面向分布式软件的可扩展且高性价比数据流分析：算法与应用程序

arXiv:2303.03659v1 [cs.DC] 7 Mar 2023

By

Xiaoqin Fu

为部分满足以下学位要求而提交的博士论文

哲学博士

华盛顿州立大学 电气工程与计算机科学学院

2022年7月

©版权归 XIAOQIN FU 所有，2022。
保留所有权利

©版权归 Xiaoqin Fu 所有，2022。
保留所有权利

致华盛顿州立大学全体教职员工:

受命审查XIAOQIN FU博士论文的委员会成员认为该论文符合要求，并建议予以通过。

Haipeng Cai，博士，主席

党哲，博士

陶定文，博士

致谢

我想借此机会，向所有帮助我顺利完成博士学位的人表达最深切的感激与谢意。

首先，我谨向我的导师兼委员会主席 Haipeng Cai 博士致以最诚挚的谢意。在我于华盛顿州立大学攻读计算机科学博士学位的整个过程中，他始终给予我支持。在我的博士学习期间，他不断鼓励我，在艰难时刻成为我坚实的后盾。若没有他的关怀、帮助与支持，我绝不可能完成学业。他是一位真正的导师，也成为了我真诚的朋友。

我非常感谢党哲博士在这段求学旅程中给予我的帮助。在过去四年里，我从他身上学到了如何成为一名成功的科学家。他将成为我今后研究和职业生涯中的榜样。我还要感谢我的委员会成员陶定文博士，感谢他同意与我一同走过这段旅程。在整个过程中，他慷慨地分享自己的专业知识，并付出了宝贵的时间。

我还要感谢华盛顿州立大学电气工程与计算机科学学院的其他教职员工，在整个过程中给予我的支持与鼓励。学院院长 Partha Pratim Pande 教授为人非常友善，也给予我许多支持。Noel Schulz 博士、Venera Arnaoudova 博士、Sakire Arslan Ay 博士、Kung-Chi Wang 博士、David Bakken 博士、Janardhan Rao Doppa 博士和 Assefaw Gebremedhin 博士都把我当作朋友和同事。我尤其要感谢 Jessica Cross，从我们初次见面那天起，她就一直鼓励我、为我提出建议，并给予我无条件的支持。特别感谢莫纳什大学的 Li Li 博士，他慷慨地为我的研究和职业发展提供帮助。

最后，我要感谢美国国家科学基金会（NSF）提供编号为 CCF-1936522 的资助，支持我开展博士研究。

面向分布式软件的可扩展且高性价比数据流分析：算法与应用

摘要

作者：Xiaoqin Fu，博士，
华盛顿州立大学，2022年7月

主席：Haipeng Cai

如今，越来越多的分布式软件系统正在被开发和部署。与其他软件一样，分布式软件系统也需要强有力的质量保证支持。分布式软件通常规模庞大且复杂，具有分布式组件，并且不存在全局时钟。所有这些特征都使得分析此类系统的信息流、从而支持软件质量保证变得极具挑战性。其中一个挑战是，现有的动态分析技术很难扩展到现实世界中的大型分布式软件系统。此外，开发高性价比的动态分析方法也颇具挑战。对于分布式软件的动态分析算法和应用而言，还面临适用性和可移植性方面的挑战。

我的博士论文通过三种针对分布式软件数据流分析的新方法来应对这些挑战。第一种方法基于对进程间通信的测量，以了解分布式软件行为并预测分布式软件质量。随后，我开发了一种特定方法，通过

针对分布式软件的多阶段、基于细化的动态信息流分析，能够准确定位敏感信息。最后，我探索了分布式系统的动态依赖分析，并利用强化学习自动调整分析配置，以提升可扩展性和成本效益权衡。

目录

Page

致谢.iii 摘要.v 图目录.xi 表目录.xiii 章节

1. 引言.1

动机与问题陈述.3 贡献.4 论文结构.6 2. 背景.8

软件度量.8 分布式系统架构.10 分布式系统中的逻辑时钟.11 依赖分析.13 单进程程序的依赖分析.14 多线程程序的依赖分析.15 分布式程序的依赖分析.16 动态信息流分析和动态污点分析.17 强化学习.18 具有可变成本效益的分析.19

3. DISTMEASURE：一个通过进程间通信视角衡量和理解分布式软件系统的框架.
21

动机.22

方法.23 总体工作流程.23 IPC指标.24 与IPC指标相关的质量指标.27 用于理解和预测质量的分类.30

工具实现.32

评估.32 实验设置.32 IPC 测量.37 与 IPC 测量相关的质量测量.40 用于预测和理解质量的分类.42 有效性威胁.46 经验教训与要点.49 相关工作.52 动态耦合度量.52 运行时内聚.53 基于机器学习的软件质量评估.53

4. FLOWDIST：面向分布式软件系统的多阶段细化式动态信息流分析。56

动机.56

方法.60 概述.60 第 1 阶段：预分析.61 阶段 2：细化.64 替代设计.68 局限性.70 工具实现.70 评估.71 实验设置.71 结果与分析.73 相关工作.81 常规信息流分析（IFA）与污点分析（TA） . . . . 82 跨进程信息流分析（IFA）与污点分析（TA） 83

5. SEADS：通过强化学习实现可扩展且高性价比的动态依赖分析.8

4 动机.84 方法.85 概述.86 配置.87

插桩器.92 监控器.92 控制器.101 查询客户端.105

局限性.105 工具实现.106 评估.107 实验设置.107 结果与分析.108 有效性威胁.117 相关工作.119 分布式软件的依赖分析.119 具有可变成本效益的分析.120

6. 结论与未来工作.121

总结.121 未来工作.122 模糊测试技术.122 云系统分析.123 物联网（IoT）系统分析.123

参考文献.125

图目录

Page

1.1 我的研究概述.4

2.1 分布式系统的架构：(a) 左：客户端-服务器，(b) 中：点对点，以及 (c) 右：多层 (三层). 10

2.2 分布式程序多个进程中使用的 Lamport 时间戳算法.12 2.3 Q学习工作流.18 3.1 DistMeasure 工作流概述.24 3.2 DistMeasure所依据的参考质量模型，该模型取自 ISO/IEC 25010 [126] 标准并符合该标准.28 3.3 DistFax 架构概述.32 3.4 面向质量指标的监督（装袋法）分类器所用特征（IPC 指标）按重要性得分排序（显示在 x 轴上）。左图显示了 RMC 与 CCC 在分类运行时圈复杂度时的重要性得分，右图显示了 CCL 与 PLC 在分类攻击面 时的重要性得分.45 3.5 其他监督学习算法的平均准确率与默认算法（装袋法）的比较.46 4.1 Apache ZooKeeper 中跨三个组件（进程）的敏感信息流案例（以带箭头的线标示）.57 4.2 FlowDist 架构概述.60 4.3 FlowDist 的第 1 阶段工作流.61 4.4 FlowDist 的阶段 2 工作流.65 4.5 DistTaint架构概述.71 4.6 FlowDist 与基线 Phosphor 和 JOANA 的准确率比较 74

4.7 所有被测程序执行的运行时减速（%， y 轴）与方法执行事件实例数（x 轴）的关系.77 4.8 所有被测程序（集成测试）的总分析时间（秒， y 轴）与被测程序规模（源代码行数， x 轴）的关系.77 4.9 所有被测程序执行中， FlowDistmul 和 FlowDistsim 相对于FlowDist 的总时间开销（秒）.79 4.10 所有被测程序执行中， FlowDistmul 和 FlowDistsim 相对于FlowDist 的存储成本（MB）.80 5.1 Seads 架构和工作流概述，包括其输入、输出和关键模块.86 5.2 监控器和控制器与插桩对象同时运行 . . . . . 93 5.3 Q学习中智能体与环境之间的交互 . . . . . 103 5.4 Dads 架构概述.106 5.5 比较y 轴上显示的成本效益，即每次执行中 Doda 与 Seads 的精确率和响应时间之比（x 轴）。比值越高，成本效益越高.114 5.6 比较y 轴上显示的成本效益，即每次执行中 Doda 与 Seads 的精确率和运行时减速之比（x轴）。比值越高，成本效益越高.115 6.1 模糊测试工作流概述.122

表目录

Page

3.1 IPC 指标总结.26

3.2 DistMeasure 中实验对象的统计信息.33

3.3 系统级 IPC 测量结果.38 3.4 两项直接质量度量的测量结果：信息流路径数量 和信息流路径长度 .40 3.5 其他（六项）直接质量度量的测量结果，按逻辑源代码行数或千行源代码规范化.41 3.6 IPC 与质量指标之间的 Spearman相关系数（p 值）42 3.7 无监督学习（k-means）分类对动态可预测质量指标的有效性.43 3.8 监督学习（装袋法）分类对动态可预测质量指标的有效性.44 4.1 所用的目标分布式程序和测试输入.72 4.2 进程内（Ir）源/汇对（Pr）和信息流路径（Ps）的数量，与进程间（Int）相应数量的比较.74 4.3 FlowDist 的时间（秒）和存储成本（MB）.75 4.4 FlowDist 检测到的现有漏洞.78 4.5 FlowDist 发现的新漏洞.79 4.6 DIFA/DTA 工具选择建议.81 5.1 混合（静态/动态）配置编码.91 5.2 实验对象.107 5.3 Seads 相对于以 Doda 为基线的时间（秒）、存储成本（ MB ）和精确率（比率） .109

5.4 Q学习的迭代次数与学习时间（秒）.....116

献辞

谨以此献给.....

我支持我的妻子、父母和朋友。

第一章

引言

由于对计算可扩展性和性能的要求不断提高，越来越多的现实世界软件系统被设计为分布式系统 [52]。执行通用分布式计算的软件系-统被称为 [52], 通用分布式系统。除了通用分布式系统之外，还有一些专用分布式系统，例如基于 RMI 的系统 [206], 分布式事件驱动（DEB）系统 [169], 云系统 [156], 物联网（IoT）系统 [226], 等。

与单进程程序相比，分布式软件系统具有多个独特的特点：

1. 分布式系统中的组件/进程是并发的。

2. 多个组件/进程通常具有自主性 .

3. 硬件和软件资源可以共享。

4. 分布式软件通常没有全局时钟。

5. 价格/性能比可能会好得多。

另一方面，分布式软件系统广泛应用于关键领域（例如航空、银行业、医疗和社交媒体）。

与其他软件系统一样，分布式软件也需要强有力的质量保证支持（例如性能监控、程序优化、软件测试、漏洞检测 [141]），其中质量包括安全性以及性能等其他因素

效率、可维护性和功能适合性 [126]。然而，分布式系统的特性不仅使安全问题变得更加复杂 [22, 62, 54, 57, 61] ，也为分析这些系统中的数据流（即信息流）带来了严峻挑战。

一项主要挑战是，现有的单进程程序数据分析技术（例如 [176, 44, 186, 166, 223, 207, 16, 27]）很难扩展到分布式系统（例如 Apache ZooKeeper [7], ，一种用于实现同步和一致性的分布式协调服务，Apache Hadoop 和 Yahoo 都在使用它）。原因在于，这些技术依赖程序实体之间的显式依赖关系，却忽略了进程间的隐式依赖关系 [93]。

开发一种经济高效的动态分析也颇具挑战，因为在特定分析中，成本与有效性往往相互牵制、彼此竞争。要为现实世界中的大多数分布式系统开发此类分析方法，挑战更大，因为与单进程程序相比，这些系统通常规模更大、复杂度更高。分布式系统中非确定性、多变且通常无界的执行，进一步加剧了这些挑战。

本博士论文的主要目标是探索和研究用于分布式软件安全的数据流分析方法，以克服可扩展性、成本效益以及其他挑战（即适用性和可移植性）。总体而言，这项工作为分布式软件的质量保证提供了基础支持。具体来说，本研究旨在预测和理解分布式软件系统的质量，以及与其运行时行为和执行动态相关的特性。此外，本研究主要针对动态信息流安全、动态程序依赖关系及相应的应用程序，旨在通过多种方式实现实际可扩展性并平衡分析成本效益，例如采用有原则的多阶段分析策略、强化学习策略，以及分别利用最优强化学习和深度强化学习的两种自适应策略，

作为一种有原则的多阶段分析策略、一种强化学习策略，以及两种分别利用最优强化学习和深度强化学习的自适应策略。

1.1 动机与问题陈述

在分布式软件面临的各种安全威胁（例如代码注入）中，分布式程序信息流路径中的各类漏洞是一种主要威胁。在这些程序中，敏感信息（例如用户名或密码）可能会泄露并造成严重损失／损害。为了防范此类信息流威胁，检查贯穿整个系统（跨越其分布式组件及相应进程）的敏感数据至关重要。有效的信息流分析（即数据流分析）通常需要对控制流和数据流路径进行细粒度（例如语句级）计算。然而，精确的细粒度数据流分析通常代价高昂。分布式系统的高度复杂性，是现有大多数相关方法不适用（例如受可扩展性障碍所限）或效用非常有限（例如仅适用于单个组件／进程）的主要原因。对于许多通常持续运行的分布式系统（例如在线／云服务），持续监控它们以防范安全威胁是理想的做法。在这些场景中，实现并维持可扩展性甚至更加困难。

此外，除了可扩展性问题之外，为分布式软件系统开发数据流分析解决方案还面临许多其他问题。例如，开发经济高效的动态依赖分析颇具挑战，尤其是考虑到通常情况下动态分析已知存在显著开销。先前的研究已表明，在单进程程序的动态依赖分析中，平衡成本与有效性十分困难且复杂

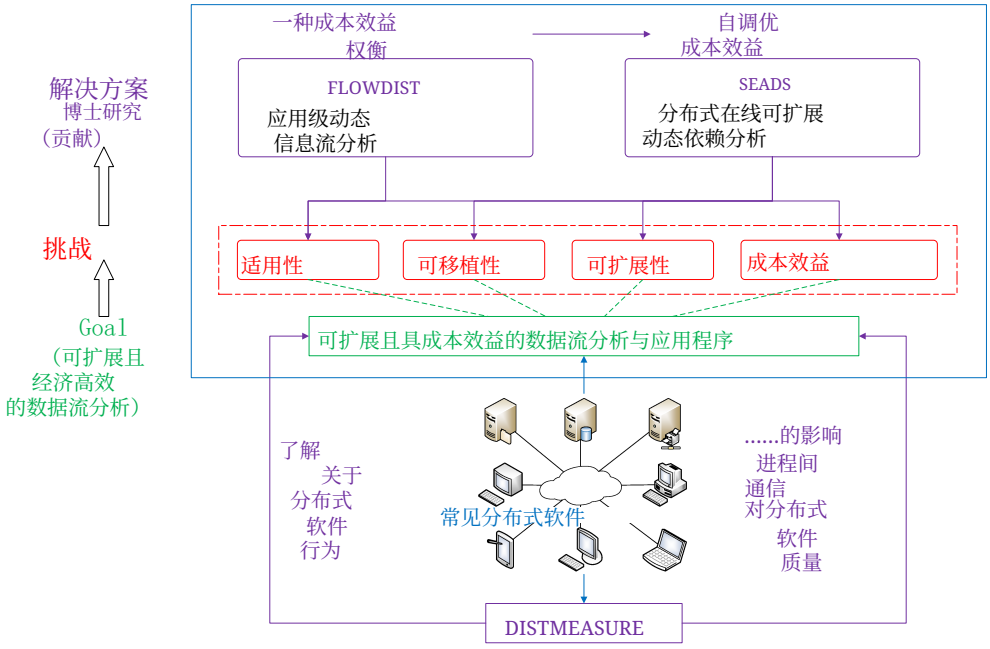
分析 [40]。执行非确定性、现实世界中运行时环境的多样性与不确定性，以及无界执行（由于其持续运行的特性），进一步加剧了大多数分布式软件系统面临的挑战。

此外，传统的动态分析方法很难适用于多进程程序，例如分布式系统。原因在于，这些方法依赖程序实体之间的显式依赖，而dismiss跨进程的隐式依赖 [93]。

一些现有的动态分析工具（例如， [128, 221]）通过在系统级进行平台定制来克服其适用性挑战。然而，由于需要针对多样且快速演进的平台进行定制，这些工具通常会面临可移植性挑战，而这将耗费大量时间，甚至难以实现 [89]。

1.2 贡献

图1.1：我的研究概述



在本博士论文中，我探讨了如何设计切实可行的解决方案，以应对大型且复杂的分布式软件系统中数据流分析的可扩展性、成本效益以及其他（即适用性和可移植性）挑战。因此，如图1.1所示，我的研究重点聚焦于三个相互关联的主题： DistMeasure、DistTaint和 Seads。

首先，我研究了 DistMeasure （ [93]的扩展），其中包括一组新颖的进程间通信（IPC）指标，用于测量通用分布式系统，以理解系统行为及系统进程间通信的影响。基于 DistMeasure，我实现了 DistFax [92],这一用于测量 IPC 和分布式软件质量的工具包。使用 DistMeasure，我针对 11 个真实世界的分布式软件系统及其多样化的执行场景，展示了 IPC 指标的实用性和可行性。实验表明，分布式进程之间较高的 IPC 耦合往往与分布式软件质量呈负相关，而单个进程的内聚性则对质量具有正面影响。对 DistMeasure基于学习的质量等级分类进行评估后发现，IPC测量在理解分布式系统行为方面展现出可观的价值，能够揭示 IPC 测量与分布式软件质量各方面之间的统计与预测关系。

其次，我探索了一种动态信息流分析框架 FlowDist [89] ，它通过一种有原则的多阶段分析方案克服了多项技术挑战（适用性、可移植性和可扩展性），并将传统的动态信息流分析扩展到分布式系统。相应的工具 DistTaint [86]——一种面向分布式软件的动态污点分析器——也已实现。针对 12 个真实世界分布式系统，并与两个基线进行评估后，结果表明其有效性更高、实际效率更优，且具备可扩展

性。 FlowDist发现了 24 个现有漏洞和 24个新漏洞，其中 17 个已得到确认，2 个已修复。此外，还提出并评估了 FlowDist的两种替代设计，以适应不同的对象。

由于 FlowDist 只提供一种成本效益权衡，我开发了 Seads [91]，这是一个面向持续运行的分布式系统的分布式在线动态依赖分析框架，可提供自调优的成本效益（权衡）。这是我为实现更好的可扩展性和成本效益所做的尝试，也是迈出的第一步：利用强化学习，根据先前的配置、相应的成本和用户定义的预算，在运行期间自动、持续地调整分析配置，从而实现实时优化。我还实现了 Dads [90]，这是一个面向持续运行的分布式软件的分布式在线动态切片器，可满足用户指定的预算约束；它通过强化学习实时、持续、自动地调整分析配置，根据给定的分析时间预算实现并维持实际可扩展性和成本效益权衡。实证结果表明，与传统动态分析方法相比， Seads 在可扩展性和效率方面具有优势，至少在方法层级计算动态依赖关系时如此。

1.3 论文结构

在本论文的其余部分中，我将在第 2 章（§2）讨论本研究的一些技术和关键概念。基于对分布式系统进程间通信的测量，我将在第 3 章（§3）从 DistMeasure （ [93] 的扩展）入手，以理解和预测分布式软件的行为与质量。随后，我将介绍面向分布式软件的基于多阶段细化的动态信息流分析 FlowDist [89] ,

并在第 4 章（§4）和第 5 章（§5）分别介绍通过强化学习实现的可扩展且经济高效的分布式系统动态依赖分析 SEADS [91] 。最后，第 6 章（§6）总结本论文，并探讨未来的若干研究方向。

第二章

背景

本节将讨论我的研究中涉及的一些技术和关键概念，包括软件度量、分布式系统架构、分布式系统中的逻辑时钟、依赖分析、动态信息流分析和动态污点分析、强化学习以及具有可变成本效益的分析。

2.1 软件度量

使用恰当选定的度量指标来衡量软件系统，是软件质量保证中的一个重要步骤 [95]。在整个软件开发生命周期中，定义适当的软件度量对于软件过程质量和产品质量都至关重要 [132]。在实施阶段之前，软件度量为依据相关质量因素规定质量要求提供了一种方法。实施之后，这些度量还可作为重要指导，帮助根据质量要求规范评估软件产品。软件度量在整体软件项目管理中也发挥着至关重要的作用（例如，用于成本和工作量估算） [80]。

软件度量中可以使用两大类度量指标：静态和动态 [222]。相比之下，静态度量指标通常比相应的动态度量指标更容易计算 [45, 75]。此外，静态度量指标不像动态度量指标那样受代码覆盖率有限或普遍适用性不足的制约。另一方面，静态度量指标不足以衡量和解释软件的动态行为，而动态度量指标则能提供更精确的指示。事实上，对于最终在运行时得到验证的质量因素

（例如性能 [75], 、可靠性 [241], 和可测试性 [110]），动态度量指标更为适用。同时，理解软件行为并不总是需要完整的代码覆盖率 [192], ，因此，动态度量指标所涉及的有限执行并不一定构成动态测量的限制。动态度量指标也不能简单地通过相应的静态度量指标进行（过度）近似——在某些情况下，两者甚至毫无相关性 [96]。

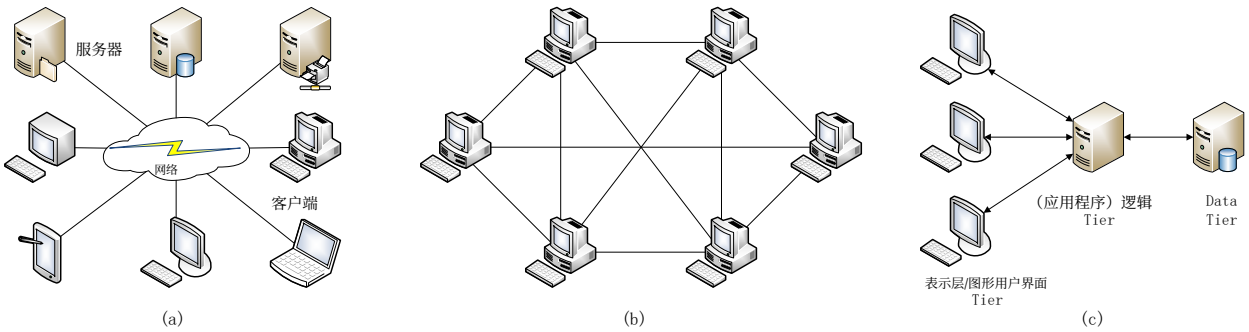
软件耦合是指软件模块之间关系的强度，用于衡量模块之间的紧密程度 [172, 172]。针对单进程系统的耦合度量已得到充分研究 [45, 132, 222]。例如，Arisholm 等人 [14] 定义了一组面向对象软件的动态耦合度量，并研究了动态耦合度量与软件变更倾向性之间的关系。动态耦合度量还被用于评估架构风险 [241] 和复杂度 [111] ，并将其与可维护性 [182, 110]等质量指标联系起来。大多数此类度量的定义都基于这样的假设：参与耦合度量的实体（例如对象、方法和类）之间存在显式的引用/调用。

软件组件的内聚性是指组件各元素之间的关联程度 [33]。高内聚的组件执行一组紧密相关的动作，且难以拆分为独立的组件 [241]。静态内聚性已在软件度量领域得到广泛研究。近期越来越多的相关工作开始关注运行时（即动态）内聚性 [167, 115, 253, 69]。

2.2 分布式系统架构

分布式系统架构通常有三种类型：客户端-服务器、对等和多层，如图2.1所示。

图2.1： 分布式系统的架构：(a) 左侧：客户端-服务器，(b) 中间：点对点，(c) 右侧：多层 (三层)



客户端-服务器 (CS) 是一种网络架构，其中网络上的每个进程/节点要么是服务器，要么是客户端。服务器具有强大的能力，可控制和管理相关资源（例如磁盘驱动器、数据库、打印机和网络流量），而客户端依赖服务器提供这些资源 [157]。客户端-服务器架构易于实现，且无需进行对等发现 [119]。例如，NioEcho 是一个包含客户端和服务器的客户端-服务器分布式程序 [215]。

点对点 (P2P) 是一种网络架构，其中每个进程/节点具有同等的职责和能力。P2P 与 CS 不同，CS 中的一些进程/节点专门为其他进程/节点提供服务 [157]。例如，OpenChord 是一个点对点分布式系统，通过分布式哈希表提供网络服务 [26]。

多层架构将应用程序拆分为多个层，为开发人员提供灵活性和可重用性；开发人员只需修改或添加特定层（层级），而不必

在决定更改应用程序时重写整个应用程序。在术语多层中，“n”可以是特定架构中所用不同层的任意数量（大于 1），例如两层、三层或四层等 [157]。例如，Microsoft Azure 是一种典型的多层分布式系统，提供云计算服务 [235, 50, 164]。

2.3 分布式系统中的逻辑时钟

分布式系统基于计算机网络，不同计算机通过传递消息或借助其他类型的中间件相互连接。这一特性使用户能够通过网络通信共享各种资源 [170]。但是，分布式组件并发执行时，并不存在全局逻辑时钟或物理时钟。

Lamport 时间戳算法（LTS 算法）用于为分布式系统中的事件生成偏序，并为所有进程维护逻辑时钟。在 LTS 算法中，每个进程都维护一个初始值为零的整数，该整数会定期递增，即每发生一次原子事件后递增一次；该值作为时间戳，附加到每个事件执行记录中，可以集中维护，也可以分别维护；跟踪通常由各个进程维护 [81]。LTS 算法支持异步和同步通信方法，并遵循以下规则：

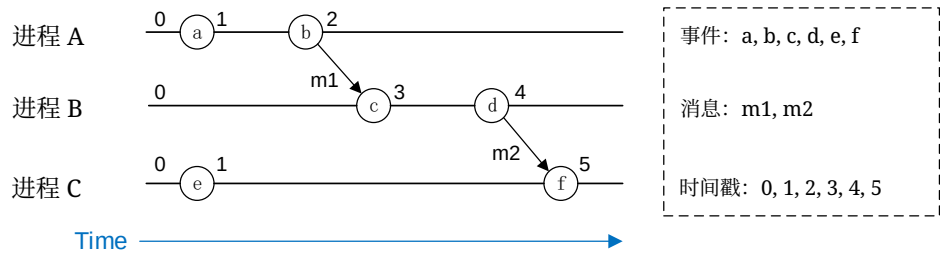
1. 进程中的每个事件发生时，该进程都会递增其计数器。

2. 发送消息时，进程会将其计数器的值附在消息中。

3. 接收消息时，接收方的计数器会更新（加 1）。

我用一个示例详细解释 LTS 算法。假设一个分布式程序中有三个进程（即 进程 *A*、进程 *B*、进程 *C*），如图

图 2.2： Lamport 时间戳算法在分布式程序的多个进程中的应用



2.2 所示。每个进程的逻辑时钟都初始化为零，并且每发生一个事件，时钟值（即时间戳）就立即递增，例如，事件 a的时间戳为 1，事件b的时间戳为 2，依此类推。当消息$m1$ 从进程 *A* 发送到进程 *B*时，搭载了时间戳 2；随后，事件c （属于进程 *B*）的时间戳为 3，即搭载的时间戳 2 与本地时间戳 0（初始值）中的最大值加 1，如 (max(0, 2) $+ 1 = 3$)。接着，事件d 的时间戳为 4（$= 3 + 1$）。然后，消息$m2$ 从进程 *B* 发送到进程 *C*，并搭载时钟值 4。最后，事件f （属于进程 *C*）的时间戳为 5（=max(1, 4) $+ 1$），其中 1 是同一进程（进程 *C*）中前一个事件（e）的时间戳 [81, 144]。

与 LTS 算法类似，向量逻辑时钟通过比较事件时间戳 [25]，用于确定事件的偏序，并检测分布式系统中的因果关系违规。对于包含 N 个进程的分布式系统，存在一个由 N 个逻辑时钟构成的向量（即数组），每个进程对应一个时钟；每个进程都维护一个本地向量（即数组），其中包含全局时钟向量 [254, 234] 的可能最大值。

然而，向量逻辑时钟存在一个重大缺陷：如果分布式系统中的进程过多，其数量 N 非常庞大，则附加在每条消息上的时间戳数据（即通信开销）会大到无法接受 [220]。

除了向量时钟和 LTS 时间戳算法之外，矩阵时钟也用于捕获分布式程序中的因果关系和时间顺序关系。作为向量时钟概念的推广，矩阵时钟为每个进程（通信主机）维护一个由向量时钟组成的向量 [73, 233]。

2.4 依赖分析

在程序执行过程中，存在两种类型的显式依赖：传统依赖和线程依赖。控制依赖和数据依赖属于传统依赖。在并发程序中，线程依赖可能有三种类型：同步依赖、就绪依赖和干扰依赖。在通用分布式软件系统中，除了代码实体之间的显式依赖（通过引用和/或调用），分布式组件/进程之间的依赖（称为组件间/进程间依赖）是隐式的，因为这些组件/进程通过网络设施彼此解耦。

分析软件系统中程序实体之间的依赖关系，有助于开发人员更好地理解系统的结构和行为。因此，依赖分析对用户开发、测试和维护系统非常有用，因为这些任务都依赖于对系统结构和行为的理解。

程序依赖关系既可以通过静态分析推导，也可以通过动态分析推导。静态依赖分析通过分析程序代码来计算依赖关系，而无需执行软件。相比之下，动态依赖分析则根据执行过程中收集的数据推断依赖关系。具体而言，作为一种特殊的动态分析，混合依赖分析结合了静态和动态分析模式 [203]。混合

依赖分析方法可以将从动态分析技术中提取的运行时信息整合到静态分析算法中，以精确计算程序依赖关系。混合依赖分析越来越受欢迎，因为与其他分析模式相比，它们能生成更准确的结果。然而，混合依赖方法通常较为复杂，其分析过程也可能耗时较长 [190]。

2.4.1 Dependence analysis for single-process programs

作为一种动态分析方法， Diver [39] 利用依赖分析技术将依赖集计算为影响集。作为（离线）动态分析领域的一项最新进展，与基于 EAS 的方法（根据执行顺序推导动态依赖关系）相比，它精确率更高，也更具成本效益，例如 PI/EAS [8]。Diver 利用静态依赖分析，大幅缩小了由 PI/EAS生成的依赖集规模。

由于最终得到的依赖集显著更小，即使增加了静态依赖分析成本，Diver 的成本效益权衡也高得多。Diver分为三个技术阶段：静态分析（第 1 阶段）、运行时跟踪（阶段 2 ）和后处理分析（阶段 3 ）。 Diver 首先在第 1 阶段计算传统的控制 / 数据依赖关系 [123] ，并对输入程序进行插桩。在阶段 2，执行该程序的插桩版本，以跟踪进入事件（即程序控制流进入某个方法）和返回进入事件（即程序控制流从被调用方法返回到调用方法）。在阶段 3，该技术根据跟踪信息，针对用户提出的任意查询计算依赖集。

Diver 的在线版本 DiverOnline [37] 通过计算依赖集，避免了离线分析中不可避免的执行跟踪成本（例如空间和 I/O 成本），在此期间。

被分析程序执行期间计算依赖集，避免了离线分析中不可避免的执行跟踪成本（例如空间和 I/O 成本）。此外， DiverOnline 提供一体化分析，可计算所有可能查询（方法）的依赖集，并在短响应时间内直接向用户提供相应的依赖集。因此，一体化在线动态依赖分析可能适用于大型软件系统。

2.4.2 Dependence analysis for multi-threaded programs

多线程（并发）程序的使用日益增加，这给依赖分析带来了挑战，同时用户也较难理解多线程系统。例如，在共享内存模型中，一个线程访问某个内存位置时，可能会干扰另一个线程对同一位置的访问，从而导致这两个线程之间产生依赖关系。

通常，线程之间存在两种特定类型的控制依赖——就绪依赖和同步依赖，以及一种数据依赖——干扰依赖 [173]。因此，线程引发的依赖动态分析的主要任务，是推断程序执行过程中多个线程之间出现的就绪依赖、同步依赖和干扰依赖。针对多线程程序，人们已经开发出若干依赖分析算法 [240, 239, 173, 98]。

Indus 是一个可靠的框架，可用于分析和切片多线程（当前）Java 程序。在 Indus 中，Java 源代码首先被转换为作为中间表示的 Jimple 代码。然而，由于它完全是静态的，Soot 只能以极低的成本对这些依赖关系进行近似分析 [51, 189]。Indus 为开发人员提供了最常见的依赖

分析，包括线程内的控制依赖和数据依赖，以及线程间的就绪、同步和干扰 依赖。

2.4.3 Dependence analysis for distributed programs

对于包含多个进程的复杂分布式系统，开发人员需要了解单个进程内部以及多个进程之间的各种（显式和隐式）依赖关系。Krinke 提出了一种切片算法，可纳入由基于套接字的消息传递所引发的分布式组件间依赖关系 [142], ，但由于这些依赖关系是通过纯静态分析计算得出的，因此其近似结果过于保守。另一种方法 [28] 可推断由进程间通信导致的各种依赖关系。然而，这种方法可能因开销较大而面临可扩展性问题。

为克服可扩展性方面的挑战，研究人员开发了一种面向分布式程序的轻量级动态分析方法 DistIa [41]。该分析方法会在系统执行期间监控并记录方法事件及其时间戳，然后根据方法执行事件之间的先发生关系，近似推断相关方法在进程内部或进程之间的运行时依赖关系。

例如，如果方法 A 的最后一次返回进入事件在另一个方法 B的首次进入事件之前执行，则偏序为 A *before* B，而 DistIa大致假设 B 依赖于 A。同样，如果一个方法 D 依赖于另一个方法 C，那么 C 必须先于 D 执行；否则， C 就无法影响 D。因此， DistIa 计算出的依赖关系对于已执行的方法是安全的，但并不适用于所有方法，因为有些方法在执行期间未被覆盖。另一方面，如果 B 是

