CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A High-Performance Golang-Based Network Intrusion Detection System

Master's thesis in Computer science and engineering

Oleksii Koshovyi and Shiqi Wu

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

A High-Performance Golang-Based Network Intrusion Detection System

Oleksii Koshovyi and Shiqi Wu

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

A High-Performance Golang-Based
Network Intrusion Detection System
Oleksii Koshovyi and Shiqi Wu

Supervisor: Victor Morel, Department of Computer Science and Engineering
Advisor: Georgios Pseiridis Pseiras, Ericsson
Examiner: Romaric Duvignau, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

A High-Performance Golang-Based
Network Intrusion Detection System
Oleksii Koshovyi and Shiqi Wu
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Distributed Denial of Service (DDoS) attacks remain a serious threat to transport networks, with recent attack volumes exceeding 30 Tbps, and the telecommunications industry being the main target. However, recent work has yet to study the impact of different architectures to host monitoring solutions, nor to assess the use of recent algorithms to improve attack detection. This thesis presents a Golang-based Network Intrusion Detection System (NIDS) for DDoS detection in transport network environments, developed in collaboration with Ericsson's Radio and Transport Engineering division.

The work builds on a baseline that utilised a statistical model and improves it in two directions. First, it improves detection effectiveness by utilising an Isolation Forest model that is trained on a wider set of flow features. These features are extracted by GoFlowMeter, an open source Go implementation of CICFlowMeter that we publish as part of this thesis. Second, it studies how the choice of software architecture affects the performance of the NIDS by comparing a monolithic deployment, a Kafka-based microservice deployment, and a gRPC-based microservice deployment. The system is evaluated on a Raspberry Pi 5 testbed using the CIC-DDoS2019 dataset, which is replayed as real network traffic through a separate sequential replayer. Extended Berkeley Packet Filter (eBPF) and Express Data Path (XDP) were also utilised to allow the NIDS to be able to process real traffic.

The results show that detection quality is governed mainly by the choice of detector rather than by the transport layer. The transport is not entirely neutral, however: the monolithic and gRPC variants reach almost the same accuracy, while the asynchronous Kafka pipeline trails them by roughly nine percentage points, an effect we attribute to its decoupled, online-updated delivery rather than to the detector itself. Compared with the statistical baseline, the Isolation Forest model achieves a higher recall and F1 score, which means that it can flag low-volume attack windows that the baseline misses. On the software side, the gRPC variant adds less than 2 milliseconds of transport time per window, while the Kafka variant adds about 27 milliseconds, which reflects the cost of the durability and decoupling that Kafka offers. The monolithic variant shows the smallest processing time degradation when the detector is switched to Isolation Forest, although this advantage may depend on the volume-heavy nature of the dataset. Together, these findings give practitioners a clearer view of the trade-off between detection quality and architectural overhead when deploying a NIDS on resource-constrained hardware.

Keywords: Network Intrusion Detection, Distributed Denial-of-Service, eBPF, XDP, Isolation Forest, microservice, gRPC, GoFlowMeter, Golang.

Acknowledgements

We are grateful to our academic supervisor, Victor Morel, for wise advice on conducting research and writing scientific papers [1]. He provided numerous comments on the article, helping to improve its quality and our understanding of academic research.
We are grateful to our industrial supervisor, Georgios Pseiridis Pseiras, who helped us with technical research, continuously supported us and gave us useful insights.
We would like to thank our examiner, Romaric Duvignau, who guided us and recommended key articles for an in-depth study of DDoS mitigation.
We want to express our gratitude to University of Gothenburg and Chalmers University of Technology for the meaningful amount of knowledge, new skills, and wisdom that we gained during our years of studying at the master's degree program.
We want to express our gratitude to Ericsson for the precious opportunity for us to gain industry insights and learn from the real-world scenarios.
Finally, we would like to express our sincere gratitude to each other. This thesis was a true team effort, and our collaboration enabled us to learn significantly more than we ever could have achieved individually.

Oleksii's acknowledgements

I'm personally grateful to my family, who supported me and believed in me at all stages of my educational process.

Shiqi's acknowledgements

I am deeply grateful to my family who not only shaped my attitude towards learning and life, but also gave me the courage to embrace my passions and bravely pursue the career I truly desire. I also want to thank everyone I have crossed paths with throughout my academic journey. Your continuous inspiration and support have constantly reinforced these lessons along the way.

Oleksii Koshovyi and Shiqi Wu, Gothenburg, June 2026

Contents

List of Figures

List of Tables

1
Introduction

Distributed Denial-of-Service (DDoS) attacks remain one of the most persistent threats to network infrastructure. Recent incidents have demonstrated attack volumes exceeding 30 Tbps, with telecommunications networks being the primary target [2]. As transport networks carry increasingly critical traffic, the need for effective and scalable intrusion detection at the network level has grown accordingly.
In the last quarter of 2025, the Aisuru-Kimwolf botnet launched a record-breaking attack that generated around 31 Tbps of attacking traffic. The amount of the traffic is almost equivalent to streaming over 2 million Netflix 4K movies, according to the press [3]. Cloudflare also reported that the main target of the Aisuru-Kimwolf botnet attack is the telecommunications industry (more than 70% among other areas) [2], which acts as the lifeline of the modern Internet. In the past two years, the amount of attacking traffic has increased from around 3 Tbps to over 30 Tbps, which is nearly a 10-fold increase [2]. Hence, DDoS attacks are not legacy problems. The attackers are constantly improving their approaches and building larger attacking infrastructure. Therefore, modern defence must scale faster to keep up.
Existing Network Intrusion Detection Systems (NIDS), the systems in charge of monitoring and detecting such attacks, face two key challenges in this domain. First, detection approaches that rely on limited feature sets, such as packet counts alone, struggle to distinguish sophisticated attack patterns from legitimate traffic surges. Second, the software architecture of the NIDS itself affects whether the system can meet the performance and maintainability demands of production transport environments, yet this architectural dimension is rarely studied alongside detection effectiveness.

1.1 Motivation

Although DDoS attack problem is not a new threat, malicious actors are constantly improving their attacking approaches. Consequently, it is important to analyse and improve the defence mechanisms. The primary motivation of this thesis is to contribute to this field by addressing the existing research gap.

1.1.1 Modern tools for DDoS mitigation

There are different types of modern DDoS attacks, based on traffic volume, targeted device, targeted OSI layer, etc. Therefore, it is more complicated to recognise the attack and filter out the attack itself. Consequently, modern NIDS use a variety of

methods in order to defend the host from the attack.
One of the modern filtering approaches is eBPF/XDP, which is a Linux kernel technology that allows the system to intercept incoming traffic and process each packet before the kernel allocates specific memory [4].
There are also different types of detection mechanisms: e.g., statistical models and machine learning models. Our research is based on previous research of Wickman and Rygaard (2025) [5]. They mostly used a statistical model in their research and Isolation Forest as a Machine Learning model. Their statistical model is used as a baseline in our research to show the improvements in detection performance.
Many research works are focused on detection mechanisms and their improvements, but it is also important to analyse the impact on the device with NIDS. Running NIDS takes the system resources, and the detection and filtering mechanism must not degrade the performance of the device. Otherwise, the defence mechanism will create more problems than it solves. Briefly put, it is important to consider all details that impact detection performance and the performance of the device that is running a NIDS.

Extended Berkeley Packet Filter

One of the modern and promising technologies is the Extended Berkeley Packet Filter (eBPF) with Express Data Path (XDP), which was introduced in the Linux kernel around 10 years ago. eBPF allows processing a packet before the kernel allocates memory. XDP is used inside an eBPF program to actually drop or pass a specific packet. eBPF with XDP can help to reduce memory usage during a DDoS attack by filtering malicious traffic at an early stage.

Isolation forest as a detection mechanism

Another modern component of DDoS-mitigation systems is Machine Learning (ML). The Isolation Forest (IF) [5], [6] is one of them. IF is lightweight and has a smaller training time, because it is an unsupervised ML algorithm. Fine-tuning and better feature selection can increase the detection performance of the model, keeping the processing time small.

1.1.2 Architecture and performance impact

Apart from the detection and filtering mechanisms, it is also important to consider the architecture of the system. Each architecture has its own positive and negative sides. Monolithic architecture has all components in one place and does not have communication overhead. Microservice architecture divides a Monolith into several parts, which can speed up the processing time in case it is running in parallel. Microservice architectures are often necessary in cloud-based systems or in systems that need to separate concerns across different deployable components. Among the communication frameworks used in such systems, Apache Kafka [7] is widely adopted in industry for data streaming [8], and Google Remote Procedure Call (gRPC) is widely used for its efficiency [9]. Therefore, both are selected as representatives of microservice communication in this study.

1.2 Research questions

We defined two research questions (RQs) to guide our research:

1. How can we improve DDoS detection effectiveness in transport networks by proposing an improved detection approach that combines eBPF kernel-level filtering with adaptive machine learning?
2. How do different software architecture patterns (monolithic, Kafka-based microservices, and gRPC-based microservices) impact the software performance of NIDS when integrated with different detection algorithms?

To answer these research questions, this thesis proposes an improved DDoS detection approach that combines eBPF kernel-level filtering with an adaptive Isolation Forest trained on a richer flow-level feature set, extracted by our open-source tool GoFlowMeter [10].

In addition, we conduct a comparative evaluation of three software architecture patterns: monolithic, Kafka-based microservices, and gRPC-based microservices, for NIDS deployment, assessing their impact on performance. The proposed system is empirically evaluated using the CIC-DDoS2019 dataset [11].

1.3 Collaboration

The DDoS attack problem is still relevant and dangerous, as the emerging attacks are continuously increasing the attacking traffic volume. One of the main targets is the telecommunications industry, wherein the amount of malicious traffic increases every year. Although modern telecommunication systems are already able to defend themselves against DDoS attacks, the continued growth in attack volume nevertheless motivates ongoing exploration of more advanced detection and mitigation technologies, so that the transport infrastructure can keep pace with the evolving threat landscape.

This thesis is presented in collaboration with Ericsson's Radio and Transport Engineering division. The products within the Transport department serve as the network's backbone, responsible for carrying and routing massive amounts of network traffic between critical nodes. Consequently, the division is investigating advanced methodologies to further enhance the reliability and safety of these systems.

One such product is MINI-LINK. It is a microwave telecommunication solution from Ericsson. The MINI-LINK product portfolio transfers the network traffic over the air using microwave transmission [12]. According to the IDS categorization types [13], the IDS that is deployed on such products could be classified as NIDS, a system that detects a DDoS attack on a network rather than on a single machine. Specifically, the interest for Ericsson is mainly in DDoS detection and prevention for their transport products. Therefore, this thesis focuses on exploring, analysing, and implementing new approaches for NIDS.

1.4 Contribution

This thesis provides a set of novel contributions. Some contributions directly relate to the previous research advice and thoughts on their future work [5], other contributions are independent and bring value to both the academic world and industry. In both cases each contribution type moved us forward and helped to improve the performance of the NIDS, analyse the impact on the device resources and implement real-world solutions working with real traffic.

- We implemented Golang version of feature extractor and fine-tuned the IF model. Authors of the previous research [5] were curious about balancing adaptability and sensitivity of the detection mechanism online, when the NIDS is working with traffic, depending on the amount of the traffic and its fluctuations. The main problem that was identified by our research was the insufficient number of features. They operated only by packet number as the only feature to identify the DDoS attack. Therefore, it was harder to tune the model and differentiate between the beginning of the DDoS attack and the increased number of real users. In our research, we decided to use more features describing traffic flows for detection, which significantly increased the performance. To extract more features, we implemented GoFlowMeter [10], based on the existing CICFlowMeter [14] provided by Canadian Institute for Cybersecurity (CIC). GoFlowMeter is a Golang implementation of the CICFlowMeter, which is implemented in Java, and is published online and available for everyone. Apart from feature extraction, the Isolation Forest model was fine-tuned, and better hyperparameters were chosen for future usage.
- We implemented a separate attacking program (Sequential replayer) to test our NIDS. Some contributions were not related to previous research. We analysed publicly available DDoS datasets and decided to proceed with the most modern: CIC-DDoS2019 [11]. In order to process and replay more than 100 GB of the traffic from the dataset, we implemented a sequential replayer, as described in section 4.1.1. Currently, it is a separate program that processes the dataset and sends the real traffic to the device with NIDS. It helps us to put the NIDS closer to the real attacking scenario with real and not simulated traffic.
- We implemented and tested the performance impact of the different architecture types. We implemented different types of architectures: Monolithic, Microservices with Kafka and gRPC. We analysed the performance impact and identified that the Microservice architecture with gRPC has much lower transport time compared to the microservice architecture with Kafka. This contribution helps us and will help other researchers to identify the best architecture for Network Intrusion Detection Systems in terms of memory usage and time delays.
- We implemented and integrated modern packet filtering with eBPF and XDP. The current NIDS works with real traffic that is sent from the replayer, and the NIDS is successfully processing and filtering the traffic, protecting the device resources. This is a solid contribution to the previous NIDS version [5], because it allows processing of the real traffic from the network interface. That

is not the first time eBPF/XDP is used as a DDoS prevention mechanism. Our work is another proof of the importance of this tool.
- We implemented a graphical user interface to visualise the acquired and filtered traffic, along with memory usage and time delays. It helps us to monitor the mitigation of DDoS attacks.

1.5 Thesis outline

The remainder of this document is organised as follows. We present an extensive background overview in Chapter 2; background represents a technical overview of the related technologies, such as IF, eBPF/XDP, gRPC, Kafka, etc. The related work research is presented in Chapter 3; in related work, we discussed related scientific papers we researched in detail, we analysed different types of NIDS, publicly available datasets and modern examples of eBPF/XDP usage. Our approaches toward implementation and evaluation are presented in Chapter 4, where we described how the latest version of NIDS is working with feature extraction, real traffic processing and modern architecture. The outcomes of the research are described in Chapter 5, and the corresponding discussions are described in Chapter 6. In the end, we provide a conclusion of our work and suggestions for future work in Chapter 7.

2
Background

Distributed Denial-of-Service (DDoS) attacks represent a growing threat to network services by targeting system availability through resource exhaustion. For example, recent botnet activities reached record-breaking traffic levels that specifically impact telecommunication infrastructure, and the variety of attack types has motivated a wide range of classification schemes and public datasets for analysis. To address these challenges, Network Intrusion Detection Systems (NIDS) offer a broader visibility into aggregated traffic patterns compared to host-based alternatives [13]. Beyond detection accuracy, a NIDS that is intended for real-world deployment must also be evaluated as a software system, which means considering aspects such as its architecture, resource usage, and processing performance. The background needed for this thesis therefore spans both the network security domain and the software engineering domain.
The remainder of this chapter is organised as follows. Section 2.1 introduces DDoS attacks in general, and Section 2.2 describes the existing taxonomies and the main attack types in each category. Section 2.3 reviews the public datasets available for DDoS research and motivates the choice of CIC-DDoS2019 for the present study. Section 2.4 introduces intrusion detection systems and explains why a network-based design is appropriate for the Ericsson use case. Section 2.5 describes the primary machine learning approach utilized in this thesis and compares it with alternative methods. Section 2.6 introduces information about fast filtering with modern Linux-kernel tools. Section 2.7 discusses microservice communication frameworks and introduces the two representatives used in the experiments, Apache Kafka and gRPC. Finally, Section 2.8 presents the software quality attributes that are later used to evaluate the proposed system.

2.1 Distributed Denial-of-Service

DDoS attacks are one of the main threats in the security area. A DDoS attack focuses on lowering the availability of the victim. In the context of the CIA triad, which balances confidentiality, integrity, and availability [15], a DDoS attack focuses on the third component. For example, in cloud-based services, users connect through central or distributed servers. An attacker can overload the system with multiple simultaneous requests to a node, so it would take all resources and cut out other users.
It is important to protect the system against this type of attack. When a DDoS attack happens, the system should filter out only malicious requests and keep real

users in the system.

