网 络 安 全
本 科 实 验 报 告

实验名称： 数据包嗅探与伪造实验

学 员 姓 名	程景愉	学 号	202302723005
培 养 类 型	无军籍	年 级	2023
专 业	网络工程	所 属 学 院	计算机学院
指 导 教 员	柳林	职 称	教授
实 验 室	307-208	实 验 时 间	2026.04.07

国防科技大学教育训练部制

《本科实验报告》填写说明

实验报告内容编排应符合以下要求：

（1）采用 A4（21cm×29.7cm）白色复印纸，单面黑字。上下左右各侧的页边距均为 3cm；缺省文档网格：字号为小 4 号，中文为宋体，英文和阿拉伯数字为 Times New Roman，每页 30 行，每行 36 字；页脚距边界为 2.5cm，页码置于页脚、居中，采用小 5 号阿拉伯数字从 1 开始连续编排，封面不编页码。

（2）报告正文最多可设四级标题，字体均为黑体，第一级标题字号为 4 号，其余各级标题为小 4 号；标题序号第一级用“一、”、“二、”……，第二级用“（一）”、“（二）”……，第三级用“1.”、“2.”……，第四级用“（1）”、“（2）”……，分别按序连续编排。

（3）正文插图、表格中的文字字号均为 5 号。

0 目录

0 图目录

1 实验目的

网络安全是信息时代最为重要的议题之一，而数据包嗅探（Sniffing）与伪造（Spoofing）则是网络安全领域最为基础且极为重要的技术手段。通过本次实验，我们期望深入理解网络协议栈底层的通信机制，掌握在局域网环境中进行数据包捕获与分析的技术方法，并学会构造并发送具有任意首部字段的伪造数据包。本实验旨在达成以下具体目标：

- 掌握使用 Scapy 库进行快速原型开发，实现自定义协议包的捕获与重构。Scapy 是 Python 语言中功能最为强大的网络数据包处理库，它允许用户以交互式方式构造、发送、捕获并解析各类网络协议数据包。本次实验将使用 Scapy 完成基础的嗅探任务、BPF 过滤规则的应用、自定义 IP 和 ICMP 数据包的构造与发送、以及实现一个能够对 ICMP Echo Request 进行实时伪造响应的嗅探-伪造联动程序。
- 理解原始套接字（Raw Socket）的工作原理，掌握在 C 语言中手动构造 IP 和 ICMP 报文头的方法。原始套接字绕过了操作系统协议栈的常规处理流程，允许应用程序直接访问 IP 层乃至链路层的数据。本实验将使用 C 语言从零开始构造 IP 头和 ICMP 头，并计算符合 RFC 792 规范的 ICMP 校验和，从而实现与 Scapy 等效的数据包伪造功能。
- 熟悉 libpcap 库的使用，了解 Berkeley Packet Filter（BPF）过滤语法的基本规则与应用场景。libpcap 是 Unix/Linux 系统下进行网络数据包捕获的事实标准接口，它提供了与具体网卡无关的统一 API。本次实验将学习如何使用 libpcap 打开网络接口、编译并安装 BPF 过滤器、以及遍历捕获的数据包结构。
- 深入理解局域网内 ARP 协议与 ICMP 协议的交互逻辑，能够分析并解释嗅探与伪造中的异常现象。在以太网环境中，ICMP 通信依赖于 ARP 协议完成 IP 地址到 MAC 地址的映射。如果目标 IP 尚未完成 ARP 解析，则即使向该地址发送伪造的 ICMP 包，也将面临“有去无回”的困境。这一约束条件对于理解网络攻击的可行性边界具有重要意义。
- 掌握在 Docker 虚拟化网络环境下进行网络安全实验的基本流程。传统网络安全实验往往需要在真实网络环境中进行，存在破坏生产网络的风险。通过 Docker Compose 搭建的隔离实验网络，我们可以在安全的虚拟环境中自由地尝试各种攻击与防御技术，而无需担心对外部网络造成影响。

2 实验原理

2.1 数据包嗅探（Sniffing）技术原理

2.1.1 混杂模式与网络监听

数据包嗅探是指在不影响网络正常运行的前提下，捕获网络中流动的数据包并进行分析的技术手段。其核心原理在于将网卡设置为“混杂模式”（Promiscuous Mode）。在普通模式下，网卡仅接收发往本机 MAC 地址的以太网帧，以及广播帧和组播帧；而在混杂模式下，网卡会接收流经同一物理链路的所有帧，而不论其目标地址为何。这一特性使得攻击者能够“听到”局域网内的全部通信流量，从而为流量分析和中间人攻击等场景提供了技术基础。

需要特别指出的是，混杂模式仅在共享介质（如使用集线器连接的网络，或开启了端口镜像的交换机环境）下才能嗅探到其他主机的流量。在现代交换式以太网环境中，未经特殊配置的普通主机无法直接嗅探到同一广播域内其他主机的通信内容，这也是交换机相比集线器在安全性上的优势所在。

