BUDOWA SYSTEMU ANALIZY SIECIOWEJ + POC

Matuszewski Kamil, Matuszewski Maciej

POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH

KRYCY

Spis treści

1. Wstęp

Nasze rozwiązanie zapewnia zarówno statystyczną analizę, jak i *live capture* poprzez program *live_capture.py*. Ponadto, pokryliśmy każdą funkcjonalność z wymagań (zarówno *Must-have*, jak i *Nice-to-have*). Dla wygody dowodzenia przykłady najczęściej będą tyczyły się analizy statycznej, ale udowodnimy także możliwość *live capture*.

2. Nasz system analizy sieciowej

2.1. Modułowa struktura

Program składa się z 5 głównych modułów:

- *ml_flows_analyzer.py* - moduł uczenia maszynowego
- *flows_analyzer.py* - główny moduł analizy
- *detection_rules.py* - moduł reguł detekcyjnych z użyciem *NFStream* i *Scapy*
- *raport_generator.py* - moduł generowania raportów
- *read_sigma.py* - moduł obsługi reguł Sigma

2.2. Reguły detekcyjne

Program zawiera 8 wbudowanych reguł detekcyjnych:

- *detect_large_flow* - wykrywa duże przepływy (>1M bajtów) na portach 80 i 443
- *asymmetrical_flow* - wykrywa asymetryczne przepływy (>1000 pakietów i <10 pakietów zwrotnych)
- *unusual_ports_flow* - wykrywa ruch na nietypowych portach (poza 80, 443, 22, 53)
- *find_DNS_users* - analizuje użycie DNS
- *detect_SYN_flood* - wykrywa ataki SYN flood
- *detect_http_get* - monitoruje żądania HTTP GET
- *detect_ping_flood* - wykrywa ataki ping flood (>1000 pakietów ICMP)
- *detect_ports_scanner* - wykrywa skanowanie portów (>10 unikalnych portów)

2.3. Reguły Sigma

Program obsługuje 3 reguły Sigma:

- *multiple_echo_request.yml* - wykrywa wiele zapytań ICMP Echo do różnych hostów (Próg: >5 różnych hostów w ciągu 60 sekund; Poziom: medium)
- *syn_flood.yml* - wykrywa ataki SYN flood (Próg: >100 pakietów SYN w ciągu 10 sekund; Poziom: high)
- *unusual_http_port.yml* - wykrywa ruch na nietypowych portach (Wykluczane porty: 80, 443, 22, 53; Próg: >5 połączeń; Poziom: medium)

2.4. Funkcjonalności ML

Klasyfikacja ruchu na podstawie cech:

- *bidirectional_packets*
- *bidirectional_bytes*
- *src2dst_packets/bytes*
- *dst2src_packets/bytes*
- *bidirectional_duration_ms*

2.4.1. Optymalizacja modelu:

GridSearchCV z parametrami:

- n_estimators: [100, 200]
- max_depth: [None, 10, 20]
- min_samples_split: [2, 5]
- min_samples_leaf: [1, 2]
- class_weight: ['balanced', 'balanced_subsample']

2.4.2. Możliwości trenowania:

- Tworzenie nowego modelu
- Doszkalanie istniejącego modelu
- Walidacja krzyżowa (5-krotna)

2.5. Funkcje wzbogacania danych (Enrichment)

- Sprawdzanie reputacji IP przez AbuseIPDB (Próg wykrycia: confidence score ≥ 30)
- Geolokalizacja podejrzanych IP przez ipinfo.io
- Generowanie mapy z Folium

2.6. Wizualizacje

Wykresy:

- Rozkład protokołów
- Statystyki portów źródłowych/docelowych
- Rozkład wielkości przepływów
- Wykresy słupkowe dla każdego typu detekcji
- Wykres kołowy proporcji ruchu normalnego/złośliwego
- Histogram pewności predykcji

Mapa:

- Interaktywna mapa Folium
- Znaczniki dla podejrzanych IP
- Popup z informacjami o IP

2.7. Raportowanie

Generowanie raportów tekstowych zawierających:

- Statystyki ogólne
- Szczegóły wykrytych zagrożeń
- Podsumowanie alertów
- Timestamp i informacje o analizowanym pliku

Katalogowanie wyników:

- Automatyczne tworzenie struktury katalogów
- Zapisywanie wykresów
- Zapisywanie mapy HTML
- Generowanie timestampów dla plików

2.8. Tryby działania

- Analiza plików PCAP
- Przechwytywanie na żywo:
  - Wybór interfejsu sieciowego
  - Filtrowanie po porcie
  - Określanie czasu przechwytywania

2.9. Dodatkowe funkcje

- Wykrywanie perspektywicznego IP - jakie urządzenie (o jakim IP) najprawdopodobniej było hostem na którym zbieraliśmy dane
- Możliwość wykluczania IP z analizy
- Automatyczne czyszczenie plików tymczasowych
- Obsługa błędów i logowanie
- Interfejs CLI z bogatymi opcjami konfiguracji

Po poniższych zrzutach ekranów prezentujących zbudowanie przyjemnego dla użytkownika interfejsu CLI z użyciem biblioteki *click* można zauważyć znaczący rozmiar naszego rozwiązania poruszającego wiele sposobów analizy flow.

```
(myvenv) maciej@DESKTOP-0GJ602Q:~/semester5/ml-flow-analysis/ml-flows-analysis$ python3 ml_flows_analyzer.py --help
Usage: ml_flows_analyzer.py [OPTIONS] COMMAND [ARGS]...

  Network Flow Classification Tool

Options:
  --help  Show this message and exit.

Commands:
  finetune  Finetune existing model with new data
  predict   Predict on new PCAP file using trained model
  train     Train a new model using normal and malicious PCAP files
(myvenv) maciej@DESKTOP-0GJ602Q:~/semester5/ml-flow-analysis/ml-flows-analysis$ python3 ml_flows_analyzer.py train --help
Usage: ml_flows_analyzer.py train [OPTIONS] NORMAL_PCAP MALICIOUS_PCAP

  Train a new model using normal and malicious PCAP files

Options:
  --model-output TEXT  Path to save the trained model
  --help               Show this message and exit.
(myvenv) maciej@DESKTOP-0GJ602Q:~/semester5/ml-flow-analysis/ml-flows-analysis$ python3 ml_flows_analyzer.py finetune --help
Usage: ml_flows_analyzer.py finetune [OPTIONS] NORMAL_PCAP MALICIOUS_PCAP
                                     MODEL_PATH

  Finetune existing model with new data

Options:
  --iterations INTEGER  Number of fine-tuning iterations
  --help                Show this message and exit.
(myvenv) maciej@DESKTOP-0GJ602Q:~/semester5/ml-flow-analysis/ml-flows-analysis$ python3 ml_flows_analyzer.py predict --help
Usage: ml_flows_analyzer.py predict [OPTIONS] PCAP_FILE MODEL_PATH

  Predict on new PCAP file using trained model

Options:
  --report-dir TEXT  Directory for saving reports
  --help             Show this message and exit.
```

Rysunek 1: ML-flows-analyzer z wyświetleniem wszystkich możliwych funkcjonalności wszystkich jego funkcji

```
Usage: flows_analyzer.py [OPTIONS] [PCAP_FILE]

  Comprehensive tool for analyzing PCAP files and live network traffic.

  Can be used in two modes: 1. PCAP file analysis: Provide a pcap file path 2.
  Live capture: Use --live flag with optional --interface and --port options

  For listing available network interfaces, use --list-interfaces

  For PCAP analysis, if you want to filter packets and exclude packets where
  your probable IP is source: 1. Use '--find-perspective-ip' first (or check
  your IP manually) 2. Use '--ip-to-remove <IP> --overall'

  Examples:

  # List available interfaces

  python flows_analyzer.py --list-interfaces

       # Capture from specific interface

  python flows_analyzer.py --live --interface eth0 --overall

       # Capture specific port

  python flows_analyzer.py --live --interface eth0 --port 80 --overall

       ATTENTION! Perspective IP may be wrong, because it simply checks what
      IP is the most frequent in pcap file packets. We highly suggest checking
      your IP manually.

Options:
  --live                  Enable live capture mode
  --list-interfaces       List available network interfaces
  --interface TEXT        Network interface to capture from (e.g., eth0)
  --port INTEGER          Port to capture on
  --duration INTEGER      Duration of capture in seconds
  --overall               Perform overall execution. This option will also
                          provide results charts and location map.
  --generate-data-charts  Generate charts from PCAP data.
  --get-reputation-ip     Check reputation from IPs.
  --generate-map          Generate map with IPs.
  --find-perspective-ip   Find the most likely perspective IP.
  --large-flow            Detect large network flows.
  --asymmetrical-flows    Detect asymmetrical flows.
  --unusual-ports         Detect unusual port usage.
  --dns-users             Find DNS users.
  --syn-flood             Detect potential SYN flood attacks.
  --http-get              Detect HTTP GET requests.
  --ping-flood            Detect Ping flood attacks.
  --ports-scanner         Detect port scanning.
  --ip-to-remove TEXT     Specify an IP to exclude from analysis.
  --chart BOOLEAN         Generate result charts.
  --sigma-analysis        Perform Sigma rules analysis.
  --help                  Show this message and exit.
```

