基于机器学习的网络流量分析审计系统
设计文档

郑宇森 **王鑫** **赵鸿宇**

韩志鹏 **姜来** **喻路稀**

1 引言

TrafficCat 是我们小组开发的一款基于机器学习的网络流量分析审计系统。**该系统为网络流量分析审计应用场景提供了一整套解决方案，包括网络流量捕获、流量解析、数据存储和机器学习分类等功能**。

通过我们的系统，用户可以轻松地进行网络流量分析和审计工作。系统可以高效地捕获网络中的流量数据，并对这些数据进行解析和处理，提取出有价值的信息和特征。捕获及解析后的数据可以进行灵活的存储，以便用户进行后续的分析和查询。此外，我们的系统还应用了机器学习算法，可以对网络流量进行分类和识别，帮助用户发现潜在的安全威胁和异常行为。无论是网络运营商、安全团队还是企业组织，都可以从我们的系统中获得准确、可靠的网络流量分析结果，提高网络安全性和监控能力。

除此之外，我们还将无 GUI 界面的精简化系统进行了 Docker 封装，以便在教学演示等场景中使用。这个小型化系统具有精简的功能集，可以轻松部署和运行，无需繁琐的配置过程。通过 Docker 技术，用户可以快速地搭建起整个系统环境，提供了更加便捷和灵活的使用方式。这样的封装使得系统的部署和使用变得简单而高效，不论是教师、学生还是技术爱好者，都可以通过我们的小型化系统，深入了解网络流量分析和相关应用，获得实际操作和体验的机会。

2 系统架构

TrafficCat 的整体系统架构如图 1所示，主体由 5 个模块构成。

- WireCat 网络嗅探器：该模块是一个基于 Qt 平台, 主要由 C++ 语言编写的网络嗅探器软件，用于捕获和分析网络流量。底层使用 libpcap 库捕获原始网络数据包, 实现了数据包解析, 地址, 端口号与协议过滤, 载荷内容查找, IP 分片重组, 日志记录等功能。我们提供了 GUI（用户图形界面）和 CLI（命令行交互）两种使用模式，能够满足用户的多样化需求。运行该模块，我们可以获得原始的 pcap 文件，并交给后续的模块解析处理。
- MinIO 数据库：该模块基于开源对象存储库 minio 构建，用于存放嗅探器抓取到原始流量文件、预处理生成的 extractor 文件以及模型输出的 label 文件。系统运行时将自动调取 upload 脚本上传文件，可通过访问数据库开放的管理端口查看文件信息，也可调取 download 脚本进行下载与统一审计。
- KDD99 特征提取器：该模块由 C++ 语言编写，负责从 WireCat 抓取的 pcap 文件中提取出与网络流量相关的特征，并对特征进行处理与编码，以便输入 NTML 模块进行流量分析。
- NTML (Network Traffic Machine Learning) 模型：该模块基于 PyTorch 的 nn 库，使用 KDD Cup99 数据集训练了一个网络流量分析神经网络。使用该模型，可以分析出 20 余种不同的恶意流量。

- Dash Board 网页：该模块用于最终呈现我们系统的分析结果，向用户提供直观的可视化和动态交互。包括流量分析时序图，流量属性表，IP 拓扑关联图谱，端口和协议计数，流量标签等。这可以帮助用户快速把握流量的整体趋势和关键特征，识别特定类型或特定标签的流量。

我们的系统的开源地址为 `https://github.com/ysyszheng/TrafficCat` ，该开源项目提供了系统的完整源代码和相关文档。任何用户都可以基于源码二次开发，使系统适应不同的网络流量分析需求和场景。

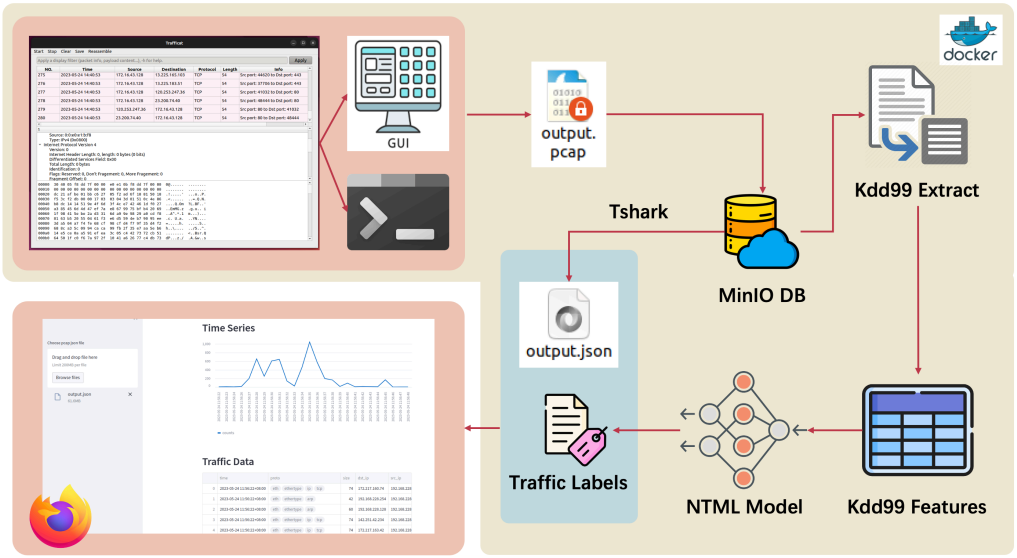

Figure 1: 系统架构图

3 网络嗅探器

3.1 简介

网络嗅探器 (Sniffer)，也被称为包嗅探器、协议分析器或网络分析器，是一种计算机软件或硬件工具，可以监控、记录、分析网络上的数据包交换。这些数据包是网络上所有通信的基础，也是构建网络分析审计系统所需要的基础工具。

我们团队自主开发了一个网络嗅探器 WireCat，支持**图形化界面运行 (GUI)** 和**命令行运行 (CLI)** 两种模式，满足用户的个性化需求。在 GUI 模式中，实现了友好的用户界面。项目采用 QT5 实现用户友好的 GUI，整体布局和交互模仿主流抓包软件 Wireshark；在 CLI 模式中，支持抓包与数据包的保存，通过“start/stop/exit”三个命令实现简洁高效的抓包流程。WireCat 实现的功能有：

- **基本功能**：通过指定需要侦听的网卡，抓取进出本主机的数据包，并解析数据包的内容。支持 IPv4、IPv6、ARP、TCP、UDP、ICMP、IGMP 协议；
- **IP 分片重组**：在 GUI 模式中，对于选中的数据包，若满足 IP 重组条件，能够对其进行重组；
- **包过滤/包查询**：在 GUI 模式中，能够在过滤框中输入源/目的 IP 地址、源/目的端口、协议等条件，从已抓获的数据包中过滤出满足条件的包；同时能根据一定的查询条件（如包内容包含“SEARCH”）显示所有满足查询条件的数据包；
- **数据包保存**：在 GUI 或 CLI 模式中，停止抓包之后，数据包会被以.pcap 文件的格式自动保存，并且保存文件具有可读性；

