南京大學

研 究 生 毕 业 论 文

（申请工学硕士学位）

论 文 题 目 持续集成下面向代码安全的提交优先级排序方法研究

作 者 姓 名 戴启铭

学科、专业名称 工学硕士（软件工程领域）

研 究 方 向 软件工程

指 导 教 师 张贺 教授

2021 年 **4** 月 **6** 日

学　　号：**MG1832001**

论文答辩日期：**xxxx** 年 **xx** 月 **xx** 日

指 导 教 师：　　（签字）

Research on the Code-Security Oriented Prioritization Ranking Method of Commit in Continuous Integration

by

Qiming Dai

Supervised by

Professor He Zhang

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
Master of Engineering
in
Software Engineering

Software Institute
Nanjing University

April 6, 2021

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：持续集成下面向代码安全的提交优先级排序方法研究

工学硕士（软件工程领域）专业 2018 级工学硕士生姓名：戴启铭

指导教师（姓名、职称）：张贺 教授

摘　要

持续集成是现代软件开发模式下的一项重要实践，开发人员需要频繁地提交并构建开发完成的代码，以实现价值的快速交付。但实际持续集成场景中，面对高频率的代码提交和有限的服务器资源数量，提交不得不等待资源的分配才能够执行构建任务。随着当下网络安全环境愈发严峻的，软件系统中的安全问题日益突出，各大软件企业逐渐考虑保障集成的代码的安全性。在持续集成中进行安全检测任务，无疑又会进一步加重提交等待服务器资源的现象。

为处理好这一问题，本研究首次站在代码安全的角度，提出了面向代码安全的提交优先级排序方法，试图在持续集成场景下提前安全漏洞的修复时间，缩短提交的整体构建时长。本研究基于对函数方法粒度的代码片段的安全性预测结果，对本次提交中的被预测为含有安全漏洞的方法个数进行统计，并依据此数值的大小，实现对提交的优先级排序。本文选取了 Juliet 测试套件和 OWASP Benchmark 两个标准数据集进行代码片段的安全性预测研究，从测试代码中提取出抽象语法树信息，并基于 BERT 模型对抽取的语法树内容进行学习训练，实现了函数方法粒度的安全漏洞预测分类器。从分类器的预测结果来看，本文所提出的针对函数方法粒度的安全漏洞预测模型在两份标准数据集中都取得了优异的预测性能，为面向代码安全的提交优先级排序方法的有效性奠定了良好的基础。

为验证本文所提出的面向代码安全的提交优先级排序方法的有效性，本文以一个开源项目的持续集成过程数据为案例，使用基于离散事件的软件过程仿真技术，对其实际持续集成过程进行了仿真模拟，并对比了提交构建效率在使用提交优先级排序方法前后的变化情况。结果表明，本文所提出的面向代码安全的提交优先级排序方法能够在不同程度上减少提交的整体构建时间，提升安全漏洞修复的反馈效率。

进一步地，本文通过控制仿真模型中的参数变化，对总计 684 种不同场景下的提交进行了仿真实验，探究了提交频率 f_c、提交漏洞率 P_{vc}、漏洞方法占比 P_{vm} 这三种外部因素对提交的实际排序效果的影响程度，为在不同情境下实际应用本文所提出的提交优先级排序方法提供了有效参考。

关键词： 持续集成；提交优先级排序；安全漏洞预测；软件过程仿真

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Research on the Code-Security Oriented Prioritization Ranking Method of Commit in Continuous Integration

SPECIALIZATION: Software Engineering

POSTGRADUATE: Qiming Dai

MENTOR: Professor He Zhang

Abstract

Continuous integration is an important practice under the modern software development. Developers need to frequently commit and build source code to achieve rapid delivery. However, in the face of high-frequency code commit and limited quantity of server resources in continuous integration scenarios, a large number of commits have to wait for resource allocation to execeute build task. In the current increasingly severe network environment, security issues in software systems have become sharply prominent, and software companies gradually consider to guarantee the security of the integrated code. However, the security scanning task in continuous integration will undoubtedly further aggravate the phenomenon of waiting for server resources.

In order to deal with this problem well, for the first time, this research proposes a code-security oriented prioritization ranking method, trying to advance the repair of security vulnerabilities in continuous integration scenarios and shorten the overall build time of each commit. This study is based on the security prediction results of the code fragments, and implements the prioritization ranking approach according to the magnitude of the number of methods predicted to be vulnerable in each commit. Two standard data sets of Juliet test suite and OWASP Benchmark are selected for realizing the security prediction of code fragments research. The abstract syntax tree information is extracted from the test code, and the security vulnerability prediction classifier is realized based on the BERT pre-trained model. From the prediction results, the prediction model proposed in this paper has achieved excellent prediction performance in the two standard data sets, which lays a good foundation for the effectiveness of the code-security oriented prioritization ranking method.

In order to verify the effectiveness of the code-security oriented prioritization ranking method proposed in this paper, continuous integration process data of an open source project and discrete event-based software process simulation technology are applied to simulate an actual continuous integration process. The changes in the efficiency of build each commit in continuous integration scenarios before and after using our prioritization ranking method are also compared. The results show that the code security-oriented prioritization ranking method proposed in this paper can reduce the overall build time of each commit and improve the feedback efficiency of security vulnerability repair in most cases.

Furthermore, this paper conducts simulation experiments under a total of 684 different scenarios by controlling the parameter changes in the simulation model. The impact of three external factors, the commit frequency f_c, the commit vulnerability rate P_{vc}, and the proportion of vulnerability methods P_{vm}, on the ranking effect is also explored, which provides a reference for the practical application of this prioritization ranking method proposed in different situations.

keywords: Continuous Integration, Commit Prioritization Ranking, Vulnerability Prediction, Software Process Simulation

目　　录

插图清单

附表清单

第一章　绪论

1.1　研究背景

从 21 世纪开始，面对复杂多变的市场需求和频繁变更的业务场景 [2]，传统的瀑布式软件开发方式难以为继。为提高软件的交付效率，走出传统开发方法的困境，相继出现了以 Scrum [3]、极限编程（eXtreme Programming）[4] 为代表的敏捷开发方法以及 DevOps 系列运动 [5]。

其中，持续集成（CI, Continuous Integration）作为一种典型的软件工程实践，是极限编程和 DevOps 中的一项重要实践。它旨在对开发的代码进行统一的管理、频繁地对代码进行集成、快速迭代出可用的软件版本、实现价值的快速交付。

持续集成强调每个开发者应当频繁的提交自己的工作，将自己开发完成的代码及时与主分支进行合并，每天每个开发者也应当完成至少一次或多次的代码集成。尽管持续集成并不能够避免出现问题，但对提交的代码而言，却可以尽早发现在项目集成过程中出现的任何问题，并针对这些问题完成及时修复，推进项目的整体进展。持续集成的整个过程应当以自动化的方式进行，但在实际实践时，也常常伴随着人工的手动触发。开发人员提交自己的代码进入集成平台进行代码的变更与合并，根据平台的流程设定，提交的代码需要进行必要的编译、静态检查、单元测试、集成测试和打包等操作，生成可运行的二进制包 [6]。在整个过程中，提交的代码如若在某一处检查处出现错误，那么本次集成就在此处停止，并将错误信息反馈给开发人员，开发人员根据反馈的错误信息进行必要的后续处理。待处理完成后，再一次重新进行集成。如若整个过程都没有发现错误，那么就认为本次的集成成功，可以完成本次代码的变更与合并。

目前，持续集成已广泛地应用于实际的大型软件开发中，为企业带来巨大的经济和规模效益。在工业界，Google、Facebook、Microsoft 等公司已应用持续集成方法，来匹配企业实际的软件开发速度和规模 [1]。HP 公司也曾宣称，

在应用持续集成等实践后，某团队节约开发成本高达 78%[①]。在开源领域，持续集成的发展也同样迅猛。Travis CI 作为与 Github 紧密结合的持续集成基础设施服务，已经为全球超过 900,000 个项目和 600,000 个用户提供持续集成服务。用户只需将 Travis CI [②] 服务器与 Github 代码仓库完成绑定后，每进行一次代码提交（commit），均会自动触发项目的构建，并记录完整的项目构建过程。Puppet 的 DevOps 年度调查报告 [7] 也明确指出，“在版本控制普遍存在的情况下，持续集成已成为 DevOps 中必不可少的一个环节”，在未来也将获得进一步的增长和发展。

然而，持续集成在实际执行过程中仍然存在的一些不可忽视的问题。Laukkanen 等人 [8] 对这一方面进行了深入研究，他们指出测试问题是持续集成中的最为重要的问题之一。尽管完整的持续集成流程中包含单元测试、集成测试等基础的功能性测试，这些功能性测试能够帮助开发者发现提交中不满足需求的功能缺陷，然而在软件安全问题日趋严重、软件功能日趋复杂的当下，光有功能性测试还远远不够。倘若交付的软件缺乏相应的安全测试，则这些软件中就极有可能存在大量的安全漏洞。安全漏洞是指系统中的弱点[③]，攻击者可利用此弱点侵入系统，从而获取系统的部分特权，能够实施侵害系统及其资产的各种不法行为。一旦这些的安全漏洞被黑客发现并加以利用，将会给企业带来难以估量的损失。

例如，据美国联邦调查局互联网犯罪投诉中心收集的数据显示[④]，2018 年与互联网相关的欺诈和漏洞利用带来的经济损失高达 27 亿美元，联邦调查局也预计这一数字在未来几年将会进一步扩大。同年，英国航空公司遭受到黑客攻击，造成约 38 万名消费者的个人敏感信息被泄露[⑤]。究其原因，是因为应用程序中存在 22 行不安全的代码，不安全的代码刚好被黑客发现并进行攻击。该事件不仅使得公司形象严重受损，同时也被开出了巨额罚单以示惩戒。

面对层出不穷的软件安全相关问题，各个国家、各个地区也积极推出各项法律法规来提高软件自身的准入条件和安全门槛，相继颁布了加强网络安全、软件安全的若干准则和规范。譬如，欧盟颁布的《通用数据保护条例》（GDPR）加强了对个人消费者敏感数据的保护，并将这一保护责任转嫁给了企业，这意味着企业所提供的应用或服务需要具备更强的安全性，足够抵

[①] http://continuousdelivery.com/evidence-case-studies/
[②] https://travis-ci.org/
[③] https://nvd.nist.gov/vuln
[④] https://www.ic3.gov/
[⑤] https://www.riskiq.com/blog/labs/magecart-british-airways-breach/

御外来的黑客攻击行为。

但在实际的软件开发过程中，如果将全部的安全检测行为都集中于软件开发完成之后，那么软件安全检测将会成为软件开发生命周期的一大瓶颈。一方面，项目代码自身错综复杂，对整个项目进行全面的安全分析本身就会耗费大量的时间和资源。由于在日常的开发过程中未进行安全保障，那么当安全检测完成后，也极有可能出现大量的安全漏洞。在这种情况下，对庞大代码中的漏洞进行定位、修复将耗费更多的人力，势必会拖慢项目的部署和发布周期。另一方面，现代软件开发已广泛应用敏捷方法、DevOps 等开发模式 [9]，将安全独立于开发过程之外，不符合当今主流的开发方式。随着近些年软件安全要求的提高，将安全元素融入 DevOps 早已成为软件工程领域中的热门话题 [10][11][12]，并逐渐吸引更多的研究者和从业者加入其中。2019 年 Puppet 发布的 DevOps 年度调查报告 [13] 通篇聚焦于 DevOps 软件开发过程中安全问题，以保障最终交付的软件的质量。由此可见，加强软件开发过程中的安全性检测，在现有的开发步骤中添加诸如安全代码扫描等安全实践方法，已成为现阶段各企业的普遍共识。其中，持续集成最为现在软件开发过程中必不可少的一环，因其每一次提交的代码量并不大，若在此时进行安全检测，则更益于进行漏洞的定位和修复。Bolduc 等人 [14] 将这一想法付诸实践，他们规定执行构建任务前的提交都需要经历一个快速安全检测的过程。同时，越来越多的企业也逐渐在持续集成过程中加大代码安全问题的关注程度。Gitlab[①]作为全球最为著名的企业级代码托管平台之一，在其自身的 CI 任务中，也将安全检测作为一项任务置于流水线中。

然而，考虑到实际服务器资源有限，面对愈发频繁的代码提交和构建，这些提交的代码将极有可能在等待队列中等待服务器资源可用。只有当服务器资源可用时，先进入队列的提交才会出列进行实际的构建。Liang[1] 等人在对 Google 内部持续集成的实际应用情况进行分析后表示，当仅有一份计算资源进行构建时，等待队列中的提交数量峰值能够达到 3000 个。这一情况，随着计算资源份数的增加有所缓解。当计算资源达到 2 份时，等待队列中的提交数量峰值约为 1600 个。但更多的计算资源也意味着更多的成本开销。据统计 [1]，每增加一份计算资源，将会提高 87% 的成本。因此，如何在有限的服务器资源下，有效地提高持续集成中代码的安全性，值得深入的研究和探讨。

为处理好上述问题，本文着眼于等待队列中的提交。按照原先的方式，当

[①]https://gitlab.com/

计算资源被占用时，进入持续集成平台提交会按照先进先出的原则在队列中挂起等待，这些等待中的提交可能具备不同的优先级、不同的安全状况。假设存在安全漏洞的提交均处在等待队列中的后半段，那也就意味着含有安全漏洞的提交将会越晚暴露，这无疑会拉长项目实际的开发周期。如果能够改变等待队列中的排序规则，将存在安全漏洞的提交提前，便能够尽早发现、修复这些提交中的安全漏洞，节约整体的构建时间。

因此，为充分利用有限的服务器资源，有效提升持续集成过程中的代码安全性，本文提出了一种面向代码安全的提交优先级排序方法。该方法能够在单个方法的粒度上，对提交的代码中的每一个方法的安全性做出预测，并基于预测的结果加权得到本次提交的安全状况，进而实现提交的优先级排序。同时，为探究本方法在实际持续集成场景下的有效性，本文基于离散事件过程仿真技术，选取 Travis CI 中项目的持续集成过程相关数据进行模拟仿真，以此展示本文提出的面向代码安全的提交优先级排序方法在持续集成场景下应用的实际效果。

1.2 本文工作与贡献

本文的主要工作分为基于 BERT 的安全漏洞预测与基于离散事件的持续集成过程仿真两个部分。

在持续集成场景下，每一次代码的提交会涉及到对部分代码文件的改动，基于 BERT 的安全漏洞预测将会针对这部分被改动的文件中的每一个方法的安全性做出预测。本文通过静态代码分析方法抽取提交中的源代码文件中的抽象语法树信息，利用深度学习方法挖掘其中的语义信息，将对源代码中的单个方法的安全性预测转化为二分类问题，并在公开的标准数据集 Juliet 与 OWASP Benchmark 上进行训练与验证。在此工作的基础上，本文构建了面向代码安全的提交优先级排序方法的排序原型，该方法依据基于 BERT 的安全漏洞预测结果，加权统计本次提交中安全漏洞的数量以确定本次提交的安全性排序。提交中含有的安全漏洞个数越多，则该提交将会被排到等待队列的靠前位置，这意味着含有更多的安全漏洞能够更早被发现，有利于在保障代码安全性的同时提升整体的构建效率。

由于难以在实际工业级项目的持续集成过程中验证所提出的排序方法的有效性，本文首次尝试基于离散事件过程仿真技术对实际项目的持续集成过程进

行模拟。使用离散事件过程仿真技术，无需重新进行项目的开发工作，而是利用先前项目开发过程中的真实过程数据来还原实际的项目开发过程，进而为研究者展开深入研究提供可能。因此，本文收集了利用 Travis CI 进行持续集成的开源项目的过程数据，并基于获取的数据进行持续集成这一软件过程的仿真模拟。同时，本文还详细对比并分析了在应用面向代码安全的提交优先级排序方法前后的性能变化，进一步验证了本文所提出的提交排序方法在持续集成场景下的有效性。

本文的主要贡献如下：

1. 本文指出了在持续集成场景中进行的安全检测的必要性，并提出了一种面向代码安全的提交排序方法，有效兼顾了持续集成过程中代码的安全性和集成效率问题。相较于传统“先进先出”的提交等待模式，面向代码安全的提交排序方法基于提交中代码的安全性，实现了对进入持续集成平台中的提交进行重排序。
2. 本文提出了一种基于 BERT 的安全漏洞预测模型。该模型基于从源代码中提取的抽象语法树信息，在方法级别的粒度，使用深度学习方法对代码的安全性做出预测。相较于现有的漏洞预测方法，本文提出的预测模型亦表现出不错的性能效果。
3. 本文首次使用软件过程仿真技术对实际的持续集成场景进行模拟还原，比较了使用排序方法前后的集成效果，验证了本文所提出提交排序方法的有效性。
4. 本文对影响排序方法效果的三个可变参数进行了多组实验验证和结果分析。分析指出，提交频率 f_c 对排序效果影响最大，提交漏洞率 P_{vc} 次之，漏洞方法占比 P_{vm} 最小。

1.3 文章结构

本文整体组织结构如下：

第一章明确了在持续集成过程中加强代码安全性检测的必要性与可行性，同时也指出了企业场景下，提交等待服务器资源的现象，明确了本文所提出面向代码安全的提交优先级排序方法的现实意义。

第二章详细介绍了本文所涉及到的三个领域的相关工作与研究进展，包括持续集成与提交排序、漏洞预测和软件过程仿真建模。

第三章介绍了本文所提出的面向代码安全的提交排序方法的整体思路，并从理论上验证了这一方法的有效性。

第四章详细介绍了基于 BERT 的安全漏洞预测模型的全过程，包括数据的预处理、模型训练过程、模型评估实验、结果有效性分析等具体内容。

第五章详细介绍了构建基于离散事件的动态仿真模型的全过程，包括持续集成静态过程分析、动态仿真模型构建与仿真模型校准等多个步骤。

第六章基于构建完成的仿真模型，完成了对面向代码安全的提交优先级排序方法有效性验证的多组实验，并针对实验结果与研究问题，进行了深入的分析与讨论。

第七章简要总结了本文的主要工作，并基于现有的工作进展对未来研究方向做出了展望。

第二章　相关工作

2.1　持续集成中的优先级排序

持续集成中，一次完整的构建过程通常会涉及基础的单元测试和集成测试等，相应的测试套件也会被直接部署到平台中。一个测试套件中通常含有多条测试用例。当提交进入平台时，按顺序执行编排完毕的测试套件即可。因此，在以往的研究中，多以对持续集成场景下的测试用例优先级排序为主。他们期望通过安排测试用例的先后顺序，来提高提交中代码的测试覆盖率，以及尽早发现代码中存在的功能缺陷。

Elbaum 等人 [15] 在持续集成下的回归测试场景中，将提交分为预提交和后提交两个阶段。在预提交阶段，开发人员需要选择本次提交中有哪些模块需要测试。选择完成后，他们提出的方法会使用一个时间窗口来回溯最近的每一个独立测试套件（test suit）的执行结果（成功或者失败）；在后提交阶段，基于上一阶段获取的执行结果，他们的方法会对开发人员指定的模块及其依赖模块安排最有可能失败的测试套件优先执行，以此达到尽早发现提交中存在的缺陷的目的。

Spieker 等人 [16] 将强化学习方法引入到测试用例优先级排序方法中，用于自适应测试套件的增删场景。因此，除了基于测试套件的历史的执行结果信息外，他们提出的方法中还将测试套件的驻留时间考虑在内，通过强化学习的奖励机制在不断执行的持续集成任务中进行学习，甄别出测试套件的贮存状态并优先选择更容易出错的测试套件执行。

Haghighatkhah 等人 [17] 探究了历史执行信息的数据量大小对回归测试中用例优先级排序的影响程度。他们指出相较于仅使用少量的历史数据的排序方式，基于大量的用例执行结果更能够切实提高测试用例优先级排序的准确性。但在缺少历史执行记录的测试初期，他们提出可以使用将测试用例多样性与历史执行信息相结合的方式来进一步提高排序的有效性，准确帮助开发人员发现回归测试中的功能缺陷。

Liang [1] 等人却提出了不同的看法。他们直接指出了传统的对测试用例进

行优先级排序的方式很难在持续集成场景中实际应用。在持续集成场景下，测试套件一般不会随提交的代码而动态调整，一次提交也会指定执行哪些测试套件，只有当开发的功能相对完整时，对测试套件进行编排才能更好地发挥效果。面对愈发频繁的提交，对提交的代码及测试用例进行大量的分析也不符合持续集成的内在要求。因此，Liang 等人摈弃了对测试用例进行优先级排序的方式，率先在这一场景中对提交本身进行优先级排序。他们同样基于的是持续集成平台中测试套件的历史执行情况及测试套件的多样性程度，能够实现对正等待队列中等待执行资源的所有提交进行持续性的优化。

准确来讲，本文的研究内容与上述研究并不相同。他们的工作大多是利用测试套件或测试用例的历史执行信息来帮助开发者尽早发现提交中的功能缺陷，而本文的工作则专注于提交中的安全漏洞，期望在持续集成中引入安全检测的同时，有效兼顾集成代码的安全性和集成效率。在代码开发阶段，对安全漏洞的检测一般是借助静态扫描工具，对代码自身的安全性进行静态扫描。由于每一次提交的代码不尽相同、安全检测任务中也不包含可以调整次序的用例，因此本文无法效仿先前的研究经验，使用历史的执行结果信息来作为排序依据，而是基于代码自身的安全状况作为排序标准。此外，研究者们可以从开源的数据集中获取到每一个测试用例的执行结果，利用这样的后验信息，可以对测试用例优先级排序方法进行有效验证。然而，在本文的场景中，开源数据集缺少关于提交代码的安全性的后验信息，若想在后期对数据集中的每一个提交进行安全扫描，得出代码的实际安全情况将会耗费大量的人力物力。因此，本文选择使用软件过程仿真技术来评估排序方法的有效性。