Rysunek 2: flows-analyzer z wyświetleniem wszystkich możliwych funkcjonalności wszystkich jego funkcji

3. Analiza flow

3.1. A.1: Wczytywanie plików PCAP przy użyciu NFStream

Implementacja tego wymagania znajduje się w kilku miejscach kodu. Funkcjonalność ta znajduje wykorzystanie między innymi w *detection_rules.py*:

```python
# Function to find the perspective IP
def find_perspective_ip_nfstream(pcap):
    streamer = NFStreamer(source=pcap, statistical_analysis=True)
    ip_counts = {}
    for flow in streamer:
        ip_counts[flow.src_ip] = ip_counts.get(flow.src_ip, 0) + flow.bidirectional_packets
        ip_counts[flow.dst_ip] = ip_counts.get(flow.dst_ip, 0) + flow.bidirectional_packets
    if ip_counts:
        perspective_ip = max(ip_counts, key=ip_counts.get)
        print(f"Most likely perspective IP: {perspective_ip}")
        return perspective_ip
    print("No flows found.")
    return None
```

Rysunek 3: *find_perspective_ip_nfstream* - Przykładowa funkcja zawierająca wczytywanie plików PCAP z NFStream

Działanie kodu:

- Używa NFStreamer do analizy pliku PCAP
- Zlicza pakiety dwukierunkowe dla każdego IP (źródłowego i docelowego)
- Znajduje IP z największą liczbą pakietów używając funkcji max()
- Wyświetla znalezione IP jako „najbardziej prawdopodobne”
- Zwraca None jeśli nie znaleziono żadnych przepływów
- Służy do identyfikacji najbardziej aktywnego IP w ruchu sieciowym

Ta funkcja akurat nie ma roli detekcyjnej, ponieważ jej celem jest wykrywanie perspektywicznego IP, tj. jakie urządzenie (o jakim IP) najprawdopodobniej było hostem na którym zbieraliśmy dane. Oczywiście, zastosowanie NFStream mają także reguły detekcyjne, jak np. *detect_large_flow*.

```
(myvenv) maciej@DESKTOP-0GJ602Q:~/semester5/ml-flow-analysis/ml-flows-analysis$ python3 flows_analyzer.py --find-perspective-ip pcap-folder/normal_traffic.pcap
Finding perspective IP...
Most likely perspective IP: 192.168.1.112
```

Rysunek 4: Wynik wywołania *find_perspective_ip_nfstream* dla danego pliku PCAP

3.2. A.2: Wyświetlanie podsumowania statystyk flow

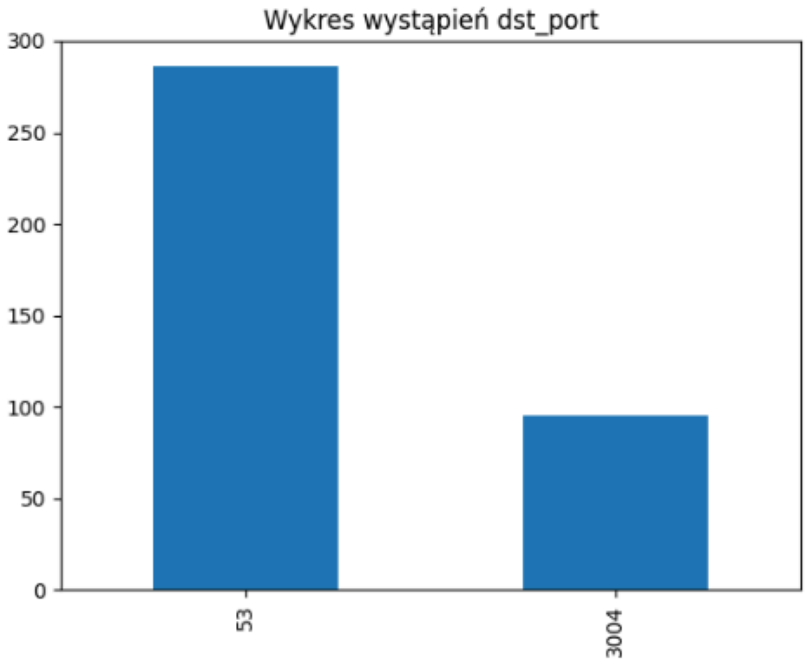
W pliku raport_generator.py zaimplementowano funkcję *charts*, która generuje statystyki dla kluczowych parametrów flow:

```python
# Raw data from flow into charts
def charts(pcap):
    base_name = os.path.basename(pcap)
    output_dir = create_dir(base_name, 'data')

    streamer = NFStreamer(source=pcap, statistical_analysis=True)
    data = streamer.to_pandas()

    # Remove columns with one unique value or missing data
    data = data.loc[:, data.nunique() > 1]

    columns = ['src_ip', 'dst_ip', 'protocol', 'src_port', 'dst_port', 'bidirectional_bytes']
    for column in columns:
        if column in data:
            data[column].value_counts().plot(kind='bar')
            plt.title(f'{column} distribution')
            plt.ylabel('')
            plt.savefig(os.path.join(output_dir, f'{column}.png'))
            plt.close()
```

Rysunek 5: Funkcja *charts* generująca grafiki dla poszczególnych parametrów flow

Działanie kodu:

- Analizuje plik PCAP używając NFStreamer i konwertuje dane do pandas DataFrame
- Usuwa kolumny z jedną unikalną wartością lub brakującymi danymi
- Generuje wykresy słupkowe dla kluczowych parametrów:
  - IP źródłowe/docelowe
  - Protokoły
  - Porty źródłowe/docelowe
  - Liczba bajtów dwukierunkowych
- Zapisuje każdy wykres jako osobny plik PNG
- Pomaga w wizualnej analizie charakterystyki ruchu



Rysunek 6: Przykładowa grafika wygenerowana przez *charts* przy analizie

4. Detection as a Code

4.1. D.1: Implementacja reguł detekcyjnych

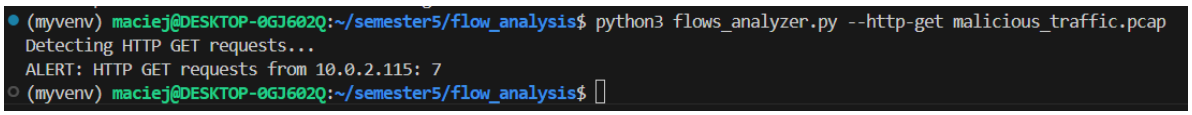
W pliku *detection_rules.py* zaimplementowano szereg funkcji detekcyjnych, w tym przykładowo *detect_http_get*:

```python
# HTTP GET detection
def detect_http_get(pcap, ip_to_remove=None, chart=False):
    packets = rdpcap(pcap)
    http_get_counts = {}
    for pkt in packets:
        if ip_to_remove and pkt.haslayer(IP) and pkt[IP].src == ip_to_remove:
            continue
        if pkt.haslayer(TCP) and pkt[TCP].dport == 80:
            if b"GET" in bytes(pkt.payload):
                src_ip = pkt[IP].src
                http_get_counts[src_ip] = http_get_counts.get(src_ip, 0) + 1
    for src_ip, count in http_get_counts.items():
        increment_threat_count("http_get")
        print(f"ALERT: HTTP GET requests from {src_ip}: {count}")
    if chart and http_get_counts:
        plot_bar_chart(http_get_counts, "HTTP GET Requests Detected", "Source IP", "Count", pcap)
    return http_get_counts
```

