8-7-2018 2:00 PM

# Real-time Intrusion Detection using Multidimensional Sequence-to-Sequence Machine Learning and Adaptive Stream Processing

Gobinath Loganathan
*The University of Western Ontario*

Supervisor
Samarabandu, Jagath
*The University of Western Ontario* Co-Supervisor
Wang, Xianbin
*The University of Western Ontario*

Graduate Program in Electrical and Computer Engineering
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Engineering Science

# Abstract

A network intrusion is any unauthorized activity on a computer network. There are host-based and network-based Intrusion Detection Systems (IDS's), of which there are each signature-based and anomaly-based detection methods. An anomalous network behavior can be defined as an intentional violation of the expected sequence of packets. In a real-time network-based IDS, incoming packets are treated as a stream of data. A stream processor takes any stream of data or events and extracts interesting patterns on the fly. This representation allows applying statistical anomaly detection using sequence prediction algorithms as well as using a stream processor to perform signature-based intrusion detection and sequence extraction from a stream of packets. In this thesis, a Multidimensional Sequence to Multidimensional Sequence (MSeq2MSeq) encoder-decoder model is proposed to predict sequences of packets and an adaptive and functionally auto-scaling stream processor: "Wisdom" is proposed to process streams of packets. The proposed MSeq2MSeq model trained on legitimate traffic is able to detect Neptune Denial of Service (DoS) attacks, and Port Scan probes with 100% detection rate using the DARPA 1999 dataset. A hybrid algorithm using Particle Swarm Optimization (PSO) and Bisection algorithms was developed to optimize Complex Event Processing (CEP) rules in "Wisdom". Adaptive CEP rules optimized by the above algorithm was able to detect FTP Brute Force attack, Slow Header DoS attack, and Port Scan probe with 100% detection rate while processing over 2.5 million events per second. An adaptive and functionally auto-scaling IDS was built using the MSeq2MSeq model and "Wisdom" stream processor to detect and prevent attacks based on anomalies and signature in real-time. The proposed IDS adapts itself to obtain best results without human intervention and utilizes available system resources in functionally auto-scaling deployment. Results show that the proposed IDS detects FTP Brute Force attack, Slow Header DoS attack, HTTP Unbearable Load King (HULK) DoS attack, SQL Injection attack, Web Brute Force attack, Cross-site scripting attack, Ares Botnet attack, and Port Scan probe with a 100% detection rate in a real-time environment simulated from the CICIDS 2017 dataset.

**Keywords:** Intrusion Detection, MSeq2MSeq, Machine Learning, Stream Processing

# Acknowlegements

I would first like to express my sincere gratitude to my supervisor Dr. Jagath Samarabandu for the opportunity, guidance and continuous support throughout the research. Prof. Samarabandu was always there whenever I ran into a trouble spot or had a question about my research or writing. Without his support, the work presented in this thesis would not be possible. Furthermore, I would like to thank Dr. Xianbin Wang for his feedback on my writing and support to find funding for this research.

I would also like to thank my labmates Nadun Rajasinghe and Wafaa Anani for their valuable feedback shared throughout the lab meetings and informal discussions we had. Friends shape us and support us whenever we are in need. I have no words to thank Sayon, Malar, Jerad, Sri, Vinothan, Sathya, Yogananth, and Kirubana for their support from the day one until now. Though we were far away from each other, I would not be able to complete my research without the moral and emotional support of my family. I would like to express my very profound gratitude to my parents, aunt, sister, brother, and my fiance for their love and support.

I also would like to thank professors at Western including Dr. Vijay Parsa, Dr. Aleksander Essex, Dr. Roy Eagleson, Dr. Roberto Solis-Oba, Dr. Olga Veksler, and Dr. Hamada Ghenniwa who provided me knowledge and guidance. Last but not least, thank you very much for all the staff in Western University for taking good care of me.

# Contents

# List of Figures

# List of Tables

# List of Appendices

# List of Abbreviations

**BiRNN** Bidirectional Recurrent Neural Network

**BOA** Bayesian Optimization Algorithm

**CEP** Complex Event Processing

**DDoS** Distributed Denial of Service

**DoS** Denial of Service

**FTP** File Transfer Protocol

**GA** Genetic Algorithm

**HTTP** Hypertext Transfer Protocol

**HULK** HTTP Unbearable Load King

**ICMP** Internet Control Message Protocol

**IDS** Intrusion Detection System

**IoT** Internet of Things

**LSTM** Long Short-Term Memory

**MQTT** Message Queuing Telemetry Transport

**MSeq2MSeq** Multidimensional-Sequence-to-Multidimensional-Sequence

**NMT** Neural Machine Translation

**pcap** packet capture

**PSO** Particle Swarm Optimization

**RNN** Recurrent Neural Network

**Seq2Seq** Sequence-to-Sequence

**SQL** Structured Query Language

**TCP** Transmission Control Protocol

**TLS** Transport Layer Security

**UDP** User Datagram Protocol

# Chapter 1

# Introduction

## 1.1 Motivation

The rise in popularity of cloud computing has caused data and software housed in local workstations to migrate to remote servers. The Internet of Things (IoT) and mobile devices heavily rely on services provided over the Internet. Availability of user data in a central point and the single point of failure attract attackers to breach servers. Intrusion Detection Systems are being used to detect such attacks in a network or a host. However, these days there are a lot of tools developed to automate attacks. New attacks are being invented by evading the detection techniques used to prevent existing attacks. For example, GoldenEye Denial of Service (DoS) [1] attack evades HTTP Unbearable Load King (HULK) DoS [2] detectors by adding randomness to packets. Therefore, an Intrusion Detection System (IDS) should be able to detect unknown attacks or at least should be able to adapt to detect evolving attacks.

There are host-based and network-based intrusion detection systems, of which there are each signature and anomaly based methods [3]. Anomaly-based systems detect intrusions based on anomalous behaviors observed in a network. There are no ideal machine learning models to detect anomalous packets with a 0 prediction accuracy. Therefore a minimum prediction accuracy threshold is used to classify anomalous packets. Such anomaly-based IDS's

require a large amount of anomalous traffic to detect attacks with enough confidence. For example, in a *SYN* Flood DoS attack, the attacker sends a large number of *SYN* requests to the victim, which is a noticeable anomalous traffic in a network. However, attacks targeting selected system vulnerabilities using a few number of packets are hard to detect for anomaly-based systems. Signature-based systems detect attacks based on predefined attack specific rules regardless of the number of packets involved in the attack. For example, a Structured Query Language (SQL) Injection attack is easy to detect by looking for SQL keywords in request parameters.

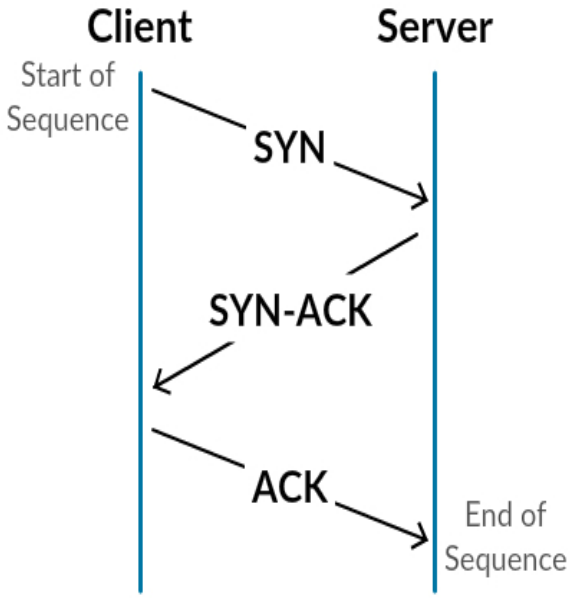



Figure 1.1: **TCP Three-way Handshake** – a legitimate TCP sequence starts with *SYN*, followed by *SYN-ACK* and ends with *ACK*.

In this research, an anomaly-based network intrusion is defined as an intentional violation of expected behavior or network protocols. A network protocol is a well-defined sequence of packets often represented by a state machine. For example, a Transmission Control Protocol (TCP) handshake as shown in Figure 1.1 is a sequence of three TCP packets with TCP flags: (*SYN, SYN-ACK, ACK*). An intentional violation of this rule in Direct SYN Flood DoS attack

as depicted in Figure 1.2 is an anomalous sequence with a large number of (*SYN, SYN-ACK, SYN, SYN-ACK, ...*) packets in which the order of *SYN-ACK* may differ. The ability of machine learning algorithms trained on legitimate traffic to detect unseen attacks makes them suitable for anomaly-based IDS's. Researchers have already proposed several machine learning algorithms to detect network intrusions [4, 5]. To the best of my knowledge, none of them except Bontemps *et al.* have taken advantage of the sequential relationship of packets in a legitimate network connection to detect intrusions. Bontemps *et al.* aligned packets arrived in a predefined interval as a sequence and trained a stacked Long Short-Term Memory (LSTM) Recurrent Neural Network (RNN) to predict last $n - 3$ packets using first 3 packets of a sequence where $n$ is the number of packets in a sequence [6]. However, their definition of the sequence may include completely unrelated packets from different connections which can drastically reduce the accuracy. Grouping packets arrived in a predefined interval as a sequence gives rise to unrealistic sequences depending on external factors such as peak business time.

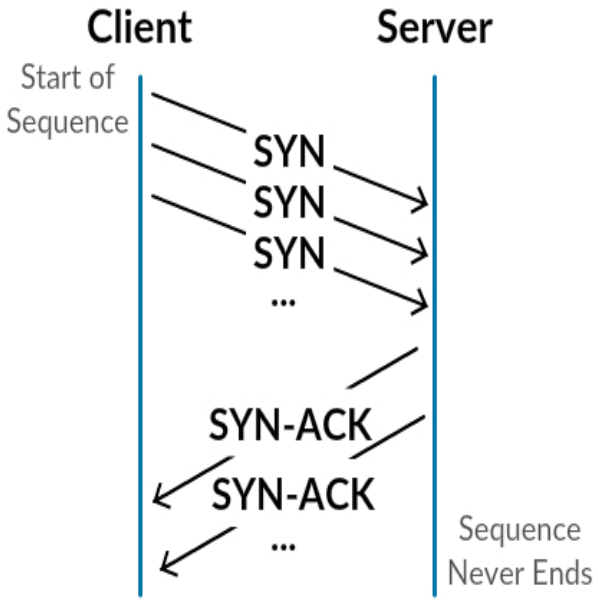



Figure 1.2: **Direct SYN Flood** – an intentional violation of TCP three-way handshake in which the attacker keeps sending *SYN* requests.

Sommer and Paxson discussed some of the challenges in using machine learning for intrusion detection including a) high cost of errors, b) semantic gaps like alert generation and organization-specific policies, c) diversity of network traffic, and d) lack of appropriate public datasets [7]. Even though these problems still exist in machine learning based intrusion detection, signature-based IDS's like Snort [8] are less affected by these problems because a carefully developed signature-based rule generates a relatively smaller number of false alarms than machine learning methods [3]. Commercial signature-based IDS's provide the necessary infrastructure to send alarms to interested stakeholders. In addition, signatures developed by domain experts do not require public datasets to define those rules. However, diversity of network traffic is a problem for signature-based IDS's. For example, receiving more traffic in a peak season is normal for an online retail service but receiving a similar amount of traffic on a regular day can be a symptom of DoS attack. Signature-based IDS's must be able to adapt according to external conditions to overcome this problem. In addition, existing IDS's do not utilize available resources other than network packets. Using all data sources like network packets, application logs, system logs, and policy changes can improve the detection ratio.

Complex Event Processing (CEP) is a reactive programming paradigm used in responding to real-time events based on predefined rules. Stream processors provide the necessary infrastructure to develop and deploy CEP rules for a wide range of applications including intrusion detection [9], healthcare [10], fleet management [11], and power grid [12]. Commercial stream processors like Apache Flink [13], Esper [14], and WSO2 Stream Processor [15] support different input sources and output sinks like File, Email, Hypertext Transfer Protocol (HTTP) connection, TCP connection, Message Queuing Telemetry Transport (MQTT) connection, Databases, and Message Queues [16, 13]. Some products even support writing custom receivers and producers with minimal effort [16, 13]. The ability of stream processors to receive data from multiple sources and emit events to different receivers makes them a better alternative to signature-based IDS's. SQL like stream processing queries offered by most of the stream processors are much more expressive and easy to learn than IDS rules. A common

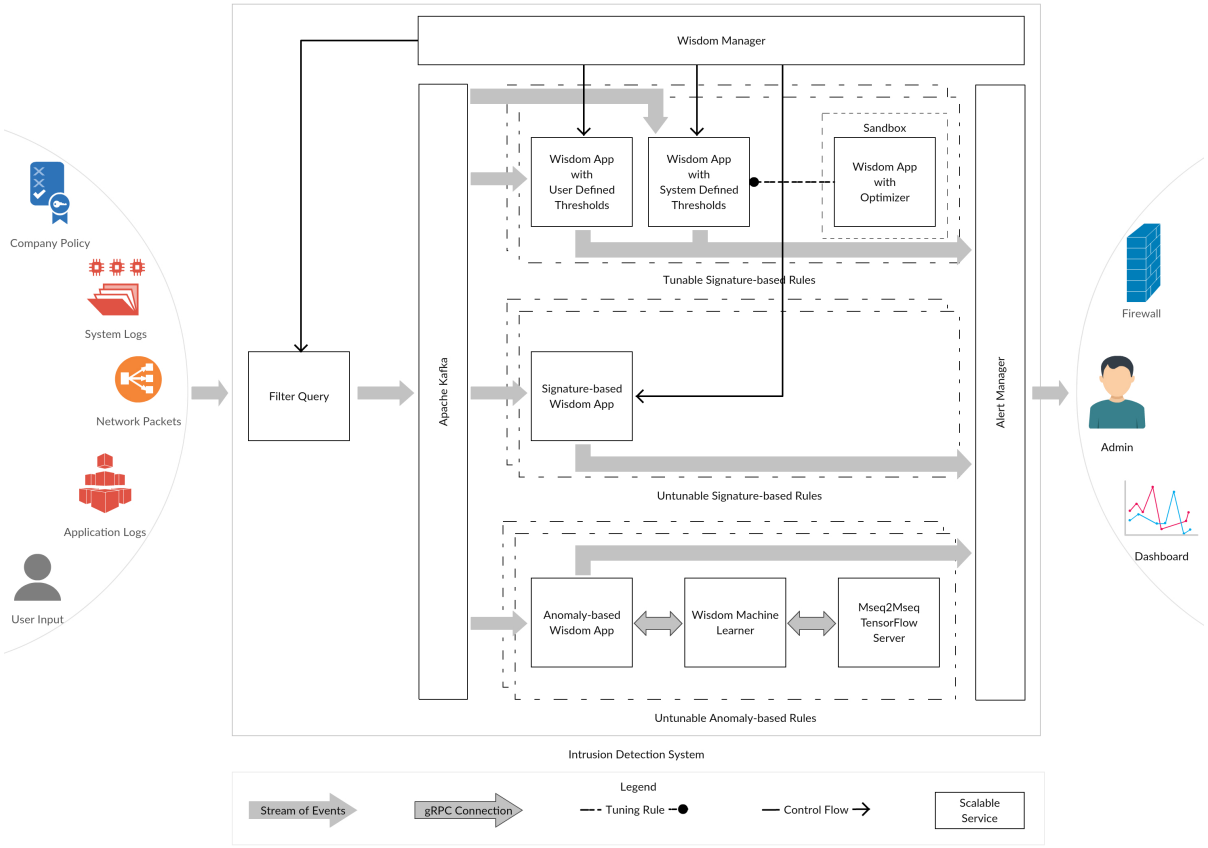



Figure 1.3: **Intrusion Detection System** – a high-level architecture of the adaptive and functionally auto-scaling IDS developed using Wisdom Stream Processor and MSeq2MSeq model.

weakness of existing stream processors is their inability to adapt according to external conditions. There were several attempts made on dynamic query deployment [17], adaptive stream processing [18], and automatic query mining with an intention to replace domain experts with machines [19, 20, 21, 22]. However, none of them were used outside of research environments due to the unrealistic assumptions made by researchers such as raw events not being complex, or a single CEP rule template being able to represent all complex events. Therefore, stream processors still require manual deployment of new queries when there is a change in requirement and do not address the challenge of "diversity of network traffic" when it comes to intrusion detection using stream processors.

