# 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bpftrace，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

本节主要介绍eBPF技术和libbpf库。

### 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、系统调用拦截、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 |

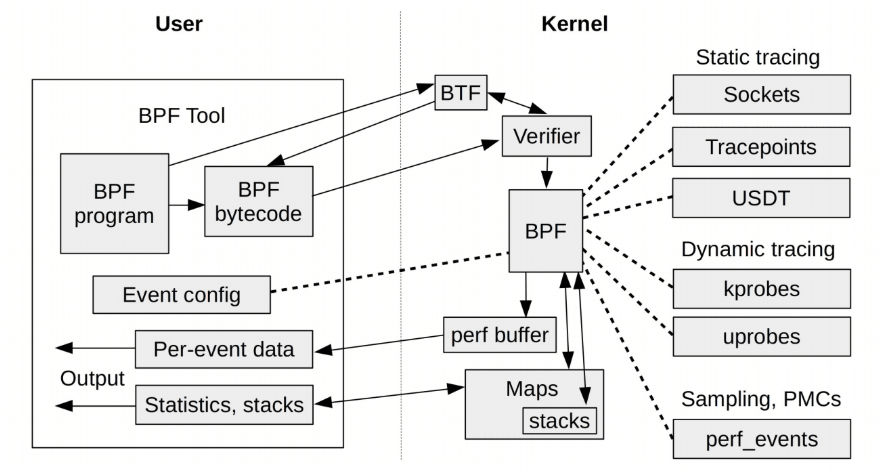
### 2.1.2 工作原理

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

本节主要介绍Linux上的动态链接器、动态链接库以及对动态链接器中的符号解析与重定位操作。

### 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]：

（1）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。

## 2.2 Qt5

本节主要介绍程序开发框架Qt5的可视化方面功能。

### 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 系统分析与设计

## 3.1 系统分析

目前已经存在一些用于分析程序动态链接过程的工具。例如，Linux系统自带的ldd命令可以列出一个可执行文件或共享库所依赖的动态库列表[15]。而strace工具则可以跟踪程序的系统调用，包括与动态链接相关的open，mmap等系统调用。然而，这些工具通常只能提供较为粗粒度的信息，而无法提供动态链接过程内部的详细信息，例如符号解析和重定位的具体过程，以及这些过程发生的实时状态。

在学术界和开源社区中，目前还没有利用动态分析技术来研究动态链接过程的项目。本课题的提出正是为了弥补现有的缺口。通过利用eBPF技术，我们可以在内核层面对动态链接过程进行细致的观测，以极低的性能开销捕获到关键的动态链接事件和数据。结合Qt5强大的可视化能力，我们可以将这些复杂的数据以清晰、直观的方式实时地展示给用户，从而提供一个更加深入和易于理解的动态链接分析和可视化系统。这样的系统能够帮助开发者更好地理解程序的启动过程、库的加载顺序、符号的解析过程以及潜在的性能瓶颈。

## 3.2 系统设计

本节主要介绍系统的设计原则、开发环境和对提取信息的处理设计。

### 3.2.1 设计原则

在设计本系统时，遵循以下几个关键的设计原则：

易用性：可视化界面应该直观易懂，操作简单，即使对于不熟悉动态链接底层细节的用户也能快速上手并理解所展示的信息。

清晰性：系统展示的动态链接信息应该准确、清晰、有条理，避免信息过载，使用户能够快速找到并理解关键数据。

实时性：系统应该能够实时地捕获和展示动态链接过程中发生的事件和相关数据，反映过程的动态变化。

可靠性：系统收集和展示的数据必须准确可靠，能够真实反映动态链接器的运行状态。

可扩展性：系统架构应该具有良好的可扩展性，方便未来添加新的功能或支持分析更多类型的动态链接信息。

### 3.2.2 开发环境

* 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

开发工具：VScode

调试工具：GDB

开发语言：C/C++

编译器：gcc/g++

构建工具：qmake

### 3.2.3 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。整个信息处理流程如下：

（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。

通过以上信息处理流程，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全过程。

动态链接信息结构如下表所示：

表2 动态链接信息数据结构

|  |  |  |
| --- | --- | --- |
| 字段名称 | 类型 | 描述 |
| pid | int | 进程ID |
| comm | char[] | 进程名称 |
| lib\_path | char[] | 库路径 |
| symbol\_name | char[] | 符号名称 |
| load\_addr | uint64\_t | 加载地址 |
| symbol\_addr | uint64\_t | 符号地址 |
| timestamp | uint64\_t | 时间戳 |
| event\_type | int | 事件类型 |
| flags | uint64\_t | dlopen标志 |
| parent\_lib | char[] | 父库 |

