分类号：按中国图书分类法，学位办网上可查　　单位代码：10335

密　级：注明密级与保密期限　　学　　号：

浙江大学

硕士学位论文

中文论文题目：毕业论文/设计题目

英文论文题目：Graduation Thesis Title

申请人姓名：王家伟

指导教师：陈焰

合作导师：合作导师

学科(专业)：计算机科学与技术

研究方向：研究方向

所在学院：计算机学院

论文递交日期　2025 年 12 月

毕业论文/设计题目

论文作者签名： _____

指导教师签名： _____

论文评阅人 1：_____ 姓名

评阅人 2：_____ 姓名

评阅人 3：_____ 姓名

评阅人 4：_____ 姓名

评阅人 5：_____ 姓名

答辩委员会主席：_____

委员 1：_____

委员 2：_____

委员 3：_____

委员 4：_____

委员 5：_____

答辩日期 _____

Graduation Thesis Title

Author's signature: _____

Supervisor's signature: _____

External reviewers: _____ Name _____

_____ Name _____

_____ Name _____

_____ Name _____

_____ Name _____

Examining Committee Chairperson:

_____

Examining Committee Members:

_____

_____

_____

_____

_____

Date of oral defence: _____

浙江大学研究生学位论文独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 **浙江大学** 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名: 签字日期： 年 月 日

学位论文版权使用授权书

本学位论文作者完全了解 **浙江大学** 有权保留并向国家有关部门或机构送交本论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 **浙江大学** 可以将学位论文的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名: 导师签名:

签字日期： 年 月 日 签字日期： 年 月 日

勘误表

致谢

序言

摘要

在微服务架构主导的云原生时代，云原生应用弹性、敏捷、轻量的特性对网络安全的提出了更高的要求。传统防火墙的粗粒度限制已难以应对动态扩展、容器化部署和零信任环境的复杂挑战，其依赖固定边界和手动配置，导致响应迟缓、资源浪费，并易受横向移动攻击影响，无法适应 Kubernetes 等平台的瞬时迁移和多租户场景。

本文旨在系统地探讨云原生应用安全防护的挑战，分析微隔离技术的原理，并提出一种动静态分析结合的微隔离策略自动生成技术。通过分析云原生应用的代码，实时感知应用集群的环境，动态调整微隔离规则，实现细粒度的访问控制和最小权限原则。本文还设计并实现了一个基于 Kubernetes 的微隔离防护系统，集成了自动化规则生成、实时监控和异常检测功能。

论文的主要工作包括：

1. 提出了一种结合静态代码分析和动态环境感知的微隔离规则自动生成方法，提升了规则的准确性和适应性。

2. 设计并实现大模型辅助的微隔离策略生成系统，利用大模型对代码进行深度理解，生成更符合实际需求的隔离规则。

3. 分析容器的系统调用行为，构建异常检测模型，提高对潜在威胁的识别能力。

Abstract

缩略词表

英文缩写	英文全称	中文全称
ZJU	Zhejiang University	浙江大学
ZJU	Zhejiang University	浙江大学
ZJU	Zhejiang University	浙江大学
ZJU	Zhejiang University	浙江大学
ZJU	Zhejiang University	浙江大学
ZJU	Zhejiang University	浙江大学
ZJU	Zhejiang University	浙江大学
ZJU	Zhejiang University	浙江大学

目录

图目录

表目录

1 绪论

1.1 研究背景与意义

随着云计算技术的深入普及，云原生（Cloud Native）架构已成为构建现代企业级应用的主流标准。根据云原生计算基金会（CNCF）发布的2024年年度报告显示，全球云原生开发者的数量已达到1560万，超过89%的企业正在生产环境中使用或尝试云原生技术[1]。这一技术范式的核心变革，在于利用容器（Container）替代了传统的虚拟机（VM）作为最小计算单元，并利用Kubernetes等编排系统实现了资源的自动化调度[2]。这种转变不仅让软件的开发与交付变得更加敏捷，也彻底重塑了底层基础设施的网络架构。

然而，云原生环境带来的极致弹性与动态性，也让传统的网络安全防御体系面临失效的风险。在传统的物理数据中心中，安全团队通常依赖防火墙在网络边缘构建一个坚固的“防线”，也就是所谓的边界防御（Perimeter-based Defense）。但在云原生环境中，容器的生命周期极短，IP地址频繁变化，服务间的调用关系错综复杂，导致固定的物理边界逐渐模糊甚至消失。Gartner的研究指出，随着云环境复杂度的增加，绝大多数（预计到2025年将达99%）的云安全事故将源于客户的配置错误，而非云平台本身的漏洞[3]。一旦攻击者突破了外围防线，由于内部网络往往缺乏有效的隔离，他们便可以利用“默认可信”的机制在微服务之间进行肆无忌惮的横向移动（Lateral Movement）。

为了应对这种“内网默认可信”带来的安全隐患，Google等业界先驱提出了“零信任（Zero Trust）”的安全理念，主张“永不信任，始终验证”，即无论请求来自网络内部还是外部，都必须经过严格的身份认证与授权[4]。在此背景下，微隔离（Micro-segmentation）技术应运而生。与传统防火墙不同，微隔离将防护的颗粒度细化到了每一个工作负载（Workload）甚至每一个容器，试图通过精细化的流量控制策略，阻断攻击者的横向渗透路径[5]。

虽然微隔离技术的理念已经非常成熟，但在实际的云原生工程落地中，运维人员仍面临着巨大的挑战，主要体现在以下三个方面：

1. 策略维护难：在Kubernetes集群中，容器的生灭往往以分钟甚至秒为单位，IP地

址的剧烈变动使得基于 IP 的传统黑白名单瞬间失效。运维人员很难手动维护一套能够实时跟随业务变化的动态策略[6]。

2. 管理复杂度高：微服务架构将单体应用拆分为成百上千个独立服务，服务间的调用链路呈指数级增长。面对如此庞大的分布式网络，依靠人工梳理业务逻辑并配置数万条隔离规则，不仅效率极低，而且极易出现配置冲突或遗漏。

3. 流量可视性差：现有的安全工具大多只能监控到网络层（L3/L4）的连接状态，缺乏对应用层（L7）业务语义的理解。这意味着安全团队很难区分正常的业务调用与利用合法端口进行的恶意攻击，难以制定出既安全又不影响业务的精准策略[7]。

综上所述，传统的、依赖人工配置的静态安全手段已无法适应云原生环境的高动态特征。因此，如何深入理解业务逻辑，自动化地生成准确、实时且具备可解释性的微隔离策略，已成为当前云原生安全领域急需解决的关键问题。

1.2 国内外研究现状

为了应对微服务环境下的服务实例生命周期大幅缩短，IP 地址不再固定，服务间的依赖关系呈现出随业务逻辑动态变化的网状拓扑特征[8]等挑战，围绕如何自动化生成适应云原生环境的微隔离策略，现有的技术方案主要根据数据源的不同分为三类：基于流量分析的策略生成技术、基于角色的策略生成技术以及基于静态分析的策略生成技术。接下来，本文将逐一介绍这些技术。

1.2.1 基于流量分析的策略生成技术

基于流量分析的策略生成技术是目前工业界应用最为广泛的方案，其核心思想是“以观测代替预设”。该技术不依赖源代码或人工配置，而是通过持续监控生产环境中的实际通信行为，逆向推导服务依赖关系，进而生成白名单策略。

从技术实现来看，主要通过 Sidecar 代理或 eBPF（Extended Berkeley Packet Filter）技术进行采集。Isovalent 公司的技术白皮书指出，基于 eBPF 的无侵入式观测能够以极低的性能损耗，在内核层直接捕获全集群的网络拓扑[9]。尽管该方法具备部署便捷的优势，但针对构建严密安全防线的这一需求看，单纯依赖流量分析存在三个本质缺陷：

1. 观测覆盖率不足与长尾效应：流量分析本质上是一种基于时间窗口的采样。它只能记录观测期间发生的“显式行为”，无法覆盖“隐式合法行为”。Zhou 等人[10]在 IEEE Transactions on Software Engineering 上的实证研究表明，微服务故障传播路径往往涉及低频调用的边缘服务，这些长尾路径在正常监控中很难显现。Chen 等人[11]的经典研究也证实，即使经过长达数周的采样，仍会有大量合法的边缘业务逻辑（如灾备切换、审计任务）因未被触发而未被记录。若仅依据不完整的流量样本生成白名单，一旦这些低频业务在未来被触发，就会被误判阻断，导致生产事故。

2. 策略生成的滞后性与冷启动问题：基于学习的机制存在天然的“冷启动”窗口。在服务刚上线或版本更新初期，系统必须积累足够的数据才能计算出收敛的行为基线。He 等人[12]指出，机器学习模型在面对数据分布快速变化时表现出显著的滞后性。此外，云原生工作负载具有高度瞬态性。Sysdig 的监测数据显示，超过 70% 的容器生命周期不足 5 分钟[13]。对于这些短生命周期的对象，往往是探针还未收集到足够的流量样本，容器就已经销毁，导致无法生成有效的防护策略。

3. 脏数据引发的策略“毒化”风险：流量分析方案通常假设“观测到的流量均为良性”。然而，Gnutti 等人[14]的研究指出，在存量系统中，攻击者可能利用伪造的正常流量模式来混淆检测模型。如果缺乏人工甄别，将所有历史连接直接固化为白名单，实际上等同于将潜在的恶意行为（如探测扫描）合法化了，这种由脏数据导致的策略“毒化”将永久性地破坏安全边界。

针对上述局限性，近年来开始出现结合大语言模型（LLM）进行策略优化的尝试。例如，KubeGuard[15] 提出利用 LLM 分析运行时日志来辅助生成最小权限配置。同时，KubeAegis 框架[16]也尝试通过统一策略管理来解决流量观测到的策略规则碎片化问题。然而这些尝试仍然不能从根本上解决上述缺陷。

1.2.2 基于角色的策略生成技术

为了解决云原生环境下 IP 地址频繁变动导致网络策略失效的问题，研究人员提出了将访问控制策略与底层网络标识解耦的思路，主张提取工作负载的高层身份属性来构建抽象的角色模型，并依据“用户/服务-角色-权限”的映射关系来生成逻辑隔离策略。

在理论层面，Sandhu 等人[17]提出的 RBAC96 模型奠定了这一领域的基石，通过引入“角色”作为中间层，极大地简化了大规模系统中的权限管理复杂度。在 Kubernetes 等现代编排平台中，这种机制通常表现为利用 ServiceAccount 作为策略锚点。这种方法实现了防护逻辑的语义化，使得安全策略能够跟随容器的生命周期自动迁移，而无需像防火墙那样频繁更新 IP 规则。

然而，尽管基于角色的方法在理论上很完善，但在面对大规模微服务集群的实际工程落地中，这种静态的授权模型逐渐暴露出了三个显著的缺陷：

1. 角色爆炸与管理熵增：微服务架构的核心原则是职责单一，这意味着随着业务的发展，服务的数量会呈线性甚至指数级增长。Ferraiolo 等人[18]指出，当系统规模扩大时，如果严格遵循“最小特权原则”为每一个微服务实例定制专属角色，将导致角色数量急剧膨胀，即所谓的“角色爆炸（Role Explosion）”现象。在云原生环境下，这一问题尤为严重。Rostami 等人[19]指出，Kubernetes 的 RBAC 系统包含极其复杂的动词（Verbs）和资源（Resources）组合，这种组合的复杂性使得在大型集群中维护成千上万个“服务-角色”映射关系变得极难管理。

2. 过度授权带来的权限泄露风险：为了规避角色爆炸带来的管理难题，工程实践中往往会出现“走捷径”的现象。Kuhn 等人[20]指出，通过“角色复用”来减少配置量虽然降低了管理成本，但不可避免地扩大了攻击面。这一点在近年来的安全报告中得到了验证。Red Hat 发布的 2024 年 Kubernetes 安全报告[21]显示，超过 46% 的组织曾因权限配置错误（Misconfiguration）导致安全事故。最新的自动化检测研究 EPScan[22]也证实，在生产环境中，大量的微服务拥有它们实际业务逻辑并不需要的“僵尸权限”，这意味着一旦单一容器被攻陷，攻击者就可以继承该角色所有的多余权限进行横向移动。

3. 静态授权无法适应动态风险：传统的 RBAC 模型本质上是一种静态授权机制。策略决策完全依赖于预先定义的静态绑定关系，缺乏对运行时上下文（Context-Awareness）的感知能力。NIST 的研究员 Hu 等人指出，静态角色难以将动态属性（如当前的负载压力、访问时间窗口、客户端地理位置或实时的威胁情报等级）纳入决策逻辑[23]。在零信任架构中，访问请求的合法性往往取决于瞬时的系统状态，基于静态角色的策略在应对突发威胁时，往往显得过于僵化。

综上所述，基于角色的策略生成技术虽然解决了网络标识漂移的问题，但在细粒度控制与动态适应性方面仍存在天然的逻辑局限。

1.2.3 基于静态分析的策略生成技术

基于静态分析的策略生成技术体现了“安全左移（Security Shift Left）”的防御思想。与前两种依赖运行时数据的方案不同，这种技术路线主张在应用部署之前的构建阶段（Build Phase），通过扫描源代码、字节码或配置文件，提前推导服务之间的调用关系，从而生成最小特权白名单。

在学术界，这一方向的研究主要集中在如何通过代码分析来提取微服务的拓扑结构。Li 等人[24]提出的 AutoArmor 系统是该领域的代表性工作，它通过对微服务源代码进行静态切片分析，自动提取服务间的 API 调用链并生成访问控制策略。类似地，Ghavamnia 等人[25]提出的 Confine 系统，则专注于通过静态分析容器内的二进制文件来削减系统调用（System Calls）的攻击面，证明了静态方法在收敛权限方面的有效性。

