西南科技大学

Southwest University of Science and Technology

信息安全产品开发实践

题目名称：基于 WinpCap 的协议分析程序

学院名称	计算机科学与技术学院
专业名称	信息安全
学生姓名	
学　　号	
指导教师	毛光灿、韦勇、魏蓉

二〇二五年六月

设计题目

摘要：随着信息技术的迅速发展，网络已成为现代社会生活和生产的重要基础设施，深刻影响着人们的衣食住行。然而，网络环境日益复杂，资源的广泛共享也带来了诸多安全隐患和性能挑战，促使网络分析、故障诊断、性能评估与安全防护工作愈发重要。网络扫描技术作为实现网络状态监测、问题定位及安全漏洞发现的关键手段，发挥着不可替代的作用。通过网络嗅探技术，能够高效识别本地计算机及网络中的异常状况。本文基于 WinPcap 库，结合 Windows MFC 开发平台，设计并实现了一套集数据包捕获、协议解析、网络行为分析及流量统计于一体的综合性网络分析工具。该系统支持实时捕获和解析网络数据包，能够准确反映网络运行状态，及时预警潜在的安全风险，有效提升网络管理与维护的效率和水平。实验验证结果表明，该工具在网络性能测试与安全性检测领域具有显著的应用价值和推广潜力。

关键词：WinPcap；数据包捕获；过滤规则；协议解析；网络监控

Title

Abstract: With the rapid development of information technology, the network has become an important infrastructure for modern social life and production, which has a profound impact on people's clothing, food, housing and transportation. However, the increasingly complex network environment and the extensive sharing of resources have brought many security risks and performance challenges, making network analysis, fault diagnosis, performance evaluation, and security protection more and more important. Network scanning technology plays an irreplaceable role as a key means to achieve network status monitoring, problem location and security vulnerability discovery. Through network sniffing technology, it can efficiently identify abnormal conditions in the local computer and network. Based on the WinPcap library and combined with the Windows MFC development platform, this paper designs and implements a comprehensive network analysis tool that integrates packet capture, protocol parsing, network behavior analysis and traffic statistics. The system supports real-time capture and analysis of network packets, which can accurately reflect the network operation status, timely warn potential security risks, and effectively improve the efficiency and level of network management and maintenance. Experimental results show that the tool has significant application value and promotion potential in the field of network performance testing and security testing.

Key words: WinPcap; packet capture; filter rules; protocol parsing; Network monitoring

目 录

第 1 章 需求分析......5
1.1 系统背景......5
1.2 选题目的......5
1.3 说明......5
1.4 具体需求......6
1.4.1 网卡选择与数据捕获......6
1.4.2 数据处理与显示......6
1.4.3 数据过滤......6
1.4.4 数据包详情分析......6
1.4.5 网络行为监测......6
1.4.6 数据保存与读取......7
1.4.7 用户界面......7
1.4.8 数据持久化与数据库存储模块......7
1.4.9 DNS 协议数据解析模块......7
1.4.10 高性能无锁并发队列模块......7
1.4.11 前端可视化的实现......8
第 2 章 框架设计......9
2.1 设计思路......9
2.2 系统整体架构......9
2.3 核心模块设计......10
2.3.1 多网卡数据捕获模块......10
2.3.2 协议解析模块......11
2.3.3 数据处理与行为分析模块......12
2.3.4 用户交互模块......13
2.3.5 数据库管理模块......13
2.3.6 前端可视化模块......14
第 3 章 基于 Winpcap 的数据包捕获与分析程序开发流程......16
3.1 预备知识......16
3.1.1 网络协议......16
3.2 winPcap......16
3.2.1 WinPacp 简介......16
3.2.2 WinPcap 的组成与结构......17
3.2.3 WinPcap 的基本原理......19
3.2.4 WinPcap 常用函数说明......20
3.3 程序分析......22
3.4 开发环境准备......22
3.4.1 硬件与软件要求......22
3.4.2 MFC 简介......23
3.4.3 工程创建与配置......23

3.5 核心模块设计与实现 26
3.5.1 数据包捕获模块 26
3.5.2 协议解析模块 29
3.5.3 数据过滤模块 30
3.5.4 用户交互模块 31
3.5.5 数据库管理模块 32
3.5.6 前端可视化模块 37
第 4 章 实现功能测试 40
4.1 功能测试 40
4.1.1 列出本机的网卡列表，选择要监听的网卡，实时捕获所有流经网卡的数据包 40
4.1.2 分析捕获到的数据包的包头和数据，按照各种协议的格式进行格式化显示 40
4.1.3 捕获过程中可以提前结束捕获 40
4.1.4 输入过滤条件，对捕获到的数据进行筛选 41
4.1.5 对数据包中各个协议数量进行统计 41
4.1.6 显示被选中的数据包头详细信息 41
4.1.7 以十六进制显示被选中的数据包内容 43
4.1.8 以本地文件形式保存和读取已捕获的数据包信息 43
4.1.9 用户行为监控 44
4.1.10 数据包统计前端可视化 46
第 5 章 结论 47
5.1 完成度 47
5.2 不足与提高点 47
5.3 总结 47
第 6 章 参考文献 48
第 7 章 附录 49

第 1 章 需求分析

1.1 系统背景

随着信息化时代的到来，计算机网络已成为社会运行和信息交流的核心基础设施。各类网络应用不断涌现，网络规模和复杂度迅速提升，带来了数据流量激增及多样化的通信协议体系。在这种复杂环境下，网络的性能保障和安全防护面临严峻挑战。网络协议作为网络通信的基本规则和载体，其准确分析对于理解网络行为、诊断故障、检测入侵及优化性能具有重要意义。

传统的网络管理手段往往难以满足实时、高效且精细化的分析需求。基于 WinPcap 的协议分析程序，能够直接访问网络接口，实时捕获和解析传输的数据包，提供底层协议的数据细节，有助于深入洞察网络通信过程。通过对协议数据的细致解析，不仅可以及时发现网络异常和安全威胁，还能为网络性能调优提供有力支持。

因此，设计并实现一款基于 WinPcap 的协议分析程序，具备捕获、解析、统计及可视化功能，能够为网络管理员和安全分析人员提供强大的技术工具，提升网络监控和维护的智能化水平，具有重要的理论意义和实际应用价值。

1.2 选题目的

本课题旨在设计并实现一款基于 WinPcap 的协议分析程序，通过对网络中传输的数据包进行实时捕获与解析，深入识别常见通信协议及其行为特征，从而实现对网络流量的有效监控与统计分析。通过构建具备协议识别、数据过滤、内容解析及图形化展示等功能的应用工具，为网络管理者提供直观、高效的分析手段，提升网络故障排查和安全隐患识别的能力，为网络性能优化与信息安全保障提供技术支持

1.3 说明

本软件旨在截取和分析各种网络数据包，显示其详细信息，并进行基础的网络协议分析和行为监控。通过该软件，用户能够全面了解当前网络的运行状况。软件由数据包捕获器、数据包分析器和网络行为分析器，前端可视化，数据库存储五大部分组成。数据包捕获器负责实时截取网络中的数据包，并将其存储以供后续分析；数据包分析器对捕获的数据包进行解析，提取并显示封包的详细信息，如源地址、目的地址、协议类型等；网络行为分析器则通过分析数据包的传输行为，帮助用户识别异常网络活动和潜在

的安全威胁，从而提升网络管理和安全防护能力。前端可视化则是通过 python 语言利用 flash 框架实现，便于实现对数据包信息的统计。数据库用来存储抓取到的数据包信息以及配合实现前端可视化。

数据包捕获器：实际上是根据用户的需要（例如，捕获某种协议，某种端口等），将网络上流动到本地网卡的网络数据包抓取下来，保存到内存或磁盘上。

数据包分析器：就是从第一步得到的数据包分析，分析它的协议类型、行为及内容等，并以一定的方式显示给用户。

网络行为分析器：该系统可以帮助监控用户网络行为，查看以便于帮助用户解决网络上的一些问题。

1.4 具体需求

1.4.1 网卡选择与数据捕获

1) 能够列出本机的所有可用网卡列表。
2) 用户可自由选择需要监控的网卡进行数据包捕获操作。
3) 系统需及时、有效地捕获选定网卡上传输的网络数据包。

1.4.2 数据处理与显示

1) 对捕获到的原始数据包进行高效处理。
2) 将处理后的数据包信息清晰、有序地展示给用户，确保用户能准确捕获到目标数据。

1.4.3 数据过滤

1) 支持用户设置过滤规则（如基于协议、IP 地址、端口号等）。
2) 能够依据设定的过滤规则对捕获到的数据包进行实时筛选，帮助用户快速聚焦于所需数据。

1.4.4 数据包详情分析

1) 用户可选择查看任意捕获数据包的详细信息（如各层协议头、负载内容等）。
2) 提供直观的展示方式，方便用户对数据进行深入分析。

1.4.5 网络行为监测

1) 具备对网络流量模式、连接状态等网络行为进行监测的能力，提供基本的网络状态洞察。

1.4.6 数据保存与读取

1) 支持将捕获的数据包信息保存为本地文件（如.dmp 格式）。
2) 支持从本地文件读取已保存的数据包信息。
3) 实现数据离线分析功能，并为后续系统参考或考证提供数据支持。

1.4.7 用户界面

1) 提供友好的图形用户界面（GUI）。
2) 界面布局合理，操作便捷，使用户能够方便地使用各项功能。
3) 注重视觉体验，提供清晰、美观的数据展示效果。

1.4.8 数据持久化与数据库存储模块

1) 系统应能够使用 SQLite 数据库进行本地数据持久化，便于后续网络数据分析与日志溯源。

2) 系统应支持基础数据包的存储功能，能够从 pcap_pkthdr 与 datapkt 结构中提取各类字段，完成协议、端口、地址等数据的格式转换与绑定后插入数据库。

3) 系统应支持保存识别出的敏感信息，包括敏感信息类型（如“密码”、“Token”等）及其文本内容，绑定到对应的基础数据包记录（通过 base_id 关联）。

1.4.9 DNS 协议数据解析模块

1）系统应具备 DNS 协议数据报文的解析能力，能够从原始数据包中提取 DNS 报文结构信息，并填充到统一的数据结构 headerPack 中。

2）系统应首先判断输入数据的合法性，如数据长度是否小于 DNS 报文头部 dns_header 所需的最小长度，避免越界访问与解析异常。

3）系统应能够正确解析 DNS 报文中的 **问题部分（Question Section）**，包括域名（QName）、查询类型（QType）与类（QClass）等字段，并支持多个查询项的顺序解析。

1.4.10 高性能无锁并发队列模块

1）系统需提供一个支持多线程并发访问的队列结构，用于在不加锁的情况下高效执

行入队与出队操作。

2）队列基于链表结构实现，使用原子操作维护节点指针，确保数据一致性与线程安全。

3）支持 enqueue 方法将数据加入队列尾部，dequeue 方法从队列头部取出数据，空队列时返回失败。

4）队列初始化时需构造哑节点，避免空指针访问；析构时需释放所有节点，防止内存泄漏。

5）禁止复制构造和赋值，确保每个队列实例唯一操作其内部资源。

1.4.11 前端可视化的实现

1）能够对存储进数据库的数据包信息进行可视化的展示

2）对于不能够访问数据库的开发人员也能够了解抓取到的数据包信息。

第 2 章 框架设计

2.1 设计思路

（1）列出本机的网卡列表，选择要监听的网卡，实时捕获所有流经网卡的数据包；

（2）分析捕获到的数据包的包头和数据，按照各种协议的格式进行格式化显示；

（3）捕获过程中可以随时中止或提前结束捕获；

（4）可以选择过滤条件，对捕获到的数据进行筛选；

（5）对数据包中各个协议数量进行统计；

（6）显示被选中的数据包头详细信息；

（7）以十六进制显示被选中的数据包内容；

（8）实现用户网络行为监控并展示；

（9）以本地文件形式保存和读取已捕获的数据包信息；

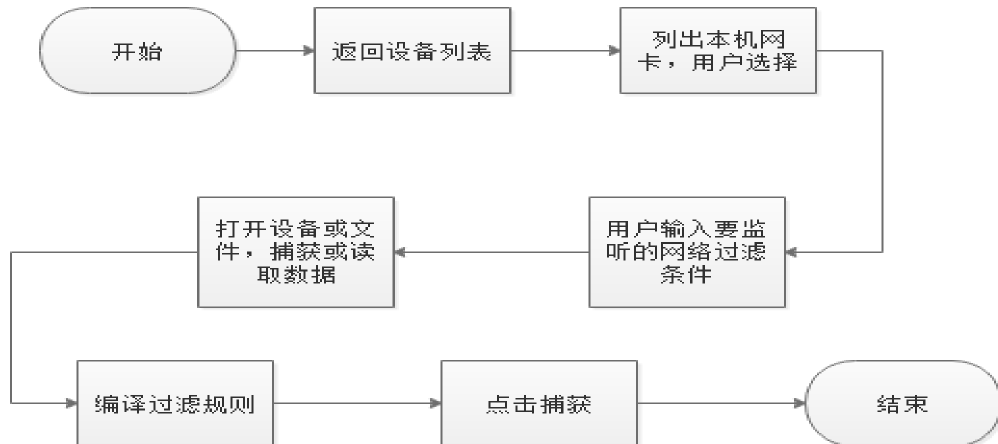


图 2-1 程序流程图

2.2 系统整体架构

本系统采用**分层**架构设计，将复杂的网络协议分析功能拆分为多个逻辑清晰的层次，各层之间通过标准化接口交互，既保证了模块独立性，又便于系统扩展与维护。整体架构从下至上可分为**数据捕获层、协议解析层、数据处理层和用户交互层**四个主要层次，各层协同工作实现完整的网络数据包分析功能。

数据捕获层通过 WinPcap 枚举本地网卡，支持多网卡同时监控，利用 pcap_open、pcap_loop 等函数实时捕获原始数据包，解决多网卡并行监控、数据包高效采集问题。协

议解析层对捕获的原始数据进行多层协议解析，按照网络协议栈结构（链路层、网络层、传输层、应用层）逐层解析数据包，提取各层协议字段。数据处理层结合过滤规则筛选数据包，通过多线程+缓冲队列优化处理效率，统计协议分布、网络行为（游戏/聊天等），实现敏感信息（账号密码、涉密内容）嗅探。用户交互层基于 MFC 构建 GUI，可视化呈现数据包列表、协议统计、行为监控结果，支持数据包详情（十六进制+协议头解析）查看、数据存储/读取，保障操作便捷性与分析直观性。

