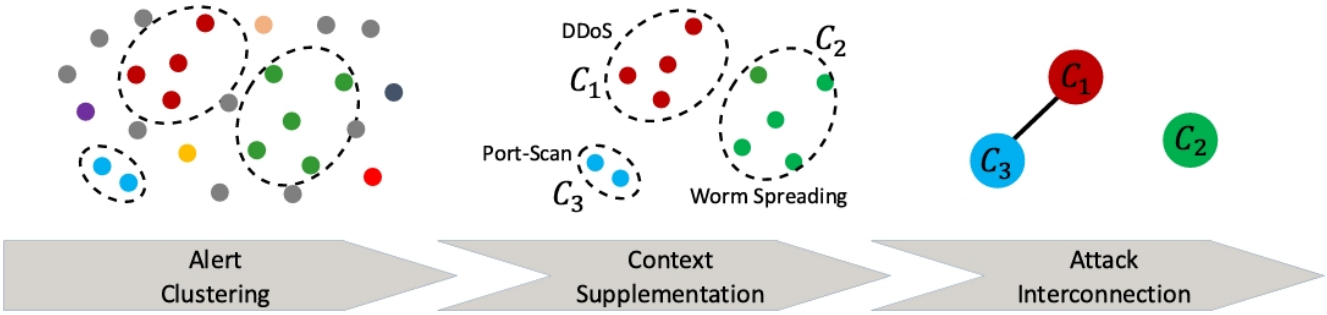

Figure 1: General alert correlation process with example for scenario labeling

图片说明

这张图片展示了一个 **通用的告警关联处理流程**，并以场景标注为例解释了各个步骤。这种方法主要用于网络安全告警的降噪和关联分析，目的是从大量告警数据中提取有意义的攻击信息，降低误报警率，并识别真正的安全威胁。

解决了什么问题

这种流程主要解决以下问题：

1. **告警过载**：面对大量告警（噪音多、信号少），需要自动化的方法筛选重要告警并关联相关事件。
2. **误报警率高**：通过聚类和上下文补充，可以过滤掉无意义的噪声告警。
3. **攻击识别不足**：基于上下文和攻击链分析，能够更准确地识别多阶段攻击行为。

步骤说明

（1）Alert Clustering（告警聚类）

- **目标**：根据告警的特征（如时间戳、源 IP、目的 IP、端口、协议、攻击类型、严重程度等）对告警进行聚类，识别出潜在的攻击模式或安全事件，降低告警的冗余度，提升分析效率。
- **技术**：特征工程
- **输出**：生成多个聚类组，每个聚类代表一个可能的攻击场景。

（2）Context Supplementation（上下文补充）

- **目标**：为每个聚类组补充背景知识（上下文），例如攻击模式或安全事件类型。
- **技术**：
  - 通过规则引擎或机器学习模型匹配模式。
  - 使用威胁情报库（Threat Intelligence）或历史数据标注攻击类型（如 DDoS、端口扫描等）。
- **输出**：将聚类映射为具体攻击场景，例如 `C1`（DDoS），`C2`（Worm Spreading），`C3`（Port Scan）。

（3）Attack Interconnection（攻击关联）

- **目标**：分析不同攻击场景之间的关联关系，构建攻击链，帮助安全团队全面了解攻击全貌，便于进一步调查和响应。
- **技术**：
  - 基于因果关系（例如攻击路径、源/目的 IP 共享）构建图模型。
  - 使用图分析算法（如图匹配、Pagerank）找出相关联的攻击。
- **输出**：关联告警场景，形成攻击链，便于进一步调查或应对。

总结

这张图描述了从原始告警数据到关联攻击链的处理流程。其关键意义在于：

1. **降低告警处理复杂度**：通过自动化手段减少误报警和告警数据量。
2. **增强攻击检测能力**：补充上下文信息和场景标注后，可以识别复杂的多阶段攻击。
3. **支持响应决策**：通过构建攻击链，帮助安全团队快速了解攻击全貌并采取行动。

