AnDO: A Lightweight Feature Extraction Framework for IDS Modelling in Low-Resource Software-Defined Networks

Emmanuel Ackerson[1][0009−0002−6793−7407] and Josiah Chavula[2][0000−0002−6774−7526]

[1] University of Cape Town, Cape Town, South Africa
ackemm001@myuct.ac.za
[2] University of Cape Town, Cape Town, South Africa
josiah.chavula@uct.ac.za

Abstract. Network traffic features serve as predictor variables for machine learning-based intrusion detection systems (ML-IDS), yet practical deployment is often hindered by computational overhead and latency introduced during capture and extraction. While benchmark datasets like UNSW-NB15 and InSDN have accelerated IDS research through pre-engineered features, they exhibit limited generalization in live or heterogeneous environments. Critically, SDN-oriented datasets omit architecture-intrinsic attributes such as control-plane state, data-plane interactions, and flow-rule dynamics essential for accurately modeling SDN behavior. This paper proposes AnDO, an efficient real-time feature extraction framework for low-resource Software-Defined Networking environments. AnDO implements an end-to-end extraction pipeline directly within the live network, eliminating external database dependencies. The architecture integrates Argus for flow generation, nDPI for protocol classification, and ONOS control-plane intelligence, augmented by a custom sliding-window connection-tracking engine for contextual flow statistics. By fusing packet-level metrics, protocol labels, and SDN state information, AnDO extracts 50 per-flow features under a linear computational model. Experimental evaluation in a virtualized SDN testbed demonstrates predictable scalability, stable resource utilization, and bounded overhead. These results validate AnDO as an efficient, resource-aware feature extraction framework suitable for constrained and community-oriented network environments.

Keywords: Feature extraction · SDN-Specific Features · ONOS Controller · Machine-Learning IDS · AnDO framework · Network Traffic · Software-Defined Networking · Low-Resource CWNs · lightweight feature extraction framework.

1 Introduction

Network traffic features are measurable metrics and categorical attributes derived from packets, flows, or sessions, providing a structured representation of network behavior and state [1]. These features are extracted across multiple layers of the protocol stack, ranging from packet-level attributes (e.g., packet length, inter-arrival time, protocol flags) to application-layer semantics such as HTTP methods, DNS query types, and TLS parameters. They are generally classified into three categories: packet-based (individual packet header features), flow-based (aggregated statistics over packets sharing common five-tuple attributes), and session-based (higher-layer aggregation across related flows forming a complete interaction). In machine learning pipelines, these features function as predictor variables for traffic classification, behavioral analysis, and intrusion detection. An Intrusion Detection System (IDS) is a critical security mechanism that monitors and analyzes network traffic to detect malicious activity before damage occurs [2]. However, deploying IDS solutions in resource-constrained environments, such as Community Wireless Networks (CWNs), presents significant challenges. Unlike conventional networks with substantial computing resources, CWNs are inherently low-resource networks and are characterized by limited processing power, memory, storage capacities and bandwidth [3]. These resource limitations significantly affect network management and security operations, rendering many existing IDS designs impractical to function in such settings due to their high computational and storage requirements. [4]. For instance, signature-based IDSs rely on extensive databases of predefined rules to identify known attacks, requiring substantial storage resources. In contrast, anomaly-based IDSs normalize network behavior and flag deviations as malicious, but struggle with the highly dynamic traffic and class imbalance typical of such environments. These limitations reduce detection accuracy and increase missed or novel attacks, hindering IDS effectiveness in low-resource and unconventional settings [5].

Artificial Intelligence (AI) has become instrumental in designing effective IDS classifiers due to its capacity to model complex, high-dimensional traffic patterns and support real-time detection in dynamic environments [6,7]. A typical ML-based IDS pipeline consists of three stages: (1) traffic capture and analysis, (2) feature extraction, and (3) feature selection and attack classification using ML/DL models. In the first stage, network traffic is captured, commonly in PCAP format using tools such as NetFlow, Wireshark, or tcpdump. The traffic is then preprocessed, labeled, aggregated into flows, and transformed into a structured tabular dataset (e.g., CSV) containing extracted features. These features subsequently serve as inputs for training and constructing security ML models. Despite the potentials of ML-based intrusion detection system (ML-IDS) models, pipeline complexities, particularly the traffic capture and feature extraction, often introduce considerable computational overhead and processing latency. To circumvent such labor-intensive feature engineering processes, researchers have lately resort to automated-based feature extraction tools such as CICFlowMeter, NetFlow, Argus. Consequently, public datasets with pre-engineered features (e.g UNSW-NB15, CIC-IDS2017, and InSDN) have accelerated IDS research by reducing the significant effort required for traffic collection and feature engineering [8, 9]. Observably, models trained on these datasets frequently exhibit limited generalization when deployed in live or heterogeneous network environments [10–13]

In this investigation, Rosay et al. [14] identified miscalculations and inconsistent flag representations in datasets generated using CICFlowMeter feature extraction tool, specifically CIC-IDS2017, raising broader reliability concerns. Inaccurate or inconsistent features can introduce bias and reduce effectiveness of ML-IDS against certain attacks. These concerns are particularly worrying for benchmark datasets such as InSDN, which was created to address the scarcity of SDN-specific datasets for ML–based IDS modeling, as it was generated using CICFlowMeter, and has a significant number of inheritable attributes from CIC-IDS2017 [15]. Moreover, InSDN, and other datasets purportedly constructed for SDN environments omit features intrinsic to SDN architecture such as control plane–data plane interactions, controller state information, and flow-rule dynamics [16,17]. These intrinsic features are essential for accurately characterizing SDN behavior and detecting attacks that exploit SDN-specific mechanisms such as abnormal path manipulation or flow-rule poisoning. Although models trained on such datasets frequently report strong experimental performance, such results may not reliably translate to real-world SDN deployment, where operational dynamics and architectural dependencies differ significantly. By design, SDN separates network control logic (control plane) from packet forwarding (data plane), with coordination typically managed mainly through OpenFlow protocols [18]. This architectural decoupling introduces distinct behavioral patterns and unique vulnerabilities that must be explicitly incorporated into feature extraction processes and IDS design. Additionally, the complexity associated with existing feature extraction mechanisms, in terms of computational resources and time demands, makes it extremely difficult for their practical deployment in live networks, especially environments with limited resources, let alone integrate into a full end-to-end IDS pipeline [19,20]. In this study, we propose an efficient feature extraction framework designed to operate within low-resource Software-Defined Networking environments and extract traffic features in real time. The framework captures live network traffic and systematically generates structured flow-level and session-based features suitable for downstream analysis and machine learning integration. The proposed framework, termed AnDO (Argus + nDPI + ONOS), leverages the comprehensive visibility of SDN architecture through three integrated components: the ONOS Intelligence Engine for control-plane awareness, Argus for robust flow generation, and nDPI for deep packet inspection and protocol classification. Together, these components extract over 50 network traffic features per flow, largely aligned with the UNSW-NB15 dataset while incorporating novel SDN-specific features currently unavailable in existing datasets. The specific contributions of this research are as follows:

1. A real-time feature extraction framework that processes live network traffic to generate behavioral features directly suitable for ML applications.
2. A technique for producing UNSW-NB15-compatible datasets enriched with SDN-specific features contextualized for low-resource network SDN environments
3. An integrated lightweight monitoring framework combining ONOS control-plane intelligence, Argus flow generation, and nDPI protocol classification for comprehensive network analysis
4. self-contained data generation methodology that performs end-to-end feature extraction within the monitoring pipeline, eliminating external database dependencies to enhance portability and reduce system overhead.

2 Related Work

A review of prior works on feature engineering and benchmark dataset for ML-IDS. B. Lypa et al. [1] highlighted the importance of AI-based intrusion detection and conducted a comparative analysis of feature extraction tools for flow-based ML IDS development. Their study emphasized that model performance depends heavily on the chosen feature set, which can significantly affect results even when derived from identical raw traffic. Kun Young et al. [13], propose a systematic framework, along with an open-source toolkit and Python library (NetML), to facilitate feature extraction from network traffic and enable end-to-end evaluation using modern novelty detection models. They release a complete pipeline supporting novelty detection tasks and validate it across five networking scenarios, including attack and novel device detection. Their study provides practical insights and guidelines on selecting appropriate features for different detection contexts. Alshammari et al. [21] proposed an ML-based IDS framework using the ISOT-CID dataset to detect malicious cloud network traffic. They added flow-based, time-interval, and payload-length features to improve accuracy, showing that effective feature extraction is critical. However, the large-scale dataset increases complexity and may hinder real-time deployment.

