西安郵電大學

毕业设计（论文）

题　　目：基于 eBPF 的 ELF 文件加载过程分析及可视化系统设计与实现

学　　院：计算机院

专　　业：网络工程

班　　级：网络 2101

学　　号：04212016

学生姓名：骆祎豪

导师姓名：王亚刚　职称：副教授

起止时间：2024 年 11 月 20 日 至 2025 年 6 月 6 日

年　　月　　日

毕业设计（论文）承诺书

本人所提交的毕业设计（论文）《基于 eBPF 的 ELF 文件加载过程分析及可视化系统设计与实现》是本人在指导教师指导下独立研究、写作的成果，毕业设计（论文）中所引用他人的文献、数据、图件、资料均已明确标注；对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式注明并表示感谢。

本人深知本承诺书的法律责任，违规后果由本人承担。

论文作者签名：骆祎豪　　日　　期：_____

关于毕业设计（论文）使用授权的声明

本人在导师指导下所完成的论文及相关的职务作品，知识产权归属西安邮电大学。本人完全了解西安邮电大学有关保存、使用毕业设计（论文）的规定，同意学校保存或向国家有关部门或机构送交论文的纸质版和电子版，允许论文被查阅和借阅；本人授权西安邮电大学可以将本毕业设计（论文）的全部或部分内容编入有关数据库进行检索，可以采用任何复制手段保存和汇编本毕业设计（论文）。本人离校后发表、使用毕业设计（论文）或与该毕业设计（论文）直接相关的学术论文或成果时，第一署名单位仍然为西安邮电大学。

本毕业设计（论文）研究内容：

□可以公开

□不宜公开，已办理保密申请，解密后适用本授权书。

（请在以上选项内选择其中一项打“√”）

论文作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

西安邮电大学本科毕业设计(论文)选题审批表

申 报 人	王亚刚	职　　称	副教授	学　院	计算机院
题目名称	基于 eBPF 的 ELF 文件加载过程分析及可视化系统设计与实现				
题目来源	☐其他 ☐教师专业实践 ☑教师科研课题				
题目类型	☐硬件设计 ☐艺术作品 ☑软件设计 ☐论文				
题目分类	☑工程实践 ☐社会调查 ☐实习 ☐实验 ☐其他				
题目简述	ELF 文件是软件开发工具链生成的主要文件格式之一,分析操作系统中 ELF 目标文件的加载和执行过程，可以准确提取操作系统对于 ELF 目标文件的使用需求，例如 ELF 文件中需要提供给操作系统的特定信息及其意义，这些信息的提取对于分析软件开发工具链中 ELF 文件的编译和链接、库的组织等有重要意义。在软件开发工具的相关科研项目中，需要对软件工具链中编译、汇编、链接以及相关库进行代码分析，其中有部分信息与最终的 ELF 文件的加载执行是直接相关的，本课题就是要基于 eBPF 工具深入 Linux 内核，观测 ELF 程序加载的过程以及对 ELF 文件信息的使用情况，并对这些信息进行采集、整理以及最终的展示。 本课题基于 eBPF 工具深入 Linux 内核，观测 ELF 程序加载的过程以及对 ELF 文件信息的使用情况，并对这些信息进行采集、整理以及最终的展示，实现 ELF 文件加载过程的可视化，为深入理解 ELF 文件的信息提供帮助，也为前期的软件开发工具链分析提供辅助。				
对学生知识与能力要求	1.熟悉 Linux 操作系统 2.熟悉 eBPF 框架，能够利用 eBPF 程序进行内核监控程序的开发 3.了解 Linux 系统中 ELF 文件的加载过程 4.熟悉一般的 WEB 开发，将最终的观测数据进行可视化展示。				
具体任务以及预期目标	本课题的具体任务包括: 在 Linux 系统上开发基于 eBPF 的 ELF 文件加载监控程序,观测 ELF 程序加载的过程，提取重要过程信息及相关 ELF 文件信息的使用情况，并对这些信息进行采集、整理，最后将这些数据基于 web 进行展示，实现 ELF 文件加载过程的可视化。 具体的预期目标及成果形式包括: 1.Linux 系统中 ELF 加载过程分析，给出分析文档 1 份 2.使用 eBPF 框架，开发一个系统，实现 ELF 文件加载过程中重要环节的信息提取。 3.开发 web 前后端，实现信息的可视化。				

<table>
<tr><td>时间
进度</td><td colspan="3">2024 年 11 月 25 日-11 月 24 日：完成毕业设计选题
2024 年 11 月 25 日-2025 年 1 月 10 日：提交开题报告，前期检查
2025 年 1 月 11 日-3 月 29 日：完成环境搭建，并完成前后台接口设计，中期检查
2025 年 3 月 30 日-5 月 17 日：完成设计实现代码，进行代码验收
2025 年 4 月 1 日-5 月 25 日：撰写毕业论文
2025 年 5 月 26 日 6 月 1 日：完善毕业论文，进行论文答辩。</td></tr>
<tr><td>专业负责人审核意见</td><td colspan="3">经审核，题目符合专业培养目标和教学要求，同意该毕业设计题目申报

签字:　　年　月　日</td></tr>
<tr><td>系（教研室）主任签字</td><td>年　月　日</td><td>主管院长签字</td><td>年　月　日</td></tr>
</table>

西安邮电大学本科毕业设计（论文）开题报告

学生姓名	骆祎豪	学号	04212016	专业班级	网络 2101
指导教师	王亚刚	题目	基于eBPF的ELF文件加载过程分析及可视化系统设计与实现		

选题目的

ELF（Executable and Linkable Format）是Linux/Unix系统下可执行文件、共享库及核心转储的标准格式，其加载过程涉及程序头解析、动态链接、内存映射等复杂机制。传统分析方法（如strace、gdb）存在性能开销大、侵入性强、无法实时跟踪内核态行为等局限性。

eBPF（extended Berkeley Packet Filter）是一种革命性的内核扩展技术，允许用户在内核安全沙盒[1]中运行自定义程序，具备低开销、高实时性的优势，在性能分析、安全监控等领域广泛应用。根据 Martin KaFai Lau 等人的研究，eBPF 在网络监控和性能分析中能够显著降低性能开销，同时提供高精度的实时数据捕获能力[2]。

与此同时，eBPF作为 Linux 内核中的一项创新技术，能够在内核中安全、动态地加载和执行自定义程序，从而实现高效、实时的系统监控。eBPF 的高性能和灵活性使其在网络监控[3]、性能分析、安全检测等领域得到了广泛应用，但其在系统行为分析中的潜力尚未完全挖掘。如 Daniel Borkmann 所述，eBPF 在操作系统内核行为分析和动态跟踪方面具有巨大的应用前景，特别是在系统性能优化和故障排查领域[4]。

本课题旨在利用 eBPF 的实时监测能力，对 ELF 文件加载的全过程进行细粒度分析，包括文件解析、段和节的加载、动态链接过程及加载后的内存布局等。同时，设计并实现一个可视化系统，将这些复杂的系统行为直观呈现，以便开发者和研究者更好地理解和优化程序加载流程。

近年来，eBPF 的技术发展和应用场景不断扩展。根据 Alexei Starovoitov 等人的研究，eBPF 不仅在网络和性能优化领域表现出色，还能够对系统事件进行精准捕获和实时分析[5]。此外，Brendan Gregg 在其著作 BPF Performance Tools 中强调，eBPF 是操作系统性能调优和行为分析的重要工具[6]。在国外，Google 和 Facebook 等科技巨头也在积极探索 eBPF 在大规模集群管理和性能优化中的应用，进一步证明了该技术的实用性和前瞻性[7]。在国内，阿里云和腾讯等企业也开始探索 eBPF 的应用，表明该技术在实践中的重要性[8]。学术界对 eBPF 技术在系统行为分析和性能优化方面的研究兴趣日益浓厚[9]。

本课题结合 ELF 文件加载机制和 eBPF 技术，探索系统行为的透明化和可视化设计。通过实时捕获和分析内核行为[10]，不仅为系统调试、性能优化提供支持，也为科研贡献一个创新性工具。此外，该研究也将推动 eBPF 在操作系统底层分析中

的应用，拓展其技术边界。

参考文献

[1] eBPF 官方社区（ebpf.io），探索eBPF在内核中的架构

[2] Brendan Gregg, BPF Performance Tools, Addison-Wesley, 2019. eBPF 的工作原理及其在性能调优和监控中的应用。

[3] 李明, 基于 eBPF 的网络性能监控系统设计与实现, 计算机工程与应用, 2020. eBPF 在网络性能监控中的实际应用案例。

[4] Daniel Borkmann, eBPF: Enabling Flexible and Efficient Kernel Dynamics, USENIX Annual Technical Conference, 2018. eBPF 在操作系统内核行为分析和动态跟踪方面的应用前景。

[5] Alexei Starovoitov, *eBPF—Rethinking the Linux Kernel*, Linux Plumbers Conference, 2014. eBPF 的核心设计理念和早期实现。

[6] Brendan Gregg, BPF Performance Tools, Addison-Wesley, 2019. eBPF 的工作原理及其在性能调优和监控中的应用。

[7] Google Research, Exploring eBPF in Large-Scale Cluster Management, 2021. eBPF 在大规模集群管理和性能优化中的应用研究。

[8] 阿里云开发者社区, 探索 eBPF:Linux 内核的黑科技（下）, 2021. eBPF 的实现机制及其在 Linux 内核中的应用。

[9] academia.edu, eBPF Technology in System Behavior Analysis, 2022. 学术界对 eBPF 技术在系统行为分析方面的研究。

[10] Cilium Project, BPF and XDP Reference Guide, 2019. eBPF 和 XDP 的概念、架构和应用场景。

前期基础

1. 已学课程

《数据结构》、《计算机组成原理》、《Linux 操作系统》、《计算机网络》、《软件工程》、《数据库原理及运用》、 《Web开发技术》

2. 掌握的工具

Visual Studio Code 开发工具、 Git（分布式版本控制系统）、 Linux操作系统、Golang，Python，C等编程语言、 Prometheus监控系统、 Grafana仪表盘。

3. 资料积累

eBPF官网（https://ebpf.io/）， 《eBPF学习手册》，libbpf-bootstrap开源项目

4. 软硬件条件

Linux操作系统、VSCode/Vim等IDE软件、笔记本电脑一台

要研究和解决的问题

1. 研究eBPF程序在内核的加载过程：深入研究 eBPF 在内核中的工作原理，包括字节码的验证、转换为机器码的过程，以及如何在内核安全沙盒中运行自定义程序。

2. 编写eBPF程序：研究主流的eBPF编程方案，如BCC，bpftrace，libbpf库等，选用合适的编程框架编写eBPF程序。

3. 实时监控实现：利用eBPF技术实时捕获内核中与ELF文件加载相关的关键事件如execve等系统调用作为挂载点，通过观测挂载点执行前后的数据的差异从而对ELF文件加载进行监测。

4. 性能优化： 评估eBPF在ELF加载过程中的性能开销，优化捕获机制，确保实时性与效率的平衡。

5. 可视化设计：通过开源项目ebpf_exporter作为数据源，使用Prometheus监控eBPF程序，最终将Prometheus和Grafana仪表盘结合实现整个程序的可视化设计。

1.工作思路和方案（怎么做）

(1) 需求分析: 收集相关资料，熟悉 ELF 文件格式和加载机制，以及 eBPF 技术的核心原理和使用方法。

(2) 内核监控模块: 利用 eBPF 捕获与 ELF 文件加载相关的内核事件（如execve、动态链接加载，load_elf_binary，elf_load，create_elf_tables等内核函数执行过程）。

(3) 数据处理模块: 将捕获的原始事件数据从内核传递到用户空间，并解析和存储。

(4) 可视化展示模块: 使用ebpf_exporter采集eBPF生成的数据，结合Prometheus组件对数据采集，通过Grafana将数据展示。

(5) 关键技术研究: 使用 BCC或libbpf或bpftrace编写 eBPF 程序。捕获与 ELF 文件加载相关的系统调用（如execve，load_elf_binary，elf_load，create_elf_tables等）。

(6) 性能优化: 研究如何最小化 eBPF 程序的性能开销，确保监测实时性。

2. 毕业论文进度计划

2023.11.20-2024.01.10 确认选题，明确论文内容，查找现有资料和文献，撰写

《开题报告》。

2024.01.11-2024.02.28　收集和阅读与 ELF 文件加载机制和 eBPF 技术相关的文献，整理核心理论，梳理 ELF 文件的加载流程和 eBPF 的实现方式。

2024.03.01-2024.03.29　编写 eBPF 程序，捕获 ELF 文件加载相关的内核事件（如execve、动态链接器加载等），填写《中期汇报表》

2024.03.30-2024.04.25　实现用户态与内核态的数据交互，完成原始事件数据的解析和存储，开发可视化模块，设计动态展示 ELF 文件加载过程的交互界面。

