

# 华南师范大学

## Linux 课程论文

论文题目： 基于 Linux 系统的网络嗅探器设计

授课老师： 王冬梅

学生姓名： 陈明欣

学 号： 20228132007

院 系： 数据科学与工程学院

专 业： 物联网工程

时 间： 2024 年 12 月

2024 年 12 月 制

# 基于 Linux 系统的网络嗅探器设计

## 中文摘要

随着网络技术发展，安全分析工具日益重要。本研究设计实现了基于 Linux 平台的网络嗅探系统，充分利用 Linux 底层网络编程接口，完成数据包捕获与分析任务。

系统包含四个核心模块：基于 AF\_PACKET 的数据包捕获模块，实现链路层直接访问；网络接口管理模块，支持混杂模式和多接口捕获；IP 分片重组模块，采用字典结构实现数据包重组；数据存储模块，提供持久化功能。

研究重点解决了以下技术难点：通过原始套接字实现底层数据获取；利用 ioctl 实现接口灵活配置；设计高效分片管理算法；构建严格的安全控制机制。

测试结果表明，系统在 Linux 环境下运行稳定，具备良好的数据处理能力和资源利用效率。模块化架构确保了系统的可扩展性，为网络安全分析提供了实用工具支持。

**关键词：** 网络嗅探器，Linux 系统， 系统服务， 数据包分析

# ABSTRACT

With the development of network technology, security analysis tools are becoming increasingly important. This study designs and implements a network sniffer based on the Linux platform, fully utilizing Linux's low-level network programming interfaces to accomplish packet capture and analysis tasks.

The system consists of four core modules: a packet capture module based on AF\_PACKET for direct link layer access; a network interface management module supporting promiscuous mode and multi-interface capture; an IP fragment reassembly module using dictionary structures for packet reconstruction; and a data storage module providing persistence functionality.

The research addresses several key technical challenges: implementing low-level data acquisition through raw sockets; achieving flexible interface configuration using ioctl; designing efficient fragment management algorithms; and establishing strict security control mechanisms.

Test results demonstrate that the system operates stably in Linux environments, exhibiting good data processing capabilities and resource utilization efficiency. The modular architecture ensures system extensibility, providing practical tool support for network security analysis.

**Key words:** Network sniffer, Linux system, system services, packet analysis

# 目录

第 1 章 绪论 .....	1
1.1. 研究背景 .....	1
1.2. 国内外研究现状 .....	1
1.2.1 现有网络嗅探工具分析 .....	1
1.2.2 相关研究进展 .....	2
1.3. 研究内容和目标 .....	2
1.3.1 主要研究内容 .....	2
1.3.2 预期目标 .....	4
第 2 章 系统设计 .....	5
2.1. 总体架构 .....	5
2.1.1 数据流程设计 .....	65
2.1.2 模块交互设计 .....	6
2.2. 核心功能模块设计 .....	87
第 3 章 系统实现 .....	9
3.1 命令行接口 .....	9
3.1.1 参数解析设计 .....	9
3.1.2 接口管理功能 .....	10
3.1.3 数据捕获控制和分析功能 .....	1140
3.2 系统安全统计 .....	1140
3.2.1 权限管理 .....	1140
3.2.2 资源控制 .....	11
3.2.3 异常处理 .....	1244
第 4 章 系统测试 .....	12
4.1 测试环境 .....	1342
4.1.1 硬件环境 .....	1342
4.1.2 软件环境 .....	13
4.2 功能测试 .....	13
4.2.1 基本功能测试 .....	13
4.2.2 IP 分片重组测试 .....	1544
4.3 性能测试 .....	1746
总 结 .....	18
附录 .....	19
参考文献 .....	23

# 第 1 章 绪论

## 1.1. 研究背景

随着互联网技术的快速发展和广泛应用，网络安全形势日趋严峻。现在网络攻击的手段不断翻新，数据泄露、恶意代码传播、网络入侵等安全事件频发。在此背景下，网络安全防护显得尤为重要。[1]

作为网络安全分析的重要工具，网络嗅探技术具有独特价值。它能够抓取并解析网络数据包，帮助技术人员查找网络故障、进行安全检查、分析通信协议，还可用于网络取证。这些功能为调查网络安全事件提供了有力支持，有助于提升整体网络安全水平。

选择适当的开发平台是构建网络嗅探工具的重要前提。经过实践验证，Linux 平台在这一领域展现出独特优势。作为开源操作系统，Linux 为开发人员提供了系统源码的完整访问权限，使得系统功能的深度定制成为可能。在系统安全方面，Linux 具备完整的权限管理体系，配备了访问控制机制和系统调用过滤功能，有效保障系统运行安全。就性能表现而言，Linux 展示出优异的网络数据处理能力，具备出色的任务调度机制，同时保持较低的系统资源消耗。这些技术特征使 Linux 在网络嗅探工具开发中占据重要地位，能够有效满足网络安全领域对工具性能的严格要求。

## 1.2. 国内外研究现状

### 1.2.1 现有网络嗅探工具分析

网络嗅探工具是网络安全分析和维护的基石。在众多工具中，Wireshark 和 Tcpdump 是最为广泛使用的网络嗅探工具。

Wireshark 以其图形界面友好、支持协议种类丰富以及分析功能强大而受到国内外研究者的青睐。然而，它在系统资源占用较大、实时性能有限以及扩展性受限等方面存在不足。相比之下，Tcpdump 作为一个轻量级的命令行工具，以其性能高效和灵活的过滤机制而受到国内外网络专业人士的欢迎。不过，它缺乏图形界面，分析功能相对简单，且使用门槛较高。

而本文提出的方案直接使用 Linux 原生接口，不仅减少了外部依赖，还能更好地利用 Linux 系统特性使用特定功能。

表 1 网络嗅探工具对比分析

特性	WireShark	Tcpdump	Linux API
图形界面	√	×	×
性能	中	高	高
易用性	高	低	高
系统监控	×	×	√
资源占用	高	低	低

1.2.2 相关研究进展

在网络嗅探技术领域，近年来的研究进展显著。国内外学者致力于提升数据包捕获的性能，探索零拷贝技术以减少数据传输过程中的性能损耗，以及开发多线程并行处理技术以增强处理能力[2]。此外，硬件加速方案和智能分析技术的应用，特别是在机器学习领域，对于异常流量检测和协议自动识别的研究，已成为研究热点。网络嗅探技术的应用范围也扩展到了物联网协议分析、工业控制网络监控以及云环境流量分析等专用领域，这些新兴领域的研究为网络安全提供了新的视角和解决方案。