2.2 Different Types of DDoS Attack

DDoS attacks could be classified in different ways, there are several DDoS taxonomies. Zargar et al. (2013) [16] provided a comprehensive taxonomy in their research based on the open systems interconnection (OSI)[1] layer at which the attack is conducted and also proposed the Botnet-based DDoS attacks category [16]. Yusof et al. (2019) [18] later summarized Zargar et al. [16]'s taxonomy and categorized the DDoS attacks purely according to the OSI layer, which includes network or transport-layer attacks and application-layer attacks [18]. The network or transport layer attacks aim to consume the victim's computing resource, which is used to respond to normal traffic, such as ICMP, TCP, and UDP [16], [18]. When a certain type of flood attack occurs at this layer, the victim will be unable to handle normal requests, and normal users will hence lose their connection to the victim. The application-layer attack, on the other hand, includes variations such as DNS query flooding with spoofed IP addresses, HTTP session/request flooding, asymmetric attacks, and the Slowloris attack [16], [18].
For OSI-layer-based attacks, each attack exploits a corresponding OSI layer in its own way:

- Network layer (3).
  - Smurf Attacks [19]. The attacker sends numerous Internet Control Message Protocol (ICMP) packets to a range of computers with a spoofed source. So, each device that gets an ICMP packet will respond to the specified source and flood the victim's machine.
  - ICMP Floods [20]. Unlike the Smurf attacks, the ICMP flood attack is executed directly against the victim. The attacker sends an excessive amount of ICMP packets to the victim's machine. In both cases it affects CPU and bandwidth resources.
  - IP/ICMP Fragmentation. The attacker exploits the fragmentation of big IP packets by sending malformed or incomplete IP fragments. The victim consumes significant CPU and memory resources attempting to reassemble the full packet set, which it will never receive.
- Transport layer (4).
  - SYN Floods [21]. The attacker exploits the TCP 3-way handshake. The attacker sends a SYN packet to the victim's machine. The victim's machine sends a SYN-ACK packet back and waits for the ACK packet as a response but never gets it back. Half-open connections exhaust the limit of the machine's connections and block the access to the real users.
  - UDP Floods [22]. The attacker sends a large amount of UDP packets to different ports of the victim's machine. The machine checks whether its programs listen to the specified port. In most cases they do not, and the

[1] "The open systems interconnection (OSI) model is a conceptual model created by the International Organization for Standardization which enables diverse communication systems to communicate using standard protocols. In plain English, the OSI provides a standard for different computer systems to be able to communicate with each other." [see 17]

machine sends an ICMP "Destination Unreachable" packet. The machine is flooded with UDP packets and cannot process users' packets.

  - TCP Connection Exhaustion
- Application layer (7). HTTP-encrypted attacks.
  - DNS Amplification [23]. The attacker sends DNS queries to open DNS resolvers using a spoofed source IP address matching the victim. By requesting large DNS records, the resolvers send massive response payloads directly to the victim, saturating their network bandwidth through reflection and amplification.

Sharafaldin et al. (2019) [11] analyzed the new attacks that appeared recently and proposed a taxonomy based on another criterion, which shifts the classification focus from the targeted OSI layers to the underlying execution mechanisms and classifies DDoS attacks into reflection-based and exploitation-based types [11]. In reflection-based attacks, attackers hide their identities by spoofing the target victim's IP address and routing requests through legitimate third-party servers [11]. These reflector servers subsequently overwhelm the victim with response packets. This approach can be executed utilizing application-layer protocols over TCP (e.g., MSSQL, SSDP), UDP (e.g., CharGen, NTP, TFTP), or a combination of both (e.g., DNS, LDAP) [11]. Conversely, exploitation-based attacks focus on directly exhausting the victim's network bandwidth or computing resources. This category encompasses TCP-based attacks, such as SYN floods that exploit the three-way handshake to cause server malfunctions, and UDP-based attacks, such as UDP floods that target random remote ports at a very high rate [11]. Additionally, it includes UDP-Lag attacks, which disrupt established client-server connections by hogging bandwidth [11].

Sharafaldin et al. [11] also proposed their DDoS dataset (CIC-DDoS2019) along with the taxonomy discussed above, and the CIC-DDoS2019 dataset includes the network traffic of each DDoS attack type across different time windows.

2.3 Datasets of DDoS Attack

It is important to have a realistic and varied dataset to train a precise machine learning model with a high predictive power. We found 3 publicly available datasets representing DDoS attacks: CAIDA2007 [24], CIC-IDS2017 [25] and CIC-DDoS2019 [11]. Ericsson also provided their own dataset. After detailed analysis, we decided to work with CIC-DDoS2019 since it is the most recent publicly available dataset focused exclusively on DDoS attacks, it represents different types of attacks, and it has labeled data. In addition, the researchers provided an extensive feature analysis extracted by CICFlowMeter and performed by RandomForestRegressor. Other public datasets analysed by Sharafaldin et al. [25], Goldschmidt and Chudá [26] (such as KDD99, CAIDAs, CSE-CIC-IDS 2018, etc.) were discarded from our analysis since they are now outdated or focus on other attacks.

Table 2.1: Comparison of DDoS Datasets

Property	**CAIDA2007**	**CIC-IDS2017**	**CIC-DDoS2019**
Duration	1 hour	5 days	2 days
Real traffic	✓	×	×
labeled data	×	✓	✓
Benign traffic	×	✓	✓
DDoS attacks	✓	✓	✓
DDoS-focused	✓	×	✓
Payload	×	✓	✓
Flow features	×	✓	✓
Reflection-based	×	×	✓
Exploitation-based	✓	✓	✓

2.3.1 CAIDA2007

CAIDA2007 [24] is one hour of anonymized traffic representing a real DDoS attack. The dataset mostly contains attacking traffic, and the authors excluded non-attack traffic. Also, all traces are anonymized, and packet payloads have been removed [24]. The dataset is outdated and does not represent all modern DDoS attack types. For that reason, we consider more recent datasets for our NIDS.

2.3.2 CIC-IDS2017

The CIC-IDS2017 dataset [25] consists of 5 days of generated traffic. Researchers simulated different types of attacks during each day. There are 7 main types: Brute-force, Heartbleed, Botnet, DoS, DDoS, Web Attack (SQL injection, Cross-Site Scripting) and Infiltration Attack. Benign traffic was generated using the B-profile [27] proposed by Gharib et al. Benign traffic simulates the behaviour of 25 people using HTTP, HTTPS, FTP, SSH, and email protocols [25]. The researchers also provided a feature analysis for the CIC-IDS2017 dataset. They used CICFlowMeter to extract an 80-dimensional feature set and RandomForestRegressor (RFR) to identify the most significant features for each type of traffic. Among older datasets, the advantages of CIC-IDS2017 are labeled entries, attack and protocol diversity, and feature analysis.

2.3.3 CIC-DDoS2019

The CIC-DDoS2019 dataset [11] is generated by the same authors as the CIC-IDS2017 dataset [25], as their next step in network security research. CIC-DDoS2019 consists of 2 days of generated traffic. This time, researchers focused on different types of DDoS attacks: reflection-based and exploitation-based. The benign traffic

of CIC-DDoS2019 is using the same B-profile as in CIC-IDS2017, simulating the behavior of 25 users. The features were also extracted with CICFlowMeter and analyzed using RFR. The authors provided RadViz analyses that represent the importance of each feature for different types of DDoS.
The CIC-DDoS2019 dataset is more suitable for our research. This dataset has extracted features and labeled data. Furthermore, CIC-DDoS2019 is the most recent publicly available dataset focused exclusively on DDoS attacks. However, its data are generated artificially. Real traffic, if captured, is often owned privately and is not disclosed due to security and privacy concerns.

2.4 Intrusion Detection Systems

Intrusion Detection Systems (IDS) are commonly categorized as Host-based IDS (HIDS) and Network-based IDS (NIDS) [13]. A HIDS is installed on individual hosts and monitors local system behaviour, whereas NIDS is placed at network vantage points and inspects traffic across monitored links [13]. As a result, the HIDS provides fine-grained visibility at the endpoint level, while NIDS provides a broader view of communication patterns across the network.
For the Ericsson use case to prevent DDoS attacks, NIDS is the more suitable primary approach for DDoS defence. First, network-level monitoring enables earlier detection of anomalous traffic upstream, before target hosts become saturated. Ericsson provides the transportability to transfer all the network traffic. Second, because DDoS attacks are distributed by nature, the relevant attack signal appears in aggregated traffic behaviour rather than on a single host. Third, network-level detection can be directly connected to mitigation at the ingress, which helps preserve bandwidth and protect critical transport resources.
A NIDS that operates on network traffic typically does not classify individual packets, but works on higher-level summaries derived from groups of related packets. The following subsections introduce the two building blocks that the proposed NIDS relies on: `network flows and sliding time windows`, which define the unit of traffic that the detector inspects in Section 2.4.1, and `CICFlowMeter`, which is the reference tool for extracting statistical features from those flows and is described in Section 2.4.2.

2.4.1 Network Flows and Sliding Time Windows

A network flow is a sequence of packets that share the same connection identity, typically the combination of source and destination IP addresses, source and destination ports, and the transport protocol. When the packets travelling in both directions of a connection are grouped together, the result is a bidirectional flow, which captures the full request and response behaviour of that connection. Flow-level statistics, such as the number of packets, the duration, and the timing between packets, summarise the behaviour of a connection more compactly than individual packets and are the standard input for flow-based intrusion detection [14], [28].
Because a continuous traffic stream has no natural boundaries, packets are also grouped into fixed-length time windows so that detection can be performed on a

bounded amount of traffic at a time. A sliding time window advances over the stream by a fixed step, so that consecutive windows cover successive periods of traffic and each completed window is classified independently. A single network flow may extend across several windows, in which case the portion of the flow that falls inside one window is only a part, or sub-flow, of the complete flow.

2.4.2 CICFlowMeter

In academic cybersecurity, CICFlowMeter is a popular open-source tool used to analyze network traffic and generate flow-based features. It was developed by the Canadian Institute for Cybersecurity (CIC) to process both PCAP files and live network traffic. The tool organizes packets into bidirectional flows and extracts 80 statistical features, such as the features related to flow duration, packet count, and time intervals. These features are commonly used to train machine learning models for detecting network intrusions or classifying traffic [14], [28]. The main advantage of CICFlowMeter is the high level of detail it provides compared to standard exporters like NetFlow. It also saves data directly into CSV files, which makes it easy to use for data analysis.

2.5 Machine Learning

Modern NIDS are using Machine Learning (ML) models as a detection mechanism. Sharafaldin et al. [11], Farasat et al. [29], Tebbaa et al. [30] conducted research on detection performance using different ML algorithms, including Long Short-Term Memory (LSTM), Support Vector Machines (SVMs), Decision Tree and other. These ML algorithms are supervised. Wickman et al. [5] used Isolation Forest (IF) in their NIDS.

2.5.1 Isolation Forest

The Isolation Forest [5], [6] is an unsupervised machine learning algorithm designed for anomaly detection. Introduced by Liu et al. [6] in 2008, this approach constructs an ensemble of isolation trees from the data. Each isolation tree is built by recursively partitioning the dataset using randomly selected features and randomly chosen split values within their respective ranges, until each data point is fully isolated in its own leaf node. The anomaly score of a point is derived from its average path length across all trees in the forest: the shorter this average path length, the higher the likelihood that the point is an anomaly. Since anomalous points are rare and require fewer partitions to be isolated, their average path length is shorter.

2.5.2 Supervised Learning versus Unsupervised Learning

Supervised learning algorithms rely on labeled data for training and generally achieve better detection performance than unsupervised learning algorithms. Among the mentioned approaches the best detection results were presented by LSTM and Bidirectional LSTM [11], [29], [30]. However, the IF algorithm is more lightweight,

consumes fewer computational resources [31] and has better detection performance in case of zero-day attacks [5], when a specific type of attack is not presented in the training data. Furthermore, IF can be further improved, using fine-tuning and better feature selection.

2.6 Fast filtering with eBPF/XDP

There are modern approaches to filter and process network traffic, and Extended Berkeley Packet Filter (eBPF) is one of them. eBPF is a Linux kernel tool that uses Express Data Path (XDP) hooks to operate packets at the kernel level. This technology is already presented in research related to the DDoS mitigation problem.

2.6.1 eBPF

eBPF is a virtual machine running on the kernel level of Linux OS. It allows loading a program from user level to kernel level and processing packets efficiently. eBPF programs can be executed in response to kernel-level events. This packet filter uses shared maps between all eBPF programs. Maps are key/value data structures and can be used for storing packet statistics [4]. Maps have general control commands: lookup, update, and delete.

2.6.2 XDP

XDP is another kernel-level Linux tool that allows the system to efficiently process network packets. XDP is triggered by hooks, it is receiving packet data directly from the network hardware. XDP processes the packet data before the kernel allocates memory or starts parsing the packet using zero-copy mode with AF_XDP sockets [4], [29]. XDP can drop, pass, or redirect packets with high performance, achieving processing rates that exceed 20 million packets per second [4], [29].

2.7 Microservice Communication Frameworks

A microservice architecture decomposes a system into independently deployable services that communicate over well-defined interfaces rather than through in-process function calls. The way these services exchange information has a direct effect on both the performance and the structure of the overall system, and two communication styles are particularly common. The first is *asynchronous messaging* through a message broker, in which a producer writes messages to an intermediate queue and one or more consumers read them at their own pace. The second is *synchronous remote procedure calls*, in which a client invokes a method on a remote server and waits for the response. Apache Kafka and gRPC are widely used representatives of these two styles, and both are used in this thesis.

2.7.1 Apache Kafka

Apache Kafka is a distributed publish-subscribe message broker that persists messages in append-only logs and exposes them to consumers organized in consumer groups [7]. Messages are written by producers to named channels called topics, where a topic is a set of queued messages identified by its topic name, and they are read by consumers that subscribe to those topics. Because Kafka follows a producer-consumer model with a broker placed between the two sides, the producer and the consumer are decoupled in time and do not need to operate at the same rate.
This model provides three properties that are relevant for a workload such as a NIDS. **First**, the broker buffers messages on disk, so a transient burst of incoming work does not have to be absorbed by the consumer in real time, since the queue is allowed to grow and the consumer catches up afterwards. **Second**, multiple consumer instances can join the same consumer group and share the partitions of a topic (a topic is split into partitions that consumers divide among themselves), which makes horizontal scaling possible without changing the producer. **Third**, because Kafka persists messages, a consumer restart does not lose the messages that arrived during the downtime. The cost of these properties is that every message has to pay for serialization, network transmission, and broker round-trip, so the end-to-end latency becomes a function of the broker configuration rather than of in-process function calls.

2.7.2 gRPC

gRPC is a remote procedure call framework that uses Protocol Buffers as its interface definition and serialization format [32]. Protocol Buffers is a compact data serialization format that reduces the load of network communication [33]. In contrast to a message queue such as Kafka, gRPC follows a synchronous request-response model in which the client issues a call and blocks until the server returns a response. The contract between the client and the server is described in a `.proto` file, from which the language-specific interface code is generated by the `protoc` compiler and shared by both sides.
The synchronous nature of gRPC means that the client waits for each request to be processed before it continues, so the per-request processing time directly includes the network round trip together with the cost of serializing and deserializing the Protocol Buffers payload. This eliminates the buffering offered by a broker, but it also avoids broker persistence and consumer-group coordination, and it produces a tighter feedback loop between the client and the server, since each result is available in the same call that produced it.

2.8 Software Evaluation

The deployability of a NIDS software in an embedded environment depends most directly on whether the system can keep up with incoming traffic within the available memory and time budget. Performance is therefore an important attribute when comparing software architectures for NIDS, and it is the attribute that this thesis

evaluates. Two aspects of performance are particularly relevant in this setting: latency and memory consumption. Latency refers to the time the system takes to process a unit of work, from the moment the input arrives to the moment a result is produced. Memory consumption refers to the amount of memory that the running system occupies.

3
Related Work

We analysed the general background and the existing research of Distributed Denial-of-Service (DDoS) attack mitigation to identify key research questions. We investigated existing works related to Network Intrusion Detection Systems (NIDS) for DDoS detection and prevention, advanced kernel-level packet filtering processes using eBPF/XDP, and different machine learning detection approaches.

3.1 NIDS for DDoS Defence

Research on NIDS for DDoS attacks encompasses various methodologies, ranging from statistical models to machine learning techniques. This section reviews the progression of these defence mechanisms. The initial approaches for this topic in Ericsson, including previous research conducted by Wickman and Rygaard (2025) [5], established statistical baselines and explored early unsupervised anomaly detection using Isolation Forest (IF). Other works also utilised supervised learning, such as Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) networks applied to capture artificially created spatial patterns and temporal patterns of the network traffic. Methods like Random Forests (RF) and the Autoencoder as unsupervised learning approaches are also utilised to improve classification accuracy. Finally, to address the limitations of relying on labelled datasets, the section examines recent studies that investigate unsupervised learning methods to detect novel attack vectors.

3.1.1 Wickman and Rygaard

Wickman and Rygaard (2025) [5], in a previous master thesis conducted at Ericsson, also contributed to this field by combining machine learning techniques with probabilistic data structures such as Count-Min Data Sketch and HeavyKeeper Algorithm. A data sketch is a compact probabilistic structure that summarises a high-volume traffic stream using a fixed and small amount of memory, at the cost of small and bounded estimation errors. In their system, the Count-Min Sketch provides approximate per-source packet frequency counts, while the HeavyKeeper algorithm identifies the heavy hitters, that is the small number of sources responsible for a disproportionate share of the traffic. Their work was evaluated on CAIDA2007 [24] and Ericsson's internal datasets. IF was chosen as an unsupervised learning approach because it does not require labelled data for the training process. And not all datasets had labelled data. The authors developed 2 IFs in their system. First, IF identified

anomalies based on the whole traffic flow (packet counts per time window). Second, IF identified anomalies for individual IPs. Their system has a warmup procedure that trains threshold IFs on benign traffic. During warmup, the threshold is constantly improved after each time window. After the warmup is finished, the system uses a threshold to filter out malicious traffic that exceeds the threshold. However, the authors used only one feature for IF training - packet number. The detection accuracy was lower than that of the statistical baseline approach: 73.63% vs. 96.09%. We are improving the IF approach by extracting and using more features to identify DDoS attacks.
The statistical method proposed in the previous master's thesis will be our baseline in this study. In later chapters, we will use "Baseline" to refer to their statistical method.

