

# A Machine Learning-Based Network Anomaly Detection System Prototype Using Isolation Forest <sup>†</sup>

Viktoria Ivanova  and Delyan Genkov 

Department of Computer Systems and Technologies, Technical University of Gabrovo, 5300 Gabrovo, Bulgaria; v.ivanova@tugab.bg

\* Correspondence: dgenkov@tugab.bg

<sup>†</sup> Presented at the International Conference on Electronics, Engineering Physics and Earth Science (EEPES2026), Bandirma, Turkey, 24–27 June 2026.

## Abstract

Due to the large amount of network traffic, manual network security management has become increasingly difficult. This paper proposes an open-source network anomaly detection system prototype that operates with the Security Onion 2.4 platform. The system relies on a custom Python engine that uses the machine learning (ML) algorithm Isolation Forest to detect anomalies in multiple Zeek datasets. The prototype was developed as a virtual machine (VM), which is hosted on a server running the software for virtualization VMware ESXi 6.0.0. For the experimental tests, live network telemetry from a university network was used. The results show that the system achieves a constant anomaly detection rate of 5.4% after a volumetric threshold of 5000 logs is reached within operating periods ranging from fifteen to thirty minutes. During testing, it was discovered that the detection logic has a minimal resource impact (less than 1 GB) on the running system beyond the defined baseline of average Random Access Memory (RAM) usage. This proposed solution is an open-source and zero-cost alternative to other paid network security solutions, demonstrating that abnormal behavior detection in network traffic can be effectively integrated on existing computer configurations without the need for expensive licenses.

**Keywords:** network anomaly detection; network security; unsupervised machine learning; isolation forest; Security Onion 2.4; Suricata; VMware ESXi 6.0.0; Zeek

## 1. Introduction

As the volume of current network traffic grows, manual administration of network security becomes more complex [1,2]. Network security solutions in the past had depended on intrusion detection and prevention systems like Suricata to identify known malicious patterns. While these systems are effective for known threats, they struggle to identify novel or subtle behavioral deviations that do not match pre-existing signatures [2,3].

Nowadays, network security solutions rely on the integration of anomaly detection. This anomaly-based detection takes advantage of a variety of algorithms for machine learning (ML) to detect novel threats in real time [4]. Security Onion version 2.4 is a popular open-source solution that provides network visibility and security. To achieve this, the platform uses a wide range of services. Some of these are Zeek's network traffic monitoring and analysis capabilities and Suricata's signature-based threat detection [5–7]. The way various logs in Elasticsearch 8.18.8 are handled is also another important aspect of the platform. The database's data is a considerable choice as a source for system and network anomaly detection [8].



Academic Editors: Teodor Iliev, Ivaylo Stoyanov, Grigor Mihaylov and Selahattin Kosunalp

Published: 28 August 2026

**Copyright:** © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

While these solutions offer improved visibility, network security experts are concerned about alert fatigue [9]. Many anomaly detection solutions are either exclusive or resource-intensive. This fact makes them impractical for smaller organisations. Moreover, there is a clear need for lightweight and flexible solutions that can detect outliers from existing live data streams, such as those in Elasticsearch, and not from labeled training datasets [10].

This study provides a localized network anomaly detection system prototype that uses a custom Python engine. The method detects anomalous patterns in Zeek datasets using the Isolation Forest, an unsupervised ML algorithm specifically built for high-dimensional outlier detection. For the creation of the proposed system, the software for virtualization VMware ESXi 6.0.0 was used. Isolation Forest, unlike traditional clustering approaches, is highly efficient, making it a suitable choice for lightweight setups [11,12].

The remainder of this paper is organized as follows: Section 2 discusses the proposed system design and implementation. Section 3 presents the experimental results and discusses key findings. Finally, Section 4 concludes the paper and addresses plans for future work.

## 2. Materials and Methods

### 2.1. Experimental Environment and Network Integration

The proposed system in this study is a localized prototype for a university building network. It incorporates near-real-time anomaly detection directly into the Security Onion 2.4 platform. This solution takes advantage of the platform's capacity to acquire data from multiple sources. It also utilizes its integrated components, such as Suricata for signature-based detection and Zeek for monitoring and analysing network traffic. For the detection of anomalies, a custom Python version 3.9 script was created. The system makes use of the libraries Scikit-Learn (version 1.6) for ML, Pandas (version 2.2) for data manipulation, and the Rich library (version 13.9) for system output visualization. To detect and display abnormal behavioral shifts that deviate from the established network baseline, the engine extracts Zeek logs from the Elasticsearch database.

1. **Experimental Environment:** The prototype is a virtual machine (VM) running on an enterprise-grade physical server. The software used for virtualization on the server is VMware ESXi 6.0.0. As a result, the detection engine has dedicated access to the necessary computational resources without interference from other processes. The hardware and software specifications of the VM are shown in Table 1.
2. **Network Integration:** The system connects to the university's physical network using a dual network adapter setup, with both interfaces configured in bridged mode. This guarantees that the system perceives the network environment the same way a hardware device would, as both adapters can be assigned an Internet Protocol (IP) address.
  - **Management interface:** The physical interface ens160 has an assigned static IP Version 4 (IPv4) address within the range of the university's Local Area Network (LAN). The main purpose of this adapter is to run Security Onion services and not capture or inspect network traffic. In addition, this interface is responsible for the management of the connection between the Python engine and the Elasticsearch Representational State Transfer (REST) Application Programming Interface (API) via localhost:9200.
  - **Monitor interface:** The physical interface ens192 does not have an assigned IP address. It is configured in promiscuous mode, and its main role is to maintain data integrity while being a passive listener in the network.

