



西安邮电大学

毕业设计（论文）

题目： 基于 eBPF 的动态链接器运行机制分析
及可视化系统设计与实现

学院： 计算机学院

专业： 网络工程

班级： 网络 2102

学号： 04212069

学生姓名： 袁野

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 的动态链接器运行机制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：_____ 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	计算机学院
题目名称	基于 eBPF 的动态链接器运行机制分析及可视化系统设计与实现				
题目来源	<input checked="" type="checkbox"/> 教师科研课题 <input type="checkbox"/> 教师专业实践 <input type="checkbox"/> 其他				
题目类型	<input type="checkbox"/> 艺术作品 <input type="checkbox"/> 硬件设计 <input checked="" type="checkbox"/> 软件设计 <input type="checkbox"/> 论文				
题目分类	<input checked="" type="checkbox"/> 工程实践 <input type="checkbox"/> 社会调查 <input type="checkbox"/> 实习 <input type="checkbox"/> 实验 <input type="checkbox"/> 其他				
题目简述	<p>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 文件的动态链接是直接相关的，分析动态链接器的运行机制，可以深入了解动态链接 ELF 目标文件的关键信息及其在程序加载、运行中的作用，对分析和研究软件开发工具链中 ELF 文件的编译和链接、库的组织等有重要意义。</p> <p>本课题就是要基于 eBPF 工具深入 Linux 内核，在动态链接器运行的关键节点设置观测点，分析其运行的主要过程，收集动态链接器运行的关键信息，并对这些信息展示，从而为理解 ELF 文件中动态链接相关信息提供帮助。</p>				
对学生知识与能力要求	<p>1.熟悉 Linux 操作系统。</p> <p>2.熟悉 eBPF 框架，能够利用 eBPF 程序进行内核监控程序的开发。</p> <p>3.了解动态链接器的基本运行过程。</p> <p>4.熟悉 WEB 开发，将最终的观测数据进行可视化展示。</p>				
具体任务以及预期目标	<p>本课题的具体任务包括：</p> <p>在 Linux 系统上开发基于 eBPF 的动态链接器运行过程的监控程序，观测动态链接器的运行过程，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。</p> <p>具体的预期目标及成果形式包括：</p> <p>1.Linux 系统中动态链接器运行过程的分析，给出分析文档 1 份</p> <p>2.使用 eBPF 框架，开发一个系统，实现动态链接器运行过程中重要环节的信息提取。</p> <p>3.开发 web 前后端，实现信息的可视化。</p>				
时间进度	<p>2024 年 11 月 20 日-11 月 24 日：完成毕业设计选题</p> <p>2024 年 11 月 25 日-2025 年 1 月 10 日：提交开题报告，前期检查</p> <p>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</p> <p>2025 年 3 月 30 日-5 月 17 日：完成设计实现代码，进行代码验收</p> <p>2025 年 4 月 1 日-5 月 25 日：撰写毕业论文</p> <p>2025 年 5 月 26 日 6 月 6 日：完善毕业论文，进行论文答辩。</p>				

专业负责人审核意见	签字：年 月 日		
系（教研室）主任签字	年 月 日	主管院长签字	年 月 日

西安邮电大学本科毕业设计（论文）开题报告

学生姓名	袁野	学号	04212069	专业班级	网络 2102
指导教师	王亚刚	题目	基于 eBPF 的动态链接器运行机制分析及可视化系统设计与实现		
选题目的					
<p>在计算机科学领域中,对于动态链接器的理解和运行机制分析对于系统优化和软件开发至关重要。传统的动态链接器运行机制分析通常需要通过代码阅读和调试,这种方式存在一些局限性,无法提供足够的直观性和易用性,限制了开发者对于动态链接器运行机制的深入学习和理解^[1]。</p> <p>目前市面上存在一些工具,如GDB和Valgrind,能够对动态链接器进行分析。然而,这些工具往往存在使用难度大和功能局限性的问题^[2]。使用这些工具需要具有较强的编程和调试技能,这对于许多开发人员和研究者来说可能是一个挑战。此外,这些工具的功能可能不全面,无法满足所有的需求^[3]。</p> <p>随着eBPF（扩展的伯克利数据包过滤器）技术的不断发展和普及,基于eBPF的动态链接器运行机制分析及可视化系统设计与实现成为一种重要的解决方案。通过使用eBPF,我们可以实时地监控和分析动态链接器的运行状态,而无需修改程序或重启系统。这为开发者和研究者提供了更加灵活和便捷的分析方式^[4]。</p> <p>为了解决传统工具使用难度大和功能局限性的问题,我们计划设计和实现一个基于eBPF的动态链接器运行机制分析及可视化系统。使用eBPF作为核心技术,我们可以构建一个能够实时监控和分析动态链接器运行状态的系统。此外,通过结合前端可视化技术,我们可以将分析结果以图形的方式展示出来,提高了分析的直观性和易用性^[5]。</p> <p>在软件开发领域,对于动态链接器的深入分析和理解能够为系统优化和程序调试提供重要的依据。然而,传统的分析工具通常需要专业的技能支持,使用起来复杂且功能有限^[6]。基于eBPF的动态链接器运行机制分析及可视化系统的设计与实现,可以提供实时的监控和分析功能,为开发者和研究者提供一个直观、高效的分析平台。通过该系统,用户可以进行实时的监控和分析,提高分析的深度和准确性^[7]。</p> <p>总而言之,基于eBPF的动态链接器运行机制分析及可视化系统设计与实现在软件开发和系统优化中具有重要的意义。它可以提供实时的监控和分析功能,加深对动态链接器运行机制的理解和记忆。它还可以为开发者和研究者提供一个直观、高效的分析平台,促进动态链接器运行机制的深入分析和理解。通过该系统的开发和应用,可以推动计算机科学领域的技术创新和进步,为软件开发和系统优化带来显著的改进。</p> <p>在我的设计方案中,主要采用一种分层架构,由内核态监控层、用户态数据处理层和图形化展示层三部分组成。</p> <p>在内核态监控层,系统将利用eBPF技术通过uprobe和uretprobe机制在用户态注入动态跟踪探针^[8],主要监控动态链接器相关的核心函数如dlopen、dlclose、dlsym,以全面捕获动态链接过程中的关键事件。并且设计扩展的链接事件数据结构,记录基础的库加载信息（路径、加载地址）,以及符号解析信息、加载标志参数等细节数据。</p> <p>在用户态数据处理层,系统将对内核态监控层返回的数据进行收集和整理,根据不同的事件类型存储不同的信息。同时可以对获取到的信息进行筛选,例如只关注特定程序的动态库</p>					

链接信息，避免其他程序的干扰。

在图形化展示层，系统将基于Qt5框架构建现代化界面，将用户态数据处理层整理的信息进行可视化展示。前端界面将包括多个不同视图表格^[9]，例如：时间轴视图、动态库加载视图、符号解析视图、动态库卸载视图。系统将实时把获取到的事件的信息显示在界面上，方便用户查看。

总的来说，基于eBPF的动态链接器运行机制分析及可视化系统将为软件开发者，系统管理员和研究人员提供一个强大而易用的工具，帮助他们更好地理解动态链接器的运行机制，从而进行更有效的系统优化和软件开发。

参考文献：

- [1] 马云峰.动态链接技术研究[D].北京理工大学，2011.
- [2] 张志强、陈艳、张亮等.GDB 调试器的使用与原理分析[J].电子测试，2020，36(02):1-5.
- [3] 李晓明、郭煜、李志鹏.使用Valgrind调试器进行内存泄漏检测[J].软件开发与应用，2020，37(12):23-26.
- [4] 赵明、王鑫、李嘉诚.基于eBPF的Linux内核性能分析工具研究[J].计算机科学，2020，47(09):8-15.
- [5] 孙寅春、郭振江、詹志宏等.全栈Web开发基础与实践[M].北京:清华大学出版社，2018.
- [6] 赵芳、胡晓亮、杨翔.动态链接技术在嵌入式Linux中的应用[J].电子设计应用，2019，47(07):98-101.
- [7] 何明、王书宏、张伟.基于eBPF的系统性能分析与优化技术[J].计算机系统应用，2020，29(06):1-7.
- [8] 曹洪卫、陈福霖.Linux性能观测新工具: eBPF[J]. 软件, 2019，40(11): 127-131.
- [9] 张俊杰、陈松林、刘轶男等. 基于Echarts的数据可视化设计与实现[J]. 计算机编程技巧与维护, 2020(8): 16-18.

前期基础

1.已学课程

《数据结构》、《计算机组成原理》、《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》、《高级语言程序设计》

2.掌握的工具

Visual Studio Code 开发工具、Git（分布式版本控制系统）、C++编程语言(后端开发语言)、QT（前端开发工具）。

3. 资料积累

《C++ Primer 中文版》、《QT5 C++开发指南》

4.软硬件条件

Archlinux 系统、Visual Studio Code 1.96 版本、笔记本电脑一台

要研究和解决的问题

1.动态链接器运行过程的监控与信息提取：在 Linux 操作系统中，动态链接器负责加载和链接动态库，这一过程对于理解软件开发工具链中 ELF 文件的运行机制至关重要。研究的关键问题是如何在不影响系统性能的前提下，使用 eBPF 工具有效地监控动态链接器的运行过程，提取出涉及动态链接的关键信息。

2.eBPF 程序的开发与内核监控：eBPF 作为一个强大且灵活的 Linux 内核监控工具，如何编写和部署 eBPF 程序以捕获动态链接器的运行数据，并确保其准确性与实时性，是一个重要的

技术挑战。

3.数据的整理与分析：收集到的动态链接器运行数据需要进行有效的整理和分析，以提取出对 ELF 文件有重要意义的动态链接信息。这包括识别动态链接过程中涉及的符号解析等关键环节。

信息的可视化展示：将动态链接器的运行机制和关键数据进行可视化展示是研究的另一重点。如何通过 Qt 开发的桌面应用程序直观地展示这些信息，使其对软件开发者和研究人员具有实际的参考价值，是需要解决的问题。

工作思路和方案

(1) 需求分析：首先，明确系统的目标和功能需求，包括动态链接器监控的具体信息需求以及最终展示的形式。通过查阅相关文献和网页资料，获取对 eBPF 监控程序和可视化系统的具体期望。深入调研 Linux 动态链接器 (ld.so) 的工作机制和关键函数调用流程，确定需要监控的关键事件点，如库加载/卸载、符号解析、重定位等过程。分析动态链接性能指标和调试信息，包括加载时间、符号解析、依赖关系等。同时，评估系统的技术可行性和潜在限制，如 eBPF 对内核版本的依赖、权限要求以及性能开销等因素。

(2) BPF 程序开发：利用 C++语言和 eBPF 技术，开发能够在动态链接器运行的关键节点处插入探针的程序。这些探针将负责捕获动态链接过程中的重要数据，如符号解析、库加载等。eBPF 程序需要确保低开销和高效的内核数据传输。具体实现中，将使用 libbpf 库简化 eBPF 程序的编写和部署。设计 uprobe 和 uretprobe 探针，分别注入到 dlopen、dlclose、dlsym 等关键函数的入口点和返回点，全面捕获函数参数和返回值。设计高效的数据结构用于事件传递，确保在高频事件发生时不会造成性能瓶颈。实现内核态和用户态之间的高效数据传输机制，通过 perf 环形缓冲区实现低延迟数据交换。

(3) 用户空间程序开发：通过 eBPF 探针收集到的动态链接数据，将被传输到用户空间进行进一步处理。使用 C++编写的数据处理程序负责对原始数据进行整理、过滤和分析，以提取出有用的信息。建立高效的事件处理管道，包括事件接收、解析、分类和存储等环节，实现对提取到的数据信息的妥善处理。