Rysunek 7: *detect_http_get* z użyciem *Scapy*

Działanie kodu:

- Czyta pakiety z pliku PCAP
- Filtruje pakiety:
  - Pomija określone IP jeśli podano ip_to_remove
  - Szuka pakietów TCP na porcie 80
  - Wykrywa ciąg „GET” w payloadzie
- Zlicza żądania GET per IP źródłowe
- Wyświetla alerty dla wykrytych żądań
- Opcjonalnie generuje wykres statystyk
- Służy do wykrywania potencjalnego skanowania HTTP lub innej nietypowej aktywności



Rysunek 8: Rezultat *detect_http_get*

4.2. D.2: Wczytywanie reguł Sigma

W pliku *read_sigma.py* zaimplementowano obsługę reguł Sigma. Wczytywanie reguł realizuje poniższa funkcja:

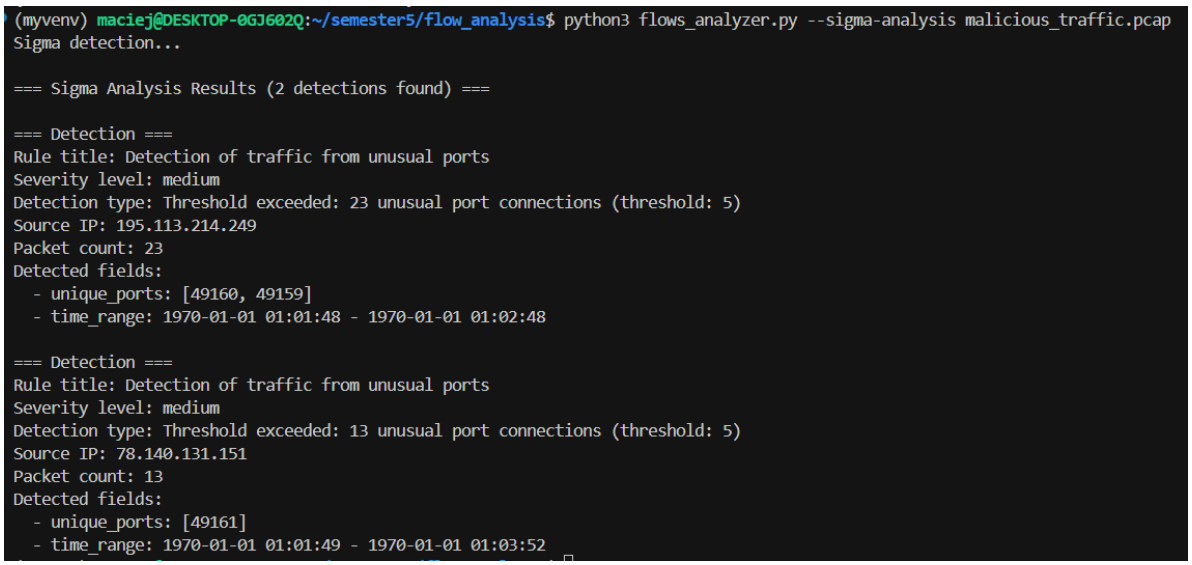
```python
def load_sigma_rule(rule_path):
    with open(rule_path, 'r') as file:
        rule_content = yaml.safe_load(file)
    rule = SigmaRule.from_dict(rule_content)
    return rule, rule_content
```

Rysunek 9: Funkcja *load_sigma*

Działanie kodu:

- Otwiera plik z regułą Sigma w formacie YAML
- Używa yaml.safe_load do bezpiecznego wczytania zawartości
- Konwertuje zawartość na obiekt SigmaRule używając metody from_dict
- Zwraca zarówno obiekt reguły jak i oryginalną zawartość
- Służy do wczytywania reguł detekcji zagrożeń w formacie Sigma

Z racji na rozmiar *read_sigma.py* jego całokształt nie zostanie zamieszczony w sprawozdaniu (kody są zamieszczone wraz ze sprawozdaniem).



Rysunek 10: Rezultat wczytywania i obsługiwania reguł Sigma

4.3. V.1: Wykres liczby wykrytych zagrożeń

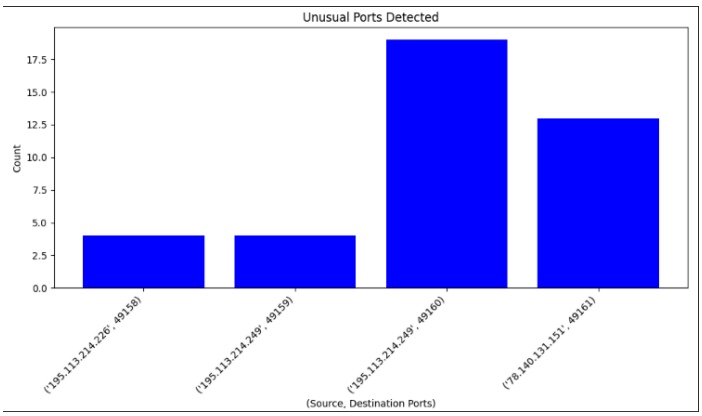
Oprócz wykresów generowanych z parametrów, uwzględniona została możliwość generowania statystyk rezultatów - przedstawienie wyników poszczególnych reguł detekcji (jeśli zostały wykryte podejrzane adresy IP, to zliczane było ile razy zostały one zarejestrowane dla danego rodzaju detekcji), a także Threat Summary, czyli zliczenie ile alertów zostało wygenerowanych dla danego typu detekcji.

```python
# Generates a bar chart
def plot_bar_chart(data, title, xlabel, ylabel, pcap):
    base_name = os.path.basename(pcap)
    output_dir = create_dir(base_name, 'results')

    plt.figure(figsize=(10, 6))
    keys = [str(key) for key in data.keys()]
    values = list(data.values())

    plt.bar(keys, values, color='blue')
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, f'{title}.png'))
    plt.close()
```

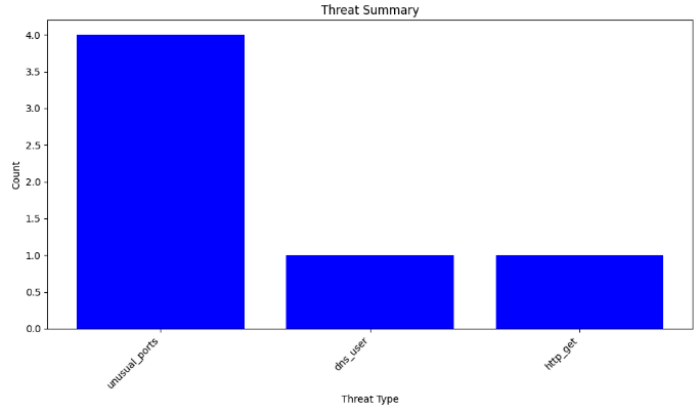
Rysunek 11: *plot_bar_chart* - funkcja generująca wykresy rezultatów detekcji

Działanie kodu:

- Tworzy wykres słupkowy używając matplotlib
- Przyjmuje słownik z danymi, tytuł i etykiety osi
- Konwertuje klucze i wartości ze słownika na listy
- Tworzy wykres z niebieskimi słupkami
- Zapisuje wykres jako PNG w katalogu wynikowym
- Automatycznie dostosowuje układ wykresu



Rysunek 12: Przykładowa grafika dla wybranej reguły detekcji - w tym prypadku *unusual_ports_flow*