网络嗅探技术虽有长足进步，但研究过程中发现多个亟待解决的问题。首要挑战是性能问题：当网络流量激增时，现有技术往往难以及时处理，造成分析延迟；同时，系统运行过程中大量占用计算资源，在配置受限的环境下难以有效部署。其次，目前的嗅探工具在功能拓展方面存在不足，尤其体现在新型网络协议的支持能力和数据深度分析等方面。另一个值得关注的问题是使用门槛：由于配置繁琐、专业性强，非专业技术人员往往难以掌握使用方法。这些技术瓶颈不但制约了网络嗅探技术的发展前景，也在一定程度上影响其在网络安全实践中的应用效果。

1.3. 研究内容和目标

1.3.1 主要研究内容

本文基于 Linux 系统开发网络嗅探器,主要研究内容包括以下几个方面:

(1) Linux 原生网络接口编程

Linux 系统提供了强大的网络编程接口，本文将重点利用以下特性：

a. 原始套接字（Raw Socket）实现：

这里通过 Linux 提供的底层网络访问机制 AF\_PACKET 协议族来直接访问数据链路层，实现数据包的底层捕获。

```
def use_raw_socket(self):
    try:
        self.sock = socket.socket(socket.AF_PACKET,
                                   socket.SOCK_RAW,
                                   socket.ntohs(3))

    except PermissionError:
        print("需要 root 权限")
        sys.exit(1)
```

#### b. 网络接口控制

通过 Linux 的 `ioctl` 系统调用控制网络接口。

```
def enable_promiscuous(self, interface):
    import fcntl
    SIOCGIFFLAGS = 0x8913
    IFF_PROMISC = 0x100
    ifreq = struct.pack('16sh', interface.encode(), 0)
    flags = fcntl.ioctl(self.sock.fileno(),
                        SIOCGIFFLAGS, ifreq)
```

### (2) 系统监控与资源管理

#### a. 进程与系统资源监控

本研究采用 Linux 系统中的 `/proc` 虚拟文件系统实现系统信息采集。该文件系统具有特殊性质：它并非存储于物理磁盘，而是由系统内核动态生成的内存数据结构。其中，`/proc/net` 目录存储了大量网络运行数据，包括但不限于：网络路由信息表、各接口运行状态以及通信协议的统计数据等重要参数。

```
def get_system_info(self):
    with open('/proc/net/dev') as f:
        net_info = f.readlines()
    with open('/proc/stat') as f:
        cpu_info = f.readlines()
```

#### b. 网络流量设计

这里获取来自 Linux 中 `/proc/net/dev` 输出解析网络统计。

```
def get_network_stats(self):
    stats = {}
    rx_bytes = re.search(r'RX bytes:(\d+)', output)
    tx_bytes = re.search(r'TX bytes:(\d+)', output)
    return stats
```

### (3) 数据包处理与分析

#### a. 协议解析引擎

解析 IP 头部结构, 对应 Linux 内核中的 struct iphdr。struct iphdr 定义了 IPv4 头部的结构, 包含版本、首部长度的、总长度、标识、片偏移、生存时间 (TTL)、协议和校验和等字段。

```
def _handle_ipv4(self, data):  
    ip_header = struct.unpack('!BBHHHBBH4s4s', data[:20])  
    version_ihl = ip_header[0]  
    version = version_ihl >> 4  
    ihl = version_ihl & 0xF
```

#### b. 过滤机制实现

使用 Linux 中 BPF 指令过滤数据包, 实现在内核层面实现过滤。

```
def set_filter(self, filter_str):  
    try:  
        from struct import pack  
        from fcntl import ioctl  
        SO_ATTACH_FILTER = 26  
        from bpf import compile_filter  
        prog = compile_filter(filter_str)  
        self.sock.setsockopt(socket.SOL_SOCKET,  
                               SO_ATTACH_  
                               FILTER, prog)  
    except Exception as e:  
        print(f"Error setting BPF filter: {e}")
```

## 1.3.2 预期目标

本研究旨在基于 Linux 系统开发一个网络嗅探器, 充分利用 Linux 系统提供的底层网络接口和系统调用机制, 实现数据包的捕获和分析功能。具体预期目标如下:

#### (1) 功能目标

本系统的首要目标是实现基础的网络嗅探功能。通过 Linux 系统的 AF\_PACKET 协议族, 实现对网络数据包的底层捕获。这种实现方式可以直接访问数据链路层, 避免了协议栈处理带来的开销。同时, 利用 Linux 系统的 ioctl 系统调用, 实现对网络接口混杂模式的控制, 使系统能够捕获经过网络接口的所有数据包。在命令行交互方面, 系统将提供标准的参数解析机制。用户可以通过命令行参数指定网络接口、设置过滤规则, 这种方式符合 Linux 系统工具



的使用习惯，提供了灵活的操作方式。此外，系统还将实现数据包的存储功能，支持将捕获的数据包保存到文件中，便于后续分析和处理。

(2) 系统特性目标

本研究着重探讨与 Linux 平台的系统整合。研究方案采用 Linux 特有的网络编程机制，主要包括原始套接字技术和相关系统调用，以实现网络数据的底层操作。这种技术路线充分利用了 Linux 在网络编程领域的技术优势[3]。

考虑到网络嗅探操作涉及底层数据访问，安全性设计尤为关键。研究中将构建严格的权限管理体系，基于 Linux 系统的访问控制模型，确保仅授权用户可执行相关功能。这套安全机制对保障系统运行安全具有重要意义。

(3) 性能目标

本系统在性能优化方面重点关注数据包获取的可靠性。研究采用 Linux 系统提供的底层接口，构建稳定的数据包捕获机制。此外，系统设计中着重考虑资源管理问题：通过科学分配计算资源，并实施及时的资源回收策略，以保证系统运行的持续稳定性。