- Logical interface aggregation: The system captures the network traffic using the virtual interface (bond0). By default, this interface is configured as the master of the physical interface ens192. In this case, the Security Onion components such as Zeek and Suricata receive the captured telemetry from bond0. This data flow is important for consistency before the metadata is indexed into Elasticsearch for further Artificial Intelligence (AI) examination.

**Table 1.** Prototype system specifications.

| Component/Resource           | Specification Details                 |
|------------------------------|---------------------------------------|
| Processor                    | Intel Xeon E5-2609 v4                 |
| Virtualization Platform      | VMware ESXi 6.0.0                     |
| Number of Logical Processors | 4                                     |
| System Memory                | 64 GB RAM                             |
| Preallocated Storage         | 200 GB                                |
| Base Operating System        | Oracle Linux 9.7                      |
| Security Distribution Suite  | Security Onion 2.4 (Standard Edition) |
| Detection Engine             | Python 3.9+/Scikit-Learn              |
| Database                     | Elasticsearch 8.18.8                  |

## 2.2. Data Capture, Normalization and Feature Engineering

1. Data capture and ingestion: The first step of the data pipeline is to acquire raw telemetry from the Elasticsearch 8.18.8 database. For communication with the REST API, a parameterized Python function is designed. The system checks the *logs-\* index*, which contains a variety of telemetry types such as *zeek.conn*, *zeek.dns*, *zeek.dhcp*, *zeek.snmp*, and *suricata.alert*. To accomplish the log retrieval, a system-level curl command is used that forwards a JavaScript Object Notation-Domain Specific Language (JSON-DSL) query, as illustrated in Figure 1. The query sets the log extraction limit, the chronological ordering, and a boolean filter for the used dataset and time window. The prototype remains reliable without overloading the database because this technique enables a modular data pipeline.

```
def fetch_logs(dataset, window="30d", size=10000):
    query = {"size": size, "sort": [{"@timestamp": "desc"}], "query": {"bool": {"must": [{"range": {"@timestamp": {"gte": f"now-{window}"}}], {"term": {"event.dataset": dataset}}]}}}
    # System-level REST API call
    cmd = f'curl -u {USER}:{PASS} -k -s -X GET {URL} -H 'Content-Type: application/json' -d '{json.dumps(query)}''
    res = os.popen(cmd).read()
    return json.loads(res).get('hits', {}).get('hits', [])
```

**Figure 1.** Parameterized Python function `fetch_logs` for network telemetry acquisition.

2. Data normalization and feature engineering: A dynamic preprocessing pipeline was developed to guarantee that the prototype can adapt to the various network logs generated by Zeek. These logs include Domain Name System (DNS), Dynamic Host Configuration Protocol (DHCP), Simple Network Management Protocol (SNMP), general network connection (Conn) and other metadata. The Suricata alert logs especially skip the normalization and encoding stages. The data is output directly by the system without it being processed by the Isolation Forest. This guarantees that all known signature-based anomalies are reported immediately with their predetermined severity levels and rule names, and without unnecessary processing costs. The

pipeline, defined in the `run_ai_analysis` function for the Zeek network telemetry, has the following steps:

- JSON object conversion: The engine uses a method called `pandas.json_normalize` to convert the nested JSON objects from Elasticsearch's `_source` field into a flat tabular dataframe. This means that all the network telemetry characteristics, varying from the transport layer to protocol-specific metadata, are captured and analysed.
- String to numeric encoding: The Isolation Forest demands numerical input. Therefore, the system employs a global casting technique. Before the *label encoder* handles the received columns, they are converted to a string format to handle data types ranging from nulls, integers, to IP addresses. The result is a complete numerical mapping of the entire log's state. This allows the model to detect anomalies based on the relationship between all the data fields in the log, rather than an array of explicitly selected features.

### 2.3. Anomaly Detection and Model Persistence

For anomaly detection, the system uses the Isolation Forest algorithm. Algorithm 1 contains the system's core function.

---

#### Algorithm 1: *RunAIAAnalysis*(*H*, *ds\_id*, *is\_init* = *False*)

---

**Inputs:** *H*—raw log hits; *ds\_id*—dataset identifier; *is\_init*—Boolean flag

**Outputs:** *A*—detected anomalies

```

1: If H is empty, return empty tabular DataFrame DF
2: Increment global STATS["total_logs"] by |H|
3: Path ← persistent_models/brain_ + ds_id + .pkl
4: Normalize nested JSON from H into tabular DF
5: Cast all column names of DF to string format
6: DFencoded ← LabelEncoder(DF)
7: If is_init is False and file exists in Path :
8: Load model M from Path
9: If feature names in M ≠ columns in DFencoded :
10: Re-train M ← IsolationForest(contamination = 0.05, random_state = 42) of DFencoded
11: Else: M ← IsolationForest(contamination = 0.05, random_state = 42) of DFencoded
12: Generate predictions preds ← M.predict(DFencoded)
13: Save M to Path via joblib.dump()
14: A ← Rows in DF where preds = −1
15: count ← |A|
16: Update global STATS["total_anomalies"] ← STATS["total_anomalies"] + count
17: If "dns" ∈ ds_id : STATS["zeek_dns"] ← STATS["zeek_dns"] + count
18: Else if "conn" ∈ ds_id : STATS["zeek_conn"] ← STATS["zeek_conn"] + count
19: Else if "dhcp" ∈ ds_id : STATS["zeek_dhcp"] ← STATS["zeek_dhcp"] + count
20: Else if "snmp" ∈ ds_id : STATS["zeek_snmp"] ← STATS["zeek_snmp"] + count
21: Return A