2024.04.26-2024.05.15　完成毕业设计论文初稿。

2024.05.16-2024.05.29　进行毕业论文的修改及完善并准备答辩PPT。

2024.05.30-2024.06.05　进行论文答辩。

2024.06.06-2024.06.10　根据答辩意见修改论文，并提交最终版本论文。

指导教师意见

签字　　年　月　日

西安邮电大学毕业设计（论文）成绩评定表

<table>
<tr><td>学生姓名</td><td></td><td>性别</td><td></td><td>学号</td><td></td><td>专业班级</td><td></td></tr>
<tr><td>课题名称</td><td colspan="7"></td></tr>
<tr><td rowspan="3">指导教师意见</td><td>支撑指标点/赋分</td><td>3-2/20</td><td>4-2/20</td><td>5-3/10</td><td>7-2/10</td><td>8-2/10</td><td>11-2/10</td><td>12-2/20</td><td>合计</td></tr>
<tr><td>得分</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="8">指导教师(签字)：　　年　月　日</td></tr>
<tr><td rowspan="3">评阅教师意见</td><td>支撑指标点/赋分</td><td></td><td></td><td></td><td></td><td></td><td></td><td>合计</td></tr>
<tr><td>得分</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="8">评阅教师(签字)：　　年　月　日</td></tr>
<tr><td rowspan="3">验收小组意见</td><td>支撑指标点/赋分</td><td></td><td></td><td></td><td></td><td>合计</td></tr>
<tr><td>得分</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="6">验收小组组长(签字)：　　年　月　日</td></tr>
<tr><td rowspan="3">答辩小组意见</td><td>支撑指标点/赋分</td><td></td><td></td><td></td><td></td><td>合计</td></tr>
<tr><td>得分</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="6">答辩小组组长(签字)：　　年　月　日</td></tr>
<tr><td rowspan="3">学生总评成绩</td><td>评分比例</td><td>指导教师(20%)</td><td>评阅教师(30%)</td><td>验收小组(20%)</td><td>答辩小组(30%)</td><td>合计</td></tr>
<tr><td>评分</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="5">毕业论文(设计)最终等级制成绩（优秀、良好、中等、及格、不及格）</td><td></td></tr>
<tr><td>答辩委员会意见</td><td colspan="7">学院答辩委员会主任(签字、学院盖章)：　　年　月　日</td></tr>
</table>

摘　要

随着 Linux 内核的演进与安全需求的提升，eBPF（Extended Berkeley Packet Filter）技术因其内核级动态插桩能力与轻量级特性，逐渐成为系统行为分析的重要工具。ELF（Executable and Linkable Format）作为主流的可执行文件格式，其加载过程的动态性与复杂性对系统安全与性能优化提出了挑战。

本文设计并实现了一种基于 eBPF 的 ELF 文件加载过程分析及可视化系统，通过动态追踪 ELF 文件加载时的系统调用、内存映射、符号解析等关键行为；通过探针并结合 eBPF 的实时数据采集与低开销特性，构建了 ELF 加载流程的动态关联模型，使用 ebpf_exporter 提取 eBPF 程序生成的数据到 Prometheus 中，最终通过 Grafana 实现可视化系统的展示。实验表明，该系统能够有效识别 ELF 文件加载，相比传统静态分析工具具有更高的时效性与准确性，为系统安全分析和性能优化提供了新的技术路径。

关键词：eBPF；Linux 内核；ebpf_exporter；Prometheus

ABSTRACT

With the continuous evolution of the Linux kernel and the increasing demand for security, eBPF (Extended Berkeley Packet Filter) technology has emerged as a crucial tool for system behavior analysis due to its kernel-level dynamic instrumentation capabilities and lightweight characteristics. ELF (Executable and Linkable Format), as the mainstream executable file format, presents challenges in system security and performance optimization due to the dynamic and complex nature of its loading process.

This paper designs and implements an eBPF-based system for analyzing and visualizing the ELF file loading process. By dynamically tracing critical behaviors such as system calls, memory mapping, and symbol resolution during ELF file loading, and leveraging eBPF's real-time data collection and low-overhead characteristics, the system constructs a dynamic association model of the ELF loading workflow. The ebpf_exporter is utilized to extract data generated by eBPF programs into Prometheus, which is subsequently visualized through Grafana. Experimental results demonstrate that the system effectively identifies ELF file loading processes with higher timeliness and accuracy compared to traditional static analysis tools, offering a novel technical approach for system security analysis and performance optimization.

Key words： eBPF；Linux Kernel；ebpf_exporter；Prometheus

目　　录

第 1 章　绪论

1.1　研究背景及意义

随着 Linux 内核的不断升级，如果想要直接通过 Linux 内核源码分析[1]操作系统内部工作机理显然愈加复杂且难以调试，通过 Linux 传统的安全和性能分析工具因为需要频繁地在用户态和内核态之间切换导致效率低下，而 eBPF 在内核中直接运行，并且支持其他第三方工具，能够在内核中进行数据采集，减少了内核态和用户态的切换开销，提高了整体效率和资源利用率。

BPF（Berkeley Packet Filter），即伯克利数据包过滤器，最初构想提出于 1992 年，其目的是为了提供一种过滤包的方法，并且要避免从内核空间到用户空间的无用的数据包复制行为。eBPF 是起源于 Linux 的一个技术，eBPF 可以通过探针或 Hook 挂载等方式挂载到 Linux 内核的系统调用或者内核函数上[2]，这种方式可以无侵入性的不需要更改内核源代码来实现扩展内核功能，这对 Linux 内核的安全性提供了基本的保障。

如今，eBPF 程序的应用范围极广，比如，可以通过 XDP[3]低消耗的实现 Linux 中网络流量的转发，网络负载均衡。再比如通过 eBPF 程序开发出的开源项目 Cilium，在 Kubernetes 中充当网关的 kube-proxy 在 Cilium 项目出现前，仍然在使用效率低下的 iptables 进行流量转发，通过 Cilium 可以简化 Kubernetes 的网络复杂度，并且可以使得 Kubernetes 对并发网络请求具有更高的处理能力，为维护的错误定位降低了难度。

ELF 文件格式是一种广泛使用的标准文件格式，用于定义程序、库（如共享库）、核心转储和其他在类 Unix 操作系统中执行的文件类型[4]。分析操作系统中 ELF 目标文件的加载和执行过程，可以准确提取操作系统对于 ELF 目标文件的使用需求，例如 ELF 文件中需要提供给操作系统的特定信息及其意义[5]，这些信息的提取对于分析软件开发工具链中 ELF 文件的编译和链接、库的组织等有重要意义。

1.2　国内外研究现状

国外学者对 eBPF 的研究较早且深入，2014 年 Alexei Starovoitov 实现了 eBPF 并扩展到用户空间[6]，这对 Linux 系统研究影响力巨大。如今 eBPF 技术在容器，网络，系统安全等领域进行了深入的研究，eBPF 作为一项具有里程碑意义的内核技术，正在深刻地改变着我们构建和管理计算机系统的方式。

在网络领域，由于近年云原生领域飞速发展，容器生命周期较为短暂。传统网络管理工具（如 iptables）因依赖用户态流量处理机制，逐渐面临高延迟、高资源消耗等性能瓶颈。在此背景下，基于 eBPF 技术开发的 Cilium 通过绕过内核网络协议栈直接处理数据包[7]，

将网络转发效率提升超过 40%；它利用 eBPF 实现高效的网络策略执行、负载均衡、加密以及深度网络可见性，能够直接在 Linux 内核中操作网络流量[8]，从而避免了传统 iptables 等机制带来的性能瓶颈。与 Cilium 并肩的其他网络插件如 Katran，Calico，XDP 作为 eBPF 技术的典型实践，为云原生网络优化和容器运行时安全的核心方案[9]。

eBPF 在安全领域的应用也日益增多，Falco 是一个基于 eBPF 的运行时安全监控工具，它可以检测异常的系统行为[10]，如未预期的进程执行、文件访问、网络连接等。Tracee 同样利用 eBPF 追踪系统事件，并专注于安全取证。

我校西安邮电大学紧跟时代步伐，至目前为止，我校已连续创办三届 eBPF 开发者大会，除此之外陈莉君教授作为我校开源杰出贡献作者，研究 Linux 内核的名师，跟踪 Linux 内核发展动向，并多次进行技术分享。国内知名企业例如，阿里云、腾讯、美团等企业，也积极参与，并推动 eBPF 技术的发展，以及在其他更多领域中的应用。

无论从企业还是学术方面来看，位于开源社区之首的 Linux 操作系统，定将成为未来主流的操作系统，而 Linux 庞大的内核代码可能对社区的开发带来不便，eBPF 对 Linux 内核调试，开发带来的便捷性无可比拟。

从历史角度来看深入了解 eBPF 技术已然是技术趋势，研究 eBPF 对深入 Linux 内核源码及其运行机制都有着深远的影响，且 Linux 系统中大部分文件为 ELF 类型文件，因此研究 ELF 文件加载过程对系统安全，系统性能优化等都将开拓新的领域。

1.3 课题研究目标

本次毕业设计的目标是在 Linux 系统上开发基于 eBPF 的 ELF 文件加载监控程序，观测 ELF 程序加载的过程，提取重要过程信息及相关 ELF 文件信息的使用情况，并对这些信息进行采集、整理，最后将这些数据基于 web 等方式进行展示，实现 ELF 文件加载过程的可视化。该程序实现的功能包括：①设计基于 eBPF 的 ELF 加载行为监控框架。②实现用户态数据校验与可视化交互系统。③验证系统性能与安全性。

1.4 本文的组织结构

本文共分六个章节，主要介绍了基于 eBPF 程序的 ELF 文件加载过程分析及可视化系统设计与实现。

第 1 章　绪论，该章节介绍了基于 eBPF 的 ELF 文件加载过程分析及可视化系统的研究背景及意义，国内外研究现状，该课题的研究目标及全文组织结构。

第 2 章　相关技术，该章节详细阐述了该系统的技术需求，读者可以清楚了解其中重点的开发技术。

第 3 章　系统设计，该章节详细阐述了本系统的功能设计，总体设计架构，包含内核态监控层，用户态处理层以及可视化展示层。

第 4 章　系统实现，该章节详细阐述了项目开发环境，系统功能实现过程。

第 5 章　成果与系统测试，该章节详细展示了开发成果和整体系统测试测试，包含测试环境，测试过程，测试数据对比，性能测试和测试结论。

第 6 章　总结与展望，该章节详细阐述了本系统可能存在的问题，以及对这些问题的改进措施。

1.5　本章小结

本章介绍了基于 eBPF 的 ELF 文件加载过程分析及可视化系统设计与实现的研究背景、研究现状、研究目标及全文组织结构。该毕业设计本着利用 eBPF 深入研究 Linux 内核及对 ELF 文件加载过程的探索，实现 eBPF 对 ELF 文件领域的扩充，为后人提供研究思路。本章为后续章节的研究提供了必要的理论基础及参考。

第 2 章　相关技术

2.1　开发技术

2.1.1　eBPF 内核态开发技术

eBPF 内核态程序采用 libbpf-bootstrap 开发框架进行开发，在开发过程中，需要通过阅读 Linux 官方内核源码定位需要被观测的内核函数或者系统调用以及学习 libbpf-bootstrap 开源社区示例来掌握该技术框架的使用及开发流程。通过 clang，llvm 以及 make 等编译工具实现对 eBPF 内核态程序的代码编译[11]。

由于本毕设中的 eBPF 内核态程序为了兼容更多 Linux 内核版本正常运行，所以还涉及到 CO-RE 功能，此功能需要 Linux 内核支持 BTF，并且通过开源项目 libbpf-bootstrap 中 bpftool 工具生成 vmlinux.h 头文件，利用该头文件中的宏定义编写 eBPF 程序可以实现不同 Linux 内核版本的正常运行，真正做到编译一次 eBPF 程序，兼容所有支持 BTF 的 Linux 内核版本运行。

2.1.2　用户态采集数据技术

用户态程序使用 ebpf_exporter 监测 BPF maps 中的数据，当内核态程序产生数据输出时输入至 BPF maps，然后 ebpf_exporter 就可以不断从中提取数据并配置 yaml 文件通过内置解码器对提取数据进行自定义解码，转换成 Prometheus 监控软件兼容的数据格式。接着通过 HTTP 端口暴露解码后的数据，由 Prometheus 通过 pull 的方式拉取 ebpf_exporter 解码后的数据并存放在 TSDB 时序数据库中，完整有序地保存这些数据，并且写入磁盘中进行持久化保存，以免断电等意外事故发生导致数据采集中断和数据缺失。