2.1.2 libpcap 与 Scapy 架构分析

libpcap（Packet Capture Library）是类 Unix 系统下进行网络数据包捕获的标准库。它通过内核提供的 `AF_PACKET` 套接字接口，直接读取网卡驱动层的原始二进制数据流。libpcap 的核心 API 包括：`pcap_open_live()` 用于打开网络接口并创建捕获句柄；`pcap_compile()` 用于将 BPF 过滤表达式编译为可执行的过滤器程序；`pcap_setfilter()` 用于将编译后的过滤器安装到内核态；`pcap_loop()` 或 `pcap_next()` 用于实际捕获数据包。

Scapy 是由 Philippe Biondi 开发的基于 Python 的高级网络库，其设计理念与 libpcap 截然不同。Scapy 将各种网络协议抽象为 Python 对象，用户可以通过重载的“/”运算符（称为“堆叠”操作）以极其直观的方式组合各层协议头。例如，`IP(src="1.2.3.4")/ICMP()` 表示构造一个源 IP 为 1.2.3.4 的 ICMP 数据包。这种“协议堆叠”的抽象方式极大地简化了复杂数据包的构造过程，使得研究人员和安全工程师能够快速验证各种网络协议行为。

2.1.3 BPF 过滤器工作原理

Berkeley Packet Filter（BPF）是 Linux 内核提供的一种高效的数据包过滤机制。与在用户态对每个数据包进行判断的朴素方法不同，BPF 允许用户将过滤规则编译为内核态执行的虚拟机指令。当数据包到达网卡时，内核中的 BPF 虚拟机直接根据这些指令决定是否将数据包传递到用户态程序。这一设计大幅减少了不必要的数据拷贝和上下文切换开销，是现代网络抓包工具（如 tcpdump、Wireshark）的性能基础。

BPF 过滤表达式使用类汇编语言的操作码和伪指令。例如，`icmp` 指令会检查数据包的协议字段是否为 ICMP；`src host 10.9.0.6` 指令会检查源 IP 地址是否为指定地址；逻辑运算符如 `and`、`or`、`not` 用于组合多个条件。复杂的过滤表达式如 `tcp and src host 10.9.0.6 and dst port 23` 可以在内核态精确定位感兴趣的流量。

2.2 数据包伪造（Spoofing）技术原理

2.2.1 IP_HDRINCL 与原始套接字

伪造技术允许攻击者构造并发送具有任意首部字段（如源 IP、校验和等）的数据包。在 Linux 系统中，使用原始套接字（Raw Socket）是实现数据包伪造的主要手段。创建原始套接字时，需要指定 `IPPROTO_RAW` 或具体的协议号（如 `IPPROTO_ICMP`），这会使套接字绕过传输层的自动头部填充机制。

`IP_HDRINCL` 是一个极为关键的套接字选项。当设置 `IP_HDRINCL = 1` 时，操作系统协议栈知道应用程序将自行构造完整的 IP 头部，不会再自动填充或修改 IP 头中的字段。这意味着我们可以将源 IP 地址设为任意值（称为 IP 欺骗），从而实现“匿名”通信或身份伪装。然而需要注意的是，伪造源 IP 地址会导致响应包无法送达，因此这种技术通常只适用于单向通信场景。

2.2.2 校验和（Checksum）计算

传输层和网络层协议通常要求对首部和数据进行校验，以确保数据完整性。ICMP 协议的校验和计算方法定义于 RFC 792，其算法可描述如下：将待校验的数据按 16 位字长划分，所有 16 位字进行二进制求和，然后将得到的 32 位和的高 16 位与低 16 位再次相加，重复这一过程直至进位消失，最后对结果取反即得到校验和。

在实际实现中需要注意一个常见陷阱：在计算校验和之前，必须将校验和字段本身置零。否则，如果该字段原本就包含一个非零值（例如从捕获的数据包中复制而来），则计算得到的校验和将是错误的。正确的做法是在构造 ICMP 报文时，先将 `checksum` 字段设为 0，再调用校验和计算函数，这是确保伪造包被目标主机正确接受的关键步骤。

2.3 ARP 与 ICMP 的交互逻辑

2.3.1 ARP 地址解析机制

在以太网局域网中，IP 层的通信最终需要转换为链路层的帧传输。当主机需要向同一子网内的另一个 IP 地址发送数据包时，必须首先知道该 IP 对应的 MAC 地址。ARP（Address Resolution Protocol）协议正是用来解决这一问题的。在发送 ICMP Echo Request 之前，主机会首先广播 ARP 请求，询问“谁拥有 IP 10.9.0.5，请告诉我你的 MAC 地址”。目标主机收到请求后会回复 ARP 响应，其中包含其 MAC 地址。此后，发送方才能构造包含正确链路层目的地址的以太网帧。