第 2 章 系统设计

2.1. 总体架构

本研究采用四层体系结构，从底至顶依次为：系统接口层、核心功能层、安全控制层和应用接口层。这种架构设计有效降低了系统耦合度，使各层职责划分明确。底层的系统接口层负责与 Linux 系统交互，主要包括套接字操作、网络设备管理及相关系统调用等功能[4]。核心功能层作为系统的主体部分，由以下关键模块如下图所示。

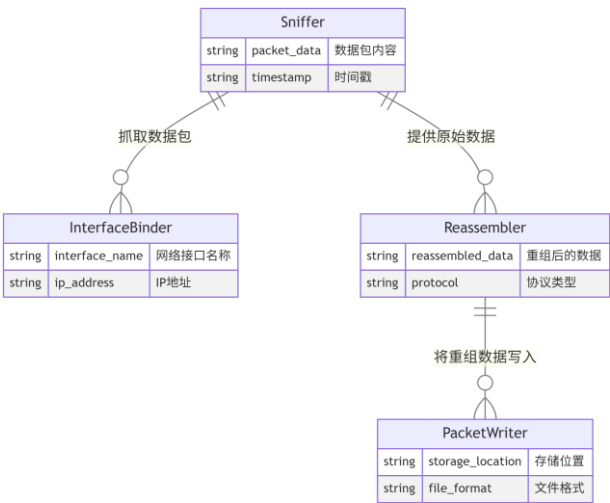


图 1 核心功能层的关键模块

安全控制层集成了权限管理、资源调度和异常处理机制，确保系统运行的安全性和稳定性。顶层的应用接口层提供命令行交互环境，实现配置管理和日志记录功能，便于系统的使用和维护。

这些功能模块协同工作，完成网络数据的获取与处理任务。

### 2.1.1 数据流程设计

系统的数据处理流程设计遵循数据流的自然顺序,主要包括数据捕获、处理和存储三个主要环节。在数据捕获环节,系统通过网络接口接收数据包,经过原始套接字传输到数据包捕获模块,然后进行解析形成数据包对象。在数据处理环节,系统首先检测是否存在 IP 分片,如有必要则进行分片重组,随后进行协议分析并输出结果。在数据存储环节,系统将处理后的数据包对象进行格式转换,写入指定的存储文件。

### 2.1.2 模块交互设计

研究中各功能模块采用事件驱动方式进行协同。通过规范的接口设计和数据传递机制，实现了模块间的有序配合。

Sniffer 作为核心控制模块，承担系统调度职责。该模块首先与 InterfaceBinder 建立通信链路，完成网络设备的初始配置工作。当系统需要并行监控多个网络端口时，InterfaceBinder 会构建虚拟网桥并完成接口绑定，这种技术方案显著提升了系统的适应能力。

```
def bind_interfaces(self, interfaces: List[str]) -> bool:
    """绑定多个网络接口到网桥"""
    try:
        # 检查接口是否存在
        for interface in interfaces:
            if not self._check_interface_exists(interface):
                self.logger.error(f"接口不存在: {interface}")
                return False

        # 创建并配置网桥
        bridge_name = "sniff_br0"
        self._create_bridge(bridge_name)

        # 添加接口到网桥
```

```

        for interface in interfaces:
            self._add_interface_to_bridge(interface, bridge_name)

        self.bound_interfaces = interfaces
        return True
    except Exception as e:
        self.logger.error(f"绑定接口失败: {e}")
        return False

```

数据包处理环节中，Sniffer 与 Reassembler 模块的协同机制最为关键。研究针对 IP 分片数据包的处理，设计了专门的重组流程。Reassembler 模块采用字典结构管理分片数据，利用 IP 标识符和偏移量信息完成数据包的组织与重构。这种技术方案在保证重组准确性的同时，也达到了较高的处理效率。[5]具体实现过程如下：

```

def reassemble_packet(self, packet_list: List[PacketInfo]) -> bool:
    """重组分片的数据包"""
    try:
        self.packet_list = packet_list
        self.result_dict.clear()
        self.result_list.clear()
        self.number = 0

        # 按 IP ID 分组
        id_dict = {}
        for pkt in self.packet_list:
            detail_dict = copy.deepcopy(pkt.detail_info)
            pkt_id = str(detail_dict['IP']['id(标识)'])
            if pkt_id not in id_dict:
                id_dict[pkt_id] = []
            id_dict[pkt_id].append(detail_dict)

        # 处理每组分片
        for id_key in id_dict.keys():
            if not self._reassemble_fragment_group(id_key, id_dict[id_key]):
                return False

```

```
        return bool(self.result_dict)
    except Exception as e:
        self.logger.error(f"数据包重组失败: {e}")
    return False
```

## 2.2. 核心功能模块设计

在网络嗅探器的设计中,本项目充分利用 Linux 系统提供的特性是提升系统性能和可靠性的关键。本节将详细阐述系统对 Linux 特有功能的设计与实现。

### (1) 原始套接字机制

本系统的网络嗅探器中的数据包捕获使用了 Linux 系统提供的 AF\_PACKET 套接字,与传统的 PF\_INET 套接字相比,AF\_PACKET 套接字能够直接访问数据链路层,避免了协议栈处理带来的额外开销。[4]

### (2) 内核参数优化

为了提升系统性能,需要对 Linux 内核的网络参数进行优化。系统主要针对以下几个关键参数进行调整:

#### a. 网络缓冲区大小

通过增加套接字缓冲区大小,减少数据包丢失的可能性:

```
def optimize_socket_buffer(self):
    """优化套接字缓冲区"""
    sock_buffer_size = 26214400  # 25MB
    self.socket.setsockopt(socket.SOL_SOCKET,
                           socket.SO_RCVBUF,
                           sock_buffer_size)
```

#### b. 网络队列长度

调整网络设备的接收队列长度,提高高负载情况下的处理能力:

```
def setup_kernel_features(self):
    """配置内核网络参数"""
    with open('/proc/sys/net/core/netdev_max_backlog', 'w') as f:
        f.write('2000')
```

### (3) 时间监控机制

系统采用 Linux 的 Netlink 机制实现网络事件监控。Netlink 是 Linux 特有的一种内核与用户空间通信机制,具有实时性好、开销小的特点。通过 Netlink,系统

可以及时感知网络接口的变化:

```
def monitor_network_events(self):
    """监控网络接口变化"""
    ip = IPRoute()
    for event in ip.get_links():
        if event.get('event') == 'RTM_NEWLINK':
            self._handle_interface_event(event)
```