2.2 安全漏洞预测

目前，针对安全漏洞的预测工作大致可以分为两类：基于软件度量的安全漏洞预测模型和基于文本挖掘的安全漏洞预测模型。

2.2.1 基于软件度量的安全漏洞预测模型

基于软件度量的安全漏洞模型通过分析软件的度量信息，如代码行数、方法个数等静态代码指标特征，对软件中是否含有安全漏洞进行判断。Zimmermann 等人 [18] 对 Windows Vista 项目中的安全漏洞进行分析，他们利用 Microsoft 内部的 CODEMINE[19] 工具进行特征提取。该工具能够提供多种

软件度量，包括代码变化相关指标、代码复杂度等代码相关的度量指标，甚至还可以获取软件开发过程中开发团队、开发过程度量的相关信息指标。在获取到相关指标数据后，Zimmermann 等人接着将这些获取的指标作为随机森林分类器 [20] 的特征输入进行训练，并对指定的文件是否存在安全漏洞做出判断。测试结果表明，在应用全部的度量特征后，安全漏洞预测的平均准确率（precision）约为 0.45，查全率（recall）为 0.2。

Shin[21] 等人对 Mozilla FireFox 和 Linux Kernal 两个开源项目分别进行了分析。他们针对每个项目在几年内的几次发布的版本，从这些版本的 bug 报告中搜索安全漏洞相关的内容信息及其对应的修复文件，这些被修复的文件被认为是存在安全漏洞的。接着使用代码复杂度指标、代码变化指标以及开发人员活动信息等共计 28 个度量指标对项目中的版本文件进行特征提取，依次使用线性分析 [22] 、贝叶斯网络模型 [23] 、随机森林分类方法进行训练和预测。Shin 等人并未直接使用准确率和召回率作为模型评价指标，而是声称他们提出的方法能够为 Mozilla Firefox 节省 71% 的代码扫描量，为 Linux Kernal 节省 28% 的代码扫描量。

Chowdhury[24] 等人同样基于软件的复杂度、内聚性和耦合性下的软件质量指标，探究了这些指标与软件安全漏洞之间的相关性。他们对四年来开发的 52 个 Mozilla Firefox 版本进行了大规模的实证研究，整理了这些 Firefox 版本中的漏洞日志、版本代码等相关信息，同时提出了一种基于复杂度、内聚性和耦合性的安全漏洞预测框架。在文件的粒度上，他们分别使用了决策树、随机森林、朴素贝叶斯、线性回归等常用机器学习方法对文件中是否含有安全漏洞进行了二分类预测。结果表明，他们的提出的预测模型的正确率约为 71%，但查全率则根据使用的机器学习算法的不同而差异显著。使用决策树算法的预测模型的召回率指标达到了 74%，而使用朴素贝叶斯的预测模型的召回率指标则为 29%。

不难发现，单纯基于软件度量进行漏洞预测的分析方法，其性能指标其实并不理想。其原因可能归咎于不同公司可能具备不同的开发规范或开发方式，开发人员的开发习惯也不尽相同，只是简单利用一些度量指标很难准确判断代码中是否真正还有安全漏洞。因此，近年来，针对安全漏洞预测进行的相关研究多需要深入代码语义层面，以期达到更好的预测效果。

2.2.2 基于文本挖掘的安全漏洞预测模型

为从代码中获取更多的特征信息，一般而言，基于文本挖掘的安全漏洞预测模型需要对源代码进行更多的额外处理，最终以向量的形式表征代码内容。Walden 等人 [25] 对 3 个 PHP 开源项目中的代码使用词袋（bag of word）技术 [26] 进行序列化来提取代码的词频特征，并将提取到的词频特征作为随机森林分类算法的输入进行训练。同时，他们也应用了基于软件度量的安全漏洞预测方法进行结果的比较，结果表明，相较于软件度量，使用文本挖掘的预测方法具备更高的准确性和查全率。然而，词袋技术将代码中的每个单词单独来看，忽略了代码的语法、语义信息，单纯通过统计单词出现频率来构建代码的词典。当代码文件过大时，构建出来的词典将会十分庞大，不利于分类模型的训练。并且，将代码文件中的每个单词割裂开来，会错失更丰富的代码特征，例如代码的结构特征及上下文语义信息。

为更多地获取代码信息，Yamaguchi 等人 [27] 融合了经典的静态程序分析的相关内容，使用了抽象语法树（AST）、数据流图（DFG）、程序依赖图（PDG）等结构性的代码静态表征信息，并将些结构化的代码信息作为特征输入机器学习算法进行分类。他们的结果表示，结构性的信息能够更好地反映程序内容，基于这些静态代码表征方式进行安全漏洞检查也具备更好的预测效果。尽管代码的静态表征方式能够反映代码的结构特征，但仍然存在一些不足，比如它们无法捕捉代码的语义信息、无法理解代码的上下文含义。

在以往安全漏洞预测模型的相关工作中，提取多种维度的度量特征并应用机器学习技术加以分类已成为主流的研究方法。但随着近年来，深度学习在自然语言处理、机器翻译等领域成就突出，在安全漏洞识别领域，已经有越来越多的研究者将目光转向深度学习技术。相较于传统的机器学习方法，深度学习能够学习抽象层次更高、更复杂的内容 [28]，也能够适用更广泛、更灵活的场景。Hindle 等人 [29] 的研究也表明，讲程序源代码视为自然语言进行处理同样具备良好的效果。

Li 等人 [30] 基于深度学习方法进行安全漏洞的检测。他们使用 word2vec 工具进行代码的向量表征，提取代码的语义信息。Word2vec 工具[①] 基于分布表达的思想，能够将特定的文本表达转化成定长的向量，是文本挖掘领域最为常用的将文本转化为向量的方法之一 [31]。但由于他们将研究范围限定在与库/API 调用有关的安全漏洞上，只针对了 CWE119 和 CWE339 两种具体的漏洞

[①] http://radimrehurek.com/gensim/models/word2vec.html

类型。因此，他们并未将全部代码向量化，而是首先从代码中提取库/API 的调用关系，并用程序切片 [32] 表示这种调用关系，这里的程序切片是指与调用特定库/API 方法的相关代码片段。接着使用 word2vec 工具将提取出来的程序切片用向量的形式加以表示，最后将得到的向量作为 BiLSTM 模型（Bidirectional Long Short-Term Memory）的输入进行训练。BLSTM 模型是一种能够处理上下文关系的神经网络模型。结果表明，相较于现有的其他方法，Li 等人提出的预测模型具备更好的性能，且能够实现细粒度的安全漏洞代码的精确定位。

Lin 等人 [33] 也同样使用了 BLSTM 模型进行训练，但相较于 Li 等人的研究，他们的识别粒度并不一致。Li 等人 [30] 的研究中针对的粒度是他们所提出的代码袋（code gadget），代码袋是由一系列代码片段组成。而 Lin 等人所提出的预测模型是针对程序方法级别的漏洞识别。他们选择使用抽象语法树结构化表征代码的静态信息，接着使用连续词袋方法进行词嵌入编码，最后输入 BiLSTM 模型进行训练和学习，获取对安全漏洞的表示方式。同时，他们还从 6 个开源项目中进行了手动标记，总计得到 457 个漏洞函数以及 30000 多个非漏洞函数。实验结果表明，他们的模型具备较好的迁移学习能力，即使是在不同的项目中，这一模型也具备较好的性能和效果。

2.3 软件过程仿真

软件过程仿真模型关注于特定的软件开发和维护过程，能够动态处理软件过程中行为的随机性，从而实现对现实或计划中的行为进行抽象、建模与实验，常用于流程改进、项目管理等多个方面。Kellner 等人 [34] 就软件过程仿真中的三大基本问题进行了回答，总结了为什么需要做软件过程仿真、软件过程仿真可以模拟哪些变量以及如何进行软件过程仿真。他们指出，当操作真实项目进行实验的成本或风险过高时，使用仿真建模技术会是一个很好的选择。在软件过程领域，应用软件过程仿真技术能够在战略和战术层面帮助企业做出合理决策，减小企业所承担的成本风险。同时，他们并将使用软件过程仿真技术的目的分为六个类别，即策略管理（Strategy Management）、企业规划（Planing）、运营管理（Control and Operational Management）、流程改进与技术革新（Process Improvement and Technology Adoption）、问题理解（Understanding）、培训学习（Traning and Learning）。进一步地，他们从建模范围、结果指标、流程抽象和输入参数这四个方面对过程仿真建模的具体内

容进行了更为深入的介绍，阐述了进行软件过程建模仿真的具体步骤，为研究者和从业者实际应用软件过程仿真建模技术提供必要的指导。

Zhang 等人 [35] [36] 围绕软件过程仿真这一主题进行多次系统化文献综述。他们肯定了软件过程仿真建模在实际应用中的巨大价值，并对使用软件过程仿真模型的原因和分类方案进行了改进和更新。在对 200 多篇学术文献进行分析后，他们认为 Kellner 等人在对使用对软件过程仿真技术的原因分类上存在交叉，并将原来总结出的的六种类别拓展为三个层级（认知层级、战术层级和战略层级）共计十种分类。同时，他们还指出了基于系统动力学（SD, System Dynamics）和基于离散事件（DES, Discrete-Event Simulation）的仿真是软件过程仿真建模领域最为常用的两种技术，对软件过程的改进则是该领域种最被频繁研究的主题。在仿真模型最终的产出结果上，大多数研究也将时间（time）作为最终的输出，以此衡量软件过程改进的实际效果。

此外，Zhang 等人 [37] 还汇报了软件过程仿真技术在不同工业场景下的应用实例。他们通过对公开出版的文献进行检索，确定了 32 个工业案例作为影响证据进行研究，并对其中被广泛应用的系统动力学仿真和基于离散事件的过程仿真进行了深入探讨，进一步论证了软件过程仿真在工业实践上的巨大影响。同时，在他们的工作中，Zhang 等人还对 Dan Houston 博士进行了访谈，Dan Houston 博士分享了他在 Honeywell 与 Aerospace 公司中实际应用软件过程仿真技术的经验，突出了使用软件过程仿真技术带来的巨大价值和效果。

Garousi 等人 [38] 为探究在给定项目中测试自动化的影响，通过在一家加拿大的软件公司中进行软件过程仿真的方式，成功帮助企业评估自身进行软件测试的性能，并辅助企业进行测试自动化过程中的决策。他们对软件测试过程中的每一个步骤进行了拆解和抽象，将测试流程分为用例设计、构建脚本、测试执行、测试评估、测试维护和测试代码重构六个部分，并基于系统动力学模型对这些部分进行仿真，构建出表征整体测试流程的仿真模型。依据软件公司的实际数据，Garousi 等人在他们构建的模型中模拟了 7 种不同的测试场景，分析了如何合理配置手动测试人员与自动测试人员之间的比重来达到测试目标，并评估了这一做法带来的实际损耗。在一定程度上，为企业的实际决策提供了理论依据。

2.4 本章小结

本章首先介绍了持续集成场景下的优先级排序问题，回顾了以往针对测试用例优先级排序的研究内容，比较了本文的研究主题与先前研究之间的异同。受限于有限的计算资源和高频率迭代开发的要求，在实际工业场景中，提交的代码在等待队列中等待计算资源已成为普遍现象。随着安全检测任务被加入到持续集成环节中，提交的排序等待现象也将愈发明显，如何在保障代码安全性的同时提高代码的集成效率，本文试图就这一问题给出解决方案。接着介绍了目前主流的两种安全漏洞预测研究，包括基于软件度量的安全漏洞预测方法和基于文本挖掘的安全漏洞预测方法。相较于软件度量，基于文本挖掘的安全漏洞预测方法通过引入深度学习技术，深入代码内容和信息，取得的实际效果也日益突出，并逐渐成为该领域的研究热点。最后本章还介绍了软件过程仿真建模方法的发展及其应用情况，证明了软件过程仿真方法的在评估软件过程和指导实际企业生产方面的巨大价值。

第三章　面向代码安全的提交优先级排序方法

3.1　研究目的

为提高集成的代码的安全性，对提交的代码进行必要的安全检测已成为各大企业一种普遍共识。然而，受限于有限的服务器资源，当项目的提交达到一定频率时，将会不可避免地出现等待服务器资源的场景。Liang 等人 [1] 展示了 Google 内部的代码提交情况，如图 3-1 所示。

当只有一份服务器资源时，队列长度的峰值达到了 3000，这些提交按照“先进先出”的规则在队列中等待服务器资源。倘若在提交等待时，对这些提交加以处理，优先让含有安全问题的提交出列，那么这些安全问题将会在接下来的安全检测任务中被提前发现。相较于先前在队列中继续等待，排序方法能够通过人工干预的方式，提前完成对这部分提交中的安全漏洞的修复，缩短所有提交的平均工作时间。

以图 3-2 为例，阐述一下本文所提出的面向代码安全性的提交优先级排序方法的基本思想。假设等待队列中有 10 个提交（C1 至 C10），其中有 2 个提交（C4、C10）的代码中存在安全漏洞。假设每个提交的安全检测时长分别为 X_i $(X_i > 0)$，对于无漏洞的提交，执行安全检测任务后，结束整个工作流；对于有漏洞的提交，需要花费额外的时间进行漏洞处的修复。假设修复本次提交中所有漏洞所需时长为 Y_i $(Y_i > 0)$，本次提交在完成所有的漏洞修复工作后，也同样会结束工作流。

具体地，C1 获取到服务器资源执行安全检测任务时，余下的 9 个提交在队列中继续等待。直到 C4 获取到服务器资源执行安全检测任务后，会发现 C4 中的代码中存在安全漏洞，于是需要额外花费时间 Y_4 对这些漏洞进行修复。在漏洞修复的过程中，队列中的其他提交继续依次获取服务器资源，执行安全检查任务，直至队列中的所有提交结束工作流。因此，在图 3-2 所示的样例中，按提交的到达顺序，即“原先执行顺序”出列的所有提交的工作平均流时长为：

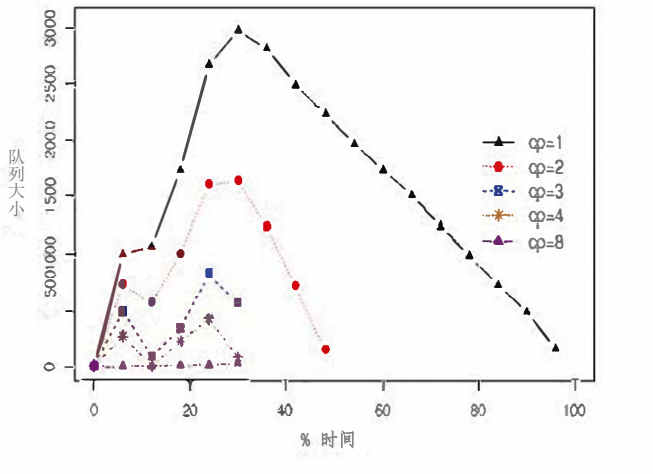


图 3-1: Google 内部的提交等待情况 [1]

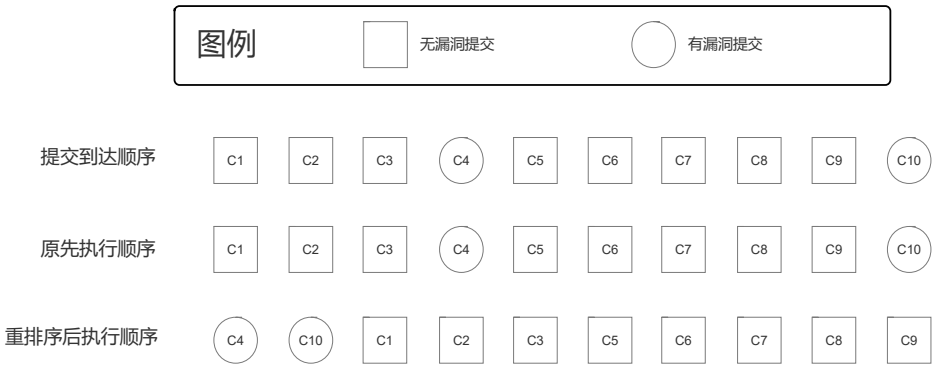


图 3-2: 提交排序示例

$$T_{orig} = \frac{\sum_{i=1}^{10}\sum_{i=1}^{i} X_i + max\left(0, Y_4 - \sum_{i=5}^{10} X_i\right) + Y_{10}}{10} \tag{3-1}$$

若对等待队列中的提交进行排序，将存在安全漏洞的提交前置，对这部分漏洞代码的修复工作也将提前，那么可以得到按照重排序后顺序出列的所有提交的平均工作流时长为：

$$T_{prior} = \frac{\sum_{i=1}^{10}\sum_{i=1}^{i} X_i + max\left(0, Y_4 - \sum_{i=2}^{10} X_i\right) + max\left(0, Y_{10} - \sum_{i=3}^{10} X_i\right)}{10} \tag{3-2}$$

比较上述两种方法的平均工作时长，不难发现，相较于原先“先进先出”

的出队方式，在对等待队列中的提交进行排序后，提交的平均工作流时长得到减少。为此，本文提出了面向代码安全提交排序方法，力图在持续集成场景下减少提交的平均工作流时长，提高了漏洞修复反馈效率，在保障代码安全性的同时兼顾了代码的集成效率。

3.2 研究框架

本文提出的面向代码安全的提交优先级排序方法的主要内容可以分为两个部分，第一部分为基于 BERT 的安全漏洞预测，第二部分为持续集成过程仿真，具体内容如图 3-3 所示。

基于 BERT 的安全漏洞预测的工作主要包括数据处理、数据解析和模型训练、性能评估四个阶段，具体细节将在第四章进行阐述。该预测模型能够对方法级别的源代码进行分析，并对该段代码是否含有安全漏洞做出判断。本文并未选取文件粒度作为模型预测的粒度，这是因为持续集成场景下，代码提交将会十分频繁，一般而言，其中包含的文件数量差异不会过于悬殊。若只是预测文件的安全性，统计漏洞文件的个数，那么提交之间的区分度将不会很明显，这会对排序效果造成一定的影响。本文借鉴了 Lin[33] 等人的研究思路，同样认为在方法粒度上进行安全漏洞的识别是一个较好的选择。一次代码提交中一般会包含多种方法实现，尽管方法体的长度可能长短不一，但这取决于开发人员的开发习惯以及公司的编码规范。若严格遵循编码规范，那么这一数值将不会相差太大，但不同的提交却极有可能包含不同的方法数量，这增加了提交之间的区分度，有利于增强提交的排序效果。

同时，本文完成了对分类器的原型工具封装。这样，该排序原型工具就能够解析本次提交中的所有代码内容，在方法粒度上，就代码的安全性做出预测，并统计本次提交中所有方法的预测结果。统计结果的数值大小反映的是本次提交中可能含有的安全漏洞的方法个数，排序原型就依据这一数值的高低，实现对提交的优先级进行排序。

由于本文的选取了两个不同的标准数据集，并以在这两个数据集上的平均测试效果作为分类器的性能参数。故此，我们认为本文训练所得得预测模型具备一定的泛化能力。在对提交中的具体方法进行预测并统计结果时，也应当服从这一性能参数。在本文的工作中，便可以认为这一参数能够反映本文所提出的优先级排序方法的实际性能指标。在进行仿真实验时，本文也就将其用于仿

真模拟的结果验证。

为探究本文所提出的排序方法在持续集成场景中的实际使用效果，我们使用软件过程仿真技术对持续集成的过程进行了仿真模拟。分类模型的性能评估数据可以通过在现有的数据集进行测试得到，但对于排序模型来说，目前尚缺乏对每一次提交的代码的安全性信息，即使在开源项目中对一段时间内的提交进行安全检测试图构建这样一份数据集，将会耗费大量的人力成本，更为重要的是，很难在开源项目中对这些存在安全漏洞的提交进行有效跟踪。这意味着，当提交中的代码被检测出含有安全漏洞时，我们无法获悉修复本次提交中的漏洞所需的实际耗时，这会极大地限制对排序效果的定量评估。

在以往关于测试用例优先级排序的研究中 [15][16][17]，他们基于测试用例的历史执行结果，将可能不通过的测试用例的位置前移，以期尽早发现提交中的功能缺陷，但却忽略了修复这些功能缺陷的实际耗时情况。因此，在定量评估他们的排序方法的实际性能时，流程并未形成闭环，排序方法的实际效果也相应地会存在一定的片面性。此外，受后验结果影响，他们的工作也无法定量分析参数变化对排序效果的影响。例如，无法探究测试用例通过率、测试用例所需耗时等数值的大小变化对排序方法造成的实际影响。

在本文的工作中，我们可以基于软件过程仿真建模技术探究提交频率 f_c，提交漏洞率 P_{vc} 和漏洞方法占比 P_{vm} 对排序实际效果的影响。在构建完成的模型中，通过改变这三种参数的数值大小，可以实现对不同场景的仿真模拟。该部分的工作主要分为四个阶段，包括持续集成过程分析、静态过程建模、构建仿真动态模型及方案验证，具体细节将在第五章进行阐述。本文基于对 Travis CI 的实际持续集成过程进行的分析、抽象及静态建模，构建了相应的仿真动态模型。仿真动态模型是对现实持续集成流程的抽象实现，具备这一过程的内在特征，能够反映持续集成的实际运行过程。在这一模型上进行排序方法的效果验证及分析，亦能够有效对比使用排序方法前后所带来的效率上的变化。

3.3 排序方案

算法3.1展示了本文提出的面向代码安全的提交优先级排序方法的几个重要步骤。提交等待队列 *commitQ* 及服务器资源 *resources* 是本算法的两个重要输入。服务器资源是执行构建任务时所需要占用的资源，提交等待队列是等待服务器资源的提交暂存地点。当进入流水线平台中的提交执行构建任