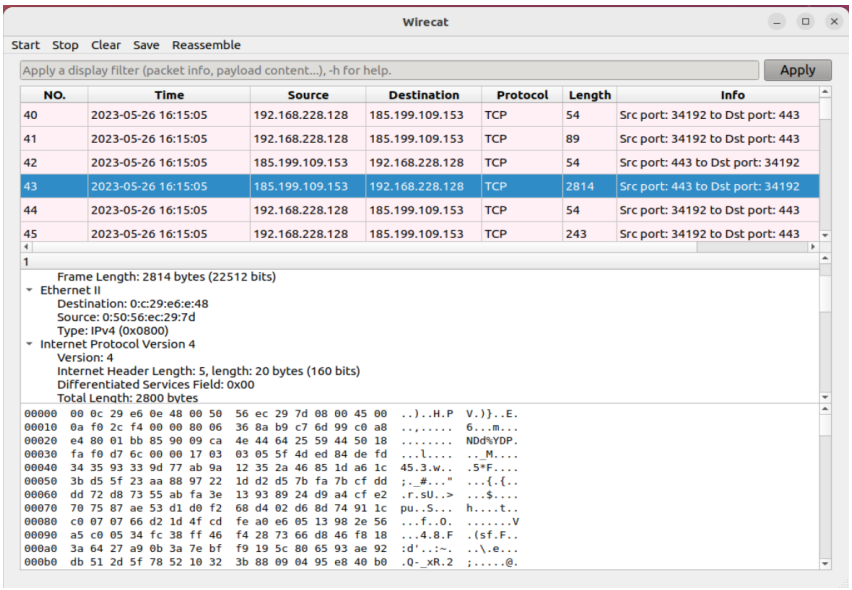

Figure 2: Wirecat 用户图形界面设计

3.2 程序逻辑

Wirecat 程序创建了 5 个类，分别为 MainWindow, DevWindow, View, Filter, Sniffer，并将工具函数和数据结构定义封装在 utils 文件夹中。其中 GUI 模式用到了所有的类，而 CLI 模式只用到 Sniffer 类。整体逻辑框架如图 3所示。

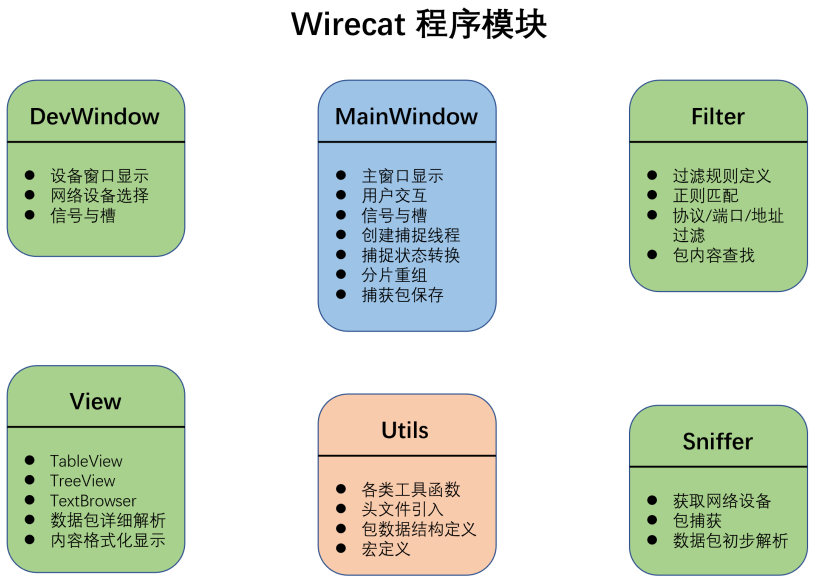

Figure 3: 程序模块

3.3 功能实现

3.3.1 包解析

GUI 模式：在 MainWindow 类中，通过 moveToThread 启动 Sniffer 类的数据包捕捉线程。通过槽函数 MainWindow::showMainWnd() 发送信号，开启捕捉线程的 sniff() 函数。该函数

判断当前的捕获状态，若为 Start，调用 pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user) 函数，每次捕捉一批量的数据包。每当抓获一个数据包后，通过函数指针调用回调函数 get_packet()，解析数据包的捕获时间、序号、协议类型、长度等信息，并将捕获到的数据包实时存储到指定的存储文件中。将这些信息记入 packet_struct 结构体，并通过 View:::add_pkt() 函数传入 View 类中。View 类接收新数据包后，更新 TableView 的内容。当用户点击 TableView 中的某行数据包时，TreeView 和 TextBrowser 中会分别呈现数据包头部的详细信息和十六进制打印的包内容信息。GUI 模式下的模块交互设计如图 4所示。

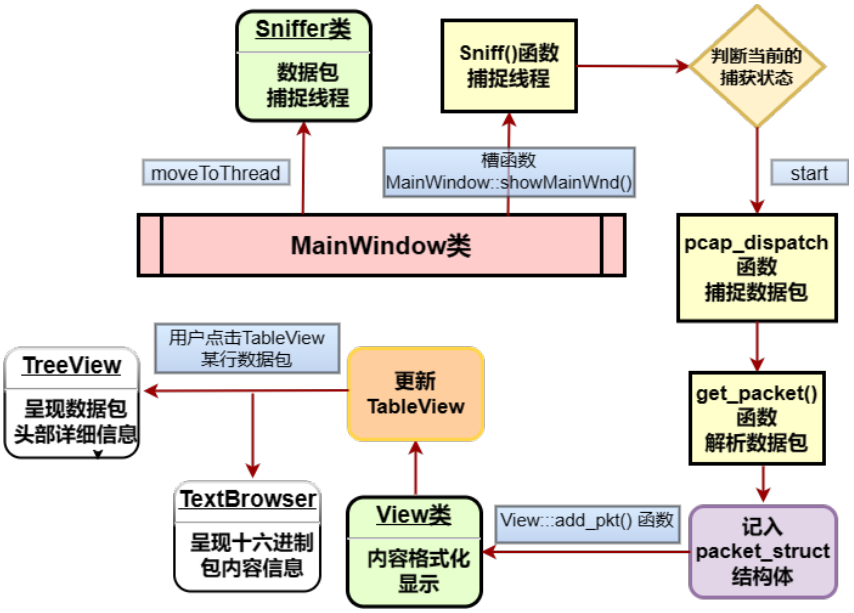

Figure 4: GUI 模式模块交互

CLI 模式：与 GUI 模式相比，CLI 模式下的 Sniffer 只需要完成抓包的功能即可，不需要与 UI 界面进行交互，是精简化系统的抓包部分。CLI 模式运行时，会在命令行中提供三个命令选项：start/stop/exit。输入 start 命令，会修改当前捕捉状态为 start，并调用捕捉线程的 sniff() 函数。该函数获取到当前的捕获状态为 Start，调用 pcap_dispatch(pcap_t *p, int cnt,pcap_handler callback, u_char *user) 函数，捕捉一定量的数据包。每当抓获一个数据包后，通过函数指针调用回调函数 get_packet()，解析数据包的捕获时间、序号、协议类型、长度等信息，并将捕获到的数据包实时存储到指定的存储文件中。输入 stop 命令，会调用 sniffer.stopSniffing() 函数，修改当前捕捉状态为 stop，使抓包进程停止。输入 exit 命令，会调用 sniffer.stop() 函数，程序会关闭抓包进程，并退出。此时，抓取到的数据包会被以.pcap 文件的格式保存到数据路径中 (data/traffic.pcap)。CLI 模式下的模块交互设计如图 5所示。

