标识符”序列，一个标识符可能代表一行代码或一行代码中的部分组件 [17]。在基于树的表示中，树表示源代码中变量、常量、函数调用和其他标识符的语法结构，树结构的代码表示已用于代码克隆检测、漏洞检测、漏洞预测等研究领域。在基于图的表示中，函数表示为图，其中节点表示表达式或语句，边表示控制流和数据流，此类图结构可体现代码的语义特征，用于代码克隆检测、漏洞检测及缺陷检测。

确定相似度算法。有两种比较方法：向量比较 [18] 和基于特征比较 [19]。向量比较法首先将代码特征转换为向量，然后通过向量的相似度/距离算法来检测漏洞代码的相似性。基于特征比较方法与代码特征表示方法有关，如文本表示方法可用字符串匹配算法进行相似度比较，树和图的结构都可比较子树、子图

的相似性，进而进行漏洞代码检测。

上述方法适用于传统的代码相似度检测，但是漏洞代码存在独有的特征。对于粒度选择，漏洞代码经常是涉及多个函数，只检测单个函数间相似度无法找到所有的相似片段，如果检测文件间的相似度，粒度又偏大，因此需要一种适合检测漏洞代码相似度的粒度，本系统引入静态调用图，来将函数连接为功能，并在该粒度上检测相似性。对于代码特征表示，现有相似度检测算法是提取语法特征或者语义特征，但是这两种特征单独使用无法准确描述漏洞代码，本系统使用语法和语义特征的联合，来描述漏洞代码，并且加入了漏洞相关特征，包括分支数量、代码行数等特征。

1.3　本文主要工作内容

本文所设计和实现的 C++ 源代码漏洞静态扫描系统，其目标是针对现有开源工具误报率高、实用性不强的特点，帮助开发者和漏洞审核人员解决审核漏洞数量大、误报漏洞数量多的问题。

本系统通过考虑以下三个方面的设计来实现这一目标。

一方面在于融合不同开源漏洞静态扫描工具的结果，得到一个统一的、更加全面的漏洞扫描结果 [20]。目前使用广泛的 C++ 漏洞静态扫描开源工具有 Cppcheck 和 TscanCode，两款开源工具在检测功能、准确率和检测效率上面存在差别，在检测功能上存在一定的互补性。因此，本系统选用了这两款开源工具对 C++ 源代码进行漏洞静态扫描，并将两个开源工具的结果融合起来。这部分的难点在于不同开源工具漏洞扫描结果的融合。不同的漏洞静态扫描工具的输出都是工具的开发者自己定义的，没有按照一个统一的标准，两种开源工具的检测规则数量，输出结果粒度都有很大的差异。本系统融合不同开源工具提高源代码漏洞扫描的覆盖率，解决单一工具因为检测侧重点不同引起的覆盖率低的问题。

另一方面在于于过滤漏洞静态扫描结果中的误报，方便开发者进行漏洞审核。开发者在使用静态扫描工具时，往往会因为工具输出的大量漏洞列表且其中大部分为误报而感到困扰。在研究了误报漏洞代码后，作者发现误报漏洞是存在代码特征的，因此本系统提取误报漏洞源代码的代码特征，结合机器学习技术对误报进行过滤 [21–23]。这一阶段的难点主要在于代码特征的提取和误报过滤器的设计。误报漏洞代码存在一些相同的语法和语义特征。这些特征和误报漏洞存在关联，通过这些特征可以对误报漏洞和正确漏洞进行分类，从而将误报漏洞过滤掉。

最后一方面在于使用审核后的漏洞进行迭代反馈，提高漏洞过滤器的准确率。漏洞静态扫描器产生的漏洞列表经过误报过滤器过滤之后还是可能存在误报，因此需要专家进行审核。在专家审核之后，可以得到漏洞列表中的误报和正报信息。误报和正报信息可以反馈给误报过滤器，提高准确率。这一部分的主要难点是专家审核成本高，并且专家审核得到的少量数据，相比于用于训练误报过滤器的数据量很小，神经网络很可能对于少量数据不敏感，导致审核得到的结果对于误报过滤器的改进不显著。因此本系统使用相似性算法，在漏洞列表中寻找与审核为误报的漏洞相似的漏洞，并标记为误报。这样不仅可以直接提高误报过滤器的过滤效果，还可以扩增误报数据集的规模，使得迭代反馈提高误报过滤器准确率效果更好。

结合上述设计和需求，本系统基于 Docker 容器技术 [24] 开发，将 C++ 源代码漏洞静态扫描作为独立服务，向外部提供 Restful API，与其他服务实现松耦合。系统使用 HTTP 协议进行数据交换，支持文本、文件等多种格式的数据传输，具有方便灵活的优势。为实现漏洞静态扫描任务的分发，系统使用 RabbitMQ 实现消息队列进行支持。为实现误报过滤器的迭代更新后的自动化部署，系统使用 Jenkins 进行流水线部署。

对于 C++ 源代码漏洞静态扫描，本系统将其分解为静态扫描、误报过滤和迭代反馈三个阶段。静态扫描阶段中，系统会使用多个开源 C++ 源代码漏洞静态扫描工具对源代码进行扫描，并将结果融合得到一个统一的漏洞报告。误报过滤阶段中，系统使用训练集预先训练好的误报过滤器对上一阶段得到的漏洞报告进行过滤，得到经过过滤的漏洞报告。迭代反馈阶段中，首先漏洞专家对过滤后漏洞报告中的部分漏洞进行审核，筛选出其中的误报漏洞，接着使用相似度算法，寻找与误报漏洞相似的漏洞，并将这些漏洞数据反馈给误报过滤模型，提升误报过滤器的效果。这样可以提升源代码漏洞静态扫描的准确率，减少其中的误报漏洞数量，降低漏洞专家审核成本，便于开发者快速、准确的发现项目中存在的漏洞，并进行修复。

1.4　本文的组织结构

本文的组织结构如下：

第一章 引言部分。本章概述 C++ 源代码漏洞静态扫描背景以及目前面临的问题，国内外工业界现状，国内外误报过滤研究现状，以及本文在解决该问题上的主要思路。

第二章 技术综述。本章概述项目中使用的程序源代码切片技术、源代码特征提取技术、误报过滤器的训练和使用技术、误报漏洞迭代反馈技术、漏洞代码

相似度检测技术、容器技术 Docker 和消息队列技术 RabbitMQ。

第三章 需求分析和概要设计。本章首先根据实际用户场景对系统进行了功能需求和非功能需求分析；接着介绍了系统的模块划分，并从多个维度对系统进行了设计；最后对系统的各个核心模块从架构设计、流程设计、核心类图等角度进行分析与设计。

第四章 系统实现。本章主要是对系统的具体实现方式进行细节描述。按照模块的划分，对系统中的 C++ 源代码漏洞静态扫描模块、源代码特征提取模块、误报过滤模块以及误报漏洞反馈模型的顺序图和核心代码进行描述。最后还对系统中的主要功能界面进行展示。

第五章 系统测试。本章主要是对系统的功能和非功能需求进行详细的测试。首先对于系统的可靠性进行测试，保证系统可以给用户提供持续的服务；接着对系统提供的功能进行功能测试，保证系统能满足用户需求；最后对本文提出的降低漏洞扫描误报率的算法效果进行测试。

第六章 总结与展望。本章是对本文内容的总结以及未来深入方向的展望。对系统中漏洞扫描的整个流程进行了介绍，并对其中的误报过滤和误报漏洞专家审核反馈的流程进行了具体描述。对本文的主要工作进行阐述。最后对系统可以优化的三个点进行了阐述。

第二章　相关技术概述

2.1　程序切片技术简介

由于程序的某一个输出只与源程序中部分语句和控制谓词（control predicate）有关，删除其他的语句和谓词并不影响该输出的结果。程序切片是代码简化的一个重要技术手段，程序切片是将源代码减少到最小的代码量，同时保留原始程序的语法和结构属性 [25]。

2.1.1　程序切片的方法原理

程序切片技术的方法可根据代码的图形化中间表示进行分类。

第一类是根据程序的控制流进行切片。Weiser [26] 首次提出了基于程序控制流图 (CFG) 的静态代码切片方法。该方法首先将源代码转化为程序控制流图，在控制流图的基础上建立数据流方程，通过求解数据流方程来获得相应的程序切片。但该算法存在着一定的缺陷，如运行效率低、无法处理面向对象的复杂程序等问题，因为在求解一系列迭代方程方程时需消耗大量时间和空间，而且数据流方程较为复杂，即使存在很小的错误都可能导致切片的结果不准确。另外，由于传统的程序控制流图只局限于函数粒度，若函数内部存在着函数调用关系，则该算法无法计算准确的程序切片 [27]。

第二类是根据程序的依赖图进行切片。Karl 等首先提出了程序依赖图 (PDG) 来计算程序切片，Horwitz 等人 [28] 提出使用系统依赖图（SDG）来计算程序切片，相比于程序控制流图，这类图结构更加复杂，同时包含了程序的控制流和数据流，图中的节点表示程序语句或控制谓词，边表示依赖关系。此类方法是基于依赖图的图可达性完成的程序切片，相比于传统的数据流方程式计算，其效率和准确性都更高。

2.1.2　程序切片的主要流程

基于控制流和数据流的程序切片过程可分为三个阶段 [29]。

第一，使用程序分析技术，分析程序源代码之间的数据流关系和控制流关系，并依据该关系构造程序源代码对应的 PDG 或 SDG。

第二，根据程序切片准则，如前向切片或后向切片，从指定的节点出发，运用图可达性算法，计算该节点的依赖图，标记所有的依赖节点。

第三，根据依赖图和源代码映射关系，分析依赖图切片相对应的语句和控

制谓词，构建源代码关于切片准则的程序切片。

2.2　代码特征提取技术简介

在程序分析和编译器设计等领域，研究者开发了各种不同的代码表示方法，以解释和分析程序的性质。代码特征提取可视为程序分析的预处理阶段，选择合适的代码特征表示方式，对于后续训练深度学习分类模型有着重要的帮助 [30]。

2.2.1　抽象语法树

抽象语法树通常是由编译器的代码解析器产生的第一批中间表示，因此构成了生成许多其他代码表示的基础 [31]。这种树形结构准确地体现了程序中的表达式及语句的嵌套关系。一方面，与普通源代码相比，AST 是抽象的，不包括标点符号和分隔符等所有细节。另一方面，AST 可以用来描述源代码的词汇信息和句法结构，因为树结构上的每个节点可对应源代码中的所有标识符和符号，粒度较为精细。

抽象语法树是有序树，其中内部节点表示操作符，叶节点对应于操作数（例如常量或标识符）。虽然抽象语法树非常适合于简单的代码转换，但是由于其不能体现程序的控制流和数据依赖关系，它不适用于更进化的代码分析，例如检测死代码或未初始化的变量，同时，抽象语法树也无法体现条件判断、循环等特殊的代码结构 [32]。

2.2.2　程序控制流图

程序控制流图在程序分析领域有着广泛的应用，例如，检测已知恶意应用程序的变体和指导模糊测试 [33]。此外，它已经成为逆向工程中的标准代码表示，以帮助理解程序。程序控制流图有利于分析代码的结构特征，如程序中的条件判断在程序控制流图中体现为分支节点；程序中的循环判断在程序控制流图中体现为有闭环的图结构。

控制流图显式地描述了执行代码语句的顺序以及需要满足的条件，并且描绘了程序所有的可达路径。在程序控制流图中，每个节点表示一个代码执行语句，节点通过有向边的连接，以表示控制的转移。然而，相比于抽象语法树，程序控制流图中节点的代码粒度较粗，因此控制流图在分析代码的结构上有明显优势，在分析代码的语法细节上有一定局限 [34]。

2.3　基于机器学习的漏洞误报分类模型简介

为了过滤漏洞静态检测工具的误报，本系统使用基于机器学习的方法来训练误报过滤模型，将误报过滤问题转化为二分类问题。基于机器学习的方法中

的一个关键因素就是输入数据，本系统使用的数据是漏洞静态检测工具产生的漏洞报告和被扫描源代码的代码特征，包括语法特征、语义特征和代码结构特征 [35]。基于机器学习的方法的另一个关键因素就是模型的选择，常见的模型可以分为两类：传统机器学习分类模型、基于神经网络的分类模型。基于神经网络的分类模型在数据规模较大的时候效果是优于传统机器学习分类模型，但是在小数据集情况下可能无法收敛，这时候传统机器学习分类模型的效果就更好一些。因此，本系统在面对小数据集情况下，选择使用传统机器学习分类模型，当数据集增长到一定规模时，使用基于神经网络的分类模型。

2.3.1　传统机器学习分类模型

传统机器学习分类模型包括支持向量机（SVM）[36]、决策树算法 [37]。这些模型通过学习特征空间中的边界或者特征空间与派生空间的映射来进行分类。一旦训练好了，这些模型就可以通过在决策空间中记录一个引入特征的位置来进行预测。同时，传统机器学习分类模型的可解释相比于基于神经网络的方法要强一些。

SVM 算法在特征空间搜索一个超平面，这个超平面能够根据最大边界条件来对标签数据进行最优划分。给定一组特征 x_i，标签值为 $y_i \in \{-1, 1\}$，SVM 负责求解一个超平面 w，该超平面求解是严格的凸优化问题：

$$\min_{w,\gamma} \frac{1}{2}\|w\|^2 + \lambda \sum_i \max\left(0, 1 - y_i\left(x_i^T w - \gamma\right)\right) \tag{2.1}$$

观察公式2.1可知，该方法通过一个鲁棒性的分类方法来对正则项 $\|w\|^2$ 加权，当预测标签 $x_i^T w - \gamma$ 符号为正时，i 为 0，当符号为负时，呈线性增长。该问题是一个严格的凸优化问题，所以可以保证总能有一个唯一解，通过迭代计算的方法可以很容易的找到解。

SVM 的一个弱点是模型空间的丰富性和复杂度，线性分类方法的性能总是有限制的 [38]。为了解决这个问题，基于核函数的 SVM 神经网络模型根据从数据派生的特征训练了多层模型。基于核函数的 SVM 模型使用了核函数，将特征映射到另一个特征空间 Φ，并且允许对内积 $\Phi(x_i)^T \Phi\left(x_j\right)$ 进行简单评估。

2.3.2　基于神经网络的分类模型

基于神经网络的分类也已经在误报分类上广泛应用，包括卷积神经网络（CNN）[21]、长短期记忆网络（LSTM）[13]、深度神经网络（DNN）。这些神经网络主要的区别在于中间神经元的结构，不同的误差传播机制。因为源代码的上下文之间是有强关联的，CNN 可以很好的提取到这种上下文信息，本系

统选择使用 CNN 模型。

CNN 是一个包含卷积层的神经网络。卷积层包含一组等价的神经元，通常称为过滤器，它们只连接输入数据中的一个局部区域，多个过滤器实例应用于整个输入数据，以给定的间隔移动这些过滤器 [39]。CNN 模型可以学习源代码中的结构模式。误报模式与固定大小的特定标识符出现的部分顺序有关，但是标识符之间的距离有可能不一致。

使用 CNN 模型进行误报分类的具体步骤如下。源代码片段通过标识符和正则化转换为标识符序列，接着每一个标识符使用 word2vec [40] 算法转化为特征向量。每个标识符对应的向量堆叠在一起，每个源代码片段对应为一个特征矩阵。由于每个代码片段的标识符的数量不一样，在矩阵的顶部填充向量，使所有的矩阵维度相同。使用卷积核在特征矩阵上做卷积操作，使用最大池化算法对数据进行降维，最后的输出层使用 *sigmoid* 激活函数，来对正报和误报漏洞进行二分类。

2.4　误报漏洞模式的构建与迭代反馈简介

针对静态代码漏洞扫描工具的误报率高的问题，构建了基于深度学习的自动识别误报模型。为了降低漏洞专家审核成本，在模型过滤误报基础上，采样部分漏洞给专家进行审核，并构建误报漏洞模式。当有新的漏洞报告产生时，可先与误报漏洞模式进行匹配，完成模型的迭代反馈更新，不断提高自动识别误报漏洞模型的准确率。

2.4.1　误报漏洞模式的构建

在通过程序切片获取了精简代码之后，需提取精简代码的相关特征属性，本系统使用程序切片体和相关特征属性来构建误报漏洞模式。构建误报漏洞模式的目的是识别代码库中反复出现的特殊的源代码结构，该结构会导致静态代码分析工具产生误报 [8]。构建漏洞模式可以对导致误报的漏洞结构进行描述。这种方法类似于记录软件设计模式或编目软件缺陷。在所有这些场景中，目标是从大量实例中捕捉其中的相似性，根据此构建的模式可以检测新的代码是否存在同种模式，进而将同种模式的代码段标记为误报漏洞。以下是代码特征属性。

代码片段：切片后的精简代码片段。

名称：一个简短的短语，描述误报模式的代码结构。

工具：此属性记录产生误报模式的静态代码分析工具列表。

频率：用于记录训练样本中指定类型的误报漏洞出现的频率。

描述：对代码结构的非正式、概要的解释，说明导致误报的原因。

漏洞警告信息：来自静态代码分析工具的警告消息。

变更：对精简的代码片段的微小修改，以及静态代码分析工具结果。

2.4.2　迭代反馈的过程简述

迭代反馈是根据专家审核的结果，来改进误报漏洞过滤器的精度 [41]。具体流程如下所示。

使用开源静态代码扫描工具检测源代码，形成漏洞报告后，使用误报漏洞报告分类模型进行预测，输出模型判断的正报与误报结果。

根据模型输出的正报与误报结果，进行采样选取，例如选择置信度低于指定阈值的漏洞报告，交给安全专家进行审核，专家需审核该代码片段是否存在漏洞，若无漏洞需用自然语言描述其产生误报漏洞的原因。

根据专家的自然语言描述，针对高频出现的漏洞类型进行归纳，构建误报漏洞模式，并针对不同漏洞类型设计不同的误报检测器。当有新的待检测代码出现时，首先使用误报检测器进行预判，若存在已知假阳性模式的代码段，则判定为其漏洞为假阳性结果。

2.5　漏洞代码相似度检测方法

在不同的项目中，同一种类型的漏洞具有相似形式的代码表示，根据代码语法、语义特征的相似度检测，可根据已知的漏洞代码对待检测的代码进行漏洞代码相似度检测，进而识别出相似的漏洞代码 [42]。本系统根据专家评审后的标准漏洞代码 [43]，并做为漏洞代码样本，提取语法和语义特征，搜索相似代码片段，辅助进行源代码的漏洞检测。

2.5.1　代码相似度特征提取

为了有效进行漏洞代码相似度的检测，设定漏洞代码的粒度并提取代码特征是十分有必要的，在整个代码文件中，有大量与漏洞无关的语句，这部分内容会影响代码相似度检测的准确性，因此本系统首先定位出现漏洞的语句，并根据程序的控制依赖和数据依赖关系，提取与之有依赖关系的语句，做为漏洞代码片段，即代码粒度为代码片段。在代码片段的基础上，需对代码进行标准化处理，例如删除注释等无关内容，将变量名和函数名进行统一，根据标准化的代码片段，可提取代码段的标识符序列，进而使用词嵌入的方式形成代码的特征向量；同时，可根据代码片段生成代码段的抽象语法树，形成树形的代码语法特征结构。

2.5.2　代码相似度计算

代码相似度的算法可分为两类。一类是基于代码的特征向量，比较两个代码片段对应的特征向量间的距离，如欧几里得距离、曼哈顿距离等 [19]。若距离小于某一阈值时，则判定两个特征向量对应的代码片段相似；若大于此阈值则认为该代码片不相似。基于距离的方法的优点是计算方便、快速，但是也存在一些问题，包括阈值合理设定比较困难和距离特征过于简单难以描述代码片段之间是否相似。

另一类是基于机器学习的技术 [18]。该技术包含两个关键点，一个是代码特征的选择，另一个是机器学习模型的选择。计算代码相似度中使用的代码特征一般是根据相应的代码表示来提取的。常用的代码表示有标识符、字节码、抽象语法树和程序依赖图。本系统选择使用抽象语法树来获取源代码的语法特征，程序依赖图获取源代码的语义特征，标识符的信息包含在抽象语法树内部，字节码需要编译不适合本系统。

对于标识符代码表示，代码片段是表示为标识符流。对于抽象语法树代码表示，先序遍历抽象语法树并将所有的叶子节点提取出来，构成节点流，抽象语法树的节点包含节点类型、节点代码片段等信息。对于程序依赖图代码表示，是由控制流图和数据流图构成。图结构的抽象程度较高，不需要进行特殊处理。标识符代码表示和抽象语法树代码表示，都是字符串序列，直接使用 word2vec 算法，提取每个字符串的特征向量，并对所有字符串特征向量使用平均池化获取整个代码片段特征向量。对于程序依赖图，使用可以获取图结构特征的 graph2vec [44] 算法，得到控制流图和数据流图的图特征向量，进行平均池化后得到整个程序依赖图的特征向量。

基于机器学习的方法进行代码相似度检测也是将问题转化为二分类问题。该部分与误报过滤模型在方法上基本相似，但是不同的模型的适用场景不一样。DNN 模型很适合进行代码相似度检测，可以将语法和语义特征拼接在一起成对输入，足够捕获到源代码中与代码相似度相关的特征，并且模型结构较简单，效率很高。

2.6　容器技术 Docker

2.6.1　Docker 简介

本系统使用 Docker 容器技术来对 C++ 源代码漏洞静态扫描系统进行封装，降低系统耦合性。Docker 的中文含义是“搬运工”，其搬运的东西就是镜像（image），运行中的镜像就是容器（container） [24]。Docker 是一个基于 LXC

（Linux container）的高级容器引擎，可以直接将软件以及软件的环境依赖直接部署到容器内部，并且可以实现软件和数据分离，保证数据的独立性。Docker 镜像是分层的，支持在基础镜像上添加新功能，不需要改动原有的功能，扩展起来很方便。Docker 具有跨平台的特性，支持在主流的操作系统 Windows 和 Unix 系统上运行。

构建 Docker 镜像通常需要经历以下几个步骤：首先，编写需要打包成 Docker 镜像的软件，软件提供相应的功能；接着，编写 Dockerfile 脚本文件，该脚本主要负责安装软件的环境依赖以及设置镜像的入口；最后，根据 Dockerfile 脚本构建镜像，运行中的镜像功能上和实际的软件没有区别。

2.6.2　使用 Docker 技术的优势

Docker 容器技术作为现在使用最广泛的容器技术有很多优点：

第一，高资源利用率。相比于虚拟机技术，Docker 容器技术不需要虚拟硬件环境，对系统资源的利用率更高。在应用执行的内存开销、文件读写速度上，也比传统虚拟机技术更高效。

第二，一致运行环境。开发者在开发过程中经常遇到的一个问题就是开发环境、测试环境、生产环境不一致，导致程序在开发完成之后无法立刻应用到测试、生产环境。Docker 容器技术提供了一致性的环境，实现三个环境的统一，可以大幅度节约开发成本。

第三，低维护成本和高扩展性。Docker 容器采用分层存储的技术，每一层的功能可以轻松复用，每一层的功能相对独立，层与层之间通过简单的数据交换降低耦合性，增加新功能时需要在原有基础上增加新的层即可，扩展方便。

2.7　消息中心 RabbitMQ

2.7.1　RabbitMQ 简介

本系统使用 RabbitMQ 消息队列组件，在 C++ 源代码漏洞静态扫描服务和扫描器之间进行通信，RabbitMQ 是目前主流的消息队列组件之一。RabbitMQ 的全称是 Rabbit Message Queue，基于 Erlang 开发的，实现了高级消息队列协议（AMQP），该协议是一个进行异步消息传递所使用的应用层协议 [45]。AMQP 为三层协议，其中模型层负责实现业务逻辑；会话层基于可靠传输负责客户端和服务器端之间的异步信号传输；传输层负责二进制流的传输，提供帧处理、信道复用、差错检测。三层协议互不干扰、共同协作，实现异步消息的可靠传输。

2.7.2　RabbitMQ 的优势

RabbitMQ 技术作为主流的消息中间件技术存在很多优势：

第一，高并发性能。RabbitMQ 是基于 Erlang 语言开发的，Erlang 对于高并发支持很好，RabbitMQ 性能较好。吞吐量达到万级，可以很好的支持常用场景。

第二，解耦子系统。子系统之间使用异步队列进行通信，系统间不需要相互等待，降低系统的整体延时，提升用户使用系统时的体验。

第三，高社区活跃度。该消息中间件的社区活跃度很高，问题反馈机制建立的很完善，发现问题后能够得到及时的解决，对于开发者很友好。

2.8　本章小结

本章主要概述本系统研发过程中使用到的相关技术和算法。首先，对提取代码片段，去除无关代码的程序切片技术进行概述。其次，对提取抽象语法树、程序控制流图等代码特征的技术进行概述。接着，对基于机器学习进行漏洞误报检测的两类方法，包括基于传统机器学习方法和基于深度学习方法进行概述。然后，对构建误报模式使用的属性和结合专家审核进行迭代反馈的流程进行了概述。再然后，对相似度检测中使用的代码特征提取技术和基于机器学习进行代码相似度检测的方法概述。最后，对系统中使用 Docker 容器技术和 RabbitMQ 消息中间件的流程和优势进行了概述。

第三章　C++ 源代码漏洞扫描系统的需求与设计

本章将要概述 C++ 源代码漏洞扫描系统的需求分析和设计。首先，本章对系统的目标和设计方式进行了整体概述。其次，本章对系统的非功能需求和功能需求进行了分析，并设计了相应的系统用例。然后，根据系统的目标和设计要求，对系统进行了整体架构设计并划分为多个子模块。最后，对系统中的各个子模块分别进行架构设计以及相应的流程分析。

3.1　系统整体概述

由于 C++ 源代码漏洞扫描系统自身机制的问题和源代码复杂度高的原因，导致系统在对源代码进行漏洞扫描时遇到漏洞报告数量大、误报漏洞多等困难。本系统针对现有 C++ 源代码漏洞静态扫描过程中存在的问题和难点，利用程序切片技术和机器学习技术对误报进行过滤来降低开发者审核漏洞的难度，帮助开发者交付健壮性更高的程序。