这种方法的显著优势在于其覆盖率高且具备确定性。静态分析可以遍历代码中的所有逻辑分支，理论上能够发现那些在运行时很难被触发的“冷路径”依赖（例如灾难恢复逻辑），且生成的策略直接映射代码意图，具备天然的可解释性。然而，在云原生微服务的实际工程应用中，纯静态分析面临着三个难以逾越的“语义盲区”：

1. 动态语言特性的解析难题：现代微服务应用广泛使用反射（Reflection）、动态代理和依赖注入等动态特性。Cerny 等人[26]指出，现有的静态分析工具在处理微服务架构时存在显著的“上下文缺失”问题，特别是当调用目标由配置文件或数据库动态决定时，静态工具很难推断出准确的调用图。最新的 arXiv 研究[27]进一步提出“语义依赖”的概念，指出微服务间大量存在的隐式逻辑依赖无法仅通过语法分析来捕获，这导致纯静态生成的策略往往是不完整的。

2. 基础设施层的不可视：在云原生架构中，许多关键的网络行为并非由业务代码直接控制，而是由底层基础设施接管。例如，在 Istio 等服务网格环境中，Sidecar 代理会自动注入到 Pod 中并接管流量。这种运行时的架构变化无法在静态的源代码或 Dockerfile 中被预先捕获。Tigera 的技术报告[28]指出，静态分析无法感知由 Kubernetes 控制平面或 Sidecar 引入的额外网络路径，这意味着仅凭业务代码生成

的策略，可能会因为遗漏基础设施组件的通信需求而导致服务启动失败。

3. 多语言异构环境的适配成本：微服务架构的一个重要特性是技术栈的异构性。Pereira-Vale 等人[29]中分析了 50 种现有的安全方案，发现绝大多数静态分析工具缺乏能够跨越语言边界追踪调用链的通用解决方案。这种工具链的碎片化，使得在多语言混合的微服务系统中构建全局一致的依赖拓扑变得极具挑战性。

综上所述，虽然静态分析能够提供高精度的代码级依赖视图，但受限于动态特性和基础设施层的不可见性，它无法独立生成完整的运行时微隔离策略。这也为本文提出“动静态协同分析”的思路提供了有力的事实依据。

1.3 研究内容与意义

1.3.1 当前云原生零信任微隔离领域面临的主要挑战

随着云原生架构（Cloud-Native Architecture）逐步确立为现代数据中心的基础设施标准，零信任微隔离技术已成为在不可信网络环境中保障大规模分布式微服务集群安全的核心范式。然而，尽管该技术在理论层面已趋于成熟，但在实际的工程化落地过程中，仍面临着严峻的挑战。这些挑战不仅源于云原生环境本身固有的高动态性（High Dynamism）与架构复杂性，更深层次地涉及到安全策略在技术实现、运维管理以及合规审计等多重维度的制约。

鉴于此，本节将从策略生成的准确性、异构环境的一致性、运行时防护的自愈能力以及决策逻辑的可解释性这四个关键维度展开详细剖析。通过结合国内外最新的研究成果与行业实践报告，本文旨在系统性地解构制约零信任微隔离技术发展的核心瓶颈，从而明晰当前的研究空白，为本文提出的创新性解决方案奠定立论基础。

1.3.1.1 策略生成的准确性

在云原生环境中，微服务的高频部署与容器的短生命周期特性，对零信任策略的准确性与响应速度提出了极高的要求。然而，现有的技术路线普遍陷入了“静态覆盖不足”与“动态噪声干扰”的两难境地，很难在准确性与时效性之间找到平衡点。

一方面，基于静态分析的策略生成技术虽然能从代码逻辑推导出最小权限基线，但存在显著的“检查时与使用时不一致”（Time-of-Check to Time-of-Use, TOCTOU）问题。在 Kubernetes 集群中，工作负载的动态行为——例如 Pod 的故障迁移、Sidecar 代理的热加载——往往发生在部署之后。这意味着静态预设的规则很容易与运行时真实的拓扑产生时空错位，导致短暂的权限漏洞窗口[30]。相关的行业调查显示，超过 35% 的安全团队在实施微隔离时，因为无法动态适应应用变更而遭遇了严重的性能瓶颈与业务阻断，最终迫使运维人员不得不放宽策略粒度，牺牲安全性来换取可用性[31]。

另一方面，基于流量学习的动态生成方法虽然具备实时性，但极易受环境噪声干扰而产生“过拟合”或“欠拟合”现象。Palo Alto Networks 的研究指出，在东西向流量高度复杂的微服务网格中，单纯依赖流量基线很难有效区分合法的异常波动与恶意的横向移动。在零日攻击场景下，这种方法的有效阻断率仅徘徊在 15% 到 30% 之间，且存在较高的误报风险[32]。

此外，在多云分布式架构下，跨域策略数据的同步往往存在显著的“收敛延迟”。CISA 在其零信任成熟度模型中强调，传统策略引擎在面对云原生“瞬生灭变”的特征时，策略下发的滞后性已成为制约防御效能的主要瓶颈[30]。尽管近年来有学者尝试利用机器学习算法来优化流量聚类，但这类黑盒模型在面对对抗样本时泛化能力有限，且缺乏可解释性，很难作为关键基础设施的唯一决策依据[12]。

综上所述，单一的静态预判或动态学习均无法满足云原生微隔离的高标准要求，构建一种融合静态语义与动态行为的混合式策略生成机制势在必行。

1.3.1.2 异构环境的一致性

云原生架构的演进呈现出显著的 ** 多云混合（Multi-Cloud Hybrid）** 趋势，微服务往往跨越私有云（Kubernetes）、公有云托管平台（如 AWS EKS, Azure AKS）以及边缘计算节点（Edge Nodes）进行分布式部署。这种基础设施的异构性导致零信任微隔离策略在跨域迁移时面临严重的一致性挑战。

首先，底层控制平面的碎片化导致了严重的“厂商锁定”与策略孤岛。现有的微隔离方案高度依赖特定的网络插件（CNI）或服务网格实现（如 Istio 基于 Envoy 代理，Cilium 基于 eBPF 钩子）。不同技术栈之间的策略模型存在巨大的语义鸿沟，缺乏统一的标准描述语言[1]。例如，在多云架构中，将业务从私有 IDC 迁移至公有云时，往往需

要将数千条 iptables 规则手动重构为云厂商特定的 Security Group 规则。这种重复劳动不仅导致运维成本指数级增长，更违背了云原生“声明式（Declarative）”与“一次编写，到处运行”的设计初衷[33]。

其次，“棕地（Brownfield）”集成是企业数字化转型中不可回避的痛点。现实系统中往往共存着现代化的微服务容器与遗留的单体应用（Legacy Monoliths）。这些遗留系统缺乏 Sidecar 注入能力或对现代零信任协议（如 SPIFFE/SPIRE）的支持，导致微隔离策略在实施时出现兼容性断层[34]。

此外，在 5G 核心网（5G Core）与 Serverless 无服务器计算等新兴场景下，策略的执行效率与粒度平衡面临严峻考验。3GPP 在其安全架构标准中明确指出，服务化架构（SBA）中的高吞吐量信令交互要求安全策略必须具备极低的时延，然而各设备厂商对标准的不同解读与自定义实现（Proprietary Implementation），导致跨厂商设备的微隔离互操作性极差[35]。特别是在 Serverless 场景下，由于底层基础设施被云厂商深度抽象，用户失去了对操作系统内核的控制权，传统的基于代理（Agent-based）的微隔离机制失效。研究表明，在无服务器环境中移植策略往往需要针对特定运行时进行繁琐的手动调整，极易引入“配置漂移（Drift）”风险，导致防护效能显著下降[36]。

权威调研报告指出，这种跨环境的策略碎片化已成为多云安全治理的最大障碍，构建一个跨平台、标准化的策略抽象层已成为行业迫切需求[37]。

1.3.1.3 运行时防护的自愈能力

在云原生环境中，持续集成/持续部署（CI/CD）流水线带来的灰度发布、自动扩缩容以及故障自愈机制，使得基础设施处于极度不稳定的“流体”状态。这种高频的动态变化导致预定义的静态白名单与实际通信路径之间产生显著的“时空错位”，即所谓的“运行时漂移（Runtime Drift）”*。

首先，微服务实例的“瞬态性（Ephemerality）”加剧了策略的“腐化（Decay）”速率，Datadog 的大规模测量研究也证实，Kubernetes 集群中超过三分之一的 Pod（如 Job 或 CronJob 类型）在 1 分钟内即完成销毁[38]。然而，现有的策略执行点（PEP）往往依赖于静态的 IP 地址绑定或非实时的标签同步。当新实例上线或发生故障迁移时，基于 iptables 或 IPVS 的底层规则更新往往存在分钟级的延迟，导致新实例在启动初期缺乏管控，或旧规则残留形成“幽灵策略（Ghost Policies）”，极易被攻击者利用进行持久化驻

留[39]。

其次，缺乏针对“概念漂移（Concept Drift）”的闭环矫正机制是当前技术的重大缺陷。在多租户与多集群场景下，业务流量模式并非一成不变，而是随着业务逻辑迭代发生动态偏移。现有的监控体系多侧重于系统指标（Metrics）而非安全语义，导致漂移检测存在严重的滞后性（Lag）。NIST 在其微服务安全标准（SP 800-204B）中指出，若缺乏对应用层语义（L7）的持续再验证，任何静态授权模型都会随着时间推移而退化为过权状态（Over-privileged）[7]。Gartner 的市场指南进一步量化了这一风险：超过 60% 的微隔离失效并非源于初始策略的逻辑错误，而是源于策略未能跟随工作负载的变更而及时演进，最终导致运维团队被迫将防护模式回退至“监控模式”[3]。

尽管学术界尝试引入深度学习（Deep Learning）构建自适应的自愈框架，但在生产环境中部署此类“重型”模型面临严峻的 **“资源-精度”权衡 ** 难题。一方面，高频的流量推理抢占了业务容器宝贵的计算资源（CPU/Memory），违背了云原生轻量化的原则；另一方面，研究表明[40]，基于统计学的异常检测模型在面对非平稳分布的流量时，极易混淆正常的突发流量（Flash Crowd）与恶意攻击，高误报率（False Positive）使得自动化阻断功能在实际工程中难以落地。

1.3.1.4 决策逻辑的可解释性

在零信任微隔离的落地实践中，自动化策略生成算法往往面临着“计算高效”与“逻辑透明”的零和博弈。尽管自动化工具能够基于流量或依赖关系快速生成海量的访问控制列表（ACL），但其生成过程的 **“黑盒化”** 特征严重阻碍了在严格合规环境下的工程应用。

首先，监管法规对自动化运维系统的 **“可审计性（Auditability）”** 提出了硬性约束。我国《网络安全等级保护基本要求》（GB/T 22239-2019）明确规定，安全管理中心必须具备对安全策略配置依据的追溯与核查能力[41]。欧盟《通用数据保护条例》（GDPR）亦强调了对“自动化决策（Automated Decision-Making）”的解释权。然而，现有的策略生成引擎大多仅输出最终的阻断/放行规则，缺乏对 **“归因链条（Attribution Chain）” ** 的完整记录——即无法回答“某条规则是基于哪个业务意图、哪段通信历史或哪项合规基线而生成”的因果问题[42]。

其次，**“语义鸿沟（Semantic Gap）” ** 与 **“规则爆炸” ** 导致了认知负荷的极

限挑战。大规模微服务集群往往产生数以万计的底层网络规则（L3/L4），这些规则以 IP、端口或标签为载体，彻底丢失了上层的业务语义（L7）。ACM Computing Surveys 的最新综述指出，随着微服务数量的增长，底层规则之间的 **“逻辑冲突（Logical Conflicts）”**（如影子规则、冗余规则）呈指数级上升，且难以通过人工手段识别[43]。

此外，在多策略源并存的复杂场景下，算法内部的 **“冲突消解机制（Conflict Resolution）”** 往往是不透明的。NIST 在其属性访问控制（ABAC）标准中指出，当静态基线与动态流量推导出的规则发生逻辑互斥时，自动化系统通常基于预设的优先级（如“拒绝优先”）进行静默处理，而未向管理员暴露决策依据，导致实际生效的安全边界与预期设计严重偏离[44]。

因此，构建一种具备全链路溯源能力、能够将底层规则映射回高层业务意图的 **“白盒化”** 策略生成框架，是解决信任危机的关键。

1.4 解决方案及其意义

针对上述云原生微隔离领域在策略准确性、环境一致性以及可解释性方面面临的严峻挑战，本文提出了一种基于动静态协同分析与大模型语义增强的零信任微隔离策略生成框架。该框架旨在打破传统网络层（L3/L4）与业务应用层（L7）之间的语义壁垒，构建具备全生命周期感知与自适应能力的防护体系。

本文的核心解决方案包含以下两个层面的技术创新：

1.4.1 动静态深度融合的策略生成机制

为了解决单一分析视角存在的“盲区”，本文设计了静态代码分析（Static Analysis）与动态运行时感知（Dynamic Monitoring）的双向校验机制，以实现策略生成的全覆盖与高精度。

在静态侧，系统引入了程序语言分析领域的经典工具 Soot 框架。Vallée-Rai 等人在 1999 年提出的 Soot 框架能够对 Java 字节码进行全程序分析（Whole-Program Analysis），构建高精度的类粒度调用图（Call Graph）[45]。利用这一能力，系统可以预先识别代码中潜在的隐式依赖与远程调用逻辑，从而覆盖那些在测试阶段未被流量触发的“冷路径”（Cold Paths），有效解决了纯流量学习方案覆盖率不足的问题。