3.3.2 包过滤

GUI 模式中可对抓取到的数据包进行过滤选择。on_filter_textChanged() 函数调用 checkCommand() 对 filter 输入框中的文本进行实时检查，若满足过滤规则显示为绿色，若不满足过滤规则显示为红色。当按下 Apply 按键时，调用 on_filter_Pressed() 函数，根据输入的规则进行过滤；launchOneFilter() 对单个包进行判断，若满足规则则将其显示到 GUI 中，若不满足则不显示。

过滤规则有：

1. `-p protocal(TCP、UDP、ICMP、IGMP、ARP、IPv4、IPv6)`
2. `-s SourceIP(xxx.xxx.xxx.xxx)`
3. `-d DestinationIP(xxx.xxx.xxx.xxx)`
4. `-sport sourcePort(port number)`

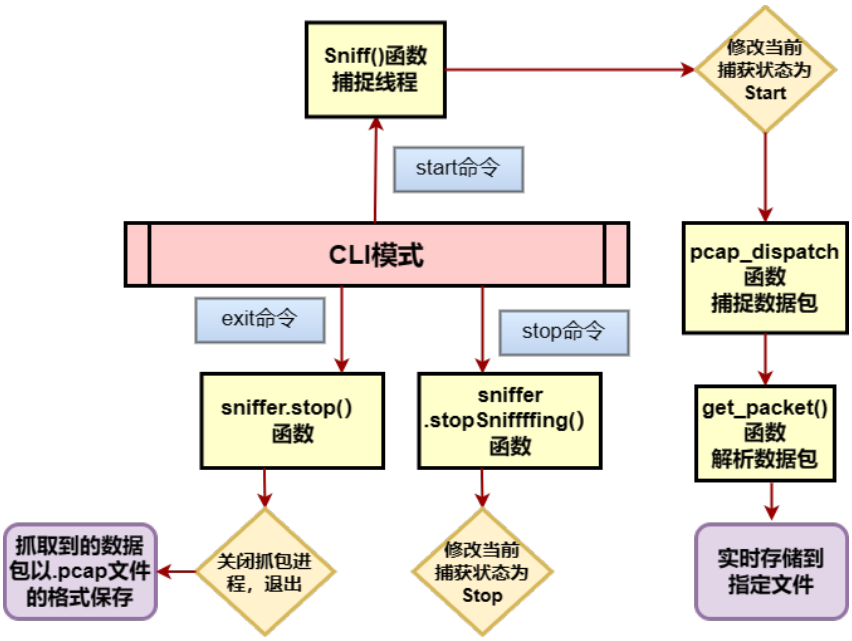

Figure 5: GUI 模式模块交互

5. `-dport destinationPort(port number)`

需按照上述规则的格式在命令行中输入命令进行包过滤。

3.3.3 包查询

包查询与包过滤实现方法相同，并且在 GUI 中共用一个输入框。若要根据数据包内容进行查询，可使用规则"-c content(content you want to search)"，会将含有对应 content 的数据包筛选出来。

3.3.4 IP 分片重组

当路由器受到一个数据包时，它会检查目的地址、接出口使用以及该分组的大小。如果分组的大小比 MTU 大，且该分组的 DF 位不为 1，则路由器可分片传送该数据包。

1. **确定需进行 IP 重组的数据包**：对于选中的数据包进行分析，若满足：协议类型为 IPv4 且 DF 位为 0，则为分片后的数据包。记录其 ip_id，由同一数据包的各个分片的 ID 相同这一性质，在装有所有捕获的数据包的向量 pkt 中查找与其 ip_id 相同的数据包，并将满足条件的数据包放入新建的向量 repkt 中。最终 repkt 中的数据包即为和选中的数据包同属一个数据包的各分片。
2. **将各分片的数据包进行重组**：根据 IP 分片及重组的原理，重组方法为：根据 Offset 确定各个数据包的顺序，重组数据包的报头为第一个分片 (Offset=0) 的报头，数据部分为各个分片的数据组合。根据此方法，对 repkt 中的所有数据包进行重组，最终得到重组后的数据包，并将重组后的数据包内容显示在 GUI 中。

3.4 嗅探结果

我们将 WireCat 抓取的数据包保存为.pcap 文件格式。Pcap 是 Packet Capture 的英文缩写，是一种行业标准的网络数据包捕获格式。Pcap 文件格式是一种二进制格式，支持纳秒级精度的时间戳，Wireshark、Tcpdump 或 WinDump 等主流网络分析器捕获 TCP/IP 数据包后存盘的文件格式就是.pcap 文件。以.pcap 格式存储保证了我们的系统与主流网络分析系统的可交互性。

此外，我们使用 tshark 工具将 pcap 文件进一步转换为 json 文件，方便后续流量分析过程的进行。tshark 是网络分析工具 wireshark 下的一个工具，主要用于对抓取的数据包进行分

析，尤其对协议深层解析时，tcpdump 难以胜任的场景中。其可在命令行运行，简单高效，且便于集成到自动化运行脚本中。

4 基于 MinIO 的数据存储

4.1 MinIO 简介

MinIO 是一个开源的对象存储服务器软件，它兼容 Amazon S3 协议，可用于构建和部署私有云存储解决方案。MinIO 旨在提供高性能、可扩展和易于使用的对象存储服务。它具有如下优势：

- **高可扩展性**：MinIO 可以轻松地扩展存储容量和处理能力，通过添加更多的 MinIO 节点来满足不断增长的数据包存储需求。这使得我们可以处理大规模的网络流量数据包集合，包括大量的文件和高并发访问。
- **高性能**：MinIO 专注于对象存储，并针对大型文件和海量数据进行了优化。它使用并行处理和分布式架构，以实现高吞吐量和低延迟的数据访问，有助于更高效地处理和分析大量的网络流量数据包。