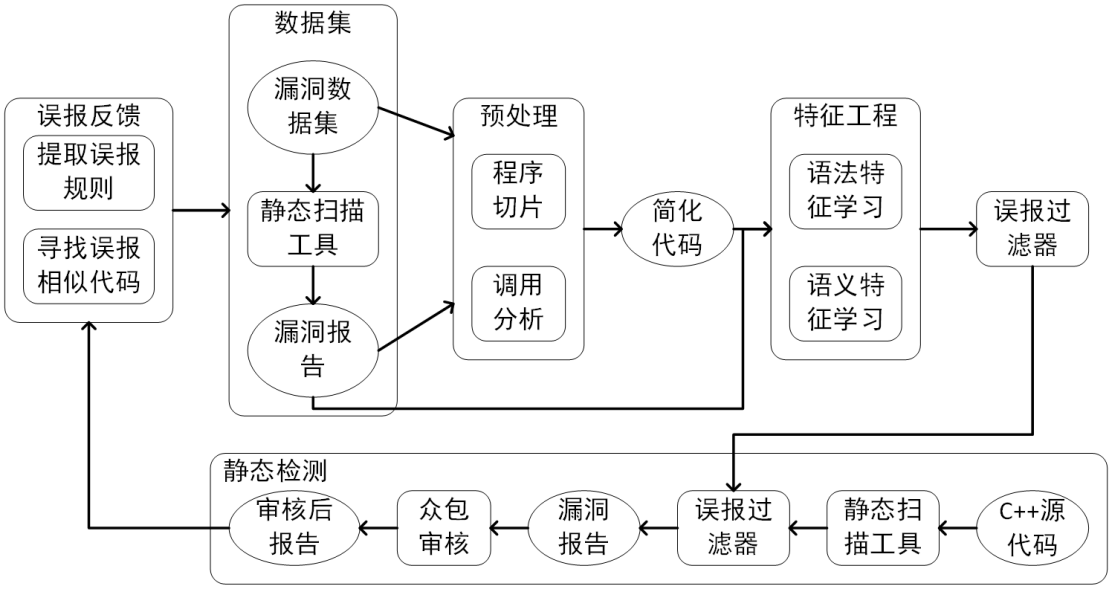


图 3.1: 源代码静态漏洞扫描流程图

图3.1是本系统的进行 C++ 源代码漏洞静态扫描的处理流程图。首先，对于待审核 C++ 源代码使用融合多种工具的源代码漏洞静态扫描工具进行扫描，得

到初步的漏洞报告；接着，使用相同的源代码漏洞静态扫描工具对源代码漏洞数据集进行扫描，根据得到的数据训练误报过滤器，并使用误报过滤器对初步漏洞报告进行过滤，得到过滤后的漏洞报告；最后，漏洞专家对过滤后的漏洞报告进行人工审核来寻找其中的误报漏洞，并使用相似度算法在源代码中寻找与误报漏洞相似的代码片段进行二次过滤，并将这部分数据加入到漏洞训练集中对误报过滤器迭代训练。通过融合多种 C++ 源代码漏洞扫描工具，可以有效地弥补单个工具覆盖率低的问题，有效地提升漏洞覆盖率为后续的误报过滤提供更全面的漏洞报告。通过使用 C++ 源代码漏洞静态扫描工具对带标注漏洞数据集进行扫描，可以获取误报漏洞数据集，并用该数据训练误报过滤模型，可以有效地过滤掉大部分误报漏洞，并提升漏洞扫描系统的可用性和减轻开发者使用漏洞扫描系统的负担。通过使用相似度算法寻找专家审核后的误报漏洞代码的相似漏洞代码，可以扩增漏洞数据集并使用该数据集提升误报过滤器的准确率，该过程可以迭代进行。最后，通过上述步骤，系统显著地的降低了源代码漏洞扫描中的误报数量，减少了漏洞报告的大小，降低了开发者审核漏洞的人力成本，提升了代码质量，为公司的稳定发展做出了一定的贡献。

3.2　系统需求分析

3.2.1　功能性需求

功能需求按照三个部分进行分析，分别是 C++ 源代码漏洞静态扫描、误报过滤和迭代改进误报过滤器。主要需求如表 3.1所示。

在 C++ 源代码漏洞静态扫描流程里，首先，系统需要从浏览器获取待扫描的源代码。其次，系统需要对源代码进行预处理，包括去除注释代码，去除非 C++ 源代码文件。然后，系统使用漏洞扫描工具集对源代码进行扫描，获取漏洞扫描报告。对于漏洞报告中漏洞对应的源代码需要先抽取程序切片，并依据程序切片提取其中的统计特征、语法特征、语义特征，方便后续的误报过滤和相似度计算。扫描完成后，系统会使用误报过滤器对漏洞报告中的误报进行过滤，来减少漏洞报告中的漏洞和误报数量。在进行了初步的自动误报过滤后，系统将漏洞报告发布为审核任务，给漏洞专家进行人工审核，审核其中存在的误报，目标是使漏洞结果尽可能准确。人工误报过滤后，系统会使用相似度算法，搜索与人工审核为误报的漏洞代码相似的漏洞项，并将此数据标注为误报。得到扩增误报漏洞后，系统会使用此数据重新训练误报过滤器，提升误报过滤器的准确率，下次过滤误报时相同的误报项可以被过滤掉。

表 3.1: 功能需求列表

ID	需求名称	需求描述
R1	源代码漏洞静态扫描	用户点击“开始扫描”，系统开始对 C++ 源代码的扫描。系统使用多个开源 C++ 源代码漏洞静态扫描工具对源代码分别进行扫描，并将结果融合起来得到更丰富的漏洞列表，使漏洞项尽可能完善。
R2	查看漏洞扫描任务列表	用户可以查看所有的漏洞扫描任务，默认按照扫描时间进行排序，可以看到每个扫描任务的进度，包括未开始、进行中、完成、失败等状态。
R3	查看漏洞扫描任务初步报告	用户选择任一漏洞扫描任务，可以查看由源代码漏洞静态扫描器生成的扫描报告，包括扫描任务名称、扫描时间、使用的开源扫描工具集以及漏洞列表。
R4	训练误报过滤器	当系统新增误报数量达到阈值后，系统开始训练误报过滤器。系统有初始漏洞代码集，使用源代码漏洞静态扫描工具集对该漏洞代码集进行漏洞扫描，获取误报数据。基于带标签的漏洞数据来提取语法特征、语义特征，并用来训练机器学习模型，可以获取用于过滤误报的二分类模型，来对漏洞报告进行过滤获取去除误报的过滤后报告。
R5	专家审核漏洞报告	专家可以对漏洞报告进行人工审核，审核内容主要是漏洞项，可以按照漏洞风险等级进行排序，可以按照漏洞类型进行分类。漏洞项包含漏洞名称、风险等级等信息。
R6	漏洞代码相似检测	系统对于审核为误报的漏洞，使用相似度算法搜索与误报相似的代码片段，并将这部分数据也加入到误报数据集中，达到扩增数据集的目的。
R7	查看过滤后的漏洞报告	用户选择进行误报过滤后，可以查看过滤后的漏洞报告，报告中也包含被过滤掉的漏洞以及过滤原因，即被专家审核过滤或被误报过滤器过滤。漏洞报告支持导出静态网页文件，可以在浏览器打开。
R8	代码漏洞报告管理	管理员在漏洞报告管理页面，选择“编辑漏洞报告”，可以对漏洞报告进行编辑，包括漏洞类型、漏洞风险等级、漏洞对应源代码。管理员可以将指定漏洞标记为误报，标记后的漏洞不在导出的报告中出现。

在扫描任务管理页面，可以看到所有扫描任务的进度，以及扫描任务使用的漏洞扫描工具集。对于扫描完成的任务，可以选择转人工审核，将扫描任务以及扫描结果转到人工审核，便于对漏报报告进行进一步确认。

在漏洞报告管理页面，用户首先可以查看所有的漏洞报告中的所有漏洞项，并将未审核的漏洞项置顶，以及可以按照漏洞风险等级排序，并可以按照漏洞类型进行筛选，方便用户可以统计漏洞报告结果。

在漏洞审核页面，可以看到每一个漏洞项的详细信息，包括漏洞名称、漏洞类别、风险等级、相关代码切片、建议解决方案等信息。漏洞专家对漏洞项进行审核，一旦判定为误报，漏洞项的状态就变更为误报漏洞。同时该页面可以下载漏洞报告，方便漏洞专家在无网络环境下进行审核。

3.2.2　非功能性需求

表 3.2: 非功能需求列表

需求名称	需求描述
可扩展性	系统应该对调用漏洞扫描工具进行扫描的接口进行抽象，方便增加新的扫描工具。系统应该对加载误报过滤模型进行误报过滤的部分的数据和程序分离，方便模型的更新、替换。
健壮性	系统上线前，应该使用测试集和相应的测试用例进行测试，通过测试的版本才可以发布。
可维护性	系统运行期间的维护需要尽可能的简化，运维人员不需要去大量阅读项目源代码就可以进行简单的增加漏洞扫描工具、更新误报过滤模型等操作，系统将功能进行模块化，对外提供统一的接口，并将更新误报过滤模型等操作固化为脚本，实现一键操作。
可用性	系统应该符合交互设计中的规范，提供帮助文档，并为界面上的输入框和按钮等控件提供文字说明，并提供便捷的搜索、筛选等功能，保证系统的易用性。

本系统的非功能性需求如表 3.2所示。为应对漏洞扫描工具集的扩展，系统需要将漏洞扫描工具设计为插件的形式，当有新的漏洞扫描工具时，可以直接加入系统，当误报过滤模型更新时，可以自动替换，不必修改太多的代码，增强系统的可扩展性。为应对用户上传内容的任意性，系统需要对各种异常情况进行处理防止系统崩溃，并给出合理的反馈。同时使用测试集对系统进行测试，提高系统健壮性。为减轻系统后期的维护压力，系统内部的各个子功能都进行

了模块化，各模块也都使用容器化技术进行打包，提高系统的可维护性。本系统需要保证界面简洁、操作简单，界面上的每个表单都有明确的指向性，用户操作时没有二义性，不会误导用户，保证系统的易用性。

3.2.3　系统用例图

依据上文所述本系统功能性需求的描述，得出如图 3.2所示的系统用例图。本系统可以分为 6 个用例，分别是查看扫描任务列表、触发漏洞扫描、触发误报过滤、查看漏洞详细信息、查看漏洞审核列表、审核漏洞，用户分为系统用户和审核专家。

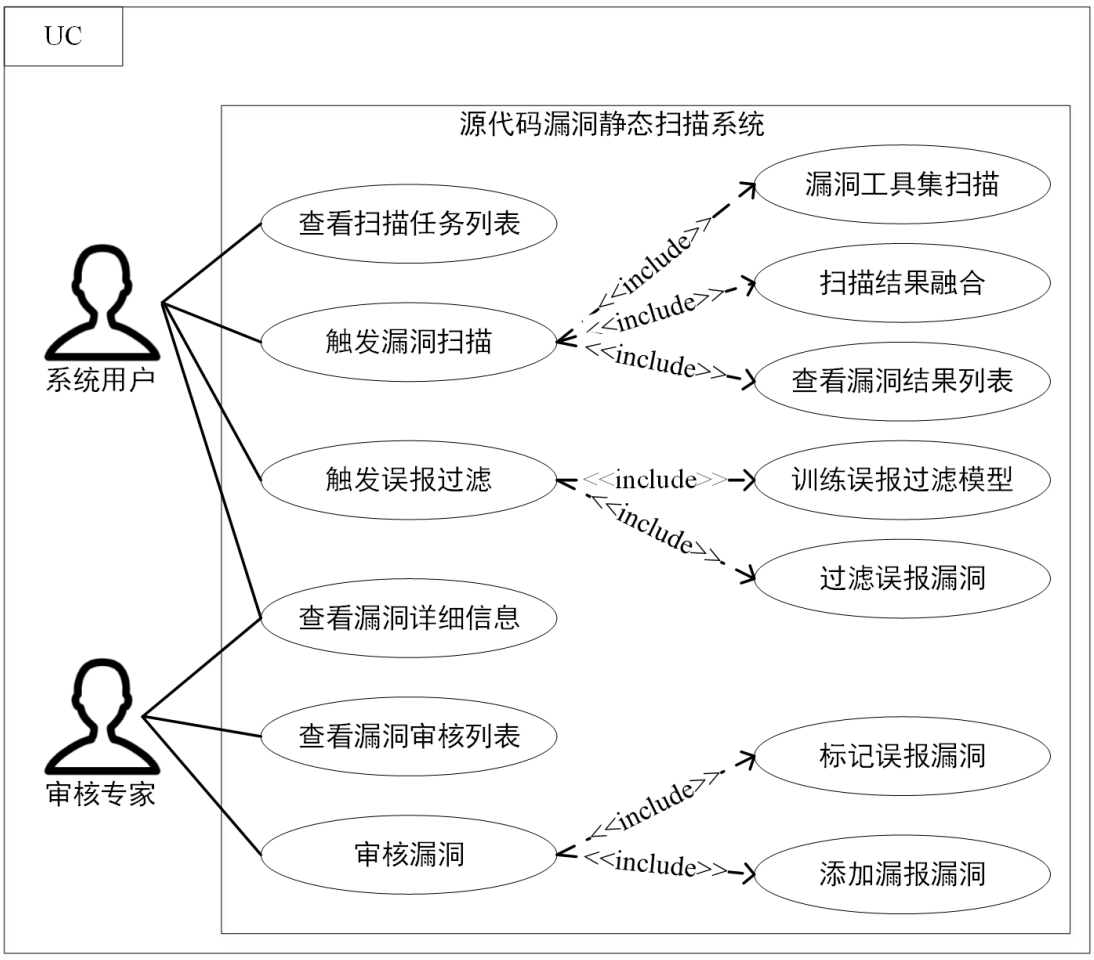


图 3.2: 系统用例图

3.2.4　系统用例描述

查看扫描任务列表是系统的主页面，系统用户通过该功能快速对漏洞扫描任务状态进行了解，方便系统用户快速发现并开始未开始的扫描任务，因此需要将未开始的任务放在最显眼的位置上，并将其状态标注为灰色表示未开始，有利于系统用户掌握漏洞扫描系统当前的状态。同时，因为扫描时间与文件中的函数数量、函数间调用关系的复杂度有关，所以扫描时间与文件大小不是成正比，难以用进度条来精确表示扫描进度，本系统中直接使用文字来表示漏洞扫描任务的状态。用例描述如表 3.3所示。

表 3.3: 查看扫描任务列表用例描述

UC1	查看扫描任务列表
参与者	系统用户，目的是查看漏洞扫描任务的进度
触发条件	系统用户进入查看扫描任务列表页面
前置条件	系统用户已经过认证和授权
后置条件	无
优先级	高
正常流程	1. 系统用户登录完成； 2. 系统显示所有的漏洞扫描任务，并将未开始的漏洞扫描任务按照时间顺序置顶显示； 3. 系统用户查看漏洞扫描任务的状态； 4. 系统显示已完成任务、未开始任务、进行中任务数量。
特殊需求	已完成任务、未开始任务、进行中任务的状态使用不同颜色的字体。

触发漏洞扫描是本系统最重要的流程之一。虽然系统用户的操作很简单，只需要点击开始扫描，但是这一步的背后进行的处理很重要。触发漏洞扫描后，首先系统会调用集成好的两个开源漏洞扫描工具：TscanCode 和 Cppcheck，分别对用户上传的源代码进行漏洞扫描；接着系统会将两份不同的漏洞报告映射到本系统选定的漏洞标准，即 CWE 数据库[1]；最后，系统根据漏洞位置、漏洞类型对映射后的漏洞报告进行融合，产生一份统一的漏洞报告。系统后续进行的误报过滤都是在这个基础上进行的，而且这部分操作不需要用户参与，因此设计的尽可能简单，给用户最好的体验。用例描述如表 3.4所示。

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

表 3.4: 触发漏洞扫描用例描述

UC2	需求描述
参与者	系统用户，目的是触发误报过滤的任务，启动自动化漏洞扫描
触发条件	系统用户选择“开始扫描”
前置条件	系统用户已经过认证和授权
后置条件	漏洞扫描完成，初步漏洞扫描结果持久化存储
优先级	高
正常流程	1. 系统用户在漏洞扫描任务列表页面选择“开始扫描”； 2. 系统显示进行中，扫描结束后显示查看报告入口，列表显示漏洞扫描任务 ID； 3. 系统用户点击查看报告按钮，显示漏洞报告页面。

触发误报过滤是本系统的核心功能之一。每次执行扫描任务时，误报过滤会在漏洞扫描之后自动执行，同时用户也可以选择主动触发此流程。将这个流程单独出来的一个重要的原因就是，误报过滤模型训练时所使用的数据集是会随着系统的使用而发生变更的，包括增加新的误报数据、增加新的漏洞数据。随着训练使用的数据集增多，模型需要重新进行训练，以不断提高模型的准确率。尽管过滤误报洞时，也会过滤掉少部分真实漏洞，但是相比于过滤的误报，这部分漏洞是可以接受的，并且过滤的误报在系统中也是可以查看的，系统中设置了误报漏洞列表。用例描述如表 3.5所示。

表 3.5: 触发误报过滤用例描述

UC3	需求描述
参与者	系统用户和管理员，目的是触发误报过滤模型训练和使用
触发条件	系统用户选择“重新过滤误报”
前置条件	系统用户已经过认证和授权
后置条件	漏洞数据集发生变化
优先级	高
正常流程	1. 系统用户在漏洞扫描任务列表页面选择“重新过滤误报”； 2. 系统显示数据集变更情况，并在数据集发生变更时，管理员重新启动误报模型训练； 3. 模型训练结束后，使用测试集对模型进行评估，并反馈测试结果。

查看漏洞详细信息是本系统的基础功能。主要展示有关源代码中存在的漏洞的详细信息，包括漏洞名称、风险等级、解决方案等信息，同时提供基础的排序、搜索、过滤功能，为系统的使用者提供便利。用例描述如表 3.6所示。

表 3.6: 查看漏洞详细信息用例描述

UC4	需求描述
参与者	系统用户和漏洞专家，目的是查看有关漏洞的详细信息
触发条件	系统用户和漏洞专家在漏洞列表中选择特定漏洞
前置条件	系统用户是该扫描任务的拥有者，漏洞专家已登录
后置条件	无
优先级	高
正常流程	1. 系统用户和漏洞专家在漏洞列表页面选择特定漏洞； 2. 系统显示该漏洞扫描任务结果中漏洞列表和漏洞相关类型以及风险等级的统计信息； 3. 系统用户和漏洞专家根据风险等级和漏洞类型进行查看； 4. 系统用户和漏洞专家按名称进行检索。

查看漏洞审核列表是本系统供漏洞专家使用的基本功能。主要是为漏洞专家展示需要审核的漏洞列表，包含被误报过滤器过滤掉的漏洞，为漏洞专家提供最全的漏洞列表。同时，还按照误报过滤模型计算出来的为误报的概率进行排序，方便漏洞专家按照疑似误报的概率进行审核，以及基础的分类查看、搜索功能。用例描述如表 3.7所示。

表 3.7: 查看漏洞审核列表用例描述

UC5	需求描述
参与者	漏洞专家，目的是查看漏洞审核的概要情况
触发条件	漏洞专家进入漏洞审核页面
前置条件	漏洞专家登录并领取该漏扫审核任务
后置条件	漏洞扫描和误报过滤已完成
优先级	高
正常流程	1. 漏洞专家选定项目进入其漏洞审核页面； 2. 系统显示该扫描任务的待审核漏洞列表； 3. 漏洞专家按照风险等级对待审核漏洞排序； 4. 漏洞专家根据疑似误报概率和漏洞类型进行查看。

审核漏洞是本系统需要人工参与比较多的流程。漏洞专家根据漏洞名称、风险等级、漏洞相关代码的程序切片对漏洞进行审核，判断该漏洞是否是误报。其中程序切片是以源代码的控制流图和数据流图的形式展现的，去除掉了无关代码的干扰，方便漏洞专家快速掌握程序的结构，厘清程序中的调用关系，对于误报是否是判断很有帮助。用例描述如表 3.8所示。

表 3.8: 审核漏洞用例描述

UC6	需求描述
参与者	漏洞专家，目的是审核漏洞列表中存在的误报
触发条件	漏洞专家选中特定的漏洞，进入漏洞审核界面
前置条件	漏洞专家登录并进入相应的漏洞列表界面
后置条件	无
优先级	高
正常流程	1. 漏洞专家选中特定的漏洞； 2. 漏洞专家根据风险等级对漏洞进行从高到低的排序； 3. 漏洞专家查看漏洞相关的描述以及相应的程序切片和代码是否符合； 4. 漏洞专家将误报漏洞标记为误报。

3.3　系统总体设计

3.3.1　系统整体架构设计

C++ 源代码漏洞静态扫描系统的系统架构如图 3.3所示，下面按模块之间的关联从工程角度进行详细阐述。

本系统前端页面使用 AngularJS 模板引擎，具有强大的模板功能，能够完成比较复杂的功能，同时有完善的 MVC 框架，开发方便快捷。前端项目的构建使用的 Gulp，该工具遵循“代码优于配置”得原则，引入流的概念来简化前端项目复杂的构建。前后端之间的通信使用的 Ajax 完成异步请求，进行延迟加载和局部更新，避免请求后页面卡住的情况。UI 组件库使用的是轻量级前端框架 layui.js，基于 DOM 驱动并拥有自己的模式，符合后端人员的开发习惯。

服务端部分向外提供 Restful API，这是一种分布式系统的应用层解决方案，使用简单，提高客户端的便捷性和服务端的可伸缩性，实现客户端和服务端解耦。服务内部的子模块均使用 Docker 容器技术将其进行容器化，容器化技术可以提升子模块的可复用性和易用性，并且可以简化子模块的升级换代。服务端

需要承担与系统使用者交互的所有业务逻辑，包括获取用户上传的待扫描源代码、用户的鉴权、漏洞扫描任务状态的查看和管理、获取初步漏洞扫描报告和过滤后的漏洞报告。服务端的这些服务，有的服务耗时比较长并且可能有多个相同服务同时被调用，因此使用 RabbitMQ 消息队列中间件对服务进行排队，保证服务最终可以正常执行。

数据存储部分主要存储系统使用过程中产生的数据以及初始的漏洞数据集，这些数据均使用数据库进行持久化存储。随着系统的使用，漏洞数据会逐渐增多，当数据增加到一定规模时就会触发误报过滤模型的再训练。同时数据库中的漏洞数据之间的相似度信息也被记录在数据库中，数据存储部分有着类似于知识库的作用，对扩增漏洞数据集很有帮助。

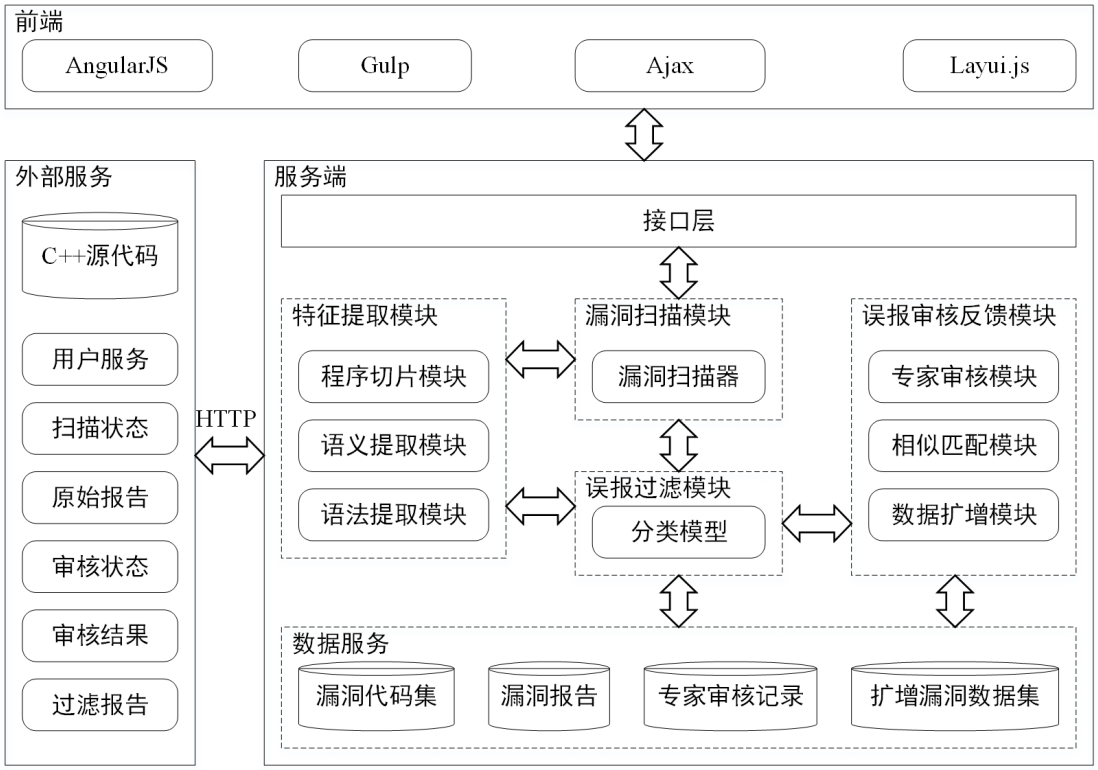


图 3.3: 系统架构图

3.3.2　系统模块划分

如图3.3所示，本系统从功能上分为四个主要部分，第一部分是漏洞静态扫描，主要进行多个开源漏洞扫描工具结果的融合；第二部分是特征提取模块，主要进行程序切片，并从切片后代码中提取语法特征和语义特征；第三部分是误

报过滤，主要负责误报过滤模型的训练、更新和使用；第四部分是误报漏洞审核反馈模块，包含专家审核模块、相似匹配模块和数据扩增模块。下面会根据完成一次漏洞扫描任务完整的流程来介绍各个模块的设计。

漏洞扫描模块。获取到用户上传的待扫描 C++ 项目后，系统会进行格式检查，判断项目中的 C++ 文件格式是否符合 C99 标准，并对符合标准的源代码文件进行漏洞扫描。系统使用开源工具 Cppcheck 和 TscanCode 进行漏洞扫描，两款扫描工具都会输出一份漏洞报告，并将两份漏洞报告融合。因为两款扫描工具没有参照统一的标准，在融合之前需要先将漏洞报告映射到统一的标准。

特征提取模块。系统对融合漏洞报告中的漏洞项进行前向程序切片，切片的起点为漏洞代码行，终点为函数入口。程序切片是根据漏洞函数的 CFG（控制流图）和 DFG（数据流图），并从漏洞代码行对应节点开始，遍历 CFG 和 DFG，去除与漏洞代码行无数据流关联节点，获取切片代码。对于语法特征，使用词嵌入算法提取 AST（抽象语法树）序列语法特征向量。对于语义特征，使用图嵌入算法提取 CFG 语义特征向量。

误报过滤模块。首先，系统对漏洞数据集进行漏洞扫描，获取漏洞报告，并根据标签将漏洞报告中的漏洞项判定为正报和误报。接着，系统对漏洞报告进行特征提取，提取每一个漏洞项的语法特征和语义特征。然后，系统将漏洞项的语法和语义特征向量以及正报或误报标签作为数据集，用于 DNN 模型训练。最后，系统将训练得到的误报过滤模型存入到 MongoDB，进行持久化存储。

审核反馈模块。漏洞专家审核准确率很高，但是相比于漏洞静态扫描技术存在扫描效率低、经济成本高的不足。系统从每一种漏洞中随机选择部分漏洞，并交给漏洞专家进行人工审核。对于漏洞专家审核出来的误报漏洞，使用相似度算法搜索与误报漏洞相似的漏洞，并将其标记为误报。使用相似度算法搜索完误报漏洞后，将这些漏洞数据作为误报过滤模型输入，对模型进行再训练。

3.3.3　4+1 视图

本部分从场景视图、逻辑视图、开发视图、过程视图和物理视图这几个角度来讲述系统总体设计。其中场景视图是从用户的角度来识别业务需求，是系统架构设计最基本的视图，在前文中的系统用例图部分已经讲述过，如图3.2所示，这里就不再赘述。

逻辑视图是从用户角度来进行逻辑分层、子模块划分、模块间依赖关系分析。对应 UML 中的类图，如图 3.4所示。

VulScan 负责进行源代码漏洞扫描，分为 VulScanService 和 ReportFusionService。其中 VulScanService 负责使用开源工具集对源代码进行扫描，ReportFusion-