2.1.3　可视化数据技术

可视化数据阶段使用 Grafana 进行图表展示，Grafana 依据 Prometheus 向外暴露的端口为数据源，使用 PromQL 语句通过 HTTP 请求查询 Prometheus Server 中 TSDB 时序数据库，获取 TSDB 中存储的 Prometheus 格式的数据，然后集成在 Grafana 的仪表盘中[12]。

2.2　开发技术介绍

2.2.1　libbpf-bootstrap 框架

eBPF 内核态程序的开发框架有许多，例如：通过 Python 语言和 C 语言开发的 BCC，使用脚本语言开发 eBPF 程序的 bpftrace，使用纯 Golang 语言开发的 ebpf-go 等开发框架。

但以上的开发框架都存在一定的弊端。例如 BCC 框架难以调试，在调试报错代码过程中，出现过多无用信息，对调试造成较大干扰。BCC 在便捷性方面较差，如果需要移植 BCC 框架开发的 eBPF 程序必须在宿主机安装相同版本的 Python 开发环境。性能不够高，Python 程序占用系统资源较多。

Bpftrace 开发框架是一种新型的开发框架，采用全新的脚本语言对 eBPF 程序进行开发，虽然便捷，但也存在一些问题。例如，对 Linux 系统内核版本兼容性较差，只能对支持 bpftrace 的发行版系统中才能使用，对于内核版本较低的 Linux 操作系统则可能出现脚本无法执行，甚至可能无法安装 bpftrace 工具。

ebpf-go 开发框架采用纯 Golang 语言编写 eBPF 程序。虽然其性能几乎与 C 语言不分上下，但相应文档还不够完善。

libbpf-bootstrap 作为 eBPF 程序开发框架之一，内核态和用户态程序采用纯 C 语言开发。相比较上述几种开发框架，libbpf-bootstrap 作为最经典的纯 C 语言架构，其框架优势有如下几个方面：

① 适配性强

我们知道 Linux 底层代码由 C 语言和汇编语言组成的，因此 libbpf-bootstrap 框架与 Linux 适配性极强，通过系统调用和内核函数载入 eBPF 程序，并且可以丝滑地调用 Linux 相关的接口

② 开发便捷性高

在编写完 eBPF 内核态程序后，libbpf-bootstrap 框架会自动通过 bpftool 工具生成 skel.h 结尾的头文件，该头文件中包含操作 eBPF 内核态程序的各种接口，在用户态程序中可通过这些接口操作和管理 eBPF 内核态程序，可以使我们更加专注于开发 eBPF 内核态程序，减少对用户态程序的开发负担。。

③ 程序通用性强

libbpf-bootstrap 支持 CO-RE 功能，该技术主要功能是通过宏定义消除 Linux 不同版本内核参数的差异，可直接通过参数名操作内核中的数据，再通过 Makefile 使用 cmake 编译将 eBPF 内核态程序和用户态程序编译为一个二进制文件，此时的二进制文件可以脱离开发环境在任意支持 CO-RE 功能的 Linux 中使用，且无需额外安装的任何相关依赖包，完成真正的一次编译多次使用。。

④ 低资源占用

由于 eBPF 程序使用 C 语言编译，而 C 语言的执行效率在所有开发语言中居于前列，其编译完成的二进制文件体积小，运行时占用系统资源小，这对于系统负担极小，不会影响其他应用程序的正常使用。

⑤ 安全性

使用 libbpf-bootstrap 框架在编译完成后 eBPF 程序运行后，tracepoint/kprobe 等探针挂载到内核函数/系统调用前会对程序安全性进行自动检测，例如程序某个申请变量大小不够

明确等可能会对系统稳定性造成威胁，则会自动退出，而不会执行，保证系统内核的安全性。

2.2.2 ebpf_exporter 数据采集

ebpf_exporter 是由 Cloudflare 公司对于原生 exporter 采集的进行二次开发得到的项目，它的主要功能是对 eBPF 内核态程序中的数据进行采集并且转化为 Prometheus 支持的数据格式，还可以自行定义查询这些数据的 API 接口，Prometheus 可以通过这些 API 接口获得到 eBPF 内核输出的数据。

因此 ebpf_exporter 作为强大的内核数据采集工具，能够高效地收集和导出 Linux 内核中的自定义指标数据。ebpf_exporter 主要具有如下优势：

① 低开销性

传统数据采集过程需频繁地在用户态和内核态之间进行上下文切换，导致数据采集对系统开销过大，并且由于需要用户态和内核态转换，因此对于采集效率来说也并不理想。比如，在采集系统调用相关数据时，传统方法采集可能每个数据都需要通过系统调用或其他方式从用户态进入内核态获取数据，然后再返回用户态输出数据，这对于整体系统的性能来说非常低效。

ebpf_exporter 的 eBPF 程序可以直接在内核态运行，数据采集过程更加高效。与传统数据采集方法相比，大大减少了上下文切换的开销，显著提高了数据采集的效率。同时，ebpf_exporter 在用户态的资源占用相对较低，使得 ebpf_exporter 能够在不影响系统正常运行的前提下，长期稳定地进行数据采集工作，为系统监控提供可靠的数据支持。

② 监控能力强大

借助 eBPF 技术，ebpf_exporter 能够深入到操作系统内核的各个关键环节，采集到传统方法难以获取的丰富数据。支持定制化的 eBPF 程序，可以根据自身的监控需求定义各种独特的监控指标。无论是特定应用程序的性能指标，还是系统中某种罕见的异常事件，都可以通过编写相应的 eBPF 程序进行采集和监测。

③ 可扩展性

ebpf_exporter 作为二次开发的 exporter，与 Prometheus 监控系统兼容性极强。它能够将采集到的 eBPF 数据按照 Prometheus 的指标格式进行导出，使得这些数据可以方便地被 Prometheus 服务器抓取、存储和查询

④ 数据传输效率高

ebpf_exporter 通过在内核态对数据进行预处理，只将经过过滤和聚合后的关键数据传递到用户态，大大减少了数据传输量，优化了数据传输过程，提高了系统的整体性能。

⑤ 自定义指标名称

ebpf_exporter 在其配置文件 yaml 中，可为所采集所有的数据自定义所有指标名，例如

本毕设中 elf_open_total，通过该指标名即可在 Prometheus 中获取这些数据，同时可以通过添加标签如本毕设中 filename="/usr/bin/cat"、Magic="ELF" 来区分不同 ELF 文件加载过程中的信息。该毕设系统中涉及到的使用 ebpf_exporter 内置解码器解码的数据类型如下表 1 所示。

表 1　数据格式解码表

变量名	含义	原数据类型	解码后数据类型
fname	ELF 文件名	string	string
Magic	ELF 魔数	string	string
e_type	ELF 类型	uint16	static_map
e_entry	入口地址	uint64	ksym
e_machine	CPU 架构	uint16	static_map
e_shoff	节头偏移	uint64	ksym
e_phoff	段偏移	uint64	uint
e_shnum	节头数	uint16	uint
e_shentsize	节头大小	uint16	uint
e_phnum	段头数	uint16	uint
e_phentsize	段大小	uint16	uint

2.2.3　应用监控组件 Prometheus

Prometheus 作为开源的监控系统。它采用 pull 模型获取监控数据，基于时间序列存储数据，并提供强大的查询语言 PromQL 以及灵活的告警规则。

① 内置筛选过滤器

Prometheus 的数据模型基于多维度时间序列。每个时间序列由指标名称和一组键值对标签唯一标识，例如 instance="localhost:9543"、job="ebpf_exporter"，通过 PromQL 筛选语句过滤出我们想观测的 ELF 文件加载过程信息，减少其他 ELF 数据加载的干扰，能够从多个角度分析 ELF 文件信息。

② 可自定义监控时间范围

Prometheus 监控系统可以自定义监控时间范围，例如：近期 15min 内，10s 内等等，可自行选取合适的监控始末时间，进行更加精准的监测，或者可以查询从某个具体时间点（如 "2025-05-14 00:00:00"）开始到另一个时间点（如 "2025-05-14 23:59:59"）之间的数据。自定义时间范围如图 2.1 所示。

图 2.1　自定义监控时间范围图

③ 持久化存储

对于传统的数据采集或者程序监控系统，对历史数据的保留可能性极低，而 Prometheus 自置 TSDB 时间序列数据库，可以将历史数据存放在 TSDB 中，然后持久化在本机的硬盘/磁盘中，实现数据的持续存储，解决了丢失历史采集的数据的缺陷。

2.2.4　可视化仪表盘 Grafana

Grafana 是一款功能强大的开源数据可视化平台，它最初由 Rajdell 信息技术公司开发，现已成为全球广泛使用的可视化工具之一。可以为我们提供个性化的仪表盘和图表创建、数据分析以及监控告警等全面的可视化解决方案。

使用 Grafana 具有以下优势：

① 图表类型多样化

为了满足不同数据展示的需求，Grafana 提供了丰富多样的图表类型，支持的部分图表类型如下图 2.2 和图 2.3 所示。

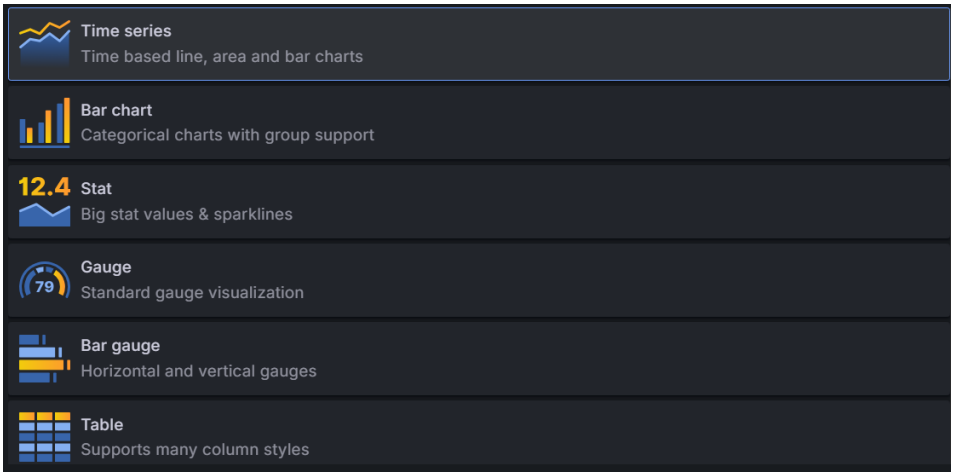

图 2.2　图表类型图

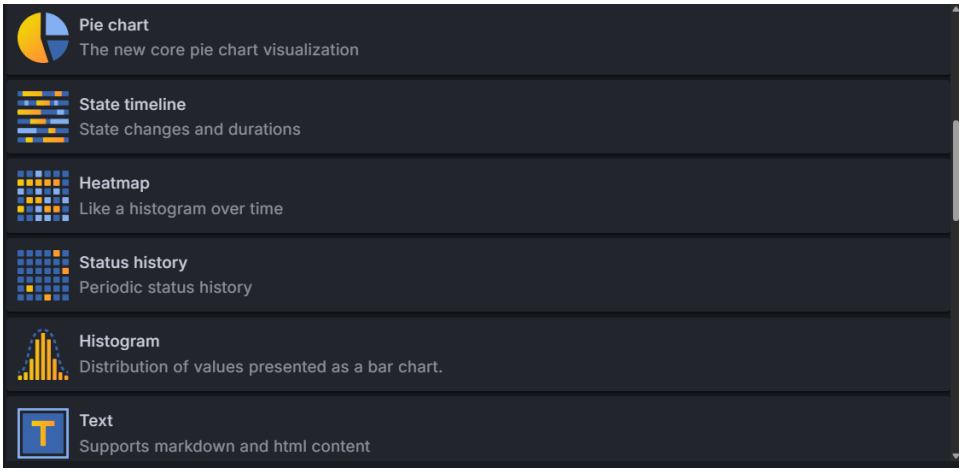

图 2.3　图表类型图

② 可自定义告警

虽然，本毕设中并没有与监控告警相关的设计，但在有需要的情况下 Grafana 可以与 Prometheus 进行对接，并且能提供比 Prometheus 监控告警组件 AlertManager 更多的告警类型，可以通过电子邮件，微信或其他 WebHook 方式进行告警，能够在短时间内得到告警通知。

2.2.5 Docker 容器技术

本部分属于调优部分，如果没有需求可以不进行对应用的容器化，但为了丰富毕业设计，增加整个系统的可移植性，而不单单是 eBPF 内核态程序的可移植性，所以我在这里引入了 Dcoker 容器化技术，当然也可以使用其他的容器化技术。

Docker 的底层逻辑依然是 Linux 内核的特性[13]，Docker 引擎运行的容器具有自己的命名空间以及独立的资源限制，这是基于 Linux 内核中的 Cgroup 和 Namespace 功能实现的[14]，如果读者感兴趣可以自行查阅相关文档。基于以上的特性，Docker 具有如下优势：