特征工程

特征工程是机器学习中非常关键的一步，目的是将原始数据转换为合适算法模型的特征，来提高模型的性能和效果，在告警聚类中

特征工程主要包括：**数据预处理和特征选择与提取**

数据预处理

数据预处理的目的是清洗和整理原始数据，为后续的特征提取和模型训练做好准备

数据清洗

- 处理重复告警
  - 方法：检查数据集中的重复行，根据唯一标识符来识别重复的告警记录
  - 工具：使用python中的Pandas库中的drop_duplicates()方法
  - 示例：

```
import pandas as pd
#读取对应的数据集
df = pd.read_csv('alerts.csv')
#进行去重操作
df_cleaned = df.drop_duplicates(subset=['alert_id', 'timestamp', 'src_ip',
'dst_ip', 'alert_type']
```

- 处理缺失值
  - 方法：

    - 删除含有大量缺失值的记录：如果一条告警记录缺失了大部分重要信息，考虑将其删除。
    - 填充缺失值：使用平均值、中位数、众数或其他合理的值填充缺失的数据
    - 工具：Pandas库中的isnull()和fillna()方法。
    - 示例：

```
# 检查缺失值
missing_values = df.isnull().sum()
# 填充缺失值（例如，将缺失的端口号填充为-1）
df['port'].fillna(-1, inplace=True)
```

- 处理异常值（目的是识别出明显跟别的数据不一样的数据）
    - 方法：使用Z-score识别并处理异常值（后续有更好的选择可以进行替换）
    - 示例：

```
from scipy import stats

# 计算 Z-score
df['z_score'] = np.abs(stats.zscore(df['event_count']))

# 标记异常值
df['is_outlier_zscore'] = df['z_score'] > 3

print(df[['alert_id', 'event_count', 'z_score', 'is_outlier_zscore']])
```

数据标准化

- 目的：在告警数据中，不同特征的取值范围和量纲（我理解是一个计量单位）可能差异很大。例如：
    - 数值型特征：如告警次数可能在1到10的6次方之间
    - 时间特征：如响应时间可能在0.1秒到1000秒之间

  如果直接使用这些未处理的特征进行聚类，取值范围大的特征会对距离度量产生更大的影响（后续会用到基于距离度量的算法），导致模型偏向这些特征，影响聚类效果
- 方法：
    - 标准化：将特征转换为均值0，标准差为1的分布
    - 归一化：将特征缩放到[0,1]的范围
    - 示例：

```
# 初始化 StandardScaler
scaler_standard = StandardScaler()

# 对特征进行标准化
features = ['event_count', 'response_time']
df_standardized = scaler_standard.fit_transform(df[features])

# 转换为 DataFrame
```

```
df_standardized = pd.DataFrame(df_standardized, columns=['event_count_std',
'response_time_std'])

print("\n标准化后的数据：\n", df_standardized)

# 初始化 MinMaxScaler
scaler_minmax = MinMaxScaler()

# 对特征进行归一化
df_normalized = scaler_minmax.fit_transform(df[features])

# 转换为 DataFrame
df_normalized = pd.DataFrame(df_normalized, columns=['event_count_norm',
'response_time_norm'])

print("\n归一化后的数据：\n", df_normalized)
```