Service 负责将不同扫描工具的结果融合为一个统一的漏洞报告。漏洞扫描结束后，就需要对扫描结果中的误报进行过滤。FalsePositiveFilter 负责进行误报过滤，分为 FeatureExtraction 和 ModelTraining。其中 FeatureExtraction 负责提取源代码中的语法和语义特征，ModelTraining 负责使用上一步提取到的特征进行误报过滤模型的训练。使用机器学习模型对误报进行过滤后，还需要人工对结果进行二次审核，保证误报过滤器的准确率以及提升机器学习模型的准确率。Feedback 部分负责将漏洞专家审核的结果反馈给误报过滤器，分为 VulReviewService 和 DataAmplify 两部分。其中 VulReviewService 负责对扫描结果进行人工审核，DataAmplify 负责使用相似度算法，寻找与误报代码片段相似的漏洞，进行误报数据集扩增，并对模型进行再训练。ScanTaskService 主要负责漏洞扫描任务列表的查看和管理等功能，记录漏洞扫描任务上传时间、扫描时间、唯一标识。

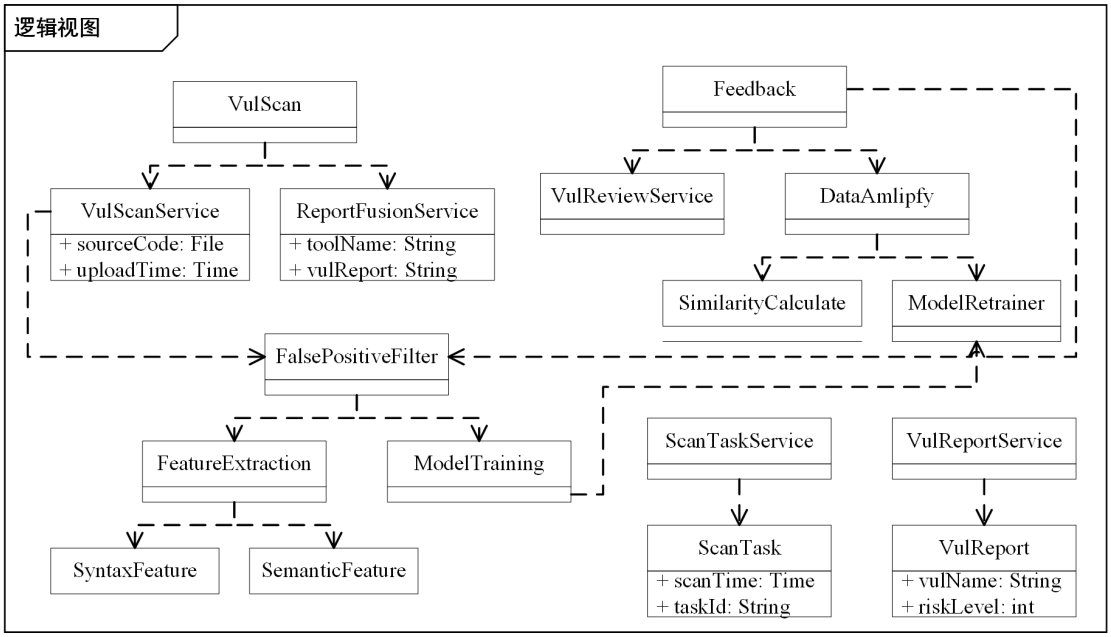


图 3.4: 逻辑视图

如图 3.5所示，进程视图主要描述系统的并发和同步设计以及线程间是如何进行交互和通信。系统初始在主线程，当进行漏洞扫描时，异步调用漏洞扫描线程，并向扫描线程发送心跳，获取扫描状态，当扫描完成时获取扫描结果并存入漏洞数据库。当进行误报过滤时，系统同步调用误报过滤线程，等到过滤结果返回后，返回给主线程显示。当触发迭代反馈时，系统调用漏扫审核线程，等审核结果返回后，使用该结果进行相似度检测，并将数据存储到误报数据库。系统接收到请求时，需要在 RabbitMQ 线程进行排队，避免请求丢失。触发误报

过滤模型的更新时，系统根据数据集中的误报数据，调用误报过滤线程的模型训练部分，更新误报过滤模型。

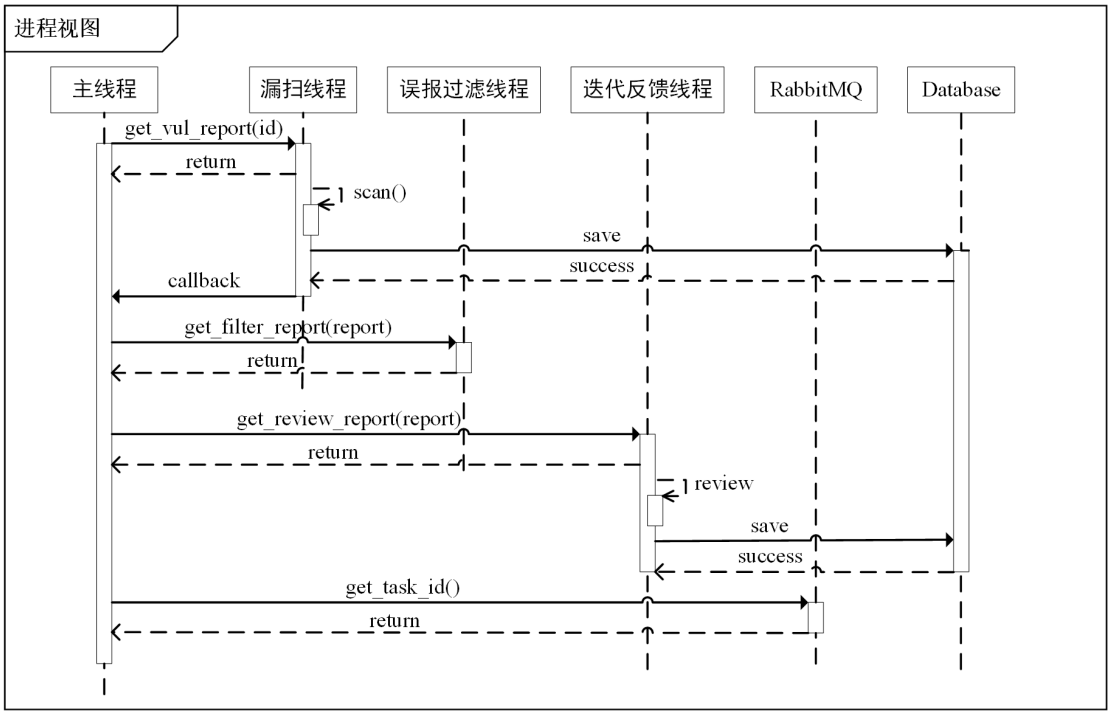


图 3.5: 进程视图

如图 3.6所示，开发视图面向开发者，从开发者的角度描述软件的静态组织结构，包括使用的框架、函数库以及组织结构。UI 部分包含系统中使用的技术框架和外部依赖库，包含报告模板、Lay.ui 模板引擎、程序切片的展示、数据流图和控制流图的展示以及使用的 JS 前端技术。Service Logic 指的是服务端逻辑，按照层级进行划分。上层 Controller 负责向外部提供服务，包括 TaskController 提供漏洞扫描任务管理接口，ScanController 提供漏洞扫描执行接口，FilterController 提供误报过滤模型使用的相关接口，ReviewController 提供漏洞审核和相似度检测相关接口。Controller 负责处理所有的外部请求，并交给 Service 来处理。Service 负责进行实际的服务，负责实现 Controller 层向外提供的接口，具体是通过调用相应的子模块实现实际的功能。Scan 进行实际的 C++ 源代码漏洞静态扫描，Filter 进行误报模型的训练和误报过滤。Common 提供公共服务，包括程序切片、特征提取、相似度计算、程序分析。底层技术框架是上层服务的基础，包括容器技术 Docker、消息队列中间件 RabbitMQ、数据库服务 MySQL、机器学习框架 Pytorch、日志服务 log4j，这些技术相互配合向上层提供服务。

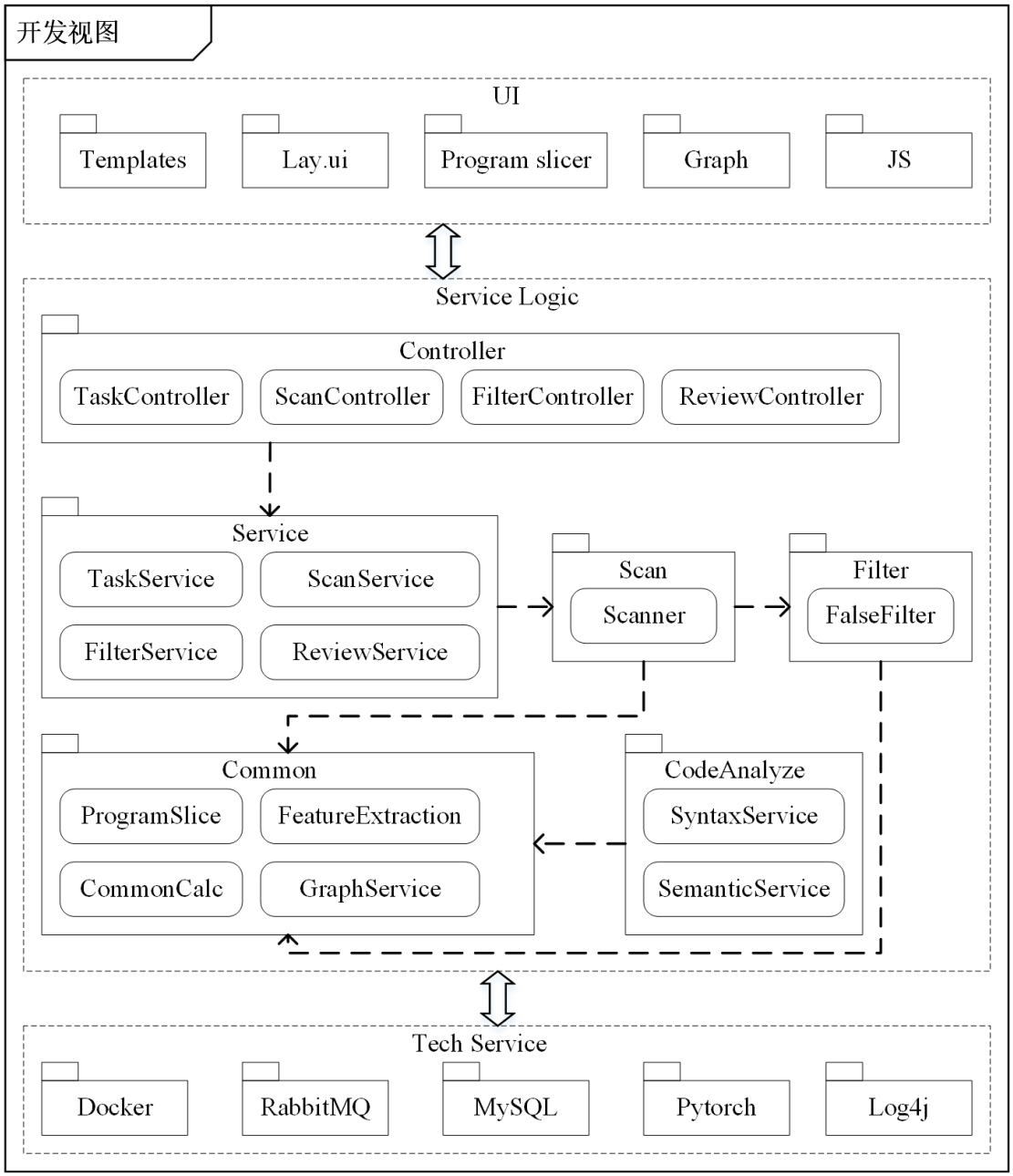


图 3.6: 开发视图

如图 3.7所示，物理视图从系统部署角度，描述系统在安装/部署以及物理通信上的问题。客户端在浏览器上，经过防火墙之后通过 HTTP 连接可以与服务器端进行通信。客户端发送请求后，先在 RabbitMQ 消息队列中间件进行排队，并按照先到先服务的原则将任务分发给 App Server。App Server 主要包含两个 Docker 镜像，分别是 Scanner 和 Filter，其中 Scanner 是负责进行漏洞扫描的容

器，Filter 是负责进行误报过滤的容器。使用容器技术对于提升系统的可扩展性有很大的帮助，当新请求到来的时候，系统只需要启动一个新的容器即可。系统投入使用过程中产生的漏洞扫描结果数据、误报数据、人工审核数据以及相似代码片段数据都持久化到数据库中，方便查看和后续训练误报过滤器时使用。

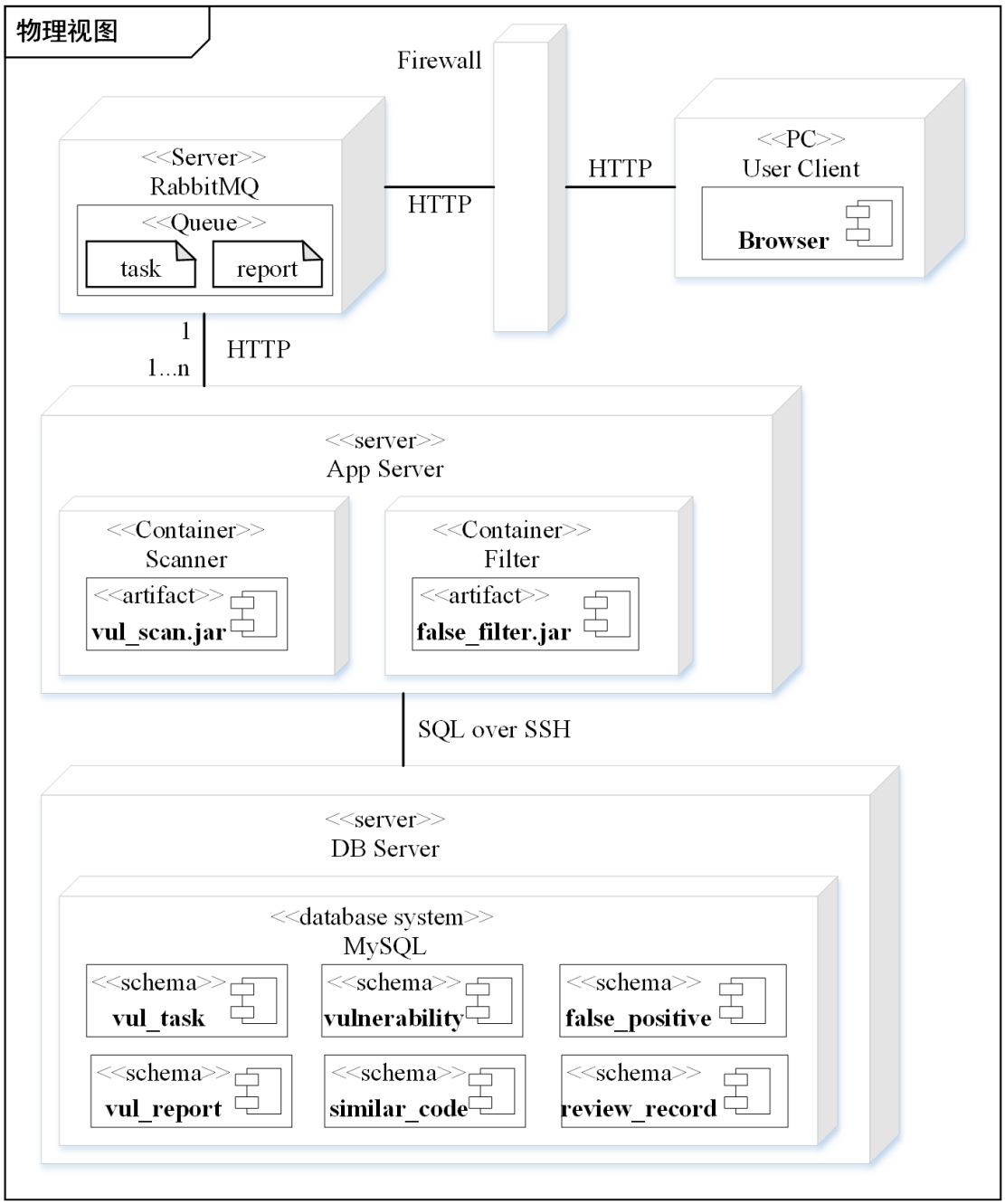


图 3.7: 物理视图

3.4　C++ 源代码漏洞静态扫描模块设计

3.4.1　架构设计

如图 3.8所示，C++ 源代码漏洞静态漏洞扫描模块为系统提供基本的漏洞扫描功能，即对 C++ 源代码进行扫描获取漏洞报告。对于用户上传的项目，需要进行程序清洗和规范检查，程序清洗是将源代码中非代码的部分去除，包括注释等无用代码行；规范检查是检查 C++ 源代码是否符合代码规范，比如是否包含无法识别的关键字，这些无法识别的字符不符合 C++ 编码规范，无法成功扫描，因此需要在这一步进行规范检查。漏洞扫描是使用开源工具 TscanCode 和 Cppcheck 进行漏洞扫描，并将扫描结果映射到同一个漏洞数据集，获取统一的漏洞列表以及对应的漏洞代码片段。扫描结果最终存储到数据库中，数据库使用主从模式，提高容错率，使用线程池减少数据库建立、断开连接消耗的时间。本部分对外提供统一的漏洞扫描接口。

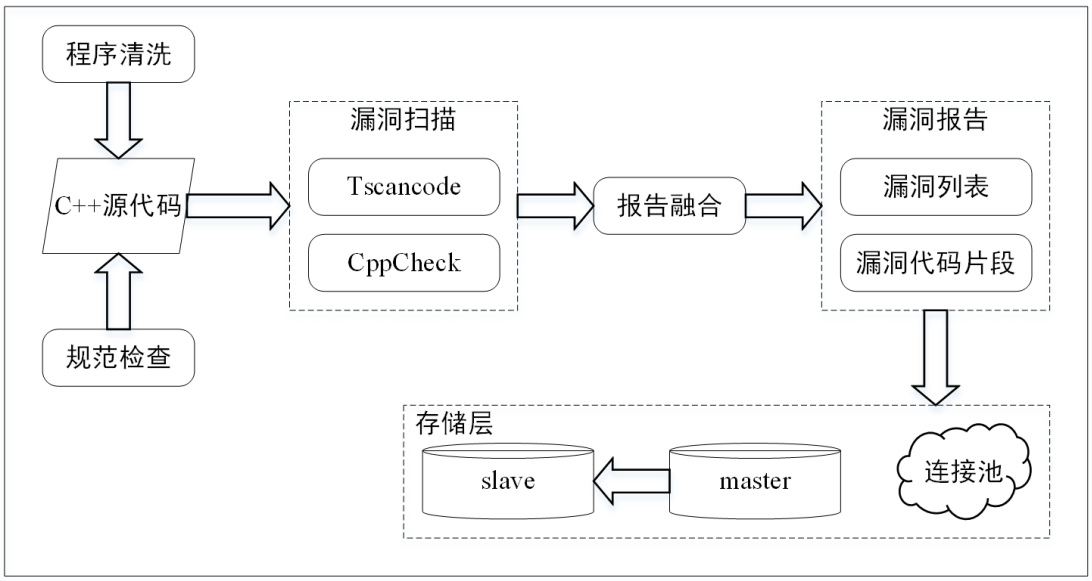


图 3.8: C++ 源代码漏洞静态扫描模块架构图

3.4.2　流程设计

C++ 源代码漏洞静态漏洞扫描模块流程如图 3.9所示。该模块主要根据系统实际需求，并结合不同开源工具的优势进行实现。具体步骤如下：

项目预处理是对待审核项目中的源代码文件进行处理，筛选掉非 C++ 源代码文件的项目文件，包括后缀不是.cpp 的文件以及包含无法识别字符的文件，并

且在此过程中将源代码文件中的注释去除掉，保证代码的简洁性。

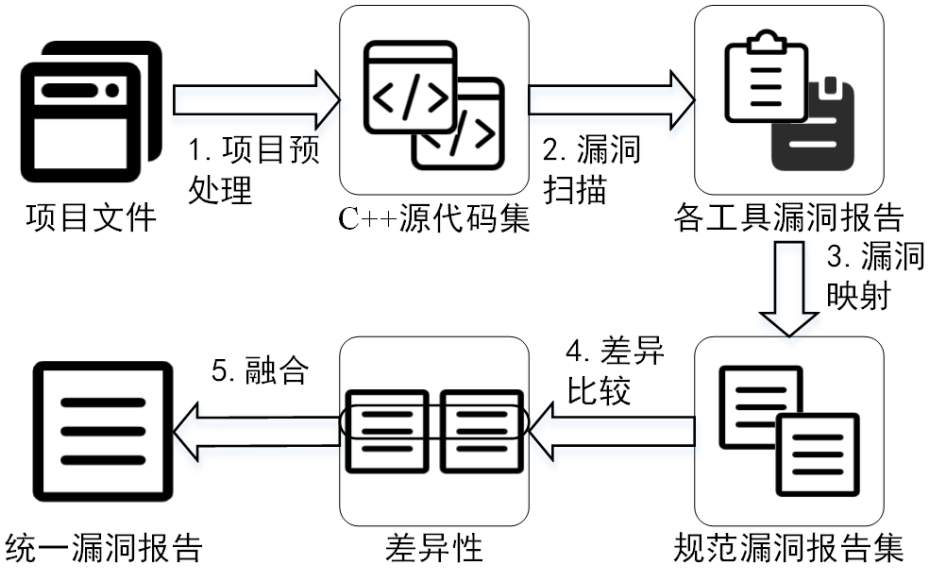


图 3.9: C++ 源代码漏洞静态扫描模块流程图

漏洞扫描是调用系统中的开源 C++ 源代码漏洞静态扫描工具集，目前有 TscanCode 和 Cppcheck，对源代码文件进行漏洞扫描，获取各个开源工具的漏报报告。

漏洞映射是将各个开源工具的漏洞报告映射到一个事先设定好的统一的标准，目前都是对标到 CWE 漏洞数据库 [46]。不同工具的漏报报告采用统一的漏洞名称、漏洞描述以及统一的漏洞代码描述方式，即定位到存在漏洞的代码行。虽然有些工具会输出漏洞的 source 点和 sink 点，但是由于有的工具并没有进行污点分析，没有 source 点和 sink 点信息。因此本系统统一使用漏洞代码行，并以该代码行为起点，进行前向切片。切片可以去除掉无关代码的干扰。

差异比较是比较各个开源工具扫描结果的差异之处，包括不同代码行有不同的漏洞、不同代码行有相同的漏洞、相同代码行有不同的漏洞这些情况，找出不同工具之间的差异性。

融合是针对上一步找出的各个开源工具之间的差异性进行相应的处理，同一代码行的相同漏洞直接合并为一个漏洞即可；对于不同代码行有不同的漏洞，将这些漏洞都合并到最终结果；对于不同代码行有相同的漏洞，将它们合并到最终漏洞报告；对于相同代码行有不同的漏洞，将两者都加入到最终漏洞报告。

3.4.3　核心类图

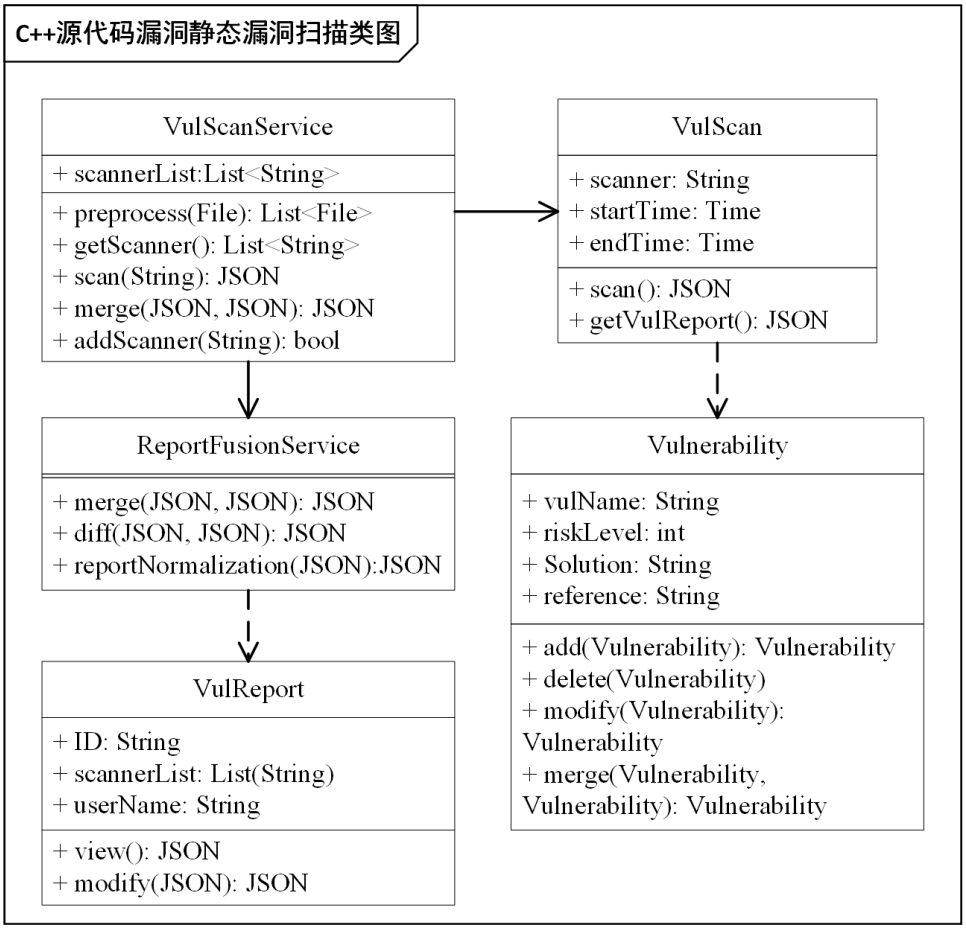


图 3.10: C++ 源代码漏洞静态漏洞扫描模块类图

如图 3.10所示是 C++ 源代码漏洞静态漏洞扫描模块的核心类图，VulScanService 是向外提供漏洞扫描服务的类，向外提供上传项目预处理（preprocess），获取扫描器列表（getScanner），指定扫描器进行漏洞扫描（scan），融合漏洞扫描结果（merge）和新增漏洞扫描器（addScanner）。

实际的漏洞扫描依赖 VulScan 类，包含扫描器名字、扫描开始时间、扫描结束时间、进行漏洞扫描（scan）、获取漏洞扫描报告（getVulReport）。Vulnerability 类包含漏洞的名称、风险等级、建议解决方案、参考链接等属性，还有增加漏洞（add）、删除漏洞（delete）、修改漏洞（modify）、合并漏洞（merge）等方法。

报告融合依赖于 ReportFusionService 类，该类向外提供三个接口，报告归一化（reportNormalization）是将漏洞报告映射到一个统一的标准格式，比较漏洞报告差别（diff）是比较归一化之后的漏洞报告之间的差别，合并漏洞报告（merge）

是将多个不同的漏洞扫描工具的归一化后的结果融合为一个统一的漏洞报告。

3.4.4　数据库设计

源代码漏洞静态漏洞扫描模块 ER 图如图 3.11所示。图中描述了漏洞扫描模块涉及到的三个实体，其中 vulTask 代表漏洞扫描任务，vulReport 代表漏洞报告，vulnerability 代表源代码中发现的漏洞。如图中所示，vulTask 和 vulReport 是一对多的关系，一个漏洞扫描任务对应多份漏洞报告，这是因为本系统使用多个开源扫描工具进行扫描，每个扫描工具都会产生一份漏洞报告；vulReport 和 vulnerability 是一对多的关系，一份漏洞报告中包含多个漏洞；vulTask 和 vulnerability 也是一对多的关系，一个漏洞扫描任务包含多个漏洞。