3.1.2 CNN-based NIDS

Convolutional Neural Networks (CNN) provide a method for data classification based on labelled training data. They perform well in image classification and processing two-dimensional data. Shaaban et al. (2019) restructured one-dimensional network feature vectors into two-dimensional matrices with padding to apply a CNN model. Their approach achieved 99 % accuracy tested on a simulated network dataset and the NSL-KDD dataset [34]. Similarly, Akgun et al. [35] developed an inception-like CNN model to detect DDoS attacks. They emphasised data preprocessing by removing redundant or duplicate records and applying feature selection to the CIC-DDoS2019 dataset. Their model achieved 99.99 % binary classification accuracy and 99.30 % multiclass accuracy.
However, CNN architectures are inherently designed for spatial data, while the network traffic is sequential. Converting one-dimensional network packet features into two-dimensional matrices introduces an artificial structure that may neglect important temporal patterns.

3.1.3 LSTM-based NIDS

To utilise the temporal patterns of network traffic, Shurman et al. [36] developed a model based on Long Short-Term Memory (LSTM) networks to detect Distributed Reflection Denial-of-Service attacks. They evaluated their approach using the CIC-DDoS2019 dataset and achieved a test accuracy of 99.19 % using a three-layer, 128-unit LSTM architecture. Djama et al. [37] also proposed a hybrid architecture that integrates CNN and LSTM networks. Their design uses convolutional layers to extract spatial features from network traffic and recurrent layers to capture temporal dependencies. When tested on the CICIDS2017 dataset, this hybrid model reached an accuracy of 99.96 %.

3.1.4 Random Forest-based NIDS

RF is an ensemble learning method that constructs multiple decision trees to improve accuracy and reduce overfitting. Bhawsar et al. (2025) [38] evaluated the

performance of RF and Support Vector Machines (SVM) for detecting DDoS attacks using the CIC-DDoS2019 dataset as well. Their results indicated that the RF model achieved an accuracy of 99.92%, which produced fewer false negatives than the SVM model. However, the execution time of their RF method is around 10 times larger than the execution time of the SVM method they used, indicating that the RF method could affect the performance in time-sensitive use scenarios.

3.1.5 Unsupervised Learning-based NIDS

CNN, LSTM, and RF are all supervised learning approaches. Training these models requires a labelled dataset, and the quality of the dataset affects the detection results. In the DDoS domain, a pre-trained model might be outdated due to the rapid growth of the approaches, characteristics, or amounts of DDoS attacks. Therefore, unsupervised learning could be potentially helpful.
Recent studies investigate unsupervised methods like Autoencoders and IF to address the reliance on labelled datasets. Vinothina et al. (2025) [39] proposed an anomaly detection scheme utilising an autoencoder combined with XGBoost for feature selection. The model learns a representation of standard behaviour by training the autoencoder exclusively on normal network traffic. It identifies malicious traffic by flagging data that exceeds a specific reconstruction error threshold. This method reduces data dimensionality and computational overhead during detection on the UNSW-NB15 dataset. Ghani et al. (2025) [40] applied the IF algorithm to detect network anomalies involving excessive login attempts and abnormal port activity. The algorithm isolates outliers by randomly partitioning data, which allows it to identify new attack patterns without prior labelling. Their system uses this unsupervised step as a primary warning mechanism to mitigate threats like backdoor exploits and DDoS attacks before passing the data to a supervised classifier. These approaches show that unsupervised learning can model normal traffic patterns and detect deviations caused by unknown attack vectors.

3.1.6 Fast Packet Processing with eBPF/XDP

Farasat et al. [29] proposed a framework for NIDS with an eBPF/XDP filtering procedure. The researchers used eBPF/XDP to process and pass/drop packets. They also compared different ML algorithms, including SVM, Decision Tree, LSTM and Bidirectional LSTM. The detection and prevention performance of the eBPF/XDP framework with the ML algorithm were examined using a generated dataset [41] and the Bidirectional LSTM (BiLSTM) algorithm showed the best result with 99.3 accuracy, 99.3 F-Score and 99.9 ROC. Also, the authors used CICFlowMeter-V4.0 to extract features from the traffic flow and provided and used SelectKBest to select the top 20 features.

3.2 Gap Analysis

A review of existing NIDS literature indicates a trend of reliance on supervised machine learning models for DDoS detection, while the application of unsupervised

learning methods could be investigated further. Both supervised models, such as CNN and RF, and unsupervised models, like Autoencoders, have demonstrated high detection rates when evaluated on standard datasets such as CIC-DDoS2019. However, for supervised models, securing accurately labelled network traffic for training is difficult and time-consuming in practical environments, including the specific operational context at Ericsson Radio and Transport Engineering. Network configurations change frequently, and new attack vectors emerge constantly, making a reliance on static labelled data impractical. Consequently, a gap exists for an unsupervised method that leverages multiple network features to identify anomalies without requiring labelled training data.
Beyond the challenge of data labelling, bridging this research and industry gap requires focusing on operational performance, a factor often neglected in existing studies. Current research primarily focuses on maximising detection accuracy, yet various models and methods require relatively significant computational resources. For instance, Bhawsar et al. (2025) [38] demonstrated that RF execution times are longer than those of simpler models, and neural networks introduce inherent latency during feature extraction and classification. In practical network environments, an NIDS must process packets at high speeds to mitigate DDoS attacks before they overwhelm the infrastructure. Therefore, optimising system performance parameters, such as memory usage and latency, is a critical requirement for transferring the effective unsupervised detection methods from academia to the industry.

4

Methods

This study proposes a Network Intrusion Detection System. Testing the proposed NIDS involves several core components. The first is a network replayer that replays captured traffic accurately while maintaining acceptable performance. The second is a network flowmeter that extracts a richer set of features from each network flow, in order to provide the detector with more information. The third is a set of three software architectures, namely the monolithic architecture and two communication frameworks that implement the microservice architecture, whose impact on system performance is also analysed. We then introduce the embedded testbed, which relies on a Raspberry Pi 5. Showcasing the feasibility of our approach on a device with restricted computational capabilities is crucial, since the primary usage of the NIDS in the industry is to install the system on hardware, and the performance of the NIDS using restricted computing resources is meaningful.

4.1 Practical Considerations

To design and implement a NIDS that meets the operational constraints of a network infrastructure, such as limited memory, bounded latency, and continuous processing under high traffic volume, practicality in terms of performance impact is an important aspect to consider. To replay the network traffic in the dataset in an efficient yet continuous way, a sequential replayer is proposed in Section 4.1.1. It aims to reduce the memory pressure of loading the large dataset, such as CIC-DDoS2019 [11].

4.1.1 Sequential Replayer

In order to test the detection performance and impact on the system, we used a modern DDoS dataset CIC-DDoS2019 [11]. This dataset represents a high volume of traffic and different types of attacks. Researchers split the traffic into 1000 PCAP (packet capture file format) files due to the total size of captured traffic (over 150GB). We implemented a sequential replayer with a lazy loader to process the CIC-DDoS2019 dataset efficiently in terms of memory usage and time consumption. The replayer is “lazy” since it takes 2 consecutive PCAP files from the list: the first file for direct processing and the second file for preloading. In this case, when the replayer finishes one pcap file, the next one is already preloaded into the memory, and the processing procedure is working without stops. Figure 4.1 represents the replaying procedure.

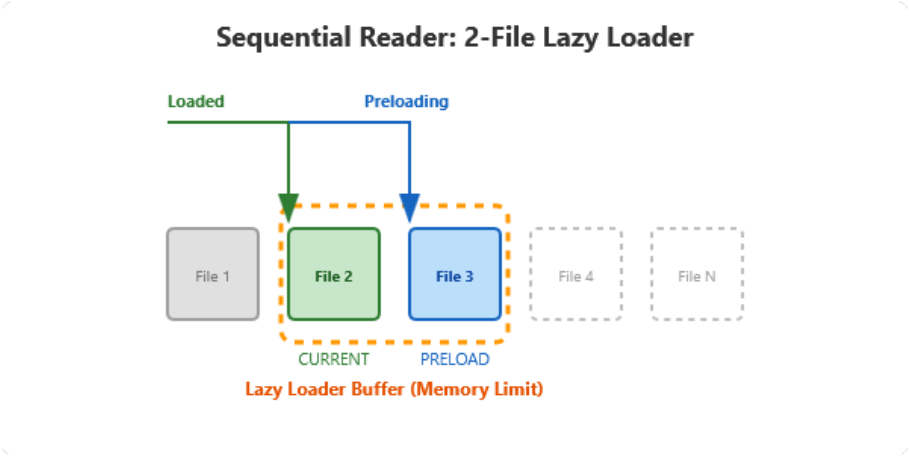


Figure 4.1: Sequential replayer

Each packet is replayed with a delay consideration [5]. Inside the replayer, we store intended and actual delays. Intended delay is based on the time difference between the first and the current packet. And the actual delay is based on the time difference between the replay time of the first packet and the current time. In case the intended delay is higher than the actual delay, the replayer waits until the intended and actual delays are equal. All processing procedures take time, so there is a maximum rate for the replayer. This limitation is mentioned in section 6.4.
The new NIDS is processing real traffic using eBPF/XDP Linux kernel tools. The NIDS attaches an eBPF program to a network interface to capture and process the traffic. Consequently, we extracted the sequential replayer into an independent program. Attacking device replays real traffic to a victim device with NIDS, specifying its network interface MAC address and the network interface of the attacker. Figure 4.2 shows data flow in the extracted version of the replayer.

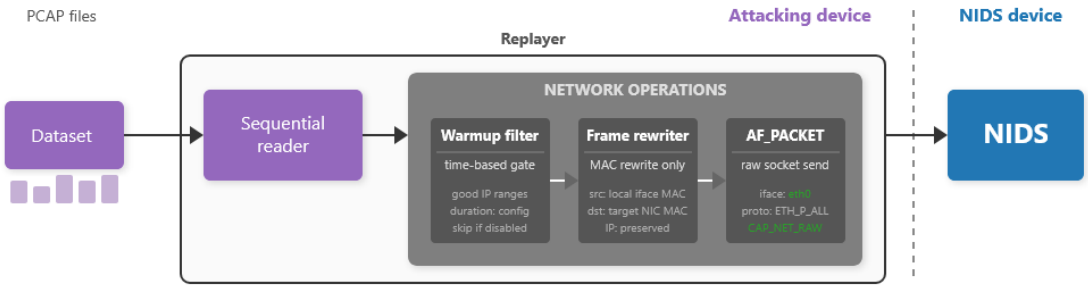


Figure 4.2: Extracted sequential replayer. Each packet read from the dataset passes through three network operations – a warmup filter, a frame rewriter, and an AF_PACKET raw socket – before reaching the NIDS device.

As shown in Figure 4.2, each packet passes through three network operations before it reaches the NIDS device. The warmup filter is a time-based gate that, during a configurable startup period, forwards only packets matching predefined good IP ranges, so that the NIDS can stabilise before attack traffic begins. The

frame rewriter then rewrites only the Ethernet source address (to the attacker interface) and the destination address (to the victim NIC), while the IP addresses and ports are kept unchanged. Finally, the rewritten frame is transmitted through an AF_PACKET raw socket (`ETH_P_ALL`, requiring `CAP_NET_RAW`) onto the configured interface.

Since the replayer is now separated, the rate limitation of the replayer and the replayer itself can only impact the performance of the attacker device and affect the attack quality. It is important to verify the traffic speed of the separated replayer.

4.1.2 Network Feature Analysis Using GoFlowMeter

Despite the fact that CICFlowMeter is a powerful tool in network flow analysis, as discussed in Section 2.4.2, the tool has some downsides. For example, its Java implementation makes it difficult to integrate into projects written in a modern system in Golang or other languages. Since there is no native CICFlowMeter feature-compatible library for Golang, developers have to use external execution calls or complex wrappers, which is inefficient for a high-performance application like the NIDS we proposed.

Building on the limitations of CICFlowMeter, GoFlowMeter was designed to provide a native Go implementation for extracting network features. GoFlowMeter addresses the shortcomings of CICFlowMeter by using Go's built-in library to handle packet analysis and flow assembly, which is more efficient in a modern Go project.

The tool processes Ethernet packets inside a time window to generate bidirectional flows and extracts the same statistical features found in the original CICFlowMeter. These features are fully compatible with the CICFlowMeter output, which ensures that datasets generated by GoFlowMeter can be used interchangeably with existing machine learning models, and the results could be analysed together with previous research results that utilised CICFlowMeter. GoFlowMeter is also written in Go, both because Go is widely used in cloud-based services and because the NIDS proposed in this thesis is implemented in the same language, which allows GoFlowMeter to integrate smoothly into the detection pipeline. We have made GoFlowMeter open-source to support researchers who need a high-performance flow meter that fits naturally into the Go ecosystem [10]. The new GoFlowMeter is published on GitHub and available online.

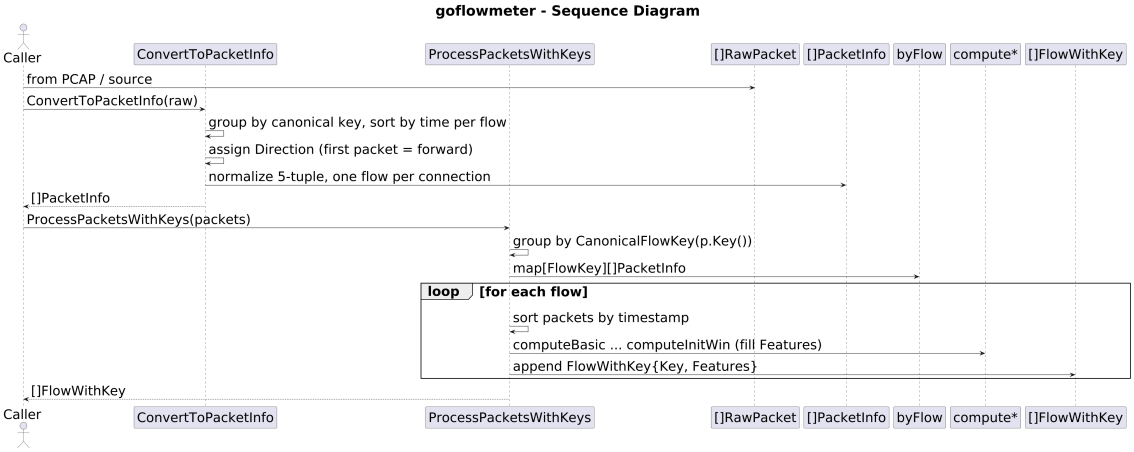


Figure 4.3: The sequence diagram of GoFlowMeter

Figure 4.3 illustrates the operational sequence of GoFlowMeter. The process begins with the user converting a PCAP file into an array of raw packets. Following this, the user calls the ConvertToPacketInfo API to organise these packets into specific flows and assign direction information to each. This step adds the necessary metadata for further processing to the raw packets. Finally, the ProcessPacketsWithKeys API is invoked to perform the feature analysis. This call generates a FlowWithKey array, which contains the unique flow identifier and the complete list of statistical features extracted for each flow.
GoFlowMeter produces one feature vector per bidirectional flow, whereas the detector consumes a single vector per time window, so the engine aggregates the per-flow vectors into one window-level vector. For each completed window, the packets buffered during that window are passed to GoFlowMeter, which assembles them into bidirectional flows and computes the 80 CICFlowMeter-compatible features (Section 2.4.2) per flow. Each flow f is mapped to an 80-dimensional vector x_f in a fixed feature order, and the window vector $\bar{x}$ is the element-wise arithmetic mean over the set F of flows observed in the window:

$$\bar{x}_k = \frac{1}{|F|} \sum_{f \in F} x_{f,k}, \qquad k = 1, \ldots, 80. \tag{4.1}$$

Any NaN or infinite component (for example, a rate computed over a zero-duration flow) is replaced by zero, and a window that contains no flows is represented by the 80-dimensional zero vector. The per-IP feature vectors used by the IP-level detector are built with the same element-wise mean, restricted to the flows whose source address is that IP.

4.1.3 Ground Truth Determination

The CIC-DDoS2019 dataset provides pre-classified network flow labels alongside flow features extracted by CICFlowMeter. However, these labels are coarse-grained, as each label covers the entire duration of a given flow. In this thesis, incoming network packets are buffered and grouped into sliding time windows. Consequently, a flow within a single window may represent only a sub-flow of the full flow described

by the dataset label. Since the objective is to identify time windows containing malicious traffic, a mechanism is required to derive window-level ground truth from the flow-level labels provided by the CIC-DDoS2019 dataset.