(4) Qt5 桌面应用程序开发：使用 C++语言和 Qt5 框架开发桌面应用程序，负责将分析后的数据进行可视化展示。Qt5 提供了丰富的 UI 组件和图形库，可以用于构建直观的用户界面和数据展示图表。采用模块化架构设计。开发多视图界面，使用列表清晰地展示监测到的不同事件的信息。设计实时数据更新机制，通过信号-槽机制实现前端界面与后端数据的连接。

(5) 系统测试与优化：对开发的监控和可视化系统进行综合测试，确保其性能和稳定性。使用自己编写的测试程序来进行测试，测试程序将显式加载动态库，例如数学库 libm.so，并调用其中的函数，进行符号解析，最后卸载动态库。这样便可以产生动态链接信息以供监测程序监测。同时，考虑系统的扩展性以支持未来的功能拓展和定制需求。观察是否获取到对应的信息并在前端界面正确展示。最后，需要编写详细的用户文档和技术文档，便于系统的使用和后续维护。

毕业论文进度计划

2024.11.20-2024.11.25	确认选题，明确论文内容，查找现有资料和文献。
2024.11.25-2025.01.10	提交开题报告，前期检查。
2025.01.11-2025.03.29	完成环境搭建，并完成前后台接口设计，中期检查。
2025.01.11-2025.03.29	完成环境搭建，并完成前后台接口设计，中期检查。
2025.03.30-2025.05.17	完成设计实现代码，进行代码验收。
2025.04.01-2024.05.25	撰写毕业论文。
2024.05.26-2025.06.06	完善毕业论文，进行论文答辩。

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

学生姓名	袁野	性别	男	学号	04212069	专业 班级	网络工程 网络 2102		
课题名称	基于 eBPF 的动态链接器运行机制分析及可视化系统设计与实现								
指导教师意见	支撑指标点/赋分	3-2/20	4-2/20	5-3/10	7-2/10	8-2/10	11-2/10	12-2/20	合计
	得分								
	指导教师(签字): 年 月 日								
评阅教师意见	支撑指标点/赋分								合计
	得分								
	评阅教师(签字): 年 月 日								
验收小组意见	支撑指标点/赋分								合计
	得分								
	验收小组组长(签字): 年 月 日								
答辩小组意见	支撑指标点/赋分								合计
	得分								
	答辩小组组长(签字): 年 月 日								
学生总评成绩	评分比例	指导教师(20%)		评阅教师(30%)		验收小组(20%)		答辩小组(30%)	
	评分								
	毕业论文(设计)最终等级制成绩(优秀、良好、中等、及格、不及格)								
答辩委员会意见	学院答辩委员会主任(签字、学院盖章): 年 月 日								

摘 要

Linux 动态链接器是现代操作系统不可或缺的组成部分，负责在程序运行时解析和加载共享库依赖、执行符号重定位等关键任务，其内部机制复杂且对程序性能有直接影响。深入理解动态链接器的工作原理对于系统优化和问题排查至关重要。

本课题的主要工作内容如下：

（1） eBPF 内核态监控程序开发：

设计并实现了基于 eBPF 的内核态监控程序。利用 libbpf 库，该程序能够无侵入式地挂载到 Linux 动态链接器的关键函数点，精确捕获动态库加载、符号解析等核心事件与数据，并通过环形缓冲区（Ring Buffer）高效地将信息传递至用户空间。

（2） 用户空间数据处理程序开发：

开发了 C++ 用户空间应用程序。该程序通过 libbpf 库负责管理 eBPF 程序的生命周期，接收并解析来自内核的原始动态链接事件数据。它将这些数据进行结构化处理，并准备好用于前端展示的数据格式。

（3） Qt5 可视化前端程序开发：

构建了一个基于 Qt5 框架的图形用户界面。该前端通过 Qt 的信号与槽机制接收来自用户空间程序的数据，并利用 QTableView、QGraphicsView 等组件，将复杂的动态链接过程以实时、直观的方式（如事件纵览、事件时间线）呈现给用户，实现了对动态链接行为的可视化分析。

关键词：eBPF；动态链接器；Qt5

ABSTRACT

The Linux dynamic linker is an indispensable component of modern operating systems, responsible for critical tasks such as resolving and loading shared library dependencies and performing symbol relocations during program runtime. Its internal mechanisms are complex and directly impact program performance. A deep understanding of the dynamic linker's working principles is crucial for system optimization and troubleshooting.

The main work of this thesis includes the following aspects:

(1) Development of an eBPF Kernel-Space Monitoring Program:

An eBPF-based kernel-space monitoring program was designed and implemented. Utilizing the libbpf library, this program can non-intrusively hook into key function points of the Linux dynamic linker, accurately capturing core events and data such as dynamic library loading and symbol resolution. It efficiently transmits this information to user space via a Ring Buffer.

(2) Development of a User-Space Data Processing Program:

A C++ user-space application was developed. This program, through the libbpf library, is responsible for managing the lifecycle of the eBPF program, and for receiving and parsing the raw dynamic linking event data from the kernel. It processes this data into a structured format, preparing it for presentation by the frontend.

(3) Development of a Qt5 Visualization Frontend Program:

A graphical user interface (GUI) was built based on the Qt5 framework. Through Qt's signal and slot mechanism, the frontend receives data from the user-space application. It utilizes components like QTableView and QGraphicsView to present the complex dynamic linking process in a real-time and intuitive manner (such as event overview and event timeline), achieving visual analysis of dynamic linking behavior.

Key words: eBPF; Dynamic Linker; Qt5

目 录

第 1 章 绪论	1
1.1 概述	1
1.2 相关研究	1
1.3 内容安排	2
第 2 章 相关技术	4
2.1 eBPF 技术	4
2.1.1 概述	4
2.1.2 工作原理	5
2.1.3 libbpf 库	6
2.2 动态链接器	6
2.2.1 概述	7
2.2.2 动态链接库	7
2.2.3 符号解析与重定位	7
2.3 Qt5	8
2.3.1 概述	8
2.3.2 数据可视化常用组件	8
第 3 章 系统设计	10
3.1 信息处理设计	10
3.2 系统架构整体设计	11
3.2.1 架构分层与层级关系	11
3.2.2 内核态-用户态交互设计	12
3.3 系统架构组件设计	13
3.3.1 BPF 程序设计	13
3.3.2 用户空间程序设计	14
3.3.3 Qt5 可视化程序设计	15
第 4 章 系统实现	17
4.1 BPF 程序实现	17
4.2 用户空间程序实现	18
4.3 Qt5 可视化程序实现	21
第 5 章 系统测试	25
5.1 测试环境	25

5.2 功能测试	25
结 论	28
参考文献	30
致 谢	31

第 1 章 绪论

1.1 概述

在计算机软件发展的历史长河中，程序的构建和执行方式经历了从静态链接到动态链接的深刻变革。早期的程序开发通常采用静态链接的方式，即将程序代码所需的所有库函数在编译链接阶段完全合并到可执行文件中。这种方式虽然简化了程序的部署，但同时也带来了诸多问题，例如可执行文件体积庞大、内存占用高，以及库函数的更新和维护困难等。随着软件规模的不断扩大和复杂性的日益增加，静态链接的局限性日益凸显^[3]。

为了克服静态链接的不足，动态链接技术应运而生。动态链接的核心思想是将程序依赖的库函数在程序运行时才进行加载和链接，而不是在编译链接阶段就将其嵌入到可执行文件中^[4]。这种方式显著减小了可执行文件的大小，多个程序可以共享同一份动态库的内存副本，从而降低了系统资源的消耗。此外，当动态库需要更新时，只需要替换库文件，而无需重新编译链接依赖该库的所有程序，极大地提高了软件的维护性和可升级性。动态链接技术已经成为现代操作系统和应用程序开发中不可或缺的关键组成部分^[3]。

扩展伯克利封包过滤器（extended Berkeley Packet Filter，简称 eBPF）是一种强大的内核观测技术^[1]。eBPF 最初被设计用于网络数据包的过滤和监控，但其灵活的架构和强大的功能使其逐渐扩展到内核跟踪、性能分析、安全监控等多个领域^[2]。eBPF 允许用户在内核空间安全地运行自定义的沙箱程序，从而能够以极低的性能开销对内核行为进行精细的观测。

利用 eBPF 技术，开发者可以深入 Linux 内核，观测动态链接器在程序加载和动态库链接过程中的行为，提取关键的动态链接信息，并使用可视化前端界面展示出提取的信息。这能够加深开发者对 Linux 动态链接器运行机制以及程序的理解。

1.2 相关研究

随着 Linux 系统在服务器、云计算以及嵌入式设备等领域的广泛应用，对系统运行时行为的深度洞察与监控变得日益重要。在动态链接机制下，程序在运行时加载和卸载动态库、解析符号、进行地址重定位等过程，不仅是程序执行的关键环节，也可能成为安全攻击（如动态链接库劫持）的潜在入口。目前有一些传统的系统监控工具，例如 ptrace、LD_PRELOAD 环境变量以及 SystemTap/Perf 等内核跟踪工具，虽然能够提供一些运行时信息，但它们往往存在性能开销大、需要修改环境变量、部署和维护复杂，或难以提供足够细粒度信息的局限性。这些方

法在监控动态链接过程中的库加载、卸载、符号重定位等方面表现出明显的不足，限制了对程序运行时行为的全面理解和安全威胁的有效检测。

近年来，eBPF（extended Berkeley Packet Filter）技术的兴起，为解决上述问题提供了全新的思路和强大的能力。eBPF 允许在 Linux 内核中执行沙盒程序，凭借其安全、高效、无需修改内核源码的特性，在系统可观测性、性能分析和安全监控等领域展现出巨大潜力。其核心优势在于能够在不影响内核稳定性和性能的前提下，动态地在内核层面的关键点（如系统调用、函数入口/出口、tracepoints 等）注入自定义逻辑，从而获取传统方法难以触及的细粒度信息。在动态链接器监控方面，eBPF 的研究和应用主要集中在动态库加载/卸载事件的捕获和符号重定位事件的追踪。通过附加到内核中负责加载和卸载共享库的系统调用（如 `execve`、`openat`、`mmap`）或特定内核函数上，eBPF 可以精确追踪动态库的生命周期事件。同时，通过在动态链接器（一般是 `ld.so`）的用户态函数入口点（`uprobes`）设置探针，eBPF 能够捕获符号解析和重定位过程中的详细信息，包括符号名称、重定位类型、原始地址和最终地址等关键数据。这对于理解程序运行时行为、分析性能瓶颈以及检测恶意代码注入都具有重要意义。

基于 eBPF 的动态链接器监控不仅有助于深入理解程序行为，还在安全监控与威胁检测方面发挥着关键作用^[13]。由于动态链接过程是许多攻击（如共享库注入、符号劫持）的切入点，eBPF 通过对加载/卸载、符号重定位事件的实时监控，可以建立行为基线并检测异常行为。例如，如果一个非特权进程尝试加载一个位于非标准路径的可疑库，或者在不寻常的时间进行大量的符号重定位，都可能被 eBPF 程序捕获并触发警报。传统的静态分析工具难以涵盖动态加载的库，而 eBPF 可以在内核层面透明地捕获程序运行时的动态链接行为，这对于软件供应链安全分析至关重要。目前已有许多基于 eBPF 的开源工具和框架，例如 BCC（BPF Compiler Collection）^[10] 和 `bpftool`，以及 Cilium、Falco、Tracee 等安全和可观测性平台，它们虽然不一定直接针对“动态链接器”进行全面监控，但其提供的能力和监控范式为本论文提供了宝贵的参考和基础。这些工具的实践经验表明，eBPF 技术在实现高性能、细粒度系统监控方面的强大潜力，为本课题利用 eBPF 技术实时监测 Linux 动态链接器核心事件，并结合 Qt5 前端进行可视化展示提供了可行性与坚实基础。