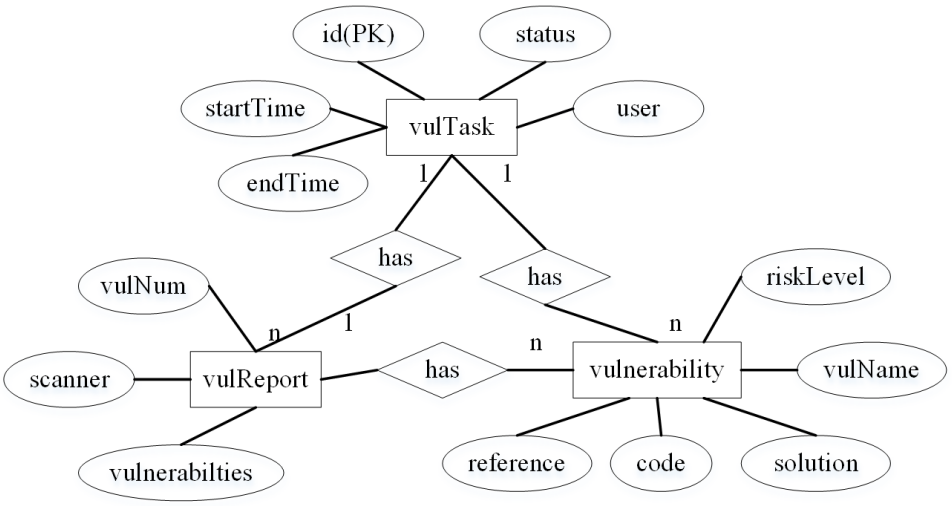


图 3.11: C++ 源代码漏洞静态漏洞扫描模块 ER 图

如表 3.9所示为 VulReport 表的数据库字段说明。该表主要用来存储漏洞扫描报告信息。便于在系统查看漏洞扫描报告概要信息、统计信息和漏洞列表。

表 3.9: VulReport 表

字段名	类型	描述
id	BIGINT	任务唯一 id，漏洞报告表主键
vulNum	VARCHAR	不同风险等级的漏洞数量，格式为 riskLevel:number，不同风险等级之间以；间隔
scanner	VARCHAR	扫描器名称，标识使用的扫描器，多个扫描器之间以；间隔

如表 3.10所示为 VulTask 表的数据库字段说明。该表主要用来存储漏洞扫描的任务信息。便于在系统中查看漏洞扫描任务列表和漏洞扫描任务详细信息。

表 3.10: VulTask 表

字段名	类型	描述
id	BIGINT	任务唯一 id，任务表主键
taskName	VARCHAR	任务名称，用于表示该任务
description	VARCHAR	任务描述，一段简短描述，介绍项目概要信息
user	VARCHAR	漏洞扫描任务发起者，表示扫描任务发起者身份
status	INT	漏洞扫描任务状态。0：成功，1：未开始，2：进行中，3：失败
startTime	TIME	漏洞扫描任务开始时间
endTime	TIME	漏洞扫描任务结束时间，未开始和进行中任务字段为空，成功和失败任务字段为结束时间

如表 3.11所示为 Vulnerability 表的数据库字段说明。该表主要用来存储漏洞信息。便于在系统中查看漏洞名称、风险等级、建议解决方案等详细信息。

表 3.11: Vulnerability 表

字段名	类型	描述
id	BIGINT	漏洞唯一 id，漏洞表主键
vulName	VARCHAR	漏洞名称，用于表示该漏洞
riskLevel	INT	漏洞风险等级。0：信息，1：警告，2：低危，3：中危，4：高危
solution	VARCHAR	通用的建议解决方案，方便用户查看
reference	VARCHAR	漏洞参考信息，为超链接，链接到 CWE 页面
code	VARCHAR	漏洞上下文代码片段，与扫描任务相对应

3.5　C++ 源代码特征提取模块设计

3.5.1　架构设计

如图 3.12所示，C++ 源代码特征提取模块主要目的是提取代码中的语法特征、语义特征和统计特征。项目通过 HTTP 连接传输到服务器，在完成程序清洗和规范检查后得到 C++ 源代码。拿到 C++ 源代码之后，使用开源代码分析工

具 Joern[2]，提取源代码的 Control Flow Graph（控制流图）、Data Flow Graph（数据流图）、Call Graph（调用关系图）、Abstract Syntax Tree（抽象语法树）。提取 C++ 源代码特征的开源工具较少，Joern 是使用 Fuzz 技术来提取 C++ 源代码中的特征，但是该工具也存在一些缺陷，比如获取调用关系图时无法获取调用语句中的参数类型。因此本系统根据调用语句中的变量名，在调用函数中进行类型匹配，基本可以识别到所有的参数类型。

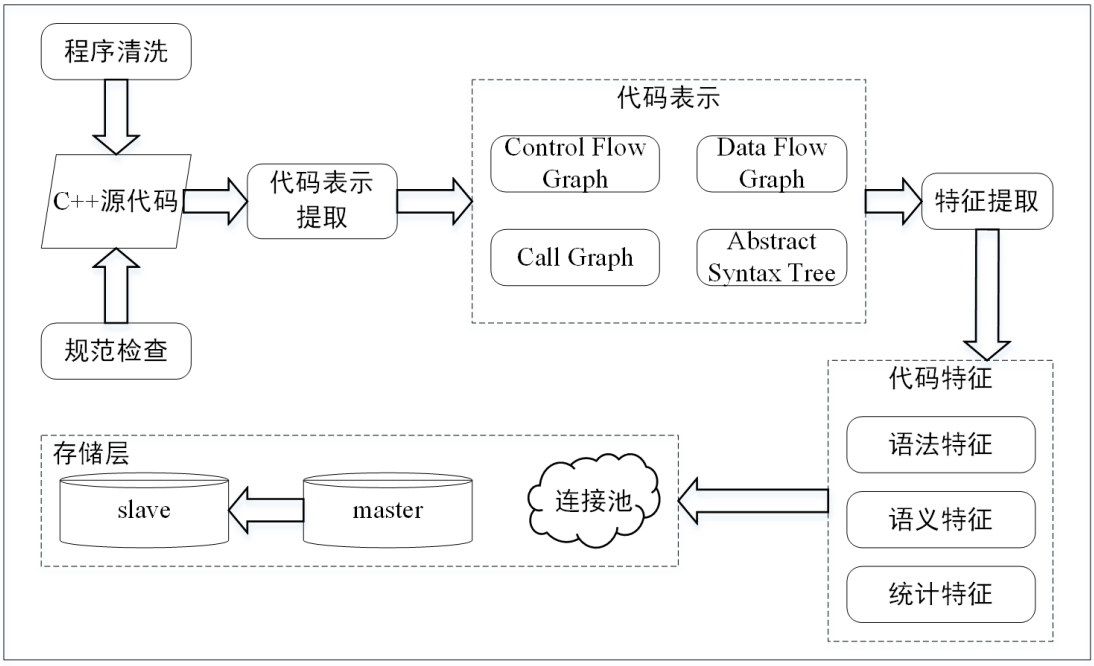


图 3.12: C++ 源代码特征提取模块架构图

接着使用代码特征提取算法，来提取代码表示中的特征向量。抽象语法树包含源代码的语法信息，为了提取语法特征，对抽象语法树进行先序遍历，获取抽象语法树的所有叶子节点组成的标识符序列，接着使用 word2vec 算法提取每个标识符的特征向量，最后进行平均池化获取抽象语法树的特征向量。word2vec 可以捕获上下文的关系，代码存在明显的上下文关系，因此可以提取到代码序列中的语法特征。控制流图和数据流图包含源代码的语义信息，为了提取语义特征，使用 graph2vec 算法，提取两个图的特征向量。但是 Joern 提取的控制流图和数据流图是函数粒度的，不包含函数间的调用关系，因此根据调用关系图将调用关系加到两个图中，捕获完整的语义特征。统计特征包含代码行数、变量

[2]https://joern.io/

数量、分支数量、循环结果数量等统计数据。

3.5.2　流程设计

C++ 源代码特征提取模块流程如图 3.13所示。C++ 源代码特征提取模块的处理流程主要是参考的常用的几种代码表示，并结合特征工程技术来提取代码表示的特征向量。下面是 C++ 源代码特征提取的主要步骤：

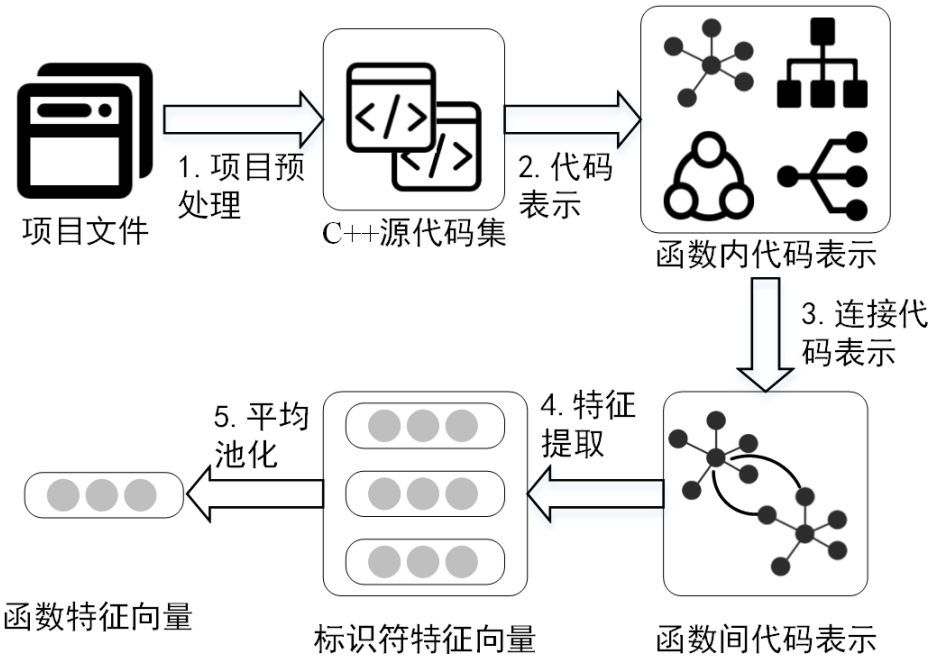


图 3.13: C++ 源代码特征提取模块流程图

项目预处理是为了将一些非 C++ 源代码文件过滤掉，以及将 Joern 无法提取代码特征的源代码文件过滤掉。

代码表示是将源代码表示为不同的代码表示形式。Joern 可以提取函数内的控制流图、数据流图、抽象语法树和函数调用图。但是 Joern 对于函数间的调用关系的识别不够准确，对于函数重载的情况，Joern 无法判断具体调用的哪一个函数。因此需要根据调用语句实参的标识符来识别实参具体的类型。

连接代码表示是为了将函数内的代码表示扩展到函数间的代码表示。根据函数调用关系中的调用语句、被调语句，将调用函数和被调函数的控制流图和数据流图连接起来，形成一个整体，即为函数间的代码表示。

特征提取是将上一步的控制流图、数据流图和抽象语法树表示为特征向量。具体是先对代码表示做归一化，将抽象语法树变成标识符流，控制流图和数据流图的抽象程度较高，不需要做归一化；接着对标识符流使用 word2vec 算法提

取每个标识符的特征向量，对控制流图和数据流图使用 graph2vec 算法提取特征向量。

平均池化是为了将特征向量矩阵变成单独的特征向量，本质上和取平均值相似，这是机器学习技术中的表述方式。

3.5.3　核心类图

C++ 源代码特征提取模块核心类图如图 3.14所示。功能主要通过 FeatureExtractionService 特征提取类提供，下面详细介绍该模块涉及的类。

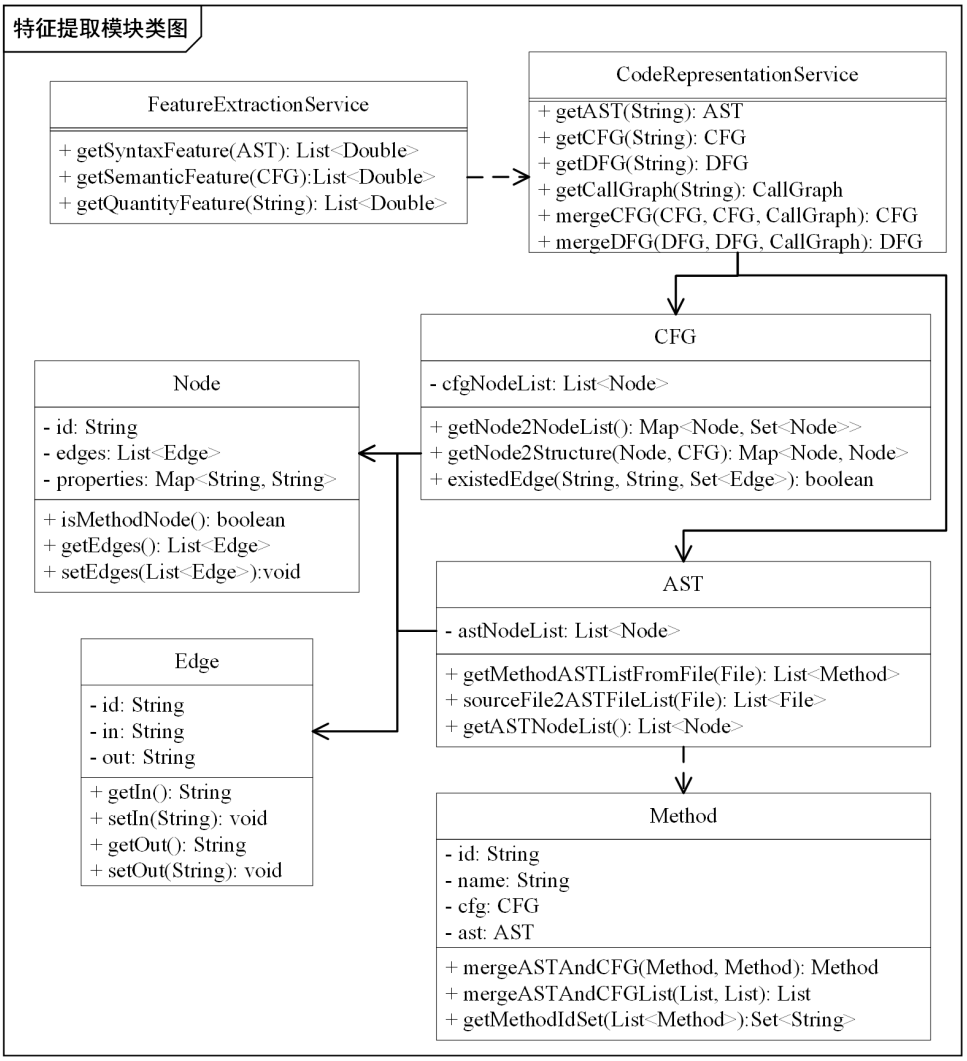


图 3.14: C++ 源代码特征提取模块核心类图

特征提取类向外提供三个接口，分别是获取语法特征（getSyntaxFeature）、获取语义特征（getSemanticFeature）和获取统计特征（getQuantityFeature）。特

征提取类依赖于代码表示类 CodeRepresentationService，代码表示类向外提供获取抽象语法树、控制流图、数据流图、函数调用图等代码表示的功能，这些功能都是通过开源 Joern 提供的，该工具将所有的代码表示以代码属性图（Code Property Graph）的形式输出，并向外提供从代码属性图中获取这些代码表示的接口，但是提供的并不全面，因此大部分脚本需要自己去实现。特征提取类还提供了将函数内的代码表示扩展到函数间代码表示的接口。当获取到代码表示之后，特征提取类会调用 word2vec 和 graph2vec 的可执行脚本来获取特征向量矩阵，最后进行平均池化获取最终的特征向量。

上述类依赖于 AST 和 CFG 两个类，分别表示抽象语法树和控制流图两个数据结构。CFG 提供了一些方法，包括获取节点相连的节点列表（getNode2NodeList）、获取节点对应的结构节点（getNode2Structure）、判断边是否已经在控制流图中存在（existedEdge），还有包含控制流图中节点列表（cfgNodeList）的属性。AST 提供了一些方法，包括从 Json 文件中提取所有方法的抽象语法树（getMethodASTList FromFile）、源代码文件转换为抽象语法树 Json 文件（sourceFile2ASTFile）、获取抽象语法树节点列表（getASTNodeList）。

Node 类和 Edge 类是基础数据结构类，分别表示节点和边。Node 类包含一些属性，包括唯一标识（id）、节点相连的边（edges）、节点包含的属性（properties）。Node 类也提供了一些方法，包括判断是否是方法中的节点（isMethodNode）、获取所有的边（getEdges）、设置边（setEdges）。Edge 类包含一些属性，包括唯一标识（id）、边的入口节点（in）和边的出口节点（out），以及相应读取、修改节点属性的方法。

3.6　漏洞静态扫描误报过滤模块设计

3.6.1　架构设计

静态漏洞扫描误报过滤模块架构设计如图 3.15所示。误报过滤模块的主要功能是作为漏洞扫描结果的过滤器，过滤掉扫描出来的漏洞列表中的误报。误报过滤的主要步骤是：首先，对带漏洞标签的源代码数据集进行漏洞扫描，获取正报漏洞和误报漏洞数据集；接着，对带漏洞标签的源代码进行特征提取，提取语法特征、语义特征和统计特征；然后，将特征向量和误报标签输入到分类模型中，进行机器学习模型的训练，得到误报过滤模型；最后，将误报过滤模型存入数据库，进行持久化存储。

为了应对漏洞扫描器和代码特征提取算法的更新和替换，静态漏洞扫描误报过滤模块屏蔽了漏洞扫描模块和特征提取模块的细节，直接提供相应的接口

给误报过滤模块使用。误报过滤模块只需要调用相应的接口即可，返回结果的格式已经固定，保证模块之间的松耦合。机器学习模型的训练是十分消耗时间的，因此本系统将训练得到的误报过滤模型存储到数据库中，下次对于新的漏洞扫描结果进行过滤时，直接将模型拿过来使用即可，不需要每次都重新训练模型。

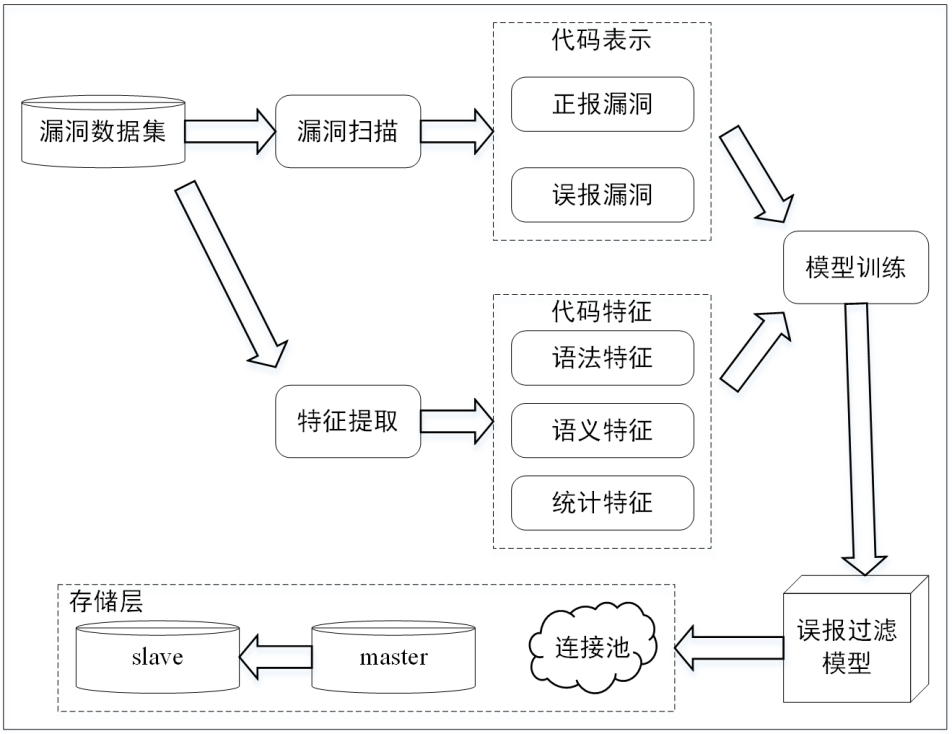


图 3.15: 漏洞静态扫描误报过滤模块架构图

3.6.2　流程设计

静态漏洞扫描误报过滤模块流程设计如图 3.16所示。静态漏洞扫描误报过滤模块流程设计主要是按照机器学习模型的训练和使用的常用方法。下面是静态漏洞扫描误报过滤的主要步骤：

误报扫描是使用漏洞扫描工具对带有漏洞标签的源代码数据集进行扫描，分析扫描的结果可以得到漏洞报告中的正报和误报数据，这些数据可以表示漏洞扫描工具的能力边界，以及从导致误报的代码段中挖掘出某些代码特征。

代码特征提取和代码特征提取模块进行的处理类似。为了减低系统的耦合性，这里的代码特征提取直接调用代码特征提取模块提供的接口，可以获取代码的语法特征向量、语义特征向量和代码统计特征，转化为特征向量之后就可以输入到机器学习模型中使用。

分类模型训练是使用上一步得到的漏洞代码段的语法特征向量、语义特征向量、统计特征以及相应正报、误报标签来训练机器学习模型。该模型的主要目的是过滤掉漏洞报告中的误报漏洞，误报漏洞存在共有的特征，使用机器学习方法来学习这些误报间共有的特征，最后将训练得到的模型持久化存储下来。

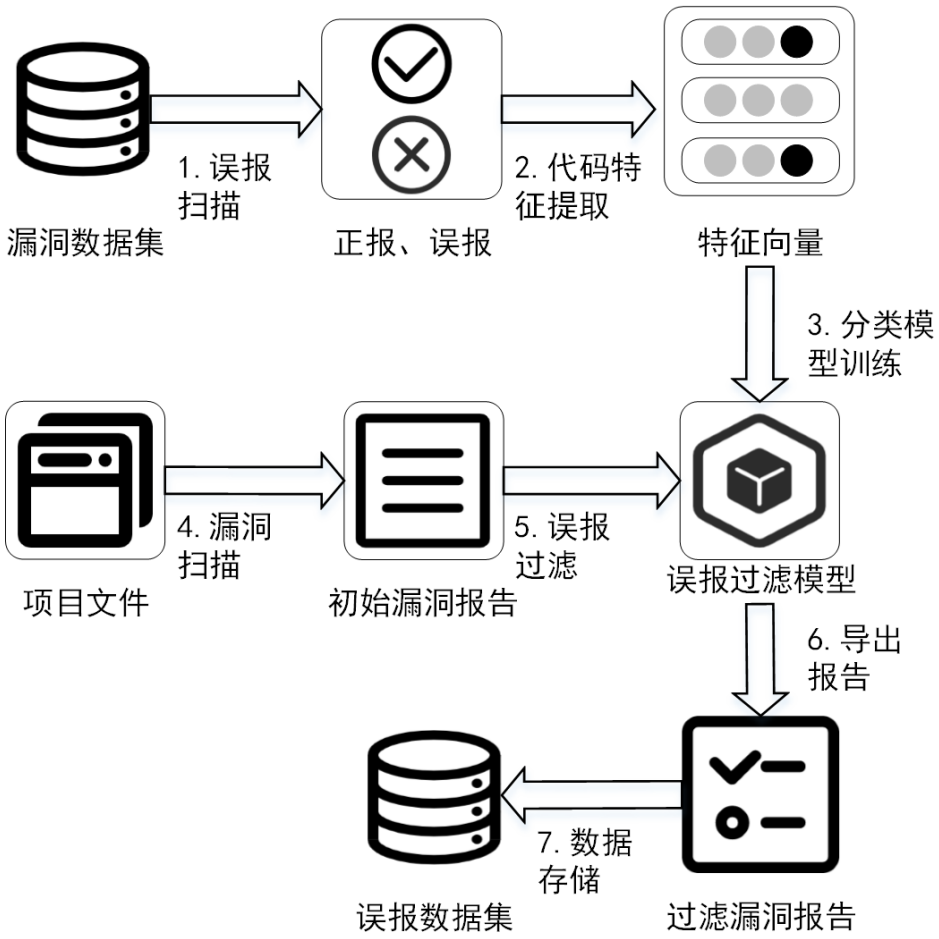


图 3.16: 漏洞静态扫描误报过滤模块流程图

漏洞扫描和漏洞扫描模块进行的操作类似。同样是为了降低系统的耦合性，这里的漏洞扫描也是直接调用的漏洞扫描模块提供的接口，当漏洞扫描工具更新和替换时，不需要修改这部分代码。

误报过滤是使用误报过滤模型对漏洞扫描得到的漏洞报告进行过滤，过滤掉其中的误报漏洞，将正报漏洞留下来，这样可以大幅度减少漏洞和误报数量，减轻审核人员压力。

导出报告是将经过误报过滤后的漏洞报告导出为静态网页文件，该文件不依赖于网络，在浏览器端可以随时打开，方便用户随时随地的查看漏洞报告。

数据存储是将经过过滤的漏洞报告存储到数据库中，包括正报漏洞还有误报漏洞，因为误报过滤器的准确率无法达到 100%，后续还需要经过人工审核，主要是审核误报是否准确。

3.6.3　核心类图

静态漏洞扫描误报过滤模块核心类图如图 3.17所示。误报过滤模块的核心就是训练误报过滤模型，训练模型使用的数据是代码特征向量和正报、误报标签，该数据是从 Juliet 测试集中提取的源代码中函数对应的语法、语义、统计特征向量，同时 Juliet 中的函数名就表示该函数是否是漏洞。