```

---

The reasoning behind the choice of Isolation Forest is its linear time complexity  $O(t\psi \log \psi)$  and low memory usage. This gives the algorithm several advantages over the density-based method LOF, the near-linear time complexity distance-based method ORCA and the unsupervised method Random Forests. For example, the average time complexity of LOF is  $O(n \log n)$  with *n* data points, but for high-dimensional data, the complexity increases to  $O(n^2)$ . After the empirical evaluation of these methods, Isolation Forest demonstrated higher performance in terms of processing time and efficiency, specifically in large datasets. Therefore, the algorithm is a preferable choice for handling high-volume

databases, especially for real-life applications, like the proposed Security Onion 2.4 setup for the university network in this paper [13,14].

To implement the detection phase and long-term preservation of the model's training progression, the following steps are taken:

1. **Model retrieval and initial validation:** The engine implements a model persistence mechanism that reuses the previous training progression of the models to optimize the hardware load. After capturing the raw network telemetry, the engine performs an initial validation check, as shown in Step 1 of Algorithm 1. If log entries are found, in Step 2 the system increments the global log counter to make sure that the final session report accurately reflects the total traffic volume processed. In Step 3, the system creates a path to the local directory *persistent\_models* and defines the name identifier structure for each *.pkl* file. The files that are saved in this folder are represented in Figure 2. To avoid execution errors during feature comparison, all column headers are converted to a string format in Step 5. Logs are converted into numeric forms using the *label encoder* in Step 6. Whereas the retrieval of the model takes place in Steps 7–8, with the current location being used for the persistent memory of the pre-trained Isolation Forest models for each Zeek dataset. To guarantee that the system identifies the correct model, a specific number of files are saved in the file format *.pkl*. The name of each file is based on a Zeek dataset's name, for example *brain\_zeek\_dns.pkl*. In Steps 9–11, the engine deserializes the model and validates the schema by comparing the *feature\_names\_in\_* property to the active dataframe columns. This is done by using the Joblib library. If a file is missing or the names mismatch, the engine will perform an automatic *fit()* procedure to initialize or retrain the model.
2. **Anomaly detection:** By finding an anomaly score  $s$  for each log in Step 12, the engine locates outliers. Based on [13], this score is calculated using the path length  $h(x)$ . The length represents the number of edges the instance  $x$  requires from the root to a terminal external node. The average path length  $E(h(x))$  is normalized against  $c(n)$ , which is the expected path length of an unsuccessful search in a binary search tree [13]:

$$c(n) = 2H(n-1) - \left( \frac{2(n-1)}{n} \right), \quad (1)$$

where  $n$  is the size of the training data and  $H(i)$  is the harmonic number. To find this number, the equation  $\ln(i) + 0.5772$  (Euler's constant) must be solved. The anomaly score  $s$  is:

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}}. \quad (2)$$

The short path lengths (anomalies) are indicated with  $s \rightarrow 1$ , whereas  $s \approx 0.5$  suggests a normal distribution. The model, using a contamination parameter equal to 0.05 in Steps 10–11, predicts  $-1$  for the top 5% of the cases as anomalies in Step 14, filtering them out for the final set  $A$ .

3. **Memory retention and statistical accounting:** To ensure training consistency spanning across multiple runtime sessions, the newly modified state of model  $M$  is saved on disk in Step 13 by calling *joblib.dump()*. Finally, in Step 15, the function calculates the number of elements in the anomaly set  $|A|$  and increments counters depending on the dataset type in a global dictionary named STATS. This process, represented in Steps 16–20 of Algorithm 1, gives the system the ability to produce a consistent, numerical evaluation of identified abnormalities across distinct Zeek logs, culminating in the final result in Step 21.

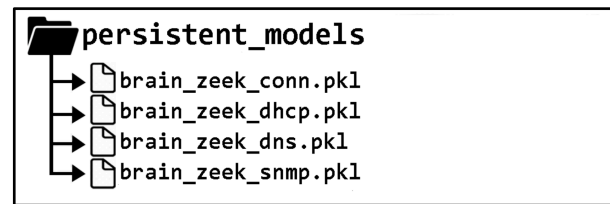


**Figure 2.** File hierarchy of the system's AI models.

#### 2.4. System Logic and Real-Time Reporting

The system is run by a while loop that connects specific phases: the model training on historical network logs, the near real-time behavioral detection and the automatic visual session reporting. As implemented in the core logic, the system, with its unique operational phases, can provide continuous detection throughout sequential monitoring sessions.