不同于传统的关系型数据库，需要将数据拆分为表格形式的行和列进行存储，MinIO 以对象为基本单位进行管理和存储，对象数据可以是任意类型的二进制数据，例如图像、视频、文档等。因此，MinIO 很适合存储大文件和非结构化数据，很适合我们的流量分析与审计系统的应用场景。

4.2 数据存储的实现

4.2.1 MinIO 的 Python API

MinIO 的 Python API 是一个用于与 MinIO 服务器进行交互的 Python 软件包，它提供了丰富的功能来管理和操作存储在 MinIO 服务器上的对象，包括：

```
#初始化Minio客户端对象:
Minio(endpoint,access_key,secret_key,secure)
#创建新的存储桶:
make_bucket(bucket_name)
#删除指定的存储桶:
remove_bucket(bucket_name)
#列出所有存储桶:
list_buckets()
#检查指定的存储桶是否存在:
bucket_exists(bucket_name)
#将本地文件上传为对象到指定的存储桶:
fput_object(bucket_name, object_name, file_path)
#从指定的存储桶下载对象到本地文件:
fget_object(bucket_name, object_name, file_path)
#删除指定存储桶中的对象:
remove_object(bucket_name, object_name)
```

4.2.2 上传数据

MinIO 用于存储嗅探器抓取到的 traffic.pcap 文件、traffic.pcap 文件转换成的 traffic.json 文件、机器学习模型的特征 extractor.txt 文件、机器学习模型生成的标签 label.txt 文件，在./run.sh 脚本运行时，会调取 minio/upload.py 将上述数据上传至数据库。minio/upload.py 的部分代码如下：

```
# Determine if a bucket exists, create if it does not
# bucket pcap
if not minio_client.bucket_exists("pcap"):
    minio_client.make_bucket("pcap")
# bucket json
```

```python
if not minio_client.bucket_exists("json"):
    minio_client.make_bucket("json")
# bucket label
if not minio_client.bucket_exists("label"):
    minio_client.make_bucket("label")
# bucket extractor
if not minio_client.bucket_exists("extractor"):
    minio_client.make_bucket("extractor")

# Upload file to minio
# Upload pcap file with time as file name
minio_client.fput_object(bucket_name="pcap", object_name="pcap"+time_str, file_path=pcap_path)
# Upload json file with time as file name
minio_client.fput_object(bucket_name="json", object_name="json"+time_str, file_path=json_path)
# Upload the label file with the time as the file name
minio_client.fput_object(bucket_name="label", object_name="label"+time_str, file_path=label_path)
# Upload extractor file with time as file name
minio_client.fput_object(bucket_name="extractor", object_name="extractor"+time_str, file_path=extractor_path)
```

实现逻辑如下：

1) 首先，通过预先配置好的用户名和密码连接至 minio 服务器
2) 接着，判断 minio 中是否存在 pcap、json、label、extractor 四个桶，若不存在则创建相应的桶
3) 调用相关函数，将本地文件上传至数据库的对应桶中

4.2.3 数据的查看与管理

MinIO 在部署时，可指定控制台的服务端口，在浏览器访问对应端口，可实现对数据的查看与管理，本项目中的数据对象采用生成时的时间戳命名，图 6展示的是控制台中 label 桶的情况。

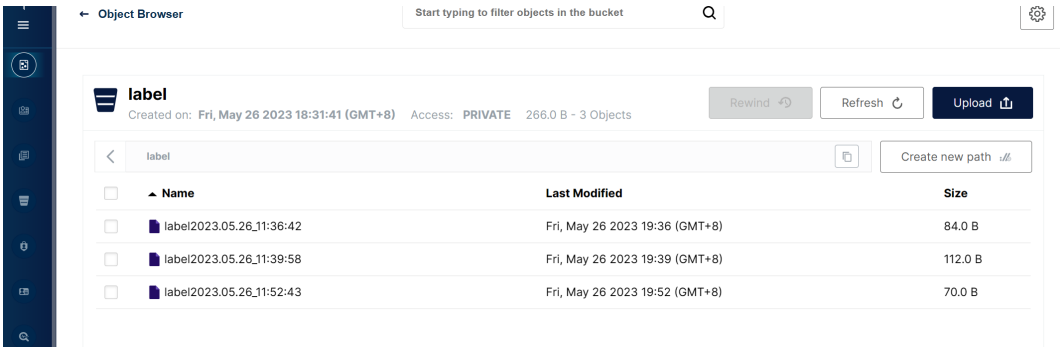

Figure 6: 控制台中 label 桶的情况

我们也提供了 minio/clean.py 脚本用于数据的清除，运行它将清除当前数据库存储的所有数据，部分代码如下：

```python
# Delete the file
# Delete the pcap file
if minio_client.bucket_exists("pcap"):
    for file in minio_client.list_objects("pcap"):
        minio_client.remove_object("pcap", file.object_name)
# Delete the json file
if minio_client.bucket_exists("json"):
```

```
    for file in minio_client.list_objects("json"):
        minio_client.remove_object("json", file.object_name)
# Delete the label file
if minio_client.bucket_exists("label"):
    for file in minio_client.list_objects("label"):
        minio_client.remove_object("label", file.object_name)
# Delete the extractor file
if minio_client.bucket_exists("extractor"):
    for file in minio_client.list_objects("extractor"):
        minio_client.remove_object("extractor", file.object_name)
```

4.2.4 数据的下载与合并

MinIO 的 json、label、extractor 桶中存储了后续 Dash Board 需要呈现给用户的关键信息，我们提供了将它们下载到本地与合并的功能，方便用户对全部数据的统一审计。该功能由 minio/download.py 实现，部分代码如下，这部分代码实现了 label 数据的下载与合并：

```
# Download all objects in the label bucket locally and merge them into one file
file_name = "minio/label_tmp/label-all.txt"
open(file_name, "w")
for label_object in label_object_list:
    minio_client.fget_object(
        bucket_name="label", object_name=label_object.object_name, file_path=label_folder  + label_object.object_name
    )
    with open(file_name, "ab") as f:
        with open(label_folder + label_object.object_name, "rb") as f1:
            f.write(f1.read())
    # print("download label file: "+label_object.object_name)

# Overwrite the original data/label.txt with label-all.txt
os.rename(file_name, "data/label.txt")

# Emptying the label_tmp folder
for file in os.listdir(label_folder):
    os.remove(label_folder + file)
```

实现逻辑如下：

1) 通过预先配置好的用户名和密码连接至 minio 服务器
2) 在本地创建临时文件 minio/label-tmp/label-all.txt
3) 对于 label 桶中的每个对象，将它下载到本地临时文件夹，并将内容写入上面的临时文件
4) 用合并完成的临时文件覆盖 Dash Board 将调取的 data/label.txt 文件，并清空临时文件夹