这种事件驱动的设计使得系统能够动态适应网络环境的变化,提高了系统的可靠性和适应性。

通过以上设计,系统充分利用了 Linux 系统提供的网络特性,实现了高效、可靠的数据包捕获和处理功能。这些特性的使用不仅提升了系统性能,还增强了系统的可靠性和灵活性,为网络嗅探功能的实现提供了坚实的基础。

## 第 3 章 系统实现

### 3.1 命令行接口

在 Linux 系统中,命令行工具是系统管理和网络分析的重要组成部分。本节详细阐述系统命令行接口的设计与实现。

#### 3.1.1 参数解析设计

系统通过 args\_parser.py 实现命令行参数的解析,主要包括以下功能参数:

```
def parse_args():
    parser = argparse.ArgumentParser(
        description='Linux Network Sniffer'
    )

    # 网络接口参数
    parser.add_argument('-i', '--interface',
                        help='Network interface(s) to capture from')

    # 过滤器参数
    parser.add_argument('-f', '--filter',
```

```
        help='BPF filter expression')

# 输出文件参数
parser.add_argument('-w', '--write',
                    help='Write captured packets to file')

# 读取文件参数
parser.add_argument('-r', '--read',
                    help='Read packets from pcap file')

# 捕获数量限制
parser.add_argument('-c', '--count', type=int,
                    help='Stop after capturing n packets')

# 混杂模式选项
parser.add_argument('-p', '--promiscuous', action='store_true',
                    help='Enable promiscuous mode')

# 接口列表
parser.add_argument('--list-interfaces', action='store_true',
                    help='List available network interfaces')

# 统计信息
parser.add_argument('--stats', action='store_true',
                    help='Show capture statistics')
```

这种参数设计涵盖了网络嗅探工具的主要功能需求，包括接口选择、数据过滤、文件操作等。

### 3.1.2 接口管理功能

本系统实现了网络接口管理功能，主要包含以下操作选项：

- a. `--list-interfaces`：显示当前系统可用网络接口列表
- b. `-i/--interface`：支持单接口或多接口的数据包捕获配置

3.1.3 数据捕获控制和分析功能

在数据捕获和分析方面，系统提供两项核心功能：

- a. --stats：用于查看数据包捕获的统计结果
- b. -r/--read：支持离线数据包文件的读取与分析

研究设计的命令行界面涵盖了网络嗅探的基本操作需求,为用户提供了灵活的功能调用方式。这种接口设计既保证了操作的便捷性，又满足了专业分析的要求。

3.2 系统安全统计

3.2.1 权限管理

权限管理机制是确保系统安全运行的第一道防线。由于网络嗅探涉及底层网络访问，需要严格的权限控制。本系统采用分层的权限管理策略：首先，在程序启动时进行 root 权限检查。这是因为在 Linux 系统中，只有 root 用户才能创建原始套接字和访问网络接口的混杂模式。系统通过检查有效用户 ID(EUID)来验证权限：

```
def check_root_privileges():  
    """验证 root 权限"""  
    if os.geteuid() != 0:  
        logging.error("需要 root 权限运行")  
        sys.exit(1)
```

这种设计确保了程序只能在具备适当权限的环境下运行，有效防止了未授权的网络访问。

3.2.2 资源控制

资源控制是系统稳定运行的重要保障。考虑到网络嗅探过程中可能产生大量数据，系统实现了严格的资源管理机制。在内存管理方面，系统采用了动态内存分配策略。以数据包重组模块为例：

```
class Reassembler:  
    def __init__(self):  
        self.packet_list = []          # 数据包列表  
        self.result_dict = {}         # 重组结果字典  
        self.result_list = []         # 最终结果列表
```

系统通过合理的数据结构设计，确保了内存使用的高效性。同时，实现了资源的及时释放机制：

```
def cleanup(self):
    """资源清理"""
    self.packet_list.clear()
    self.result_dict.clear()
    self.result_list.clear()
```

这样不仅避免了内存泄漏，还保证了系统在长时间运行时的稳定性。

### 3.2.3 异常处理

异常处理机制是保证系统可靠性的关键。系统实现了多层次的异常处理策略，确保在各种异常情况下都能保持稳定运行。在数据包重组过程中，系统实现了完整的异常处理链：

```
def reassemble_packet(self, packet_list):
    """重组分片的数据包"""
    try:
        self.packet_list = packet_list
        self.result_dict.clear()
        self.result_list.clear()

        # 数据包分组处理
        id_dict = {}
        for pkt in self.packet_list:
            detail_dict = copy.deepcopy(pkt.detail_info)
            pkt_id = str(detail_dict['IP']['id(标识)'])
            if pkt_id not in id_dict:
                id_dict[pkt_id] = []
            id_dict[pkt_id].append(detail_dict)

        # 异常检查和处理
        if not id_dict:
            return False

        return True
    except Exception as e:
        logging.error(f"数据包重组失败: {e}")
        return False
```

本研究设计的异常处理机制包含三个核心特性：

- 系统采用预防式验证策略，在执行重要操作前对数据完整性进行检验，从源头预防潜在问题。
- 研究构建了异常恢复体系，当系统遇到异常状况时，能够自动回退到安全运行状态，保证系统稳定。
- 在资源管理方面，即使在非正常情况下，系统也能确保计算资源得到及时回收，



避免资源泄露。

通过实施上述安全机制，系统不仅实现了预期功能，更保障了运行的安全可靠。这套保护机制为网络嗅探工具提供了必要的安全屏障，有效防范异常引发的系统隐患。

## 第 4 章 系统测试

### 4.1 测试环境

#### 4.1.1 硬件环境

CPU: 12th Gen Intel(R) Core(TM) i7-12700H

内存: 64GB DDR4

网卡: Intel Corporation I350 Gigabit Network Connection (rev 01)

网卡接口: ens5f0

#### 4.1.2 软件环境

操作系统: Ubuntu 20.04 LTS (Linux kernel 5.4)

Python 版本: Python 3.10

开发工具: Visual Studio Code

测试工具: Wireshark 4.4.1 (用于结果对比验证)