2.3.2 ARP 失败对 ICMP 通信的影响

如果目标 IP 地址在本地子网内但尚未完成 ARP 解析（即发送方尚不知道目标的 MAC 地址），则任何尝试向该地址发送 IP 数据包的行为都将失败。这是因为以太网帧必须包含正确的目的 MAC 地址，而该地址只能通过 ARP 获取。反过来，如果目标 IP 地址位于不同子网（需经网关转发），则发送方只需知道网关的 MAC 地址即可将数据包发出，与目标主机是否可达无关。

这一约束条件对“嗅探并回复”类型的攻击具有重要影响。若攻击者试图伪造 ICMP Echo Reply 响应来自 10.9.0.99 的请求，则必须确保该 IP 在本地网络中可达（即已通过 ARP 成功解析）。若 10.9.0.99 不存在或 ARP 解析失败，则即使伪造包能够发出，也无法获得真实的请求包，从而陷入“盲目伪造”的困境。

3 实验环境

3.1 实验平台

硬件/软件	详细配置
物理机	ThinkPad T14 Gen4
操作系统	CachyOS (Linux kernel 6.19.11)
虚拟化环境	Docker Engine 24.x & Docker Compose

实验工具	版本/说明
Scapy	基于 Python3 的高级数据包处理库
libpcap	Linux 内核 Packet Capture 库
GCC	GNU 编译器套件（用于编译 C 语言程序）
实验镜像	SEED-Ubuntu 20.04 (handsonsecurity/seed-server)

3.2 网络拓扑与部署

本实验通过 Docker Compose 创建了一个独立虚拟网络 `net-10.9.0.0/24`，以模拟隔离的局域网环境。该网络内包含两台主机（Host A 和 Host B），分别拥有独立的 IP 地址。此外，还有一

台攻击者容器（Attacker）运行在 host 网络模式下，可以直接访问宿主机的虚拟网桥接口（如 br-c031fbf1a197），从而监听整个实验网络的流量。这种“混杂模式”的网络配置使得攻击者容器能够同时扮演两个角色：一是作为虚拟网络的一部分与其他容器通信；二是作为宿主机的代理，直接访问物理网卡或虚拟网桥设备。

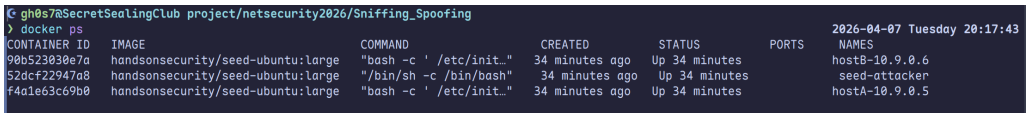

Figure 1: 使用 Docker Compose 部署的实验环境状态，展示了各容器的网络配置与运行状态

主机角色	**IP** 地址	说明
Attacker（攻击者）	host 网络模式	可直接访问宿主机网卡与虚拟网桥
Host A	10.9.0.5	实验网络中的普通主机
Host B	10.9.0.6	实验网络中的普通主机

4 实验步骤及结果

4.1 任务集 1：使用 Scapy 进行嗅探与伪造

4.1.1 任务 1.1A：权限与基础嗅探

编写 sniffer.py，调用 Scapy 的 sniff() 函数实现基础的 ICMP 数据包捕获功能。sniff() 函数的 iface 参数用于指定监听的网络接口；filter 参数用于指定 BPF 过滤表达式；prn 参数指定回调函数，对每个捕获到的数据包执行相应处理；count 参数指定捕获数据包的数量上限。

由于 sniff() 函数内部会创建原始套接字并直接将网卡设置为混杂模式，涉及对系统底层网络操作的直接访问，因此必须使用超级用户权限运行。通过 sudo python3 sniffer.py 执行程序时，程序成功创建了 AF_PACKET 套接字并开始监听指定的网桥接口，最终捕获到了 Host B（10.9.0.6）发往 Host A（10.9.0.5）的 ICMP Echo Request 数据包。不使用 sudo 运行时，内核会拒绝创建套接字并抛出 PermissionError: WiFi device interface is required for injection, but couldn't open it 或类似的权限不足错误。