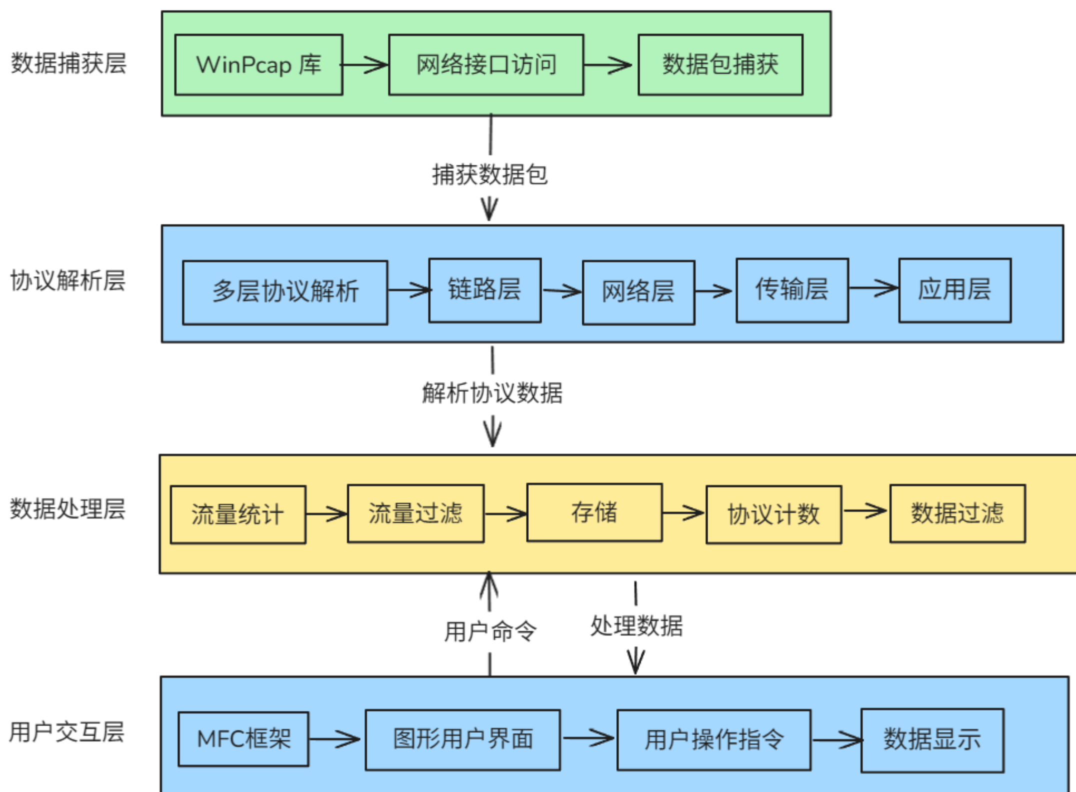


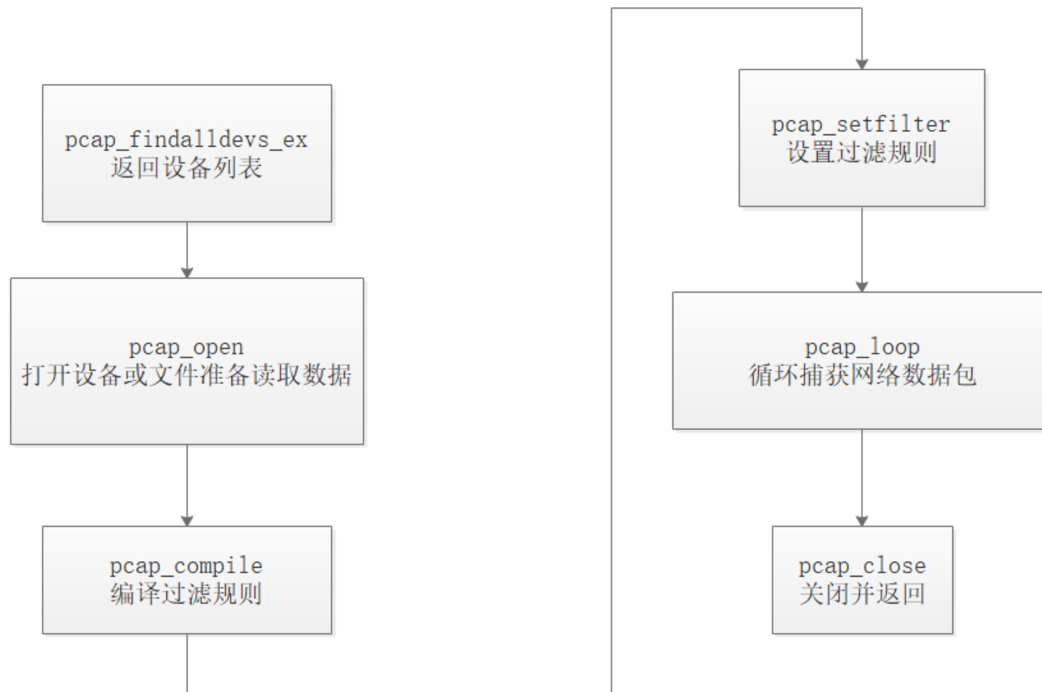
图 2-2 系统架构图

系统架构设计充分考虑了功能模块化和接口标准化，各层之间通过定义明确的数据结构和接口进行交互，例如数据捕获层通过特定数据结构将捕获的数据包传递给协议解析层，协议解析层解析后生成标准化的协议头部信息供数据处理层使用，数据处理层处理后的数据通过事件机制通知用户交互层更新界面显示。这种分层架构设计使得系统各部分职责明确，便于开发、测试和维护，同时也为系统功能扩展提供了良好的基础。