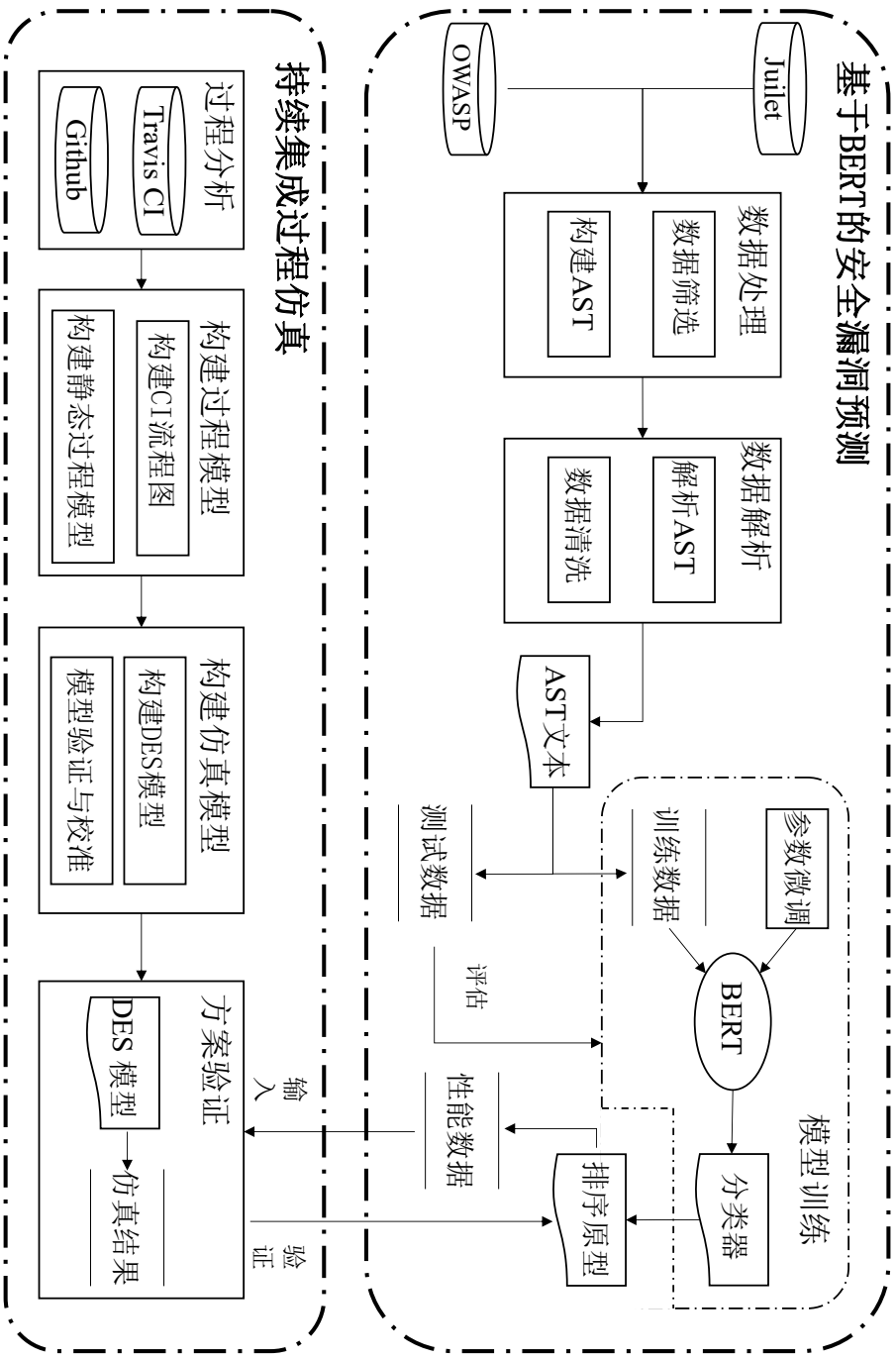


图 3-3: 面向代码安全的提交优先级排序方法整体研究框架

务时，服务器资源将被占用。当服务器资源都被占用时，若提交仍然不断进入持续集成平台中，那么这些提交将会进入等待队列中等待资源。提交到达 *onCommitArrival*、提交构建结束 *onCommitBuildEnding* 是面向代码安全的提交优先级排序方法的两个重要阶段性步骤。

onCommitArrival 是指提交的起始阶段，此时新提交 *commit* 进入到持续集成平台中，算法会对本次提交中代码的安全性能做出预测。本次提交中所有的代码文件将会被遍历，并从这些代码文件中提取出所有的方法片段，执行基于 BERT 的安全漏洞预测模型对每一个方法粒度的代码片段的安全性做出预测并记录预测结果。待所有的代码片段的预测工作结束后，排序算法会统计本次提交中被预测为含有安全漏洞的方法个数，并根据这一数值的大小为等待队列 *(*commitQ) 中的提交进行排序，它反映的是本次提交更可能含有更多的安全漏洞，优先对这部分提交执行构建任务，将会尽早识别出提交中实际安全性情况并进行修复，提高了漏洞修复的反馈效率。排序后的 *commitQ* 是按照预测提交所含漏洞方法个数从大到小排序的有序队列，这样，当服务器资源空闲时，可以直接从有序队列中弹出提交，避免了服务器资源空闲等待 *commitQ* 进行优先级排序的情况，提高了服务器资源的利用效率。

onCommitBuildEnding 是指提交完成构建任务后的阶段。完成构建任务就意味着本次提交集成成功，通过了持续集成平台中的所有既定任务。此时，提交便会释放所占用的服务器资源供等待队列中的提交使用。如果此时提交队列 *commitQ* 非空，仍然有提交正在等待服务器资源，那么在确认服务器资源可用后，就会弹出有序队列 *commitQ* 中的部分提交，并为这些提交分配相应的服务器资源，用于执行持续集成中的所有任务。如此循环往复，直至 *commitQ* 中再无等待服务器资源的提交。

需要说明的是，本文中能够进行优先级排序的提交是指一系列独立的提交，这些提交不会存在相互的依赖关系也不需要按照特定的顺序执行构建任务，因此可以打乱这些提交的构建顺序，而不会造成任何不利影响。Liang 等人 [1] 在进行提交优先级排序时也曾考虑到这一问题，从他们的实践经验来看，在他们所研究的工业数据集中并未出现提交依赖的情况。但如果考虑到提交执行顺序依赖的相关场景，将会使得提交优先级排序算法变得更加复杂，也是值得进一步研究的方向之一。

理论上来讲，如果队列 *commitQ* 中一直存在正在等待服务器资源的提交，那么就有可能存在部分提交一直无法被分配到服务器资源的情况。如果先前进入等待队列中的提交$commit_1$ 被预测为不存在任何含有漏洞的方法，那么，当后续被预测为存在漏洞方法的提交$commit_2$ 将会被提前执行构建任务，即使$commit_1$ 是先进入等待队列中的，也仍然需要在 *commitQ* 中等待。倘若之后的提交均被预测为含有漏洞方法，那么$commit_1$ 将会一直无法得到出列获取服

务器资源的机会，造成提交“饿死”的现象。但事实上，随着每一次提交的出列并执行构建任务，等待队列中的提交数量也会相应的减少，即使在这个过程中，又有新的提交进入到等待队列中，但随着时间的推移，这些提交也终将会被分配到相应的服务器资源。倘若在这个过程中，需要对等待时长做出人为干预，那么可以设定等待的阈值，从理论上消除这一现象发生的可能。但至于等待阈值的设定需要考虑哪些因素、等待阈值的设定又会给排序方法造成何种影响等问题也都是未来的重点研究方向之一。

算法 3.1 面向代码安全的提交优先级排序算法

```
输入：commitQ; resources
步骤 onCommitArrival(commit):
  for 提交中的所有代码文件 do
    提取方法片段
    预测方法安全性
    统计预测结果信息
  end for
  更新提交属性信息
  插入 commitQ
  根据预测漏洞方法个数大小排序
步骤 onCommitBuildEnding():
  释放服务器资源
  if commitQ 非空 then
    if 服务器资源可用 then
      commitQ 中的提交出列
      为出列的提交分配服务器资源
    end if
  end if
```

3.4 本章小结

本章进一步突出了本研究的必要性，并基于一个例子，从理论上证明了本文所提出的面向代码安全的提交优先级排序的有效性。同时，本节还简述了本文的基本研究思路和工作内容，包括基于 BERT 的安全漏洞预测和持续集成过程仿真两大部分，并对各部分工作的方法选用作了详细说明。最后介绍了本文提出的优先级排序算法的主要思路和一些假设条件。

第四章　基于 BERT 的安全漏洞预测

4.1　方法概述

图 3-3 展示了基于 BERT 的安全漏洞预测部分工作的整体方法，包括数据处理、数据解析、模型训练等多个阶段。图 4-1对数据处理及数据解析的过程做了进一步展示说明。本文选取了 Juliet 测试套件以及 OWASP Benchmark 这两份标准数据集作为原始数据，在分析其测试代码特征后。首先我们对其进行了筛选，保留了符合标准的部分 CWE 漏洞类型的测试代码展开实验研究。接着我们提取了保留的代码的抽象语法树，并在方法的粒度上进行解析，得到基本的抽象语法树文本语料，同时也解析出该方法所对应的标签内容。然后，我们对解析得到的方法抽象语法树进行了标准化处理，防止携带标签信息的方法名、方法体对预测模型的强干扰性，保障了预测模型的客观性和有效性。最后基于标准化的抽象语法树文本语料及标签内容，我们按照八二的比例，将其拆分为训练集和测试集，并使用 BERT 模型完成了相应的训练和测试。测试完成后，本文根据既定的排序规则将预测模型封装为排序原型工具，并将得到的性能参数作为持续集成过程仿真部分工作的输入，展开排序方法有效性的验证。

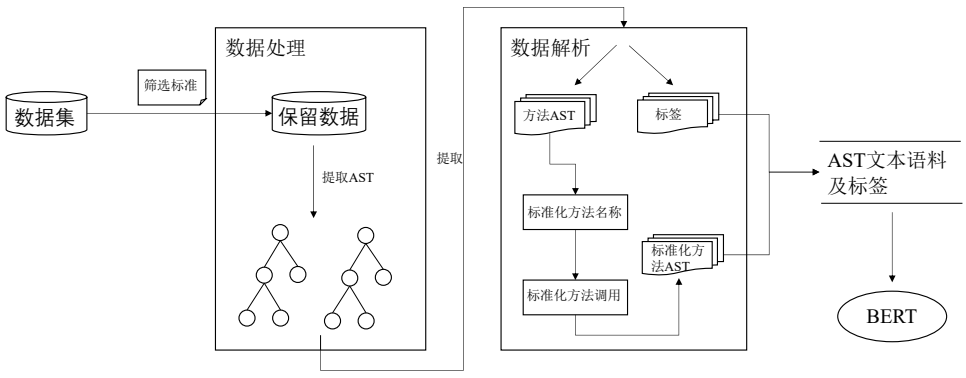


图 4-1: 基于 BERT 的安全漏洞预测模型数据处理过程示意图

4.2 数据获取

通用缺陷列表（CWE, Common Weakness Enumeration）是一个对软件脆弱性和易受攻击性的分类系统，它基于开发者视角，重点关注了开发过程的安全性，汇集了软件开发社区中较为常见的的漏洞类型，如缓冲溢出、SQL 注入、跨站脚本攻击等，是由美国国土安全部国家计算机安全部门资助的软件安全战略性项目，也是目前较为权威的源代码安全缺陷研究项目。因其对源代码漏洞描述的准确性和权威性，其成果已经被越来越多的企业和研究者所认可，并逐渐成为衡量产品安全性的重要标准。

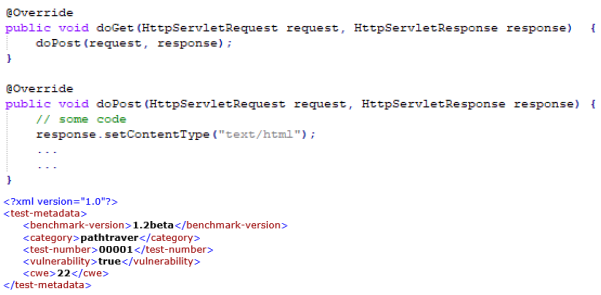
开放式 Web 应用程序安全项目（OWASP，Open Web Application Security Project）是一个开源的、非盈利的全球性安全组织，它更偏向攻击者的视角，致力于应用软件的安全研究，为企业和组织能够对应用安全风险做出更清晰的决策。OWASP 被视为 web 应用安全领域的权威参考，其发布的 OWASP TOP 10 早已成为 IBM APPSCAN、HP WEBINSPECT 等扫描器漏洞参考的主要标准，其开源的 OWASP Benchmark[①]项目，也被广泛应用于安全扫描工具的性能测试。

鉴于 CWE 漏洞分类信息与 OWASP Benchmark 的广泛认可和应用，本文使用了由美国国家安全软件中心开发的针对 Java 语言的 Juilet 测试套件[②]（v1.3 版本）与的 OWASP Benchmark 这两份数据集 (v1.2 版本)。

Juilet 测试套件由 112 种不同类型的 CWE 安全漏洞组成，总计包含 28881 个测试用例，被广泛应用于代码静态分析工具性能评估、软件产品安全质量检测及安全漏洞相关的学术研究中 [39][40][41]。Juliet 测试套件中的测试用例代码结构相对简单，每一种漏洞类型均会有一系列测试用例，每一个用例文件中又会存在多个源代码文件。在这些源代码文件中同样会有多个人为构建的方法，如果这个方法含有安全漏洞，方法名会显示使用类似“bad”的单词进行提示。相似的，如果这个方法不存在安全漏洞，方法名也会使用“good”类似的单词进行标识。

OWASP Benchmark（v1.2 版本）包括了 11 种不同类型的 CWE 安全漏洞，总计 2740 个测试用例。与 Juliet 测试套件中的数据结构不同的是，OWASP Benchmark 中的代码并未直接展示函数方法是否含有安全漏洞。而是通过附上 XML 文件的方式给出对应文件的安全性结果。图 4-2展示了 OWASP Benchmark

[①]https://owasp.org/www-project-benchmark
[②]https://samate.nist.gov/SARD/testsuite.php



```
@Override
public void doGet(HttpServletRequest request, HttpServletResponse response)  {
    doPost(request, response);
}

@Override
public void doPost(HttpServletRequest request, HttpServletResponse response) {
    // some code
    response.setContentType("text/html");
    ...
    ...
}
```

```
<?xml version="1.0"?>
<test-metadata>
    <benchmark-version>1.2beta</benchmark-version>
    <category>pathtraver</category>
    <test-number>00001</test-number>
    <vulnerability>true</vulnerability>
    <cwe>22</cwe>
</test-metadata>
```

图 4-2: OWASP Benchmark 测试用例片段及标签信息展示

的部分测试用例片段及对应的标签信息。一般而言，OWASP 中的测试用例中会存在两个方法，一个是 doGet() 方法，但它的具体实现是直接调用 doPost() 方法。因此，可以认为这一测试用例的安全性结果反映的是 doPost() 方法的安全性。在附上的 XML 文件中会显示该测试用例所属的安全漏洞类型的种类、CWE 分类编号以及安全性结果等内容。如图4-2展示的代码片段所属的安全漏洞类型为 Path Traversal，对应的 CWE 编号为 22。vulnerability 标签中的内容反映的是该测试用例的安全性结果，结果为 true，说明该测试用例存在安全漏洞，结果为 false，则说明该测试用例不存在安全漏洞。

因此，Juliet 测试套件与 OWASP Benchmark 数据集均是携带标签信息的数据内容，免去了进行人工标注的繁重工作，在通过简单的处理步骤后即可直接使用。在本文的工作中，我们便使用了这两份数据集中的数据进行了模型的训练及测试工作。

4.3 数据处理

4.3.1 数据筛选

Juliet 测试套件与 OWASP Benchmark 中的安全漏洞类型与测试用例数量众多且不同漏洞种类的用例比例各不相同，若不加以区分，则数据量会过于庞大且极不平衡，会导致模型的学习和训练成本过高。另一方面，现有的研究大都针对特定类别的安全漏洞类型，为了更为方便地与他们的结果进行对比，本文在充分研究这两份数据集中的内容后进行了筛选，筛选标准见表 4-1、表 4-2。

表 4-1是对 Juliet 测试套件数据集的筛选标准。Juliet 测试套件的测试内容中依然存在部分未打上标签的方法，如 main 方法等，这部分的数据无法直

接用于训练。OWASP TOP 10 是 OWASP 组织公布的“10 项最严重的 Web 应用程序安全风险列表”，是 Web 应用程序最可能、最常见、最危险的十大安全漏洞。在本文中，我们选取了 OWASP TOP 10 的前三名，即 OWASP TOP 3 进行实验。它们分别是 A1 注入（Injection）、A2 失效的身份认证（Broken Authentication）、A3 敏感数据泄露（Sensitive Data Exposure）。

同时，为了更好地与 Patil 等人 [42] 的分类结果进行对比，突出本文所提出的安全漏洞预测方法的有效性。本文在已经选定的 OWASP TOP 3 范围中，优先考虑了与他们所使用的 CWE 安全漏洞类型相一致的数据作为初始训练内容。此外，考虑到少量的训练样本容易使深度学习方法的训练结果产生过拟合的现象，因此在筛选标准中本文排除了方法总数量未达到 100 的 CWE 安全漏洞种类。

表 4-2是对 OWASP Benchmark 数据集的筛选标准。与 Juliet 测试套件的筛选标准类似，我们保留了 OWASP TOP 3 的作为纳入标准之一。同时，区别于 Juliet 测试套件中测试用例的具体实现，OWASP Benchmark 中 doPost 方法才是具备代码安全性的内容。

在对两份数据集分别应用相应的纳入标准后，得到的数据信息如表 4-3和表 4-4所示。表中的 A1、A2、A3 表示 OWASP TOP 3 所对应的漏洞类型，在每一种 OWASP 漏洞类型下也同样会存在多种 CWE 安全漏洞类型。ID 列的其余数字是指具体的 CWE 安全漏洞类型，OWASP TOP 10 与 CWE 安全漏洞类型详细对应情况参见（https://cwe.mitre.org/data/definitions/1026.html）。基于不同数据集的实际测试用例情况，Juliet 测试套件共保留了 4 种 CWE 安全漏洞类型，OWASP Benchmark 数据集中由于 CWE 安全漏洞类型数量有限，并未覆盖全部的 OWASP TOP 3，且考虑到与现有工作的对比情况，因此共保留了 1 种 CWE 安全漏洞类型。

表 4-1: Juliet 测试套件中测试用例纳入标准

	Juliet 纳入标准
IN-1	OWASP TOP 3
IN-2	含有标签的测试方法
IN-3	Patil 等人使用的 CWE 安全漏洞类型
EX-1	方法总数量不足 100

表 4-2: OWASP Benchmark 中测试用例纳入标准

	OWASP Benchmark 纳入标准
IN-1	OWASP TOP 3
IN-2	doPost 方法
IN-3	Koc 等人使用的 CWE 安全漏洞类型
EX-1	方法总数量不足 100

表 4-3: Juliet 测试套件保留数据基本信息

ID	描述	不含有安全漏洞的方法数量	含有安全漏洞的方法数量
A1	注入		
89	SQL Injection	11340	3660
90	LDAP Injection	1380	732
A2	失效的身份认证		
256	Plaintext Storage of Password	189	61
A3	敏感数据泄露		
319	Sensitive Cleartext Transmission	1890	610

4.3.2 程序表达

不同于一般的自然语言，代码需要符合特定的语法规则，具有很强的结构性。部分研究表明，获取源代码的语法结构特征有益于对加强对源代码内容的理解 [43][44]。Alon 等人 [45] 进一步指出了程序理解和静态分析之间的关系。倘若完全不对代码进行静态分析，就需要大量的数据和语料来重新学习代码的语法和语义内容，这一过程将会十分缓慢。于是，他们使用了抽象语法树来进行程序分析，并基于提取出来的信息内容进行学习训练来加速这一过程。他们发现，这样的方式更能够反映代码中的一般规律，也能够显著降低模型的学习成本。因此，我们也采用了 Alon 等人的建议，平衡程序分析和文本学习之间的成本，对筛选出来的 Juliet 测试套件与 OWASP Benchmark 中的测试代码进行静

表 4-4: OWASP Benchmark 保留数据基本信息

ID	描述	不含有安全漏洞的方法数量	含有安全漏洞的方法数量
A1	注入		
89	SQL Injection	272	232

图 4-3: 抽象语法树片段示意图

态代码分析，从中提取出抽象语法树信息，并基于提取出的信息内容展开深度学习任务。

抽象语法树是程序语言代码语法结构的一种抽象表示，它以树状结构的形式表征特定代码内容，树上的每一个节点都表示源代码中的一种结构，但也省去了现实语法中的一些非必须的细节，如括号、分号等，常用于语法检查、代码提示、代码自动补全等场景。图 4-3 展示了 CWE563 安全漏洞中部分测试代码的抽象语法树片段。生成的抽象语法树是一段编译单元，以类似 JSON 的格式表现代码的语法描述信息。在这段生成的抽象语法树片段中，我们可以清晰地获取这段代码的变量声明、方法声明、方法调用等结构化信息。利用这部分信息，我们可以更高效地完成对代码文本内容的学习。

4.4 数据解析

基于 BERT 的安全漏洞预测模型需要对方法级别的代码片段的安全性做出预测，这就意味着图 4-3 中展示的抽象语法树片段无法直接应用于模型的训练。本文对得到的抽象语法树内容进行了解析，提取了其中描述方法主体的部分，即在抽象语法树中以“bodyDeclarations”标记的大括号内的内容。这部分内容详细描述了方法体中每一行代码的具体操作，但在实际解析过程中，我们对提取的方法 AST 内容进行了信息删减，这在一定程度上缩减了语料长度，如

我们并未提取代码行信息。

此外，由于 Juliet 测试套件数据集的自身特性，方法名中会指出该方法是否含有安全漏洞、方法体中也会存在方法调用的情况，这样的信息会带有明显的标签内容，极易对训练的结果造成很强的干扰，降低分类模型的泛化能力。因此，本文对这部分的干扰信息统一进行了标准化处理，将带有标签信息的方法名称统一变更为“f1”，将带有标签信息、漏洞信息的方法调用统一更改为“MethodInvocation”。我们也对 OWASP Benchmark 数据集中提取的测试用例应用了同样的标准化步骤。