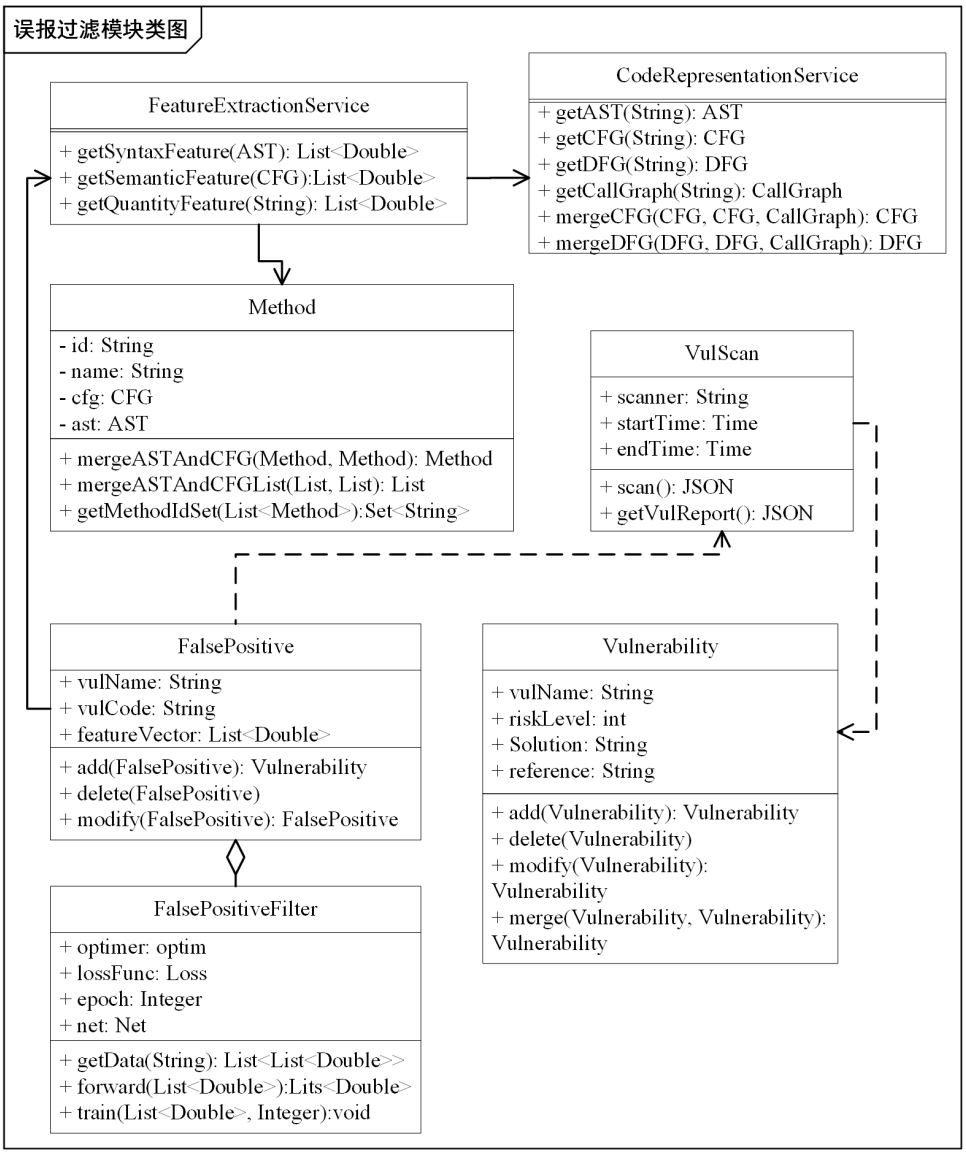


图 3.17: 漏洞静态扫描误报过滤模块类图

FalsePositiveFilter 类（误报过滤类）为误报过滤模块的核心类，表示误报过滤器。误报过滤器本质上是一个机器学习二分类模型，拥有机器学习中通用的属性，其中 optimer（优化器）是控制模型训练过程中调整模型参数方式的变量，本系统使用 Adam 优化器，learningRate（学习率）为 0.001；lossFunc（损失函数）是指导模型优化方向和衡量模型优劣的指标，模型的优化的目标通常都是最小化损失函数，本系统使用的损失函数为 CrossEntropyLoss（交叉熵损失函数）；epoch（轮次）为模型训练的次数，表示模型训练多少次之后停止；Net（网络结构）为模型的核心部分，表示模型的结构。同时误报过滤类还提供了一系列方法，包括 getData（获取训练和预测使用数据）、forward（模型的输入到输出之间的转换关系，和模型的结构有关）、train（模型训练过程中使用的函数，包括损失函数、优化器、反向求导函数），这些方法共同完成了误报过滤模型的训练和预测。

FalsePositive 类（误报类）为误报过滤模块的数据类，用于存储误报信息。误报信息包括该漏洞的名称（vulName）、漏洞上下文代码（vulCode）以及相应的语法特征向量、语义特征向量、统计特征，这些特征数据拼接为一个数组。同时误报类还提供了添加误报（add）、删除误报（delete）、修改误报（modify）的方法，这些方法对于实时根据误报过滤器结果调整误报过滤结果很有帮助。

3.6.4　数据库设计

静态漏洞扫描误报过滤模块 ER 图如图 3.18所示。vulReport 表示漏洞扫描工具扫描得到的漏洞报告；FalsePositive 表示误报漏报数据；Feature 是漏洞对应的特征向量；Node 是代码特征中的节点，抽象语法树和控制流图等结构中使用了该数据；Edge 是代码特征中的边，与 Node 一起使用，构成了抽象语法树和控制流图等结构。其中 VulReport 和 FalsPositive 是一对多的关系，一份漏洞报告经过误报过滤后可能得到很多误报信息；VulReport 和 Feature 是一对多的关系，一份漏洞报告包含多个漏洞项，每一个漏洞项对应的代码都需要提取其中的包含语法特征、语义特征、统计特征的代码特征；Feature 和 Node 是一对多的关系，代码特征属于的代码表示会存在多个 Node，每个 Node 表示抽象语法树或控制流图中的节点，除了节点自身包含属性之外，节点还包含相邻的边的信息；Feature 和 Edge 是一对多的关系，代码特征属于的代码表示可能存在多个 Edge，Edge 的作用是连接 Node。因为提取代码特征是很耗时的，因此将这些数据持久化到数据库中，下次使用的时候可以直接从数据库读取，不用重新计算，大幅度节省时间。

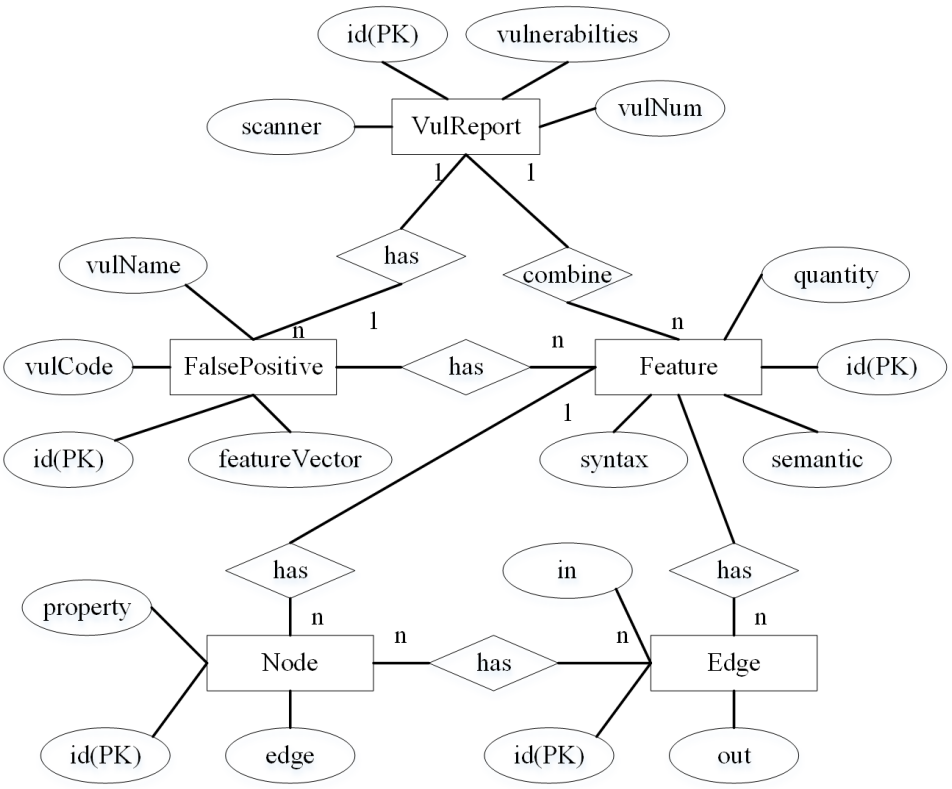


图 3.18: 漏洞静态扫描误报过滤模块 ER 图

如表 3.12所示为 FalsePositive 表的数据库字段说明。该表主要用来存储误报漏洞信息。包含误报代码片段以及相应的特征向量等信息。

表 3.12: FalsePositive 表

字段名	类型	描述
id	BIGINT	任务唯一 id，误报漏洞表主键
vulName	VARCHAR	判断为误报的漏洞名称，便于后续人工审核时判断是否确实为误报
vulCode	VARCHAR	误报上下文代码片段，为精简后代码段
featureVector	VARCHAR	存放特征向量表的主键，每一个代码段在特征向量表中都有语法、语义、统计三种特征向量

如表 3.13所示为 Feature 表的数据库字段说明。该表主要用来存储误报漏洞信息。包含误报代码片段以及相应的特征向量等信息。

表 3.13: Feature 表

字段名	类型	描述
id	BIGINT	任务唯一 id，代码特征表主键
syntax	VARCHAR	语法特征向量，以字符串形式存储，以；间隔
semantic	VARCHAR	语义特征向量，以字符串形式存储，以；间隔
quantity	VARCHAR	统计特征，包含代码行数、分支数量等统计特征，特征间以；间隔

3.7　误报漏洞审核反馈模块设计

3.7.1　架构设计

误报漏洞审核反馈模块架构设计如图 3.19所示。误报漏洞审核反馈模块是在漏洞扫描和误报过滤模块基础之上，经过专家审核来完成的。

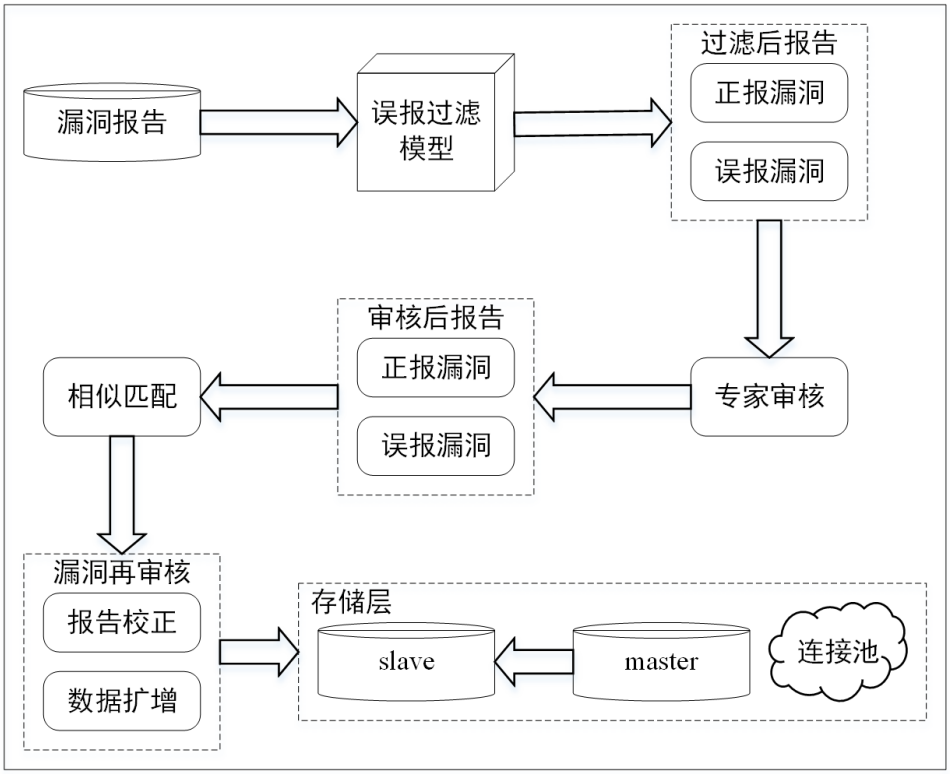


图 3.19: 误报漏洞审核反馈模块架构图

误报漏洞审核反馈模块的处理过程中需要调用漏洞扫描模块、误报过滤模块和特征提取模块的相关接口，因为该过程涉及到漏洞扫描、误报过滤和代码

特征提取。除了这些功能之外，该模块还涉及代码段间的相似度计算。因为专家审核成本高、耗时长，所以本系统支持漏洞审核专家在每一个漏洞类别中选择部分漏洞进行审核，对于判断为误报的漏洞，使用相似度算法寻找与判断为误报的漏洞相似的漏洞，并且将这些漏洞也判定为误报。这样一方面可以减轻漏洞审核专家的压力，另一方面可以通过相似度算法寻找更多的误报漏洞，扩增误报漏洞数据集，对于后续训练误报过滤器很有帮助。

专家审核为误报的漏洞和使用相似度算法匹配到的漏洞都需要存到数据库中，方便后续用户的查看和使用。同时，审核结束后，还需要根据误报数据集的规模来决定是否要对误报过滤器重新训练，重新训练模型之后将模型更新到数据库中。

3.7.2　流程设计

误报漏洞审核反馈模块流程如图 3.20所示。该模块的设计是因为机器学习模型准确率目前还不是很高，但是随着数据量的增多，准确率和召回率会逐步提升。同时单次人工审核的误报漏洞数量较少，因此加入了相似度计算模块。下面是误报漏洞审核反馈模块主要流程：

误报过滤是使用训练得到的误报过滤器对漏洞报告进行初步过滤，分别得到正报漏洞和误报漏洞，但是因为机器学习模型的准确率还不是很高，因此需要再进行专家审核。专家审核是漏洞专家对于初步漏洞报告进行人工审核，对机器学习模型判断的结果进行二次审核，判断其中的正报、误报审核结果是否准确，如果审核结果与误报过滤模型结果不一致，需要将漏洞结果存下来。

相似漏洞匹配是针对人工结果与机器审核结果不一致的漏洞项，使用相似度匹配算法，在所有的漏洞项中寻找与不一致漏洞项相似度的漏洞，这样可以对专家审核的结果再进行一次校验，并且可以扩增误报数据集。

模型再训练是使用专家审核和相似度匹配的结果，对误报过滤模型进行再次训练，捕获这些误报的代码特征，提高模型的准确率。

误报过滤是使用更新过的误报过滤模型，对漏洞报告进行重新过滤，目标是将专家审核为误报的漏洞过滤掉。

更新漏洞数据库是将重新过滤的漏洞报告结果，更新到漏洞数据库中，将误判为正报的漏洞更新为误报。

更新误报数据集是将专家审核和相似度计算判断为误报的漏洞数据，加入到误报过滤器的数据集中，后续训练模型时可以直接使用。

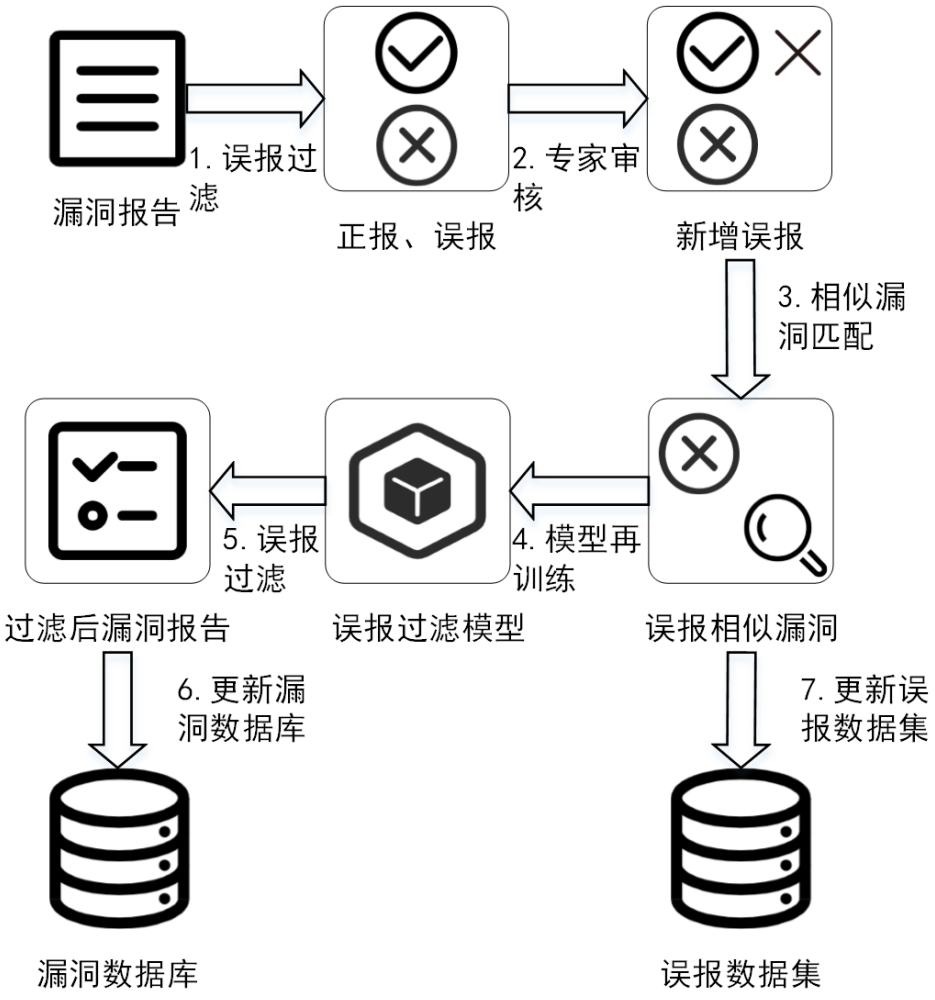


图 3.20: 误报漏洞审核反馈模块流程图

3.7.3　核心类图

误报漏洞审核反馈模块类图如图3.21所示。误报漏洞审核反馈模块的核心类是 Reviewer（审核专家）类，该类负责对漏洞进行人工审核。Reviewer 类具有漏洞专家唯一标识、漏洞专家名、审核报告唯一标识等属性，还有关于漏洞审核的一系列方法，包括漏扫审核（review）、修改漏洞项（modify）、新增漏洞项（add）和删除漏洞项（delete）。

误报漏洞审核反馈模块的另一个核心类是 SimilarityTool（相似计算工具）类，该类负责计算漏洞项之间的相似度。相似计算工具类根据漏洞项的语法特征、语义特征、统计特征以及漏洞名称等信息，使用二分类机器学习模型判断两个漏洞项之间是否相似。该类有一系列属性，包括漏洞项所属项目、漏洞项对于代码片段、漏洞项对应特征向量，还有一些方法，包括获取相似分类模型（getModel）、判断是否相似（getSimilarity）、获取特征向量。该类依赖于特征提取类和误报类。

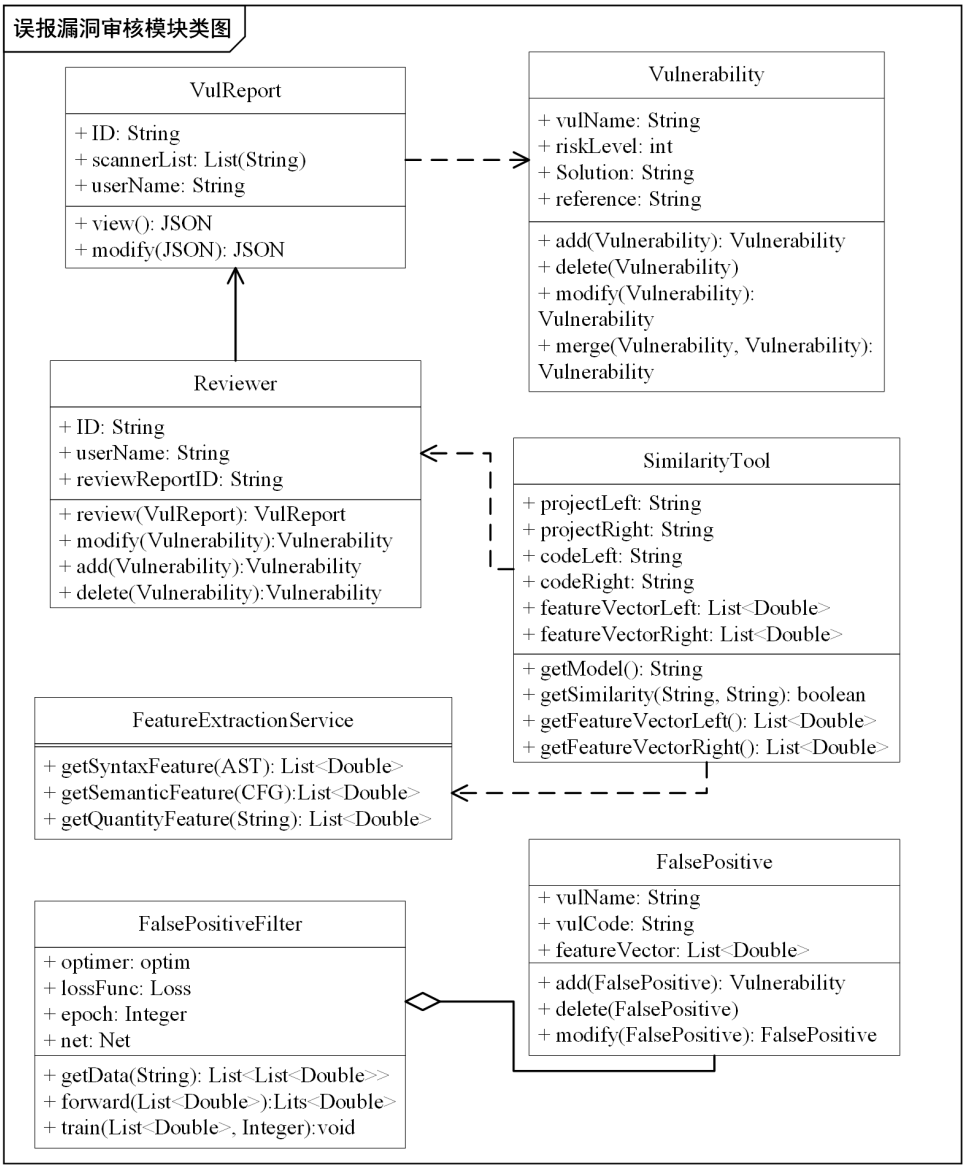


图 3.21: 误报漏洞审核反馈模块核心类图

3.8　本章小结

本章首先对 C++ 源代码漏洞静态扫描平台的功能、非功能需求以及相应的用户场景经常分析；其次，对整个系统的技术架构和使用的基础技术进行分析，介绍了不同模块间的划分，并使用 4+1 视图从多个维度对系统进行了分析；最后，对系统的四个核心模块：C++ 源代码漏洞静态扫描模块、C++ 源代码特征提取模块、静态漏洞扫描误报过滤模块和误报漏洞反馈模块，从架构设计、流程设计、核心类图解析和数据库设计多个角度进行了分析、介绍。为了方便功能的增加和更新，系统利用 Docker 容器技术将系统的子功能进行模块化；为了提高

数据的可靠性，数据库采用主从模式，当主服务器出现宕机等故障是，从服务器可以直接替换上去；为了保证服务的可靠性，系统使用 RabbitMQ 消息队列中间件，将漏洞扫描任务请求排队，依次进行处理。

第四章　C++ 源代码漏洞扫描系统的实现

本章在第三章系统需求分析和设计的基础上，对各个模块具体的实现方式进行详细描述，主要以顺序图和核心代码进行描述，其中顺序图描述系统中各模块间的调用顺序，核心代码是系统中模块关键功能的代码，系统的主要界面在最后介绍。

4.1　C++ 源代码漏洞静态漏洞扫描模块实现

4.1.1　顺序图

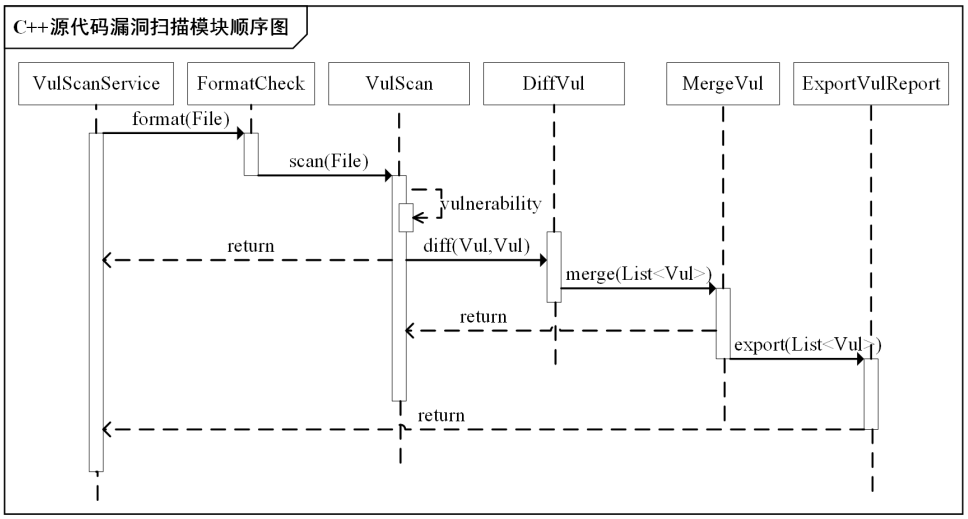


图 4.1: C++ 源代码漏洞静态扫描模块顺序图

如图 4.1所示是进行 C++ 源代码漏洞静态扫描的调用顺序，主要模块是漏洞扫描和报告融合，是本系统最核心的部分。在接收到用户上传的待审核 C++ 项目后，VulScanService 会调用 FormatCheck 类进行格式检查，判断项目中的 C++ 文件格式是否符合 C99 标准，格式检测完成后，会将符合标准的 C++ 源代码文件传给漏洞扫描器。VulScan 类进行实际的漏洞扫描，该类会调用 scan 方法进行漏洞扫描，scan 方法会根据配置文件中漏洞扫描器的名称和调用命令对 C++ 源代码文件进行扫描，每个扫描器都会输出一份漏洞报告。

获取到不同漏洞扫描器的报告后，DiffVul 类负责对比漏洞报告间的区别。因为不同的漏洞扫描工具之间没有一个统一的标准，在对比之前需要先将漏洞

报告映射到一个统一的标准，具体映射方式是按照最小粒度原则将工具的漏洞项映射到与该漏洞项相关的 CWE 漏洞。本系统选择映射到 CWE 漏洞数据集，是因为该数据集漏洞覆盖率广且使用广泛。比较方法是判断相同漏洞代码行位置的漏洞 ID 是否相同，将具有差异性的漏洞记录下来。接着 MergeVul 类负责将这些漏洞报告合并为一个漏洞报告，具体方式是如果漏洞 ID 相同，直接合并为一个漏洞项，如果漏洞 ID 不同，则作为两个单独的漏洞项，分别加入到统一漏洞报告中。最后 ExportVulReport 类负责将统一漏洞报告导出为静态漏洞报告，其中包含对不同风险等级漏洞的统计、不同类别风险报告的统计以及漏洞相关代码的格式化展示。漏洞报告最终持久化存储到数据库中。

4.1.2　关键代码

```
public Output doScan(CommandLine commandLine) {
    //…初始化代码省略
    // TScanCode 扫描
    TScanner tScanner = new TScanner();
    JSONArray tScanRes = tScanner.doScan(decompressFilePath);
    // CppCheck 扫描
    CScanner cScanner = new CScanner();
    JSONArray cScanner = cScanner.doScan(decompressFilePath);
    // 合并两种扫描结果
    if (tScanRes != null && !tScanRes.isEmpty()
            && cScanner != null && !cScanner.isEmpty()) { // 判断扫描成功
        JSONArray res = new JSONArray();
        for (int i = 0; i < tScanRes.length(); i++) { // 添加 TscanCode 结果
            res.put(tScanRes.getJSONObject(i));
        }
        for (int i = 0; i < cScanner.length(); i++) { // 添加 CppCheck 结果
            res.put(cScanner.getJSONObject(i));
        }
        Output.SCAN_RESULT.setData(res); // 设置扫描结果
        return Output.SCAN_RESULT;
    }
}
```