Rysunek 13: Threat Summary dla reguł detekcji

4.4. V.2: Mapa geograficzna

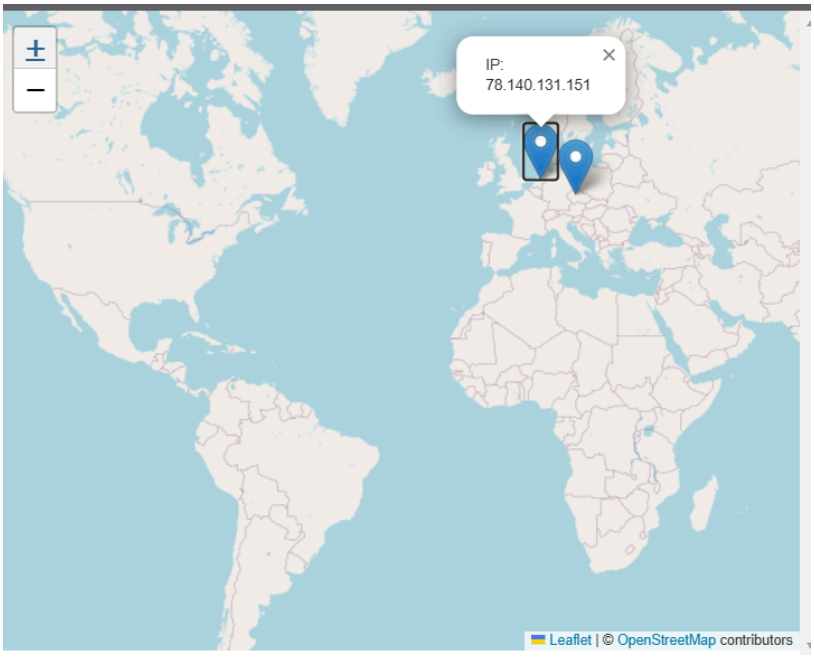
Implementacja mapy w *raport_generator.py* realizuje generowanie mapy z użyciem ipinfo.io. Zamieszcza wszystkie podejrzane IP zaraportowanych przez stworzony flow analyzer na interaktywnej mapie.

```python
def generate_ip_location_map(*args, pcap):
    base_name = os.path.basename(pcap)
    output_dir = create_dir(base_name, 'results')
    output_file = os.path.join(output_dir, "map.html")

    API_KEY = ""
    map_center = [0, 0]
    m = folium.Map(location=map_center, zoom_start=2)

    unique_ips = set()  # Zbiór, aby uniknąć duplikatów

    # Walidacja argumentów i zbieranie kluczy
    for a in args:
        if isinstance(a, dict):  # Sprawdzenie, czy `a` jest słownikiem
            unique_ips.update(a.keys())
        else:
            print(f"WARNING: Skipping non-dict argument: {a}")

    print(f"Unique IPs from alerts: {unique_ips}")

    for ip in unique_ips:
        try:
            response = requests.get(f"https://ipinfo.io/{ip}", headers={"Authorization": f"Bearer {API_KEY}"})
            data = response.json()

            loc = data.get("loc")
            if loc:
                lat, lon = map(float, loc.split(","))
                folium.Marker(location=[lat, lon], popup=f"IP: {ip}").add_to(m)
        except Exception as e:
            print(f"ERROR: cannot process {ip}: {e}")

    m.save(output_file)
    print(f"Map written to {output_file}")
```

Rysunek 14: *generate_ip_location_map*

Działanie kodu:

- Tworzy katalog wynikowy 'results' bazując na nazwie pliku PCAP
- Inicjalizuje mapę Foliu
- Zbiera unikalne adresy IP z przekazanych argumentów
- Dla każdego IP:
  - Pobiera dane geolokalizacyjne przez API ipinfo.io
  - Wyciąga współrzędne
  - Dodaje marker na mapę z popup zawierającym IP
- Zapisuje interaktywną mapę HTML
- Obsługuje błędy przy nieudanych zapytaniach do API



Rysunek 15: Przykładowa mapa dla jednej z analiz

5. Enrichment

5.1. E.1: Pobieranie informacji o IP/domenach

W pliku *raport_generator.py* zaimplementowano integrację z API AbuseIPDB:

```python
# Check IP reputation using AbuseIPDB API
def check_ip_reputation(pcap, chart=False):
    base_name = os.path.basename(pcap)
    output_dir = create_dir(base_name, 'jsons')

    API_KEY = ""
    headers = {
        "Accept": "application/json",
        "Key": API_KEY
    }

    streamer = NFStreamer(source=pcap, statistical_analysis=True)
    unique_ips = {flow.src_ip for flow in streamer}
    unique_ips.update({flow.dst_ip for flow in streamer})
    abuseIPs = {}

    for ip in unique_ips:
        response = requests.get(f"https://api.abuseipdb.com/api/v2/check?ipAddress={ip}", headers=headers)
        data = response.json()


        if data['data']['abuseConfidenceScore'] >= 30: # or data['data']['totalReports'] >= 50: # You can change that
            print(f'AbuseIPDB considers {ip} as dangerous!')
            if ip in abuseIPs:
                abuseIPs[ip] += 1
            else:
                abuseIPs[ip] = 1


        with open(os.path.join(output_dir, f"{ip}_reputation.json"), "w") as f:
            f.write(str(data))

    if abuseIPs and chart:
        plot_bar_chart(abuseIPs, "Dangerous IPs Detected By AbuseIPDB", "Source IP", "Count", pcap)

    print(abuseIPs)

    return abuseIPs
```

Rysunek 16: *generate_ip_location_map*

Działanie kodu:

- Zbiera unikalne IP z pliku PCAP używając NFStreamer
- Dla każdego IP:
  - Sprawdza jego reputację przez API AbuseIPDB
  - Identyfikuje IP jako niebezpieczne jeśli score >= 30
  - Zapisuje wyniki do pliku JSON
- Opcjonalnie generuje wykres słupkowy wykrytych niebezpiecznych IP
- Zwraca słownik z niebezpiecznymi IP i ich licznikiem wystąpień

Funkcja sprawdza IP występujące w danym ruchu w bazie danych AbuseIPDB. Jeśli AbuseIPDB wykryje jakieś IP jako podejrzane to generuje alert.

```
(myvenv) maciej@DESKTOP-0GJ602Q:~/semester5/ml-flow-analysis/ml-flows-analysis$ python3 flows_analyzer.py --get-reputation-ip pcap-folder/malicious_traffic.pcap
Checking IP reputations...
AbuseIPDB considers ff02::16 as dangerous!
AbuseIPDB considers 10.0.2.2 as dangerous!
AbuseIPDB considers ff02::2 as dangerous!
AbuseIPDB considers 8.8.4.4 as dangerous!
AbuseIPDB considers ff02::1:2 as dangerous!
AbuseIPDB considers ff02::1:3 as dangerous!
AbuseIPDB considers 8.8.8.8 as dangerous!
AbuseIPDB considers fe80::a866:72ce:9643:c9b1 as dangerous!
AbuseIPDB considers ff02::1:ff43:c9b1 as dangerous!
AbuseIPDB considers 10.0.2.115 as dangerous!
AbuseIPDB considers 78.140.131.151 as dangerous!
AbuseIPDB considers 224.0.0.252 as dangerous!
AbuseIPDB considers 195.113.214.249 as dangerous!
AbuseIPDB considers 195.113.214.226 as dangerous!
AbuseIPDB considers :: as dangerous!
AbuseIPDB considers 10.0.2.255 as dangerous!
```

Rysunek 17: Odnajdowanie niebezpiecznych IP w ruchu według IPDB - w celu prezentacji działania celowo ustawiono warunek sprawdzania „if data['data']['abuseConfidenceScore'] >= 0" na 0%, zamiast standardowych 30%