In this research, a network-based IDS was built using both anomaly-based and signature-based detection techniques as shown in Figure 1.3 with a preliminary focus on detecting attacks

in real-time. Even though the proposed IDS is tested using network packets, the signature-based rules of the proposed IDS can be used as a host-based IDS. The proposed IDS can be hosted anywhere in a network and fed with packets captured using network monitoring tools like Wireshark [23] and other required information like application logs in real-time. The anomaly-based intrusion detection problem is mapped to multidimensional sequential anomaly detection problem (see chapter 3.1 for details) and a novel Multidimensional-Sequence-to-Multidimensional-Sequence (MSeq2MSeq) model based on the traditional Sequence-to-Sequence (Seq2Seq) model [24] is proposed for real-time anomaly-based intrusion detection [25]. The model was trained using packets from regular network traffic split into sequences. In testing, actual packets highly deviating from predicted packets are classified as anomalies. Training the model on normal traffic instead of intrusion traffic gives access to large training data and lets the model detect even new unknown attacks those are deviating from a regular pattern. Humans are better than machines at designing logical CEP rules due to their cognition. However, expecting a human to modify deployed rules in real-time based on the dynamics of the operating environment is not practical. To address this, a new adaptive and *functionally auto-scaling* stream processor: "Wisdom" has been developed which can optimize its queries automatically using *swarm intelligence*[1] [27]. We use the term *"functionally auto-scaling"* to mean the ability of "Wisdom" stream processor to add more features by starting or to reduce resource consumption by stopping unwanted rules. Using such a self-tuning stream processor in intrusion detection solves the problem of "diversity of network traffic" by adjusting thresholds on the fly based on the dynamics of the environment. Humans also need an expressive language to code their knowledge in defining an attack signature. Semantics used to define rules in popular IDS's like Snort are not expressive enough and hard to interpret for a novice user. Therefore, "Wisdom" supports SQL like "Wisdom Query" language which is expressive enough for humans to code their knowledge. At the same time, "Wisdom Query" also provides variables and annotations which are used by the stream processor itself to optimize the rule and to automate deploy-

[1] Intelligent behavior of a decentralized population emerged by intractability and non-representability of individual agents [26]. Swarm Intelligence is widely being used in decision making and optimization algorithms.

ment. “Wisdom” stream processor is used to build other core components of the IDS including signature-based detectors. “Wisdom” stream processor supports the following input sources: HTTP endpoints, gRPC [28] endpoints, Apache Kafka [29] Topics, CSV files, and packet capture (pcap) files from which events may be read. Output event sinks implemented in “Wisdom” include: HTTP endpoints, gRPC endpoints, Apache Kafka Topics, Console and Text files. New input sinks and output source can be developed with minimal effort using “Wisdom” extension library[2]. Variety of sources and sinks let the IDS read data from multiple sources including raw network packets, system logs, application logs, and user feedback and send alerts to different stakeholders like system admin, network firewall or even a dashboard for statistical purposes. However, within the IDS, Apache Kafka is used to provide loosely coupled communication between “Wisdom” applications[3]. Quality of a self-tuning CEP rule is highly determined by the data used to tune threshold values at the runtime. Since we do not have control over what is received at the runtime, the self-tuned rule may miss some attacks which could be detected with original user-defined thresholds. To overcome this problem, a *Minimum Rate Guaranteed* deployment of tunable signature-based rules is proposed (see Figure 1.3) using two additional clones of the actual CEP rule which are used to tune the CEP rule in a sandbox and to detect attacks using tuned CEP rule along with the actual rule (discussed in chapter 4.6.1).

In addition to the adaptiveness of stream processor, particular attention was paid on utilizing system resources for intrusion detection. Stream processors like Apache Flink [13] supports distributed deployment of rules where each CEP operators can be scaled up and down. Even though stream processors can distribute and scale operators, not all stateful operators are horizontally scalable [30]. Especially when it comes to dynamic CEP operators, it is hard to track and atomically update them in a horizontally scaled environment. Therefore, a novel functionally auto-scaling deployment of CEP rules in a distributed environment is proposed to utilize the available resources. As depicted in Figure 1.3, each “Wisdom” rule deployed in the IDS

[2]How to write a Wisdom extension is explained at `https://slgobinath.github.io/wisdom/wisdom-extensions`

[3]In this thesis, “Wisdom application”, “Wisdom rule” and “Wisdom query” are used interchangeably to refer a CEP logic developed using Wisdom stream processor.

is a standalone microservice [31] application. Therefore they can be scaled up and down individually depending on the requirement. The "Wisdom Manager" and "Filter Query" along with other rules form a functionally auto-scaling infrastructure to reduce the overall resource consumption of the IDS. In this deployment, "Wisdom" queries with low priority are kept off by the "Wisdom Manager" until some events related to those queries are sent by the "Filter Query". More details about functionally auto-scaling deployment is discussed in Chapter 4.6.2 and the minimum resource consumption of the proposed functionally auto-scaling IDS is demonstrated in Chapter 5.3.

The proposed machine learning model itself cannot solve the problems addressed by Sommer and Paxson. However, the signature-based IDS built on top of the adaptive stream processor generates a smaller number of false positives, and false negatives and adapts at the runtime based on the diversity of network traffic. The ability of "Wisdom" stream processor in receiving and sending events through various protocols let the proposed IDS receive organization-specific policies and send alerts to different stakeholders. Signature-based IDS's do not require datasets to define rules because attack signatures are developed by domain experts based on how an attack works. It makes signature-based IDS's not affected by the availability of public dataset. Therefore, the proposed hybrid IDS supporting both signature-based detection and anomaly-based detection successfully overcomes common challenges encountered in machine learning based IDS's. In Phase 1 evaluation, the proposed MSeq2MSeq model was compared with the LSTM RNN model used by Bontemps *et al.* [25]. The model developed by Bontemps *et al.* was able to detect Neptune DoS attack with 100% detection rate and 63 false alarms in DARPA 1999 dataset [6]. The MSeq2MSeq model along with the definition of sequence was able to detect Port Scan probe and Neptune DoS attack with 100% detection rate and only a single false alarm was produced in DARPA 1999 dataset [25]. The proposed IDS was tested with four signature-based rules and an anomaly-based rule against CICIDS 2017 dataset [32]. It was able to detect File Transfer Protocol (FTP) Brute Force attack, HTTP Slow Header DoS attack, HULK DoS attack, SQL Injection attack, Web Brute Force attack, Cross-site scripting

attack, Ares Botnet [33] attack, and Port Scan probe with 100% detection rate.

## 1.2 Contributions

To summarize, this thesis presents the following contributions:

- A novel MSeq2MSeq model which can be used in any multidimensional sequence prediction problem like intrusion detection, weather prediction, and stock prediction is developed.

- An adaptive and functionally auto-scaling stream processor: "Wisdom" is implemented which can be used for any stream processing applications such as intrusion detection, fraud detection, and fleet management.

- Suitability of Bayesian Optimization Algorithm (BOA) and Particle Swarm Optimization (PSO) for CEP rule optimization is compared, and a hybrid algorithm using PSO and Bisection algorithm is implemented as part of the "Wisdom" Stream Processor.

- Prototype of a distributed IDS using MSeq2MSeq model for anomaly-based intrusion detection and "Wisdom" stream processor for signature-based intrusion detection is developed and tested using CICIDS 2017 dataset. The proposed IDS supports adaptive rules and functionally auto-scaling deployment with the support of "Wisdom" stream processor. *Minimum Rate Guaranteed* deployment ensures that the adaptive rules will never reduce the detection rate of initial rules with user-defined thresholds. Semantics of the SQL like "Wisdom" query used in the proposed IDS is easy to learn than the semantics of Snort [8] rules.

## 1.3 Document Structure

The remainder of this thesis is organized as follows: The mathematical formulation of mapping intrusion detection to sequence prediction problem and CEP rule optimization problem are presented in Chapter 3. A literature review is presented in Chapter 2 which describes different intrusion detection techniques and early research activities on building dynamic stream processors. In Chapter 4, the methodology of the proposed method is described in detail. It contains detailed description and architecture of the MSeq2MSeq model, "Wisdom" stream processor and the proposed IDS. Tests conducted to evaluate the MSeq2MSeq model, the IDS, and functionally auto-scaling deployment are elaborated in Chapter 5. Obtained results in all test cases are presented and discussed in Chapter 6. Finally, the conclusion of the research and future work are discussed in Chapter 7.

# Chapter 2

# Background

## 2.1 Intrusion Detection

Maheshkumar and Gursel used nine different pattern recognition, and machine learning algorithms to detect attacks in the KDD CUP 1999 dataset [4]. The authors proposed a multi-classifier model using Multilayer Perceptron for probe attacks, K-means algorithm for DoS and User-to-Root (U2R) attacks, and Gaussian classifier for Remote-to-Local (R2L) attacks based on the results they obtained. The proposed multi-classifier was able to detect probe attacks, DoS attacks, U2R attacks, and R2L attacks with 0.887, 0.973, 0.298, and 0.096 detection rates respectively. Since KDD CUP 1999 is a pre-processed and flattened dataset, this work does not take advantage of the sequential relationship of packets.

Bontemps *et al.* used LSTM RNN based model to detect Neptune DoS attack with 100% true positives and 63 false positives in the time series version of KDD CUP 1999 dataset using collective anomalies in a network [6]. 63 false alarms in a five day period are not acceptable for an IDS. In their research, the model was trained using attack-free traffic to predict a sequence of packets using the first three packets. The average prediction error was used to classify anomalies. Bontemps *et al.* defined all packets arrived within a fixed time window as a sequence. However, within a time window, there can be packets from multiple connections

and multiple clients which are independent of each other. Furthermore, packets arrived within a time window is depending on external factors like peak business hours which do not have a constant pattern. Therefore their proposed solution may not give the claimed accuracy with real datasets.

Lobato *et al.* used stream processors with machine learning algorithm to detect DoS and probe attacks in real-time [34]. They used a stream processor to receive logs from applications and packets from Bro Network Security Monitor [35]. The authors used Decision Tree, Artificial Neural Network (ANN), and Support Vector Machine (SVM) algorithms for unsupervised classification and obtained above 90% accuracy with ANN and SVM algorithms. Though they used stream processor, it was not used for signature-based detection. Furthermore, the ANN used in this research is just a linear regression model which does not take advantage of the sequential order of packets. Therefore it has no significant difference from Maheshkumar *et al.*'s work [4].

Massimo and Luigi proposed a signature-based IDS using a stream processor [9]. In this work, authors assumed that an attack requires some mandatory pre-steps to complete and developed CEP rules to detect those patterns. For example, they assumed that an SQL Injection attack always starts with a Port Scan followed by Directory Traversal and Buffer Overflow within a predefined time frame. However, the authors agree that it is hard to define an interval which covers all these steps because an attacker may execute each of them at different times depending on his level of patience. Furthermore, a hacker can attack a known public endpoint using SQL Injection without scanning open ports. Therefore, this methodology is not suitable for a real IDS.

## 2.2 Sequence to Sequence Encoder Decoder Model

Intrusion detection based on anomalies in a sequence of packets requires a machine learning model to predict sequences of packets. Similar sequence prediction problem has been already

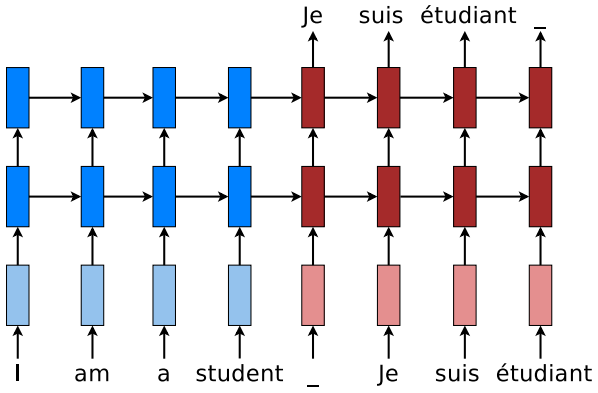



Figure 2.1: **Sequence to Sequence Encoder Decoder Model** – a stacking recurrent architecture for translating a source sequence "I am a student" into a target sequence "Je suis étudiant". Here, _ marks the end of a sequence. [45]

addressed in Neural Machine Translation (NMT) in which source and translated sentences are treated as sequences of words. NMT researchers have moved from LSTM RNN to Seq2Seq Encoder-Decoder model since the work of Sutskever *et al.* [24]. The model was later improved by other researchers and became the state-of-the-art solution in NMT [24, 36, 37, 38, 39, 40]. For example, the sentence "I am a student" in English can be translated into "Je suis étudiant" in French using Seq2Seq model as shown in Figure 2.1. The Seq2Seq model is also used in object recognition in video [41, 42] and anomaly detection in series of events [43]. Malhotra *et al.* used stacked LSTM RNN and Seq2Seq model to detect anomalous sensor data [43, 44]. In their research, Seq2Seq model gave better results for unpredictable datasets, whereas stacked LSTM RNN gave better results for predictable datasets [43].

The sequence in NMT is a sentence formulated by words in a given order. Similarly, the sequence in video analysis is a grid of pixels aligned sequentially. In both cases, elements of a sequence are one-dimensional items such as words or pixels. Datasets used by Malhotra *et al.* also contain sequences of single dimensional sensor readings. However, in packet prediction, both input and output are sequences of multidimensional packets. Selected attributes of a packet may or may not have interdependencies. In such a multidimensional Seq2Seq problem, the model must learn those interdependencies and the contribution of each attribute in predicting the following element.