① 独立运行环境

这一特性可以使得运行中的容器都具有独立的运行环境，且单个容器的崩溃不会导致其他容器应用的运行崩溃，这对调试以及故障排查都提供了更加便捷的方式。

② 防止内存泄露

因为我们可以对容器进行硬件资源限制，当容器使用的硬件资源超过预定值，则会被 Dcoker 引擎自动重启容器，防止容器资源泄露导致操作系统异常崩溃的可能性。

③ 强移植性

Dcoker 容器引擎可以把运行中的容器制作为容器镜像，此镜像中会存放容器运行过程中的所有数据，因此提供了强大的移植能力，而且容器内部环境几乎可以兼容任何支持容器运行的 Linux 操作系统，在本毕业设计中主要使用容器的此功能。

2.3 本章小结

本章主要围绕着基于 eBPF 的 ELF 文件加载过程分析及可视化系统实现所需要的开发技术进行了基本的介绍。通过本章的介绍，可以了解整套系统的开发使用到的技术和开发工具。通过本章节的技术介绍可以为下一章系统架构层次的了解起到帮助性的作用，然后通过深刻理解系统每一层的实现过程，可以进一步深刻掌握系统实现的效果和具体细节，为后续的系统的使用和维护提供了指导思路。

第 3 章　系统设计

3.1　系统功能设计

3.1.1　基本 ELF 信息和生命周期

记录 ELF 加载基本事件，例如：时间、进程 PID、文件路径等基本信息。追踪 ELF 文件加载的全生命周期和重要信息包括 ELF 头部，ELF 文件类型，适用的 CPU 架构，加载程序和数据段节物理和虚拟偏移地址，动态链接库加载地址、内存映射及执行入口 e_entry 跳转等关键环节[15]。对 ELF 文件的关键属性进行动态校验。如：ELF 段节的权限（只读，只写，可执行，可读写等权限）。

3.1.2　ELF 加载可视化功能

可视化 ELF 加载过程。通过 ebpf_exporter 和 Prometheus 形式采集 ELF 加载过程中信息并实时通过 Grafana 进行展示，统计各类 ELF 加载次数，方便对操作系统整体的 ELF 加载进行宏观掌控。

3.1.3　系统高性能性与兼容性

从事件捕获到可视化展示的端到端延迟需控制在 10ms 以内，避免该系统影响对采集数据的实时有效性。内核态 eBPF 程序的 CPU 占用率≤5%，用户态组件内存占用≤50MB。整套系统的 CPU 占用率≤10%，用户态组件内存占用≤100MB 防止对操作系统正常运行造成负担。采用 CO-RE 技术，消除不同 Linux 内核版本中参数的差异性，使得本程序适用于更多 Linux 发行版及内核版本。

3.2　逻辑架构设计

基于 eBPF 的 ELF 文件加载过程分析及可视化系统的逻辑架构如下图 3.1 所示。

图 3.1　eBPF 程序及可视化逻辑结构图

从左往右，该系统可分为内核态监控层、用户态处理层、可视化展示层。

内核态监控层：利用编写完成的 eBPF 程序中挂钩机制自动识别指定的 Linux 系统调用以及内核函数，当系统进行这些函数的调用动作时，自动触发相应部分的代码对这些系

统调用和内核函数中关键数据进行提取[16]。

用户态处理层：主要通过第三方工具 ebpf_exporter 对内核态数据进行收集，再使用解码器对内核态监控层的原生数据进行解码，转化为 Prometheus 数据格式[18]和人类易于理解的数据，便于用户在可视化层便捷的查询数据。

可视化展示层：主要用于自定义图标功能，并实现用户通过简单的 API 调用或 PromQL 语句即可查询和筛选出想要得到的数据。

3.3 内核态监控层

内核态监控层（内核态 eBPF 程序）选取 eBPF 中合适的探针通过挂钩系统调用（execve、mmap、openat 等）和 ELF 解析函数（load_elf_binary，elf_load，create_elf_tables），捕获以下事件：

ELF 加载的开始：进程启动时解析的 ELF 文件路径、PID、UID。

段节加载信息：LOAD 段或节的物理地址和虚拟地址、权限（R/W/X）等信息。

BPF Maps：通过 maps 将内核态事件数据高效传输至用户态。

eBPF 内核态程序从编译到运行的整个过程[17]如下图 3.2 所示。

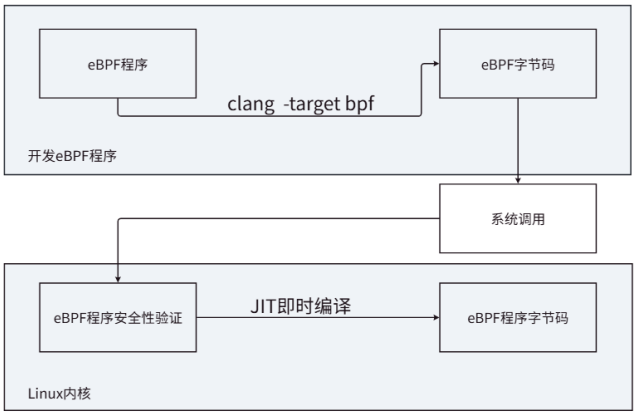

图 3.2 eBPF 内核态程序编译及运行流程图

ELF 文件的加载过程主要涉及到以下 7 个阶段：

ELF 文件的执行过程：

① 执行 execve 系统调用

首先，当用户需要新的 ELF 文件加载时，通过 execve()或者 execve_at()该系统调用从用户态转变为内核态，清空当前进程内存空间，其目的是为了替换为指定的 ELF 文件的内容，在这个过程中会判断 ELF 的真实有效性[19]，如果不存在这样的有效文件则会返回错误，若正常我们就可以获取到该 ELF 文件的进程号 PID，用户名 UID，文件描述符，文件路径等相关信息，其中 PID，文件描述符等信息会继承清空前进程的 PID 和文件描述符。

② 预处理 ELF 文件

通过调用 prepare_binprm 以及 kernel_read 预处理函数，这个过程主要将 ELF 的部分基础信息载入到 linux_binprm 结构体中，例如该结构体中的 buf 用于存放 ELF 头的 128 字节，该 ELF 文件在内存中的顶部指针，在上一步执行 execve 传入的运行和环境变量等参数信息，该结构体的部分信息如下图所示 3.3。接着调用 search_binary_handler 内核函数，此函数主要用于寻找当前 ELF 文件合适的文件处理器。

```
struct file *executable; /*
struct file *interpreter;
struct file *file;
struct cred *cred;      /*
int unsafe;             /*
unsigned int per_clear; /*
int argc, envc;
const char *filename;   /*
const char *interp;     /*


const char *fdpath;     /*
unsigned interp_flags;
int execfd;             /*
unsigned long loader, exec;

struct rlimit rlim_stack;

char buf[BINPRM_BUF_SIZE];
```

图 3.3　linux_binprm 结构体部分图

ELF 文件真正被内核加载的过程：

③ 调用内核函数 load_elf_binary

这个函数是整个 ELF 文件从执行到加载过程中最核心的函数，其中包含了大量的其他函数调用。首先该函数会读取之前写入到 linux_binprm 结构体中存放 ELF 头信息的 buf 数组，验证并判断该数组的第二到第四字节是否为 ELF 文件头 Magic，正常 ELF 文件头的第二到第四字节分别为 0x45、0x4c、0x46，第二到第四个字节这三部分分别对应“ELF”这三个字母的 ASCII 码，load_elf_binary 函数在加载文件的过程中确保魔数 Magic 真实有效，如果不正确则会直接退出该函数并返回错误。接着检查 ELF 文件的 CPU 架构兼容性。

④ 加载程序头

load_elf_binary 函数完成以上工作后，紧接着继续调用 load_elf_phdrs 读取程序头表，确定待加载的程序加载段 PT_LOAD，把 PT_LOAD 的数据存放在 ELF 程序头数据即 elf_phdata 中。

⑤ 确定 ELF 文件是否需要动态链接

如果当前加载的 ELF 文件类型是需要动态链接的 ELF 文件，则从 PT_INTERP 段获取所需的动态链接器文件路径，由于动态链接器也是 ELF 文件，所以 Linux 系统会接着递归调用 load_elf_binary 内核函数，重复步骤③以检查该解释器是否为 ELF 文件，CPU 架构信息是否适用，然后调用步骤④加载解释器程序头表内容。如果当前的 ELF 文件是静态的 ELF 文件[20]，则跳过无需此步骤。

⑥ 载入 LOAD 段程序

遍历 ELF 文件的所有程序段，通过 elf_load 内核函数将不同 PT_LOAD 段设置合适的权限，然后通过虚拟地址，物理地址，以及偏移量等信息进行地址映射。

⑦ 返回 ELF 文件的入口地址

在以上步骤全部完成以后，如果 ELF 文件不依赖动态链接器，则直接将 ELF 文件的入口地址 e_entry 作为程序的入口地址，如果该 ELF 文件还需要动态链接器才能正确加载，则最终将动态链接器的入口地址作为程序的入口地址。返回用户态，执行入口点，完成整个 ELF 文件执行和加载过程。

上面整个 ELF 文件执行和加载的流程图如下图 3.4 所示。

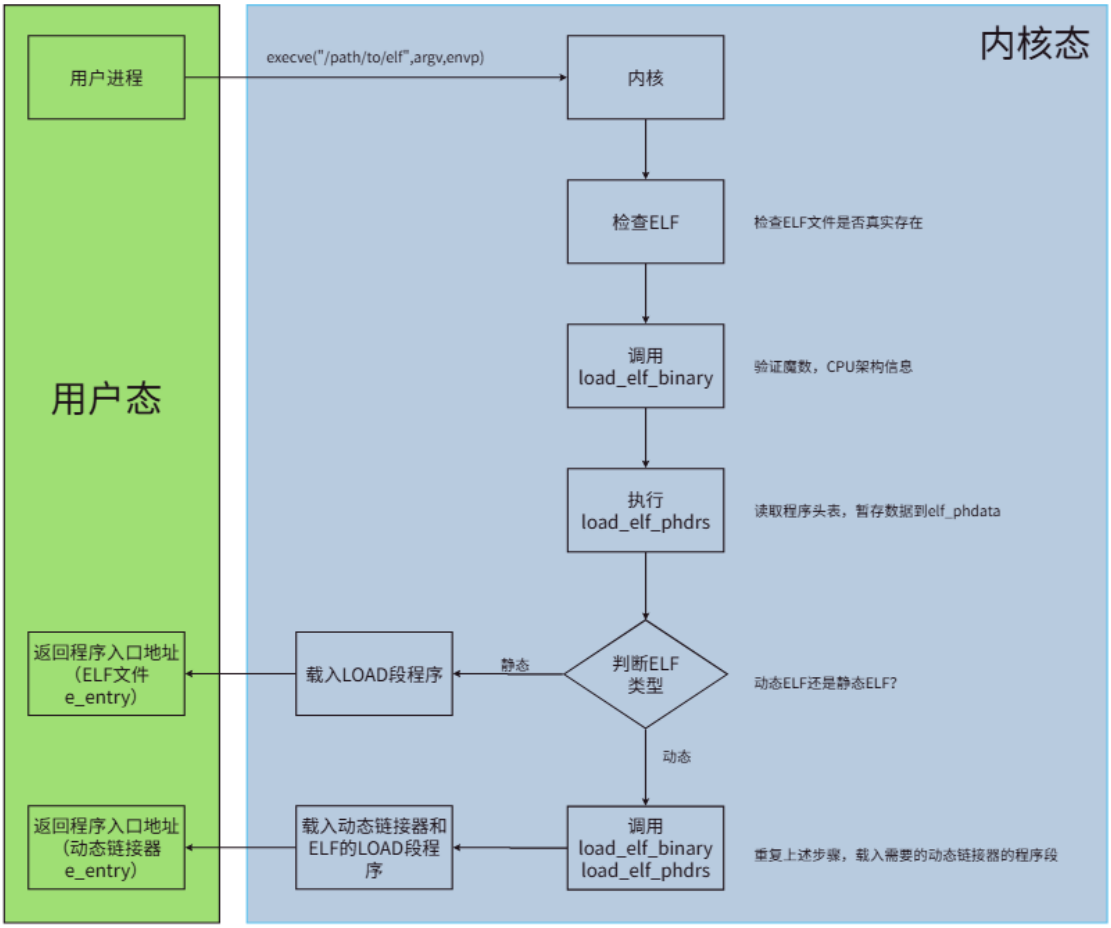

图 3.4　ELF 执行加载流程图

以上即为 ELF 文件执行和加载的整体过程，而内核态 eBPF 程序则是以多个挂载点的形式挂载到上述每一步的内核函数和系统调用的起始点，每当 Linux 执行特定函数时，触发特定的挂钩程序运行其中代码，获取 ELF 文件每个阶段的信息，最终整合数据进行输出。eBPF 程序中的探针和 Linux 内核函数/系统调用之间的关系如下图 3.5 所示。