1.3 内容安排

本论文主要内容为设计并实现一个基于 eBPF 的程序来对动态链接器运行机制进行分析并进行信息的可视化展示。利用 eBPF 技术在捕获动态链接器在 ELF 文件加载、符号查找、库依赖解析等关键阶段的运行时数据。通过 `libbpf` 库与用户态控制程序交互，将采集到的数据进行处理和结构化。最终，借助 Qt5 图形用户界面库，以实时、直观的方式将动态链接过程可视化。

论文的主要内容安排如下：

第一章讲述了 eBPF 的生态和发展以及动态链接器的关键作用。第二章介绍了所用到的相关技术，包括 eBPF 技术、libbpf 库、Linux 动态链接机制以及 Qt5 可视化框架。第三章分析和规划了系统的整体功能模块。第四章重点阐述了系统的整体架构设计。第五章详细描述了系统各核心功能的具体实现过程。第六章使用自己编写的程序程序对所开发的系统进行了功能测试。

第 2 章 相关技术

2.1 eBPF 技术

2.1.1 概述

eBPF（扩展型伯克利包过滤器）作为现代 Linux 系统的革命性内核技术，实现了用户定义程序的安全内核态执行范式^[2]。其创新之处在于构建了基于寄存器的虚拟机架构，使得非特权用户能够在受控环境中扩展内核功能。2014 年，Linux 社区借鉴了 Solaris DTrace 的动态追踪理念，针对 Linux 内核模块开发门槛高、稳定性风险大的痛点，Linux 3.18 版本首次引入 eBPF 架构，并在 4.x 系列内核中逐步完善其功能集合^[6]。相较于传统 BPF 单一的网络层过滤能力，eBPF 通过以下多维创新实现了技术突破：

- （1）执行域扩展：支持网络流量处理、性能事件分析等多维观测场景。
- （2）安全模型：引入验证器（Verifier）确保程序内存安全，防止内核崩溃。
- （3）交互机制：提供高效的内核-用户空间数据通道（BPF Maps）。
- （4）开发范式：采用 LLVM 中间表示编译技术，支持 C 语言子集编程。

有一些类似 eBPF 的工具。例如，SystemTap 是一种开源工具，可以帮助用户收集 Linux 内核的运行时数据。它通过动态加载内核模块来实现这一功能，类似于 eBPF。另外，DTrace 是一种动态跟踪和分析工具，可以用于收集系统的运行时数据，类似于 eBPF 和 SystemTap。下面的表格展示了这三种工具的不同之处：

表 1 eBPF、SystemTap 和 DTrace 的对比表格

工具	eBPF	SystemTap	DTrace
定位	内核技术，可用于多种应用场景	内核模块	动态跟踪和分析工具
工作原理	动态加载和执行无损编译过的代码	动态加载内核模块	动态插接分析器，通过 probe 获取数据并进行分析
常见用途	网络监控、安全过滤、性能分析等	系统性能分析、故障诊断等	系统性能分析、故障诊断等
优点	灵活、安全、可用于多种应用场景	功能强大、可视化界面	功能强大、高性能、支持多种编程语言
缺点	学习曲线高，安全性依赖于编译器的正确性	学习曲线高，安全性依赖于内核模块的正确性	配置复杂，对系统性能影响较大

2.1.2 工作原理

eBPF 技术体系构建了一个分层的安全执行环境，其核心运行逻辑可解构为程序注入、静态验证、动态执行三个阶段。用户态管理进程通过特定系统调用接口发起加载请求，触发内核执行内存隔离传输机制。该过程采用写时复制技术确保用户空间字节码安全迁移至内核地址空间，同时建立程序与目标事件的绑定关系，例如关联到指定的 kprobe 插桩点。

在代码验证阶段，LLVM 工具链将高级语言源码编译为平台无关字节码后，内核验证器实施多维度安全审查。首先进行控制流完整性检查，确保指令序列不存在不可达代码段或超过允许的循环嵌套深度。接着执行指针别名分析，预防非法内存访问行为，同时过滤非白名单内的特权指令。最后评估资源消耗，限定栈空间使用不超过 512 字节且指令总数低于百万量级，从而构建安全执行沙箱。

通过验证的字节码进入自适应执行阶段，经历两级性能优化过程。JIT 编译器首先将中间表示转换为目标架构原生指令集，例如将通用字节码映射为 x86_64 的 SIMD 指令以提升执行效率。优化后的程序通过事件驱动模型激活，典型触发条件包括内核函数边界探测、用户空间函数调用拦截以及网络数据平面事件捕获。这种设计使得 eBPF 能够以接近原生代码的性能进行实时系统观测。

eBPF 映射子系统采用键值存储范式实现跨调用周期状态保持，其技术实现涉及多种数据结构变体。内核通过原子化操作原语确保哈希表、数组等结构的并发访问安全，用户空间则借助文件描述符标识符与映射对象交互。这种设计既保证了内核态数据的高效存取，又提供了灵活的用户态数据分析接口。

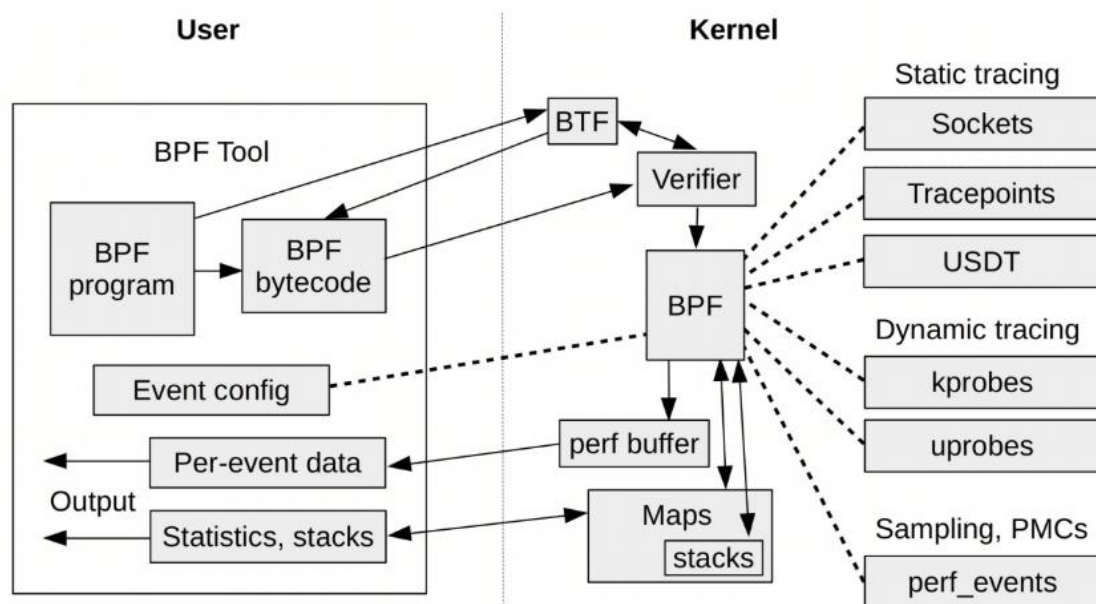


图 2.1 BPF 追踪工具工作流程

上图是一个追踪工具的简化工作流程。在图的示例中，用户空间里的 BPF 工具具有一个 BPF 程序，它被转换为了 BPF 字节码。紧接着字节码被传输到内核部分的验证器（verifier），这个过程可能用到 BTF（byte type format）用于提供结构体信息。验证通过后可以连接到不同的事件源（图右侧），这些事件是在用户空间就已经决定好的，并且在执行过程中，可以通过 perf buffer 或者 map 进行内核态到用户空间的传输（前者单向，后者双向）。

相较于传统内核模块开发模式，eBPF 通过沙箱隔离机制限制程序仅能访问经批准的内核接口。验证器在预执行阶段实施静态代码分析，JIT 编译器动态插入内存保护指令，形成纵深防御体系。能力分级模型进一步细化了权限控制，根据 CAP_BPF 标志集实施最小特权原则，从多维度保障系统稳定性。

2.1.3 libbpf 库

libbpf 作为现代 BPF 生态的核心组件，构建了用户态与内核态协同工作的桥梁。该库采用分层架构设计，底层封装 bpf 系统调用原语，上层提供类型安全的抽象接口，显著降低了 BPF 应用的开发复杂度^[11]。其架构创新主要体现在三个维度：

在接口抽象层面，libbpf 通过双模式 API 满足不同场景需求。底层 API 直接映射 bpf 系统调用，为开发者提供精确控制 BPF 对象生命周期的能力。高层 API 则封装了通用操作流，支持以声明式方法管理 BPF 程序的加载-验证-挂载全生命周期。

针对可移植性挑战，libbpf 集成 CO-RE（Compile Once-Run Everywhere）技术框架。该机制通过 BTF（BPF Type Format）类型信息实现跨内核版本兼容，允许开发者构建适应不同内核数据布局的 BPF 程序。具体实现依赖 LLVM 编译器的 `_builtin_preserve_access_index` 属性记录结构体偏移，配合 libbpf 的重定位加载器在运行时动态调整内存访问指令。

在开发范式革新方面，libbpf 引入 BPF 骨架生成技术。通过 bpftool 工具解析目标文件的 ELF 格式元数据，自动生成包含字节码与映射描述符的结构化接口。这种设计将传统的分散式资源管理转化为类型安全的对象操作。

libbpf 通过状态机模型管理 BPF 程序运行阶段，提供从对象加载到资源释放的全流程控制：

- （1）初始化阶段解析 ELF 段结构，提取字节码与重定位信息。
- （2）加载时执行验证器预检与 JIT 编译，确保代码符合内核安全策略。
- （3）挂载阶段动态配置探针点，支持 kprobe/tracepoint 等多类事件源。
- （4）卸载过程采用引用计数机制，保证资源有序释放。

2.2 动态链接器

2.2.1 概述

动态链接器是现代操作系统的核心组件之一，主要负责在程序运行时加载程序所依赖的共享库（也称为动态库），并将程序中对库函数的符号引用解析为库中实际的函数地址，最终完成程序的链接过程^[12]。在 Linux 系统中，动态链接器通常是 `/lib/ld-linux.so.*` 文件。当一个使用了动态库的可执行程序被启动时，内核会首先加载动态链接器到内存中，然后由动态链接器负责加载程序本身以及其依赖的所有动态库。

2.2.2 动态链接库

动态链接器的工作过程可以概括为以下几个主要步骤^[7]：

程序启动：当用户执行一个使用了动态库的程序时，操作系统内核会识别出这是一个需要动态链接的可执行文件。

（2）内核加载动态链接器：内核不会直接加载应用程序，而是首先加载指定的动态链接器到进程的地址空间。

（3）动态链接器加载程序本身：动态链接器被激活后，会首先加载程序的可执行文件到内存中。

（4）检查依赖库列表：动态链接器会读取程序文件中的特定段，获取程序依赖的动态库列表。

（5）加载依赖的动态库：对于每个依赖的动态库，动态链接器会在文件系统中查找库文件（通常根据预设的搜索路径，如 `/lib`、`/usr/lib` 等，以及 `LD_LIBRARY_PATH` 环境变量），并将其加载到进程的地址空间。如果一个库还依赖于其他库，这个过程会递归进行。