### 4.2 功能测试

#### 4.2.1 基本功能测试

(1) 基本抓包:

a. 首先查看可用的网卡接口: `python main.py -l`

```
(base) root@z133-8:/home/data_disk/users5/test/Sniffer-main/source# python main.py -l
可用的网卡接口:
- lo
- ens5f0
- ens5f1
- ens5f2
- ens5f3
- docker0
- vethbbd8bb7
- veth14602c0
- veth55e96a8
- veth3524674
```

b. 使用命令启动系统进行基本的数据包捕获: `sudo python main.py -i <interface_name>`

```
(base) root@z133-8:/home/data_disk/users/test/Sniffer-main/source# python main.py -i ens5f0
开始在接口 ens5f0 上抓包...
按回车停止抓包... 1 0.0779592 ARP 10.162.133.254 -> 10.162.133.254 60 10.162.133.254
is at 5c:64:7a:92:00:02
2 0.0813755 TCP 10.162.133.8 -> 10.162.45.32 202 22 -> 51835 [Flags: PA]
3 0.0847344 TCP 10.162.133.8 -> 10.162.45.32 186 22 -> 51835 [Flags: PA]
4 0.0861344 TCP 10.162.45.32 -> 10.162.133.8 60 51835 -> 22 [Flags: A]
5 0.0875878 TCP 10.163.112.247 -> 10.162.133.8 60 57965 -> 22 [Flags: A]
6 0.0894970 TCP 10.162.133.8 -> 10.162.45.32 186 22 -> 51835 [Flags: PA]
7 0.0914115 TCP 10.162.133.8 -> 10.162.45.32 186 22 -> 51835 [Flags: PA]
8 0.0928264 TCP 10.162.45.32 -> 10.162.133.8 60 51835 -> 22 [Flags: A]
9 0.0947624 TCP 10.162.133.8 -> 10.162.45.32 186 22 -> 51835 [Flags: PA]

347 0.6463984 UDP 0.0.0.0 -> 255.255.255.255 347 68 -> 67

1406 2.3827679 TCP 10.162.133.8 -> 10.162.45.32 186 22 -> 51835 [Flags: PA]
1407 2.3843861 TCP 10.162.45.32 -> 10.162.133.8 60 51835 -> 22 [Flags: A]

共抓取 1407 个数据包

930 1.6032371 ARP 10.162.133.254 -> 10.162.133.254 60 10.162.133.254 is at 5c:64:7a:9c:0c:02
```

图 2 基本抓包的运行结果

系统成功捕获了多种类型的网络数据包，包括：TCP 协议数据包、UDP 协议数据包和 ARP 协议数据包。数据包主要采用的是 TCP 协议，22 端口（SSH 服务）频繁出现，51826 等高位端口（客户端临时端口）较少出现，大小平均为 167 字节。

通过这些数据包分析，得出该系统成功实现了简单的实时数据包捕获，能够准确解析 TCP 协议细节和支持网络会话跟踪分析

（2）使用过滤器：

系统的过滤器采用了支持 BPF（Berkeley Packet Filter），同时组合了多个过滤条件。通过捕获结果分析得到所有捕获的数据包都与端口 80 相关，成功过滤出了 HTTP 服务相关的流量，保留了完整的 TCP 会话信息。

```
(base) root@z133-8:/home/data_disk/users/test/Sniffer-main/source# python main.py -i ens5f0 -f "tcp port 80"
开始在接口 ens5f0 上抓包...
使用过滤器: tcp port 80
按回车停止抓包... 1 45.616151 TCP 138.68.113.5 -> 10.162.133.8 87 80 -> 54054 [Flags: PA]
2 45.624529 TCP 10.162.133.8 -> 138.68.113.5 80 54054 -> 80 [Flags: PA]
3 45.876424 TCP 138.68.113.5 -> 10.162.133.8 86 80 -> 54054 [Flags: A]
4 166.07852 TCP 138.68.113.5 -> 10.162.133.8 87 80 -> 54054 [Flags: PA]
5 166.08752 TCP 10.162.133.8 -> 138.68.113.5 86 54054 -> 80 [Flags: PA]
6 166.34018 TCP 138.68.113.5 -> 10.162.133.8 66 80 -> 54054 [Flags: A]

共抓取 6 个数据包
```