In NMT, some words are more important than others. Attention layers were introduced to the Seq2Seq model to focus on the more relevant word in source sentence [37, 38, 39]. In a video frame, some pixels may be more important than others. Such important pixels are extracted using convolutional encoders [41, 42]. Sutskever *et al.* reversed the encoder input to introduce short-term dependencies and obtained a better result in English to French NMT because it reduces the average distance between target words [24]. Luong *et al.* also obtained better results by reversing the input sentence in English to German NMT [38]. Bahdanau *et al.* used Bidirectional Recurrent Neural Network (BiRNN) instead of LSTM in Seq2Seq model which can read input sequence in both directions [37].

## 2.3 Adaptive Stream Processor

Machine learning algorithms for anomaly detection and publicly available stream processors do not address the problem of "diversity of network traffic" as discussed in Chapter 1. Even though several attempts were made to build adaptive stream processors, all of them were limited to certain domains and datasets due to the unrealistic assumptions made by researchers. This section briefly analyzes such attempts and their applicability to intrusion detection. Mousheimish *et al.* proposed automatic predictive CEP rule mining from classified multivariate time series dataset [19]. Their learning algorithm first searches for subsequences across a time series input. The length of possible subsequences is limited by user-defined lower and upper bounds. Later, it prunes redundant parts of subsequences and builds a CEP rule using subsequence with the highest accuracy. This approach is limited by user-defined sequence lengths and limited CEP rule templates which are not guaranteed to fit all use cases.

Margara *et al.* developed iCEP which can generate CEP rules using time window, selector, logical operator, pattern, and aggregator [20]. iCEP learns interesting events and time frame followed by aggregators and filters, and finally parameters and sequences in an independent three-phase pipeline. In this approach, errors made in early stages of the pipeline can prop-

agate and affect the following learners. For example, if "Time Window Learner" does not capture all the necessary events, "Sequence Learner" cannot learn a sequence at the end of the pipeline. Isolated learning phases of iCEP fail to address the correlation between CEP operators. Therefore, a rule generated by iCEP may not perform well on a highly correlated domain.

Lee *et al.* proposed CEP rule mining based on similarity match [21]. In this work, authors clustered event sequences, extracted a complex event based on similarity across sequences from the same cluster and finally generated a complex event pattern using Markov Transition model. The proposed clustering algorithm calculates the distance between two sequences based on the cosine similarity between individual events and their position in sequence. The attribute to calculate cosine similarity is determined by domain experts.

Mehdiyev *et al.* used Elitist Pareto-based Multi-Objective Evolutionary Algorithm to select event attributes and Fuzzy Unordered Rule Induction Algorithm to classify events [22]. Authors compared the proposed algorithm with other classification algorithms but did not propose a way to generate CEP rules using the proposed algorithm. They also admit that generating CEP rules from the output of their classifier is a difficult challenge to address.

All above CEP rule mining methodologies were developed with an intention to replace domain experts with machines. However, they rely on false assumptions like raw events not being complex, TimeWindow being enough to collect events in all scenarios, or a single CEP rule template being able to represent all complex events. These assumptions oversimplify the problem and do not capture the real world requirements. Furthermore, these solutions mainly focus on generating rules for frequently occurring patterns. In anomaly driven domains like intrusion detection, such frequently occurring patterns represent legitimate traffic in training data. Hence rules developed using frequently occurring patterns may not work well for detecting anomalous traffic.

Turchin *et al.* defined CEP rules based on probability score of selected attributes and tuned threshold values using Discrete Kalman Filter based on expert feedback and event history [18].

The concept of tuning rule parameters and the application of adaptive rules to detect attacks in DARPA 1999 dataset are close to this research. Therefore, Turchin *et al.*'s results have been chosen as a benchmark to compare the results obtained in this research. However, their contribution to CEP rule optimization may not be widely applicable because their rules neither use any CEP operators like *Filter*[1] or *Window*[2] nor follow CEP semantics. Therefore, their solution does not address any problems we raised in Chapter 3.2 such as correlated parameters or discontinuous function. Instead, they calculate anomalous probability score of request length, response length, possible "SYN" error, and hostname for each packet. A packet is classified as an anomaly by comparing the total score of these four attributes with two threshold values. Using anomalous probability score of the hostname will give high accuracy in a simulated dataset but not in an actual deployment. For example, in CICIDS 2017 dataset, Firewall translates the hostnames of most of the attackers to the same IP address. Similarly, in DARPA 1999 dataset, same hosts were reused multiple times to simulate different attacks. In both these datasets, the anomalous probability score of attackers will be high and yield a high accuracy. In a real attack, there is a high chance of an attacker to spoof the IP address so that the anomalous probability score of that unknown IP will be close to zero which will reduce the accuracy of the system.

BOA [46] is widely being used by researchers for hyperparameter optimization and black-box optimization [47]. In this method, an unknown objective function is mapped into a prior belief and sequentially refined by a Bayesian posterior update [47]. Snoek *et al.* used BOA to tune machine learning hyperparameters [48]. It is also used by Pooyan *et al.* to optimize the performance of Apache Storm stream processor [49]. Among the population-based optimization algorithms, Genetic Algorithm (GA) and PSO [50] are widely used for hyperparameter tuning [51, 52]. GA and PSO optimized a selected set of problems with equal accuracy in a test conducted by Hassan *et al.* [53]. Though GA has been successfully applied for optimization problems, it is inefficient for applications with highly correlated parameters [52]. In

[1] A CEP operator used to filter events based on a given condition.

[2] A CEP operator used to subset events based on some conditions like a time frame or number of events per window.

addition, GA is much more complex to implement than PSO. Therefore, both BOA and PSO are evaluated for suitability in this research due to their simplicity and popularity in similar optimization problems.

# Chapter 3

# Problem Formulation

## 3.1 Map Intrusion Detection to Sequence Prediction

Seq2Seq model is widely being used in NMT [24, 36, 37, 38, 39, 40]. In general, a Seq2Seq problem can be defined as below:

$$X = (x_1, x_2, x_3, ....x_n)$$

$$Y = (y_1, y_2, y_3, ...y_m)$$

$$Y' = (\sigma_s, y'_1, y'_2, y'_3, ...y'_m, \sigma_e)$$

$$Y'_{i+1} = f(X, \bigcup_{t=0}^{i} Y'_t)$$

where $X$ is the input sequence which is the source sentence in NMT, and $Y$ is the expected output sequence which is the ideal translated sentence in NMT. $\sigma_s$ is a flag indicating *Start of Sequence* known as $<SOS>$ in NMT and $\sigma_e$ is a flag indicating *End of Sequence* known as $<EOS>$ in NMT. $Y'$ is the input to the decoder which starts with $\sigma_s$ and ends with $\sigma_e$. Naively, $y'_i$ is the predicted item for the decoder input $y'_{i-1}$. In teacher forcing method [54], $y_{i-1}$ is used as the decoder input to predict $y'_i$. $f$ is the Seq2Seq encoder-decoder model which generates

the $i + 1^{th}$ item of output sequence using the input sequence $X$ together with the first $i$ elements of decoder input $Y'$.

The intrusion detection problem can be mapped to the Seq2Seq prediction problem by defining a sequence of packets $P$ as given below in which $p_i$ is the $i^{th}$ packet in the sequence. The first $k$ packets are used as input sequence $X$ and the remaining packets are treated as output sequence $Y$. Sequences with a prediction accuracy $\alpha$ less than a threshold $T$, are classified as an anomaly.

$$P = (p_1, p_2, p_3, ...p_n)$$

$$X = (p_1, p_2, p_3, ...p_k)$$

$$Y = (p_{k+1}, p_{k+2}, ...p_n)$$

$$Y' = (\sigma_s, p'_{k+1}, p'_{k+2}, ...p'_n, \sigma_e)$$

$$\alpha = 1 - \sum_{i=k+1}^{n} \delta(p_i, p'_i)$$

Though the intrusion detection problem can be mapped to sequence prediction problem, the traditional Seq2Seq algorithm cannot be directly applied for packet prediction. In NMT, Seq2Seq model receives and generates sentences which are sequences of one-dimensional items: words. A word can be encoded into a one-hot vector [55] with a size equal to the number of words in the vocabulary. In intrusion detection problem, the model must receive and generate a sequence of packets where a packet is a multidimensional item. Flattening all selected attributes into a single one-hot vector requires a large vector and does not let the model learn the contribution of each attributes in predicting the next packet. Therefore, a new mapping technique or a model to predict multidimensional sequence is required to apply Seq2Seq model for intrusion detection.

Figure 3.1: **HTTP Slow Header DoS Detector** – a Wisdom query to detect HTTP Slow Header DoS attack based on the fact that the attacker opens several incomplete connections simultaneously to keep the server busy.

```
def stream PacketStream;
def stream AttackStream;

@config(trainable=true, minimum=100, maximum=60000, step=-1)
def variable time_threshold = 101;

@config(trainable=true, minimum=3, maximum=1000, step=1)
def variable count_threshold = 998;

from PacketStream
    filter 'http' == app_protocol and
      destPort == 80 and '\r\n\r\n' in data
        and 'Keep-Alive: \\d+' matches data
    partition by destIp
    window.externalTimeBatch('timestamp', $time_threshold)
    aggregate count() as no_of_packets
    filter no_of_packets >= $count_threshold
    select srcIp, destIp, timestamp
insert into AttackStream;
```

## 3.2 Stream Processor Query Optimization

Logical stream processor queries are developed by domain experts using their experience and domain knowledge. Several recent studies have proposed automatic CEP rule generation using unsupervised machine learning algorithms to replace domain experts by machines [19, 20, 21, 22]. Machine learning algorithms require a lot of preprocessed data and training time. Instead, the traditional way of defining CEP rules based on human cognition and domain-specific facts is easier than mining rules from training data. For example, HTTP Slow Header DoS attacks open several incomplete connections to keep an HTTP server busy for a long time [56]. A "Wisdom" CEP rule was developed based on the above definition to detect HTTP Slow Header DoS attack as given in Figure 3.1. Though it is easy to define the filter condition (*filter* keyword with boolean conditions in Figure 3.1) based on attack signature, determining threshold values (*time_threshold* and *count_threshold* in Figure 3.1) requires manual inspection of

training data. Humans are experts in logical reasoning using their experience and cognition but poor in handling a large volume of numbers. Especially in intrusion detection, it is a tedious task for a human to analyze raw pcap files and to determine those threshold values.

Threshold values in a CEP rule can be an integer, a real number or a constant. Considering all possible constants as a list of candidates, they can be treated as integer values. These numbers may or may not have lower and upper bounds. For example, the minimum *count_threshold* in Figure 3.1 has a lower bound 0 but not an upper bound because the number of packets in an interval can be an arbitrarily large positive number but cannot be a negative number. However, these parameters are correlated to each other in such way that they cannot take all possible values in the space. According to these facts, a CEP rule optimization problem can be defined as

$$\begin{aligned} max/min \quad & f(x_1, x_2, x_3, ..., x_n) \\ s.t \quad & AX \leq B \\ & x \epsilon \mathbb{R} \end{aligned}$$

where $A$ is a rational matrix, and $B$ is a rational vector. CEP rule is a discontinuous function which takes streams of events as input and optionally generates complex events as output. Therefore, it is hard to fit a CEP rule itself in an optimization problem. Instead, $f$ is a continuous profit or loss function defined using the output of a CEP rule in such a way that optimizing $f$ will optimize the CEP rule. This way, optimizing a CEP rule can be defined as a Mixed Integer Linear Programming (MILP) problem if $f$ is linear or Mixed Integer Non-Linear Programming (MINLP) problem if $f$ is non-linear. Both MILP and MINLP are NP-Hard problems and as such, finding a solution in polynomial time is not feasible in a worst-case scenario [57, 58].

# Chapter 4

# Methodology

The proposed IDS has two main components: (1) a machine learning model for anomaly-based intrusion detection and (2) an adaptive stream processor for signature-based intrusion detection. This section covers both these components and how the problems discussed in Chapter 3 have been addressed in this research. In addition, the architecture of proposed IDS is explained in this chapter.

## 4.1 Dataset Selection and Preprocessing

Available intrusion detection datasets such as DARPA 1999 [59], KDD CUP 1999 [60], UNSW-NB15 [61] and CICIDS 2017 come with raw packet capture files and pre-processed data in text or CSV format. Pre-processed datasets in human readable format do not capture the sequential relationship of packets. For example, the KDD CUP 1999 dataset has 42 attributes but among them, the *duration* of a connection and the *number of connections* in the last two seconds are the only two attributes related to the temporal relationship of packets. Still, KDD CUP 1999 dataset does not provide any hint on packets which are part of each connection and how those packets are aligned in the connection. Therefore, raw network packets[1] were used to train

[1]The application developed to process raw packet capture files is available at `https://github.com/slgobinath/pcap-processor`

the MSeq2MSeq machine learning model. In Phase 1 evaluation, the model was trained and tested using raw packets from DARPA 1999 dataset. However, the simulated traffic in DARPA 1999 dataset is highly unrealistic. For example, manual inspection of raw packets revealed that some port scans were generated by iterating from port 1 to 1000 five times repetitively. In addition, DARPA 1999 dataset is more than fifteen years old. On the other hand, CICIDS 2017 dataset is relatively newer than other datasets and covers the most common attacks based on 2016 McAfee report [32]. Therefore, raw packet capture files from CICIDS 2017 dataset were chosen to train and test the IDS due to the availability of recent attacks in raw packet capture format.

In the Phase 1 evaluation using DARPA 1999 dataset, a sequence of packets $P$ was defined as a TCP connection. However, it cannot be used with other connectionless protocols like User Datagram Protocol (UDP). Therefore, in later evaluations, streams of packets transferred between the same client and server were split into sequences $P_i$ as given below:

$$
\begin{aligned}
& P_i = (p_{i_1}, p_{i_2}, p_{i_3}, ...p_{i_n}) \\
& s.t \quad TIME(p_{(i+1)_1}) - TIME(p_{i_n}) \geq 1sec \\
& and \quad TIME(p_{i_{t+1}}) - TIME(p_{i_t}) < 1sec \\
& and \quad IP(p_{i_{t+1}}) \equiv IP(p_{i_t})
\end{aligned}
$$

where $p_{i_k}$ is the $k^{th}$ packet in the $i^{th}$ sequence, $TIME$ is the timestamp of the packet, and $IP$ is the hostname of the source or destination. Here an assumption was made that most of the time, network traffic is generated directly or indirectly by humans. Therefore, a delay is expected between two independent traffic generated by a person. For example, if a user clicks on a link, there will be a delay before another click. In such a case, packets transferred for the first click and the second click are treated as two separate sequences. In this research, a $1sec$ delay was considered which was determined by trial and error using CICIDS 2017 dataset. Suppose a user

watches more than a video from the same server, all the packets transmitted between the client and server may be considered as a sequence because of the lack of significant delay between those packets. No actions were taken to separate them because it is a legitimate scenario and the model should be able to learn such patterns.