6. Flows Analyzer - prezentacja działania

W tym rozdziale przedstawiamy ogólne działanie naszego narzędzia Flows Analyzer.

```
(myvenv) maciej@DESKTOP-0GJ602Q:~/semester5/ml-flow-analysis/ml-flows-analysis$ python3
flows_analyzer.py  --overall  --ip-to-remove  fe80::a866:72ce:9643:c9b1  pcap-folder/
malicious_traffic.pcap
Executing all analyses...
Generating data charts...
Detecting large flows...
Detecting asymmetrical flows...
Detecting unusual port usage...
ALERT: Unusual port usage detected: to port 49158 by 195.113.214.226, Count: 4
ALERT: Unusual port usage detected: to port 49159 by 195.113.214.249, Count: 4
ALERT: Unusual port usage detected: to port 49160 by 195.113.214.249, Count: 19
ALERT: Unusual port usage detected: to port 49161 by 78.140.131.151, Count: 13
Finding DNS users...
DNS user: 10.0.2.115, Count: 1
Detecting SYN flood attacks...
Detecting HTTP GET requests...
ALERT: HTTP GET requests from 10.0.2.115: 7
Detecting Ping flood attacks...
Detecting port scanning...
Sigma detection...

=== Sigma Analysis Results (2 detections found) ===

=== Detection ===
Rule title: Detection of traffic from unusual ports
Severity level: medium
Detection type: Threshold exceeded: 23 unusual port connections (threshold: 5)
Source IP: 195.113.214.249
Packet count: 23
Detected fields:
  - unique_ports: [49160, 49159]
  - time_range: 1970-01-01 01:01:48 - 1970-01-01 01:02:48

=== Detection ===
Rule title: Detection of traffic from unusual ports
Severity level: medium
Detection type: Threshold exceeded: 13 unusual port connections (threshold: 5)
Source IP: 78.140.131.151
Packet count: 13
Detected fields:
  - unique_ports: [49161]
  - time_range: 1970-01-01 01:01:49 - 1970-01-01 01:03:52
Checking IP reputations...
Generating map and saving to HTML...
Unique IPs from alerts: {'195.113.214.249', '78.140.131.151', '10.0.2.115'}
Map written to malicious_traffic.pcap-wrapped/results/map.html

Generating comprehensive report...

Detailed report saved to: reports/analysis_report_20241223_191510.txt

Final IPs Threat Report:
unusual_ports: 4
dns_user: 1
http_get: 1
```

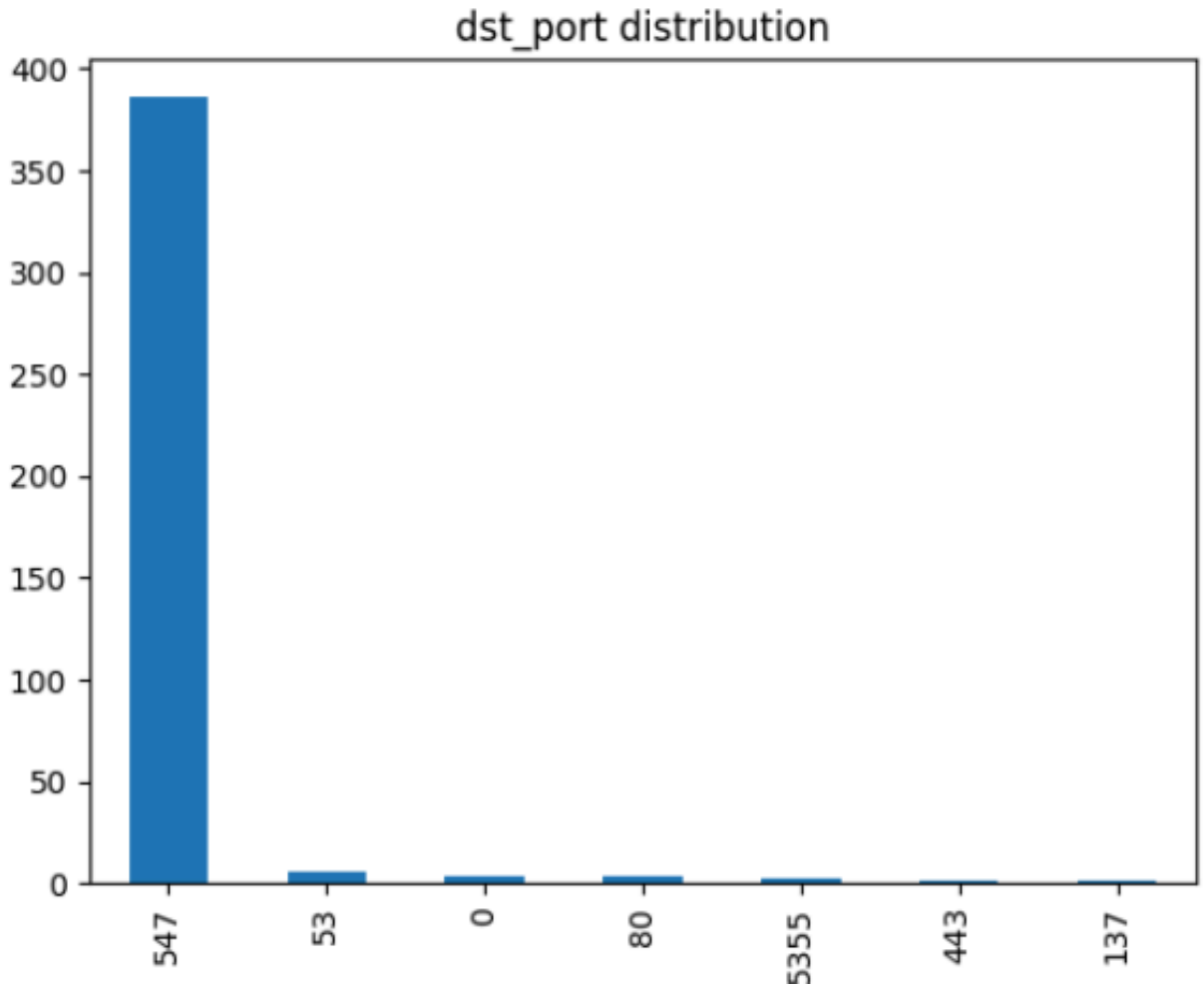
Jak można zauważyć, w tym przykładzie użyliśmy analizy statycznej dla pliku *malicious_traffic.pcap*. Najpierw sprawdziliśmy programem perspektywiczne IP, tak aby w naszej analizie nie identyfikować naszego IP, jako podejrzanego. Oczywiście jest to ryzykowne i zgodnie z naszym zaleceniem w instrukcji programu

najlepiej podać własne/perspektywiczne IP sprawdzone przez użytkownika.
Rezultatami działania programu są:

- **Raport w formacie .txt**

```
=== Network Traffic Analysis Report ===
Generated at: 2024-12-23 19:33:44.846735
Analyzed file: pcap-folder/malicious_traffic.pcap

=== Sigma Rules Detections ===
IP 195.113.214.249: 1 detections
IP 78.140.131.151: 1 detections

=== Unusual Ports Usage ===
IP 195.113.214.226: 4 connections to unusual ports
IP 195.113.214.249: 23 connections to unusual ports
IP 78.140.131.151: 13 connections to unusual ports

=== DNS Usage Analysis ===
IP 10.0.2.115: 1 DNS queries

=== HTTP GET Requests ===
IP 10.0.2.115: 7 requests


=== Summary of Detections ===
Total number of alerts: 7
Number of detection types triggered: 4
```