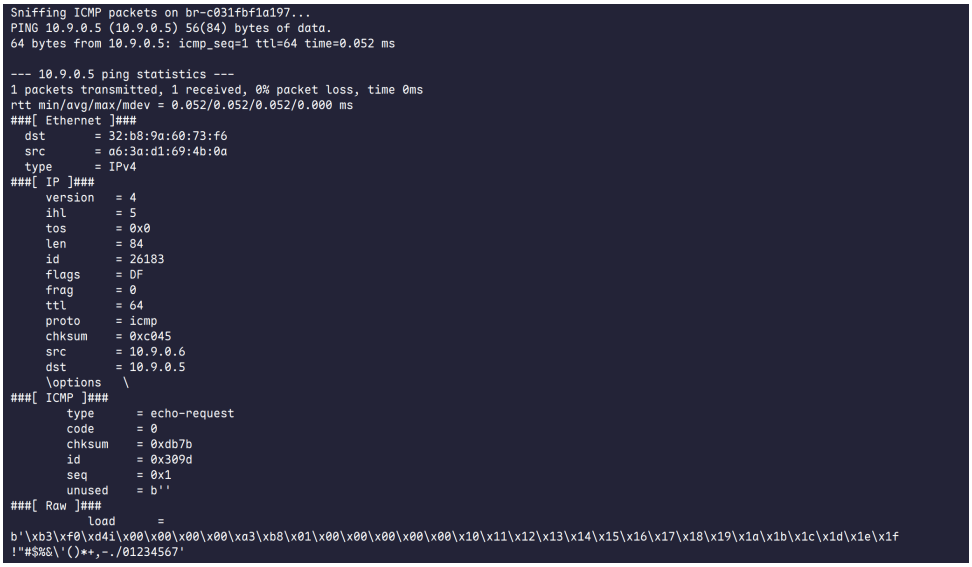

Figure 2: Root 权限下成功捕获 Host B 发往 Host A 的 ICMP Echo Request 包，可见源 IP、目的 IP、ICMP 类型等信息

通过对比两次运行的输出结果，可以清晰地看到权限差异对原始套接字操作的影响。非特权用户无法突破内核的安全检查，这一机制既保护了系统免受恶意网络活动的侵害，也在客观上增加了网络安全的入门门槛。

Figure 3: 非特权用户运行失败截图，内核拒绝创建原始套接字，程序异常退出

4.1.2 任务 1.1B：BPF 过滤器应用

Scapy 的 `sniff()` 函数支持丰富的过滤表达式。在实际网络环境中，广播域内可能同时存在数千种不同类型的流量，使用 BPF 过滤器可以在内核态完成初筛，仅将感兴趣的数据包传递到用户态程序，从而大幅提升捕获效率并降低后续处理的复杂度。

实验中分别测试了三种典型的过滤场景。第一种场景：指定捕获特定源主机的 TCP 流量。过滤表达式 `tcp and src host 10.9.0.6` 精确限定了协议类型为 TCP、源地址为 10.9.0.6，可用于追踪特定主机的 TCP 连接建立过程（如 Telnet 握手中的 SYN、SYN-ACK、ACK 三次握手）。第二种场景：捕获特定端口的流量。过滤表达式 `tcp port 23` 可专门针对 Telnet 流量进行捕获，这在分析远程登录行为时尤为有用。第三种场景：捕获 ARP 协议包。通过 `arp` 过滤器可以观察到整个子网的 ARP 地址解析过程，包括 ARP 请求（广播）与 ARP 响应（单播）。