Splitting DARPA 1999 dataset based on the above definition led to sequences with packets from several independent connections. It may be due to the way DARPA 1999 dataset was created using continuously repeating scripts without concerning real human behavior. A model trained using such a dataset will fail to differentiate the traffic generated by bots from legitimate traffic. Compared to DARPA 1999, CICIDS 2017 is better in representing human behavior. However, CICIDS 2017 dataset also has some pitfalls. For example, the creators of the dataset claim that there are 21 SQL Injection attacks in the dataset; but only 15 packets related to SQL Injection attack were found in the packet capture file. CICIDS 2017 dataset has only one long-running simulation of every selected attack. Since the proposed IDS processes raw packets, the labeled dataset cannot be used to compare the number of detections. Instead, the timestamp and IP addresses of anomalous packets detected by the IDS were manually compared with raw packets and labeled dataset to confirm the accuracy.

| Bucket | Protocol | No of packets | Encoder Input | Decoder Input |
|---|---|---|---|---|
| 1 | TCP | 4-10 | 3 | 7 |
| 2 | TCP | 11-20 | 7 | 13 |
| 3 | TCP | 21-40 | 13 | 27 |
| 4 | TCP | 41-80 | 27 | 53 |
| 5 | TCP | 81-160 | 53 | 107 |
| 6 | TCP | 161-320 | 107 | 213 |
| 7 | UDP | 4-10 | 3 | 7 |
| 8 | UDP | 11-20 | 7 | 13 |
| 9 | UDP | 21-40 | 13 | 27 |

Table 4.1: **Buckets of Sequences** – sequences grouped based on transport protocol and the maximum number of packets per sequence. Sequences with less than 3 packets were ignored and sequences with more than 320 packets were pruned to 320.

Packets from the first day of CICIDS 2017 dataset which contain attack free traffic were

split into sequences. In an initial experiment with CICIDS 2017 dataset, a single model was unable to learn all sequences with enough accuracy. Therefore, the extracted sequences were grouped into buckets based on transport protocol and the maximum number of packets as shown in Table 4.1 and dedicated MSeq2MSeq models were developed for each bucket. Using different models per transport protocol lets each model learn a single pattern. Internet Control Message Protocol (ICMP) packets were ignored because the dataset does not have enough number of ICMP sequences to train the machine learning model. Sequences with less than the maximum number of packets of a bucket were padded with $max\_bucket\_size - sequence\_length$ number of $EMPTY_PACKET$s to meet the expected length of the input vector. Here, $EMPTY\_PACKET$ is a vector representing a packet with default values assigned to each selected attributes. Sequences with less than four packets were ignored because the encoder required at least three to predict with enough accuracy in our preliminary test. Sequences having more than 320 TCP packets were pruned to 320, and more than 160 UDP packets were pruned to 160 because there were no enough TCP sequences with more than 320 packets and UDP sequences with more than 160 packets to train a model with enough accuracy. From each packet, following attributes were selected to be used with the proposed machine learning model: $transport\_layer$, $src\_ip$, $dst\_ip$, $ip\_flag$, $transport\_flag$. Increasing the number of selected attributes reduces the accuracy of the model. Above attributes were selected by trial and error to obtain better accuracy.

## 4.2 Sequence to Sequence Model

Seq2Seq model has two RNN's named encoder and decoder. The goal of the encoder is converting an input sequence $X = (x_1, \ldots, x_n)$, into a vector $c$ and the goal of the decoder is converting $c$ into an output sequence $Y = (y_1, \ldots, y_n)$.

At each time step $t$, hidden state of an RNN $h_t$ is computed by (4.1) where $f$ is a non-linear activation function and $x_t$ is the input at time $t$.

$$h_t = f(x_t, h_{t-1}) \tag{4.1}$$

Usually the non-linear activation function $f$ is an LSTM and $c$ is the final hidden state $h_n$ of that LSTM. Decoder is another LSTM which generates $Y = (y_t, ..., y_i)$ using $Y = (y_1, ..., y_{t-1})$ as the input and $c$ as the initial internal state. Hidden state of the decoder is computed by:

$$h_t = f(h_{t-1}, Y_{t-1}, c) \tag{4.2}$$

## 4.3 Multidimensional Sequence to Multidimensional Sequence Model

Though Seq2Seq model is widely being used in NMT, it cannot receive or generate sequences with multidimensional items as discussed in Chapter 3.1. Therefore the traditional Seq2Seq model is improved in this research to receive and generate sequences with multidimensional items as depicted in Figure 4.1. This figure depicts only a single layer of sigmoid functions. However, there may be more than one layer depending on the problem. The proposed MSeq2MSeq model has $k$ number of input and output branches where $k$ is the number of selected attributes in a packet. Depending on the problem, the number of branches of the encoder and decoder can vary. This way, MSeq2MSeq model learns the contribution of every single attribute in predicting the next packet and adjusts the weight in input and output branches. Except for the first few layers and last few layers, the model reflects the exact traditional Seq2Seq model. The attention layer proposed by Luong *et al.* [38] is used to obtain better results as discussed later in this chapter.

Sutskever *et al.* and Luong *et al.* obtained better results by reversing the encoder input in NMT. The impact of reversing input sequence in packet prediction was tested by comparing the training loss of reversed and non-reversed encoder input. Though there was no significant dif-

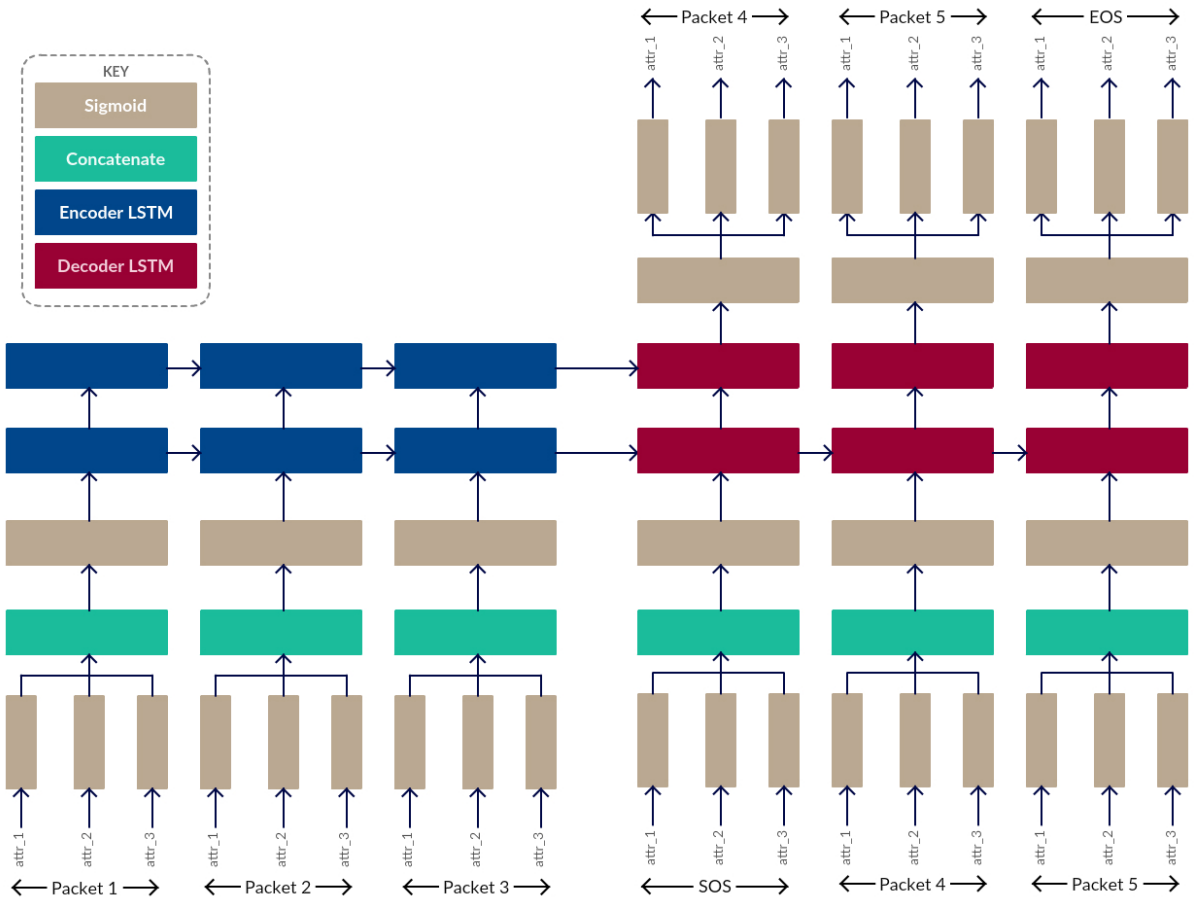
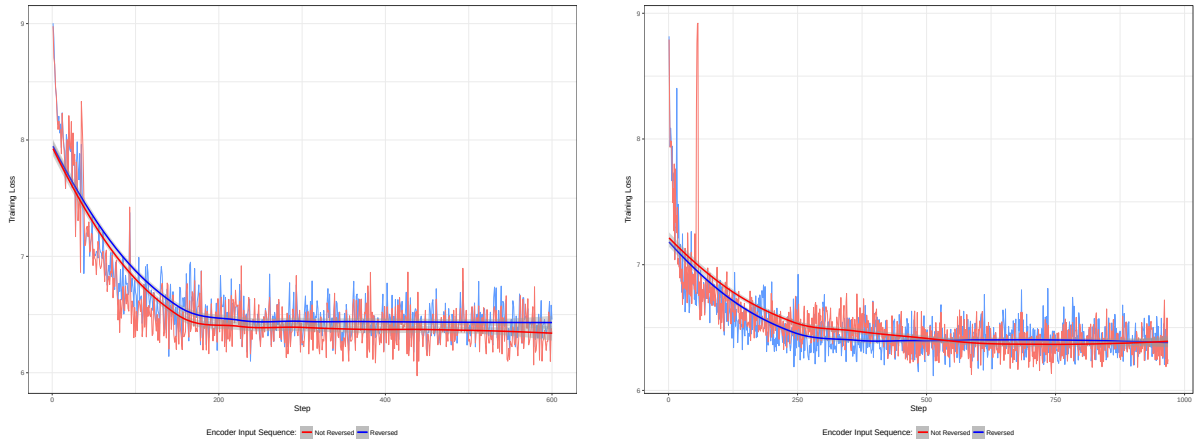



Figure 4.1: **Multidimensional Sequence to Multidimensional Sequence Model** - an enhanced Seq2Seq model to receive and generate sequences of multidimensional items. This model learns the individual contribution of each attribute of elements in predicting the sequence.

ference in prediction accuracy and training loss, the model trained with direct input sequences learned slightly better than the model trained with reversed input sequence as shown in Figure 4.2. As shown in Table 4.1, the encoder input sequence $X$ is always guaranteed to be complete. Furthermore, in a sequence of packets, the $i + 1^{th}$ packet is highly depending on $i^{th}$ packet than the first packet. Therefore, reversing the encoder input is not necessary in packet prediction problem.

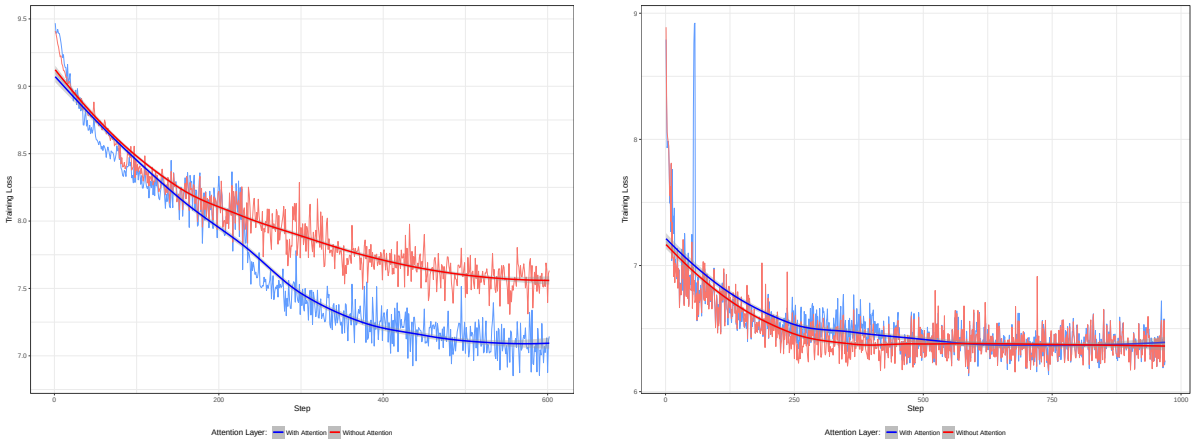
The impact of attention layer in the proposed MSeq2MSeq model for packet prediction was tested using the attention layer proposed by Luong *et al.* Two MSeq2MSeq models were developed with and without the attention layer proposed by Luong *et al.* and trained using




(a) (b)

Figure 4.2: **Impact of Reversing Encoder Input** – comparison of the impact of reversing encoder input in (a) Bucket 1 (b) and Bucket 3 from Table 4.1. The model trained with not-reversed sequences with a maximum of 10 TCP packets learned slightly faster than the model trained with reversed sequences with a maximum of 10 TCP packets. As the number of packets in a sequence increases, the difference becomes less transparent.

sequences from "Bucket 1" and "Bucket 3" respectively. As shown in Figure 4.3, the model with attention layer learned faster than the model without attention layer while training sequences from "Bucket 1". However, there is no significant difference in training the model with sequences from "Bucket 3". Sequences in "Bucket 1" has a maximum of ten packets and sequences in "Bucket 3" has a maximum of 40 packets. Packets from a small network connection like TCP three-way handshake heavily rely on preceding packets compared to packets in a long network connection like video streaming. As discussed earlier, the definition of sequence allows a sequence to have packets from more than one connection. As the number of packets in a sequence increases, there is a high chance of having packets from different network connections. Therefore, as the number of packets increases, the importance of reversing the encoder input or having an attention layer becomes less.




(a) (b)

Figure 4.3: **Impact of Attention Layer** – comparison of the impact of attention layer in (a) Bucket 1 (b) and Bucket 3 from Table 4.1. The model with attention-layer learned significantly faster than the model without attention layer when training them using sequences with a maximum of 10 TCP packets. As the number of packets in a sequence increases, the difference becomes less transparent.