时间窗口划分

- 目的：考虑告警的时许特征，捕捉在特定时间段内发生的告警模式
- 方法：
  - 固定时间窗口：将时间划分为固定长度的窗口（如5分钟、1小时），将该时间段内的告警聚集在一起。
  - 滑动窗口：使用固定长度的窗口，在时间轴上以固定的步长滑动，每个窗口可能包含重叠的时间段
- 工具：使用Pandas的时间序列处理功能
- 关于固定时间窗口的示例

```
import pandas as pd

data = {
    'alert_id': [1, 2, 3, 4, 5, 6],
    'timestamp': [
        '2023-10-01 10:05:00',
        '2023-10-01 10:07:00',
        '2023-10-01 10:15:00',
        '2023-10-01 10:20:00',
        '2023-10-01 10:35:00',
        '2023-10-01 10:50:00'
    ],
    'src_ip': ['192.168.1.1', '192.168.1.2', '192.168.1.1', '192.168.1.3',
'192.168.1.2', '192.168.1.4'],
    'dst_ip': ['10.0.0.1', '10.0.0.2', '10.0.0.3', '10.0.0.4', '10.0.0.2',
'10.0.0.5'],
    'alert_type': ['Port Scan', 'DDoS', 'Port Scan', 'Malware', 'DDoS', 'Port Scan']
}

df = pd.DataFrame(data)
df['timestamp'] = pd.to_datetime(df['timestamp'])
# 设置 'timestamp' 为索引
df.set_index('timestamp', inplace=True)
```

```
22
23  # 以15分钟为时间窗口
24  fixed_window = df.resample('15T').agg({
25      'alert_id': 'count',
26      'alert_type': lambda x: x.mode()[0] if not x.empty else None
27  })
28
29  fixed_window.rename(columns={'alert_id': 'alert_count', 'alert_type':
    'most_common_alert_type'}, inplace=True)
30  print(fixed_window)
```

特征选择与提取

特征选择与提取旨在预处理后的数据中提取有用的信息，以便算法能够更好地学习数据的特征

类别型特征编码

- 目的：在机器学习模型中，大多数算法只能处理数值型数据，无法直接处理非数值的类别特征（如字符串、分类标签等）。因此，需要将这些类别型特征转换为数值型，以便模型能够理解和处理
- 方法：
  - 独热编码：为类别型特征中的每个可能取值创建一个新的二进制特征。对于每个样本，在对应的类别位置上标记为1，其余位置标记为0
- 工具：使用sklearn.preprocessing的OneHotEncoder
- 示例：

```
1   from sklearn.preprocessing import OneHotEncoder
2   import numpy as np
3
4   # 假设有一个类别特征：颜色
5   colors = np.array(['红', '绿', '蓝', '绿', '红', '蓝']).reshape(-1, 1)
6
7   # 初始化独热编码器
8   onehot_encoder = OneHotEncoder(sparse=False)
9
10  # 进行独热编码
11  encoded_colors = onehot_encoder.fit_transform(colors)
12
13  print("原始类别：", colors.flatten())
14  print("编码结果：\n", encoded_colors)
```

文本特征处理

- 目的：从自由文本字段（如告警描述、日志信息）中提取有价值的特征，使模型能够理解和利用文本信息进行训练和预测
- 方法：
  - TF-IDF（词频-逆文档频率）：是一种用于评估词语对于一份文档或一个文档集合的重要程度的统计方法。它广泛应用于信息检索、文本挖掘和自然语言处理等领域

        - TF（词频）：衡量一个词在文档中出现的频率
        - IDF（逆文档频率）：衡量词语在整个语料库中的普遍程度。词语越普遍，其IDF值越低；词语越稀有，其IDF值越高
- TF-IDF的核心思想是：
    - 提升那些在当前文档中出现频率，单在整个语料库中较少出现的词语的重要性。
    - 降低那鞋在所有文档中都常见的词语的重要性（如“的”,“是”,”和”等中文词）
- 工具：scikit-learn提供的TfidVectorizer
- 示例

```
from sklearn.feature_extraction.text import TfidfVectorizer

# 定义文档列表
documents = [
    "我 喜欢 看 电影",
    "我 不 喜欢 看 电视剧",
    "我 喜欢 看 动画片 和 电影"
]

# 初始化 TfidfVectorizer
vectorizer = TfidfVectorizer()

# 计算 TF-IDF 矩阵
tfidf_matrix = vectorizer.fit_transform(documents)

# 获取特征名称（词汇表）
feature_names = vectorizer.get_feature_names_out()

# 将 TF-IDF 矩阵转换为可读的形式
import pandas as pd
df_tfidf = pd.DataFrame(tfidf_matrix.toarray(), columns=feature_names)

print(df_tfidf)
```