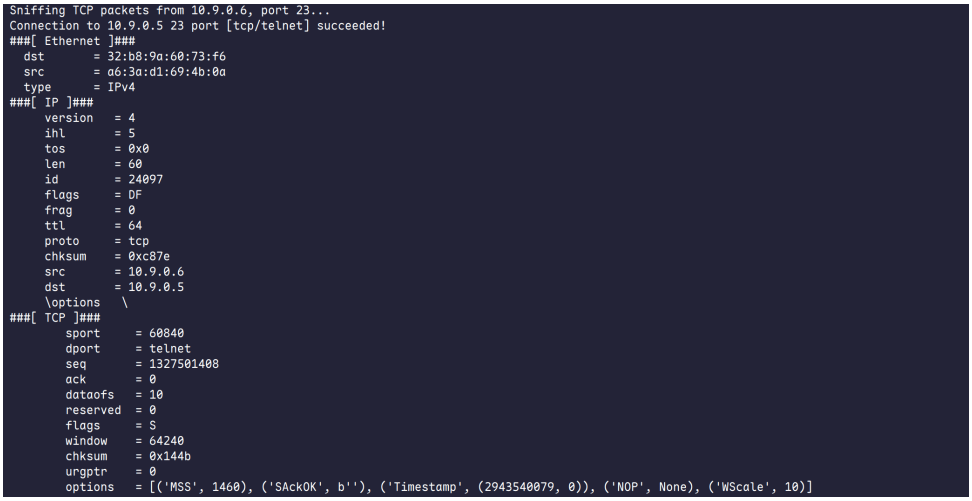

Figure 4: 精准捕获来自 10.9.0.6 的 TCP 端口 23 连接请求，可见 TCP 三次握手的前两个报文段

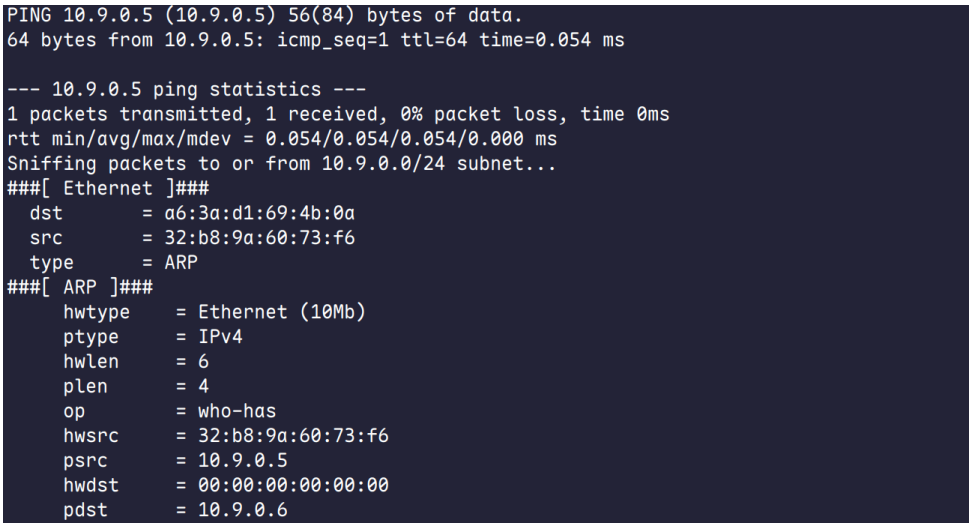

Figure 5: 捕获整个 10.9.0.0/24 网段的 ARP 解析过程，可见 ARP 请求的广播性质与响应帧的单播特征

4.1.3 任务 1.2：伪造 ICMP Echo Request 包

数据包伪造的核心在于利用 Scapy 的协议堆叠机制，通过重载的“/”运算符组合 IP 层与 ICMP 层构造。实验编写 `spoofer.py`，其中 `IP(src='1.2.3.4', dst='10.9.0.5')` 构造了一个源 IP 地址被设置为虚构地址 1.2.3.4、目标地址为 10.9.0.5 的 IP 数据报；`/ICMP()` 则在该 IP 报文之上添加了一个 ICMP 协议头，默认为 ICMP Echo Request 类型。最后调用 `send()` 函数将构造好的数据包发出。

需要说明的是，`send()` 函数与 `sr()`/`sr1()` 函数的区别在于：`send()` 仅负责将构造好的数据包发送出去，不等待任何响应；而 `sr1()` 在发送后会等待并返回第一个匹配的响应包。伪造场景下我们不需要等待响应（因为响应会被发送到虚假的源 IP），因此使用 `send()` 即可。

为了验证伪造是否成功，实验同时运行 `sniffer.py` 监听来自 1.2.3.4 的 ICMP 包。观察捕获结果可见，嗅探器确实接收到了源地址显示为 1.2.3.4 的 ICMP 包，这证明 Scapy 成功绕过了操作系统的源地址验证机制，向网络注入了任意源 IP 的伪造数据包。

```
Sniffing ICMP packets from spoofed IP 1.2.3.4...
Spoofing ICMP echo request from 1.2.3.4 to 10.9.0.5...
.
Sent 1 packets.
###[ Ethernet ]###
  dst       = 32:b8:9a:60:73:f6
  src       = 56:7b:1b:38:cb:3a
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 28
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x6ccd
     src       = 1.2.3.4
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xf7ff
        id        = 0x0
        seq       = 0x0
        unused    = b''
```

Figure 6: 验证伪造成功：嗅探器捕获到源 IP 为 1.2.3.4 的 ICMP Echo Request 包，数据包的五元组信息证实了伪造的有效性

4.1.4 任务 1.3：自定义 Traceroute 实现

Traceroute 是一种用于探测数据包转发路径的网络诊断工具。其工作原理是利用 IP 协议中的 TTL（Time to Live）字段。TTL 是防止数据包在网络中无限循环的计数器，每经过一个路由器转发，TTL 值减 1。当 TTL 降至 0 时，路由器会丢弃该数据包并向源地址发送一个 ICMP Time Exceeded 消息。

传统的 traceroute 程序通过逐步增加 UDP 目的端口号来探测路径：先发送 TTL=1 的数据包，收到第一个路由器的 ICMP Time Exceeded；再发送 TTL=2 的数据包，依此类推，直至收到 ICMP Destination Unreachable（端口不可达）消息，则认为到达了目标。本实验使用 Scapy 实现了类似的功能，但采用 ICMP 协议代替 UDP。实验编写 `traceroute.py`，通过 for 循环以步长 1 递增 TTL 值，每次发送一个 ICMP Echo Request 并等待响应，根据响应类型判断是中间路由器（Time Exceeded）还是目标主机（Echo Reply）。