- **Diagramy wybranych atrybutów wejściowych**



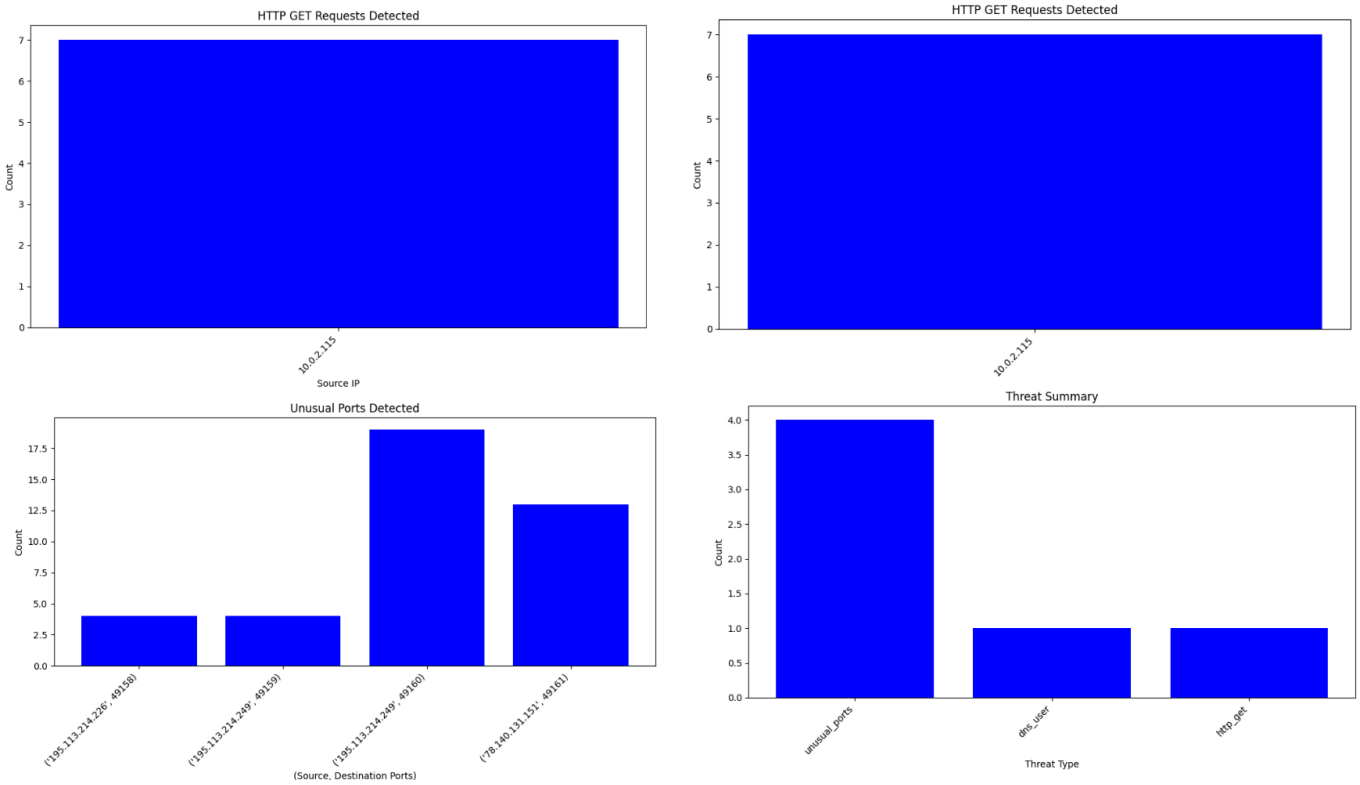
Rysunek 18: Przykładowy diagram dla atrybutów wejściowych

- **Pliki .json zwrócone w zapytaniach IPDB**

{'data': {'ipAddress': '78.140.131.151', 'isPublic': True, 'ipVersion': 4, 'isWhitelisted': None, 'abuseConfidenceScore': 0, 'countryCode': 'NL', 'usageType': 'Data Center/Web Hosting/Transit', 'isp': 'Webzilla B.V.', 'domain': 'webzilla.com', 'hostnames': [], 'isTor': False, 'totalReports': 0, 'numDistinctUsers': 0, 'lastReportedAt': None}}

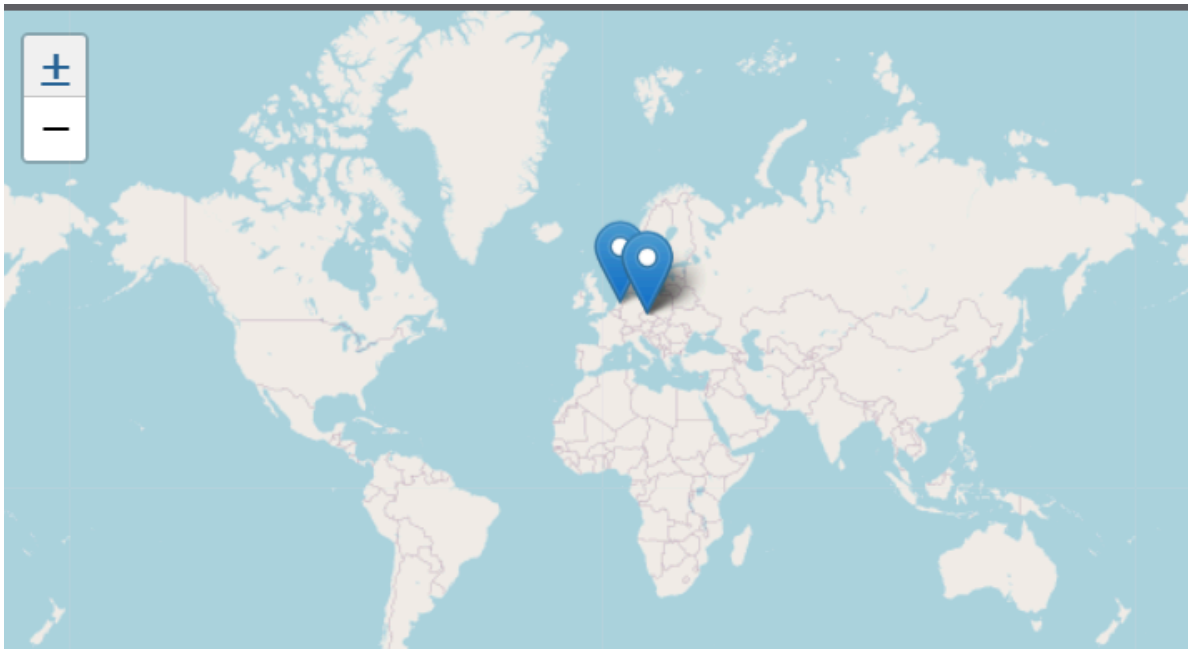
Rysunek 19: Przykładowy zwrócony .json

- **Diagramy rezultatów**



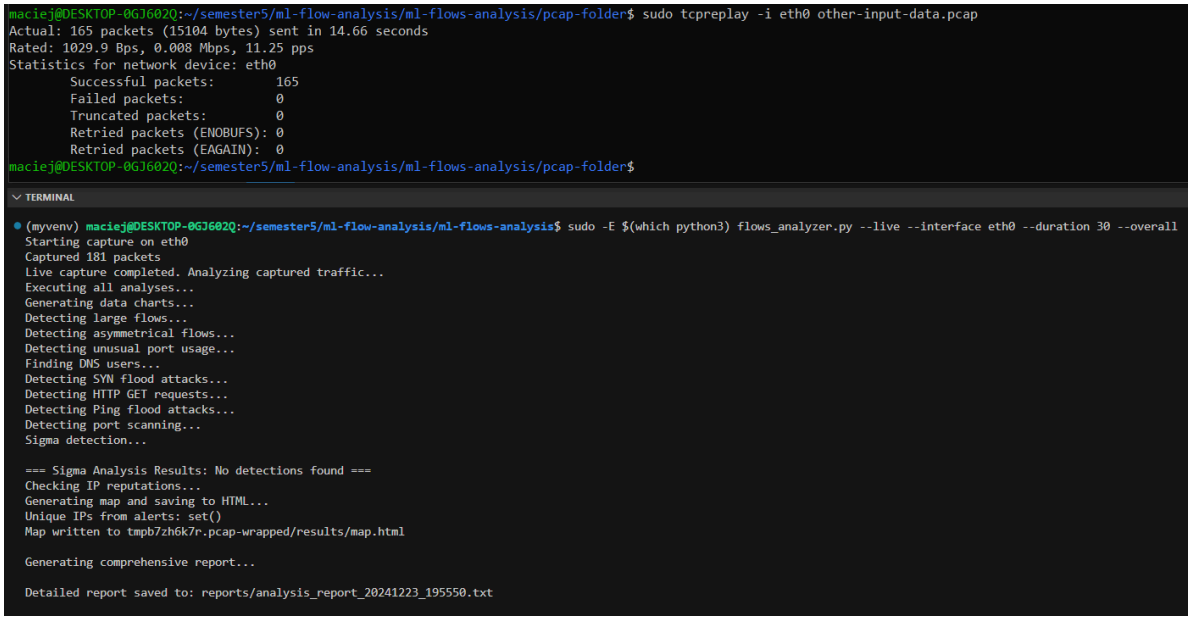
Rysunek 20: Diagramy rezultatów

- **Mapę podejrzanych adresów IP**



Rysunek 21: Mapa podejrzanych adresów IP

Program działa również dla *live capture*, czemu dowodzi poniższy zrzut ekranu:



Rysunek 22: Skorzystanie z programu w wersji *live capture*

Zgodnie z założeniem program nie wykrył żadnego zagrożenia - była to symulacja ruchu z bezpiecznego pliku .pcap.