通过对象存储库 MinIO，我们实现了网络流量分析数据的高效组织与存储，有效提升了关键数据的可靠性。

5 网络流量特征提取与数据处理

该部分承接网络嗅探器的功能，旨在从网络嗅探器抓取的 pcap 格式数据包中提取 KDD Cup99 数据集所对应的特征属性并进行预处理和编码。

KDD Cup99 的原始数据为 7 周的网络流量，这些网络流量包含有约 500 万条网络连接; 实验的测试数据为 2 周的网络流量，包含有约 200 万条网络连接，因此，KDD Cup99 是网络入侵检测领域的事实 Benckmark，为基于计算智能的网络入侵检测研究奠定基础。

数据集中每个网络连接被标记为正常（normal）或异常（attack），异常类型被细分为 4 大类共 39 种攻击类型，其中 22 种攻击类型出现在训练集中，另有 17 种未知攻击类型出现在测试集中，这样设计的目的是检验分类器模型的泛化能力。训练集的 22 种恶意流量见下一节所述。

KDD Cup99 数据集中每个连接记录包含了以下特征：

- TCP 连接基本特征（共 9 种）：该部分包含了一些连接的基本属性，如连续时间，协议类型，传送的字节数等。
- 基于时间的网络流量统计特征（共 9 种）：该部分统计当前连接记录与之前一段时间内的连接记录之间存在的联系，可以更好的反映连接之间的关系。这类特征又分为两种集合：一个是“same host”特征，只观察在过去两秒内与当前连接有相同目标主机的连接，例如相同的连接数，在这些相同连接与当前连接有相同的服务的连接等等；另一个是“same service”特征，只观察过去两秒内与当前连接有相同服务的连接，例如这样的连接有多少个，其中有多少出现 SYN 错误或者 REJ 错误。
- 基于主机的网络流量统计特征（共 10 种）：基于时间的流量统计只是在过去两秒的范围内统计与当前连接之间的关系，而在实际入侵中，有些攻击使用慢速攻击模式来扫描主机或端口，当它们扫描的频率大于 2 秒的时候，基于时间的统计方法就无法从数据中找到关联。所以增加基于主机的网络流量统计特征，该部分按照目标主机进行分类，使用一个具有 100 个连接的时间窗，统计当前连接之前 100 个连接记录中与当前连接具有相同目标主机的统计信息。

由于 KDD Cup99 数据集可以提供大量的真实数据，贴近真实生活，并且便于进行深度学习，训练良好的模型，所以我们决定从嗅探器抓到的网络流量包中提取该数据集对应的 28 中特征，进行下一步的模型训练。

5.1 从 pacp 文件提取 KDD99 数据集特征

由于从 pcap 文件中提取诸如过去两秒内，在与当前连接具有相同目标主机的连接中，出现“SYN”错误的连接的百分比之类的特征耗时较长，难以在两周内完成，所以我们借鉴了 Github 上的开源项目 `https://github.com/AI-IDS/kdd99_feature_extractor.git`，并在其基础上进行了修改，实现的功能有：

1) 从 sniffer 抓到的 pcap 文件中提取 28 个 kdd99 数据集特征
2) 将提取到的特征保存到 extractor.txt 中，以便后续的数据处理

上述功能通过命令 sudo kdd/kdd99extractor > data/ectractor.txt 执行，该部分的具体实现如下：

5.1.1 读取 pcap 文件

该部分的实现代码如下：

```
try {
    Config config;
    parse_args(argc, argv, &config);

    if (config.get_files_count() == 0) {
      // Input from interface
      int inum = config.get_interface_num();
      if (config.should_print_filename())
        cout << "INTERFACE " << inum << endl;
      Sniffer *sniffer = new Sniffer(inum, &config);
      extract(sniffer, &config, true);
    }
    else {
      // Input from files
      int count = config.get_files_count();
      char **files = config.get_files_values();
      for (int i = 0; i < count; i++) {
```

```
        if (config.should_print_filename())
          cout << "FILE '" << files[i] << "'" << endl;

        Sniffer *sniffer = new Sniffer(files[i], &config);
        extract(sniffer, &config, false);
      }
    }
  }
```

实现逻辑如下：

1) 首先，创建一个 Config 对象 config，用于存储配置信息；并调用 parse_args 函数，解析命令行参数，并将配置信息存储到 config 对象中。
2) 接着，检查是否存在输入文件，当存在输入文件时，获取文件数量 count 和文件名数组 files
3) 调用 extract(sniffer, &config, false) 函数，处理数据。

5.1.2 提取基本特征

kdd99 的基本特征包括 service（目标主机的网络服务类型），protocol_type（协议类型）等，这些特征的特点是可以直接从数据包中读取，所以特征提取较为简单。以 service 为例，介绍该部分特征提取的实现逻辑：

```
const char *Conversation::get_service_str() const
  {
    // Ensure size of strins matches number of values for enum at compilation time
#ifdef static_assert
    static_assert(sizeof(Conversation::SERVICE_NAMES) / sizeof(char *) == NUMBER_OF_SERVICES,
      "Mapping of services to strings failed: number of string does not match number of values");
#endif

    return SERVICE_NAMES[get_service()];
  }
```

该函数从 SERVICE_NAMES 数组（一个包含所有服务类型的数组）中取出对应的服务名称字符串；并判断 get_service()(从数据包中读取相应字符的函数）所读取的字符是否在数组中，确保返回合法的服务类型。

5.1.3 提取统计特征

统计特征一般需要计数或计算得出，如 count（过去两秒内，与当前连接具有相同的目标主机的连接数），srv_count（过去两秒内，与当前连接具有相同服务的连接数），以 srv_diff_host_rate 为例，分析其实现逻辑：

```
double ConversationFeatures::get_srv_diff_host_rate() const {
    return (srv_count == 0) ? 0.0 : ((srv_count - same_srv_count) / (double)srv_count);
  }
```

如果 srv_count 的值为零，则返回 0.0，以避免除以零的错误。否则，计算 (srv_count - same_srv_count) / (double)srv_count 的结果，其中：

- srv_count 是服务的总数。
- same_srv_count 是具有相同服务的主机对的数量。
- (double)srv_count 将 srv_count 转换为 double 类型，以确保结果为浮点数。

最终，函数返回服务的不同主机比率，表示具有不同服务的主机对占服务总数的比例。

5.2 数据预处理与编码

5.2.1 数据处理

kdd99_feature_extractor 将提取到的特征保存到 extractor.txt 后，要对每一列分别标记，并滤掉不在 kdd99 数据集的特征，具体代码如下：

```
def read_data(path):
    data = pd.read_csv(path, header=None)
    data.columns = [
        "duration",  # 持续时间，范围是 [0, 58329]
        "protocol_type",  # 协议类型，三种：TCP, UDP, ICMP
        "service",  # 目标主机的网络服务类型，共有70种，如 'http_443 ,'imap4 等
        ...
    ]

    #删除非kdd99特征

    valid_services = ['uucp_path', 'uucp', 'supdup', 'whois', 'nnsp', 'pm_dump', 'tftp_u', 'tim_i', 'netbios_ssn', 'link', ...]

    data = data[data["service"].isin(valid_services)]

    ...
    # 对每一项数据进行编码或正则化
    encode_numeric_zscore(data, "duration")
    encode_text_dummy(data, "protocol_type")
    ...

    return data.values
```