2.3 核心模块设计

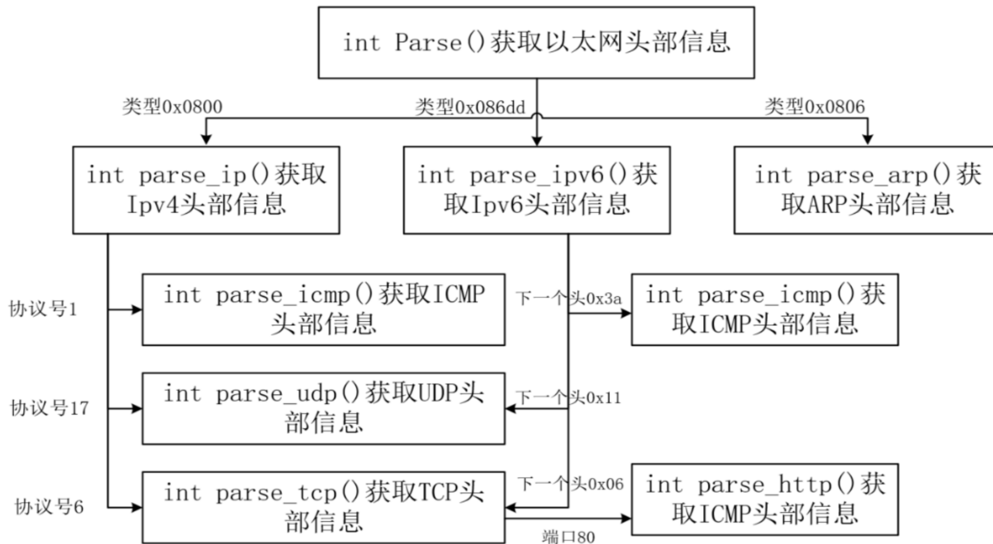
2.3.1 多网卡数据捕获模块

支持 WinPcap 枚举网卡→用户选网卡→抓包线程采集原始数据包→通过 packetCallback 传递给协议解析模块。



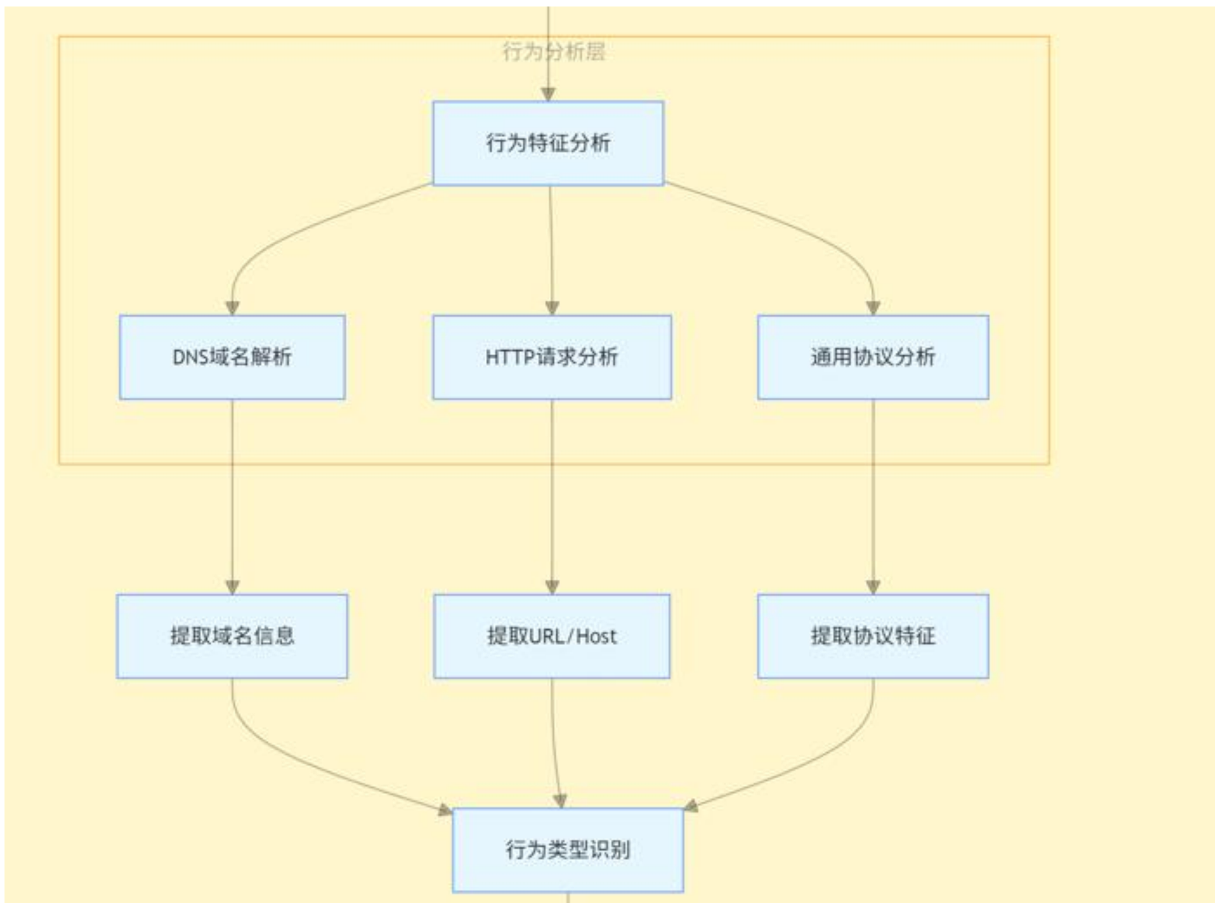
2.3.2 协议解析模块

按 TCP/IP 栈分层处理→提取 IP/端口/应用数据→识别网络行为、嗅探敏感信息→传递结构化数据到处理层。



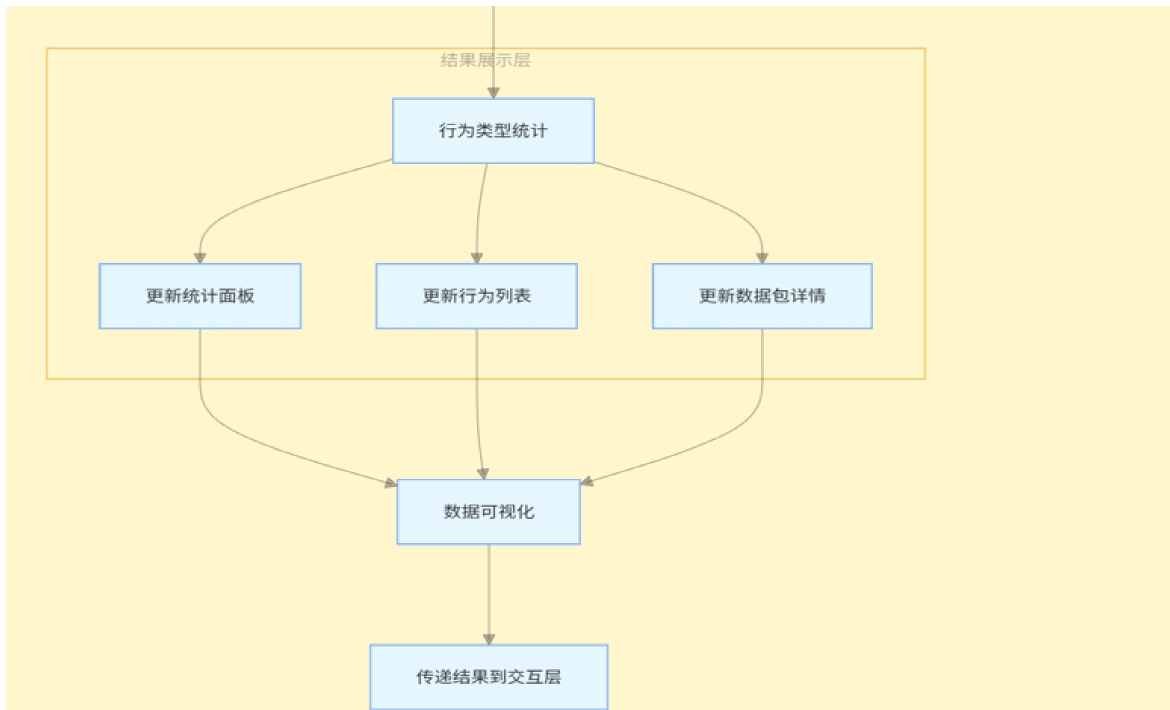
2.3.3 数据处理与行为分析模块

开始抓包 → 数据包捕获 → 协议解析 → 行为分析 →UI 更新



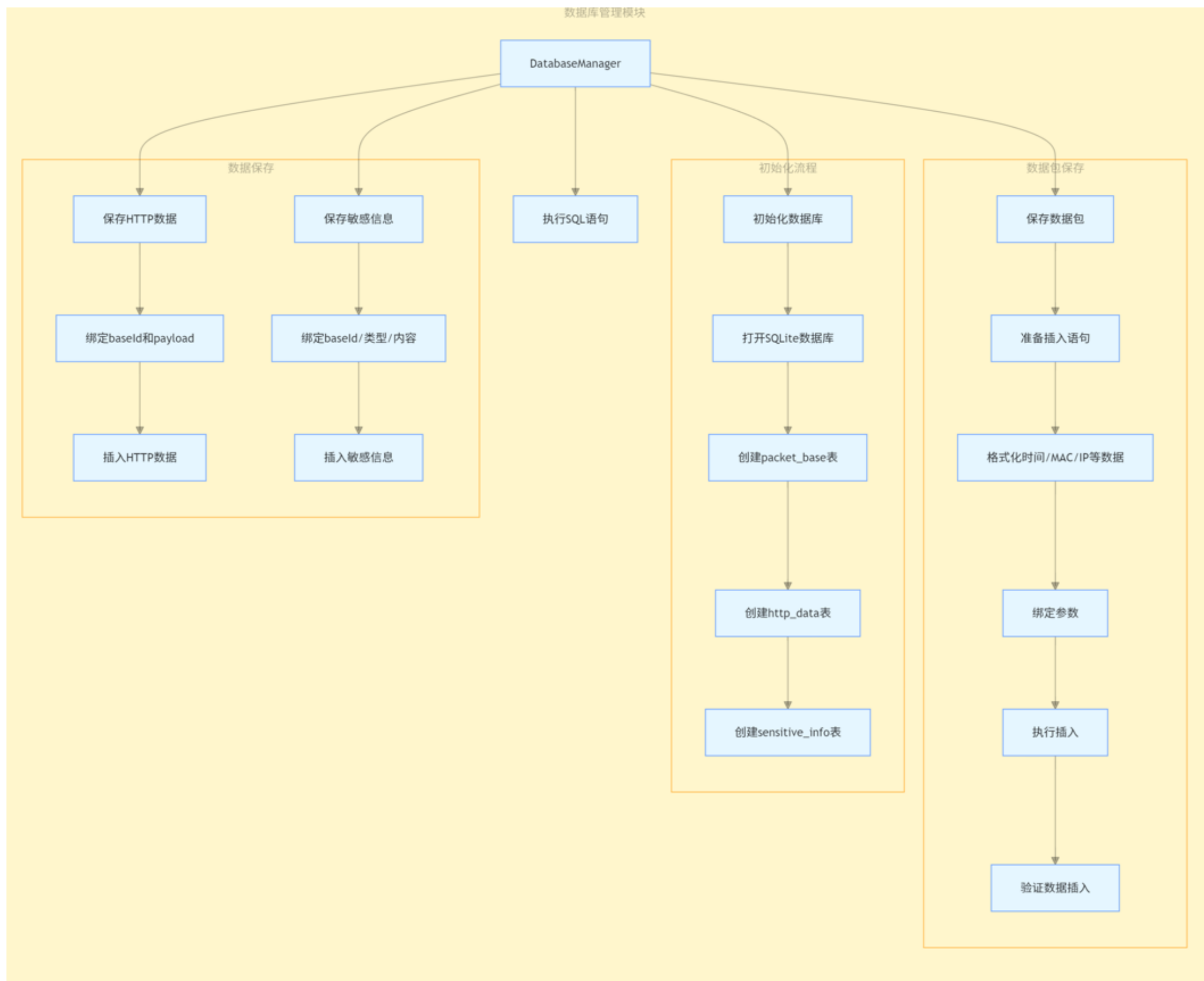
2.3.4 用户交互模块

可视化展示数据包列表、协议头详情、行为统计→接收用户操作→调用底层模块执行存储/读取。

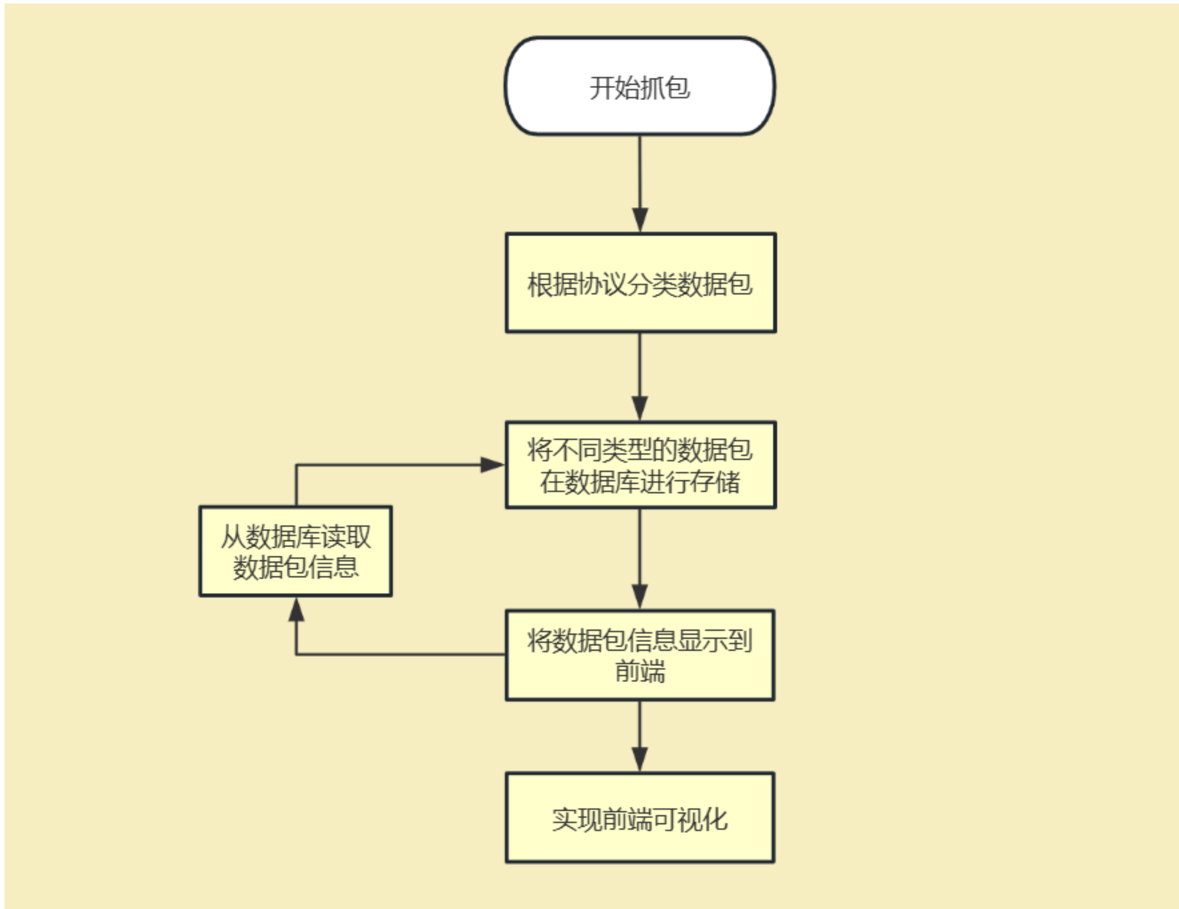


2.3.5 数据库管理模块

整体分为初始化、SQL 执行和三种数据保存功能。始化数据库及表结构，存储网络数据包的基础信息，保存 HTTP 请求 / 响应数据，存储识别出的敏感信息，提供 SQL 执行的通用接口



2.3.6 前端可视化模块


开始抓包
根据协议分类数据包
将不同类型的数据包
在数据库进行存储
从数据库读取
数据包信息
将数据包信息显示到
前端
实现前端可视化

第 3 章 基于 Winpcap 的数据包捕获与分析程序开发流程

3.1 预备知识

3.1.1 网络协议

网络协议为计算机网络中进行数据交换而建立的规则、标准或约定的集合。

（1）应用层协议

HTTP：超文本传输协议 <端口号 80>， 面向事务的应用层协议。

DHCP：动态主机分配协议，使用 UDP 协议工作，主要有两个用途：给内部网络或网络服务供应商自动分配 IP 地址，给用户或者内部网络管理员作为对所有计算机作中央管理的手段。实现即插即用连网。

DNS: 域名解析协议 DNS 协议将域名转换为 IP 地址（也可以将 IP 地址转换为相应的域名地址）。

（2）传输层协议

TCP：传输控制协议提供可靠的面向连接的服务，传输数据前须先建立连接，结束后释放。可靠的全双工信道。可靠、有序、无丢失、不重复。

UDP：用户数据报协议发送数据前无需建立连接，不使用拥塞控制，不保证可靠交付，最大努力交付。

（3）网络层协议

IP：网络之间互连的协议。

ARP：即地址解析协议，实现通过 IP 地址得知其物理地址。

ICMP：Internet 控制报文协议。它是 TCP/IP 协议族的一个子协议，用于在 IP 主机、路由器之间传递控制消息。

3.2 winPcap

3.2.1 WinPacp 简介

WinPcap: Winpcap 是一个基于 Win32 平台的，用于捕获网络数据包并进行分析的开源库.大多数网络应用程序通过被广泛使用的操作系统元件来访问网络，比如 sockets。这是一种简单的实现方式，因为操作系统已经妥善处理了底层具体实现细节（比如协议处理，封装数据包等等），并且提供了一个与读写文件类似的，令人熟悉的接口。然而，

有些时候，这种“简单的方式”并不能满足任务的需求，因为有些应用程序需要直接访问网络中的数据包。也就是说，那些应用程序需要访问原始数据包，即没有被操作系统利用网络协议处理过的数据包，WinPcap 产生的目的，就是为 Win32 应用程序提供这种访问方式。

WinPcap 提供的功能是捕获原始数据包，无论它是发往某台机器的，还是在其他设备（共享媒介）上进行交换的。在数据包发送给某应用程序前，根据用户指定的规则过滤数据包，将原始数据包通过网络发送出去，收集并统计网络流量信息。以上这些功能需要借助安装在 Win32 内核中的网络设备驱动程序才能实现，再加上几个动态链接库 DLL。所有这些功能都能通过一个强大的编程接口来表现出来,易于开发，并能在不同的操作系统上使用。

3.2.2 WinPcap 的组成与结构

WinPcap 由一个数据包监听设备驱动程序（NPF）、一个底层的动态连接库（packet.dll）和一个高层的不依赖于操作系统的静态库（wpcap.dll）共三个部分构成。这里，NPF 在操作系统的内核级，packet.dll、wpcap.dll 在用户级。

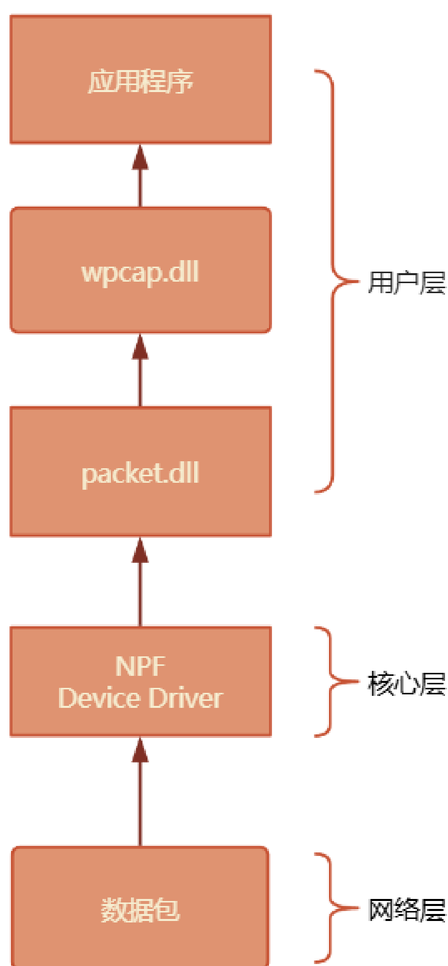


图 3 WinPcap 结构

1）数据包监听设备驱动程序

技术实现上，为了实现抓包，系统必须绕过操作系统的协议栈来访问在网络上传输的原始数据包（raw packet）。这就要求 WinPcap 的一部分运行在操作系统核心内部，直接与网络接口驱动交互。由于这个部分是系统依赖（system dependent）的，在 Winpcap 的解决方案中它被视为是一个设备驱动，称作 NPF（Netgroup Packet Filter）。

2）底层的动态连接库（packet.dll）和高层静态库（wpcap.dll）

为了方便编程，WinPcap 必须提供一个编程接口（API），这就是 WinPcap 的底层的动态连接库（packet.dll）和高层静态库（wpcap.dll）。这里，packet.dll 提供了一个底层 API，伴随着一个独立于 Microsoft 操作系统的编程接口，这些 API 可以直接用来访问驱动的函数；wpcap.dll 导出了一组更强大的与 libpcap 一致的高层抓包函数库（capture primitives），这些函数使得数据包的捕获以一种与网络硬件和操作系统无关的

方式进行。

底层动态链接库运行在用户层，它将应用程序和数据包监听设备驱动程序隔离开来，使得应用程序可以不加修改地在不同的 WINDOWS 系统上运行。高级的静态链接库和应用程序编译在一起，它使用低级动态链接库提供的服务，向应用程序提供完善的监听接口。

3.2.3 WinPcap 的基本原理

抓包是 WinPcap 的基本功能，也是 NPF 最重要的操作。在抓包的时候，驱动（例如 NIC Driver）使用一个网络接口监视着数据包，并将这些数据包完整无缺地投递给用户级应用程序。

WinPcap 的 NPF 抓包主要依靠两个组件。

1）数据包过滤器（filter）。

数据包过滤器决定是否接收进来的数据包并把数据包拷贝给监听程序。数据包过滤器是一个有布尔输出的函数。如果函数值是 true，抓包驱动拷贝数据包给应用程序；如果是 false，数据包将被丢弃。NPF 数据包过滤器更复杂一些，因为它不仅决定数据包是否应该被保存，而且还决定要保存的字节数。被 NPF 驱动采用的过滤系统来源于 BSD Packet Filter（BPF），一个虚拟处理器可以执行伪汇编书写的用户级过滤程序。