通过上述标准化操作，我们能够得到相对无干扰的方法抽象语法树文本语料及其标签信息，并以“id，标签，语料内容”的形式作为 BERT 模型的训练内容输入。

4.5 模型训练

4.5.1 BERT 模型

BERT（Bidirectional Encoder Representation from Transformers）模型 [46] 是 2018 年 10 月由 Google AI 团队发布，它在 11 项不同的自然语言处理（Natural Language Processing, NLP）测试任务上均创出最佳成绩，被誉为是近年来自然语言处理领域的里程碑式的突破性成果。BERT 模型使用双向 Transformer 来预训练未标记文本的深层双向表示，通过在海量预料上运行自监督学习方法，将传统方法中大量在下游具体 NLP 任务中做的操作上移到预训练词向量中，形成更为通用的预训练模型。在此基础上，用带标记的训练数据微调（fine-tuning）既有模型，使之能够适用于具体的场景，完成复杂的 NLP 任务。预训练过程需要完成以下两个任务：

任务一：标记语言模型（Masked Language Model）

BERT 在预训练时会随机遮挡（MASK）每一个句子中 15% 的令牌（token），然后用该句子的上下文来预测被遮挡住的令牌。然而对于这些遮挡的令牌在下游任务中可能并不存在，即 BERT 不知道它将会被要求用来预测哪些令牌。因此，在本阶段的任务中，BERT 被要求按照一定的比例在需要预测的令牌位置上输入原来的令牌或者是某个随机的令牌，下面以句子“你喜欢吃桃子”为例，展示 BERT 的标记-学习过程。

1. 有 80% 的概率用“[MASK]”标记随机替换——“你喜欢吃桃 [MASK]”
2. 有 10% 的概率用随机采样的一个 token 来替换——“你喜欢吃桃 [的]”
3. 有 10% 的概率不做替换——“你喜欢吃桃子”

值得注意的是，尽管在标记-学习的过程中，有一定的概率使用随机的令牌对原语句形成干扰，但这一概率为 1.5% ，并不会对学习能力造成过多的影响，反而会迫使 BERT 保持每个输入的令牌的分布式上下文表示，进而增加预训练模型的信息存储和学习能力。

任务二：下一句预测（Next Sentence Prediction）

很多复杂的 NLP 任务需要捕捉句子级别的模型来完成自然语言推理、句子间交互与匹配等任务。为了训练句子级别的模型关系，BERT 预先进行一个下一句测任务，这一任务可以从任何单语语料库中生成。具体来说，当选择句子 A 和句子 B 作为与训练样本对时，B 有 50% 的概率可能是 A 的下一句，也有 50% 的概率可能是来自语料库中的随机句子。即首先给定一个句子，它的下一句即为正例，随机采样一个句子作为负例，然后在该句子级上判断当前子句的下一句是正例还是噪声。以此，可以让 BERT 学习到连续文本片段之间的关系，以适用于更为复杂的具体 NLP 场景。

4.5.2 参数微调

完成预训练的抽象模型已经具备一定的泛化能力，对于具体的下游任务，只需要使用携带标签的下游任务数据对预训练模型的参数进行微调即可。在本文安全漏洞预测的场景下，由于程序代码不同于一般的自然语言文本，尤其是在生成的抽象语法树文本中，会经常性地出现该领域内特有的一些关键词，如 var、null 等，但这些关键词在预训练模型的词汇表中不一定存在。因此，为了使得微调后的预训练模型能够更好地使用代码的安全漏洞预测领域，我们对预训练模型的词汇表进行了内容上的补充。

由于预训练模型已经考虑将来所应用具体场景的特殊性，因此在预训练模型的原始词汇表中已经预留了一部分占位符，它们以 [unusedXX] 的形式存储在词汇表中。其中 XX 是指序号，在预训练模型的词汇表中共有 101 个占位符，从 [unused1] 一直到 [unused101]，供未来在特定场景下微调使用。

于是，我们将其中的部分占位符替换为 Java 语言中的关键字和保留字，以抽象语法树中生成你的结构性提示词。在进行参数微调时，我们同时也注意到表 4-6中的不同标签的数据量差别不是很大，不含有安全漏洞的方法数量与含

有安全漏洞的方法数量的比例大致在 1：3 左右；表4-7中的不同标签的数据量基本接近，我们对这部分用于训练的数据，本文并未做数据不平衡方面的处理。至此，我们完成了预训练模型参数微调的前置工作，而后，我们便可以使用带标签的训练数据来对预训练模型的训练参数进行实际调整，以完成既定的预测任务。

4.6 模型评估实验

4.6.1 评价指标

本研究中，对方法的安全性预测本质是一个二分类的任务，将需要进行预测的方法分为有漏洞和无漏洞两种类别之一。分类结果通常以混淆矩阵的形式呈现，它将结果数据分为四个类别，具体见表4-5。

表 4-5: 混淆矩阵

混淆矩阵		真实值	
		Positive	Negative
预测值	Positive	TP	FP
	Negative	FN	TN

其中，TP（True Positive）表示实际值为正，且预测值也为正的数量；FP（False Positive）表示实际值为负，但预测值为正的数量；FN（False Negative）是指实际值为负，但预测值为正的数量；TN（True Negative）是指实际值为负，且预测值也为负的数量。

对于这种二分类的任务，本文选取了常用的正确率、准确率、召回率和 F1 值指标来具体评价预测模型的性能表现，具体公式如下：

正确率（Accuracy）表示分类模型中，所有预测正确的数量占总测试样本数量的比重。

$$Accuary = \frac{TP + TN}{TP + TN + FP + FN} \tag{4-1}$$

准确率（Precision）表示分类模型中，对于所有预测结果为正的样本，预

测正确的样本比重。

$$Precision = \frac{TP}{TP + FP} \tag{4-2}$$

召回率（Recall）是指在分类模型中，对于实际结果为正的所有样本，预测正确的样本比重。

$$Recall = \frac{TP}{TP + FN} \tag{4-3}$$

F1 值综合了准确率和召回率，用于综合反映分类模型的整体指标。

$$F1 = \frac{2 \times Accuracy \times Recall}{Accuracy + Recall} \tag{4-4}$$

4.6.2 实验结果及对比

本文选取的预训练模型版本为 $BERT - Base, Cased$，全部实验在 Google Colab 上完成。对于每一种数据集，本文使用了五折交叉验证的实验结果作为最终的性能指标，以拓展分类模型的泛化能力。实验结果见表 4-6和表 4-7。

表 4-6展示的是 Juliet 测试套件中针对 OWASP TOP 3 的分类结果，同时，在表中我们对比了 Patil 等人 [42] 的分类结果。从表中的结果来看，无论是本文使用的基于 BERT 的安全漏洞预测方法还是 Patil 等人使用的图神经网络（GNN，Graph Neural Network）方法，针对 OWASP TOP 3 中的 CWE 安全漏洞的分类情况，均取得了不俗的效果，且总体来说，两种方法的性能表现相差不大，对表中大部分的 CWE 安全漏洞分类结果的正确率和查全率均能够达到 95% 以上。

这一现象，可能是与标准数据集本身的构造情况有关。区别于实际工业场景下的业务代码，标准数据集中的代码片段一般较为精简。在人为编写的过程中，为了迅速达到某一特定目的，针对某种漏洞类型的编写规则会可能相对简单并会趋于一致。在使用深度学习进行代码内容理解的过程中，这种规则会被很轻易的学习到，进而造成了性能指标的优越性。Lee 等人 [40]、Gao 等人 [47] 等人在应用 Juliet 测试套件进行实验时，也应证了这一现象。根据他们各自的实验结果，均取得了超高的正确率，平均正确率也达到 96% 以上。这从另一方面也表明，结合代码静态表征和深度学习方法对于理解代码内容是一个值得借鉴的思路。但其实际的应用情况，还需要借助真实的工业项目进一步检验，这

也是本文未来工作的一个重要方向。

针对 CWE256 安全漏洞类型，Patil 等人的方法的查全率只有 83.33%，而本文所使用的方法达到了 92.86%，这是由于 CWE256 的训练样本和测试样本数量过少导致的结果，Patil 等人在该安全漏洞类型上的测试样本量大小仅仅为 30。当测试样本数量过少时，一旦出现错误的分类结果，都将会导致最终性能参数结果的不佳。

表 4-7展示的是使用 OWASP Benchmark 数据集的分类结果，在表中我们与 Koc 等人 [48] 的分类结果进行了对比。在 Koc 等人的工作中，他们只针对 CWE89 这一类型的安全漏洞进行预测，并使用了多种不同的数据处理方法，本文选取的是他们所使用的方法中，所取得的分类效果最好时的性能指标，各项性能指标均达到了 100%。而本文所使用的安全漏洞预测办法的各项指标也均在 96% 以上。这进一步说明了，标准数据集的代码构造原则和内容相对简单，对代码进行静态分析后的中间内容差异不大，使用深度学习方法能够较为容易地识别出其中的规则，进而取得较好的性能指标。

表 4-6: Juliet 测试套件预测结果及对比情况

	CWE ID	Accuary(%)		Precision(%)		Recall(%)		F1(%)	
		ours	Patil's	ours	Patil's	ours	Patil's	ours	Patil's
A1	89	98.33	98.40	98.34	97.82	94.92	96.67	96.60	97.24
	90	99.05	98.85	99.32	98.63	98.00	97.29	98.52	97.95
A2	256	96.00	96.66	92.86	100.0	92.86	83.33	94.40	90.90
A3	319	99.20	98.76	99.20	95.71	97.64	98.52	98.41	97.09
Average		98.28	98.17	97.66	98.04	96.26	93.95	97.25	95.80

表 4-7: OWASP Benchmark 预测结果及对比情况

	CWE ID	Accuary(%)		Precision(%)		Recall(%)		F1(%)	
		ours	Koc's	ours	Koc's	ours	Koc's	ours	Koc's
A1	89	96.01	100.0	96.49	100.0	98.21	100.0	97.56	100.0

为了进一步验证本文所提出的安全漏洞预测模型在混合预测实验中的性能，即在多种 CWE 安全漏洞类型同时存在的情况下，预测模型是否能够正确预测出方法级别的代码片段的安全性。即使是在混合了多种安全漏洞类型的数据作为训练集的情形下，这里的安全性预测仍然是一个二分类的问题。因为在

本文的提交优先级排序场景下，只需要对一个函数方法是否含有漏洞做出判断，而不需要对这个预测为存在安全漏洞的方法进一步判断其隶属于哪种具体的安全漏洞。

我们将4-3中保留的几种 CWE 安全漏洞类型进行了混合。在混合的过程中，我们发现不同安全类型的训练数据数量级也不同，CWE90 漏洞类型与 CWE319 漏洞中类的方法总数基本相似，但 CWE89 漏洞类型与 CWE256 漏洞类型的方法总数与它们的方法总数的数量级却相差较大。CWE89 漏洞类型的总方法数量达到了 15000，而 CWE256 漏洞类型的方法总数量只有 250。面对如此悬殊的数量级差距，预测模型很难学习到小部分样本的特征。因此，在数据的处理上，本文并未将 CWE256 漏洞种类的数据纳入到混合预测的实验中，同时对 CWE89 漏洞类型的总方法数量进行了缩减，减少到与 CWE90 和 CWE319 的方法总数大体保持一致，随机选取了 CWE89 漏洞类型的 20% 的内容作为混合实验的训练数据，数据的基本信息内容见表4-8。

表 4-8: 混合实验数据基本信息

ID	描述	方法总数量	不含有安全漏洞的方法数量	含有安全漏洞的方法数量
89	SQL Injection	3000	2268	732
90	LDAP Injection	2112	1380	732
319	Sensitive Cleartext Transmission	2500	1890	610
	Total	7612	5538	2074

针对表4-8中的数据，本文训练得到的预测模型的性能结果如表4-9所示。从实验结果来看，在混合实验的场景下，本文所提出的预测模型的性能仍然保持在一个较高的水准，正确率达到了 96.65%、查全率达到了 93.53%，这也进一步应证了人造数据集中规则的可循性及基于 BERT 的安全漏洞预测模型能够在代码片段安全性的预测领域应用的有效性。在接下来的提交优先级排序方法的效果验证实验中，我们也将以这一性能数据为基础，进行仿真实验。

表 4-9: 混合实验结果

Accuary(%)	Precision(%)	Recall(%)	F1(%)
96.65	95.01	93.53	95.06

4.7 排序原型

基于 BERT 的安全漏洞预测模型能够针对方法级别的代码，就其安全性做出判断。但一次代码提交中会存在多份代码文件，每一份的代码文件中又会包含多个函数方法。在进行提交优先级排序时，需要准确预测本次提交中存在安全漏洞的方法个数，并以此作为提交优先级排序的依据。因此，本文以基于 BERT 的安全漏洞预测模型为基础，构建了面向代码安全的提交优先级排序方法的排序原型。

排序原型所实现的排序逻辑参照算法 3.1，利用安全预测模型对方法级别的代码片段的预测能力，实现对提交的优先级排序。假设一次提交中的函数方法总数为 n，单个函数方法被分类器预测为含有安全漏洞的概率为 p，那么，从理论上来讲，排序原型会预测本次提交中含有的漏洞方法个数的期望为 $E(x) = n * p$。由于函数方法的数量大小并不会影响到对提交中方法安全性的预测判断，因此，在接下来的仿真实验中，本文依然沿用了对单个方法的预测性能指标数据来表现对提交中所有函数方法安全性的预测性能。

排序原型是直接对本次提交中的所有方法的安全性预测结果进行累加，这一作法基于的假设是方法之间是相互独立的。也正因如此，本文可以在方法级别的粒度上，获取方法级别的代码片段的抽象语法树信息，对其应用深度学习方法做出安全性的预测，而未考虑其他诸如信息流、数据流等额外的代码表征信息。从本文选取的两个数据集中的测试代码来看，由于是人为设计的标准数据集，在实现上，不同的函数方法之间具备较强的独立性。但在实际工业场景下，函数之间的调用和依赖情况会比较频繁，仅仅考虑从抽象语法树中提取的部分信息不一定能够准确预测代码片段的安全性，这会对安全漏洞预测模型本身提出一定的挑战。受限于方法级别的公开安全漏洞数据集的匮乏，本文并未选择耗费大量时间和精力去构建这样一个安全数据集，但这将作为本文未来工作的一个重点方向之一。

4.8 本章小结

本章详细介绍了基于 BERT 的安全漏洞预测模型训练的全过程。从数据获取开始，本文选择了 Juliet 测试套件与 OWASP Benchmark 两份标准数据集作为实验对象。根据不同数据集中测试用例的实际特征，本文对两个不同的原始数

据集应用了不同的筛选标准，得到相应的训练数据和测试数据。同时，本文从测试用例代码中提取出抽象语法树信息，并使用 BERT 预训练模型进行参数微调，得到适用于特定任务下的分类模型。此外，本文将训练得到的分类结果与最新的一些研究进行了对比分析，表明了本文所使用的预测方法的性能优势。最后，本文对提出安全漏洞预测模型全流程进行了原型封装，并基于一定的提交排序规则，形成面向代码安全的提交优先级排序方法。

第五章　基于离散事件仿真的持续集成模型

为有效评估本文提出的面向代码安全的提交优先级排序方法的实际效果，本文使用基于离散事件的软件过程仿真进行模拟实验。基于离散事件的软件过程仿真能够根据特定的离散事件触发不同的流程状态，精确地模拟出整体流程的状态变化，从而反映实际执行流程。

5.1　静态过程模型的构建

本文所分析和抽象的持续集成步骤是基于开源的持续集成工具 Travis CI 的实际过程，其大致过程如图 5-1所示。

开发人员提交的代码首先会进入持续集成平台中的等待队列，当存在空闲的服务器资源时，提交将会从队列中出列，并被分配相应的服务器资源，去执行既定的 CI 脚本，如编译、测试、打包等。若本次提交在整个执行过程中未发现任何问题，将会通过本次集成，并完成对提交的代码的合并工作；倘若在整个构建过程中的任意一个环节出现问题，那么本次集成就会失败，开发人员需要修复本次集成中发现的问题，并进行再次提交，直至完成提交的合并。

在 Travis CI 现有的持续集成流程中，一次提交分可以为两种状态：Pull Request 和 Push。Pull Request 是指开发者向团队提出合并自己提交的代码的申请，团队会执行拉取这部分代码的操作，对拉去的代码进行后续的构建任务；

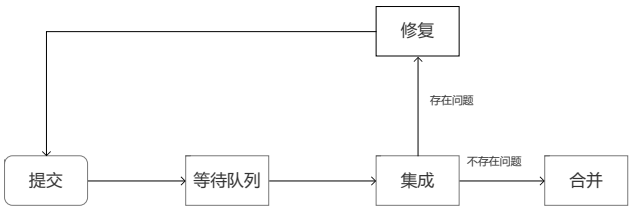


图 5-1: 持续集成全流程示意图

而 Push 是指开发者提交自己的代码至指定仓库，直接执行构建任务，而不需要通知团队进行代码的拉取。实际上，我们也注意到部分提交可以通过人为配置，跳过构建任务的执行，这部分代码将直接集成到主干中，并在后续某一次的提交到来时一并进行验证。对于执行构建任务的提交，通常会存在以下几种状态之一：

- 通过：本次提交成功执行集成流程中的所有步骤。
- 取消：用户在执行集成流程中途取消执行。
- 错误：在执行构建任务前由于环境、网络等外部因素而造成的集成错误。
- 失败：未通过构建任务。

通过的提交意味着本次提交的代码通过了所有的构建任务，并能够顺利完成合并，而失败则意味着开发者需要花费额外的时间对本次集成进行修复。取消和错误两种状态均是由外部环境因素或者人为干扰而导致的问题，其本身并未执行 CI 脚本中的构建任务。因此，并不能说明本次提交的内容本身是否存在问题。在本文的工作中，我们忽略了这两种状态，只对通过和失败两种状态的提交进行了考虑。

进一步地，当安全检测被加入到持续集成的构建任务中时，提交需要通过先前的编译测试任务后，再执行安全检测任务，这个整体过程仍然被视为一个完整的构建任务。但为了与先前不进行安全检测任务的构建区分开来，本文将新的构建称为进行安全检测的构建任务，这个全新的构建任务需要串行执行编译测试任务和安全检测任务两个部分。同时，为保持整体持续集成的完整性，开发人员仍然可以人为设定跳过本次的构建任务，跳过构建任务的提交代码，也将会在接下来的提交中批量完成需要的编译测试及安全检测任务。

当在持续集成过程中应用本文所提出的面向代码安全的提交优先级排序方法时，执行构建任务的提交将不再遵循“先进先出”的原则，这些提交需要在等待队列中按照特定的规则进行排序，即提交优先级排序方法会统计本次提交的代码中所包含的函数方法数量，并对每一个方法级别的代码片段进行安全预测，统计出本次提交中可能含有安全漏洞的方法数量，并以此项数据作为提交排序的依据，优先让含有漏洞个数更多的提交出列，从而获得服务器资源进行实际的构建任务。实际执行构建任务时，提交会进入持续集成平台依次执行编译测试和安全检测两个不同的任务，并最终分为成功以及失败两种状态。对于状态为失败的提交，它未能通过构建任务中的所有子任务，若未能通过编译测试任务，则需要进行相应的故障定位与缺陷修复；若未能通过安全检测任务，

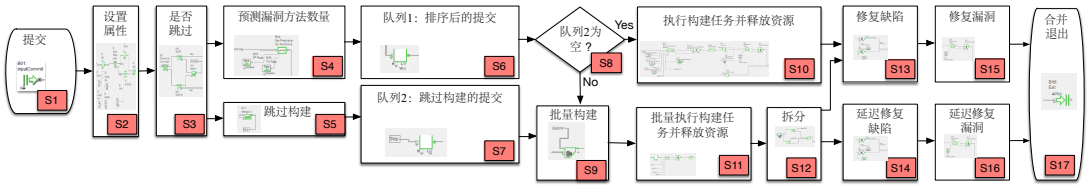


图 5-2: 动态仿真模型流程图

则需要针对所提示的危险代码内容，进行针对性的源代码修改；而对于状态为成功的提交，其代码才能够顺利进入主干进行合并。

5.2 动态仿真模型的构建

上一节基于 Travis CI 的实际持续集成过程，构建了持续集成的静态流程。本节将会进一步使用基于离散事件的软件过程仿真方法构建动态的仿真模型，其内容如图 5-2所示。

构建完成的仿真模型共分为 17 个步骤，为表述方便，本文使用 S1 至 S17 对每一个步骤进行编号加以区分，并在每一步的内部，附上了由 ExtendSim 实现的具体抽象模块，完整的 ExtendSim 实现见附录 B。同时为方便阐述，本文给这些具体的抽象模块进行了编号，如 B01、B02 等，具体模块表示见附录 A-1。接下来，本文就每一个步骤的具体内容做出解释。

S1：提交。进入持续集成平台中的提交会执行相应的构建任务，这些提交中一般包含开发人员开发完成的代码或配置文件等。当模型开始运行时，B01 会随机产生一系列提交。通过对 Travis CI 中的提交日志进行分析后，我们发现，当提交数量达到一定规模时，这些提交之间的时间间隔近似满足泊松分布。

S2：设置属性。当提交被创建后，每一项提交将按照一定比例随机分配相应五个属性。一是提交的类别 B03。提交的类别可以是 PullRequest 或者是 Push，分配的比例可以由具体项目的构建日志分析得到；二是本次提交的编译测试任务结果 B04。在排除人为和外部环境因素的干扰后，当一个提交开始执行构建任务时，它的结果就已经确定。三是本次提交中是否含有安全漏洞 B05。当提交进行安全检测任务时，这一属性值的结果也已经确定。在本研究中，这一属性值由于自身的不确定性，被设置为可以动态进行改变，以便进一步探究提交漏洞率对提交优先级排序效果的影响。四是本次提交中所含的方法