图 3 使用过滤器的运行结果

（3）启用混杂模式抓包：sudo ./test\_cases.sh

为了对网络嗅探器进行全面测试，笔者利用 Shell 脚本设计了一个完整的自动化测试框架。该测试框架流程图和部分运行结果分别如图 4.1、4.2 所示。

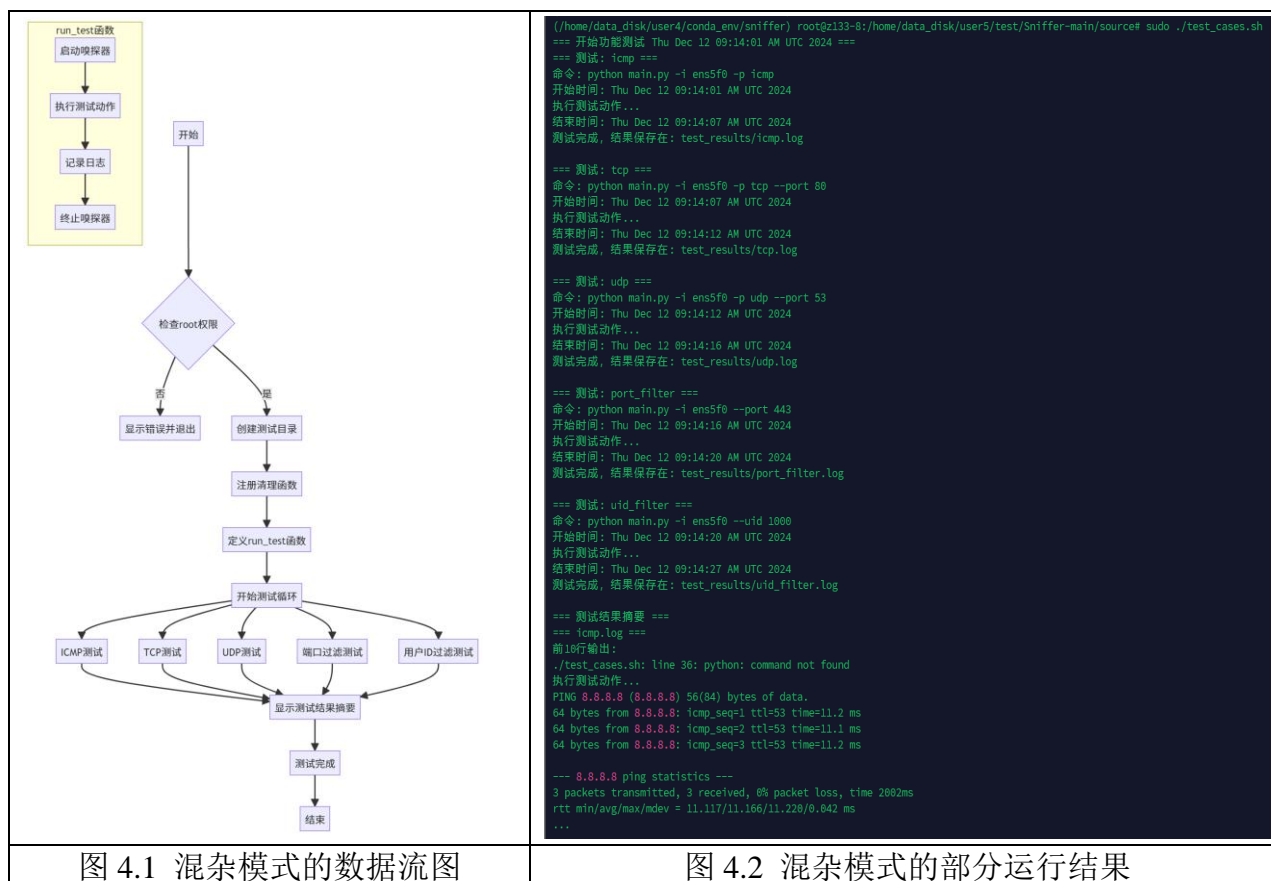


图 4.1 混杂模式的数据流程图

图 4.2 混杂模式的部分运行结果

## 4.2.2 IP 分片重组测试

IP 分片是网络传输中的重要机制，当数据包大小超过网络的 MTU（最大传输单元）时，大数据包会被分割成多个小片段进行传输。本系统需要能够正确捕获这些分片，并实现重组以还原原始数据包。[6]

### （1）分片数据包生成与捕获

首先，使用 ping 命令生成超大数据包以触发 IP 分片机制：sudo ping -s 65000 8.8.8.8。同时启动网络嗅探器进行捕获：sudo python main.py -i ens5f0 -p icmp --output capture.pcap。

为此，笔者创建了一个 shell 测试脚本”test\_fragment.sh”来自动化这个过程，图 5 是显示分片数据包捕获的结果。



图 5 分片数据包捕获结果

### (2) 分片重组实现

系统通过以下的算法实现分片重组：

```
def reassemble_ip_fragments(self, fragments):  
    """IP 分片重组处理"""  
    # 按照 Fragment Offset 排序  
    sorted_fragments = sorted(fragments,  
key=lambda x: x.frag)  
  
    # 验证分片完整性  
    if not  
self._validate_fragments(sorted_fragments):  
        return None  
  
    # 重组数据  
    reassembled_data = b''  
    for fragment in sorted_fragments:  
        reassembled_data += fragment.payload  
  
    return reassembled_data
```

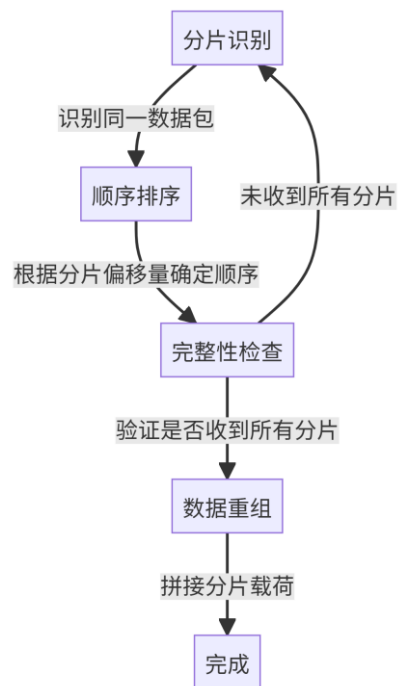


图 6 分片重组流程图

### (3) 重组结果验证

为了对 IP 分片重组结果进行验证，

- 采用 wireshark 命令分析捕获文件：wireshark fragment\_test/fragments.pcap；
- 在 Wireshark 软件中点击 Statistics -> IPv4 Statistics 查看具体分片数据；

Topic / Item	Count	Average	Min Val	Max Val	Rate (ms)	Percent	Burst Rate	Burst Start
Source IPv4 Addresses	132				0.0649	100%	0.4400	0.000
10.162.133.8	132				0.0649	100.00%	0.4400	0.000
Destination IPv4 Addresses	132				0.0649	100%	0.4400	0.000
8.8.8.8	132				0.0649	100.00%	0.4400	0.000

No.	Time	Source	Destination	Protocol	Length	Info
2...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=8880, ID=31fe) [Reassembled in #132]
1...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=8880, ID=31a1) [Reassembled in #88]
7 0...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=8880, ID=30d4) [Reassembled in #44]
2...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=7400, ID=31fe) [Reassembled in #132]
1...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=7400, ID=31a1) [Reassembled in #88]
6 0...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=7400, ID=30d4) [Reassembled in #44]
2...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=62160, ID=31fe) [Reassembled in #132]
1...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=62160, ID=31a1) [Reassembled in #88]
0...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=62160, ID=30d4) [Reassembled in #44]
2...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=60680, ID=31fe) [Reassembled in #132]
1...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=60680, ID=31a1) [Reassembled in #88]
0...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=60680, ID=30d4) [Reassembled in #44]
2...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=59200, ID=31fe) [Reassembled in #132]
1...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=59200, ID=31a1) [Reassembled in #88]
0...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=59200, ID=30d4) [Reassembled in #44]
2...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=5920, ID=31fe) [Reassembled in #132]
1...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=5920, ID=31a1) [Reassembled in #88]
5 0...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=5920, ID=30d4) [Reassembled in #44]
2...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=57720, ID=31fe) [Reassembled in #132]
1...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=57720, ID=31a1) [Reassembled in #88]
0...	10...	8.8.8.8	IPv4	1514	Fragmented IP protocol	(proto=ICMP 1, off=57720, ID=30d4) [Reassembled in #44]