在 A之后执行， B 可能确实依赖于 A，也可能并非如此。因此，我们知道DistIa 并不精确。

2.5 动态态信息流分析与动态污点分析

跟踪/检查动态信息流是各种安全应用的基础（例如， [218,159, 161, 198, 107]）。它解决的是程序执行中的一般源-汇问题：源 是机密或不可信（即敏感）信息产生并流入程序的位置，而汇 则会使用这些信息，并使其流出程序执行过程 [79, 4]。由于这种方法侧重于对实际执行进行推理，因此相较于静态推断信息流具有精确率方面的优势。

实现该方法的一种技术是计算动态控制依赖和数据依赖链，从而推断执行期间给定源和汇之间的完整信息流路径（例如， [210, 160, 159, 161]），这种技术称为动态信息流分析（DIFA）。另一种技术是动态污点分析（DTA），它会对从源进入程序的数据应用标签（即标记为污点），在执行期间传播污点标签，并检查汇处的数据是否带有这些标签（例如， [49, 247,
176, 202, 166, 207, 16, 27, 256, 134, 128, 221, 246, 244, 77, 243, 121, 47, 133]）。与 DIFA 不同，DTA 不会计算完整的信息流路径。因此，对于需要更详细流信息的使用场景（例如，诊断路径中的数据泄漏），DIFA 能提供更好的支持。

DIFA 和动态污点分析可通过以下两点加以区分：(1) 根据前文所述的内部工作机制（即，DIFA 通过计算动态依赖关系来工作，而动态污点分析则通过数据污点标记和污点传播来工作）；(2) 根据二者的结果——DIFA 提供完整的信息流路径

而动态污点分析只能说明哪些数据被标记为污点。另一方面，DIFA 和动态污点分析都关注给定源和汇之间的信息流路径。

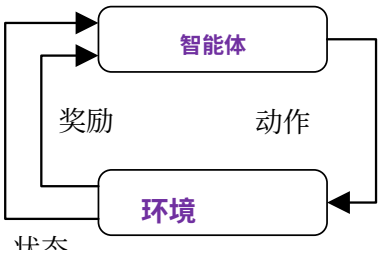
2.6 强化学习

强化学习（RL）是机器学习的一个领域，超越了监督学习，旨在提出让软件智能体在环境中采取动作，从而使所有可能的后续动作的总奖励最大化 [151]。RL 基于从环境中测得的信息，因此也称为基于动作的学习。RL 指一种学习方法，其中智能体或行动者会根据其与环境交互所得到的响应来调整动作。RL 几乎不需要先验知识，因此可应用于多变且不确定的环境，而标准的监督学习/无监督学习方法并不适用于这类环境 [146]。

与监督学习和无监督学习不同，RL 不需要训练数据。根据马尔可夫决策过程的描述，状态由先前的状态决定 [185]，RL 的输出取决于相应的输入，而下一个输入则取决于当前输出 [3]。

具体而言，作为一种特定类型的强化学习，Q学习使用贝尔曼方程来最小化其累积成本 [29]。它不需要对软件系统进行明确或精确的描述，只需在其反馈控制回路中获取状态测量值 [146]。Q学习还能够仅根据先前的状态准确地学习下一个状态。从初始状态开始，Q学习会尝试通过最大化累积奖励值来寻找一种方式，以选择动作来最大化

图2.3： Q学习工作流



累积奖励值：在特定状态下测量动作的效果后选择一个动作 [229]。它是一种离策略、无模型算法，因为不需要现有的策略或模型 [230]。Q-learning算法的整体工作流程如图2.3所示，包括以下步骤：

(1) 算法初始化 Q学习 组件和 Q表（即用于存储计算所得预期奖励的查找表）。

(2) 在当前状态下，智能体参考 Q表中的最大值或通过随机方式选择一个动作。

(3) 智能体从环境中获得一个状态和一项奖励。

(4) 算法使用贝尔曼方程 [71] 更新 Q表。

(5) 算法重复步骤 (2)、(3) 和 (4)，直到学习满足预定义条件（例如，智能体完成其最终任务）。

2.7 可变成本效益分析

大多数现有分析方法普遍面临着平衡分析成本与有效性的挑战。它们要么精确但成本过高，要么高效但精度不足。为应对这些挑战，一种现有解决方案是提供不同的成本效益平衡，以满足用户的不同需求。例如， DiaPro 框架为不同层级的成本效益权衡提供了灵活的选择，从而针对不同的用户需求和预算提供最佳选项 [40]。通过结合静态和动态数据， DiaPro 统一了 PI/EAS [8]、 Diver [39]，以及三种基于依赖的动态依赖分析技术：一种使用覆盖率和跟踪，一种使用别名和跟踪，另一种则使用所有这些动态数据（别名、覆盖率和跟踪）。

另一个例子是 D^2ABS [38]，它旨在实现实际可扩展性，并针对分布式程序的动态依赖分析提供四种版本，涵盖不同层级的成本效益权衡。

第三章

DISTMEASURE：一种通过进程间通信视角测量和理解分布式软件系统的框架

我的博士研究旨在解决分布式软件分析及其在其他质量问题中的应用所面临的可扩展性、成本效益和其他（适用性/可移植性）挑战。首先，我研究了分布式系统中的安全问题。分布式系统中存在一种通信机制，基本上用于连接不同的进程。进程间通信（IPC）可能是分布式系统最重要的特性之一。因此，我开发了 DistMeasure （ [93, 85]的扩展），其中包含一组针对通用分布式系统的指标，重点关注其 IPC——这是系统运行时行为的一个重要方面。我还实现了一个工具包 Dist-Fax [92] ，用于测量 IPC 和分布式软件的质量。通过针对 11 个真实世界分布式系统的大量测试输入计算相关度量，我证明了利用所提出的 IPC 指标刻画 IPC 动态特性和复杂度的实用性。为了展示 IPC 测量的实际用途，我深入研究了所提出的指标如何帮助预测和理解分布式系统的各种质量指标，具体做法是根据 IPC 测量结果，针对多个不同的质量方面对系统质量等级进行分类。

3.1 动机

一般而言，分布式系统由多个协作组件构成，每个组件都在一个进程中运行，而该进程通常位于独立的计算节点上。由于这些组件主要通过 IPC 进行交互， [204], 测量 IPC 对于理解分布式系统的行为至关重要。

显然，IPC主导着该系统的各项活动；如果排除 IPC，各个组件本身基本上都微不足道。因此，该系统执行的复杂度本质上在于其 IPC 的复杂度。首先，我想利用能够复现相关问题的输入，诊断用户报告的该系统通信安全问题。

一个颇有价值的第一步，是了解当前通信如何围绕用户输入通过 IPC 运作。测量 IPC 将有助于分析这一场景。此外，IPC 测量或许还能帮助开发者进一步评估 IPC 所引发系统的质量方面。例如，如果进程间通信耦合非常高，直观而言，该系统可能难以更新。因此，其维护成本也可能相当高。

遗憾的是，目前缺乏用于测量分布式系统中与 IPC 相关执行情况的工具支持，而且尚不清楚哪些质量因素可以通过 IPC 测量进行分析。为此，我通过开发新的 IPC 指标并将其应用于分布式软件质量评估，来解决这些问题。

与此同时，直接测量分布式系统的某些质量指标可能很困难，甚至不可能。例如，代码变更量 这一质量指标通常用于量化不断演进的程序中发生变化的部分。然而，计算这一质量指标通常需要程序发布版本的数据，而程序的第一个发布版本无法直接测量

。开发人员可以利用 IPC指标与分布式系统质量（指标）之间的相关性，间接评估该系统中相关的质量指标，进而了解该系统及其质量方面。

3.2 方法

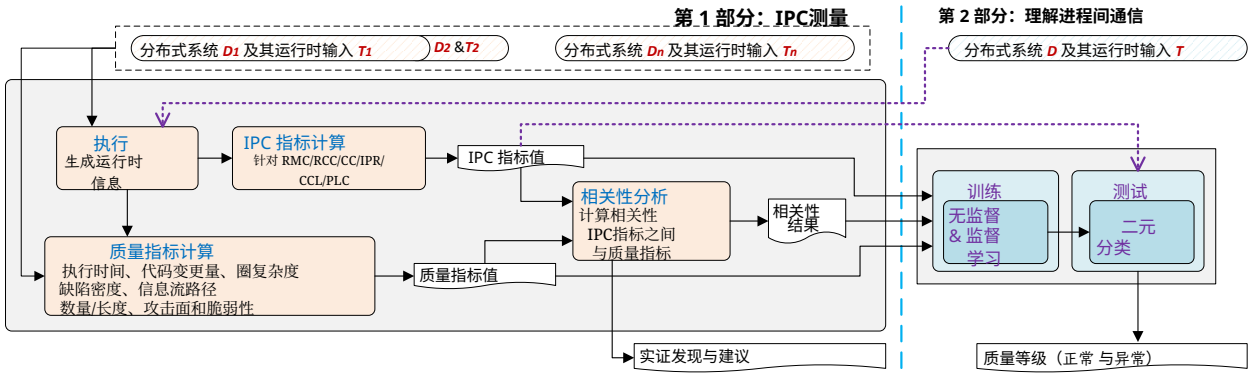
我首先概述 DistMeasure 高级工作流程和架构。然后，我介绍 DistMeasure中使用的进程间通信指标和质量指标。

3.2.1 Overall workflow

图 3.1 描述了 DistMeasure的概述，其中包括两个紧密相连的部分：*IPC*测量（第 *1* 部分）和理解 *IPC*（第二部分）。在第 *1* 部分中， DistMeasure从用户处获取两个**输入**：多个（N）分布式系统 D_1、 D_2、.....、 D_n 及其运行时输入 T_1、 T_2、.....、 T_n ，包括系统命令、SQL语句、文本消息等。 DistMeasure 首先针对相应输入运行这些系统，以生成系统运行时数据。然后， DistMeasure 根据这些数据计算六个进程间通信指标（即 RMC/RCC/CCC/IPR/CCL/PLC）。

与此同时，直接测量系统的八项质量指标（即，执行时间、代码变更量、圈复杂度、缺陷密度、信息流路径数量、信息流路径长度、攻击面和脆弱性）。根据这些 IPC 指标和质量指标的值， DistMeasure 计算它们之间的相关性。这些相关性结果引出了关于系统质量的实证发现和建议，作为 DistMeasure 第 1 部分的**输出** 。

图 3.1： 工作流程概述 DistMeasure



在第 *2* 部分中， DistMeasure 将 IPC 和质量指标值、分布式系统D以及 D的输入 T 作为**输入**，用于对 D的相同质量指标进行分类。它首先根据度量值训练无监督学习和监督学习模型，然后使用这些模型将每项此类质量指标分类为 `anomalous` 或 `normal`。 `Anomalous`表示质量较低（低于平均水平）的警告，而 `normal` 则表示没有此类警告（即，相应的质量指标值高于或等于平均水平）。具体而言，这些模型经过训练后用于分类质量指标；这些指标均与六项 IPC 指标中的一项或多项显著相关（正相关或负相关），相关系数的绝对值为 $>= 0.4$。这些分类结果为开发人员在软件维护中的质量保证任务提供指导，例如软件调试、软件测试等。

3.2.2 IPC metrics

如表 3.1 所示，我定义了五个进程间通信耦合指标（即 运行时消息耦合（*RMC*）、运行时类耦合（*RCC*）、类中心耦合（*CCC*）、进程间复用（*IPR*）和类通信负载（*CCL*））以及一个 IPC耦合度量（即 进程-

层级内聚性（***PLC***） [92]），并说明了它们的计算方法（第二列）和合理性说明（第三列）。六个进程间通信指标涵盖四个测量粒度层级：方法（IPR）、类（RCC、 CCC 和 CCL ）、进程 / 组件（ RMC 、 RCC 和 PLC ）以及系统（ RMC 、 RCC 、 CCC、 IPR、 CCL 和 PLC），列于表 3.1 的最后一列（**NS. L.**，即非系统级）。系统级未列于表 3.1 中，因为它涵盖全部六个 IPC指标。

RMC表示进程之间运行时交互的程度。进程级 RMC 是从一个进程发送到另一个进程的消息数量。而系统级 RMC 是所有通信进程对的进程级 RMC 平均值。

RCC 涉及一个进程中某个类的方法访问其他进程中方法的情况。类级远程类耦合度是第一个进程中第一个类里依赖于第二个进程中第二个类任一方法的方法总数，与除第一个进程外任一进程中的方法总数之比；而进程级 RCC 指标则设为该进程所有类中所有方法的完整依赖集并集的规模，其中分子为这些方法的远程依赖集并集的规模。此外，系统级 RCC 指标是所有进程级 RCC 度量的平均值。

CCC 涉及某个类对远程进程中其他类的影响程度。类级 CCC 是该类与所有远程进程中其他类之间所有 RCC 度量的总和。系统级 CCC 是进程级 CCC 度量的平均值。

对于进程间代码重叠与复用，方法级 IPR 是本地依赖集（进程中依赖给定方法的方法集合）与远程依赖集（依赖给定方法、但属于第一个进程以外任意进程的方法集合）的交集，除以执行中涵盖的整个方法集合的规模所得的比值。

表 3.1： IPC指标总结

	类型 & 定义 & 计算	合理性说明	NS. L.
RMC	耦合 & 进程间消息耦合 & 消息数量 从一个进程发送到另一个进程	运行时交互的 进程之间的	进程
RCC	耦合，以及两个进程之间的类耦合，并进一步在系统 层级上，第二个类中依赖于 第一个类中任一方法的方法总数与方法总数之比， 这些依赖于第二个类中的任何方法以及所有进程中的方法 但第一个进程	一个进程中的类的方法如何 一种进程访问方法 in 其他进程中的方法	类 ， 进程
CCC	耦合与聚合耦合，涉及单个类的执行方面 各远程进程中的类，聚合 该类与各远程进程中其他类之间的 RCC 度量	类的重要性 相对于其耦合而言 强度	类
IPR	方法级别的耦合与进程间耦合，以及 本地和远程依赖集的交集除以 已执行的方法	代码重叠与复用 跨进程	方法
CCL	耦合与单个类与其他进程通信时产生的通信负载 以及所有远程进程中远程依赖集的规模总和除以所有已执行 类的方法的依赖集合数量除以所有已执行项的集合规模 类中的方法集合的规模	一个类的贡献程度 to 通信 负载 进程之间	类
PLC	单个进程内的内聚性与内部连接，以及.....的总和 进程的方法的本地依赖集的规模除以 进程中所有已执行方法的集合的规模	进程的方法彼此 归属的程度 在一起	进程

然后，将系统级 IPR 指标设为执行中所有方法的方法级 IPR 度量之和，除以这些方法的整个集合规模所得的比值。

CCL 衡量一个类对执行该类的进程与其他进程之间通信负载的贡献程度。CCL 越高，表示系统组件之间的通信越多。类级 CCL 的计算方法是：将该类各方法的远程依赖集规模相加，再除以该类所有已执行方法集合的规模。系统级 CCL 是所有已执行类（跨所有进程）的类级 CCL 指标的平均值。

与之前所有关于耦合的度量不同，*PLC* 衡量的是内聚性。它表示一个进程中已执行的所有方法彼此相关的程度。进程级 PLC 的计算方法是，将该进程中所有方法的本地依赖集规模相加，

然后，系统级 PLC 是所有进程的进程级 PLC 度量值的平均值。

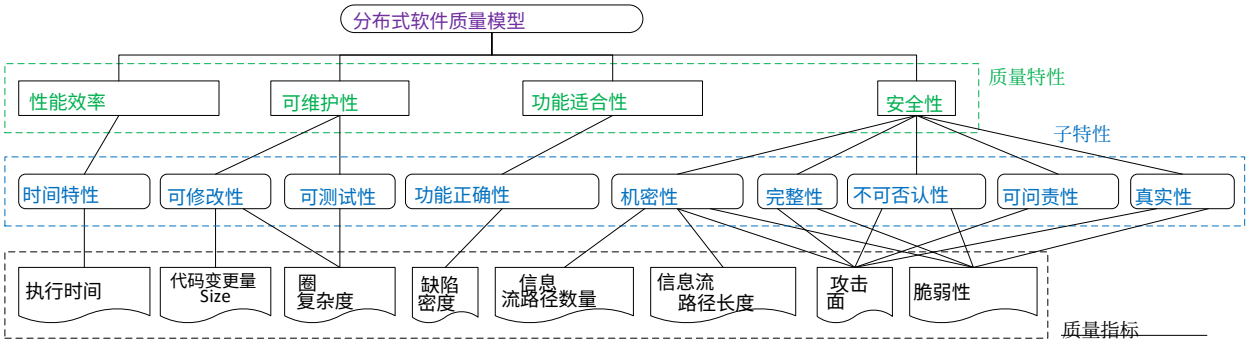
其中一些度量（即 RMC、 RCC）明确衡量进程间耦合，而另一些度量（即 CCC、IPR 和 CCL）则是导出耦合度量。 IPR 衡量两个进程的共同依赖，这与另外四种耦合度量（即 RMC、 RCC、 CCC 和 CCL）不同；后者衡量一个进程对另一个进程的相互依赖关系。

然而，与衡量进程内聚不同，在分布式系统中衡量进程间耦合既有其特殊性，也颇具挑战性 [41]。耦合度量通常基于某些关系（例如依赖和继承）来定义 [222]。然而，在分布式系统执行的上下文中，推导出计算这些度量所需的进程间依赖关系并非易事。主要原因在于系统缺乏全局时序，同时分布式组件之间也缺少显式引用或调用。为克服这一挑战，我采用了一个用于分布式程序动态依赖分析的框架 [38]。借助该框架，我通过执行不同进程中的方法之间的先发生关系来推理进程间依赖关系；这种关系源自方法执行事件的全局偏序。我还进一步利用消息传递的语义，提高此类导出依赖关系的精确率，从而增强 IPC 耦合度量和内聚度量的有效性。

3.2.3 Quality metrics correlated to IPC metrics

为了展示耦合和内聚性度量在辅助分布式系统质量分析方面的实用性，我采用了标准质量模型 ISO/IEC25010 [126] ，并进行了必要的定制选择，如图3.2所示。该模型

图3.2： DistMeasure所依据的参考质量模型，采用自 ISO/IEC 25010 [126] 标准并符合该标准



该模型分为三层：质量特性（即因素）、子特性（即子因素）和质量指标。前两层采用标准规定的质量特性/子特性名称（见 ISO/IEC 25010 [126]），而底层则根据 DistMeasure 中使用的质量指标名称进行定制。

有四项质量特性（即 性能效率、可维护性、功能适用性和安全性）、九项子特性（即 时间特性、可修改性、可测试性、功能正确性、机密性、完整性、不可否认性、可问责性和真实性），以及八项质量指标（即 执行时间、代码变更量、圈复杂度、缺陷密度、信息流路径数量、信息流路径长度、攻击面和 脆弱性），其中一些指标用于衡量多个子特性。

DistMeasure 使用直接指标 执行时间 来衡量分布式系统的时间特性，然后根据被分析系统的逻辑源代码行数（SLOC）对其进行规范化。

版本级代码变更量是程序 [248]两个相邻版本之间发生变更（即新增、删除或更新）的源代码行数。随后，使用后一个版本的逻辑源代码行数对其进行规范化。最后，系统级代码变更量为

该程序所有版本的版本级代码变更量的平均值。

特定系统的圈复杂度 是一项直接质量指标 [163], ，以该系统的逻辑源代码行数进行规范化。

特定系统的缺陷密度 是与该系统相关的缺陷数量，以系统的逻辑源代码行数进行规范化。

信息流路径数量 是动态信息流路径的总数，而信息流路径长度 则通过计算动态信息流路径的平均长度得出。

攻击面 是一项质量指标，从方法、信道和数据三个维度衡量系统的相对安全性，以确定该系统是否比其他系统更安全 [158]。首先，我考虑一个三元组 <**N1**、**N2**、**N3**>，其中**N1** 是包含源/汇（即入口/出口点）的方法数量，**N2** 是所使用的网络端口（即信道）数量，而**N3** 是对象在执行期间读取/写入的文件数量。接下来，我计算该三元组与原点 **(0,0,0)** 之间的欧几里得距离，即 $\sqrt{N1^2 + N2^2 + N3^2}$。最后，将该距离按对象的逻辑源代码行数进行规范化，得到系统级攻击面。

大多数被测程序的脆弱性依据公共安全数据库（例如国家漏洞数据库（NVD） [67]）中的 CVSS 2.0 评分，以及漏洞在 CVSS 中的时间来确定。如果某个漏洞是最近发现的，其受关注程度应高于很久以前发现的漏洞。然而，NVD 并未收录某些被测程序（例如 Grizzly、OpenChord、Voldemort、XNIO、xSocket）的缺陷。因此，也应考虑其他在线缺陷资源（例如 Jira、SourceForge、GitHub）。被测程序的脆弱性计算如下：

$$N_{nn} + \sum(CVSS_v * (100 - Diff_v/100)) \tag{3.1}$$

其中， N_{nn} 表示在非 NVD 缺陷源（没有 CVSS 评分）中发现的漏洞数量， $CVSS_v$ 表示漏洞 v 的 CVSS 评分， $Diff_v$ 表示当前年份与发现年份 v 之间的差值，

随后，我探究了 IPC 测量与通过相应直接测量量化的八项质量指标之间的关系，以展示 IPC 测量的实用性及其与分布式软件质量测量之间的统计关系，并证明 DistMeasure 能够通过深入刻画分布式软件中与 IPC 相关的行为，帮助理解软件质量。

为此，我进行了广泛的统计分析，以考察六项进程间通信指标与八项质量指标之间各组配对的相关性。我选择了斯皮尔曼等级相关分析 [199]，而非其他方法（例如肯德尔 [1], 、皮尔逊 [32] ）。原因在于，前者不假设两个相关变量之间存在任何关系。

此外，我根据斯皮尔曼等级相关系数 r 在 [197] 中的取值评估相关强度：如果 $|r| >= 0.4$，则相关性显著；否则（即 $|r| < 0.4$），相关性较弱。此外，还计算了 p 值，用于表示相关系数的统计显著性。

3.2.4 Classification for understanding and predicting quality

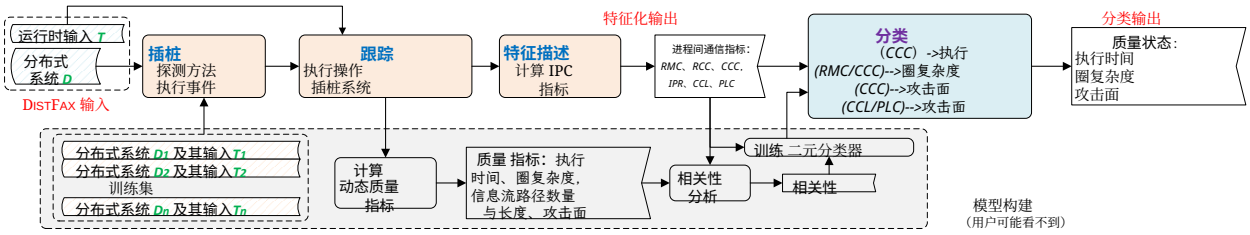
在理解 *IPC*（第二部分）中， DistMeasure 首先根据 IPC/质量度量训练无监督学习和监督学习模型，然后利用这些模型对与 IPC 指标显著相关的质量指标进行分类。

学习特征。提出的六种 IPC 指标（即 RMC、RCC、CCC、IPR、CCL 和 PLC）是用于对与一个或多个 IPC 指标显著相关的直接质量度量进行无监督和监督分类的潜在特征。正相关和负相关均纳入考虑。在 DistMeasure的**第 2 部分**中，仅使用系统级度量值作为特征值。

无监督学习。将具有相似特征的数据点归为一组，并将数据点划分为 k 个簇，方法是将它们分配到最近的簇中心。 [183]k-均值聚类算法（由 Python Scikit-learn库 [179]实现）被选用于无监督学习质量分类器。我设置 $k = 2$ ，因为 DistMeasure 旨在将质量指标分类为 `normal` 或异常，以检测质量异常并进一步了解系统质量。

监督学习。在比较探索了若干替代方案后，我选择了自助聚合（简称装袋法）来构建监督式质量分类器。装袋法通过对训练数据集进行自助抽样，生成多个模型版本，并将这些版本用作新的训练数据集，从而实现较高的预测准确率，尤其适用于不稳定的数据——即使训练数据 [35]存在较大变化，准确率仍能保持。分布式系统涵盖各种应用领域、执行场景和规模。因此，在系统执行中，预计不同训练样本之间会存在较大差异，而这类情况适合采用装袋法。与那些无监督分类器一样， DistMeasure 中的监督分类器也是二元分类器，可用于发现质量异常，进而预测和理解分布式系统的质量。

图3.3： DistFax 架构概述



3.3 工具实现

为了描述通用分布式系统，我实现了 DistFax （Distributed software systems facts 的简称，读音类似 Fax） [92], ，这是一个用于描述通用分布式系统的工具包，关注系统运行时行为的一个重要方面：进程间通信（IPC）。如图3.3所示， DistFax 通过定义的 IPC 指标衡量系统耦合度和内聚性，并参照标准质量模型 ISO/IEC 25010 [126]，使用动态质量指标描述系统的运行时质量。随后， DistFax 分析 IPC 指标与质量指标之间的统计相关性。此外，它还利用这些相关性构建学习模型，根据系统的 IPC 指标对系统质量状态进行分类。

3.4 评估

3.4.1 Experiment Setup

我测量了九个真实世界分布式软件项目的进程间通信特征，这些项目大多是企业级规模的系统。表3.2列出了被测程序名称/版本（**第一列**）、逻辑源代码行数（**第二列**）、测试类型（**第三列**）以及测试输入数量（**最后一列**）。

表3.2： DistMeasure中实验对象的统计数据

对象（版本）	逻辑源代码行数	测试类型	测试输入数
XNIO（2.0.0）	3,963	集成	247
OpenChord（1.0.5）	6,391	集成	1,999
xSocket（2.8.15）	11,628	集成	1,698
QuickServer（1.4.6）	13,369	集成	34
Thrift (0.11.0)	13,543	集成	525
Grizzly (2.4.0)	22,725	集成	2,000
Karaf (2.4.4)	46,810	集成	26
ZooKeeper（3.4.11）	50,577	集成	2,000
		Load	1,409
		系统	2,000
Voldemort（1.9.6）	66,754	集成	1,631
Netty (4.1.19)	109,450	集成	1,919
Derby（13.1.1）	423,662	集成	1,392

对于所有被测程序，我都通过整理其官方网站快速入门指南中的步骤来创建集成测试。具体而言，对于六种框架/库（XNIO、xSocket、QuickServer、Thrift、Grizzly 和 Netty）的集成测试，我开发了应用程序以覆盖其主要函数，然后运行了所有这些应用程序。此外，ZooKeeper 的负载测试和系统测试随项目包一同提供。现在，我将介绍所有评估对象、其原始集成测试操作，以及相应扩展后的运行时输入。

1. XNIO 是一个非阻塞 I/O 库，用于构建高效的网络应用程序 [211]。在其原始集成测试中，我启动了一个服务器和一个客户端，然后从客户端向服务器发送任意文本消息。为了扩充测试套件，我创建了 2,000 个文件，每个文件都包含随机生成的不同文本内容。每次执行时，XNIO 客户端都会读取一个文件，并将其全部内容发送到服务器。

2. OpenChord 是一项点对点网络服务 [63]。在最初的集成测试中，我

首先启动了节点 A、B 和 C。然后，我执行了以下操作：在节点 A 上创建覆盖网络；在节点 B 和 C 上加入该网络；在节点 C 上向网络插入新的数据条目；在节点 A 上搜索并删除该数据条目；最后，在节点 B 上列出所有数据条目。为了扩充测试套件，我创建了 2,000 个文件，每个文件都包含一组 OpenChord 命令（例如，`retrieveN -key test`）。其中一些文件包含无效命令或命令组合，用于构造格式错误的输入。在每次执行期间，节点都会从一个文件中读取命令，然后按顺序执行这些命令。

3. xSocket 是一个基于非阻塞输入/输出的库，用于构建高性能计算（HPC）软件 [118]。在其原始集成测试中，启动一个服务器和一个客户端后，客户端向服务器发送了一系列手动编写的文本消息。我采用与 XINO 相同的方式扩展了测试套件。

4. QuickServer 是一个开源库，供用户快速开发多客户端 TCP 应用程序 [23]。在其原始集成测试中，启动服务器后，客户端连接到服务器，然后向服务器发送了一组文本消息。我以扩展 XINO 测试套件的方式扩展了原始测试套件。

5. Thrift 是一个用于开发可扩展跨语言服务的框架 [187]。在其原始集成测试中，我使用其库开发了一个由服务器和客户端组件组成的计算器。我通过客户端运行该计算器，测试基本算术运算（即加法、减法、乘法和除法）。为了扩展测试套件，我创建了 2,000 个文件，每个文件都包含一个不同的算术表达式（例如 `1 add 2 minus 3 multiply 4`），其中一些无效表达式用于表示格式错误的

输入。在每次执行过程中，Thrift 客户端从一个文件中读取表达式，将其发送到服务器，然后从服务器获取计算结果。

6. Grizzly 是 GlassFish 社区 [180] 推出的基于非阻塞输入/输出的服务器框架。在其原始集成测试中，我启动了一个服务器和一个客户端，然后从客户端向服务器发送随机文本消息，最后等待每条消息的回显。我扩展了原始测试输入，方式几乎与 XNIO 相同，唯一不同的是增加了一条命令，用于等待客户端发送的每条消息的回显。

7. Karaf 是一个模块化 Container，也是支持标准开放服务网关倡议（OSGi） [177] 的开源运行时环境。在其原始集成测试中，我创建了一个由服务器承载的 Container，然后执行了两条命令：列出所有软件包（`la`）和列出 OSGi 捆绑包（`list`）。为了扩展原始输入集，我创建了 2,000 个文件，每个文件都包含各种 Karaf 命令（例如 `config:proplist`）。我特意在部分文件中加入了无效命令，以构造异常输入。每次执行期间，Karaf 客户端都会从一个文件中读取命令，并按顺序执行。

8. ZooKeeper 在分布式系统中实现一致性和同步 [125]。在原始集成测试中，操作包括：创建两个节点、搜索它们、查找其属性、更新其数据关联，并删除这两个节点。为了扩展集成测试输入，我创建了 2,000 个文件，每个文件都包含各种 ZooKeeper 命令（例如 `ls /zk-temp`），其中也包括无效命令。每次执行集成测试时，ZooKeeper 客户端都会从一个文件中读取命令，然后按顺序执行。

在原始负载测试中，服务器启动后，我启动了一个容器实例，然后生成工作负载。为了扩展负载测试套件，我使用 2,000 组不同的配置参数生成工作负载（例如 `the number of clientsinvolved in the load, the request size`）。每次执行负载测试时，我都会使用一组参数运行负载测试。

在原始系统测试中，我启动了一个服务器实例和一个系统测试容器，然后运行系统测试。为了扩展系统测试套件，我创建了 2,000 个配置文件，每个文件都包含不同的配置参数（例如， `initLimit`、 `tickTime`、`syncLimit`）。在每次系统测试执行期间，我运行原始系统测试代码，但只使用一个配置文件。

9. Voldemort 是 LinkedIn 的服务 [219] 所依托的分布式键值存储。在其原始集成测试中，我依次执行了以下操作：添加一个键值对、查找与其值对应的键、删除该键，然后检索该键值对。为了扩展原始测试套件，我创建了 2,000 个文件，每个文件都包含各种 Voldemort 命令（例如， `getmetadata`）。对于无效输入，还加入了无效命令或命令序列。每次执行期间，Voldemort 客户端都会从一个文件中读取命令，然后逐一执行这些命令。

10. Netty 是一个用于快速高性能计算应用开发的框架 [162]。在其原始集成测试中，启动服务器和客户端后，我从客户端向服务器发送了文本消息。我用同样的方法扩展了 XINO 的原始输入集。

11. Derby 是一个开源关系型数据库 [48]。在其原始集成测试中，我从一个关系型数据库中搜索了所有数据记录（`SELECT *`）（该数据库事先创建，包含一张

表）。为扩展原始测试，我创建了 2,000 个文件，每个文件都包含一组与 Derby 兼容的不同 SQL语句（例如，`show settable roles`）。我还加入了无效 SQL语句或无效语句序列，以构造无效输入。每次执行时，Derby客户端都会从一个文件中读取 SQL语句，然后按顺序执行这些语句。

每个被测程序和测试类型对应 26 至 2,000 个测试输入（**最后一列**），这些评估对象的测试输入总计为 16,880 个。这 16,880 次被测程序执行构成了 DistMeasure 所有实验的基础。

使用了两个动态分析器 [38, 89] 来检测信息流路径，因此基于这些路径采用了两个直接质量度量（即信息流路径数量和信息流路径长度）。根据 JDK 安全/密码学 API 生成了相关的源和汇。使用源代码度量工具 LocMetrics [17, 135] 计算分布式系统的规模（即逻辑源代码行数）。为计算每次被测程序执行的动态（运行时）圈复杂度，我对被测程序进行了插桩，执行其插桩版本，并统计执行过程中经过的简单条件判断数量。使用比较工具 `diff` [24] 计算每个被测程序相邻版本之间的代码变更规模。所有评估实验均在 Ubuntu 18.04.1 工作站上进行，每台工作站均配备 256GB 动态随机存取存储器和 2.27GHz CPU。