1. System initialization and baseline establishment: After running the Python script, the engine enters the initialization sequence in phase 1. In the beginning, a search in the Elasticsearch database is done for a 30-day historical time period of network telemetry. This stage is essential for establishing a good baseline of the usual university network behavior. The AI models are trained according to the specific network activity in the LAN. The system guarantees that the training knowledge is saved throughout system restarts by a retention mechanism that stores and loads the encoded models for each Zeek dataset in *.pkl* files. This allows consistent anomaly detection during periodic run sessions.
2. Live telemetry ingestion and detection: After the initialization phase, the system enters a second one. This phase is a continuous loop where the engine obtains the Suricata and Zeek logs from the database every thirty seconds. The specific time period was chosen to balance the processing demand while ensuring that anomalies are detected quickly. In this approach, each found Suricata alert is treated as an immediate signature-based anomaly. Meanwhile, the Zeek data is passed to the anomaly detection engine for behavioral outlier analysis. The system calculates the overall Anomaly detection rate (R) at the end of each session run to determine the network's security state:

$$R = \left( \frac{\sum_{z \in Z} |A_z|}{\sum_{z \in Z} |H_z|} \right) 100\%, \quad (3)$$

where  $Z$  is the set of processed Zeek log types,  $z$  represents a singular log within  $Z$ ,  $|A_z|$  is the total amount of found anomalies in the  $Z$  set, and  $|H_z|$  indicates the number of raw telemetry hits processed for a given log type.

3. Automated analytic reporting and visualization: In order to provide an effective overview, the orchestration loop features a visualization output in the console terminal. The global statistic counters (STATS) are summed up into a session report table over one-minute intervals. Aside from the global measurements, the engine generates detailed outputs for each Zeek dataset. The results from the generation show essential features such as source and destination IP addresses and ports, Media Access Control (MAC) addresses, domain and host names, descriptions of the connection statuses, and threat severity levels.
4. Operational flexibility: It should be mentioned that the system is designed to be modular. All operating time intervals, including the historical training window, the main while loop delay, and reporting intervals, are fully adjustable. This allows the system to scale based on the experimental environment's traffic and hardware boundaries. Section 3, which is named Experimental Results, presents the visual outputs and experimental data generated by the system's detection logic.

Figure 3 shows the workflow of the system's key components. The process begins with capturing the network traffic using the network adapter, then it passes through the log extraction and detection phases, and finally it finishes with the visualization of the results in the terminal.

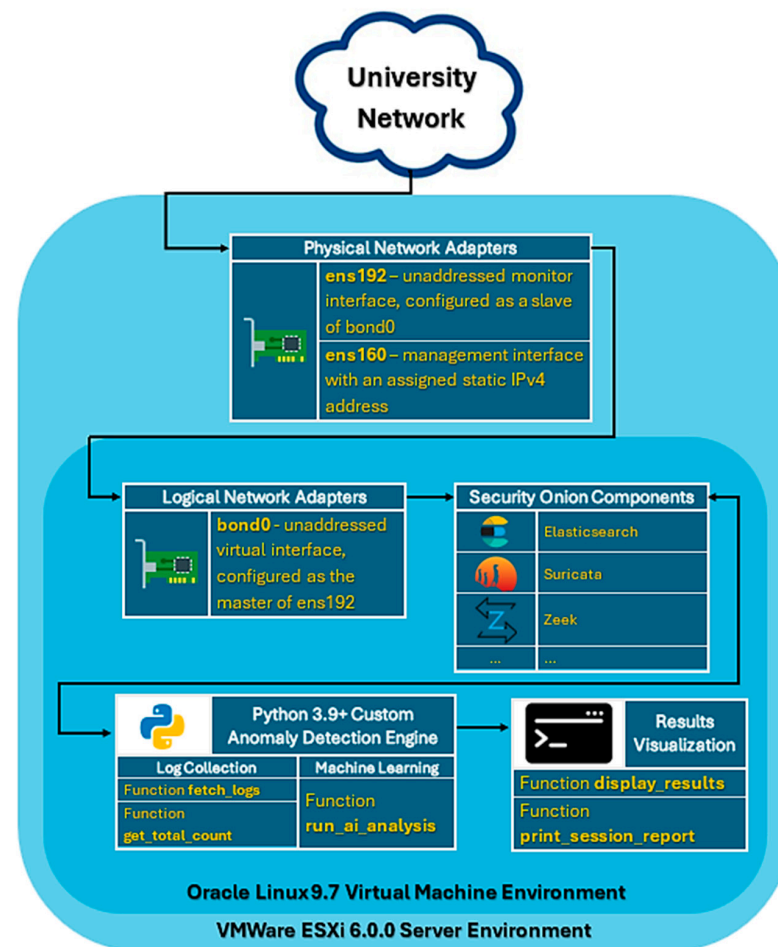


Figure 3. Operational workflow of the system.

### 3. Experimental Results

#### 3.1. Data Scale and Repository Audit

The first step of the evaluation begins with a quantitative test. Using the custom `get_total_count` function, the system's input capabilities were represented. This function provides a multi-layered audit based on its direct interaction with Security Onion's Elasticsearch database. The goal is to define the hierarchy between the global repository volume and the specific network telemetry needed for analysis.