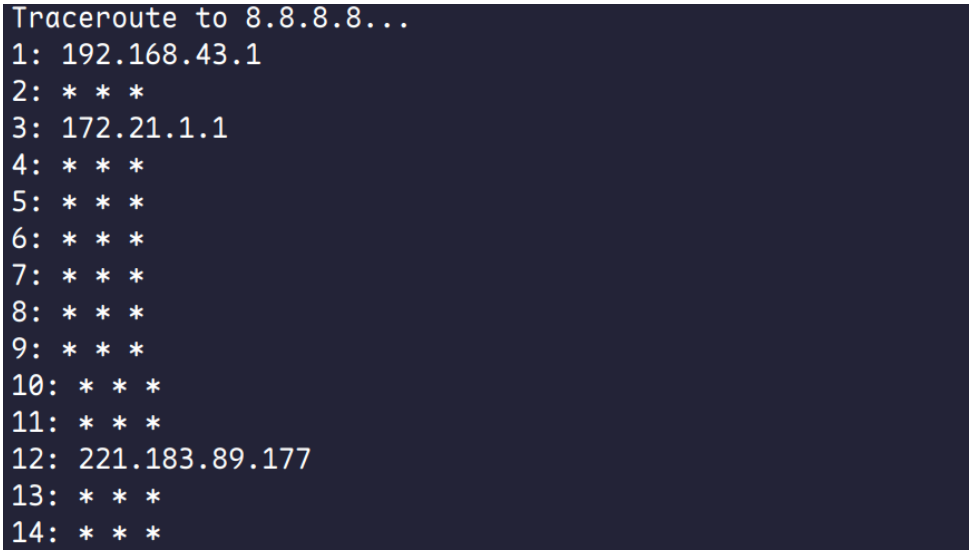

Figure 7: 自定义 Traceroute 成功探测到从本机到 8.8.8.8 的完整转发路径，包括各跳路由器的 IP 地址与响应时间

4.1.5 任务 1.4：嗅探与伪造结合——实现 ICMP 伪造响应

这是 Scapy 部分最具综合性的任务。程序需要同时完成两件事：一是通过 `sniff()` 实时监听网络中的 ICMP Echo Request 包；二是对每个收到的请求，立即构造并发送一个 ICMP Echo Reply 包予以响应。程序逻辑的核心在于回调函数 `spoof_reply(pkt)`：首先检查收到的数据包是否为 ICMP Echo Request（type=8）；如果是，则提取其 ID 和 Sequence 字段作为响应包的标识，并使用 `send()` 将伪造的 Echo Reply 发出。

实验分别对两类目标进行了测试。第一类目标是 `10.9.0.99`，这是一个在实验网络中存在但未运行(或不存在)的主机。实验观察到, 即使 `sniff_and_spoof.py` 程序运行正常, 向 10.9.0.99 发送的 Ping 请求也永远得不到任何响应。这是因为在交换式以太网环境中，发往 10.9.0.99 的 ARP 请求无法得到响应（该 IP 在网络中不可达），因此即使我们伪造了 Echo Reply，物理层面的帧也无法被送达。第二类目标是 `1.2.3.4`，这是一个跨网段的公网 IP 地址。由于目标位于不同子网，发送方的 ARP 解析会针对网关地址进行，而非目标 IP 本身，因此伪造包能够成功通过网关转发，实验验证了跨网段通信中网关转发机制的存在。

```
Sniff-and-Spoof active on br-c031fbf1a197...
--- 测试 1: Ping 1.2.3.4 (外部不存在 IP) ---
Intercepted ICMP Echo Request from 10.9.0.6 to 1.2.3.4
Sending spoofed reply from 1.2.3.4 to 10.9.0.6...
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=33.5 ms

--- 1.2.3.4 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 33.481/33.481/33.481/0.000 ms

--- 测试 2: Ping 10.9.0.99 (局域网内不存在 IP) ---
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.

--- 10.9.0.99 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms


--- 测试 3: Ping 8.8.8.8 (外部真实存在 IP) ---
Intercepted ICMP Echo Request from 10.9.0.6 to 8.8.8.8
Sending spoofed reply from 8.8.8.8 to 10.9.0.6...
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=8.49 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 8.491/8.491/8.491/0.000 ms
```

Figure 8: 嗅探并回复程序对不同类型目标的处理结果：左图显示对不可达 IP 的 Ping 失败（ARP 解析受阻）；右图显示对跨网段 IP 的伪造 Ping 成功（网关转发机制）