（5）符号解析和重定位：这是动态链接的核心步骤。程序和各个动态库中可能存在对外部符号（例如函数、全局变量）的引用。动态链接器需要找到这些符号在被加载的动态库中的实际地址，并将程序和动态库中的引用指向这些地址。

（6）控制权交给应用程序：当所有必要的动态库都被加载和链接完成后，动态链接器会将程序的控制权交给应用程序的入口点，程序开始正常执行。

2.2.3 符号解析与重定位

当程序调用一个定义在动态库中的函数，或者访问一个定义在动态库中的全局变量时，程序本身在编译链接时并不知道这些符号的实际内存地址。动态链接的关键在于符号解析（Symbol Resolution）和重定位（Relocation）^[4]。

符号解析是指在运行时，动态链接器根据程序或动态库中对符号的名称引用，在已加载的动态库中找到该符号的定义。这个过程通常涉及到查找符号表。每个动态库都维护着一个符号表，其中包含了该库导出的所有符号的名称及其在库中

的地址。当需要解析一个符号时，动态链接器会搜索已加载的动态库的符号表，找到与引用名称匹配的符号。

重定位是指在符号解析之后，动态链接器需要修改程序和动态库中对已解析符号的引用，将其指向符号在内存中的实际地址。这是因为动态库被加载到内存中的地址在每次程序运行时都可能不同，因此编译时确定的地址是无效的。重定位的过程会修改程序和动态库中的特定位置，将占位符地址替换为符号的实际运行时地址。

2.3 Qt5

2.3.1 概述

Qt5 是一个跨平台的 C++ 应用程序开发框架，它提供了丰富的工具和库，用于创建具有图形用户界面（GUI）的应用程序，以及非 GUI 应用程序，例如控制台工具和服务端应用。Qt5 以其强大的功能、易用性和良好的跨平台性而闻名，支持 Windows、Linux、macOS、Android 和 iOS 等多个操作系统。Qt5 提供了包括窗口部件、图形、多媒体、网络、SQL 数据库、XML 处理、并发编程等在内的众多模块，极大地简化了应用程序的开发过程^[9]。

在数据可视化方面，Qt5 提供了强大的支持，其灵活的架构和丰富的组件使得开发者能够创建出美观且交互性强的数据展示界面^[14]。Qt5 的信号与槽机制为实现数据的动态更新和用户交互提供了便利^[5]。

2.3.2 数据可视化常用组件

Qt5 提供了许多强大的组件，可以用于创建各种类型的数据可视化界面，比如以下组件：

- **QWidget**：Qt 中所有用户界面对象的基础类。虽然 QWidget 本身并不直接提供高级的数据可视化功能，但它是构建复杂 UI 布局的基础，可以作为其他可视化组件的容器。
- **QGraphicsView 和 QGraphicsScene**：一个强大的 2D 图形框架，允许开发者创建自定义的图形和交互式可视化效果。QGraphicsScene 提供了一个用于管理大量 2D 图形项的表面，而 QGraphicsView 则提供了一个用于在屏幕上查看场景的窗口部件。这对于需要高度定制化的动态链接过程可视化非常有用。
- **Qt Charts**：Qt 的一个模块，专门用于创建各种常见的图表类型^[16]，例如柱状图、折线图、饼图等。虽然本毕业设计的主要目标是实时展示动态链接的过程，但 Qt Charts 可以在总结和分析动态链接数据时提供有价值的辅助可视化手段。

- **QTableView 和 QAbstractTableModel**：这两个类用于显示表格数据。QAbstractTableModel 是一个抽象模型，开发者可以继承它来提供自定义的数据模型，而 QTableView 则是一个用于显示该模型数据的视图。这对于展示动态链接过程中加载的库列表、符号信息等详细的表格数据非常合适。
- **QPainter**：一个底层的绘图类，提供了丰富的 API 用于绘制各种图形元素，例如线条、形状、文本等。对于需要进行精细控制和高度定制的可视化场景，QPainter 可以提供强大的支持。
- **信号与槽（Signals and Slots）**：Qt 框架中对象之间进行通信的核心机制。当一个特定事件发生时，对象会发出一个信号，而其他对象可以连接到这个信号，并在信号发出时执行相应的槽函数。在动态链接可视化系统中，用户空间程序可以通过信号将处理后的动态链接数据发送给 Qt5 前端，前端的槽函数接收到数据后更新 UI，实现实时的数据可视化。

第 3 章 系统设计

3.1 信息处理设计

本系统的核心在于对动态链接过程信息的捕获、处理和展示。因此，系统需要有一个合适的动态链接信息结构体和一个完整的动态链接信息处理流程。

动态链接信息数据结构如下：

表 2 动态链接信息数据结构

字段名称	类型	描述
timestamp	__u64	事件发生的时间戳（纳秒）
pid	__u32	进程 ID
uid	__u32	用户 ID
comm	char[]	进程名
lib_path	char[]	动态库路径
lib_addr	__u64	动态库加载地址或句柄
symbol_name	char[]	符号名称
event_type	int	事件类型
flags	int	dlopen 的标志（1:加载, 2:卸载, 3:符号解析）
symbol_addr	__u64	符号地址
result	int	操作结果

动态链接信息处理流程如下：

（1）**eBPF 程序挂载**：eBPF 程序将被挂载到 Linux 内核中与动态链接器操作相关的特定跟踪点或内核探测点。这些挂载点包括与加载共享库相关的函数，以及与符号解析和重定位相关的函数。

（2）**信息提取**：当动态链接器执行到这些挂载点时，eBPF 程序将被触发，并从内核上下文中提取关键信息。这些信息包括加载的库名称、库在内存中的起始地址、解析的符号名称、符号的原始地址和重定位后的地址、事件发生的时间戳以及相关的进程 ID 等。

（3）**数据存储到 eBPF 映射**：提取到的关键信息将被存储到预先创建的 eBPF 映射中。

（4）**用户空间程序数据读取**：用户空间应用程序（使用 C++编写）将利用 libbpf 库提供的 API 与内核中的 eBPF 程序进行交互。这包括打开 eBPF 程序、加载程序到内核、将程序附加到跟踪点，以及打开和读取 eBPF 缓冲区映射。

(5) 数据解析与处理：用户空间程序从 BPF 程序获取到动态库名字符串、内存地址数值、符号名称等。处理后的数据可以存储在用户空间程序内部的数据结构中。

(6) 数据传递给 Qt5 前端：用户空间程序需要将处理后的动态链接信息传递给 Qt5 前端进行可视化展示。通过 Qt 框架提供的信号与槽机制来实现这个功能。用户空间程序定义特定的信号，当有新的动态链接数据准备好时，就发出这些信号，并将数据作为信号的参数传递出去。

(7) Qt5 前端数据展示：Qt5 前端应用程序创建相应的槽函数来接收用户空间程序发出的信号。在槽函数中，前端程序将接收到的动态链接数据更新到用户界面上。

通过上面的动态链接信息结构体和动态链接信息处理流程，系统能够实现从内核动态链接事件的捕获到用户界面实时可视化的数据处理全过程。

3.2 系统架构整体设计

3.2.1 架构分层与层级关系

系统采用分层架构设计，主要分为三个层次：内核态监控层、用户态数据处理层和图形化展示层。这种分层设计有助于模块化开发、提高系统的可维护性和可扩展性。

- 内核态监控层：这是系统的最底层，主要包含 Linux 操作系统内核以及运行在其中的 eBPF 子系统。eBPF 监控程序作为内核层的一个组件，负责在内核态监测动态链接器的运行状态，捕获动态链接器进行动态库链接、卸载和符号解析的关键信息。
- 用户态数据处理层：位于内核态监控层之上，主要包含两个核心组件：BPF 控制与数据处理模块。该模块负责使用 libbpf 库加载和管理内核层的 eBPF 监控程序，并接收从内核传递过来的原始动态链接数据。同时，它还负责对这些原始数据进行解析、处理和初步分析，为上层的 Qt5 前端提供格式化的数据。
- 图形化展示层：这是系统的顶层，负责图形用户界面的展示。它使用 Qt5 框架开发，接收来自用户空间层处理后的动态链接数据，并通过 UI 组件（如表格）将这些数据以直观的方式呈现给用户。

这三个层次之间存在清晰的层级关系。Qt5 前端层依赖于用户空间层提供的数据，而用户空间层则依赖于内核层运行的 eBPF 程序来获取数据。数据流向是从内核层到用户空间层，再到 Qt5 前端层。控制流向则通常是从用户空间层到内核层，例如用户空间程序通过 libbpf 控制 eBPF 程序的加载和卸载。

下表描述了系统的架构层次及其主要职责：

表 3 内核态监控层、用户态数据处理层和图形化展示层及其职责

层次名称	主要组件	主要职责
内核态监控层	eBPF 子系统, eBPF 监控程序	在内核态监测动态链接器的运行, 捕获关键事件和数据
用户态数据处理层	BPF 控制与数据处理模块 (使用 libbpf)	加载、管理 eBPF 监控程序, 接收和处理来自内核的原始数据, 为 Qt5 前端提供格式化数据
图形化展示层	Qt5 图形用户界面	接收来自用户空间层处理后的数据, 并通过图形化的方式实时展示动态链接的过程和关键信息, 提供用户交互功能

下图展示了系统的整体架构和数据流动:

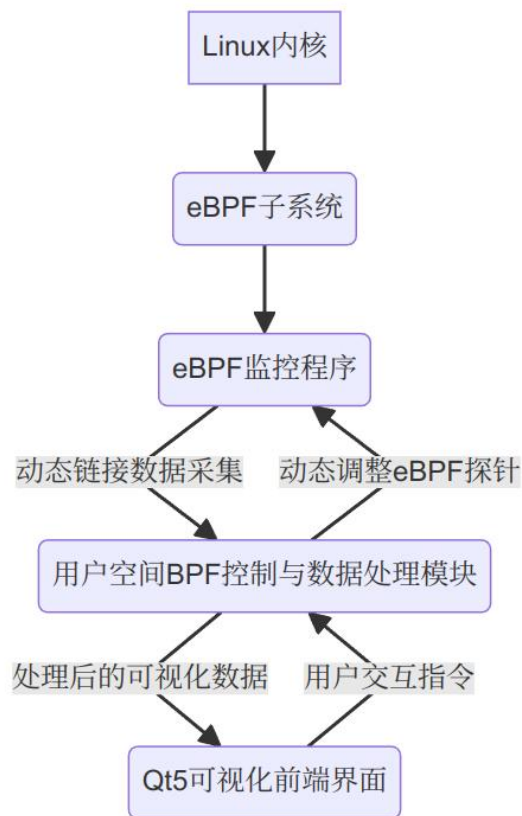


图 3.1 三层间的交互和数据流动示意图

3.2.2 内核态-用户态交互设计

系统中的内核态 eBPF 程序与用户态应用程序之间的交互主要通过 eBPF 映射 (Maps) 来实现。环形缓冲区 (Ring Buffer) 类型的映射被用于从内核高效地向用户空间传递动态链接的事件数据。

数据写入: 在内核态, eBPF 监控程序通过挂载到动态链接器相关的内核事件点, 捕获所需的动态链接信息, 并将这些信息封装成特定的数据结构。然后, eBPF 程序使用内核提供的辅助函数将这些数据写入到预先创建的环形缓冲区映射中。

环形缓冲区是一种先进先出的数据结构，非常适合于单向的数据流，能够保证数据的顺序性和高效性。

数据读取：在用户态，BPF 控制与数据处理模块使用 libbpf 库提供的 API 来打开和读取这个环形缓冲区映射。libbpf 库提供了 `perf_buffer__poll()` 等函数，可以用于轮询环形缓冲区，检查是否有新的数据到达。当有新的数据时，用户空间程序会将其从环形缓冲区中读取出来，并进行后续的解析和处理。