3.4.2 IPC measurements

IPC 测量结果汇总于表 3.3 中。每个数值代表针对一个对象测试计算出的六项拟议 IPC指标之一。

表 3.3：系统级 IPC 测量结果

被测程序执行	RMC	RCC	CCC	IPR	CCL	PLC
XNIO	16.82	58.96	2.76	0.51	74.43	41.10
OpenChord	3.14	67.15	5.29	0.78	267.36	255.78
xSocket	20.74	76.77	3.63	0.36	257.98	208.19
QuickServer	2.12	36.21	2.42	0.40	113.27	70.85
Thrift	11.59	25.27	3.17	0.56	23.40	41.59
Grizzly	39.68	152.09	2.16	0.67	665.13	673.34
Karaf	2.85	22.31	1.06	0.45	70.16	78.32
ZooKeeper	6.26	191.83	3.04	0.42	506.32	461.36
ZooKeeper 负载	4.01	90.02	1.19	0.37	391.41	369.29
ZooKeeper 系统	4.84	131.32	2.62	0.39	382.56	332.70
Voldemort	40.17	301.54	5.02	0.54	528.32	569.79
Netty	1.00	129.00	2.38	0.54	863.39	765.36
Derby	3.31	29.45	2.22	0.72	717.70	734.12

两个真实世界分布式系统 Grizzly 和 Voldemort 的 RMC 值最大。这是因为在集成测试执行过程中，这两个系统表现出最高层级的组件间依赖，并且进程之间存在大量消息交换（以及相应的网络通信）；这些测试需要进程紧密协作，因此 RMC 层级最高。此外，RMC 值似乎通常与系统规模（以逻辑源代码行数衡量）无关。例如，在相同测试类型（即集成测试）下，最大系统 Derby 的 RMC 值小于最小系统 XNIO 的 RMC 值。

Voldemort 的 RCC 值最大，因此其类级别的进程耦合最高，其次是 ZooKeeper；而 Karaf 的 RCC 值较小，这意味着在 Karaf 进程中，一个类几乎不会影响其他进程中的类。RCC 值与 RMC 值以及按逻辑源代码行数衡量的系统规模似乎没有显著关联。

从数量上看，平均 CCC 大多在 1 到 5 之间，这意味着每个类都会与远程进程中的约 1 至 5 个其他类协作。

OpenChord 的 IPR 值最高（0.78），而 xSocket 的值最低（0.36）。这意味着 OpenChord 的进程之间存在大量功能共享。相反，xSocket 的进程之间的共同依赖关系非常少。

CCL 值介于 23 到 863 之间，这意味着每个类的通信负载是其他远程进程中类的 23 至 863 倍。Netty 的 CCL 值最高，为 863；Thrift 的 CCL 值最低。这表明 Netty 的进程间通信复杂度最高，因此 Netty 的 IPC 正确性/性能调试也最困难。相比之下，Thrift 的进程间通信复杂度最低，因此更容易调试。

我发现 PLC 值介于 41.10 到 765.36 之间，这意味着一个进程的方法依赖于同一进程中约 41 至 765 个其他方法。我还观察到，通常 CCL 值高时 PLC 值也高，CCL 值低时 PLC 值也低。原因在于评估中的执行整体上体现了进程级耦合与内聚性之间的接近关系，其中 CCL 衡量耦合，PLC 衡量内聚性。

总之，我发现四项 IPC 指标（即 RMC/RCC/CCC/IPR）既与研究对象规模无关，也与其他任何 IPC 指标无关。这表明，这四项 IPC 指标之间的关联很少，而且 RMC/RCC/CCC/IPR 中的每一项几乎不会影响其他指标。此外，我观察到 RMC/RCC/CCL/PLC 与 CCC/IPR 之间存在明显差异：CCC/IPR 只有很小的变化，RMC/RCC/CCL/PLC 却会随之大幅波动。这表明，RMC/RCC/CCL/PLC 在不同系统中的变化范围相对较大，而 CCC/IPR 的变化很小。

表 3.4： 两个直接质量度量的测量结果：信息流路径数量和信息流路径长度

主体执行	Thrift	xSocket	Voldemort	ZooKeeper 负载	Netty
信息流路径数量 2.22E-04	1.72E-04		6.28E-04	1.27E-03	1.83E-05
信息流路径长度 1.26E-02	4.69E-03		5.43E-03	9.22E-03	2.26E-05

3.4.3 Quality measurements correlated to IPC measurements

信息流路径数量是此类路径的总数，信息流路径长度则是这些路径的平均长度。二者均按主体逻辑的千行源代码（KSLOC）进行规范化，如表 3.4 所示。

所有被测程序中六项质量指标（信息流路径数量和长度除外）的直接度量结果汇总于表3.5。量化某一质量因子时，我根据该质量因子度量值的取值范围，选择逻辑源代码行数/千行源代码作为归一化单位，以避免度量值过小而被忽略。

随后，我采用斯皮尔曼方法 [199]进行非参数相关性分析，开展了大量统计分析，以发现耦合度量与质量指标之间的显著关联。通过经过实证验证的相关性，我展示了 IPC 指标在预测和理解复杂分布式系统的质量与行为方面的应用前景。结果显示，6 个 IPC 指标与 8 个质量指标在 6 x $8 = 48$ 对组合中呈现出差异很大的相关性。我在表 3.6 中以**粗体**标出了显著相关性。

根据结果，六项 IPC 指标中的五项（即 RMC、RCC、CCC、CCL 和 PLC）与八项质量指标中的六项（即执行时间、代码变更量、圈复杂度、

表3.5： 其他六项直接质量度量（按逻辑源代码行数或千行源代码规范化）

对象	执行时间（秒）	代码变更量 Size	圈复杂度	缺陷密度	攻击表面		脆弱性
XNIO	1.51E-03	9.28E-02	2.03E-01	9.34E-03	4.55E-03		1.01E+00
OpenChord	8.45E-03	8.87E-02	1.72E-01	2.35E-03	1.13E-02		6.26E-01
xSocket	9.46E-04	4.83E-02	2.38E-01	5.16E-04	1.98E-03		2.58E-01
QuickServer	8.23E-04	4.73E-02	2.45E-01	2.99E-04	7.10E-04		4.38E-01
Thrift	5.91E-04	6.57E-02	1.45E-01	4.28E-03	2.51E-03		8.14E+00
Grizzly	3.08E-04	7.81E-02	2.21E-01	1.98E-03	6.30E-04		8.80E-01
Karaf	5.13E-04	1.17E-02	1.66E-01	9.08E-03	3.01E-04		1.96E+00
ZooKeeper	4.60E-04	1.37E-02	5.17E-02	7.41E-03	6.16E-03		9.00E-01
Voldemort	4.19E-04	3.97E-03	2.42E-01	4.93E-04	4.61E-03		1.80E-01
Netty	1.10E-04	1.11E-02	2.33E-01	5.03E-03	7.13E-04		1.80E-01
Derby	5.90E-05	2.70E-02	1.55E-01	9.70E-03	4.91E-04		5.12E-01

缺陷密度、攻击面和脆弱性）显著相关，并且能够提供相关信息，尽管相关性强度各不相同。基于进程与其他进程之间依赖关系的高 IPC 耦合（即高 RMC/RCC/CCC）与较长的执行时间、较高的复杂度、较低的缺陷密度、较大的攻击面以及较少的已报告漏洞显著相关。此外，类与远程进程中的其他进程之间存在大量通信负载（即高 CCL），表明攻击面较小、脆弱性较低。同时，单个进程中的高内聚性（即高 PLC）意味着代码变更量较小、攻击面较小且漏洞较少。

根据表 3.6，我基于用于预测动态质量指标的 IPC指标，构建了四个分类器（IPC指标 -> 对应质量指标）：(CCC) -> 执行时间、 (RMC,CCC) ->(运行时) 圈复杂度、 (CCC) -> 攻击面，以及 (CCL, PLC) -> 攻击面。对于静态质量指标，我以类似方式构建了一个分类器：(CCL, PLC) -> 脆弱性。

此外，相关系数的 p 值表示该系数不显著的概率。如果 p 值小于 5%，则该相关系数称为具有统计显著性

表 3.6： IPC 与质量指标之间的 Spearman相关系数（p 值）

质量指标			进程间通信指标					
层级	**Type**	**Name**	**RMC**	**RCC**	**CCC**	**IPR**	**CCL**	**PLC**
系统	静态	代码变更量	0.2320 (4.46E-01)	-0.3260 (2.77E-01)	0.2541 (4.02E-01)	0.2707 (3.71E-01)	-0.3978 (1.78E-01)	**-0.4972** (8.38E-02)
		缺陷密度	-0.2210 (4.68E-01)	-0.3149 (2.95E-01)	**-0.4475** (1.25E-01)	0.0497 (8.72E-01)	0.0166 (9.57E-01)	0.0331 (9.14E-01)
		脆弱性	0.0553 (8.58E-01)	**-0.4205** (1.53E-01)	-0.2407 (4.28E-01)	-0.0470 (8.79E-01)	**-0.6224** (2.31E-02)	**-0.5616** (4.58E-02)
执行动态		执行时间	-0.2845 (0.00E+00)	-0.0864 (2.39E-29)	**0.4735** (0.00E+00)	0.0461 (2.11E-09)	-0.1554 (1.10E-91)	-0.2471 (3.32E-233)
		圈复杂度 复杂度	**0.4239** (0.00E+00)	0.0573 (9.16E-14)	**0.4831** (0.00E+00)	0.1988 (5.77E-150)	-0.3644 (0.00E+00)	-0.3941 (0.00E+00)
		信息流 路径计数	0.1295 (4.99E-64)	0.0777 (4.97E-24)	-0.0091 (2.38E-01)	-0.3771 (0.00E+00)	-0.1327 (3.31E-67)	-0.1570 (1.15E-93)
		信息流 路径长度	-0.1616 (3.45E-99)	0.0349 (5.77E-06)	-0.1159 (1.50E-51)	-0.2494 (1.24E-237)	0.1134 (2.07E-49)	0.0757 (6.59E-23)
		攻击面	0.1556 (5.35E-92)	-0.2132 (9.49E-173)	**0.4534** (0.00E+00)	-0.0868 (1.42E-29)	**-0.7424** (0.00E+00)	**-0.7968** (0.00E+00)

。如表 3.6 所示，五项（共六项）IPC指标（即 RMC/CCC/CCL/IPR/PLC）与所考虑的四项（共五项）动态质量指标（即执行时间、圈复杂度、信息流路径长度和攻击面）中的一项或多项之间存在显著相关性，其对应系数的 p 值为零（0.00E+00）。然而，对于静态质量指标（相对于 IPC 指标），由于静态数据点过少（仅 11 个），相应的 p 值（大多数为 > 0.05）无法表明系数具有显著性。

3.4.4 Classification for predicting and understanding quality

在本节中，我将介绍实验结果和结论，重点关注以大量（16,880 次）被测程序执行作为样本时，对动态质量指标进行分类的结果。然而，对于静态质量指标的分类，可用样本（即被测程序）只有 11 个，数量太少，无法训练出实用模型，因此也难以得出可靠结论。

表3.7： 无监督学习（k-均值）分类对动态可预测质量指标的有效性

模型		留出验证			10折交叉验证		
IPC指标	**质量指标**	**精确率**	**召回率**	**F1**	**精确率**	**召回率**	**F1**
CCC	执行时间	100.00%	37.84%	54.90%	83.33%	41.27%	55.20%
RMC，CCC	圈复杂度	73.23%	84.91%	78.64%	99.68%	66.85%	80.03%
CCC	攻击面	90.87%	83.22%	86.88%	98.10%	77.73%	86.73%
CCL, PLC	攻击面	99.73%	61.47%	76.06%	71.70%	81.01%	76.07%
平均值：		**90.96%**	**66.86%**	**74.12%**	**88.20%**	**66.71%**	**74.51%**

对于可预测动态质量指标的无监督分类器，表3.7展示了其精确率、召回率和F1准确率。对于该分类器，前两列显示 IPC 指标与-> 对应质量指标。第 3–5 列和第 7–8 列分别显示留出验证结果和 10折交叉验证（CV）结果。平均 F1值为 74%，表明使用无监督学习（k均值）算法，依据动态质量指标（即执行时间、运行时圈复杂度和攻击面）对分布式系统质量进行分类，通常能够取得实用的准确率。此外，两种验证结果的精确率、召回率和 F1准确率都非常相似（平均值分别为 90.96% 对 88.20%、66.86% 对 66.71%，以及 74.12% 对 74.51%），总体平均值和各个分类器的结果均如此。这意味着评估结果具有一致性，因此开发人员可以进一步确认结果的可靠性和有效性。

然而，无监督学习（k-means）算法在执行时间分类方面并未达到较高的效果 （召回率低于 42%，F1准确率低于 56%）。这表明，根据系统 IPC 测量结果（CCC），聚类算法在捕捉分布式系统的时间行为状态方面效果不佳。主要原因是（k-means）算法基于数据点之间的相似性，但 IPC 测量结果（例如 CCC）相近的系统执行，其执行时间

可能并不接近。因此，无监督学习（例如 k-means）分类可能不是通过 CCC 评估执行时间质量指标的最佳选择。

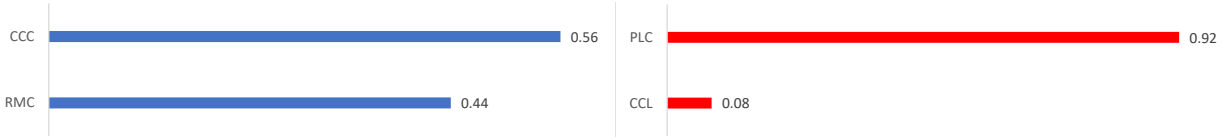
表3.8：监督学习（装袋法）分类对动态可预测质量指标的有效性

模型		留出验证			10折交叉验证		
IPC指标	**质量指标**	**精确率**	**召回率**	**F1**	**精确率**	**召回率**	**F1**
CCC	执行时间	99.80%	99.80%	99.80%	99.70%	99.70%	99.70%
RMC、CCC	圈复杂度	99.10%	99.10%	99.10%	99.70%	99.70%	99.70%
CCC	攻击面	96.50%	96.50%	96.50%	95.20%	95.20%	95.20%
CCL, PLC	攻击面	97.50%	97.50%	97.50%	96.80%	96.80%	96.80%
平均值：		**98.23%**	**98.23%**	**98.23%**	**97.85%**	**97.85%**	**97.85%**

与此相反，如表3.8所示，IPC指标与动态可预测质量指标之间的监督学习分类在预测这些质量指标方面更为有效。对于相同 IPC 指标与质量指标之间的相同分类任务，监督分类器在留出验证和10折交叉验证中均表现出较高的精确率、召回率和 F1准确率，且这些指标均超过95%。尤其是，总体平均值 F1准确率约为98%。与无监督分类结果相比，这些监督分类结果在这三项有效性指标上也都具有一致性，不仅适用于单个分类器，也适用于整体。由此可见，考虑到可用的带标签训练样本，监督分类器显然比无监督分类器更适合在 DistMeasure 中进行准确分类。

我使用 Python Scikit-learn库 [179] 计算监督分类器中的特征重要性分数 DistMeasure，即不纯度减少累积的标准差和均值 [42]（也就是 Python 中的特征重要性）。结果（即按重要性得分排列的特征）如图3.4所示。不过，有两个分类器“(CCC) -> 执行”

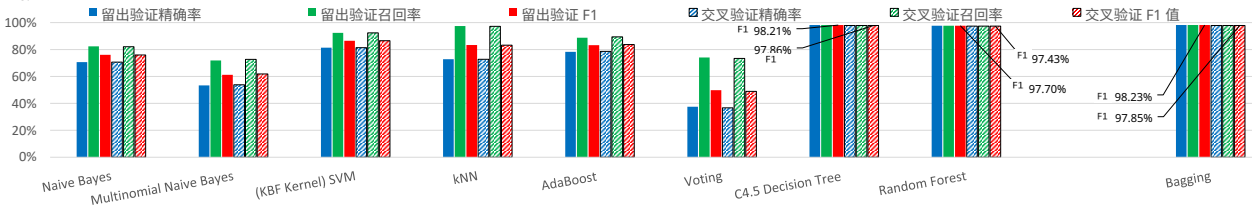
图3.4： 质量指标的监督式（装袋法）分类器中，按重要性得分（显示在 x 轴上）排列的特征（IPC指标）。左图显示了在分类运行时圈复杂度时 RMC 与 CCC 的重要性得分，右图则显示了在分类攻击面时 CCL 与 PLC 的重要性得分



和“时间 (CCC) -> 攻击面”都只有一个特征（CCC），无需进行特征选择。结果表明，正如前文所述，最强的 IPC耦合度量 CCC 在分类质量指标运行时圈复杂度时，明显比 RMC 更重要（重要性得分分别为 0.56 和 0.44）。此外，在 DistMeasure 中唯一的 IPC内聚度量 PLC，在分类质量指标 攻击面 时，比 CCL 重要得多（重要性得分分别为 0.92 和 0.08）。请注意，特征重要性和排名与 k 均值聚类无关，因此本研究仅针对我的监督分类器进行。

在基于 IPC 指标对动态、可预测的质量度量进行分类时，我将默认的监督分类（装袋法）算法与另外八种监督学习算法进行了比较。图 3.5 展示了在留出验证和 10 折交叉验证下，各算法的精确率、召回率或 F1准确率的总体平均值。从图中可以看出，在所考虑的所有监督学习算法中，装袋法 算法的性能最佳。具体而言，*C4.5*决策树 算法的 F1准确率排名第二，其次是随机森林；而投票法 和多项式朴素贝叶斯这两种算法的 F1准确率最差。无论采用留出验证还是 10 折交叉验证，其他算法在精确率、召回率或 F1准确率方面的分类效果都相近。

图 3.5： 替代监督学习算法与默认算法（装袋法）的平均准确率



3.4.5 Threats to Validity

根据 [236]，实证结果受到多种常见效度威胁的影响。下文将逐一介绍主要类型，并讨论我如何控制或减轻相关效度威胁。

内部效度。 对内部 效度的主要威胁在于：在测量/计算 IPC/质量度量指标、分析相关性以及对质量进行分类时，实现中可能存在错误。为降低这一威胁，我仔细检查了代码，并手动验证了两个最小型被测程序的分析/预测结果。特别是，我确认了用于 DistMeasure 计算的这些被测程序执行过程中的动态依赖关系正确无误。因此，我没有发现此类问题。

此外， DistMeasure 结果可能受到各种未予考虑的因素影响，这些因素却影响了收集数据的过程，而这些数据是直接量化所研究质量指标所需的。例如，为了测量代码变更量这一质量指标，我需要从各种来源收集系统发布/版本历史记录的数据。然而，某些系统版本可能无法在线获取。为减轻这一威胁，我仔细搜索并审查了所有可能的对象版本数据，并联系相关开发人员，请求他们提供缺失的版本。

某些特定质量指标的测量结果，对于我未考虑的其他可能相关的质量测量指标而言，可能并不有效。例如，目前使用运行时圈复杂度来衡量可测试性这一质量子特性，却未考虑其他可能影响可测试性的指标。具有相似（运行时）圈复杂度的系统，其可测试性未必也相似。例如，两个程序的代码相似，但读写各种外部文件。前者处理大型文件和复杂的数据结构，而后者处理简单文件（例如文本文件）。因此，它们的（运行时）圈复杂度接近，但可测试性不同。由于我仅关注与 IPC 相关的系统行为所涉及的质量指标，因此未考虑其他一些质量指标（例如可用性、网络吞吐量、响应时间和可扩展性）。

外部效度。 对外部 效度的主要威胁在于，我的研究结果可能无法推广到其他分布式程序和执行。为降低这一威胁，我选择了涵盖不同架构、领域和规模的被测程序。

由于运行时输入有限，针对**第 2 部分**中根据静态 质量指标对软件质量进行分类，存在相关的有效性威胁。使用不同的运行时输入时，IPC指标与静态质量指标之间的相关性，以及基于IPC指标对质量指标进行分类的分类器性能，可能不同于报告中的结果。

另一个外部效度威胁是，很难收集仅针对 ZooKeeper 的所有三种测试类型（例如集成测试、系统测试和负载测试）。幸运的是，每个被测程序都有可用的集成测试，并且这些测试覆盖了整个系统 的行为。此外，我还为被测程序手动构造了大量不同的运行时输入，以显著

提高所有被测程序的运行时覆盖率总和，从而降低这一威胁。

构念效度。 统计分析的使用可能导致我的结论，这是对构念效度的主要威胁。在计算系统级 IPC 指标时，我对低层级指标取了平均值，而没有考虑其变化情况（例如标准差）。为降低相关性分析所面临的威胁，我特意选择了斯皮尔曼方法，而非其他方法，因为它是一种非参数方法，不要求底层数据分布或数据组之间的关系满足正态性。

另一方面，尽管我大幅增加了被测程序的运行时输入，但这种扩充并未增加样本数量，因而无法为基于静态质量指标的质量分类器训练和测试提供更多样本。因此，对这些分类器的评估可能还为时过早。若要在这方面进行更有力的评估，我需要数千个不同的 Java 分布式系统作为被测程序。

结论效度。 分布式系统被测程序数量不足（仅 11 个），对静态质量指标分类器的结论效度构成了威胁。在构建这些分类器时，我测量了被测程序执行期间的 IPC 指标，并汇总了相应结果。这一过程忽略了同一被测程序多次执行中动态行为的不同特征，实际上将该被测程序的所有执行都视为其整体行为的体现。

这种聚合忽略了每个对象在不同执行过程中动态行为特征的差异，实际上将每个对象的所有执行都视为共同代表该对象的整体行为。这种处理方式可能会导致对这些分类器性能的评估产生偏差。

最终，质量分类器只能在较高层级上估计质量指标是否异常（是否值得发出警告）。目前， DistMeasure尚不足以在发出警告时精确定位此类异常。

3.5 经验教训与启示

我对 IPC 指标的探索不仅证明了在大型真实世界分布式系统中测量 IPC 的可行性，还揭示了分布式组件之间普遍存在隐式耦合（尽管程度各不相同；这些组件在架构设计中通常是解耦的）。我还指出，揭示这种隐式耦合的一种方法是测量进程间耦合。我的 IPC 测量结果表明，就进程间依赖而言，较高的耦合通常不利于质量（它显著表明，在考察的六个因素中，有五个因素的质量较低）。

具体而言，在分布式程序执行中，较高的进程间通信耦合通常与某些特性/子特性的低质量相关（例如，时间特性较差，进而导致性能效率低下；可修改性/可测试性较低，进而导致可维护性较差；安全性较低），也与相应质量指标的较差表现相关（例如，执行时间长、复杂度高、攻击面大）。这在很大程度上印证了高耦合的弊端，即它通常会导致质量较低 [132]。

相反，更高的进程级内聚性通过多个子特性（即更高的机密性、完整性、不可否认性、可追责性和真实性）以及相关质量指标（例如较低的攻击面和脆弱性），凭借更高的安全性提升了系统质量。这证实了先前的发现：较高的内聚性（在单个软件进程/组件中）通常与较高的质量相关 [97]。

总之，建议分布式系统开发人员降低并维持系统组件之间较低的整体（隐式）耦合，以实现较高的系统质量，尤其是在关注性能效率、功能适合性和安全性等质量特性时。另一方面，在追求更好的可维护性和安全性时，开发人员倾向于编写高内聚组件，以发挥更高进程级内聚性的优势 [126]。简而言之，低耦合和高内聚应是分布式系统开发人员的目标。

根据 DistMeasure **第 1 部分**的统计结果，对于一种静态可预测质量度量，只有一个分类器：(CCL, PLC) -> 脆弱性，详见 §3.4.3。然而，相应的数据集并不充分（仅有 11 个数据点，如表 3.5 所示），无法用于 DistMeasure **第 2 部分**中的无监督学习和监督学习。尤其是，由于样本太少，无法进行 10 折交叉验证。要训练/测试一个可靠的静态可预测质量度量（即脆弱性）分类器，还需要更多数据点和样本。

作为 DistMeasure六项进程间通信指标之一，IPR 与所考察的任何直接质量度量都没有显著相关性。不过，它仍有助于理解分布式系统中的运行时代码复用。与此同时，其他五项进程间通信指标已被证明能够反映分布式软件质量，至少与一项静态和/或动态质量指标相关。

尤其是，最能体现进程间耦合的指标 CCC，能够以细粒度级别捕捉所有分布式进程之间的耦合（而 RCC 衡量的是两个进程之间的耦合），其对分布式软件质量的指示作用往往最强（而 RMC 衡量的是粗粒度级别的耦合）。

事实上，实验发现，CCC 与质量指标的显著相关数量（四项）多于任何其他 IPC 指标。作为一项学习特征，在分类器使用多个特征的情况下，CCC 对质量分类的贡献也超过其他特征。同时，PLC 是该框架中唯一能够捕捉进程内聚的指标，其表现与 CCC 相近：研究发现，PLC 与三项质量指标显著相关，相关数量仅次于 CCC，且对相关分类器的贡献远高于其他特征。这些结果表明，在进程层级同时测量耦合和内聚，对评估分布式软件质量具有重要价值。

结果还表明，至少对于分布式系统中动态、可预测的质量指标，监督分类器明显优于无监督分类器。这一比较说明，仅根据不同系统执行中对应 IPC 指标的相似性来预测这些质量指标，可能并不奏效。主要原因在于，即使 IPC 测量值足够接近，不同的系统执行也可能产生截然不同的质量测量结果。相反，更好的方法是基于标注数据进行监督分类。

另一方面， DistMeasure 中监督分类优于无监督分类也要付出代价：处理相关质量指标值可能成本高昂，尤其是因为在 DistMeasure 的评估中，计算并标注为训练样本的此类值数量庞大。

此外，对于监督式（装袋法）分类器，计算得到的斯皮尔曼等级相关系数和特征重要性分数表现出一致性，如表 3.6 和图 3.4 所示。对于质量指标运行时圈复杂度，IPC指标 CCC 的

相关系数大于 RMC（0.448 对 0.42）；而在运行时圈复杂度 分类中，CCC 的重要性也高于 RMC（重要性得分分别为 0.56 和 0.44）。在分类器“(CCL, PLC) ->攻击面”中也观察到了类似的一致性——IPC指标 CCL 与攻击面的相关性更强，在分类器中的重要性也高于 PLC。

3.6 相关工作

在本节中，我将从三个类别讨论其他已有研究：动态耦合度量（§3.6.1）、运行时内聚（§3.6.2），以及基于机器学习的预测软件质量评估（§3.6.3）。

3.6.1 Dynamic coupling metrics

Jin等人 [130] 直接基于从带有时序信息的方法执行中导出的组件间依赖，定义了一种动态组件耦合度量（CPC）。从概念上讲，CPC度量与我们的 IPR度量密切相关，因为两者均基于跨组件的近似动态依赖。然而，根据[41]，作为我们 IPR 计算基础的进程间依赖，比 [130], 中近似得到的纯粹基于控制流的依赖精确得多。此外，CPC被定义用于度量结构复杂度，而 IPR则主要被提出作为一种可复用性度量。以往的复用度量主要关注库代码的复用以及服务器和客户端节点整体之间的连通性 [82]。相反，我根据基于代码依赖定义的度量，在代码级别度量进程间可复用性。

3.6.2 Run-time cohesion

Jin等人 [129] 还提出了一种用于分布式软件的动态内聚度度量方法，该方法通过扩展单进程程序的经典内聚度量方法缺乏内聚性（*LCOM*），包括两种组件级内聚度量（即 CC 和 CCW）。随后，还针对分布式软件引入了一种结构质量属性组件内聚因子（*CHC*）[130]。这些内聚度量仅针对专用分布式程序（例如 Netflix RSS Reader、RSS Reader Recipes 和/或 iBATIS JPetStore 的分布式版本）进行了评估 [129, 130]。所使用的底层度量工具也仅为这些专用系统而设计。相比之下，内聚度量 PLC 基于方法级动态依赖定义，并针对现实世界的通用分布式系统进行了评估。

3.6.3 Software quality assessment based on machine learning

软件质量预测有助于开发人员更好地利用资源、估算工作量，并为可能存在缺陷的组件制定测试计划，从而在早期阶段降低开发成本并减轻风险 [188, 208]。各种机器学习技术采用无监督或监督学习算法（例如逻辑回归、支持向量机、神经网络和 k-均值聚类）进行软件质量评估。

例如，Khoshgoftaar 和 Allen 提出了一个采用逻辑回归的软件质量评估模型 [137]。Xing 等人采用支持向量机技术，根据复杂度度量对软件模块进行分类，以预测软件质量，适用于

早期开发阶段 [191]。Abe 等人使用贝叶斯分类器来估计软件项目的成功或失败 [2]。此外，神经网络技术也被用于预测软件质量 [138, 224, 227, 208]。此外，Zhang 等人 [252]提出了 SMPLearner，该工具通过收集根据代码变更历史计算出的实际维护工作量，训练软件可维护性预测模型；它测试了 24 种通用机器学习技术，包括支持向量机回归、随机森林、 K-star 等。

作为一种软件质量评估模型，已有许多软件缺陷预测模型基于单个或多个软件项目构建，分别用于项目内或跨项目预测 [250, 251, 255], 。例如，Zhang 等人通过根据多个预测变量的分布相似性对项目进行聚类，推导出秩变换，并在预先转换的数据上拟合模型，利用来自不同上下文的大量软件项目构建了通用缺陷预测模型 [250]。

无监督学习模型不需要任何训练数据，因此不受项目间同质性要求（例如，度量指标的分布相似）所限 [251]。此外，对于开源软件系统中的一种特殊软件缺陷预测类型（即工作量感知的即时缺陷预测），一些简单的无监督模型表现优于有监督模型 [242, 83]。具体而言，与基于距离的无监督学习模型（例如， k-均值聚类）不同，基于连通性的无监督缺陷分类器建立在一种相似的直觉假设之上：有缺陷的实体可能聚集在同一区域周围 [251]。

与这些以往的缺陷预测模型不同，这些模型通常是静态的——它们依据的是项目特征，而非系统执行特征； Dist-Measure 中的质量分类器则是动态的，因为它们基于特定执行中的 IPC 测量。我还

与以往的缺陷预测研究不同，我将 IPC 特征作为系统行为的一个特定方面加以利用，而此前尚未有人对此进行研究。 DistMeasure也致力于解决分布式软件系统的质量问题，而早期研究并未特别关注这一问题。最后，与以往主要旨在预测某个软件单元（例如文件）是否包含功能缺陷的缺陷预测研究不同，我的质量分类器关注多种质量特性（包括功能性和非功能性特性）。同时，我记得， DistMeasure 的主要目标是支持并探索对 IPC 诱发行为进行测量，并从质量相关性的角度理解测量结果，而不是为分布式软件系统开发缺陷预测模型。

第四章

FlowDist：面向分布式软件系统的基于多阶段细化的动态信息流分析

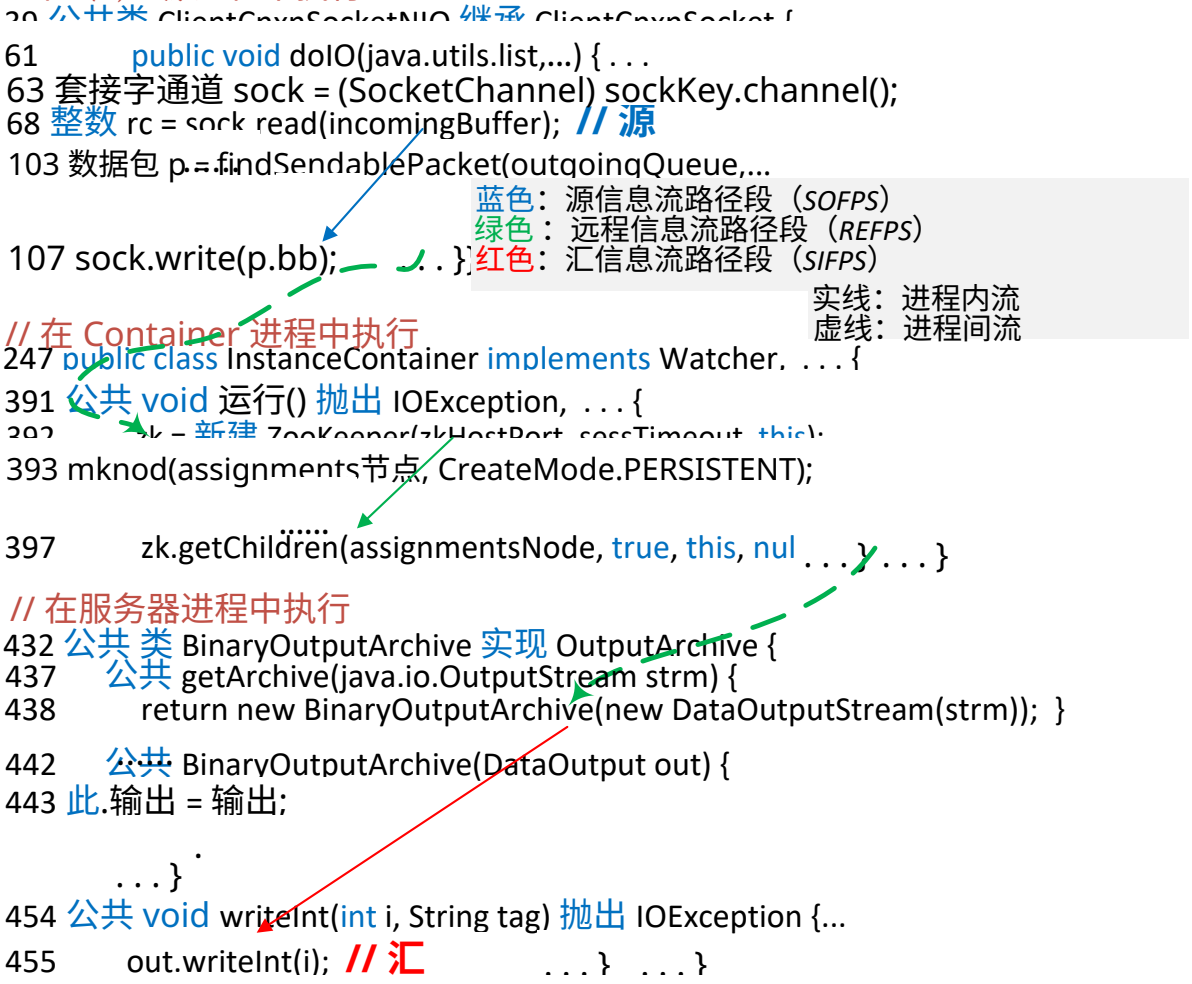
在研究 DistMeasure 的过程中，我发现进程间通信对于理解分布式系统中的信息流安全问题非常重要。DistMeasure 可以帮助我们间接了解潜在的信息问题。然而，它无法直接检测可能泄露敏感数据的信息流路径。这促使我在针对分布式系统的整体数据流分析中考虑进程间依赖和信息流路径。作为研究的下一步，我开发了一个面向分布式系统的动态信息流分析（DIFA）框架，该框架能够检测具体的敏感信息流路径。