## 4.4 Wisdom Stream Processor

Even though dynamic stream processors have been proposed in early works, commercial stream processors do not support dynamic query modification at the runtime. Analysing the open source stream processors: Apache Flink and WSO2 Siddhi revealed that their underlying data structures that are being used to store events in memory do not support dynamic query modification. On the other hand, authors of iCEP dynamic complex event processor claim that their complex event processor can analyze "thousands of events in a few minutes" [20] which is much less than the throughput of commercial stream processors which typically can handle several million events per second. Therefore, an adaptable and functionally auto-scaling stream processor: "Wisdom"[2] was developed without compromising the performance [27]. The underlying architecture of "Wisdom" using *Observer* design pattern and *Mediator* design pattern [62] to implement variables and dynamic CEP operators yields performance comparable to commercial stream processors as shown in Table 4.2 and significantly better performance than

[2]The Wisdom Stream Processor is available at `https://slgobinath.github.io/wisdom`

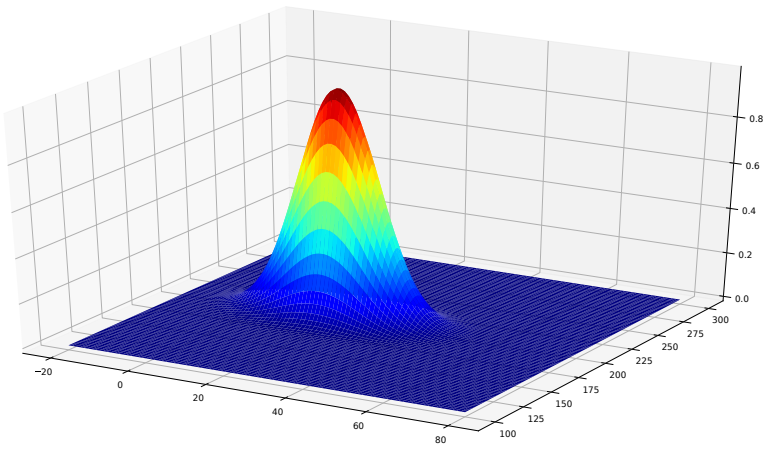



Figure 4.4: **Made-up Profit Function** – representing an imaginary CEP rule with two threshold values to test Bayesian Optimization and Particle Swarm Optimization algorithms. The imaginary CEP rule produces more accurate events when those threshold values are close to 20 and 200.

iCEP.

| Stream Processor | Throughput | Latency |
|---|---|---|
| Apache Flink | 6,711,544 events/sec | 100 nanoseconds |
| WSO2 Siddhi | 3,811,876 events/sec | 216 nanoseconds |
| Wisdom | 2,543,299 events/sec | 332 nanoseconds |
| Esper | 2,247,807 events/sec | 334 nanoseconds |

Table 4.2: **Performance Comparison of Stream Processors** – comparing "Wisdom" with commercial stream processors using a filter query in single thread setup.

The profit or loss function $f$ defined in Chapter 3.2 is a black box of correlated variables because its output depends on the underlying CEP rule. CEP rules looking for anomalies emit output only for a limited set of threshold values. For any other values, they emit nothing. A simple profit function was developed as shown in Figure 4.4 to simulate the behavior of an imaginary CEP rule which generates output only if its threshold values are closed to 20 and 200. Both PSO and BOA were used to optimize this function. The accuracy and execution time were recorded for each optimization algorithms. As shown in Table 4.3, PSO outperforms BOA in both accuracy and performance. PSO starts with random initial points and quickly converges to the optimum once a particle finds an improvement in the profit. Though BOA had

Figure 4.5: **CEP Rule Optimization Algorithm** – a hybrid optimization algorithm developed using PSO and Bisection algorithms. PSO algorithm is used to find an initial optimum point and Bisection algorithm is used to push the optimum point towards a desired boundary.

**Input:** $function$, $constraints$, $steps$
**Output:** $optimal\_values$

1: $optimal\_values, loss \leftarrow PSO(function, constraints)$
2: **for all** $val \in optimal\_values$ **do**
3: $val \leftarrow Bisection(function, val, constraints[val], step[val])$
4: **end for**
5: **return** $optimal\_values$

some initial points close to the optimum, it was distracted by the plateau where profit is 0 and spent more time in building the prior model. Therefore, the PSO algorithm has been chosen to implement the actual CEP parameter tuning algorithm.

| | **Bayesian** | **Particle Swarm** |
|---|---|---|
| Initial points/Swarm size | 100 | 100 |
| Maximum iterations | 10 | 10 |
| Avg. execution time (seconds) | 255.788 | **0.029** |
| Avg. optimal points (x, y) | 8.742, 409.921 | **20.681, 199.919** |
| Avg. Profit | 0 | **0.998** |

Table 4.3: **Bayesian Optimization vs Particle Swarm Optimization** – comparison of Bayesian and Particle Swarm optimization algorithms using the profit function shown in Figure 4.4. The maximum obtainable profit is 1 for the optimum point (20, 200).

A real CEP rule can have more than one optimum points adjacent to each other. For example, the above imaginary CEP rule may produce the same output for threshold values in between 20 – 25 and 200 – 250. Depending on the requirement we may be interested in either the upper bound or the lower bound of a threshold. For example, we prefer to have minimum time threshold and maximum count threshold for the CEP rule given in Figure 3.1 to reduce latency and false positives. Therefore, a hybrid optimization algorithm was implemented using PSO and Bisection algorithms as shown in Figure 4.5. PSO is used to find the initial optimum values, and Bisection algorithm is used to push them towards desired boundaries. In traditional Bisection algorithm, the convergence speed reduces with every iteration as the step size de-

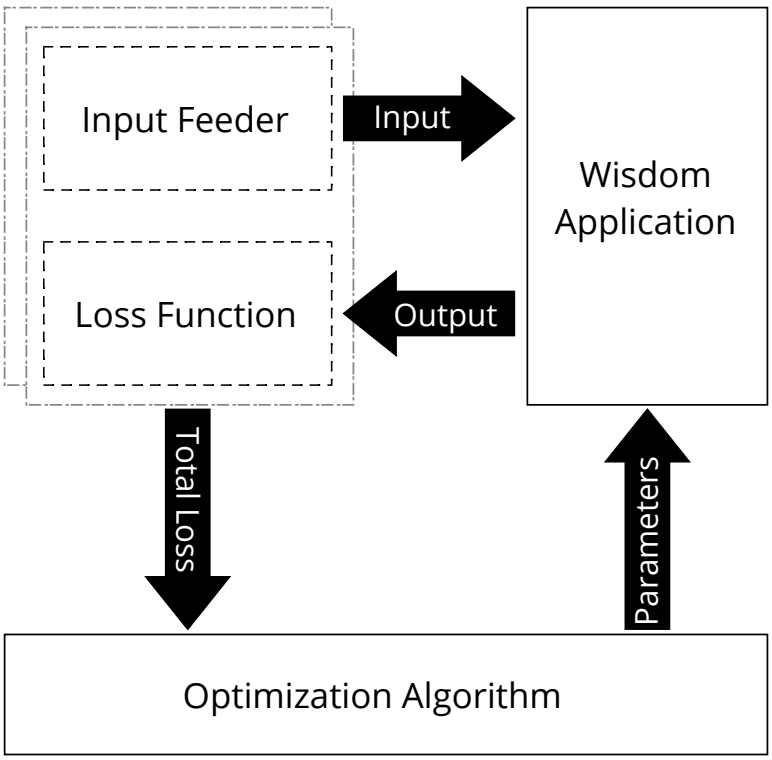



Figure 4.6: **Wisdom Optimizer Architecture** – a framework developed to optimize Wisdom rules. The Input Feeder and Loss Function must be defined by the user depending on the requirement and data format.

creases. To overcome this problem, the user-defined *step* value ($-1$ and 1 in Figure 3.1) is used as the step size if the actual step size is smaller than the user-defined step value. Sign of the user-defined *step* value indicates the desired direction to move the optimum threshold values.

The proposed optimization algorithm requires a domain expert to limit the range of threshold variables to find a solution in polynomial time. The *minimum* value, *maximum* value, and *step* size are tailored to "Wisdom" query using $@config(trainable = true, ...)$ annotation as shown in Figure 3.1. Figure 4.6 depicts the architecture of Wisdom Optimizer. In this architecture, "Input Feeder" and "Loss Function" must be defined by domain experts depending on the domain requirements where "Input Feeder" feeds input events to the optimizer and "Loss Function" converts the output of CEP rule to a real number. "Wisdom Application" is the runtime environment compiled from a Wisdom query and the "Optimization Algorithm" is the implementation of the algorithm given in Figure 4.5 with additional features to coordinate with

Figure 4.7: **Wisdom Java API** – a sample Wisdom application developed using Java API to filter TCP packets from PacketStream.

```
WisdomApp app = new WisdomApp("TCPFilter", "1.0.0")

app.defineStream("PacketStream");
app.defineStream("OutputStream");

app.defineQuery("FilterQuery")
.from("PacketStream")
.filter(event -> "TCP".equals(event.get("protocol")))
.insertInto("OutputStream");

app.addCallback("OutputStream", EventPrinter::print);
```

Figure 4.8: **Wisdom Query** – a sample Wisdom application developed using Wisdom Query Language to filter TCP packets from PacketStream.

```
@app(name='TCPFilter', version='1.0.0')

def stream PacketStream;

@sink(type='console')
def stream OutputStream;

@query(name='FilterQuery')
from PacketStream
filter symbol == 'TCP'
insert into OutputStream;
```

"Wisdom" application.

"Wisdom" provides Java [63] API and Wisdom Query Language to develop CEP applications. For example, Figure 4.7 shows a Wisdom application developed in Java to filter TCP packets from *PacketStream*. The same application can be developed using Wisdom Query Language as given in Figure 4.8. A rule defined using Wisdom Query Language can be still parsed in Java to create a Java based application. Wisdom Stream Processor package provides the necessary infrastructure to run "Wisdom" applications as a service using query files. Even though "Wisdom" can be used as a Java library, running "Wisdom" application as a service is recommended for scalability and resource allocation. The Wisdom Stream Processor package

also provides an additional tool named "Wisdom Manager" which can be used to automatically deploy and control Wisdom rules. Currently, Wisdom Manager supports the following RESTful APIs:

- Get details about a deployed Wisdom rule.
- Get details about all deployed Wisdom rules.
- Start a deployed Wisdom rule.
- Stop a running Wisdom rule.
- Stop the Wisdom Manager.

These APIs are subject to change in future work according to the requirement. Wisdom Query Language provides built-in semantics to define the priority of a query within the system and streams on which the query is depending on. A stream defined in a query can report its statistics to third parties including Wisdom Manager. Using these details, Wisdom Manager can automatically stop running Wisdom instances if streams on which they are depending on have not received any events for a long time. However, functionally auto-scaling deployment is an optional feature and only used in the IDS if system resources are limited. Distributing and scaling a stream processor at the operator level can cause to coordination problems in CEP operators depending on the order of events. Therefore, "Wisdom" stream processor is designed using microservice architecture [31] to deploy each CEP rule as a microservice with required memory and CPU allocation.

## 4.5 Integrating the Stream Processor with Machine Learner

Another framework named "Wisdom Machine Learner"[3] is developed using TensorFlow [64] to train and serve the MSeq2MSeq model developed in Chapter 4.3. The framework is configurable to pre-process, train, test or serve the MSeq2MSeq model in a pipeline. The Wisdom

[3]The Wisdom Machine Learner is available at `https://github.com/slgobinath/wisdom-ml`

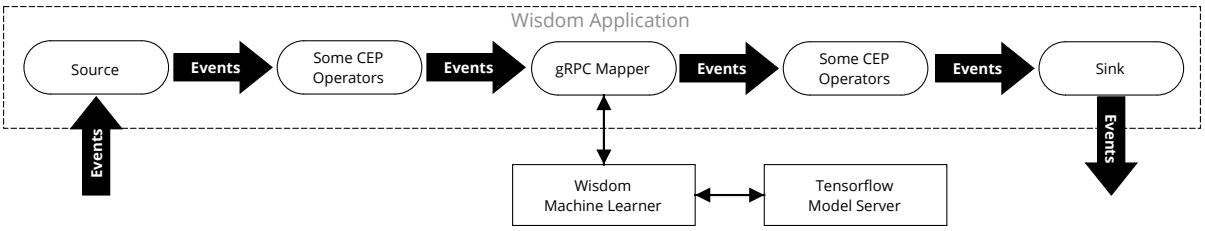



Figure 4.9: **Wisdom Stream Processor and Machine Learner** – integrating the Wisdom Machine Learner into a Wisdom application. The gRPC Mapper seamlessly sends events to the Wisdom Machine Learner and injects the result back into the stream of events.

Machine Learner exposes gRPC [28] endpoint which can be accessed from "Wisdom" stream processor through *gRPC Mapper* as given in Figure 4.9. The predictive "Wisdom" query given in Figure 4.10 splits the stream of packets transferred between same source and destination, aligns them into a sequence using $IdleLengthTimeBatchWindow$[4] and seamlessly calculates the prediction accuracy using the Wisdom Machine Learner. The Wisdom Machine Learner converts the events to a one-hot vector, calculates the prediction accuracy using TensorFlow Model Server, and returns the accuracy to the stream processor. The above query is also responsible for selecting the correct endpoint according to the maximum number of packets in the sequence. During runtime, this query is served using three independent services: (1) Wisdom Stream Processor, (2) Wisdom Machine Learner, and (3) TensorFlow Model Server. Since they are independent of each other, these instances can be scaled up and down depending on the requirement.

## 4.6 Intrusion Detection System

The proposed IDS combines both adaptive intrusion detection and functionally auto-scaling deployment together as depicted in Figure 4.11. Even though only raw network packets were used in this research, CEP rules can be written to take advantage of other sources like application logs and company policy changes. Every individual "Wisdom" service running as part of

[4] A CEP window which collects events from a stream and emits them if there is an idle period in the stream or the number of events in the window exceeds a threshold.