# 第4章 系统架构设计

## 4.1 系统架构整体设计

本节主要介绍系统的分层设计以及内核态和用户态之间的交互设计。

### 4.1.1 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：内核层、用户空间层和Qt5前端层。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，捕获关键事件和数据。
* 用户空间层：位于内核层之上，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。
* 前端层：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，并通过各种UI组件（如图表、列表、时间线等）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2 内核层、用户空间层和Qt5前端层及其职责

|  |  |  |
| --- | --- | --- |
| 层次名称 | 主要组件 | 主要职责 |
| 内核层 | 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 |
| 用户空间层 | 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 |
| Qt5前端层 | 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：

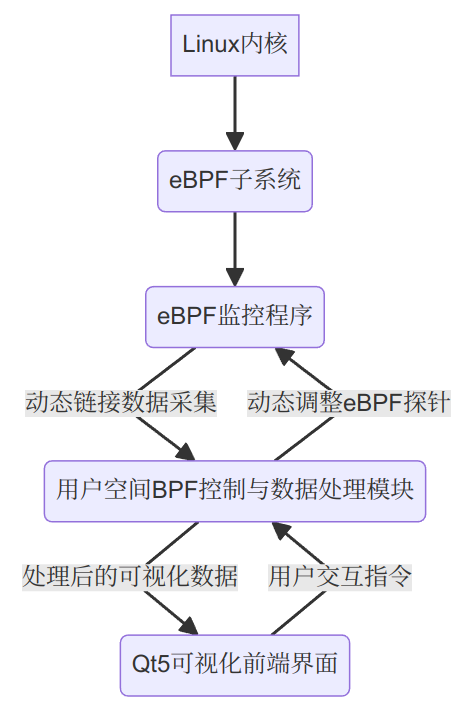


图4.1 内核层、用户空间层和Qt5前端层间的交互和数据流动示意图

### 4.1.2 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ring\_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。

## 4.2 系统架构组件设计

本节主要介绍BPF程序、用户空间程序和前端程序的具体设计。

### 4.2.1 BPF 程序设计

本系统的核心是eBPF监控程序的设计，它负责在内核态捕获动态链接过程中的关键信息。该程序需要挂载到与动态链接相关的特定内核跟踪点上。关键的跟踪点包括：

（1）与加载共享库相关的函数：\_dl\_map\_object (glibc中加载共享对象的关键函数)，dlopen (用户空间程序请求加载动态库的函数)[8]。通过跟踪这些函数的入口和出口，可以获取加载的库名称、加载的内存地址等信息。

（2）与符号解析和重定位相关的函数：\_dl\_relocate\_object (执行共享对象的重定位操作)，\_dl\_lookup\_symbol (查找符号的定义)。跟踪这些函数可以获取解析的符号名称、原始地址和重定位后的地址等信息.

在这些挂载点，eBPF程序需要提取以下关键信息：

（1）加载的动态库名称。

（2）动态库被加载到的内存地址。

（3）被解析的符号名称。

（4）符号在动态库中的地址。

（5）事件发生的时间戳，用于记录事件发生的先后顺序。

（6）触发动态链接操作的进程ID。

eBPF程序内部使用结构体来临时存储捕获到的信息。为了将这些信息传递到用户空间，需要使用eBPF映射，特别是环形缓冲区。程序会使用bpf\_ringbuf\_output()等辅助函数将填充好的数据结构写入到环形缓冲区中。