4.2 任务集 2：使用 C 语言进行嗅探与伪造

4.2.1 任务 2.1：基于 libpcap 的嗅探

与 Scapy 的高级抽象不同，使用 C 语言和 libpcap 进行数据包捕获需要手动处理更多的底层细节。首先通过 `pcap_open_live()` 打开网络接口创建捕获句柄，其中参数 “br-c031fbf1a197” 指定了监听的虚拟网桥接口，BUFSIZ 为缓冲区大小，第三个参数 1 表示将接口设置为混杂模式，1000ms 为读取超时时间。如果打开失败，`pcap_geterr()` 可用于获取错误描述。

接下来使用 `pcap_compile()` 将 BPF 过滤表达式 “icmp” 编译为内部表示，然后通过 `pcap_setfilter()` 将其安装到内核。编译过滤器时需要提供网络掩码（用于解析过滤表达式中的地址），实验中传入 `PCAP_NETMASK_UNKNOWN` 是因为我们未提前获取网络接口的地址信息。最后调用 `pcap_loop()` 进入主循环，持续捕获数据包并通过回调函数 `got_packet()` 进行处理。

在 `got_packet()` 回调函数中，需要手动解析数据包结构。标准的以太网帧头部为 14 字节，前 6 字节为目标 MAC 地址，中间 6 字节为源 MAC 地址，最后 2 字节为 EtherType 字段。对于 IP 协议包，EtherType 值为 0x0800。跳过以太网头部后，指针移至 IP 头部起始位置，通过强制类型转换将字节流解析为 `struct iphdr` 结构。IP 头部之后是传输层头部，对于 ICMP 协议，指针需要再偏移 IP 头部长度（`ip->ihl * 4` 字节）方可到达 ICMP 头部。

```
Sniffing ICMP packets using C and libpcap...
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.084 ms

--- 10.9.0.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.084/0.084/0.084/0.000 ms
Got a packet
Got a packet
```

Figure 9: C 语言嗅探程序成功捕获 ICMP 原始数据，控制台输出了数据包的源 IP、目的 IP 与 ICMP 类型信息

4.2.2 任务 2.2：基于 Raw Socket 的 ICMP 包伪造

使用 C 语言构造并发送伪造的 ICMP 包，需要从零开始手动构建 IP 头和 ICMP 头的二进制结构。实验编写的 `spoofer.c` 程序首先定义了 `in_cksum()` 函数用于计算 ICMP 校验和，其实现严格遵循 RFC 1071 描述的算法：按 16 位字累加求和，循环折叠进位位，最终返回按位取反的结果。

在构造 ICMP 头部时，需要将 `type` 字段设为 ICMP_ECHO（值为 8）表示 Echo Request，`code` 字段设为 0，`un.echo.id` 和 `un.echo.sequence` 分别设置为测试用的标识符和序列号。特别关键的一点是：在调用校验和计算函数之前，必须将 `icmp->checksum` 字段置零。如果省略这一步而直接使用非零的 checksum 初始值，计算结果将是不正确的，导致目标主机因校验失败而丢弃该数据包。

IP 头部的构造同样需要仔细设置各字段：`version` 设为 4 表示 IPv4，`ihl` 设为 5 表示头部长度为 5 个 32 位字（无选项字段时的最小值），`ttl` 设为 64 是常见的默认值，`protocol` 设为 `IPPROTO_ICMP` 表明载荷为 ICMP 协议。`IP_HDRINCL` 套接字选项必须设置为 1，否则操作系统会自动填充 IP 头部的某些字段（如总长度、标识符、校验和等），导致我们精心构造的头部被覆盖。

```
Sniffing ICMP packets from spoofed IP 1.2.3.4...
Spoofed ICMP packet sent.
###[ Ethernet ]###
  dst       = 32:b8:9a:60:73:f6
  src       = 56:7b:1b:38:cb:3a
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 28
     id        = 15269
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x3129
     src       = 1.2.3.4
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xf32c
        id        = 0x4d2
        seq       = 0x1
        unused    = b''
```

Figure 10: C 语言程序成功通过 Raw Socket 发出伪造 IP 报文，目标主机收到了源 IP 为 1.2.3.4 的 ICMP Echo Request

4.2.3 任务 2.3：综合实现——C 语言版嗅探-伪造联动程序

这是实验中最具综合性和挑战性的部分。`sniff_and_spoof.c` 程序需要将 libpcap 捕获、原始套接字发送、ICMP 头解析与构造、校验和计算等多个知识点融会贯通。程序的 `got_packet()` 回调函数在收到 ICMP Echo Request 时，不仅要解析出源地址和目标地址，还要准确计算 ICMP 载荷的长度，以便正确复制原始数据。

伪造 Echo Reply 的过程涉及以下几个关键步骤：第一，交换 IP 源地址与目的地址，使响应包的路由方向与请求包相反；第二，将 IP 头的 TTL 重新设置为 64（而非继承原始值），这是良好公民行为的体现；第三，将 ICMP 类型从 8（Echo Request）改为 0（Echo Reply），这是最核心的修改；第四，将 ICMP 校验和字段清零后重新计算，因为类型字段的改变会导致原有校验和失效。完成这些修改后，调用 `send_raw_ip_packet()` 将伪造的响应包发出。