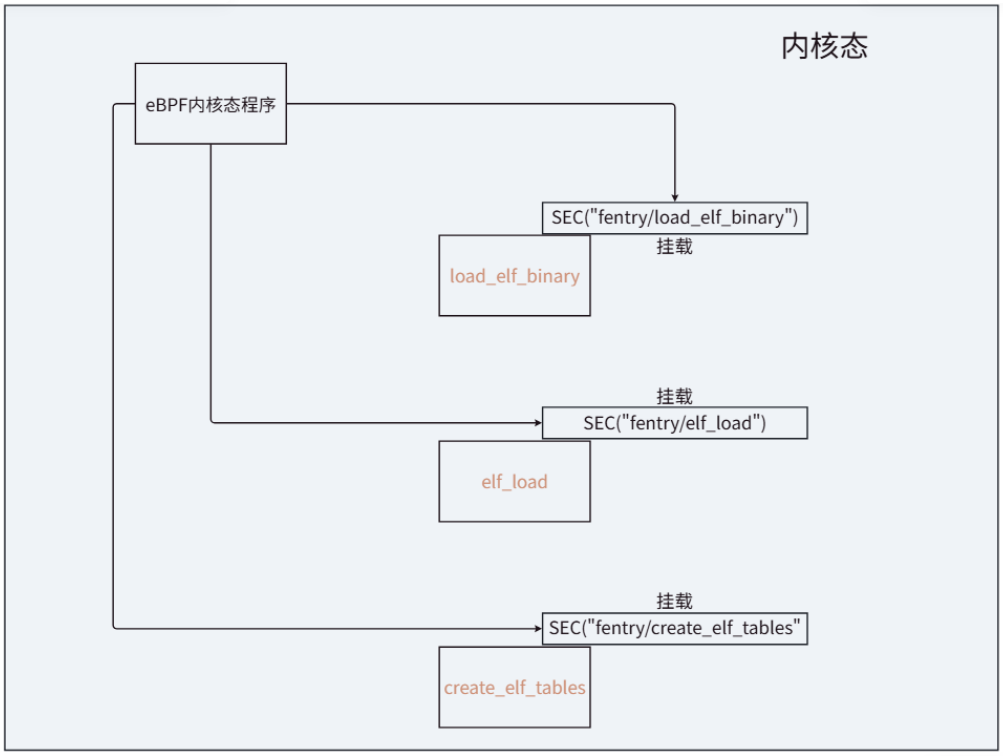

图 3.5　eBPF 程序执行流程图

3.4　用户态处理层

采用 ebpf_exporter 从 maps 中读取原始事件数据，通过 HTTP 端点（如/metrics）提供，在 ebpf_exporter 内置的解码器 Decoders（如：hex，static_map，string，syscall，uint 等）中对提取数据解码为 Prometheus 支持的格式数据，再通过 HTTP 端口暴露数据给 Prometheus，将数据存放在 Prometheus 内置的时序数据库 TSDB 中，并自定义 API 用于后续在 Prometheus 或 Grafana 中通过 PromQL 语言查询该数据。用户态处理层架构如下图 3.6 所示。

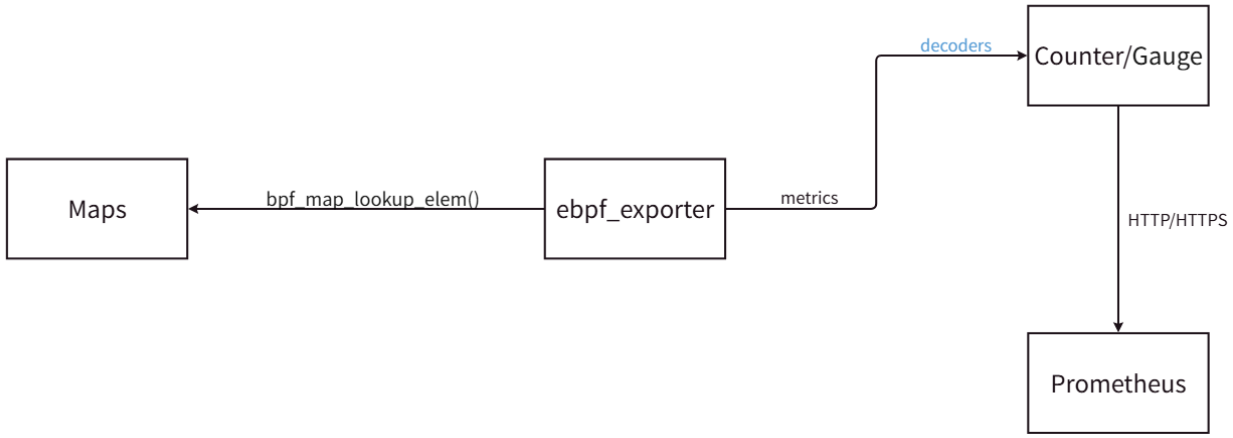

图 3.6　用户态处理层架构图

3.5 可视化展示层

在 Grafana 中通过 PromQL 语句查询，Counter/Gauge 等实时反应每秒 ELF 加载请求数及加载过程信息，可自定义按照结果过滤数据。可视化展示层架构如图 3.7 所示。

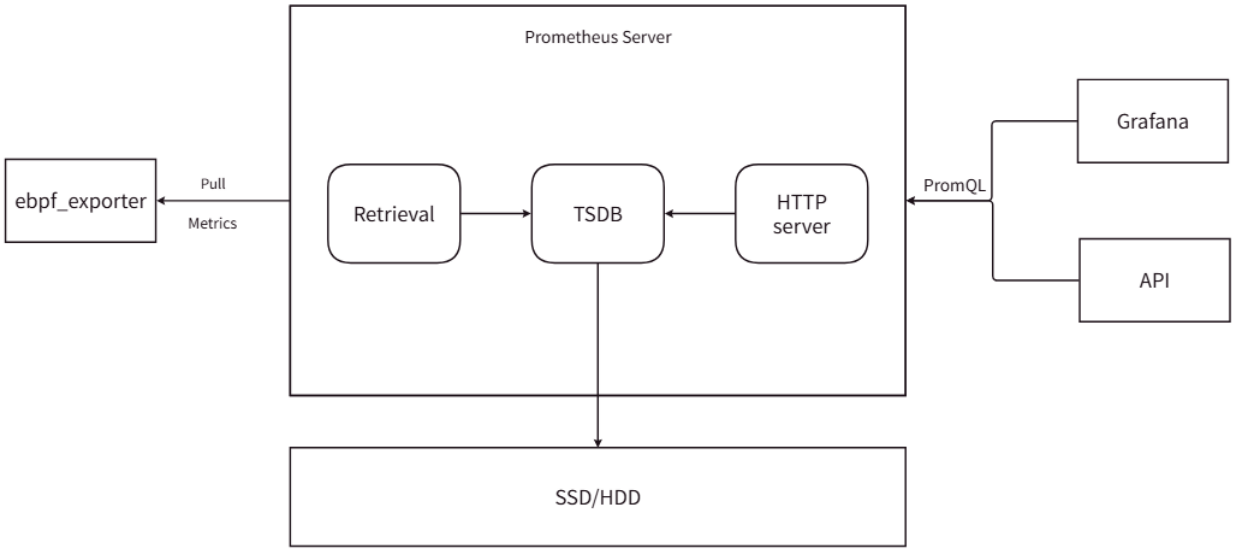

图 3.7　可视化展示层架构图

3.6 本章小结

本章主要介绍了基于 eBPF 的 ELF 文件加载过程分析及可视化系统的设计方案，包括系统功能设计，逻辑架构设计，内核态监控层设计，用户态处理层设计，可视化展示层设计。逻辑架构设计主要围绕着整个系统的功能和技术架构，以及各层次间的关系，方便用户更加整体直观地了解该系统。内核态监控层设计则主要对 eBPF 内核态程序代码编写规范，编译过程，及运行（载入内核的过程）过程，便于新手更加直观地了解 eBPF 的工作机理。用户态处理层设计主要针对 eBPF 内核态程序输出结果的采集以及处理解码等操作，方便对接 Prometheus 官方接口数据类型。最后可视化展示层设计主要通过 HTTP 端口向 Prometheus Server 发起数据请求、获取数据，并将数据通过合适的表盘进行实时展示，本毕设因没有包含 ELF 文件安全的检测机制，所以没有利用到 Prometheus 中的 AlterManager 组件，如需要告警，可自行通过官方文档学习如何使用告警组件。

第 4 章　系统实现

4.1　开发环境

开发环境主要包括软件环境，其中基于 eBPF 的 ELF 文件加载过程分析及可视化系统设计与实现系统软件开发环境如表 2 所示。

表 2　软件开发环境

操作系统	发行版本	内核版本	IDE
Linux	Ubuntu22.04	6.8.0-57	VSCode

4.2　系统功能的实现

4.2.1　内核态 eBPF 程序

ELF 文件加载过程如下：

首先通过系统调用 execve 获取待加载内容的路径和文件名，进程 ID，此进程在内存中的起始地址，被打开的时间戳等信息。

检查 ELF 文件头的魔数，通过 ELF 头文件结构体 ELF Header[21]中的 e_ident 前四个字节 ASCII 码进行判断，ASCII 码对应的字符如下表 3 所示。

表 3　e_ident 取值

e_ident[i]	ASCII 码	解码后字符
0	0x7f	0x7f
1	0x45	E
2	0x4c	L
3	0x46	F

接着检查 ELF 头部中的 e_type 即 ELF 文件类型，阅读 Linux 内核相关源码以及测试后，e_type 取值含义如下表 4 所示：

表 4　e_type 取值类型

e_type	ELF 文件类型
0	非文件类型
1	可重定位文件

2	可执行文件
3	共享库
4	核心文件

这个过程中还需要检查 ELF 文件 e_machine 取值，代表着适配的 CPU 架构，其取值类型如下表 5 所示。

表 5　e_machine 取值类型

e_type	CPU 架构
62	X86_64
183	ARM 64
40	ARM 32

读取并处理 ELF 文件的程序头表，首先获取 LOAD 段在 ELF 文件中的虚拟地址和物理地址，以及偏移量等信息，然后加载程序头表中的可加载 LOAD 段，这些段包含了后续加载到内存中的代码段和数据段，并且需要给不同 LOAD 段设置不同的权限 p_flags，p_flags 对应的不同权限如下表 6 所示。

表 6　p_flags 取值类型

p_flags	段权限
0	没有权限
1	可执行
2	可写
3	可写可执行
4	只读
5	可读可执行
6	可读可写
7	可读可写可执行

上述阶段完成后还需要查询该 ELF 文件加载过程中涉及到的加载动态链接器，并重复上述操作，然后返回动态链接器的 e_entry，如果没有则跳过，这样在用户空间可以直接根据 e_entry 进入并执行该程序。

整个 eBPF 内核态程序主要涉及到 load_elf_binary，elf_load，create_elf_tables 等 Linux 内核函数，采用 fentry 探针挂载到这些内核函数的入口点，一旦系统调用到这些内核函数，即可触发 eBPF 内核态程序中对应的 SEC。其中核心部分如图 4.1 所示。

```
my_elf.e_entry =  BPF_CORE_READ(exec, e_entry);       /
my_elf.e_type = BPF_CORE_READ(exec, e_type);         //
// my_elf.e_type = BPF_CORE_READ(exec, e_type);
my_elf.e_machine = BPF_CORE_READ(exec, e_machine);   //

my_elf.e_shoff = BPF_CORE_READ(exec, e_shoff);       //
my_elf.e_shnum = BPF_CORE_READ(exec, e_shnum);       //
my_elf.e_shentsize = BPF_CORE_READ(exec, e_shentsize);

my_elf.e_phoff = BPF_CORE_READ(exec, e_phoff);       //
my_elf.e_phentsize = BPF_CORE_READ(exec, e_phentsize);
my_elf.e_phnum = BPF_CORE_READ(exec, e_phnum);       //
my_elf.fname = BPF_CORE_READ(bprm, filename);     //读取
```

图 4.1　eBPF 内核态程序部分图

虽然包括 execve 等系统调用，但通过这些系统调用可获取的数据量极少，还有部分内核函数因受 Linux 限制，无法使用探针进行探测，因此主要观测上述三个内核函数。

4.2.2　用户态数据处理程序

用户态使用 ebpf_exporter 通过 maps 采集内核输出的数据，在 eBPF 内核态程序中创建用于数据传输的 maps，如下图 4.2 代码所示，定义哈希表存放数据，采用键值对的方式存储数据，prom_key 结构体作为键，把获取到的数据作为之分别传入 prom_key 结构体中的变量中。