特征降维

- 目的：在数据集中特征数量过多会导致计算复杂度高、模型过拟合等问题。特征降维的目的是减少特征数量，消除冗余信息，降低计算复杂度，同时尽可能保留原始数据的重要信息。
- 方法：
    - 主成分分析（PCA）：通过高维数据投影到低维空间，找到数据中心差最大的方向（主成分），这些方向彼此正交
- 示例

```
from sklearn.decomposition import PCA
import numpy as np

```

```
# 假设有高维特征矩阵X
X = np.random.rand(100, 10)  # 100个样本，10个特征

# 初始化PCA，指定降维后的维度
pca = PCA(n_components=2)

# 进行降维
X_reduced = pca.fit_transform(X)

print("原始维度：", X.shape)
print("降维后：", X_reduced.shape)
```

Context Supplementation（上下文补充）

目标

- 为每个聚类组补充背景知识（上下文）：在经过聚类分析后，我们得到了多个告警的聚类组。为了更好地理解这些聚类组代表的安全事件，需要为每个聚类组补充背景知识，识别其对应的攻击模式或安全事件类型

特征提取与描述符生成

- **目标**：为每个聚类组提供能够代表其特征的关键信息，生成用于后续匹配的描述符号。
- **技术与方法**：
  - **频率统计**：计算聚类组内各类告警的出现频率，如告警类型、源IP、目的IP、端口等。
  - **事件特征**：分析告警的时间分布特征，如告警发生的高峰期、持续时间、间隔等。
  - **数量特征**：统计告警数量、涉及的唯一IP数量、端口数量等。
- **文本特征提取**：
  - **关键词提取**：从告警描述中提取高频关键词
  - **主题模型**：使用LDA等方法，提取潜在的主题信息
  - **N-gram模型**：捕捉词序信息，提取有代表性的短语或句子片段
- **网络特征提取**：
  - **拓扑特征**：分析聚类组中告警涉及的网络节点和连接关系，构建网络拓扑图。
  - **关联关系**：识别聚类组内告警之间的关联关系，如共同的源IP、目的IP、相似的攻击手法等
- **工具与实现**：
  - **数据处理**：使用Pandas、NumPy进行数据统计和分析。
  - **文本处理**：使用 **scikit-learn** 的 feature_extraction.text 模块、**NLTK**、**jieba**（中文分词）等库进行文本特征提取。
  - **网络分析**：使用 **NetworkX** 库构建和分析网络拓扑。
- **示例**

