# 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、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

### 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

### 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 |
| 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 |

# 第4章 系统架构设计

## 4.1 系统架构整体设计

### 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，捕获关键事件和数据。
* 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4.1 三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。

## 4.2 系统架构组件设计

### 4.2.1 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，以实现对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 |

### 4.2.2 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。

### 4.2.3 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MainWindo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。

下表总结了前端程序主要组件及其核心功能：

表5 Qt5前端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，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6.1 测试时的时间线视图

图6.2 测试时的库加载事件视图

图6.3 测试时的符号解析事件视图

图6.4 测试时的库卸载事件视图

# 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。