7. Machine Learning

7.1. ML.1: Klasyfikacja flow

W pliku *ml_flows_analyzer.py* zaimplementowano klasyfikację flow przy użyciu *scikit-learn* (funkcja *train_model* służy do utworzenia nowego modelu ML):

```python
def train_model(X_train, y_train):
    """Train the model with hyperparameter tuning"""
    # Define parameter grid for GridSearch
    param_grid = {
        'n_estimators': [100, 200],
        'max_depth': [None, 10, 20],
        'min_samples_split': [2, 5],
        'min_samples_leaf': [1, 2],
        'class_weight': ['balanced', 'balanced_subsample']
    }

    # Initialize base model
    base_model = RandomForestClassifier(random_state=42)

    # Perform GridSearch
    grid_search = GridSearchCV(
        estimator=base_model,
        param_grid=param_grid,
        cv=5,
        scoring='f1',
        n_jobs=-1
    )

    # Fit the model
    grid_search.fit(X_train, y_train)

    return grid_search.best_estimator_, grid_search.best_params_
```

Rysunek 23: *train_model*

7.2. ML.2: Redukcja liczby fałszywych pozytywów

System wykorzystuje *GridSearchCV* do optymalizacji parametrów modelu:

```
# Perform GridSearch
grid_search = GridSearchCV(
    estimator=base_model,
    param_grid=param_grid,
    cv=5,
    scoring='f1',
    n_jobs=-1
)

# Fit the model
grid_search.fit(X_train, y_train)
```

Rysunek 24: Fragment funkcji definiującej *GridSearchCV*

7.3. ML.3: Trenowanie modelu z nowymi danymi

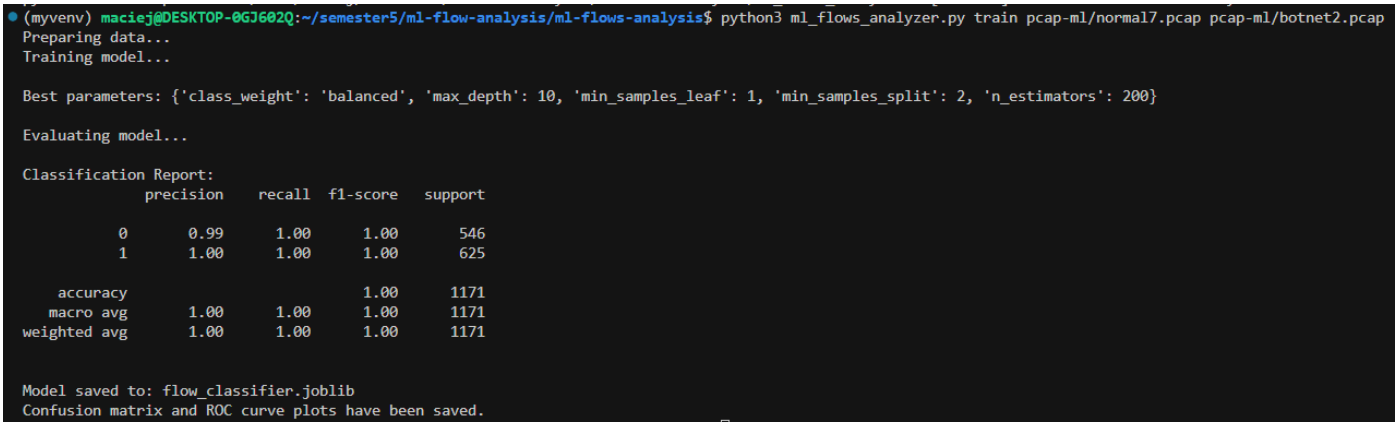
Wprowadziliśmy implementację CLI do trenowania utworzonego modelu:

```
@cli.command()
@click.argument('normal_pcap', type=click.Path(exists=True))
@click.argument('malicious_pcap', type=click.Path(exists=True))
@click.option('--model-output', default='flow_classifier.joblib', help='Path to save the trained model')
def train(normal_pcap, malicious_pcap, model_output):
    """Train a new model using normal and malicious PCAP files"""
    # Prepare data
    print("Preparing data...")
    data = prepare_data(normal_pcap, malicious_pcap)

    # Split data
    X = data.drop('label', axis=1)
    y = data['label']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Train model
    print("Training model...")
    model, best_params = train_model(X_train, y_train)
    print("\nBest parameters:", best_params)

    # Evaluate model
    print("\nEvaluating model...")
    report, cm, roc_auc = evaluate_model(model, X_test, y_test)
    print("\nClassification Report:")
    print(report)

    # Save model
    joblib.dump(model, model_output)
    print(f"\nModel saved to: {model_output}")
    print("Confusion matrix and ROC curve plots have been saved.")
```

Rysunek 25: Moduł CLI do trenowania modelu

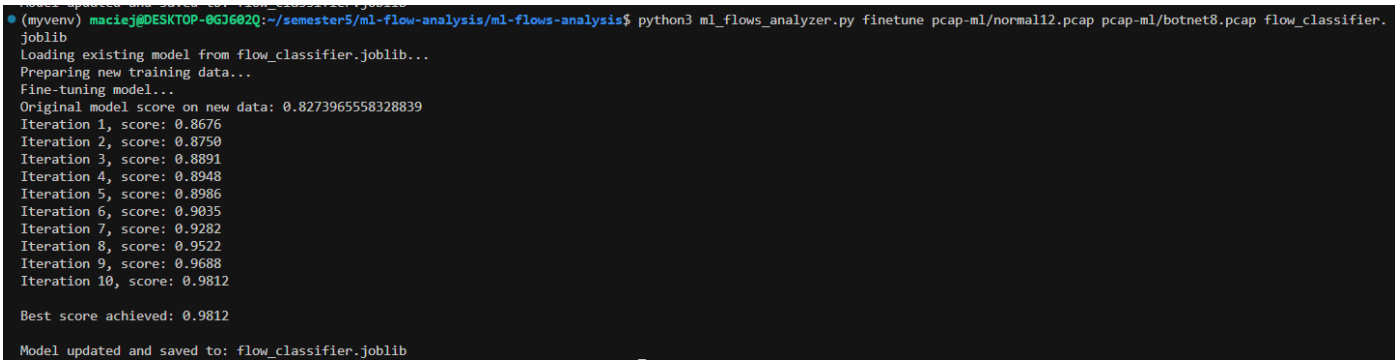
8. ML Flows Analyzer - prezentacja działania

1. Najpierw tworzymy nowy model ML poprzez podanie modelowi złośliwego oraz poprawnego ruchu za pomocą pliku .pcap:



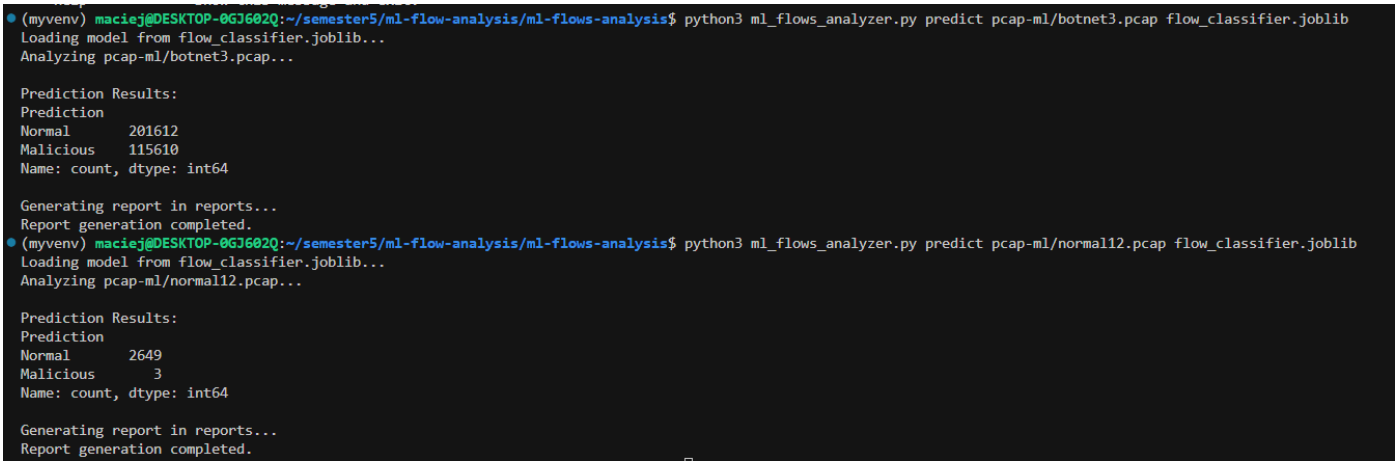
Rysunek 26: Tworzenie modelu ML

2. Następnie należy trenować model - czym więcej danych do nauki, tym nasz model działa lepiej:



Rysunek 27: Trenowanie modelu ML

3. Nastepnie możemy sprawdzić, jak działa nasz model:



Rysunek 28: Predykcja z modelem ML

Raport ze złośliwego .pcap:

```
=== Network Traffic Analysis Report ===

Generated at: 2024-12-23 20:25:33.405129

=== General Statistics ===
Total flows analyzed: 317222
Normal flows: 201612
Malicious flows: 115610

=== Malicious Flow Details ===

Flow 4578:
Source IP: 10.0.2.29
```