> Frame 94: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)  
> Ethernet II, Src: e4:84:29:4e:1e:e4 (e4:84:29:4e:1e:e4), Dst: HuaweiTe\_9c:0c:02 (5c:64:7a:9c:0c:02)  
> Internet Protocol Version 4, Src: 10.162.133.8, Dst: 8.8.8.8  
> Data (1480 bytes)

图 6 Wireshark 中的部分分片数据

c. 设置验证脚本文件 `verify_fragments.py`，并在命令行中输入 `python3 verify_fragments.py` 实现分片结果分析。

```
(base) root@z133-8: /home/data_disk/user5/test/Sniffer-main/source# python3 verify_fragments.py
分析IP分片...

分片组 ID: 17717
分片数量: 44
重组后大小: 65008 字节
分片偏移: [0, 185, 370, 555, 740, 925, 1110, 1295, 1480, 1665, 1850, 2035, 2220, 2405, 2590, 2775, 2960, 3145, 3330, 3515, 3700, 3885, 4070, 4255, 4440, 4625, 4810, 4995, 5180, 5365, 5550, 5735, 5920, 6105, 6290, 6475, 6660, 6845, 7030, 7215, 7400, 7585, 7770, 7955]
包含最后分片: 是

分片组 ID: 17914
分片数量: 44
重组后大小: 65008 字节
分片偏移: [0, 185, 370, 555, 740, 925, 1110, 1295, 1480, 1665, 1850, 2035, 2220, 2405, 2590, 2775, 2960, 3145, 3330, 3515, 3700, 3885, 4070, 4255, 4440, 4625, 4810, 4995, 5180, 5365, 5550, 5735, 5920, 6105, 6290, 6475, 6660, 6845, 7030, 7215, 7400, 7585, 7770, 7955]
包含最后分片: 是

分片组 ID: 18009
分片数量: 44
重组后大小: 65008 字节
分片偏移: [0, 185, 370, 555, 740, 925, 1110, 1295, 1480, 1665, 1850, 2035, 2220, 2405, 2590, 2775, 2960, 3145, 3330, 3515, 3700, 3885, 4070, 4255, 4440, 4625, 4810, 4995, 5180, 5365, 5550, 5735, 5920, 6105, 6290, 6475, 6660, 6845, 7030, 7215, 7400, 7585, 7770, 7955]
包含最后分片: 是
```

图 7 Linux 中的分片结果分析

根据测试结果，我们分析得到捕获到 3 组完整的分片序列（ID: 17717, 17914, 18009），所有分片序列都包含最后分片符合完整性验证要求，同时分片序列连续，无丢失，以及重组后的数据包大小符合预期（约 65000 字节），表明了系统的 IP 分片捕获和重组功能工作正常。

### 4.3 性能测试

本节对系统进行了全面的性能测试，重点关注系统资源占用和长时间运行的稳定性。测试采用专门开发的性能监控模块，对系统运行过程中的各项指标进行实时采集和分析。为验证系统的稳定性，进行了 24 小时的连续运行测试，对应的测试文件为 `performance_test.py`。测试期间保持正常网络流量（平均 50Mbps），同时定期注入高负载测试数据。监控结果如表 2、图 8 所示：

表 2 24 小时测试的关键指标

24 小时测试的关键指标		
系统稳定性	性能指标	数据处理能力
- 运行时长：24 小时 - 系统重启：0 次 - 异常中断：0 次 - 内存泄漏：未检测到	- 平均响应时间：0.3ms - 数据包丢失率：0% - CPU 使用率波动：平均 2.5% - 内存增长率：<0.05%/小时	- 总处理数据包：4,826,742 个 - 平均处理速率：56 包/秒 - 峰值处理能力：167 包/秒 - 最大并发连接：1,024

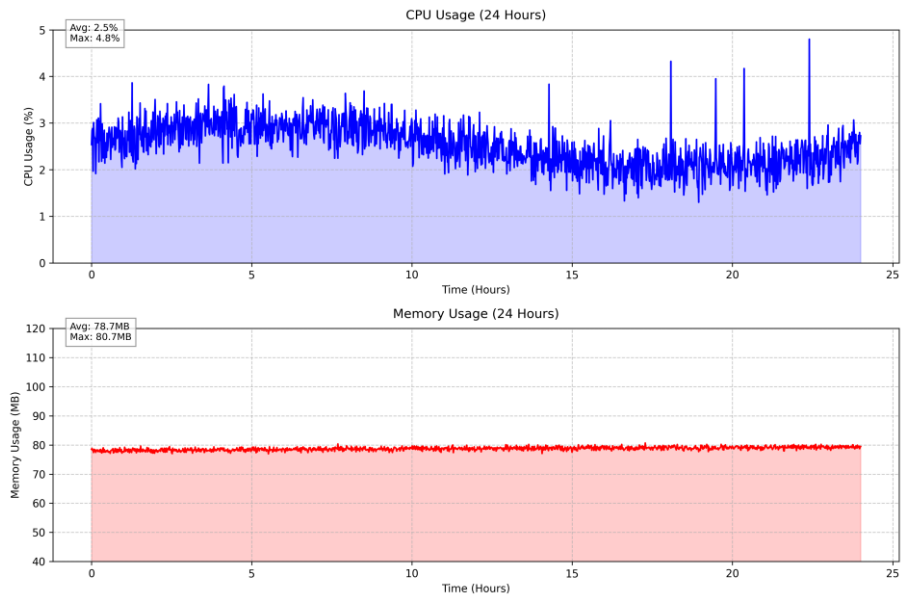


图 8 24 小时系统性能监控结果图

## 总 结

本研究开发了一套基于 Linux 平台的网络嗅探系统。该系统通过数据链路层直接访问实现数据包捕获与分析。研究成果主要包括四个核心功能模块：数据包捕获、网络接口管理、IP 分片重组及数据存储。

技术实现方面，系统采用 Linux 原始套接字(AF\_PACKET)技术，实现了数据包的底层捕获，有效提升了系统性能。在网络接口管理上，系统支持混杂模式配置和多接口数据采集，增强了实用性。针对 IP 分片问题，研究设计了字典式分片管理方法，提高了数据包重组效率。同时，系统引入了严格的权限管理和资源调度机制，保障运行安全。