应用程序采用用户自定义的过滤器并使用 wpcap.dll 将它们编译进 BPF 程序。然后，应用程序使用 BIOCSETF IOCTL 写入核心态的过滤器。这样，对于每一个到来的数据包该程序都将被执行，而满足条件的数据包将被接收。与传统解决方案不同，NPF 不解释（interpret）过滤器，而是执行（execute）它。由于性能的原因，在使用过滤器前，NPF 提供一个 JIT 编译器将它转化成本地的 80x86 函数。当一个数据包被捕获，NPF 调用这个本地函数而不是调用过滤器解释器，这使得处理过程相当快。

2）循环缓冲区（Buffer）。

NPF 的循环缓冲区用来保存数据包以免丢失（如果一个包符合过滤器的要求，就被复制到循环缓冲区）。一个保存在缓冲区中的数据包有一个头，它包含了一些主要的信息，例如时间戳和数据包的大小，注意：它不是协议头。另外，循环缓冲区以队列插入的方式来保存数据包，提高数据的存储效率。程序员可以以组的方式将数据包从 NPF 缓冲区拷贝到应用程序，这样就提高了性能，因为它降低了读的次数。如果一个数据包到来的时候缓冲区已经满了，那么该数据包将被丢弃，这时就发生了丢包现象。

3）Network Tap

是一个用于探听网络中所有数据流的函数。

4）数据统计

为了提高数据处理的速度，WinPcap 将统计和监听功能移到内核中，这样避免了将任何数据都传递给用户。WinPcap 通过使用从 NPF 中得到的过滤器来执行一个内核级的可编统计模块，这使其变成一个强大的分级引擎，而不只是个简单的包过滤器。应用程序可以构造这个模块来监听网络活动的任意方面（例如：网络负荷、两台主机间的流量、每秒 web 请求的次数等等），并在预定的时间间隔内接收内核传来的数据。

统计模式避免了复制数据包并且执行 0-copy 机制（当包仍存放在 NIC（网络接口卡）驱动的内存中时开始进行统计，随后丢弃这个包）。而且，环境转换的次数可以保持最低，这是因为结果通过一次系统调用就可以返回给用户。它不需要缓冲区（内核或用户），因此当监听开始时不用为它分配内存。可见，统计模式是一种很有效的网络监听方式，在高速网络中利用 libpcap 来工作也没任何问题。

WinPcap 为程序员提供了一套系统调用和高层函数来进行网络监听，这使得已经知道 libpcap API 的程序员能很容易使用。

5）构造数据包

BPF 和 NPF 都提供了构造包的函数，使用户可以将原始数据包发送到网络中。在 Windows 环境下，WinPcap 就成为首选的构造数据包的函数库，它提供了一套标准稳定的函数。另外，NPF 增加了一些新的函数，这些函数可以使数据包通过一次用户和内核模式之间的转换就发送几次。数据复制到内核中，然后通过调用一次 NDIS 将包发送到网络中。尽管 WinPcap 提供了一套新的函数来开发这些特性，但它没有提供那些强大的创建数据包的抽象函数，这需要通过其它现有的工具来实现。程序员可以利用著名的 Libnet Packet Assembly Library 的 Windows 版本实现，这个函数库增加了数据包结构层并在 WinPcap 上构造数据包。

3.2.4 WinPcap 常用函数说明

WinpCap 有丰富的数据结构和各种方便实用的函数，下面就 WinpCap 的几个常用函数介绍:

```
int pcap_findalldevs_ex(char *source,
struct pcap_rmtauth *auth,
pcap_if_t **alldevs,
char * errbuf)
```

该函数返回一个整型数据如果返回 0，则表示正常，返回-1 表示失败，并且错误提示在 errbuf 内，sourc 表示目的主机，auth 存放着连接远程主机需要的信息，alldevs 返回一个指向所有设备的第一设备的指针。

```
            char *    errbuf)
```

该函数返回一个特定的接口，source 指明目标主机，snaplen 表 要保留的数据包的长度，flags 表示所设置的标志，read_timeou,表示超时时间，auth 是目标主机所必要的信息，errbuf 存放错误提示信息。

```
            char *    errbuf)
```

该函数用来收集捕获到的数据，p 表示一个网络接口，从这接收数据，cnt 表示捕获的数据包的个数，如果是-1 则一直捕获，callback 是一个回调函数，用来实现数据包的循环捕捉，在里面可以实现一些数据包的分析。

```
int pcap_next_ex(pcap_t * p,
    struct   pcap_pkthdr   **   pkt_header,
    const u_char ** pkt_data)
```

该函数不用回调的方式来实现数据包的捕获，需要将它加入一个循环，p 表示获取数据的一接口，pkt_header 返回时间信息的一个头指针，该数据结构包含了时间戳和包的大小，pkt_data 返回一个指向获取的数据包的首地址。

```
int pcap_compile(pcap_t * p,
    struct bpf_program * fp,
    char *  str,
    int optimize,
    bpf_u_int32 netmask)
```

该函数用来过滤数据包的函数之一，p 指向一个接口地址，fp 是一个指向 bpf_program 的指针，会在 pcap_compile 中赋值，str 表示所要过滤的过滤规则，optimize

是控制结果代码的优化，netmask 表示子网掩码。

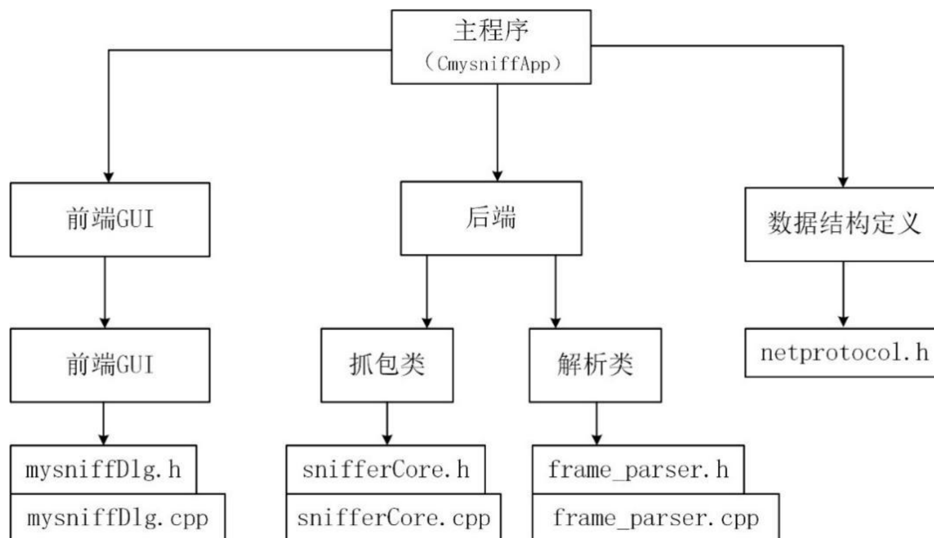
```
int pcap_setfilter(pcap_t * p,
        struct bpf_program * fp     )
```

该函数设置过滤规则，p 表示一个网络接口，此函数和上面组合使用来获取用户需要的数据。

WinPcap 是 Politecnico di Torino 的 NetGroup 小组开发的基于 Win32 平台的数据包捕获和网络分析的基础构架, 从 UNIX 下的 Libpcap 库移植而来, 用于用户层次的数据包捕获工作，由一系列开源的网络数据包捕获函数构成。它为底层网络监控编程提供了一个易于移植的应用框架。

3.3 程序分析

本课程设计采用 Visual Studio 2022 C++，基于应用程序 Winpcap 来实现数据包的捕获与分析。界面采用 MFC 实现一个单文档的程序，由菜单项中的按钮触发操作，同时改进了程序自带的保存、另存为等图标，成功加上了自己的图标，并与按钮 ID 相匹配。



这个程序基本实现了预期功能，下面是程序开发的过程。

3.4 开发环境准备

3.4.1 硬件与软件要求

硬件要求：AMD R9 7945 处理器、64M 以上内存、4M 可用硬盘空间（满足 WinPcap 驱动与程序运行）；

软件要求：Windows 11 操作系统、Visual Studio 2022 开发工具、WinPcap 4.1.3 驱动及开发包（包含头文件与库文件）。

3.4.2 MFC 简介

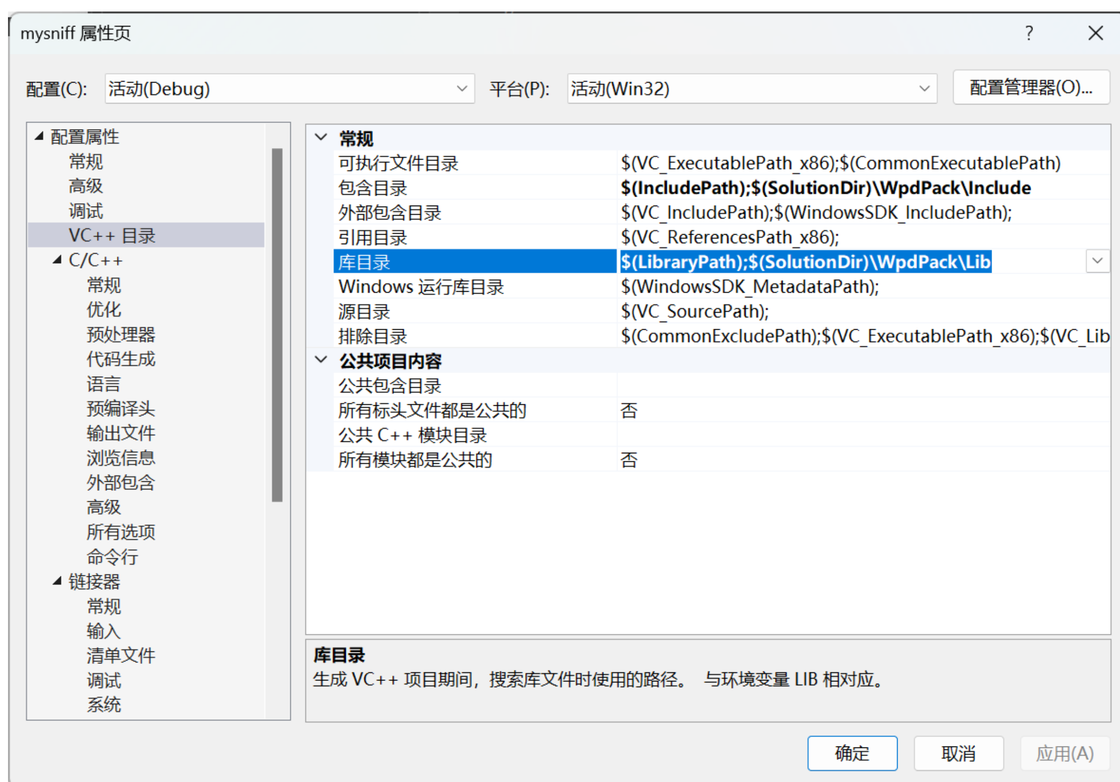
MFC： MFC(Microsoft Foundation Classes)，是微软公司提供的一个类库（ class libraries），以 C++类的形式封装了 Windows 的 API，并且包含一个应用程序框架，以减少应用程序开发人员的工作量。其中包含的类包含大量 Windows 句柄封装类和很多 Windows 的内建控件和组件的封装类。

MFC 其实就是一个在 Visual C++环境下编写程序的一个框架或者说是一个引擎，是微软公司提供的一个类，它包含一个应用程序的框架，windows 的 API 它也已经封装好，它的封装是以 C++类的形式来封装的，并且 Vc++本身的很多语言扩展和特征都是为了 MFC 设计或者量身打造的，这样就在一定程度上应用程序开发人员的工作量就会减少。本设计中用到的主要控件有 Pitcure 控件，用于用户显示图片，图片格式必须 BMP 文件； Edit box 控件，是一个最常用的控件，是用来接收用户输入的信息的；Static Text 控件，是一个文本控件，它的功能相对来说是比较简单的，是静态的，是能够用来作为位图、显示字符串、图标用的；Button 控件，用于按钮；Group Box 组框控件，Combo Box 组合框控件，是有一个输入框和一个列表框组成。

3.4.3 工程创建与配置

（1）新建 MFC 项目：在 Visual Studio 2022 下创建一个单文档的 MFC 应用程序，工程名：mysniffe

（2）项目属性配置：添加包含目录、库目录，并且链接库文件。以管理员权限运行 Visual Studio，确保 WinPcap 驱动正常加载。



（3）界面设计：双击资源视图中的 IDD_PACKETANALYZER_DIALOG（主对话框），进入设计模式。右侧“工具箱”会显示可用控件，

a) 添加下拉列表，点击“工具箱”，选择“combo box”，在“底板”上拖动，信息安全综合实验 4 增添一个合适长宽的控件，然后右键点击此控件，选择“属性”，在“外观-Type” 栏中选择“DropList”，再复制粘贴一个此控件。

b) 添加四个按钮：开始、结束、保存、读取。点击“工具箱”中的“button”，在“底板”增添一个适当大小的的按钮，点击此按钮，把“外观-Caption”修改为“开 始”，这样，按钮上的文字就被设置为“开始”了，其他三个按钮方法相同。

c) 添加列表，用来显示接受到的包信息，点击“工具箱”，选择“List Control”，添加一个适当大小的列表框控件，设置其属性，“外观-View”项设置为“Report”，“外-single selection”设置为“true”。

d) 添加树形列表，显示每一个包的具体字段信息，点击“工具箱”，选“Tree Control”，添加一个合适大小树形列表控件，将其“属性”中的“外观”下面“Full Row Select”、“Has Buttons”、“Info Tip”、“Lines At Root”均设置“True”。

e) 添加文本框，显示数据包的二进制信息，添加一个合适大小的“EditControl”，将“Multiline”、“Readonly”、“Auto Vscroll”、“Horizon Scroll”、“Vertical Scroll”均设置为“True”。

f) 再添加若干适当大“Edit Control”及“Static Text”，用于显示，并添加一 个“Group Box”它们包围起来。

至此为止，所有的 GUI 算是画完了，后续数据及属性的设置需要通过后台对相应的控件来完成，需为控件绑定变量：右键控件→添加变量，为所有需要操作的控件添加变量。

系统界面采用分区布局，包含四大功能区域：

- 数据包列表区：使用 CListCtrl 控件，以报表形式（LVS_REPORT）显示数据包摘要（编号、时间、协议、IP 等）；
- 协议详情区：使用 CTreeCtrl 控件，树形展示选中数据包的各层协议头字段；
- 十六进制显示区：使用 CEdit 控件，以十六进制格式展示数据包原始字节流；
- 统计区：使用 CStatic 控件，实时显示各协议数据包数量（TCP、HTTP、ARP 等）。