下表列出了关键的跟踪点以及在每个点可能提取的信息：

表4 BPF程序的钩子点及提取信息

|  |  |
| --- | --- |
| 钩子点 | 提取的信息 |
| \_dl\_map\_object | 加载的库文件名，加载标志等 |
| \_dl\_map\_object | 库被加载到的内存起始地址 |
| dlopen | 加载库事件信息 |
| dlsym | 符号名称，解析地址 |
| dlclose | 卸载库事件信息 |
| \_dl\_relocate\_object | 正在重定位的共享库的内存地址 |
| \_dl\_lookup\_symbol\_x | 需要查找的符号名称，查找范围等 |
| 任意以上钩子点 | 当前时间戳，触发操作的进程ID |

### 4.2.2 用户空间程序设计

用户空间程序的主要职责是与内核中的eBPF程序进行交互，接收并处理捕获到的动态链接数据，最终将处理后的数据传递给Qt5前端进行可视化。

用户空间程序会使用libbpf库来完成以下任务：

* 加载eBPF程序：使用骨架API加载编译好的eBPF目标文件到内核中。
* 附加eBPF程序：将eBPF程序附加到之前确定的内核跟踪点上，使得当这些事件发生时，eBPF程序能够被执行。这可以通过骨架API提供的相应函数完成。
* 管理环形缓冲区：打开之前在eBPF程序中创建的环形缓冲区映射，并设置回调函数，当有新的数据到达时，回调函数会被触发。可以使用ring\_buffer\_\_poll()函数来主动轮询数据。
* 读取数据：从环形缓冲区中读取eBPF程序写入的原始数据。

接收到原始数据后，用户空间程序需要进行以下处理步骤：

* 数据解析：根据eBPF程序写入数据的格式，解析原始的字节流，提取出库名称、内存地址、符号名称等关键信息。
* 数据结构化：将解析后的信息存储到合适的数据结构中，例如可以创建一个表示动态链接事件的类或结构体，包含库名称、地址、符号、时间戳等属性。可以使用链表或向量来存储一系列的动态链接事件。
* 数据过滤：根据需要，对事件进行过滤，只显示特定进程的链接信息。
* 数据传递：使用Qt5的信号与槽机制，将处理后的动态链接事件数据发送给Qt5前端。可以定义一个信号dynamicLinkEventOccurred，并在有新的事件发生时发出该信号。

用户空间程序需要处理异步到达的内核数据，并确保数据的完整性和正确性。

### 4.2.3 Qt5前端界面设计

Qt5前端界面的主要目标是以清晰、实时的方式可视化展示动态链接的过程。界面设计包含以下关键元素：

主窗口：作为应用程序的主容器，包含所有的可视化组件和控制元素。

加载的动态库列表：使用QListView来展示当前进程已经加载的动态库列表。每一项可以显示库的名称和加载到的内存地址。当有新的库被加载时，列表应能实时更新。

动态链接事件时间线：使用自定义的QGraphicsView来展示动态链接事件的发生顺序。每一项可以显示事件的类型、相关的信息以及发生的时间戳。时间线按照时间顺序排列，方便用户追踪链接过程。

进程信息显示：显示当前正在监控的进程的ID和名称。

控制面板：提供用户交互的控件，开始/停止监控按钮，过滤选项（只显示特定进程的链接信息）。

界面能够实时地响应用户空间程序发出的信号，当接收到新的动态链接数据时，立即更新相应的UI元素。

# 第5章 系统功能实现

## 5.1 eBPF监控程序实现

针对动态链接器的监控需要设置多个跟踪点和相应的处理函数。首先确定了以下关键跟踪点：动态库加载事件(dlopen函数调用)、动态库卸载事件(dlclose函数调用)、符号解析事件(dlsym函数调用)以及相应的重定位操作。通过这些跟踪点，可以全面观察动态链接过程的各个阶段。

BPF程序通过在这些关键位置附加kprobes和uprobes来实现无侵入式监控。对于dlopen、dlclose和dlsym函数，使用uprobes实现用户空间函数拦截；对于内核中相关的内存映射函数，使用kprobes进行跟踪。这种双层跟踪方式确保了动态链接过程的全覆盖监控。