用户空间程序还通过其他类型的 eBPF 映射与内核态的 eBPF 程序进行控制和配置。使用哈希映射来传递过滤规则给 eBPF 程序，让 eBPF 程序只捕获特定进程的动态链接事件。用户空间程序使用 libbpf 提供的 API 来更新这些映射中的数据，从而动态地控制 eBPF 程序的行为。

3.3 系统架构组件设计

3.3.1 BPF 程序设计

eBPF 监控程序是本系统的核心，它负责在 Linux 内核态捕获动态链接过程中的关键信息，需要精确挂载到与动态链接相关的特定用户空间跟踪点上。

本系统选定的关键跟踪点主要集中在 glibc 库中与动态链接密切相关的函数。具体包括：`dlopen`（用户空间程序请求加载动态库的函数）、`dlclose`（用户空间程序请求卸载动态库的函数）和 `dlsym`（用户空间程序请求查找符号定义的函数）。通过在这些函数的入口（`uprobe`）和出口（`uretprobe`）设置探针，BPF 程序能够全面获取动态库的加载、卸载以及符号解析过程中的详细信息。

在这些预设的挂载点上，BPF 程序被设计用于提取以下关键信息：

- 加载的动态库名称：识别被加载的共享库的文件名或路径。
- 动态库被加载到的内存地址：获取动态库在进程虚拟内存空间中的起始加载地址，对于卸载事件，则是被卸载库的句柄。
- 被解析的符号名称：记录 `dlsym` 调用请求查找的具体符号。
- 符号在动态库中的地址：获取 `dlsym` 解析成功后返回的符号实际内存地址。
- 事件发生的时间戳：精确记录事件发生的纳秒级时间。
- 触发动态链接操作的进程 ID：识别执行动态链接操作的进程。

BPF 程序内部采用结构体 `struct event` 来封装捕获到的所有信息，确保数据传递的完整性和一致性。为了将这些在内核态捕获并填充好的数据结构高效、异步地传递到用户空间，本系统采用了 eBPF Perf Event Array 映射。BPF 程序通过 `bpf_perf_event_output()` 辅助函数将填充好的 `struct event` 数据写入到该 Perf Event Array 中，用户空间程序则从该缓冲区读取数据。此外，为了解决 `dlopen` 的入口和出口探针之间参数传递的问题，程序还使用了一个临时的 eBPF Hash Map 来存储文件名，确保在出口探针能够将句柄与对应的文件路径关联起来。为了实现灵活

的监控策略，BPF 程序还包含两个 eBPF Array Map，分别用于存储用户指定的目标进程名和监控程序自身进程名，以实现特定进程的监控或避免对自身的监控。

下表详细列出了 BPF 程序挂载的关键钩子点以及在每个点能够提取的信息：

表 4 BPF 程序的钩子点及提取信息

钩子点	提取的信息
/usr/lib/libc.so.6:dlopen	加载的库文件名、dlopen 调用时的标志（如 RTLD_LAZY）、库被加载到的内存起始地址（句柄）
/usr/lib/libc.so.6:dlclose	待卸载库的句柄、卸载事件的库路径
/usr/lib/libc.so.6:dlsym	需要查找的符号名称、解析到的符号地址、所属库句柄、所属库路径
任意以上钩子点	当前时间戳、触发操作的进程 ID、触发操作的进程名

3.3.2 用户空间程序设计

用户空间程序是本系统的交互前端，其核心职责在于与内核中的 eBPF 程序进行高效交互，接收、处理并最终展示捕获到的动态链接事件数据。该程序采用 libbpf 库进行开发，该库提供了与 eBPF 程序进行通信和管理所需的底层接口，极大地简化了开发复杂性。

用户空间程序在启动时会执行一系列初始化任务。首先，它利用 libbpf 提供的骨架 API 加载编译好的 eBPF 目标文件到内核中。加载成功后，程序会进一步将 eBPF 程序附加到之前确定的用户空间跟踪点上（即 dlopen、dlclose、dlsym 的入口和出口），从而使 BPF 程序能够在相应的动态链接事件发生时被内核执行。在此阶段，用户空间程序还会将其自身的进程名写入到 BPF 程序的 Map 中，以避免监控自身的动态链接行为，并且根据命令行参数设置需要监控的目标进程名。

数据接收是用户空间程序的关键环节。它会通过 libbpf API 管理 Perf Event Array，打开 BPF 程序中创建的 events Map，并设置一个回调函数 handle_event。当 BPF 程序通过 bpf_perf_event_output() 将新的事件数据写入 Perf Event Array 时，该回调函数会被异步触发，从而接收到内核态传递的原始数据。为了确保事件的实时性，程序会持续使用 perf_buffer__poll() 函数主动轮询 Perf Buffer，等待并处理新到达的数据。

接收到原始数据后，用户空间程序会进行以下详细的处理步骤：

数据解析：根据 BPF 程序写入数据时所遵循的 struct event 格式，程序会解析接收到的原始字节流，准确提取出动态库名称、内存地址、符号名称、事件类型、时间戳等关键信息。

数据结构化：解析后的信息将被存储到内部定义的数据结构中，例如 struct event 的用户空间对应版本。这种结构化的存储方式便于后续的数据管理和访问。程序可能会使用 std::vector 等容器来存储一系列连续发生的动态链接事件。

数据增强：为了提供更丰富和精确的信息，用户空间程序会执行额外的数据增强操作。例如，它会利用 `dlfcn.h` 和 `link.h` 提供的 `dlopen` 和 `dlinfo` 函数，将 BPF 程序在内核态可能捕获到的相对路径或链接名解析为动态库的真实文件系统路径。同时，它会将内核态的单调时间戳转换为用户友好的本地真实时间格式，并解析 `dlopen` 的标志位为可读的字符串。

数据展示：处理完成的动态链接事件数据将直接通过标准输出展示，同时可以进一步通过 Qt5 的信号与槽机制传递给图形用户界面前端，实现事件的可视化展示。例如，可以定义一个信号，并在有新的事件发生时发出该信号，由前端接收并更新显示。

用户空间程序的设计考虑了异步数据流的特性，并确保了所接收数据的完整性和正确性，从而为用户提供准确、实时的动态链接监控信息。

3.3.3 Qt5 可视化程序设计

本监控系统的用户界面采用 Qt5 框架进行开发，旨在提供直观、实时的动态链接事件展示。前端程序通过模块化设计，将数据管理、进程控制和用户界面展示分离，确保了系统的可维护性和可扩展性。核心设计思想是建立一个事件驱动的架构，通过信号与槽机制实现各组件间的松耦合通信。

前端程序主要由以下几个核心组件构成：

- **MainWindow(主窗口类)：**作为应用程序的顶层窗口，负责整体界面的布局、用户交互的响应以及协调各功能模块。它集成了开始/停止监控按钮、目标进程输入框、状态显示标签以及多标签页视图，为用户提供统一的操作入口。
- **EventData(事件数据管理类)：**该类专注于事件数据的存储和管理。它从后端接收原始的事件文本，并负责将其解析成结构化的 `Event` 对象。`EventData` 内部维护一个事件列表，并提供按事件类型（加载、卸载、符号解析）进行过滤和查询的接口。当有新事件被解析并添加到其内部存储时，它会发出信号通知其他对事件感兴趣的组件。
- **ProcessManager(进程管理器类)：**此组件是用户空间程序与后端 eBPF 监控程序之间的桥梁。它负责启动和停止后端程序（通常通过 `QProcess` 对象执行外部进程），捕获后端程序的标准输出和标准错误，并将其转发给 `EventData` 进行解析。`ProcessManager` 还处理后端进程的异常情况，如启动失败或崩溃，并通过信号通知 `MainWindow`。
- **TimelineView(时间线视图类)：**这是一个专门用于以时间顺序展示所有动态链接事件的独立视图组件。它以表格形式呈现事件的时间戳、类型、关联进程信息和详细内容，提供了一种全局概览。

- 表格视图组件：除了时间线视图，前端还设计了三个独立的 `QTableWidget` 实例，分别用于分类展示动态库加载事件、符号解析事件和动态库卸载事件。每个表格都有预定义的列，以适应不同事件类型的特定信息，例如加载事件会显示库路径和加载基址，符号解析事件会显示符号名和解析地址等。

这种分层的设计使得每个类都承担单一职责，提高了代码的清晰度和可测试性。数据流通常从 `ProcessManager` 接收原始输出，传递给 `EventData` 进行解析和存储，`EventData` 再通过信号通知 `MainWindow` 和 `TimelineView` 进行界面更新。用户通过 `MainWindow` 上的控件与 `ProcessManager` 交互，进而控制后端监控的启停。

通过各层组件之间的互相配合，前端可以及时收到后端程序传递的事件信息并实时展示给用户。

下表总结了 Qt5 可视化程序主要组件及其核心功能：

表 5 Qt5 可视化程序主要组件及其功能

组件名称	核心功能	主要交互对象
MainWindow	应用程序主窗口，布局管理，用户交互响应，协调各模块	EventData, ProcessManager, TimelineView, QTableWidget
EventData	解析、存储和管理所有动态链接事件数据，提供查询接口	ProcessManager(接收数据), MainWindow, TimelineView(发送事件)
ProcessManager	启动/停止后端监控进程，捕获其输出和错误，转发给 EventData	MainWindow(控制启动/停止), EventData(传递输出)
TimelineView	以时间线方式显示所有动态链接事件的表格视图	EventData(接收事件)
分类表格视图	以表格形式分类显示加载、符号解析、卸载事件	EventData(接收事件)

第 4 章 系统实现

4.1 BPF 程序实现

本系统的 eBPF 程序通过 C 语言编写, 并利用 clang/LLVM 工具链编译成 eBPF 字节码, 最终加载到 Linux 内核中执行。其实现细节涵盖了数据定义、探针函数的逻辑以及与用户空间的数据交互机制。

1. 数据结构定义:

在 BPF 程序中, `struct event` 结构体是核心数据载体, 它定义了从内核态捕获的每一条动态链接事件所包含的所有字段, 例如时间戳、进程标识符(PID)、用户标识符(UID)、进程命令名(comm)、动态库路径(lib_path)、库加载地址或句柄(lib_addr)、符号名称(symbol_name)、事件类型(event_type)、dlopen 调用标志(flags)和符号地址(symbol_addr)。这些字段的设计旨在全面覆盖动态链接事件的关键属性, 为用户空间提供足够的信息进行分析和展示。

2.eBPF Map 的具体实现:

BPF 程序中定义了多种类型的 eBPF Map, 它们在实现中扮演着不同的角色:

`handle_to_path`(哈希表): 使用 `__uint(type, BPF_MAP_TYPE_HASH)` 声明, 键为 `__u64` 类型的库句柄, 值为 `char[64]` 类型的库路径。该 Map 在 `dlopen` 返回时更新, `dlclose` 和 `dlsym` 调用时查询并使用, 有效维护了库句柄与路径的映射关系。

`events`(Perf Event Array): 通过 `__uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY)` 定义, 用于将 `struct event` 实例异步、高效地传递到用户空间。BPF 程序通过 `bpf_perf_event_output()` 辅助函数将事件数据原子地推送到此缓冲区。

`temp_path`(哈希表): 声明为 `__uint(type, BPF_MAP_TYPE_HASH)`, 键为 `__u32`, 值为 `char[64]`。该 Map 用于在 `dlopen` 的入口探针中临时存储库路径, 并在对应的返回探针中取出, 解决了跨探针传递上下文信息的挑战。其最大条目数设置为 1, 确保只存储当前 `dlopen` 调用的临时路径。