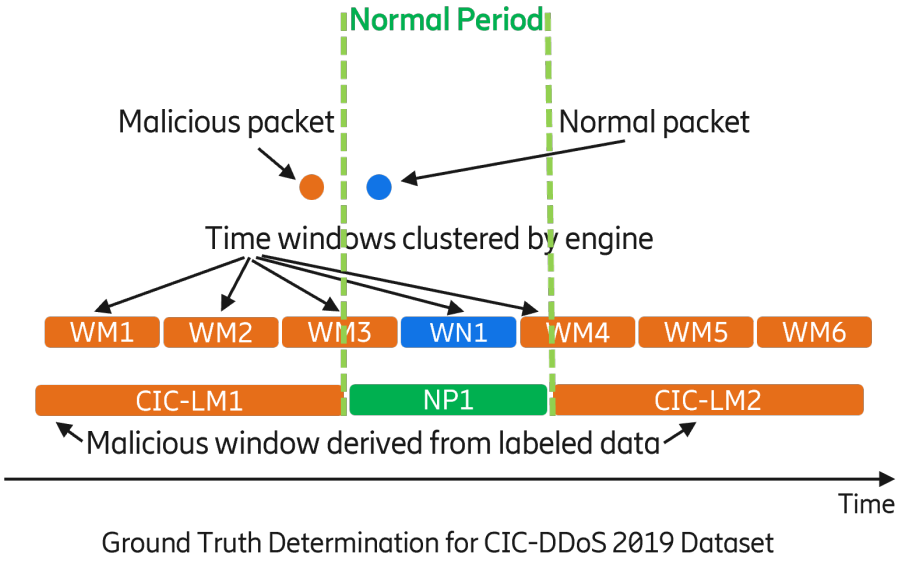


Figure 4.4: Ground-truth determination for packets and sliding windows. Orange and blue dots mark packets inside and outside a labelled malicious interval (CIC-LM), respectively. Windows WM1–WM6 overlap a malicious interval and are labelled malicious, while WN1 lies entirely within a normal period (NP1) and is labelled normal.

The ground truth (the known correct label for each packet or window) determination method used in this thesis is illustrated in Figure 4.4. In the figure, the malicious time intervals derived from the CIC-DDoS2019 labels are denoted CIC-LM, and the normal periods between them are denoted NP. Each packet is drawn as a dot: an orange dot lies inside a CIC-LM and is therefore classified as malicious, whereas a blue dot lies in a normal period and is therefore classified as normal. A sliding window labelled WM (WM1–WM6 in the figure) overlaps at least one CIC-LM and is labelled malicious, while a window labelled WN (WN1) lies entirely within a normal period and is labelled normal.
As a first step, per-flow labels are merged into malicious time intervals. There could be multiple flows in one time duration, and that duration is classified as malicious if any of the flows inside that duration is classified as malicious. Next, each incoming network packet is classified as malicious if it falls within a malicious interval, and as normal otherwise. A sliding window is then classified as malicious if it overlaps with any malicious interval; it is considered normal only if the entire window falls within a normal period.
It is worth clarifying that the window label and the packet label are derived independently. For example, the blue dot in Figure 4.4 is a normal packet because it lies inside the normal period NP1, even though it also falls inside WM3; the window WM3 is marked malicious only because it overlaps CIC-LM1, which has no bearing on the label of the packet. In short, both labels are obtained solely by checking

overlap with the CIC-LM intervals.
The rationale behind this determination method is that the resulting ground truth should reflect how the NIDS is expected to behave in the production environment. In a production deployment, the detector has to raise an alert as soon as any malicious activity is present within a time window, regardless of whether the window contains only a sub-flow of a longer attack or covers the full duration of one. Labelling a window as malicious whenever it overlaps with a known malicious interval is consistent with this operational requirement, and it rewards the detector for identifying attacks that begin or end inside a window boundary. Similarly, a window is labelled as normal only when no part of it overlaps with a malicious interval, which prevents partially malicious windows from being counted as clean traffic. Together, these rules make the ground truth conservative in the security-relevant direction, namely that no window in which an attack is in progress is ever marked as normal.

4.2 Fine-Tuning of Isolation Forest

To evaluate the accuracy based on different Isolation Forest hyperparameters, we developed a probe program that parsed every PCAP file in the selected dataset once, grouped packets into one-second windows under the same warmup stage that the NIDS engine applies, and caches both the per-window packet count and the corresponding 80-dimensional flow feature vector. This vector is the per-window aggregate of the flow statistics that GoFlowMeter extracts, formed as the element-wise mean of the per-flow feature vectors described in Section 4.1.2 (Equation 4.1), and it matches the 80-feature CICFlowMeter set introduced in Section 2.4.2 (covering flow duration, packet counts, byte counts, and inter-arrival timing).
This probe is an offline tool that is kept separate from the live NIDS engine, and it is used only to search for suitable hyperparameters before deployment rather than as a component of the deployed system.
The three hyperparameters varied by the probe are `num_trees`, `sample_size`, and `anomaly_fraction`.

1. `num_trees` is the number of trees in the forest. A larger number generally produces more stable anomaly scores at the cost of additional memory and computational resources.
2. `sample_size` is the number of samples drawn to train each tree. It controls the size of the subsample passed to the isolation procedure and affects both the training cost and the ability of each tree to isolate small anomaly regions.
3. `anomaly_fraction` is the expected proportion of anomalies in the data, which determines the threshold above which a score is interpreted as an anomaly.

The probe then sweeps a hardcoded grid of hyperparameters in which `num_trees` takes the values 50, 100, and 200, `sample_size` takes the values 128, 256, and 512, and `anomaly_fraction` takes the values 0.01, 0.05, 0.1, and 0.2. The probe supports two input modes for the detector, one in which only the packet count of each window is used and one in which the full 80-dimensional vector is used. The hyperparameter combination that ranks first among these results is the input that the live monolithic, Kafka-based, and gRPC-based deployments produce.

4.3 Software Architecture

Software architectures, such as monolithic and microservice, are adopted in various contexts depending on performance requirements and the need to accommodate fluctuating workload demands. In the context of DDoS detection and mitigation, monolithic architecture offers the simplest approach to system development and deployment. However, it lacks elasticity under varying workloads. Microservice architectures address this limitation. Apache Kafka, a distributed message queue with persistent storage, decouples services and buffers traffic spikes, potentially enabling elastic scaling. gRPC provides low-latency inter-service communication and straightforward service replication. Together, these technologies yield a more flexible system than monolithic alternatives. This thesis also investigates the impact of software architecture on system performance, and this section describes the architectural configurations employed in the study.

4.3.1 Monolithic Architecture

Monolithic software architecture usually refers to software that integrates the entire application into a single process [42]. In this thesis, the monolithic NIDS is implemented as one Go application where packet ingestion, aggregation, detection, mitigation, logging, and visualisation are executed in the same operating system process. This design is used as a baseline architecture because it minimises orchestration overhead and allows direct in-memory communication between components. The architecture diagram, which also indicates the pipeline, is shown in Figure 4.5. The monolithic implementation starts from a single entry point that loads a JSON configuration and initialises the IDS engine. The engine coordinates traffic replay, window management, detection, online updates, and output generation. In this context, an online update means that the model is adjusted incrementally as each new window of non-malicious traffic arrives, rather than being retrained offline on a fixed dataset. Traffic is replayed from CSV or PCAP traces with preserved timing and duration so that packet arrival behaviour remains close to the original capture. The system supports separate normal and attack traces for the CAIDA2007 dataset, and has a special sequential replayer (discussed in Section 4.1.1) and IP-filtering mechanism tuned for the CIC-DDoS2019 dataset. Flow-level features are extracted using our open-sourced GoFlowMeter, which is described in Section 4.1.2. In the monolithic pipeline, these features are consumed directly inside the same process, without external service calls or network serialisation. During execution, the system also exposes lightweight HTTP endpoints for health checking and GUI monitoring, while keeping recent detection results in local memory for fast access.

For each incoming packet, the engine first updates unfiltered window statistics and per-source counters. If mitigation is enabled, the packet is checked against the current per-IP limit and may be filtered before further processing by a module called Rate Limiter. Packets that are not filtered are inserted into active sliding windows. When feature-based detection is enabled, packets are buffered and transformed into flow-level representations before inference.

Detection is performed at two levels in each completed window. At the *time-window*

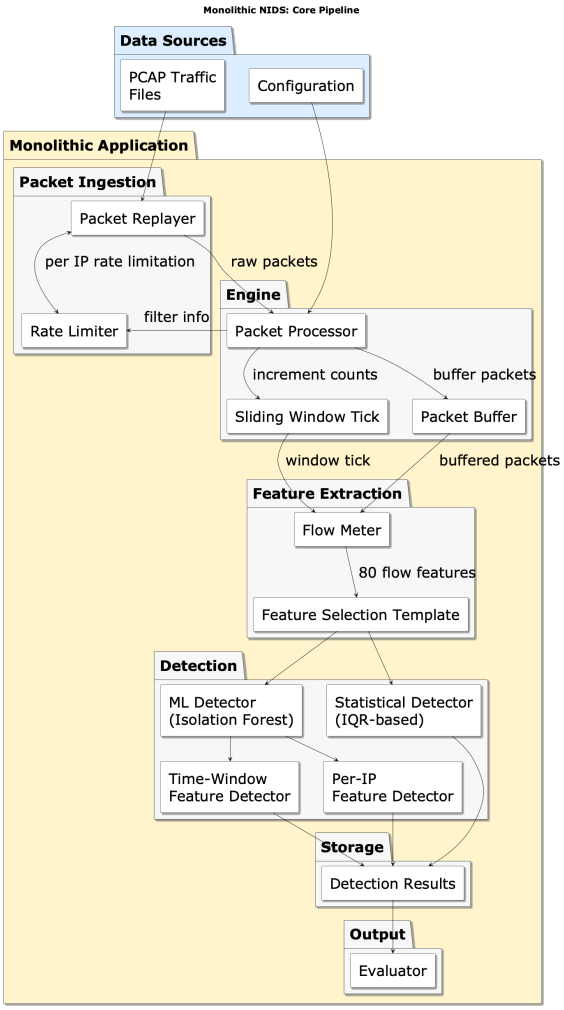


Figure 4.5: Monolithic NIDS architecture and pipeline used in this thesis

level, the system decides whether the current aggregate traffic behaviour indicates an attack. At the *per-IP level*, the system identifies suspicious sources within the same window. The detector can run in baseline mode using an IQR-based threshold strategy, or in machine-learning mode using Isolation Forest. In machine-learning mode, both count-based inputs and feature-based inputs are supported, and the active path is selected through configuration.
After each decision, thresholds are propagated to the rate limiter, and the corresponding window event is stored. The model is then updated online using non-malicious observations so that it can adapt to evolving normal traffic patterns. During detected attack periods, the IP threshold can be frozen to reduce instability caused by attack-dominated traffic.

4.3.2 Microservice Architectures

This study uses two microservice architectures that differ only in the communication framework placed between the NIDS engine and the detector. One uses Apache Kafka and the other uses gRPC. The general characteristics of these two frameworks

are described in Section 2.7; this section describes how each of them is integrated into the detection pipeline.

4.3.2.1 Kafka-based Architecture

In the Kafka-based architecture, Kafka decouples feature extraction from detection by placing a broker between the IDS engine and the detector. The engine keeps the same pipeline introduced in the monolithic configuration, namely the packet replayer, flow assembly based on a sliding time window, and feature extraction through GoFlowMeter. The difference is the way the completed time windows are delivered to the detector. Instead of invoking the detector through an in-memory function call, the engine serialises each completed time window into a feature message and produces it to the Kafka topic `flow-features`. A separate detector process subscribes to that topic, runs the configured detection mode on each consumed message, and produces the outcome to a second topic called `detection-results`. The engine consumes that topic asynchronously and feeds the results into the same storage, evaluator, and visualisation components that are used in the monolithic version, so that only the transport between the engine and the detector differs. Figure 4.6 illustrates this flow.
The buffering, horizontal scaling, and persistence properties described in Section 2.7 apply directly to this configuration. The broker absorbs bursts of incoming traffic, additional detector instances may join the same consumer group, and a detector restart does not lose the windows that arrived during the downtime. The deployment used in this study runs a single broker reachable at `localhost:9092` together with the engine and the detector on the same host, which keeps the focus on the architectural overhead rather than on networking conditions, while preserving the producer-consumer semantics that distinguish this configuration from the other two.

4.3.2.2 gRPC-based Architecture

In the gRPC-based architecture, the in-memory call from the engine to the detector is replaced by a unary RPC (a single request that returns a single response, as opposed to streaming). The engine acts as the client, and the detector runs as a stand-alone server that listens on a configurable address, with `localhost:50051` used as the default in this study. The contract between the two processes is defined in a single `.proto` file, and the Golang interface definition is generated by `protoc` and shared by both implementations. The service exposes one method, `Detect`, which takes a `DetectRequest` structure that contains the window identifier, the aggregated 80-dimensional window feature vector, the per-IP feature vectors, the per-IP packet counts, and the detection mode flags, and returns a `DetectResponse` that carries the attack decision, the anomaly score, the threshold, the list of malicious source addresses, and the time spent inside the detector. Apart from this transport boundary, the detector reuses the same Isolation Forest and baseline detection code that the monolithic and Kafka variants use, which keeps the detection logic comparable across all three architectures. Figure 4.7 shows the resulting interaction.
Because the call is synchronous, the engine waits for each window to be classified before it moves on, which produces a tighter feedback loop between the detector and

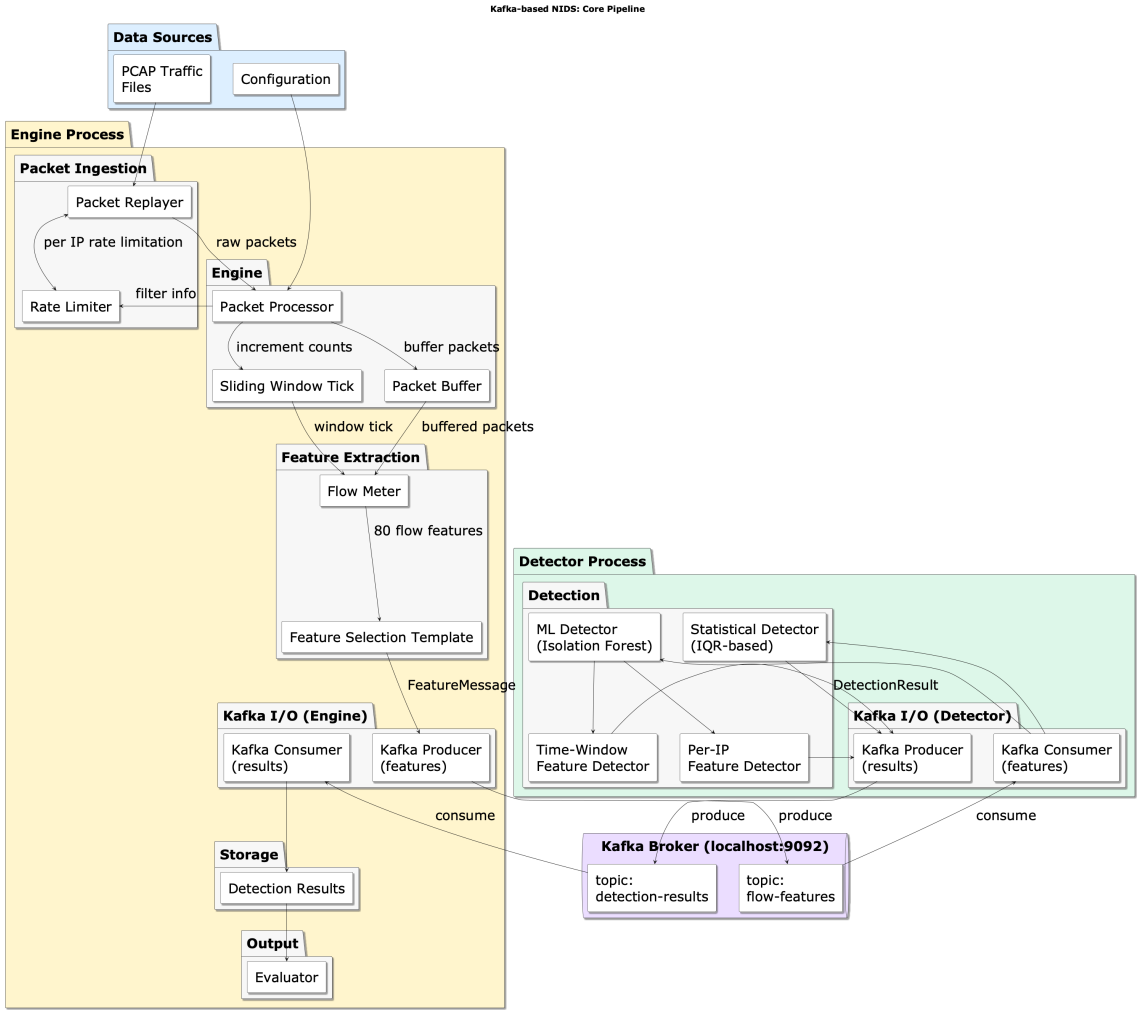


Figure 4.6: Microservice NIDS architecture and pipeline based on Kafka

the rate limiter, since each detection result is available to the engine in the same call that produced it. As with the Kafka deployment, the engine and the detector are placed on the same host in this study and communicate over the loopback interface (the host's internal localhost network path without a physical link), so that the comparison between the three architectures isolates the cost of the communication framework itself rather than the cost of crossing physical network boundaries.