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 1024);
    __type(key, struct prom_key);
    __type(value, u64);
} elf_open_total SEC(".maps");
```

图 4.2　定义 BPF MAP 图

prom_key 结构体定义如下图 4.3 所示。

```
struct prom_key {
    char fname[16];     //16    文件名
    char magic[4];      //4     ELF头魔数
    u16 e_type;         //2     ELF文件类型:0非文件类型，1可重定位文件，2可执行文件，3共享库，4Core file
    u64 e_entry;        //8     ELF文件代码运行入口
    u16 e_machine;      //2     适用的CPU架构

    u64 e_shoff;        //8     节头偏移量
    u64 e_phoff;        //8     程序头偏移量

    u16 e_shnum;        //2     节头条目数
    u16 e_shentsize;    //2     节头每个条目大小

    u16 e_phnum;        //8     程序头条目数
    u16 e_phentsize;        //8     程序头每个条目大小
};
```

图 4.3　prom_key 结构体图

在上节中介绍了 ebpf_exporter 的数据采集方式，知道它是在内核中进行高效采集，所以我们需要在 eBPF 内核态程序中添加相关代码，引入 maps.bpf.h 头文件，在本毕业设计中使用该头文件中的官方接口 increment_map 操作内核数据的采集以及数据聚合，当然也有许多其他聚合方式，读者可以自行阅读官方文档。

在上述步骤完成后，还需要配置 yaml 文件，自定义 Prometheus 中查询数据接口 API，例如下图中 elf_open_total 就是一个 PromQL 语言支持的 API，使用解码器对数据进行解码为 Prometheus 支持的数据格式后才能正常显示数据。由于本科毕业设计篇幅有限，部分 yaml 配置文件如下图 4.4 所示。

```yaml
metrics:
  counters:
    - name: elf_open_total
      help: ELF opened in the kernel
      labels:
        - name: filename
          size: 16
          decoders:
            - name: string

        - name: Magic
          size: 4
          decoders:
            - name: string

        - name: e_type
          size: 4
          decoders:
            - name: uint
            - name: static_map
              static_map:
                0: NONE
                1: REL
                2: EXEC
                3: DYN
                4: CORE
```

图 4.4　ebpf_exporter 配置文件图

完成上述配置文件后，使用该 yaml 文件启动 ebpf_exporter 进行数据监测，然后在 Prometheus 的运行配置文件中配置 ebpf_exporter 的监听端口，用于将其采集到的数据存放在 Prometheus 中的 TSDB 时序数据库中。如图 4.5 展示的为 ebpf_exporter 检测到的 eBPF 程序中探测的函数信息。

```
# HELP ebpf_exporter_build_info A metric with a constant '1' value labeled by version, revision, branch, goversic
build.
# TYPE ebpf_exporter_build_info gauge
ebpf_exporter_build_info{branch="master",goarch="amd64",goos="linux",goversion="go1.23.0",revision="16cb6b5",tags
# HELP ebpf_exporter_decoder_errors_total How many times has decoders encountered errors
# TYPE ebpf_exporter_decoder_errors_total counter
ebpf_exporter_decoder_errors_total{config="hello"} 0
# HELP ebpf_exporter_ebpf_program_attached Whether a program is attached
# TYPE ebpf_exporter_ebpf_program_attached gauge
ebpf_exporter_ebpf_program_attached{id="64"} 1
ebpf_exporter_ebpf_program_attached{id="66"} 1
ebpf_exporter_ebpf_program_attached{id="67"} 1
ebpf_exporter_ebpf_program_attached{id="68"} 1
# HELP ebpf_exporter_ebpf_program_info Info about ebpf programs
# TYPE ebpf_exporter_ebpf_program_info gauge
ebpf_exporter_ebpf_program_info{config="hello",id="64",program="load_elf_binary",tag="c33534d2df7bf2d5"} 1
ebpf_exporter_ebpf_program_info{config="hello",id="66",program="elf_load_hook",tag="db3823018469ad12"} 1
ebpf_exporter_ebpf_program_info{config="hello",id="67",program="create_elf_tables",tag="11c3726928273019"} 1
ebpf_exporter_ebpf_program_info{config="hello",id="68",program="load_elf_binary_exit",tag="412271613f31b0a3"} 1
```

图 4.5　ebpf_exporter 的 Metrics 指标图

4.2.3 容器化技术

在本毕业设计系统中，为了使得整套系统能够更好地移植，在完成原本的任务后，增添了容器化技术，将配置完成后的 Prometheus 以及 Grafana 的配置通过镜像打包的形式重新制作为新的镜像，在其他 Linux 平台中只需要获取本镜像文件，并且下载容器驱动引擎一键运行即可得到本套系统实现的全部内容，省去开发过程中的重复的环境搭建所消耗的时间，并且不用担心因配置出错而导致程序无法按照预期效果正常运行。整个容器镜像打包过程如下图 4.6 所示。

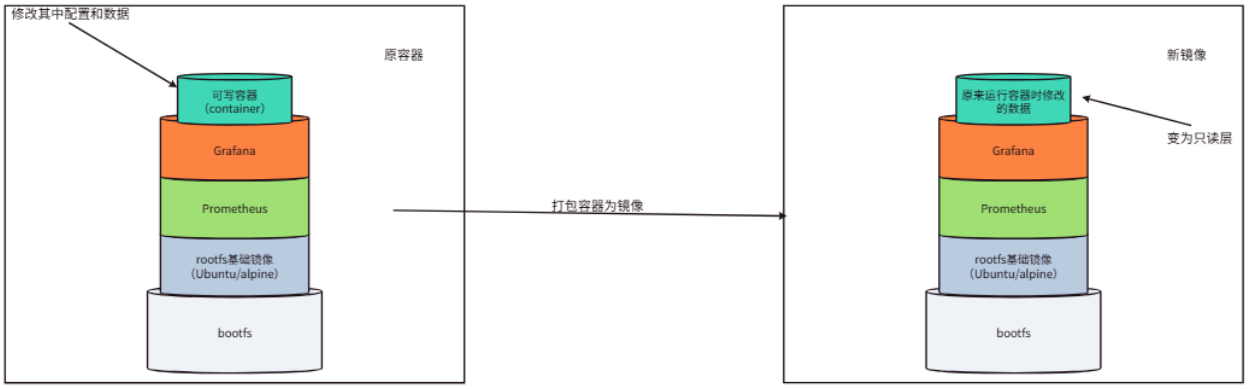

图 4.6 容器镜像打包过程图

此外，由于容器中的网络与原宿主 Linux 操作系统处于两个不同的网络[22]，要实现在 Linux 系统中访问容器内的 HTTP 应用程序，需要将 Linux 系统与容器 container 进行网络连接，由于容器网络模式较多，这里仅介绍较为常见的网络模式：网桥模式，如下图 4.7，通过 Docker0 网桥+端口暴露实现宿主与 container 容器的网络连接，这样宿主可以通过本地网络 ip+与容器进行映射的端口号即可访问到容器内部的应用程序。

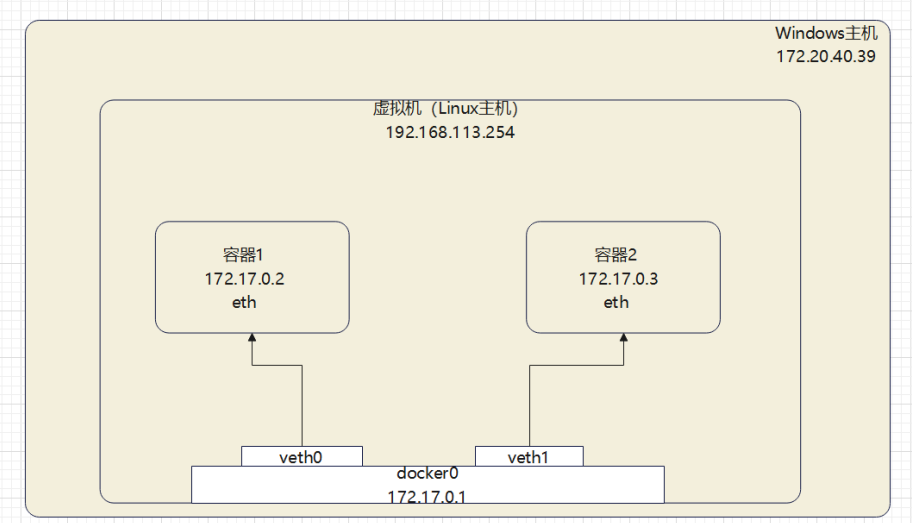

图 4.7 容器网桥模式结构图

4.3 本章小结

本章主要围绕着该系统的开发环境和系统功能实现的关键部分进行了展示和介绍，方便读者更加清晰地了解到该系统实现过程中的重难点，这部分对以后系统调试有关键性指导作用。

第 5 章　成果与系统测试

5.1　开发成果展示

5.1.1　内核态 eBPF 程序输出

内核态 eBPF 监测 ELF 文件加载的内容通过用户态程序打印输出到终端展示，此过程主要在调试过程中使用，属于中间产物，其输出如下图 5.1 所示。

```
pid = 5198,
uid = 0,
filename = /usr/bin/cat,
interp = /usr/bin/cat,

            cat-5198    [001] ...11   129.573666: bpf_trace_printk: fentry:
pid = 5198,
filename = /usr/bin/cat,
interp = /usr/bin/cat,
mem_addr = 7fffffffe898,
buf =ELF,
interp_flags = 0
            cat-5198    [001] ...11   129.573852: bpf_trace_printk:
Loading ELF segment:
         p_type=0x1, , p_offset=0x0, p_paddr=0x0, p_flags=0x4, p_align=0x1000
            cat-5198    [001] ...11   129.573858: bpf_trace_printk:
Loading ELF segment:
         p_type=0x1, , p_offset=0x2000, p_paddr=0x2000, p_flags=0x5, p_align=0x1000
            cat-5198    [001] ...11   129.573860: bpf_trace_printk:
Loading ELF segment:
         p_type=0x1, , p_offset=0x6000, p_paddr=0x6000, p_flags=0x4, p_align=0x1000
            cat-5198    [001] ...11   129.573862: bpf_trace_printk:
Loading ELF segment:
         p_type=0x1, , p_offset=0x7ab0, p_paddr=0x8ab0, p_flags=0x6, p_align=0x1000
            cat-5198    [001] ...11   129.573869: bpf_trace_printk:
Loading ELF segment:
         p_type=0x1, , p_offset=0x0, p_paddr=0x0, p_flags=0x4, p_align=0x1000
            cat-5198    [001] ...11   129.573874: bpf_trace_printk:
Loading ELF segment:
         p_type=0x1, , p_offset=0x2000, p_paddr=0x2000, p_flags=0x5, p_align=0x1000
            cat-5198    [001] ...11   129.573876: bpf_trace_printk:
Loading ELF segment:
         p_type=0x1, , p_offset=0x2c000, p_paddr=0x2c000, p_flags=0x4, p_align=0x1000
            cat-5198    [001] ...11   129.573878: bpf_trace_printk:
Loading ELF segment:
         p_type=0x1, , p_offset=0x37620, p_paddr=0x38620, p_flags=0x6, p_align=0x1000
            cat-5198    [001] ...11   129.573889: bpf_trace_printk:
ELF Header:
         entry=0x3760, magic[1]=0x45, magic[2]=0x4c, magic[3]=0x46
         e_type=3, e_machine=62
```

图 5.1　内核态数据输出图

5.1.2　ebpf_exporter 采集数据

ebpf_exporter 通过监测 maps 中内容并实时解码后输出到 HTTP 端口的部分内容如下图 5.2 所示。