Wan et al. [20] present CATO, a framework for end-to-end optimization of ML-based traffic analysis pipelines. Using multi-objective Bayesian optimization with a realistic pipeline generator and profiler, CATO constructs serving pipelines that jointly optimize accuracy and system performance. Experiments on live and offline traffic demonstrate improved inference latency, throughput, and execution time while preserving or enhancing predictive performance. To address the heterogeneity of feature sets and the lack of standardization in IDS model design and evaluation, Sarhan et al. [22] proposed a standardized NIDS feature set derived from the NetFlow protocol. They consolidated four benchmark datasets using UNSW-NB15, BoT-IoT, ToN-IoT, and CSE-CIC-IDS2018 into a unified corpus using their proposed NetFlow-based features. Experimental evaluation with an Extra Trees classifier showed that this 43-feature set achieved superior classification performance (measured by F1-score) compared to the original proprietary feature sets across all datasets, in both binary and multi-class settings. In this study, Russo et al. [23] utilized Zeek for feature extraction and applied machine learning models to analyze the CICModbus 2023 dataset. Algorithms such as K-means and Isolation Forest were used to detect traffic patterns and anomalies, while a neural network achieved an accuracy of 84.21%. The findings highlight the effectiveness of integrating traditional network analysis tools with ML techniques to improve malicious activity detection. CICFlowMeter is a widely used network flow generation tool that extracts over 80 statistical features for ML-based intrusion detection [24]. Producing benchmark datasets including InSDN and supports feature selection, custom extensions, and configurable timeouts with CSV output. However, Rosay et al. [14] identified critical issues in datasets generated with the tool relating to miscalculations, missing IP addresses and incomplete flag information, raising concerns about their reliability. Indeed Liu et al. identified feature-generation (6.67%) and labeling (7.53%) errors in the CIC-IDS2017 and CSE-CIC-IDS2018 datasets generated by CICFlowMeter. After correcting these issues and reverse-engineering the datasets, they produced improved representations that enhanced attack detection performance. Datasets created by Sarica et al. [5] and Elsayed et al. [15] sought to address the scarcity of SDN-specific data for ML-based IDS modeling. These studies utilized SDN testbed environments and feature extraction tools such as CICFlowMeter to generate network traffic features. Nonetheless, the resulting datasets still lacked SDN-intrinsic attributes, including control–data plane interactions, controller state information, and flow rule dynamics.

Moustafa et al. [8], present the design and evaluation of the UNSW-NB15 dataset and compare it with established benchmarks such as KDDCup'99 and NSL-KDD. To construct the dataset, they propose a feature extraction framework that integrates Zeek and Argus, generating flow-based attributes that comprehensively characterize network behavior. The resulting dataset combines real modern benign traffic with contemporary synthesized attack activities, using both existing and novel techniques to derive its feature set. This dataset is widely regarded within the research community as a reliable and high-quality benchmark for intrusion detection system (IDS) modeling, owing to its realistic traffic composition, comprehensive feature set, and balanced representation of modern attack scenarios. Our work closely aligns with this approach in that the extracted features are largely compatible with those of the UNSW-NB15 dataset, though the methodologies differ significantly. In UNSW-NB15, feature extraction relies on an external SQL database to store and correlate outputs from Argus and Bro-IDS (Zeek), particularly for connection tracking and stateful feature derivation. In contrast, our approach implements an end-to-end feature extraction pipeline directly

within a live network environment without external database dependencies. We leverage the temporal retention capabilities of the ONOS controller, where a custom connection tracking engine maintains behavioral context and computes temporal time-window–based statistics for each flow's contextual relationships. This approach consequently optimizes resource utilization and eliminates pipeline complexity.

3 AnDO System Design

This study introduces AnDO, a lightweight SDN-aware framework for generating accurate, scalable, and semantically rich ML-ready features through multi-plane network data extraction and integration. Leveraging the SDN controller's global visibility, AnDO obtains contextual intelligence on network behavior, including flows, protocols, links, paths, devices, and connection dynamics. The framework extracts flow-level features aligned with UNSW-NB15 [8] while incorporating SDN-specific attributes using Argus, nDPI, and ONOS. In addition, lightweight connection-tracking and statistical aggregation mechanisms are employed to derive contextual behavioral features suitable for resource-constrained ML-based IDS environments. Subsequent sections present the design and architecture of AnDO.

3.1 System Architecture

The proposed framework employs a tri-planar integration model that unifies data-plane traffic capture, control-plane intelligence, and real-time feature analysis. As illustrated in Figure 1, the system operates across three synchronized planes aligned with the canonical SDN decomposition; the data plane, the control plane and the application plane (analysis pipeline), all converging at the Integrated Feature Engine.

- *Data Plane*: The Mininet-based SDN environment generates and forwards live network traffic through Open vSwitch instances configured with OpenFlow 1.3. A strategic monitoring point at the core switch (s3-eth1) captures all transit flows between network branches without interfering with production forwarding.
- *Control Plane*: The ONOS controller maintains the global network view, including topology state, device inventories, host locations, and installed flow rules. AnDO accesses this intelligence exclusively through northbound REST APIs, preserving the architectural separation between control and application planes.
- *Application Plane*: The AnDO framework resides entirely within this plane, operating as a passive observer that consumes but does not modify network state. The Integrated Feature Engine synthesizes multi-source data into standardized machine learning-ready output.

Each plane contributes a distinct class of information, and no plane subsumes the responsibilities of another. The data plane provides raw packet observations, the control plane supplies contextual network state, and the application plane performs extraction and analysis. These components converge at the Integrated Feature Engine, which fuses packet-level metrics, protocol labels, and SDN state information into comprehensive feature vectors.

3.2 Feature Extraction Process

AnDO operates entirely within the application plane, consuming information from both the data and control planes without modifying their behavior. Internally, the framework is structured into two tightly coupled components, the Analysis Pipeline and the Integrated Feature Engine, each with a clearly defined functional role. Together, these components extract three distinct classes of features from the network: stateless flow metrics, stateful connection-tracking attributes derived from temporal analysis, and ONOS-specific contextual features harvested from the SDN control plane:

(a) Analysis Pipeline The Analysis Pipeline is internal to the AnDO framework and performs plane-local feature extraction using only data-plane observations. It converts raw packet streams into structured, semantically enriched flow records while preserving connection-level context. The pipeline consists of four sequential operations:

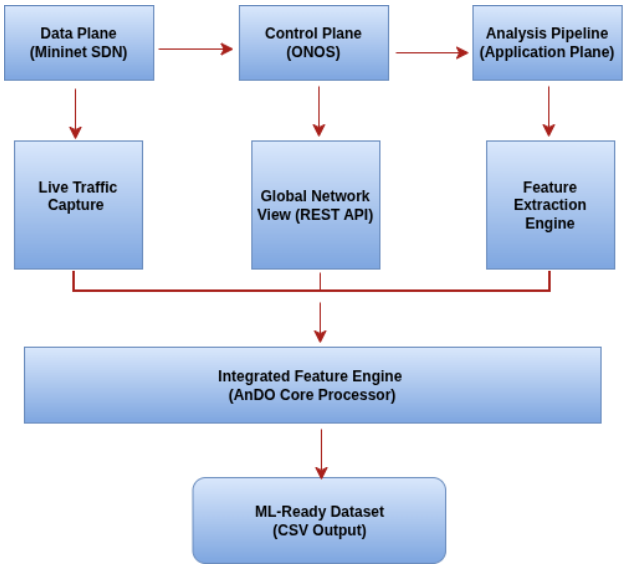

Fig. 1. Overview of AnDO Architectural Framework

1. *Flow parsing and aggregation*: Raw packets are aggregated into bidirectional flow records using Argus, identified by the standard five-tuple of source and destination IP addresses, source and destination ports, and transport protocol. Argus computes core flow statistics, including duration, packet and byte counts in each direction, inter-packet timing variations, and transport-layer state indicators derived from TCP flags and session semantics.
2. *Service identification*: Each flow is enriched using nDPI, which assigns coarse-grained application and protocol labels such as HTTP, DNS, or TLS through signature-based and header-inspection techniques. This classification operates without payload decryption, preserving privacy while providing essential service-level context.
3. *Statistical feature aggregation*: Derived metrics including packet rates, byte rates, mean packet sizes, and directional ratios are computed from the raw Argus telemetry. These aggregated statistics characterize macroscopic traffic behavior and normalize variability across flows of differing durations and sizes, enabling consistent comparison.
4. *Connection tracking*: Lightweight local state is maintained within a sliding-window data structure to compute connection tracking features *(ct_* features)*, including source-destination interaction frequency, port reuse patterns, and temporal locality metrics. The detailed mechanism for these calculations is presented in the following subsection.