个数 B06。若本次提交的内容中存在源代码内容，那么这一属性值反映的是这些源代码中存在的函数方法的总数量，这是进行提交优先级排序的重要数据之一。五是本次提交中实际含有的漏洞方法的数量 B08 与 B09。若本次的提交内容中含有源代码，经过安全检测任务后，提交中的漏洞方法会被检测出来，这些实际的漏洞方法数量将会直接影响安全漏洞的修复时间。在仿真模型的构建时，这一属性值的设定根据本次提交中是否含有安全漏洞被分成了两种情形，即有漏洞提交和无漏洞提交。对于无安全漏洞方法的提交，直接设置实际含有的漏洞方法数量为 0；而对于有漏洞方法的提交，其数值则有本次提交中所含的方法总数量和漏洞方法占比的乘积决定。

S3：是否跳过。在提交进入持续集成平台中时，不一定都会执行构建任务，如在一些特殊情况下，可以人为设定跳过构建任务，直接进入主干完成合并。如果本次提交跳过构建任务，那么会直接进入到 Queue-2，当下一次提交到来时，再一并完成构建任务；如果本次提交没有跳过构建任务，那么就会进入到 S4 对这些提交进行安全漏洞方法预测。考虑到不同的提交类型可能具备不同的跳过概率，在实际仿真时需要对具体项目的提交日志进行分类别的分析。

S4：预测漏洞方法数量。在提交执行具体的构建任务之前，我们需要对到来的提交的安全性做出预测，以便提交进入等待队列 Queue-1 中进行排序。这一数值是通过预测模型的性能参数 B19 和 B20 计算得到，B19 是指模型将未含有安全漏洞的方法错误地预测为含有安全漏洞方法的概率，B20 是指模型正确将出含有漏洞的方法预测为含有漏洞的方法的概率。

S5：跳过构建。如果需要跳过本次提交的构建任务，此步骤将会合并 Pull Request 和 Push 两种类型的提交，并进入到 Queue-2 进行等待。

S6：排序后的提交。进入持续集成平台中的提交在完成步骤 S2 中的基本属性设定和在步骤 S4 中预测漏洞方法数量后，进入 Queue-1 等待服务器资源的分配。当一个新的提交进入到 Queue-1 时，若此时队列为空，则直接进入队首；如果此时队列不为空，那么将会根据 S4 中预测得到的漏洞方法数量的大小进行排序，让含有更多漏洞方法的提交尽早出列，尽早执行提交的构建任务，让真正的缺陷和安全漏洞暴露出来，并对进行及时的修复，以提高安全漏洞的反馈效率，减少整体的构建时长。

S7: 跳过构建的提交。跳过构建的提交会进入到 Queue-2 中进行等待，其实质上是跳过本次的构建。当下一次的提交到来时，检测到 Queue-2 不为空，会进入 S9 完成提交的批量构建任务。

S8：队列 2 是否为空。在提交执行构建任务之前，需要进行队列 2 是否为空的判断。若队列 2 不为空，说明有先前跳过的提交中的代码还没有执行构建任务就完成了代码的合并，那么接下来从 Queue-1 中出列的提交进行构建任务时，将会一并执行构建任务。反之，若队列 2 为空，那么从 Queue-1 出列获取到服务器资源的提交将会直接进行 S10 的构建任务。

S9: 批量构建。批量构建会一次性完成多个提交的构建任务，其包含的提交由两个部分组成。一部分是从 Queue-1 中出列的提交，另一部分是 Queue-2 中的所有提交。只有当检测到 Queue-2 中的提交数量不为 0 时，从 Queue-1 中出列的提交才能与之完成绑定，去执行批量的构建任务。

S10：执行构建任务并释放资源。在仿真模型中，进入持续集成平台中的提交所需要经历的构建任务分为编译测试任务和安全检测任务两个部分。提交在执行这两个部分的任务时是串行执行，需要依次通过编译测试任务和安全检测任务才能算作是完成了提交的构建任务。而两个任务所需的服务器资源并不相同，这意味着前一个提交在执行安全检测任务时，将会释放其先前占有的编译测试任务的服务器资源 B22，后一个提交可以被分配到刚刚释放的服务器资源进行编译测试任务。在执行编译测试任务时，不同类型、不同编译测试结果的提交一般具备不同的任务时长 B24-B27。通常，成功通过编译测试任务的提交耗时更长，因为它们会执行完毕 CI 脚本中编译测试任务的所有内容，而未能通过的提交则会在故障出直接停下，不会继续执行。因此，在这一步中，我们为不同执行结果、不同类型的提交分别设置了不同的延迟时长。当提交执行编译测试任务完毕后，将会释放相应的服务器资源。释放完毕后，通过编译测试任务的提交将会再依次获取安全检测所需的服务器资源 B50，执行相应的安全检测任务。执行安全检测任务的耗时将由 B52 决定。当本次提交的安全检测任务完毕时，将会释放获取的服务器资源，并得到本次提交的实际安全漏洞信息。

S11：批量执行构建任务并释放资源。如 S9 中所阐述的那样，当 Queue-2 不为空时，从 Queue-1 中出列的提交将会与 Queue-2 中的所有提交一并执行构建任务。执行的构建任务总体流程如 S10 中所描述的那样，分为编译测试任务和安全检测两个任务串行执行。

S12: 拆分。S9 中已经阐明批量执行构建任务的提交由两个部分组成，这两部分的提交来源不同。若本次批量执行构建任务的结果为失败，那么对于从 Queue-1 中出列的提交的修复属于及时修复；而对于从 Queue-2 中出列的所有

先前跳过构建任务的提交，它们已经不属于其原始提交的次序，对它们的修复是延迟的修复。因此，在仿真模型中，我们对 S11 批量执行构建任务的提交进行了拆分。

S13: 修复缺陷。执行编译测试任务是为了发现提交中的功能缺陷及其他相关问题。对于未能通过编译测试任务的提交需要进行修复工作，以便下一次提交能够顺利通过。从实际构建日志的分析结果来看，不同提交类型出现功能缺陷时，所需的修复时长也不尽相同。因此，在本步骤中，根据提交类型的不同，我们将这些提交区别开来进行分别修复，修复时长分别依赖 B34 与 B36。

S14: 延迟修复缺陷。与 S13 类似，延迟修复缺陷是对从 Queue-2 中出列的先前跳过构建任务的提交的功能缺陷的修复。在所构建的仿真模型中，本文认为 S13 及时的修复和 S14 延迟的修复只是功能缺陷发现时机的不同，当展开相应的修复工作时，将不会展现出明显的差异。因此，在模型的处理上上，本文并未对 S13 与 S14 中的修复耗时加以显性区分。

S15：修复漏洞。安全检测任务会对本次提交的代码中的所有安全漏洞进行修复。区别于 S13 步骤中修复编译测试任务中的缺陷问题，S15 中的安全漏洞精确到方法的个数，能够准确获悉含有安全漏洞的方法的数量。因此，本步骤中修复安全漏洞的耗时也会因为含有安全漏洞的方法数量的不同而不同。在对本提交中的所有安全漏洞完成修复后，即可进入 S17 进行代码的合并和退出。

S16：延迟修复漏洞。与 S15 类似，延迟修复漏洞你是对从 Queue-2 中出列的先前跳过构建任务的提交的安全漏洞的修复。与 S14 中所陈述的原因类似，在所构建的仿真模型中，本文也并未将 S15 与 S16 中所需的修复耗时加以显性区分。

S17：合并退出。本次提交的代码能够顺利与主干合并，并结束本次提交的生命周期。

5.3 仿真模型的验证和校准

动态仿真模型构建完成后，需要对构建完成的模型进行验证和校准，进一步确保所构建模型的正确性和可用性。参照现有的软件过程仿真模型的验证和校准方面的研究 [49][50]，本文从语法和语义两个方面对模型质量进行校验。

语法校验是为了验证语法的正确性。在本研究中，多名经验丰富的软件过

程仿真的研究者对本文所构建的仿真模型进行了手动检查。他们检查了所构建的仿真模型的整体逻辑，并对每一个模块的参数及模块之间的联系进行了仔细核对，确保了构建过程中所使用的模块不存在语法问题。

语义校验是为了验证模型的可用性和完备性。我们从模型结构、模型变量和模型结果三个方面进行了验证。首先，为了验证模型结构的一致性，我们手动检查了持续集成的静态模型和动态仿真模型之间的映射关系，核实了图5-2中每一步的仿真模型的实现细节。为保障模型变量的正确性与合理性，我们统一了模型所使用的参数的度量单位，并对模型所使用的所有变量的概率分布进行了校准。

在本文所构建的动态仿真模型中，使用到的唯一计量单位是时间。于是我们手动检查了模型中每一个模块的计量单位，将所有模块的计量单位统一，单位为分钟。接着，基于从 Travis CI 中获取得到的特定项目的提交日志信息，我们应用了 Kolmogorov-Smirnov 测试 [51] 方法对这些数据变量进行校验，以判断这些变量近似服从何种分布函数，校准后的变量数据见表5-1。在完成对模型结构和模型变量的校准后，我们最后对模型结果进行了校验。对模型结果的校验是为了确保模型的每一个分支在给定的变量参数下都能够按期执行。在结果校验的过程中，我们将模型中的每一条分支分别设置为 0%，50%，100% 进行多组极限实验，并通过 ExtendSim 中的慢动画模式观察所创建的提交在工作流中的流动情况。结果表明，模型中的每一条分支在给定参数下的执行结果都符合预期。

表 5-1: 项目数据变量分布情况

变量名称	内容描述	变量分布
CommitType	本次提交的具体类型占比 PR，Push	{0.06，0.94}
Build&Test Result	执行编译测试任务的结果为失败或者通过失败，通过	{0.06，0.94}
TotalMethodNum	每条提交所含的方法总数，服从泊松分布	7
SkipPR	PR 类型的提交跳过编译测试任务的概率	0.61
SkipPush	Push 类型的提交跳过编译测试任务的概率	0.28
PRPassedDelay	成功执行 PR 类型的提交所需要的时间，服从均匀分布	[11,23]
PRFailedDelay	失败执行 PR 类型的提交所需要的时间，服从均匀分布	[3,13]
PushPassedDelay	成功执行 Push 类型的提交所需要的时间，服从均匀分布	[4,21]
PushFailedDelay	失败执行 Push 类型的提交所需要的时间，服从均匀分布	[3,12]

至此，我们从语法和语义两个方面完成了对动态仿真模型的校验，保障了构建出来的模型能够客观地反映现实地持续集成过程。接下来，本文将通过实

际的项目数据对所提出的面向代码安全的提交优先级排序方法的有效性进行实验与分析。

5.4 本章小结

本章介绍了构建软件过程仿真模型进行方法有效性验证的全过程。首先我们对 Travis CI 的实际持续集成过程进行了分析，接着将持续集成的静态模型和动态仿真模型流程进行了映射，利用 ExtendSim 工具具体实现动态仿真模型的构建，最后从语法和语义两个方面对构建完成的模型进行手动校验，进一步保障了构建完成的模型的准确性和可用性。

第六章　基于仿真的排序效果评价

6.1　实验设计

为了评估本文提出的面向代码安全的提交排序方法的实际效果，我们基于软件过程仿真技术，对真实的开源项目展开案例研究。

我们选择了 Graylog/Graylog2-server 这一开源项目进行仿真实验。该项目使用 Travis CI 进行构建，累计的构建日志数据丰富，相应的构建日志信息也能够在 TravisTorrent 上获取。此外，这一项目也曾被 Liu 等人 [52] 用于持续集成场景下的相关研究，具有一定的代表性。因此，我们基于该项目的原始数据，通过更改参数变量的数值大小，进行了一系列仿真实验，探究本文所提出的提交优先级排序方法在不同场景下的实际效果。

开源项目的基本统计信息见表 6-1。2012 年至 2014 年期间，该项目的累计提交达到 11683 次，其中触发构建的次数为 5199 次，构建成功的提交为 4648 次，构建失败的提交为 307 次。我们对这 11683 次提交的内容进行了回溯，统计得到所有提交的 Java 文件中的方法总数为 77946 个。由于再仿真模型中，实际参数输入需要符合某种具体的函数分布，本文在对获取的统计数据及构建日志应用 Kolmogorov-Smirnov 方法进一步分析后，得到各属性参数分布如表 5-1所示。这部分数据能够反映该开源项目在执行构建任务及其相关修复时的基础特征，是本文构建软件过程仿真模型的基础数据。

但仍然有部分仿真参数无法从构建日志中直接分析得到，尤其是当安全检测任务被加入到持续集成流程中时，现有的构建日志无法获取安全扫描时长、漏洞修复情况等信息。因此，我们基于现有的研究资料和实践情况，对这些仿真参数进行了人为设定。

对于未通过编译测试任务的提交需要进行修复工作，但其修复时长却因出现的具体问题不同、开发人员具备的技能不同而有所差异。我们在进行相关调研后发现，不同的研究中观察得到的持续集成场景下的缺陷修复时长差别很大 [53][54][55]，从 5 分钟到 60 分钟不等，因此，在本文对取了这些观察得到的数字进行了折中处理，将编译测试任务的修复时长设置为符合均值为 30，方差为

10 的高斯分布。对于安全检测中发现的安全漏洞的修复时长的设定，我们也使用了近似的漏洞修复时长数据。但由于安全问题的迫切性，修复安全漏洞的平均时长相较于普通缺陷的要短 [56]。因此，我们将这一属性值设置为符合均值为 20，方差为 10 的高斯分布。对于安全检测任务的耗时，我们基于 Bolduc 等人 [14] 的实践情况，将这一属性值设置为符合均值为 10，方差为 10 的高斯分布。

此外，由于实际提交中漏洞情况存在很大的不确定性，在实际仿真时，我们并未量化这些属性，而是对相关属性进行多值设置，以探究本文所提出的提交优先级排序方法在不同情况下具备怎样的实际表现。因此，我们将提交中含有漏洞的概率 P_{vc} 设置为从 0.1 到 0.9 不等，含有漏洞的提交中漏洞方法占比 P_{vm} 也设置为从 0.1 到 0.9 不等。同时，为更好仿真出提交的排队情况，我们参考了 Liang 等人 [1] 指出的 Google 实际集成场景的提交频率及等待队列长度等信息，将仿真实验的提交频率 f_c 设置为 10/8/6/4 不等的四个层级，但提交之间的间隔并不是一个固定值，而是被设置为服从泊松分布，增加了提交的随机性，以此探究不同影响因素的干扰下，本文所提出的优先级排序方法的实际表现。

在仿真实验中，我们共探究了 648 种不同的情况（81 ），比对了使用提交排序方法和未使用两种不同的情况下的结果。对每一种情况，我们执行了 200 次模拟，并取模拟结果的平均值作为每一种情况的实际结果，以平滑仿真实验的随机性。同时，为限定每一种情况的仿真时长，我们将一次仿真的时长设定为 43,200 分钟（即 30 天）。

表 6-1: 项目基本信息

项目名称	开始时间	结束时间	提交次数	构建次数	成功次数	失败次数	方法总数
Graylog2-server	2012/04/16	2016/08/31	11683	5199	4648	307	77946

6.2 评价指标

在本研究中，为评估面向代码安全的提交优先级排序方法的实际效果，我们使用相对提交节约时间 S_{rc} 这一指标来进行度量。这一指标反映的是，相对

于未进行提交优先级排序的持续集成过程，使用提交排序方法的集成过程工作流时长的相对减少量占比。其定义如下：

$$S_{rc} = (T_{orig} - T_{prior})/T_{orig} \tag{6-1}$$

其中，T_{orig} 是未使用提交优先级排序的持续集成场景下的提交工作流平均耗时，T_{prior} 为使用了本文所提出的提交优先级排序方法的持续集成场景下的提交工作流平均耗时。

一次完整的提交工作流耗时是指一次提交进入持续集成流程直至完成持续集成流程中的所有任务的全过程的整体耗时。在我们的仿真模型中，一次完整的提交工作流应当是以下几种状态之一。

- 成功通过编译测试任务和安全检测任务的提交。
- 成功通过编译测试任务，但在进行安全检测任务时发现安全漏洞，即未能通过安全检测任务，但完成了所有安全漏洞的修复的提交。
- 未能通过编译测试任务，但完成对编译测试中的故障问题修复工作后，通过安全检测任务的提交。
- 未能通过编译测试任务，也未能通过安全检测任务，但完成了对编译测试中的故障问题和安全检测任务中的所有安全漏洞的修复工作的提交。

6.3 仿真结果与分析

6.3.1 提交漏洞率 P_{VC} 影响分析

图6-1、图6-2、图6-3以及图6-4的仿真结果分别展示了，在不同提交频率 f_c（从 10 次/min 至 4 次/min），不同漏洞方法占比 P_{vm} 下（从 0.1 至 0.9），提交漏洞概率 P_{vc} 对最终排序方法有效性的影响。其中，横坐标表示提交漏洞率 P_{vc}，纵坐标表示本次提交在持续集成流程中的平均工作流耗时 T。

结果表明，当 f_c=10 时，无论 P_{vm} 与 P_{vc} 如何变化，T_{prior} 与 T_{orig} 基本相差不大。但当 P_{vm} 一定时，随着 P_{vc} 的增加，T 却呈现出增长趋势。这是由于进入流水线的提交中，含有漏洞的提交占比较大，即使通过优先级的排序方法被置于队列前部位置，缩短了提交的等待时间，但这些有漏洞的提交仍然需要花费一定的时长进行修复。相较于无漏洞直接完成工作流的提交而言，这些累计的修复耗时仍然会在一定程度上增大提交工作流的平均耗时 T。

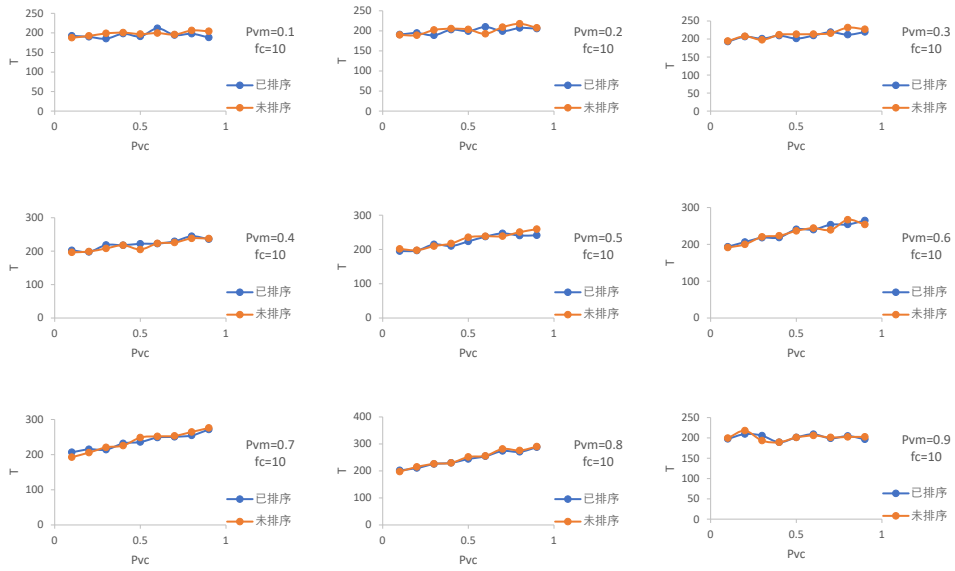


图 6-1: f_c=10，P_{vc} 变化时的仿真结果

随着 f_c 的提高，排序的效果才逐渐明显。当 f_c=8 和 f_c=6 时，无论 P_{vm} 和 P_{vc} 如何变化，T_{orig} 基本维持稳定，T_{orig} 变化的幅度数量级显著小于提交的平均工作流耗时 T 的数量级。但随着 P_{vc} 的增加，T_{prior} 呈现出明显的减少种趋势。尤其是自 P_{vc}=0.5 开始，T_{prior} 的耗时曲线骤然减少，并逐渐趋于稳定。这一现象在 f_c=4 时尤为明显，从 P_{vc}=0.6 开始，T_{prior} 的时耗变化幅度缩窄。这说明，在 f_c 满足一定的提交频率前提下，随着 P_{vc} 的增加，含有漏洞的提交数量也将会增长，那么累计的修复时长也会随着增加。通过前移含有漏洞的提交，来尽量消除这部分的影响能够切实减少提交的平均工作流耗时。但当 f_c 的数值超过某一频率、P_{vc} 也达到某一概率时，在有限的仿真时长内，仍然等待服务器资源和未修复完成的提交将无法完成相应的集成任务，也就无法计算这部分提交的平均工作流耗时。因此，在这种情况下，通过集成任务的提交数量差别不大，显示的仿真结果图形中也会相应地呈现出趋于平缓的特征。

6.3.2 漏洞方法占比 P_{vm} 影响分析

图 6-5、图 6-6、图 6-7和图 6-8的仿真结果展示了，在不同提交频率 f_c（从 10 次/min 至 4 次/min），不同提交漏洞率 P_{vc} 下（从 0.1 至 0.9），漏洞方法占比 P_{vm} 对最终排序方法有效性的影响。其中，横坐标表示漏洞方法占比 P_{vm}，纵坐标表示本次提交在持续集成流程中的耗时 T。