Figure 4 provides the results defining the scope of this study. The engine detected a repository of more than 10.3 million records, representing the total network capture of the system, as shown in Figure 4a. Approximately 5.8 million logs are found in the thirty-day operational window. The engine extracts specific event datasets (e.g., `zeek.dns`, `zeek.conn`, `zeek.dhcp`, `zeek.snmp`, and `suricata.alert`) from this time window, as visualized in Figure 4b. The success of this phase is based on the high-ratio data funneling. By extracting 65,145 network events from a repository of roughly 6 million logs, the system demonstrated its ability to minimize data dimensionality while avoiding memory overloads and retrieval latency from the query. This guarantees that in the next phase the engine focuses on the live network traffic rather than redundant system telemetry, resulting in a defined baseline for the AI models.

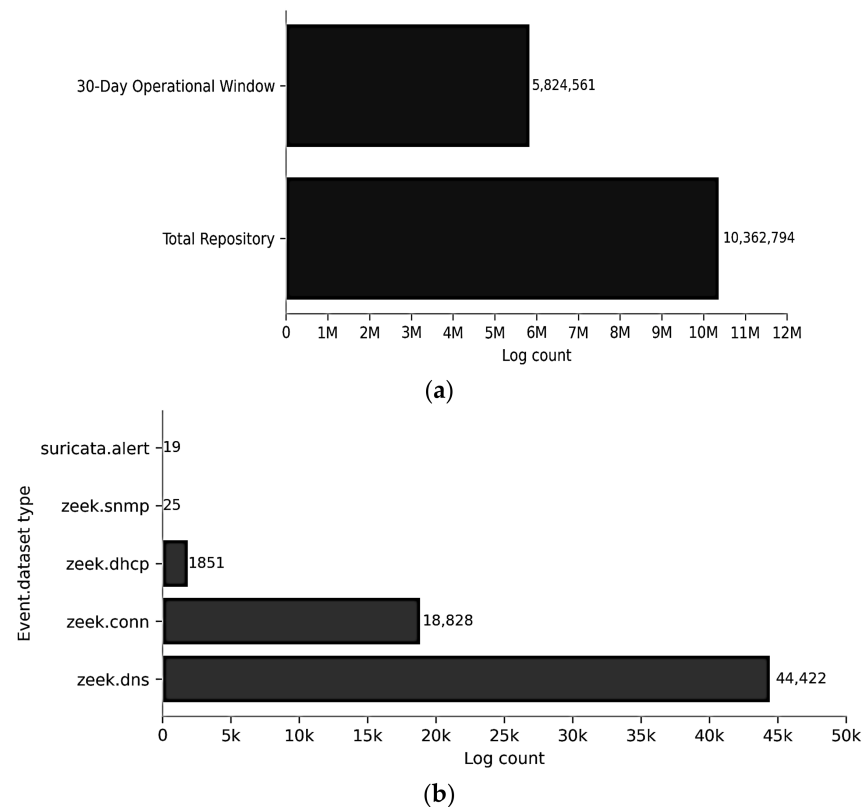


**Figure 4.** System repository log breakdown: (a) comparison of total captured log volume to date. (b) log distribution of Suricata and Zeek datasets for the 30-day training window.

### 3.2. Historical Phase and OUTPUT Evaluation

The historical log processing phase starts with the output of all available *suricata.alert* logs to establish a baseline. During validation, this process detected 19 alerts. The engine has an extraction limit of 10,000 logs per dataset to maintain speed across Zeek datasets, such as *zeek.dns*, which has over 44,000 records. This phase relies on the Isolation Forest algorithm for anomaly detection in the LAN and the Elasticsearch database with its RESTful query engine.

The engine for each processed event does a complete analysis. It evaluates the network's traffic behavioral pattern by using a *label encoder* and flattening nested JSON fields. The resulting outliers are found throughout the entire log structure. In this approach, all kinds of metadata are unique attributes in a high-dimensional feature set.

Tables 2 and 3 present the results, which highlight dataset-specific features such as source and destination IP addresses, ports, and transport protocols. This comparative view highlights a significant distinction between the Suricata alert and the anomaly seen in the *zeek.snmp* logs. It also confirms the system's ability to detect network events that the signature-based detection of Suricata missed.

**Table 2.** Suricata signature-based alert results during the historical phase.

| Date          | Time     | Rule Name                  | UUID <sup>1</sup> | Severity <sup>2</sup> | Source IP  | Source Port | Destination IP | Destination Port |
|---------------|----------|----------------------------|-------------------|-----------------------|------------|-------------|----------------|------------------|
| 16 April 2026 | 11:10:34 | ET INFO Spotify P2P Client | 2027397           | Low                   | 10.3.48.15 | 57621       | 10.3.255.255   | 57621            |

<sup>1</sup> UUID represents the unique identifier integer of each rule as shown in the Security Onion 2.4 console [5]. <sup>2</sup> Severity shows a specific Suricata alert severity level (e.g., info, low, medium or high) [5].

**Table 3.** Zeek SNMP anomaly detection results during the historical phase.

| Date          | Time     | Source IP  | Source Port | Destination IP  | Destination Port | Community <sup>1</sup> | Version <sup>2</sup> |
|---------------|----------|------------|-------------|-----------------|------------------|------------------------|----------------------|
| 30 March 2026 | 06:03:45 | 10.3.40.73 | 39325       | 255.255.255.255 | 161              | public                 | 2c                   |

<sup>1</sup> Community represents the string of the first SNMP packet associated with the session, used as part of SNMP's (v1/v2c) administrative/security framework. The value public indicates default read-only access [7]. <sup>2</sup> Version shows the used SNMP version. The value 2c refers to SNMPv2c [7].