4.1 动机

随着大规模计算需求的不断增长，分布式软件得到了越来越广泛的开发。与其他领域的软件应用程序一样，分布式软件也存在各种安全漏洞。例如，据 CVE-2018-8012 [65] 报告，一个真实的分布式系统 Apache ZooKeeper [7]，存在一项安全漏洞：*“ZooKeeper 服务器尝试加入仲裁组时，没有强制执行身份验证或授权……”*。相关的信息流如图 4.1 所示。

利用这一漏洞，攻击者可能轻易访问 ZooKeeper 服务器，并造成严重损害或损失。尤其是，与集中式程序相比，分-

图 4.1： Apache ZooKeeper 中敏感信息流（由带箭头的线标示）跨越三个组件（进程）的一个案例



分布式系统通常具有更大的代码规模，其解耦组件运行在物理上相互分离的机器上，且没有全局定时机制。这些特性以及其他因素增加了分布式软件的复杂度，使得保障这些系统的代码安全更加困难。

作为广受欢迎的替代技术， [133], 污点分析方法已被用于防御这些漏洞。它们帮助用户识别敏感数据可能泄露给不可信方的位置，这可通过污点信息流路径（即污点路径）揭示出来。然而，作为保守的污点分析器，静态污点分析工具常因现代软件中使用了动态语言结构（例如反射调用和动态代码加载）而可能缺乏可靠性。

相比之下，动态污点分析（DTA）被视为一种强大的软件安全技术，它比静态方法更精确，因为它会监控和/或计算程序执行期间实际发生的信息流 [194]。遗憾的是，这类技术大多是为单进程程序开发的，无法立即应用于或适配常见的分布式系统。适用性问题的一个主要原因在于，这些工具依据显式依赖关系计算信息流，而常见分布式软件中的分布式（解耦）组件之间并不存在这种依赖关系。

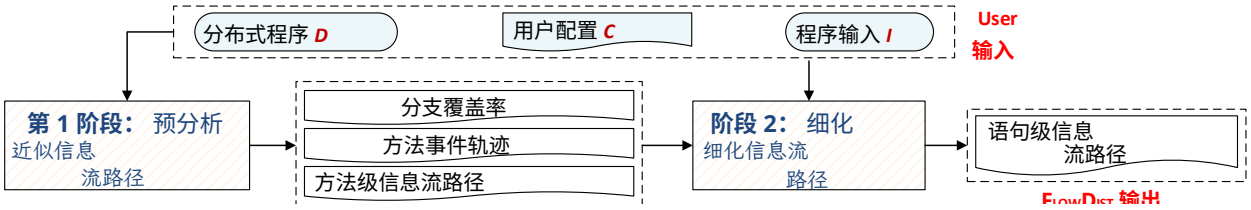
其他现有的动态污点分析器，如 Panorama [243]和 Taintdroid [77],，可能不受适用性 障碍的影响，但通常依赖于经过定制或修改（例如，经过插桩）的底层运行时平台（例如操作系统或 JVM）。因此，这些工具存在可移植性 问题——每次更新版本时，都可能需要再次修改运行时平台，这可能需要付出大量努力，而且并非总是可行。

在分布式系统中，对于一个信息流路径段，如果其语句或方法全部位于同一个组件/进程中，则称其为本地信息流路径段。相对地，远程信息流路径段 （*REFPS*）中的所有语句或方法都位于不同的组件/进程中。如果本地信息流路径段 包含一个或多个源语句或方法，则称其为源信息流路径段 （*SOFPS*）。相反，包含汇语句或方法的本地信息流路径段称为汇信息流路径段 （*SIFPS*）。

分布式程序中的信息流可能通过消息传递跨越解耦组件。图 4.1 展示了广泛使用的分布式协调服务 Apache ZooKeeper（3.4 版本）的一段代码，其中包含一条导致漏洞案例 CVE-2018-8012 [65] 的敏感流。

数据泄漏流经过了三个进程：（1）敏感数据（安全密钥）被读入*incomingBuff* 类 `ClientCnxnSocketNIO` 中，该类属于客户端进程（位于源处）；（2）流经 `InstanceContainer` 类，该类属于*Container* 进程；（3）到达 `BinaryOutputArchive` 类，该类属于服务器进程，数据在此处泄漏出系统（位于汇处）。数据泄漏导致 ZooKeeper 服务器尝试加入仲裁组时发生身份认证/授权失败，因此可能会向 ZooKeeper 领导节点传播伪造的变更。此外，信息流中的进程之间不存在显式依赖：客户端、*Container*和服务器之间没有显式依赖。不过，它们之间可能存在隐式依赖，如图 4.1 中虚线所示，这些依赖体现在进程间信息流中。

图 4.2： FlowDist 架构概述



4.2 方法

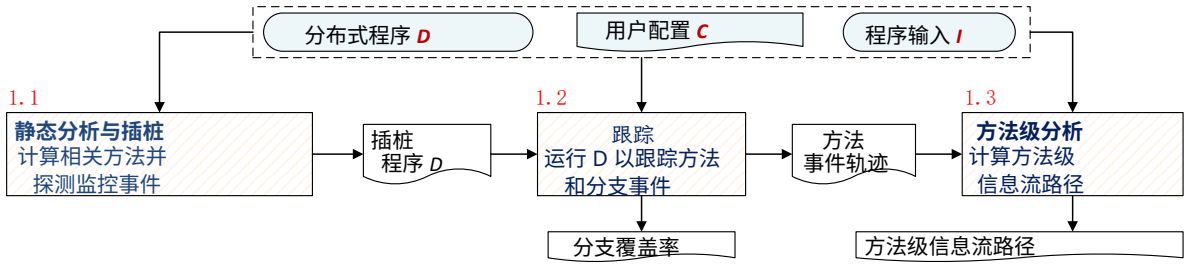
我开发了 FlowDist [89]，这是一种面向常见分布式软件的 DIFA，仅在应用层运行，以避免平台定制，从而实现高可移植性。它从全局偏序执行事件中推断隐式的进程间依赖，以适用于分布式软件。最重要的是，它通过引入一种面向应用层 DIFA 的多阶段细化方案，在保持有效性的同时实现了高可扩展性：该方案利用低成本预分析得出的方法级结果，降低了原本成本高昂的数据流分析的开销。

4.2.1 Overview

FlowDist 架构概述如图 4.2 所示。 FlowDist 需要三项**用户输入**：分布式程序 D 、用于驱动D的运行时输入集I ，以及用户配置C。此配置指定用户感兴趣的源和汇，以及一系列消息传递 API，供 FlowDist探针用于监控和分析进程间消息通信事件。借助这些用户输入， FlowDist 会根据I，分两个阶段计算C 中任意源与任意汇之间的信息流路径。

在第一阶段（**预分析**）中， FlowDist计算方法级流路径及路径上方法的语句级覆盖率，以避免不必要的高昂计算开销

图 4.3： FlowDist的第 1 阶段工作流程



（从而克服下一阶段的可扩展性挑战），缩小分析范围。随后，在第二阶段（**细化**）中，FlowDist根据预分析结果的指引，通过混合依赖分析推断语句级信息流路径，并将其作为最终 FlowDist **输出**（即细粒度的敏感流）。

4.2.2 Phase 1: Pre-analysis

本阶段的目标是通过粗略（保守）且快速的分析，提供粗粒度的（方法级）分析结果，从而降低下一阶段的成本，进而提升整体可扩展性。在本阶段，FlowDist 分三个主要步骤计算每个源-汇对（在配置C中定义）的分支覆盖率和方法级信息流路径，如图 4.3 所示。

步骤 1.1。在此步骤中，FlowDist在混合依赖分析中利用三种类型的动态数据，以平衡分析成本效益：(1) 进入（即程序控制进入某个方法）和返回至（即程序控制从被调用方法返回至调用方法）方法事件，(2)发送和接收消息事件，以及 (3) 语句分支覆盖事件。

FlowDist 通过插入探针来监控这些事件，从而生成程序 D的插桩版本 D' 。为了确定消息传递事件的探针位置，FlowDist

指的是C中的消息传递 API 列表。如果用户未提供该列表， FlowDist将使用 Java SDK 中最常用 API 的默认列表。只有位于源-汇对之间的静态控制流路径上的方法才会被插桩，这些方法称为相关方法。此外，只有相关方法中的语句分支（即相关分支）才会被插桩。

对于 D 中的每个分布式组件，都会构建一个过程间控制流图（ICFG）。组件中的每个消息发送和消息接收 API 调用点，分别被视为额外的汇和源。

步骤 1.2。 在此步骤中， FlowDist 记录每个方法的首次 进入事件和最后一次 返回调用方事件。之所以只监控这些事件，是因为它们足以推断所有方法执行事件之间的先发生关系，从而推断所有进程中相关方法之间近似的（基于控制流的）依赖关系 [41] 。同时，尽管消息传递事件本身不被跟踪，但在此步骤中会根据 Lamport 时间戳算法 [232]，对方法执行事件进行偏序处理，以便稍后确定任意两个方法执行事件之间的先发生关系。执行过程中每个 n 进程（P_i）都会生成方法事件跟踪（T_i）。此外，还会为 P_i 生成一个映射（$pid2fm[i]$），用于记录从另一个进程接收第一条消息的每个事件的时间戳（$p2fm[i][j]$） P_j（$i,\ j \in [1,n]$）。

步骤 1.3. 在此步骤中， FlowDist通过识别执行期间任意源与任意汇之间经历的方法序列，计算方法级信息流路径，并将其作为此阶段的输出。

算法 1 计算方法级流路径

令 SO 和 SI 分别为源和汇所在的方法列表
令 T_i 为进程 P_i, $i \in [1, n]$中的带时间戳的方法执行事件轨迹

1: $ps = \emptyset$ // 初始化给定配对之间所有方法级路径的集合
2: **对** i=1到 n **执行** // 遍历给定执行中的 n 进程
3: $S_d = \{s | s \in SO \wedge s \in T_i\}$
4: **if** S_d==$\emptyset$**then 继续**
5: **结束 if**
6: **对每个** 方法 $q \in S_d$ **执行** // 首先计算进程内依赖
7: $DS(q) = \{m | m \in T_i \wedge fe(q) \leq lr(m)\}$
8: **对于** j=1至 n **执行** // 然后计算进程间依赖关系
9: **如果** $i == j \vee p2fm[i][j] == null$ **则继续**
10: **结束 if**
11: $DS(q) \cup = \{m | m \in T_j \wedge fe(q) \leq p2fm[i][j] \leq lr(m)\}$
12: **结束 for**
13: **如果** $DS(q) \cap SI == \emptyset$ **则继续**
14: **结束如果**
15: $ps \cup = \{< m1, ..., m_k > | m1 == q \wedge m_k \in SI \wedge \forall_{i<j,\ i,j \in [1,k]} fe(m_i) \leq lr(m_j) \wedge \forall_{i \in [1,k]} m_i \in DS(q)\}$
16: **结束循环**
17: **结束循环**
18: **返回** ps

根据步骤 1.2 中的事件轨迹和映射， FlowDist 按照算法 1 计算方法级信息流路径。其核心思想是， FlowDist结合方法级控制流和进程级数据流，以近似动态方法级依赖关系。

该算法通过（第 2–15 行）遍历 n 每进程跟踪，以搜索路径 ps 。在每个跟踪 T_i 中，获取 S_d 被覆盖的源代码外围方法集合（第 3 行）。如果 P_i 中没有执行源代码，那么就不会有路径从 P_i 开始（对应于 T_i）（第 4 行）。否则，对于 S_d 中的每个方法 q ，算法通过计算其动态依赖集 DS（q），识别从 q 开始的路径。（第 6–13 行）。

令 $fe(m)$ 和 $lr(m)$ 分别表示方法 m 的首次进入事件和最后一次返回事件的时间戳。根据 q 与在 P_i 中执行的其他每个方法 m 之间的先发生关系，首先识别 本地 （进程内）依赖关系（第 7 行）。（将 P_i 视为一个本地 进程。）这样设计的原因是：如果 $m2$ 并非在 $m1$ [39] 之后执行，那么方法 $m2$ 就不依赖于方法 $m1$ 。

接下来，识别其他每个远程进程 P_j 中的依赖关系（第 8–11 行）。如果 P_j 从未发送消息（第 9 行），或者消息传递的时序表明不存在依赖关系，则不考虑 P_j 中相关的方法。否则，将它们添加到 $DS(q)$（第 11 行）。其依据是：对于分别在两个进程 $p1$ 和 $p2$ 中执行的两个方法 $m1$ 和 $m2$，只有当 $p2$ 在 $lr(m2)$ 之前收到至少一条消息，且该消息是在 $fe(m1)$ 之后由 $p1$ （直接或间接）发送的，$m2$ 才依赖于 $m1$。

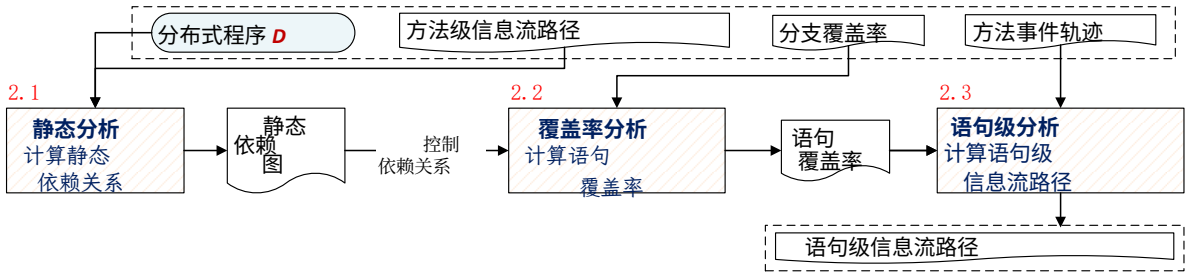
计算 $DS(q)$ 时，如果其中包含一个包含汇点的方法 m_k （第 13 行），那么 $DS(q)$ 中这些具有偏序关系的方法便构成了从 q 到 m_k 的信息流路径。所有此类路径都会被收集到 ps （第 15 行），随后返回（第 18 行）。

4.2.3 Phase 2: Refinement

在这一阶段，FlowDist旨在通过细化第 *1* 阶段推断出的粗粒度结果（即方法级信息流路径），利用两种模态的程序数据（即静态 和动态），分三步计算细粒度（语句级）信息流路径，如图 4.4 所示。这种混合分析设计的主要动机是在总分析成本与精确率之间取得平衡 [40]。

步骤 2.1. 此步骤主要进行静态分析，构建程序 D 的静态依赖图。与全系统依赖图不同，该图并不完整，仅涉及第一阶段（预分析）计算出的（方法级）流路径上的方法。其依据是，从源到汇的语句级信息流路径上的方法，必定也位于包含源和汇的两个方法之间的语句级信息流路径上。 FlowDist 在遇到不属于（方法级）信息流路径的方法时，会停止相关流事实的过程间传播。

图 4.4： FlowDist 的第 2 阶段工作流程



静态依赖关系在语句级进行计算，作为最后一步混合数据流分析所需的一种关键信息。具体而言， FlowDist会计算线程内和线程间的数据依赖关系和控制依赖关系 [123] 。另一方面，为了在计算此类静态依赖关系时涵盖 D 中的所有组件，静态分析会搜索所有可能的入口点（即所有包含 `main`方法的类），并从找到的每个入口点开始进行数据流计算。同时，也会计算所有控制依赖关系。

此处选择采用上下文不敏感的静态依赖分析，因为其结果仅用于步骤 2.3，而该步骤将利用方法执行事件提供必要的上下文。此外，过程间分析部分采用流不敏感分析，因为这些事件按其时间戳排序，而过程内分析部分仍采用流敏感分析。这些选择降低了 FlowDist 的总分析成本。

步骤 2.2。 此步骤旨在为每个进程生成语句覆盖率，并收集细粒度动态数据，仅用于在步骤 2.3 中改进混合分析；此过程参考步骤 2.1 中的静态依赖关系和步骤 1.2 中的分支覆盖率。重要的是，只考虑第 1 阶段计算出的方法级路径上的方法。原因是其他方法中的语句不会出现在语句级信息流路径中。

步骤 2.3。 借助进程 P_i 中的语句覆盖率（SC）、方法事件轨迹（T_i）以及 D、 FlowDist 的部分静态依赖图（sDG）， D、 FlowDist 现在可以推断每个源-汇调用点对（$<s$、$t>$）之间的语句级信息流路径。 FlowDist 利用消息发送与接收 API，识别方法级流路径上各方法中的调用点，从而确定信息流出和流入各进程的位置，分别称为出口 和入口。

在算法 2 中，首先按事件时间戳排序，将每进程方法事件序列合并为全局事件序列 ES （第 2 行）。然后，函数 `build-dDG`在遍历 ES时，参考 sDG中的静态依赖关系，并采用受 Diver [39]启发的混合依赖分析，构建动态依赖图 dDG （第 3 行）。其中的要点包括：(1) sDG 中的过程间依赖分为相邻 （由参数传递或返回值传递导致）和后续（由定义-使用关联导致）；(2) 扫描 ES时，如果该依赖是相邻依赖，且 $m2$ 紧接在 $m1$ 之后发生于 ES中；或者该依赖是后续依赖，且 $m2$ 在 ES中的 $m1$ 之后任意位置发生，那么方法 $m2$对另一方法 $m1$ 的静态依赖就会被激活，并因此添加到 dDG中。

该算法将ES 中某个方法的所有静态进程内 依赖关系视为已激活，然后将它们添加到 dDG。接着，算法从源 s 构建图，仅包含能够到达汇 t 的依赖关系。由此得到的 dDG 中的依赖关系在语句级上并不精确。因此， FlowDist 使用函数`prune-dDG` ，按照语句覆盖率 SC剪除 dDG 中的伪依赖：从图中删除与未覆盖语句对应的顶点及其相关边，从而得到剪枝后的图（第 4 行）。我做出这一选择，是为了控制 FlowDist 的总体分析成本，提升其可扩展性。

算法 2 计算语句级流路径

令 S_1, S_2, ..., S_n 为 n 每进程实例级事件轨迹
令 M_1, M_2, ..., M_k 为方法级路径上的 k 方法
令 sDG为部分静态依赖图
令 $<s$、$t>$ 为一个源-汇调用点对，在二者之间计算路径
令 $outlets$ 为所有出口的列表
令 $inlets$ 为所有入口的列表

1: $SOFPS$=∅, $REFPS$=∅, $SIFPS$=∅, $intraFP$=∅
2: 将 T_i, i∈$[1, n]$ 合并为一个全局偏序序列 ES
3: dDG = `build-dDG`(sDG, $<s,t>$, ES) // 混合分析
4: dDG' = `prune-dDG`(dDG, SC) // 语句级剪枝
5: $SOFPS$ = `findPaths`(dDG', {s}, $outlets$, $tr(s)$) // 计算进程间路径的远程片段
6: **对** i=至 n **做**
7: $intraFP$ ∪= `findPaths`(dDG',{s},{t},S_j) // 进程内路径
8: **如果** $tr(s) == T_i \ \vee \ tr(t) == T_i$ **，则继续**
9: **结束条件判断**
10: $REFPS$ ∪= `discoverPaths`(dDG', $inlets$, $outlets$, T_i)
11: **结束循环**
12: $SIFPS$ = `discoverPaths`(dDG', $inlets$, {t}, $tr(t)$)
13: **返回** [`spliceSegs`($SOFPS$, $REFPS$, $SIFPS$), $intraFP$]

借助 dDG'，算法随后使用函数 `discoverPaths(G, In, Out, T)`推导进程内和进程间的信息流路径。该函数在图 `G` 上检测从 `In` 中的任意语句到 `Out` 中任意语句的路径，同时仅考虑 `T`中的节点。算法通过简单遍历 dDG'，计算每个进程中的进程内信息路径（$intraFP$）（第 7 行）。

然而，对于任何进程间信息流路径，由于 dDG' 中不存在连通性，因此无法从源显式到达汇。因此， FlowDist 会分别计算这三个片段：

1. 源 (s) 所在进程中的片段 ($SOFPS$)，通过遍历 dDG' （第 5 行）计算，该遍历会从 s 获取路径，直至该进程跟踪中的相关出口——$tr(x)$ 表示包含封装语句 x 的方法事件的跟踪。

2. 汇（t）的进程（$SIFPS$）中的片段采用类似的方法计算（第 12 行），但通过搜索从任意入口到 t 的路径来完成。

3. 剩余片段（*REFPS*）会在除包含 s 或 t 的外层进程之外的每个进程中进行搜索（第 6–10 行），方法是遍历 dDG'，并查找进程中从所有入口到所有出口的路径。

最后，根据相关入口和出口的时间戳，利用函数 `spliceSegs` 将这些片段拼接成进程间信息流路径。按照全局偏序序列 ES，这种拼接可确保 *SOFPS* 的结束与*REFPS*的开始之间，以及 *REFPS* 的结束与 *SIFPS* 的开始之间均没有事件。随后，将这些拼接后的进程间路径与进程内路径一并作为算法输出返回（第 13 行）。

4.2.4 Alternative designs

如上所述， FlowDist 的默认设计面向一般的通用分布式软件系统。为了更系统地探索用于动态信息流分析的多阶段细化方法，我设计了两种替代设计：FlowDistsim 和 FlowDistmul。对于满足特定条件的系统，这些设计或许能进一步降低分析成本，同时不损害精确率和可靠性，从而提供更高的成本效益和可扩展性。

FlowDistsim。 在 FlowDist 的第 1.1 步中，静态分析的目标是缩小插桩范围，从而降低跟踪方法和语句分支事件的成本。然而，对于某些系统，探测并跟踪所有此类事件的成本很低，而静态分析本身产生的成本可能高于其节省的成本。针对符合这些条件的系统进行了优化后， FlowDistsim 会跳过静态分析，直接对所有方法和语句分支进行插桩。

FlowDistmul。 对于某些系统， FlowDistsim 的设计很有道理。但在 D 中探测并跟踪所有方法和分支事件会产生高昂的成本。为了降低这些成本，我在 FlowDistsim 中引入了一个中间阶段，并在第 1 阶段采用多阶段、基于细化的设计。

首先，新的第 1 阶段仅针对每个方法的首次进入事件和最后一次返回进入事件进行探测与跟踪，然后根据这些事件计算方法级流路径。随后，中间阶段会对语句分支的覆盖率以及此类路径上方法的两类事件的所有实例进行探测与跟踪。最后，从阶段 2 中移除步骤 2.2。

由于 FlowDistmul 需要针对相同输入多次执行同一系统（在第一阶段和中间阶段），因此该设计针对执行具有确定性的系统进行了优化。第一个条件是，两次执行之间的不一致可能会损害 DIFA 整体的可靠性。另一个条件是，成本降低应当超过中间阶段产生的成本。

FlowDistsim **与 FlowDist 相比**mul。 根据各个备选设计的原理，对于没有非确定性执行的此类系统， FlowDistmul 是最佳选择；对于具有非确定性执行的小型/简单系统， FlowDistsim 预计表现最佳。对于大型/复杂分布式系统， FlowDist（默认设计）的表现会最好。这些差异由条件 （如上所述）决定。如果系统不满足其中任何条件，通常而言， FlowDist的默认设计更胜一筹。

4.2.5 Limitations

FlowDist 无法解决识别目标源/汇的问题；这些源/汇默认已包含在列表中，或由用户提供。此外， FlowDist中的分析仅限于已执行的程序部分。因此，其漏洞发现能力取决于以下两点：(1) 已指定相关源和汇；(2) 所考虑的运行时输入覆盖了该源和汇。此外，考虑到特定使用场景中的安全上下文（例如，对源或汇应用的外部保护机制），
FlowDist可能会产生误报，因为它们既不会分析这些外部/上下文因素，也无法获取这些因素。

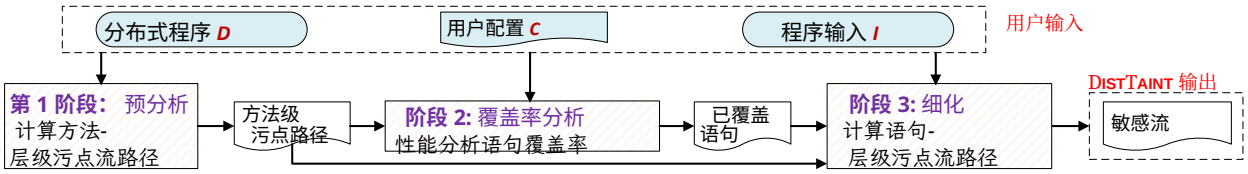
此外， FlowDist 需要静态插桩，因此不适用于无法修改的系统。 FlowDistsim 和 FlowDistmul 的其他局限性，则是前文所述相应系统条件所带来的。

4.3 工具实现

对于分布式系统，信息流安全解决方案面临多项挑战，包括技术适用性、工具可移植性以及分析可扩展性。为应对这些挑战，我实现了 DistTaint [86, 88, 84, 85]，这是一款面向分布式系统的动态信息流（污点）分析器，如图 4.5 所示。

通过对执行期间的方法事件进行偏序处理， DistTaint 能够计算分布式程序中的隐式依赖，从而解决适用性挑战。它完全在应用层运行，无需定制运行时平台，因此克服了可移植性挑战。为实现可扩展性，它采用多阶段分析来降低分析成本，

图 4.5： DistTaint架构概述



其中，预分析阶段生成的方法级结果用于缩小后续语句级分析的范围。将其应用于不同的大型分布式软件，并针对多种执行进行分析后， DistTaint展示了其对工业规模分布式系统的适用性、可移植性和可扩展性，以及发现现有漏洞和新漏洞的能力。

4.4 评估

4.4.1 Experiment setup

如表 4.1 所示，我使用了 12 个 Java 分布式系统作为被测程序。被测程序的规模通过非空、非注释 Java 源代码行数（#源代码行数）、被测程序中定义的方法数（#方法）以及执行场景/架构（场景）来衡量，包括客户端-服务器、点对点和多层架构。

下面我将介绍各个被测程序及其集成测试操作，但此前已介绍的除外（§3.4.1）。

1. NioEcho [215] 为客户端发送的任意消息提供回显服务。在其集成测试中，我启动了一个服务器和一个客户端，从客户端向服务器发送随机文本消息，然后等待每条消息的回显。

2. MultiChat [108] 是一个广播客户端消息的聊天服务。在其集成测试中 ，

表 4.1：所使用的目标分布式程序和测试输入

对象	#源代码行数	#方法	场景	测试
NIOEcho	412	27	客户端-服务器	集成
MultiChat	470	37	点对点	集成
ADEN	4,385	260	点对点	集成
Raining Sockets	6,711	319	客户端-服务器	集成
OpenChord	9,244	736	点对点	集成
Thrift	14,510	1,941	客户端-服务器	集成
xSocket	15,760	2,209	点对点	集成
ZooKeeper	62,194	5,383	客户端-服务器	集成
			N层系统	Load
			N层系统	系统
RocketMQ	105,444	6,198	多层	集成
			多层	系统
Voldemort	115,310	20,406	客户端-服务器	集成
			多层	Load
			多层	系统
Netty	167,961	12,389	多层	集成
HSQLDB	326,678	10,095	客户端-服务器	集成
			多层	系统

我启动了一个服务器和三个客户端。从其中一个客户端向服务器发送随机文本消息，服务器再将这些消息广播给所有其他客户端。

3. ADEN [214] 提供了一种基于 UDP 的 TCP套接字替代方案。在其集成测试中，我启动了两个节点，每个节点都会向另一个节点发送消息并接收来自另一个节点的消息。

4. Raining Sockets [213] 是一个非阻塞的基于套接字的框架。在其集成测试中，我启动了一个服务器和一个客户端，然后客户端向服务器发送了文本消息。

5. RocketMQ [193] 是一个分布式消息平台。在其集成测试中，有四个组件：名称服务器、代理、生产者和消费者。名称服务器提供读写服务并记录完整的路由信息，

代理存储消息，生产者向代理发送消息，而客户从代理接收消息。

6. Netty：在其集成测试中，我开发了一个由三个节点组成的多层（三层）应用程序。第一个节点从文件中读取电子邮件列表，然后将相关电子邮件发送给第二个节点。接着，第二个节点使用 RSA 算法加密这些电子邮件，并将其发送给第三个节点。最后，第三个节点使用 Postfix 发送收到的电子邮件。

7. HSQLDB（HyperSQL 数据库） [216] 是一种 SQL 关系数据库系统。我启动了一个数据库服务器和一个客户端。随后，客户端向服务器发送 SQL 查询，并接收服务器返回的 SQL 结果。

对于所有被测程序，我都根据其官方网站上的指南创建了集成测试。特别是针对五个框架/库（ADEN、Raining Sockets、Thrift、xSocket 和 Netty）的集成测试，我开发了应用程序以覆盖其主要功能，然后运行了所有这些应用程序。

我选择了最先进的动态污点分析器 Phosphor [30] 和静态污点分析器 JOANA [114], 作为与 FlowDist进行比较的两个基线。实验使用了搭载 Ubuntu 16.04.3 LTS 的工作站，每台工作站均配备 Intel E7-4860 2.27GHz CPU 和 32GB DMI RAM。

4.4.2 Results and analysis

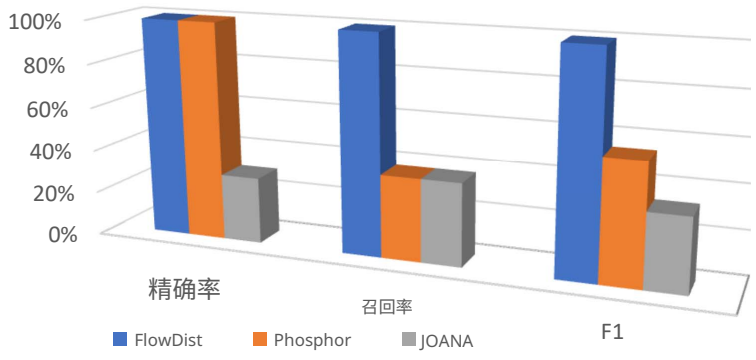
有效性。 表 4.2 分别展示了进程内和进程间路径中，每次执行（即被测对象-测试类型）所覆盖的源-汇对数量，以及这些源-汇对之间的信息流路径数量

表 4.2：进程内（*Ir*）源/汇对（*Pr*）和信息流路径（*Ps*）的数量，与进程间（*Int*）数量的比较

执行	#IrPr	#IrPs	#IntPr	#IntPs	IntPs/AllPs
NioEcho	66	21	12	6	22.22%
MultiChat	42	0	12	0	0.00%
ADEN	0	0	5	0	0.00%
Raining Sockets	12	3	0	0	0.00%
OpenChord	14	0	24	0	0.00%
Thrift	4	0	4	3	100.00%
xSocket	10	8	26	2	20.00%
ZooKeeper 集成	9	0	33	0	0.00%
Zookeeper 负载	1086	1	6522	64	98.46%
Zookeeper 系统	124	0	1116	46	100.00%
RocketMQ 集成	19	23	46	17	42.50%
RocketMQ 系统	24	0	187	50	100.00%
Voldemort 集成	198	30	193	138	82.14%
Voldemort 负载	6	0	6	0	0.00%
Voldemort 系统	80	30	77	42	58.33%
Netty	9	3	7	2	40.00%
HSQLDB 集成	140	10	668	0	0.00%
HSQLDB 系统	7	2	11	4	66.67%

。对于配置C、 FlowDist 中给出的每个源/汇，污点分析将其每个实际执行的调用点作为单独的源/汇来统计源-汇对并计算路径。最后一列显示了每次执行中，进程间信息流路径占所有路径的百分比。灰色行表示不包含任何信息流路径的执行。

图4.6： FlowDist 与基线 Phosphor 和 JOANA 的准确率



两个基线都捕获了所有进程内路径，就像 FlowDist 一样，但遗漏了所有进程间路径。因此，如图4.6所示，它们的召回率相同但较低（37.5%）。出于同样的原因，它们都未能发现任何已知或未知漏洞。此外，JOANA 报告了许多未被所考虑的执行覆盖的额外路径，并将其视为误报，导致其精确率最低（30%）。因此， FlowDist 取得了最高的 F1值（100%）（Phosphor 的 F1值为 54.6%，JOANA 的 F1值为 33.3%）。

表4.3： FlowDist的时间（秒）和存储（MB）成本

执行	Norm	阶段1时间				阶段2时间			存储
	运行	静态运行	减速	查询	静态覆盖率	查询			
NioEcho	39	53	41	5.16%	0.2	50	1	1.0	1.6
MultiChat	26	55	28	6.12%	0.2	50	1	0.1	1.0
ADEN	21	117	23	10.23%	0.3	59	3	0.3	4.0
Raining Sockets.	6	40	6	7.67%	0.3	122	6	0.4	14.5
OpenChord	54	177	59	8.54%	0.3	740	41	4.7	26.7
Thrift	8	146	10	24.83%	0.5	79	45	0.6	26.1
xSocket	11	101	19	63.99%	0.5	70	14	0.1	29.3
ZooKeeper 集成	71	292	121	70.16%	0.5	193	108	1.8	231.2
Zookeeper 负载	99	292	177	78.83%	0.6	137	67	2.0	404.0
Zookeeper 系统	98	292	178	81.87%	0.5	250	93	1.1	417.5
RocketMQ 集成	105	56	196	87.05%	0.6	704	49	21.5	291.0
RocketMQ 系统	339	156	753	122.09%	0.6	727	52	34.0	463.2
Voldemort 集成	28	1206	58	106.06%	0.6	566	317	9.1	560.4
Voldemort 负载	11	1206	23	113.37%	0.6	435	260	14.4	523.1
Voldemort 系统	31	1206	65	109.81%	0.6	618	344	22.2	545.1
Netty	12	1132	22	81.65%	0.6	381	317	30.1	417.6
HSQLDB 集成	9	659	19	107.46%	0.7	2227	96	41.5	591.1
HSQLDB 系统	15	684	36	142.71%	0.7	2771	408	49.7	733.7
总体平均值	**55**	**437**	**102**	**68.20%**	**0.5**	**565**	**124**	**13.0**	**293.4**