Figure 4.10: **Predictive Wisdom Query** – detects anomalies using Wisdom Machine Learner serving on *localhost* port *9001*. Sequences with a prediction accuracy less than 0.125 are classified as anomalies.

```
@app(name='WisdomApp', version='1.0.0', playback='timestamp')
@source(type='kafka', bootstrap='localhost:9000')
def stream PacketStream;

def stream TCPStream;

@source(type='console')
def stream AttackStream;

from PacketStream
filter transport_layer == '\ac{tcp}'
select src_ip, dst_ip, ip_flag
partition by src_ip, dst_ip
window.idleTimeLengthBatch(time.sec(1), 1000)
limit 320
aggregate collect('src_ip') as src_ip, collect('dst_ip') as dst_ip,
    collect('ip_flag') as ip_flag
map len('ip_flag') as no_of_packets
insert into TCPStream;

from TCPStream
filter no_of_packets > 20 and no_of_packets <= 40
map grpc('localhost:9001', 'accuracy') as accuracy
filter accuracy < 0.125
select src_ip, dst_ip, no_of_packets, accuracy
insert into AttackStream;
```

the IDS has the flexibility to read external sources and to produce output to external endpoints. Therefore, users can write individual rules using different sources of events without worrying about other rules already deployed in the IDS. However, a common source is recommended for inputs which are likely to be used by all "Wisdom" rules to improve the performance. For example, in the above architecture, the "Filter Query" receives all network packets and share them with other "Wisdom" rules. The deployment method of "Wisdom" rules in the proposed IDS differs from native "Wisdom" stream processor. Both anomaly-based rules and untunable signature-based rules are deployed as normal CEP rules. However, signature-based rules with tunable parameters are deployed in *Minimum Rate Guaranteed* mode. Since the IDS is developed using "Wisdom" stream processor, it takes all advantages of "Wisdom" stream processor including self-tuning rules, functionally auto-scaling instances in a distributed deployment, SQL like "Wisdom" query for rule definition and multiple input sources for advanced decision making. Following sections cover the in-depth architecture of adaptive IDS and functionally auto-scaling IDS.

### 4.6.1 Adaptive Intrusion Detection System

The intrusion detection system is built using both "Wisdom" stream processor and Wisdom Machine Learner. During deployment time, IDS reads all "Wisdom" rules and create "Wisdom" instances based on those rules. If a query has at least one trainable variable, the IDS will create three runtime instances of the query: (1) a static instance which will use user-defined thresholds forever, (2) an adaptive instance starting with user-defined thresholds but will be tuned later by the IDS and (3) a sandboxed instance to optimize the query without producing output to the user. If a query does not have trainable variables, only one CEP instance will be created. Predictive "Wisdom" rules connected with machine learner also can be tunable. However, in this research, predictive rules were not tuned to reduce complexity and resource consumption.

As shown in Figure 4.12, the "Packet Receiver" receives raw packets, converts them into

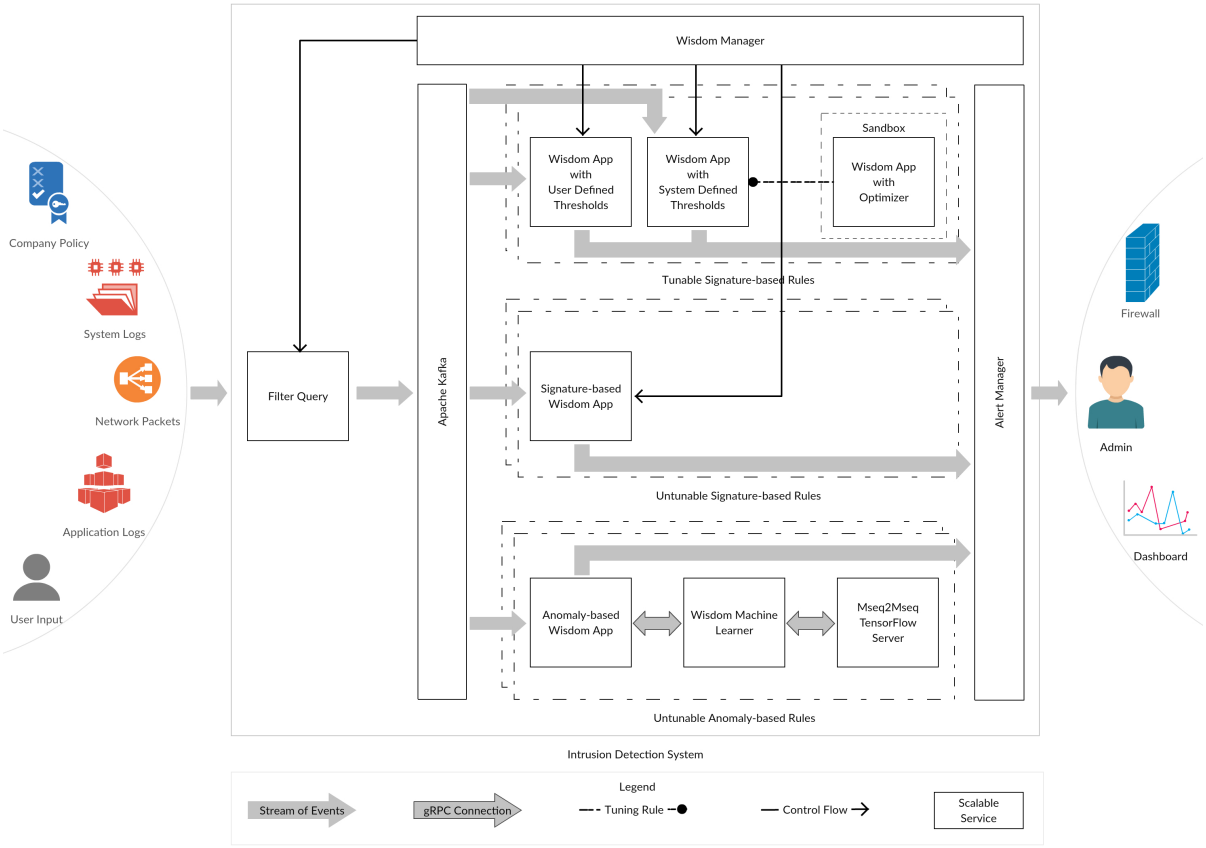



Figure 4.11: **Intrusion Detection System** – a high-level architecture of the adaptive and functionally auto-scaling IDS developed using Wisdom Stream Processor and MSeq2MSeq model.

events and feeds them to the "Packet Cache" and all deployed "Wisdom" applications. A "Wisdom" application can be a signature-based detector as in Figure 3.1 or an anomaly-based detector with the support of "Wisdom" machine learner as in Figure 4.10. "Packet Cache" keeps packets arrived in last $t$ minutes which will be later used to tune "Wisdom" rules. Alerts generated by "Wisdom" applications trigger the Wisdom Optimizer deployed in a sandbox and update the threshold variables of "Wisdom" applications. Missing an attack can cause severe damage in safety-critical domains like intrusion detection. The above *Minimum Rate Guaranteed* deployment ensures that the IDS will not miss any attacks that could be captured by the user-defined rule by tuning a clone of CEP rule and running both user-defined rule and self-tuning rule concurrently with a cost of additional system resources. Running duplicate instances of the same Wisdom rule generates duplicate alerts for the same attack. Such dupli-

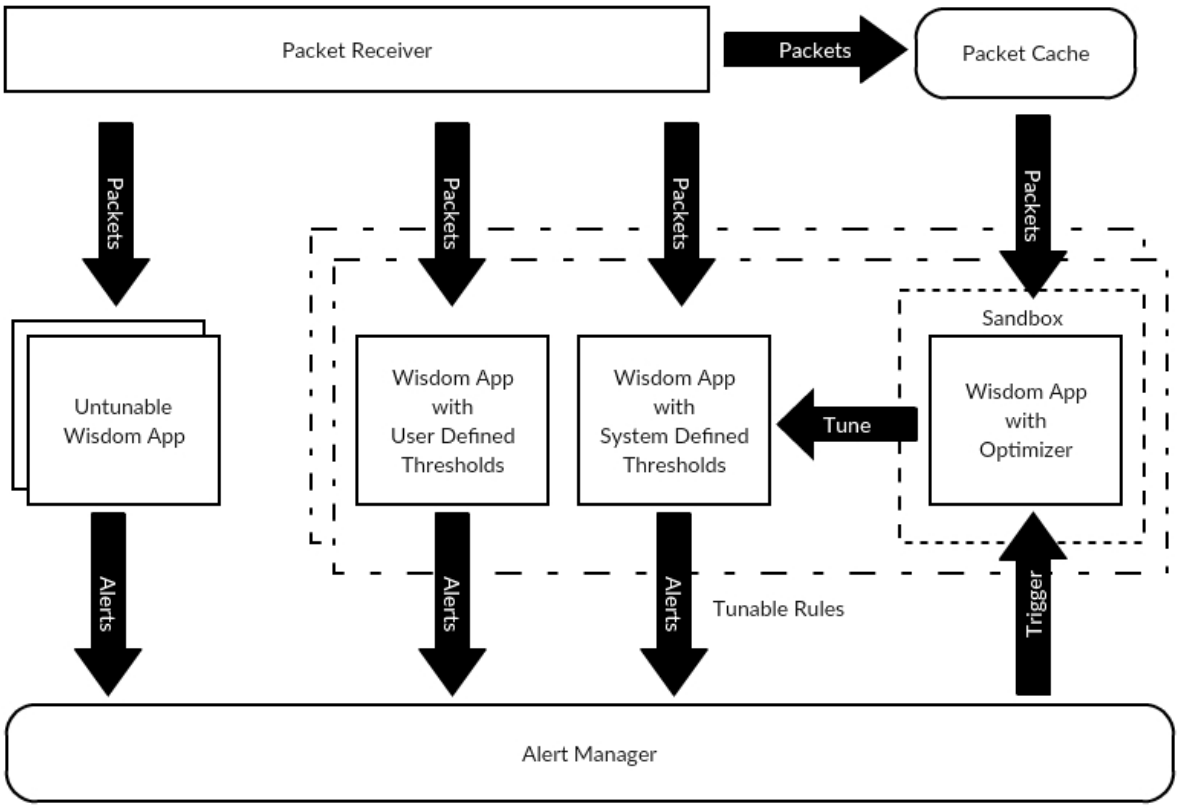



Figure 4.12: **Self-tuning Intrusion Detection System** - the Minimum Rate Guaranteed deployment of self-tuning Wisdom rules. Two additional clones of tunable rules are used to optimize the rule in a sandbox and to guarantee the minimum detection rate.

cate alerts are filtered by the "Alert Manager". It is also responsible for triggering the relevant Wisdom Optimizer to tune the rule. By combining stream processor and machine learner, the IDS can detect attacks based on both signature and anomalous traffic.

### 4.6.2 Functionally Auto-scaling Intrusion Detection System

In an IDS, some attack detectors may need more resources than others. For example, a DoS attack detector may need more system resources than an SQL attack detector due to the large amount of traffic involved in DoS attack. The native microservice architecture of "Wisdom" stream processor lets the IDS to control system resource allocation per each IDS rule. Running all intrusion detectors all the time consumes a lot of system resources. In the world of IoT, miniature computers like Raspberry Pi [65] are getting popular. In such systems, running a

Figure 4.13: **Filter Query Used in Functionally Auto-scaling Deployment** – filters network packets based on some initial conditions and to send them to relevant possible attack stream.

```
@app(name='packet_filter', version='1.0.0', priority=10, stats='
   StatisticsStream', stats_freq=time.sec(5), stats_vars=['port'])

@source(type='kafka', bootstrap='localhost:9092', topic='
   PacketStream')
def stream PacketStream;

@config(stats=true)
@sink(type='kafka', bootstrap='localhost:9092', topic='
   PossibleDosStream')
def stream PossibleDosStream;

@config(stats=true)
@sink(type='kafka', bootstrap='localhost:9092', topic='
   PossibleBruteForceStream')
def stream PossibleBruteForceStream;

@config(stats=true)
@sink(type='kafka', bootstrap='localhost:9092', topic='
   PossiblePortScanStream')
def stream PossiblePortScanStream;

@sink(type='kafka', bootstrap='localhost:9092', topic='_Statistics')
def stream StatisticsStream;

@query(name='FilterDosAttacks')
from PacketStream
filter 'http' == app_protocol and destPort == 80 and '\r\n\r\n' in
   data and 'Keep-Alive: \\d+' in data
insert into PossibleDosStream;

@query(name='FilterBruteForceAttacks')
from PacketStream
filter '\ac{ftp}[Control]' == app_protocol and '530 Login incorrect'
    in data
insert into PossibleBruteForceStream;

@query(name='FilterPortScanAttacks')
from PacketStream
filter syn == true and ack == false
insert into PossiblePortScanStream;
```

Figure 4.14: **DoS Attack Detector Used in Functionally Auto-scaling Deployment** – uses packets filtered by "packet_filter" shown in Figure 4.13. This query is depending on 'PossibleDosStream'. Therefore, it will not run until 'PossibleDosStream' receive some events.

```
@app(name='dos_detector', version='1.0.0', priority=5, requires=['
   PossibleDosStream'])

@source(type='kafka', bootstrap='localhost:9092', topic='
   PossibleDosStream')
def stream PossibleDosStream;

@sink(type='file.text', path='/temp/dos.txt')
def stream DosAttackStream;

from PossibleDosStream
    partition by destIp
    window.externalTimeBatch('timestamp', 1189)
    aggregate count() as no_of_packets
    filter no_of_packets >= 3
    select srcIp, destIp, no_of_packets, timestamp
insert into DosAttackStream;
```

full-fledged IDS is not feasible. Therefore, the proposed IDS is designed to start or stop its rules using the advantage of functionally auto-scaling "Wisdom" stream processor.

Functionally Auto-scaling IDS always requires a query to filter events as given in Figure 4.13. The above query reads packets from Apache Kafka message queue. Even though the input source can be configured to any other supported sources in "Wisdom" stream processor, Apache Kafka or any other message queues are recommended to handle unexpected unavailability of "Wisdom" instances. This query filters the packets and inserts them into different streams if they meet certain conditions. The $priority = 10$ property in $@app$ annotation informs the "Wisdom Manager" that the "packet_filter" query should not be stopped by any chance.

Output streams of "packet_filter" are used for further processing in separate "Wisdom" queries. For example, the "dos_detector" given in Figure 4.14 collects filtered packets from "packet_filter" through $PossibleDosStream$ and process them further to detect HTTP Slow Header DoS attack. The $priority = 5$ and $requires = ['PossibleDosStream']$ properties in

$@app$ annotation informs the "Wisdom Manager" that the "dos_detector" query should not run unless there are some events passed to the $PossibleDosStream$. Wisdom Manager keeps looking at the throughput of each stream decorated with $@config(stats = true)$ (See Figure 4.13) and starts all depending queries if a stream has a throughput greater than a predefined threshold $\tau$ which is 0 in the proposed IDS. If a stream has a throughput of 0 for a long time, the Wisdom Manager will stop all depending "Wisdom" instances with a priority less than a predefined threshold $\Theta$ to save system resources. In this method, latency is compromised for resource utilization because starting a new "Wisdom" instance takes some time. However, the system will not miss any packets since they are written to and read from Apache Kafka queue.

# Chapter 5

# Evaluation

The proposed IDS was evaluated in three phases: (1) to ensure the applicability of the MSeq2MSeq model in intrusion detection using DARPA 1999 dataset, (2) to test all components of the IDS using CICIDS 2017 dataset, and (3) to test the effectiveness of functionally auto-scaling deployment. First two evaluation phases have three tests per each and the last phase has a single test. This chapter covers all test case setups and their purpose. Results obtained for each test are discussed in Chapter 6.