### 3.3. Near Real-Time Anomaly Detection and Memory Persistence

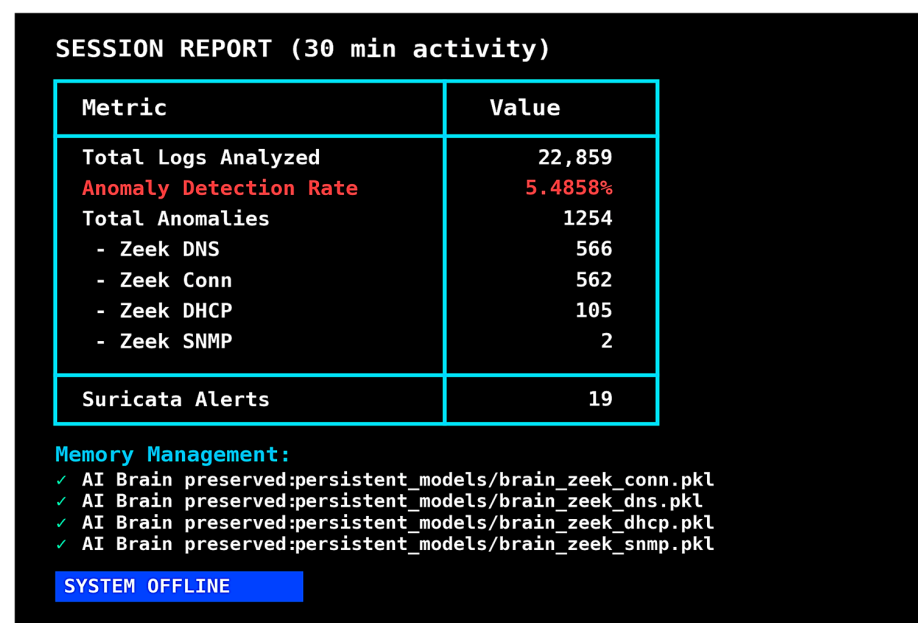
The near real-time detection phase utilizes thirty-second retrieval cycles to transition the engine into a permanent monitoring state. Using the Elasticsearch database, the system extracts captured Zeek logs from the last five minutes for the trained models. Unlike the historical phase, the incoming logs are compared to a predefined behavioral baseline to identify novel outliers. As seen in Table 4, the system successfully detected an anomaly in the live *zeek.dns* stream, isolating a Web Proxy Auto-Discovery (WPAD) query.

**Table 4.** Zeek DNS anomaly detection results during the near real-time phase.

| Date          | Time     | Source IP  | Source Port | Destination IP | Destination Port | Query (Domain) | Status <sup>1</sup> | Protocol <sup>2</sup> |
|---------------|----------|------------|-------------|----------------|------------------|----------------|---------------------|-----------------------|
| 18 April 2026 | 16:10:58 | 10.3.30.61 | 137         | 10.3.255.255   | 137              | WPAD           | NIMLOC              | UDP                   |

<sup>1</sup> Status displays the name of the DNS response code (e.g., NIMLOC indicating the domain name does not exist in the DNS records). <sup>2</sup> Protocol shows the connection's transport layer protocol (e.g., UDP).

To guarantee constant network visibility and situational awareness, the engine creates session reports every minute. These updates provide a real-time record of the total examined Zeek logs, Suricata alerts, discovered outliers, and the anomaly detection rate. As shown in Figure 5, when the session is terminated, a final report table is generated.

**Figure 5.** Cumulative session audit and model state saving.

At the end of each system runtime session, the engine saves individual model states as *.pkl* files, as illustrated in Figure 5. The prototype has permanent memory, ensuring that

the training progress and learned network behavioral patterns for each Zeek dataset are preserved over time. The memory transfer between system sessions eliminates the need for full re-initialization of models, maximizing visibility while controlling computing overhead through log extraction and model serialization.

### 3.4. Experimental Tests

To identify the most optimal operational settings for the prototype, six controlled tests were performed. These experiments, which are presented in Table 5, highlight the correlation between the historical log limit ingestion ( $L$ ) and the training time window periods. The results determine the impact on anomaly detection, model training state persistence, and resource consumption. A thirty-minute control test was run before the initialization to establish the system RAM utilization idle baseline (in GBs). The usage was recorded over five-minute intervals, and the test achieved an average of 36.92 GBs, demonstrating a constant overhead of the Security Onion services. The historical baseline logs ( $L$ ) obtained for initial training from the Elasticsearch database and live telemetry ingested during near real-time detection are combined into a single data source value. This value is used for the metric called total logs processed.