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
import networkx as nx
```

```
4
5  # 假设 cluster_df 是某个聚类组的 DataFrame
6
7  # 1. 统计告警类型分布
8  alert_type_counts = cluster_df['alert_type'].value_counts()
9  print("告警类型分布: \n", alert_type_counts)
10
11 # 2. 提取高频关键词
12 vectorizer = TfidfVectorizer(max_features=10)  # 取前 10 个关键词
13 tfidf_matrix = vectorizer.fit_transform(cluster_df['alert_description'])
14 keywords = vectorizer.get_feature_names_out()
15 print("高频关键词: ", keywords)
16
17 # 3. 构建网络拓扑图
18 G = nx.from_pandas_edgelist(cluster_df, source='src_ip', target='dst_ip')
19 print("网络节点数: ", G.number_of_nodes())
20 print("网络边数: ", G.number_of_edges())
```

模式匹配与识别

- 目标:
  - 根据提取的特征，匹配已知的攻击模式，识别聚类组可能代表的安全事件类型
  - 简化安全分析流程，自动识别并分类安全事件，辅助安全人员快速响应
- 技术与方法:
  - 基于规则的匹配:
    - 规则引擎：使用预定义的规则，将聚类组的特征与已知攻击模式进行匹配
    - 规则定义：规则可以基于告警类型组合、IP行为模式、端口扫描特征、事件特征等
  - 基于机器学习的匹配
    - 分类模型：训练监督学习模型（**随机森林（Random Forest）：抗噪能力强，易于使用，梯度提升树（如 XGBoost、LightGBM）：高准确率，处理复杂特征，高效性， 支持向量机（SVM）：适合于高维数据**），根据聚类组的特征预测攻击类型
    - 特征向量化：将聚类组的统计和文本特征转换为模型可用的特征向量

威胁情报融合

- **目标**：利用外部威胁情报数据，丰富聚类组的上下文信息，提高攻击类型识别的准确性
- **情报查询**：将聚类组中的ip地址、域名等与威胁情报库进行比对，获取相关的威胁信息。
- **情报补充**：利用情报数据，进一步确认和细化攻击类型，丰富上下文

输出攻击场景

- **映射结果**：将每个聚类组标注为具体的攻击场景
- **生成报告**：汇总聚类特征和攻击类型，生成分析报告，供安全团队参考

Attack Interconnection（攻击关联）

第一步：数据准备与预处理

- **收集攻击场景数据**：
  - 汇总所有已识别的攻击场景（如 `C1`、`C2`、`C3`）及其对应的告警信息。
- **数据清洗**：
  - 处理缺失值、重复值和异常值，确保数据质量。
- **时间同步**：
  - 统一时间格式和时区，对告警事件进行时间序列排序。

技术：使用 **Pandas**、**NumPy** 等数据处理库进行数据清洗和整理。

第二步：构建关联关系

- **定义节点和边**：
  - **节点（Nodes）**：将攻击场景或告警事件视为图的节点。
  - **边（Edges）**：根据告警之间的关联关系建立连接。
- **建立关联规则**：
  - **共享属性关联**：如果两个攻击场景共享相同的源 IP、目的 IP、端口等，建立关联。
  - **时间关联**：在特定时间窗口内连续发生的攻击场景可能具有关联性。
  - **因果关系**：根据已知的攻击路径或攻击步骤，建立前后关联。

技术：制定关联规则，使用 **NetworkX** 等图处理库建立节点和边。

第三步：构建攻击关系图

- **创建图模型**：
  - 根据定义的节点和边，构建攻击关系图，表示各攻击场景之间的关联关系。

技术：使用 **NetworkX** 库创建图模型，管理节点和边的关系。

第四步：应用图分析算法

- **图遍历与搜索**：
  - **目的**：识别可能的攻击路径和关键节点。
  - **方法**：使用深度优先搜索（DFS）、广度优先搜索（BFS）等算法遍历图。
- **节点重要性评估**：
  - **目的**：评估攻击场景在整个攻击链中的重要程度。
  - **方法**：应用 **PageRank** 算法计算节点的权重。
- **社区检测**：
  - **目的**：发现图中高度关联的节点群体，可能代表协同攻击或复合攻击。

- **方法**：使用 **Louvain** 算法等社区发现算法。

技术：利用 **NetworkX** 提供的图算法功能进行分析。

第五步：构建和识别攻击链

- **攻击链生成**：
  - 根据图分析结果，连接相关的攻击场景，形成有序的攻击链。
- **路径优化**：
  - **目的**：找到最有可能的攻击路径，考虑路径长度、节点重要性等因素。
  - **方法**：使用最短路径算法（如 Dijkstra 算法）或自定义的路径评分机制。

技术：使用图算法库中的路径搜索和优化功能。

第六步：输出与可视化

- **攻击链展示**：
  - **图形化**：以直观的图形方式展示攻击链，突出关键节点和关联关系。
- **报告生成**：
  - **内容**：包括攻击链的详细信息、节点和边的属性、分析结论等。
  - **形式**：生成可阅读的报告文件，如 PDF、HTML 等。

技术：使用 **Matplotlib**、**Plotly** 等可视化库；使用 **Jinja2** 等模板引擎生成报告。