图 4.2: C++ 源代码漏洞静态扫描模块关键代码

漏洞扫描模块关键点在于将开源工具 TscanCode 和 Cppcheck 的漏洞报告进行融合，获取整体漏洞报告，关键代码如图4.2所示。首先使用 TscanCode 对上传的源代码进行漏洞扫描，同时将该工具的漏洞结果映射到系统的漏洞库中，包括漏洞的名称、类型、描述、参考链接、建议解决方案等信息；接着使用 Cppcheck 进行与用 TscanCode 工具类似的操作，将漏洞结果也映射到系统的漏洞库；最

后根据漏洞名称、类型以及在源代码中的位置信息，融合两个开源工具的报告，获取整体漏洞报告，这样可以降低漏报率。

4.2　C++ 源代码特征提取模块实现

4.2.1　C++ 源代码特征提取模块顺序图

如图 4.3所示是 C++ 源代码特征提取模块的顺序图。描述了特征提取过程中，各个类之间的交互过程。

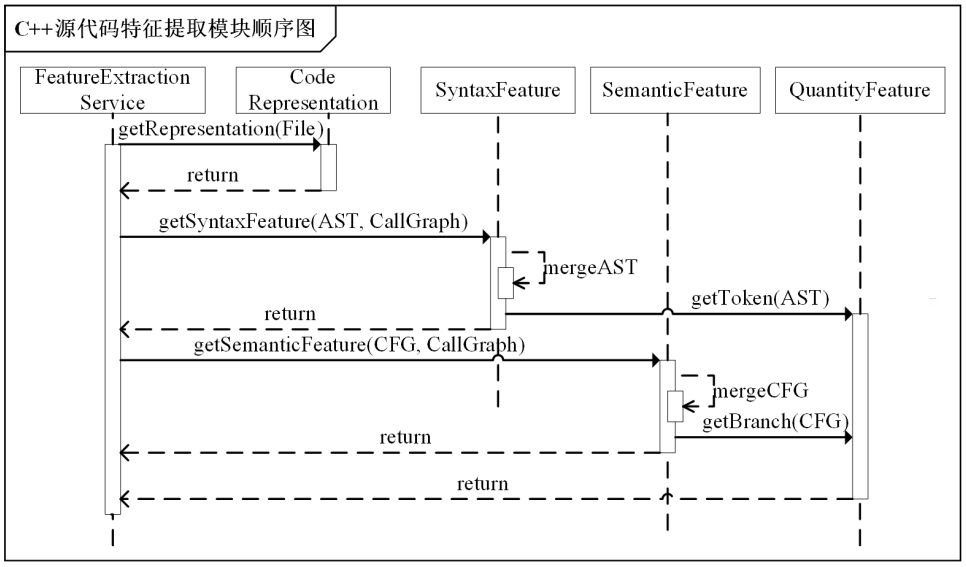


图 4.3: C++ 源代码特征提取模块顺序图

FeatureExtractionService 类是向外统一提供接口的类。首先该类调用 CodeRepresentation 类的 getRepresentation 方法获取 C++ 源代码文件的抽象语法树、控制流图、数据流图和函数调用图，该方法主要是根据开源工具 Joern 提供的接口，编写相应的 Scala 脚本来使用。然后调用 SyntaxFeature 类获取语法特征，该类需要根据函数调用图将函数内的抽象语法树连接为函数间的抽象语法树，因为使用的开源工具 Joern 只能提取函数内的抽象语法树，而很多源代码漏洞是跨函数的。因为树结构有多个叶子节点，树与树之间无法直接连接，因此先对抽象语法树进行先序遍历获取树的所有叶子节点，得到标识符序列，这样树结构的数据就变成了一维的数据，函数之间连接只需要将序列数据首尾相连即可，同时由于叶子节点包含所有的变量，也不会丢失很多语法信息。得到一维的标识符序列之后，使用 word2vec 算法进行词嵌入，可以获得每个标识符特征向量，最后使用平均池化获得函数间的语法特征向量。

接着调用 SemanticFeature 类获取语义特征，该类需要根据函数调用图将函数内的控制流图连接为函数间的控制流图。图结构有唯一的入口节点和出口节点，可以根据调用关系将控制流图连接起来。

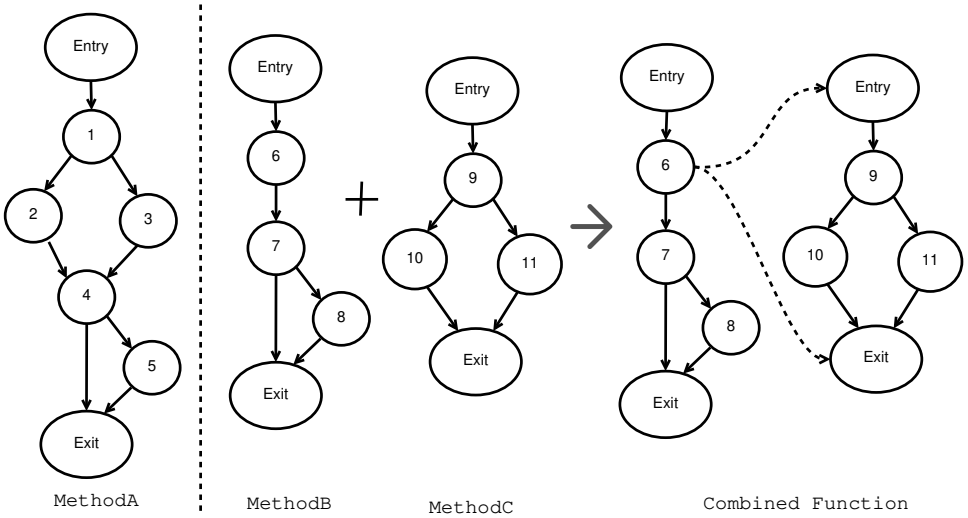
4.2.2　C++ 源代码特征提取模块具体实现

```
int A() {
    int a, b, c;
    bool flag;
    cin >> a >> b >> flag;
    if(flag) {
        c = a*b;
    } else {
        c = a+b;
    }
    if(c>0) {
        c = -c;
    }
    int res = c;
    return 0;
}
```

(a) 代码示例 1

```
int B(int a, int b, bool flag) {
    int c;
    if(flag) {
        c = a*b;
    } else {
        c = a+b;
    }
    return c;
}
int c() {
    int a, b, c;
    bool flag;
    cin >> a >> b >> flag;
    c = B(a, b, flag);
    c = c > 0 ?-c : c;
    int res = c;
    return 0; }
```

(b) 代码示例 2



(c) 函数控制流图连接方式

图 4.4: 结合函数调用图连接控制流图

数据流图依托于控制流图，是在控制流图结构的基础上添加数据传递的边，因此数据流图不需要单独说明，下面主要以控制流图为例进行说明。如图 4.4所示为连接控制流图示例，将被调函数的入口节点连接到调用语句的父亲节点，被调函数出口节点连接到调用语句的孩子节点，构成函数间的控制流图。代码示例 1 和代码示例 2 是两段 C++ 源代码，代码示例 2 中的函数 B 和函数 C 之间存在调用关系。本系统首先构造各个函数的控制流图，控制流图中的节点为代码片段，边为代码段之间的控制流，包括分支、循环等结构；接着依据函数间的调用关系来连接控制流图，图中的函数 B 和函数 C 之间存在调用关系，函数 B 中的代码段 6 调用了函数 C，则将函数 C 的入口节点连接到函数 B 中代码段 6 的父亲节点，函数 C 的出口节点连接到函数 B 中代码段 6 的出口节点，构成一个大的控制流图。得到控制流图后，使用 graph2vec 进行图嵌入，获取源代码语义特征向量。

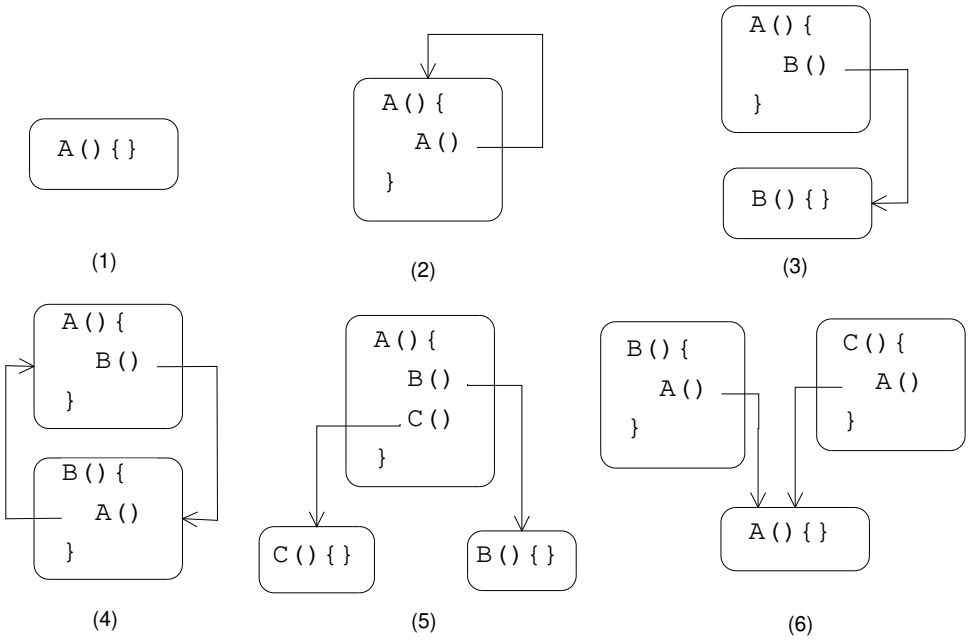


图 4.5: C++ 源代码函数调用关系分析

上文提到根据函数调用关系来连接控制流图构成函数间的控制流图，图 4.5展示了 6 种基础的函数间调用关系。从图中可以发现连接函数内的控制流图，只需要将调用节点的父亲节点和被调函数的入口节点相连，调用节点的孩子节点和被调函数的出口节点相连即可，该方法适用于各种调用关系。

4.2.3　关键代码

```
Public CFGGraph getFeatureCFG (Feature feature,
        Map<Method, CFGGraph> method2Graph) {
    Set<MethodCall> methodCalls = feature.getMethodCallSet();// 获取调用关系
    // 获取所有方法的节点和边集
    Set<CFGNode> cfgNodeSet = getAllNodes(method2Graph);
    Set<CFGEdge> cfgEdgeSet = getAllEdges(method2Graph);
    // 根据调用关系，添加边来连接 CFG
    for (MethodCall methodCall : methodCalls) {
        int lineNum = methodCall.getLineNum();
        // 获取需要添加边的节点
        CFGNode selectedNode = getSelectCfgNode(lineNum);
        // 添加入度节点
        Set<CFGNode> inNodeSet = new HashSet<>();
        // 添加出度节点
        Set<CFGNode> outNodeSet = new HashSet<>();
        // 获取被调 CFG
        CFGGraph methodCalleeCFGGraph = method2Graph.get(methodCall
            .getCalleeMethod());
        //获取 CFG 的入口和出口节点
        CFGNode entry = getEntry(methodCalleeCFGGraph);
        CFGNode exit = getExit(methodCalleeCFGGraph);
        // 添加与 entry 节点连接的边
        for (CFGNode cfgNode : inNodeSet) {
          CFGEdge edge = new CFGEdge(entry.getId(), cfgNode.getId());
          cfgEdgeSet.add(cfgEdge);
        }
        // 添加与 exit 节点连接的边
        for (CFGNode cfgNode : outFromCFGNodeSet) {
          CFGEdge edge = new CFGEdge(cfgNode.getId(), exit.getId());
            cfgEdgeSet.add(cfgEdge);
        }
    }
    return new CFGGraph(cfgNodeSet, cfgEdgeSet);
}
```

图 4.6: C++ 源代码控制流图连接关键代码

图4.6展示了控制流图连接关键代码。该方法调用的前提是根据静态调用图来分析函数之间的调用关系，并且将存在调用关系的控制流图放到集合中，这个集合本文中称为功能。首先需要获取集合中所有函数的控制流图，以节点和边来表示；然后根据具体的静态调用图找出调用关系对应的控制流图中调用节点；接着找出静态调用图中被调函数的入口和出口节点，每个函数的控制流图

有唯一的入口和出口节点；最后将函数间的节点连接起来，生成功能控制流图，由边和节点构成。

4.3　误报过滤模块实现

4.3.1　误报过滤模块顺序图

如图 4.7所示是本系统进行漏洞数据下载、误报过滤模型训练、误报过滤的调用流程，是本系统的核心部分，可以减少误报漏洞数量，提高 C++ 源代码漏洞静态扫描系统的可用性。本部分的入口是 FalsePositiveFilterService 类。首先 VulScanService 类负责进行 C++ 源代码漏洞静态扫描，并将漏洞报告融合起来，得到初始漏洞报告；其次 VulDownload 类负责从漏洞数据库中下载所有的漏洞数据集，该数据是对漏洞数据集进行漏洞扫描得到的正报、误报数据；然后将带漏洞标签的数据集送入到 ModelTraining 误报过滤模型训练类，进行模型的训练，因为深度学习网络存在冷启动的问题，在误报数据集小于 1000 的时候，使用传统机器学习分类模型 SVM 进行模型训练，当误报数据集大于等于 1000 时，使用深度学习分类模型 DNN 进行模型训练；接着使用训练得到的误报过滤器对漏洞报告进行过滤获取过滤后的漏洞报告；最后将过滤好的漏洞报告存储到数据库中，进行持久化处理。

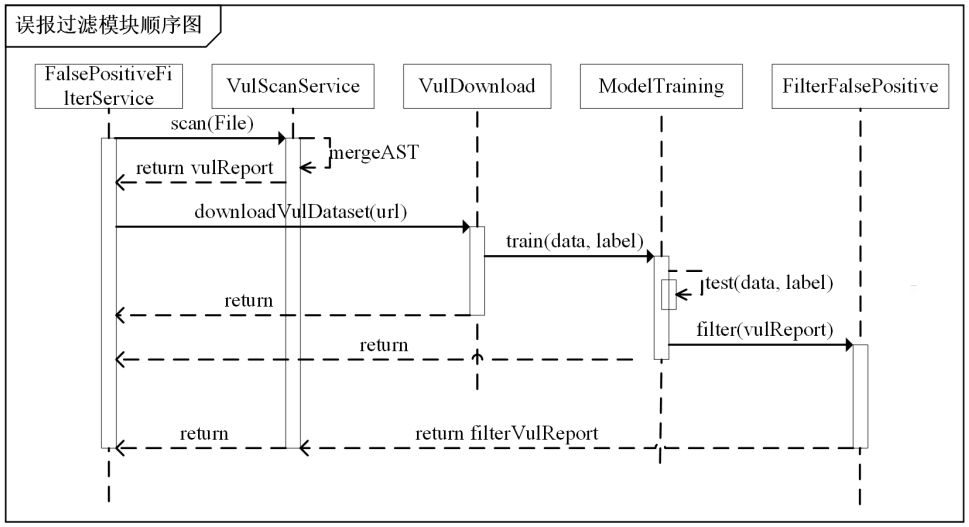


图 4.7: 误报过滤模块顺序图

4.3.2　误报过滤模型训练的实现

上文提到源代码漏洞漏洞静态扫描工具为了保证扫描效率，会进行近似操作，包括忽略分支条件。如图4.8所示，右侧的代码是漏洞，其中 *data.s* = *read*();

是污点分析中的 source 点，该数据从不可信的源获取，*data.s* 可能为空，同时 *wirte*(*data.s*); 是污点分析中的 sink 点，会导致空指针解引用漏洞，左侧的代码不是漏洞，因为污点分析中的 source 点和 sink 点在 if-else 两个分支中，不会同时运行。但是开源工具 TscanCode 会将左侧的代码判断为漏洞，因为该工具不检测控制流，认为 if 和 else 分支中的语句会同时运行，判断存在空指针解引用漏洞。本系统进行误报过滤时，会提取源代码的控制流图，从控制流图中可以清晰看到，if 语句和 else 语句中的内容位于两个不同的分支，进而判断其不是漏洞。但是单纯的分析不同的分支情况，会带来资源消耗严重的问题，因为大型项目中分支语句数量很多，而不同的分支语句组合起来可能性是指数级别。因此本系统借助于机器学习技术，使用图嵌入技术提取控制流图的结构特征、词嵌入提取抽象语法树的语法特征，将复杂的图分析问题转化为较简单的分类问题。

```
#include<iostream>
class Data{
    string s;
}
void main(int argc, char* argv[]) {
    bool flag = true;
    Data data = new Data();
    data.s = “”;
    if(flag) {
        data.s = read();
    }
    if(!flag) {
        write(data.s);
    }
}
```

```
#include<iostream>
class Data{
    string s;
}
void main(int argc, char* argv[]) {
    bool flag = true;
    Data data = new Data();
    data.s = “”;
    data.s = read();
    write(data.s);
}
```

图 4.8: 误报漏洞与正报漏洞示例

如图 4.10所示为误报过滤模型训练流程图。误报过滤模型的训练数据来源是使用漏洞扫描模块提供的接口，对带漏洞标签的源代码数据集进行扫描，将识别为漏洞的代码段和数据集标签进行对比，判断是正报还是误报，对于漏洞数据集标记为漏洞而漏洞扫描器未发现的漏洞判定为漏报，但是本系统主要目标是降低误报率来提升系统的可用性，因此不关注漏报的问题。得到正报、误报数据后，根据数据量的大小选择使用传统机器学习方法还是使用深度学习方法来训练模型，本系统中设置的阈值是 1000。

```
# 获取模型训练和预测数据
def get_training_data(csv_path):
    data = get_data(csv_path)
    label = get_label(csv_path)
    # 获取数据
    train_data = split_data(0: 0.7).float()
    test_data = split_data(0.7: 1.0).float()
    # 获取标签
    train_label = split_data(0: 0.7).long()
    test_label = split_data(0.7: 1.0).long()
    return train_data, test_data, train_label, test_label
# 模型参数设置
class Net(nn.Module):
    # 模型结构设置
    def __init__(self, n_feature, n_hidden1, n_hidden2, n_output):
        super(Net, self).__init__()
        self.hidden1 = torch.nn.Linear(n_feature, n_hidden1)
        self.hidden2 = torch.nn.Linear(n_hidden1, n_hidden2)
        self.out = torch.nn.Linear(n_hidden2, n_output)
    # 前向传播
    def forward(self, x):
        x = F.relu(self.hidden1(x))    # activation function for hidden layer
        x = F.relu(self.hidden2(x))
        x = self.out(x)
        return x

if __name__ == '__main__':
    net = Net(n_feature=128, n_hidden1=20, n_hidden2=30, n_output=2)
    # 设置优化器
    optimizer = torch.optim.Adam(net.parameters(), lr=0.001)
    # 设置 loss 函数
    loss_func = torch.nn.CrossEntropyLoss()
    train_data, test_data, train_label, test_label = get_training_data (path)
    # 模型训练
    for epoch in range(1000):
        out = net(train_data)
        loss = loss_func(out, train_label)
        optimizer.zero_grad() # 梯度消除
        loss.backward() # 反向传播
        optimizer.step()
```

图 4.9: 误报过滤 DNN 模型训练代码

4.3.3　关键代码

如图4.9所示为本系统中误报过滤模型训练关键代码。该模型训练使用主流的机器学习框架 Pytorch，因此关键代码使用 Python 脚本语言编写。该模型使用

70% 数据训练，30% 数据预测，模型采用三层线性结构，损失函数使用交叉熵损失函数，优化器使用 Adam 优化器，训练轮次为 1000，学习率为 0.001。该网络结构简单，训练效率较高。

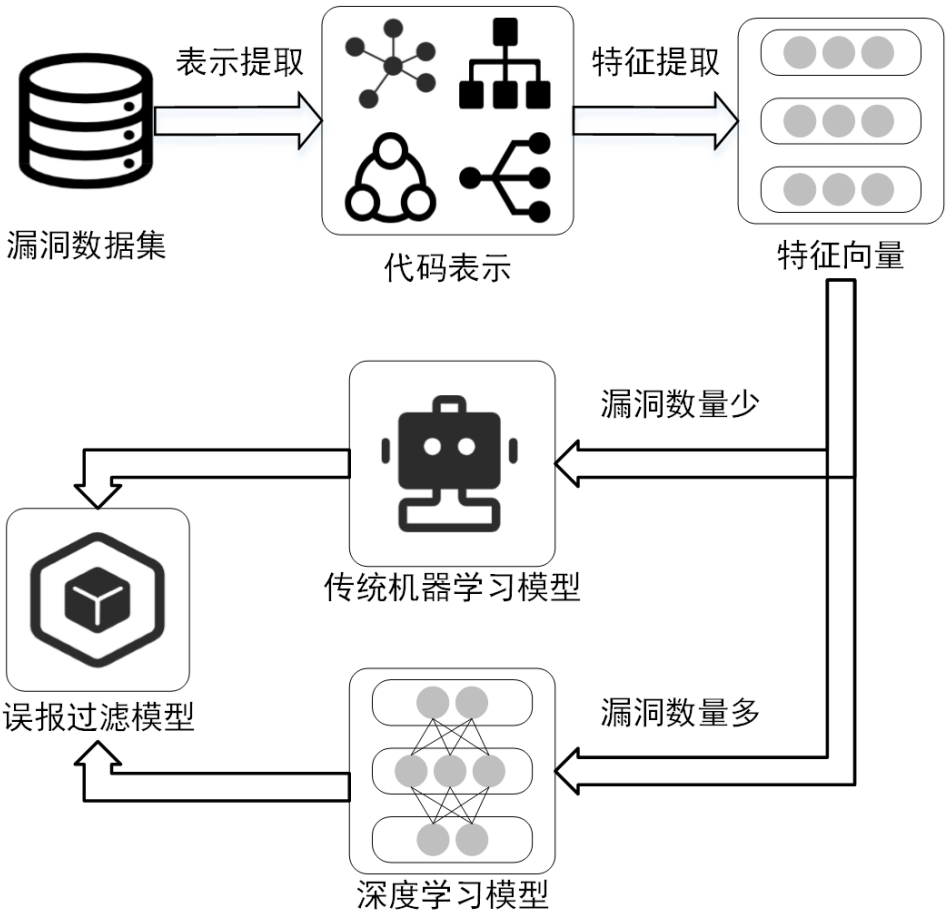


图 4.10: 误报过滤模型训练图

4.4　误报漏洞反馈模块实现

4.4.1　误报漏洞反馈模块顺序图

如图 4.11所示为误报漏洞反馈进行专家审核、相似漏洞寻找的调用流程，是本系统一个重要的创新点。FeedbackService 为本模块向外提供服务的接口。首先该模块调用 FalsePositiveFilter 类，进行 C++ 源代码漏洞静态扫描，接着对漏报报告进行初步过滤；接着漏洞审核专家对初步过滤后的漏洞报告进行人工审核，这是因为误报过滤模型对现有误报以及新出现的误报情况没有足够的识别能力；然后使用相似度工具寻找与漏洞专家审核为误报漏洞相似的漏洞，并将

其标记为误报；最后将误报漏洞加入到数据集中，用于误报过滤模型的再训练，并将更新后的模型替换上去。

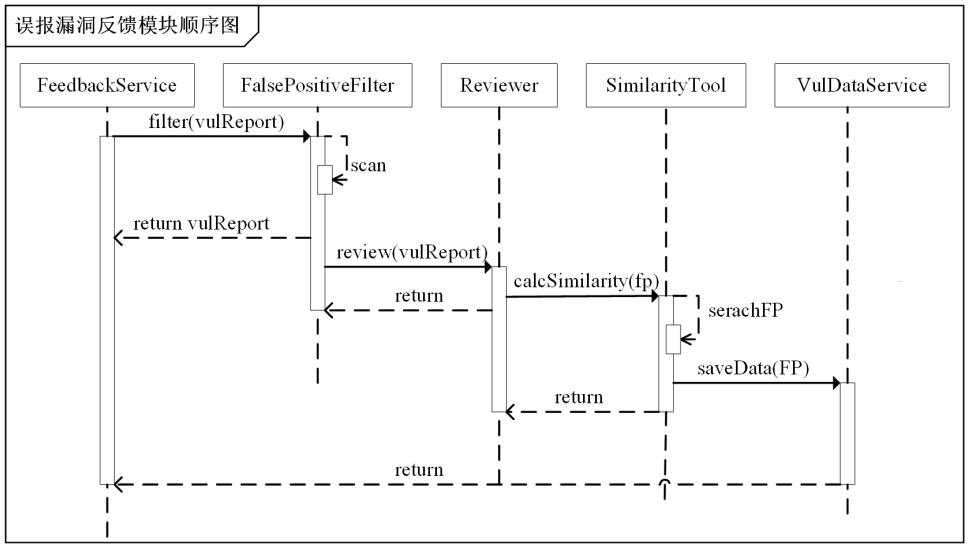


图 4.11: 误报漏洞反馈模块顺序图

4.4.2　基于相似度的误报漏洞反馈实现

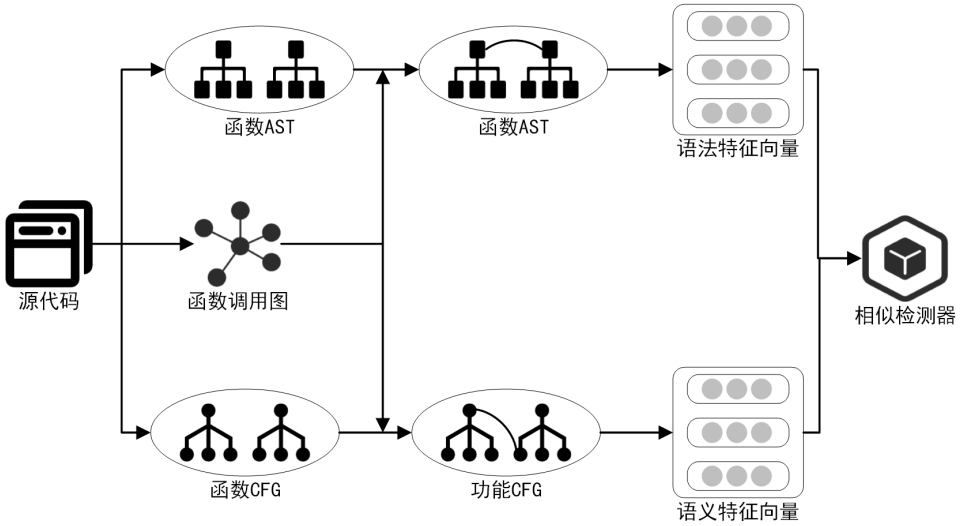


图 4.12: 代码相似度检测模型架构图

本模块的核心是漏洞代码的相似性判断，现有的一些相似度计算方法不适用于漏洞代码的相似度计算，因为漏洞经常是因为多个函数间的调用才产生的，而现有的方法大部分是计算函数内的相似度，同时漏洞代码也存在一些漏洞独有特征，因此需要设计专用于计算漏洞代码间相似度的方法。本模块结合抽象

语法树和控制流图，提取 C++ 源代码的语法特征和语义特征，并将两种特征融合来表征代码段。如图 4.13所示，Sample1 和 Sample2 不是相似代码段，Sample2 和 Sample3 是相似代码段。对于 Sample1 和 Sample2 来说，从控制流图上来看两者是相似的，但是从抽象语法树的角度来看，可以发现 Sample1 和 Sample2 中 *res*+ = *data*[*i*] 和 *res*∗ = *data*[*i*] 的抽象语法树是存在区别的，可以检测到两者是不相似的。对于 Sample2 和 Sample3 来说，两者的抽象语法树是存在很大不同的，无法识别到两者是相似代码片段，但是从控制流图的角度来看，两者的控制流图是一致的。因此本系统将抽象语法树和控制流图结合起来，分别提取它们的特征进行相似度计算。