**Table 5.** Comparative test results and resource metrics.

| Test ID | Training Window (Days) | $L$ <sup>1</sup> | Total Logs Processed | $R(\%)$ | Total Anomalies | Model Size Deviation (MB) | Average RAM Usage (GB) | Runtime (min) |
|---------|------------------------|------------------|----------------------|---------|-----------------|---------------------------|------------------------|---------------|
| 1       | 30                     | 10,000           | 22,699               | 5.3703  | 1219            | −0.4 (1.9)                | 37.18                  | 30            |
| 2       | 30                     | 5000             | 12701                | 5.5950  | 711             | −0.3 (2)                  | 37.5                   | 30            |
| 3       | 30                     | 1000             | 3773                 | 17.4132 | 657             | +1.6 (3.9)                | 37.6                   | 30            |
| 4       | 15                     | 10,000           | 18,764               | 5.4529  | 1023            | 0 (2.3)                   | 37.86                  | 15            |
| 5       | 15                     | 5000             | 11,872               | 5.3571  | 636             | −0.2 (2.1)                | 37.8                   | 15            |
| 6       | 15                     | 1000             | 3688                 | 17.8145 | 657             | +1.6 (3.9)                | 37.87                  | 15            |

<sup>1</sup> Log limit ( $L$ ) represents the maximum historical log ingestion number per individual dataset.

The key experimental analysis and observations are as follows:

- The model size deviation measures the change in the persistent models directory size from the initial 2.3 MB during the system experimental setup. A negative deviation (as seen in Tests 1, 2, and 5) demonstrates predictive reduction, in which the engine detects excessive repetition in network traffic events. It then optimizes the model state, decreasing total directory size to 1.9 MB. On the other hand, the positive deviation of +1.6 MB in Tests 3 and 6 indicates a level of growth. In these instances, the model fails to reach a conclusion on normal behavior and attempts to serialize a significantly higher number of distinct outliers.
- The data confirms a direct inverse relationship between the log limit ( $L$ ) and the anomaly detection rate ( $R$ ). At  $L \geq 5000$ , the detection rate stabilizes between 5.3% and 5.6%. This range indicates a good understanding of the network baseline, where the Isolation Forest can efficiently differentiate ordinary traffic from true outliers. Reducing  $L$  to 1000 logs resulted in a dramatic increase in the detection rate, which reached 17.8%. This is a consequence of the limited number of network traffic logs. In this case, the detection engine classifies legitimate but rare network events as anomalies.
- During the execution of Tests 3 and 6, a quick transition through the historical phase was observed. The engine, when  $L$  has a lower value, completes its initial training

almost instantly, compared to Tests 1 and 4. However, the increase in speed resulted in a reduction in the anomaly detection accuracy. As long as the  $L$  value remained high, the system achieved nearly equal accuracy across both training time windows. These findings imply that the system prototype's accuracy depends on the number of logs.

- The log limit ( $L$ ) threshold is a volumetric threshold, which represents the volume of processed network logs needed by the Isolation Forest model to establish a reliable baseline and overall stability. The log threshold value  $L \geq 5000$  was chosen at the end of the iterative testing phase, after viewing and empirical validation of the experimental results. The decision is not only based on the overall results but also on the observed changes in the model's behavior, which further supports the direct inverse relationship between the log limit threshold value and the anomaly detection rate ( $R$ ).
- The system was stable across all six tests. Despite the wide variety in log ingestion, the average RAM usage was within the narrow range of 37.18 to 37.87 GB. When compared to the 36.92 GB idle baseline, the overall load of the detection engine is less than 1.0 GB. This confirms that the system can provide near real-time network anomaly detection without impacting the stability of other Security Onion services.
- The contamination parameter is set to 0.05 as a heuristic to achieve prioritization of network visibility. However, this creates an evident trade-off: reducing the chance of missing a real threat comes at the cost of detecting more false positives. This allows the system to be able to capture subtle abnormal behavior, which has never been detected before. For example, if the parameter was set to 0.01 or 0.02, these types of unusual events could be overlooked, and only the most extreme anomalies would be detected, which is not optimal and creates risks of missing a real threat for a complex and diverse environment like the university network.

The current detection logic behind the prototype isolates the top 5% of the total processed logs for each of the Zeek datasets. The number of these logs represents the most statistically abnormal captured entries compared to the established baseline by the model. The prototype flags successfully network traffic novelties. However, the current operation of the system can be described as a "black box". This means that the system does not yet have the capabilities to automatically distinguish and categorize the found anomalies, as well as give feedback if they are actual security threats or false positives. To mitigate this, the system outputs Suricata alerts alongside the detected anomalies in the Zeek logs. This approach provides additional context to help determine if the outliers are true security threats. The presented results in this paper confirm the need for further research and work focusing on explainable AI techniques, designing a superior approach that provides greater clarity of the specific type of detected anomalies, fine-tuning of the parameters used by the Isolation Forest algorithm, and overall Python logic behind the detection module.

#### 4. Conclusions

This study presents an open-source, modular anomaly detection solution that is integrated with the Security Onion 2.4 platform. It was developed in an experimental virtual environment with the main goal of improving visibility and security of a university network. The system has a custom engine that detects anomalies using the unsupervised learning algorithm Isolation Forest while optimising resource usage. The achieved stability at  $L \geq 5000$  logs is a clear indicator that the accuracy is affected by the data volume, instead of the time duration of training. The anomaly detection in this system is designed without the need for expensive AI licenses or additional hardware. This is a clear indicator that the proposed prototype is a scalable and zero-cost alternative to enterprise network traffic analysis and threat detection solutions.