5.2.2 特征编码

对于数值型特征，我们对其进行 Z-score 标准化，将其平均值和标准差归一化，实现代码如下：

```
def encode_numeric_zscore(df, name, mean=None, sd=None):
    if mean is None:
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd
```

对于数值型特征，使用该特征列的样本均值和样本标准差进行标准化：先减去均值并除以标准差。标准化后的值具有均值为 0，标准差为 1 的标准正态分布。最后，更新原始的 DataFrame 中的相应列，将标准化后的值存储到 df[name] 列中。

对于分类型特征，我们在设计之初采用了独热编码，但是由于部分特征（例如"service"）的类别较多，达 70 种，所以在模型预测时实际用于预测的数据无法覆盖全部类别，导致了模型对于 kdd99extractor 提取出的特征无法预测；为了解决这一问题，我们采用了标签编码 (Label Encoding)，它将每个类别映射到一个整数值，相比独热编码，标签编码不会增加数据维度，因为它只使用一个整数列来表示类别特征。实现代码如下：

```
def encode_text_dummy(df, name, encoder=None):
    if encoder is None:
```

```
        encoder = LabelEncoder()
        encoder.fit(df[name])
        with open("./encoder/" + name + "_encoder.pkl", "wb") as f:
            pickle.dump(encoder, f)
    else:
        encoder.classes_ = np.load(encoder + ".npy")
    df[name] = encoder.transform(df[name])
```

对于某个分类特征列，使用 LabelEncoder 将类别映射为整数标签，并将编码器对象保存为 pickle 文件（如果是新的编码器对象）。并且在编码后，将整数标签值更新到原始的 DataFrame 中的相应列。

6 网络流量分析机器学习模型

我们团队使用 KDD Cup99 数据集[1]训练了一个针对入侵检测和攻击识别的机器学习模型。使用该模型，可以从网络流量中识别出恶意的攻击流量，并识别出具体的攻击类型。可以识别的流量类型如表 1所示。

6.1 模型训练

由于 kdd99 的原始数据集规模较大，特征较多。因此，我们通过分布式架构训练我们的模型，将计算任务分布在多个计算节点上进行并行计算，充分利用集群或分布式系统的计算资源，提高训练速度和效率。我们使用开源的无服务器平台 OpenWhisk 搭建了分布式训练框架，具体的训练流程如下：

1. 首先根据 worker 的数量，将训练集平均分成几份，分配给每一个 worker；
2. 每个 worker 在完成自己这部分数据集的训练之后，将模型的参数也平均分为几部分，上传到远程的数据库中；
3. 之后，每个 worker 下载一部分的模型参数，将这一部分模型参数聚合，而其他部分的模型参数由其他的 worker 聚合；
4. 每个 worker 在完成聚合之后，将自己负责聚合的模型参数上传到远程数据库；
5. 每个 worker 下载其他 worker 负责聚合的模型参数，更新自己的模型。

训练流程可以表示成图 7的形式

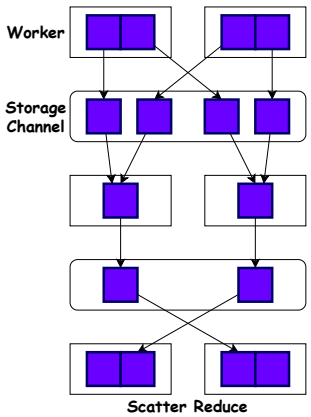

Figure 7: 分布式训练流程

在对数据进行处理和编码后，基于 PyTorch 的 nn 库，我们构建了自己的神经网络，由 5 个全连接层组成，如图 8所示。该模型的具体构成如下：

[1] https://www.kaggle.com/datasets/galaxyh/kdd-cup-1999-data

- 输入层：28 个神经元，对应输入数据的 28 个特征；
- 隐藏层 1：64 个神经元，使用 ELU 激活函数；
- 隐藏层 2：64 个神经元，使用 ELU 激活函数；
- 隐藏层 3：32 个神经元，使用 ELU 激活函数；
- 隐藏层 4：16 个神经元，使用 ELU 激活函数；
- 输出层：23 个神经元，对应 23 个类别的概率输出；

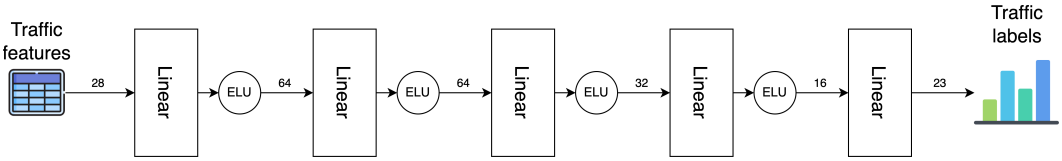

Figure 8: NTML 模型架构

我们团队将 KDD Cup99 数据集划分为训练集和测试集，采用 20 个 epoch，每个 batch 的大小为 128，并在每个 epoch 结束后评估模型的 loss 和准确率。在 20 轮的训练过后，我们的模型对于给定的流量分析的准确率可以达到 99.75% 以上。

为了提高模型训练的效率，我们团队采用了分布式的训练方法。

6.2 模型分析

在分析阶段，我们首先导入训练好的模型，然后对待预测的数据做和训练阶段相同的预处理操作，最后使用模型输出预测的结果。

Table 1: 模型可以识别的流量类型

流量类型	具体攻击类型
NORMAL	
PROBE	ipsweep, mscan, nmap, portsweep, saint, satan
DOS	apache, back, land, mailbomb, neptune, pod, processtable, smurf, teardrop, udpstorm
U2R	buffer_overflow, httptunnel, loadmodule, perl, ps, rootkit, sqlattack, xterm
R2L	ftp_write, guess_passwd, imap, multihop, named, phf, sendmail, snmpgetattack, snmpguess, spy, warezclient, warezmaster, worm, xlock, xsnoop

7 Dash Board 网页设计

我们的系统通过网页以统计信息和图表的形式呈现流量分析的结果，使用户能够直观地浏览和查看数据。在构建 Dash Board 网页的过程中，我们团队采用了开源的 app 框架 Streamlit[2]。Streamlit 框架非常适合展示数据和图表，能够帮助我们创建交互式的、美观的用户界面，适合我们的项目需求。

我们系统的前置模块负责将捕获的网络流量包解析为层次结构，并为每个流量包添加相应的标签。基于解析和分类的结果，我们的 Dash Board 网页将呈现以下内容：

- 流量时序图：如图 9所示，以时间为横坐标，展示流量的变化趋势和波动情况。这可以帮助用户了解被监测的计算机系统的流量时序特征，判断流量高峰，识别异常事件和分析流量的周期性模式。该图表可以直观地展示流量的变化情况，使用户能够快速把握流量的整体趋势和关键特征。