```
ebpf_exporter_ebpf_program_info{config="hello",id="61",program="load_elf_binary",tag="c33534d2df7bf2d5"} 1
ebpf_exporter_ebpf_program_info{config="hello",id="63",program="elf_load_hook",tag="db3823018469ad12"} 1
ebpf_exporter_ebpf_program_info{config="hello",id="64",program="create_elf_tables",tag="11c3726928273019"} 1
ebpf_exporter_ebpf_program_info{config="hello",id="65",program="load_elf_binary_exit",tag="412271613f31b0a3"} 1
# HELP ebpf_exporter_elf_open_total ELF opened in the kernel
# TYPE ebpf_exporter_elf_open_total counter
ebpf_exporter_elf_open_total{Magic="ELF",e_entry="unknown_addr:0x128a0",e_machine="X86_64",e_phentsize="56",e_phnum="13",e_phoff="unknown_addr:0x40",e_shentsize="64",e_shnum="30",e_sho
ff="unknown_addr:0xe0748",e_type="DYN",filename="/usr/sbin/sshd"} 2
ebpf_exporter_elf_open_total{Magic="ELF",e_entry="unknown_addr:0x12f0",e_machine="X86_64",e_phentsize="56",e_phnum="13",e_phoff="unknown_addr:0x40",e_shentsize="64",e_shnum="29",e_shof
f="unknown_addr:0x3158",e_type="DYN",filename="/usr/bin/locale"} 1
ebpf_exporter_elf_open_total{Magic="ELF",e_entry="unknown_addr:0x1990",e_machine="X86_64",e_phentsize="56",e_phnum="13",e_phoff="unknown_addr:0x40",e_shentsize="64",e_shnum="31",e_shof
f="unknown_addr:0x31c0",e_type="DYN",filename="/usr/bin/mesg"} 1
ebpf_exporter_elf_open_total{Magic="ELF",e_entry="unknown_addr:0x1c6a0",e_machine="X86_64",e_phentsize="56",e_phnum="13",e_phoff="unknown_addr:0x40",e_shentsize="64",e_shnum="31",e_sho
ff="unknown_addr:0x83ba8",e_type="DYN",filename="/usr/bin/file-rr"} 1
ebpf_exporter_elf_open_total{Magic="ELF",e_entry="unknown_addr:0x27e0",e_machine="X86_64",e_phentsize="56",e_phnum="13",e_phoff="unknown_addr:0x40",e_shentsize="64",e_shnum="31",e_shof
f="unknown_addr:0x71c8",e_type="DYN",filename="/usr/bin/dirnam"} 1
ebpf_exporter_elf_open_total{Magic="ELF",e_entry="unknown_addr:0x2840",e_machine="X86_64",e_phentsize="56",e_phnum="13",e_phoff="unknown_addr:0x40",e_shentsize="64",e_shnum="31",e_shof
f="unknown_addr:0x8248",e_type="DYN",filename="/usr/bin/basena"} 13
ebpf_exporter_elf_open_total{Magic="ELF",e_entry="unknown_addr:0x2a90",e_machine="X86_64",e_phentsize="56",e_phnum="13",e_phoff="unknown_addr:0x40",e_shentsize="64",e_shnum="31",e_shof
f="unknown_addr:0x8248",e_type="DYN",filename="/usr/bin/uname"} 6
ebpf_exporter_elf_open_total{Magic="ELF",e_entry="unknown_addr:0x2b20",e_machine="X86_64",e_phentsize="56",e_phnum="13",e_phoff="unknown_addr:0x40",e_shentsize="64",e_shnum="31",e_shof
f="unknown_addr:0x191e8",e_type="DYN",filename="/usr/bin/expr"} 2
```

图 5.2　ebpf_exporter 数据采集图

5.1.3　通过 Grafana 仪表盘展示数据

最终 ELF 文件加载信息图如图 5.3 所示，其中左侧显示为 ELF 文件被调用的次数，下侧为当前系统时间戳，与系统本地时间同步，通过折线图对 ELF 文件调用次数进行动态展

示。

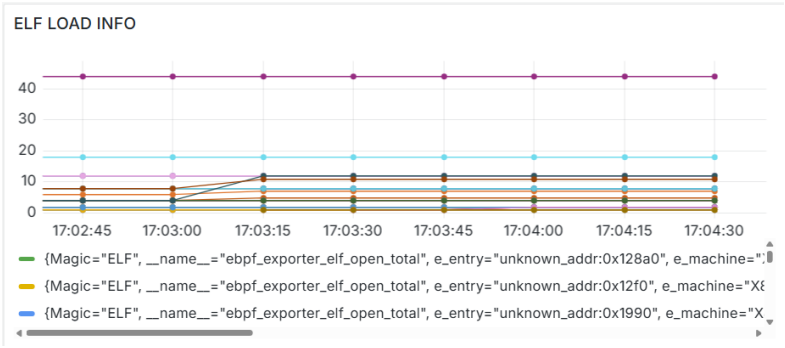

图 5.3　ELF 文件加载信息图

然后当鼠标指针落在折线图上时，会显示 ELF 加载信息，ELF 加载部分信息如下图 5.4 所示。

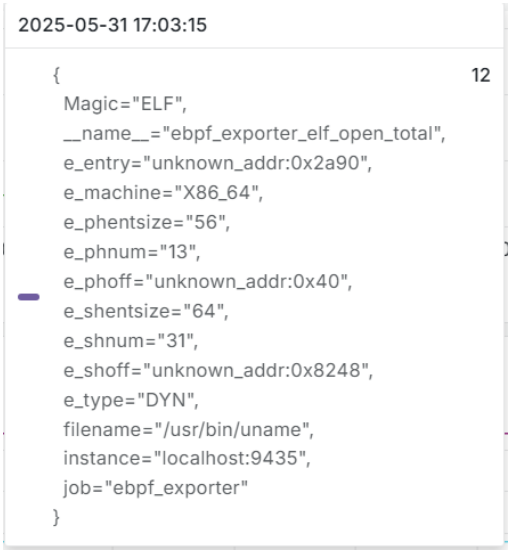

图 5.4　ELF 部分加载信息图

在 Linux 中对于如此庞大的 ELF 文件加载，如果只有图表显然非常混乱，在 Grafana 中可以通过 PromQL 语句对查询的数据进行过滤，例如只查询和 firefox 相关的 ELF 文件加载信息，则可以得到类似于下面图 5.5 所展示的数据，通过筛选语句对仪表盘中无关的数据进行过滤，可以更加专注于想要观测的数据。

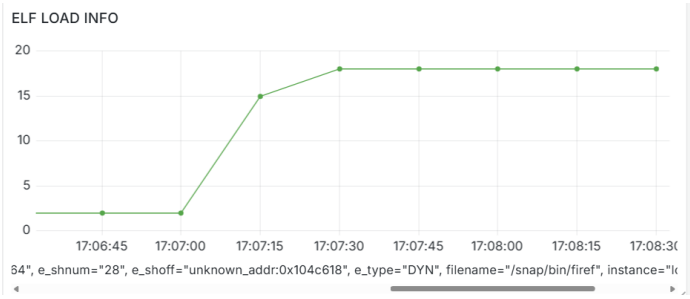

图 5.5　过滤特定 ELF 文件信息图

5.2 测试环境

为了保证该系统的强移植性，兼容性，稳定性，低资源消耗等特性，在测试环境中分别对硬件和软件进行测试，硬件测试环境如下表 7 所示。

表 7 硬件测试环境

硬件资源	硬件配置
CPU	Intel(R) Core(TM) i5-8265U
运行内存	2GB

软件测试环境如下表 8 所示。

表 8 软件测试环境

软件资源	硬件配置
操作系统	Ubuntu20.04
内核版本	5.4.0-26-generic
开发环境	不搭建开发环境

在测试前期准备过程中，不进行环境的搭建，直接通过打包好的容器镜像在测试环境中进行部署，发现测试环境中的程序正常运行，并且与预期效果一致。

5.3 测试结果对比

首先，eBPF 程序检测 execve 系统调用，获取 ELF 文件加载初始阶段的数据，例如，ELF 文件的时间戳，PID，UID 等信息，其测试结果如下图 5.6 所示。

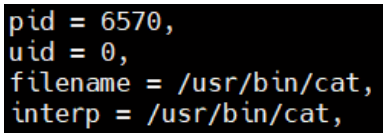

图 5.6 ELF 文件执行过程信息图

在第一部分结束后，当系统调用 search_binary_handler 内核函数时会触发 eBPF 程序中的第二个宏定义执行其中代码，捕获存放 ELF 文件的预处理信息的结构体 linux_binprm 中的内容，例如该结构体中的 buf 用于存放 ELF 文件的前 128 字节，ELF 文件在内存中的顶部指针，该 ELF 文件的解释器，其测试结果如下图 5.7 所示。

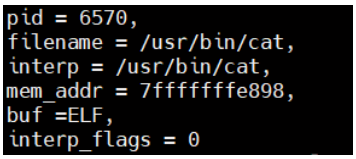

图 5.7 ELF 文件预加载信息图

接着 Linux 内核会调用 load_elf_binary 内核函数正式加载 ELF 文件，在正式加载文件前，首先获取 ELF 头文件的前四个字节和 ELF 文件适用的 CPU 架构用于比对结果是否符

合规范，获取的信息如下图 5.8 所示。

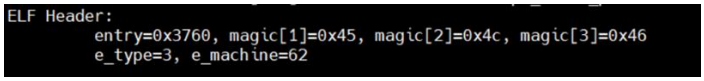

图 5.8　ELF 文件魔数校验图

load_elf_binary 函数内部调用 load_elf_phdrs 读取程序头表，确定待加载的程序加载段 PT_LOAD，本过程主要获取 ELF 文件加载的地址，偏移量，段头类型，以及对各段设置的权限信息，其内容如下图 5.9 所示。

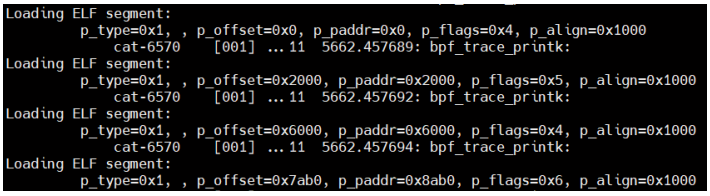

图 5.9　ELF 文件程序加载段图

接着需要确定该 ELF 文件是否需要动态链接器进行动态链接，本过程主要获取动态链接器的名称路径和动态连接器的程序加载段 PT_LOAD，完成以上信息后最终返回 ELF 文件的入口点，即入口地址 e_entry，主要结果如下图 5.10 所示。

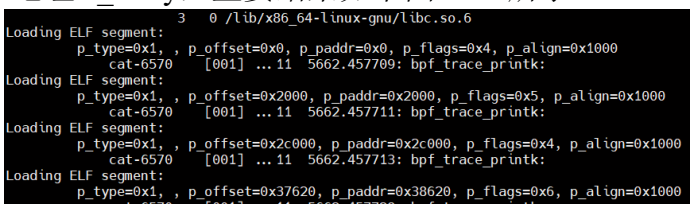

图 5.10　动态链接程序加载段图

以下是在测试环境中通过自己的 eBPF 程序和 readelf 得到的 ELF 文件加载信息进行对比，来验证本毕业设计系统的真实有效性。由于篇幅有限，这里只展示部分得到的对比结果如下表 9 所示。

表 9　测试对比

对比信息	eBPF 程序	readelf
文件路径	/usr/bin/cat	/usr/bin/cat
是否为 ELF 文件	是	是
ELF 入口地址	0x3760	0x3760
LOAD[0]偏移量	0000	0x0000000000000000
LOAD[1]偏移量	2000	0x0000000000002000
LOAD[2]偏移量	6000	0x0000000000006000
LOAD[3]偏移	7ab0	0x0000000000007ab0

量		
LOAD[0]权限	R	R
LOAD[1]权限	RX	RX
LOAD[2]权限	R	R
LOAD[3]权限	RW	RW
节偏移量	0x8218	0x8218
节大小	64bytes	64bytes
节数量	31	31

通过对比可以发现，eBPF 内核态程序采集数据基本与 Linux 工具 readelf 所得信息一致。

5.4 性能测试

完成基本的数据真实性测试后，还需要对本系统的性能进行测试，保证不影响操作系统以及其他应用的正常运行。eBPF 内核态程序对系统内存和 CPU 占用如下图 5.11 和图 5.12 所示所示。

图 5.11　eBPF 占用内存图

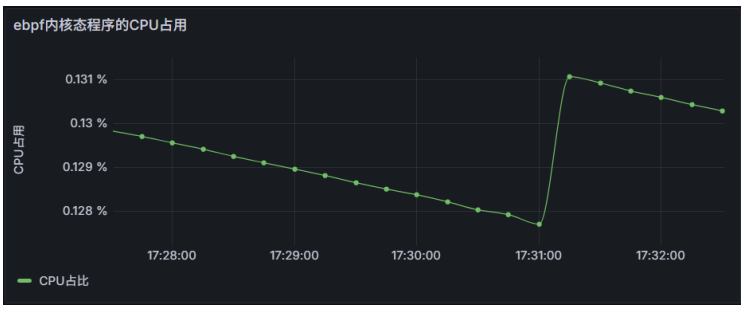