在动态侧，系统利用 Kubernetes 的 Informer 机制与 eBPF 技术，实时监听集群内的资源变更（如 Pod 漂移、Service 变动）及实际流量轨迹。Isovalent 的研究表明，通过内核层的 eBPF 探针可以无侵入地获取服务间的真实拓扑[9]。本文通过将静态分析提取的调用链路与动态采集的运行时拓扑进行图同构映射（Graph Isomorphism Mapping），既剔除掉了静态代码中未被加载的死代码（Dead Code），又补全了静态分析无法感知的动态反射调用，从而实现了策略生成的“高召回”与“低误报”。

1.4.2 大模型驱动的语义对齐与合规解释

针对自动化策略生成过程中的“黑盒化”与“语义鸿沟”问题，本文引入大语言模型（LLM）作为安全知识的推理引擎，利用其强大的语义理解能力来提升策略的可解释性。

首先，系统集成了 DeepWiki 仓库级依赖挖掘技术。Du 等人在 ACL 2025 上发表的研究指出，DeepWiki 能够跨越文件边界，精准提取代码库中的深层依赖关系[46]。利用这一技术，系统可以将底层的 Kubernetes NetworkPolicy 规则（YAML）逆向映射回具体的业务代码片段与高层设计意图，解决策略与业务割裂的问题。

其次，系统采用了思维链（Chain-of-Thought, CoT）推理技术。Wei 等人在 NeurIPS 上证明，通过引导大模型生成中间推理步骤，可以显著提升其处理复杂逻辑任务的准确性[47]。本文利用 CoT 技术构建了双向解释机制：一方面，将自然语言描述的安全需求自动转化为形式化的 ACL 规则；另一方面，为每一条生成的阻断策略提供符合自然语言逻辑的“归因证据”，例如指出某条规则是基于哪个业务模块的数据库访问需求而生成的。Chen 等人的研究表明，这种基于代码语义的解释机制能够显著降低安全审计的认知门槛，使自动化策略满足合规性要求[48]。

1.4.3 研究意义

本研究的学术与应用意义主要体现在以下两个方面：

1. 理论意义：本文提出了一种融合程序语言分析（PL）与网络安全（Security）的跨域建模方法。通过将代码级的静态语义与网络级的动态行为进行对齐，验证了利用大模型解决安全策略“语义鸿沟”的可行性，为零信任架构下细粒度访问控制的自动化研究提供了新的理论范式。

2. 工程价值：通过自动化替代人工配置，本方案显著降低了大规模微服务集群的运维熵增（Entropy）。实测表明，该框架可将策略部署效率提升一个数量级，同时确保了在多云异构环境下的策略一致性，为企业核心业务上云提供了可落地、可审计且具备自愈能力的内生安全保障。

1.5 论文结构

本文围绕云原生环境下微服务安全防护的核心痛点，遵循“背景分析—理论基础—系统实现—实验验证—总结展望”的逻辑脉络展开论述，全文共分为六章，各章节的具体组织结构如下：

第一章：绪论。本章首先阐述了云原生技术范式的演进背景，剖析了在容器化与微服务架构下，传统边界防御失效、东西向流量不可视以及动态漂移等安全挑战，明晰了研究零信任微隔离技术的迫切性与工程意义。随后，系统梳理了国内外在基于流量分析、静态分析及角色访问控制等领域的最新研究进展，指出了现有方案在策略覆盖率、实时性与可解释性方面存在的局限。最后，概括了本文的主要研究内容、创新点及全篇的组织架构。

第二章：相关技术概述。本章为后续系统的设计与实现奠定理论与技术基础。首先，详细介绍了容器核心原理（Namespace/Cgroups）与 Kubernetes 编排机制，重点解析了 NetworkPolicy 的执行流程与 Informer 事件驱动模型。其次，阐述了 Soot 静态分析框架在 Java 字节码切片与调用图构建中的应用。最后，介绍了大语言模型（LLM）的基础原理，重点探讨了 Transformer 架构、DeepWiki 仓库级依赖挖掘技术以及 LLM 在结构化输出与安全约束方面的技术路径。

第三章：动静态分析的零信任网络策略生成系统。本章详细论述了“确定性”策略生成子系统的设计与实现。首先，介绍了基于 eBPF 与 K8s Informer 的运行时数据采集模块，涵盖 Pod、Service 及注册中心配置的多维感知。其次，深入剖析了基于 Soot 的静态分析引擎，阐述了第三方依赖过滤、自定义类提取及类/微服务粒度调用关系图（Call Graph）的构建算法。最后，提出了从服务拓扑到 Pod 标签的映射机制，以及具备冗余消除与冲突检测能力的最小化白名单生成算法。

第四章：基于大模型代码分析的零信任微隔离策略双向智能生成。本章重点论述了

引入 LLM 进行语义增强的“智能化”子系统。首先，构建了统一的多源信息采集机制，实现了 GitHub 仓库与 Kubernetes 运行时资源的精确绑定。在此基础上，设计了双向生成模块：一方面，实现了以 NetworkPolicy 为起点的逆向解释，利用大模型对规则合理性进行校验并生成自然语言解释；另一方面，实现了以 Pod 为起点的正向生成，结合 DeepWiki 安全知识库与代码语义理解，自动合成符合最小权限原则的微隔离策略。

第五章：实验与分析。本章对提出的框架进行了系统性的验证。首先介绍了基于双集群 Kubernetes 的实验环境搭建方案与包含 14 个典型云原生组件（如 Airflow, TiDB）的测试数据集。随后，定义了真阳性率（TP）与假阳性率（FP）等评价指标。最后，对比分析了不同大语言模型（如 GPT-4o 与 DeepSeek R1）在策略校验任务中的性能表现，并对自动生成的策略数量与质量进行了统计验证，证明了本方案在精确性与安全性上的优势。

第六章：总结与展望。本章对全文的研究工作与核心成果进行了全面总结，客观分析了当前系统在极端高并发场景下的性能瓶颈与局限性，并对未来在多云异构架构支持、边缘计算场景适配等方向的研究前景进行了展望。。

1.6 本章小结

本章介绍了云原生应用对分布式系统安全带来动态威胁与碎片化挑战的课题背景，并将零信任微隔离技术分类。从基于流量分析的策略生成、基于角色的访问控制、基于静态分析的策略生成这三类防护技术分别介绍了国内外的最新研究进展。之后，本章介绍了本文所设计和实现的零信任微隔离策略生成框架，并阐述了其中的挑战，工作与意义。最后，介绍了本文的论文结构。

2 相关技术

本章重点阐述支撑零信任微隔离策略生成系统的核心技术基础。首先分析容器技术的隔离机制及其在网络安全层面的局限性；其次解析 Kubernetes 编排系统的网络模型与事件驱动机制；随后探讨 Soot 静态分析框架在字节码依赖提取中的应用；最后阐述大语言模型在代码语义理解与自动化推理方面的技术原理。

2.1 容器技术

容器技术通过操作系统级的虚拟化实现应用的轻量级封装与隔离，是云原生架构的基石。不同于传统虚拟机（Virtual Machine, VM）依赖 Hypervisor 模拟硬件层，容器直接运行于宿主机内核之上，具备毫秒级启动与高密度部署特性[49]。

2.1.1 命名空间与控制组机制

容器的隔离性主要依赖于 Linux 内核提供的命名空间（Namespaces）与控制组（Cgroups）机制。命名空间负责实现资源的逻辑隔离，其中 PID Namespace 实现了进程视图的隔离，Network Namespace 提供了独立的网络协议栈（包括网卡、路由表、防火墙规则），而 Mount Namespace 则通过挂载点隔离构建了独立的文件系统视图[50]。控制组（Cgroups）则负责对 CPU、内存、磁盘 I/O 等物理资源进行配额限制与优先级调度，防止单一容器耗尽节点资源[51]。

然而，容器的隔离本质上是共享内核的进程级隔离。相比于虚拟机的硬件级虚拟化，容器并未实现彻底的攻击面隔离。研究表明，一旦发生内核漏洞逃逸，攻击者极易突破命名空间限制获取宿主机权限[52]，这对容器间的网络微隔离提出了更高的防护要求。

2.1.2 联合文件系统

容器镜像采用分层存储架构，通过联合文件系统（UnionFS）将不同的物理目录挂载到同一虚拟文件系统下。典型的实现如 OverlayFS，利用下层（LowerDir）的只读镜像层与上层（UpperDir）的可读写容器层进行堆叠[53]。这种写时复制（Copy-on-Write,

CoW）机制不仅优化了存储空间，也为静态分析提供了便利：通过解析只读镜像层的文件结构，可以在不运行容器的情况下提取应用制品（Artifacts），为后续的依赖分析提供数据源。

2.1.3 容器网络接口标准

随着容器生态的发展，容器网络接口（Container Network Interface, CNI）成为连接容器运行时与网络插件的标准规范[54]。CNI 插件负责在容器创建时分配 IP 地址、配置网桥或虚拟网卡（veth pair），并负责路由表的维护。在云原生环境中，CNI 插件（如 Calico、Cilium）是实施微隔离策略的执行主体，它们通过拦截进出 Pod 的网络流量，依据预定义的策略进行放行或阻断。

2.2 Kubernetes 容器编排技术

Kubernetes（K8s）作为当前云原生生态的事实标准，源于 Google 内部的 Borg 集群管理系统。它不仅是一个部署工具，更提供了一套用于构建分布式系统的声明式 API 和控制平面，为微服务的自动化治理提供了基础设施层面的支持[55]。

2.2.1 核心架构与资源抽象

Kubernetes 采用典型的“控制平面-数据平面”解耦架构。控制平面（Control Plane）包含 API Server、Etcd、Scheduler 和 Controller Manager 等核心组件，负责维护集群的全局状态与调度决策；数据平面（Data Plane）则由运行在各工作节点上的 Kubelet 和 Kube-Proxy 组成，负责容器生命周期的具体执行与网络规则的维护。

在资源抽象层面，Kubernetes 引入了 Pod 作为最小调度单元，通过将紧密耦合的容器封装在共享的 Network 和 IPC 命名空间中，解决了微服务进程间的协同问题。Service 资源则提供了一组 Pod 的逻辑抽象与稳定的虚拟 IP（ClusterIP），利用标签选择器（Label Selector）屏蔽了后端 Pod IP 动态变化带来的寻址复杂性。这种基于标签的松耦合关联机制，是本文构建动态微隔离策略的核心依据。

2.2.2 网络策略模型

为了在扁平化的 Pod 网络中实现访问控制，Kubernetes 定义了 NetworkPolicy 资源对象。该对象本质上是一种以应用为中心的白名单防火墙规则，允许管理员通过 `podSelector` 和 `namespaceSelector` 精确界定流量边界，控制入站（Ingress）与出站（Egress）的通信权限[56]。

值得注意的是，NetworkPolicy 仅定义了策略规范，其实际执行依赖于底层的 CNI 插件（如 Calico, Cilium）。虽然基于 iptables 或 IPVS 的传统实现能够满足基本的 L3/L4 隔离需求，但随着集群规模的扩大，海量规则的查找与更新面临显著的性能瓶颈。此外，原生 NetworkPolicy 缺乏对应用层协议（L7）的感知能力，无法防御基于合法端口的逻辑攻击，这为引入代码级语义分析提供了必要性。

2.2.3 Informer 事件驱动机制

在零信任安全体系中，策略引擎需要实时感知工作负载的生灭与漂移。若采用传统的轮询（Polling）方式查询 API Server，不仅时效性差，且会给控制平面带来巨大的负载压力。Kubernetes 客户端库（client-go）为此提供了 Informer 机制，基于 List-Watch 接口实现了高效的事件驱动模型[57]。

Informer 的核心组件包括 Reflector 和 Indexer。Reflector 负责与 API Server 建立长连接，实时监听（Watch）资源变更事件，并将增量数据写入 DeltaFIFO 队列；Indexer 则在客户端本地维护一份线程安全的只读缓存，通过索引加速数据检索。本文提出的策略生成系统深度利用了 SharedInformer 机制，能够在毫秒级延迟内捕获 Pod 的创建、销毁及标签变更事件，从而触发微隔离策略的动态计算与下发，解决了静态策略无法适应云原生动态环境的难题。

2.3 Soot 静态分析技术

Soot 是 Java 语言分析领域最通用的静态分析与优化框架之一，最初由麦吉尔大学 Sable 研究组开发。它通过将 Java 字节码（Bytecode）转换为中间表示（Intermediate Representation），提供了一套用于分析和转换 Java 程序的 API。在云原生微服务场景下，Soot 能够直接处理编译后的 `.class` 文件或 JAR 包，无需源代码即可提取类结构、继承

关系及成员变量定义，为构建服务间的依赖关系提供了底层技术支撑[58]。

2.3.1 Java 字节码与类文件解析

Java 字节码是运行在 Java 虚拟机（JVM）上的二进制指令集。根据 JVM 规范，类文件（Class File）存储了类的全限定名、父类引用、接口实现列表、字段表（Fields）及方法表（Methods）。Soot 框架首先通过 `SootResolver` 组件加载类路径下的所有类文件，将其解析为内存中的 `SootClass` 对象。这一过程能够完整保留类的元数据信息，包括修饰符（public/private）、注解（Annotations）以及泛型签名，是后续识别 AOP 切面标记或依赖注入注解的基础[59]。

2.3.2 Jimple 中间表示

为了克服原生字节码基于栈架构（Stack-based）难以分析的缺点，Soot 引入了 Jimple 中间表示。Jimple 是一种基于类型的三地址码（3-Address Code），它将复杂的栈操作指令转换为显式的赋值语句（如 `x = y + z`）。