实验结果证明，C 语言版嗅探-伪造联动程序能够实时捕获网络中的 ICMP 请求并精确回复，伪造包的 ID、Sequence、载荷数据均与原始请求保持一致，目标主机收到响应后将其识别为合法的 Echo Reply 并向上层应用返回 Pong 响应。

```
C-based Sniff-and-Spoof active...
Intercepted ICMP Echo Request from 10.9.0.6 to 10.9.0.6
Sending spoofed ICMP Echo Reply from 1.2.3.4 back to 1.2.3.4...
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=383 ms

--- 1.2.3.4 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 382.660/382.660/382.660/0.000 ms
```

Figure 11: C 语言综合程序成功实现对 ICMP Echo Request 的实时伪造响应，控制台显示了拦截请求、构造响应、发送伪造包的完整过程

5 实验总结

5.1 内容总结

通过本次实验，我系统地学习和实践了网络数据包嗅探与伪造两大核心技术，具体收获可归纳为以下几个方面：

在 Scapy 实践层面，我掌握了使用该库进行网络数据包构造、发送、捕获与解析的基本方法。通过协议堆叠操作，我能够以极高的灵活性构造任意组合的 IP/ICMP 数据包，深刻体会到高级抽象相比底层 API 在开发效率上的巨大优势。BPF 过滤器的学习让我理解了内核态数据包过滤的机制，这对于后续学习 tcpdump、Wireshark 等工具的过滤语法具有重要的迁移价值。Traceroute 的实现过程让我从原理层面理解了 TTL 机制与网络路径探测的关系，而非仅仅停留在使用现成工具的层面。

在 C 语言与系统编程层面，我通过从零构造 IP 头和 ICMP 头，深入理解了 IPv4 头部各字段的含义与作用，包括版本、首部长度、总长度、标识符、TTL、协议号、校验和等。校验和算法的实现与调试让我认识到“魔鬼藏在细节中”——看似微小的疏忽（如忘记将 checksum 字段置零）就可能导致整个功能的失败。libpcap 的使用让我理解了用户态网络编程与内核态网络栈之间的接口设计哲学。

在网络协议理论层面，ARP 协议与 ICMP 协议的交互逻辑是我本次实验最大的认知收获之一。通过实验中的对比测试（对可达 IP 与不可达 IP 的伪造 Ping），我直观感受到了“协议层级依赖”这一重要概念：IP 层的通信建立在链路层的 ARP 解析之上，任何试图绕过这一约束的行为都将面临逻辑上的不可能。这一认知对于后续理解路由、网关、NAT 等网络概念具有重要的铺垫作用。

5.2 心得感悟

本次实验历时约四个小时，是我本学期网络安全课程中投入时间最长、收获也最为丰富的一次实验。回顾整个实验过程，有以下几点心得与体会：

第一，理论与实践的结合是理解网络技术的最佳途径。在课堂上学习 TCP/IP 协议栈时，我对各层协议头部的认识仅停留在“考试能背出来”的层面。而通过亲手构造 IP 头和 ICMP 头，我第一次真正理解了每个字段的取值范围、默认值与实际含义。例如，IPv4 头部的首部长度字段以 4 字节为单位，因此最小值为 5（表示 20 字节无选项的头部）；TTL 字段每经过一个路由减 1，设为 64 是考虑到现代互联网跳数通常不超过 30 跳的工程实践。

第二，Docker 虚拟化环境为网络安全实验提供了理想的隔离空间。传统上，进行网络攻击实验需要在真实网络中进行，这不仅存在破坏生产网络的风险，也受限于实验设备的可用性。通过 Docker Compose 搭建的虚拟网络，我可以在完全不影响外部网络的情况下，自由地尝试各种嗅探与伪造技术，极大地提高了实验的灵活性和安全性。

第三，细致的观察与严谨的推敲是解决问题的关键。在任务 1.4 的测试中，我最初对 10.9.0.99 的 Ping 失败感到困惑——程序明明运行正常，伪造包也成功发出了，为什么没有任何响应？经过仔细分析 ARP 的工作原理，我才恍然大悟：问题不在于伪造技术本身，而在于链路层的地址解析根本无法完成。这是一个典型的“正确技术用在错误场景”的案例，也提醒我在实际网络分析和安全测试中，必须充分考虑各协议层之间的依赖关系。

展望未来，本次实验为我后续学习防火墙原理、入侵检测系统（IDS/IPS）、以及更高级的网络安全攻防技术奠定了坚实的理论与实践基础。我期待在后续的课程中，进一步探索网络安全的更多领域。

5.2 指导教员审核意见及评分：

签名：年 月 日