3.5 核心模块设计与实现

3.5.1 数据包捕获模块

（1）本地网卡枚举与选择，通过 pcap_findalldevs 函数获取本地网卡列表。通过链表遍历获取网卡名称与描述，使用 m_adapterName2dev 映射表存储适配器信息，便于后续选择。

```
int SnifferCore::snif_initCap() {
    // 用来记录找到的网卡设备
    int devCount = 0;
    char * errbuff = new char[PCAP_ERRBUF_SIZE];
    if (pcap_findalldevs(&m_alldevs, errbuff) == -1)
    {
        int x = MessageBox(GetForegroundWindow(), _T("没有找到网卡设备，请确认
Npcap-1.60 驱动已经安装！"), _T("错误"), 2);
        if (x == 3) { // 终止
            PostMessage(NULL, WM_QUIT, 0, 0);
        }
        if (x == 4) { // 重试
```

```
            while (1) {
                x = MessageBox(GetForegroundWindow(), _T("没有找到网卡设备，请
确认 Npcap-1.60 驱动已经安装！"), _T("错误"), 2);
                if (x != 4) break;
            }
            if (x == 3) {
                PostMessage(NULL, WM_QUIT, 0, 0);
            }
        }
    }

    for (auto dev = m_alldevs; dev; dev = dev->next) {
        printf("\n%d : 网卡名称: %s\n", ++devCount, dev->name);
        if (dev->description)
            printf("\t 描述 (%s) \n", dev->description);

        m_adapterName2dev[std::string(dev->description)] = dev;
    }

    if (devCount > 0) return devCount;
    return -1; // No adapter found
}
```

（2）网卡打开与数据包捕获，使用 pcap_open_live 打开网卡，设置混杂模式（PCAP_OPENFLAG_PROMISCUOUS）以捕获所有流经数据包：

```
int SnifferCore::snif_startCap() {
    pcap_if_t* curAdapter = getChoosedIf();
    if (!curAdapter) {
        MessageBox(GetForegroundWindow(), _T("请选择要监听的网卡"), _T("提示
"), MB_ICONWARNING);
        return -1;
    }
    // 打开网卡，设置捕获长度 65536 字节、超时 1000ms
    m_opened_if_handle = pcap_open_live(
        curAdapter->name, 65536, PCAP_OPENFLAG_PROMISCUOUS, 1000, errbuff
    );
    if (!m_opened_if_handle) {
        // 错误处理：显示网卡打开失败原因
        MessageBox(GetForegroundWindow(),
            _T("无法打开接口：") + CString(curAdapter->description) +
            _T("\n 错误详情：") + CString(errbuff),
            _T("错误"), MB_ICONERROR);
```

```
        return -1;
    }
    // 设置过滤规则（见 3.3.3 节）
    if (snif_setupFilter() != 0) return -1;
    // 启动抓包线程
    return m_snif_CreateCapThread();
}
```

（3）多线程抓包实现，通过 CreateThread 创建独立抓包线程，在回调函数中处理数据包：

```
int SnifferCore::m_snif_CreateCapThread() {
    // 关闭旧线程（如有）
    if (m_threadHandle) CloseHandle(m_threadHandle);
    // 创建新线程，绑定回调函数 m_snif_CapThreadFun
    m_threadHandle = CreateThread(
        NULL, 0, (LPTHREAD_START_ROUTINE)m_snif_CapThreadFun, this, 0, NULL
    );
    if (!m_threadHandle) {
        int errCode = GetLastError();
        CString errMsg;
        errMsg.Format(_T("创建抓包线程失败，错误码：%d"), errCode);
        MessageBox(GetForegroundWindow(), errMsg, _T("错误"), MB_ICONERROR);
        return -1;
    }
    return 0;
}

// 线程回调函数：实时捕获数据包并传递给解析模块
DWORD WINAPI SnifferCore::m_snif_CapThreadFun(LPVOID lpParam) {
    SnifferCore* pThis = (SnifferCore*)lpParam;
    pcap_t* handle = pThis->getOpenedIfHandle();
    struct pcap_pkthdr* header;
    const u_char* packet;
    // 循环捕获数据包，-1 表示无限循环
    while (pcap_next_ex(handle, &header, &packet) >= 0) {
        if (packet) {
            // 传递数据包给解析模块
            pThis->data_parser.set(packet, header);
            pThis->data_parser.parse();
            // 通知 UI 更新显示
            pThis->tellGuiToUpdate(
                &pThis->data_parser.getStatistics(),
```

```
                &pThis->data_parser.getParsedHeaderPack()
            );
        }
    }
    return 0;
}
```

3.5.2 协议解析模块

协议解析遵循 TCP/IP 协议栈分层结构，从链路层到应用层逐级解析：

- 链路层解析：提取 MAC 地址与上层协议类型（如 IPv4、ARP）；
- 网络层解析：解析 IP 头部（版本、TTL、协议字段）或 ARP 头部（硬件类型、操作码）；
- 传输层解析：根据网络层协议类型（TCP/UDP/ICMP）解析端口、序列号等；
- 应用层解析：识别 HTTP、SMTP 等协议（基于端口或数据特征）。

```
// 数据解析类核心方法：入口函数
int DataParser::parse() {
    // 保存原始数据包头
    m_hdr_pack.pcaph = shallowCopy(const_cast<PKTHDR*>(m_pacp_header));
    // 解析以太网帧头部
    eth_header* ethdr = (eth_header*)m_pkt_data;
    u_short protocol = ntohs(ethdr->proto);
    m_pkt_counter.n_sum++;
    m_hdr_pack.ethh = shallowCopy(ethdr);
    // 根据协议类型调用下层解析函数
    switch (protocol) {
        case  ETH_PROTOCOL_IP:        parse_ip((u_char*)ethdr  +  ETH_HEAD_LEN);
break;
        case  ETH_PROTOCOL_ARP:      parse_arp((u_char*)ethdr  +  ETH_HEAD_LEN);
break;
        case  ETH_PROTOCOL_IPV6:    parse_ipv6((u_char*)ethdr  +  ETH_HEAD_LEN);
break;
        default: /* 未支持协议 */ break;
    }
    // 保存解析结果
    m_idx2data.push_back(pair<pktCount,              headerPack>(m_pkt_counter,
m_hdr_pack));
    return 0;
}
```

```
// IP 协议解析（以 IPv4 为例）
int DataParser::parse_ip(const u_char* pkt) {
    ip_header* iphdr = (ip_header*)pkt;
    int ip_head_len = (iphdr->ver_ihl & 0xF) * 4;  // IP 头部长度（字节）
    u_char protocol = iphdr->proto;
    m_hdr_pack.iph = shallowCopy(iphdr);
    m_pkt_counter.n_ip++;
    // 根据传输层协议类型继续解析
    switch (protocol) {
        case IP_PROTOCOL_TCP:  return parse_tcp(pkt + ip_head_len);
        case IP_PROTOCOL_UDP:  return parse_udp(pkt + ip_head_len);
        case IP_PROTOCOL_ICMP: return parse_icmp(pkt + ip_head_len);
        default: return -1;
    }
    return 0;
}

// TCP 协议解析（含 HTTP 协议识别）
int DataParser::parse_tcp(const u_char* pkt) {
    tcp_header* tcphdr = (tcp_header*)pkt;
    m_hdr_pack.tcph = shallowCopy(tcphdr);
    m_pkt_counter.n_tcp++;
    // 识别 HTTP 协议（端口 80）
    if (ntohs(tcphdr->dport) == 80 || ntohs(tcphdr->sport) == 80) {
        m_pkt_counter.n_http++;
        strcpy(m_hdr_pack.pktType, PKTTYPE_HTTP);
    } else {
        strcpy(m_hdr_pack.pktType, PKTTYPE_TCP);
    }
    return 1;
}
```

3.5.3 数据过滤模块

使用 pcap_compile 与 pcap_setfilter 实现数据包过滤，支持协议、IP、端口等条件：

```
int SnifferCore::snif_setupFilter() {
    struct bpf_program fcode;
    pcap_if_t* curIf = getChoosedIf();
    int netmask = 0xffffff;  // 默认掩码
    // 获取网卡子网掩码
    if (curIf->addresses && curIf->addresses->netmask) {
        netmask                              =                              ((struct
```

```
sockaddr_in*)curIf->addresses->netmask)->sin_addr.S_un.S_addr;
    }
    // 编译过滤规则（如"tcp and port 80"表示过滤 TCP 协议且端口 80 的包）
    if (pcap_compile(getOpenedIfHandle(), &fcode,
        const_cast<char*>(getChoosedRule().c_str()), 1, netmask) < 0) {
        MessageBox(GetForegroundWindow(), _T("过滤规则语法错误，请检查！"), _T("
错误"), MB_ICONERROR);
        return -1;
    }
    // 设置过滤规则
    if (pcap_setfilter(getOpenedIfHandle(), &fcode) < 0) {
        MessageBox(GetForegroundWindow(), _T("设置过滤规则失败"), _T("提示"),
MB_ICONWARNING);
        return -1;
    }
    return 0;
}
```

3.5.4 用户交互模块

列表控件更新逻辑如下：

```
void CmysniffDlg::update_listCtrl(const pktCount* npkt, const datapkt*
hdrs_pack) {
    CString num, ts, len, s_mac, d_mac, proto, s_ip, d_ip;
    // 格式化数据包基本信息
    num.Format(_T("%d"), m_snifferCore.getnpkt());
    len.Format(_T("%d"), hdrs_pack->pcaph->len);
    // 解析 MAC 地址
    u_char* mac_arr = hdrs_pack->ethh->s_mac;
    s_mac.Format(_T("%02x:%02x:%02x:%02x:%02x:%02x"),
        mac_arr[0], mac_arr[1], mac_arr[2], mac_arr[3], mac_arr[4],
mac_arr[5]);
    // 解析时间戳
    time_t t = hdrs_pack->pcaph->ts.tv_sec;
    struct tm* ltime = localtime(&t);
    char timeBuf[32] = {0};
    strftime(timeBuf, sizeof(timeBuf), "%Y/%m/%d %H:%M:%S", ltime);
    ts = CString(timeBuf);
    // 解析 IP 地址（根据协议类型）
    u_short code = ntohs(hdrs_pack->ethh->proto);
    if (code == ETH_PROTOCOL_IP) {
        struct in_addr ip;
        ip.S_un.S_addr = *((u_long*)(void*)(&hdrs_pack->iph->saddr));
```

```
        s_ip = CString(inet_ntoa(ip));
        ip.S_un.S_addr = *((u_long*)(void*)(&hdrs_pack->iph->daddr));
        d_ip = CString(inet_ntoa(ip));
    } else if (code == ETH_PROTOCOL_ARP) {
        s_ip.Format(_T("%d.%d.%d.%d"),
            hdrs_pack->arph->saddr.byte1, hdrs_pack->arph->saddr.byte2,
            hdrs_pack->arph->saddr.byte3, hdrs_pack->arph->saddr.byte4);
        d_ip.Format(_T("%d.%d.%d.%d"),
            hdrs_pack->arph->daddr.byte1, hdrs_pack->arph->daddr.byte2,
            hdrs_pack->arph->daddr.byte3, hdrs_pack->arph->daddr.byte4);
    }
    // 更新列表控件
    int nitem = m_listCtrl.InsertItem(m_snifferCore.getnpkt(), num);
    m_listCtrl.SetItemText(nitem, 1, ts);
    m_listCtrl.SetItemText(nitem, 2, len);
    m_listCtrl.SetItemText(nitem, 3, s_mac);
    m_listCtrl.SetItemText(nitem, 4, d_mac);
    m_listCtrl.SetItemText(nitem, 5, hdrs_pack->pktType);
    m_listCtrl.SetItemText(nitem, 6, s_ip);
    m_listCtrl.SetItemText(nitem, 7, d_ip);
    // 滚动到最新项
    m_listCtrl.EnsureVisible(nitem, FALSE);
}
```