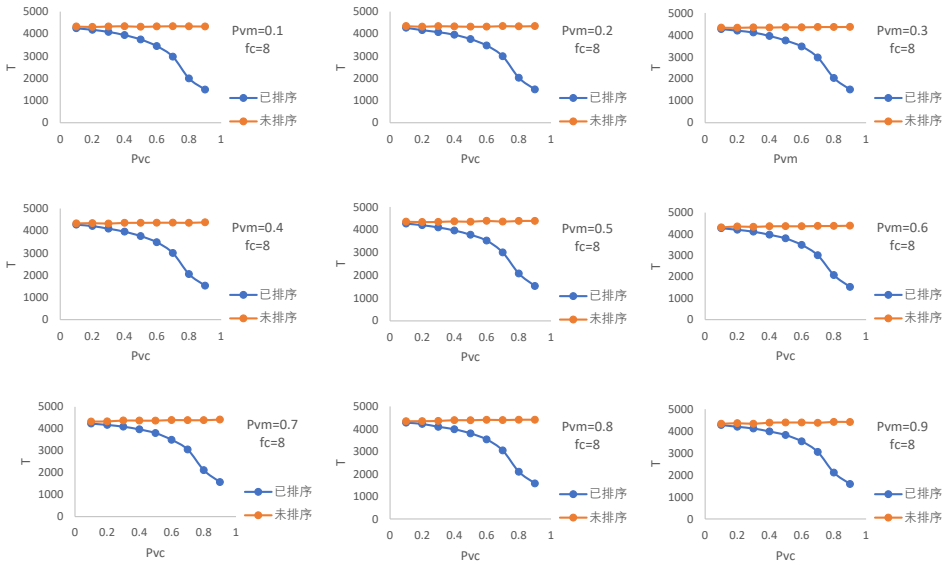


图 6-2: f_c=8，P_{vc} 变化时的仿真结果

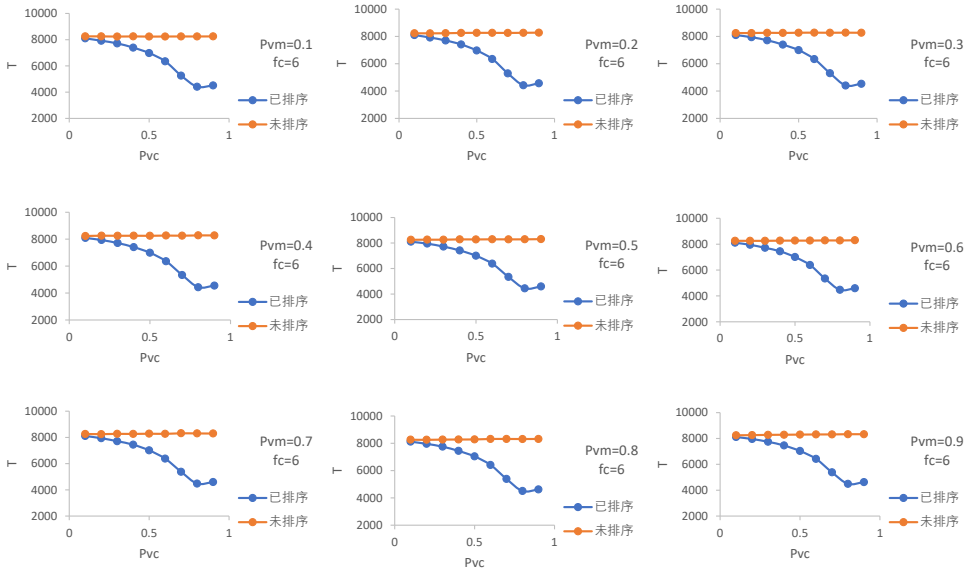


图 6-3: f_c=6，P_{vc} 变化时的仿真结果

结果表明，当 f_c=10 时，无论 P_{vm} 与 P_{vc} 如何变化，T_{prior} 与 T_{orig} 相差不大，使用排序方法并未能够真正减少提交的工作流耗时 T。但从结果曲线可以看出，当 P_{vc} 一定时，随着 P_{vm} 的增加，T 会呈现出增长趋势。这是由于进入流水线的提交中，含有漏洞的提交中漏洞方法占比增多，这意味着开发者需要花费更多的时间对本次提交中的安全问题进行修复。相较于无漏洞直接完成工作流的提交而言，这些提交中累计的修复耗时会在一定程度上增大提交工作流

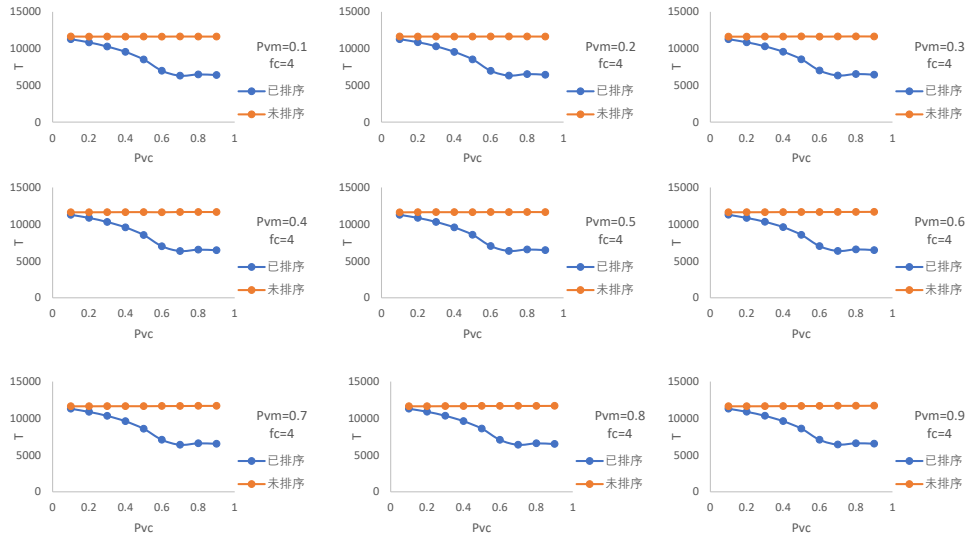


图 6-4: f_c=4，P_{vc} 变化时的仿真结果

的平均耗时 T。

随着 f_c 与 P_{vc} 的提高，排序的效果逐渐显现，但在高频率的提交下，无论 P_{vm} 如何变化，T_{orig} 与 T_{prior} 均基本维持稳定，只是在很小的范围内波动，在这种情况下，对含有漏洞的提交进行修复的时长比提交在等待队列中等待的时长要小得多，因此会呈现出波动幅度的数量级远小于在该提交频率下的平均工作流耗时 T 的数量级。这也从另一个方面说明了，相较于 f_c 与 P_{vc}，P_{vm} 对提交的平均工作流耗时影响有限。

f_c 与 P_{vc} 的变动能够极大地影响 T，而 P_{vm} 只有在 f_c 较小的情况下才能够较为明显地显示出对 T 的影响趋势。此时不会存在大量的提交在等待资源，提交可以较为顺利地完成集成过程。没有了需要等待资源的时耗，T 的数量级也就不会太大，只与编译测试任务耗时、安全检测耗时、漏洞修复耗时这三者密切相关。倘若本次提交存在安全问题，那么 P_{vm} 越大意味着本次提交中需要修复的安全问题越多，所需要的漏洞修复时耗也就越长，在结果上就会呈现出 T 增大的趋势。但当 f_c 达到某一频率时，等待队列中的提交数量急剧增加，提交的平均等待时间将成为影响 T 的数量级的主要因素。

6.3.3 提交频率 f_C 影响分析

从6.3.1与6.3.2中展示的仿真结果来看，f_c 对 T_{orig} 与 T_{prior} 大小的影响均至关重要。不同的提交频率意味在相同的仿真时长内，会有不同数量的提交产

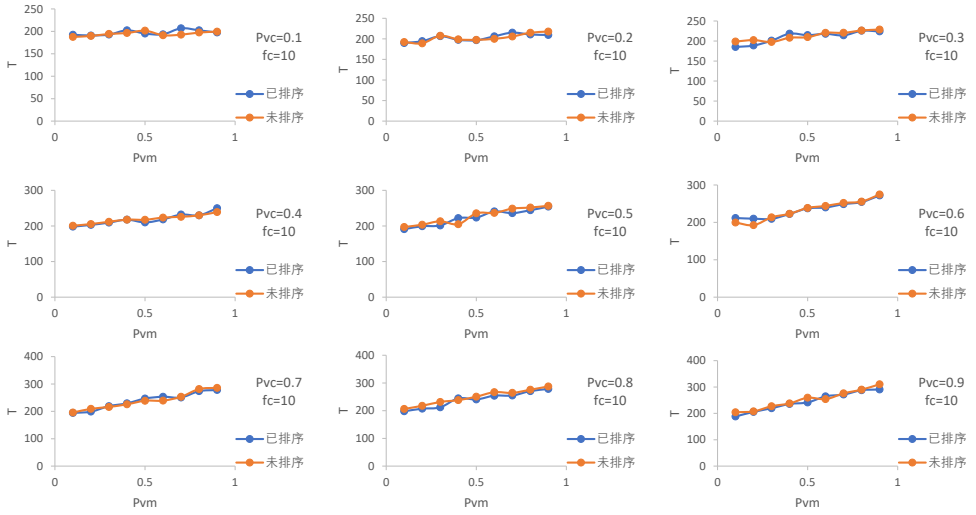


图 6-5: f_c=10，P_{vm} 变化时的仿真结果

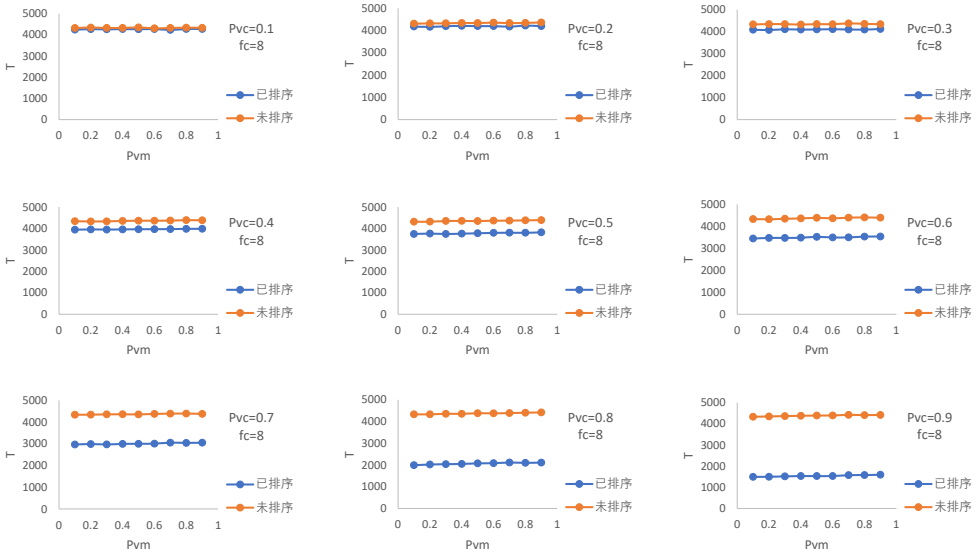


图 6-6: f_c=8，P_{vm} 变化时的仿真结果

生。在有限的服务器资源的情况下，这些频繁产生的提交将无法得到及时的处理，需要在等待队列中进行等待，只有当服务器资源可用时，才能够执行相应的任务。表 6-2展示了提交频率 f_c 与执行安全检测任务前的等待列队长度 L_{Q2} 的对应关系。当提交之间的平均间隔为 10min 时，即使是在有限的服务器资源下，这些提交也能够得到有效的处理，提交不会过多地在等待队列前堆积。正如图 6-1与 6-5显示的那样，T_{orig} 与 T_{prior} 的区别并不大。

但随着提交之间的时间间隔的缩短，在等待队列中堆积的提交数量将会陡然上升。但等待的队列长度并未呈现出明显的规律性，从结果来看，即便是小

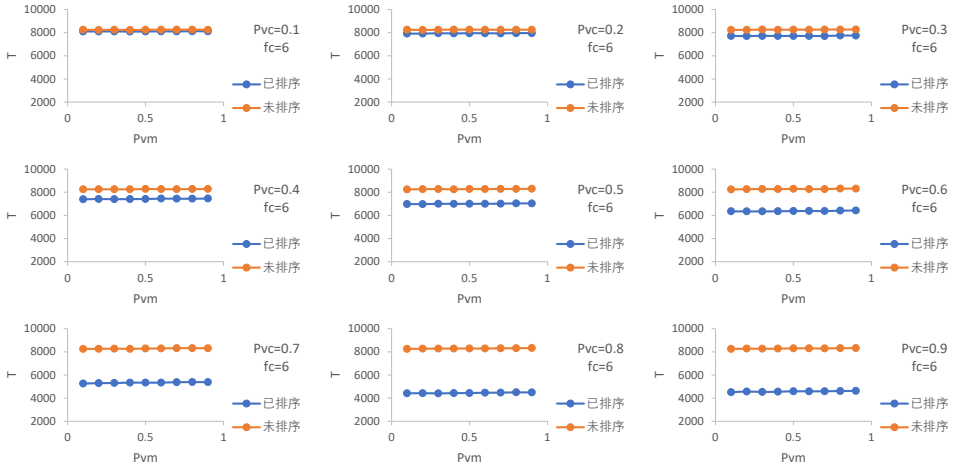


图 6-7: f_c=6，P_{vm} 变化时的仿真结果

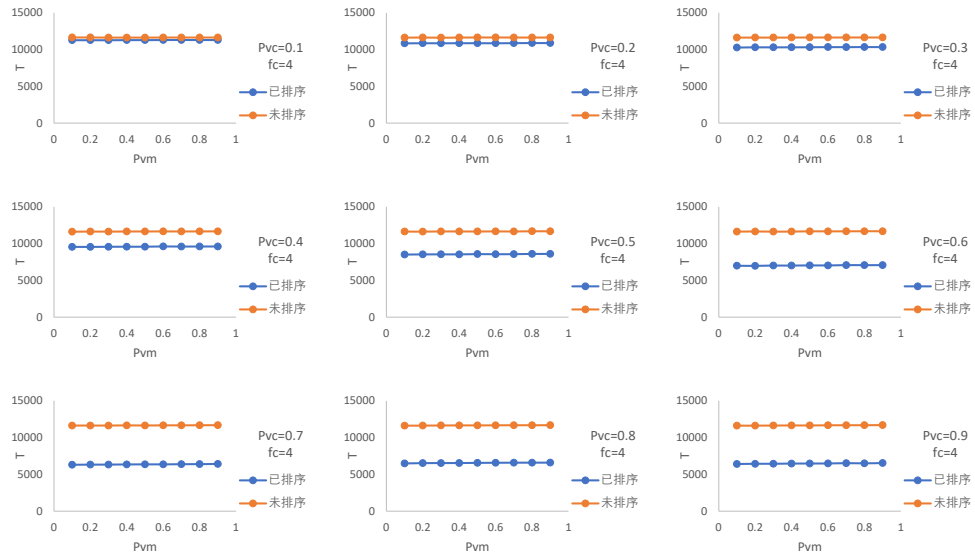


图 6-8: f_c=4，P_{vm} 变化时的仿真结果

幅度的频率提升，等待队列的长度也会成倍增长。当提交频率由 10min/次提高到 8min/次时，等待队列的长度会增长到 397，提高了近 26 倍。当平均提交间隔达到 4min 时，等待队列中的长度将会达到 1208，相较于 8min 的提交时间间隔，提高了近 3 倍。面对这种长度的提交堆积，将会严重影响到最终的提交平均工作流耗时 T 的数量级，造成 T_{orig} 与 T_{prior} 的显著上升。

表 6-2: 提交频率 f_c 与等待队列长度 L_{Q2} 对应关系表

提交频率	队列长度
10	15
8	397
6	670
4	1208

6.3.4 f_C、P_{VC}、P_{VM} 关系分析

由上述几节的分析结果可知，f_c 的提高会显著影响 T_{orig} 与 T_{prior} 的数量级。在不同提交频率下，T_{prior} 的数值会随着 P_{vc} 和 P_{vm} 的变化而变化，与 T_{orig} 呈现出一定的差异性，达到减少提交平均工作流时长 T 的目的。表6-3展示了在 f_c=10 时的 S_{rc} 的数值变化情况。从结果来看，在 f_c=10 时，优先级排序方法并未展现出很好的效果，最佳的相对节约时间占比 S_{rc} 为 0.084。考虑到这与等待队列的长度 L_{Q2} 密切相关，参照表6-2，当 f_c=10 时，L_{Q2} 的平均长度为 15。在短长度的队列中，如果含有漏洞的提交修复时长不是特别长的话，在此队列中进行提交的优先级排序意义不大，因为这些提交都会很快地获取到服务器资源执行接下来的任务。

在表6-3中的 S_{rc} 会存在一些负值，这说明提交优先级排序方法在次情形下并未节约提交的平均工作流时长，反而起到了相反的作用。这与预测模型的实际性能和仿真结果的随机性存在一定的关系。在动态仿真模型的实现中，提交的优先级排序依据的是预测模型对提交中漏洞情况的预测结果，这一结果与提交的实际情况可能存在一定的偏差。当等待队列中的提交数量较少时，错误的排序的负面效果可能会被放大，造成 S_{rc} 的数值为负数。另一方面，仿真模型的实际执行结果是由 200 次的模拟得到的平均值，这一平均值会在一定范围内小幅度波动。当 f_c=10 时，进入持续集成流水线的提交均能够得到及时的处理，因此提交的平均工作流耗时 T 不会过大。在这种情况下，模拟得到的平均值的波动现象将会被放大，在结果上就呈现出有正有负的 S_{rc} 属性值。

表6-4展示的是 f_c=8 时的 S_{rc} 的数值变化情况。此时的等待队列的平均长度 L_{Q2} 为 397，对等待队列中的提交进行优先级排序具备较好的效果。从结果来看，相对节约时间 S_{rc} 随着 P_{vc} 的增加而增加。当 P_{vc} 为 0.1 时，使用提交优先级排序方法产生的相对节约时间 S_{rc} 大致在 0.010 至 0.020 之间，最差的相对节约时间约为 0.008。这是由于 P_{vc} 为 0.1 时，提交漏洞率较小，在等待队列

表 6-3: f_c=10，S_{rc} 与 P_{vc} 和 P_{vm} 的关系表

f_c=10		P_{vm}								
		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
	0.1	-0.026	-0.005	0.008	-0.031	0.032	-0.010	-0.074	-0.025	0.010
	0.2	0.012	-0.029	0.006	0.005	0.005	-0.031	-0.045	0.020	0.038
	0.3	0.068	0.066	-0.017	-0.050	-0.024	0.011	0.029	0.003	-0.063
	0.4	0.013	0.012	0.011	-0.001	0.035	0.023	-0.028	-0.001	0.001
P_{vc}	0.5	0.031	0.021	0.056	-0.084	0.052	-0.018	0.051	0.030	-0.001
	0.6	-0.061	**-0.090**	0.018	0.003	0.005	0.017	0.013	0.005	-0.016
	0.7	0.007	0.048	-0.014	-0.017	-0.034	-0.059	0.010	0.027	0.012
	0.8	0.041	0.048	**0.084**	-0.027	0.040	0.048	0.037	0.019	-0.011
	0.9	0.078	0.010	0.033	0.008	0.069	-0.042	0.016	0.008	0.029

中仅有较少的提交含有安全漏洞，且这些提交不大可能会集中于等待队列的尾部。在对这些提交进行优先级排序时，相对节约时间的提升效果便会不明显。

随着 P_{vc} 的提高，使用提交优先级排序方法的效果逐渐明显，S_{rc} 也随之增大。P_{vc} 为 0.8 时，S_{rc} 就已经具备了较好的效果，能够达到 0.5 以上，当 P_{vc} 为 0.9 时，S_{rc} 的峰值达到了 0.656。在高提交漏洞率 P_{vc} 时，每当有提交进入等待队列时，排序模型将会按照对该提交的漏洞情况的预测值进行插入排序，将该提交插入到等待队列中的适当位置。由于 P_{vc} 下产生的含有漏洞的提交很多，按照可能的修复时间对这些提交进行排序则显得更有意义。需要更长修复时间的提交被前移到队列的前部，在有限仿真的时间内，将会有更多的提交会完成持续集成中的所有任务，提交的平均耗时也将会有所减少。

另一方面，P_{vm} 对 S_{rc} 的效果则影响不大。当 P_{vc} 一定时，无论 P_{vm} 如何变化，S_{rc} 的波动幅度并不大，也没有展现出明显的规律性。这与仿真过程存在一定的随机性有关。由于漏洞的修复时长符合高斯分布，而不是定值，这就意味着即使含有相同漏洞个数的提交，它们的修复时长也不一定会一样，甚至可能会出现含有漏洞个数较少的提交反而需要更长的修复时长的情况，但这种情况反而更加符合现实提交修复的情况。

表 6-5展示的是 f_c=6 时的 S_{rc} 的数值变化情况。此时的等待队列的平均长度 L_{Q2} 长达 670，对等待队列中的提交进行优先级排序具备不错的效果。但从结果来看，与 f_c=8 时的情形类似，相对节约时间 S_{rc} 总体随着 P_{vc} 的增加而增加。但当 P_{vc} 增长到 0.8 时，S_{rc} 将稳定在 0.45 左右，与 P_{vc} 为 0.9 时的 S_{rc} 数值相差不大，没有发生太大波动。

表 6-4: f_c=8，S_{rc} 与 P_{vc} 和 P_{vm} 的关系表

f_c=8		P_{vm}								
		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
	0.1	0.018	0.017	0.015	0.013	0.019	**0.008**	0.021	0.014	0.012
	0.2	0.030	0.036	0.030	0.030	0.032	0.035	0.037	0.028	0.037
	0.3	0.056	0.062	0.054	0.053	0.056	0.052	0.064	0.060	0.050
	0.4	0.092	0.086	0.089	0.091	0.092	0.091	0.091	0.092	0.091
P_{vc}	0.5	0.132	0.129	0.139	0.137	0.131	0.130	0.129	0.133	0.130
	0.6	0.204	0.196	0.201	0.202	0.198	0.200	0.204	0.198	0.195
	0.7	0.316	0.312	0.318	0.313	0.311	0.313	0.304	0.308	0.303
	0.8	0.541	0.534	0.532	0.530	0.527	0.525	0.519	0.524	0.522
	0.9	**0.656**	**0.656**	0.653	0.651	0.652	0.652	0.644	0.642	0.640