`target_process` 和 `monitor_process`(数组 Map): 两者都使用 `__uint(type, BPF_MAP_TYPE_ARRAY)` 声明, 键为 `__u32`, 值为 `char[16]`。这些 Map 的设计是为了实现灵活的进程过滤, 用户空间程序在加载时会将监控目标或自身进程名写入, BPF 程序则通过 `bpf_map_lookup_elem()` 查询并根据内容决定是否处理当前进程的事件。

3. Uprobe 探针的实现:

BPF 程序通过 `SEC("uprobe//usr/lib/libc.so.6:function_name")` 和 `SEC("uretprobe//usr/lib/libc.so.6:function_name")` 宏将 BPF 函数附加到特定的用户空间函数上。

trace_dlopen(Uprobe)：在 `dlopen` 函数入口处执行。它首先调用 `is_target_process()` 进行进程过滤。若符合条件，则获取当前时间戳、进程和用户 ID、进程名。关键一步是使用 `bpf_probe_read_user_str()` 安全地从用户空间读取 `filename` 参数，并将其拷贝到 `temp_path` Map。最后，填充 `event` 结构体，并通过 `bpf_perf_event_output()` 发送初步的加载事件。

trace_dlopen_ret(Uretprobe)：在 `dlopen` 函数返回时执行。同样进行进程过滤。它获取 `dlopen` 的返回值（即库句柄 `retval`），并将其赋值给 `e.lib_addr`。随后，从 `temp_path` 中查找并读取之前保存的库路径，并将其与 `retval` 一同存入 `handle_to_path` Map，完成句柄与路径的永久关联。最终，再次发送加载事件，补充 `lib_addr` 信息。

trace_dlclose(Uprobe)：在 `dlclose` 函数入口处执行。在过滤后，获取句柄并设置为 `e.lib_addr`。它会尝试从 `handle_to_path` Map 中查找对应的库路径，并将路径拷贝到事件结构体中。成功找到路径后，为了防止内存泄露和保持 Map 的实时性，会将该句柄从 `handle_to_path` Map 中删除。最后，发送卸载事件。

trace_dlsym(Uprobe)：在 `dlsym` 函数入口处执行。在过滤后，获取库句柄和符号名称，并通过 `bpf_probe_read_user_str()` 读取符号字符串。它会从 `handle_to_path` Map 中查找与句柄对应的库路径，并填充到事件结构体中，然后发送符号解析事件。

trace_dlsym_ret(Uretprobe)：在 `dlsym` 函数返回时执行。在过滤后，获取 `dlsym` 的返回值（即解析到的符号地址 `retval`），将其赋值给 `e.symbol_addr`，并发送带有符号地址信息的符号解析事件。

4. 进程过滤逻辑 `is_target_process()`:

这是一个静态内联函数，旨在优化性能。它首先使用 `bpf_get_current_comm()` 获取当前进程名。接着，通过查找 `monitor_process` Map，判断当前进程是否为监控程序自身，若是则直接返回 `false`。然后，检查 `target_process` Map 是否设置了目标进程名。如果没有设置，则表示监控所有进程（除了监控程序自身），返回 `true`。否则，将当前进程名与目标进程名进行精确比较，只有匹配时才返回 `true`。

4.2 用户空间程序实现

用户空间程序是连接 BPF 监控程序和可视化界面的桥梁，主要负责从 BPF 程序接收事件数据，进行处理、分析，并为可视化界面提供结构化的信息。

用户空间程序是基于 C++ 语言和 `libbpf` 库构建的，负责加载、附加 eBPF 程序，以及从内核接收、解析和展示动态链接事件。其实现围绕与 BPF 程序的交互、数据处理和用户界面输出展开。

1. 程序初始化与命令行解析:

`main` 函数是程序的入口点。它首先设置标准输出为无缓冲模式，以确保实时打印事件。然后，通过 `signal()` 函数设置 `SIGINT` 和 `SIGTERM` 信号处理函数 `sig_handler`，用于捕获中断信号以实现程序的优雅退出。程序支持命令行参数，通过 `argc` 和 `argv` 解析，允许用户指定需要监控的特定进程名，或者通过 `-h/--help` 显示使用说明。

2. BPF 程序的加载与附加：

用户空间程序使用 `libbpf` 的骨架 API 进行 BPF 程序的生命周期管理：

`dynlib_monitor_bpf__open_and_load()`：此函数用于打开预编译的 BPF 骨架文件并加载 BPF 字节码到内核中。如果加载失败，程序将报错并退出。

Map 初始化：在 BPF 程序加载成功后，用户空间程序通过 `bpf_map__fd()` 获取 `monitor_process` 和 `target_process` Map 的文件描述符。

通过 `bpf_map_update_elem()` 将当前用户空间监控程序的进程名（通过 `strchr(argv[0], '/')` 提取）写入 `monitor_process` Map 的键 0。

如果命令行指定了目标进程名，同样通过 `bpf_map_update_elem()` 将其写入 `target_process` Map 的键 0。这些操作确保了内核态 BPF 程序能够正确执行进程过滤逻辑。

`dynlib_monitor_bpf__attach(skel)`：此函数将 BPF 程序中定义的所有 Uprobe 和 Uretprobe 探针附加到 `/usr/lib/libc.so.6` 库中的相应函数上。如果附加失败，程序将报错并退出。

3. 事件接收与处理：

用户空间程序通过 `perf_buffer__new()` 函数创建并管理与 BPF 程序 events Map 对应的 Perf Buffer：

`perf_buffer__new(bpf_map__fd(skel->maps.events), 64, handle_event, handle_lost_events, NULL, NULL)`：此调用创建了一个 Perf Buffer 实例，指定了 BPF Map 的文件描述符、每个 CPU 的缓冲区大小（64 页）、事件处理回调函数 `handle_event` 和丢失事件处理回调函数 `handle_lost_events`。

`handle_event(void *ctx, int cpu, void *data, __u32 data_size)`：这是核心事件处理回调函数。当 Perf Buffer 接收到来自内核的事件数据时，此函数被触发。

它将原始数据 `data` 转换为 `const struct event*` 类型，从而可以访问事件结构体中的各个字段。

通过 `get_formatted_timestamp()` 函数将内核态传递的单调时间戳转换为易读的本地时间格式（年月日时分秒.微秒）。

使用 `switch(e->event_type)` 语句根据事件类型进行分支处理：

加载事件(`event_type = 1`)：

当 `e->lib_addr == 0` 时，表示这是 `dlopen` 入口处的事件，此时程序会调用 `get_lib_real_path()` 函数，尝试通过用户空间的 `dlopen` 和 `dlopen` API 获取动态库的真

实路径，以克服 BPF 程序在内核态获取路径的局限性。同时解析并打印 dlopen 的标志位。

当 `e->lib_addr != 0` 时，表示这是 dlopen 返回处的事件，此时打印加载成功的基地址。

卸载事件(`event_type = 2`): 打印被卸载库的句柄、路径和卸载结果。

符号解析事件 (`event_type = 3`):

当 `e->symbol_addr ==` 时，表示这是 dlsym 入口处的事件，打印查找的库句柄、请求的符号名和所属库路径。

当 `e->symbol_addr != 0` 时，表示这是 dlsym 返回处的事件，打印解析到的符号地址。

`handle_lost_events(void *ctx, int cpu, __u64 lost_cnt)`: 当内核态的 Perf Buffer 发生溢出导致事件丢失时，此函数被调用，用于向用户报告丢失的事件数量。

4. 辅助函数实现:

`get_monotonic_ns()`和 `get_realtime_ns()`: 使用 `clock_gettime()`函数获取单调时间（用于计算事件的相对时间）和实时时间（用于转换为可读的挂钟时间）。

`get_formatted_timestamp()`: 结合单调时间差和实时时间，将 BPF 程序传递的事件单调时间戳转换为标准的格式。

`get_dlopen_flags()`: 将 dlopen 函数的整数标志位（如 `RTLD_LAZY`, `RTLD_NOW`, `RTLD_GLOBAL`, `RTLD_LOCAL`）解析为可读的字符串组合，提高了日志的可解释性。

`get_lib_real_path()`: 利用用户空间的 dlopen 和 dlopen 函数，接收一个库名或路径作为输入，尝试加载该库并查询其 `link_map` 信息，以获取该库在文件系统中的真实、规范化路径。这对于处理符号链接、相对路径等情况非常重要。

5. 主循环与清理:

程序进入一个 `while (!exiting)` 循环，通过 `perf_buffer__poll(pb, 100)` 以 100 毫秒的超时时间持续轮询 Perf Buffer。如果 poll 返回错误且不是中断信号，则打印错误并退出循环。当 `exiting` 标志被信号处理函数设置为 `true` 时，循环结束。最后，程序通过 `perf_buffer__free(pb)` 和 `dynlib_monitor_bpf__destroy(skel)` 释放 Perf Buffer 资源并卸载 BPF 程序，确保资源被正确回收。

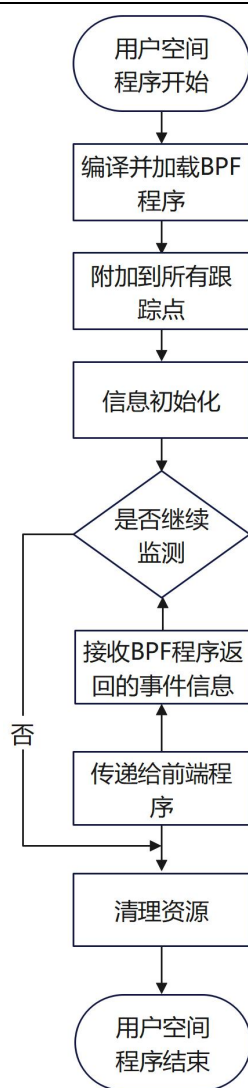


图 4.1 用户空间程序流程图

4.3 Qt5 可视化程序实现

Qt5 可视化程序的实现细节体现在各个类的具体方法中，通过信号与槽机制将独立的组件连接起来，实现了数据流和用户交互的响应。

1. MainWindow 的实现：

MainWindow 的构造函数负责初始化 EventData 和 ProcessManager 实例，并调用 setupUI() 和 setupEventTables() 方法构建界面。它建立了多对信号与槽连接：

processManager 的 monitorStarted、monitorStopped 和 errorOccurred 信号分别连接到 MainWindow 的 handleMonitorStarted、handleMonitorStopped 和 handleError 槽，用于更新界面状态或显示错误信息。

eventData 的 eventAdded 信号连接到 MainWindow 的 handleNewEvent 槽，这是接收新事件并更新界面的主要入口。setupUI() 方法负责创建主窗口的布局，包括 QLineEdit 用于输入目标进程名，QPushButton 用于开始/停止监控，以及 QLabel

显示当前状态。它还创建了 `QTabWidget` 来管理不同的视图，并初始化了 `timelineView`。

`setupEventTables()` 方法创建并配置了三个 `QTableWidget` 实例 (`loadTable`, `symbolTable`, `unloadTable`)，为它们设置了相应的表头，并配置了通用表格属性（如禁止编辑、单行选择、自动调整列宽、启用排序）。用户交互逻辑通过 `startMonitoring()` 和 `stopMonitoring()` 槽函数实现。

`startMonitoring()` 获取用户输入的进程名，并调用 `processManager->startMonitor()`。如果启动成功，则更新按钮和状态标签的可用性。`stopMonitoring()` 简单调用 `processManager->stopMonitor()`。