4.4 eBPF/XDP

The previous NIDS provided by Wickman and Rygaard [5] had a good performance of the detection model, but the system itself was not autonomous, and the traffic replayer and the processing procedure were part of the NIDS. There was no real traffic running through a network interface. The new NIDS we are working with is fully autonomous and has the ability to interact with real traffic instead of simulated traffic. We decided to keep the rate limiter in the system and use its threshold as an indicator of the DDoS attack [29].

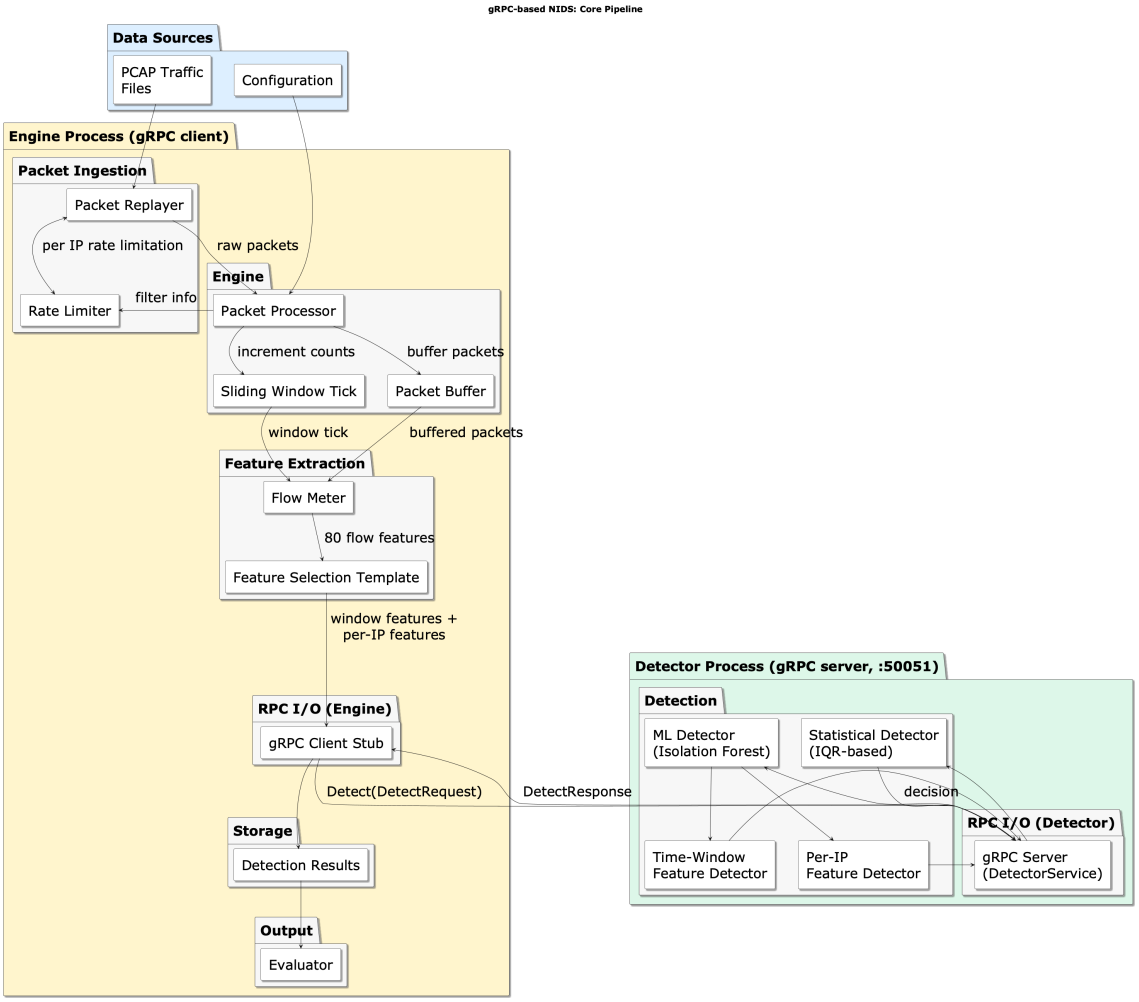


Figure 4.7: Microservice NIDS architecture and pipeline based on gRPC

4.4.1 New system improvements

As it was previously mentioned in section 6.4, the new replayer is entirely extracted and separated from the NIDS into a separate program. The replayer now represents an attacker, and the DDoS dataset is replayed through a real network interface to another device with a running NIDS. The old system used a rate limiter that filtered all exceeding traffic per IP address. The threshold of normal traffic was determined by an Isolation Forest model. The new version uses XDP hooks to capture traffic for analysis and filter all malicious packets from IP addresses identified as malicious. In the current version, we decided to keep the rate limiter as an indicator of malicious IP addresses. Any source IP with traffic exceeding the threshold will be added to a blocklist and filtered out in the next time windows (figure 4.8).

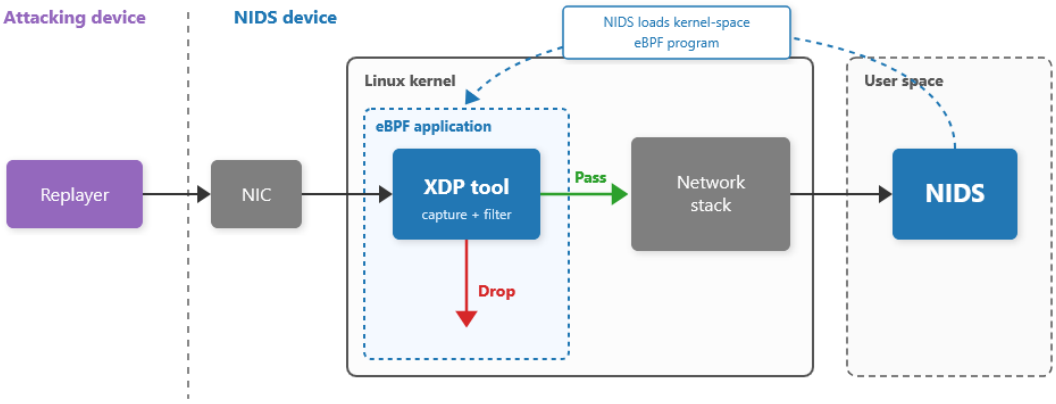


Figure 4.8: NIDS with eBPF integration

4.4.2 eBPF integration

To make the system interact directly with real traffic, we wrote a new eBPF program in C. This program is compiled and then automatically loaded into the Linux kernel by the NIDS engine. We attached the XDP hook directly to the network interface of the machine running the NIDS. This allows the system to inspect packets at the lowest level, right when they arrive at the network card and before the operating system processes them.
To share data between the kernel space and our Go application in the user space, we use BPF maps. When a new packet arrives, the XDP hook checks its source IP address against a BPF blocklist map. If the IP address is found in this blocklist, the eBPF program executes an XDP_DROP command, which instantly discards the packet and saves system resources. If the IP is not in the blocklist, it executes an XDP_PASS command. This allows the packet to pass through the network stack. The packet information is sent to the user space for further analysis.
On the Go side, we added two new components that replace the old replayer as the source of traffic: the processor and enforcer. The processor loads the compiled program, attaches it to the network interface, and reads the captured packets from the kernel. The enforcer writes malicious IP addresses to the blocklist map so the XDP hook can drop their traffic.

4.4.3 New processing and filtering logic

At startup, the NIDS loads the compiled BPF object, attaches the XDP program to the specified network interface, and starts a go routine that continuously reads packet events from the perf ring buffer[1]. Each event carries source and destination IP addresses, ports, protocol, TCP flags, direction, and IP total length. These fields are decoded and forwarded to the detection mechanism as a PacketData structure.

[1] "The ring buffer is a fundamental mechanism for data transfer. perf uses ring buffers to transfer event data from kernel to user space; another kind of ring buffer, which is the so-called auxiliary (AUX) ring buffer, also plays an important role for hardware tracing with Intel PT, Arm CoreSight, etc." [43]

The filtering logic is also updated to work together with the new kernel tools. At the end of each time window, the Isolation Forest model analyses the traffic and calculates a rate limit threshold. If the processor sees that a specific IP address has sent traffic exceeding this threshold, the engine confirms that it is an attacker. The Go application then updates the BPF blocklist map by adding this malicious IP address. Because the map is shared with the kernel, the XDP hook will automatically start dropping all future traffic from this attacker in the next time windows.

4.5 Experimental Setup and Evaluation

We used a Raspberry Pi 5 single-board computer as the testbed for the proposed NIDS. There are two reasons for us to choose the Raspberry Pi 5. First, the Raspberry Pi provides an embedded environment with limited computing resources compared to a standard workstation. This setting exposes the cost of each architectural choice more clearly than a less constrained machine would, and it approximates the conditions under which a NIDS could possibly be deployed close to the edge of a network.

Second, using a single fixed hardware platform across all experiments removes a source of variability. Because the platform is dedicated to running only the NIDS, the comparison between the monolithic, Kafka-based, and gRPC-based architectures more directly reflects the differences between the variations themselves.

The Raspberry Pi 5 has a Broadcom BCM2712 system on a chip with a 2.4 GHz quad-core 64-bit Arm Cortex-A76 CPU. The CPU includes the Arm Cryptographic Extension, 512 KB of L2 cache per core, and a 2 MB shared L3 cache [44]. The board runs a 64-bit Linux distribution. The Raspberry Pi 5 in this experiment has 16GB RAM.

All components used in a given experiment run on the same Raspberry Pi. In the monolithic configuration, the IDS engine runs as a single process. In the Kafka configuration, the engine, a single Kafka broker, and the detector service run as separate processes on the same board. The broker is reachable through the loopback interface. In the gRPC configuration the engine and the detector server run as two separate processes that communicate over loopback as well. The network conditions are identical across the three architectures and remove the influence of external links, while keeping the inter-process communication overhead that is characteristic of each design, which this thesis aims to compare.

Evaluating not only the accuracy-related metrics but also the software performance metrics could provide more insights into algorithm and software architecture selection for both academia and industry.

4.5.1 Evaluation Metrics

We evaluate the system with two groups of metrics: detection quality metrics and software performance metrics. The detection quality metrics are accuracy, precision, recall, and F1 score, and they are defined as follows:

$$\begin{aligned} \text{Accuracy} &= \frac{TP + TN}{TP + TN + FP + FN} \\ \text{Precision} &= \frac{TP}{TP + FP} \\ \text{Recall} &= \frac{TP}{TP + FN} \\ \text{F1 Score} &= 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \end{aligned} \tag{4.2}$$

Here, TP, TN, FP, and FN denote the number of true positives, true negatives, false positives, and false negatives, respectively. The ground truth is defined based on the method demonstrated in Section 4.1.3.
The software performance metrics consist of three time measurements and the memory consumption of the engine process. The three time measurements are processing time, detector time, and transport time, and their precise definitions depend on the software architecture. They are introduced together with the corresponding equations in Section 5.3.

4.5.2 Overview of Experiments

Three experiments are reported in the next chapter. The first tunes the Isolation Forest hyperparameters with the offline probe of Section 4.2 and fixes the configuration that the live system uses. The second compares detection quality, using the metrics defined above, between the statistical baseline and Isolation Forest across the monolithic, Kafka-based, and gRPC-based architectures. The third compares software performance, namely processing, detector, and transport time, together with memory consumption, across the same six architecture–detector combinations.

5
Results

The results in this study focus on three aspects: the fine-tuned hyperparameters for the Isolation Forest model to reach a better accuracy in the NIDS setup; the accuracy comparison between the baseline model and the proposed model; also the software performance results that report the processing time and memory consumption analysis.

5.1 Hyperparameter Tuning Results

To identify the hyperparameter combinations of Isolation Forest that are most suitable for DDoS detection on the CIC-DDoS2019 dataset, we executed the probe program introduced in Section 4.2. The probe swept the full grid of 72 configurations (36 per input mode: 3 values of `num_trees` × 3 values of `sample_size` × 4 values of `anomaly_fraction`, over the feature-vector and packet-count modes) and ranked them by F1 score and accuracy.

Table 5.1 lists the 19 top-ranked combinations out of the 72 evaluated, for both input modes of the Isolation Forest detector, the full 80-dimensional feature vector and the per-window packet count. All nine combinations use an `anomaly_fraction` of 0.2 and reach a mean F1 of 0.932 and a mean accuracy of 0.873 over three runs, regardless of the values of `num_trees` and `sample_size`. Since the nine combinations are tied at the displayed precision (identical mean F1 and accuracy over the three runs), the configuration carried forward to the live monolithic, Kafka-based, and gRPC-based deployments was selected arbitrarily from this tied set, namely the one with 200 trees and a sample size of 128. The same hyperparameter triple is also optimal for the packet-count mode, so the two modes differ only in the input representation and not in the chosen hyperparameters.

At their respective best settings, the two input modes perform almost identically. The best feature-vector configuration reaches an F1 of 0.932 and an accuracy of 0.873, while the best packet-count configuration at rank 10 reaches an F1 of 0.929 and an accuracy of 0.868, a difference of only about 0.3 and 0.5 percentage points. The feature-vector mode is also more stable: its nine combinations with `anomaly_fraction` set to 0.2 occupy the first nine positions with identical F1 and accuracy, which indicates that `num_trees` and `sample_size` have no measurable effect once the contamination parameter is set appropriately. The packet-count mode shows a slightly wider spread, with F1 from 0.909 to 0.929 and accuracy from 0.833 to 0.868 across the same two parameters, so these hyperparameters influence the count-based detector marginally more than the feature-based one.

Table 5.1: Hyperparameter combinations for the Isolation Forest across different input modes

Rank	Input Mode	num trees	sample size	anomaly fraction	F1	Accuracy
1	Feature Vector	200	128	0.2	0.932	0.873
2	Feature Vector	100	256	0.2	0.932	0.873
3	Feature Vector	200	256	0.2	0.932	0.873
4	Feature Vector	100	128	0.2	0.932	0.873
5	Feature Vector	100	512	0.2	0.932	0.873
6	Feature Vector	200	512	0.2	0.932	0.873
7	Feature Vector	50	256	0.2	0.932	0.873
8	Feature Vector	50	512	0.2	0.932	0.873
9	Feature Vector	50	128	0.2	0.932	0.873
10	Packet Count	200	128	0.2	0.929	0.868
11	Packet Count	200	512	0.2	0.927	0.864
12	Packet Count	200	256	0.2	0.927	0.864
13	Packet Count	100	512	0.2	0.924	0.859
14	Packet Count	100	128	0.2	0.923	0.858
15	Packet Count	50	256	0.2	0.922	0.856
16	Packet Count	50	128	0.2	0.918	0.848
17	Packet Count	100	256	0.2	0.918	0.848
18	Packet Count	50	512	0.2	0.909	0.833
19	Packet Count	200	512	0.1	0.830	0.710

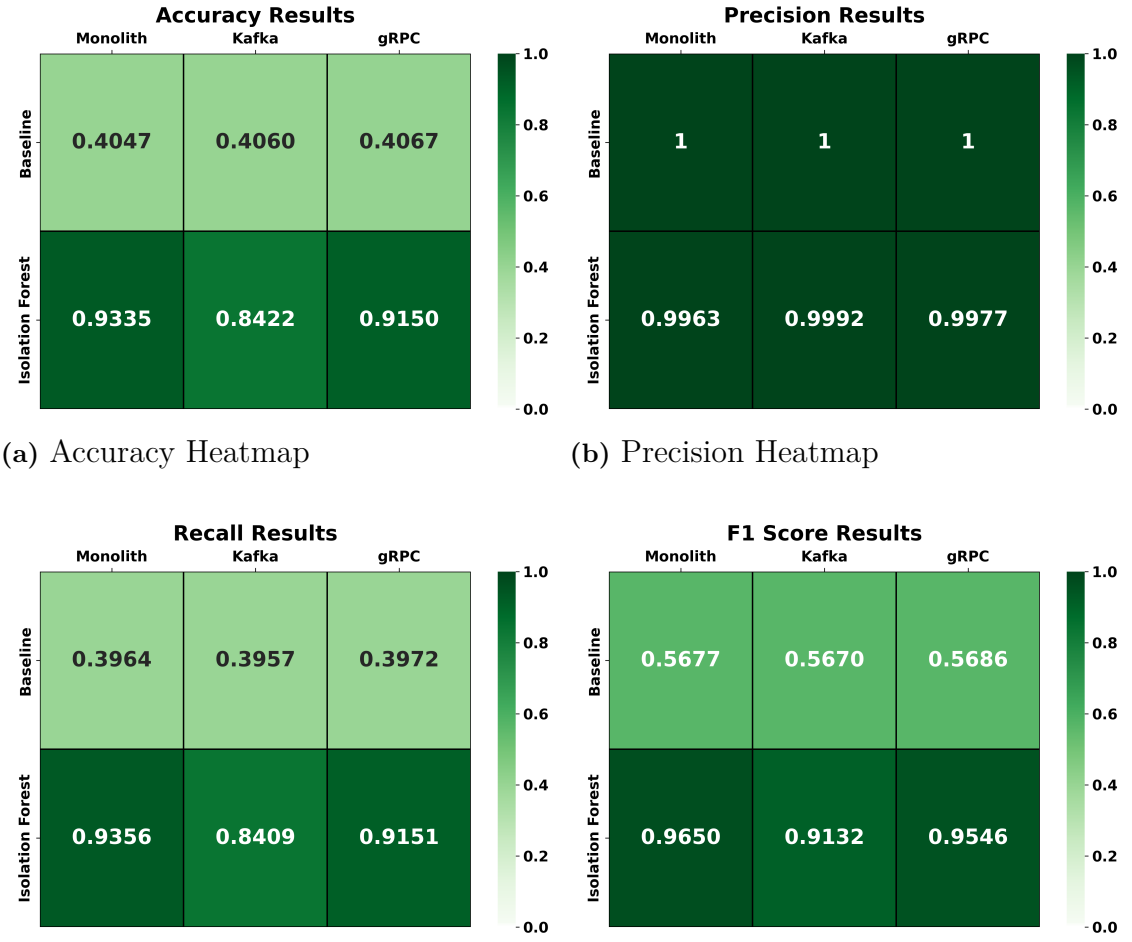
In both modes, `anomaly_fraction` is the dominant hyperparameter. Every combination with `anomaly_fraction` = 0.2 sits far above the single combination with `anomaly_fraction` = 0.1 at rank 19, whose F1 and accuracy fall to 0.830 and 0.710. This is consistent with the attack-heavy distribution of CIC-DDoS2019, where a higher contamination value matches the true proportion of malicious windows. The near-equivalence of the two input modes under this evaluation is discussed further in Section 6.1.