3.5.5 数据库管理模块

savePacketBase 函数是 DatabaseManager 类的核心方法，负责将捕获的网络数据包基础信息存储到 SQLite 数据库的 packet_base 表中。该函数处理从数据包解析到 SQL 参数绑定的完整流程，支持多种网络协议的信息提取和存储。

```
bool DatabaseManager::savePacketBase(const struct pcap_pkthdr* header, const struct datapkt* data) {
    sqlite3_stmt* stmt;
    const char* sql = "INSERT INTO packet_base (timestamp, length, src_mac, dst_mac, eth_protocol, "
        "src_ip, dst_ip, ip_protocol, transport_protocol, src_port, dst_port, app_protocol) "
        "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);";

    if (sqlite3_prepare_v2(m_db, sql, -1, &stmt, nullptr) != SQLITE_OK) {
        const char* err = sqlite3_errmsg(m_db);
```

```
            OutputDebugStringA(("SQL        prepare        error:        "        +
std::string(err)).c_str());
            return false;
        }

        // 时间戳转换 - 修复时间类型转换问题
        char timestamp[64];
        time_t timestamp_sec = static_cast<time_t>(header->ts.tv_sec);
        struct tm* ltime = localtime(&timestamp_sec);
        strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", ltime);

        // MAC 地址格式转换
        char src_mac[18], dst_mac[18];
        snprintf(src_mac, sizeof(src_mac), "%02x:%02x:%02x:%02x:%02x:%02x",
            data->ethh->s_mac[0],                        data->ethh->s_mac[1],
data->ethh->s_mac[2],
            data->ethh->s_mac[3],                        data->ethh->s_mac[4],
data->ethh->s_mac[5]);

        snprintf(dst_mac, sizeof(dst_mac), "%02x:%02x:%02x:%02x:%02x:%02x",
            data->ethh->d_mac[0],                        data->ethh->d_mac[1],
data->ethh->d_mac[2],
            data->ethh->d_mac[3],                        data->ethh->d_mac[4],
data->ethh->d_mac[5]);

        // 绑定参数
        sqlite3_bind_text(stmt, 1, timestamp, -1, SQLITE_STATIC);
        sqlite3_bind_int(stmt, 2, header->len);
        sqlite3_bind_text(stmt, 3, src_mac, -1, SQLITE_STATIC);
        sqlite3_bind_text(stmt, 4, dst_mac, -1, SQLITE_STATIC);
        sqlite3_bind_text(stmt, 5, data->pktType, -1, SQLITE_STATIC);

        // IP 相关信息 - 修复 IP 地址转换问题
        if (ntohs(data->ethh->proto) == ETH_PROTOCOL_IP && data->iph) {
            struct in_addr src_ip, dst_ip;

            // 使用内存复制解决类型转换问题
            memcpy(&src_ip.s_addr, &data->iph->saddr, sizeof(u_long));
            memcpy(&dst_ip.s_addr, &data->iph->daddr, sizeof(u_long));

            // 绑定 IP 地址
            sqlite3_bind_text(stmt, 6, inet_ntoa(src_ip), -1, SQLITE_STATIC);
            sqlite3_bind_text(stmt, 7, inet_ntoa(dst_ip), -1, SQLITE_STATIC);
```

```
        // 获取 IP 协议名称
        auto it = IP_PROT2STR.find(data->iph->proto);
        if (it != IP_PROT2STR.end()) {
            sqlite3_bind_text(stmt, 8, it->second.c_str(), -1, SQLITE_STATIC);
        }
        else {
            sqlite3_bind_text(stmt, 8, "Unknown", -1, SQLITE_STATIC);
        }

        // 传输层信息
        if (data->iph->proto == IP_PROTOCOL_TCP && data->tcph) {
            sqlite3_bind_text(stmt, 9, "TCP", -1, SQLITE_STATIC);
            sqlite3_bind_int(stmt, 10, ntohs(data->tcph->sport));
            sqlite3_bind_int(stmt, 11, ntohs(data->tcph->dport));

            // HTTP 检测
            if (ntohs(data->tcph->dport) == 80 || ntohs(data->tcph->sport) == 80) {
                sqlite3_bind_text(stmt, 12, "HTTP", -1, SQLITE_STATIC);
            }
            else {
                sqlite3_bind_null(stmt, 12);
            }
        }
        else if (data->iph->proto == IP_PROTOCOL_UDP && data->udph) {
            sqlite3_bind_text(stmt, 9, "UDP", -1, SQLITE_STATIC);
            sqlite3_bind_int(stmt, 10, ntohs(data->udph->sport));
            sqlite3_bind_int(stmt, 11, ntohs(data->udph->dport));
            sqlite3_bind_null(stmt, 12);
        }
        else {
            sqlite3_bind_null(stmt, 9);
            sqlite3_bind_null(stmt, 10);
            sqlite3_bind_null(stmt, 11);
            sqlite3_bind_null(stmt, 12);
        }
    }
    else {
        // 其他协议处理
        sqlite3_bind_null(stmt, 6);
        sqlite3_bind_null(stmt, 7);
```

```
            sqlite3_bind_null(stmt, 8);
            sqlite3_bind_null(stmt, 9);
            sqlite3_bind_null(stmt, 10);
            sqlite3_bind_null(stmt, 11);
            sqlite3_bind_null(stmt, 12);
        }

        int result = sqlite3_step(stmt);
        sqlite3_finalize(stmt);
        // 添加验证代码
        if (result == SQLITE_DONE) {
            // 获取最后插入的行 ID
            sqlite3_int64 lastRowId = sqlite3_last_insert_rowid(m_db);
            TRACE("成功保存数据包到数据库，行 ID: %lld\n", lastRowId);

            // 验证数据是否实际写入
            sqlite3_stmt* checkStmt;
            const char* checkSQL = "SELECT COUNT(*) FROM packet_base WHERE id
    = ?;";
            if (sqlite3_prepare_v2(m_db, checkSQL, -1, &checkStmt, nullptr)
    == SQLITE_OK) {
                sqlite3_bind_int64(checkStmt, 1, lastRowId);
                if (sqlite3_step(checkStmt) == SQLITE_ROW) {
                    int count = sqlite3_column_int(checkStmt, 0);
                    TRACE("验证结果: %s\n", count > 0 ? "成功" : "失败");
                }
                sqlite3_finalize(checkStmt);
            }
            return true;
        }
        else {
            TRACE("保存数据包失败，错误代码: %d\n", result);
            return false;
        }
        return result == SQLITE_DONE;
    }
```

结果：

Sniff_UI

RZ616 Wi-Fi 6E 160MHz

输入过滤规则(默认不过滤)　...帮助　开始　结束

数据包列表

编号	时间	长度	源MAC地址	目标MAC地址	协议	源IP地址	目
1	2025/06/23 23:25:44	1414	6e:06:81:2e:c3:07	14:ac:60:b5:1f:bf	TCP	120.236.198.165	192.1
2	2025/06/23 23:25:44	235	6e:06:81:2e:c3:07	14:ac:60:b5:1f:bf	TCP	120.236.198.165	192.1
3	2025/06/23 23:25:44	54	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	TCP	192.168.164.179	120.2
4	2025/06/23 23:25:44	162	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	TCP	192.168.164.179	120.2
5	2025/06/23 23:25:44	180	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	TCP	192.168.164.179	120.2
6	2025/06/23 23:25:44	1414	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	TCP	192.168.164.179	120.2

行为分析

编号	行为类型	协议	时间	域名/应用	具体行为	源端口	目的...	源IP地址	目的IP地址
	DNS解析	UDP	2025/06/23 23:25:...	le3-api.game.bilibili.co...	正在观看B...	60124	53	179.164.168.192	114.114.114.114
	DNS解析	UDP	2025/06/23 23:25:...	le3-api.game.bilibili.co...	正在观看B...	60123	53	179.164.168.192	29.29.29.119
	DNS解析	UDP	2025/06/23 23:25:...	le3-api.game.bilibili.co...	正在观看B...	60126	53	179.164.168.192	5.5.5.223
	DNS解析	UDP	2025/06/23 23:25:...	le3-api.game.bilibili.co...	正在观看B...	60125	53	179.164.168.192	5.5.5.223
	DNS解析	UDP	2025/06/23 23:25:...	le3-api.game.bilibili.co...	正在观看B...	60127	53	179.164.168.192	29.29.29.119
	DNS解析	UDP	2025/06/23 23:25:...	le3-api.game.bilibili.co...	正在观看B...	60128	53	179.164.168.192	114.114.114.114
	DNS解析	UDP	2025/06/23 23:25:...	le3-api.game.bilibili.co...	正在观看B...	53	60126	5.5.5.223	179.164.168.192

接受到的第3个数据包
- 链路层数据
  - 源MAC: 14:ac:60:b5:1f:b
  - 目的MAC: 6e:06:81:2e:c
  - 类型码：0x0800
- IP协议头
  - 版本: 4
  - IP头部长度: 20(bytes)
  - 服务类型: 0
  - 总长度: 40
  - 标识: 58166
  - 段偏移: 0
  - 生存期: 128
  - 协议: 6

数据包分析

```
0000: 54 43 50 00 4f 57 4e 00 70 30 20 01 30 2e 20 01 TCP.OWN.p0 .0. .
0010: f0 32 20 01 30 33 20 01 00 00 00 00 00 00 00 00 .2 .03 ........
0020: 18 1a 21 01 f0 30 20 01 00 00 00 00 00 00 00 00 ..!..0 ........
0030: 00 00 00 00 00 00                               ......
```

16进制数据

统计数据

TCP包	1413	HTTP包	30	IPv4包	965
UDP包	0	ARP包	4	IPv6包	522
ICMP包	3	ICMPv6	5	其它	0
				总计	2965

协议类型数量统计

保存　读取

3.5.6 前端可视化模块

```
bool CmysniffDlg::FindPythonExecutable(CString& path) {
    // 尝试从注册表查找 Python 安装路径
    HKEY hKey;
    if (RegOpenKeyEx(HKEY_LOCAL_MACHINE,
        _T("SOFTWARE\\Python\\PythonCore\\3.11\\InstallPath"),
        0, KEY_READ, &hKey) == ERROR_SUCCESS) {

        TCHAR pythonPath[MAX_PATH];
        DWORD bufSize = MAX_PATH;
        if (RegQueryValueEx(hKey, _T(""), NULL, NULL,
            (LPBYTE)pythonPath, &bufSize) == ERROR_SUCCESS) {

            path.Format(_T("%s\\python.exe"), pythonPath);
            if (PathFileExists(path)) {
                RegCloseKey(hKey);
                return true;
            }
        }
        RegCloseKey(hKey);
    }

    // 尝试常见安装路径
    const CString commonPaths[] = {
        _T("D:\\python\\python.exe"),
        _T("C:\\Python310\\python.exe"),
        _T("C:\\Program Files\\Python311\\python.exe"),
        _T("C:\\Program Files\\Python310\\python.exe")
    };

    for (const auto& p : commonPaths) {
        if (PathFileExists(p)) {
            path = p;
            return true;
        }
    }

    return false;
}

void CmysniffDlg::OnBnClickedButton7()
```

```
{
    // 1. 直接指定虚拟环境的 Python 解释器路径（跳过查找逻辑）
    CString                          pythonExe                          =
_T("E:\\reverse\\python\\venv\\Scripts\\python.exe");

    // 2. 设置日志文件路径（调试用）
    CString logPath = _T("E:\\reverse\\python\\script_log.txt");

    // 3. 目标 Python 脚本路径
    CString pythonScriptPath = _T("E:\\reverse\\python\\2.py");

    // 4. 构建完整命令行（重定向输出到日志）
    CString commandLine;
    commandLine.Format(_T("\"%s\" \"%s\" > \"%s\" 2>&1"),
        pythonExe, pythonScriptPath, logPath);

    // 5. 使用 CreateProcess 启动
    STARTUPINFO si = { sizeof(STARTUPINFO) };
    PROCESS_INFORMATION pi;
    BOOL success = CreateProcess(
        NULL,                          // 必须设为 NULL 才能使用完整命令行
[4,6](@ref)
        commandLine.GetBuffer(), // 完整命令行
        NULL, NULL, FALSE,
        CREATE_NO_WINDOW,        // 隐藏控制台窗口
        NULL,
        _T("E:\\reverse\\python"), // 工作目录（脚本所在目录）
        &si, &pi
    );
    commandLine.ReleaseBuffer();

    // 6. 进程处理与结果反馈
    if (success)
    {
        CloseHandle(pi.hThread);
        CloseHandle(pi.hProcess);
        MessageBox(_T("Python 脚本已启动"), _T("成功"));
    }
    else
    {
        DWORD err = GetLastError();
        CString msg;
        msg.Format(_T("启动失败 (错误代码 %d)\n 检查日志: %s"), err,
```

```
        logPath);
                MessageBox(msg, _T("错误"));
            }
        }
```

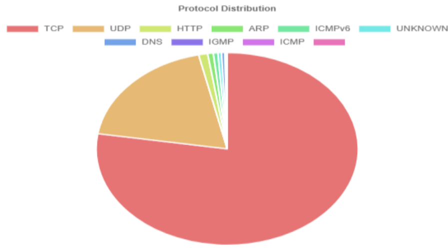
Network Activity Dashboard

All Packets HTTP Data Sensitive Info Table Stats **Chart Stats**

Chart Statistics

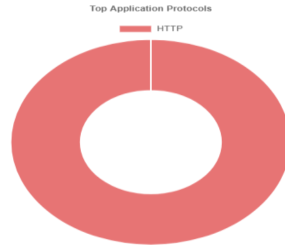
Protocol Distribution

Protocol Distribution



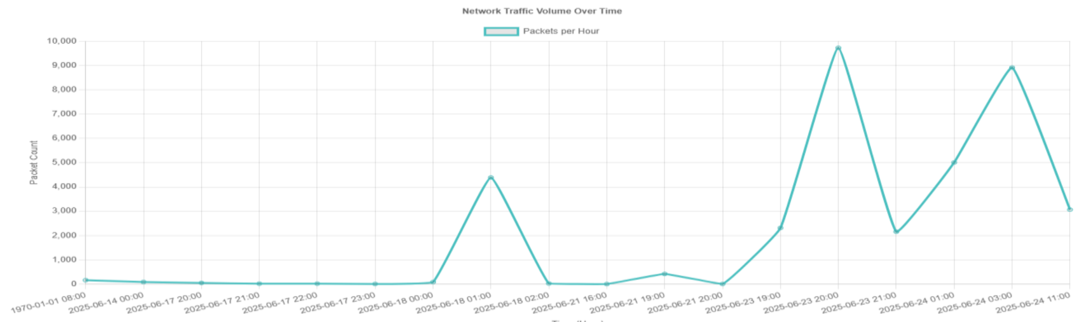
Application Protocol Distribution

Top Application Protocols



Traffic Over Time (Packets per Hour)