`handleNewEvent()` 槽是核心事件分发逻辑。它接收 `Event` 对象，首先将其添加到 `timelineView`。然后，根据 `event.eventType` 判断事件类型，选择对应的表格 (`loadTable`, `symbolTable`, `unloadTable`)。接着，根据事件的详细信息 (`event.details`) 提取出所需字段，创建 `QTableWidgetItem` 列表，并在目标表格的第一行插入新行并填充数据。为了保证时间顺序，表格会按时间戳降序排序。

2.EventData 的实现：

`EventData` 类的核心方法是 `addEvent()` 和 `parseEventText()`。

`parseEventText(const QString& eventText)`：此方法负责解析从后端接收到的原始文本事件。它使用 `QRegularExpression` 从文本的第一行中匹配时间戳和事件类型。对于剩余的行，它通过查找冒号来提取键值对，并将这些详细信息存储在 `Event` 结构体的 `details (QMap<QString, QString>)` 成员中。

`addEvent(const QString& eventText)`：此方法接收原始事件文本，调用 `parseEventText()` 进行解析。如果解析成功（即时间戳非空），则将解析后的 `Event` 对象添加到内部的 `events (QVector<Event>)` 容器中，并发出 `eventAdded(event)` 信号，通知所有连接的槽函数有新事件到来。

`getLoadEvents()`, `getSymbolEvents()`, `getUnloadEvents()`：这些方法遍历内部存储的所有事件，并根据 `eventType` 字段过滤出特定类型的事件列表，供其他组件查询使用。

3.ProcessManager 的实现：

`ProcessManager` 使用 `QProcess` 对象来管理后端监控程序的生命周期。

构造函数中，设置 `QProcess::MergedChannels` 以统一处理标准输出和标准错误，并设置环境变量 `PYTHONUNBUFFERED` 和 `GLIBC_UNBUFFERED` 来禁用输出缓冲，确保实时获取后端输出。`connect()` 语句将 `QProcess` 的 `readyReadStandardOutput`、`errorOccurred` 和 `finished` 信号分别连接到 `ProcessManager` 的 `handleProcessOutput`、`handleProcessError` 和 `handleProcessFinished` 槽。

startMonitor(const QString& targetProcess): 此方法构建执行 `pkexec` 命令及其参数的 `QStringList`，其中包含后端监控程序的完整路径和可选的目标进程名。然后调用 `process->start()` 启动外部进程。启动成功后发出 `monitorStarted()` 信号。

stopMonitor(): 此方法向后端进程发送 `SIGINT` 信号 (`process->terminate()`) 尝试终止进程。如果进程在超时时间内未结束，则强制杀死 (`process->kill()`)。

handleProcessOutput(): 此槽函数在后端有新输出时被触发。它读取所有可用的输出，将其追加到 `currentBuffer`，并立即调用 `processEventText()` 进行解析。

processEventText(): 这是一个关键的解析函数。它循环查找 `currentBuffer` 中的事件开始标记[和]事件：，将完整的事件文本提取出来。每当识别出一个完整的事件，就将其传递给 `eventData->addEvent()` 进行进一步处理。

handleProcessError()和 **handleProcessFinished():** 这两个槽函数分别处理后端进程启动失败、崩溃或正常结束的情况，并发出 `errorOccurred()`或 `monitorStopped()` 信号。

4. TimelineView 的实现：

`TimelineView` 类主要负责其表格(`eventTable`)的初始化和事件的添加。

setupTable(): 设置表格的列头（时间戳、事件类型、进程名、进程 ID、详细信息），并配置表格的通用属性，使其适合显示时间线数据。

addEvent(const Event& event): 此方法接收一个 `Event` 对象，在 `eventTable` 的第一行插入新行。它将事件的时间戳、类型、进程名、进程 ID 以及从 `event.details` 构建的详细信息字符串填充到对应的单元格中。特别地，详细信息单元格被设置为自动换行(`setWordWrap(true)`)，以适应多行文本。表格会根据内容的添加自动调整列宽。

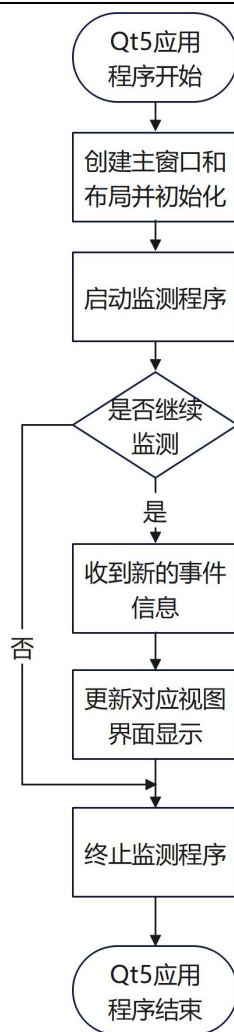


图 4.2 Qt5 可视化程序流程图

第 5 章 系统测试

5.1 测试环境

- 硬件环境

内存：16GB+16GB (swap)

硬件空间：256GB

处理器：AMD Ryzen 7 5800H 16 核

显卡：RTX3050

- 软件环境

操作系统：x86_64 GNU/Linux

内核版本：6.14.4-arch1-1

开发语言：C/C++

编译器：gcc/g++

5.2 功能测试

为了验证信息提取过程的准确性，使用自己编写的测试程序来进行测试，测试程序加载三个动态库 libm.so、libpthread.so 和 libcrypt.so，并使用其中的部分函数，但是只显式卸载了动态库 libm.so 和 libpthread.so，而没有卸载 libcrypt.so，这些过程信息应该被监控程序所监测到。

测试程序设计为典型的动态链接场景，主要通过显式调用动态链接器函数 (dlopen, dlsym, dlclose)^[8]来模拟常见的动态加载模式。测试程序运行时动态加载 libm.so、libpthread.so 和 libcrypt.so 库、调用其中函数并卸载 libm.so 和 libpthread.so 库，以此来产生动态链接事件。选择这三个动态库作为测试对象是因为他们包含多种常用函数，调用频率高，且在实际应用中具有广泛的代表性。

测试程序在运行时首先通过 dlopen 函数加载 libm.so、libpthread.so 和 libcrypt.so 库，采用 RTLD_LAZY 标志表示延迟绑定模式。随后通过 dlsym 依次获取 sin、cos、sqrt、log、exp、pthread_mutex_init、pthread_mutex_destroy 和 crypt 函数的符号地址，并在每个函数调用之间随机睡眠 100~1000ms，模拟真实延时。

测试执行时，首先启动开发的监控系统，使 BPF 程序附加到相关的跟踪点上，然后运行测试程序，记录并分析监控系统捕获的事件数据。测试过程中，监控系统正确识别并记录了测试程序的进程信息、动态库加载事件、符号解析事件以及相关的内存映射信息。

最后获取到的过程信息如下图：



图 5.1 测试时的时间线视图

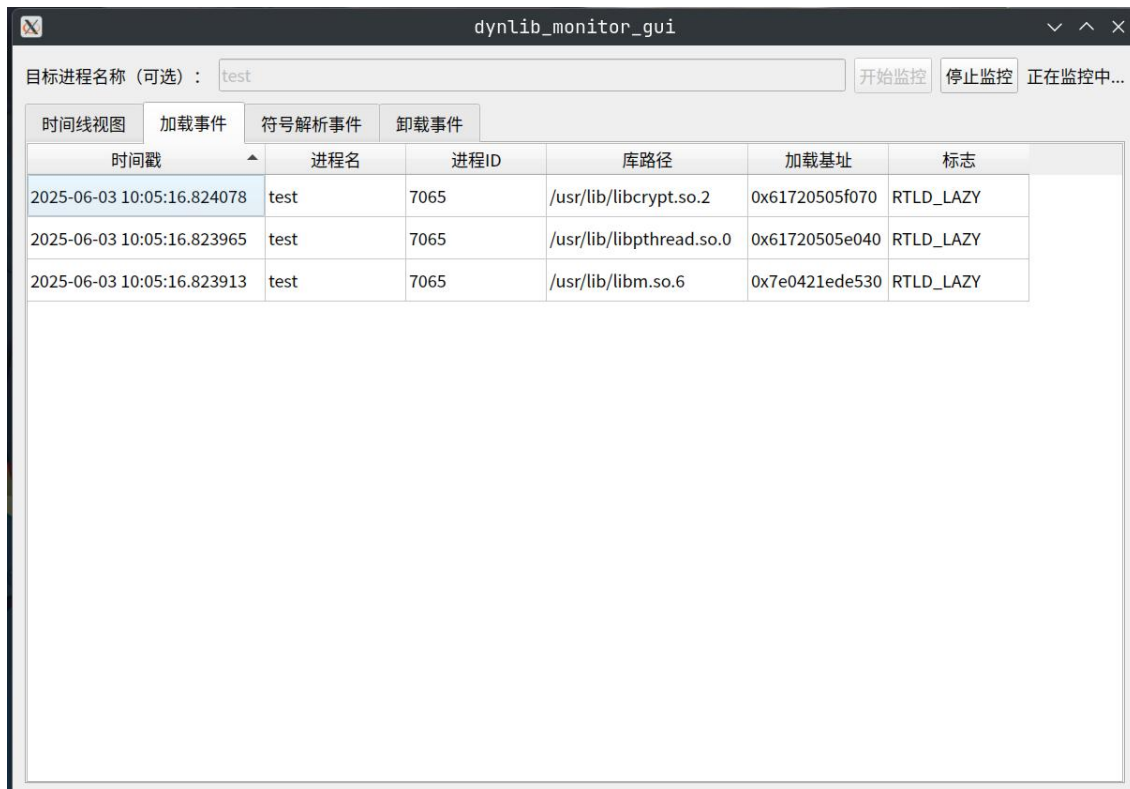
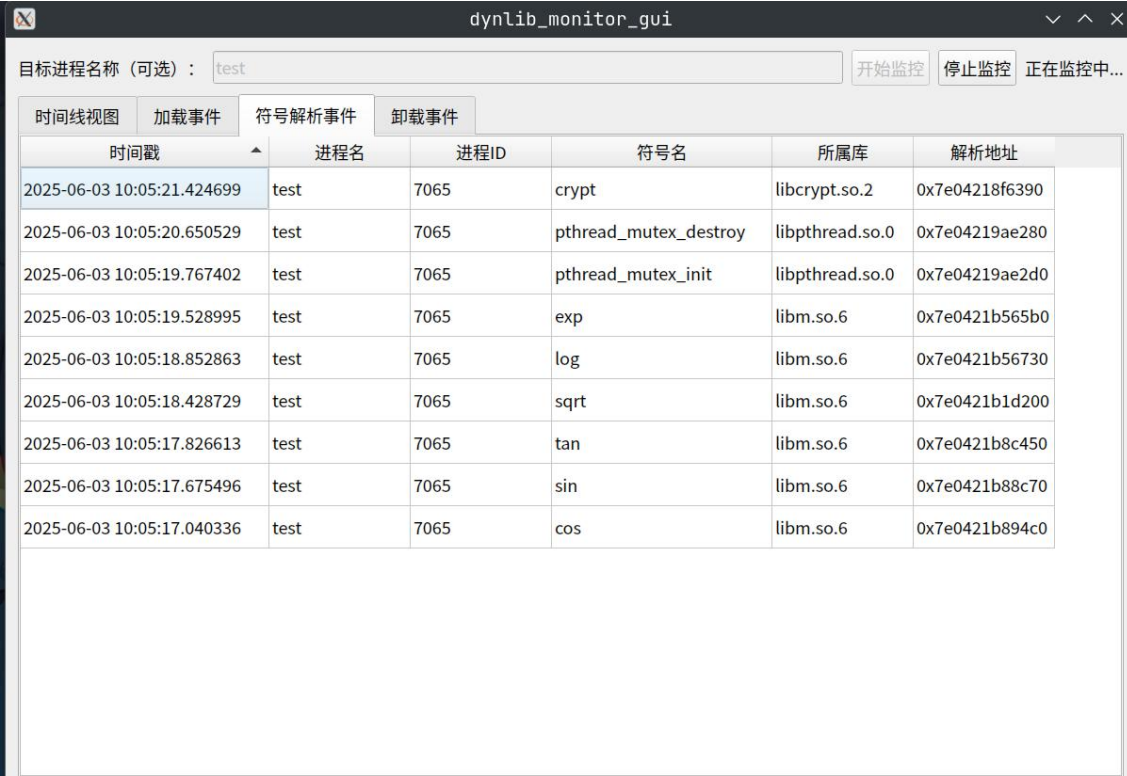