为了在不同类型的探针之间共享数据，使用BPF映射机制创建了多个映射表格，主要包括：

表4 BPF程序主要映射表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 映射表名称 | 映射类型 | 键类型 | 值类型 | 用途 |
| event\_ringbuf | BPF\_MAP\_TYPE\_RINGBUF | - | - | 事件数据传输至用户空间 |
| pid\_info | BPF\_MAP\_TYPE\_HASH | pid\_t | proc\_info | 进程信息存储 |
| lib\_info | BPF\_MAP\_TYPE\_HASH | addr\_t | lib\_metadata | 库加载信息存储 |
| sym\_lookup | BPF\_MAP\_TYPE\_HASH | addr\_t | sym\_info | 符号查找过程跟踪 |

每当触发相关跟踪点时，BPF程序会捕获状态信息，将其格式化为特定的事件结构体，通过环形缓冲区传输至用户空间。对于动态库加载事件，记录进程信息、库名称、加载地址和标志；对于符号解析事件，记录符号名称、解析地址和相关上下文。针对时序性要求，每个事件都附加精确的时间戳，确保后续分析的准确性。

具体实现中，使用了libbpf提供的高级API进行BPF程序的开发，相比传统的BPF开发方式，这种方法具有更高的抽象级别和更简便的接口，使得开发过程更加高效。在处理动态链接器符号解析过程时，通过设计的BPF程序追踪了符号解析的完整流程，包括查找、匹配和最终绑定等关键步骤。

eBPF监控程序的实现主要涉及确定需要挂载的内核跟踪点，编写C代码以捕获相关信息，并将其存储到eBPF映射中。可以使用 SEC("uprobe/<函数名>") 宏将一个eBPF函数挂载到内核函数的入口。可以使用SEC("uretprobe/<函数名>") 宏将一个eBPF函数挂载到内核函数的出口。进而获取到内核在该函数出入口时获得的信息。

图5.1.1 BPF程序的部分钩子函数追踪点

当在挂载的内核函数中捕获到关键信息后，会创建一个自定义的数据结构来存储这些信息，包含库名（char）、加载地址（unsigned long）、符号名（char）等字段。然后，将该数据结构写入到环形缓冲区中。

## 5.2 用户空间程序数据采集与处理实现

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。

用户空间程序首先初始化与BPF程序的通信机制，加载并验证BPF程序，然后附加到相应的跟踪点。通过libbpf提供的API，程序加载BPF对象、设置映射表，并建立事件回调机制。程序启动后进入主循环，持续监听来自BPF程序的事件数据。

表5 用户空间程序主要组件结构

|  |  |
| --- | --- |
| 组件名称 | 功能描述 |
| 事件接收器 | 从环形缓冲区接收BPF事件 |
| 数据解析器 | 解析原始事件数据为结构化信息 |
| 数据存储器 | 维护进程、库和符号信息的状态 |
| 事件关联器 | 关联相关事件构建完整调用链 |
| 数据导出器 | 提供查询接口和数据导出功能 |

用户空间程序的核心是使用libbpf库与内核中的eBPF程序进行交互。首先，会使用bpf\_prog\_load()函数加载编译好的eBPF目标文件。然后，通过遍历eBPF对象中的程序列表，找到需要附加的程序，并使用bpf\_attach()函数将其附加到相应的内核跟踪点。

接下来，需要打开并管理eBPF环形缓冲区映射。可以使用ring\_buffer\_\_new()函数创建一个环形缓冲区对象，并指定一个回调函数，该回调函数将在有新的数据到达时被调用。

在回调函数handle\_event中，接收到的data是eBPF程序写入的原始数据。需要根据eBPF程序中定义的数据结构来解析这些数据，提取出库名、地址、符号等信息。解析后的数据存储在用户空间程序内部的std::vector容器中。