Jimple 的核心优势在于简化了控制流分析。在 Jimple 中，所有的实例方法调用（Invokevirtual）和接口调用（Invokeinterface）都被显式保留，且变量的定义与使用关系（Def-Use）清晰可见。这使得分析工具能够直接通过扫描 Jimple 语句（Statement）来识别代码中的方法调用行为，而无需模拟 JVM 的操作数栈行为[60]。

2.3.3 类层次与结构分析能力

Soot 提供了强大的类层次分析（Class Hierarchy Analysis, CHA）与结构提取能力，这是构建类之间静态依赖关系的核心机制：

- 继承与实现关系分析：Soot 维护了一个全局的类层次结构树（Hierarchy），能够快速查询任意两个类是否存在继承（Extends）或接口实现（Implements）关系。这为识别基于多态的调用链路提供了必要的类型约束信息[61]。

- 成员变量与字段分析：通过 `SootField` 对象，Soot 允许分析人员遍历类中定义的所有成员变量及其类型。这在微服务分析中尤为重要，因为许多服务依赖是通过

成员变量注入（如 `@Autowired` 或 `@Resource`）建立的。Soot 能够提取字段的声明类型，从而建立当前类与字段类型之间的关联边。

- 方法体与切面逻辑可见性：对于采用面向切面编程（AOP）的代码，编译后的字节码中往往包含由编译器织入（Weaving）的额外调用逻辑。Soot 对字节码的分析基于最终的编译产物，因此能够“看到”并解析这些由 AOP 框架自动生成的字节码指令，从而建立起切面类与目标类之间的关联。

综上所述，Soot 不仅是一个字节码转换工具，更提供了一个包含类结构、字段定义及方法签名的完整对象模型，为静态提取微服务内部复杂的类间依赖提供了完备的视图。

2.4 大语言模型技术

大语言模型（Large Language Model, LLM）是近年来人工智能领域的重要进展。在云原生安全领域，LLM 不仅需要理解单文件的代码逻辑，更需要具备跨文件的仓库级理解能力，以弥合“非结构化代码逻辑”与“结构化安全策略”之间的语义鸿沟[62]。

2.4.1 Transformer 架构与代码建模

现代大语言模型主要基于 Transformer 架构，其核心创新在于引入了自注意力（Self-Attention）机制。对于代码分析任务，Transformer 的优势在于能够捕捉长距离的依赖关系（Long-range Dependencies）。通过多头注意力（Multi-Head Attention）机制，模型能够从不同的语义子空间同时关注代码的语法结构、数据流向及控制流逻辑，从而在没有显式编译过程的情况下理解程序的业务意图[63]。

然而，通用 LLM 在处理代码任务时面临着显著的上下文窗口（Context Window）限制。微服务项目通常由数百个源文件组成，直接将整个仓库（Repository）输入模型会超出长度限制或导致“迷失中间（Lost-in-the-Middle）”现象，这使得模型难以捕捉跨文件的隐式依赖。

2.4.2 DeepWiki：仓库级依赖挖掘框架

为了解决大模型在全仓库理解上的瓶颈，DeepWiki 项目提出了一种基于 LLM 的仓库级依赖挖掘新范式。DeepWiki 的核心原理是将代码仓库视为一个巨大的知识库，通过“静态索引 + 动态检索”的方式实现对跨文件依赖的精准捕获[46]。

DeepWiki 的技术路径主要包含两个关键步骤：

1. 语义索引构建：利用静态分析工具提取代码中的实体（如类、方法、变量）及其拓扑关系（如继承、调用、引用），构建项目级的代码知识图谱（Code Knowledge Graph）。同时，利用代码大模型生成每个实体的语义摘要（Summary），将其向量化后存入向量数据库。

2. 上下文感知检索：当需要分析某个微服务的对外调用时，DeepWiki 不再输入全部代码，而是根据当前分析的焦点（如一个 Controller 类），在知识图谱中游走检索其关联的依赖节点（如 Service 接口及其实现类）。

这种机制使得 LLM 能够在有限的上下文窗口内，获取到跨越多个文件的完整调用链路信息，从而准确识别出微服务间隐藏的远程调用逻辑。

2.4.3 思维链推理与结构化生成

在获取了完整的代码上下文后，利用思维链（Chain-of-Thought, CoT）技术可以进一步增强模型的推理鲁棒性。CoT 引导模型在生成最终的安全策略之前，显式地生成中间推理步骤（例如：“首先定位到 `OrderController`，发现其调用了 `PaymentService` 接口，经 DeepWiki 检索该接口实现类绑定了 `/api/pay` 路径”）。这种分步推理机制显著提高了策略生成的准确率，并缓解了深度学习模型的“黑盒”不可解释问题[47]。

2.5 本章小结

本章对支撑零信任微隔离策略生成系统的关键技术进行了梳理。首先，分析了容器与 Kubernetes 编排机制，指出原生网络策略在动态环境下的局限性，并确立了基于 Informer 的实时数据采集方案。其次，探讨了 Soot 静态分析框架，利用 Jimple 中间表示与类结构分析能力，为从字节码中提取确定的服务依赖提供了理论支撑。最后，介绍了

大语言模型及 DeepWiki 框架，论证了其在跨文件依赖挖掘与策略语义增强方面的优势。上述技术共同构成了本文动静态协同分析框架的工程基础。

3 动静态分析的零信任网络策略生成系统

3.1 运行时数据采集

构建精确的零信任微隔离策略，首要前提是建立对云原生集群运行状态的全景感知。传统的静态分析技术往往因缺乏运行时上下文（Runtime Context），难以解析配置文件中的占位符或动态绑定的服务地址，导致生成的依赖关系图存在断裂或误判。为此，本系统设计并实现了一个轻量级的运行时采集探针（Collector），作为独立服务部署于 Kubernetes 集群中。

采集探针的设计遵循“无侵入”与“低开销”原则，负责收集以下四类关键数据，为后续的动静态协同分析提供确定性的环境输入：

1. 编排元数据：包括 Pod、Service 等 Kubernetes 原生资源对象的生命周期状态与标签信息，用于构建网络层面的拓扑基座；

2. 容器运行时信息：包括容器进程的环境变量、启动参数及文件系统挂载点，用于还原应用启动时的配置上下文；

3. 应用资产（Artifacts）：指运行态容器内的 Java 字节码文件（JAR/WAR），作为静态依赖提取的直接输入源；

4. 服务注册信息：包括 Nacos、Eureka 等注册中心维护的服务实例列表，用于解析微服务架构中的逻辑调用寻址。

本节将详细阐述上述数据的采集机制及其工程实现。

3.1.1 Kubernetes 资源

Kubernetes 集群中的资源对象处于高频变更状态，Pod 的扩缩容、故障迁移（Failover）以及 Service 定义的更新都会直接影响微隔离策略的有效性。为了实现对集群资源拓扑的毫秒级感知，本系统摒弃了高延迟的 API 轮询（Polling）模式，转而采用基于 Kubernetes 官方客户端库 `client-go` 的 **SharedInformer** 事件驱动机制[64]。

3.1.1.1 基于 List-Watch 的增量同步机制

采集器内部维护了一个针对核心资源的本地缓存（Local Cache）。在启动初期，采集器通过 List 接口从 API Server 拉取全量资源快照，建立初始索引。随后，通过建立长连接并调用 Watch 接口，实时监听资源对象的增量变更事件（Add、Update、Delete）。

具体而言，底层的 Reflector 组件负责将 API Server 推送的二进制流反序列化为资源对象，并存入 DeltaFIFO 增量队列中。采集器的主控制循环（Controller Loop）从队列中消费事件，更新本地的 Indexer 索引，并触发回调函数将变更数据结构化存储。这种机制确保了采集器与 API Server 之间的数据一致性，同时将对控制平面的网络压力降至最低[65]。

3.1.1.2 关键资源对象与采集维度

为了支撑后续的静态分析与策略生成，系统重点采集以下几类 Kubernetes 资源对象：

- Pod：主要的采集对象是 metadata.labels（业务标签）、status.podIP（实际 IP）以及 spec.nodeName（所在节点）。其中，标签是微隔离策略 podSelector 的直接映射依据，必须确保实时准确。

- Service：主要的采集对象是 spec.selector（后端 Pod 选择器）与 spec.clusterIP（虚拟 IP）。这对于解析代码中硬编码的 ClusterIP 或通过 DNS 发现的服务名称至关重要。

- Endpoints/EndpointSlice：主要的采集对象是 Service 对应的实际后端 IP 列表。在某些未定义 Selector 的特殊服务（如外部数据库映射）场景下，Endpoints 是确定流量目标的唯一依据。

- ConfigMap 与 Secret：重点采集被业务 Pod 挂载的配置资源。Java 应用通常将数据库连接串、注册中心地址等敏感信息存储于这些对象中。采集器通过解析 Pod 的 spec.volumes 字段建立关联，获取配置文件的真实内容，从而在静态分析阶段将代码中的配置占位符（如 ${DB_URL}）替换为真实值。

- Namespace：主要的采集对象是命名空间的名称与标签，用于生成跨命名空间的隔离规则（`namespaceSelector`）。

所有采集到的资源数据被封装为统一的 JSON 结构，并附带采集时间戳与集群标识（Cluster ID），通过消息队列异步推送至后端分析引擎，为构建全集群的服务依赖拓扑图提供元数据支撑。

3.1.2 容器资源

容器作为微服务代码的运行时载体，其文件系统中往往冗余了大量与核心业务无关的基础设施组件（如 Shell 工具、包管理器、调试工具等）。若直接对静态镜像进行全量分析，不仅会引入巨大的计算开销，更会导致微隔离策略生成算法产生大量误报（例如误将调试工具的网络行为纳入业务白名单）。

借鉴基于运行时分析的沙箱挖掘思想[25]，本系统设计了一套“活跃资产锁定（Active Artifact Locking）”机制。该机制通过深度交互容器运行时接口（CRI）与 Linux 内核数据结构，精准识别并提取“积极参与业务逻辑”的进程与文件，从而收敛分析边界。

3.1.2.1 基于 CRI 的容器定位与元数据映射

在 Kubernetes 的动态调度环境下，容器的生命周期与宿主机解耦。采集器首先通过 gRPC 协议连接节点上的 CRI Socket（如 `/run/containerd/containerd.sock`），调用 `ListContainers` 接口获取当前节点所有活跃容器的运行时状态。

系统通过解析 `ContainerStatus` 结构体，提取出容器的唯一标识符（Container ID）、关联的 Pod UID 以及镜像的哈希摘要（ImageRef）。这一步骤建立了“逻辑 Pod ↔ 物理容器”的确定性映射，解决了 Overlay 网络掩盖下真实负载难以定位的问题，为后续进入容器命名空间（Namespace）提供了必要的句柄[66]。

3.1.2.2 基于内存映射的活跃二进制集构建

为了剔除镜像中“存在但未运行”的僵尸文件，本系统利用 Linux 内核的内存管理机制来识别当前活跃的业务代码。具体而言，采集器通过 `setns` 系统调用进入目标容器的 PID Namespace，读取目标进程的 `/proc/<PID>/maps` 内存映射文件。

该文件记录了进程虚拟地址空间中加载的所有可执行文件与动态链接库（Shared Libraries）。系统通过解析这些内存段，构建出当前容器的“活跃二进制集（Active Binary Set）”：

1. 主执行体定位：识别映射具有 r-xp（可读可执行）权限且路径指向文件系统的主程序，过滤掉由 entrypoint.sh 或 tini 启动的父进程包装器；

2. 解释器透视：对于 Java 应用，系统识别出 JVM 进程（如 java 二进制文件），并进一步扫描其打开的文件描述符（/proc/<PID>/fd），从而精准定位到被加载到内存中的 JAR 包或类文件，而非扫描整个 /app 目录；

3. 动态库依赖：记录进程加载的 glibc 等基础库版本，这为后续基于二进制分析提取系统调用（Syscall）特征提供了必要的依赖上下文。

这种基于运行时内存视图的采集方法，能够从根本上排除“镜像内打包了 curl 但业务从未调用”带来的策略噪声，确保后续分析仅聚焦于真实的业务实体。

3.1.2.3 配置上下文与分层文件系统解析

依据云原生应用的配置最佳实践，微服务的关键拓扑信息（如数据库地址、注册中心 URL）通常在启动时通过环境变量注入，而非硬编码于镜像中。

采集器通过读取容器进程的 /proc/<PID>/environ 文件，无侵入地提取全量环境变量键值对。同时，针对容器镜像采用的联合文件系统（UnionFS, 如 OverlayFS），系统解析 /proc/mounts 信息，区分 LowerDir（只读镜像层）与 UpperDir（读写容器层）。这使得分析引擎能够优先处理运行时产生的文件修改（Copy-on-Write），不仅还原了应用启动时的完整配置上下文，也确保了对热更新（Hot-Swap）代码的实时感知[67]。

3.1.3 Java 字节码制品提取

在云原生应用中，业务逻辑被封装在 JAR（Java ARchive）或 WAR 包等二进制制品中。传统的静态分析方法往往采用“全量扫描”策略，即对容器镜像中的所有类库进行无差别分析。然而，生产环境的镜像中往往包含大量冗余的第三方依赖（Bloatware），这些代码在实际运行时从未被加载或者对实际的服务调用没有关联。若将这些非必要代

码纳入分析范围，不仅会引入与业务无关的误报，还会导致后续调用图构建阶段出现严重的路径爆炸（Path Explosion）。

依据本文提出的“动静态结合”理念，本节设计了一种基于运行时内存映射的“活跃依赖锁定（Active Dependency Locking）”机制，旨在精准提取当前微服务实例真正使用的字节码制品。