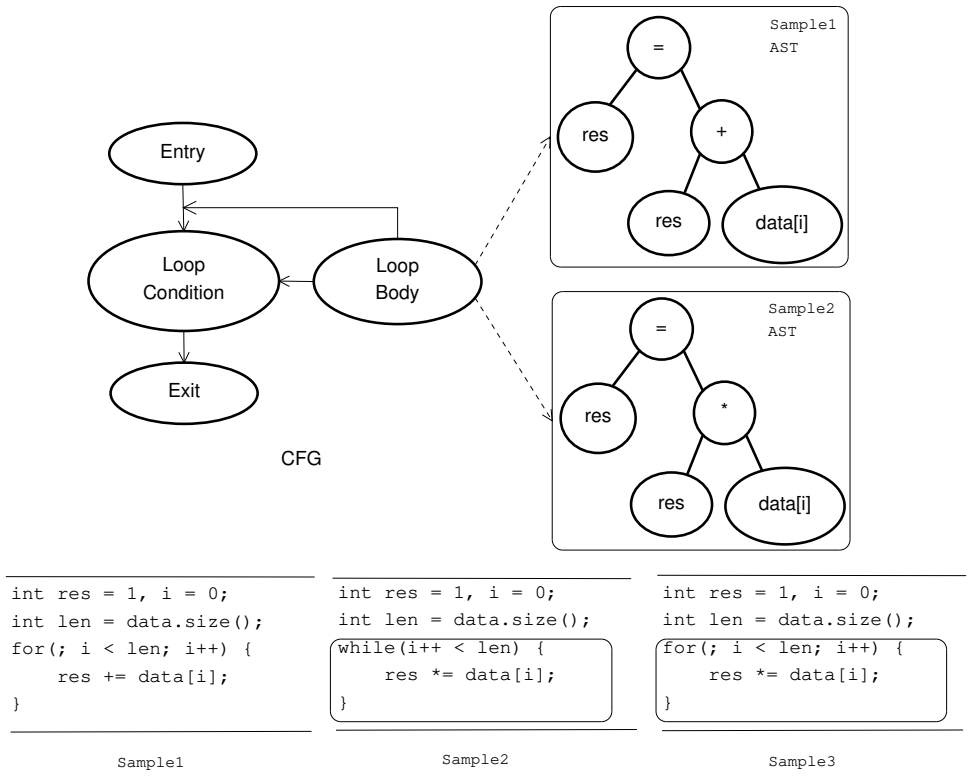


图 4.13: 语法和语义特征用于相似度计算对比图

如图 4.12所示为代码相似度检测模型架构图。首先系统提取源代码的抽象语法树、控制流图和函数调用图；接着根据函数调用图分别连接函数内的抽象语法树和控制流图；然后对连接后的抽象语法树使用 word2vec 算法获取语法特征向量，对连接后的控制流图使用 graph2vec 算法获取语义特征向量；最后将语法特征向量和语义特征向量一起送入 DNN 模型中进行模型的训练。该模型本质上是一个二分类器，输入一对代码片段的特征向量，输出为相似或不相似。

4.5　C++ 源代码漏洞静态扫描系统的实现

图 4.14为创建源码扫描任务的页面，首先需要上传源码文件夹，格式为 zip 或 tar.gz；接着选择服务类型，包括 Java 字节码扫描和 C++ 源码扫描，本系统是针对 C++ 源码扫描；然后填写项目名；最后填写项目描述。

图 4.14: 创建源码扫描任务

图 4.15为漏洞扫描任务审核列表页面，可以查看所有的漏洞扫描任务的信息，包括测试类型、提测时间、任务名称、目标名称、提测人、测试进度，其中测试进度分为任务待审核、任务审核通过、任务审核不通过、创建众审报告成功、执行完成等状态。同时还支持按任务目标关键字搜索和测试进度进行筛选，该功能方便用户快速定位到未启动任务。

图 4.15: 漏洞扫描任务审核列表

图 4.16所示为用户拥有的漏洞扫描任务列表。该页面和漏洞扫描任务审核列表页面很相似，包含提测时间、任务名、扫描文件、测试服务、测试进度、操作。其中操作栏初始只有执行一个按钮，点击执行按钮后系统会向服务端发送漏洞扫描异步请求，服务端接收到漏洞扫描任务请求后，会对漏洞扫描任务进行排队，等到服务器有空闲的时候执行漏洞扫描任务。

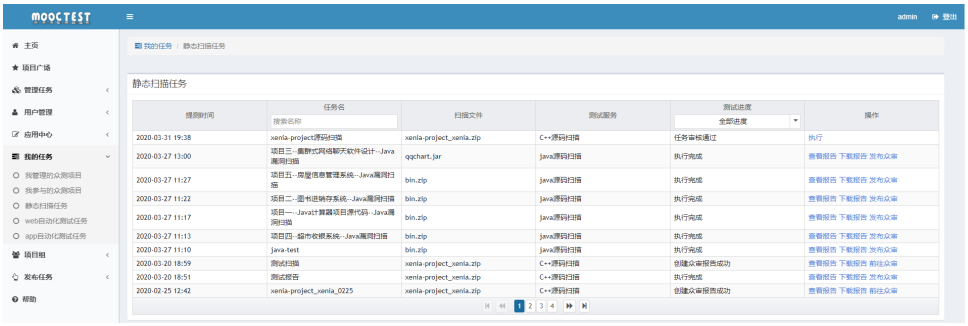


图 4.16: 用户漏洞扫描任务列表

图 4.17所示为源码安全扫描报告的结果概述。该页面分为基本信息和统计信息两个部分，基本信息包括项目名称、提测用户、提测时间、扫描耗时，统计信息包括按信息、警告、低危、中危、高危五个风险等级分类的统计信息以及按漏洞类型进行分类的漏洞数量统计信息。在统计图中点击扇形图，可以跳转到相应的漏洞列表页面。

图 4.17: 漏洞扫描报告结果概述

图 4.18所示为源码安全扫描报告漏洞列表页面。该页面包含报告中所有的漏洞项，每一个漏洞项报告漏洞名称、漏洞类型、漏洞等级和查看详细信息的详情查看按钮。并且支持按漏洞类型和漏洞等级进行筛选。

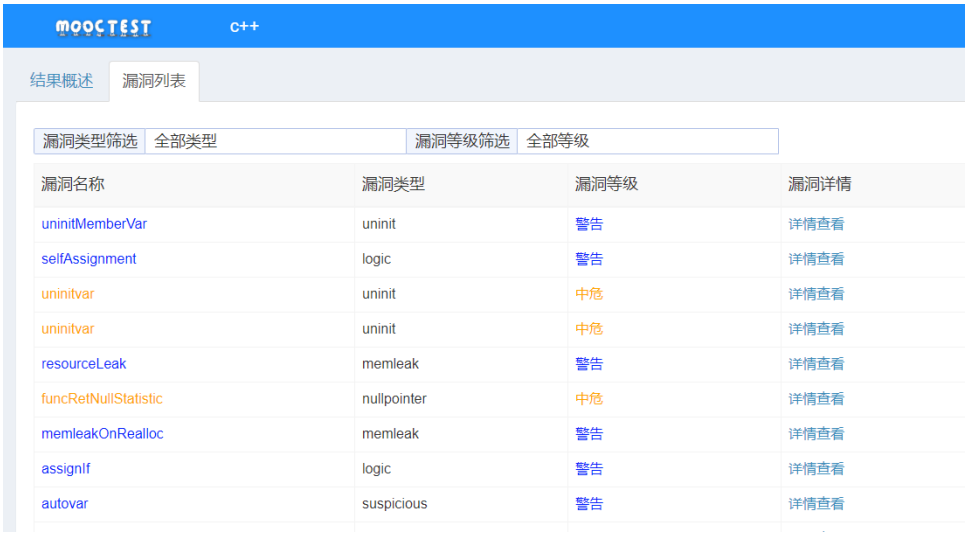


图 4.18: 漏洞扫描报告漏洞列表

图 4.19所示为漏洞详情页面，可以查看漏洞的详细信息。该页面包含漏洞名称、风险等级、漏洞类型、漏洞概述、漏洞位置（精确到文件中的代码行）、漏洞上下文（漏洞位置上下文代码，方便用户查看漏洞具体信息）、参考链接（对标到 CWE 的漏洞信息）。

图 4.19: 漏洞详细信息

图 4.20所示为发布众审任务页面，提供查看报告信息、编辑报告信息、定制审查选项等功能。该页面同时还支持上传附件和众审需求文件和以 JSON 文件

格式编辑报告信息，修改其中的内容，并进行保存。定制审查选项功能支持对于指定漏洞项设置审查项，设置审核标签和数据说明，同时支持文件框、单复选框、多选框。

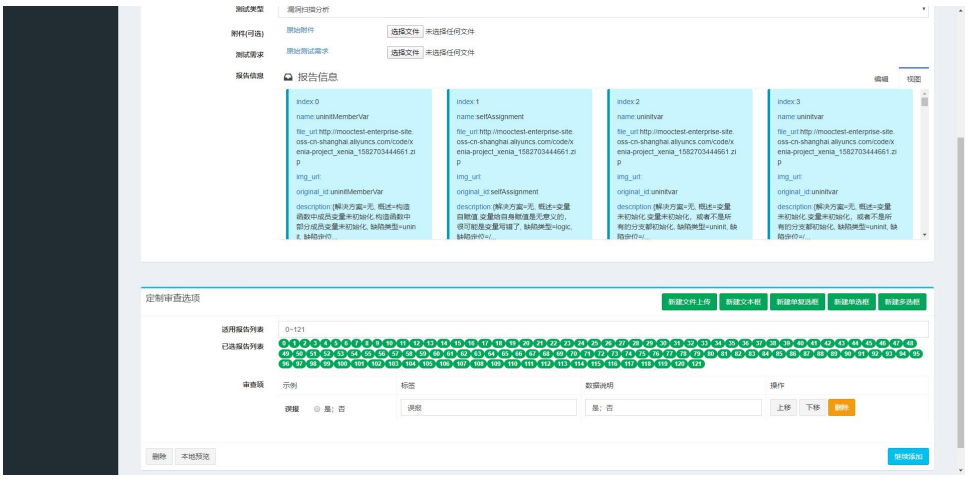


图 4.20: 发布众审页面

图 4.21所示为漏洞扫描众包审核页面。对于发布为众审任务的漏洞进行人工审核，中间可以看到漏洞信息，包含漏洞名称、解决方案、风险等级、漏洞上下文，众包工人根据这些信息对漏洞进行人工审核，右侧是待审核漏洞列表，众包工人可以选择需要审核的漏洞进行审核。

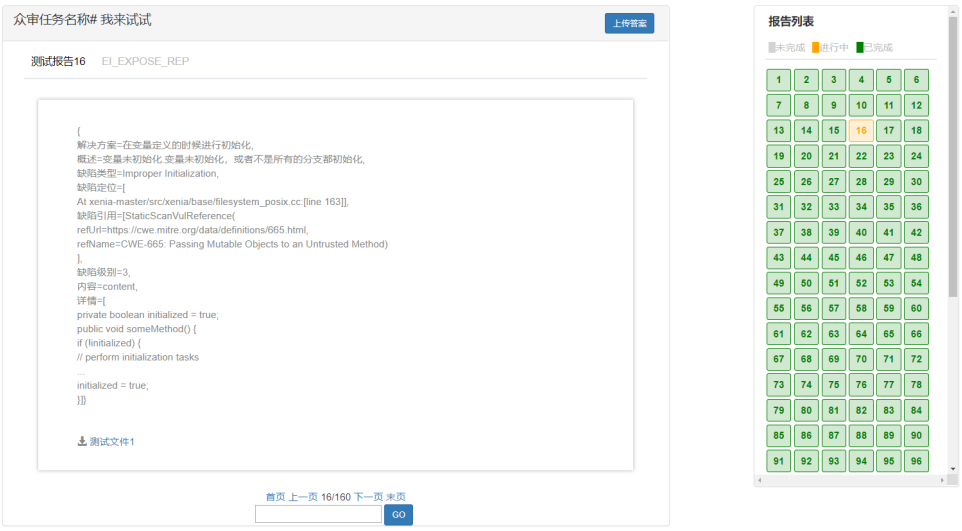


图 4.21: 漏洞扫描众包审核

4.6　本章小结

本章主要是对 C++ 源代码漏洞静态扫描的实现细节进行描述。按照模块划分，首先对 C++ 源代码漏洞静态漏洞扫描模块的漏洞扫描和融合部分使用顺序图和关键代码进行详细描述；其次对 C++ 源代码特征提取模块中提取抽象语法树、控制流图、数据流图和函数调用图等代码表示，以及代码表示间的融合和使用词嵌入和图嵌入算法提取语法特征、语义特征、统计特征的流程，借助顺序图和关键代码进行详细描述；然后对误报过滤模块中根据数据集大小选择使用传统机器学习算法还是深度学习算法来训练误报过滤器进行描述；接着对基于相似度的误报漏洞反馈模块中的融合语法和语义特征进行代码相似度计算的方法进行了详细的描述；最后对系统各主要界面以及相应功能进行了介绍。

第五章　系统测试与实验分析

5.1　测试准备

5.1.1　测试目标

本系统的软件测试主要针对三方面进行，第一个是验证软件的可靠性，尽可能减少软件中存在的缺陷，以及保证各种情况下系统能够正常运行；第二个是验证系统是否能够提供相应的功能和以及功能的效果相比于现有方法是否有提升；第三个是验证系统的性能是否在用户可以接受的范围内。

根据第三章对 C++ 漏洞静态扫描系统的需求分析、架构设计和第四章的实现，本章的测试与分析将包括以下几个方面：

可靠性测试。该测试主要是针对系统在各种情况下是否可以正常运行，以及可以持续运行的时间是否满足要求，保证系统能够为用户提供可靠的服务。

功能测试。该测试主要是检查系统是否能够提供相应的服务，还有就是系统提供服务的效果相比于现有方法在效果上是否有提升。

效果测试。该测试主要是检查系统提供服务的效果是否满足用户需求，误报率是否在一个用户可以接受范围内，证明系统是真实可用。

5.1.2　测试环境

本系统作为独立服务进行部署并进行测试，测试配置如表5.1所示。协同使用服务器资源，共同为 C++ 源代码漏洞扫描系统提供服务。

表 5.1: 系统测试环境

设备与软件	描述信息
漏洞扫描服务器	1 台，Ubuntu 16.04 LTS，JDK 1.8，2 核，4G 内存，10M 带宽
误报过滤服务器	1 台，Ubuntu 16.04 LTS，Python 3.6，2 核，4G 内存，10M 带宽
数据库服务器	2 台，Ubuntu 16.04 LTS，MySQL 5.7，2 核，4G 内存，10M 带宽
部署软件	Jenkins 2.222.1 LTS

5.2　可靠性测试

本小节的测试目标是检测系统的可靠性，即系统应对各种情况的输入以及处于各种外界环境下，能够正常运转以及在无法正常运转的情况下，能够有相应的反馈信息，以证明系统设计符合工业界要求的可靠性。

5.2.1　测试设计

该部分测试的主要思路是在各种常见的外部环境条件下，使用特定的数据集对系统进行测试，观察系统是否可以正常运行，以及系统可以持续运行的时间是否满足要求。具体步骤是，首先是模拟各种外部环境，包括弱网络环境、高并发请求等，然后监控系统是否保持正常运行，有没有出现崩溃、卡死的现象，最后在一个时间周期内，对系统进行监控，观察系统持续运行的能力。

下面介绍该部分测试使用的数据集。本系统中使用的数据集主要来源于两个部分，包括 Github 项目[1]和 Juliet 测试集[2]。其中 GitHub 项目是使用 GitHub API V3，从 GitHub 网站上根据 star 数目排序爬取的 top200 的 C++ 项目，下面选取几个典型的 C++ 项目的来介绍数据集概要情况，如表 5.2所示。

表 5.2: GitHub 数据集概要信息

项目名称	语言	文件数	代码行数	链接地址
360Controller	C++	50	8257	https://github.com/360Controller
abseil-cpp	C++	522	162101	https://github.com/abseil/abseil-cpp
git-crypt	C++	26	5002	https://github.com/AGWA/git-crypt
aria2	C++	1189	188315	https://github.com/aria2/aria2
xgboost	C++	156	33488	https://github.com/dmlc/xgboost

Juliet 是一个用 C/C++ 语言编写的测试用例集合，它包含 118 类 CWE 漏洞示例代码，以及 64099 个独立的测试用例。该数据集是人工构造而成，文件名和函数名都遵循一定的规则，并且文件名和函数名中包含漏洞标签。例如，

[1]https://github.com/

[2]https://samate.nist.gov/SRD/testsuite.php

*CWE*415*_Double_Free__malloc_free_char_*01*.c* 文件中的函数 *CWE*415*_Double_Free__malloc_free_char_*01*_bad*() 为漏洞函数，双短划线前的表示 CWE 漏洞 ID，双短划线后的 *malloc_free* 为具体漏洞名称，char 表示特定变量名称，01 表示流复杂度，数字越高复杂度越高。该人造数据集主要是用来评估 C++ 源代码漏洞静态检测系统，因为工业界实际项目用来评估存在一些缺陷，如下所示。

评估工具的结果来确定其正确性。当一个漏洞检测工具对实际项目进行检测，漏洞报告每个结果需要进行审查以确定指定位置上是否真的存在此缺陷。

比较不同漏洞检测工具的结果。比较漏洞检测工具在实际项目代码上是复杂的，因为不同的工具展示结果的方式不同。比如有的工具报告污点分析的源和汇，然而有的工具只报告污点分析的汇。

识别工具找不到的代码缺陷。在评估漏洞检测工具时，需要有代码中所有漏洞的列表，以便识别工具没有报告的缺陷。在实际项目中没有工具可以识别这些缺陷，只能通过人工代码检查来发现。

使用 C++ 漏洞静态扫描系统对上述两个数据集中的项目进行顺序扫描，通过 fiddler 创造弱网络环境，同时测试全程使用 Prometheus 进行监控。表 5.3展示了测试用例设计的细节。

表 5.3: 系统可靠性测试用例

测试 ID	TC1
测试接口	源代码漏洞静态扫描接口
测试工具	Selenium、Grafana、Prometheus
测试方案	1. Selenium 脚本设置对 GitHub 和 Juliet 两个数据集的漏洞扫描自动化测试任务； 2. 使用 Prometheus 对系统的运行情况进行监控，Grafana 负责展示监控页面状态。

5.2.2　测试执行

图5.1左侧展示了对于 GitHub 项目数据集和 Juliet 数据集 [47] 测试的结果。从结果中可以看到本系统测试用例的通过率达 95%，剩余的未通过的的测试用例，是因为使用的 rar4 的压缩技术，在 Linux 系统无法解压缩，还有就是扫描时间超时，本系统设定的超时时间为 3h。右侧展示了漏洞的统计信息，从图中可以看出来空指针异常（nullpointer）和未初始化变量（uninit）是数量最多的两类漏洞，这两类漏洞的危害程度也是很高的，可能会导致系统的崩溃。平均漏洞数量为 4.6 个/千行，单项目最高漏洞为 1074 个。

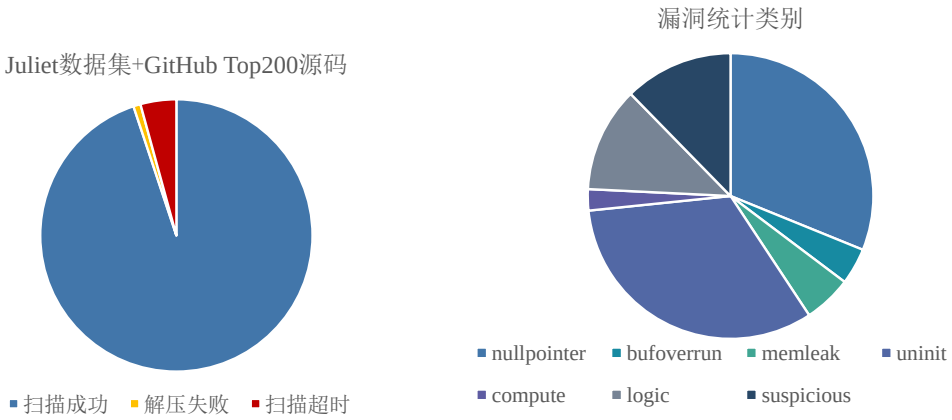


图 5.1: 漏洞扫描结果统计图

5.3　功能测试

本小节测试的目标是检测 C++ 源代码漏洞静态扫描系统的功能可以满足用户需求，以及系统提供功能的效果相比于现有的工具的提升，以证明系统是符合用户实际需求，具备一定的可用性。

5.3.1　测试设计

表 5.4: 查看扫描任务列表测试用例

测试 ID	TC2
测试名称	查看扫描任务列表
测试功能	系统用户可以查看、按关键字搜索、排序漏洞扫描任务列表，筛选未开始、已完成的漏洞扫描任务，显示漏洞扫描任务的进度。
测试步骤	1. 系统用户登录； 2. 筛选正在排队漏洞扫描任务； 3. 筛选正在执行漏洞扫描任务； 4. 输入任务名关键字； 5. 点击执行按钮。
预期结果	1. 显示所有正在排队漏洞扫描任务； 2. 显示所有正在执行的漏洞扫描任务； 3. 显示关键字相关的漏洞扫描任务； 4. 任务进度变更为正在执行。

本小节是本系统进行测试的重点，根据第三章需求分析中提到的功能需求列表来设计测试用例，面向系统提供的功能进行测试，达到功能覆盖的标准，目标是保证最终的系统符合要求，能够顺利的提供预期的功能，基本满足支撑 C++ 源代码漏洞扫描系统业务需求。

表 5.4展示了查看扫描任务列表的具体测试过程和预期结果，该测试用例是针对用例描述 UC1 来设计的，主要关注点是覆盖查看扫描任务列表功能的各项细节。

表 5.5展示触发漏洞扫描测试用例，测试的是系统的基础功能源代码漏洞扫描是否能够正确的被触发，主要关注点是页面上的状态变化是否正确，以及扫描完成后能否正常生成报告。

表 5.5: 触发漏洞扫描测试用例

测试 ID	TC3
测试名称	触发漏洞扫描测试
测试功能	系统用户可以触发漏洞扫描流程，扫描结束后显示“查看漏洞报告”按钮。
测试步骤	1. 点击“执行”按钮，任务状态是否由“任务审核”通过转变为“正在排队”； 2. 排队一段时间后任务状态是否转变为“正在执行”； 3. 执行完成后是否显示“查看报告”、“下载报告”、“发布众审”，是否隐藏“执行”按钮； 4. 点击“查看报告”按钮是否显示漏洞报告。
预期结果	1. 任务状态由“任务审核”转变为“正在排队”； 2. 任务状态由“正在排队”转变为“执行中”； 3. 操作栏的“执行”按钮消失，“查看报告”、“下载报告”、“发布众审”按钮显示； 4. 显示漏洞扫描报告页面。

表 5.6展示了触发误报过滤测试用例，测试的系统进行误报过滤模型的训练以及过滤漏洞报告中误报的能力，主要关注点是误报过滤模型的训练是否可以顺利完成，并对漏洞报告进行过滤。

表 5.6: 触发误报过滤测试用例

测试 ID	TC4
测试名称	触发误报过滤测试
测试功能	系统用户可以触发误报过滤，系统会进行误报过滤模型的训练，以及对漏洞报告的过滤。
测试步骤	1. 点击“触发误报过滤”按钮，系统是否会进行误报过滤模型训练，显示“模型训练中”； 2. 模型训练结束后，系统是否会对当前扫描任务进行重新扫描，并过滤漏洞报告中的误报。
预期结果	1. 系统根据误报数据再训练模型，显示“模型训练中”； 2. 误报过滤模型重新训练后，系统使用新的模型对当前漏洞报告再过滤。

表 5.7展示了查看漏洞详细信息测试用例，测试的查看漏洞报告中漏洞项详细信的能力，主要关注点是漏洞信息的是否完整，展示形式对于查看漏洞关联代码是否便捷。

表 5.7: 查看漏洞详细信息测试用例

测试 ID	TC5
测试名称	查看漏洞详细信息测试
测试功能	系统用户可以查看漏洞详细信息，包括漏洞名称、风险等级、上下文代码等信息。
测试步骤	1. 用户点击“查看报告”按钮，是否跳转到漏洞报告页面； 2. 点击“结果概述”按钮，是否展示漏洞报告概述信息； 3. 点击漏洞等级和漏洞类型饼状图，是否跳转到相应漏洞； 4. 点击“漏洞类型筛选”-“nullpointer”和“漏洞等级筛选”-“高危”下拉框，是否按照预设的条件筛选漏洞； 5. 点击“详情查看”按钮，是否显示漏洞详细信息。
预期结果	1. 跳转到漏洞报告页面，页面包含结果概述和漏洞列表； 2. 显示扫描任务基本信息，包括项目名称、提测用户、提测时间、扫描耗时，以及漏洞风险等级和漏洞类型统计信息； 3. 显示“nullpointer”类型中的所有“高危”漏洞； 4. 显示漏洞详情页面，包含漏洞更详细信息。

表 5.8展示了查看漏洞审核列表测试用例，测试的是漏洞专家查看漏洞审核列表的功能，主要关注点是漏洞专家是否可以快速的按照要求筛选出待审核的漏洞，并进行漏洞审核。需要测试的具体功能包括漏洞专家查看待审核漏洞列表，漏洞专家查看被误报过滤器判断为误报的漏洞。漏洞专家需要对判断为误报的漏洞进行人工审核是因为误报过滤器的准确率无法达到百分之百。

表 5.8: 查看漏洞审核列表测试用例

测试 ID	TC6
测试名称	查看漏洞审核列表测试
测试功能	漏洞专家可以查看需要审核漏洞列表，可以按照风险等级、类型进行过滤，可以按照风险等级对漏洞列表进行排序。
测试步骤	1. 漏洞专家点击“漏洞审核”按钮，页面是否跳转到漏洞审核列表页面，并且展示所有待审核的漏洞； 2. 点击漏洞类型饼状图中的 nullreference 漏洞类型，是否跳转到相应漏洞类型的的待审核漏洞列表； 3. 点击漏洞风险等级饼状图中的高危漏洞，页面是否跳转到相应高危漏洞类型的待审核漏洞列表； 4. 点击“审核状态”-“未审核”下拉框，是否按照条件筛选出所有的未审核漏洞列表； 5. 点击“误报漏洞列表”，是否显示所有被误报过滤器判定为误报的漏洞列表； 6. 点击风险等级标签栏，是否按照漏洞风险等级进行排序。
预期结果	1. 系统跳转到漏洞审核页面，显示漏洞报告概述信息； 2. 跳转到漏洞列表页面，显示 nullreference 类型漏洞列表； 3；跳转到高危漏洞列表页面，显示高危等级漏洞列表； 4. 显示所有的未审核漏洞列表，已审核的漏洞隐藏； 5. 将所有判断为误报漏洞显示出来，并且按照风险等级从高到低进行排序，便于审核人员查看； 6. 按照漏洞风险等级对漏洞列表从高到低进行排序。