Connection Tracking : Connection tracking features are stateful, flow-level metrics that capture the temporal behavior and inter-relationship of network connections over time. They model how connections evolve and interact with other flows, enabling detection of coordinated, repetitive, or malicious activity patterns. To derive these features, a custom stateful tracking engine was designed using a sliding-window temporal model, implemented via the *count_ connections()* method, which performs pattern matching over recent flow records maintained by the *FlowHistory* structure *(pseudocode presented in Algorithm 1 below)*. Unlike static dataset computation, this approach preserves a dynamic context of recent flows typically within a 60-second window to compute real-time relational metrics. Within AnDO's multi-stage feature extraction pipeline, the Feature Extraction Engine computes the complete set of UNSW-NB15 ct_* features using FlowHistory as its temporal analysis core. The FlowHistory class maintains a stateful sliding window of recent flows, enabling computation of all seven connection tracking metrics, which quantify source–destination associations, port reuse behavior, and communication sequencing. For each incoming flow, the engine queries this maintained temporal state to compute the corresponding ct_* features. The output is a set of enriched flow records containing statistical, service-level, and connection-aware attributes derived exclusively from the data plane, before SDN control-plane semantics are incorporated.

SDN-Specific Extended Features : The information returned by the controller is parsed and deterministically mapped into ten SDN-native feature dimensions. From the flow rule itself, the engine extracts rule priority and OpenFlow table identifier, capturing the control logic's precedence and pipeline placement. Controller-maintained statistics including flow duration, packet count, and byte count—provide a control-plane-validated perspective on flow longevity and volume, independent of packet-level data-plane observation. The identifier of the forwarding switch and the ingress port recorded by ONOS spatially anchor the flow within the physical network fabric. The number and type of forwarding actions associated with the rule quantify action complexity. Analysis of the retrieved topology enables computation of estimated path length, defined as the number of switch hops between the flow's endpoints as determined by the controller's routing decisions. Finally, flow age, as maintained by the controller, reflects the temporal persistence of the rule within the control plane, resulting ten (10) extracted features (*(pseudocode presented in Algorithm 2 below)*. These ten features collectively enable downstream machine learning models to detect anomalies originating in the control plane itself, including flow rule poisoning, abnormal path manipulation, or divergences between controller intent and data-plane behavior, threats that remain fundamentally invisible to traditional, non-SDN-aware monitoring approaches.

Algorithm 1 FlowHistory (Connection Tracking)

init: $w \leftarrow 300s$, $Q \leftarrow \emptyset$
function cnt(f): *// count matching flows*
  $c \leftarrow 0$; **for** $r \in Q$: **if** match(r, f): $c \leftarrow c + 1$
  return c
function get_ct($flow$):
  return [$state(flow)$, cnt($\{dst : flow.dst, t : 100s\}$),
  cnt($\{src : flow.src, t : 100s\}$), cnt($\{src : flow.src, port : flow.dst\_port, t : 100s\}$),
  cnt($\{dst : flow.dst, port : flow.src\_port, t : 100s\}$), cnt($\{src : flow.src, dst : flow.dst, t : 100s\}$)]
function add($flow$): Q.append($flow$); age_out()

Algorithm 2 ONOSEngine (SDN-Specific)

init: api, $timeout \leftarrow 5s$
function refresh():
  if elapsed() $> timeout$: $F \leftarrow$ GET(/flows); $T \leftarrow$ GET(/topology)
function ext($flow$):
  refresh(); $r \leftarrow$ match($flow, F$)
  return [$r.prio$?0, $r.table$?-1, now()$-r.install$?0,
  $r.pkts$?0, $r.bytes$?0, $flow.switch$, $r.port$?0,
  count($r.actions$)?0, path_len($flow, T$), now()$-r.install$?0]

Algorithm 3 Main Feature Extraction Pipeline

$flows \leftarrow$ nDPI.classify(Argus.convert(capture($iface$, $10s$)))
for each $flow \in flows$: write_csv(merge($flow$, FlowHistory.get_ct($flow$), ONOSEngine.ext($flow$))); FlowHistory.add($flow$)

(b) Integrated Feature Engine : The SDN-derived attributes, together with the data-plane features extracted by the Analysis Pipeline, are fused within the Integrated Feature Engine to produce the final 50-feature vector. Within AnDO, this engine performs multi-plane feature fusion by combining data-plane observations with control-plane intelligence, yielding a unified representation of network behavior through four core functions:

- *Control-plane correlation:* For each flow generated by the Analysis Pipeline, the engine issues on-demand queries to the ONOS northbound REST API to retrieve relevant flow rules, device metadata, and topology information.
- *Flow-rule alignment:* Observed flows are mapped to their corresponding controller-installed rules using match fields and switch context, establishing a direct link between packet-level behavior and controller intent.
- *SDN-specific feature injection:* Control-plane attributes, including rule priority, table placement, rule lifetime, controller counters, ingress switch and port identifiers, and topology-derived path properties, are extracted and appended to the feature vector, encoding SDN logic and structural constraints.
- *Final normalization and structuring:* All data-plane and control-plane features are normalized, validated, and organized into a consistent schema to ensure dimensional homogeneity and eliminate ambiguity from heterogeneous sources.

The output is a unified, ML-ready dataset, formatted as CSV, where each row represents a single flow enriched with statistical, semantic, temporal, and SDN-specific attributes for downstream analytical tasks.

4 Methodology

The proposed AnDO framework was deployed and evaluated within a low-resource SDN testbed implemented using Mininet and ONOS. Mininet, a Python-based SDN network emulator, was used to create and manage virtual network topologies comprising hosts, switches, links, and controllers within isolated Linux namespaces [18]. The testbed was deployed on Ubuntu Linux to emulate realistic SDN behavior under resource-constrained conditions. ONOS, an open-source SDN controller, was adopted to provide centralized control, network visibility, and control-plane intelligence within the virtualized environment. Network traffic was introduced into the testbed by replaying PCAP traces from a low-resource Community Wireless Network (CWN) dataset using TCPreplay [3]. The resulting traffic was captured and processed through the AnDO pipeline to generate structured ML-ready flow records. Experimental evaluation focused on both operational and computational performance. Operational analysis considered CPU utilization, memory consumption, throughput, overhead, stability, and scalability, while computational analysis examined temporal and spatial complexity under increasing traffic and processing demands. Subsequent sections present the experimental evaluation, performance analysis, and findings.

4.1 Experimental Procedure

To verify the functionality and operability of the proposed AnDO framework, we conducted experiments within a live SDN environment. The procedure comprises two main stages: (1)network environment setup including topology design, and (2)traffic replay and feature extraction.