The configuration selected here, the feature-vector input with 200 trees, a sample size of 128, and an `anomaly_fraction` of 0.2, is held fixed for every subsequent experiment, and the detection accuracy reported in the next section is produced with exactly this configuration running live rather than through the offline probe.

5.2 Detection Accuracy Results

Whereas the previous section used an offline probe to select a detector configuration, this section reports how that selected configuration performs when the NIDS is run live. The live evaluation is carried out on a fixed set of 1,533 windows drawn from CIC-DDoS2019 and is repeated across every architecture and detector variant. These windows come from the same dataset that the probe used to rank configurations, and we did not hold out a separate test set; the hyperparameters were therefore both tuned and evaluated on CIC-DDoS2019. The live numbers should accordingly be read as the performance of the deployed system on this dataset, not as an independent test of generalisation to unseen traffic. The difference between the two is that the probe scores a configuration offline over the full dataset, whereas the live run measures the same configuration inside the running NIDS, including its warm-up and online-update behaviour, which is why the live and probe figures do not coincide.

Figure 5.1 shows the results of the detection accuracy heatmaps. The results are demonstrated using heatmaps because there are many software variations in this study, and heatmaps could help developers to visualise the best-performing model. The baseline row showed the performance of the baseline model.



(a) Accuracy Heatmap

(b) Precision Heatmap

(c) Recall Heatmap

(d) F1 Score Heatmap

Figure 5.1: Detection accuracy heatmaps across four evaluation metrics

5.3 Software Performance Results

The software performance analysis focuses on two metrics: the per-window processing time and the memory consumption for each software architecture.
In this study, three time metrics are defined for every processed window: *Processing Time*, *Detector Time*, and *Transport Time*. Based on the implemented measurement logic in the codebase, the equations are as follows.
In the monolithic architecture, all detection logic runs in the same process, so there is no inter-process stage; the transport time only captures the negligible in-process bookkeeping between the outer window boundary and the detector call. As in the other two architectures, it is therefore defined as the difference between processing time and detector time:

$$\begin{aligned} \text{Processing Time} &= t_{\text{end}} - t_{\text{start}} \\ \text{Detector Time} &= t_{\text{detector\_end}} - t_{\text{detector\_start}} \\ \text{Transport Time} &= \text{Processing Time} - \text{Detector Time} \end{aligned} \tag{5.1}$$

where t_{start} and t_{end} are the timestamps at the beginning and the end of the window processing routine, and $t_{\text{detector\_start}}$ and $t_{\text{detector\_end}}$ bound the detector call inside that routine. Because there is no inter-process communication, this residual is zero on average and stays in the millisecond range even at its maximum (Figure 5.4).

For the microservice architecture using Apache Kafka, the processing time is measured from feature message production until result consumption at the engine side:

$$\begin{aligned} \text{Processing Time} &= t_{\text{consume}} - t_{\text{produce}} \\ \text{Detector Time} &= t_{\text{detector\_end}} - t_{\text{detector\_start}} \\ \text{Transport Time} &= \text{Processing Time} - \text{Detector Time} \end{aligned} \tag{5.2}$$

where t_{produce} is when the engine publishes a feature message, $t_{\text{detector\_start}}$ and $t_{\text{detector\_end}}$ are measured inside the detector service, and t_{consume} is when the engine receives the detection result.

For the microservice architecture using gRPC, latency is measured from request emission until response handling:

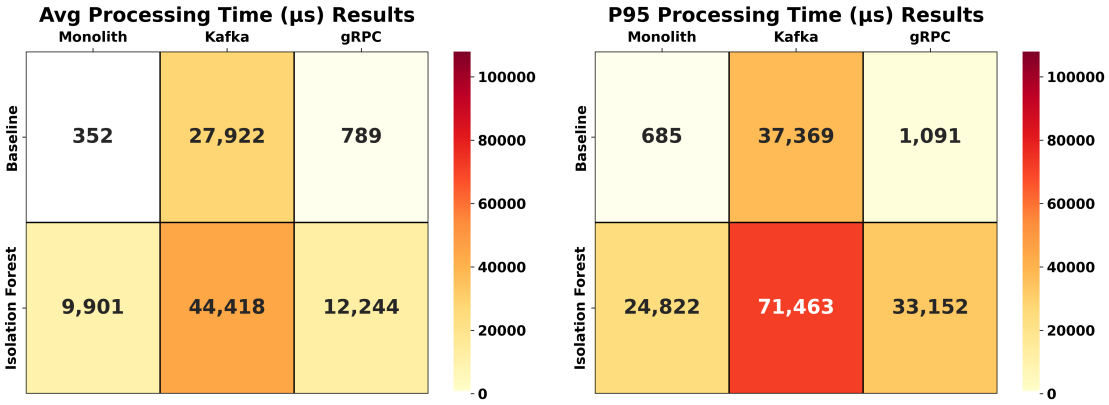
$$\begin{aligned} \text{Processing Time} &= t_{\text{response}} - t_{\text{request}} \\ \text{Detector Time} &= t_{\text{detector\_end}} - t_{\text{detector\_start}} \\ \text{Transport Time} &= \text{Processing Time} - \text{Detector Time} \end{aligned} \tag{5.3}$$

where t_{request} is when the gRPC request is sent by the engine, t_{response} is when the response is received, and detector timestamps are measured in the gRPC detector service.

By using the calculation methods in Equation 5.1, Equation 5.2, and Equation 5.3, time performance is evaluated consistently and accurately across all architectures.

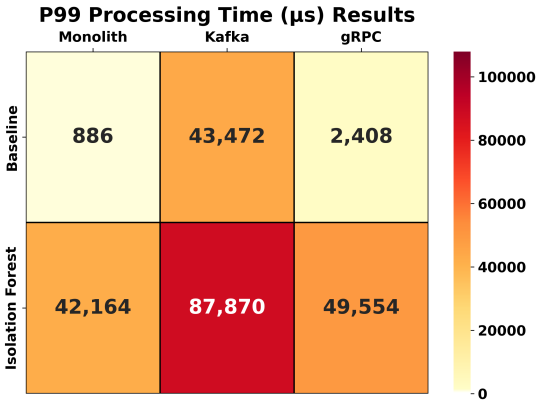
Two points about this processing time should be emphasised. First, it is measured per completed window and never includes packet capture or the accumulation of packets into the one-second sliding window; it covers only the work that follows once a window has closed. Second, the start of the timer differs slightly across architectures: in the monolithic variant it begins at the start of the post-window routine, whereas in the Kafka and gRPC variants it begins only when the engine emits the feature message or request, so both microservice figures start after feature extraction and measure transport plus detection. The reported processing time is therefore a post-window, detector-side latency rather than the full end-to-end latency of the NIDS from packet arrival to verdict, and it should be read as such.

Four sets of data were measured for each time category: average time, 95 percentile time, 99 percentile time, and max time. The time measurements for processing time, detector time, and transport time are shown in Figure 5.2, Figure 5.3, and Figure 5.4, respectively. For better visualisation, the colour of each cell is based on the logarithmic value of the data.

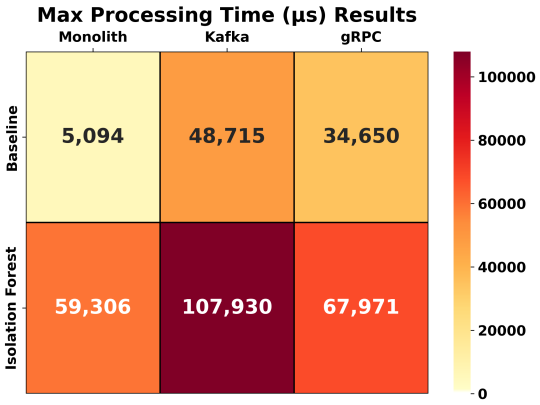


(a) Average Processing Time (μs)

(b) 95 Percentile Processing Time (μs)

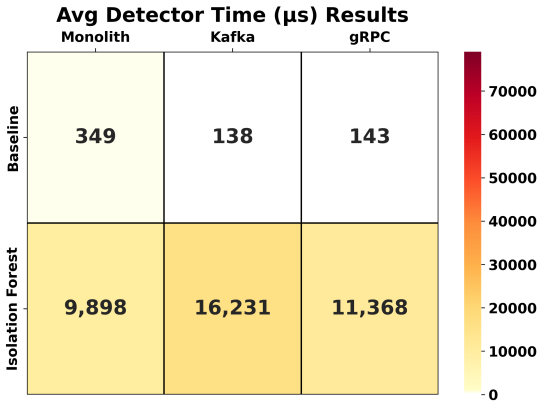


(c) 99 Percentile Processing Time (μs)

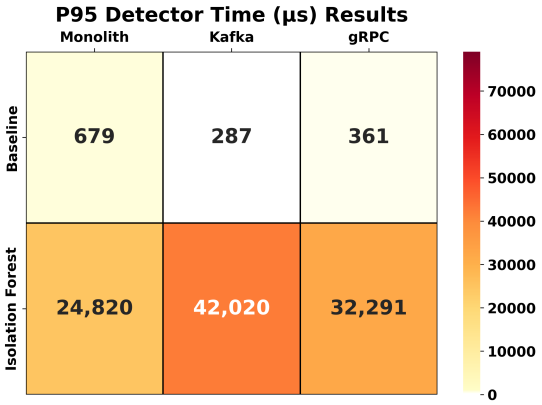


(d) Max Processing Time (μs)

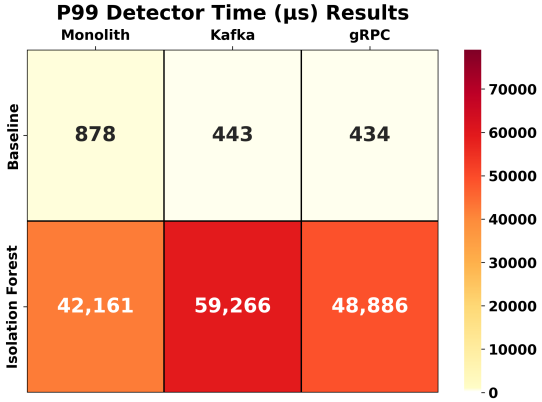
Figure 5.2: Processing Time measurement result heatmap



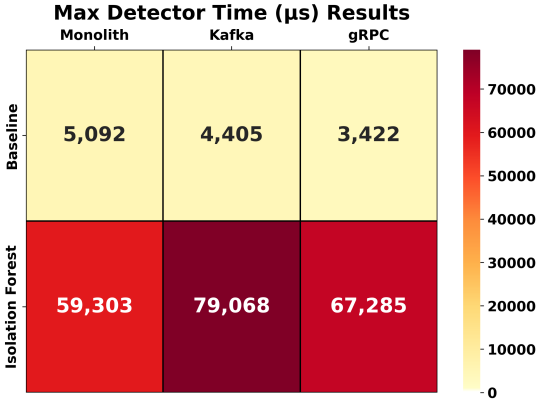
(a) Average Detector Time (μs)



(b) 95 Percentile Detector Time (μs)

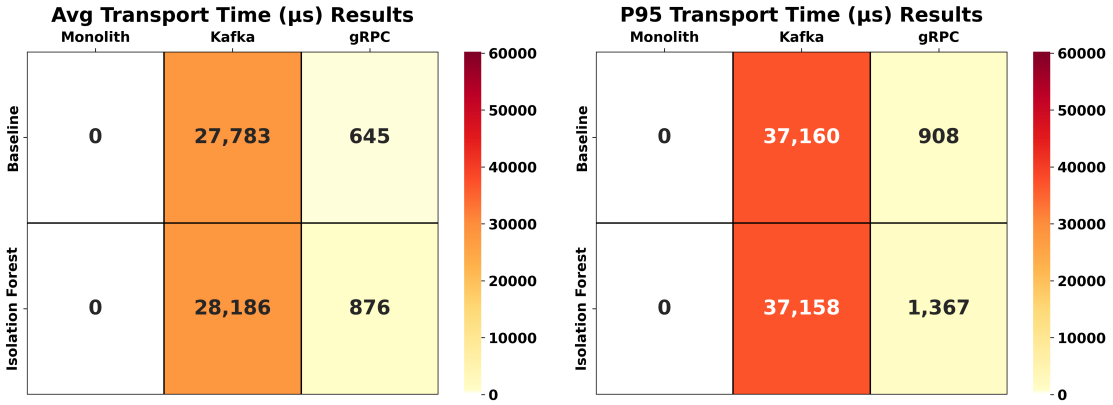


(c) 99 Percentile Detector Time (μs)



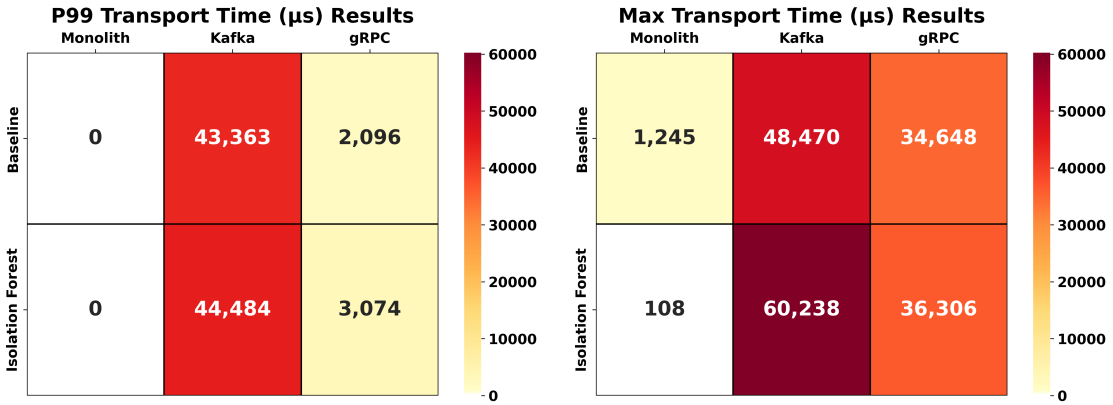
(d) Max Detector Time (μs)

Figure 5.3: Detector Time measurement result heatmap



(a) Average Transport Time (μs)

(b) 95 Percentile Transport Time (μs)

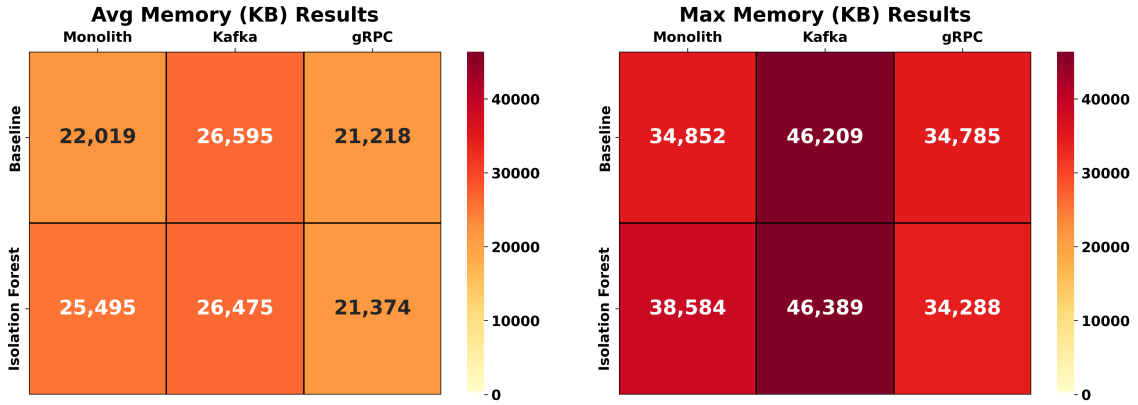


(c) 99 Percentile Transport Time (μs)