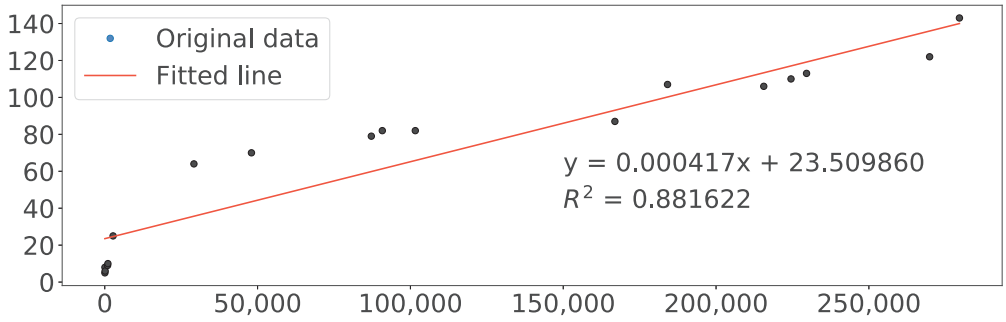
效率。 表4.3列出了 FlowDist在两个阶段及其主要步骤中的时间和存储成本明细。时间开销包括静态分析（以及可能进行的插桩）所需的时间（**静态**）、性能分析所需的时间（**运行**），以及平均而言计算每个源-汇对之间（方法级或语句级）信息流路径所需的时间

（**查询**）。第二列列出了每次执行的原始运行时间（**正常运行时间**），据此计算出性能分析开销的运行时间减速比（**减速**）。第八列显示了语句覆盖率分析的时间开销（**覆盖率**）。最后一列列出了每次执行各阶段的总存储成本（**存储**），用于存储第 1 阶段的方法事件和分支事件跟踪、第 2 阶段的语句覆盖率和部分静态依赖图，以及插桩程序。最后一行给出了总体平均值（所有执行的平均值）。

在 18 次执行中，平均而言， FlowDist 所有一次性分析耗时 19 分钟（（$437 + 565 + 124$）/ 60），包括所有静态分析、插桩和覆盖率分析所需的时间，如表4.3最后一行所示。我将它们视为一次性分析，因为对于给定的对象执行和源/汇配置，其结果可供所有查询共享。具体而言，部分依赖分析（由第 1 阶段的方法级路径引导）比未经过预分析阶段的全系统分析高效得多。例如，在针对 Voldemort 的其他设置都相同的情况下，后者甚至在 12 小时内都未能完成。

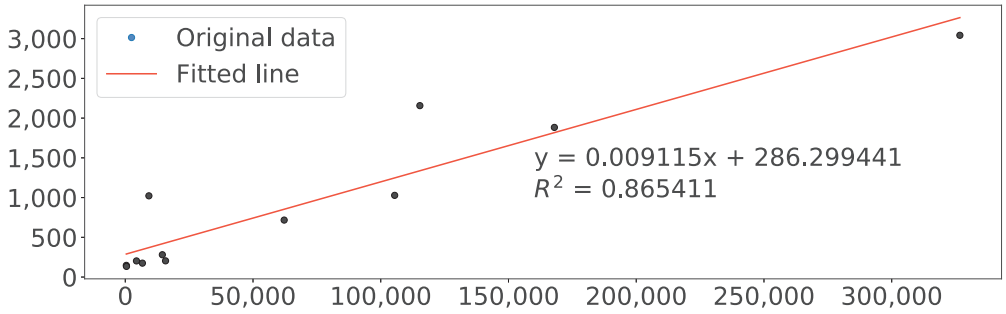
另一方面，Phosphor 和 JOANA 对每个源/汇对的平均耗时分别为 1.38 秒和 0.43 秒，低于 FlowDist 的查询成本（平均 13 秒）。 FlowDist 的平均存储成本（293.4 MB）也高于 Phosphor（21.2MB）和 JOANA（35.2MB）。这是因为，与基线相比， FlowDist 执行了更多、更复杂的分析（例如探测、构建动态依赖图、剖析实例级方法事件）。不过， FlowDist 产生的这些额外成本并不高，而且其有效性大幅提升，足以抵消这些成本。此外，它无需付出大量人工工作（例如开发测试用例或添加源代码注释）。

图 4.7： 所有被测程序执行的运行时减速（%，y 轴）与方法执行事件实例数（x 轴）的关系。



可扩展性。 图 4.7 展示了 FlowDist 的可扩展性，衡量指标是运行时减速；图中涵盖了全部 18 次执行，每次执行均以其中实例级方法执行事件序列的长度作为运行时复杂度度量。拟合曲线的 $R^2 > 0.88$ 表明，就运行时开销而言，FlowDist 能够良好地扩展至大型系统；决定系数 $R^2 \in [0,1]$ 表示数据与曲线的拟合程度（R^2 越接近 1，拟合效果越好）。

图 4.8： 所有被测程序（integrationtest）的总分析时间（秒，y 轴）与被测程序规模（#源代码行数，x 轴）的关系



与此同时，图 4.8 展示了 FlowDist 随被测程序规模增大时的总时间开销（一次性分析时间、性能分析开销以及查询所有可能源/汇对所需时间之和），并与集成测试进行对比，因为每个被测程序都有

表 4.4： FlowDist 检测到的现有漏洞

对象	漏洞参考	检出#漏洞	#假阴性		
HSQLDB	CVE-2005-3280	[58]	✓	1	0
Netty	CVE-2014-0193	[55]	✗	10	5
	CVE-2014-3488	[64]	✗		
	CVE-2015-2156	[56]	✗		
	CVE-2016-4970	[59]	✗		
	问题 8869	[100]	✗		
	问题 9112	[101]	✓		
	问题 9229	[102]	✓		
	问题 9243	[103]	✓		
	问题 9291	[104]	✓		
	问题 9362	[105]	✓		
RocketMQ	CVE-2019-17572	[60]	✓	1	0
Thrift	CVE-2015-3254	[57]	✓	1	0
Voldemort	问题 101	[99]	✓	6	1
	问题 381	[12]	✓		
	问题 387	[13]	✓		
	问题 352	[9]	✓		
	问题 378	[11]	✓		
	问题 377	[10]	✗		
xSocket	漏洞 21	[106]	✓	1	0
ZooKeeper	CVE-2014-0085	[54]	✓	4	0
	漏洞 2569	[5]	✓		
	CVE-2018-8012	[65]	✓		
	CVE-2019-0201	[66]	✓		

这样的测试。与图 4.7 的格式相同，图 4.8 中的拟合曲线表明， FlowDist 的总时间开销呈线性增长。

发现漏洞。我从各种来源（例如漏洞库和 CVE 报告）中搜寻了我们被测程序中的现实漏洞，筛选出与信息流安全相关的漏洞，随后在 7 个受研究的被测程序中识别出一个或多个漏洞，如表 4.4 所示。表中列出了每个被测程序（**被测程序**）的漏洞（**漏洞**）、参考链接（**参考**）和漏洞数量（**#漏洞**），并用标记（**已发现**）表示是否发现了该漏洞。最后一列给出了遗漏漏洞的数量（**#假阴性**）。

根据推断出的信息流路径（如表 4.2 所示）， FlowDist成功发现了除 Netty 之外的所有这 7 个被测程序中的大多数漏洞。Netty 中有 5 个漏洞、Voldemort 中有 1 个漏洞被遗漏。我核实了这些漏洞被遗漏的原因：执行过程中未覆盖这些漏洞。我没有刻意选择运行时

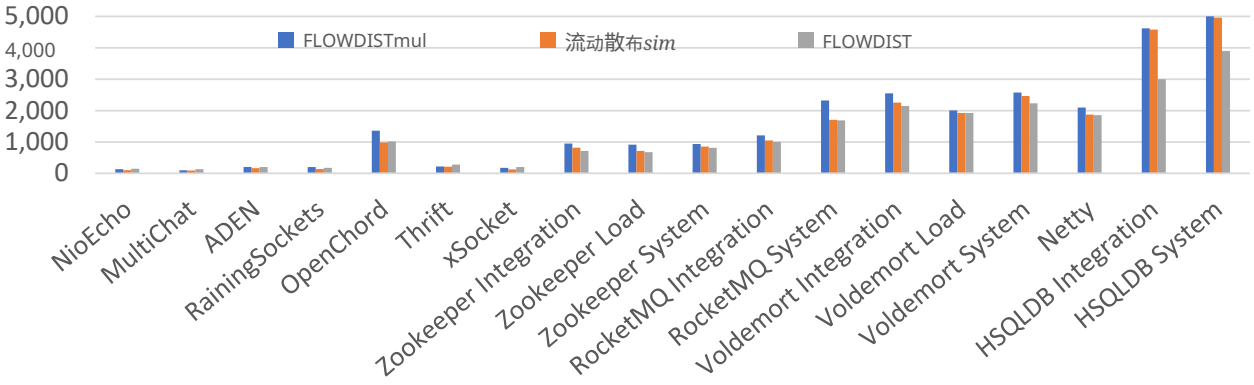
输入来覆盖这些漏洞，而只是使用了现有的、能够代表系统运行场景的输入。此外，在18 个成功案例中，相应路径均为进程间路径。

表4.5： 由 FlowDist 发现的新漏洞

对象	#已修复	#已确认	#待处理
HSQLDB	0	5	2
Netty	1	1	0
Raining Sockets	0	1	0
RocketMQ	0	4	0
Thrift	0	5	0
Voldemort	0	0	4
xSocket	0	0	1
ZooKeeper	1	1	0

此外，根据 FlowDist针对8个研究对象发现的信息流路径，识别出24个新漏洞 [87]，如表4.5所示。我报告后，其中17个已得到确认，2个目前已修复。这表明， FlowDist 能够根据给定执行中的信息流路径检测已知或新发现的漏洞，而无需错误报告；不过，此类报告可能有助于漏洞检测。

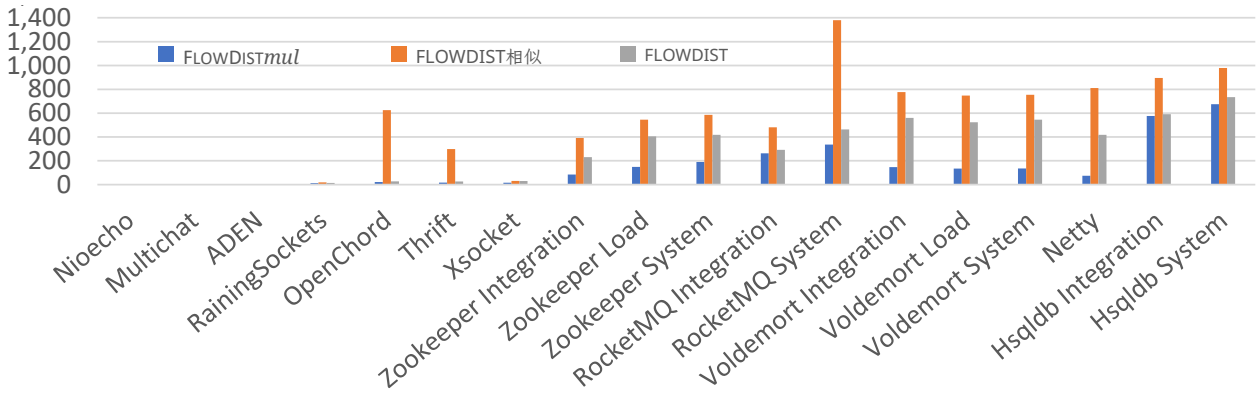
图4.9： 所有被测程序执行中， FlowDist*mul* 和 FlowDist*sim* 相对于FlowDist 的总时间开销（秒）



不同设计的对比。 不出所料，在效率方面，三个设计中表现最佳的设计因系统而异。图4.9展示了针对所研究的18次执行， FlowDist 及其各设计的总分析时间对比。对于相对大型的系统（ZooKeeper或更大型的系统）， FlowDist 效率最高。对于该系统，预分析中因减少插桩和缩小剖析范围而节省的时间，明显超过静态分析的时间开销，因此 FlowDist 优于FlowDistsim。与此同时，因缩小实例层级方法事件的性能分析范围而节省的时间，超过了中间阶段带来的额外时间开销，因此FlowDist 优于 FlowDistmul。

对于小型系统（小于 ZooKeeper 的系统），这些相反的对比结果则发生了逆转，这解释了为什么对于这些系统而言，替代设计优于 FlowDist。

图4.10： FlowDistmul 和 FlowDistsim 针对所有被测程序执行的存储成本（MB）



存储成本的比较显示差异并不显著，如图4.10所示。FlowDistmul 和 FlowDistsim 的存储空间需求分别最少和最多，而 FlowDist 的需求介于两者之间。原因是，FlowDistmul 在预分析阶段只记录首次进入事件和最后一次返回进入事件

，随后只跟踪预分析中发现的方法级流路径上的方法及这些方法中的分支。相比之下，FlowDistsim 在预分析阶段会跟踪执行过程中的所有实例级方法和语句分支事件。另一方面，FlowDist 只跟踪相关方法和分支。

表 4.6： DIFA/DTA工具选择建议

<table>
<tr><th colspan="3" rowspan="2">系统类型</th><th colspan="2">执行是否具有非确定性?</th></tr>
<tr><th>Yes</th><th>No</th></tr>
<tr><td rowspan="3">分布式
（多进程）</td><td rowspan="2">通用</td><td>小型</td><td>FlowDistsim</td><td>FlowDistsim
或 FlowDistmul</td></tr>
<tr><td>大型</td><td>FlowDist</td><td>FlowDist</td></tr>
<tr><td colspan="2">专用</td><td colspan="2">Kakute [128] （适用于 Spark [245]）
Pileus [221] （适用于 OpenStack [200]），...</td></tr>
<tr><td colspan="3">单进程</td><td colspan="2">Phosphor [30], JOANA [114],...</td></tr>
</table>

表 4.6 中展示的研究结果促使我提出了如何为特定软件系统选择合适工具的建议。总体而言，FlowDist（默认设计）最适合大型通用分布式系统，无论其执行是否具有非确定性。对于小型通用分布式系统，如果已知其执行具有确定性，那么FlowDistsim 和 FlowDistmul 都可能是不错的选择；否则，只应考虑 FlowDistsim。我还在表 4.6 中列出了一些适用于其他类型（专用分布式系统或单进程系统）的同类工具。

4.5 相关工作

与我们的工作最相关的已有研究分为两类：传统信息流分析（*IFA*）& 污点分析（*TA*）以及跨进程信息流分析（*IFA*）&污点分析（*TA*）。

4.5.1 Conventional information flow analysis (IFA) & taint analysis (TA)

此前大多数此类分析都是静态的 [171, 196, 212, 228, 15, 149, 231, 109, 249, 114]。这些方法不仅存在静态分析常见的不精确性问题，而且由于现代语言中的动态构造（例如，动态代码加载），还缺乏可靠性 [153]。应用于分布式软件时，由于分布式（解耦）组件之间存在隐式依赖关系，它们的准确性会更低。

在动态方法中，TaintDroid [77] 对 Android 操作系统进行了定制，以便在运行时跟踪整个系统的信息流，并在发现敏感信息流时向用户发出警告。Panorama [243] 基于处理器模拟器 QEMU，通过动态插桩对 Windows 恶意软件分析执行系统侧的动态信息流跟踪。在 [121], 中，研究人员使用动态污点分析，通过定制的 Linux 安全模块（即修改 Linux 内核）进行入侵检测。另有少数方法 [47, 53, 218] 需要专用硬件来执行污点跟踪。这些方法依赖平台定制。此外，它们属于动态污点分析，而非动态信息流分析，因为它们无法提供完整的代码级信息流路径。Juturna [154], 是一种与 JOANA 结合的混合方法，它采用字节码增强和经过修改的 Java API 类来进行污点跟踪。因此，它也需要平台定制。

Dytan [49] 提供了一个用于对 x86 可执行文件进行动态污点分析的通用框架，以及该框架针对 x86 二进制文件的一个实例。与 Dytan 类似，TaintEraser [256] 利用动态插桩框架 Pin [155] 跟踪 Windows 应用程序中的运行时信息流。在 [20, 21, 134], 中，作者提出了用于 JavaScript 代码动态污点分析的语言语义。具体而言， [134] 使用 J alangi 框架 [201] 来

对目标系统的 ECMAScript 5 [68] 代码进行插桩。LabelFlow [46]是 PHP 的一个扩展，用于简化 Web 应用程序中的安全策略实现。TaintMan [77] 是一种与 Android 运行时（ART）兼容的动态污点分析，它对目标 Android 应用程序和库的代码进行静态插桩，以跟踪数据流和控制流。TaintTrace [44] 使用 DynamoRIO [36] 执行插桩，其加载器通过修改 Valgrind [175]的代码实现。此外，TaintCheck [176]使用 Valgrind 执行插桩，而 LIFT [186] 基于 StarDBT 动态二进制翻译器 [225] ，DFSan [223] 则采用 LLVM 框架 [145]。

与其他分析器一样， [176, 166, 207, 16, 27], 这些方法无法妥善处理分布式软件，因为它们只跟踪单个线程/进程中的信息流。

4.5.2 Cross-process information flow analysis (IFA) & taint analysis (TA)

在众多用于跟踪/检查动态信息流的既有技术中，只有少数涉及跨进程的信息流。其中一种是 Kakute， [128], 它通过统一 API 实现引用传播和标签共享，并跟踪字段级数据流。由于 Kakute 基于 Phosphor， [31], 它还需要定制（插桩）其运行时平台（即 JVM）。此外，它面向运行在 Spark 上的大数据应用程序， [245], 并不适用于通用分布式软件。这与 Pileus 类似， [221] 后者面向运行于特殊云平台 OpenStack 上的应用程序。 [200]

Taint-Exchange [247] 是一种通用的跨进程/主机污点跟踪框架。它基于 libdft [136] 构建，用于传输主机/进程之间通过套接字和管道交换的污点信息。与 Cloudfence [178] 和 Cloudopsy [246], 类似，它依赖于定制的运行时（Pin 动态插桩框架），面向 C/C++ 软件。

第五章

SEADS：基于强化学习的可扩展且具成本效益的动态依赖分析

理解和验证软件行为的一项基本策略，是将程序实体之间的运行时交互建模为依赖关系，然后基于依赖模型 [140, 127, 34]推理程序行为。从历史上看，动态依赖建模与分析 [184, 122] 一直支持软件质量保证，应用范围从故障诊断 [237] 到安全防御 [18, 136]。动态依赖分析十分重要，因为软件质量保证中的许多应用技术都依赖动态依赖关系，例如性能监控、程序优化、安全防御、软件测试、漏洞检测等 [141]。例如，软件测试对软件质量保证也至关重要；通过在观察到错误输出的程序实体的依赖关系中进行搜索，可以利用动态依赖关系检测软件缺陷。同样，也可以沿着依赖关系链，利用依赖关系检测从源到汇的运行时敏感数据泄露。与静态方法相比，动态依赖分析的精确率更高，因为它关注特定的具体执行。因此，我开发了一种经济高效的动态依赖分析框架，并实现了一个满足用户预算约束的动态切片器。

5.1 动机

总体而言，分布式软件动态分析面临的一项根本挑战，是如何在分析开销与分析算法的有效性之间实现最佳平衡。

（例如，以最低成本实现最高分析准确率）。早期针对并行和分布式软件的依赖建模与分析曾有过尝试 [43, 139, 74, 131, 168]，但成本不平衡，且缺乏可扩展性和有效性。

例如，我尝试将一种现有的最先进的分布式程序依赖分析方法 [38] 应用于一个真实系统 [6]。该分析在配备 8 核处理器、2.7 GHz CPU 和 512 GB 动态随机存取存储器的机器上运行，耗时 12 小时仍未完成。而且，我没有找到任何现有的依赖分析方法，能够为现实中的工业规模分布式系统提供可扩展的解决方案。

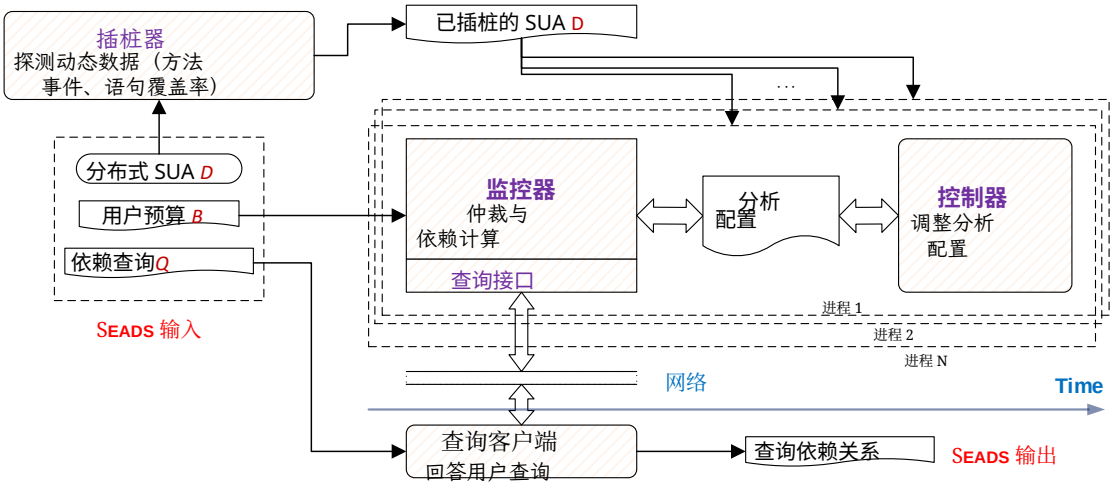
然而，越来越多的工业规模软件系统在本质上正变为分布式系统。因此，分布式系统亟需工具支持，而动态依赖分析是其中一项基础性、赋能性方法。我认为，对于分布式系统而言，在线分析比离线分析更合适。在线方案在运行期间分析系统，而离线方法则在程序执行后计算依赖关系。因此，在线分析会快得多，尽管会产生少量运行时开销。在持续不断的无限执行过程中，在线方法不需要执行轨迹。

相比之下，离线技术需要跟踪记录，其存储成本可能很高。如果执行永无止境，对轨迹存储空间的需求也将是无限的。然而，这并不现实，因此离线分析不适用于分布式系统。

5.2 方法

我开发了 SEADS（即可扩展且经济高效的分布式系统动态依赖分析）[91]，一种具成本效益的动态依赖分析框架，它能够

图 5.1： SEADS 架构和工作流程概述，包括其输入、输出和关键模块



将分析扩展到真实世界分布式系统。分析本身是分布式的，并且在线运行，以解决在分析持续运行的系统时执行轨迹无界的问题。此外，给定用户指定的时间预算，分析会根据时间开销，通过调整配置自动进行调整，以实现更好的成本效益权衡。其核心思想是采用强化学习方法，根据当前配置及其对应的分析成本，并结合给定的用户预算，决定调整哪些配置。

5.2.1 Overview

SEADS 的整体架构如图 5.1 所示，由一个插桩器、每个被分析系统（SUA）进程对应的监视器、每个 SUA 进程对应的控制器，以及一个查询客户端组成。 SEADS 接受用户提供的三项**输入**：分布式 SUA D、用户预算 B和依赖查询 Q。具体而言， B 是依赖分析的响应时间约束，而 Q 是用户输入的方法名称，用于

该方法的依赖关系，例如 *org.apache.zookeeper.server.ZooKeeperServerMain: void main(java.lang.String[])*。

插桩器首先将探针插入 D ，用于监控进入和返回到方法执行事件，和/或监控已执行语句的覆盖率，从而生成插桩系统 D'。随后，随着时间推移， D' 持续运行， SEADS 则通过进程级监视器和控制器，不断调整自身的分析配置。具体而言，在执行 D'期间，我假设存在 N 个进程（例如，一个服务器和 $N-1$ 个客户端）。在其每个 N 进程中，监视器负责仲裁（决定调整方式）和依赖计算，而控制器负责调整分析配置。简言之，总体动态依赖分析以分布式方式进行，各进程的分析配置独立于其他进程进行调整。

查询客户端接收用户发出的查询 Q ，然后通过网络设施将其发送给查询接口；该接口直接与每个进程中的监视器通信。在某个进程中的依赖计算完成后， Q 的结果依赖关系会传送到该进程中与监视器相连的查询接口，查询客户端再从该接口接收该进程的查询依赖关系。所有进程中的分布式依赖分析完成后，查询客户端接收并合并 Q 的所有进程级依赖集，然后执行进程间分析，计算总体查询依赖，作为 SEADS的最终**输出**。

5.2.2 Configuration

实现可扩展性和成本效益的关键思路是根据以下因素持续调整分析配置：(1) 用户预算，(2) 当前和先前的配置，

以及 (3) 依赖计算的时间开销。在 SEADS 中，分析配置发挥着关键作用，其中每个参数都对成本效益有独特的贡献。作为一种混合方法， SEADS 结合运用多种静态和动态分析技术，以及不同的静态/动态数据（例如静态依赖图、方法事件和语句覆盖率），同时采用不同的静态和动态配置参数。我首先分别介绍静态和动态分析中考虑的参数（即配置项）（分别称为静态配置 和 动态配置），然后说明整体（混合）配置编码。

静态配置。 静态配置有两个参数维度：数据选择和敏感性。在数据选择 维度中，一个参数静态图 涉及是否使用静态数据，并决定 SEADS 是否通过遍历每个组件的静态依赖图，使用静态依赖关系计算给定查询的动态依赖关系，从而推断出精确的依赖关系，但时间开销较高。

敏感性 维度包含两个参数（上下文敏感性 和流敏感性），预期能够实现相对较高的精确率，具体说明如下。

1. 上下文敏感性 涉及同一变量/对象（例如方法）在不同调用上下文中的效应。上下文不敏感分析对所有调用上下文中的一个变量/对象使用单一抽象。相反，上下文敏感分析会区分变量/对象的不同调用上下文，并针对这些调用上下文分别计算信息 [217, 117]。例如，如果一个方法在不同调用点被多次调用，上下文敏感分析会区分这些调用点，并分别计算分析事实（例如依赖关系）。

实现上下文敏感分析有两种方法（函数式方法和调用字符串方法），两者在上下文抽象和分析算法上有所不同。函数式方法使用方法参数，并通过一个摘要函数将每个上下文映射到该方法对分析事实产生的相应效应。调用字符串方法使用调用点字符串，并为分析事实标记上下文；随后，分析会沿着被分析系统的流图传播这些带标记的事实 [205]。使用调用点字符串作为上下文的上下文敏感分析通常称为k-CFA，其中 CFA 是控制流分析的缩写，k 是字符串长度（即调用长度）的整数上限，代表调用点的层级 [209, 165, 147]。特别是，0-CFA 中的 0 表示上下文不敏感，其运行速度很快。然而，其分析结果可能不够精确 [113]。

2. 流敏感性 会考虑控制流可达性。流不敏感分析不会考虑实体（例如语句）的执行顺序（即控制流），而流敏感分析方法在计算信息（例如实体之间的依赖关系）时会考虑执行顺序。例如，如果一个变量在程序中被定义两次，然后使用一次，流不敏感分析只会考虑第二次定义和这次使用。然而，流敏感分析会在执行过程中区分这两次定义与使用的先后顺序。因此，只有流敏感分析才会报告相关信息（例如第一次定义与使用之间的依赖关系） [195]。流敏感分析需要额外的（时间）成本来分析实体执行顺序，因此成本更高，但比流不敏感分析精确得多。

动态配置。 SEADS考虑了动态配置参数的两个维度：数据选择和数据粒度 。数据选择 维度涉及所使用的数据类型，包括方法事件 和 语句覆盖率 参数。

1. 参数 方法事件 决定 SEADS 是否使用方法（入口 和 返回-到）事件来计算依赖关系。启用该参数后， SEADS 会根据动态数据（例如方法事件）推断出更精确的依赖关系，但需要付出额外成本。否则，在禁用参数 方法事件时， SEADS 会不精确但快速地计算依赖集，而不使用方法事件。

2. 参数 语句覆盖率 决定 SEADS 是否使用语句覆盖信息对静态依赖图进行剪枝。这意味着启用 语句覆盖率后， SEADS只考虑执行过程中覆盖到的语句，并舍弃其他语句。因此，在参考静态依赖关系时舍弃其他语句， SEADS 能获得比未启用时更精确的结果。

在数据粒度维度中只有一个参数*MethodInstanceLevel*，它涉及依赖关系计算中所用动态数据（即方法事件）的粒度。该参数用于确定 SEADS 是否使用所有方法事件实例来计算依赖关系。若启用此参数， SEADS 会使用（入口和返回到）事件的所有实例，更精确地计算依赖关系，但监控成本更高，且会使用更多动态数据。否则，只使用收集到的第一个入口方法事件和最后一个返回到方法事件；计算速度更快，但较为粗略（即精确度较低）。**配置编码**。 SEADS 的整体配置由上述静态配置和动态配置组成。三个比特用于编码三个参数,

表 5.1： 混合（静态/动态）配置编码

编码	静态配置			动态配置		
	数据选择	敏感性		数据选择		数据粒度
	静态图	上下文敏感性	Flow 敏感性	方法事件	语句覆盖率	方法实例级别
000000	禁用 (0) 禁用	(0)禁用 (0)	禁用 (0)禁用	(0)		禁用 (0)
000001	禁用 (0) 禁用	(0)禁用 (0)	禁用 (0)禁用	(0)		启用 (1)
111110	启用 (1)	启用 (1) 启用 (1)	启用 (1)	启用 (1)		禁用 (0)
111111	启用 (1)	启用 (1) 启用 (1)	启用 (1)	启用 (1)		启用 (1)

静态配置，其余比特用于动态配置参数。第一至第三个比特编码为静态配置参数，第四至第六个比特用作动态配置参数。二进制数1或0分别表示相应参数已启用或已禁用。因此，包含静态和动态配置参数的整体（即混合）配置编码为一个6位二进制数，范围从000000到111111。如表 5.1所示，在配置编码中，混合配置参数的顺序为：*staticGraph*、上下文敏感性、流敏感性、方法事件、语句覆盖率和*methodInstanceLevel*。

所有 64 种可能的混合配置中，有些配置无效，无法在 Seads 中使用。原因是某些配置参数依赖于其他参数，只有在启用其他参数时才有意义。例如，三个参数（上下文敏感性、流敏感性、语句覆盖率）依赖于参数 静态图。如果禁用参数静态图 ，也就是说， Seads 不使用静态数据（即静态依赖图），那么这三个相关参数（上下文敏感性、流敏感性和语句覆盖率）便毫无意义——语句覆盖率数据仅用于在 Seads中剪枝静态依赖图。因此，配置 001xxx、010xxx、

011xxx 和 0xxx1x 均无效，其中 'x' 是可以取 0 或 1 的位。另一个参数方法实例级别 依赖于参数 方法事件 ，因此配置 xxx0x1 无效。配置 000000 也无效，这意味着分析中没有使用任何数据。总之，在 SEADS 中有 38 种无效配置和 26 种有效配置。

5.2.3 Instrumenter

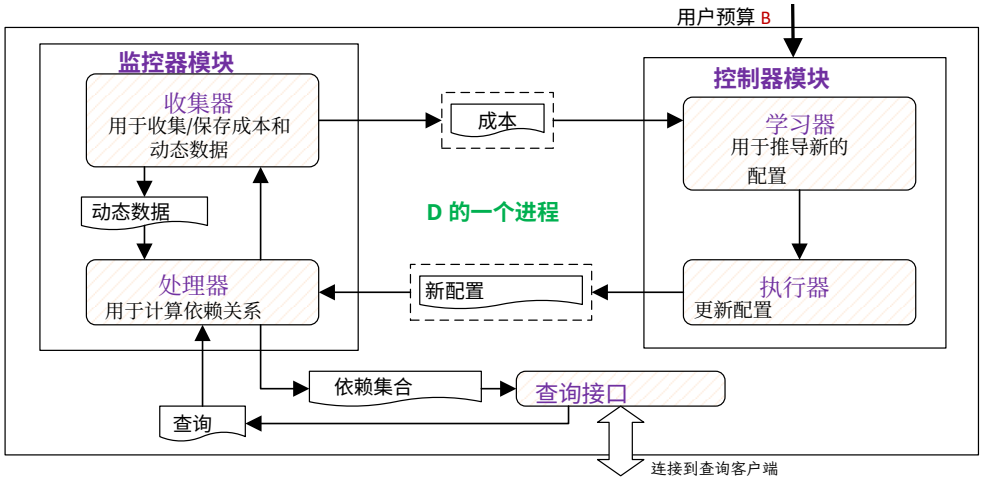
插桩器将探针插入系统 D 以生成将持续运行的插桩版本D' 。在执行期间，这些探针监控并记录所有已执行方法的进入和返回至事件，以推断方法先发生关系，并进一步近似进程内及跨进程方法之间的动态依赖关系 [39]。

插桩器还会对语句分支进行探测，以高效推断语句覆- 盖率 [40], ，这是所考虑的另一类动态数据。具体而言，除与显式谓词相关的分支外，方法进入也被视为特殊分支（即进入分支），其 `true` 边通向相应的方法（执行）进入点。