Network Environment and Topology : Two virtual machines were configured on a Windows 10 host machine running VirtualBox, with 50 GB storage, 6 GB RAM, and 3 CPU cores running to emulate resource-constrained CWNs:

- *SDN Environment:*(1) A Mininet VM (20.04.1) with 25 GB storage, 2.5 GB RAM, and 1 CPU core, pre-installed with OpenFlow libraries; (2) An Ubuntu VM hosting the ONOS container, allocated 25 GB storage, 3 GB RAM, and 2 CPU cores. The hardware resources were deliberately allocated to emulate a low-resource environment. ONOS was installed on the Ubuntu VM to provide SDN control-plane functionality. Both VMs were configured with dual network adapters: NAT for internet access and Host-Only for internal communication. The Ubuntu VM assigned NAT to enp0s3 and Host-Only to enp0s8; the Mininet VM assigned NAT to eth0 and Host-Only to eth1. This setup enabled secure SSH communication between VMs, remote access to both environments, and connectivity to the host Windows machine. NAT allows VMs to share the host's IP for external access, while Host-Only creates an isolated private network for VM-to-VM and VM-to-host communication.
- *Network Topology*: A custom network topology *c_ topo.py* was created using Python scripts and associated libraries. The network topology comprises two(2) types of devices: switches and hosts, denoted by the letters 's' and 'h', respectively. The custom topology follows a tree structure, consisting of four (4) switches (s1–s4) and eleven (11) hosts (h1–h11) interconnected to form a hierarchical network. The root switch, s1, connects vertically to s3, which in turn connects horizontally to s2 and s4 on its left and right, forming the intermediate layer of the topology. The host nodes are distributed across the switches as follows: four (4) hosts connected to s2, three (3) to s3, and four (4) to s4. The s1 switch serves as the control-plane connection point, linking directly to the ONOS controller. In this configuration, the host nodes emulate end-user devices such as laptops, desktop computers, smartphones, and tablets, while the switches function as virtual OpenFlow-enabled switches, forming the backbone of the Mininet-based Software-Defined Networking (SDN) environment. Finally, the setup was customized to emulate SDN in Mininet as in [18].

Following the Mininet SDN setup, latest Argus server and client packages were installed on the Mininet VM. Argus is an open-source network monitoring solution where both components function as an integrated unit, complementing each other's roles [25]. The Argus server attaches to the Mininet network interface (eth0) to monitor, capture, and analyze live packet streams, aggregating them into bidirectional flow records. The Argus client then connects to the server, reads these flow records, and outputs them in machine-parsable CSV format. Our AnDO feature extractor receives this output, processes it line by line, converts each flow into a dictionary structure, and feeds it into the FlowHistory class for temporal analysis. Additionally, the latest version of nDPIreader was installed within the Mininet environment to enable deep packet inspection functionality [26]. The tool analyzes packet payloads traversing the network to detect and classify application-layer protocols such as HTTP, SSL, and BitTorrent. Through this inspection process, nDPIreader provides fine-grained visibility into traffic types, enabling accurate identification of services and applications associated with each network flow.

[3] https://github.com/appneta/tcpreplay

Traffic Replay and Feature Extraction: Once the network environment was configured and operational, the feature extraction process proceeded through the systematic pipeline illustrated in Figure 2. The experiment replayed network traffic through the simulated SDN environment to enable real-time feature extraction. ONOS was first initialized on the Ubuntu host machine, followed by the launch of Mininet and deployment of the AnDO framework within the virtualized environment. A target PCAP trace obtained from the iNethi CWN dataset [27] was replayed into a designated interface within the Mininet topology. To preserve privacy and emphasize core communication patterns, the original PCAP traces underwent packet-level rewriting using tcprewrite (package in TCPreplay) [28], where all source IP addresses were remapped to 10.0.0.1 and destination IP addresses to 10.0.0.2. Layer-2 Ethernet addresses were similarly standardized, and packet checksums recalculated. The resulting *inethi_ rewritten.pcap* file was then injected into the SDN data plane using *tcpreplay* via switch interface *s1-eth1*. Simultaneously, the AnDO framework operated passively on the monitoring interface *s3-eth1*, where it observed the sanitized traffic stream, extracted flow-level features, computed temporal context through the *FlowHistory* mechanism, and exported the resulting ML-ready records to CSV format. Tables 1–3 summarize the feature categories generated by AnDO. For brevity, only the connection-tracking and SDN-specific intelligence features are presented in full, as these represent the primary contributions of the framework. The experimental topology employed a centralized monitoring strategy in which all traffic exchanged between Switch 1 (S1) and Switch 2 (S2) traversed the aggregation switch (S3). Positioning the capture interface on s3-eth1 provides a comprehensive vantage point that observes all transit flows between network branches, ensuring feature extraction captures complete network-wide activity.

Fig. 2. Feature Extraction Pipeline

Table 1. Feature Categories and Counts from AnDO Framework

Category	Features Count	Description
Basic Flow Identifiers	6	Core 5-tuple + state
Flow Statistics	8	Volume, timing, service
Network Layer	4	TTL, load metrics
Transport Layer	4	TCP window, sequence
Derived Statistical	4	Means, jitter
Temporal/Timing	5	Inter-packet, RTT estimates
Behavioral	1	Same IPs/ports detection
Connection Tracking	7	UNSW-NB15 stateful features
ONOS-Enhanced	10	SDN-specific novel features
Metadata	1	Timestamp
TOTAL	**50**	**Complete AnDO feature set**

4.2 Evaluation and Results

We examine AnDO's operational performance and computational complexity to establish a complete understanding of its practical viability and theoretical foundations. A custom performance monitoring script, launched automatically alongside AnDO during the feature capture experiments described in the previous section, executes throughout the framework's lifecycle and continuously samples system-level and process-specific resource utilization including CPU and memory consumption at regular intervals. All collected metrics are persistently logged and exported in CSV format for detailed post-execution analysis of computational overhead, resource behavior, and performance stability. Performance logs capture periodic snapshots of overall CPU and memory usage, Python process CPU and Resident

Table 2. Connection Tracking Derived Features

Feature Name	Type	Description	Time Window
ct_srv_src	Integer	Connections to same service from same source	60s
ct_srv_dst	Integer	Connections to same service to same destination	60s
ct_dst_ltm	Integer	Connections to same destination	60s
ct_src_ltm	Integer	Connections from same source	60s
ct_src_dport_ltm	Integer	Connections from source to dest port	60s
ct_dst_sport_ltm	Integer	Connections to dest from source port	60s
ct_dst_src_ltm	Integer	Connections between src-dst pair	60s

Table 3. SDN-Specific Control Plane Intelligence (ONOS) Derived Features

Feature Name	Type	Description	ONOS Source
onos_flow_priority	Integer	Flow rule priority (0–65535)	Flow entries
onos_table_id	Integer	OpenFlow table ID	Flow entries
onos_flow_duration	Float	How long flow exists in ONOS (s)	Flow statistics
onos_packet_count	Integer	Packets counted by ONOS	Flow statistics
onos_byte_count	Integer	Bytes counted by ONOS	Flow statistics
onos_switch_id	String	Which switch handled flow	Device info
onos_ingress_port	Integer	Input port on switch	Flow criteria
onos_flow_actions	Integer	Number of actions in flow rule	Flow treatment
onos_path_length	Integer	Estimated network hops	Topology analysis
onos_flow_age	Float	Time since flow installed	Flow duration

Set Size (RSS), timestamps, and derived durations. The *Time_ dur* and *Time_ dur(sec)* columns normalize these into a monotonic execution timeline starting from process launch, while the delta column reflects sampling irregularity from OS jitter, sleep drift, or CPU contention rather than computational delay. Each log row represents a sampling interval rather than an individual flow, with each interval capturing batch processing of thousands of flows to enable high throughput within limited samples. Table 4 presents a sample of the evaluation metrics, derived columns, and performance results.

Operational Performance Results : This evaluation quantifies the resource utilization across four primary dimensions: memory allocation (measured in megabytes of RSS), computational processing (expressed as percentage of CPU capacity), temporal efficiency (quantified in seconds per processing cycle), and storage requirements (documented in disk input/output operations). Specifically, the assessment examines memory utilization stability through statistical analysis of mean consumption and variability, CPU efficiency through processing load distribution between system-wide mean, and process-specific mean utilization. Furthermore, we examine throughput capacity through flow processing rates, performance overhead through system resource stratification analysis, and longitudinal stability through variance coefficients in the operational life-cycle window. Summary of resource metrics results shown in Table 5.