## 5.1 Phase 1 - Test the MSeq2MSeq model using DARPA 1999 dataset

Phase 1 tests were developed to ensure that the MSeq2MSeq model can be used for real-time intrusion detection by treating streams of packets as sequences. As discussed in Chapter 1, Bontemps *et al.* used stacked LSTM RNN model for intrusion detection in DARPA 1999 dataset based on sequential anomaly detection. Results obtained in this evaluation phase are compared with the results obtained by Bontemps *et al.*

TCP packets from attack-free outside sniffing data of DARPA 1999 dataset were split into connections based on their sessions using PcapSplitter.[1] Connections with less than 4 packets were not used for training and connections with more than 60 packets were pruned to 60 because a connection must have at least 4 packets to train the model and 96.96% TCP connections in the training dataset have less than 60 packets. Connections having packets between 3 and 60, were padded with empty packets to maintain desired batch input format. The first three packets of a connection were used as input sequence to predict the rest.

The decoder was trained to predict $\{p_4, p_5, \ldots, p_n, \sigma_e\}$ using $\{\sigma_s, p_4, p_5, \ldots, p_n\}$ as input and hidden state of the encoder as the initial state. Here, $n$ is the number of packets in the connection. The model was trained using Teacher Forcing [54] because it reduces error propagation in testing. Even if the model can predict more than one upcoming packets, it must wait for actual packets to calculate the prediction error. Therefore, predicting $i + 1^{\text{th}}$ packet after the arrival of $i^{\text{th}}$ packet is enough and gives better results.

Three datasets were prepared from the DARPA 1999 dataset: (1) attack-free tcpdumps split into connections; (2) tcpdumps containing both attack and normal traffic within a day; and (3) tcpdumps containing both attack and normal traffic over a week.

### 5.1.1 Test 1.1 - Validate the MSeq2MSeq model using legitimate TCP connections

The first dataset was used to check the accuracy of the model on predicting packets of normal TCP connections and end of connections ($\sigma_e$) with a goal of validating the model. For this test, the first three packets of valid TCP connections were fed to the encoder and the rest were fed to the decoder one by one to predict following packets. Two packets were considered equal only if all their attributes match to each other. The accuracy of a predicted connection is calculated by (5.1).

[1] PcapPlusPlus available at `https://github.com/seladb/PcapPlusPlus`

$$accuracy = \frac{No\ of\ correct\ predictions}{No\ of\ packets\ in\ connection} * 100 \tag{5.1}$$

### 5.1.2 Test 1.2 - Determine the minimum accuracy threshold

TCP packets collected on Thursday of the second week outside sniffing data were split into streams having requests and responses between same source and destination. The first three packets from each stream were fed to the encoder and the rest were fed to the decoder one by one to predict the next packet until the decoder generates a $\sigma_e$. Suppose a $\sigma_e$ is generated after $i^{th}$ packet in a stream, the first $i$ packets will be considered as a connection and compared with predicted packets. The remaining packets in the stream will be used to predict next connection. If a $\sigma_e$ is not generated within $\tau$ packets, the decoder will emit predicted $\tau$ number of packets as a connection and a new prediction cycle will start from $\tau + 1^{th}$ packet. Even though most connections have less than 60 packets, $\tau$ is set to 100 in this test to be on the safe side.

In preliminary Test 1.2, the accuracy defined using exact match of packets resulted in more false positives. Therefore, predicted packets were compared with actual packets using a distance algorithm as given in Figure 5.1. The distance algorithm calculates a weighted distance by comparing individual categorical attributes of packets. The weight of each attribute is determined based on preliminary observations. The computed distance of each connection was used to define the accuracy of prediction. Suppose a predicted connection has $n$ packets, prediction accuracy of that connection is given by (5.2).

$$accuracy = \frac{\sum_{i=1}^{n} 1 - distance_i}{n} * 100 \tag{5.2}$$

Figure 5.1: **Weighted Packet Distance Algorithm** – used in Test 1.2 to calculate prediction accuracy.

**Input:** $actual\_packet, predicted\_packet, weights$
**Output:** $distance$
*Initialize:*
1: $distance \leftarrow 0$
2: **for all** $name \in actual\_packet.attributes$ **do**
3: **if** ($actual\_packet[name] \neq actual\_packet[name]$) **then**
4: $distance \leftarrow distance + weights[name]$
5: **end if**
6: **end for**
7: $distance \leftarrow distance / \sum weights$
8: **return** $distance$

### 5.1.3 Test 1.3 - Test the ability of the MSeq2MSeq model in real-time intrusion detection

Test 1.3 was developed to validate the application of the proposed model in real-time anomaly detection. Packets collected in the second week of DARPA 1999 dataset were preprocessed in the same way as in Test 1.2 and fed to the model. The system will raise an alarm in real-time, if the average weighted prediction error of 60 packets is less than 12.5% which is the mean accuracy of anomalous packets in Test 2. The percentage of true positive alarms and number of false positive alarms were compared with the results obtained by Bontemps et al. using stacked LSTM RNN on the same dataset.

## 5.2 Phase 2 - Test the Intrusion Detection System using CICIDS 2017 dataset

The MSeq2MSeq model trained on attack-free raw packets from CICIDS 2017 dataset as explained in Chapter 4 is used in this evaluation phase. Three tests were developed (1) to validate the MSeq2MSeq model and to test the integration of stream processor with the machine learner, (2) to compare the proposed CEP rule optimization algorithm with Turchin *et al.*'s work [18],

and (3) to test the ability of the proposed IDS in detecting attacks based on both anomaly and signature.

### 5.2.1 Test 2.1 - Test the integration of Wisdom Stream Processor with Machine Learning model

In the training phase, MSeq2MSeq model was trained without the intervention of stream processor to avoid additional delay introduced by the stream processor even though it is negligible. All nine MSeq2MSeq models were trained and validated individually. Test 2.1 was developed to test the integration of stream processor and all trained MSeq2MSeq models to detect anomaly-based attacks. Predictive "Wisdom" rules were developed as in Figure 4.10 for all buckets listed in Table 4.1 and attack free stream of packets were fed to those rules. As shown in Figure 4.10, predictive rules classify sequences with a prediction accuracy less than 12.5% as anomalies. The minimum accuracy threshold was selected from Test 1.2. All sequences classified as anomalies must be false positives in this test because the input traffic does not have any attack.

### 5.2.2 Test 2.2 - Test the self-tuning ability of Wisdom Stream Processor

In Test 2.2, three "Wisdom" rules were developed to detect HTTP Slow Header DoS attack, FTP brute force attack and "nmap -sS" Port scan probe. All these rules have two optimizable variables: *time_threshold* and *count_threshold*. A loss function as given in Figure 5.2 was developed to calculate the loss based on the number of false positive packets detected by stream processor. In the five days period of CICIDS 2017 dataset, FTP brute force, HTTP Slow Header DoS and Port scan attacks were simulated on Tuesday 9:20 - 10:20, Wednesday 10:14 - 10:35 and Friday 13:55 - 14:35, respectively. Packets transferred in a randomly selected 10 minutes interval from those attack simulations were extracted and used to optimize the "Wisdom" rules. After optimization, network packets from Tuesday, Wednesday, and Friday packet capture files

Figure 5.2: **Loss Function to Tune "Wisdom" Rules** – used in the proposed Intrusion Detection System. This function increases the loss by 100 for every false positive and decreases the loss by 10 for every true positive to reduce the number of false positives.

**Input:** $output\_events$, $exp\_src\_ip$
**Output:** $loss$
*Initialize:*
1: $loss \leftarrow 1,000,000$
2: **for all** $event \in output\_events$ **do**
3: **if** $exp_src_ip \neq event[`src_ip'] $ **then**
4: $loss \mathrel{+}= event[`no\_of\_packets'] * 100$
5: **else**
6: $loss \mathrel{-}= event[`no\_of\_packets'] * 10$
7: **end if**
8: **end for**
9: **return** $loss$

were fed to FTP brute force, HTTP Slow Header DoS and Port scan detectors respectively.

### 5.2.3 Test 2.3 - Test the complete Intrusion Detection System using CICIDS 2017 dataset

In Test 2.3, all queries developed in Test 2.1 and Test 2.2 were deployed together as depicted in Figure 5.3 to detect attacks based on both anomalies and signature. Another rule to detect SQL Injection based on the signature was developed without any variables to optimize. A filter to the anomaly-based "Wisdom" rules was added to ignore Transport Layer Security (TLS) packets since they caused false alarms in Test 2.1. Signature-based queries were deployed in self-tuning mode with initial threshold values obtained in Test 2.2. Even though the IDS is supposed to generate unique alert per attack, in this test the IDS was set to log all alerts to compare the precision of each component. The IDS groups packets into a sequence in anomaly-based detection or into a group of packets collected by the $IdleLengthTimeBatchWindow$ in signature-based detection and classify them as anomaly or not. The labeled CICIDS 2017 dataset has $flow$ of packets marked as an attack or not but the definition of a $flow$ is not available to the public. Furthermore, the complete dataset was used in this test which makes counting packets involved

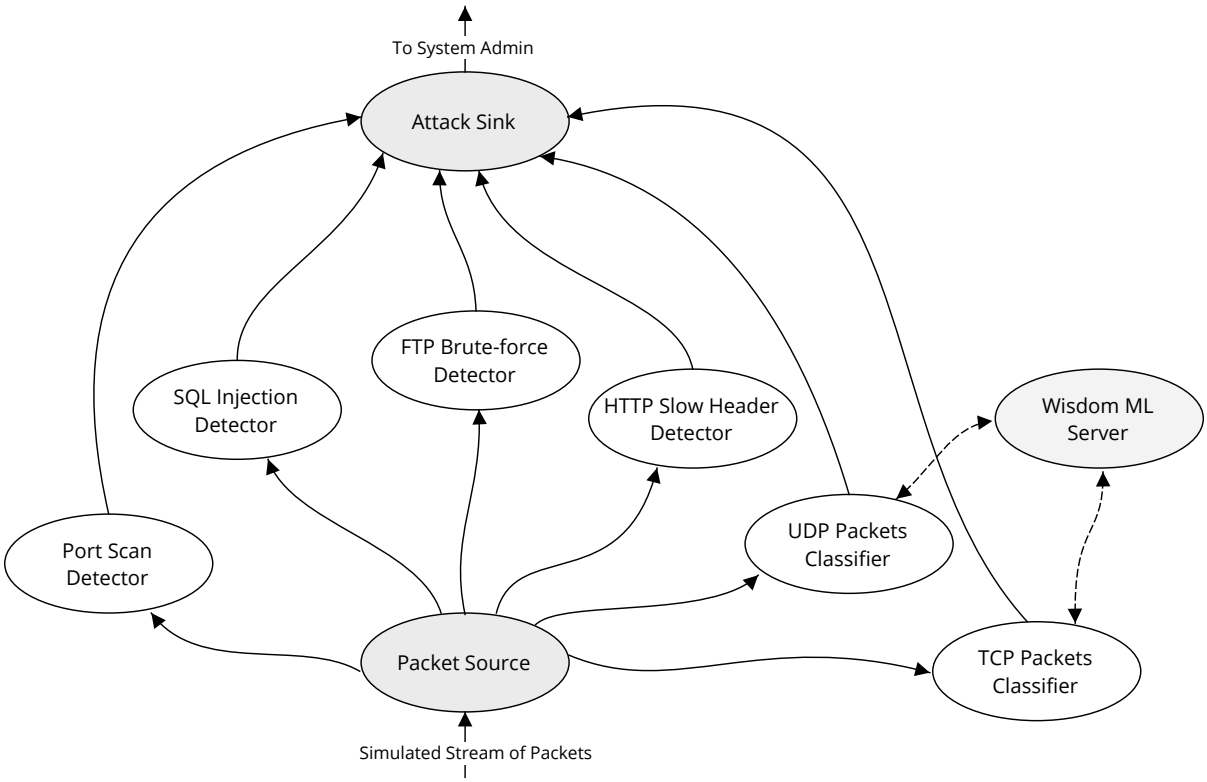



Figure 5.3: **Test 2.3 Deployment** – signature-based rules and anomaly-based rules deployed together in the IDS.

in attacks impossible. Therefore, the accuracy of the IDS was measured only in precision by manually comparing packets classified as anomalies with raw packets. Every attack simulated in the CICIDS 2017 dataset occurs only once but last for a long time. Therefore, detecting at least one anomalous sequence which is part of an attack is enough for an IDS to prevent the attack.

## 5.3 Phase 3 - Test the functionally auto-scaling ability of Wisdom Stream Processor

### 5.3.1 Test 3.1 - Compare the memory consumption of functionally auto-scaling deployment with manual deployment

Test 3.1 was developed to show the effectiveness of functionally auto-scaling stream processor in intrusion detection. To avoid the complexity, only the signature-based rules developed in Test 2.2 were used without enabling self-tuning ability. Deploying all three signature-based rules requires three "Wisdom" instances to run all the time. Instead, a filter query as given in Figure 4.13 was developed to filter incoming packets which can be part of HTTP Slow Header DoS attack, FTP brute force attack or Port scanning. Rest of the attack specific CEP operations were defined in separate rules and deployed as shown in Figure 5.4. If there is a possibility of any of these attacks, "Filter Query" sends the packet to the relative output stream. Wisdom Manager monitors the throughput of "Filter Query" output streams and starts a "Wisdom" instance with relevant CEP rule if the throughput is greater than 0. If there is no input for an attack detector for a long time, Wisdom Manager will stop the attack detector. To simulate real-time behavior, pcap files having no attack, FTP brute force attack, HTTP Slow Header DoS attack, and Port Scan were fed in order. The memory consumption of this automatic functionally auto-scaling deployment was compared with manual deployment where all three "Wisdom" instances were deployed and started using a Wisdom Manager instance without enabling the functionally auto-scaling feature.