5.2.4 Monitor

插桩完成后，启动插桩系统 D' ，并使其在各个 N 分布式进程中持续运行。每个进程中的监控器和控制器 D' 也会随进程启动并持续运行，如图5.2所示。在进程执行期间，作为 SEADS的核心组件，监控器决定何时 需要调整分析配置（即仲裁），并使用当前配置计算动态依赖关系（即依赖计算）。该模块由

图 5.2： 监控器和控制器与经过插桩的对象一起运行 t



两个子模块组成：一个收集器，用于收集动态数据（方法执行事件和/或语句覆盖率）；另一个处理器，用于接收数据并进行依赖计算。具体而言，当 (i) 距离上一次依赖计算的时间间隔超过阈值（例如 15 分钟），且 (ii) 自上一次依赖计算以来累积的方法执行事件数量超过另一个阈值（例如 1000）时，处理器会计算/更新所有可能查询（即已执行的方法）的动态依赖关系。作为 SEADS 设置的一部分，用户可以自定义这两个阈值。

仲裁。 算法 3 展示了仲裁伪代码，该代码决定何时以及如何触发依赖计算和配置调整。在此算法中，我将一些变量设为输入： $method$ 表示已执行的方法， $gCounter$ 表示方法事件计数器的值， $LastT$ 表示上次计算的时间， B 表示用户预算，QU表示方法事件队列，$isTimeOut$表示用于记录超时的布尔值， TC/TT分别表示事件数量和分析时间间隔的阈值， $sgc\ T/sgl\ T/d\ T$ 分别表示构建/加载静态图以及计算依赖关系时的超时时间， $TCN/oldTCN$分别表示当前配置和紧邻其之前的配置。

算法 3 仲裁

1: 设置 $gCounter = 0$、 $LastT = 0$、 $QU = \emptyset$、 TCN =111111、 $oldTCN$ =为空
2: 将 $sgc\ T$、$sgl\ T$和 $d\ T$ 从 B中分配
3: **当** true**时**
4: **如果** 事件($method$)==entry**则**
5: $gCounter{+}{+}$
6: 将 (-$method$) 添加到 QU
7: **结束条件判断**
8: **如果** 事件($method$)==返回Into**则**
9: $gCounter{+}{+}$
10: 将 $method$ 添加到 QU
11: **如果** $gCounter > TC\ and (CurrentTime\text{-}\ LastT) > TT$**则**
12: 读取当前配置参数
13: $isTimeOut =$ 假
14: **如果** $Configuration_staticGraph$ **那么**
15: **如果** TCN/$oldTCN$ 的静态配置参数不同 **那么**
16: 构建新的静态（依赖）图
17: **如果** （非 $isTimeOut$） and （$constructTime > sgc\ T$）**那么**
18: 取消静态图构建，并将 $isTimeOut =$ 设置为 true
19: **结束如果**
20: **结束 if**
21: **如果** 静态图 $exists$ **则**
22: 加载静态图
23: **如果** (not $isTimeOut$) and($loadTime > sgl\ T$) **则**
24: 取消静态图加载，并将 $isTimeOut =$ 设置为 true
25: **结束 if**
26: **结束条件判断**
27: **结束条件判断**
28: **如果**非 $isTimeOut$ **则**
29: 调用处理器，使用 TCN 计算依赖关系
30: **如果**(非 $isTimeOut$) and($ComputeTime > d\ T$) **则**
31: 取消依赖计算，并将 $isTimeOut =$ 设为 true
32: **结束如果**
33: **结束如果**
34: $gCounter = 0, LastT = CurrentTime$
35: 调用收集器记录分析的时间开销
36: 调用控制器计算新的配置 $newTCN$
37: $oldTCN = TCN$
38: $TCN = newTCN$
39: **结束如果**
40: **结束如果**
41: **结束循环**

SEADS首先初始化 $gCounter$、 QU和 $LastT$，并将当前配置 TCN设为111111，以便从最精确但可能也是最慢的分析开始（第 1 行）。变量 $oldTCN$ 初始化为 `None` ，用于保存先前的配置。变量 $sgc\ T$、 $sgl\ T$和 $d\ T$ 分别是构建/加载静态依赖图以及计算依赖关系的超时值，其数值根据给定的用户预算 B 进行经验分配（第 2 行）。例如，我可以将用户预算设为 30 秒 $sgc\ T$，并分别将 $sgl\ T$和 $d\ T$ 设为 21 秒、6 秒和 3 秒。

接下来进入一个无限循环，通过调用（协同使用）与该监控器处于同一进程的控制器模块，在依赖计算和配置调整之间进行仲裁（第 3–40 行）。执行期间，每发生一次 $method$ 进入事件， SEADS就递增 $gCounter$ ，并将负数 $method$ （id）加到 QU 上（第 4–6 行）。例如，$gCounter$ 为 1，负数 $method$ （id）（如 -571）会被加到 QU上。每发生一次 返回事件， SEADS 也会递增 $gCounter$ ，但会将 $method$ （id）加到 QU 上（第 8–10 行）。例如， $gCounter$ 为 2， $method$ （id）（如 -571）会被加到 QU上。

如果 $gCounter$ 大于 TC ，且时间跨度（当前时间与上次计算时间 $LastT$之间）大于 TT（即条件 (i) 和 (ii) 均满足），SEADS 将开始新一轮分析（例如，为所有可能的查询更新动态依赖关系），具体如下。首先， SEADS 读取当前配置，并将 $isTimeOut$ 设为 false（第 11–13 行）。例如，发生 1000 个事件且经过 3 分钟后， $gCounter > TC$（例如，1000）以及（当前时间 - $LastT$） $> TT$（例如，3 分钟）。

然后， SEADS 读取当前配置 TCN $= 111111$，且 $isTimeOut =$ 为 false。如果启用了静态图 参数，并且至少一个静态分析参数（即配置的第一至第三位）在当前配置与紧邻的前一配置之间发生变化，

那么 SEADS 会使用静态配置构建新的静态依赖图（第 14–16 行）。例如，当前 TCN $= 111111$ 中，第一至第三位（三个静态参数）均为 1（启用）。因此， SEADS会构建新的静态依赖图，并同时应用上下文敏感性和流敏感性。静态依赖图准备就绪后， SEADS 会加载该图（第 21–22 行）。静态分析会复用 DIVER [39] 和 DIVERONLINE [37]的相关算法。

此外，处理器会在当前配置 TCN下被调用以计算依赖关系（第 29 行），具体如算法4所示。当 $isTimeOut$为 false 时，如果静态或动态分析的任一部分——（1）构建静态依赖图、（2）加载静态依赖图以及（3）计算动态依赖关系（分别耗费 $construct\ Time$、$loadTime$ 和 $computeTime$ 时间）——运行超时， SEADS 就会取消相应的分析部分，并将 $isTimeOut$ 设为 true（第 17–18、23–24、28–31 行）。例如，在当前配置 TCN $= 111111$下， SEADS 未能在 $sgc\ T$ 时间内（例如 21 秒）完成静态图的构建，因此会取消静态图的构建。

接着， $isTimeOut$ 为 true，因此 SEADS 会跳过静态图加载和依赖计算。重置 $gCounter$ 和 $LastT$后， SEADS 会调用收集器，收集当前配置下上述分析（即构建/加载静态图以及计算依赖关系）的时间开销，然后调用控制器计算下一个配置 $newTCN$ （第 34–36 行），具体内容将在算法 5 中详述。例如，此时 $gCounter = 0$，时间开销为 $= 22\ (> 21)$ 秒，并且 $newTCN = 000101$。

最后，算法会相应地更新当前配置（TCN）和先前配置（$oldTCN$），以便进行下一次仲裁迭代（第 37–38 行）。例如，我现在得到了 $oldTCN = 111111$ 和 $TCN = 000101$ 。

依赖计算。 当 SEADS 调用处理器来计算依赖关系时，会采用基于 DIVERONLINE [37] 的在线分析，以避免执行跟踪，从而节省分析成本，例如存储和 I/O 成本。算法4给出了用于计算依赖关系的在线算法伪代码。在算法4中， QU 与算法 3 中的相同，都是方法事件队列； DS（m）是方法 m 的依赖集。

首先，从当前配置中读取四个配置参数变量（即 $Configuration$ $staticGraph$、$methodEvent$、 $statementCoverage$ 和 $methodInstanceLevel$）（第 1 行）。例如，根据当前配置 000101，这些变量分别被赋值为 0、1、0、1。如果参数 方法实例级别 被禁用，SEADS 会从 QU中的事件序列中过滤掉第一个 进入 事件和最后一个 返回到 事件（第 2–3 行）。例如，由于 $methodInstanceLevel$ 已启用， SEADS 会跳过过滤。如果 $staticGraph$ 和 $statementCoverage$均已启用，则会根据语句覆盖率对静态依赖图进行剪枝（第 5–6 行）。例如，由于 $staticGraph$和 $statementCoverage$ 均已禁用， SEADS 会跳过剪枝。然后， SEADS 遍历 QU，计算与 QU 中的方法事件相对应的动态依赖关系（第 8–34 行）。

对于 QU中的每个事件 e ，我将 m 设为 e 对应的方法，并将 DS（m）设为 m 的依赖集，该依赖集初始为空（第 9 行）。例如，对于 e = -571， DS（e）当前为空，其中 571 是方法 *org.apache.zookeeper.server.ZooKeeperServerMain:void main(java.lang.String[])*的 ID。如果参数 方法事件 已启用、 e 的值为负（即 e 是一个 进入 事件），且执行了 m ，则 SEADS 会将其自身添加到依赖集 DS（m）中（第 8–10 行）。例如，由于参数方法事件 已启用，且 e =-571（< 0），ID 为 $= 571$的方法会被添加到 DS（m）中。

如果两个参数 方法事件 和静态图 均已启用， SEADS 会添加依赖 -

算法4 计算依赖关系

令 $Configuration\ staticGraph$ 为当前配置的 $staticGraph$ 参数
令 $Configuration\ methodEvent$ 为当前配置的 $methodEvent$ 参数
令 $Configuration\ statementCoverage$ 为当前
配置
令 $Configuration\ methodInstanceLevel$ 为当前
配置
设 $DS(m)$ 为方法 m 的依赖集

```
1: 读取当前配置参数设置
2: 如果 Not Configuration_methodInstanceLevel，则
3:     QU=获取首末实例(QU)
4: 结束 if
5: 如果 Configuration staticGraph 且 Configuration statementCoverage 则
6:     根据语句覆盖率剪枝静态图
7: 结束如果
8: 对于每个 方法事件 e ∈ QU 执行
9:     m=绝对值(e), DS(m) = ∅
10:    如果 Configuration methodEvent 则
11:        如果 e < 0 那么
12:            DS(m) ∪ = {m}
13:        结束如果
14:        如果 Configuration_staticGraph 那么
15:            如果 e < 0 那么
16:                AddDSEntry(m)
17:            else
18:                添加数据结构返回(m)
19:            结束如果
20:        else
21:            如果 e < 0 则
22:                对于每个 最后一次返回进入事件 e′ 在 e ∈ Q 之后发生执行
23:                    m′=绝对值(e′)
24:                    DS(m) ∪ = {m′}
25:                结束循环
26:            结束条件判断
27:        结束条件判断
28:    else
29:        如果 Configuration staticGraph 则
30:            AddDSEntry(m)
31:            添加数据结构返回(m)
32:        结束条件判断
33:    结束条件判断
34: 结束循环
```

通过调用函数 AddDSEntry 获取负 e 值（进入 事件），或调用另一个函数 AddDSReturnInfo 获取正 e 值（返回调用方 事件）（第 14–18 行）。如果未启用参数 静态图， SEADS 会跳过这两个子程序。我利用DIVERONLINE [37] 开发了这两个子程序，其中 SEADS 遍历静态依赖图，将依赖关系添加到 $DS(m)$ 中，并分别对方法（m）的 进入 事件和返回调用方 事件应用不同的依赖传播规则。如果启用了参数方法事件，但禁用了参数 静态图，那么每次出现负 e （即 进入 事件）时， SEADS 都会将 QU 中最后一个（返回调用方）事件发生在 e 之后的所有方法添加到依赖集 $DS(m)$ 中（第 20–24 行）。例如，启用 方法事件 且禁用 静态图时， SEADS 会将 QU 中最后一个（返回调用方）事件发生在 e （-571）之后的所有方法添加到依赖集 $DS(m)$ 中。

如果方法事件 已禁用而静态图 已启用， SEADS 只需调用这两个函数 AddDSEntry & AddDSReturnInfo （第 28–31 行），通过遍历静态依赖图，在不使用任何动态数据的情况下，将依赖关系添加到 $DS(m)$。如果方法事件已启用而静态图 已禁用， SEADS 则会跳过此处的这两个函数。

值得注意的是，对于每个进程，监控器只计算该进程内的运行时依赖关系（称为进程内依赖）。进程之间的依赖关系（即进程间依赖）则由查询客户端 在收到来自每个进程的 Q 的进程内依赖关系后，针对给定的查询 Q 进行计算。

查询接口。为了与用户交互，每个进程的监控器模块都包含一个查询接口，它接收来自 Q 的依赖查询，并通过网络将相应的依赖集发送回查询客户端模块。

设施（见图 5.1）。当监控器模块的查询接口收到查询时，需要分别处理以下两种情况：（1）如果监控器正在计算或更新所有可能查询的依赖集，则查询接口会等待依赖计算或更新完成，然后运行 Q的依赖集。（2）如果监控器正在执行仲裁功能，但在上一轮依赖计算完成后当前并未计算依赖关系，而是在等待下一轮计算，则查询接口会立即返回查询 Q最近计算出的依赖集。

每个进程的查询接口（附接于监控器）负责计算进程内运行时依赖关系，而查询客户端模块则通过合并所有每个进程的进程内依赖集，推导进程间依赖，从而为用户提供的查询 Q生成最终依赖集。一旦收到 Q，查询客户端便将其发送给每个进程的查询接口，然后等待所有每个进程的接口返回各自针对所有可能查询的进程内依赖集。原因在于，计算查询 Q的进程间依赖需要所有这些依赖集。

更具体地说，查询客户端将识别最先执行查询 Q 的进程 P_i（即 Q 最早的首次进入事件发生所在的进程）。

1. 如果没有任何进程执行过 Q，则将返回一个空依赖集。

2. 否则，将 Q 的最终依赖集（记作 $fDS(Q)$）初始化为从针对 P_i 的 查询接口 返回的 Q 的进程内依赖集（记作$intraDS(Q,\ i)$）。

然后，对于每个其他进程 P_j，如果 P_j 也执行了 Q，则将 $intraDS(Q,\ j)$ 直接合并到 $fDS(Q)$；否则，对于在 P_j 中执行的每个方法 m，如果最后一次返回事件发生在 Q 的首次进入事件之后，则将 $intraDS(m,\ j)$ 合并到 $fDS(Q)$。根据系统所有进程中方法执行事件之间的先发生关系，这一合并过程会隐式推导出 Q 的进程间依赖，并将其添加到 $fDS(Q)$ 中。

5.2.5 Controller

对于系统中的每个进程， SEADS都会使用一个控制器来调整其分析配置，如图5.2所示。控制器将当前配置的成本和用户预算 B 作为输入，以确定分析应采用哪种下一步配置，从而在用户预算（即预算下的总分析时间限制）方面实现比当前配置更好的成本效益。

每个控制器模块由两个子模块组成：学习器和执行器。学习器参考用户预算，利用收集器提供的数据（即当前配置下的分析成本）来调整配置。调整后的配置可能与当前配置相同，也可能不同。然后，执行器将 SEADS 更新为学习器输出的新配置，并将该配置直接传递给协作监控器。接下来，我将详细说明学习器进行配置调整的内部机制。

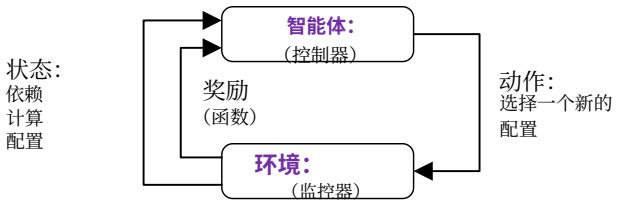
SEADS 采用强化学习方法，尤其是Q学习方法来确定新的分析配置。由于监督学习需要大型训练集，因此不适用于 SEADS中的配置调整。数据量不足以

在 SEADS 针对特定系统启动时用于训练。与此同时，由于不同系统的执行动态特性可能差异很大，事先从其他系统学习也可能无效。因此，鉴于系统执行期间环境 会不可预测地变化，不受这些限制的强化学习更为适合。此外，作为一种特殊类型的强化学习，Q学习尤其适用于 SEADS中的配置调整，因为在没有现有调整策略或模型的情况下，依赖计算时间开销在执行期间会不断变化 [230]。因此，我们采用了作为离策略、无模型学习策略的Q学习方法。

在 SEADS 中应用的 Q学习中，智能体从环境接收一个状态（当前分析配置），并采取一个动作（即选择新的配置）；它可以从 Q表中选择，也可以通过随机探索可能的动作来选择。随后，智能体会收到根据该动作的性能计算得出的奖励反馈。如图5.3所示，状态代表当前的依赖计算配置，监控器 是环境，而控制器 是智能体。根据用户预算和依赖计算的时间开销计算出奖励，并将其作为反馈发送给智能体。奖励为正时，会鼓励/强化相应的动作；否则，则会抑制该动作。Q学习利用奖励更新 Q表，其中数值最大的项预计会被选作后续动作 [94]。换言之，奖励越大，相应配置被选中的可能性就越高。

Q学习根据贝尔曼方程更新Q表，以寻找最优策略 [124]。利用该方程，下一状态在Q表中的值（即 Vq）通过以下两个参数（γ 和 α）计算得出：

图 5.3： Q学习中智能体与环境之间的交互



$$Vq = Vq + \alpha * [reward + \gamma * \text{最大}\{q|q \in Q\text{-}table\} - Vq] \tag{5.1}$$

这里， γ 是一个介于0到1之间的折扣因子，与未来奖励的重要性相关。如果γ 为0，Q学习智能体只考虑当前奖励；如果 γ 为1，智能体则会努力获得长期高奖励。而 α 是一个介于0到1之间的学习率，用于控制对先前值与新 Vq 值之间差异的考虑程度。如果 α 为0，智能体只利用先验知识；如果 α 为1，智能体则忽略先验知识，只考虑最新信息来探索各种可能性。

算法 5 展示了用于配置调整的学习伪代码。首先， SEADS初始化 Q学习组件/变量（例如， $learner$、 $Agent$、 $Qtable$、 $actions$、 $rewards$、ϵ 和 $states$）以及贝尔曼方程的参数： γ 和 α （第 1–2 行）。例如，我将 Q表中的值全部设为零，并分别将参数 γ、α 和 ϵ 设为 0.9、0.9 和 0.2。由于 γ=0.9 略低于 1，Q学习智能体更倾向于长期高奖励，而非当前奖励。由于 α (0.9)，智能体更倾向于使用最新的数据（例如奖励、Q表中的值）。

ϵ 用于控制智能体从 Q表中采取一个动作（即选择一个新配置），或通过随机探索可能的动作。如果随机计算出的变量值 <= (1 - ϵ)，智能体就使用 ε-贪心策略 [238] ，根据 Q表中最大的 Vq 采取最佳动作。否则，智能体会随机选择一个动-

算法 5 使用 Q学习进行配置调整

令 TCN和 $newTCN$分别表示当前配置和新配置；令 $Qtable$、 $actions$、 $rewards$、 $states$、 γ、 α 和 ϵ 表示 Q学习使用的组件和参数；令 T 表示当前配置下动态依赖分析的总时间开销；令 B 表示用户预算；令 $probability$ 表示根据 $Qtable$中的最大值选择动作的概率。1：初始化 Q学习组件： $learner$、 $Agent$、 $Qtable$、 $actions$、 $rewards$和 $states$。2：设置贝尔曼方程参数 γ α ϵ 0 1 、 和 以及 和 之间的关系：更新 /(-) $reward = 1 \quad B \quad T \quad 1000$ * 。4：使用贝尔曼方程更新 $Qtable$ 。5： $probability$=random(0,1)。6：如果 $probability <=$ (1 - ϵ), 则 7：根据 $Qtable$中的最大值采取最佳动作。8：否则 9：随机采取一个动作。10：结束 if。11：返回新配置 $newTCN$

置 [78]。（第 8–9 行）。现在 $\epsilon = 0$。2 和 (1 - ϵ) $= 0.8 = 80\%$。因此，智能体根据 Q表中的最大值采取最佳动作的可能性为 80%，而随机选择的可能性为 20%。

接下来，更新 $reward$ 和 $Qtable$ （第 3–4 行）。 $reward$ 定义为 1/(用户预算 B - 当前依赖分析时间开销 T) * 1000，其依据是：分析成本越接近用户预算（此处的学习目标），奖励就越高。例如，假设用户预算 B 为 60000（毫秒），当前分析时间开销 T 为 40000 毫秒。那么， $reward$ 就是 1/(60000 - 40000) * $1000 = 0.05$，用于更新 Q表。在算法中，动作表示从当前状态（配置）转移到下一状态（配置）。如果随机计算出的 $probability$ 值等于或小于 (1 - ϵ)，Q学习智能体 $Agent$ 就会使用 ε-贪心策略 [19]，根据 $Qtable$ 中的最大值采取最佳动作（第 5–7 行）。例如，假设 SEADS 随机计算出的 $probability$ 值为 0.803（> 0.8）。

此时， SEADS 会跳过根据 $Qtable$ 中的最大值选择最佳动作。否则， $Agent$ 会随机选择一个动作（第 8–9 行）。例如， $Agent$ 可能会

随机选择一个动作（例如选择下一个配置（如 000101））。最终，会返回一个新配置 $newTCN$ （第 11 行）。

5.2.6 Querying client

Seads 使用查询客户端与用户进行交互。通过此模块，用户向插桩系统 D' 中每个进程内的查询接口发送依赖查询 Q，并等待其响应。当某个进程中的进程内依赖计算完成后，计算得到的依赖关系会被传回该进程内的查询接口，查询客户端则从中接收这些已计算出的进程内依赖。

-

所有进程中的进程内分析完成后，查询客户端会执行进程间分析，根据与部分有序的方法事件相关联的时间戳拼接收到的所有进程内依赖集，同时利用基本的消息传递语义来减少误报 [91]。

5.2.7 Limitations

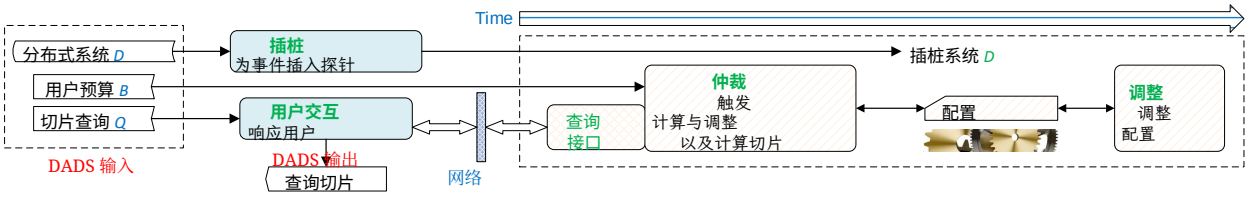
在插桩过程中， Seads 需要在被分析系统（SUA）的字节码中插入探针，以监控方法事件。如果管理员不允许修改SUA， Seads 就无法工作。

Seads 旨在提供在响应时间约束（即用户预算）内获得的最具成本效益的结果（即依赖集）。然而，当前的控制器（Q-learning算法）可能并非最优。动态依赖计算所采用的配置不一定能实现相对于 Seads可能达到的最优成本效益权衡。例如，Q-learning算法可能会在某些

步骤中采取错误的动作（尤其是在选择是随机的情况下，如算法 5所示）。因此，动态依赖分析可能并不总是最具成本效益的。

此外，如果用户设置了不合适的预算（例如，与特定系统的典型响应时间相差甚远的预算），控制器执行的分析配置调整可能会更加低效（即导致分析 SEADS 进一步偏离最优的成本效益平衡）。另一方面，当用户未指定预算时，SEADS 将不得不使用默认预算，这可能不符合用户的期望，或不适用于给定的系统。

图 5.4： DADS 架构概述



5.3 工具实现

为了在分析持续运行的分布式软件时避免跟踪及相关的时间/空间成本，我开发了 DADS [90, 85]。如图 5.4 所示，它是一款能够根据用户指定的预算约束工作的在线、可扩展且高成本效益的动态切片器。 DADS 采用分布式设计，以利用并行和分布式计算/存储资源。此外， DADS 通过强化学习持续自动地即时调整其分析配置，以根据给定预算实现并维持实际可扩展性和成本效益权衡。针对现实世界中的八个 Java 分布式系统进行适配后， DADS展现出其可扩展性和成本效益优势

表 5.2： 实验对象

对象（版本）#方法	#源代码行数		测试类型
NioEcho（r69）	27	412	集成
MultiChat（r5）	37	470	集成
OpenChord（v1.0.5）	736	9,244	集成
Thrift（v0.11.0）	1,941	14,510	集成
xSocket (v2.8.15)	2,209	15,760	集成
ZooKeeper（v3.4.11）	5,383	62,194	集成、负载、系统
Netty（v4.1.19）	12,389	167,961	集成
Voldemort（v1.9.6）	20,406	115,310	集成、负载、系统

与不具备配置学习和调整能力的类似切片器相比， DADS 具有优势。通过在给定预算内生成动态切片， DADS 能够在时间预算约束下助力软件维护、测试和安全性任务。

5.4 评估

5.4.1 Experiment setup

我将 SEADS 应用于八个通常持续运行的 Java 分布式系统，如表 5.2 所示。这些被测程序的规模以被测程序源代码中定义的方法数量（#方法数）以及不包括空行和代码注释的 Java 源代码行数（#源代码行数）来衡量。最后一列（测试类型）显示了测试类型，包括集成测试、负载测试和系统测试。我选择了涵盖不同架构、领域和规模的这些被测程序。

为了展示这项创新的影响和优势，由于缺乏可直接比较的同类技术，我创建并使用了一个D^2ABS [38]的在线版本（称为 DODA）作为评估中的基线；据我所知，D^2ABS [38]是针对分布式程序的最先进动态依赖分析。在实现方面，

Doda 本质上是 Seads 的一个版本，它不会动态更改其分析配置，而是始终使用固定配置（111111，用于可能具有最高精确率但代价高昂的分析）。

5.4.2 Results and analysis

本节报告并讨论实证结果。表5.3展示了主要结果，包括针对（第一列中的）12次被测程序执行， Seads 相对于基线 Doda的开销和精确率指标。第三至第五列和第六至第八列分别列出了使用 Doda和Seads时，每次插桩受试程序执行的总运行时间（**运行时间**）、运行时减速（**减速**）以及平均查询响应时间（**响应时间**）。每次被测程序执行的正常运行时间（第二列），是被测程序在与相应插桩受试程序执行相同的一系列运行时输入驱动下的总执行时间。在两种技术进行分析时，每次被测程序执行所使用的运行时输入完全相同，此外，两种技术接收的查询序列也完全相同。

在第九列（精确率）中，我报告了 Seads 相对于基线的平均相对精度：对于每个查询，该指标计算为 Doda 计算出的查询依赖集合规模与 Seads 计算出的查询依赖集合规模之比。我计算并报告这一相对指标有两个原因：(1) 样本查询没有可用的真实依赖集合，也没有现有工具能够扩展到这些受试系统并与之协同工作，以计算此类真实值；(2) 评估目标主要是验证 Seads 相较于采用某种配置的传统分析，在可扩展性和成本效益方面的优势。

表5.3： 时间（以秒为单位）和存储成本（以 MB 为单位），以及 SEADS 相对于 DODA 基线的精确率（比率）

执行	正常 Run Time	DODA Run Time	DODA Slow Down	DODA 响应 Time	SEADS Run Time	SEADS Slow Down	SEADS 响应 Time	精度存储	
NioEcho	158.37	228.17	44.07%	14.39	214.15	35.22%	13.71	100.00%	2.00
MultiChat	148.96	241.89	62.39%	15.12	223.67	50.15%	14.32	100.00%	2.00
Openchord	233.78	606.87	159.59%	51.36	359.37	53.72%	25.33	77.44%	14.00
Thrift	199.87	573.49	186.93%	45.23	345.19	72.71%	23.82	90.68%	25.00
xSocket	380.59	1,817.61	377.58%	170.82	772.38	102.94%	65.37	83.25%	21.00
Netty	589.39	4,218.85	615.80%	409.56	1,226.16	108.04%	115.16	87.12%	105.00
ZooKeeper 集成	598.34	3,543.47	492.22%	343.19	1,139.25	90.40%	103.21	66.29%	96.00
ZooKeeper Load	616.16	3,804.37	517.43%	368.43	1,209.77	96.34%	111.97	65.55%	96.00
ZooKeeper 系统	598.17	3,676.91	514.69%	355.56	1,183.97	97.93%	107.94	63.28%	96.00
Voldemort 集成	398.06	-	-	-	791.37	98.81%	69.62	-	200.00
Voldemort Load	194.44	-	-	-	731.71	276.32%	67.58	-	200.00
Voldemort 系统	355.78	-	-	-	719.38	102.20%	66.96	-	200.00
平均值:	372.66	2,079.07	330%	197.07	743.03	99%	65.41	81.5%	88.08

最后一列（存储）列出了 SEADS 的总存储成本（占用的磁盘空间），从两个最小型被测程序（NioEcho 和 MultiChat）的 2MB，到最大型系统（Voldemort）的 200 MB 不等；12 次执行的平均成本为 88MB。这些空间主要用于存储每个被分析组件的静态依赖图以及插桩后的被分析对象版本。基线的存储成本与 SEADS 的几乎相同，因此未予列出。

针对 DODA Voldemort 执行的结果不可用（因此未列入表格），因为基线无法扩展到该系统：运行 12 小时后，我终止了分析。因此， SEADS 这些执行的相对精度也缺失。接下来，我将讨论这些结果中的主要发现和观察，以回答研究问题。

效率：响应时间。如第五列和第八列所示，响应时间是指从向每种依赖分析技术发送依赖查询，到用户收到返回的依赖集之间，用户等待的时间

。在所有被测程序执行中，对于随机间隔发送的随机用户查询以及用户指定的预算，SEADS 平均需要 65 秒作出响应。对于单次执行，针对 NioEcho 执行，SEADS 的平均响应时间最短（14 秒），这是因为其规模最小。另一方面，在被测程序中规模最大的系统 Netty 上，SEADS 的平均响应时间最长（115 秒）。然而，综合考察所有被测程序执行的这项效率指标，会发现被测程序规模与平均响应时间之间不存在一致的相关性。原因之一是，被测程序的源代码规模并非影响这项效率指标的唯一因素。例如，所分析执行的复杂度也是一个重要因素。其他因素（如网络通信的时间开销）也可能影响用户体验到的响应时间。事实上，对于同一被测程序 Voldemort，在三次被测程序执行中，使用 SEADS 得到的平均响应时间存在明显差异。

DODA在与发送给 SEADS 的相同查询请求下，对 12 次被测程序执行的平均耗时为 197 秒；单次被测程序执行的平均响应时间从 NioEcho 的 14 秒到 Netty 的 410 秒不等。特别是，对于 Voldemort，DODA 在 12 小时内无法回答任何查询。实际上，大多数用户（无论是人类用户，还是使用动态依赖结果的应用程序）都不愿意为了查询依赖集而等待超过 12 小时。换言之，DODA 可能面临严重的可扩展性问题，从而妨碍其在现实世界的大型分布式系统中的实际应用。相反，在所考察的 Voldemort 的三次执行中，SEADS 的响应时间很短，展现了其相较于传统依赖分析的可扩展性和效率优势。

就平均响应时间而言， SEADS 相较于 DODA 的效率优势通常会随着被测程序执行规模的增大而更加显著。如图所示，随着被测程序的源代码规模和执行复杂度增加，两种技术在这一效率指标上的差距也趋于扩大。具体而言，对于两个最小规模的被测程序，DODA 的平均响应时间与 SEADS 非常接近（每次查询的平均差距不到一秒）；对于 Thrift 这样的中等规模被测程序， SEADS 的速度约快 2 倍；对于规模最大的被测程序，SEADS 的速度快逾 3 倍。这进一步表明， SEADS 相较于 DODA 的可扩展性和效率优势在大型分布式系统中尤为显著。

效率：分析开销。表5.3显示， SEADS的运行时减速幅度介于35%（NioEcho）和276%（Voldemort负载测试）之间，在12次被测程序执行中平均为99%。与此同时， DODA 的运行时开销介于44%（NioEcho）和616%（Netty）之间，平均为330%。总体而言，正如预期，面对规模更大、执行过程更复杂的被测程序时，两种技术的减速幅度都更大。

由于 DODA 在12小时内未能针对Voldemort的三次执行回答任何依赖查询，因此相应的减速指标没有意义，故予以省略（$>$ (12 * 3600 - 398.06 / 398.06) $= 4$，319,900%）。对于其余9次被测程序执行，SEADS 的效率始终高于基线。

与平均响应时间方面的对比类似，随着被测程序的规模和执行复杂度不断增加，SEADS 相较于 DODA的优势也愈发显著。例如，对于两个规模最小、最简单的被测程序，两种技术的减速幅度相近；对于中等规模的被测程序 xSocket， DODA造成的减速幅度超过 SEADS的两倍；而对于