3.1.3.1 基于内存映射的活跃依赖识别

JVM 的类加载机制（ClassLoader）决定了并非所有 ClassPath 下的 JAR 包都会被加载。为了获取真实的加载视图，采集器利用 Linux 内核提供的进程文件系统（procfs）进行透视。

具体而言，采集器通过读取容器内 Java 主进程的 `/proc/<PID>/maps` 内存映射文件，筛选出所有映射路径以 `.jar` 结尾的内存段。该文件记录了进程虚拟地址空间中所有动态加载的文件描述符。与静态扫描磁盘相比，这种方法的优势在于：

1. 排除冗余：自动忽略存在于磁盘但未被 JVM 加载的“僵尸依赖”（例如未启用的数据库驱动或测试框架）；
2. 解析动态路径：能够识别被自定义 ClassLoader 动态加载到非标准路径下的 JAR 文件；
3. 处理软链接：通过解析文件描述符的真实路径（Real Path），自动处理符号链接导致的路径混淆[25]。

3.1.3.2 无侵入式流式传输通道

为了将锁定的活跃 JAR 包提取至后端分析引擎，系统设计了基于 Kubernetes API 的流式传输通道。采集器不依赖 `docker cp` 等宿主机特权命令，而是与 API Server 建立 HTTP/2 长连接。

通过调用 `PodExec` 接口，采集器在目标容器内执行轻量级的 `tar` 打包命令，将筛选出的活跃 JAR 文件列表打包为二进制流，并通过标准输出（Stdout）管道实时传输。该过程采用了“零落地（Fileless）”策略，数据流直接在内存中转发，避免了在业务容器内创建临时文件，从而消除了对业务磁盘 I/O 的干扰及潜在的残留风险[68]。

3.1.3.3 增量指纹比对与传输优化

在微服务集群中，同一服务的多个副本（Replicas）共享完全相同的代码层，且不同服务之间往往存在公共的基础架构组件（如 Spring Cloud 全家桶）。全量传输所有 JAR 包将产生巨大的网络带宽开销。

为此，系统引入了基于 SHA-256 的增量指纹比对机制。在传输前，采集器在容器内计算目标文件的哈希摘要（Digest），并利用布隆过滤器（Bloom Filter）快速判断该哈希值是否已存在于分析引擎的制品库中。仅当检测到全新的哈希值（对应版本更新或新引入的依赖）时，才触发实际的数据传输。

3.1.4 注册中心配置

在云原生微服务架构中，服务间的通信不再依赖静态 IP，而是通过服务注册与发现机制实现动态寻址。静态代码分析虽然能够提取出微服务间的逻辑调用关系（例如代码中硬编码的 `"http://payment-service/pay"`），但无法直接推导出 Kubernetes NetworkPolicy 所需的确切网络五元组（IP Block 或 Pod Selector）。为了建立“逻辑服务标识”与“物理网络实体”之间的映射视图，本系统设计了针对服务注册中心（Service Registry）的实时采集模块[8]。

3.1.4.1 异构注册中心适配器设计

鉴于企业技术栈的演进往往导致多种注册中心共存（例如存量应用使用 Eureka，新应用使用 Nacos），采集器采用了适配器模式（Adapter Pattern）构建统一的接入层。该层屏蔽了底层异构协议（如 Nacos 的 gRPC/UDP 推送、Eureka 的 HTTP 长轮询）的差异，向上层分析引擎提供标准化的 `ServiceInstance` 模型。

系统目前已内置支持云原生领域主流的服务发现协议：

- Nacos：作为阿里巴巴开源的云原生服务发现平台，Nacos 支持基于 UDP 的服务变更推送。采集器通过模拟 Nacos Client 行为，订阅指定命名空间下的服务列表变更事件，能够实现毫秒级的实例上下线感知。
- Eureka/Consul：针对基于 HTTP 的注册中心，采集器实现了自适应轮询机制（Adaptive Polling）。通过分析服务变动的历史频率，动态调整拉取间隔（例如在发布窗口

期加密轮询，在稳定期降低频率），在保证数据新鲜度的同时降低控制面负载[69]。

3.1.4.2 逻辑服务名到物理地址的动态映射

采集器将从各注册中心获取的数据聚合为一张全局的“服务寻址映射表（Service Addressing Map）”。该表维护了 `Service Name` → `List<Instance IP, Port, Metadata>` 的实时对应关系。

在后续的策略生成阶段，该映射表发挥着关键的“桥梁”作用：例如当静态分析引擎识别到 Java 代码中发起对 `user-service` 的 OpenFeign 调用时，系统查询该表即可锁定目标服务在当前集群中所有活跃实例的 IP 集合。

3.2 字节码静态分析

在完成运行时的制品采集后，分析引擎获得了去重后的业务 JAR 包。然而，现代微服务应用通常包含大量的第三方依赖库（Third-party Libraries），这些库的代码量往往是业务逻辑代码的数十倍。若直接对全量字节码构建调用图，不仅计算开销巨大，且容易引入与业务无关的调用路径（如日志框架内部的网络请求）。

为此，本节设计了一个“三级过滤漏斗”模型，依次执行第三方依赖剔除、非业务类过滤及核心调用链提取，确保静态分析仅聚焦于能够产生服务间交互的关键代码片段。

3.2.1 第三方依赖包过滤

为了解决 Java 应用普遍存在的代码膨胀（Bloatware）问题，系统首先在 JAR 包粒度进行粗筛。该过程基于软件成分分析（SCA）思想，构建了一个包含 Maven Central 主流组件的“指纹黑名单库”[70]。

具体算法流程如下：

1. 指纹计算：对输入的 JAR 包计算 SHA-256 哈希值；

2. 黑名单匹配：将哈希值与指纹库进行 $O(1)$ 匹配。指纹库预置了 Spring Boot Starter、Netty、Jackson 等基础设施组件的哈希特征。

3. 包名启发式过滤：对于未命中的 JAR 包（如企业私有二方库），进一步检查其包名（Package Name）。若包名匹配 `org.apache.*`、`org.springframework.*` 或 `io.kubernetes.*` 等标准前缀，且不包含特定业务关键词，则将其标记为非业务依赖进行剔除。

通过该步骤，分析引擎能够过滤掉约 90% 的基础设施代码，显著降低了后续 Soot 分析的内存压力。

3.2.2 自定义类的启发式过滤

在保留下来的业务 JAR 包中，仍存在大量不参与远程调用的数据对象（POJO）、工具类（Utils）或配置类（Configuration）。为了进一步收敛分析范围，系统利用轻量级的 ASM 字节码操作框架，在不加载完整类结构的情况下扫描常量池（Constant Pool），执行启发式过滤[71]。

系统仅保留满足以下特征之一的“活跃类”：

- 入口特征：包含 `@Controller`、`@RestController` 或 `@Schduled` 等注解的类，这通常是服务调用的起点；
- 出口特征：包含 `@FeignClient`、`@DubboReference` 等注解的接口或字段，这代表了明确的远程调用意图；
- 行为特征：常量池中包含 `"http://"`、`"https://"` 或常见服务名前缀字符串的类，提示可能存在编程式的网络请求。

3.2.3 基于类的调用信息提取

经过前两级过滤后，剩余的类被加载至 Soot 框架中生成 Jimple 中间表示。本节针对云原生环境中最常见的两种调用模式，设计了专用的提取器（Extractor）。

3.2.3.1 声明式调用提取（OpenFeign/Dubbo）

对于基于接口代理的声明式调用，分析引擎重点解析接口上的元数据注解。以 OpenFeign 为例，系统扫描所有带有 `@FeignClient` 注解的接口，提取 `name` 或 `value` 属性作为

目标服务标识，提取 path 属性作为基础路径。同时，解析接口方法上的 @RequestMapping 注解，获取具体的 HTTP 动词与子路径。这些信息组合后，构成了确定的 <Caller, CalleeService, Path> 三元组。

3.2.3.2 编程式调用提取（RestTemplate/WebClient）

对于直接使用 HTTP 客户端的编程式调用，系统采用基于 Jimple 的静态污点分析（Static Taint Analysis）技术。

1. Source 识别：将 URL 字符串常量或配置项标记为污点源；
2. Sink 识别：将 RestTemplate.getForObject() 等网络请求方法标记为污点汇聚点；
3. 数据流追踪：利用 Soot 的 SmartLocalDefs 分析器，逆向追踪网络请求方法的 URL 参数定义点。如果 URL 是由字符串拼接而成（如 "http://" + serviceName + "/api"），系统通过常量传播分析尝试还原其静态部分，并将动态部分标记为需在运行时解析的变量[72]。

这一机制使得系统能够从非结构化的代码中还原出隐含的服务依赖关系。

3.3 调用关系图构建

经过字节码静态提取与运行时配置解析，系统已获取了大量离散的调用链路片段（Call Segments）。为了推导微服务间的全局依赖拓扑，本节提出了一种两阶段图构建算法：首先构建细粒度的类级调用图（Class-level Call Graph），以此为基础，结合微服务部署元数据，聚合生成粗粒度的微服务依赖图（Microservice Dependency Graph）。

3.3.1 基于类的调用关系图

在 Spring Cloud 等现代微服务框架中，服务间的调用往往被封装在复杂的对象关系之中。为了还原这些隐式联系，分析引擎构建了一个有向图 $G_{class} = (V_c, E_c)$，其中节点 V_c 代表业务类（Class），边 E_c 代表类与类之间的关联关系。

不同于传统仅基于函数调用的构图方法，本文针对云原生应用的开发范式，设计了以下三种特定类型的边生成规则，以解决依赖注入（DI）与面向切面编程（AOP）带来的断链问题：

3.3.1.1 基于成员变量注入的关联边

在 Spring 容器中，服务类通常通过 `@Autowired` 或 `@Resource` 注解将其他组件（如 `FeignClient` 接口）注入为成员变量。Soot 的类结构分析能力使得系统能够扫描所有字段定义。若类 A 包含一个类型为 B 的成员变量，且该变量带有注入注解，则建立一条 $A \rightarrow B$ 的“持有边（Has-A）”。这种边揭示了潜在的调用能力，即使代码中未显式出现 `new` 关键字[73]。

3.3.1.2 基于继承与实现的泛化边

为了处理多态调用，系统利用 Soot 的类层次分析（CHA）算法处理继承关系。若类 A 继承自父类 B 或实现了接口 I，则建立 $A \rightarrow B$ 或 $A \rightarrow I$ 的**“泛化边（Is-A）”**。在后续的路径搜索中，若遇到针对接口 I 的调用，算法将自动扩展至其所有具体实现类 A，从而覆盖多态场景下的动态分派路径。

3.3.1.3 基于切面的增强边

微服务架构广泛使用 AOP 技术实现熔断、限流或日志记录。切面类（Aspect）通常通过 `@Before`、`@Around` 等注解绑定到特定的目标方法上，导致控制流在运行时发生隐式跳转。系统通过解析切面表达式（Pointcut Expression），识别出切面类 A 与被增强类 B 之间的绑定关系，建立 $A \rightarrow B$ 的“增强边（Advise-A）”。这确保了在分析调用链时，能够正确穿越由中间件引入的代理逻辑。

3.3.2 基于微服务的调用关系图

类级调用图虽然精确，但其粒度过细，无法直接用于 Kubernetes NetworkPolicy 的生成。为此，系统执行图聚合操作，构建以微服务为节点的有向图 $G_{service} = (V_s, E_s)$。

3.3.2.1 节点映射与聚合

首先，利用 3.1 节采集的 Kubernetes Pod 标签信息，建立“类 $\rightarrow$ 微服务”的归属映射函数 $M(c) = s$。对于 G_{class} 中的每一个节点 $c \in V_c$，将其映射到对应的微服务标识 s。所有映射到同一 s 的类节点被合并为一个微服务节点 $v_s \in V_s$。

3.3.2.2 跨服务边的解析与生成

遍历 G_{class} 中的所有边 (u, v)，若 $M(u) \neq M(v)$，则表明存在潜在的跨服务调用。此时，分析引擎需结合 3.1.4 节采集的注册中心数据进行地址解析：

- 声明式调用解析：若节点 v 是一个被标记为 `@FeignClient("order-service")` 的接口，系统查询服务寻址映射表，找到 `order-service` 对应的微服务节点 v_{target}，并在 $M(u)$ 与 v_{target} 之间添加一条有向边。
- 编程式调用解析:若节点 u 中包含指向 `"http://inventory-service"` 的 `RestTemplate` 调用（经 3.2.3 节污点分析提取），系统同样通过服务发现机制解析目标服务，并在图中建立依赖关系。

最终生成的 $G_{service}$ 清晰描述了集群内所有微服务之间的“谁调用谁”的拓扑结构，且每一条边都附带了具体的调用端口与协议元数据，为后续生成最小权限的 ACL 规则提供了完备的决策依据[10]。

3.4 网络策略生成

微服务依赖图 $G_{service}$ 描述了系统内部合法的通信链路。为了将这些抽象的拓扑关系转化为 Kubernetes 集群可执行的访问控制规则，本节设计了一套自动化的策略生成算法。该算法遵循零信任架构的“最小权限（Least Privilege）”原则，通过标签映射、白名单合成及冲突消解三个步骤，生成确定性的 NetworkPolicy 资源对象。

3.4.1 Pod 标签与服务拓扑映射机制

Kubernetes NetworkPolicy 的核心机制是基于标签选择器（Label Selector）来界定策略的作用域（Target）与对端（Peer）。因此，策略生成的首要任务是将 $G_{service}$ 中的节点

v_s 映射回集群运行时的 Pod 标签集合。

利用 3.1 节采集的 Kubernetes 编排元数据，系统建立了一个双向索引表：

$$Map : ServiceName \rightarrow \{\langle Key, Value\rangle\}_{labels} \tag{3-1}$$