Network Traffic Volume Over Time

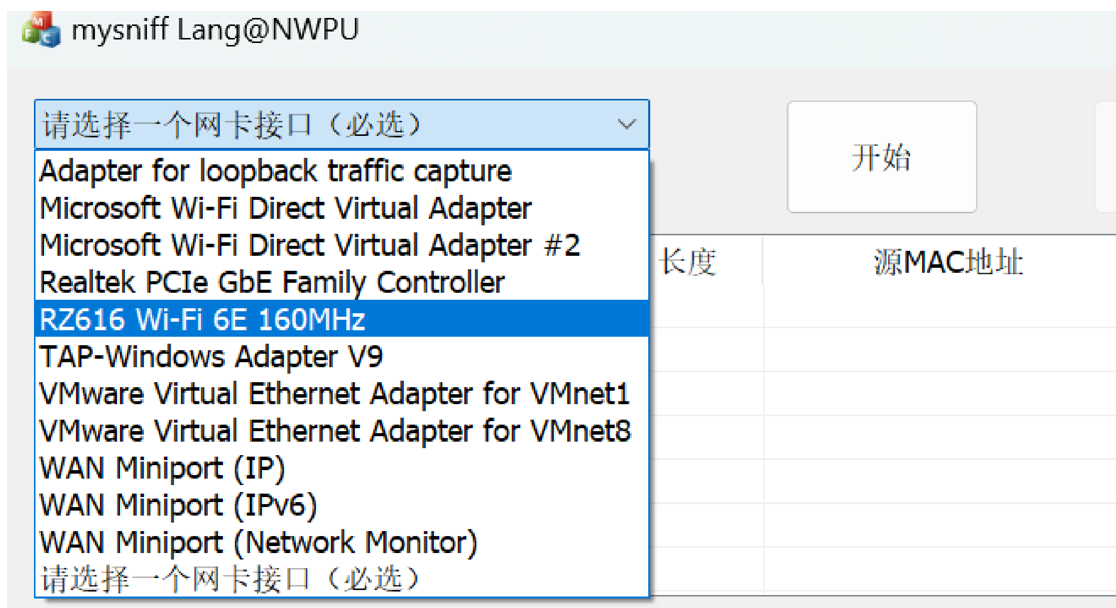


第 4 章 实现功能测试

4.1 功能测试

4.1.1 列出本机的网卡列表，选择要监听的网卡，实时捕获所有流经网卡的数据包

程序运行后，显示界面，首先先对网卡进行选择，才能截取相应的数据包。



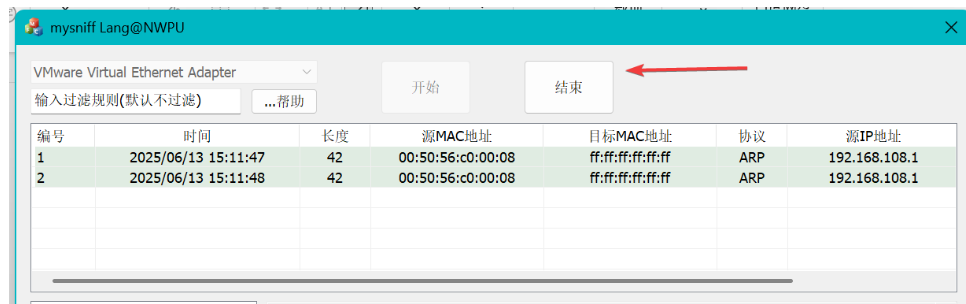
4.1.2 分析捕获到的数据包的包头和数据，按照各种协议的格式进行格式化显示

下面的一个列表控件，用来显示具体捕获的数据包的信息，包括协议名称，源 IP 地址，源 MAC 地址，目的 IP 地址，目的端口，以及该数据包的大小

编号	时间	长度	源MAC地址	目标MAC地址	协议	源IP地址
1	2025/06/13 14:51:04	42	00:50:56:c0:00:08	ff:ff:ff:ff:ff:ff	ARP	192.168.108.1
2	2025/06/13 14:51:05	42	00:50:56:c0:00:08	ff:ff:ff:ff:ff:ff	ARP	192.168.108.1
3	2025/06/13 14:51:06	42	00:50:56:c0:00:08	ff:ff:ff:ff:ff:ff	ARP	192.168.108.1
4	2025/06/13 14:51:34	42	00:50:56:c0:00:08	ff:ff:ff:ff:ff:ff	ARP	192.168.108.1
5	2025/06/13 14:51:35	42	00:50:56:c0:00:08	ff:ff:ff:ff:ff:ff	ARP	192.168.108.1
6	2025/06/13 14:51:36	42	00:50:56:c0:00:08	ff:ff:ff:ff:ff:ff	ARP	192.168.108.1

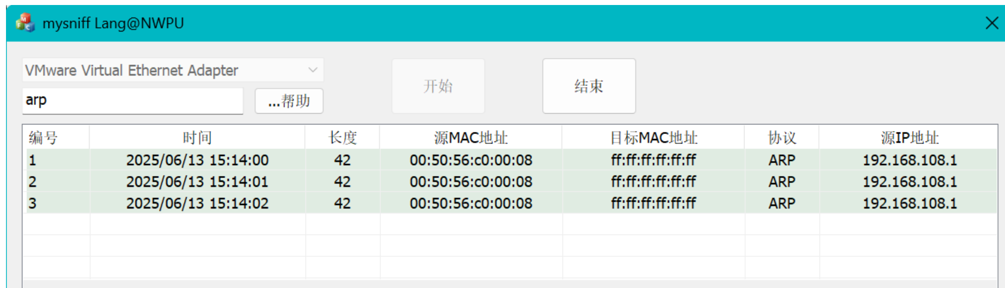
4.1.3 捕获过程中可以提前结束捕获

点击结束按钮，可以随时终止捕获数据。



4.1.4 输入过滤条件，对捕获到的数据进行筛选

输入不同的协议名称，可以对捕获到的数据进行筛选，能够尽快帮助用户找到自己想要的数据，本程序暂时只支持对协议类型进行筛选。



4.1.5 对数据包中各个协议数量进行统计

在捕获数据包的同时，在界面上实时显示各个协议数据包的统计数目以及数据包总数。

统计数据

TCP包	0	HTTP包	0	IPv4包	0
UDP包	0	ARP包	32	IPv6包	0
ICMP包	0	ICMPv6	0	其它	0
				总计	32

4.1.6 显示被选中的数据包头详细信息

随机点击显示列表中的某一行数据，单独显示被选中的数据包头详细信息。

Arp 报文：

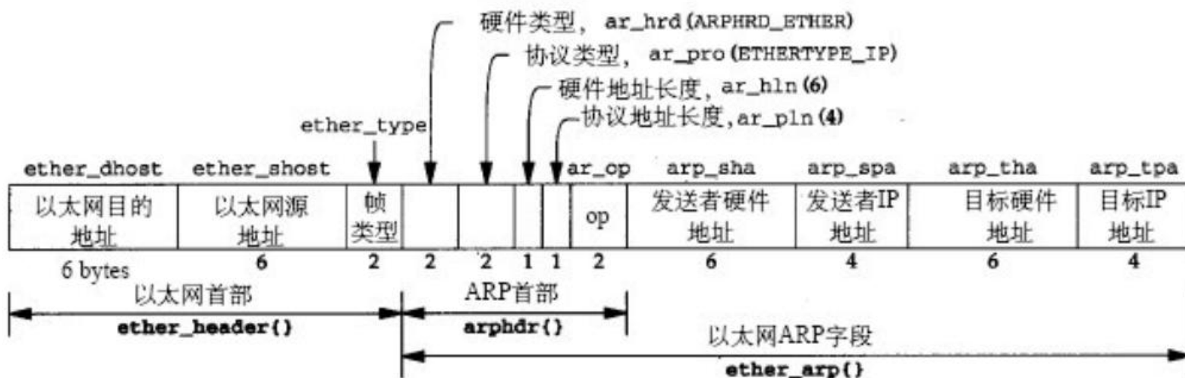
接受到的第5个数据包
⊟ 链路层数据
源MAC: 00:50:56:c0:00:08
目的MAC: ff:ff:ff:ff:ff:ff
类型码：0x0806
⊟ ARP协议头
硬件类型: 8573
协议类型: 0x217d
硬件地址长度: 2013798781
协议地址长度: 2013798781
操作码: 0x217d
发送方MAC: 00:50:56:c0:00:0
发送方ip: 192.168.108.1
接受方MAC: 00:00:00:00:00:0
接收方ip: 192.168.108.2

0000: 41 52 50 00 00 00 00 00 d8 e3 94 00 58 e1 94 00 ARP.........X...
0010: 98 de 94 00 98 e1 94 00 b0 c9 90 00 58 e5 94 00 ....X...
0020: 00 00 00 00 00 00 00 00 00 00 ....

统计数据

TCP包	0	HTTP包	0	IPv4包	0
UDP包	0	ARP包	32	IPv6包	0
ICMP包	0	ICMPv6	0	其它	0
				总计	32

保存 读取



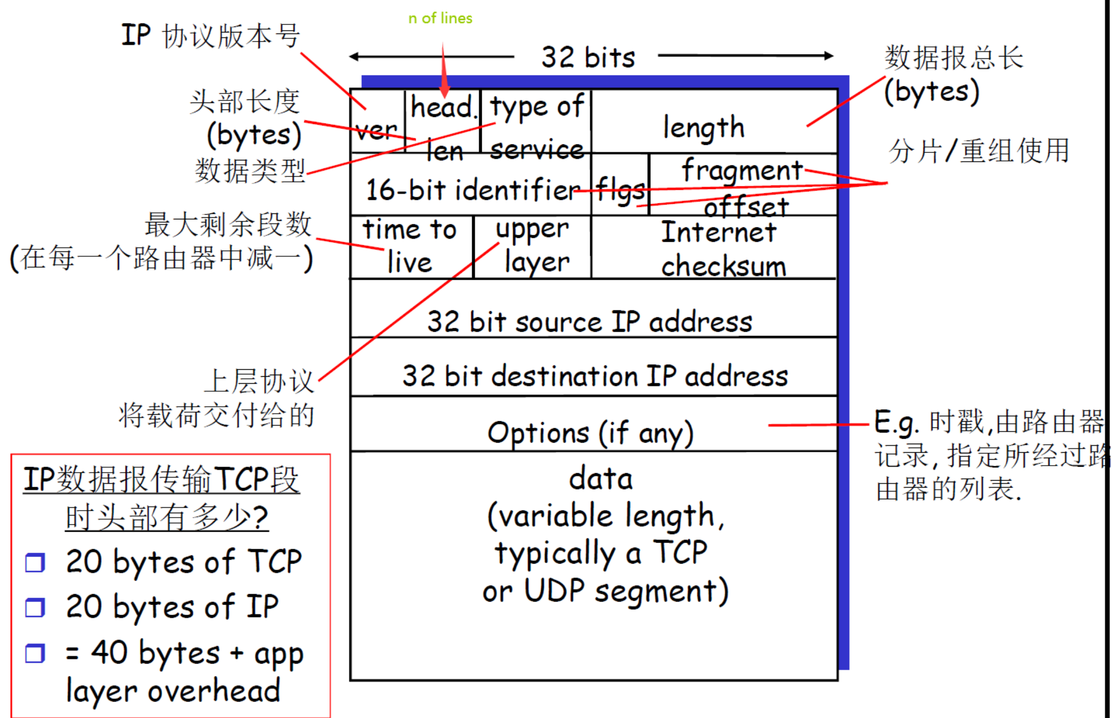
IP 报文：

⊟ 接受到的第1个数据包
⊟ 链路层数据
源MAC: 58:69:6c:5f:b2:3
目的MAC: 14:ac:60:b5:1
类型码：0x0800
⊟ IP协议头
版本: 4
IP头部长度: 20(bytes)
服务类型: 4
总长度: 101
标识: 30325
段偏移: 0
生存期: 50
协议: 6
头部校验和: 24605

0000: 54 43 50 00 00 00 00 00 a8 bc 90 00 28 be 90 00 TCP.........(...
0010: 28 bd 90 00 68 ba 90 00 00 00 00 00 00 00 00 00 (...h...........
0020: 80 3f 8e 00 e8 ba 90 00 00 00 00 00 00 00 00 00 .?..............
0030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0040: 00 00 00 00 00 00 00 00 c8 3f 8e 00 68 bb 90 00 ....?..h...
0050: 00 00 00 00 00 00 00 00 cc cc cc cc cc cc cc cc ....

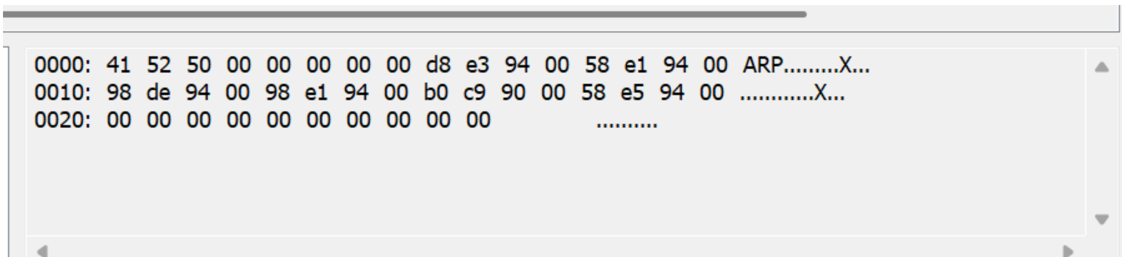
统计数据

TCP包	69	HTTP包	8	IPv4包	69
UDP包	0	ARP包	0	IPv6包	0
ICMP包	0	ICMPv6	0	其它	0
				总计	69

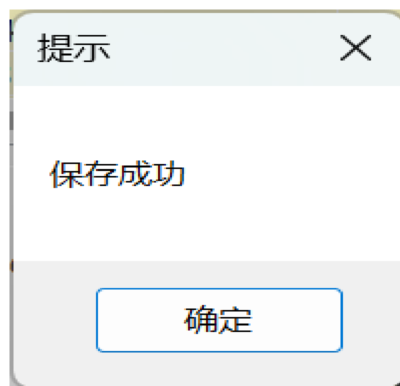


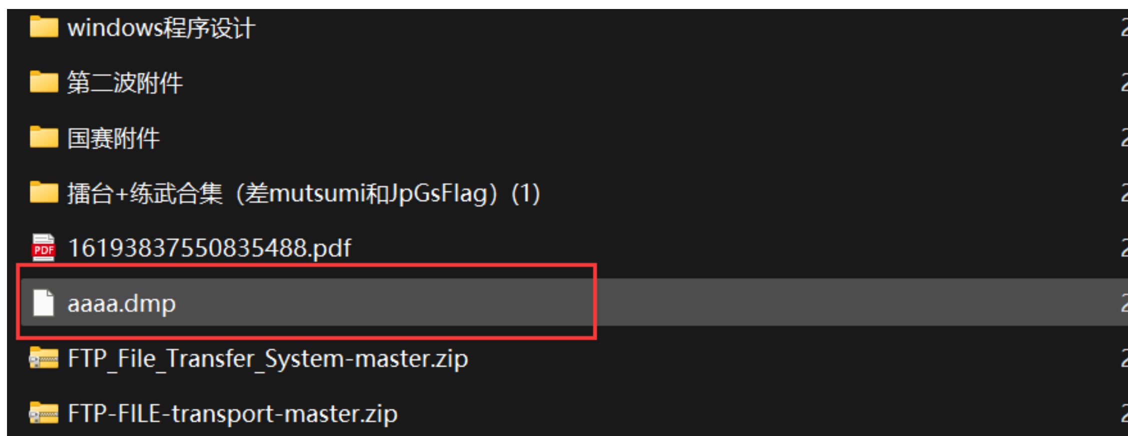
4.1.7 以十六进制显示被选中的数据包内容

随机点击显示列表中的某一行数据，以十六进制单独显示被选中的数据包内容。



4.1.8 以本地文件形式保存和读取已捕获的数据包信息





4.1.9 用户行为监控

Sniff_UI

RZ616 Wi-Fi 6E 160MHz

输入过滤规则(默认不过滤) ...帮助

开始 结束 看微信聊天记

编号	时间	长度	源MAC地址	目标MAC地址	协议	源IP地址	目
1	2025/06/24 03:14:34	54	6e:06:81:2e:c3:07	14:ac:60:b5:1f:bf	TCP	120.236.198.165	192.1
2	2025/06/24 03:14:34	54	6e:06:81:2e:c3:07	14:ac:60:b5:1f:bf	TCP	120.236.198.165	192.1
3	2025/06/24 03:14:34	54	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	TCP	192.168.164.179	120.2
4	2025/06/24 03:14:35	55	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	TCP	192.168.164.179	36.
5	2025/06/24 03:14:35	66	6e:06:81:2e:c3:07	14:ac:60:b5:1f:bf	TCP	36.147.59.95	192.1
6	2025/06/24 03:14:35	74	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	HTTP	dd64806d:dd64806d:dd...	dd64806c