Destination IP: 199.89.170.197
Destination Port: 80
Confidence: 1.00

Flow 78:
Source IP: 10.0.2.29
Destination IP: 2.50.12.137
Destination Port: 27670
Confidence: 1.00

Flow 75:
Source IP: 10.0.2.29
Destination IP: 74.141.217.35
Destination Port: 7117
Confidence: 1.00

Flow 4396:
Source IP: fe80::a866:72ce:9643:c9b1
Destination IP: ff02::1:2
Destination Port: 547
Confidence: 1.00

Flow 113125:
Source IP: 10.0.2.29
Destination IP: 217.160.81.200
Destination Port: 53
Confidence: 1.00

Flow 127:
Source IP: 10.0.2.29
Destination IP: 8.8.8.8
Destination Port: 53
Confidence: 1.00

Flow 88:
Source IP: 10.0.2.29
Destination IP: 69.159.192.117
Destination Port: 8494
Confidence: 1.00

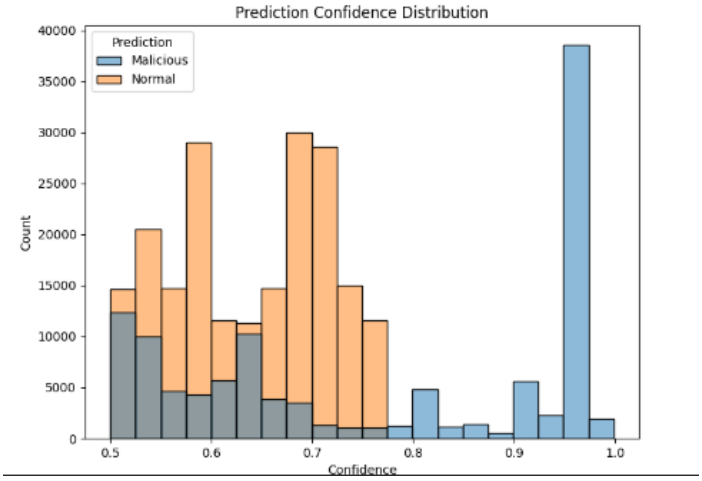
Flow 1009:
Source IP: 10.0.2.29
Destination IP: 196.210.183.125
Destination Port: 64673
Confidence: 1.00

Flow 4814:
Source IP: 10.0.2.29
Destination IP: 98.124.252.66
Destination Port: 80
Confidence: 1.00

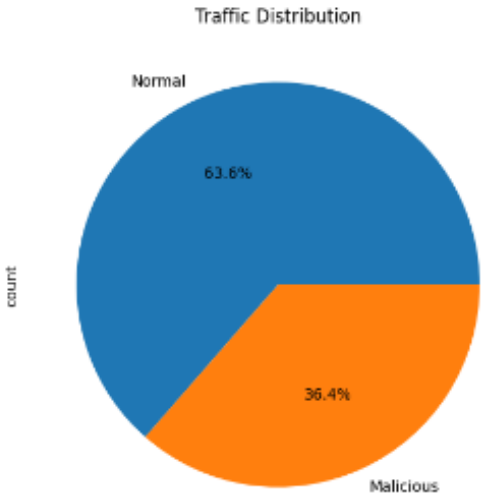
Flow 1060:
Source IP: fe80::a866:72ce:9643:c9b1
Destination IP: ff02::1:2
Destination Port: 547
Confidence: 1.00

Flow 125:
Source IP: 10.0.2.29
Destination IP: 190.42.234.9
Destination Port: 1113

```
Confidence: 1.00

...
```



Rysunek 29: Confidence Distribution złośliwego .pcap

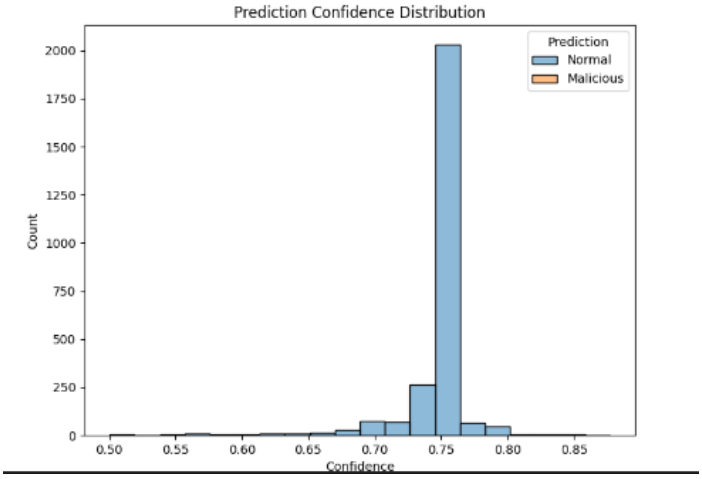


Rysunek 30: Traffic Distribution złośliwego .pcap

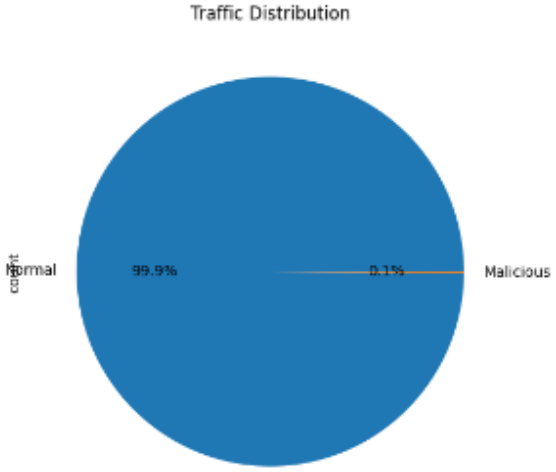
Raport z niezłośliwego .pcap:

```
=== Network Traffic Analysis Report ===

Generated at: 2024-12-23 20:26:30.917371

=== General Statistics ===
Total flows analyzed: 2652
Normal flows: 2649
Malicious flows: 3

=== Malicious Flow Details ===

Flow 62:
Source IP: 10.0.0.46
Destination IP: 124.90.142.164
Destination Port: 56085
Confidence: 0.73

Flow 1615:
Source IP: 10.0.0.46
Destination IP: 213.65.221.143
```

```
Destination Port: 6881
Confidence: 0.58

Flow 595:
Source IP: 10.0.0.46
Destination IP: 124.79.69.91
Destination Port: 1481
Confidence: 0.51

=== Summary ===
Percentage of malicious flows: 0.11%
```



Rysunek 31: Confidence Distribution niezłośliwego .pcap



Rysunek 32: Traffic Distribution niezłośliwego .pcap

Jak można zauważyć wyniki są zadowalające. Oczywiście rezulaty będą tym lepsze, im więcej treningu uzyska model (na rzecz prezentacji model był trenowany krótko), a także od wyboru atrybutów, których *zachowania* model miał się nauczyć. W naszym przypadku były to: 'bidirectional_packets', 'bidirectional_bytes', 'src2dst_packets', 'dst2src_packets', 'src2dst_bytes', 'dst2src_bytes', 'bidirectional_duration_ms'.