Table 4. Sample of system_level performance metrics of Feature Extraction algorithm (AnDO)

Indx	Total_CPU(%)	Total_RAM(mb)	Proc_CPU(%)	Proc_Mem(mb)	Elapsed_T(sec)	Time_dur(delta)	Time_dur(sec)	Time_int(sec)
0	100.0	136	0.0	19.09	1	0 days 04:11:23	15083.0	0.0
1	46.7	175	124.0	66.18	2	0 days 04:11:25	15085.0	2.0
2	94.1	180	41.3	66.18	3	0 days 04:11:26	15086.0	1.0
3	94.5	180	31.0	66.18	5	0 days 04:11:27	15087.0	1.0
4	100.0	179	24.8	66.18	6	0 days 04:11:29	15089.0	2.0
...	...	...	...	...	...	...	...	...
2893	6.2	178	5.2	71.73	3834	0 days 05:15:17	18917.0	1.0
2894	0.0	174	5.1	71.73	3835	0 days 05:15:18	18918.0	1.0
2895	6.2	174	5.1	71.73	3837	0 days 05:15:19	18919.0	1.0
2896	11.8	174	5.1	71.73	3838	0 days 05:15:21	18921.0	2.0
2897	11.8	174	5.1	71.73	3839	0 days 05:15:22	18922.0	1.0

Table 5. Summary of Resource Metrics - Experimental Evaluation

Resource Metric	Mean	Std Dev	Min	Max
Total CPU Usage (%)	97.81	9.03	0.00	100.00
Process CPU Usage (%)	4.61	2.50	0.00	124.00
Total RAM Usage (MB)	181.07	2.95	136.00	192.00
Process Memory Usage (MB)	71.05	1.15	19.09	71.73

Computational Complexity Results : The computational complexity of AnDO was empirically evaluated through sustained experimentation processing 25,684 network flows over a 64-minute period. The performance dataset comprises 2,898 valid samples, with a single initialization outlier excluded to preserve analytical accuracy, providing a reliable basis for characterizing runtime behavior under realistic conditions. We evaluate AnDO's computational and space complexity by aligning theoretical Big-O bounds with empirical runtime behavior. Time complexity is modeled as O(n) for linear stages and O(n log n) for sorting or aggregation, while space complexity is modeled as O(n) for flow state storage and O(1) for streaming components. These bounds are validated against empirical measurements as flow volumes scale to thousands of flows. Each pipeline component ONOS controller interactions, flow parsing, feature enrichment, and correlation processing is analyzed both independently and within the integrated cycle to identify dominant cost drivers and confirm theoretical expectations.

Table 6. Summary of computational (Time/Space) complexity analysis: Big-O Notation

Operation	Function/Method	Storage Location	Time Complexity	Space Complexity	Variables Description
ONOS Global View	refresh_global_view()	ONOSIntelligenceEngine instance	$O(F)$	$O(F+H+D)$	F = ONOS flows, H = Hosts, D = Devices
Flow Processing	_get_enhanced_flows()	flow_cache dictionary	$O(F)$	$O(F)$	F = ONOS flows
Flow Correlation	_find_matching_onos_flow()	flow_cache lookup	$O(F)$	$O(1)$	F = ONOS flows
Path Calculation	_calculate_path_length()	host_cache + device_cache	$O(H)$	$O(1)$	H = Number of hosts
Argus Parsing	parse_argus_data()	DataFrame (temporary)	$O(N)$	$O(N)$	N = Captured flows
Feature Calculation	calculate_unsw_features()	DataFrame columns	$O(N)$	$O(N)$	N = Current flows
Service Mapping	Service lookup in pipeline	service_map dictionary	$O(N \times S)$	$O(S)$	S = Service entries
ONOS Enhancement	correlate_flow_with_onos()	ONOS features DataFrame	$O(N \times F)$	$O(N)$	$N \times F$ product
Connection Tracking	add_connection_tracking_features()	UNSWConnectionTracker.flows deque	$O(N \times T)$	$O(T)$	T = Historical flows
Packet Processing	capture_flows() + nDPI	Temporary PCAP file	$O(P)$	$O(P)$	P = Packets
Single Cycle	process_capture_cycle()	All structures combined	$\boldsymbol{O(N \times F + P)}$	$\boldsymbol{O(F+T+S+N)}$	Combined operations
Continuous Loop	main() while running	Persistent structures	$\boldsymbol{O(C \times N \times F)}$	$O(F+T)$	C = Number of cycles

Time Complexity, Big-O Notation: Several pipeline operations execute in constant time O(1) through deliberate architectural design and bounded data structures, as summarized in Table 6. ONOS controller interactions for global state refresh occur at fixed 5-second polling intervals and operate on a refreshed, rather than cumulative, cache of controller flows. Although the refresh process traverses the current flow set, the cache size remains strictly bounded and independent of flow arrival rates, ensuring constant contribution per execution cycle. Similarly, connection-tracking primitives including flow insertion and eviction achieve amortized O(1) complexity through time-bounded deque structures with fixed capacity. Lookup operations for host identification, topology attributes, and path-length computation are performed over pre-populated bounded in-memory structures, maintaining constant-time access. Empirical observations confirm no measurable correlation between these operations and batch size or cumulative flow volume, confirming they do not contribute to asymptotic complexity growth.

Linear-Time Operations O(O(N), O(F), O(T)) : Observably, as shown in Table 6, several core components exhibit linear complexity with respect to their immediate inputs:

- *Argus Flow Parsing (O(N))*: Flow parsing and feature extraction operate linearly over the number of captured flows per batch. Each flow is processed exactly once per cycle, resulting in predictable linear scaling.
- *ONOS Flow Correlation (O(F))*: Correlation between extracted flows and ONOS controller state requires a linear scan of the current ONOS flow cache. However, since this cache is replaced rather than accumulated and bounded to a fixed size, the operation behaves as constant time per cycle in practice.
- *Connection Counting (O(T)*: Connection statistics are computed by scanning a time-bounded tracking window representing flows observed within a fixed temporal horizon (e.g., 60 seconds). This results in linear complexity with respect to the window size T. As T is strictly bounded by design, its cost remains stable across execution.

– *Dominant Composite Operations O(N × T)*: The primary computational cost per cycle arises from computing connection-based features for each flow against a time-bounded tracking window, yielding a theoretical complexity of O(N × T). Since the tracking window size T is fixed and independent of traffic volume, this reduces to effective linear complexity O(N) in practice. Empirical results confirm that per-cycle execution time scales proportionally with batch flow volume, with no superlinear growth observed.

Aggregating all pipeline stages into a single execution cycle yields theoretical time complexity of $O(N \times T \times F + P)$. Since T and F are bounded constants and P scales with N, this simplifies to $O(N)$ for *combined per-cycle complexity*. Empirical throughput measurements confirm this linear bound, demonstrating stable per-flow processing costs throughout execution.

Space Complexity, Big-O Notation : Memory utilization within AnDO exhibits effective O(1) space complexity, enforced through deliberate architectural constraints on all long-lived data structures:

– *Connection tracking* is implemented using a time-based sliding window with a fixed 60-second temporal boundary, ensuring that only recent flow state is retained while older entries are continuously expired. This design guarantees that the number of tracked connections remains bounded over time, independent of total execution duration or cumulative traffic volume.
– *ONOS flow intelligence*: Similarly, is maintained through complete cache refresh cycles at fixed 5-second intervals, replacing prior controller state rather than accumulating historical flow records. This refresh strategy ensures that the ONOS flow cache size is strictly limited to the current controller view, preventing unbounded growth as new flows are observed.
– *Service mappings*: derived from nDPI rely on a static, finite lookup table, while intermediate data structures such as Pandas DataFrames used for per-cycle feature aggregation are instantiated on demand and explicitly released at the end of each processing cycle. These temporary allocations do not persist across cycles and therefore do not contribute to long-term memory growth.

These mechanisms ensure stable, bounded memory consumption decoupled from cumulative flow count, validating the algorithm's effective O(1) space complexity.