(d) Max Transport Time (μs)

Figure 5.4: Transport Time measurement result heatmap

Memory consumption. In addition to the time metrics, the memory footprint reported here is the Go runtime heap allocation (`runtime.MemStats.Alloc`) of the NIDS engine process, sampled once per window. It measures the engine's own footprint only and excludes the separate detector process and, in the Kafka case, the JVM broker, both of which run as separate processes outside the measured one. Figure 5.5 reports the average and maximum memory consumption in kilobytes.
The memory results follow the same ordering across both the average and the maximum case. The Kafka-based engine consistently consumes the most, about 26 MB on average and up to 46 MB at peak. Because the broker and the detector are separate processes, this is not the cost of the broker but of the in-process Kafka client linked into the engine. Its producer/consumer buffers and background client threads are allocated on the engine's heap. The gRPC engine, with a lighter client, stays near 21 MB on average and 34 MB at peak, and the monolithic engine lies between the two.
The choice of detector has a smaller effect on the engine footprint than the choice of transport. Switching from the baseline to Isolation Forest leaves the Kafka and gRPC engine footprints essentially unchanged (within a few hundred kilobytes), while the monolithic configuration shows the only noticeable increase, rising by



(a) Average Memory (KB)

(b) Maximum Memory (KB)

Figure 5.5: Memory consumption measurement result heatmaps

about 3.5 MB on average (from 22,019 KB to 25,495 KB) and 3.7 MB at peak (from 34,852 KB to 38,584 KB). This pattern follows directly from where the detector runs: in the monolithic configuration, the Isolation Forest model lives inside the measured engine process, so its memory appears here, whereas in the Kafka and gRPC configurations, the model runs in the separate detector process and is therefore outside the measured footprint. In all cases the absolute footprint stays far below the 16 GB available on the Raspberry Pi 5, so engine memory is not a limiting factor for any of the evaluated configurations on this platform; within this engine-only view, the transport client rather than the detector dominates the differences.

6
Discussion

This discussion focuses on two parts: the detection accuracy among the different models, and the NIDS software performance regarding the time usage and memory usage. The Raspberry Pi OS is based on Debian and is not a real-time operating system (RTOS). As a result, the recorded time measurements may vary between runs, and the exact values do not carry independent significance. Hence, this chapter discusses the results by comparing the overall trends among each variation rather than relying on arguing the absolute numbers.

6.1 Detection Accuracy

When every variant uses the same 75-window warm-up procedure, detection quality on the Raspberry Pi 5 separates first along the detector axis. The three baseline configurations cluster tightly, with accuracies around 0.405–0.407, recall around 0.396–0.397, and F1 around 0.569, while all three Isolation-Forest configurations sit far above them. Along the transport axis, the picture is more nuanced. For the baseline, the transport has essentially no effect, because the detector receives an identical input and returns an identical verdict whether the window is delivered by an in-process function call, a unary gRPC stub, or a Kafka consumer. For Isolation Forest, the monolithic and gRPC variants are close to each other (accuracy 0.9335 and 0.9150, recall 0.9356 and 0.9151), but the Kafka variant is noticeably lower (accuracy 0.8422, recall 0.8409), about nine percentage points below the monolithic case. We therefore do not claim that detection quality is strictly transport-invariant. The detector is the same Isolation Forest in all three deployments, so the gap is not a property of the algorithm. We did not instrument the pipeline to isolate its cause, so we offer the following as a hypothesis rather than an established mechanism. The gap may be due to the way Kafka delivers windows: because the engine produces feature messages and consumes results asynchronously, and the model is updated online as each window is scored, the broker's buffering and fetch cycle may change the order and timing of these online updates relative to the synchronous in-process and gRPC paths, or delay when a result is applied, leaving the model in a slightly different state when a given window is scored. This interpretation is consistent with the Kafka variant, also showing the largest transport time and the largest processing-time degradation, though confirming it would require logging the per-window delivery and update order. The practical implication is more measured than a clean invariance result: the choice of detector matters far more than the choice of transport. A synchronous transport, such as gRPC, preserves the detec-

tion quality of the in-process deployment, while a heavily decoupled transport, such as Kafka, can introduce a measurable detection penalty that has to be weighed against the durability and decoupling it provides.
The baseline detector has a precision of 1 but low recall, which indicates that there are no false positives, but the baseline model missed the labelled attack windows that contain low-volume malicious traffic. In comparison, Isolation Forest had a precision on par with the baseline model (ranging from 0.996 to 0.999), but the recall climbed, which also increased the F1. This reflects that the Isolation Forest model widens the alarm region compared to the baseline model. This finding also suggests that Isolation Forest has more potential in a DDoS detection scenario, where the subtle and low-volume attack patterns should be discovered and flagged early. In this situation, the Isolation Forest model performs better.
The hyperparameter results also show that the feature-vector and packet-count inputs reach almost the same detection quality on this dataset. We attribute this primarily to the nature of CIC-DDoS2019 [11], which, for the most part, contains high-volume traffic flooding attacks. When the attack signal is characterised mainly by a sharp increase in traffic volume, the per-window packet count is already a near-sufficient indicator of an attack, so the additional flow features contribute little extra separation at the window level. This does not imply that the richer feature vector is generally redundant. Feature selection might further improve the accuracy and F1, but the feature selection part is not in the scope of this thesis and therefore a future work.

6.2 Software Performance Comparison

Regarding transport time, gRPC has a lower transport time compared to Kafka. The average transport time of gRPC is less than 2 milliseconds; however, the average transport time of Kafka is more than 27 ms. This indicates that the cost of switching from an in-process call to a network-based microservice is not uniform across transports: the Kafka path imposes more than an order of magnitude more transport overhead than the gRPC path on the same Raspberry Pi 5 host.
The huge difference between the two transport times could be interpreted as a difference in strategy cost rather than just a difference in network cost. Both microservice variants are co-located on the same Pi 5 and communicate through the kernel loopback interface. The bulk of the 27 ms Kafka transport budget, therefore, reflects the design decisions of the Kafka protocol itself. These include the producer's accumulation of records before flushing, the wakeup of the broker's request handler, the broker-side append to the log and the subsequent acknowledgement, the consumer's fetch poll cycle, and the consumer-side handoff between the network thread and the user-facing detector goroutine, etc. For example, the fetch poll cycle is governed by a parameter named `fetch.max.wait.ms`, which sets the maximum time the broker waits before responding to a fetch request and, by default, introduces a noticeable delay. So the 27 ms measured on the Pi 5 with default tunable is the expected order of magnitude, and is largely the cost of the durability, decoupling, and replay properties that Kafka provides over and above what gRPC offers. The synchronous, single-frame request/response shape of a gRPC unary call avoids most of these costs,

which is why its transport overhead remains in the low milliseconds.

6.3 Switching from Baseline to Isolation Forest

Figure 6.1 shows the improvements and degradation when switching from the baseline model to the Isolation Forest. In an embedded environment provided by Raspberry Pi 5, the switch produces a much higher accuracy across all three software architectures, but it also degrades the processing time in every case. The microservice architecture with Kafka shows the largest degradation, while the monolithic architecture shows the smallest.

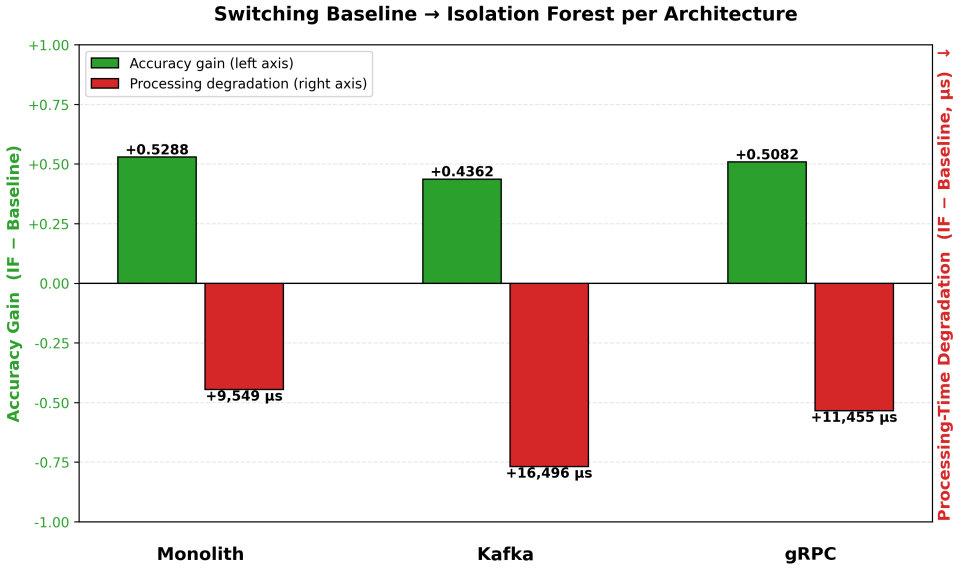


Figure 6.1: Accuracy gain and processing time degradation overview

This finding also suggests a trade-off between accuracy and performance: reaching higher accuracy might require additional time. For practitioners who require a microservice architecture, for instance, to decouple the detection pipeline in cloud-based deployments or a multi-process structure, the gRPC variant emerges as the strongest candidate, since it preserves a comparable accuracy gain while incurring a smaller processing-time degradation than the Kafka variant.

One thing that should be read with caution is that, based on the dataset used in this thesis, the monolithic architecture appears to be the most favourable choice, as it combines the accuracy gain with the smallest processing-time degradation. However, CIC-DDoS2019 mainly contains high-volume attack traffic but might not necessarily exercise the monolithic architecture under conditions in which a single process becomes a bottleneck. It is therefore possible that the monolithic advantage observed here partly reflects the characteristics of the dataset rather than a general property of the architecture. As a result, a systematic study of the conditions under which the monolithic architecture begins to degrade, and of how the microservice variants compare under heavier or more diverse workloads, is left as future work.

6.4 Effectiveness of Sequential Replayer

Wickman and Rygaard [5] introduced the first version of the replayer in their thesis. It was used to replay one PCAP file into the system and process it directly inside NIDS. In the discussion of the future work, they specified the main risks related to the replayer. It is important to identify the bottleneck of the system and understand the limitations of each part, because, as they specified, it does not make sense to improve one efficient part while another part is slowing everything down. Replayer, as part of the old system, is processing large amounts of packets. The replayer sends packets with a determined delay specified as the timestamp difference between consecutive packets in the dataset. The main risk here is that the required inter-packet delay may be shorter than the system's processing time. Consequently, the Replayer may become a bottleneck, failing to transmit packets as fast as needed.
We agree that the efficiency of the system is determined by its bottleneck or the slowest part. In the current NIDS, we should consider the processing speed of the engine, the filtering speed of the eBPF program with XDP hooks, and the detection time of the ML model. The sequential replayer is now extracted and works as an independent program. The processing speed still exists, but it affects the efficiency of the attack and does not impact the efficiency of the NIDS itself. So, the main goal is to verify the replaying speed. In case the replayer is slow, it can affect the performance analysis of the NIDS, because it is easier to defend the system from slower attacking traffic. So, we still need to consider the replayer bottlenecks, but we should not count them as part of the NIDS problems.

6.5 Limitations

This section discusses the limitations of the study. It first considers feature selection for the Isolation Forest detector, an aspect that was not explored within the scope of this thesis, but that is relevant to both detection quality and software performance. It then examines the threats to the validity of the results, covering internal, external, construct, and detection validity.

6.5.1 Feature Selection

The detection accuracy and software performance reported in this thesis were obtained from only two input configurations for the Isolation Forest detector: a single packet-count value per window and the full 80-dimensional flow feature vector produced by GoFlowMeter (Section 4.1.2). These two configurations represent the two extremes of the available feature space rather than a systematic exploration of it, since no intermediate configuration in which an informative subset of the 80 features is selected was evaluated.
This is a limitation for two related reasons. In terms of detection quality, the Isolation Forest algorithm is sensitive to irrelevant or redundant dimensions, because uninformative features can weaken the isolation behaviour that the algorithm relies on [45]. It is therefore possible that a carefully selected subset of features would

yield higher accuracy and F1 scores than the complete 80-dimensional vector, and the present comparison cannot rule this out. In terms of software performance, the size of the feature vector directly affects the cost of the system. A smaller vector reduces the amount of data that has to be assembled, scored by the model, and, in the Kafka-based and gRPC-based architectures, serialised and transported between the engine and the detector. An appropriately reduced feature set could therefore lower memory usage, detection time, and transport time simultaneously, and the magnitude of this combined effect was not measured.
Because feature selection was outside the scope of this study, these results should be read as a comparison between a minimal and a complete feature representation rather than as evidence about the best achievable trade-off between detection quality and architectural overhead.

6.5.2 Validity Threats

This subsection examines the threats to the validity of the study and the extent to which the results can be trusted and generalised. Four categories of validity are considered:

1. Internal validity concerns factors outside the experimental design.
2. External validity concerns how far the findings extend beyond the hardware and dataset used.
3. Construct validity concerns whether the chosen metrics measure what they are intended to measure.
4. Detection validity concerns the limitations of the mitigation mechanisms.

Each of these categories is discussed in turn in the subsections below.

6.5.2.1 Internal Validity

Several factors in the Raspberry Pi 5 environment could have influenced the results independently of the experimental design. The Linux operating system manages background tasks such as Wi-Fi handling and system logging, and these tasks were not isolated from the processor cores used by the detector. As a result, the software experienced unpredictable interruptions that were not introduced or controlled by the experiment. In addition, the CPU clock speed was not fixed, so the heat generated during the 25-minute runs may have caused the processor to change frequency in ways that correlate with the order in which the tests were executed. The Java Virtual Machine used by Kafka introduces a further confound, because its garbage collection pauses occur independently of the workload and do not reflect the efficiency of the underlying architecture.
A second internal threat concerns the correctness of the labels rather than the runtime environment. The labels are consistent across all tests because they are produced by a single fixed rule, but that rule defines an attack on the basis of the ground truth provided with the CIC-DDoS2019 dataset. The dataset authors generated their ground-truth CSV files using the CICFlowMeter, which may itself contain labelling inaccuracies. Any such inaccuracy is inherited by our labelling rule and then propagates into the absolute accuracy and F1 scores reported for every variant, so these absolute figures should be read with that limitation in mind.

6.5.2.2 External Validity

The Raspberry Pi 5 is only one single-board configuration, whereas gateway hardware in real deployments varies considerably. Because software performance depends on the platform, the same Isolation Forest algorithm may behave differently on other gateway machines. The results, therefore, act as an indication of the potential performance of the algorithm in an embedded-like environment rather than as values that transfer directly to arbitrary hardware.
A further external threat is the absence of a realistic traffic representation. CIC-DDoS2019 was released in 2019 and is a synthetically generated dataset, so its traffic distribution does not fully capture the characteristics of present-day network conditions. We were nonetheless constrained to use it because the available private datasets that contain more realistic traffic do not provide labelled data, which makes a quantitative evaluation infeasible on them. Our findings should therefore be interpreted within the scope of this dataset.

6.5.2.3 Construct Validity

To measure detection quality, we used fixed time windows rather than individual packets, and a window is labelled malicious if it contains even a single attack packet. This design reliably answers whether the system noticed an attack, but it has limitations. It can hide short attack bursts, and it can mask cases where a detector flags a window correctly but for the wrong underlying reason, so a high score does not by itself guarantee that the detector responded to the intended signal.
To measure architectural overhead, we recorded memory usage together with average and tail processing times, and we separated transport time from detection time in order to isolate the specific costs of Kafka and gRPC. The monolithic baseline, however, has no inter-process transport stage, so its transport time is effectively zero: it captures only negligible in-process overhead rather than a real transport cost. Any comparison of total processing time across the architectures, therefore, includes this asymmetry, and that component of the difference reflects a structural property of the design rather than a purely empirical measurement.
Finally, the choice of baseline and Isolation Forest is intended to represent a lightweight rule-based detector and an unsupervised machine learning detector. Even so, any conclusion about these algorithm classes applies strictly to these two concrete implementations under the fixed hyperparameters we used, and it should not be read as a broad claim about statistical methods or tree-based anomaly detectors in general.