图 5.2 测试时的库加载事件视图

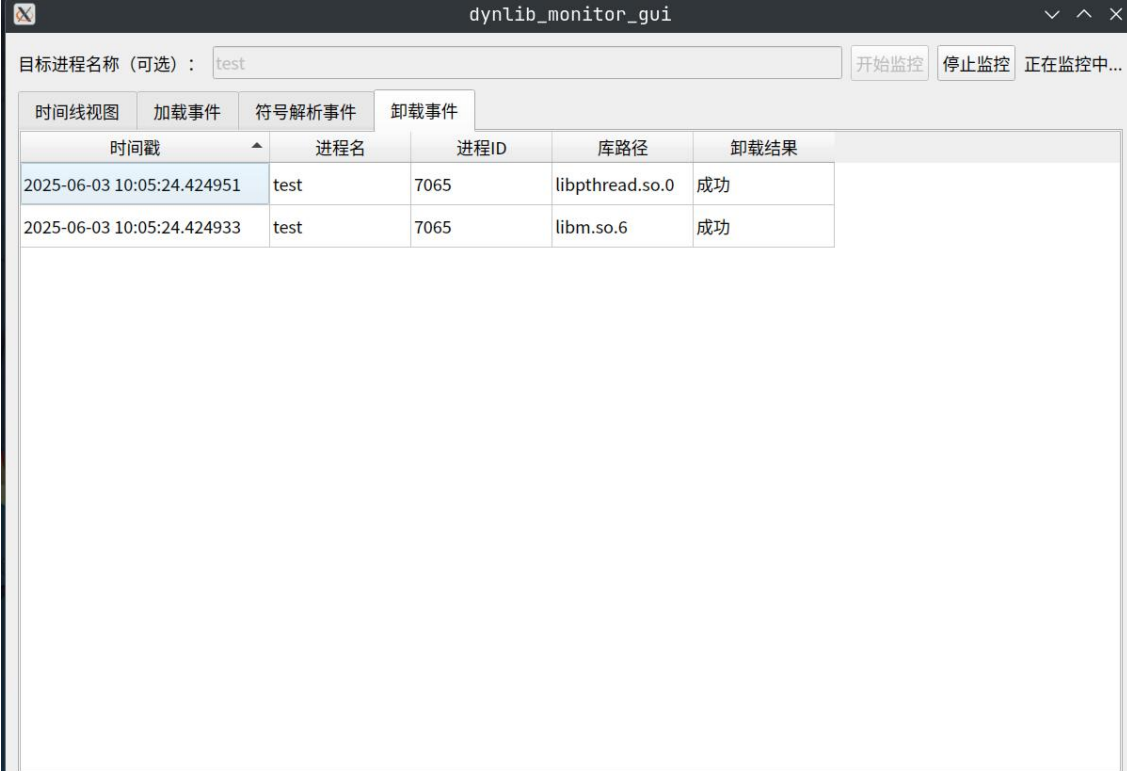


目标进程名称 (可选): 开始监控 停止监控 正在监控中...

时间线视图 | 加载事件 | **符号解析事件** | 卸载事件

时间戳	进程名	进程ID	符号名	所属库	解析地址
2025-06-03 10:05:21.424699	test	7065	crypt	libcrypt.so.2	0x7e04218f6390
2025-06-03 10:05:20.650529	test	7065	pthread_mutex_destroy	libpthread.so.0	0x7e04219ae280
2025-06-03 10:05:19.767402	test	7065	pthread_mutex_init	libpthread.so.0	0x7e04219ae2d0
2025-06-03 10:05:19.528995	test	7065	exp	libm.so.6	0x7e0421b565b0
2025-06-03 10:05:18.852863	test	7065	log	libm.so.6	0x7e0421b56730
2025-06-03 10:05:18.428729	test	7065	sqrt	libm.so.6	0x7e0421b1d200
2025-06-03 10:05:17.826613	test	7065	tan	libm.so.6	0x7e0421b8c450
2025-06-03 10:05:17.675496	test	7065	sin	libm.so.6	0x7e0421b88c70
2025-06-03 10:05:17.040336	test	7065	cos	libm.so.6	0x7e0421b894c0

图 5.3 测试时的符号解析事件视图



目标进程名称 (可选): 开始监控 停止监控 正在监控中...

时间线视图 | 加载事件 | 符号解析事件 | **卸载事件**

时间戳	进程名	进程ID	库路径	卸载结果
2025-06-03 10:05:24.424951	test	7065	libpthread.so.0	成功
2025-06-03 10:05:24.424933	test	7065	libm.so.6	成功

图 5.4 测试时的库卸载事件视图

可以看到监控程序正确提取到了测试程序执行过程中产生的动态链接信息并将其按照事件发生的事件显示在可视化界面中。包括时间戳、事件类型、进程名、进程 ID、加载库路径、加载基址、加载标志、被解析符号名、被解析符号所属库、被解析符号解析地址、卸载库路径和卸载结果。同时，通过对加载库事件和卸载库事件的观察，可以据此发现 `libcrypt.so` 未能正确显式卸载的问题。

测试结果表明程序可以通过 `ebpf` 技术监测 `dlopen`、`dlsym`、`dlclose` 等关键函数的调用来正确地获取到动态链接过程中的信息、这些信息对于深入了解程序的动态链接过程十分重要。开发者可以据此无侵入性地对程序的动态链接过程进行分析，并发现其中问题。

结 论

本课题成功设计并实现了一个基于 eBPF 技术和 Qt5 前端的 Linux 动态链接过程可视化分析系统。系统通过在内核态利用 eBPF 程序对动态链接器的关键行为进行精确观测与数据提取，结合用户态程序对原始数据的解析处理，最终通过 Qt5 图形用户界面，实现了对动态库加载、符号解析与重定位等核心环节的实时、直观展示。

研究过程中，首先深入分析了 Linux 动态链接器的工作机制及 eBPF 技术的原理与应用，确定了关键的观测点和需要提取的核心数据。随后，利用 C 语言和 libbpf 库开发了 eBPF 监控程序，实现了对 `dlopen`、`dlsym`、`dlclose` 等关键函数的跟踪，并通过缓冲区高效地将捕获的动态链接事件数据传递至用户空间。用户空间程序则负责数据的接收、解析、结构化处理，并利用 Qt 的信号与槽机制将数据动态传递给前端界面。前端界面基于 Qt5 设计，通过列表视图形式，清晰地呈现了动态链接的详细过程，包括动态库的加载和卸载、符号解析信息等。

通过对自行设计的测试程序的监控与分析，验证了本系统在捕获动态链接事件、提取关键信息以及可视化展示方面的准确性和有效性。本系统不仅加深了对 Linux 动态链接器运行机制的理解，也展示了 eBPF 技术在内核行为观测与分析领域的强大潜力。所开发的工具为开发者、系统管理员及安全研究人员提供了一个新颖的视角和有效的辅助手段，有助于进行程序调试、性能优化以及对动态链接过程的深入分析。

尽管本系统已基本实现预期功能，但在信息展示的丰富性、用户交互的便捷性以及应对更复杂动态链接场景的适应性等方面仍有提升空间。未来的工作可以考虑扩展对更多动态链接事件的监控，增加更高级的数据分析与统计功能，并进一步优化可视化界面的交互体验。

参考文献

- [1] Gregg B. BPF Performance Tools[M]. Boston: Addison-Wesley Professional, 2019.
- [2] 何明, 王书宏, 张伟. 基于eBPF的系统性能分析与优化技术[J]. 计算机系统应用, 2020, 29(6): 1-7.
- [3] Levine J R. Linkers and Loaders[M]. San Francisco: Morgan Kaufmann, 2000.
- [4] Drepper U. How To Write Shared Libraries[R/OL]. 2011 [2025-05-23]. <https://akkadia.org/drepper/dsohowto.pdf>.
- [5] The Qt Company. Qt 5 Documentation[EB/OL]. [2025-05-23]. <https://doc.qt.io/qt-5/>.
- [6] Xiong Y, Gan Y. eBPF: The Future of Linux Observability and Security[C]//2020 IEEE International Conference on Networking, Architecture, and Storage (NAS). Piscataway: IEEE, 2020: 1-10.
- [7] Linux Man Pages. ld.so(8) - Linux manual page[EB/OL]. [2025-05-23]. <https://man7.org/linux/man-pages/man8/ld.so.8.html>.
- [8] Linux Man Pages. dlopen(3) - Linux manual page[EB/OL]. [2025-05-23]. <https://man7.org/linux/man-pages/man3/dlopen.3.html>.
- [9] Ezust A, Ezust P. An Introduction to Design Patterns in C++ with Qt 5[M]. New Jersey: Prentice Hall, 2012.
- [10] BPF Core (BCC) Project. BCC - BPF Compiler Collection[EB/OL]. [2025-05-23]. <https://github.com/iovisor/bcc>.
- [11] Libbpf Documentation. libbpf - BPF library[EB/OL]. [2025-05-23]. <https://libbpf.readthedocs.io/>.
- [12] 李云. Linux动态链接机制研究与应用[D]. 西安: 西安电子科技大学, 2010.
- [13] 王帅. 基于eBPF的Linux内核网络数据包监控系统设计与实现[D]. 成都: 电子科技大学, 2021.
- [14] 张晓东, 孟坤. 基于Qt的通用数据可视化软件设计与实现[J]. 计算机与现代化, 2018, (9): 99-103.
- [15] Bovet D P, Cesati M. Understanding the Linux Kernel[M]. 3版. Sebastopol: O'Reilly Media, 2005.
- [16] Qt Project. Qt Charts Overview[EB/OL]. [2025-05-23]. <https://doc.qt.io/qt-5/qtcharts-overview.html>.

致 谢

我首先要感谢我的指导老师王亚刚，他为我的论文研究提供了指导性的意见和推荐，在论文撰写过程中给予了我悉心的知道，并不断提出有益的改善型建议。不仅如此，他同样是我加入的西邮 Linux 兴趣小组的指导老师，在我的整个大学实验室生活期间，指导了我很多学业和技术上的指导。此外，我也非常感谢小组的同学和朋友们，他们一直以来都是我最重要的支持者和帮助者，无私地为我提供了许多帮助和起始。在小组中，我不仅获得了专业的 Linux 技能和知识培新，还结交了很多志同道合的朋友，让我的大学生活风度多彩。同时，小组中熟悉 eBPF 和 Linux 内核的同学也在本论文的撰写过程中给予我很多帮助，帮助我系统的学习了 eBPF 技术。在他们的帮助下，我相信我将走向更充实、更有意义的人生旅程，我希望我能继续保持对计算机技术的热情和好奇心，不断探索和学习，提高自己的技术能力。

最后，再次想王亚刚老师和所有帮助过我的朋友们表示衷心感谢，他们的支持永远铭刻在我的心中。感谢西安邮电大学四年的美好经历，感谢所有人。