表 5.9展示了审核漏洞测试用例，测试的漏洞专家对漏洞项进行实际审核时，展示的漏洞信息是否全面以及展示形式是否方便、合理，主要关注点是页面信息的全面性以及漏洞信息的可视化程度。

表 5.9: 审核漏洞测试用例

测试 ID	TC7
测试名称	审核漏洞测试
测试功能	漏洞专家审核单独的漏洞项时，可以查看漏洞名称、风险等级、误报概率、可视化的程序切片等信息，为漏洞专家提供审核依据。
测试步骤	1. 漏洞专家点击“查看详情”按钮，是否跳转到审核页面； 2. 点击“参考链接”按钮，是否可以打开参考信息网页； 3. 点击“漏洞上下文”按钮，是否可以显示漏洞相关源代码； 4. 点击“审核”按钮，是否可以对漏洞项进行审核。
预期结果	1. 系统跳转到漏洞审核页面，可以查看到漏洞的基础信息； 2. 系统打开新的页面，内容是参考链接对应的内容； 3. 系统打开新的页面，页面包含漏洞的上下文源代码； 4. 漏洞专家对漏洞进行审核，标记误报或正报，未审核的使用误报过滤器结果。

5.3.2　测试执行

表 5.10展示了功能测试的结果，测试人员严格按照测试用例中设计的步骤执行，并将实际结果与预期结果进行对比，并将结果进行记录。从表中可以发现，本系统满足了需求分析阶段设计的功能性需求，具有一定的实际价值，可以投入实际使用。

表 5.10: 功能测试用例执行结果

测试用例 ID	对应用例描述	测试结果
TC2	UC2	通过
TC3	UC3	通过
TC4	UC4	通过
TC5	UC5	通过
TC6	UC6	通过
TC7	UC7	通过

5.4　效果测试

本小节效果测试的目标是检测系统使用的 C++ 源代码漏洞静态检测方法是否可以检测到漏洞，以及本系统产生的漏洞报告相比于目前主流的开源工具

TscanCode 和 Cppcheck 是否减少了误报数，并且在减少误报的同时，不会大幅度增加漏洞，增加安全风险，以证明本系统提出的方法是有效的。

5.4.1　测试设计

本次测试的基本思路是通过对比本系统提出的方法和开源工具在相同的 C++ 项目上，即带有漏洞标签的 Juliet 数据集，进行漏洞扫描，比较它们结果的误报率和漏报率。为了保证对比实验的有效性，需要控制变量，因此测试都是在相同配置的服务器上进行的。还有测试目标是选取的相同的 Juliet 数据集中的 C++ 源代码项目，一方面是控制变量，另一方面是便于进行漏报率和误报率的计算。本系统主要目标是降低误报率，但是降低误报率的同时还需要尽可能的保证漏报率不会大幅提升，保证系统依然可以检测出大部分的漏洞，即系统是可用的。因为每一个漏洞都很可能会导致软件的崩溃，而对于投入使用的软件来说，每一次的系统崩溃对于用户体验的影响都是巨大的，造成巨大的经济损失。下面对漏洞扫描功能进行测试，具体测试步骤如表 5.11所示。

表 5.11: 漏洞扫描测试用例

测试 ID	TC8
测试接口	基础漏洞扫描接口
测试功能	系统对 Juliet 数据集进行漏洞扫描，并进行误报过滤，检测系统的误报率和漏报率。
测试方案	1. 在系统中创建漏洞扫描任务，上传 Juliet 项目； 2. 点击“执行”按钮，系统进行漏洞扫描以及误报过滤，获得扫描 Juliet 源码的漏洞报告； 3. 使用开源工具 Cppcheck 和 TscanCode 对 Juliet 源码数据集进行漏洞扫描，获得漏洞报告； 4. 根据 Juliet 数据集标签，计算本系统方法和开源工具的漏报率和误报率，并进行比较。

本系统另一个需要测试效果的是将相似度检测加入到扩增误报数据集中是否有效。这是本系统提出的方法，将漏洞专家审核和漏洞相似度检测结合起来，漏洞专家可以挑选部分漏洞进行审核，对于和当前审核漏洞相似度很高的漏洞可以不需要审核，可以降低专家审核成本和扩增误报数据集。本测试用例需要测试该方法是否有效，具体测试步骤如表5.12所示。

表 5.12: 专家审核反馈机制测试用例

测试 ID	TC9
测试接口	专家审核反馈接口
测试功能	漏洞专家审核漏报报告，并进行相似度检测，扩增误报数据集，重训练误报过滤器。
测试方案	1. 漏洞专家对漏洞报告进行人工审核，标记误报漏洞； 2. 使用相似度检测算法，在漏洞报告中寻找相似漏洞，并标记为误报； 3. 使用新的数据集，重新训练误报过滤器，并进行误报过滤； 4. 根据 Juliet 数据集标签，计算新的漏报率和误报率，并之前的结果进行比较。

5.4.2　测试执行

如图5.2所示，本系统提出基于 DNN 模型进行误报过滤方法以及结合相似度检测的专家反馈机制应用于漏洞扫描，从结果来看相比开源工具 Cppcheck 和 TscanCode，在 precision 上提升了超过 20%，同时 recall 值相比于 Cppcheck 也未降低很多，并且综合指标 F1 值也提升了超过 20%，因此本系统提出的方法是真实有效的。另外从图中可以看出来 Cppcheck 在 precision、recall 等方面表现都比 TscanCode 更好，但是因为 TscanCode 可以扫描到 Cppcheck 未涉及的一些漏洞，比如 CWE606[3]未检测的循环条件等，因此还是需要使用 TscanCode 来扫描更多的漏洞。

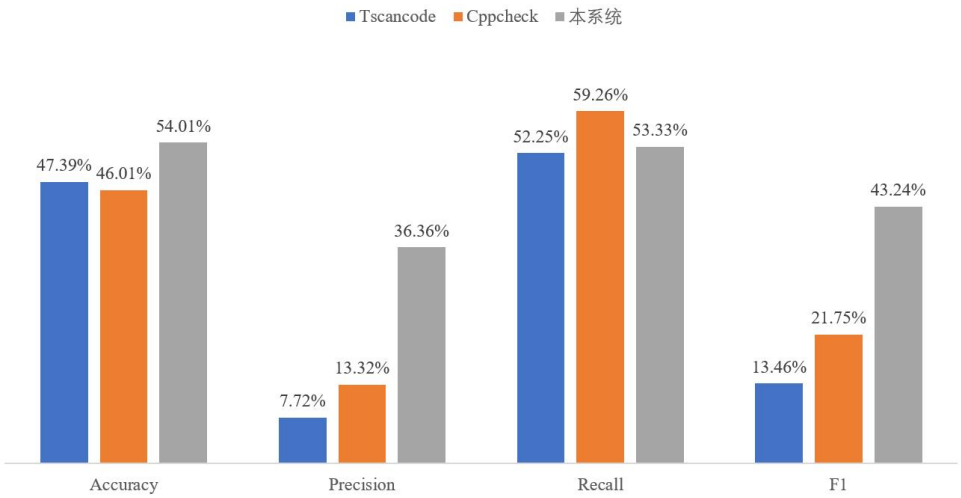


图 5.2: 漏洞扫描结果对比统计图

[3]https://cwe.mitre.org/data/definitions/606.html

本系统对于空指针解引用、内存管理例程不匹配、重复释放内存、错误参数数量函数调用、除零异常这五种漏洞进行单独的漏洞静态扫描，测试系统提出的基于机器学习的迭代反馈式误报过滤机制适用于哪些漏洞类型。与上文实验不同的是，这部分实验对于每一种漏洞类型，单独训练误报过滤器，进行实验评估与分析，具体结果如表5.13所示，其中 P 代表 precision，R 代表 recall。从表中可以看到，本系统提出的方法相比于开源工具 Cppcheck 和 TscanCode 漏洞静态扫描结果的融合在空指针解引用、内存管理例程不匹配、错误参数数量函数调用、除零异常这四种类型漏洞在 precision 上均有显著提升，recall 有所下降，但综合指标 F1 值还是有显著提升。这是因为当前误报过滤器是使用每种漏洞类型对于的漏洞数据进行模型训练，对应到每种漏洞类型的数据量较小，可能会让误报过滤模型无法达到全局最优。同时，误报过滤模型准确率无法达到百分之百，会将真实漏洞判断为误报，导致 recall 值有所下降。重复释放内存漏洞的 precision 保持不变，但是 recall 有所下降。这是因为本系统使用的代码分析工具 Joern 无法分析全局变量，而该漏洞类型存在全局变量的使用，无法提取到相应语法和语义特征，导致无法识别部分漏洞。

表 5.13: 不同漏洞类型漏洞静态扫描结果分析

工具名称	Cppcheck + TscanCode			本系统方法		
漏洞名称	P	R	F1	P	R	F1
空指针解引用	0.34	0.94	0.49	0.66	0.70	0.68
内存管理例程不匹配	0.09	1.0	0.17	0.63	1.0	0.77
重复释放内存	0.24	1.0	0.39	0.24	0.77	0.37
错误参数数量函数调用	1.0	1.0	1.0	1.0	1.0	1.0
除零异常	0.02	1.0	0.05	0.26	0.72	0.38

如图5.3所示，左右两段代码功能上是相似的，同时两段代码都在 printf 函数处产生了空指针解引用的误报。在漏洞专家审核完左边的漏洞后，使用相似度算法发现右边的代码和左边的代码是高度相似度的，因此可以直接将右边的代码判定为误报。本系统提出的专家审核反馈结合相似度计算的方法，是可以将人和机器很好的结合起来，用人工审核的结果来进一步减少误报漏洞，降低误报率，对于提升本系统的可用性很有效。

```
#include <stdio.h>

int main ( void) {
   for(;;){
      char * ch;
      char * &chRef = ch;
      // Warning : ch 存储的值未读.
      ch = "Good" ;
      printf ( "%c \n" , chRef [ 0 ] ) ;
      return 0 ;
   }
}
```

```
#include <stdio.h>

int main ( void) {
   while(1){
      char * data;
      char * &dataRef = data ;
      // Warning : data 存储的值未读.
      data = "Bad" ;
      printf ( "%c \n" , dataRef [ 0 ] ) ;
      return 0 ;
   }
}
```

图 5.3: 相似误报漏洞代码示例图

5.5　本章小结

本章主要是对 C++ 源代码漏洞静态扫描系统的功能需求和非功能需求进行详尽的测试。对于功能需求的测试是针对第三章描述的功能逐一进行测试，验证系统是否可以提供必要的功能，满足用户需求。对于非功能需求的测试是对系统可靠性以及系统提出方法的有效性进行测试。经过测试系统具备一定的可靠性，可以满足用户对于系统鲁棒性需求。最后，对于系统中提出的降低 C++ 源代码漏洞静态扫描误报率的方法进行测试，验证了该方法确实可以降低误报率，同时对于漏报率增加较小。

第六章　总结与展望

6.1　总结

C++ 源代码漏洞静态扫描漏洞报告数量大、误报率高，给开发人员审核报告带来了工作压力大、无效报告审核数量多的问题。为解决这些问题，本系统设计开发了结合机器学习技术过滤误报的 C++ 源代码漏洞静态扫描系统。

用户在对 C++ 源代码进行漏洞静态扫描时，通过浏览器访问系统，执行漏洞扫描任务，系统会使用漏洞静态扫描工具集对 C++ 源代码进行扫描，得到融合后的漏洞报告。接着使用系统当前的误报过滤器对漏洞报告进行过滤，获取过滤后的漏洞报告，并返回前端页面展示，显示包括漏洞名称、误报标签、漏洞位置、上下文信息、风险等级、解决方案和参考链接等信息。漏洞扫描任务完成后可以选择转为专家审核任务，漏洞专家对部分漏洞报告进行人工审核，并使用相似度算法搜索误报漏洞相似漏洞。系统返回专家审核后报告，并将搜索到的漏洞数据加入误报过滤器训练集。

该系统结合了机器学习中分类技术的思路，将漏洞报告误报过滤的问题转化为机器学习中的二分类问题。在进行误报过滤之前，系统使用开源 C++ 源代码漏洞静态扫描工具集对项目进行扫描，并将漏洞扫描的结果映射到 CWE 数据集来融合不同工具的报告，输出一个统一的漏洞报告。误报过滤分为训练和过滤两个阶段。在训练阶段，系统首先提取带有漏洞标签的数据集的抽象语法树、控制流图；接着使用词嵌入和图嵌入算法来获取源代码的特征向量代码表示；最后将特征向量作为机器学习模型的输入，根据模型的结构对模型进行训练，得到误报过滤模型。在过滤阶段，系统也需要先提取漏洞代码相对应的抽象语法树、控制流图；接着使用相同的嵌入算法获取源代码的代码表示；最后使用误报过滤模型对漏洞报告进行过滤。

该系统创新的将相似度检测和专家审核结合起来，结合人工和机器的力量来扩增误报漏洞数据集。首先漏洞专家对漏洞报告进行人工审核，标记出其中的误报漏洞；接着使用相似度检测算法，寻找和人工审核出来的误报漏洞相似的漏洞，并标记为误报，进一步扩展误报数据集；最后使用新的误报数据集来训练误报过滤器。

本文的主要工作如下：

首先，对 C++ 源代码漏洞静态检测、降低扫描器误报和相似度检测领域的研究进行综述。本文对 C++ 源代码漏洞静态检测、降低扫描器误报和相似度检

测领域的研究现状进行了较为深入的研究。受到现有工作的启发，将机器学习中的分类技术应用到误报过滤中，将误报过滤问题转化为二分类问题。

其次，完成对 C++ 源代码漏洞静态扫描系统进行需求分析。本文对漏洞静态扫描系统的业务需求和其中的难点进行了深入的分析，并针对实际业务需求设计了为了迭代反馈式降低误报的系统业务流程。

然后，完成 C++ 源代码漏洞静态扫描系统的概要设计。本系统分为 C++ 源代码漏洞静态扫描模块、C++ 源代码特征提取模块、漏洞扫描误报过滤模块、误报漏洞审核反馈模块。为对系统中的各个模块进行解耦，提升系统的可扩展性，本系统充分使用 Docker 容器技术、RabbitMQ 消息中间件技术对各模型进行模块化和异步化的设计。

最后，完成 C++ 源代码漏洞静态扫描系统的实现，并对系统进行了细致的功能测试和非功能测试。本文依照详细设计和用户实际需求实现了系统的功能。并且对系统提供的功能、系统的可靠性、提出算法的有效性进行了详细的测试。结果显示系统通过了所有的测试，并且提出的算法是有效的。

目前，本系统已经投入使用，对外提供源代码漏洞扫描服务。该系统通过对 C++ 源代码进行自动化漏洞静态扫描、误报过滤来减少漏洞报告和误报漏洞数量，提升开发者的用户体验和使用效率，结合漏洞代码相似度检测算法减少漏洞专家人工审核漏洞的数量，降低了漏洞专家的工作量，提升了漏洞审核的效率。该系统已经成功对数十个企业项目进行扫描，帮助公司向用户提交更高质量、更专业的漏洞报告，降低了人工成本，为企业创造了一定的价值。

6.2　进一步工作展望

C++ 源代码漏洞静态扫描系统目标是对 C++ 源代码项目进行漏洞扫描，该系统还有诸多可以优化之处：

漏洞扫描的结果存在限制。本系统的基础扫描部分是使用的开源工具集对 C++ 项目进行扫描，因此本系统的扫描能力依赖于开源工具的扫描能力。目前市面上的开源 C++ 源代码漏洞扫描工具数量有限，并且比如 clang 之类的工具需要可以编译的源码，存在一些限制。后续将会着手研发扫描工具。

人工参与度过高。目前本系统需要漏洞专家对漏洞报告进行人工审核，来标记误报漏洞，扩增误报数据集。人工审核存在成本高、审核结果不稳定等缺陷。后续可以考虑使用对抗神经网络技术，来提升误报过滤器的能力。

误报过滤模块缺乏可解释。本系统中的误报过滤模块使用了深度学习模型，该模型可解释性较差。在过滤掉相应的漏洞时，开发者和漏洞专家会比较关注

过滤漏洞的标准。而目前基于神经网络的深度学习技术不具备很强的可解释性。因此后续可以尝试挖掘误报分类的特征，提升可解释性。

综上，本文所设计的 C++ 源代码漏洞静态扫描系统还有一些可以继续提升、深化的点。通过自研漏洞审核工具，系统可以进一步提升漏洞覆盖率。使用对抗神经网络，可以降低人力成本以及人工带来的不确定性。通过深入分析漏洞代码特征，找出误报漏洞特征，可以提升系统的可解释性。

参考文献

[1] 王林章, 陈恺, 王戟, 软件安全漏洞检测专题前言, 软件学报 29 (5) (2018) 1177–1178.

[2] 邵思豪, 高庆, 马森, 段富尧, 马骁, 张世琨, 胡津华, 缓冲区溢出漏洞分析技术研究进展, Journal of Software 5 (2018) 1179–1198.

[3] J. Viega, J. Bloch, Y. Kohno, G. McGraw, Its4: A static vulnerability scanner for c and c++ code, in: Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00), IEEE, 2000, pp. 257–267.

[4] 李珍, 邹德清, 王泽丽, 金海, 面向源代码的软件漏洞静态检测综述, 网络与信息安全学报 5 (1) (2019) 1–14.

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

[6] W. Kirchmayr, M. Moser, L. Nocke, J. Pichler, R. Tober, Integration of static and dynamic code analysis for understanding legacy source code, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2016, pp. 543–552.

[7] 陈平, 韩浩, 沈晓斌, 殷新春, 茅兵, 谢立, 基于动静态程序分析的整形漏洞检测工具, 电子学报 38 (8) (2010) 1741–1747.

[8] Z. P. Reynolds, A. B. Jayanth, U. Koc, A. A. Porter, R. R. Raje, J. H. Hill, Identifying and documenting false positive patterns generated by static code analysis tools, in: 2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER&IP), IEEE, 2017, pp. 55–61.

[9] 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.

[10] 陈秋远, 李善平, 鄢萌, 夏鑫, 代码克隆检测研究进展, 软件学报 30 (4) (2019) 962.

[11] 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, 2014, pp. 762–774.

[12] E. Alikhashashneh, R. Raje, J. Hill, Using software engineering metrics to evaluate the quality of static code analysis tools, in: 2018 1st International Conference on Data Intelligence and Security (ICDIS), IEEE, 2018, pp. 65–72.

[13] 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, arXiv preprint arXiv:1801.01681.

[14] T. Kremenek, D. Engler, Z-ranking: Using statistical analysis to counter the impact of static analysis approximations, in: International Static Analysis Symposium, Springer, 2003, pp. 295–315.

[15] U. Yüksel, H. Sözer, Automated classification of static code analysis alerts: a case study, in: 2013 IEEE International Conference on Software Maintenance, IEEE, 2013, pp. 532–535.

[16] J. Yoon, M. Jin, Y. Jung, Reducing false alarms from an industrial-strength static analyzer by svm, in: 2014 21st Asia-Pacific Software Engineering Conference, Vol. 2, IEEE, 2014, pp. 3–6.

[17] G. M. Selim, K. C. Foo, Y. Zou, Enhancing source-based clone detection using intermediate representation, in: 2010 17th Working Conference on Reverse Engineering, IEEE, 2010, pp. 227–236.

[18] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, D. Song, Neural network-based graph embedding for cross-platform binary code similarity detection, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 363–376.

[19] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, C. V. Lopes, Sourcerercc: Scaling code clone detection to big-code, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 1157–1168.

[20] V. B. Livshits, M. S. Lam, Finding security vulnerabilities in java applications with static analysis., in: USENIX Security Symposium, Vol. 14, 2005, pp. 18–18.

[21] S. Lee, S. Hong, J. Yi, T. Kim, C.-J. Kim, S. Yoo, Classifying false positive static checker alarms in continuous integration using convolutional neural networks, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), IEEE, 2019, pp. 391–401.

[22] U. Koc, P. Saadatpanah, J. S. Foster, A. A. Porter, Learning a classifier for false positive error reports emitted by static code analysis tools, in: Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2017, pp. 35–42.

[23] M. Schnappinger, M. H. Osman, A. Pretschner, A. Fietzke, Learning a classifier for prediction of maintainability based on static analysis tools, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), IEEE, 2019, pp. 243–248.

[24] L. Ma, S. Yi, Q. Li, Efficient service handoff across edge servers via docker container migration, in: Proceedings of the Second ACM/IEEE Symposium on Edge Computing, 2017, pp. 1–13.

[25] I. Mastroeni, D. Zanardini, Abstract program slicing: An abstract interpretation-based approach to program slicing, ACM Transactions on Computational Logic (TOCL) 18 (1) (2017) 1–58.

[26] M. Weiser, Program slicing, IEEE Transactions on software engineering (4) (1984) 352–357.

[27] P. Ohmann, B. Liblit, Lightweight control-flow instrumentation and postmortem analysis in support of debugging, Automated Software Engineering 24 (4) (2017) 865–904.

[28] K. B. Gallagher, J. R. Lyle, Using program slicing in software maintenance, IEEE transactions on software engineering (8) (1991) 751–761.

[29] 孙继荣, 李志蜀, 王莉, 殷锋, 金虎, 程序切片技术在软件测试中的应用, 計算機應用研究 24 (5) (2007) 210–213.

[30] U. Alon, M. Zilberstein, O. Levy, E. Yahav, code2vec: Learning distributed representations of code, Proceedings of the ACM on Programming Languages 3 (POPL) (2019) 1–29.

[31] I. Neamtiu, J. S. Foster, M. Hicks, Understanding source code evolution using abstract syntax tree matching, in: Proceedings of the 2005 international workshop on Mining software repositories, 2005, pp. 1–5.

[32] E. Söderberg, T. Ekman, G. Hedin, E. Magnusson, Extensible intraprocedural flow analysis at the abstract syntax tree level, Science of Computer Programming 78 (10) (2013) 1809–1827.

[33] C. Lemieux, K. Sen, Fairfuzz: A targeted mutation strategy for increasing grey-box fuzz testing coverage, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 475–485.

[34] J. Chen, Q. Bao, Q. Zhang, J. Hu, P. K. Kudjo, A detection approach for buffer overflow vulnerability based on data control flow graph, in: Chinese Conference on Trusted Computing and Information Security, Springer, 2019, pp. 310–324.

[35] 张涛, 郝红卫, 基于安全扫描信息降低入侵检测系统的误报率, 微计算机信息 22 (8-3) (2006) 61–63.

[36] J. A. Suykens, J. Vandewalle, Least squares support vector machine classifiers, Neural processing letters 9 (3) (1999) 293–300.

[37] S. R. Safavian, D. Landgrebe, A survey of decision tree classifier methodology, IEEE transactions on systems, man, and cybernetics 21 (3) (1991) 660–674.

[38] D. Anguita, A. Ghio, N. Greco, L. Oneto, S. Ridella, Model selection for support vector machines: Advantages and disadvantages of the machine learning theory, in: The 2010 international joint conference on neural networks (IJCNN), IEEE, 2010, pp. 1–8.

[39] R. Girshick, Fast r-cnn, in: Proceedings of the IEEE international conference on computer vision, 2015, pp. 1440–1448.

[40] W. Ling, C. Dyer, A. W. Black, I. Trancoso, Two/too simple adaptations of word2vec for syntax problems, in: Proceedings of the 2015 Conference of the

North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2015, pp. 1299–1304.

[41] M. Striewe, M. Goedicke, A review of static analysis approaches for programming exercises, in: International Computer Assisted Assessment Conference, Springer, 2014, pp. 100–113.

[42] 甘水滔, 秦晓军, 陈左宁, 王林章, 一种基于特征矩阵的软件脆弱性代码克隆检测方法, 软件学报 26 (2) (2015) 348–363.

[43] R. Ploesch, H. Gruber, G. Pomberger, M. Saft, S. Schiffer, Tool support for expert-centred code assessments, in: 2008 1st International Conference on Software Testing, Verification, and Validation, IEEE, 2008, pp. 258–267.

[44] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, S. Jaiswal, graph2vec: Learning distributed representations of graphs, arXiv preprint arXiv:1707.05005.

[45] D. Dossot, RabbitMQ essentials, Packt Publishing Ltd, 2014.

[46] R. A. Martin, Common weakness enumeration, Mitre Corporation, 2007.

[47] P. E. Black, P. E. Black, Juliet 1.3 Test Suite: Changes From 1.2, US Department of Commerce, National Institute of Standards and Technology, 2018.

简历与科研成果

基本情况 史洋洋，男，汉族，1996 年 8 月出生，江苏省东台市人。

教育背景

2018.9 ~ 2020.6 南京大学软件学院 硕士

2014.9 ~ 2018.7 大连理工大学软件学院 本科

专利

1. 房春荣，**史洋洋**，“一种基于众包测试的 Android 和 iOS 移动应用静态安全测试系统的构建方法”，申请号：2018013101109150，已受理。
2. 房春荣，**史洋洋**，“一种基于微服务的安卓众包在线验证方法”，申请号：2019100901275450，已受理。，**史洋洋**，蒋燕，陈振宇，李玉莹，“一种功能粒度上基于语义信息的源代码相似度评估方法”，申请号：2019100901275550，已受理。
3. 房春荣，蒋燕，**史洋洋**，陈振宇，李玉莹，“一种结合源代码语义与语法特征的基于 CNN 的 bug 定位方法”，申请号：2019109519990，已受理。
4. 何铁科，严格，廉昊，秦泽民，**史洋洋**，陈振宇，“一种结合源代码语义与语法特征的基于 CNN 的 bug 定位方法”，申请号：2019109519990，已受理。

论文

1. Yunshan Tang , Nibin Zou, **Yangyang Shi**, et al. Research Progress of Security Protection for Dispatching Automation System. PIC 2018: 339-343.
2. Chunrong Fang , Zixi Liu, **Yangyang Shi**, et al. Proceedings of the Nineteenth International Symposium on Software Testing and Analysis. ISSTA 2020.

参与项目

1. 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.
2. 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.

致　　谢

论文接近尾声，我首先想感谢我的导师陈振宇老师和房春荣老师在我研究生期间对我的谆谆教导和耐心帮助。在毕业设计中，感谢两位导师从开题、中期到最后的耐心指导。在系统开发阶段，感谢两位导师帮助我不断的明确系统需求，并在开发技术选型上给予的帮助，不断地完善系统。恰逢疫情特殊时期，感谢两位导师对于论文进展不断的关心和督促，对我按时、保证质量的完成论文写作有很大的帮助。在此，我要向两位导师表示最崇高的敬意。

其次，我要感谢实验室的蒋燕、徐文远同学。感谢他们在整个项目的需求分析、实际开发过程中，提供的热心帮助和耐心指导，一起讨论解决问题，让我收获良多。同时，我还要感谢刘子夕、黎宇、张舒、门铎、韦志宾、徐悠然同学，感谢他们与我一同探讨问题、解决问题，同时也让我学到了很多他们的优点，也明白自身存在的不足。

感谢我的家人，在我遇到困难时，对我的鼓励和支持，同时感谢我的女朋友，与我一起分享喜悦忧愁，给生活增添光彩。

最后，我要感谢南京大学软件学院在我研究生期间对我的培养，让我从一个对计算机编程懵懂的人，成长为一个具备相对较为完整计算机技能的研究生毕业生，并且找到了心仪的工作。感谢软件学院全体教职工付出的辛勤劳动，为我们创造一个学风优良、积极向上、环境优美的环境。

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：史洋洋

日期：2020 年 5 月 27 日