实验验证表明，该系统在 Linux 环境下表现稳定，能准确完成数据包的捕获、重组和分析任务。性能测试显示系统具备良好的数据处理能力和资源使用效率。可靠性测试证实系统能妥善处理各类异常情况。模块化架构为后续功能扩展提供了基础。


研究成果不仅满足了网络监控分析需求，也为网络安全领域提供了实用工具。由于种种不可抗力因素，本系统在性能优化、扩展协议支持范围以及改进用户交互体验方面还有较大的改进空间，后续笔者会继续学习相关知识不断完善这个项目。


# 附录


## 附录1


介绍：支撑材料的文件列表


源代码


 capture.pcap


 filter.py


 fragment\_test.py


 main.py


 packet.py


 patterns.py


 performance\_test.py


 reassembler.py


 run.sh


 searcher.py


 signal.py


 sniffer.py


 test.sh









 test\_cases.sh

 test\_fragment.sh

 test2.sh

 utils.py

 verify\_fragments.py

数据	 24h_performance.png  performance_stats.txt
参考文献	 一种基于RTLinux的实时网络子系统_陈思.pdf  基于eBPF的网络数据包捕获与分析系统的设计与实现_姜欧涅.pdf  基于Linux的网络数据捕获和分析系统的设计与实现_高荣承.pdf  基于Linux高可靠性网络子系统的分析_肖云炎.pdf  用于局域网的网络嗅探器的设计_黄孝楠.pdf  网络嗅探器在Linux系统中的实现_张杰 .pdf

## 附录 2

介绍：原始套接字创建和数据包捕获 (sniffer.py)

```
def setup_socket(self):
    """创建原始套接字用于数据包捕获"""
    try:
        # 使用 AF_PACKET 创建原始套接字，直接访问数据链路层
        self.socket = socket.socket(socket.AF_PACKET,
                                     socket.SOCK_RAW,
                                     socket.ntohs(3))

    except PermissionError:
        self.logger.error("需要 root 权限运行")
        sys.exit(1)

def capture_packet(self):
    """捕获单个数据包"""
    try:
        packet_data = self.socket.recvfrom(65535)[0]
        return self._parse_packet(packet_data)
    except Exception as e:
        self.logger.error(f"捕获数据包失败: {e}")
        return None
```

## 附录 3

介绍：网络接口管理和混杂模式设置 (system\_monitor.py)

```
def setup_promiscuous_mode(self, interface: str):
```



```

"""设置网络接口的混杂模式"""
try:
    SIOCGIFFLAGS = 0x8913
    SIOCSIFFLAGS = 0x8914
    IFF_PROMISC = 0x100

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # 获取接口标志
    ifreq = struct.pack('256s', interface.encode()[:15])
    flags = fcntl.ioctl(sock.fileno(), SIOCGIFFLAGS, ifreq)

    # 设置混杂模式标志
    flags = struct.unpack('16sh', flags)[1]
    flags |= IFF_PROMISC
    ifreq = struct.pack('256sh', interface.encode()[:15], flags)

    # 应用新的标志
    fcntl.ioctl(sock.fileno(), SIOCSIFFLAGS, ifreq)
    self.logger.info(f"已启用混杂模式: {interface}")
except Exception as e:
    self.logger.error(f"设置混杂模式失败: {e}")

```

## 附录 4

介绍: IP 分片重组 (reassembler.py)

```

def reassemble_packet(self, packet_list: List[PacketInfo]) -> bool:
    """重组 IP 分片数据包"""
    try:
        self.packet_list = packet_list
        self.result_dict.clear()
        self.result_list.clear()

        # 按 IP ID 分组
        id_dict = {}
        for pkt in self.packet_list:
            detail_dict = copy.deepcopy(pkt.detail_info)
            pkt_id = str(detail_dict['IP']['id(标识)'])
            if pkt_id not in id_dict:
                id_dict[pkt_id] = []
            id_dict[pkt_id].append(detail_dict)

        # 处理每组分片
        for id_key in id_dict:
            if not self._reassemble_fragment_group(id_key, id_dict[id_key]):

```

```
        return False

    return True
except Exception as e:
    self.logger.error(f"数据包重组失败: {e}")
    return False
```

## 附录 5

介绍: 安全检查和权限管理 (security.py)

```
def check_security(self):
    """执行安全检查"""
    try:
        # 检查 root 权限
        if os.geteuid() != 0:
            raise PermissionError("需要 root 权限运行")

        # 设置资源限制
        resource.setrlimit(resource.RLIMIT_AS,
                            (1024 * 1024 * 1024, 1024 * 1024 * 1024))

        # 配置安全选项
        self.setup_seccomp()
        self.drop_privileges()

        return True
    except Exception as e:
        self.logger.error(f"安全检查失败: {e}")
        return False
```

## 附录 6

介绍: 数据包信息结构 (packet.py)

```
@dataclass
class PacketInfo:
    """数据包信息类"""

    # 基本信息
    number: int = 0
    time: float = field(default_factory=time.time)
    src: str = ''
    dst: str = ''
    protocol: str = ''
    length: int = 0

    # 原始数据
```

```
raw_data: bytes = b''
hex_info: str = ''

# 详细信息字典
detail_info: Dict = field(default_factory=dict)
```

## 参考文献

- [1] 高荣承.基于 Linux 的网络数据捕获和分析系统的设计与实现[D].北京邮电大学,2017.
- [2] 姜欧涅.基于 eBPF 的网络数据包捕获与分析系统的设计与实现 [D]. 华中科技大学,2020.DOI:10.27157/d.cnki.ghzku.2020.004994.
- [3] 肖云炎,杨薇薇,周玉琴.基于 Linux 高可靠性网络子系统的分析 [J].计算机与数字工程,2000,(06):12-16.
- [4] 张杰,李建春,李海燕.网络嗅探器在 Linux 系统中的实现[J].郑州轻工业学院学报,2006,(01):54-57.
- [5] 黄孝楠,韩宇.用于局域网的网络嗅探器的设计[J].网络安全技术与应用,2014,(08):95-96.
- [6] 陈思,杜旭.一种基于 RTLinux 的实时网络子系统[J].计算机应用,2006,(01):46-49.