5 Analysis and Discussion

This section analyzes the proposed feature extraction framework and discusses the implications of the evaluated results. The discussion focuses on two main aspects: the operational performance of AnDO, including resource utilization (CPU and memory), throughput efficiency, system overhead, stability, and scalability; and the algorithmic time complexity of the framework, particularly its scaling behavior as input size increases.

Operational Performance

– *CPU Utilization*: CPU utilization profile exhibits a clear transition from an initialization-intensive phase to a stable steady state, as shown in Table 5. During startup, system-wide CPU utilization reached full saturation (mean: 97.81%), corresponding to pipeline initialization, library loading, ONOS REST interactions, and initial flow processing. The Python feature extraction process peaked at 72% CPU usage, with occasional values exceeding 100% (maximum: 124%), reflecting multi-core accounting during parallel execution, Figure 3*(a)*. Notably, 96.58% (2,799 samples) of recorded measurements exceeded 90% CPU, concentrated within this initialization interval. Following system stabilization after data structures, caches, and ONOS state queries were established, CPU demand decreased substantially. The process-level CPU mean reduced to 4.61% (SD: 2.50%), while overall system utilization stabilized within the 12–15% range. The coefficient of variation (9.24%) indicates controlled and predictable fluctuations in steady state. Monitoring frequency averaged 0.75 Hz with consistent 1.32-second sampling intervals, confirming sustained operational stability. Overall, the results demonstrate an expected high-cost initialization phase followed by efficient steady-state execution, reflecting effective caching, communication stabilization, and reduced per-cycle processing overhead.
– *Memory Utilization*: The AnDO feature extraction pipeline demonstrates stable and efficient memory management throughout the 64-minute monitoring period. Total system memory usage remained low, with a mean of 181.07 MB and a small standard deviation of 2.95 MB, indicating predictable allocation behavior. Memory consumption was confined within a narrow range of 136.00 MB to 192.00 MB, yielding a bounded variation of 56 MB. At peak usage (192 MB) relative to the available 2,464 MB, the system utilized only 7.8% of total RAM, maintaining substantial operational headroom and showing no signs of memory pressure. At the process level, the Python-based AnDO engine maintained an average resident memory usage of 71.05 MB (SD: 1.15 MB), as

depicted in Table 5 and Figure 3*(b)*. The footprint increased from 19.09 MB during initialization to a stable maximum of 71.73 MB during active processing, after which it remained constant with no observable drift or spikes. No samples exceeded this maximum, indicating deterministic memory allocation with clearly bounded limits and no evidence of leakage. Overall, the pipeline accounts for approximately 2.9% of total system memory capacity during steady operation. The sustained stability under continuous and increasing flow processing confirms efficient allocation/deallocation mechanisms and a scalable memory architecture suitable for long-duration monitoring deployments.

- *Throughput*: Over 3,839 seconds, the pipeline processed 25,684 flows, each with 50 features, yielding an average throughput of 6.69 flows per second. This performance is noteworthy given AnDO's integrated pipeline including packet capture, flow reconstruction, multi-dimensional feature extraction, ONOS REST interactions, and feature consolidation, demonstrating a balance between analytical depth and computational efficiency. Throughput aligns with CPU behavior: system-wide CPU peaks during initialization (library loading, controller setup, first-batch processing) but stabilizes at lower levels thereafter while throughput remains steady. Memory usage is similarly stable, with system RAM averaging 176.25 MB (SD: 3.19 MB) and the Python process 71.05 MB (SD: 1.63 MB), showing no growth with increasing flows. As represented in Figure 3*(f)*, correlation analysis indicates decoupled resource usage: near-zero correlation between Python memory and system CPU (-0.04), moderate positive correlation with system RAM (0.36), and slight negative correlation between system and process CPU (-0.11). Sustained throughput is limited by algorithmic complexity rather than resource saturation, with CPU and memory comfortably within capacity and monitoring overhead negligible. These results confirm that AnDO maintains consistent throughput and efficient resource utilization, suitable for long-duration, high-volume network monitoring.
- *Overhead Analysis*: The performance data indicates a clear separation between system-wide resource consumption and Python process utilization, reflecting infrastructure-level overhead within the SDN emulation environment. The mean total system RAM usage was 176.25 MB (range: 128–189 MB), while the Python resident set size averaged 71.05 MB (range: 12.19–72.06 MB), yielding an approximate 105 MB differential. This overhead corresponds to essential components including the Linux kernel (20–30 MB), Mininet (15–25 MB), Open vSwitch (25–35 MB), Argus services (10–15 MB), tcpdump utilities (5–10 MB), system buffers and cache (15–25 MB), and background services (5–10 MB). The stability of this overhead is confirmed by a low total RAM standard deviation of 3.19 MB over the 3,839-second execution period. A similar stratification is observed in CPU utilization. Total system CPU averaged 97.05%, whereas the Python process accounted for 9.22%, producing an 88-percentage-point differential attributable to orchestration and virtualization overhead. This includes kernel scheduling (10–15%), Mininet simulation (15–20%), Open vSwitch processing (25–30%), TCP/IP stack operations (10–15%), monitoring activities (5–8%), and auxiliary services (3–5%). The slight negative correlation between system and process CPU (-0.11) suggests complementary rather than competing utilization patterns. Overall, Python consumes approximately 9–10% of CPU resources while coordinating core extraction tasks, whereas the remaining capacity sustains virtualization, switching, and networking infrastructure. This distribution confirms an efficient architectural division of labor, supporting stable and sustained pipeline operation.
- *Stability and Scalability*: Monitoring results confirm stable and predictable operation throughout the experimental window. System CPU utilization averaged 97.05% (SD: 7.82%), indicating sustained computational engagement without degradation. Memory usage remained tightly bounded, with total system RAM averaging 176.25 MB (SD: 3.19 MB) and the Python AnDO process maintaining 71.05 MB (SD: 1.63 MB). Sampling intervals were consistent (mean: 1.39 s) with no significant gaps, demonstrating temporal reliability suitable for prolonged SDN experimentation. Distinct execution phases are evident. Initialization periods frequently reached full CPU saturation (100%), driven by Argus parsing, Pandas aggregation, ONOS communication setup, and initial flow processing. Notably, 93.82% of samples (2,824 measurements) exceeded 90% CPU, reflecting operation near the computational limits of the single-processor Mininet VM during peak phases. While expected, this suggests that further scaling, for instance, higher flow rates or reduced polling intervals would require additional resources or targeted optimization of these stages. Despite CPU intensity, memory remained stable within a narrow 128–189 MB range, with no growth or pressure observed. This disciplined memory profile provides a robust baseline for scalability improvements, such as streaming-based extraction or subsystem offloading, without introducing memory bottlenecks.