[2] `https://streamlit.io/`

- 流量数据表：如图 10所示，以表格的形式呈现捕获的流量数据，包括时间戳、源 IP 地址、目的 IP 地址、协议类型、包大小等信息。用户可以通过该数据表进行快速的筛选、排序和过滤，以便深入分析特定时间段或特定属性的流量数据。用户还可以在输入框中输入流量的索引序号，查看该流量的详细层次结构内容，如图 11所示。
- 协议计数柱状图：如图 12所示，统计各个协议在流量中的出现次数，并以柱状图的形式展示。该图表可以帮助用户了解不同协议的流量分布情况，识别主要协议和潜在的异常协议行为。
- 源 IP 与目的 IP 地址计数饼状图：如图 13所示，统计流量中不同源 IP 地址和目的 IP 地址的出现次数，并以饼状图的形式展示。通过该图表，用户可以快速了解流量的来源和目的地分布情况，发现与特定 IP 地址相关的流量模式或异常行为。
- 流量包大小饼状图：如图 14所示，将流量按照包大小进行分组统计，并以饼状图的形式展示不同大小的流量比例。这有助于用户了解流量的大小分布情况，判断流量中是否存在异常的特大或特小包。这些异常的包可能是由于网络拥塞、恶意攻击、数据损坏或传输错误等因素引起的。通过观察该图表，用户可以快速识别异常大小的流量包，并进一步分析其产生原因，采取相应的措施来优化网络性能和确保数据的安全传输。
- 端口计数饼状图：如图 14所示，统计流量中不同端口号的出现次数，并以饼状图的形式展示。该图表可以帮助用户了解不同端口的流量分布情况，发现哪些端口被频繁连接，哪些不常用端口被连接，并揭示与特定端口相关的流量模式或异常行为。通过观察该图表，用户可以迅速识别流量集中的热门端口和流量分散的冷门端口。此外，对于特定端口的异常流量活动，例如频繁连接或异常数据传输，用户可以进一步调查其背后的原因，识别潜在的安全风险或异常行为，并及时采取适当的措施加以应对。
- IP 拓扑关联图谱：如图 15所示，基于流量中的源 IP 和目的 IP 地址，绘制网络中不同 IP 之间的关联关系图。通过该图谱，用户可以直观地了解网络中不同 IP 之间的通信情况和连接模式，有助于发现潜在的网络异常或异常流量行为。
- 流量标签分布饼状图：如图 15所示，根据前置模块为流量包添加的标签，统计不同标签在流量中的分布比例，并以饼状图的形式展示。这有助于用户了解不同标签所代表的流量类型或行为的比例，识别特定类型的流量或关注特定标签的流量分布情况。

我们的 Dash Board 网页通过以上多种可视化方式，为用户提供了丰富的信息展示和分析工具，帮助用户深入理解网络流量的特征和行为，提供决策支持和网络安全监测的参考依据。

8 实际应用前景

基于以上内容，TrafficCat 作为一款基于机器学习的网络流量分析审计系统，由于其在设计时具有的完整性、结构性、开源性、简便性等特点，拥有广泛的应用前景。

首先，TrafficCat 有巨大的教学应用价值。由于本系统开放源代码，学生和研究者可以直接观察和理解网络流量分析的具体实现，从中学习网络协议、数据分析方法以及机器学习在网络流量分析中的应用，是理论教学与实践操作的极佳桥梁。同时，它也可以用于研究人员的实验环境，通过修改和调整其参数和算法，研究网络流量的特征和模式，探索更有效的网络流量分析技术。

其次，TrafficCat 的开源意义巨大。开源不仅意味着代码的公开，更代表了一种共享和合作的精神。TrafficCat 的开源将吸引更多的开发者和用户参与其中，通过用户反馈和开发者贡献，持续优化和完善系统功能，提升系统的稳定性和效能，推动国产网络流量分析技术的发展。

最后，TrafficCat 提供了丰富的二次开发和拓展可能性。基于 TrafficCat 清晰的系统架构和功能，开发者可以轻松地进行二次开发和功能拓展，如添加新的网络协议解析、开发新的数据可视化工具、引入新的机器学习模型等，以满足特定场景的需求。

9 团队协作方式

我们团队在开发本系统时采用了多种团队协作工具，确保我们的开发任务高效有序进行。

- Github: 我们的代码开发协作使用 Github 代码托管平台完成，项目地址为 `https://github.com/ysyszheng/TrafficCat`
- Overleaf: 我们的文档撰写使用 Overleaf 平台。项目地址为`https://www.overleaf.com/read/gbcktbykvjhj`
- 腾讯会议与小组自习室: 我们在开发本项目的过程中进行了多次讨论，线上通过腾讯会议进行，线下则利用小组自习室进行集体研讨。

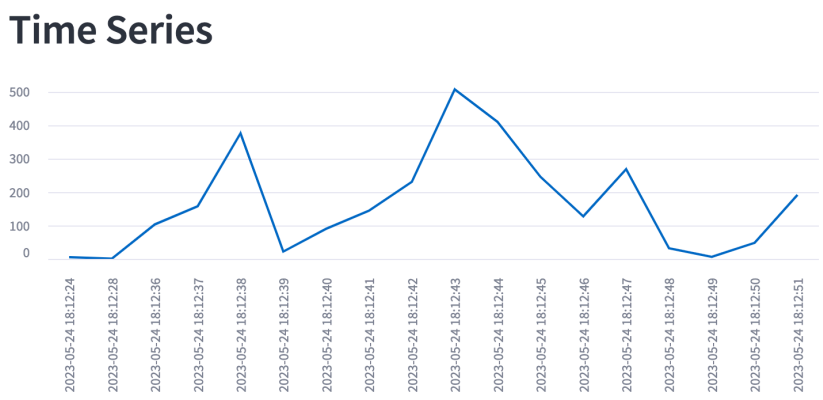

Figure 9: 时序图

Traffic Data

	time	proto	size	dst_ip	src_ip
2,196	2023-05-24 18:12:45+08:00	eth ethertype ip tcp tls	360	112.83.140.13	192.168.228
2,197	2023-05-24 18:12:45+08:00	eth ethertype ip tcp	60	192.168.228.128	112.83.140.
2,198	2023-05-24 18:12:45+08:00	eth ethertype ip tcp	60	192.168.228.128	112.83.140.
2,199	2023-05-24 18:12:45+08:00	eth ethertype ip tcp tls	1,721	101.69.198.54	192.168.228
2,200	2023-05-24 18:12:45+08:00	eth ethertype ip tcp	60	192.168.228.128	101.69.198.
2,201	2023-05-24 18:12:45+08:00	eth ethertype ip tcp	60	192.168.228.128	101.69.198.
2,202	2023-05-24 18:12:45+08:00	eth ethertype ip tcp tls	1,645	101.69.198.54	192.168.228
2,203	2023-05-24 18:12:45+08:00	eth ethertype ip tcp	60	192.168.228.128	101.69.198.
2,204	2023-05-24 18:12:45+08:00	eth ethertype ip tcp	60	192.168.228.128	101.69.198.
2,205	2023-05-24 18:12:45+08:00	eth ethertype ip tcp tls	1,650	101.69.198.54	192.168.228