最后，在用户空间程序中定义一个信号，使用Qt的信号与槽机制将解析后的数据发送给Qt5前端。

在handle\_event函数中，解析出数据后，创建一个包含相关信息的信号，并使用emit关键字发送该信号，将数据传递给Qt5前端。

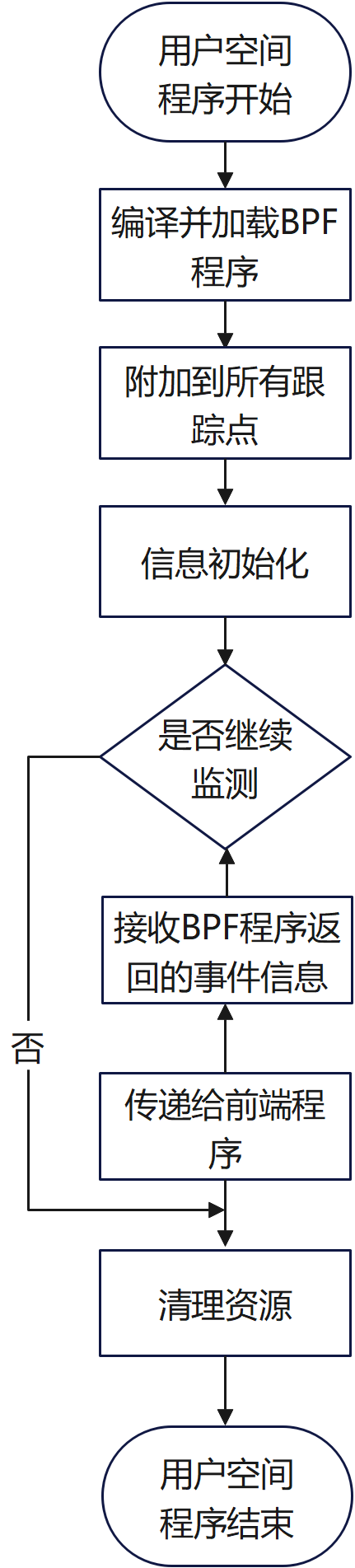


图5.2 用户空间程序流程图

## 5.3 Qt5可视化界面实现

Qt5可视化界面的实现主要使用QMainWindow作为主窗口，并使用不同的QWidget子类来展示动态链接信息。

表5 Qt5前端程序主要功能模块

|  |  |
| --- | --- |
| 模块名称 | 功能描述 |
| 库加载视图 | 展示动态库加载状态和详情 |
| 符号解析视图 | 显示符号解析过程及结果 |
| 时间线视图 | 以时间轴方式展示事件序列 |
| 内存映射视图 | 可视化展示内存布局和映射关系 |
| 实时监控面板 | 展示系统实时状态和关键指标 |

使用QTableView来显示加载的动态库列表。创建一个继承自 QAbstractTableModel的自定义模型，用于存储库名和地址信息[9]。当用户空间程序发出dynamicLinkEventOccurred信号时，相应的槽函数会被调用，解析信号中包含的数据，并更新模型中的数据，从而更新QTableView的显示。

对于动态链接事件的时间线，使用自定义的QGraphicsView来实现。每当接收到 dynamicLinkEventOccurred信号时，在列表中添加一个新的项，显示事件的类型、相关信息和时间戳。为了实现实时更新，需要确保用户空间程序发出的信号连接到了Qt5前端的相应槽函数。这在主窗口的构造函数中完成。