两个大型被测程序（Netty 和 ZooKeeper）， DODA的减速幅度是 SEADS 的 4 到 5 倍。总体而言，按照减速指标衡量， SEADS 的效率比传统动态依赖分析方法高出 3 倍以上，这进一步体现了通过强化学习在 SEADS中即时调整分析配置的优势。**成本效益：精度-成本比**。为评估 SEADS的成本效益，我首先需要比较由 DODA 和 SEADS计算出的查询依赖集合的平均规模，以计算精确率。如表5.3第九列（精确率）所示，每次主体执行的精确率是这样计算的：对于每个查询，将 DODA 计算出的依赖集规模除以 SEADS 针对同一查询计算出的依赖集规模，再取平均值。对于每个依赖查询，我还比较了两个依赖集的内容，并发现，对于评估中涉及的所有查询， SEADS的依赖集始终包含 DODA给出的依赖集。这证实，尽管通过调整分析配置牺牲了精确率以提高可扩展性和效率， SEADS 的召回率并未降低。这些相对有效性指标本质上将基线结果视为真实值。因此，鉴于两种技术的召回率均为 100%，在计算两种技术的成本效益比时，我只考虑 SEADS 的相对精度（ DODA 的精确率始终为 100%）。

对于有基线依赖集可用于测量 SEADS 相对精度的 9 次被测程序执行， SEADS 达到的精确率介于 63% 至 100% 之间，总体平均值为 82%。在最佳情况下，对于两个最小、最简单的被测程序执行（即 NioEcho 和 MultiChat）， SEADS 相较于基线没有损失任何精确率。这主要是因为在线分析

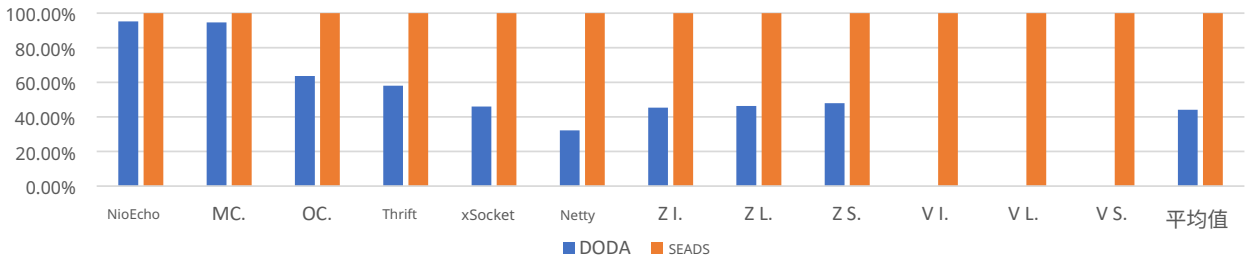
通过 SEADS 始终产生低于用户预算的时间开销，即使对这些被测程序采用最精确的分析配置也是如此。因此， SEADS 无需离开精确率最高的配置。

同样， SEADS 在 Zookeeper 系统测试中的精确率最低，为 63.28%；最可能的原因是，为了在给定用户预算的条件下保持可扩展性和效率， SEADS 对其分析配置进行了最积极且最频繁的调整。 SEADS 针对某次主体执行所达到的平均相对精确率（十个查询的平均值）与该主体执行的规模和复杂度有关，这一点也部分体现在两个效率指标中：响应时间和运行时减速。

为计算 SEADS 和 DODA的成本效益，我针对每次主体执行，计算了平均精确率（十个查询的平均值）与我所考虑的两种成本度量之一的比值：平均响应时间（十个查询的平均值）以及运行时减速（整个执行过程中十个查询的总体运行时减速）。因此，在对这两种技术进行成本效益评估和比较时，我分别针对两种成本度量采用了两个指标，如图 5.5（以平均响应时间为成本因素）和图 5.6（以运行时减速为成本因素）所示。

为了节省篇幅，便于展示，我在两张图中都使用了以下缩写（位于 x 轴上）：MC. 表示 MultiChat，OC. 表示 OpenChord，V 表示 Voldemort，Z 表示 ZooKeeper，I. 表示集成测试，L. 表示负载测试，S. 表示系统测试。具体而言，为了突出 SEADS 相较于 DODA的优势，两张图都只展示了每次主体执行的成本效益比较，即 DODA的成本效益占 SEADS成本效益的百分比。因此，两张图中SEADS 对应的柱形始终为 100%，而 DODA 的成本效益则显示为其占比

图 5.5: DODA 和 SEADS 每次执行的精确率与响应时间之比所表示的成本效益对比（y 轴）。比值越高，成本效益越高



的分数。由于 DODA 无法应用于 Voldemort，因此 DODA在 Voldemort 的三次执行中的相应成本效益指标均为零。计算成本效益时未考虑存储成本，因为它们几乎可以忽略不计（平均仅为 88MB，最多为 200MB）。

图 5.5 显示， DODA在响应时间方面的成本效益平均约为 SEADS 的 44%。对于单个被测程序执行，在最小、最简单的两个被测程序 NioEcho 和 MultiChat 上， DODA和 SEADS 的成本效益非常接近。在前文中，我观察到，与基线相比， SEADS 在应用于规模更大、复杂度更高的系统时，效率和可扩展性优势更为显著。这里也呈现出相同的比较趋势：与DODA相比， SEADS在规模较大、执行更复杂的被测程序上，成本效益优势更加明显。例如，对于 Open Chord 和 Thrift 等中等规模的被测程序， DODA 的成本效益约为 SEADS 的 60%；对于 ZooKeeper 等更大型系统，这一比例降至 45% 以下，而对于规模更大的被测程序 Netty，则进一步降至 30%。作为这一趋势的极端例子，在该项目中最具挑战性的被测程序 Voldemort 上， DODA的成本效益为零。

图 5.6: 对比 DODA 和 SEADS 每次执行的成本效益（y 轴），以精确率与运行时减速的比值表示（x轴）。比值越高，成本效益越高

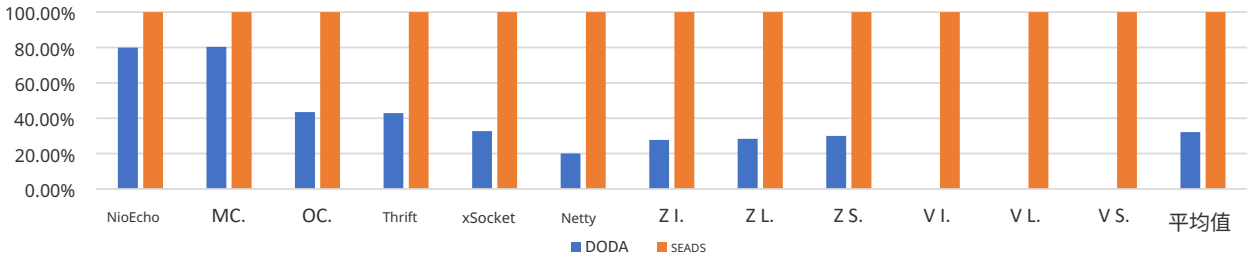


图 5.6 展示了另一种衡量方式下的成本效益，其中将运行时减速作为成本因素。总体而言，按这一成本指标衡量，DODA 的成本效益平均仅约为 SEADS 的 32%，显著低于图 5.5 所示的响应时间方面的成本效益。显然，在运行时减速方面，SEADS 相较于 DODA 的效率和可扩展性优势，大于其在平均响应时间方面的优势。另一方面，与图 5.5 所示的响应时间成本效益变化相比，运行时减速方面的成本效益变化同样与被测程序执行的规模相关：面对规模更大、复杂度更高的被测程序时，SEADS 的成本效益优势更明显。

总而言之，无论涉及哪种成本因素，SEADS 的成本效益都显著高于 DODA。这表明，SEADS 在精确率方面的牺牲，通过效率和可扩展性方面的收益得到了充分弥补，从而在总体成本效益上取得了最终优势。因此，在运行过程中动态调整分析配置的方法，似乎是现实世界中针对大规模分布式系统开展动态依赖分析的一种可扩展且具备成本效益的解决方案。

表 5　.4: Q-learning 的迭代次数和学习时间（秒）　ng

执行	#迭代次数	时间
NIOEcho	1	36.73
MultiChat	1	41.38
OpenChord	2	327.19
Thrift	2	316.70
xSocket	3	639.37
ZooKeeper 集成	4	1013.73
Zookeeper 负载	4	1097.26
Zookeeper 系统	4	1053.82
Netty	4	1132.98
Voldemort 集成	3	697.35
Voldemort 负载	3	632.42
Voldemort 系统	3	621.94
总体平均值：	**3**	**634.24**

配置学习速度。　SEADS 使用Q学习来学习分析配置，以获得并保持动态依赖分析的可扩展性和高成本效益。然而，其控制器模块需要一定时间进行学习，才能开始产生较为理想的决策（即下一步要切换到的配置）。事实上，作为近似动态规划中广泛用于马尔可夫决策过程（MDP）的一种迭代学习方法，Q学习通过多次迭代计算最优MDP策略，使平均动态特性在收敛性质下达到预期效果 [181]。为了展示 SEADS 学习具备成本效益的配置的速度，我在实证评估中收集了 SEADS 开始计算具备成本效益的结果之前的Q学习迭代次数和学习时间（即控制器开始稳定地选择能够实现成本效益动态依赖计算的下一配置时）。

表 5.4 展示了 12 次被测程序执行（执行）各自的学习迭代次数（$\#Iteration$）和学习时间（时间，单位为秒）。总体而言，对于更复杂的系统执行， SEADS 往往需要更多迭代才能学到更好的配置，这直观上

是因为这些执行的动态特性变化更大。由于学习时间包含依赖计算的时间开销（如表 5.3 所示），对于更复杂的系统执行， SEADS 计算依赖关系所需的时间更长，因此对依赖查询的响应也更慢。这解释了为何这些系统执行的学习时间通常也更长。总体平均而言， SEADS 需要经过 3 轮学习、耗时 634 秒，才能开始取得具有成本效益的结果。

5.4.3 Threats to validity

内部效度。 对内部效度的主要威胁在于 SEADS、 DODA 的实现以及我们的实验流程可能存在错误。任何一项实现中的错误都会损害实证结果的有效性，以及我们基于这些结果得出的结论。不过， SEADS 基于 Soot [143], ，这是一个历经十余年发展、日趋成熟的框架。 SEADS 和 DODA的许多关键组件，包括静态插桩、静态依赖分析、运行时监控/分析以及混合依赖计算的代码，均取自若干经过多年调试和优化的工具（例如 DIVER [39]、 DI-APRO [37]、 DISTEA [41]）。

为了尽量减少与 SEADS 的实现脚本和新开发组件（例如控制器）相关的威胁，我仔细审查了代码，并针对简单的样本（例如两个最小的被测程序）和案例（例如依赖集相对较小的查询）进行了人工检查，以确保 SEADS 的功能正确性。

外部效度。 外部效度面临的一项威胁在于，评估研究中所使用的被测程序及其执行是否具有代表性。所选被测程序可能无法很好地代表所有适用 SEADS 的真实世界分布式系统，而

针对每个所选被测程序考察的执行可能并未涵盖其所有典型行为。如果样本被测程序执行与其他分布式系统执行之间存在显著差异， SEADS 用户可能会获得与我所报告的不同的性能和优势。

我尝试通过考察涵盖不同规模、架构和应用领域的被测程序，以及多种执行场景，来降低这一威胁。尽管如此，为了最大限度地降低此类威胁，我还需要在真实世界分布式系统的实际部署环境中，使用真实的运维场景。

构念效度。 对构念效度的主要威胁在于，评估基线的选择可能并不理想。为避免潜在偏差，我使用了一款先进的在线动态依赖分析工具 DODA （由我和导师共同开发），该工具至少能够适用于一些（即使无法扩展到所有）真实世界分布式系统。此外，为降低这一威胁，我确保 DODA 和 SEADS 共享底层分析基础设施和一些实用工具。

结论效度。主要的结论效度威胁在于评估结果和结论的可推广性。同时， SEADS 与基线相比，在动态依赖分析方面的方法级结果可能无法推广到更细粒度的层级（例如语句级）。这是因为语句级动态分析将面临更大的效率和可扩展性挑战。由于所考察的被测程序及其执行数量有限、种类也不够多样，我无法声称这些结果能够推广到所有场景下的所有真实世界分布式系统。

5.5 相关工作

本节将讨论与以下主题密切相关的先前研究：分布式软件的依赖分析 以及具有可变成本效益的分析。

5.5.1 Dependence analysis for distributed software

早期方法试图将传统依赖分析算法扩展到并发程序中 [240, 239, 173, 98]。这些方法大多仅限于分析单个进程内的依赖关系。在 [142], 中，研究者提出了一种切片算法，将基于套接字的消息传递所产生的依赖关系纳入其中。由于该算法完全是静态的，因此对这些依赖关系的近似过于保守。

研究者也开发了若干动态切片算法，最初仅适用于过程式程序 [139, 43, 112, 74, 131] ，后来也适用于面向对象系统 [168, 28]。特别是， [28] 中的方法定义了由进程间通信引发的多种依赖关系。不过，该方法并未实现为可用于真实世界分布式软件的工具，而且其算法性质意味着存在可扩展性障碍。

通过基于逻辑时钟的偏序处理全局时序问题，从而推理先发生关系，是并发程序分析中的一种标准技术。该技术已用于测试并发程序和分布式系统。例如，DCatch 通过检查分布式执行是否符合一组先发生关系规则来检测并发错误 [152]。

5.5.2 Analysis with variable cost-effectiveness

如何平衡成本效益，长期以来一直是程序分析技术普遍面临的挑战。为应对这一挑战，DiaPro [40] 为动态依赖分析提供了多种成本效益权衡方案，以满足不同用户的需求。它将此前提出的多种依赖分析技术（例如，PI/EAS [8]、Diver [39]）整合起来；这些技术各自提供一种独特的成本效益权衡方案。

此外，D^2Abs [38] 旨在实现实际可扩展性，为分布式程序的动态依赖分析提供不同层次的成本效益权衡。为实现多种成本效益权衡，D^2Abs 向用户提供四个版本（例如，basicversion、msg+ version、csd+ version 和 scov+version），以便启用或禁用不同的分析步骤。

第六章

结论与未来工作

6.1 总结

在我的博士论文中，我介绍了三个相关项目（DistMeasure、 FlowDist和 Seads），以实现我的整体博士研究目标：提供可扩展且经济高效的数据流分析，以支持常见分布式软件的质量保证。为实现这一目标，我首先针对一种特定的安全应用（即信息流分析，本质上是一种定向数据流分析）解决了可扩展性、成本效益、适用性和可移植性方面的挑战，随后又针对更一般的问题（例如自适应、依赖分析）解决了这些挑战，以支持更广泛的应用，如软件维护、质量保证等。

首先，在项目 **DistMeasure** （ [93]的扩展）中，我定义了进程间通信（IPC）指标，用于衡量分布式系统的耦合性和内聚性，进而预测并了解其质量。

其次，在通过 IPC 指标了解分布式系统后，我开发并评估了一个基于细化的动态信息流分析框架 **FlowDist** [89]（包含两种替代设计）。该框架通过多阶段分析策略，能够克服适用性、可移植性和可扩展性方面的挑战。

最后，我开发了一个分布式、在线、连续且高性价比的动态依赖分析框架**Seads** [91]，用于持续运行的分布式系统。该框架通过自动调整分析配置，实现实际可扩展性和成本效益，并通过强化学习方法提供自调优性价比权衡。

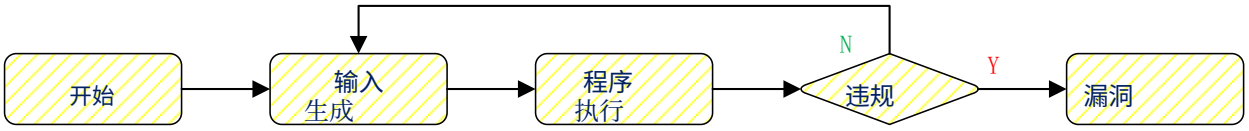
6.2 未来工作

我可以从许多方向拓展我的研究工作。下面列出三个主要方向，包括模糊测试技术、云系统分析和物联网（*IoT*）系统分析。

6.2.1 Fuzzing techniques

软件漏洞给许多工业领域（如航空、能源、金融等）造成了严重的安全问题和重大损失 [148]。然而，我的方法FlowDist 仍存在一个问题：它依赖测试输入来发现漏洞。如果某些漏洞在使用所有输入执行的过程中都未被覆盖，就无法发现这些漏洞。因此，我将开发技术来生成更多测试输入FlowDist （以及其他方法）。

图6.1：模糊测试工作流概述



为了发现并修复漏洞，研究人员提出了多种技术，其中模糊测试得到了长足发展，正逐渐成为最受欢迎的技术。模糊测试首先生成大量异常和正常的输入（即测试用例），并将其提供给目标程序，然后尝试发现并监控异常（例如，代码断言失败、崩溃、数据泄漏），[148], 如图6.1所示。因此，我将把模糊测试技术（例如 AFL（American Fuzzy Lop 的简称） [116]）应用于信息流分析，以检测常见分布式软件中的更多漏洞。

6.2.2 Analysis for cloud systems

云系统是分布式系统的应用程序 [226]。它们发展迅速，提供由亚马逊云服务（AWS）、谷歌云平台、IBM 云服务、Microsoft Azure、甲骨文云、Salesforce、SAP 等提供的平台即服务和软件即服务 [174] 。各种服务和应用程序已部署在这些云系统中，例如电子商务、存储、高性能计算等 [226]。用户只需向云服务提供商发送请求，然后等待其返回结果，无需了解云服务/平台如何运行 [150]。

然而，云系统和应用程序的发展仍面临一些挑战，例如安全性、性能、可用性等 [70]。分析技术/工具将着重应对这些挑战。例如，将采用各种方法来检测/防范云中的安全漏洞/攻击 [150]。我将探索能够分析云系统和/或部署于云系统上的技术，以应对这些挑战。

6.2.3 Analysis for Internet of Things (IoT) systems

物联网（IoT）利用嵌入式技术，构建了一个各种物理对象能够彼此交互与协作的环境 [72]。物联网系统将现实世界中的对象整合起来，帮助人们感知/控制相关事物。

过去几年，物联网的开发和应用层出不穷，例如互联汽车、智慧城市、智能电网、智能工厂等。许多物联网系统/平台也已开发并部署，例如 AWS IoT、IBM Watson IoT 平台、Microsoft Azure IoT Hub、OpenMTC、SiteWhere 等 [156]。

近年来，学术研究着重于应对物联网系统面临的安全挑战（例如，机密性、完整性、密钥管理、策略执行、隐私）。传统技术（例如，密码学）和新方法（例如，区块链、软件定义网络（SDN））已被用于解决当前的物联网安全挑战 [120]。除了物联网安全之外，物联网还有一些未来的研究方向，例如数据分析、性能建模和监控。物联网可以在云端转换、汇总、存储和分析数据，而云能提供诸多优势，例如高级安全、低成本资源、测量服务、资源整合、自助访问等 [76]。因此，我计划开发适用于物联网系统的分析方法。

参考文献

[1] Herv´e Abdi。肯德尔秩相关系数。测量与统计百科全书。*Sage*，加利福尼亚州千橡市，第508–510页，2007。

[2] Seiya Abe、Osamu Mizuno、Tohru Kikuno、Nahomi Kikuchi 和 Masayuki Hirayama。使用贝叶斯分类器估算项目成功率。载于第*28*届国际软件工程会议论文集，第600–603页，2006年。

[3] David H Ackley 和 Michael L Littman。强化学习中的泛化与扩展。载于神经信息处理系统进展，第550–557页，1990。

[4] Abdullah Mujawib Alashjaee、Salahaldeen Duraibi 和 Jia Song。动态污点分析工具：综述。《国际计算机科学与安全杂志》，13(6)：231，2019年。

[5]Apache。ZooKeeper/zookeeper-2569。issues.apache.org/jira/browse/ZOOKEEPER- 2569，2012年。

[6] Apache。Voldemort。github.com/voldemort，2015年。

[7] Apache。ZooKeeper。zookeeper.apache.org/，2015年。

[8] Taweesup Apiwattanapong、Alessandro Orso 和 Mary Jean Harrold。使用执行后序列进行高效且精确的动态影响分析。载于第*27*届国际软件工程会议论文集，第432–441页。ACM，2005年。

[9]Google Code Archive。V project-voldemort - 问题 352。code.google.com/archive/p/project-voldemort/issues/352，2011年。

[10]Google Code Archive。V project-voldemort - 问题 377。code.google.com/archive/p/project-voldemort/issues/377，2012年。

[11]Google Code Archive。V project-voldemort - 问题 378。code.google.com/archive/p/project-voldemort/issues/378，2012年。

[12]Google Code Archive。V project-voldemort - 问题 381。code.google.com/archive/p/project-voldemort/issues/381，2012年。

[13]Google Code Archive。V project-voldemort - 问题 387。code.google.com/archive/p/project-voldemort/issues/387，2013。

[14] Erik Arisholm、Lionel C Briand 和 Audun Foyen。《面向对象软件的动态耦合度量》。*IEEE*软件工程汇刊，30(8):491– 506，2004。

[15]Steven Arzt、Siegfried Rasthofer、Christian Fritz、Eric Bodden、Alexandre Bartel、Jacques Klein、Yves Le Traon、Damien Octeau 和 Patrick McDaniel。FlowDroid：面向 Android 应用的精确上下文、流、字段、对象敏感和生命周期感知污点分析。载于*ACM*编程语言设计与实现会议论文集，第 259–269 页，2014年。

[16] Mohammadreza Ashouri 和 Christoph Kreitz。《Scala 中的混合污点流分析》。载于*IEEE*计算智能研讨会系列论文集，第 657–663 页。IEEE，2019年。

[17] S Aswini 和 M Yazhini。使用

LOC 指标。在电力与先进计算技术创新（*i-PACT*）论文集，第 1–6 页。IEEE，2017年。

[18] Mona Attariyan 和 Jason Flinn。《通过动态信息流分析自动化配置故障排查》。载于*OSDI*，第 10 卷，第 1–14 页，2010年。

[19] Bryan Auslander、Stephen Lee-Urban、Chad Hogg 和 H´ector Munoz-Avila。《识别敌人：将强化学习与基于案例的推理策略选择相结合》。载于欧洲基于案例推理会议，第 59–73 页。Springer，2008年。

[20] Thomas H Austin 和 Cormac Flanagan。《高效的纯动态信息流分析》。载于*ACM SIGPLAN 安全编程语言与分析第四届研讨会论文集*，第 113–124 页。ACM，2009年。

[21] Thomas H Austin 和 Cormac Flanagan。《宽容的动态信息流分析》。载于*ACM SIGPLAN 安全编程语言与分析第五届研讨会论文集*，第 3 页。ACM，2010年。

[22]Gabriel Avner。《2017年十大安全漏洞》。resources.whitesourcesoftware.com/blog-whitesource/top-10-security-vulnerabilities-of-2017，2017年。

[23] Galip Aydin、Ibrahim Riza Hallac 和 Betul Karakus。《基于云计算与大数据技术的可扩展传感器数据存储与分析系统的架构与实现》。《传感器杂志》，2015年。

[24] Nikolaus Baer 和 Robert Zeidman。《通过代码行数变化衡量软件演化》。载于 *CATA* 会议论文集，第264–170页，2009年。

[25] Roberto Baldoni 和 Michel Raynal。《分布式计算基础：向量时钟系统实用指南》。*IEEE* 分布式系统在线，3(2):12，2002年。

[26] 班贝　格大学。Open Chord。sourceforge.net/projects/open-chord/，2015年。

[27] Subarno Banerjee、David Devecsery、Peter M Chen 和 Satish Narayanasamy。Iodine：使用无回滚乐观混合分析实现快速动态污点跟踪。载于 *IEEE*安全与隐私研讨会论文集，第490–504页。IEEE，2019年。

[28] Soubhagya Sankar Barpanda 和 Durga Prasad Mohapatra。分布式面向对象程序的动态切片。*IET*软件，5(5):425–433，2011年。

[29] EN Barron 和 H Ishii。用于最小化最大值的贝尔曼方程。非线性分析：理论、方法与应用，13(9):1067–1090，1989年。

[30] Jonathan Bell 和 Gail Kaiser。《Phosphor：揭示商品级 Java虚拟机中的动态数据流》。载于*ACM SIGPLAN* 面向对象编程、系统、语言与应用大会论文集，第 83–101 页，2014年。

[31] Jonathan Bell 和 Gail Kaiser。《使用 Phosphor 对 Java 进行动态污点跟踪》。载于*2015*年国际软件测试与分析研讨会论文集，第 409–413 页。ACM，2015年。

[32] Jacob Benesty、Jingdong Chen、Yiteng Huang 和 Israel Cohen。皮尔逊相关

系数。载于《语音处理中的噪声消除》论文集，第 1–4 页。Springer，2009年。

[33] James M Bieman 和 Linda M Ott。《测量功能内聚性》。*IEEE*软件工程汇刊，20(8):644–657，1994年。

[34] David Binkley、Sebastian Danicic、Tibor Gyim´othy、Mark Harman、Akos Kiss 和 Bogdan Korel。《动态程序切片的理论基础》。理论计算机科学，360(1–3):23–41，2006年。

[35] Leo Breiman。《装袋预测器》。机器学习，24(2):123–140，1996年。

[36] Derek Bruening 和 Qin Zhao。《使用 DynamoRIO 构建动态插桩工具》。2011年。

[37] Haipeng Cai。《用于有效动态影响预测的混合程序依赖近似》。*IEEE*软件工程汇刊，44(4):334–364，2018年。

[38] HaipengCai 和 Xiaoqin Fu。《D2ABS：用于分布式程序动态依赖分析的框架》。*IEEE*软件工程汇刊，2021年。

[39] Haipeng Cai 和 Raul Santelices。Diver：使用基于依赖的轨迹剪枝进行精确的动态影响分析。载于国际自动化软件工程会议论文集，第 343–348 页，2014年。

[40] Haipeng Cai、Raul Santelices 和 Douglas Thain。DiaPro：统一动态影响分析，以提升成本效益并实现可变成本效益。*ACM*软件工程与方法学汇刊，25(2):1–50，2016年。

[41] Haipeng Cai 和 Douglas Thain。DistIA：一种针对分布式程序的高成本效益动态影响分析。发表于第*31*届 *IEEE/ACM* 国际自动化软件工程会议论文集，第344–355页，2016年。

[42] Andrew Phelps Cassidy和Frank A Deviney。利用在线随机森林计算数据流在概念漂移下的特征重要性。载于*2014 IEEE* 国际大数据会议（*Big Data*），第23–28页。IEEE，2014年。

[43] Jingde Cheng。并行与分布式程序的依赖分析及其应用。载于并行与分布式计算进展论文集，第370–377页，1997年。

[44] Winnie Cheng、Qin Zhao、Bei Yu和Scott Hiroshige。TaintTrace：通过动态二进制重写实现高效流追踪。载于*IEEE* 计算机与通信研讨会论文集，第749–754页。IEEE，2006年。

[45] Jitender Kumar Chhabra和Varun Gupta。动态软件度量综述。《计算机科学与技术学报》，25(5)：1016–1029，2010年。

[46]Georgios Chinis、Polyvios Pratikakis、Sotiris Ioannidis和Elias Athanasopoulos。遗留 Web 应用程序的实用信息流。载于*ACM SIGPLAN* 面向对象编程、系统、语言与应用会议论文集，第17–28页，2013。

[47] Jim Chow、Ben Pfaff、Tal Garfinkel、Kevin Christopher 和 Mendel Rosenblum。通过全系统模拟理解数据生命周期。载于*USENIX* 安全研讨会，第 321–336 页，2004。

[48] Binildas Christudas。Derby。载于实用微服务架构模式，第 869–876 页。Springer，2019年。

[49] James Clause、Wanchun Li 和 Alessandro Orso。Dytan：通用动态污点分析框架。载于国际软件测试与分析研讨会论文集，第 196–206 页。ACM，2007。

[50] Marshall Copeland、Julian Soh、Anthony Puca、Mike Manning 和 David Gollob。Microsoft Azure。美国纽约州纽约市：*Apress*，2015年。

[51] James C Corbett、Matthew B Dwyer、John Hatcliff、Shawn Laubach、Corina S Pasare- anu、Hongjun Zheng 等。Bandera：从 Java 源代码中提取有限状态模型。载于*IEEE* 国际软件工程会议论文集，第 439–448 页。IEEE，2000。

[52] George Coulouris、Jean Dollimore、Tim Kindberg 和 Gordon Blair。分布式系统：概念与设计。Addison-Wesley 出版公司，第 5 版，2011年。

[53] Jedidiah R Crandall 和 Frederic T Chong。《Minos：与内存模型正交的控制数据攻击防护》。载于会议论文集，第 221–232 页。IEEE，2004。

[54] CVE。CVE-2014-0085。www.cvedetails.com/cve/CVE-2014-0085/，2014年。

[55]CVE。CVE-2014-0193。cve.mitre.org/cgi-bin/cvename.cgi?name= CVE-2014-0193，2015年。

[56]CVE。CVE-2015-2156。cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2156，2015年。

[57]CVE。CVE-2015-3254。cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3254，2015年。

[58]CVE。CVE-2005-3280。cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-3280，2016年。

[59]CVE。CVE-2016-4970。cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4970，2016年。

[60]CVE。CVE-2019-17572。cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17572，2016年。

[61]CVE。CVE-2017-5637。cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5637，2017年。

[62] CVE。通用漏洞披露（CVE）。cve.mitre.org/，2018年。

[63] Frank Dabek、Emma Brunskill、M Frans Kaashoek、David Karger、Robert Morris、Ion Stoica 和 Hari Balakrishnan。《使用 Chord（一种分布式查找服务）构建点对点系统》。载于操作系统热点主题研讨会论文集，第 81–86 页。IEEE，2001 年。

[64]国家漏洞数据库。漏洞详情：CVE-2014-3488。nvd.nist.gov/vuln/detail/CVE-2014-3488，2014年。

[65]国家漏洞数据库。漏洞详情：CVE-2018-8012。”
https://nvd.nist.gov/vuln/detail/CVE-2018-8012”，2018年。

[66]国家漏洞数据库。漏洞详情：CVE-2019-0201。
nvd.nist.gov/vuln/detail/CVE-2019-0201，2019年。

[67] 国家漏洞数据库。国家漏洞数据库。nvd.nist.gov/，2020年。

[68] Harvey Deitel、Abbey Deitel 等。《互联网与万维网编程指南》。
PrenticeHallPress，2011年。

[69] Amr F Desouky 和 Letha H Etzkorn。《面向对象内聚度量：运行时行为的定性实证分析》。载于*ACM*东南地区会议论文集，第1–6页，2014年。

[70] Tharam Dillon、Chen Wu 和 Elizabeth Chang。《云计算：问题与挑战》。载于*2010*年第*24*届*IEEE*国际会议高级信息网络与应用，第27–33页。IEEE，2010年。

[71] I. Capuzzo Dolcetta 和 Hitoshi Ishii。《确定性控制理论中贝尔曼方程的近似解》。《应用数学与优化》，11(1):161–181，1984年。

[72] Mohsen Dorodchi、Maryam Abedi 和 Bojan Cukic。《分布式系统和物联网的基于信任的开发框架》。载于第*40*届年度计算机软件与应用会议（*COMPSAC*）论文集，第2卷，第437–442页。IEEE，2016年。

[73] L´ucia MA Drummond 和 Valmir C Barbosa。关于降低矩阵时钟的复杂度。*Parallel Computing*，29(7)：895–905，2003。

[74] Evelyn Duesterwald、Rajiv Gupta 和 M Soffa。面向分布式程序的分布式切片与部分重执行。载于并行计算的语言与编译器，第 497–511 页。Springer，1993。

[75] BrunoDufour、Karel Driesen、LaurieHendren 和 Clark Verbrugge。Java动态度量。载于第 *18* 届 *ACM SIGPLAN* 面向面向对象编程、系统、语言和应用的年度会议论文集，第 149–168 页，2003。

[76] Said El Kafhali 和 Khaled Salah。物联网启用的医疗监测系统的性能建模与分析。*IET Networks*，8(1):48–58，2019年。

[77] William Enck、Peter Gilbert、Seungyeop Han、Vasant Tendulkar、Byung-Gon Chun、Landon P Cox、Jaeyeon Jung、Patrick McDaniel 和 Anmol N Sheth。TaintDroid：面向智能手机实时隐私监控的信息流跟踪系统。*ACM*计算机系统汇刊 *(TOCS)*，32(2):5，2014年。

[78] Eyal Even-Dar 和 Yishay Mansour。Q学习的学习率。《机器学习研究杂志》，5(12月)：1–25，2003。

[79]Stack Exchange。术语——入口点、数据源、汇。security.stackexchange.com/questions/123854/terminology-entry-point-data-source- sink，2018年。

[80]Norman Fenton 和 James Bieman。软件度量：严谨且实用的方法。CRC出版社，2014年。

[81] Colin J Fidge。《保持偏序的消息传递系统中的时间戳》。1987年。

[82] William Frakes 和 Carol Terry。《软件复用：度量与模型》。*ACM* 计算调查（*CSUR*），28(2):415–435，1996年。

[83] Wei Fu 和 Tim Menzies。重新审视用于缺陷预测的无监督学习。载于软件工程基础联合会议论文集，第72–83页，2017年。

[84] Xiaoqin Fu。关于分布式系统的可扩展动态污点分析。载于第*27*届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集，第1247–1249页，2019年。

[85] Xiaoqin Fu。迈向分布式系统信息流安全的可扩展防护。载于第*28*届 *ACM SIGSOFT*国际软件测试与分析研讨会论文集，第438–442页，2019年。

[86] Xiaoqin Fu 和 Haipeng Cai。面向分布式系统的动态污点分析器。载于*2019*年第*27*届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集，第1115–1119页。ACM，2019年。

[87]Xiaoqin Fu 和 Haipeng Cai。FlowDist制品包。github.com/Anonymity010/FLOWDIST 资料，2020年。

[88] Xiaoqin Fu 和 Haipeng Cai。将应用层动态污点分析扩展至企业级分布式系统。载于*ACM/IEEE 第42*届软件工程国际会议论文集：配套论文集，第270–271页，2020年。