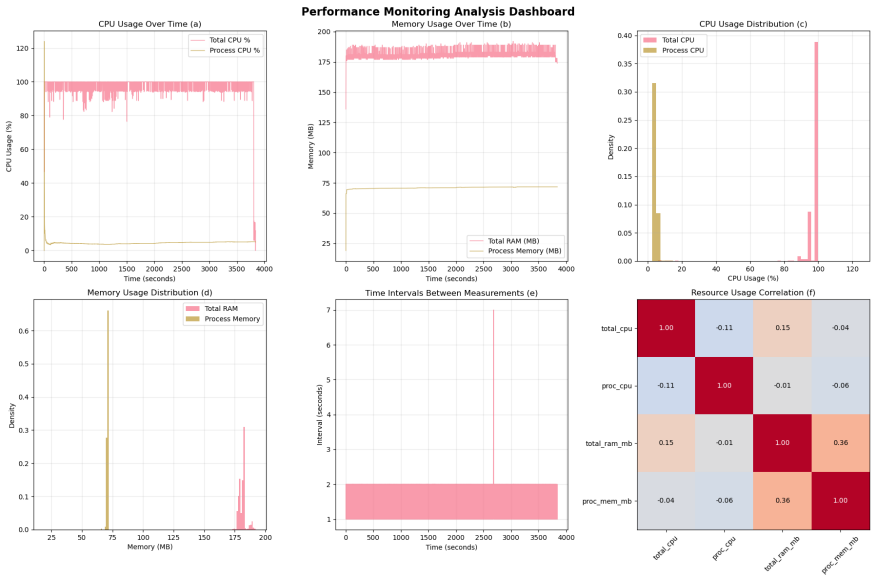

Fig. 3. Experimental Analysis of AnDO feature extraction performance

Computational Complexity

- *Time Complexity: Empirical Validation of Linear O(N) Behavior*: The AnDO feature extraction mechanism demonstrates clear linear time complexity, O(N), where **N** is the number of network flows, Table 7. Empirical measurements show that per-flow processing time remains nearly constant across the 64-minute execution window, confirming proportional scaling with input size *Figure 5(f)*. The observed per-flow processing coefficient is 0.1494 seconds, corresponding to a sustained throughput of 6.69 flows per second (401.5 flows per minute) stable throughout the runtime, as illustrated in *Figure 5(d)*. Temporal correlation analysis yields a low coefficient of 0.224, indicating only a weak dependence of processing time on execution duration. ANOVA confirms statistical significance ($p < 0.001$) across early, mid, and late phases, with CPU utilization increasing modestly from 4.436% to 5.036% (0.6 percentage points, 13.5% relative change), as depicted in *Figure 5(e) CPU Across Phases*. This minor variation does not affect the overall linear scaling, validating that AnDO maintains near-ideal O(N) performance under sustained load.
- *Space Complexity: Empirical Validation of Constant O(1) Behavior*: Memory usage remains stable and bounded, confirming constant space complexity, O(1), independent of the number of flows (*Figure 5(c)*). Key observed statistical analysis include: Average Resident Memory: 71.05 MB (SD: 1.14 MB), Operational Range: 19.09–71.73 MB (span: 52.64 MB), Net Increase: 1.339 MB over 64 minutes, and Coefficient of Variation: 1.61%. The increase from 70.32 MB at startup to 71.66 MB at the end represents only 1.91% growth, reflecting disciplined memory management with no unbounded accumulation. Memory stability is ensured through design constraints: ONOS flow state is maintained in a fixed-capacity cache ($F = 500$), connection tracking uses a time-bounded window ($T = 100$), and protocol classification relies on a constant-size service table ($S = 50$). These mechanisms enforce strict upper bounds, keeping memory usage independent of flow volume or execution duration, consistent with O(1) space complexity and robust AnDO operation.
- *Theoretical–Empirical Complexity Alignment*: Experimental results closely match the theoretical time complexity model of AnDO, expressed as $O(N\times(F+T+S)+P)$. Here, N denotes the number of flows per cycle (6.7), F the active ONOS flows, T the number of connections tracked within a bounded sliding window, S the fixed nDPI service mappings, and P the packet count, which remains proportional to N due to a stable average packets-per-flow ratio. In practice, F, T, and S are explicitly bounded. ONOS flow state is refreshed periodically, replacing rather than accumulating records, ensuring F reflects only active flows. Connection tracking uses a 60-second time-bounded sliding window, constraining T. The service mapping table S is constant by design. Because these parameters do not grow with cumulative input, the expression reduces operationally to O(N), as shown in Table 7. This simplification is empirically validated by the measured per-flow processing coefficient K=0.149432 seconds, capturing both feature extraction and system-level overheads, represented in Figure 4 an *Figure 5(f)*. The close agreement between theory and observation confirms predictable linear scaling with bounded resource usage under sustained operation.

Table 7. Empirical–Theoretical Alignment of Per-Flow Processing Cost

Component	Operation Description	Complexity Class	Avg. Time (s)	Contribution
Argus Parsing	Flow record extraction and basic parsing	$O(N)$	0.050	33.6%
Feature Calculation	Statistical and behavioral feature computation	$O(N)$	0.040	26.8%
ONOS Correlation	Flow enrichment using current ONOS state	$O(N \times F) \rightarrow O(N)$	0.030	20.1%
Connection Tracking	Temporal state updates within sliding window	$O(N \times T) \rightarrow O(N)$	0.020	13.4%
Service Mapping	Protocol and service classification (nDPI)	$O(N \times S) \rightarrow O(N)$	0.010	6.7%
Runtime Overhead	Interpreter, Garbage, System calls	$O(N)$	0.000	-0.7%
Total (Per Flow)	**End-to-end processing cost**	$\boldsymbol{O(N)}$	**0.149**	**100%**

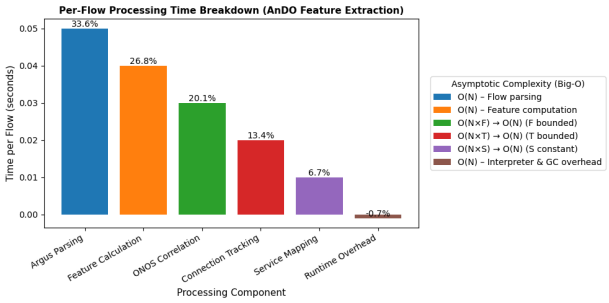

Fig. 4. Per-Flow Processing Time Breakdown (AnDO Feature Extraction)

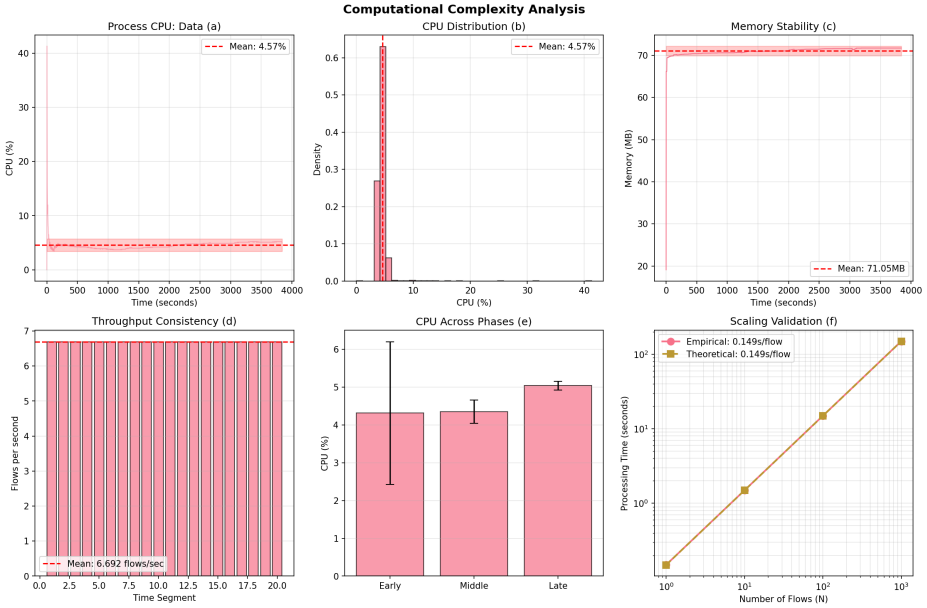

Fig. 5. Computational complexity analysis of AnDO feature extraction framework

Comparative Overview: Table 8 presents a comparative summary of the proposed AnDO feature extraction framework against traditional approaches, including UNSW-NB15 [8] and CICFlowMeter [24]. Unlike CICFlowMeter, which extracts over 80 features from PCAP files, AnDO framework extracts 50 features in real time from live traffic. Although CICFlowMeter offers higher dimensionality, it operates offline and lacks SDN-specific contextual attributes. In contrast, AnDO prioritizes real-time, lightweight extraction of 40 features compatible with UNSW-NB15, plus 10 novel SDN-specific features unavailable in traditional tools.