编号	行为类型	协议	时间	域名/应用	具体行为	源端口	目的...	源IP地址	目的IP地址
	DNS解析	UDP	2025/06/24 03:14:...	www.baidu.com	访问百度搜索	57190	53	179.164.168.192	114.114.114.114
	DNS解析	UDP	2025/06/24 03:14:...	www.baidu.com	访问百度搜索	57191	53	179.164.168.192	29.29.29.119
	DNS解析	UDP	2025/06/24 03:14:...	www.baidu.com	访问百度搜索	57193	53	179.164.168.192	29.29.29.119
	DNS解析	UDP	2025/06/24 03:14:...	www.baidu.com	访问百度搜索	57192	53	179.164.168.192	5.5.5.223
	DNS解析	UDP	2025/06/24 03:14:...	www.baidu.com	访问百度搜索	57194	53	179.164.168.192	114.114.114.114
	DNS解析	UDP	2025/06/24 03:14:...	www.baidu.com	访问百度搜索	57195	53	179.164.168.192	5.5.5.223
	DNS解析	UDP	2025/06/24 03:14:...	www.baidu.com	访问百度搜索	57195	53	179.164.168.192	165.198.236.120

统计数据

TCP包	675	HTTP包	1	IPv4包	698
UDP包	0	ARP包	0	IPv6包	49
ICMP包	1	ICMPv6	4	其它	0
				总计	1495

保存 读取

Sniff_UI

RZ616 Wi-Fi 6E 160MHz

输入过滤规则(默认不过滤)　...帮助　开始　结束　看微信聊天记

编号	时间	长度	源MAC地址	目标MAC地址	协议	源IP地址	目
1	2025/06/24 03:13:05	75	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	TCP	5a79240b:5a79240b:5a...	5a79240b
2	2025/06/24 03:13:05	529	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	TCP	192.168.164.179	112.
3	2025/06/24 03:13:05	185	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	TCP	192.168.164.179	221.
4	2025/06/24 03:13:05	1412	14:ac:60:b5:1f:bf	6e:06:81:2e:c3:07	TCP	192.168.164.179	221.
5	2025/06/24 03:13:05	89	6e:06:81:2e:c3:07	14:ac:60:b5:1f:bf	TCP	112.45.122.108	192.1
6	2025/06/24 03:13:05	74	6e:06:81:2e:c3:07	14:ac:60:b5:1f:bf	TCP	5a79240b:5a79240b:5a...	5a79240b

编号	行为类型	协议	时间	域名/应用	具体行为	源端口	目的...	源IP地址	目的IP地址
	DNS解析	UDP	2025/06/24 03:13:...	broadcast.chat.bilibili....	正在观看B...	53	55955	185.102.150.36	179.164.168.192
	DNS解析	UDP	2025/06/24 03:13:...	broadcast.chat.bilibili....	正在观看B...	53	55955	179.164.168.192	166.63.178.221
	DNS解析	UDP	2025/06/24 03:13:...	broadcast.chat.bilibili....	正在观看B...	53	55955	179.164.168.192	185.102.150.36
	DNS解析	UDP	2025/06/24 03:13:...	broadcast.chat.bilibili....	正在观看B...	53	55955	179.164.168.192	5.5.5.223
	DNS解析	UDP	2025/06/24 03:13:...	broadcast.chat.bilibili....	正在观看B...	53	55955	179.164.168.192	114.114.114.114
	DNS解析	UDP	2025/06/24 03:13:...	broadcast.chat.bilibili....	正在观看B...	53	55955	179.164.168.192	114.114.114.114
	DNS解析	UDP	2025/06/24 03:13:...	broadcast.chat.bilibili....	正在观看B...	53	55955	179.164.168.192	114.114.114.114

统计数据

TCP包	3779	HTTP包	13	IPv4包	3696
UDP包	0	ARP包	4	IPv6包	128
ICMP包	6	ICMPv6	0	其它	0
				总计	7666

0 K/s　0 K/s　保存　读取

4.1.10 数据包统计前端可视化

Network Activity Dashboard

All Packets | HTTP Data | Sensitive Info | Table Stats | Chart Stats

Packet Base Log (Page 1 of 1824)

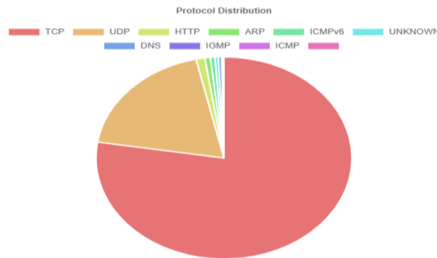
ID	Timestamp	Length	Src MAC	Dst MAC	Eth Proto	Src IP	Dst IP	IP Proto	Transport Proto	Src Port	Dst Port	App Proto
36349	2025-06-24 11:04:09	74	fa:21:94:dd:d5:ed	14:ac:60:b5:1f:bf	TCP	N/A	N/A	N/A	N/A	N/A	N/A	N/A
36358	2025-06-24 11:04:09	77	14:ac:60:b5:1f:bf	fa:21:94:dd:d5:ed	DNS	192.168.164.22	192.168.164.22	UDP	UDP	51499	53	N/A
36359	2025-06-24 11:04:09	77	14:ac:60:b5:1f:bf	fa:21:94:dd:d5:ed	DNS	192.168.164.22	192.168.164.22	UDP	UDP	55825	53	N/A
36360	2025-06-24 11:04:09	54	fa:21:94:dd:d5:ed	14:ac:60:b5:1f:bf	TCP	192.168.164.1	192.168.164.179	TCP	TCP	443	56250	N/A
36361	2025-06-24 11:04:09	54	fa:21:94:dd:d5:ed	14:ac:60:b5:1f:bf	TCP	192.168.164.1	192.168.164.179	TCP	TCP	443	56250	N/A
36362	2025-06-24 11:04:09	74	14:ac:60:b5:1f:bf	fa:21:94:dd:d5:ed	TCP	N/A	N/A	N/A	N/A	N/A	N/A	N/A
36363	2025-06-24 11:04:09	74	14:ac:60:b5:1f:bf	fa:21:94:dd:d5:ed	TCP	N/A	N/A	N/A	N/A	N/A	N/A	N/A
36364	2025-06-24 11:04:09	74	14:ac:60:b5:1f:bf	fa:21:94:dd:d5:ed	TCP	N/A	N/A	N/A	N/A	N/A	N/A	N/A
36365	2025-06-24 11:04:09	86	14:ac:60:b5:1f:bf	fa:21:94:dd:d5:ed	TCP	N/A	N/A	N/A	N/A	N/A	N/A	N/A
36366	2025-06-24 11:04:09	77	14:ac:60:b5:1f:bf	fa:21:94:dd:d5:ed	DNS	192.168.164.22	192.168.164.22	UDP	UDP	49268	53	N/A
36367	2025-06-24 11:04:09	77	14:ac:60:b5:1f:bf	fa:21:94:dd:d5:ed	DNS	192.168.164.22	192.168.164.22	UDP	UDP	65469	53	N/A
36368	2025-06-24 11:04:09	77	14:ac:60:b5:1f:bf	fa:21:94:dd:d5:ed	DNS	192.168.164.22	192.168.164.22	UDP	UDP	59138	53	N/A
36369	2025-06-24 11:04:09	77	14:ac:60:b5:1f:bf	fa:21:94:dd:d5:ed	DNS	192.168.164.22	192.168.164.22	UDP	UDP	57600	53	N/A
36370	2025-06-24 11:04:09	93	fa:21:94:dd:d5:ed	14:ac:60:b5:1f:bf	DNS	192.168.164.17	192.168.164.179	UDP	UDP	53	59138	N/A
36371	2025-06-24 11:04:09	217	fa:21:94:dd:d5:ed	14:ac:60:b5:1f:bf	DNS	192.168.164.17	192.168.164.179	UDP	UDP	53	51499	N/A
36372	2025-06-24	178	fa:21:94:dd:d5:ed	14:ac:60:b5:1f:bf	DNS	192.168.164.17	192.168.164.179	UDP	UDP	53	49268	N/A

Network Activity Dashboard

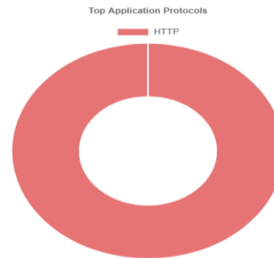
All Packets | HTTP Data | Sensitive Info | Table Stats | Chart Stats

Chart Statistics

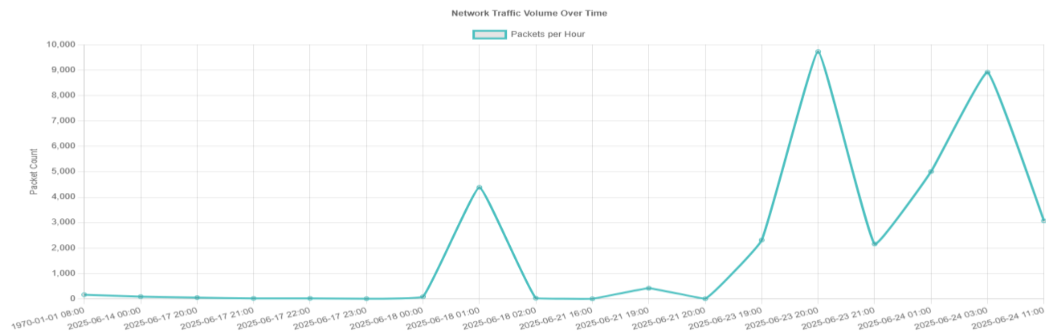
Protocol Distribution



Application Protocol Distribution



Traffic Over Time (Packets per Hour)



第 5 章 结论

5.1 完成度

本课程设计采用 Visual Studio 2012 C++，基于应用程序 Winpcap 来实现数据包的捕获与分析，使用 MFC 界面设计，界面较为友好。实现了 9 种协议数据包的抓取、分析。基于 DNS 域名解析，提供了一定的网络行为检测。实现了将抓取到的数据存储进数据库，利于抓取结束后数据包信息的统计。完成了前端可视化模块，对于数据包的统计情况能够更加直观的查询。

5.2 不足与提高点

抓包速度太快、抓包容易导致容器访问冲突

网络监控基于 DNS 解析，无法对数据包内容进行监控

5.3 总结

通过协议分析程序，我们可以将网络流量以图形化的方式呈现，使我 们能够更直观地了解网络中的通信模式和流量分布。这有助于我们快速发现异常流量或者网 络瓶颈，进而采取相应的措施。

协议分析程序可以分析捕获到的数据包，并识别出各种网络协议，如 TCP、 UDP、HTTP、SMTP 等。这可以帮助我们了解网络中正在使用的协议类型和其相应的用途。

基于 WinPcap 的协议分析程序可以提供关于网络流量的统计信息，如 流量量、流量的起源和目的地、传输速率等。通过分析这些统计信息，我们可以发现网络中 的流量模式和趋势，进而进行网络优化和性能改进。

基于 WinPcap 的协议分析程序还可以用于网络安全分析。通过检查数据包中 的内容和协议头部信息，我们可以发现潜在的安全漏洞、网络攻击行为或异常的网络行为。这对于网络安全人员来说非常重要，可以帮助他们及时发现并应对安全威胁。。

第 6 章 参考文献

[1] Lu, Xiaofan, Weijia Sun, and Huiping Li. "Design and research based on WinPcap network protocol analysis system." 2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering. Vol. 1. IEEE, 2010.

[2] Xiaoguang, An, and Lu Xiaofan. "Packet capture and protocol analysis based on Winpcap." 2016 International Conference on Robots & Intelligent System (ICRIS). IEEE, 2016.

[3] Wang, L., Li, Y. S., & Hu, L. W. Real-time detection and recovery of ARP Spoofing based on SNMP and WinPcap. In 2012 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER).

[4] Zhao, Ling, et al. "T-GCN: A temporal graph convolutional network for traffic prediction." IEEE transactions on intelligent transportation systems 21.9 (2019): 3848-3858.

[5] Shinde, Pankaj, and Thaksen J. Parvat. "DDoS attack analyzer: using JPCAP and WinCap." Procedia Computer Science 79 (2016): 781-784.

[6] Abu-El-Haija, Sami, et al. "N-gcn: Multi-scale graph convolution for semi-supervised node classification." uncertainty in artificial intelligence. PMLR, 2020.

[7] Ross, T-YLPG, and G. K. H. P. Dollár. "Focal loss for dense object detection." proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[8] Bawa, Mayank, Tyson Condie, and Prasanna Ganesan. "LSH forest: self-tuning indexes for similarity search." Proceedings of the 14th international conference on World Wide Web. 2005.

[9] Vijay, Aakanksha, and Mrs Savita Shiwani. "Intrusion Detection System Based Network Using SNORT Signatures And WINPCAP."

第 7 章 附录