[89] Xiaoqin Fu 和 Haipeng Cai。FlowDist：面向分布式软件系统的基于多阶段细化的动态信息流分析。载于《第*30*届*USENIX*安全研讨会论文集》（*USENIX Security 21*），第 2093–2110 页，2021年。

[90] Xiaoqin Fu、Haipeng Cai 和 Li Li。Dads：在预算约束下对持续运行的分布式程序进行动态切片。载于《第*28*届*ACM*欧洲软件工程会议与软件工程基础研讨会联合会议论文集》，第 1566–1570 页，2020年。

[91] Xiaoqin Fu、Haipeng Cai、Wen Li 和 Li Li。Seads：通过强化学习实现可扩展且具成本效益的分布式系统动态依赖分析。《*ACM*软件工程与方法学汇刊》（*TOSEM*），30(1):1–45，2020年。

[92] Xiaoqin Fu、Boxiang Lin 和 Haipeng Cai。DistFax：用于测量进程间通信和分布式系统质量的工具包。载于《*ACM/IEEE*国际软件工程会议论文集：伴随论文集》，2022。

[93] Fu, Xiaoqin 和 Cai, Haipeng。《分布式

系统。载于*IEEE/ACM* 国际程序理解会议论文集，第 323–334 页。IEEE，2019年。

[94] Takayasu Fuchida、Kathy Thi Aung 和 Atsushi Sakuragi。《考虑负奖励的Q学习研究》。《人工生命与机器人》，15(3):351–354，2010年。

[95] Daniel Galin。软件质量保证：从理论到实践。培生印度教育，2004。

[96] Rani Geetika 和 Paramvir Singh。《静态与动态耦合度量的实证研究》。《ACM SIGSOFT 软件工程简报》，39(1):1–8，2014年。

[97] Jean-Fran¸cois G´elinas、Mourad Badri 和 Linda Badri。《方面的内聚度量》。《面向对象技术杂志》，5(7):75–95，2006年。

[98] Dennis Giffhorn 和 Christian Hammer。《并发程序的精确切片》。《Automated Software Engineering》，16(2):197–234，2009年。

[99] 公司[美国]GitHub。voldemort/voldemort。github.com/voldemort/voldemort/issues/101，2012年。

[100] 公司[美国]GitHub。 netty/netty。github.com/netty/netty/issues/8869，2019年。[101] 公司[美国] GitHub。netty/netty。 github.com/netty/netty/issues/9112，2019年。[102] 公司[美国] GitHub。 netty/netty。 github.com/netty/netty/issues/9229，2019年。[103] 公司[美国] GitHub。 netty/netty。github.com/netty/netty/issues/9243，2019年。[104] 公司[美国] GitHub。netty/netty。 github.com/netty/netty/issues/9291，2019年。

[105] 公司[美国] GitHub。 netty/netty。
github.com/netty/netty/issues/9362，2019年。[106] 公司[美国] GitHub。
xSocket。 sourceforge.net/p/xsocket/bugs/21/，2019年。

[107] Mehran Goli、Muhammad Hassan、Daniel Große 和 Rolf Drechsler。使用动态信息流跟踪对基于VP的片上系统进行安全性验证。*it-InformationTechnology*，61(1):45–58，2019年。

[108] GoogleCode。MultiChat。code.google.com/p/multithread-chat-server/，2015年。

[109] Michael I Gordon、Deokhwan Kim、Jeff HPerkins、Limei Gilham、Nguyen Nguyen 和 Martin C Rinard。DroidSafe 中 Android应用程序的信息流分析。载于网络与分布式系统安全研讨会论文集，第110页，2015年。

[110] Anjana Gosain 和 Ganga Sharma。使用面向对象动态复杂度度量预测软件可维护性。载于信息技术与计算机通信智能趋势国际会议，第218–230页。Springer，2016年。

[111] Anjana Gosain 和 Ganga Sharma。用于软件可理解性的面向对象动态复杂度度量。系统与软件工程创新，13(2-3):177–190，2017年。

[112] Diganta Goswami 和 Rajib Mall。《并发程序的动态切片》。载于高性能计算国际会议，第 15–26 页。Springer，2000 年。

[113] Claire Le Goues。《讲义：函数式语言的控制流分析》

与面向对象调用图。 http://www.cs.cmu.edu/~clegoues/courses/15-819O-16sp/notes/notes08-functional-cfa.pdf。

[114] J¨urgen Graf、Martin Hecker 和 Martin Mohr。使用 JOANA 在 Java 程序中进行信息流控制：实用指南。载于编程语言工作会议论文集，《信息学讲义》215，第 123–138 页。Springer Berlin / Heidelberg，2013年2月。

[115] Varun Gupta 和 Jitender Kumar Chhabra。面向对象软件的动态内聚度量。系统架构杂志，57(4):452–462，2011年。

[116] Peter Gutmann。使用 AFL 对代码进行模糊测试。*login:*，41(2)，2016年。

[117]Christian Hammer 和 Gregor Snelting。基于程序依赖图的流敏感、上下文敏感和对象敏感信息流控制。国际信息安全杂志，8(6):399–422，2009年。

[118] M Hammerton、J Trevathan、T Myers 和 W Read。异构传感器网络软件连接方法的评估。国际电气与计算机工程杂志，7(4):550–557，2014年。

[119] Novita Hanafiah、Jaka Hartanto、Yulyani Arifin、Hengky Frans 和 Winardi Cris- tian. 基于多层架构模型的基于Web的中国象棋。*ProcediaComputer Science*，59:14–18，2015年。

[120] Wan Haslina Hassan 等人。物联网（IoT）安全性的当前研究：一项综述。计算机网络，148:283–294，2019年。

[121] Christophe Hauser、Fr´ed´eric Tronel、Colin Fidge 和 Ludovic M´e。基于污点标记方法的分布式系统入侵检测。载于*IEEE*国际通信会议（*ICC*）论文集，第 1962–1967 页。IEEE，2013。

[122] Susan Horwitz 和 Thomas Reps。《程序依赖图在软件工程中的应用》。发表于*IEEE/ACM*国际软件工程会议论文集，第 392–411 页，1992年。

[123]Susan Horwitz、Thomas Reps 和 David Binkley。《使用依赖图进行过程间切片》。*ACM*《编程语言与系统汇刊》（*TOPLAS*），12(1):26–60，1990年。

[124] Chi-fu Huang、Henri Pages 等。《无限期限下的最优消费与投资组合策略：存在性与收敛性》。应用概率年刊，2(1):36–64，1992年。

[125]Patrick Hunt、Mahadev Konar、Flavio Paiva Junqueira 和 Benjamin Reed。《ZooKeeper：面向互联网规模系统的无等待协调》。发表于*USENIX*年度技术会议论文集，第 8 卷，第 9 页，2010年。

[126] ISO/IEC。ISO/IEC 25010:2011 系统与软件工程——系统与软件质量要求和评价（SQuaRE）——系统与软件质量模型。www.iso.org/standard/35733.html，2011年。

[127] Daniel Jackson 和 Martin Rinard。《软件分析：路线图》。载于软件工程的未来会议论文集，第 133–145 页，2000 年。

[128] Jianyu Jiang、Shixiong Zhao、Danish Alsayed、Yuexuan Wang、Heming Cui、Feng Liang 和 Zhaoquan Gu。《Kakute：一种面向大数据安全的精确、统一信息流分析系统》。载于年度计算机安全应用会议论文集，第 79–90 页，2017年。

[129] Wuxia Jin、Ting Liu、Yu Qu、Jianlei Chi、Di Cui 和 Qinghua Zheng。《分布式系统动态内聚性度量》。载于第 *1* 届并发程序规范、理解、测试与调试国际研讨会论文集，第 20–26 页。ACM，2016年。

[130] Wuxia Jin、Ting Liu、Yu Qu、Qinghua Zheng、Di Cui 和 Jianlei Chi。《分布式软件的动态结构度量》。《软件质量期刊》，26(3)：119–1145，2018年。

[131] Mariam Kamkar 和 Patrik Krajina。《分布式程序的动态切片》。载于*IEEE* 国际软件维护会议论文集，第 222–229 页。IEEE，1995年。

[132] Stephen HKan。《软件质量工程中的度量与模型》。Addison-Wesley Longman Publishing Co., Inc.，2002年。

[133] Min Gyung Kang、Stephen McCamant、Pongsin Poosankam 和 Dawn Song。DTA++：采用目标控制流传播的动态污点分析。载于*NDSS*，2011年。

[134]Rezwana Karim、Frank Tip、Alena Sochurkova 和 Koushik Sen。《面向 JavaScript 的平台无关动态污点分析》。*IEEE*软件工程汇刊，2018年。

[135] Anureet Kaur。《代码行度量工具比较分析》。*Int J Sci Res SciEng Technol (ijsrset. com)*，2016年。

[136] Vasileios P Kemerlis、Georgios Portokalidis、Kangkook Jee 和 Angelos D Keromytis。《libdft：面向通用系统的实用动态数据流跟踪》。载于第 *8* 届 *ACM SIGPLAN/SIGOPS* 虚拟执行环境会议论文集，第 121–132 页，2012年。

[137] Taghi M Khoshgoftaar 和 Edward B Allen。《软件质量的逻辑回归建模》。*International Journal of Reliability, Quality and Safety Engineering*，6(04)：303–317，1999年。

[138] Taghi M Khoshgoftaar 和 Robert M Szabo。《使用神经网络预测测试期间的软件故障》。*IEEE Transactions on Reliability*，45(3)：456–462，1996年。

[139] BogdanKorel 和 RogerFerguson。《分布式程序的动态切片》。*AppliedMathematics and Computer Science*，2(2)：199–215，1992年。

[140] Bogdan Korel 和 Janusz Laski。《动态程序切片》。*Information processingletters*，29(3):155–163，1988。

[141] Jacob Kreindl、Daniele Bonetta 和 Hanspeter M¨ossenb¨ock。《迈向高效的多语言动态污点分析》。载于第 *16* 届 *ACM SIGPLAN* 托管编程语言与运行时国际会议论文集，第 85–94 页，2019年。

[142] Jens Krinke。《并发程序的上下文敏感切片》。载于论文集：

第九届欧洲软件工程会议与第十一届 *ACM SIGSOFT* 软件工程基础国际研讨会联合举行，第178–187页，2003年。

[143] Patrick Lam、Eric Bodden、Ondrej Lhot´ak 和 Laurie Hendren。Soot——Java 字节码优化框架。载于*Cetus* 用户与编译器基础设施研讨会，2011年。

[144] Leslie Lamport。分布式系统中的时间、时钟与事件排序。载于并发：*Leslie Lamport* 的著作，第 179–196 页。2019年。

[145] Chris Lattner 和 Vikram Adve。LLVM：用于长期程序分析与转换的编译框架。载于代码生成与优化国际研讨会论文集，第 75–86 页。IEEE，2004。

[146] Frank L Lewis 和 Kyriakos G Vamvoudakis。面向部分可观测动态过程的强化学习：利用测量输出数据进行自适应动态规划。*IEEE* 系统、人和控制论汇刊，*B* 部分（控制论），41(1):14–25，2010年。

[147] Ondˇrej Lhot´ak 和 Laurie Hendren。《使用基于 BDD 的实现评估上下文敏感的指向分析的优势》。*ACM* 软件工程与方法学汇刊（*TOSEM*），18(1)：1–53，2008年。

[148] Jun Li、Bodong Zhao 和 Chao Zhang。《模糊测试：综述》。网络安全，1(1)：1–13，2018年。

[149] Li Li、Alexandre Bartel、Tegawend´e F Bissyand´e、Jacques Klein、Yves Le Traon、Steven Arzt、Siegfried Rasthofer、Eric Bodden、Damien Octeau 和 Patrick McDaniel。Iccta：

检测 Android 应用中的组件间隐私泄露。载于第*37*届国际软件工程会议论文集（第*1*卷），第280–291页。IEEE 出版社，2015年。

[150] Zhe Li、Weiqing Sun 和 Lingfeng Wang。《基于神经网络的云平台分布式入侵检测系统》。载于*2012 IEEE* 第二届云计算与智能系统国际会议，第 1 卷，第 75–79 页。IEEE，2012年。

[151]Michael L Littman 和 Csaba Szepesv´ari。《广义强化学习模型：收敛性与应用》。载于*ICML*，第 96 卷，第 310–318 页，1996年。

[152] Haopeng Liu、Guangpu Li、Jeffrey F Lukman、Jiaxin Li、Shan Lu、Haryadi S Gunawi 和 Chen Tian。《DCatch：自动检测云系统中的分布式并发错误》。《style id='2'>ACM SIGARCH 计算机架构新闻，45(1)：677–691，2017年。

[153] Benjamin Livshits、Manu Sridharan、Yannis Smaragdakis、Ondˇrej Lhot´ak、J Nelson Amaral、Bor-Yuh Evan Chang、Samuel Z Guyer、Uday P Khedker、Anders Møller 和 Dimitrios Vardoulakis。《为可靠性辩护：一份宣言》。《style id='3'>《ACM 通讯》，58(2)：44–46，2015年。

[154] Florian D Loch、Martin Johns、Martin Hecker、Martin Mohr 和 Gregor Snelting。《Java EE 的混合污点分析》。载于年度 *ACM* 应用计算研讨会论文集，第 1716–1725 页，2020年。

[155]Chi-Keung Luk、Robert Cohn、Robert Muth、Harish Patil、Artur Klauser、Geoff Lowney、Steven Wallace、Vijay Janapa Reddi 和 Kim Hazelwood。Pin：构建

定制化的动态插桩程序分析工具。*ACM SIGPLAN* 通讯，40(6):190–200，2005。

[156] Somayya Madakam、Vihar Lake、Vihar Lake、Vihar Lake等。物联网（IoT）：文献综述。计算机与通信期刊，3(05):164，2015年。

[157] IliasMaglogiannis、Constantinos Delakouridis和Leonidas Kazatzopoulos。通过点对点分发电子健康记录，实现互联网上的协作式医疗诊断。医学系统期刊，30(2):107–116，2006年。

[158] Pratyusa K Manadhata和Jeannette M Wing。攻击面度量。*IEEE*软件工程汇刊，37(3):371–386，2010年。

[159] Wes Masri和Andy Podgurski。基于应用程序的异常入侵检测与动态信息流分析。计算机与安全，27(5-6):176–187，2008年。

[160] Wes Masri和Andy Podgurski。动态信息流分析的算法与工具支持。信息与软件技术，51(2):385–404，2009年。

[161] Wes Masri、Andy Podgurski 和 David Leon。《检测和调试不安全信息流》。载于《国际软件可靠性工程研讨会论文集》，第 198–209 页。IEEE，2004。

[162] Norman Maurer 和 Marvin Wolfthal。《*Netty* 实战》。Manning Publications，纽约，2016年。

[163] Thomas J McCabe。复杂度度量。《*IEEE* 软件工程学报》，1(4)：308–320，1976年。

[164]微软。Azure 文档。docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier，2022。

[165] Matthew Might、Yannis Smaragdakis 和 David Van Horn。《解决并利用 k-CFA 悖论：阐明函数式与面向对象程序分析》。载于第 *31* 届 *ACM SIGPLAN* 编程语言设计与实现会议论文集，第 305–315 页，2010年。

[166]Jiang Ming、Dinghao Wu、Gaoyao Xiao、Jun Wang 和 Peng Liu。TaintPipe：流水线符号污点分析。载于*USENIX* 安全研讨会，第 65–80 页，2015年。

[167] Aine Mitchell 和 James F Power。运行时内聚度量：一项实证研究。第 532–537 页。2004。

[168] Durga P Mohapatra、Rajeev Kumar、RajibMall、DS Kumar 和 Mayank Bhasin。Java 程序的分布式动态切片。《系统与软件杂志》，79(12)：1661–1678，2006年。

[169]Gero M¨uhl、Ludger Fiege 和 Peter Pietzuch。《分布式事件系统》。Springer Science & Business Media，2006年。

[170] Sape Mullender 等。《分布式系统》，第 12 卷。美国 ACM 出版社，1993 年。

[171] Andrew C Myers。Jflow：实用的近似静态信息流控制。载于第 *26* 届 *ACM SIGPLAN-SIGACT* 编程语言原理研讨会论文集，第 228–241 页。ACM，1999年。

[172]Glenford J Myers 等。《通过组合式设计实现可靠软件》。Petro- celli/Charter，1975年。

[173] Mangala Gowri Nanda 和 S. Ramesh。《多线程程序的过程间切片及其在 Java 中的应用》。《ACM《编程语言与系统汇刊》》，28(6):1088–1144，2006年。

[174] Ali Bou Nassif、Manar Abu Talib、Qassim Nasir、Halah Albadani 和 Fatima Mohamad Dakalbab。《云安全中的机器学习：系统综述》。《IEEE Access》，9:20717–20735，2021年。

[175] Nicholas Nethercote 和 Julian Seward。《Valgrind：重量级动态二进制插桩框架》。《ACM Sigplan notices》，42(6):89–100，2007。

[176] James Newsome 和 Dawn Xiaodong Song。《针对商用软件漏洞利用的自动检测、分析与特征生成的动态污点分析》。载于*NDSS*。Citeseer，2005年。

[177] Achim Nierbeck、Jamie Goodyear、Johan Edstrom 和 Heath Kesler。《 Apache Karaf 实战手册》。Packt Publishing Ltd，2014年。

[178] Vasilis Pappas、Vasileios P Kemerlis、Angeliki Zavou、Michalis Polychronakis 和 Angelos D Keromytis。《 CloudFence：作为云服务的数据流跟踪》。载于 国际

入侵检测最新进展国际研讨会，第 411–431 页。Springer，2013 年。

[179] Fabian Pedregosa、Ga¨el Varoquaux、Alexandre Gramfort、Vincent Michel、Bertrand Thirion、Olivier Grisel、Mathieu Blondel、Peter Prettenhofer、Ron Weiss、Vincent Dubourg 等。《Scikit-learn：Python 中的机器学习》。《机器学习研究杂志》，12：2825–2830，2011年。

[180] EduardoPelegri-Llopart、YutakaYoshida 和 Alexis Moussine-Pouchkine。《GlassFish 社区：交付 Java EE 应用服务器》，2007。

[181] Jing Peng 和 Ronald J Williams。《增量式多步 Q 学习》。载于《机器学习论文集》*1994*，第 226–232 页。爱思唯尔，1994。

[182] Mikhail Perepletchikov 和 Caspar Ryan。《一项用于评估耦合对面向服务的软件可维护性影响的对照实验》。*IEEE*软件工程汇刊，37(4)：449–465，2011年。

[183] Duc Truong Pham、Stefan S Dimov 和 Chi D Nguyen。K均值聚类中 K 值的选择。《机械工程师学会会刊》*C*辑：《机械工程科学杂志》，219(1):103–119，2005。

[184] Andy Podgurski 和 Lori A. Clarke。程序依赖关系的形式模型及其对软件测试、调试和维护的影响。*IEEE*软件工程汇刊，16(9):965–979，1990。

[185] Martin L Puterman。 马尔可夫决策过程：离散随机动态规划。约翰·威立父子出版公司，2014年。

[186] Feng Qin、Cheng Wang、Zhenmin Li、Ho-seop Kim、Yuanyuan Zhou 和 Youfeng Wu。LIFT：一种用于检测安全攻击的低开销实用信息流跟踪系统。载于会议论文集，第135–148页。IEEE，2006年。

[187] Krzysztof Rakowski。《学习 Apache Thrift》。Packt Publishing Ltd，2015年。

[188] ZA Rana、S Shamail 和 MM Awais。《基于度量的软件质量预测技术综述》。技术报告，技术报告，拉合尔管理科学大学，2007。

[189] Venkatesh Prasad Ranganath 和 John Hatcliff。使用 Indus 和 Kaveri 对并发 Java 程序进行切片。《软件工具与技术转移国际期刊》，9(5-6):489–504，2007。

[190] V Rao 和 K Hande。《安卓恶意软件检测的静态、动态和混合分析技术比较研究》。《国际工程发展与研究期刊》，5(2):1433–1436，2017。

[191] Ekbal A Rashid、Srikanta B Patnaik 和 VandanaC Bhattacherjee。《机器学习与软件质量预测：作为专家系统》。《国际信息工程与电子商务期刊》，6(2):9，2014年。

[192] Tamar Richner 和 St´ephane Ducasse。从静态和动态信息中恢复面向对象应用程序的高层视图。载于*IEEE* 国际软件维护会议论文集，第 13–22 页。IEEE，1999年。

[193] Apache RocketMQ。Apache RocketMQ。rocketmq.apache.org/，2020年。

[194] Alejandro Russo 和 Andrei Sabelfeld。《动态与静态流敏感安全分析》。载于第*23*届 *IEEE*国际计算机安全基础研讨会（*CSF*）论文集，第186–199页。IEEE，2010年。

[195] Barbara Ryder。《面向对象编程语言的引用分析精确率维度》。载于编译器构造，第126–137页，2003。

[196] Andrei Sabelfeld 和 Andrew C Myers。《基于语言的信息流安全》。*IEEE*通信领域精选期刊，21(1):5–19，2003。

[197] Patrick Schober、Christa Boer 和 Lothar A Schwarte。《相关系数：恰当使用与解读》。《麻醉与镇痛》，126(5):1763–1768，2018年。

[198] Edward J Schwartz、Thanassis Avgerinos 和 David Brumley。《关于动态污点分析与前向符号执行，你想知道的一切（但可能不敢问）》。载于*IEEE*安全与隐私研讨会（*SP*）论文集，第317–331页。IEEE，2010年。

[199] Philip Sedgwick。《斯皮尔曼秩相关系数》。英国医学杂志，349:g7327，2014年。

[200] Omar Sefraoui、Mohammed Aissaoui 和 Mohsine Eleuldj。《OpenStack：迈向云计算的开源解决方案》。《国际计算机应用杂志》，55(3):38–42，2012年。

[201] Koushik Sen、Swaroop Kalasapur、Tasneem Brutch 和 Simon Gibbs。Jalangi：一种面向 JavaScript 的选择性记录-回放与动态分析框架。载于软件工程基础联合会议论文集，第 488–498 页，2013。

[202]Venkatesh Gauri Shankar、Gaurav Somani、Manoj Singh Gaur、Vijay Laxmi 和 Mauro Conti。AndroTaint：一种采用动态污点分析的高效安卓恶意软件检测框架。载于*ISEA Asia* 安全与隐私论文集，第 1–13 页。IEEE，2017年。

[203] Lwin Khin Shar、Hee Beng Kuan Tan 和 Lionel C Briand。使用混合程序分析挖掘 SQL 注入和跨站脚本攻击漏洞。载于第*35*届国际软件工程会议（*ICSE*）论文集，第642–651页。IEEE，2013。

[204] Mohsen Sharifi、Ehsan Mousavi Khaneghah、Morteza Kashyian 和 Seyedeh Leili Mir-taheri。支持异构分布式系统编程的平台无关分布式IPC机制。《超级计算杂志》，59(1):548–567，2012年。

[205] Micha Sharir、Amir Pnueli 等。过程间数据流分析的两种方法。纽约大学。库朗数学科学研究所……，1978年。

[206] Mariana Sharp 和 Atanas Rountev。基于远程方法调用的 Java 软件中对象引用的静态分析。《IEEE软件工程汇刊》，32(9):664–681，2006年。

[207] Dongdong She、Yizheng Chen、Abhishek Shah、Baishakhi Ray 和 Suman Jana。《Neu-taint：基于神经网络的高效动态污点分析》。载于*IEEE*安全与隐私研讨会论文集，2020年。

[208] Kavita She oran、Pradeep Tomar 和 Rajesh Mishra。《软件质量预测》

基于 HCS 的高级神经网络模型。*Procedia ComputerScience*，92:418–424，2016年。

[209] Olin Shivers。Scheme 中的控制流分析。载于 *ACM SIGPLAN 1988*年编程语言设计与实现会议论文集，第164–174页，1988年。

[210] Paritosh Shroff、Scott Smith 和 Mark Thober。通过动态依赖监控保障信息流安全。载于*IEEE*计算机安全基础研讨会论文集，第203–217页。IEEE，2007年。

[211] David Simansky。将 JBoss Undertow HTTP服务器与 Apache Camel 项目集成，2014年。

[212] Vincent Simonet 和 Inria Rocquencourt。简述 Flow Caml。载于首届 *APPSEM-II*研讨会论文集，第152–165页，2003年。

[213] SourceForge。RainingSockets。freshmeat.sourceforge.net/projects/jniosocket/，2004年。[214] SourceForge。ADEN。sourceforge.net/projects/adenproject/，2013年。

[215]SourceForge。NioEcho。rox-xmlrpc.sourceforge.net/niotut/index.html#代码，2015年。

[216] SourceForge。HyperSQL。hsqldb.org/，2020年。

[217] Johannes Sp¨ath、Karim Ali 和 Eric Bodden。《使用同步下推系统进行上下文、流和字段敏感的数据流分析》。*ACM* 编程语言会议论文集，3(POPL):1–29，2019年。

[218] G Edward Suh、Jae W Lee、David Zhang 和 Srinivas Devadas。《通过动态信息流跟踪实现安全程序执行》。载于*ACM* 编程语言与操作系统架构支持国际会议论文集，第 85–96 页，2004。

[219] Roshan Sumbaly、JayKreps、LeiGao、Alex Feinberg、ChinmaySoman 和 SamShah。使用 Project Voldemort 提供大规模批量计算数据服务。载于*FAST*，第 12 卷，第 18–18 页，2012年。

[220] Chengzheng Sun 和 Wentong Cai。通过实时群组编辑器中的压缩向量时钟捕获因果关系。载于第 *16* 届国际并行与分布式处理研讨会论文集，第 8–pp 页。IEEE，2002 年。

[221] Yuqiong Sun、Giuseppe Petracca、Xinyang Ge 和 Trent Jaeger。Pileus：保护用户资源免受易受攻击的云服务侵害。载于计算机安全应用年会论文集，第 52–64 页，2016年。

[222] Amjed Tahir 和 Stephen G MacDonell。关于动态度量与软件质量的系统映射研究。载于*IEEE* 国际软件维护会议论文集，第 326–335 页。IEEE，2012年。

[223]Clang 团队。DataFlowSanitizer。`https://clang.llvm.org/docs/DataFlowSanitizer.html`，2019年。

[224] Mie Mie Thet Thwin 和 Tong-Seng Quah。使用面向对象设计度量指标应用神经网络预测软件开发缺陷。载于《论文集

第九届神经信息处理国际会议，第 5 卷，第 2312–2316 页。IEEE，2002 年。

[225] Cheng Wang、Shiliang Hu、Ho-seop Kim、Sreekumar R Nair、Mauricio Breternitz、Zhiwei Ying 和 Youfeng Wu。StarDBT：一种高效的多平台动态二进制翻译系统。载于亚太计算机系统结构进展会议论文集，第 4–15 页。Springer，2007。

[226] Hongyi Wang、Qingfeng Jing、Rishan Chen、Bingsheng He、Zhengping Qian 和 Lidong Zhou。分布式系统与经济学相遇：云定价。载于第二届 *USENIX*云计算热点研讨会（*HotCloud 10*），2010年。

[227] Qi Wang、Bo Yu 和 Jie Zhu。基于神经网络的软件质量预测模型中规则的提取。载于*IEEE*人工智能工具国际会议论文集，第 191–195 页。IEEE，2004。

[228] Xinran Wang、Yoon-Chan Jhi、Sencun Zhu 和 Peng Liu。STILL：通过静态污点分析与初始化分析检测漏洞利用代码。载于年度计算机安全应用会议论文集，第 289–298 页。IEEE，2008。

[229] Christopher JCH Watkins 和 Peter Dayan。Q学习。机器学习，8(3-4):279– 292，1992年。

[230] Pawe-l Wawrzynski 和 Andrzej Pacut。在连续环境中的无模型离策略强化学习。发表于*2004 IEEE*神经网络国际联合会议（*IEEE*目录号：*04CH37541*），第2卷，第1091–1096页。IEEE，2004年。

[231] Fengguo Wei、Sankardas Roy 和 Xinming Ou。Amandroid：一种用于 Android 应用安全审查的精确通用组件间数据流分析框架。发表于*ACM SIGSAC*计算机与通信安全会议论文集，第1329–1341页，2014年。

[232] 维基百科。维基百科。Lamport 时间戳。en.wikipedia.org/wiki/Lamport timestamps，2017年。

[233] 维基百科。维基百科。矩阵时钟。en.wikipedia.org/wiki/Matrix_clock, 2022。[234] 维基百科。维基百科。向量时钟。en.wikipedia.org/wiki/Vector clock，2022。

[235] Bill Wilder。 云架构模式：使用 *Microsoft Azure*。 ” O’Reilly Media, Inc.”, 2012年。

[236] Claes Wohlin、Per Runeson、Martin H¨ost、Magnus C Ohlsson、Bj¨orn Regnell 和 Anders Wessl´en。《软件工程实验：导论》。*Kluwer*软件工程国际丛书，2000年。

[237] W. E. Wong、R. Gao、Y. Li、R. Abreu 和 F. Wotawa。《软件故障定位综述》。*IEEE*软件工程汇刊，42(8):707–740，2016年8月。

[238] Michael Wunder、Michael L Littman 和 Monica Babes。《采用 ε-贪心探索的多智能体 Q 学习动态特性的类别》。载于第 *27* 届国际机器学习会议（*ICML-10*）论文集，第 1167–1174 页。Citeseer，2010年。

[239] Jianyu Xiao、Deyun Zhang、Haiquan Chen 和 Hao Dong。《面向并发程序静态切片的改进程序依赖图与算法》。载于先进并行处理技术，第 121–130 页。S pringer，2005年。

[240] Baowen Xu、Ju Qian、Xiaofang Zhang、Zhongqiang Wu 和 Lin Chen。《程序切片简要综述》。*ACM SIGSOFT* 软件工程简报，30(2):1–36，2005年。

[241] Sherif M Yacoub 和 Hany H Ammar。《架构层可靠性风险分析方法》。*IEEE*软件工程汇刊，28(6):529–547，2002年。

[242]Yibiao Yang、Yuming Zhou、Jinping Liu、Yangyang Zhao、Hongmin Lu、Lei Xu、Baowen Xu 和 Hareton Leung。《工作量感知的即时缺陷预测：简单的无监督模型可能优于有监督模型》。载于*ACM SIGSOFT* 软件工程基础国际研讨会论文集，第157–168页，2016年。

[243]Heng Yin、Dawn Song、Manuel Egele、Christopher Kruegel 和 Engin Kirda。《Panorama：捕获系统范围信息流以检测和分析恶意软件》。载于第*14*届*ACM*计算机与通信安全会议论文集，第116–127页。ACM，2007年。

[244] Wei You、Bin Liang、Wenchang Shi、Peng Wang 和 Xiangyu Zhang。《TaintMan：适用于未经修改且未获取 Root 权限的 Android 设备的 ART 兼容动态污点分析框架》。*IEEE* 可靠与安全计算汇刊，2017年。

[245] Matei Zaharia、Mosharaf Chowdhury、Tathagata Das、Ankur Dave、Justin Ma、Mur- phy McCauly、Michael J Franklin、Scott Shenker 和 Ion Stoica。《弹性分布式》

数据集：一种用于内存集群计算的容错抽象。载于作为 *USENIX* 网络化系统设计与实现研讨会的一部分发表，第 15–28 页，2012年。

[246] Angeliki Zavou. 云中的信息流审计。博士论文，哥伦比亚大学，2015年。

[247] Angeliki Zavou、Georgios Portokalidis 和 Angelos D Keromytis。《Taint-Exchange：一种用于跨进程和跨主机污点跟踪的通用系统》。发表于国际安全研讨会，第 113–128 页。Springer，2011年。

[248]Benjamin Zeiss、Diana Vega、Ina Schieferdecker、Helmut Neukirchen 和 Jens Grabowski。《将 ISO 9126 质量模型应用于测试规范——以 Ttcn-3 测试规范为例》。《软件工程 *2007*——*GI* 软件技术专业委员会会议》，2007。

[249] Chen Zhang、Jiaxin Li、Dongsheng Li 和 Xicheng Lu。《理解并静态检测分布式云系统中的同步性能漏洞》。《*IEEEAccess*》，7:99123–99135，2019年。

[250] Feng Zhang、Audris Mockus、Iman Keivanloo 和 Ying Zou。迈向构建通用缺陷预测模型。载于软件仓库挖掘工作会议论文集，第 182–191 页，2014年。

[251] Feng Zhang、Quan Zheng、Ying Zou 和 Ahmed E Hassan。《基于连通性的无监督分类器进行跨项目缺陷预测》。发表于*IEEE/ACM* 国际软件工程会议，第 309–320 页。IEEE，2016年。

[252] Wei Zhang、Liguo Huang、Vincent Ng 和 Jidong Ge。Smplearner：学习预测软件可维护性。自动化软件工程，22(1)：111–141，2015年。

[253] QingHua Zheng、ZhiJiang Ou、Ting Liu、ZiJiang Yang、YuQiao Hou 和 Chao Zheng。基于方法的交互与封装的软件结构评估。中国科学：信息科学，55(12):2816–2825，2012年。

[254] Xiong Zheng 和 Vijay Garg。适用于多线程系统的最优向量时钟算法。载于*2019*年 *IEEE* 第 *39* 届国际分布式计算系统会议（*ICDCS*），第 2188–2194 页。IEEE，2019 年。

[255] Yuming Zhou、Yibiao Yang、Hongmin Lu、Lin Chen、Yanhui Li、Yangyang Zhao、Junyan Qian 和 Baowen Xu。我们在这段旅程中走了多远？跨项目缺陷预测的考察。*ACM*软件工程与方法学汇刊，27(1):1–51，2018年。

[256] David Yu Zhu、Jaeyeon Jung、Dawn Song、Tadayoshi Kohno 和 David Wetherall。《TaintEraser：使用应用级污点跟踪防止敏感数据泄露》。*ACM SIGOPS*操作系统评论，45(1):142–154，2011年。