6.5.2.4 Detection validity

The current mitigation remains per-IP, which is effective for strong single-source anomalies but less effective when the attack load is highly distributed across many low-rate sources. Per-time-window IF has an extended list of features, and it helps significantly to identify the presence of the threat, but per-IP IF still uses only packet count as the main feature. It limits the detection performance and quality of the defence mechanism. In the future, with a better detection mechanism, it will be possible to use more features for a per-IP detector as well.

6.6 Future Work

Based on the methods, findings, and limitations in this study, we proposed some aspects of future work, such as feature selection and utilising Long-Short Term Memory.

6.6.1 Feature Selection

An extension of this work is to evaluate feature selection and dimensionality reduction techniques for the Isolation Forest detector, instead of restricting the input to a single packet count or the full 80-dimensional vector. Future work could identify an informative subset of features and quantify its joint effect on detection quality and on the per-architecture cost, in particular, the serialisation and transport overhead of the Kafka-based and gRPC-based configurations.

6.6.2 Long-Short Term Memory

Another improvement in the future can be new Machine Learning models. There are papers [11], [29], [36] specifying that Long-Short Term Memory (LSTM) models have good detection performance. LSTM can replace Isolation Forest in the future; however, this model may consume much more resources, and with better detection performance, we can get worse performance impact on the device with NIDS.

6.6.3 eBPF/XDP improvements

Current NIDS have eBPF/XDP integration to process and filter real traffic. It is dangerous to have strict filtering based only on IF detection results, because False Positive detection can block a good IP address forever. Since then, the rate limiter from the old system was kept, and the threshold mechanism was used for better and safer detection. In the future, with the usage of better detection mechanisms, such as LSTM models, it would be easier to remove the rate limiter and rely only on the detection mechanism. Also, the block logic can be modified. Instead of blocking an IP forever, we can specify the duration of the block. So, in case of True Positive detection, a good IP address will get access later.

6.7 Ethics

This section reflects on the ethical considerations of the study. Although the work is defensive in purpose, since its goal is to detect and mitigate distributed denial-of-service attacks, it still involves a public traffic dataset and tooling that can generate attack-like traffic, both of which carry ethical implications. The discussion first addresses the responsible use of the dataset, and then the measures taken to prevent the misuse of the system and the artefacts released from it.

6.7.1 Data Ethics

This study mainly utilised the CIC-DDoS2019 dataset, which is a public dataset and was created by artificial settings. It does not contain any private data, and it is specifically for intrusion detection research; we used it solely to measure detection quality. We did not attempt to identify, deanonymize, or otherwise trace any host or individual represented in the data, and no additional personal data was collected during the study.

6.7.2 Prevention of Misuse

A system that characterises attack traffic, together with a standalone replayer that can emit attack-like traffic over a real network interface using eBPF/XDP, inevitably embodies knowledge about how such traffic is produced as well as how it is detected. To ensure this knowledge was not applied harmfully, all attack replay was performed inside an isolated testbed against our own device. No attack traffic was directed at third parties or at the public internet, so no external system or user was placed at risk.
The artefacts we release publicly, in particular GoFlowMeter, are general-purpose flow-analysis tools rather than attack software, which limits their potential for misuse.

7
Conclusion

In this chapter, we summarise the research problem, our main goals, and our contributions.

This master's thesis research is focused on DDoS attack mitigation. DDoS attacks are not new threats, but attackers are evolving and improving their attack mechanisms. Therefore, we need to constantly improve our defence mechanisms using modern tools.

At the beginning of the research, we identified the main problem and two research questions presented in section 1.2. The first question investigated how to improve DDoS detection effectiveness using modern machine learning and fast packet filtering tools. The second question examined the performance impact across different architectures. Both of these questions have been successfully addressed.

We significantly improved an existing NIDS with eBPF/XDP integration, the new GoFlowMeter feature extractor, a standalone attacker program, a comparison of three deployment architectures, and a fine-tuned IF model. We also added a GUI to monitor traffic processing, memory usage, and time delays (see our contribution in Section 1.4).

Our research identified the most suitable Isolation Forest configuration for the NIDS setup: the feature-vector input with 200 trees, a sample size of 128, and an anomaly fraction of 0.2. The combination performs the best among the two test modes. These hyperparameters were chosen with the offline probe, in which this configuration reached a mean accuracy of 0.873 and a mean F1 of 0.932 over the dataset. When the same configuration was run live, the monolithic Isolation Forest reached an accuracy of 0.9335 and an F1 of 0.9650 (Figure 5.1); the probe and live figures differ because the probe scores the model offline, whereas the live run includes the warm-up and online-update behaviour of the deployed engine. We did not perform feature selection; instead, we compared two input modes: a single packet count per window and the full 80-dimensional feature vector, and the feature-vector input gave higher accuracy and F1 than the packet-count input. Both clearly improved on the statistical baseline.

The new architecture research showed the advantage of the gRPC-Microservice architecture over the Kafka-based architecture.

This research helps the industry to understand and mitigate modern threats more effectively.

Wickman and Rygaard (2025) [5] in their master's thesis specified their unsolved questions and possible future work, including the problem of adaptability of the threshold and the limitations of the data set replayer.

We solved some of the questions. The GoFlowMeter provides the system with

a variety of features to help identify threats with better accuracy. Because the detection mechanism does not rely solely on a single feature, it resolves the issue of threshold adaptability. Because the detector no longer relies on a single feature, it is better able to flag low-volume attack windows that the packet-count baseline misses. However, CIC-DDoS2019 is dominated by high-volume flooding traffic, so reliably detecting genuinely slow-growing DDoS traffic remains to be demonstrated on datasets that contain more of it.

We specified the limitations of the current version of the NIDS. The IF model could be replaced with a more accurate model, such as LSTM. The model is more resource-intensive; its usage should be investigated in both the detection performance and the device performance impact. Additional feature analysis can also improve the detection performance. Furthermore, eBPF implementation can be improved by replacing permanent IP blocks with time-bounded blocks so that a falsely flagged IP can recover access. We are looking forward to the future improvements.

Bibliography

[1] V. Morel, *Victor morel – phd in computer science / researcher in privacy & security*, https://victor-morel.net/, Accessed: 2026-05-27, 2026.

[2] Cloudflare, *Aisuru botnet: Early october attacks escalate into record-setting ddos activity*, Accessed: 2026-03-18, Dec. 2025. [Online]. Available: https://www.cloudflare.com/threat-intelligence/research/report/aisuru-botnet/.

[3] B. Ferreira, "Botnet smashes ddos traffic record, equivalent to streaming 2.2 million netflix 4k movies at once — 31.4 tb/s attack was large enough to take entire countries offline," *Tom's Hardware*, Jan. 2026, Accessed: 2026-03-18. [Online]. Available: https://www.tomshardware.com/service-providers/network-providers/botnet-smashes-ddos-traffic-record-at-31-4-tb-s-equivalent-to-streaming-2-2-million-netflix-4k-movies-at-once-attack-was-large-enough-to-take-entire-countries-offline.

[4] T. Høiland-Jørgensen et al., "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66, ISBN: 9781450360807. DOI: 10.1145/3281411.3281443. [Online]. Available: https://doi.org/10.1145/3281411.3281443.

[5] A. Wickman and D. Rygaard, "Scalable anomaly-based network intrusion detection using a statistical model and data sketches," M.S. thesis, Chalmers University of Technology, 2025.

[6] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 413–422. DOI: 10.1109/ICDM.2008.17.

[7] Apache Software Foundation, *Apache kafka*, Accessed: 2026-05-05, May 2026. [Online]. Available: https://kafka.apache.org/.

[8] V. Gogineni, "Apache kafka for low latency ai-enhanced data analytics for fraud detection system," in *2025 IEEE 6th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*, 2025, pp. 1720–1724. DOI: 10.1109/AINIT65432.2025.11035547.

[9] C. Nimpattanavong, I. Khan, T. Van Nguyen, R. Thawonmas, W. Choensawat, and K. Sookhanaphibarn, "Improving data transfer efficiency for ais in the darefightingice using grpc," in *2023 8th International Conference on Business and Industrial Research (ICBIR)*, 2023, pp. 286–290. DOI: 10.1109/ICBIR57571.2023.10147629.

[10] S. Wu and O. Koshovyi, *Goflowmeter: A High-Performance Network Flow Feature Extractor in Go*, https://github.com/Bi9River/goflowmeter, Accessed: 2026-03-11, 2024.

[11] I. Sharafaldin, A. H. Lashkari, S. Hakak, and A. A. Ghorbani, "Developing realistic distributed denial of service (ddos) attack dataset and taxonomy," in *2019 International Carnahan Conference on Security Technology (ICCST)*, 2019, pp. 1–8. DOI: 10.1109/CCST.2019.8888419.

[12] Ericsson. "Mini-link 6600," Ericsson, Accessed: Nov. 13, 2025. [Online]. Available: https://www.ericsson.com/en/portfolio/networks/ericsson-radio-system/mobile-transport/microwave/split-mount-shorthaul/mini-link-6600.

[13] R. A. Bridges, T. R. Glass-Vanderlan, M. D. Iannacone, M. S. Vincent, and Q. (Chen, "A survey of intrusion detection systems leveraging host data," *ACM Comput. Surv.*, vol. 52, no. 6, Nov. 2019, ISSN: 0360-0300. DOI: 10.1145/3344382. [Online]. Available: https://doi.org/10.1145/3344382.

[14] A. Habibi Lashkari, G. Draper Gil, M. S. I. Mamun, and A. A. Ghorbani, "Characterization of tor traffic using time based features," in *Proceedings of the 3rd International Conference on Information Systems Security and Privacy - ICISSP*, INSTICC, SciTePress, 2017, pp. 253–262, ISBN: 978-989-758-209-7. DOI: 10.5220/0006105602530262.

[15] D. Popescul, "The confidentiality – integrity – accessibility triad into the knowledge security. a reassessment from the point of view of the knowledge contribution to innovation," vol. 4, pp. 978–, Jun. 2011.

[16] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013. DOI: 10.1109/SURV.2013.031413.00127.

[17] Cloudflare, *What is the OSI model?* https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/, Accessed: 2026-05-25, 2026.

[18] A. R. Yusof, N. I. Udzir, and A. Selamat, "Systematic literature review and taxonomy for ddos attack detection and prediction," *International Journal of Digital Enterprise Technology*, vol. 1, no. 3, pp. 292–315, 2019. DOI: 10.1504/IJDET.2019.097849. eprint: https://www.inderscienceonline.com/doi/pdf/10.1504/IJDET.2019.097849. [Online]. Available: https://www.inderscienceonline.com/doi/abs/10.1504/IJDET.2019.097849.

[19] Cloudflare, *Smurf DDoS Attack*, https://www.cloudflare.com/learning/ddos/smurf-ddos-attack/, Accessed: 7 December 2025, 2025.

[20] Cloudflare, *Ping (ICMP) Flood DDoS Attack*, https://www.cloudflare.com/learning/ddos/ping-icmp-flood-ddos-attack/, Accessed: 7 December 2025, 2025.

[21] Cloudflare, *SYN Flood DDoS Attack*, https://www.cloudflare.com/learning/ddos/syn-flood-ddos-attack/, Accessed: 7 December 2025, 2025.

[22] Cloudflare, *UDP Flood DDoS Attack*, https://www.cloudflare.com/learning/ddos/udp-flood-ddos-attack/, Accessed: 7 December 2025, 2025.

[23] Cloudflare. "DNS amplification DDoS attack," Cloudflare, Inc., Accessed: Jun. 11, 2026. [Online]. Available: https://www.cloudflare.com/learning/ddos/dns-amplification-ddos-attack/.

[24] V. Shirsath, *Caida ucsd ddos 2007 attack dataset*, 2023. DOI: 10.21227/dvp9-s124. [Online]. Available: https://dx.doi.org/10.21227/dvp9-s124.

[25] I. Sharafaldin, A. Habibi Lashkari, and A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," Jan. 2018, pp. 108–116. DOI: 10.5220/0006639801080116.

[26] P. Goldschmidt and D. Chudá, "Network intrusion datasets: A survey, limitations, and recommendations," *Computers & Security*, vol. 156, p. 104 510, Sep. 2025, ISSN: 0167-4048. DOI: 10.1016/j.cose.2025.104510. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2025.104510.

[27] A. Gharib, I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "An evaluation framework for intrusion detection dataset," in *2016 International Conference on Information Science and Security (ICISS)*, 2016, pp. 1–6. DOI: 10.1109/ICISSEC.2016.7885840.

[28] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani, "Characterization of encrypted and vpn traffic using time-related features," in *Proceedings of the 2nd International Conference on Information Systems Security and Privacy - ICISSP*, INSTICC, SciTePress, 2016, pp. 407–414, ISBN: 978-989-758-167-0. DOI: 10.5220/0005740704070414.

[29] T. Farasat, J. Kim, and J. Posegga, "Smartx intelligent sec: A security framework based on machine learning and ebpf/xdp," Oct. 2024. DOI: 10.48550/arXiv.2410.20244.

[30] K. Tebbaa, O. Chakir, Y. Maleh, and M. Belaıssaoui, "Mitigating DDoS attacks in software-defined networks: A systematic literature review of machine learning and deep learning approaches," *Iranian Journal of Computer Science*, 2025. DOI: 10.1007/s42044-025-00386-x. [Online]. Available: https://doi.org/10.1007/s42044-025-00386-x.

[31] M. Reis and C. Serodio, "Edge ai for real-time anomaly detection in smart homes," *Future Internet*, vol. 17, p. 179, Apr. 2025. DOI: 10.3390/fi17040179.

[32] gRPC Authors, *Introduction to grpc*, Accessed: 2026-05-05, May 2026. [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction/.

[33] S. Popić, D. Pezer, B. Mrazovac, and N. Teslić, "Performance evaluation of using protocol buffers in the internet of things communication," in *2016 International Conference on Smart Systems and Technologies (SST)*, 2016, pp. 261–265. DOI: 10.1109/SST.2016.7765670.

[34] A. R. Shaaban, E. Abd-Elwanis, and M. Hussein, "Ddos attack detection and classification via convolutional neural network (cnn)," in *2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*, 2019, pp. 233–238. DOI: 10.1109/ICICIS46948.2019.9014826.

[35] D. Akgun, S. Hizal, and U. Cavusoglu, "A new ddos attacks intrusion detection model based on deep learning for cybersecurity," *Computers & Security*, vol. 118, p. 102 748, 2022, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2022.102748. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404822001432.

[36] M. Shurman, R. Khrais, and A. Yateem, "Dos and ddos attack detection using deep learning and ids," *International Arab Journal of Information Technology*, vol. 17, pp. 655–661, Jul. 2020. DOI: 10.34028/iajit/17/4A/10.

[37] A. Djama, M. Maazouz, H. Kheddar, M. Rahim, and B. Bengherbia, "High performance detection of ddos attacks with hybrid cnn-lstm architecture," in *2025 International Conference on Intelligent Computer Systems, Data Science and Applications (IC2SDA)*, 2025, pp. 1–6. DOI: 10.1109/IC2SDA68097.2025.11331388.

[38] P. Bhawsar et al., "Machine learning-based detection of ddos attacks: An evaluation of svm and random forest approaches," in *2025 13th International Conference on Intelligent Systems and Embedded Design (ISED)*, 2025, pp. 61–67. DOI: 10.1109/ISED67359.2025.11405254.

[39] V. Vinothina, V. Prakash, K. Revathi, C. Kumar, N.Karthikeyen, and G.Prathap, "An efficient ddos attack detection scheme using autoencoder based on feature selection approaches," in *2025 Third International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)*, 2025, pp. 372–377. DOI: 10.1109/ICAISS61471.2025.11042065.

[40] E. P. Ghani, A. Sofwan, and M. Somantri, "Ai-driven network security: Detecting and mitigating ddos, malware, and backdoor attacks with isolation and random forest algorithm," in *2025 International Conference on Smart Computing, IoT and Machine Learning (SIML)*, 2025, pp. 1–6. DOI: 10.1109/SIML65326.2025.11080951.

[41] T. Farasat, J. Kim, and J. Posegga, *Advancing network security: A comprehensive testbed and dataset for machine learning-based intrusion detection*, 2024. arXiv: 2410.18332 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2410.18332.

[42] D. Kuryazov, D. Jabborov, and B. Khujamuratov, "Towards decomposing monolithic applications into microservices," in *2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT)*, 2020, pp. 1–4. DOI: 10.1109/AICT50176.2020.9368571.

[43] The kernel development community, *Perf ring buffer — the linux kernel documentation*, https://docs.kernel.org/userspace-api/perf_ring_buffer.html, Accessed: 2026-05-27, 2026.

[44] Raspberry Pi, *Raspberry pi 5*, Accessed: 2026-05-05, May 2026. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-5/.

[45] M. Heigl, K. A. Anand, A. Urmann, D. Fiala, M. Schramm, and R. Hable, "On the improvement of the isolation forest algorithm for outlier detection with streaming data," *Electronics*, vol. 10, no. 13, 2021, ISSN: 2079-9292. DOI: 10.3390/electronics10131534. [Online]. Available: https://www.mdpi.com/2079-9292/10/13/1534.