在此前情形下，P_{vc} 为 0.8 时，S_{rc} 达到了最好的时间成本节约方面的效果，峰值达到了 0.468。这意味着在持续集成中使用本文提出的提交优先级排序方法，理论上能够节约一半左右的提交平均工作流时长。但当 P_{vc} 为 0.9 时，S_{rc} 的这一数值仍然稳定在 0.45 左右。这说明，在 P_{vc} 为 0.8 和 P_{vc} 为 0.9 时使用提交优先级排序方法，带来的实际时间成本节约方面的收益相差无几。

这一情形与 f_c=8 时的状况不同。当 f_c=8 时，P_{vc} 由 0.8 增长到 0.9 时，S_{rc} 也能够大致提升 10% 左右，由 0.53 增长到 0.65 左右。究其原因，与等待队列的平均长度 L_{Q2} 密切相关。f_c=6 时的 L_{Q2} 长度达约为 f_c=8 时的两倍。当 P_{vc} 达到较高的比例时，等待队列中的漏洞提交也相应的有较高的占比。在次情形下，通过提交优先级排序方法仍然能够缩短整体的提交修复时长，但此时，提交在长队列中的等待时长将逐渐占据主导地位，在结果上才呈现出随着 P_{vc} 的提高，S_{rc} 逐渐趋于稳定。

表 6-6展示的是 f_c=4 时的 S_{rc} 的数值变化情况。此时的等待队列的平均长度 L_{Q2} 约为 1208，提交频率的提高意味着在仿真时间内，会有更多的提交进入到持续集成平台中。受限于有限的服务器资源，这些提交只能在等待队列中堆积。当 f_c=4 时，提交之间的平均间隔为 4 分钟，面对如此高频率的提交，进入持续集成平台中的提交也只能进行等待。如此一来，与 f_c=6 时的情形相似，提交的等待时长也将会随着 P_{vc} 的提高逐渐占据主导地位。

从结果上来看，与 f_c=8 和 f_c=6 时的情形类似，相对节约时间 S_{rc} 总体随着 P_{vc} 的增加而增加。但不同的是，在此情形下，随着 P_{vc} 的增长，S_{rc} 更早地趋于稳定。当 P_{vc} 增长到 0.7 时，S_{rc} 将稳定在 0.45 左右，随着 P_{vc} 增长到 0.8、

表 6-5: f_c=6，S_{rc} 与 P_{vc} 和 P_{vm} 的关系表

f_c=6		P_{vm}								
		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
	0.1	0.019	**0.016**	0.018	0.017	0.018	0.018	0.018	0.017	**0.016**
	0.2	0.041	0.036	0.037	0.039	0.038	0.037	0.038	0.036	0.036
	0.3	0.064	0.064	0.066	0.065	0.066	0.066	0.068	0.063	0.064
	0.4	0.104	0.102	0.104	0.103	0.105	0.100	0.099	0.101	0.100
P_{vc}	0.5	0.153	0.156	0.154	0.152	0.155	0.154	0.154	0.150	0.152
	0.6	0.230	0.232	0.234	0.231	0.231	0.228	0.229	0.229	0.227
	0.7	0.362	0.359	0.358	0.353	0.356	0.356	0.353	0.352	0.353
	0.8	0.465	0.465	**0.468**	0.465	0.464	0.461	0.461	0.458	0.461
	0.9	0.454	0.448	0.452	0.450	0.447	0.448	0.446	0.445	0.444

0.9 时，S_{rc} 数值仍然变化不大，也基本维持在 0.45 上下。

表 6-6: f_c=4，S_{rc} 与 P_{vc} 和 P_{vm} 的关系表

f_c=4		P_{vm}								
		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
	0.1	0.032	0.032	0.030	0.030	0.030	**0.029**	**0.029**	**0.029**	**0.029**
	0.2	0.067	0.066	0.066	0.066	0.067	0.067	0.066	0.064	0.065
	0.3	0.116	0.114	0.114	0.114	0.114	0.112	0.113	0.112	0.112
	0.4	0.178	0.180	0.178	0.178	0.178	0.175	0.176	0.176	0.176
P_{vc}	0.5	0.268	0.266	0.267	0.267	0.264	0.266	0.264	0.264	0.263
	0.6	0.399	0.401	0.397	0.397	0.397	0.397	0.394	0.395	0.394
	0.7	**0.458**	0.456	0.455	0.455	0.453	0.454	0.452	0.452	0.451
	0.8	0.442	0.439	0.439	0.439	0.437	0.436	0.436	0.435	0.436
	0.9	0.449	0.446	0.447	0.445	0.444	0.445	0.442	0.444	0.441

上述的四张表分别展示了在不同提交频率 f_c 下，S_{rc} 随着 P_{vc} 与 P_{vm} 的变化而变化的仿真结果。整体来看，f_c 对 S_{rc} 的影响程度最高。不同的 f_c 意味着不同的提交平均间隔，只有当 f_c 达到某一数值时，才会造成等待队列中提交的堆积。否则，进入持续集成平台中的提交将无需花费额外的时长等待服务器资源。如 f_c=10 时，使用提交优先级排序的方法效果并没有呈现出明显的效果，而随着提交频率的提高，当 f_c=8 时，等待队列中的提交数量猛然增长，提交优先级排序方法才会显示出它的有效性。

P_{vc} 对 S_{rc} 的影响次之。当 f_c 一定且达到某一数值时，此时等待队列中的长度也能够达到一定长度，如 f_c=8。S_{rc} 将会随着 P_{vc} 的增长而呈现出增长趋势。

只有进入流水线中的提交被预测为含有安全漏洞时，这些提交的顺序才会被重新安排。若 P_{vc} 为 0，也就是所有进入流水线的提交不含有安全漏洞，理论上来讲，若预测模型的表现优异，我们提出的排序方法将不会对这些提交进行优先级排序，S_{rc} 也将会趋于 0。但随着 P_{vc} 的提高，进入持续集成平台中的提交才会逐渐出现含有安全漏洞的情况，我们对这部分提交进行优先级排序，改变进入等待队列中的提交顺序，才能够实现平均提交工作流时长的减少。

P_{vm} 对 S_{rc} 的影响最弱。当 f_c 与 P_{vc} 一定时，P_{vm} 对 S_{rc} 的影响程度就显得不够，只能在较小的范围内震荡。P_{vm} 影响的是对含有安全漏洞的提交进行修复所花费的时长。在本文的仿真实验中，由于将提交的平均方法个数设定为符合均值为 7 的泊松分布，单个安全问题方法的修复时长也被设定为符合均值为 20，方差为 10 的高斯分布，其本身的漏洞修复时长就存在一定的随机性。当提交频率过高时，如 f_c=4，此时的漏洞修复时长将不在占据主导地位，在结果上 S_{rc} 才呈现出小范围的波动。

仿真实验的结果是人为设定了 4 种不同的提交频率 f_c、9 种不同的提交漏洞率 P_{vc} 和 9 种不同的漏洞方法占比 P_{vm} 下的实验，进一步地，为探究更为一般的上述三种参数对提交优先级排序的影响情况，我们对第三章中的公式 3-2进行了拓展。

公式 3-2展示的是给定情境下的 T_{prior} 耗时，值得一提的是，该公式展示的 T_{prior} 所描述的耗时为进行安全检测任务及相关安全漏洞修复所需要的总耗时，但并未加上进行编译测试任务所需要的耗时 $T_{compile}$。现在我们将场景一般化并考虑进行编译测试任务所需要的耗时，假设等待队列中的长度为 n, 经过提交优先级排序后的含有漏洞的提交也将会被置于队列的前部，且不会有新的提交再进入等待队列中。此时，可以得到如下公式：

$$T_{prior} = \frac{T_{build} + \sum_{i=1}^{n}\sum_{i=1}^{i} X_i + \sum_{j=1}^{m} max\left(0, Y_j - \sum_{i=j+1}^{n} X_i\right)}{n} \tag{6-2}$$

其中，$T_{compile}$ 为对进入流水线中的提交执行编译测试任务所需的总耗时，由于执行编译测试任务的耗时满足均值为 30，方差为 10 的高斯分布，整体来说，单个提交的编译及测试耗时相差不大。那么，影响 $T_{compile}$ 数值大小的即为执行编译测试任务的提交的数量。在不考虑提交跳过该任务的情形下，这一数值仅有 f_c 决定。即提交进入持续集成平台中的间隔越短，在规定时间内，进入流水线中执行相关任务的提交就越多。但受限于有限的服务器资源，提交只能在等待队列 *Q1* 处等待，在该处等待的提交按照“先进先出”的规则依次执行编译

测试任务。T_{orig} 与 T_{prior} 在计算实际总时耗时，均需要考虑到执行编译测试任务，由于在该处的提交也需要有一个等待的过程，因此该值会拉高提交的平均工作流耗时 T，但提交优先级排序方法并不能改变这一属性值的大小。

n 为等待队列中的提交的长度，该值取决于进入持续集成平台的提交频率 f_c。正如表6-2所展示的结果，f_c 为 10 时，此时的等待队列长度为 15，而当 f_c 为 4 时，等待队列的长度将达到 1207。等待队列的长度直接决定了 T_{prior} 的大小，进而直接影响到提交相对节约时间 S_{rc}。m 为等待队列中有安全漏洞的提交数量，该值满足等待队列中的提交总数和提交中有安全漏洞的比例的乘积，其公式如下：

$$m = P_{vc} \times n \tag{6-3}$$

由上式6-3可知，P_{vc} 决定了等待队列中有多少提交需要进行漏洞修复。但由于在等待队列中的提交需要重排序的缘故，后置的修复工作能够有所提前，并与等待队列中的剩余提交进行安全检测同步进行。故此，这部分的重复时间在计算提交平均工作流时长 T 时应当只计算一次，其数值为 0 与修复该提交中的安全漏洞与等待队列中剩余提交进行安全扫描的总时长之差的较大值。一般而言，由于修复工作的提前，公式6-2中的第二项将不会过大，从而实现减少提交平均工作流时长，提高集成效率和提交安全性的目的。

X_i 为第 i 次提交进行安全扫描所需的时长，Y_j 为修复第 j 次提交中所有安全漏洞所需的时耗，该值正比于本次提交中漏洞方法的个数占比 P_{vm}。从理论上来讲，本次提交中的漏洞方法数量越多，所需的修复时长就越长，Y_j 的数值也将越大。但由于在仿真过程中，提交中含有的方法数量有限，且对发现的安全漏洞修复工作迅速，故而，这一属性值不会过大。相较于 f_c 对等待队列长度、P_{vc} 对累计修复时长相比，P_{vm} 所影响的 Y_j 将不占据主导地位。

6.4 本章小结

本章基于 Graylog/Graylog2-server 这一开源项目的基础属性展开仿真实验，以探究本文所提出的面向安全的提交优先级排序方法的实际应用效果。同时，通过改变提交频率 f_c、提交漏洞率 P_{vc} 与漏洞方法占比 P_{vm} 的属性值，对总计 684 次不同的提交情况进行仿真实验，累计仿真总次数为 136,800（684 × 200）次。仿真完成后，展示了使用面向安全的提交排序方法所带来的提交相

对节约时间 S_{rc} 的结果，并从提交频率 f_c、提交漏洞率 P_{vc}、漏洞方法占比 P_{vm} 及三者的相互关系对仿真结果进行了分析。结果显示，提交频率 f_c 对相对节约时间 S_{rc} 的影响最大，提交漏洞率 P_{vc} 的影响次之。当 f_c=8 时，S_{rc} 最高能够节约 65.6% 的提交工作流时长。

第七章　总结与未来展望

7.1　全文总结

持续集成作为现代软件开发模式中的一种典型实践，已广泛应用于国内外各大企业的实际开发过程中。在持续集成实践中，开发人员需要高频率提交自己开发完成的代码，切实提高了软件的开发和构建频率，加速了价值的交付过程。但随着近些年来软件安全问题的日益严峻，在持续集成过程中进行快速高效的安全代码控制，保障集成的代码的安全性也逐渐成为各大软件企业重点考虑的解决方案之一，并逐渐在实践中落地。

相较于先前简单的持续集成过程，加入安全检测后能够切实提高了集成的代码的安全性，但也势必会对代码的集成效率造成影响。面对愈发频繁的代码提交和构建，受限于有限的服务器资源，提交只能在等待队列中等待服务器资源的分配。若存在安全漏洞的提交集中于等待队列的后半段，那么提交中的安全漏洞也会越晚暴露，无疑会拉长项目的整体开发时长，拖慢集成进度。

在此背景下，为了提前漏洞代码的修复时机，进而缩短整体的构建时长，针对在等待队列中等待服务器资源分配的提交，本文提出了一种面向代码安全的提交优先级排序方法。该方法能够对提交中代码的安全状况做出预测，并基于预测的结果实现对提交的优先级排序。

本文的工作主要分为两个部分，一是基于 BERT 的安全漏洞预测模型，二是持续集成过程仿真建模。

基于 BERT 的安全漏洞预测模型能够针对方法级别的代码片段，就其安全性做出预测，它是面向代码安全的提交优先级排序方法的核心。在这部分工作中，本文选取了 Juliet 测试套件和 OWASP Benchmark 两个不同的开源数据集进行研究。在筛选出原始的数据集中符合一定标准的测试代码后，针对这部分的测试代码，本文提取出它们的抽象语法树信息作为原始的语料信息。基于深度学习技术，成功训练出性能较好的分类模型，并于其余分类器进行了结果比对和相关分析。而后，本文以此分类模型为核心，遵循提出的排序规则，构建了提交优先级排序原型，为实际持续集成场景中进行提交优先级排序提供了有效

的解决方案。

为了进一步验证本文所提出的排序方法的有效性，并探究该排序方法的实际效果受不同外在因素的影响程度，本文基于离散事件的软件过程仿真技术，展开了持续集成过程仿真建模的工作。在这部分工作中，本文分析了 Github 与 Travis CI 中的实际持续集成工作，并对这一过程进行了抽象，构建出动态仿真模型。而后选取了 Travis CI 中的一个实际项目作为案例，通过输入不同的参数值大小，成功对不同场景下的实际持续集成过程进行了模拟。通过横向比对应用排序方法前后的提交耗时，从理论上保障了本文所提出的排序方法的有效性，为排序方法的实际应用奠定了基础。在后续的分析过程中也指出了影响排序方法有效性的一些关键性因素，具备较强现实的参考意义。

7.2 未来展望

本文通过多组实验设计，对安全漏洞预测模型的性能及面向代码安全的提交优先级排序的效果均进行了严密验证，保障了提交排序方法从提出到验证整个过程的逻辑连贯性。但提交优先级排序方法的提出需要依赖很多外在的假设条件，且考虑到的排序场景较为单一，仍可进行更深层次的探究，以增强本文所提出的排序方法的实际应用能力。未来计划进行如下改进：

1. 本文能够进行提交优先级排序的前提是建立在提交相互独立这一基本假设的基础上。若提交之间存在依赖关系，必须按照特定的先后顺序执行构建任务，那么对于这部分提交将无法使用优先级排序方法。未来可以将提交之间的依赖关系纳入提交优先级排序方法的考量范围，针对特定类型的提交，采用针对性的处理办法。
2. 本文依据提交中代码所含存在安全漏洞的方法个数大小进行提交的优先级排序，排序依据较为单一。未来可以考虑使用多项排序规则，如提交的优先级、提交的等待时间阈值等，打造出适用于多种场景下的提交优先级排序方法，增强排序方法的实际应用能力。
3. 在对方法粒度的代码片段进行安全预测时，预测模型将函数方法视为独立的代码片段进行安全性预测，这种作法会忽视很多能够提供安全性预测的其他因素。在未来需要对预测模型进行更深入的研究，从多个方面提取信息，为提交的安全性提供更高性能指标的预测。
4. 在训练安全漏洞预测模型时，本文选取了两个标准开源的数据集。考虑到

实际工业项目中的代码与人造数据集之间可能存在较大差异，因此，在实际应用提交优先级排序方法时，需要重新进行适应性的训练。

5. 本文使用基于离散事件的软件过程仿真技术对排序方法的有效性进行了验证，尽管很多参数是从实际开源项目的持续集成过程中获取，但不同的项目之间的参数差异可能很大，也不一定会服从某种特定分布。在实际工业场景中应用时，提交的代码也会存在着巨大的不确定性。因此，提交优先级排序算法的实际性能究竟如何仍然需要结合具体的工业场景进一步探究。

致　　谢

时光飞逝，恍惚间，又一年毕业季，漫漫求学之路在这个夏天也即将迎来一个阶段性的句点。回顾这三年来所经历的点滴，可以说是夹杂着鲜花与荆棘。值此之际，我想与过去好好道个别，并向成长路上帮助过我的父母、老师、同学、朋友们表示衷心的感谢。

感谢我的导师张贺教授。三年前来到南京大学软件学院，是张老师带领我走进了学术的大门。组会上评审过的一篇篇论文，周会上一次次讨论的研究进展，无不显示出张老师在科研方面的严格要求。正是这份严谨的治学，再加上张老师积极拓展其他领域的创新精神，才能让实验室取得诸多突破性成果，并逐渐发展壮大。

感谢张津睿学长。他是带领我走进计算机相关专业的引路人，正是在他的建议下，我从一个门外汉顺利进入到软件工程领域，开启了研究生阶段的新生活。尽管本科毕业后的你，就凭借惊人的天赋直接进入到互联网企业发展，但也渐渐被无情的工作夺取了头发，但是我相信我的还可以再坚挺很多很多年。

感谢我的几位舍友黄璜、黄迪璇和胡文。黄璜同学和我是舍友，也和我属于同一个实验室。科研路上，他是一个冉冉升起的学术新星，三年光景，已有 7 篇论文，我的毕设选题也有幸受到了他的帮助。本该在学术道路上顺风顺水的他却选择了激流勇退，毅然决然去体制内，换了一种方式继续为国燃烧自己的生命，或许这才是真正的智慧吧，瑞斯拜。黄迪璇同学和胡文同学是两年制的专业硕士，他们俩早我一年毕业，后来也都去了国内互联网大厂，是真正的行业巨佬。在两年的同宿舍期间，他们会从自身的经历对我的技术成长之路进行建议，指导我进行相关技术的学习。遇到不懂的问题，也大可直接询问并展开讨论，他们在技术上的耕耘和成果令人钦羡。

感谢毛润丰博士。进入实验室课题组以来，我一直是与毛润丰博士合作进行 DevOps 安全相关的研究工作。他是与我同一年进入实验室的直博生，和我年龄相仿。但他已然具备了博士生所需的一切品质，对特定领域的科研话题有着自己的真知灼见，对学术论文写作有着自己的精巧构思。感恩我们先前一同文献调研科研的日子，也感谢你对我的毕设呕心沥血。未来路漫漫，祝毛博士

早日毕业。

感谢刘博涵博士、杨岚心博士和殷慧琳师妹。在进行实验的过程中难免会碰到这样或那样的问题，正是在你们的帮助下，我的毕业论文工作才能顺利展开，是你们为我提供了新颖的解决思路，在此万分感谢。

在去年暑假的时候，有幸在阿里巴巴实习了两个月的时间。两个月的实习时光十分短暂，第一个月一直在吃吃喝喝，每个星期都要出去喝一次酒，期间还去了一次千岛湖团建。这就直接导致了第二个月的焦头烂额。不过庆幸在阿里也认识了很多技术好、热心肠的伙伴，灵均、允布、北天、乐皓、一肖等等，当然也有具备超强领导力又很会 push 人的哈哥哈赤。特别感谢我的指导师兄六极，谢谢你对我提出的任何问题的倾囊相授，对我的任何疑虑都倾情解答。

感谢实验室中相识的每一位同学和生命中每一位不期而遇的小伙伴，是你们的出现才能让我的生活充满色彩，与你们的交流才是构成生活最基础的底色。

感谢我的父母，我的父母都是普通的打工人，但他们并不会干预我所作的任何决定，会尽他们所能去帮助我。由衷地感谢二位一直以来的默默付出，你们真的辛苦了，一定要多注意休息才行。

当然还要感谢自己。感谢自己的选择，让自己有了更为丰富的一段旅程。而现在，这段旅程即将告一段落，那就收拾好行囊，轻轻说声再见吧。路还剩长，而我一直在路上。

参考文献

[1] LIANG J, ELBAUM S, ROTHERMEL G. Redefining prioritization: continuous prioritization for continuous integration[C] // Proceedings of the 40th International Conference on Software Engineering. 2018 : 688 – 698.

[2] COHEN D, LINDVALL M, COSTA P. An introduction to agile methods.[J]. Adv. Comput., 2004, 62(03) : 1 – 66.

[3] SCHWABER K, BEEDLE M. Agile software development with Scrum : Vol 1[M]. [S.l.] : Prentice Hall Upper Saddle River, 2002.

[4] BECK K. Extreme programming explained: embrace change[M]. [S.l.] : addison-wesley professional, 2000.

[5] LWAKATARE L E, KUVAJA P, OIVO M. Dimensions of devops[C] // International conference on agile software development. 2015 : 212 – 217.

[6] SWARTOUT P. Continuous delivery and DevOps: a quickstart guide[M]. [S.l.] : Packt Publishing Ltd, 2012.

[7] ANON. 2018 State of Devops Report[R]. [S.l.] : Puppet, 2018.

[8] LAUKKANEN E, ITKONEN J, LASSENIUS C. Problems, causes and solutions when adopting continuous delivery—A systematic literature review[J]. Information and Software Technology, 2017, 82 : 55 – 79.

[9] 刘博涵, 张贺, 董黎明. DevOps 中国调查研究 [J]. 软件学报, 2019, 10.

[10] MYRBAKKEN H, COLOMO-PALACIOS R. DevSecOps: a multivocal literature review[C] // International Conference on Software Process Improvement and Capability Determination. 2017 : 17 – 29.

[11] MAO R, ZHANG H, DAI Q, et al. Preliminary Findings about DevSecOps from Grey Literature[C] // 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS). 2020 : 450 – 457.

[12] 戴启铭, 毛润丰, 黄璜, et al. DevSecOps:DevOps 下实现持续安全的实践探索[J]. 软件学报, 2021, 5.