图 5.12　eBPF 占用 CPU 百分比图

同理，ebpf_exporter 对系统内存和 CPU 占用如下图 5.13 和图 5.14 所示所示。

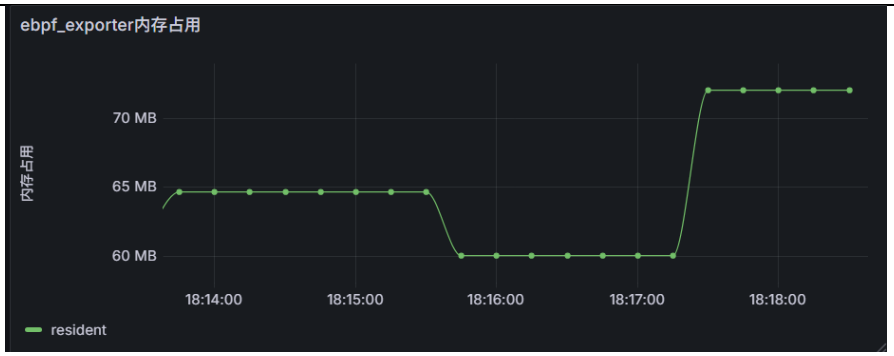

图 5.13　ebpf_exporter 占用内存图

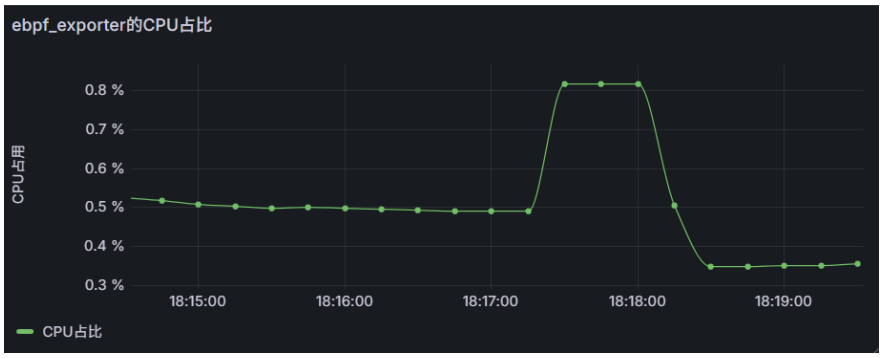

图 5.14　ebpf_exporter 占用 CPU 百分比图

通过对 eBPF 相关程序的性能分析后，可以看到比我们预期效果更好。这表明 eBPF 在性能方面具有极为突出的优势。

5.5　测试结论

通过以上对 eBPF 提取数据的真实准确性以及性能分析后，可以得到本系统具有的特征如下所示：

强移植性：整套系统对于平台迁移非常顺利，可以快速平滑地移植到其他版本的 Linux 系统中。

便捷性：本套系统在移植中非常方便，无需要复杂的前期开发环境搭建。

可靠性：与 Linux 其他分析工具对比，所得到的数据真实性几乎一致

高性能：本套系统占用系统资源小，且附带功能极多，性价比非常高。

实时性：本套系统通过图表形式可以实时对数据进行监测，系统数据读取和采集和展示反应灵敏。

综上，本套《基于 eBPF 的 ELF 文件加载过程分析及可视化系统》具有多方面优势，用户还可以在此系统上添加和扩展更多功能以便使用，并且在 ELF 文件加载领域提供了研究方向。

5.6 本章小结

本章主要介绍了该系统的开发成果与测试过程，包括测试环境、测试结果展示对比和性能分析等。

通过本章的介绍，读者可以了解一些系统测试常见的方法，通过测试结果和性能的分析和总结，可以初步定位该毕业设计系统的优势和劣势，性能表现和具体功能的执行情况。本章测试部分可以说是系统实现和实施过程中非常重要的一步，能够有效地保障系统的质量和稳定性。

第 6 章　总结与展望

6.1　研究成果总结

本文设计并实现了一个基于 eBPF 的 ELF 文件加载过程分析及可视化系统，通过运用 libbpf-bootstrap、ebpf_exporter、Prometheus、Grafana 以及 Docker 容器化技术，成功构建了一个高效、稳定且功能完善的监控与分析平台。

该系统实现了对 ELF 文件加载过程的深度监控、数据采集、存储、分析以及直观的可视化展示，通过该毕业设计系统可以深入理解 ELF 文件加载行为、为优化系统性能提供了思路。

在系统架构方面，各组件协同工作，形成了一套完整的技术解决方案。

libbpf-bootstrap 作为用户态的 eBPF 程序管理的角色，负责将 eBPF 程序加载到内核中，并与内核态的 eBPF maps 进行交互，这是进行数据采集的基础。

ebpf_exporter 则将 eBPF 程序采集到的数据以 Prometheus 指标的形式导出，实现了内核态与用户态之间的数据传递。

Prometheus 可以高效地存储这些数据到内置的 TSDB 中，并根据时间序列来管理这些数据，除此之外还提供强大的查询语言 PromQL 对数据进行查询，分析以及过滤。

Grafana 基于 Prometheus 提供的数据，通过丰富的可视化图表和自定义仪表盘功能，通过 Grafana 可以选用更加合适和丰富的图表实时展示数据，将 ELF 文件加载过程的各项指标以直观的图表形式展示出来，方便进行深入分析和监控。

最后 Docker 容器化技术则是贯穿整个毕业设计系统，在完成整套系统的搭建，并且确保系统各部分正常运作后，可以通过打包容器为新的镜像的方式，使得每层的各组件在不同的 Linux 内核中也能稳定运行而且配置文件一致，简化了部署流程，提高了系统的高可移植性和资源高效利用。

在功能实现方面，系统具备了对 ELF 文件加载过程的监控能力。能够精确地捕获 ELF 文件加载过程中的关键事件，例如文件打开，读取，内存映射，权限分配，返回程序入口地址等操作。通过对这些数据的采集和分析，用户可以分析 ELF 文件加载过程中的性能瓶颈。

此外，系统在实际测试和应用中表现出了良好的性能和可靠性。通过对多个实际场景的模拟和监控，验证了系统能够准确地采集和分析 ELF 文件加载过程数据，并且在高负载情况下依然保持较低的能开销，不会对目标系统的正常运行产生显著影响。这为系统在实际生产环境中的应用奠定了坚实基础，在性能调优、故障排查和系统稳定性保障等方面发挥实际价值。

6.2 系统中存在的不足

虽然 CO-RE 功能解决了 eBPF 程序在不同 Linux 内核版本下运行的问题，但对于不同的 CPU 架构，如 x86、ARM 平台，可能还需要对程序中部分代码进行修改才能正常移植。在 ARM 架构的嵌入式设备上运行本系统时，可能会发现某些原本在 x86 架构上正常工作的 eBPF 程序出现兼容性问题。

从 eBPF 程序采集数据到最终在 Grafana 中展示分析结果，整个过程涉及多个环节的数据处理和转换，可能会引入一定的延迟。尤其是在数据量较大或系统负载较高的情况下，Prometheus 对采集到的时间序列数据的存储和查询处理速度可能会有所下降，从而影响到 Grafana 可视化展示的实时性。

6.3 未来的改进方向

6.3.1 提高 eBPF 程序采集信息性能

对于系统中较为关键性的数据可以减少不必要的数据转换或者对 eBPF 内核态程序代码进行修改，然后启用多个 ebpf_exporter，对数据进行分批采集，提高数据从采集到展示的传输效率。

为数据处理关键环节分配更多的 CPU 和内存资源，确保系统在高负载情况下仍然能够快速响应用户请求，及时展示最新的 ELF 文件加载过程数据。

6.3.2 优化可视化功能的性能

虽然可视化在图形的呈现力方面有着不错的优势，但通过测试部分可以看出，可视化系统和数据采集系统之间数据延时较低，然而仍然能感觉到存在一定的延时，虽然目前并没有想到较好的解决方案，但在今后的学习中我会接着寻找更好的方式优化该部分系统。

结　论

本文主要开发了一套基于 eBPF 程序对 ELF 文件的加载过程中行为的监测，并最终以可视化图表的形式将其展示给用户，通过这套系统用户可以简洁清楚的了解 ELF 文件的加载过程，并可以通过可视化系统中的筛选语句对特定文件进行分析，提高了对 ELF 文件分析的工作效率。在开发技术方面，eBPF 内核态程序主要采取了 libbpf-bootstrap 的开发框架，确保了程序执行的高效性，兼容性和易调试性。数据处理方面结合 ebpf_exporter 开源项目，提升了对整体系统数据采集的可维护性和便捷性。数据展示通过 Prometheus 与 Grafana 方式通过自定义图表清晰地呈现了采集到的数据，同时这两个组件提升了整套系统的可扩展性，降低了用户使用的上手门槛。因此，在系统架构方面也主要为上述三层架构：内核态程序层，用户态程序处理层，可视化层。最终测试环节，通过对 eBPF 程序性能测试和 ebpf_exporter 程序资源占用测试，验证了整套系统的低资源消耗。

综上所述，本系统基于 eBPF 程序实现了简单的对 ELF 文件加载过程的分析，具有一定的使用价值，但该程序仍存在一些不足之处，例如 ELF 文件加载过程更深层次的数据分析还有提升空间， ebpf 采集信息的性能还需继续优化，可视化性能存在一定的弊端。在日后的学习中我仍会不断积累技术经验，提升综合实力，进一步优化整体系统的功能与性能。

参考文献

[1]毛德操. Linux 内核源代码情景分析 [M]. 杭州:浙江大学出版社 , 2001.

[2]Brendan Gregg, BPF Performance Tools, Addison-Wesley, 2019. eBPF 的工作原理及其在性能调优和监控中的应用。

[3]Cilium Project, BPF and XDP Reference Guide, 2019. eBPF 和 XDP 的概念、架构和应用场景。

[4]杨广翔.ELF 格式可执行程序的代码嵌入技术 [J]. 程序员 ,2008(3):104-106.

[5]RYAN O'NEILL. Linux 二进制分析 [M]. 棣琦,译.北京：人民邮电出版社 , 2017.

[6]Alexei Starovoitov, eBPF—Rethinking the Linux Kernel, Linux Plumbers Conference, 2014. eBPF 的核心设计理念和早期实现.

[7]李明, 基于 eBPF 的网络性能监控系统设计与实现, 计算机工程与应用, 2020. eBPF 在网络性能监控中的实际应用案例。

[8]欧涅.基于 eBPF 的网络数据包捕获与分析系统的设计与实现 [D].武汉：华中科技大学，2020.

[9]基于操作系统 eBPF 在云原生环境下的技术研究. 高巍.电子技术与软件工程,2022(17)

[10]FALCO.Falco[EB/OL].(2024-01-21).https://falco.org/.

[11]eBPF 官方社区（ebpf.io），探索 eBPF 在内核中的架构

[12]钟盈炯. 基于 Prometheus+Grafana 实现新华全媒新闻服务平台统一运维监控[J].中国传媒科技，2023(1)：154-158

[13]Docker 技术实现分析. 陈清金;陈存香;张岩.信息通信技术,2015(02)

[14]赵炯. Linux 内核完全注释 [M]. 北京 : 机械工业出版社 ,2007.

[15]Linux 系统下的 ELF 文件分析. 朱裕禄.电脑知识与技术,2006(26)

[16]Daniel Borkmann, eBPF: Enabling Flexible and Efficient Kernel Dynamics, USENIX Annual Technical Conference, 2018. eBPF 在操作系统内核行为分析和动态跟踪方面的应用前景。

[17]基于 eBPF 的系统监控和故障检测方法.王锐.中国移动通信集团广东有限公司,2024(12)

[18]基于 Prometheus 监控系统的设计与实现. 陈雪娟.数字技术与应用,2025(02)

[19]可执行文件格式分析与应用. 杨新柱. 北京邮电大学.2009(05)

[20]静态链接动态库的 ELF 文件软件设计. 陈宇;廖湘科;李慰.微计算机信息,2008(09)

[21]基于深度强化学习的恶意 ELF 文件检测对抗方法. 孙贺;张博成;耿嘉炫;吴迪;王俊峰;方智阳.网络与信息安全学报,2024(05)

[22]基于 Docker 的应用软件虚拟化研究. 马越;黄刚.软件,2015(03)

致　　谢

首先，我真诚地感谢每一位老师认真耐心地看到这里。这一章节是本论文的最后一章内容，本文的最后一个句号同时也代表着四年本科生活接近尾声，四年时光的磨砺，使我从一个懵懂少年转变为独当一面的青年。

其次，我由衷地感谢我的毕业设计指导老师，是您帮助我成功克服了对 Linux 内核的恐惧并通过本毕设使我对 eBPF 技术产生浓烈的兴趣。您严谨的教学态度，丰富的 Linux 内核经验，加深了我求知的理念。本次作为您毕设课题的学生，感受到您扎实的学科知识和循循善诱的教学理念，您恰到好处的引导提高了我对本研究的独立性和探索能力。您审查并帮助我纠错时的耐心与细心深深地烙印在我的心里，深化了我对于做人的态度。

接着，感谢我的母校西邮为我们提供了强大的师资力量，拓宽了我对知识和技术的广度，挖掘了我对研究的深度，蒂固了我与老师间的师生情谊，加深了我与同学间的友情。

最后感谢我的同学在我遇到苦难时主动积极地热心帮助我脱离困难，感谢父母家人对我的肯定和信任使我在生活中充满活力，我会继续努力，在生活中激情四射，在工作中兢兢业业。