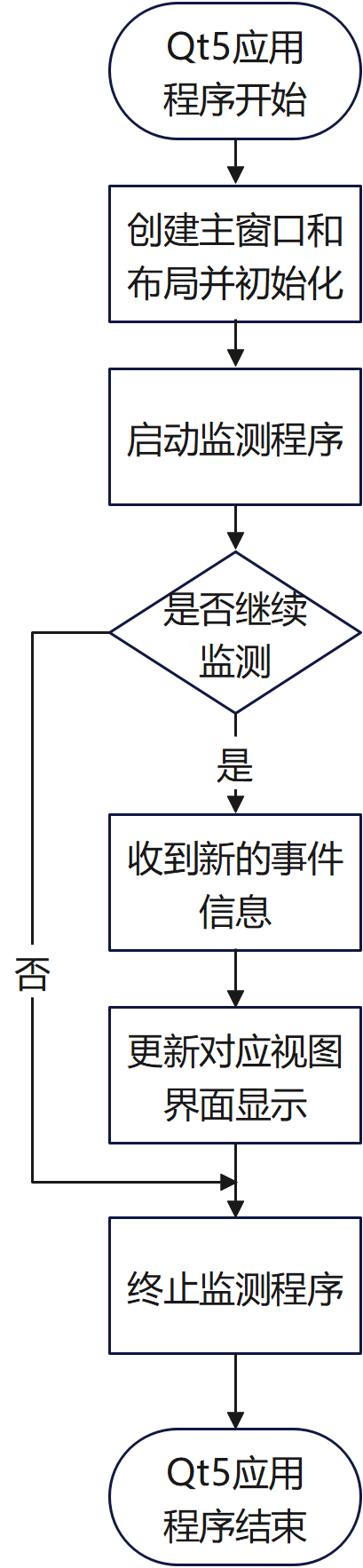


图5.3 Qt5前端程序流程图

# 第6章 系统测试

## 6.1 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++

## 6.2 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，测试程序加载数学库libm.so，并使用其中的的部分函数，这个过程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。测试程序以数学库（libm.so）为主要目标，通过运行时动态加载数学函数并执行计算来产生动态链接事件。选择数学库作为测试对象是因为其包含多种常用函数，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和exp等数学函数的符号地址。