映射逻辑如下：

1. 服务发现映射：对于通过 Service 暴露的微服务，分析系统会解析 Service 定义中的 `spec.selector` 字段，将其作为该服务对应 Pod 的标识标签。
2. 无头服务映射：对于 Headless Service 或未定义 Selector 的外部服务，系统通过解析 EndpointSlice 资源，获取后端 Pod 的实际 IP，若无法关联标签，则生成基于 `ipBlock` 的 CIDR 规则。
3. 命名空间隔离：在映射过程中，严格保留命名空间（Namespace）上下文。若依赖图中存在跨命名空间的调用边，系统将在生成的规则中自动注入 `namespaceSelector`，以确保多租户环境下的隔离性[74]。

3.4.2 最小化白名单生成

基于“永不信任，始终验证”的零信任理念，本系统采用“默认拒绝 + 显式允许”的策略生成模式。对于依赖图中的每一个微服务节点 v，生成策略流程如下：

3.4.2.1 基线策略构建

首先为目标工作负载生成一条“隔离基线（Isolation Baseline）”策略。该策略将 `policyTypes` 设置为 `["Ingress", "Egress"]`，但不包含任何允许规则。一旦应用该策略，Kubernetes CNI 插件将立即阻断进出该 Pod 的所有非授权流量，从而收敛攻击面。

3.4.2.2 白名单规则合成

遍历图 $G_{service}$ 中与节点 v 相关联的所有边，生成细粒度的放行规则：

- 入站规则（Ingress）：对于所有指向 v 的入边 $(u, v) \in E_s$，在 v 的策略中添加一条 Ingress 规则，将来源设置为 u 的标签选择器（`from: podSelector`），并严格限定目标端口（`ports`）为 v 的监听端口。

- 出站规则（Egress）：对于所有由 v 发出的出边 $(v, w) \in E_s$，在 v 的策略中添加一条 Egress 规则，将目标设置为 w 的标签选择器或 IP 段（`to: podSelector/ipBlock`）。
- 基础设施工单：自动注入对集群 DNS 服务（UDP 53 端口）的放行规则，保障微服务的基础解析能力。

3.4.3 策略冗余消除与冲突检测

在大规模集群中，自动化生成的策略可能与运维人员手动配置的规则产生重叠或冲突。为了保证策略集的有效性与性能，系统在下发前执行基于集合论的优化算法。

3.4.3.1 CIDR 聚合与冗余消除

对于基于 IP 地址的 `ipBlock` 规则，系统采用区间合并算法（Interval Merging）对重叠的 CIDR 网段进行聚合。例如，规则 R_1 : 192.168.1.0/25 与 R_2 : 192.168.1.128/25 将被合并为 192.168.1.0/24。这不仅减少了底层 iptables/eBPF 的条目数量，也降低了策略匹配的计算开销。

3.4.3.2 逻辑冲突检测

系统定义了策略间的遮蔽（Shadowing）与泛化（Generalization）关系。在生成新策略前，系统会与集群现存策略进行集合包含关系检查：

$$Conflict(R_{new}, R_{exist}) \iff (Action_{new} \neq Action_{exist}) \wedge (Scope_{new} \cap Scope_{exist} \neq \emptyset) \tag{3-2}$$

若检测到新生成的“拒绝”规则被现有的“宽泛允许”规则（如 `allow-all`）所遮蔽，系统将生成告警报告，提示管理员存在潜在的安全敞口，而非盲目覆盖，从而确保了生产环境的稳定性[75]。。

3.5 本章小结

本章构建了基于动静态协同分析的零信任微隔离策略生成底座。首先，设计了基于 SharedInformer 与 CRI 的运行时探针，实现了对 Kubernetes 元数据、容器进程上下文及业务字节码的无侵入采集；其次，利用指纹去重与 Soot 静态分析框架，精准提取了微服

务间的远程调用链路，并结合依赖注入与切面特征构建了服务级依赖拓扑；最后，遵循零信任最小权限原则，建立了从服务拓扑到 NetworkPolicy 的自动化映射与冲突消解机制，为系统提供了确定性的安全规则保障。

4 基于大模型代码分析的零信任微隔离策略双向智能生成

4.1 统一的多源信息采集与代码仓库关联机制

在前一章中，系统通过运行时探针获取了容器内的二进制制品与网络状态，解决了“由谁访问谁”的链路发现问题。然而，二进制字节码丢失了大量的语义信息（如注释、文档、变量命名意图），且无法直接被通用大语言模型（LLM）高效理解。

为了赋予 LLM 对业务逻辑的深度认知能力，本节提出了一种跨越“运行时”与“开发态”的溯源机制。系统不再局限于容器内部，而是利用大模型，将线上的 Pod 实例映射回代码仓库（Git Repository），为后续的智能分析构建完备的语义上下文。

4.1.1 运行时通用指纹特征的采集

为了在异构的多语言环境中准确识别服务归属，系统首先需要从容器运行时中提取具有普适性的“特征指纹”。这一过程的核心在于获取容器内部的进程执行上下文，而非依赖静态的镜像元数据。

采集模块主要关注以下三类关键运行时数据：

- **关键环境变量**：环境变量是云原生应用配置注入的主要载体。系统重点提取包含业务语义的变量，例如服务名称（`APP_NAME`）、应用 ID（`APP_ID`）或配置中心地址。这些变量通常在部署清单（Deployment YAML）中定义，是识别服务身份的强特征。

- **启动命令行**：解析容器主进程（PID 1）的完整启动命令（Command）及其参数（Args）。无论是 Java 的 `-jar` 参数、Python 的入口脚本路径，还是 Go 二进制文件的名称，都直接暴露了程序的入口点信息。

- **活跃进程列表**：扫描容器内当前运行的所有进程树信息，用于辅助判断容器的实际运行状态和技术栈类型（例如是否存在 `node`、`python` 或 `java` 进程）。

在数据采集的工程实现上，针对不同的镜像类型，本系统设计了自适应的采集策略：

对于包含基础 Shell 工具的标准镜像（如基于 Alpine 或 Ubuntu 构建的镜像），系统直接通过 Kubernetes 的 exec 接口进入容器命名空间，执行标准系统命令获取上述信息。

然而，为了追求极致的安全与轻量化，生产环境中大量采用了 Distroless 或 Scratch 等“极简镜像”。这类镜像移除了 Shell、ls、ps 等所有基础工具，导致常规的 exec 命令无法执行。针对这一难题，系统采用了基于 ** 临时容器（Ephemeral Container）** 的调试机制。通过 Kubernetes 的 debug 功能，系统动态地将一个包含完整调试工具集（如 BusyBox）的临时镜像注入到目标 Pod 中，并配置其共享目标容器的进程命名空间（Process Namespace）。利用这种机制，采集探针可以在不停止业务服务、不修改原有镜像的前提下，通过临时容器“透视”目标容器内的环境变量与进程状态，从而实现了全场景的无盲区采集[50,76]。

4.1.2 基于大模型的仓库推断

在获取了容器运行时的通用指纹特征（如镜像名称、启动命令、环境变量）后，系统采用了一种直观的推断机制来锁定其对应的公共代码仓库。由于云原生生态中大量采用了开源组件（如 Redis、Nginx、或 Google Microservices Demo 等），这些项目的源代码托管在 GitHub 等公共平台上。该机制的核心思想是利用大语言模型（LLM）庞大的预训练知识库，直接识别出运行时指纹所属的开源项目，并输出其 Git 地址。

整个推断过程分为“信息组装”与“知识检索”两个步骤：

首先是提示词的组装。系统会将采集到的碎片化运行时信息填入预定义的 Prompt 模板中。模板的作用是将非结构化的监控数据转化为模型可理解的自然语言描述。例如，模板会将镜像名 gcr.io/google-samples/microservices-demo/emailservice 和环境变量 PORT=8080 组合成一段提示词

其次是仓库地址的生成。将组装好的 Prompt 输入给大语言模型后，模型利用其在训练阶段阅览过的海量开源代码知识，直接检索出该特征对应的公共项目。系统要求模型只做一件事：根据提供的特征，识别出这是哪个开源软件，并直接输出其标准的 HTTPS 克隆地址（例如 https://github.com/GoogleCloudPlatform/microservices-demo）。

这种方法本质上是将大模型作为一个覆盖了全球开源生态的智能索引库使用。参考了 Chen 等人关于 Codex 模型记忆能力的研究，大模型能够准确关联知名的开源项目及其元数据，从而在无需人工介入的情况下，快速建立从运行时容器到公共代码仓库的映

射关系[48]。

4.1.3 基于 DeepWiki 的配置文件定位与采集

在通过大模型推理锁定目标代码仓库后，系统面临的下一个问题是：如何从庞大的代码库中精准定位到决定网络行为的配置文件。微服务应用通常将数据库连接串、服务调用地址等关键信息存储在 `application.yaml`、`nginx.conf` 或 `config.properties` 等文件中。然而，这些文件在仓库中的位置（如 `src/main/resources`）与它们在容器运行时中的实际路径（如 `/app/config`）往往不一致。

为了解决这一“路径映射”难题，系统引入了 DeepWiki 技术。DeepWiki 是一种专门面向代码仓库的依赖理解模型，它不仅能够解析单一的代码文件，还能理解 `Dockerfile` 或构建脚本中的文件复制（COPY）逻辑[46]。

配置文件信息的采集流程分为以下两个步骤：

首先是候选路径的静态推导。系统调用 DeepWiki 引擎对上一节识别出的代码仓库进行扫描。引擎重点关注两类文件：一是显式的配置文件，二是定义构建过程的 `Dockerfile`。DeepWiki 通过思维链推理，分析出配置文件从源码目录移动到镜像目录的轨迹。例如，它能识别出 `COPY target/classes/application.yml /opt/app/config/` 这一指令，从而推导出容器内的候选配置路径为 `/opt/app/config/application.yml`。最终，DeepWiki 会输出一个包含多个可能性的“候选路径列表”。

其次是运行时内容的实地采集。获得候选路径后，系统利用 4.1.1 节中介绍的运行时探针（直接 Exec 或通过临时容器），尝试在目标 Pod 中读取这些路径下的文件内容。这一过程起到了“验证”的作用：只有当文件在运行时真实存在且可读时，其内容才会被采集并回传。采集到的配置文本将被送入后续的分析模块，结合 KubeGuard 提出的配置分析方法，用于提取服务间隐式的网络调用依赖[15]。

通过这种“静态推导路径 + 动态验证读取”的方法，系统有效地解决了单纯静态分析无法获取运行时配置、单纯动态分析无法定位文件路径的矛盾。

4.2 基于代码上下文的策略逆向解释机制

现有的Kubernetes NetworkPolicy通常以YAML格式存储,仅包含标签选择器(Label Selector)和端口号等元数据。对于运维人员而言，像 `allow port 8080` 这样的规则缺乏业务语义，很难直观判断其合理性。为了解决这一问题，本节提出了一种逆向解释流程，通过将“运行时状态”完整地输入给 DeepWiki 引擎，让其从代码层面推导出策略的 ** 调用意图 ** 与 ** 支撑证据 **，从而生成可读性强的自然语言解释。

4.2.1 策略与工作负载信息的级联采集

解释流程的第一步是构建策略与实际运行对象的映射关系。系统首先通过 Kubernetes API 采集集群内所有的 `NetworkPolicy` 资源，提取其 `podSelector`（目标选择器）和 `policyTypes`（策略类型）字段。

随后，系统执行级联采集逻辑：根据标签选择器反向查询出受该策略管控的所有活跃 Pod 实例。对于每一个选中的 Pod，系统复用 4.1 节所述的采集机制，获取其完整的运行时画像，包括环境变量、启动命令行、活跃进程信息以及关联到的公共代码仓库地址。这些信息将作为后续分析的“上下文事实”。

4.2.2 基于 DeepWiki 的意图与证据提取

仅知道代码仓库地址是不够的,因为一个仓库可能包含多个微服务模块(Monorepo),或者同一个镜像可以通过不同的启动命令运行不同的业务逻辑。为了精准解释策略，系统将完整的 Pod 采集信息与代码仓库地址一并输入给 DeepWiki 引擎。

DeepWiki 利用大模型对代码库的深度理解能力，结合传入的运行时参数（如 `cmd: python app.py` 或 `ENV: SERVICE_TYPE=payment`），在代码库中进行精准定位。它主要输出以下两类关键信息：

1. **调用意图（Call Intent）**：DeepWiki 分析代码中的业务逻辑、注释文档或 API 定义（如 Swagger），总结该服务在该端口上的高层业务目标。例如：“该服务负责处理用户信用卡的扣款请求”。

2. **事实证据（Evidence）**：DeepWiki 提取出支撑上述意图的具体代码片段或配置项位

置。例如：“依据 src/payment/routes.py 第 45 行定义的 @app.route('/pay')，以及 Dockerfile 中暴露的 8080 端口配置”[46]。

通过这种方式，DeepWiki 将冷冰冰的运行时参数还原为了有血有肉的代码逻辑。

4.2.3 基于大模型的策略语义生成的解释

最后，系统将“原始网络策略规则”与 DeepWiki 输出的“调用意图”和“事实证据”组装成 Prompt，输入给大语言模型进行最终的文本生成。

大语言模型结合上述信息，生成符合人类阅读习惯的解释。例如，对于一条开放 8080 端口的策略，模型会解释道：“允许外部访问本 Pod 的 8080 端口。** 意图 **：该端口用于处理支付请求；** 证据 **：根据代码库中 routes.py 的定义，该端口承载了核心支付 API，且与运行时启动命令相符。”