Figure 10: 流量表

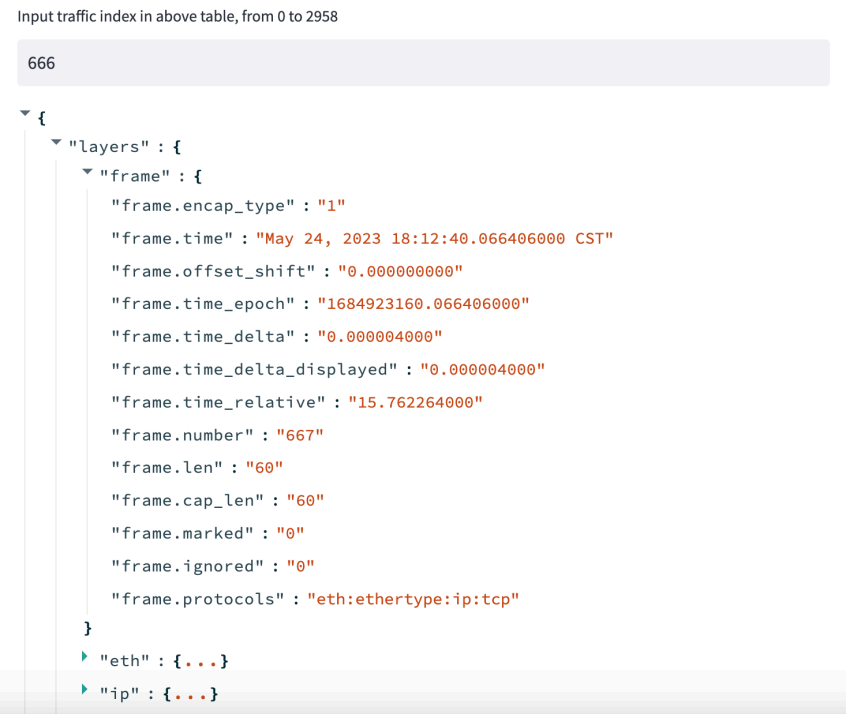

Figure 11: 流量细节分析

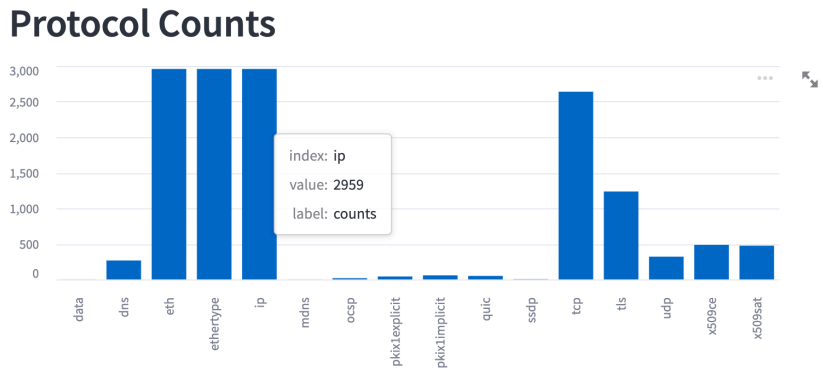

Figure 12: 协议计数

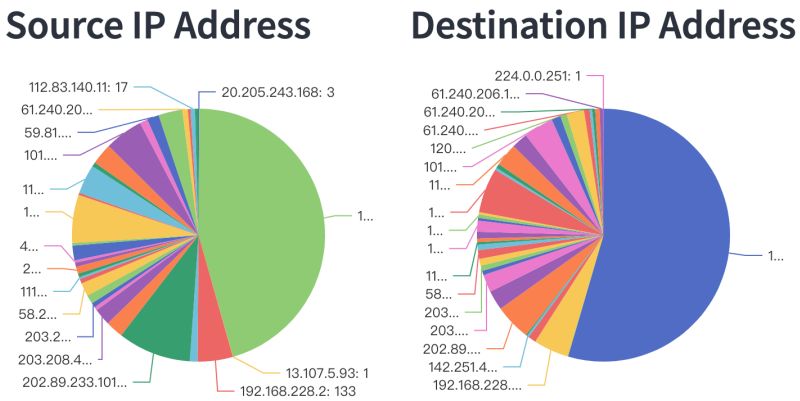

Figure 13: IP 地址计数

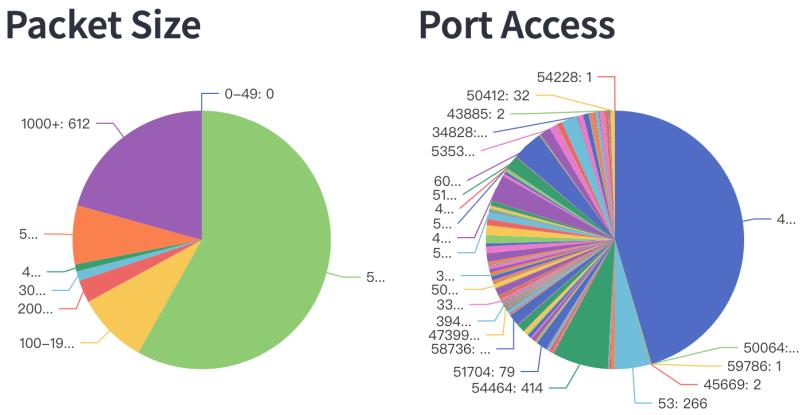

Figure 14: 包大小及端口计数

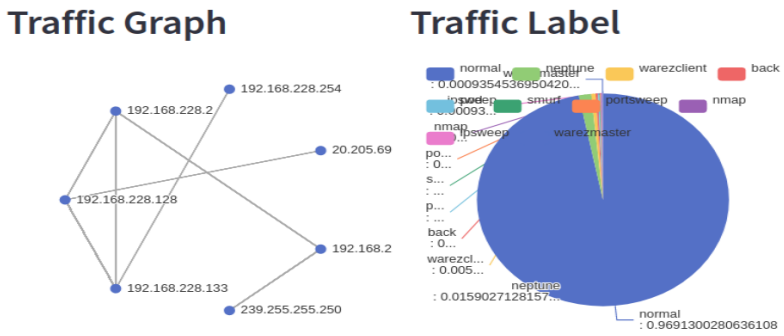

Figure 15: IP 拓扑图谱和流量标签