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。

最后捕获到的过程信息如下图：

图6.2.1 测试时的事件总览视图

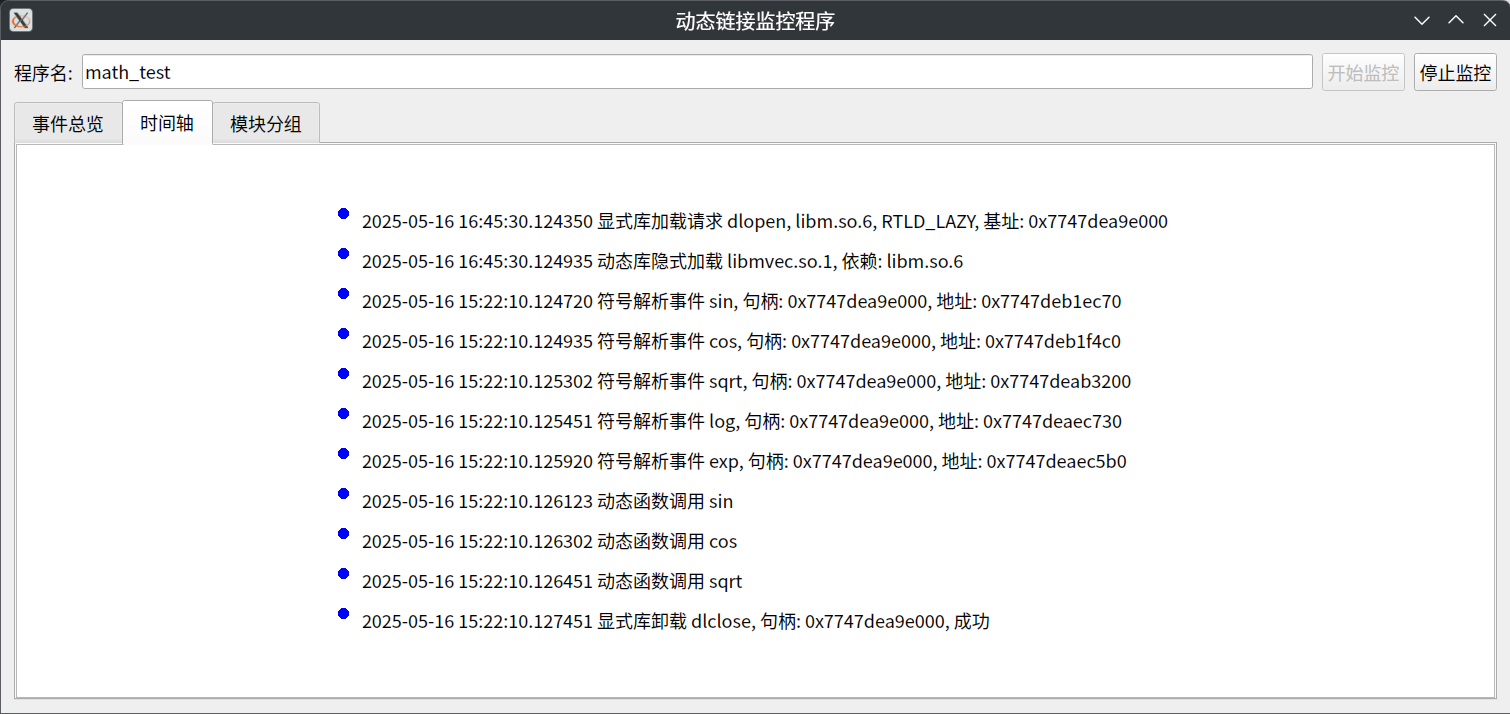
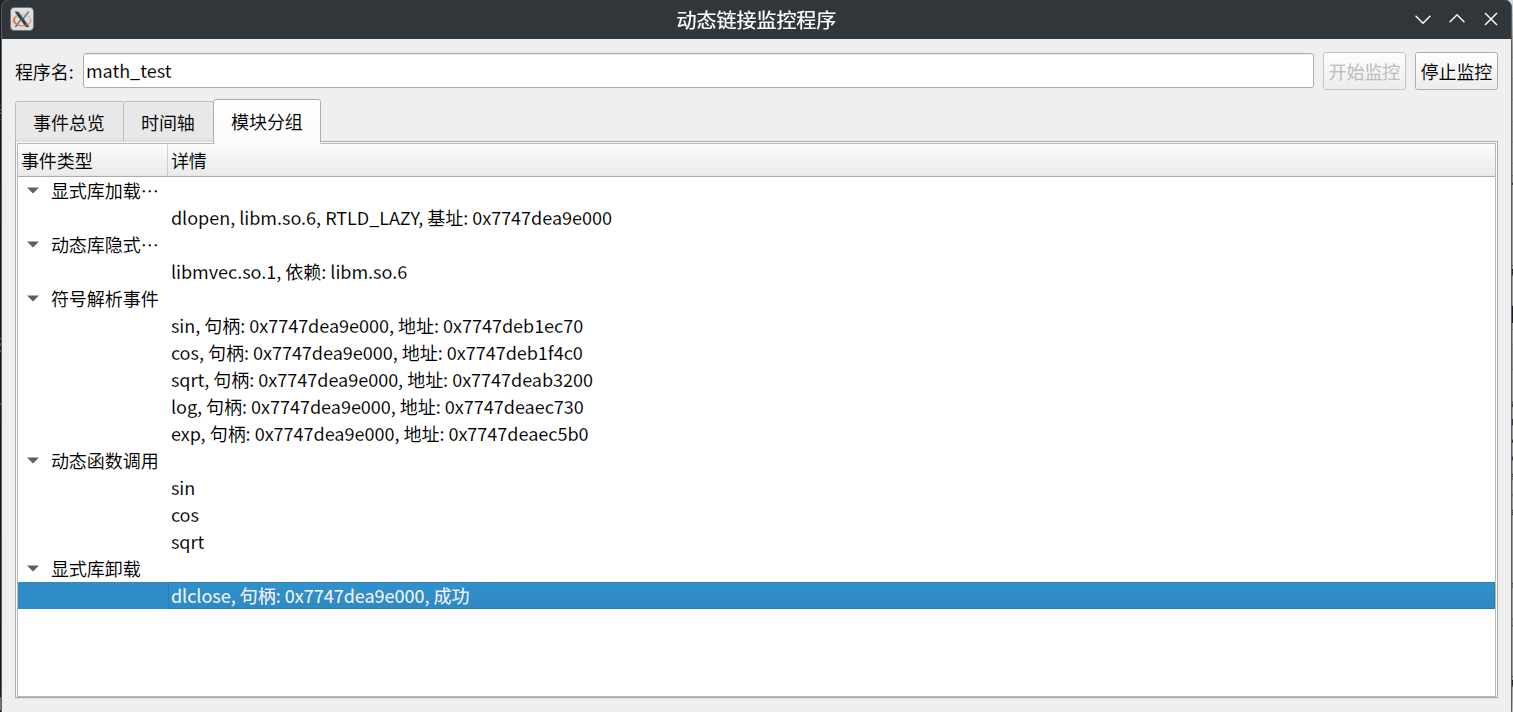


图6.2.2 测试时的时间轴视图

图6.2.3 测试时的模块分组视图