这种机制参考了 KubeGuard 的思路，通过引入代码层面的证据链，使安全策略具备了可审计性和可解释性[15]。

4.3 基于代码上下文的策略生成机制

在第 4.2 节中，系统利用 DeepWiki 成功实现了对现有策略的逆向语义解释。沿用这一思路，本节将阐述如何利用同样的机制来 ** 从头生成 ** 新的微隔离策略。其核心逻辑在于：利用 DeepWiki 强大的代码理解能力，深入分析源代码中的网络调用逻辑，并结合 4.1 节采集到的运行时配置数据，将抽象的代码逻辑“实例化”为具体的网络访问目标，从而生成精准的白名单规则。

4.3.1 基于配置数据的目标解析

策略生成的第一步是解析服务“配置了什么”。在 4.1 节中，系统已经采集到了容器运行时的环境变量和核心配置文件内容。然而，这些配置值（如 DB_URL=10.0.1.5）如果脱离了代码上下文，仅仅是无意义的字符串。

本节利用 DeepWiki 引擎将“配置数据”与“代码逻辑”进行关联分析。DeepWiki 扫描代码仓库，寻找读取配置的代码片段。例如，它能定位到 Python 代码中 os.getenv('DB_HOST') 的调用位置，或者 Java 代码中 @Value("${db.url}") 的注入点。

通过将运行时采集到的实际配置值填入这些代码变量中，DeepWiki 能够精准推导出服务在当前环境下的真实连接目标。这种方法不仅解决了硬编码地址的问题，还能完美处理云原生环境中通过 ConfigMap 动态注入后端地址的常见场景，确保了策略生成的准确性。

4.3.2 隐式网络调用逻辑的语义挖掘

除了显式的配置读取外，源代码中还包含大量隐式的网络调用逻辑。DeepWiki 利用大模型对多语言代码的语义理解能力，直接在源码层面挖掘这些行为，而无需依赖任何构建产物或依赖清单。

针对不同的技术栈，DeepWiki 关注通用的网络编程范式：

- **HTTP/RPC 调用识别**：分析代码中构建 HTTP 请求或 gRPC 调用的逻辑。例如，识别 Python 中的 `requests.post(url)` 或 Go 语言中的 `http.NewRequest`。DeepWiki 能够通过静态污点分析（Static Taint Analysis）的思想，追踪 URL 参数的来源，从而确定潜在的下游服务。

- **硬编码地址提取**：虽然不推荐，但部分遗留系统仍存在硬编码 IP 或域名的情况。DeepWiki 能够识别代码字符串常量中的 URL 模式（如 `http://user-service:8080`），将其提取为固定的访问需求。

Du 等人的研究表明，DeepWiki 能够跨越函数调用链，理解复杂的业务逻辑[46]。例如，它能分析出：“代码在 `OrderController` 中接收到请求后，会调用 `InventoryService.check()` 方法，该方法内部通过 REST 接口访问了库存服务”，从而生成一条指向库存服务的允许规则。

4.3.3 基于大模型的策略生成与文档化

经过 DeepWiki 的深度分析，系统获得了一份包含明确业务语义的“调用意图列表”（例如：需要访问 Payment DB，需要调用 User Service）。最后一步，系统将这些意图列表输入给大语言模型，要求其将其转化为 Kubernetes 标准的 `NetworkPolicy` YAML 文件。

大模型在生成策略的同时，会保留 DeepWiki 提供的上下文证据作为注释。生成的策略文件不仅包含技术规则，还具备极高的可读性。例如：

```
- to:
  - podSelector:
      matchLabels:
        app: inventory-service
  ports:
    - port: 8080
  # [Generated by DeepWiki]
  # Intent: Check item availability during order creation
  # Evidence: Found 'requests.get' in 'order_service.py' line 45
  #           using config 'INVENTORY_URL'
```

这种基于代码上下文的生成方式，完全模拟了资深开发人员“阅读代码写文档”的过程，从根本上避免了流量学习方法中常见的“冷路径遗漏”问题，确保了生成的策略既符合最小权限原则，又覆盖了所有合法的业务逻辑[15]。

4.4 本章小结

本章详细论述了基于代码上下文的零信任微隔离策略生成技术。针对云原生环境下的语义缺失难题，本章首先利用大模型推理运行时指纹，建立了运行时容器与静态代码仓库的精准关联；进而通过 DeepWiki 引擎挖掘代码中的业务意图，实现了网络策略的逆向语义解释；最后提出了基于代码逻辑与配置解析的策略生成算法，通过将静态代码分析与动态配置数据相结合，在不依赖流量基线的情况下，生成了兼顾最小权限原则与长尾业务覆盖的自适应微隔离策略，有效解决了现有技术在准确性与可解释性方面的不足。

5 实验与分析

本章对本文提出的零信任微隔离策略自动生成框架进行全面实验验证与性能评估。框架的核心技术包括 Kubernetes 事件监听、Soot 静态代码分析以及 DeepWiki 代码分析，最终解释或者生成 Kubernetes NetworkPolicy 策略。

5.1 实验环境搭建

实验在两个集群 Kubernetes 环境中进行，并确保 NetworkPolicy 功能已启用并正确实施。

软件环境包括：Kubernetes v1.23, Calico 网络插件, Soot 静态分析工具（用于代码依赖提取）, DeepWiki 知识图谱系统（用于语义增强）, LLM 模型：GPT-4o 和 DeepSeek（用于策略校验），

硬件配置如下：

表 5.1 双集群实验环境硬件配置

配置项	集群 1 (master1)	集群 2 (master2)
主机名称	master1	master1
CPU	16 核心	8 vCPUs
内存	30 GB	32 GB
硬盘	200 GB	1.024 TB
主要 IP	192.168.4.143	192.168.4.135
虚拟化平台	VMware Workstation	VMware Workstation
操作系统	CentOS 7	CentOS 7

5.2 测试数据集与服务规模

实验选取 14 个典型云原生大数据与 AI 基础设施组件，覆盖工作流调度、数据湖、向量数据库、分布式数据库、观测性等生产场景，以充分模拟真实生产环境下的复杂交

互场景。具体组件及框架自动生成的微隔离策略数量如表 5.2 所示。

表 5.2 测试应用系统及自动生成的 NetworkPolicy 数量统计

序号	应用系统	运行的服务数量
1	Airflow	5
2	Devlake	7
3	Solr	3
4	Skywalking	5
5	Mantis	5
6	Nebula	4
7	TiDB	8
8	Druid	10
9	Pachyderm	9
10	Pinot	5
11	JupyterHub	7
12	Milvus	7
13	Vitess	7
14	Sentry	58
总计		140

其中通过提前掌握的配置文件信息生成 140 条对应的正确的 network policy，在人为在原有正确策略的基础上通过修改策略信息生成 140 条对应的错误的 network policy。最终形成 280 条正反样本数据集（其中 140 条正确策略，140 条错误策略），所有策略的预期行为（Ground Truth）均经过人工复核。

5.3 评价指标

采用以下二分类指标评估大语言模型对 NetworkPolicy 正确性的判断能力：

- 真阳性率（True Positive Rate, TP）：正确识别“合规”策略的比例

- 假阳性率（False Positive Rate, FP）：将“不合规”策略误判为“合规”的比例

5.4 实验结果

本实验分别对 GPT-4o 与 DeepSeek Reasoner (R1) 两个前沿大语言模型进行测试。输入为 NetworkPolicy YAML 和针对实际运行的 kubernetes 集群采集的信息，输出为对该策略正确性的二元判断及对正确性的源码解释。

5.4.1 性能指标对比

表 5.3展示了两个模型对 140 个 NetworkPolicy 策略进行正反向校验的性能指标。

表 5.3 大语言模型对 NetworkPolicy 正确性校验性能对比

模型	真阳性率 (TP)	假阳性率 (FP)
GPT-4o	98.6%	0.7%
DeepSeek Reasoner (R1)	98.6%	0.0%

5.4.2 结果分析

两种模型在真阳性率上均达到了 98.6% 的水平，表明当前先进大语言模型已经具备对复杂 Kubernetes NetworkPolicy 规则组合的深度理解能力，证明了其在微隔离策略正确性校验任务中的有效性。

对真阴性（True Negative, TN）案例的分析表明，两个模型都能较好地拒绝不合规策略。其中，GPT-4o 的 TN 错误主要集中在放行命令行中声明的端口和 kube-scheduler 的端口被错误放宽的情形；DeepSeek 的 TN 错误则主要出现在命名空间标签识别错误而导致被放行。

在安全领域最为关键的假阳性率指标上，DeepSeek Reasoner 表现出优势，实现 0.0% 的零漏判，而 GPT-4o 仍存在 0.7% 的误放行风险。这意味着 DeepSeek Reasoner 对全部人为植入的错误配置实现了 100% 精准识别，没有发生任何安全漏判；而 GPT-4o 错误放行了约 0.7% 的高危错误策略。

进一步对假阳性案例进行分析发现，GPT-4o 的误判主要出现在以下场景：模型虽然在首次分析的上下文已经捕捉到了正确的端口信息，却仍然错误地放行了错误的端口配置，并将其解释为正确配置。

综上所述，本实验成功验证了先进大语言模型在 Kubernetes NetworkPolicy 正确性判断任务中的卓越能力。DeepSeek Reasoner 以 98.6% 的高真阳性率和 0.0% 的假阳性率，彻底杜绝了安全漏判，优于 GPT-4o，表明其更适合作为零信任微隔离策略自动生成后的最后一道安全校验防线，为本文框架在大规模生产环境的安全落地提供了可靠保障。

参考文献

[1] Cloud Native Computing Foundation. CNCF Annual Report 2024[R]. Annual Report. Available at: https://www.cncf.io/reports/. San Francisco, CA: The Linux Foundation, 2024.

[2] BURNS B, GRANT B, OPPENHEIMER D, et al. Borg, Omega, and Kubernetes[J/OL]. Communications of the ACM, 2016, 59: 50-57. https://api.semanticscholar.org/CorpusID:13246959.

[3] HILS A, KAUR R, D'HOINNE J. Market Guide for Microsegmentation[Z]. Document ID: 4435299. Available at: https://www.gartner.com/en/documents/4435299. 2023.

[4] WARD R, BEYER B. BeyondCorp: A New Approach to Enterprise Security[J/OL]. login Usenix Mag., 2014, 39. https://api.semanticscholar.org/CorpusID:56594397.

[5] ROSE S, BORCHERT O, MITCHELL S, et al. Zero Trust Architecture[C/OL]//. 2019. https://api.semanticscholar.org/CorpusID:212765583.

[6] LI W, LEMIEUX Y, GAO J, et al. Service Mesh: Challenges, State of the Art, and Future Research Opportunities[J/OL]. 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), 2019: 122-1225. https://api.semanticscholar.org/CorpusID:146119227.

[7] CHANDRAMOULI R. Attribute-based Access Control for Microservices-based Applications Using a Service Mesh[C/OL]//NIST Special Publication 800-204B. 2021. https://api.semanticscholar.org/CorpusID:238984004.

[8] JAMSHIDI P, PAHL C, das CHAGAS MENDONÇA N, et al. Microservices: The Journey So Far and Challenges Ahead[J/OL]. IEEE Softw., 2018, 35: 24-35. https://api.semanticscholar.org/CorpusID:25437582.

[9] Isovalent Inc. eBPF-based Networking, Observability, and Security[R]. Whitepaper. Available at: https://isovalent.com/resource-library/. Isovalent, 2023.

[10] ZHOU X, PENG X, XIE T, et al. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study[J/OL]. IEEE Transactions on Software Engineering, 2018, 47: 243-260. https://api.semanticscholar.org/CorpusID:57462883.

[11] CHEN X, ZHANG M, MAO Z M, et al. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions[C/OL]//USENIX Symposium on Operating Systems Design and Implementation. 2008. https://api.semanticscholar.org/CorpusID:7600750.

[12] HE K, KIM D D, ASGHAR M R. Adversarial Machine Learning for Network Intrusion Detection Systems: A Comprehensive Survey[J/OL]. IEEE Communications Surveys & Tutorials, 2023, 25: 538-566. https://api.semanticscholar.org/CorpusID:255718190.

[13] Sysdig Threat Research Team. Sysdig 2024 Cloud-Native Security and Usage Report[R]. Industry Report. Available at: https://sysdig.com/blog/2024-cloud-native-security-and-usage-report/. Sysdig, 2024.

[14] GNUTTI A, VACA F D, FACCHI C. GCN-Based Encrypted Traffic Classification: A Representation Learning Approach[J/OL]. IEEE Transactions on Network and Service Management, 2022, 19(4): 5074-5086. DOI: 10.1109/TNSM.2022.3204369.

[15] COHEN O S, MALUL E, MEIDAN Y, et al. KubeGuard: LLM-Assisted Kubernetes Hardening via Configuration Files and Runtime Logs Analysis[J/OL]. ArXiv, 2025, abs/2509.04191. https://api.semanticscholar.org/CorpusID:281103555.

[16] KIM B, LEE S. KubeAegis: A Unified Security Policy Management Framework for Containerized Environments[J/OL]. IEEE Access, 2024, 12: 160636-160652. https://api.semanticscholar.org/CorpusID:273723001.

[17] SANDHU R S, COYNE E J, FEINSTEIN H L, et al. Role-Based Access Control Models[J/OL]. Computer, 1996, 29: 38-47. https://api.semanticscholar.org/CorpusID:1958270.

[18] FERRAIOLO D F, SANDHU R, GAVRILA S, et al. Proposed NIST Standard for Role-Based Access Control[J/OL]. ACM Transactions on Information and System Security (TISSEC), 2001, 4(3): 224-274. DOI: 10.1145/501978.501980.