[13] ANON. 2019 State of Devops Report[R]. [S.l.] : Puppet, 2019.

[14] BOLDUC C. Lessons learned: Using a static analysis tool within a continuous integration system[C] // 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). 2016 : 37 – 40.

[15] ELBAUM S, ROTHERMEL G, PENIX J. Techniques for improving regression testing in continuous integration development environments[C] // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014 : 235 – 245.

[16] SPIEKER H, GOTLIEB A, MARIJAN D, et al. Reinforcement learning for automatic test case prioritization and selection in continuous integration[C] // Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2017 : 12 – 22.

[17] HAGHIGHATKHAH A, MÄNTYLÄ M, OIVO M, et al. Test prioritization in continuous integration environments[J]. Journal of Systems and Software, 2018, 146 : 80 – 98.

[18] ZIMMERMANN T, NAGAPPAN N, WILLIAMS L. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista[C] // 2010 Third International Conference on Software Testing, Verification and Validation. 2010 : 421 – 428.

[19] CZERWONKA J, NAGAPPAN N, SCHULTE W, et al. Codemine: Building a software development data analytics platform at microsoft[J]. IEEE software, 2013, 30(4) : 64 – 71.

[20] BREIMAN L. Random forests[J]. Machine learning, 2001, 45(1) : 5 – 32.

[21] SHIN Y, MENEELY A, WILLIAMS L, et al. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities[J]. IEEE transactions on software engineering, 2010, 37(6): 772 – 787.

[22] BALAKRISHNAMA S, GANAPATHIRAJU A. Linear discriminant analysis-a brief tutorial[J]. Institute for Signal and information Processing, 1998, 18(1998): 1 – 8.

[23] FRIEDMAN N, GEIGER D, GOLDSZMIDT M. Bayesian network classifiers[J]. Machine learning, 1997, 29(2): 131 – 163.

[24] CHOWDHURY I, ZULKERNINE M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities[J]. Journal of Systems Architecture, 2011, 57(3): 294 – 313.

[25] WALDEN J, STUCKMAN J, SCANDARIATO R. Predicting vulnerable components: Software metrics vs text mining[C] // 2014 IEEE 25th international symposium on software reliability engineering. 2014: 23 – 33.

[26] ZHANG Y, JIN R, ZHOU Z-H. Understanding bag-of-words model: a statistical framework[J]. International Journal of Machine Learning and Cybernetics, 2010, 1(1-4): 43 – 52.

[27] YAMAGUCHI F, GOLDE N, ARP D, et al. Modeling and discovering vulnerabilities with code property graphs[C] // 2014 IEEE Symposium on Security and Privacy. 2014: 590 – 604.

[28] SZE V, CHEN Y-H, YANG T-J, et al. Efficient processing of deep neural networks: A tutorial and survey[J]. Proceedings of the IEEE, 2017, 105(12): 2295 – 2329.

[29] HINDLE A, BARR E T, GABEL M, et al. On the naturalness of software[J]. Communications of the ACM, 2016, 59(5): 122 – 131.

[30] LI Z, ZOU D, XU S, et al. Vuldeepecker: A deep learning-based system for vulnerability detection[J]. arXiv preprint arXiv:1801.01681, 2018.

[31] WOLF L, HANANI Y, BAR K, et al. Joint word2vec Networks for Bilingual Semantic Representations.[J]. Int. J. Comput. Linguistics Appl., 2014, 5(1): 27–42.

[32] WEISER M. Program slicing[J]. IEEE Transactions on software engineering, 1984(4): 352–357.

[33] LIN G, ZHANG J, LUO W, et al. Cross-project transfer representation learning for vulnerable function discovery[J]. IEEE Transactions on Industrial Informatics, 2018, 14(7): 3289–3297.

[34] KELLNER M I, MADACHY R J, RAFFO D M. Software process simulation modeling: why? what? how?[J]. Journal of Systems and Software, 1999, 46(2-3): 91–105.

[35] ZHANG H, KITCHENHAM B, PFAHL D. Reflections on 10 years of software process simulation modeling: A systematic review[C] // International Conference on Software Process. 2008: 345–356.

[36] ZHANG H, KITCHENHAM B, PFAHL D. Software process simulation modeling: an extended systematic review[C] // International Conference on Software Process. 2010: 309–320.

[37] ZHANG H, JEFFERY R, HOUSTON D, et al. Impact of process simulation on software practice: An initial report[C] // Proceedings of the 33rd international conference on Software engineering. 2011: 1046–1056.

[38] GAROUSI V, PFAHL D. When to automate software testing? A decision-support approach based on process simulation[J]. Journal of Software: Evolution and Process, 2016, 28(4): 272–285.

[39] WAGNER A, SAMETINGER J. Using the Juliet test suite to compare static security scanners[C] // 2014 11th International Conference on Security and Cryptography (SECRYPT). 2014: 1–9.

[40] LEE Y J, CHOI S-H, KIM C, et al. Learning binary code with deep learning to detect software weakness[C] // KSII The 9th International Conference on Internet (ICONI) 2017 Symposium. 2017.

[41] AMANKWAH R, CHEN J, AMPONSAH A A, et al. Fast Bug Detection Algorithm for Identifying Potential Vulnerabilities in Juliet Test Cases[C] // 2020 IEEE 8th International Conference on Smart City and Informatization (iSCI). 2020 : 89 – 94.

[42] PATIL S S. Automated Vulnerability Detection in Java Source Code using J-CPG and Graph Neural Network[D]. [S.l.] : University of Twente, 2021.

[43] ALON U, ZILBERSTEIN M, LEVY O, et al. A general path-based representation for predicting program properties[J]. ACM SIGPLAN Notices, 2018, 53(4) : 404 – 419.

[44] BIELIK P, RAYCHEV V, VECHEV M. PHOG: probabilistic model for code[C] // International Conference on Machine Learning. 2016 : 2933 – 2942.

[45] ALON U, ZILBERSTEIN M, LEVY O, et al. code2vec: Learning distributed representations of code[J]. Proceedings of the ACM on Programming Languages, 2019, 3(POPL) : 1 – 29.

[46] DEVLIN J, CHANG M-W, LEE K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding[J]. arXiv preprint arXiv:1810.04805, 2018.

[47] GAO Y, CHEN L, SHI G, et al. A Comprehensive Detection of Memory Corruption Vulnerabilities for C/C++ Programs[C] // 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom). 2018 : 354 – 360.

[48] KOC U, WEI S, FOSTER J S, et al. An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool[C] // 2019 12th ieee conference on software testing, validation and verification (icst). 2019 : 288 – 299.

[49] KITCHENHAM B A, PICKARD L, LINKMAN S, et al. A framework for evaluating a software bidding model[J]. Information and Software Technology, 2005, 47(11) : 747 – 760.

[50] GONG H, ZHANG H, YU D, et al. A systematic map on verifying and validating software process simulation models[C] // Proceedings of the 2017 International Conference on Software and System Process. 2017 : 50 – 59.

[51] MASSEY JR F J. The Kolmogorov-Smirnov test for goodness of fit[J]. Journal of the American statistical Association, 1951, 46(253) : 68 – 78.

[52] LIU B, ZHANG H, YANG L, et al. An experimental evaluation of imbalanced learning and time-series validation in the context of CI/CD prediction[G] // Proceedings of the Evaluation and Assessment in Software Engineering. 2020 : 21 – 30.

[53] MILLER A. A hundred days of continuous integration[C] // Agile 2008 conference. 2008 : 289 – 293.

[54] YUKSEL H M, TUZUN E, GELIRLI E, et al. Using continuous integration and automated test techniques for a robust C4ISR system[C] // 2009 24th International Symposium on Computer and Information Sciences. 2009 : 743 – 748.

[55] SEO H, SADOWSKI C, ELBAUM S, et al. Programmers' build errors: a case study (at google)[C] // Proceedings of the 36th International Conference on Software Engineering. 2014 : 724 – 734.

[56] MORRISON P J, PANDITA R, XIAO X, et al. Are vulnerabilities discovered and resolved like other defects?[J]. Empirical Software Engineering, 2018, 23(3) : 1383 – 1421.

附录 A　动态仿真模型模块对应表

表 A-1: 动态仿真模型模块对应表

模块	模块名称	内容描述
B01	InputCommit	随机创建提交进入流程，产生提交的间隔服从泊松分布
B02	SetType	为创建的提交设置基本属性 B03、B04、B05、B06
B03	CommitType	按比例赋予本次提交类型为 PullRequest（PR）或者 Push
B04	Compile&Test Result	按比例赋予本次提交编译测试任务的结果为失败或者通过
B05	Vulnerability Result	按比例赋予本次提交安全描扫任务的结果为有漏洞或者无漏洞
B06	TotalMethodNum	随即赋予本次提交所包含的方法总个数，服从高斯分布
B07	Diversion By Whether have Vuls	根据是否含有漏洞对提交进行分类，不含有漏洞的提交进入 B08，否则进入 B09
B08	Set No Vuls	为不含有漏洞的提交赋值实际漏洞方法数量属性，数值为 0
B09	Set Vuls Numbers	为含有漏洞的提交赋值实际漏洞方法数量属性，数值大小由 B06 和 B32 共同决定
B10	Vuls Ratio	本次提交中漏洞方法占总方法个数的占比
B11	Merge-1	合并是否含有漏洞两种类型的提交
B12	Diversion By CommitType	根据提交类型进行区分，PR 进入 B13，Push 进入 B14
B13	SkipPR	按比例设置本次 PR 类型的提交是否跳过构建任务，跳过进入 B16，否则进入 B15
B14	SkipPush	按比例设置本次 Push 类型的提交是否跳过构建任务，跳过进入 B16，否则进入 B15
B15	Merge-2	合并未跳过编译测试任务的所有提交
B16	Merge-3	合并跳过编译测试任务的所有提交
B17	Diversion by Whether to batch	根据 B39 是否为空决定是否需要批量构建，为空进入 B18，否则进入 B40

续表

模块	模块名称	内容描述
B18	Set Predictive Vul Numbers	对提交进行安全漏洞方法预测，预测值由 B06、B10、B19、B20 共同决定
B19	FP Rate	模型将未含有安全漏洞的方法错误地预测为含有安全漏洞方法的概率
B20	TN Rate	模型正确将出含有漏洞的方法预测为含有漏洞的方法的概率
B21	Queue-1	等待队列，暂存等待服务器资源执行编译测试任务的所有提交
B22	ServerResource-1	用于执行编译测试任务的服务器资源
B23	Set Compile&Test Delay	计算编译测试任务的耗时
B24	PRPassedDelay	成功执行 PR 类型的提交所需要的时间，服从均匀分布
B25	PRFailedDelay	失败执行 PR 类型的提交所需要的时间，服从均匀分布
B26	PushPassedDelay	成功执行 Push 类型的提交所需要的时间，服从均匀分布
B27	PushFailedDelay	失败执行 Push 类型的提交所需要的时间，服从均匀分布
B28	Execute Compile&Test	执行编译测试任务
B29	Release-1	释放用于执行编译测试任务的服务器资源
B30	Diversion By Compile&Test Result	根据编译测试任务的执行结果进行分类，成功的进入 B38，否则进入 B31
B31	Merge-4	合并需要进行缺陷修复的提交，即未能通过编译测试任务的提交
B32	Diversion By CommitType	根据提交的类别对未能通过编译测试的提交进行区分，PR 进入 B33，Push 进入 B35
B33	Execute FixPR	执行 PR 类型提交的故障修复
B34	FixPR Delay	修复 PR 类型提交的故障所需的时间，服从均匀分布源
B35	Execute FxiPush	执行 Push 类型提交的故障修复
B36	FixPush Delay	修复 Push 类型提交的故障所需的时间，服从均匀分布
B37	Merge-5	合并需要进行故障修复的提交，即未能通过编译测试任务的提交
B38	Merge-6	合并经历编译测试任务的所有的提交
B39	Queue-2	等待队列，存放先前跳过执行构建任务的所有提交进行批量构建

续表

模块	模块名称	内容描述
B40	Batch	将 B39 中的所有提交与 B17 中出列的提交进行捆绑
B41	Queue-3	等待队列，存放在等待服务器资源的 B40 中完成捆绑的批量提交
B42	Set Delay	计算批量构建任务的耗时
B43	Execute CI	执行批量构建任务
B44	Release-2	释放服务器资源
B45	UnBatch	批量构建任务执行完毕后，回复各个提交的原始状态
B46	Diversion by Result from Queue-1	针对从 B17 中出列执行批量构建任务的提交，根据编译测试任务的结果进行区分，通过的进入 B48，否则进入 B31
B47	Diversion by Result from Queue-2	针对从 B39 中出列执行批量构建任务的提交，根据编译测试任务的结果进行区分，通过的进入 B48，否则进入 B31
B48	Merge-7	合并通过编译测试任务的提交
B49	Queue-4	等待队列，暂存等待服务器资源执行安全检测任务的所有提交
B50	Server Resource-2	用于执行安全检测任务的服务器资源
B51	Execute Security Scanning	执行安全检测任务
B52	Security Scanning Delay	进行安全检测所需的时间，服从均匀分布
B53	Get Vuls Nulmber	获取实际漏洞方法数量属性
B54	Diversion By Vuls Number	根据提交的实际漏洞数量进行分类，若实际安全漏洞数量大于 0，进入 B55，否则进入 B56
B55	Has Vuls	使得含有安全漏洞的提交通过
B56	Has No Vuls	使得未含有安全漏洞的提交通过
B57	Set Fix Vuls Delay	计算修复安全漏洞的时间
B58	Fix Vuls Delay	修复每个安全漏洞的平均时间，服从均匀分布
B59	Execute Fix Vuls	执行安全漏洞修复
B60	Merge-8	合并所有提交

续表

模块	模块名称	内容描述
B61	Exit	结束整个工作流

附录 B　DES 仿真模型图

图 B-1和图 B-2展示了本文构建的 DES 动态仿真模型，图中每一个模块的具体描述均可参见表 A-1。

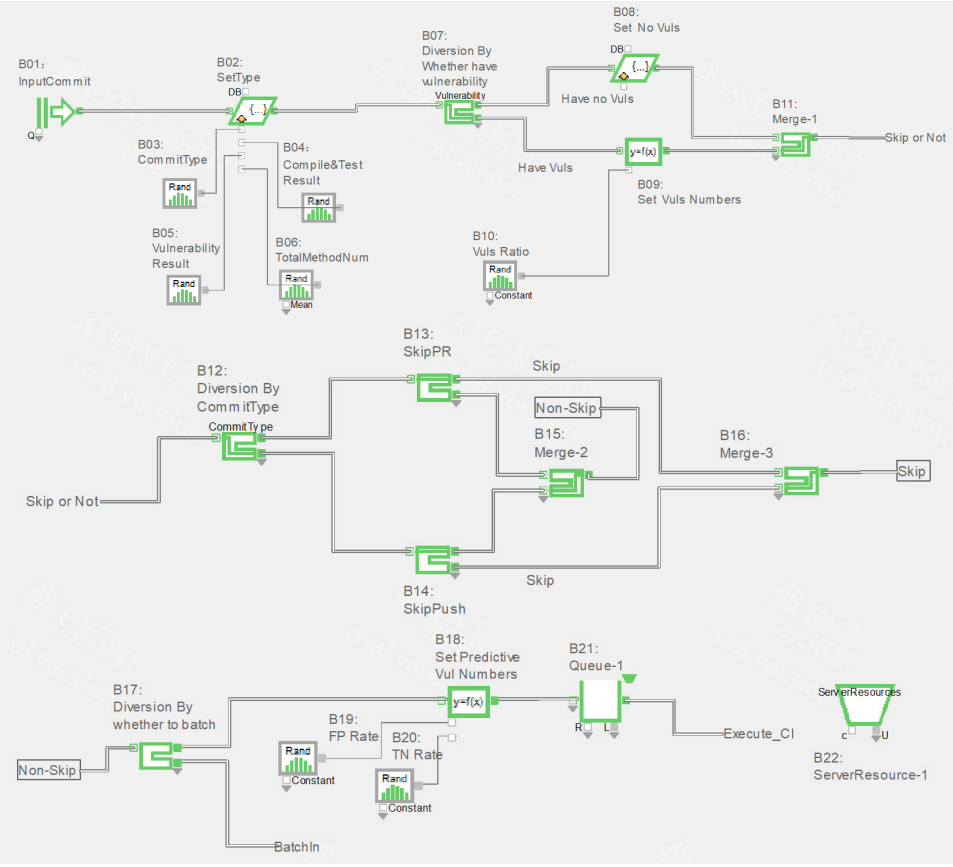


图 B-1: DES 仿真模型图（一）

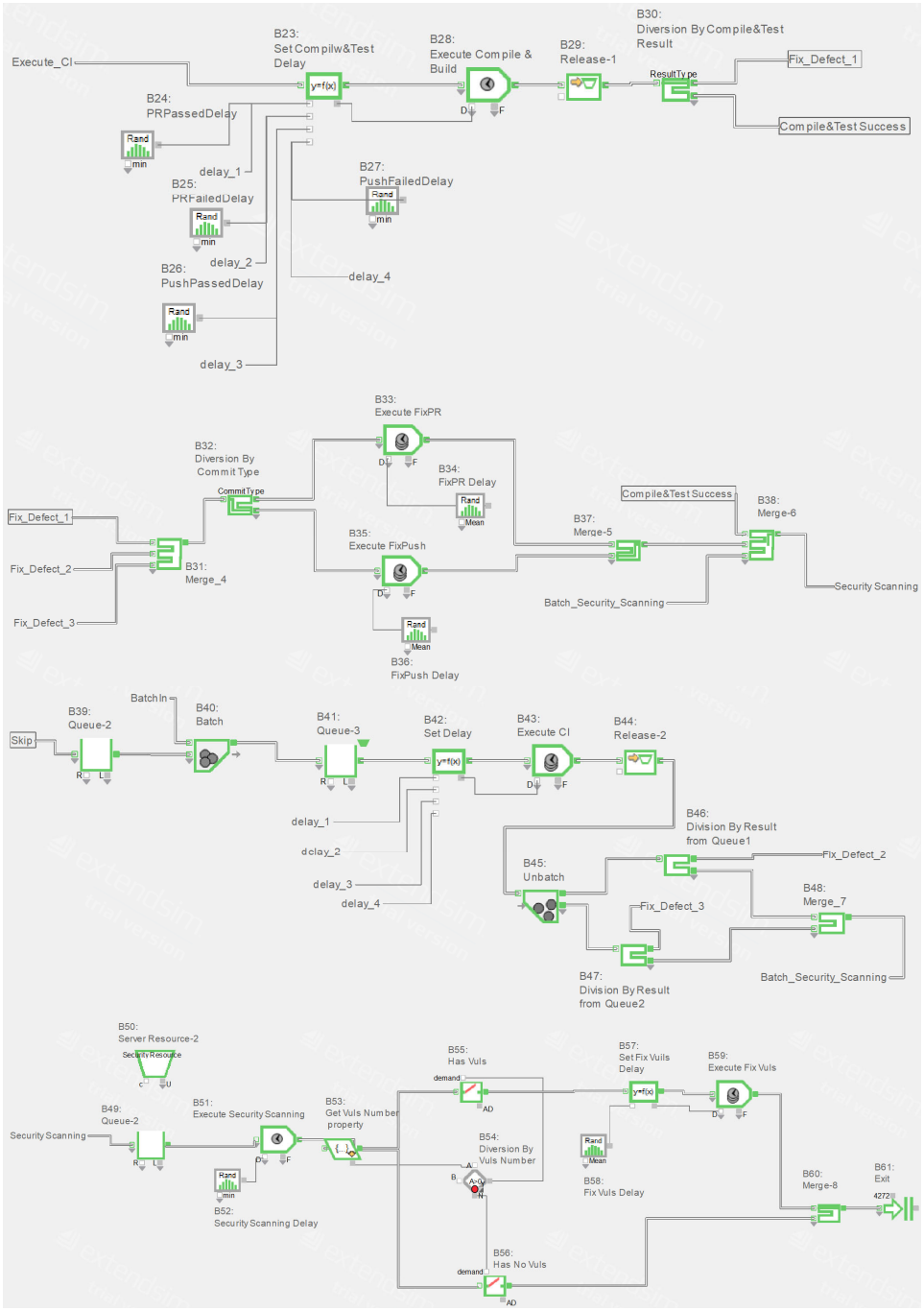


图 B-2: DES 仿真模型图（二）

简历与科研成果

基本信息

戴启铭，男，汉族，1996 年 10 月出生，江苏省泰州人。

教育背景

2018 年 9 月 — 2021 年 6 月 南京大学软件学院 硕士

2014 年 9 月 — 2018 年 6 月 南京航空航天大学能源与动力学院 本科

攻读工学硕士学位期间完成的学术成果

1. **戴启铭**, 毛润丰, 黄璜, 荣国平, 沈海峰, 邵栋, “DevSecOps:DevOps 下实现持续安全的实践探索,” 软件学报,0,(0):0.
2. Runfeng Mao, He Zhang, **Qiming Dai**, Huang Huang, Guoping Rong, Haifeng Shen, Lianping Chen, Kaixiang Lu, “Preliminary Findings about DevSecOps from Grey Literature,” in *Proc. 20th International Conference on Software Quality, Reliability and Security (QRS) 2020*, July. 2020.

《学位论文出版授权书》

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名：_____

_____年_____月_____日

论文题名	持续集成下面向代码安全的提交优先级排序方法研究				
研究生学号	MG1832001	所在院系	软件学院	学位年度	2021
论文级别	☐ 硕士　☐ 硕士专业学位 ☐ 博士　☐ 博士专业学位 （请在方框内画勾）				
作者 Email	MG1832001@smail.nju.edu.cn				
导师姓名	张贺 教授				

论文涉密情况：

☐ 不保密

☐ 保密，保密期 (_____年_____月_____日 至 _____年_____月_____日)

注：请将该授权书填写后装订在学位论文最后一页（南大封面）。