Table 8. Comparative Analysis Between AnDO and Traditional Feature Extraction Approaches

Metric	Traditional [8, 24]	Proposed (AnDO)	SDN Benefit
Data Collection	Passive sniffing (offline)	Active flow reporting (live)	Complete, switch-direct data
Processing Load	High - all packets	Low - targeted flows	Offloads filtering to data-plane
Stateful Features	Requires separate tracking	Leverages ONOS state	Provides context via API
Deployment	Complex - taps/SPAN	Simple - app-based/controller	Software-defined, flexible
Topology-Awareness	Limited	Full path intelligence	Global network view
Feature Count	80(CICFlowMeter) & 47 [8]	50 (40 UNSW-compatible)	+10 novel SDN-aware

6 Conclusion

This paper presented AnDO, a lightweight feature extraction framework designed for machine learning-based intrusion detection in low-resource Software-Defined Networking environments. The framework integrates Argus for flow generation, nDPI for protocol classification, and ONOS control-plane intelligence within a unified pipeline, extracting 50 per-flow features that maintain UNSW-NB15 compatibility while introducing ten novel SDN-specific attributes. A custom sliding-window connection tracking engine provides temporal context for behavioral analysis, and the stateless ONOS intelligence engine harvests control-plane telemetry without persistent data retention. Experimental evaluation in a virtualized SDN testbed demonstrated AnDO's operational efficiency and stability. The framework sustained processing throughput of approximately 402 flows per minute (6.7 flows per second), with sub-50 millisecond latency per flow, while maintaining minimal and stable CPU and memory utilization throughout extended execution. Resource consumption remained bounded and predictable, with computational overhead largely attributable to the emulation infrastructure rather than the framework itself. Experimental results closely match the theoretical time complexity model, empirically validated by the measured per-flow processing coefficient of 0.149 seconds, capturing both feature extraction and system-level overheads. This close agreement between theory and observation confirms predictable linear scaling with bounded resource usage under sustained operation, validating AnDO as a practical, resource-aware solution suitable for community wireless networks and other constrained environments where traditional feature extraction approaches prove prohibitive. The framework's modular architecture enables systematic, reproducible feature extraction, and its backward compatibility with established datasets facilitates immediate integration with existing machine learning workflows. By exposing SDN-specific attributes previously unavailable in benchmark datasets, AnDO empowers intrusion detection models to identify control-plane anomalies such as flow rule poisoning and topology manipulation that remain invisible to conventional monitoring approaches. Notably, six UNSW-NB15 features, primarily FTP and HTTP connection attributes could not be extracted by AnDO, as they require deep packet inspection tools (e.g., Zeek) for payload parsing beyond the scope of lightweight, real-time processing in low-resource environments. Future work will explore deployment in physical community network testbeds, and the construction of a contextualized, labeled SDN-specific dataset using AnDO in order to advance machine learning-based intrusion detection research.

References

1. Borys Lypa, Ivan Horyn, Natalia Zagorodna, Dmytro Tymoshchuk, and Taras Lechachenko. Comparison of feature extraction tools for network traffic data. *arXiv preprint arXiv:2501.13004*, 2025.
2. Lorenzo Diana, Pierpaolo Dini, and Davide Paolini. Overview on intrusion detection systems for computers networking security. *Computers*, 14(3):87, 2025.
3. Abayomi Agbeyangi and Hussein Suleman. Advances and challenges in low-resource-environment software systems: A survey. *Informatics*, 11(4):90, 2024.
4. Fawaz S Al-Anzi. Design and analysis of intrusion detection systems for wireless mesh networks. *Digital Communications and Networks*, 8(6):1068–1076, 2022.
5. Alper Kaan Sarica and Pelin Angin. A novel sdn dataset for intrusion detection in iot networks. In *2020 16th International conference on network and service management (CNSM)*, pages 1–5. IEEE, 2020.
6. Preeti Mishra, Vijay Varadharajan, Uday Tupakula, and Emmanuel S Pilli. A detailed investigation and analysis of using machine learning techniques for intrusion detection. *IEEE communications surveys & tutorials*, 21(1):686–728, 2018.
7. Shameek Bhattacharjee and Sajal K Das. Building a unified data falsification threat landscape for internet of things/cyberphysical systems applications. *Computer*, 56(3):20–31, 2023.

8. Nour Moustafa and Jill Slay. Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set). In *2015 military communications and information systems conference (MilCIS)*, pages 1–6. IEEE, 2015.
9. Rajasekhar Chaganti, Wael Suliman, Vinayakumar Ravi, and Amit Dua. Deep learning approach for sdn-enabled intrusion detection system in iot networks. *Information*, 14(1):41, 2023.
10. Lisa Liu, Gints Engelen, Timothy Lynar, Daryl Essam, and Wouter Joosen. Error prevalence in nids datasets: A case study on cic-ids-2017 and cse-cic-ids-2018. In *2022 IEEE Conference on Communications and Network Security (CNS)*, pages 254–262. IEEE, 2022.
11. Joffrey L Leevy and Taghi M Khoshgoftaar. A survey and analysis of intrusion detection models based on cse-cic-ids2018 big data. *Journal of Big Data*, 7(1):104, 2020.
12. Nataliya ZAGORODNA, Mariia STADNYK, Borys LYPA, Mykola GAVRYLOV, and Ruslan KOZAK. Network attack detection using machine learning methods. *Challenges to national defence in contemporary geopolitical situation*, 2022(1):55–61, 2022.
13. Kun Yang, Samory Kpotufe, and Nick Feamster. Feature extraction for novelty detection in network traffic. *arXiv preprint arXiv:2006.16993*, 2020.
14. Arnaud Rosay, Eloïse Cheval, Florent Carlier, and Pascal Leroux. Network intrusion detection: A comprehensive analysis of cic-ids2017. In *8th international conference on information systems security and privacy*, pages 25–36. SCITEPRESS-Science and Technology Publications, 2022.
15. Mahmoud Said Elsayed, Nhien-An Le-Khac, and Anca D Jurcut. Insdn: A novel sdn intrusion dataset. *IEEE access*, 8:165263–165284, 2020.
16. Abdulsalam O Alzahrani and Mohammed JF Alenazi. Ml-idsdn: Machine learning based intrusion detection system for software-defined network. *Concurrency and Computation: Practice and Experience*, 35(1):e7438, 2023.
17. M Sami Ataa, Eman E Sanad, and Reda A El-Khoribi. Intrusion detection in software defined network using deep learning approaches. *Scientific Reports*, 14(1):29159, 2024.
18. Neelam Gupta, Mashael S Maashi, Sarvesh Tanwar, Sumit Badotra, Mohammed Aljebreen, and Salil Bharany. A comparative study of software defined networking controllers using mininet. *Electronics*, 11(17):2715, 2022.
19. Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting traffic analysis with neural network interface cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 513–533, 2022.
20. Gerry Wan, Shinan Liu, Francesco Bronzino, Nick Feamster, and Zakir Durumeric. {CATO}:{End-to-End} optimization of {ML-Based} traffic analysis pipelines. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1523–1540, 2025.
21. Amirah Alshammari and Abdulaziz Aldribi. Apply machine learning techniques to detect malicious network traffic in cloud computing. *Journal of Big Data*, 8(1):90, 2021.
22. Mohanad Sarhan, Siamak Layeghy, Nour Moustafa, Marcus Gallagher, and Marius Portmann. Feature extraction for machine learning-based intrusion detection in iot networks. *Digital Communications and Networks*, 10(1):205–216, 2024.
23. Silvio Russo, Claudio Zanasi, and Isabella Marasco. Feature extraction for anomaly detection in industrial control systems. *Proceedings of the ITASEC*, 2024.
24. Arash Habibi Lashkari, Gerard Draper Gil, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. Characterization of tor traffic using time based features. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*, pages 253–262. INSTICC, SciTePress, 2017.
25. Siddharth Muralee, Igibek Koishybayev, Aleksandr Nahapetyan, Greg Tystahl, Brad Reaves, Antonio Bianchi, William Enck, Alexandros Kapravelos, and Aravind Machiry. {ARGUS}: A framework for staged static taint analysis of {GitHub} workflows and actions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6983–7000, 2023.
26. Luca Deri and Francesco Fusco. Using deep packet inspection in cybertraffic analysis. In *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 89–94. IEEE, 2021.
27. Amreesh Phokeer, Senka Hadzic, Eric Nitschke, Andre Van Zyl, David Johnson, Melissa Densmore, and Josiah Chavula. inethi community network: A first look at local and internet traffic usage. In *Proceedings of the 3rd ACM SIGCAS Conference on Computing and Sustainable Societies*, pages 342–344, 2020.
28. Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. DETER: Deterministic TCP replay for performance diagnosis. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 437–452, Boston, MA, February 2019. USENIX Association.