[19] ROSTAMI G. Role-based Access Control (RBAC) Authorization in Kubernetes[J/OL]. J. ICT Stand., 2023, 11: 237-260. https://api.semanticscholar.org/CorpusID:261800733.

[20] KUHN D R, COYNE E J, WEIL T R. Adding Attributes to Role-Based Access Control[J/OL]. Computer, 2010, 43: 79-81. https://api.semanticscholar.org/CorpusID:17866775.
[21] Red Hat. The State of Kubernetes Security Report: 2024 Edition[R]. Industry Report. Available at: https://www.redhat.com/en/engage/state-kubernetes-security-report-2024. Red Hat, 2024.
[22] GU Y, TAN X, ZHANG Y, et al. EPScan: Automated Detection of Excessive RBAC Permissions in Kubernetes Applications[J/OL]. 2025 IEEE Symposium on Security and Privacy (SP), 2025: 3199-3217. https://api.semanticscholar.org/CorpusID:274772137.
[23] HU V C, FERRAIOLO D F, KUHN R, et al. Guide to Attribute Based Access Control (ABAC) Definition and Considerations[C/OL]//. 2014. https://api.semanticscholar.org/CorpusID:168659974.
[24] LI X, CHEN Y, LIN Z, et al. Automatic policy generation for inter-service access control of microservices[C]//USENIX Security Symposium. 2021: 3971-3988.
[25] GHAVAMNIA S, PALIT T, BENAMEUR A, et al. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction[C/OL]//International Symposium on Recent Advances in Intrusion Detection. 2020. https://api.semanticscholar.org/CorpusID:220778345.
[26] CERNÝ T, TAIBI D. Microservice-Aware Static Analysis: Opportunities, Gaps, and Advancements [J/OL]. 2022. https://api.semanticscholar.org/CorpusID:266363693.
[27] ABDELFATTAH A S, CORDES K E, MEDINA A, et al. Semantic Dependency in Microservice Architecture[J/OL]. 2025 IEEE/ACM 22nd International Conference on Software and Systems Reuse (ICSR), 2025: 44-54. https://api.semanticscholar.org/CorpusID:275758371.
[28] Tigera. Zero Trust Network: Why It’s Important & Zero Trust for K8s[EB/OL]. 2024 [2025-11-28]. https://www.tigera.io/learn/guides/zero-trust/zero-trust-network/.
[29] RAHAMAN M S, ISLAM A, CERNÝ T, et al. Static-Analysis-Based Solutions to Security Challenges in Cloud-Native Systems: Systematic Mapping Study[J/OL]. Sensors (Basel, Switzerland), 2023, 23. https://api.semanticscholar.org/CorpusID:256647815.
[30] Cybersecurity and Infrastructure Security Agency. Zero Trust Maturity Model (Version 2.0)[R]. Guidance Document. Accessed: 2025-01-10. CISA, 2024.
[31] AccuKnox. Zero Trust Kubernetes What It Is, Benefits & Challenges[EB/OL]. 2024 [2025-11-28]. https://www.accuknox.com/blog/zero-trust-kubernetes.
[32] Palo Alto Networks. The State of Cloud-Native Security 2024[R]. Industry Report. Available at: https://www.paloaltonetworks.com/state-of-cloud-native-security. Palo Alto Networks, 2024.
[33] Axiomatics. The State of Authorization 2022[Z]. Industry Survey Report: The Road to Zero Trust. Available at: https://www.axiomatics.com/resources/the-state-of-authorization-2022/. 2022.
[34] Cloud Security Alliance. Security Guidance for Critical Areas of Focus in Cloud Computing v4.0[R]. Guidance Document. Available at: https://cloudsecurityalliance.org/artifacts/security-guidance-v4. Seattle, WA: Cloud Security Alliance, 2017.
[35] 3GPP. Security architecture and procedures for 5G System (Release 17)[R]. Technical Specification TS 33.501. Available at: https://portal.3gpp.org/. 3rd Generation Partnership Project (3GPP), 2022.
[36] LI X, LENG X, CHEN Y. Securing Serverless Computing: Challenges, Solutions, and Opportunities [J/OL]. IEEE Network, 2021, 37: 166-173. https://api.semanticscholar.org/CorpusID:235195813.
[37] Palo Alto Networks. State of Cloud-Native Security 2023[R]. Industry Report. Available at: https://www.paloaltonetworks.com/state-of-cloud-native-security. Palo Alto Networks, 2023.
[38] Datadog. State of Cloud Security 2024[R]. Industry Report. Available at: https://www.datadoghq.com/state-of-cloud-security/. Datadog, 2024.
[39] CrowdStrike. Architecture Drift: What It Is and How It Leads to Breaches[Z]. CrowdStrike Cybersecurity Blog. Accessed: 2025-02-15. 2024.
[40] ANDRESINI G, PENDLEBURY F, PIERAZZI F, et al. INSOMNIA: Towards Concept-Drift Robustness in Network Intrusion Detection[J/OL]. Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security, 2021. https://api.semanticscholar.org/CorpusID:240001863.
[41] State Administration for Market Regulation of PRC. GB/T 22239-2019: Information Security Technology - Baseline for Classified Protection of Cybersecurity[Z]. National Standard of P.R. China. 2019.
[42] ROSE S, BORCHERT O, MITCHELL S, et al. Zero Trust Architecture[R/OL]. 800-207. National Institute of Standards, 2020. DOI: 10.6028/NIST.SP.800-207.

[43] YARYGINA T, BAGGE A H. Overcoming Security Challenges in Microservice Architectures[J/OL]. 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2018: 11-20. https://api.semanticscholar.org/CorpusID:21718508.
[44] HU V C, FERRAIOLO D, KUHN R, et al. Guide to Attribute Based Access Control (ABAC) Definition and Considerations[R/OL]. 800-162. National Institute of Standards, 2014. DOI: 10.6028/NIST.SP.800-162.
[45] VALLÉE-RAI R, CO P, GAGNON E, et al. Soot - a Java Bytecode Optimization Framework[C//Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. 1999.
[46] DU J, LIU Y, GUO H, et al. DependEval: Benchmarking LLMs for Repository Dependency Understanding[C/OL]//Annual Meeting of the Association for Computational Linguistics. 2025. https://api.semanticscholar.org/CorpusID:276903041.
[47] WEI J, WANG X, SCHUURMANS D, et al. Chain-of-thought prompting elicits reasoning in large language models[C]//Advances in Neural Information Processing Systems (NeurIPS): vol. 35. 2022: 24824-24837.
[48] CHEN M, TWOREK J, JUN H, et al. Evaluating large language models trained on code[J]. arXiv preprint arXiv:2107.03374, 2021.
[49] MERKEL D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux journal, 2014, 2014(239): 2.
[50] KERRISK M. Namespaces in operation, part 1: namespaces overview[J/OL]. LWN. net, 2013, 1. https://lwn.net/Articles/531114/.
[51] MENAGE P. Cgroups[C]//Proceedings of the Linux Symposium: vol. 1. 2007: 27-29.
[52] SULTAN S, AHMAD I, DIMITRIOU T. Container security: Issues, challenges, and the road ahead [J]. IEEE Access, 2019, 7: 52976-52996.
[53] BROWN N. OverlayFS: The Multi-Layer Filesystem[J/OL]. LWN.net, 2013. https://lwn.net/Articles/718131/.
[54] Cloud Native Computing Foundation. Container Network Interface Specification[Z]. https://github.com/containernetworking/cni. Accessed: 2024-03-20. 2023.
[55] BURNS B, GRANT B, OPPENHEIMER D, et al. Borg, omega, and kubernetes[J]. Communications of the ACM, 2016, 59(5): 50-57.
[56] SOUPPAYA M, MORELLO J, SCARFONE K. Application container security guide[R]. NIST Special Publication 800-190. National Institute of Standards, 2017.
[57] SHAMIM M S I, GIBSON J A, MORRISON P, et al. Benefits, Challenges, and Research Topics: A Multi-vocal Literature Review of Kubernetes[J/OL]. arXiv preprint arXiv:2211.07032, 2022. DOI: 10.48550/arXiv.2211.07032.
[58] VALLÉE-RAI R, CO P, GAGNON E, et al. Soot - a Java Bytecode Optimization Framework[C/OL]//Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. 1999: 13. DOI: 10.1145/781995.782008.
[59] LINDHOLM T, YELLIN F, BRACHA G, et al. The Java virtual machine specification[M]. Java SE 8 Edition. Addison-Wesley Professional, 2014.
[60] VALLÉE-RAI R, GAGNON E, HENDREN L, et al. Soot: a Java bytecode optimization framework [C]//CASCON First Decade High Impact Papers. 2010: 214-224.
[61] DEAN J, GROVE D, CHAMBERS C. Optimization of object-oriented programs using static class hierarchy analysis[C]//European Conference on Object-Oriented Programming. 1995: 77-101.
[62] BOMMASANI R, HUDSON D A, ADELI E, et al. On the opportunities and risks of foundation models [R]. arXiv preprint arXiv:2108.07258. Stanford University, 2021.
[63] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is all you need[C]//Advances in Neural Information Processing Systems: vol. 30. 2017: 5998-6008.
[64] DOBIES J, WOOD J. Kubernetes Operators: Automating the Container Orchestration Platform[M]. O'Reilly Media, 2020.
[65] BURNS B, OPPENHEIMER D. Design Patterns for Container-based Distributed Systems[C/OL]//USENIX Workshop on Hot Topics in Cloud Computing. 2016. https://api.semanticscholar.org/CorpusID:9607532.

[66] Cloud Native Computing Foundation. Container Runtime Interface (CRI) - API Definition[Z]. https://github.com/kubernetes/cri-api. Accessed: 2024-03-20. 2023.
[67] Open Container Initiative. Open Container Initiative Runtime Specification[Z]. https://github.com/opencontainers/runtime-spec. v1.1.0. 2023.
[68] KERRISK M. The Linux programming interface: a Linux and UNIX system programming handbook [M]. No Starch Press, 2010.
[69] RICHARDSON C. Microservices patterns: with examples in Java[M]. Manning Publications, 2018.
[70] SOTO-VALERO C, HARRAND N, MONPERRUS M, et al. Comprehensive Analysis of Bloat in Java Applications[C/OL]//Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021: 1069-1081. DOI: 10.1109/ASE51524.2021.9678558.
[71] BRUNETON E, LENGLET R, COUPAYE T. ASM: a code manipulation tool to implement adaptable systems[C/OL]//. 2002. https://api.semanticscholar.org/CorpusID:13274869.
[72] ARZT S, RASTHOFER S, FRITZ C G, et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps[J/OL]. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014. https://api.semanticscholar.org/CorpusID:12083354.
[73] TIP F, PALSBERG J. Scalable propagation-based call graph construction algorithms[C/OL]// Conference on Object-Oriented Programming Systems, Languages, and Applications. 2000. https://api.semanticscholar.org/CorpusID:11748853.
[74] CHANDRAMOULI R. Attribute-based Access Control for Microservices-based Applications using a Service Mesh[R/OL]. NIST Special Publication 800-204B. Available at: https://csrc.nist.gov/pubs/sp/800/204/b/final. Gaithersburg, MD: National Institute of Standards, 2021. DOI: 10.6028/NIST.SP.800-204B.
[75] CUPPENS F, CUPPENS-BOULAHIA N, GARCIA-ALFARO J, et al. Detection of Configuration Vulnerabilities in Distributed Firewalls[J/OL]. IEEE Transactions on Network and Service Management, 2014, 11(3): 318-331. DOI: 10.1109/TNSM.2014.2336473.
[76] Kubernetes Documentation. Debugging with Ephemeral Containers[Z]. https://kubernetes.io/docs/tasks/debug/debug-application/debug-running-pod/. Official guide on using ephemeral containers for distroless debugging. 2024.

附录

A 一个附录

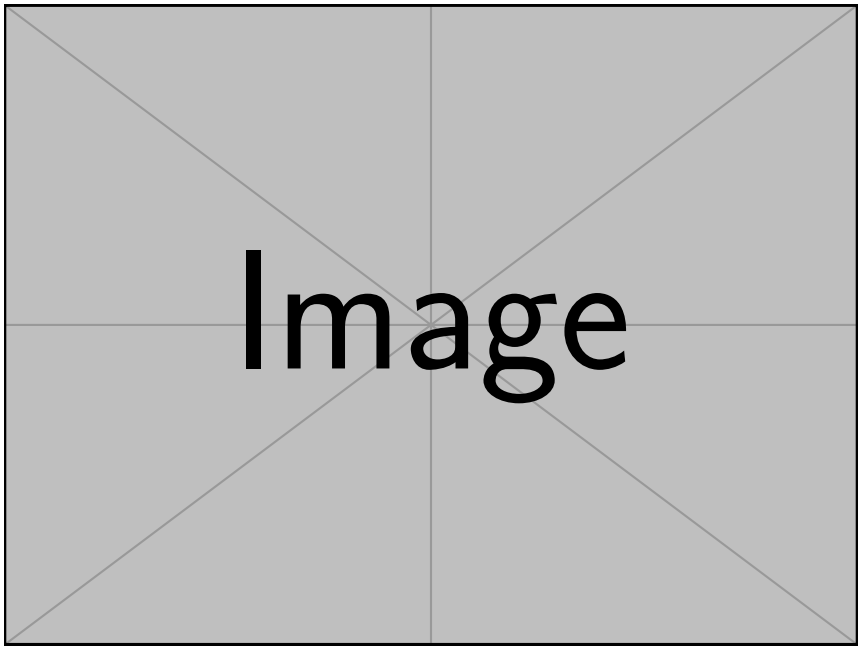

图 A.1 附录中的图片

B 另一个附录

作者简历