Future study will focus on creating an internal system log extraction function to avoid the Elasticsearch API limitations. As a result, the model training will be done on the entire log history of each Zeek dataset type without latency. Furthermore, an automated function will be implemented to autosave model states during operation. To ensure that the model training progression is saved, despite system restarts or power outages, a backup solution will be created. Support for processing external log files will also be added, as well as a second ML or deep learning algorithm to improve anomaly detection accuracy. Therefore, a comprehensive comparison with existing network anomaly detection systems using alternative algorithms (e.g., Autoencoders, RNN, k-NN, LOF, Random Forests, and One-Class SVM) is planned as future work. For these comparisons, the proposed prototype will be expanded to support standardized benchmarking of metrics (e.g., accuracy, precision, recall and f-score) using well-known datasets like UNSW-NB15, NSL-KDD, and CIC-IDS2017, as well as live Security Onion network data streams. The current terminal interface output visualization will be replaced with a web-based interface featuring automated alerting capabilities. By adding such features, it is expected that the prototype will mitigate false positives while maintaining high detection rates. To conclude, further research and testing are needed to transition the system prototype to a real-time engine with superior anomaly detection.

**Author Contributions:** V.I. and D.G. were involved in the full process of producing this paper, including conceptualization, methodology, programming and preparing the manuscript. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the University Center for Research and Technology at the Technical University of Gabrovo, project NIP2026-6 “Study and Development of a Computer Networks Anomaly Detection System”.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The raw data supporting the conclusions of this article will be made available by the authors upon request.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## Abbreviations

The following abbreviations are used in this manuscript:

|      |                                     |
|------|-------------------------------------|
| ML   | Machine Learning                    |
| VM   | Virtual Machine                     |
| RAM  | Random Access Memory                |
| REST | Representational State Transfer     |
| IP   | Internet Protocol                   |
| IPv4 | IP Version 4                        |
| API  | Application Programming Interface   |
| LAN  | Local Area Network                  |
| WPAD | Web Proxy Auto-Discovery            |
| MAC  | Media Access Control                |
| AI   | Artificial Intelligence             |
| JSON | Java Script Object Notation         |
| DSL  | Domain Specific Language            |
| DNS  | Domain Name System                  |
| DCHP | Dynamic Host Configuration Protocol |
| SNMP | Simple Network Management Protocol  |

## References

1. Divakaran, D.M. Traffic modeling for network security and privacy: Challenges ahead. *arXiv* **2025**, arXiv:2503.22161v2. [[CrossRef](#)]
2. Anis, F.M.; Alabdullatif, M.; Aljbli, S.; Hammoudeh, M. A survey on the applications of deep learning in network intrusion detection systems to enhance network security. *IEEE Access* **2025**, *13*, 185357–185373. [[CrossRef](#)]
3. Kalavadekar, P.N.; Sane, S.S. Building an effective intrusion detection system using combined signature and anomaly detection techniques. *Int. J. Innov. Technol. Explor. Eng.* **2019**, *8*, 429–435. [[CrossRef](#)]
4. Ahmad, T.; Truscan, D. Efficient early anomaly detection of network security attacks using deep learning. In Proceedings of the 2023 IEEE International Conference on Cyber Security and Resilience (CSR), Venice, Italy, 31 July–2 August 2023.
5. Security Onion Solutions. Security Onion 2.4 Documentation. 2025. Available online: <https://docs.securityonion.net/en/2.4/> (accessed on 13 March 2026).
6. OISF. Suricata User Guide. 2025. Available online: <https://suricata.readthedocs.io> (accessed on 13 March 2026).
7. Zeek Project. Zeek Documentation. 2025. Available online: <https://docs.zeek.org> (accessed on 5 April 2026).
8. Elastic. Elasticsearch Guide. 2025. Available online: <https://www.elastic.co/guide/> (accessed on 12 May 2026).
9. Aminanto, M.E.; Zhu, L.; Ban, T.; Isawa, R.; Takahashi, T.; Inoue, D. Automated threat-alert screening for battling alert fatigue with temporal isolation forest. In Proceedings of the 2019 17th International Conference on Privacy, Security and Trust (PST), Fredericton, NB, Canada, 26–28 August 2019.
10. Zamry, N.M.; Zainal, A.; Rassam, M.A.; Alkhamash, E.H.; Ghaleb, F.A.; Saeed, F. Lightweight Anomaly Detection Scheme Using Incremental Principal Component Analysis and Support Vector Machine. *Sensors* **2021**, *21*, 8017. [[CrossRef](#)] [[PubMed](#)]
11. Djidjev, C. siForest: Detecting network anomalies with set-structured isolation forest. *arXiv* **2024**, arXiv:2412.06015v1.
12. VMware. VMware vSphere 6.0 Performance Best Practices. 2015. Available online: <https://www.vmware.com/docs/vmware-perfbest-practices-vsphere6-0-white-paper> (accessed on 10 March 2026).
13. Liu, F.T.; Ting, K.M.; Zhou, Z.-H. Isolation Forest. In Proceedings of the 8th IEEE International Conference on Data Mining, Pisa, Italy, 15–19 December 2008.
14. Breunig, M.M.; Kriegel, H.-P.; Ng, R.T.; Sander, J. LOF: Identifying density-based local outliers. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, 15–18 May 2000.

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.