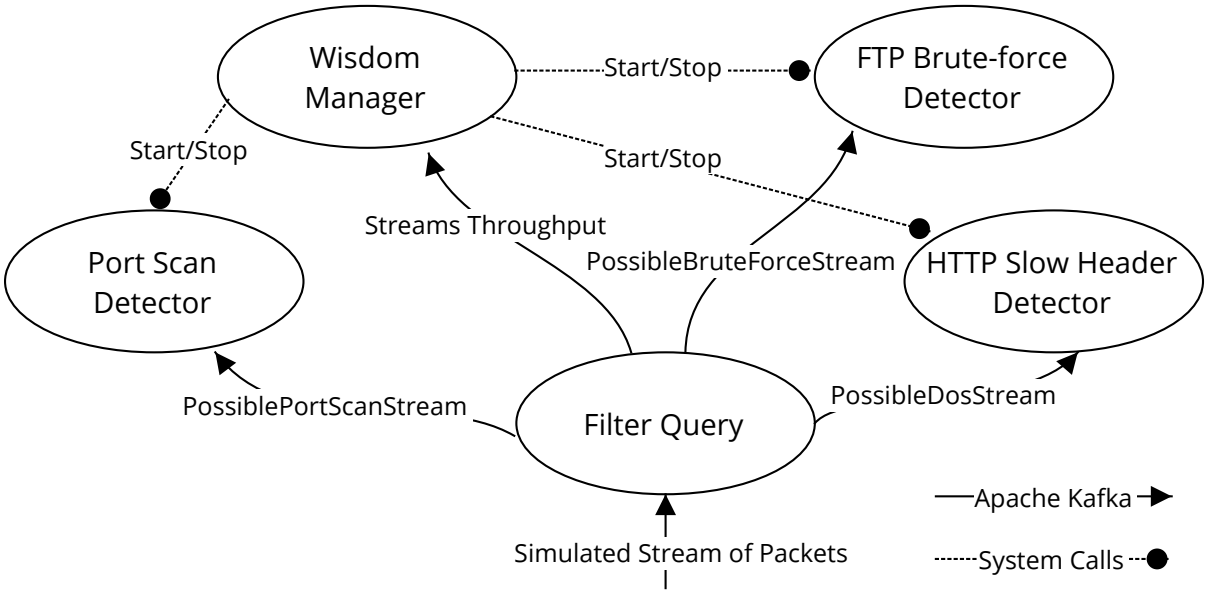



Figure 5.4: **Distributed and Functionally Auto-scaling Deployment** – Wisdom Manager controls the execution of "Wisdom" rules.

# Chapter 6

# Results

## 6.1 Phase 1 - Test the MSeq2MSeq model using DARPA 1999 dataset

### 6.1.1 Test 1.1 - Validate the MSeq2MSeq model using legitimate TCP connections

In Test 1.1, the model was able to predict connections with 84.97% accuracy and end of connections ($\sigma_e$) with 89.57% accuracy as shown in Figure 6.1. The obtained accuracy values are highly promising because two packets are considered equal only if all of their attributes are equal. As discussed in Chapter 5.1, more than 96% of connections have less than 60 packets. The prediction accuracy increases with number of packets per connection as shown in Figure 6.1.

### 6.1.2 Hypothesis

If the decoder is unable to find a connection in a stream of events, it will predict the maximum number of packets allowed for a sequence ($\tau$). Therefore, if the number of packets in a predicted connection is equal to $\tau$ (set to 100 in this experiment), either those packets are

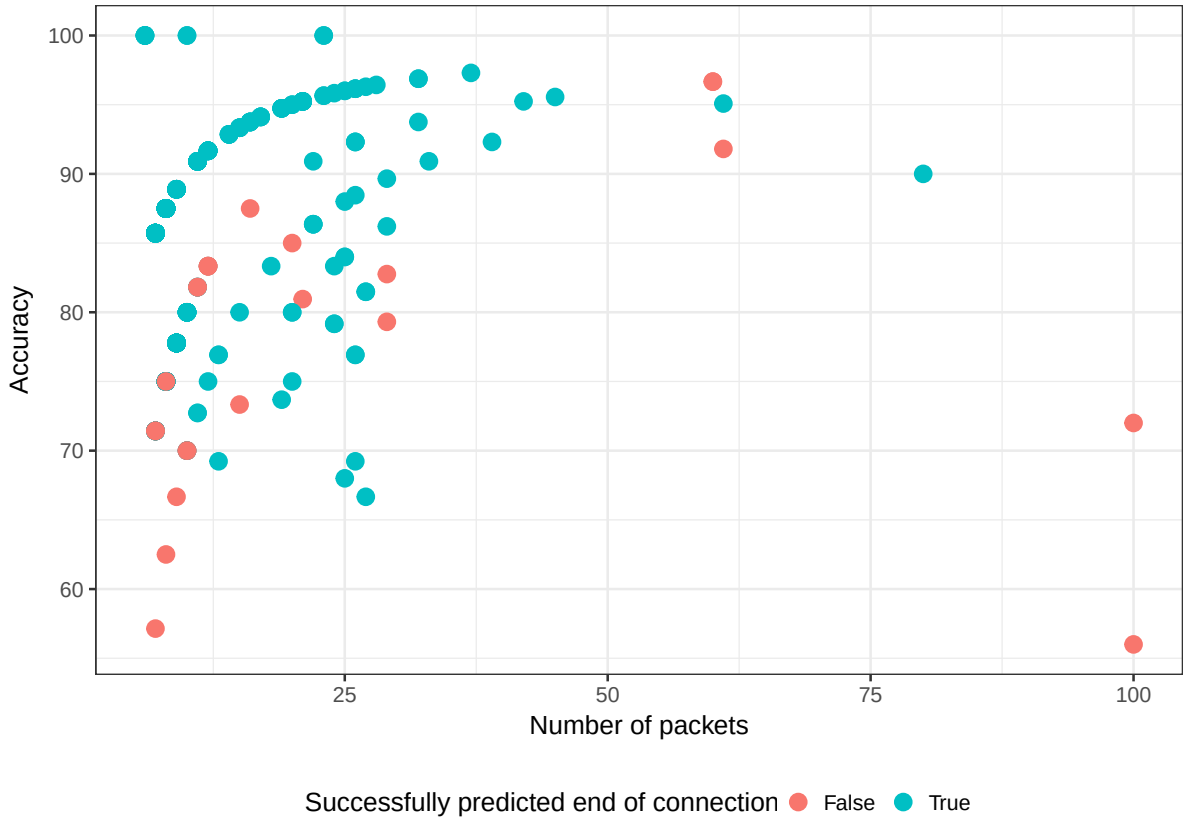



Figure 6.1: **Sequence Prediction Accuracy** – prediction accuracy and end of connection ($\sigma_e$) prediction accuracy.

anomalies or the actual connection has greater than or equal to $\tau$ number of packets. However, if those packets are from an actual connection, the model may be able to predict them with a higher accuracy even though it cannot reach the end of the connection. According to these facts, if a predicted connection has close to $\tau$ packets with less prediction accuracy, it may be an anomalous sequence.

### 6.1.3 Test 1.2 - Determine the minimum accuracy threshold

At the end of Test 1.2, a dataset with the number of packets in each predicted connection along with the prediction accuracy was prepared. This dataset was clustered using K-means clustering algorithm into six clusters as shown in Figure 6.2. The number of clusters was determined by cross-validation. As shown in Table 6.1, Cluster 6 has the lowest accuracy

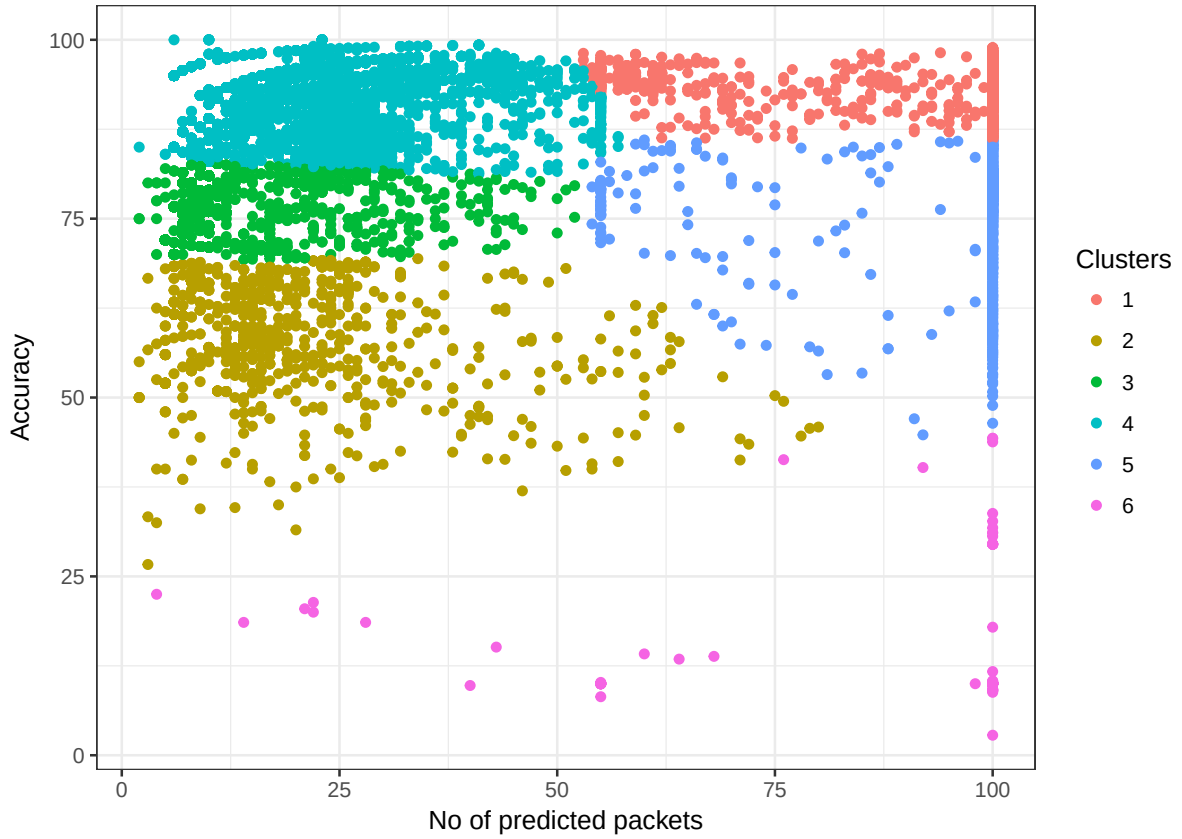



Figure 6.2: **Clusters of Predicted Connections** – cluster 6 has most of the sequences with a small prediction accuracy and large number of packets per connection.

(12.25%) and a high number of predicted packets (94.47), which supports the above hypothesis.

Even though the model classifies all attacks as anomalies without further distinctions, manual inspection of true positive packets from Cluster 6 reveals that anomalous packets are from either Port-Sweep probe or Neptune DoS attack. The proposed model is able to identify such anomalous packets with 97.02% Detection Ratio and 0.07% False Alarm Ratio.

### 6.1.4 Test 1.3 - Test the ability of the MSeq2MSeq model in real-time intrusion detection

In Test 1.3, the model was able to raise alarms on all Port-Sweep and Neptune DoS attacks with 100% true positive rate. Only a single False Alarm was raised in five days of network traffic. Bontemps *et al.* claimed 100% true positive alarms and 63 false alarms using LSTM on

| Cluster | Accuracy | No of packets |
|---|---|---|
| 1 | 95.19 | 97.08 |
| 2 | 61.80 | 9.52 |
| 3 | 76.52 | 7.68 |
| 4 | 89.16 | 13.06 |
| 5 | 76.66 | 96.49 |
| 6 | 12.25 | 94.47 |

Table 6.1: **Accuracy and Number of Packets** – cluster means of accuracy and number of predicted packets in Test 1.2.

the same dataset [6]. Furthermore, their model was able to detect only the Neptune DoS attack. Based on these results, we can claim that the proposed MSeq2MSeq model outperforms LSTM RNN in detecting anomalies in TCP traffic.

## 6.2 Phase 2 - Test the Intrusion Detection System using CICIDS 2017 dataset

### 6.2.1 Test 2.1 - Test the integration of Wisdom Stream Processor with Machine Learning model

In Test 2.1, the IDS generated 36 false alarms. Considering individual packets in each false alarms, the IDS classified 0.07% of packets as anomalies and the remaining 99.93% packets as legitimate packets. Although this is a low percentage, for an IDS, 36 false alarms in a day is not acceptable. Manual inspection of raw packets revealed that among these false alarms, 24 were caused by a massive amount of out of order packets, 9 were caused by TLS sequences and the remaining 3 were caused by a large number of HTTP GET requests sent by a client to a specific server in a short interval. Though the massive amount of out of order packets and anomalous GET requests are not intentional attacks in the dataset, they are potential network anomalies which can be used to attack systems. Therefore, they are treated as potential anomalies and the IDS is allowed to raise alarms. False alarms caused by TLS sequences are due to the inability of

the trained model in predicting such sequences. Less number of TLS sequences in the training data compared to other traffic can be a reason for lower prediction accuracy of TLS sequences. Test 2.1 also reveals that the model can be used in real-time with our stream processor and the predictive "Wisdom" rules are working as expected.

### 6.2.2 Test 2.2 - Test the self-tuning ability of Wisdom Stream Processor

| CEP Rule | Avg. Precision | Avg. Recall |
|---|---|---|
| FTP Brute Force | 100% | 99.61% |
| Slow Header DoS | 100% | 96.85% |
| Port Scan | 99.95% | 83.80% |

Table 6.2: **Precision and Recall of Optimized Wisdom Rules** – optimized using PSO and Bisection algorithm in Test 2.2.

As in Table 6.2, "Wisdom" rules optimized by the proposed optimization algorithm was able to detect selected attacks with a minimum precision of 99.95% and a maximum precision of 100%. The minimum recall was 83.80% and the maximum recall was 99.85%. The knowledge of domain expert and training data used to optimize the rule determine the accuracy of a signature-based rule. The port scan detector is looking for a large number of packets with *SYN* flags and unique destination port sent within a short interval. Therefore, there is a high chance of false port scan alarms. Turchin *et al.* obtained a maximum precision of 80% and a maximum recall of 90% with their probability-based CEP rule optimized using Kalman Filter after training the system using the complete dataset [18]. Above results, support the argument that humans are good at writing high-level signature-based rules and the proposed optimization algorithm helps to derive optimal threshold values with better accuracy.

| Attack | Static Rule | Self-tuning Rule |
|---|---|---|
| FTP Brute Force | 100% | 100% |
| Slow Header DoS | 99.96% | 100% |
| Port Scan | 71.43% | 80.38% |
| SQL Injection | 100% | N/A |

Table 6.3: **Precision of Signature-based Rules** – self-tuning "Wisdom" rules performed better than static rules with user-defined threshold values in Test 2.3.

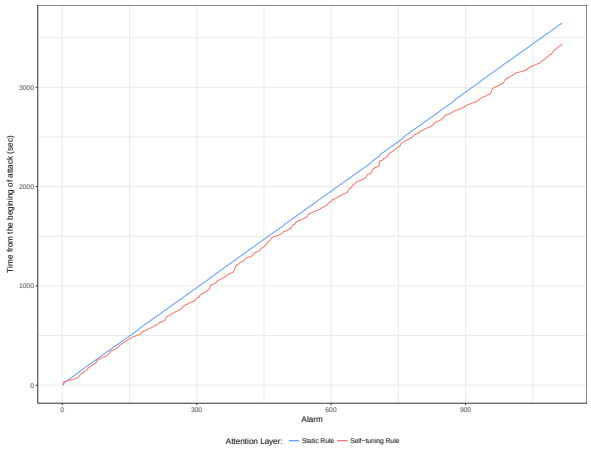



Figure 6.3: **Timestamp of Alarms Raised by FTP Bruteforce Detector** – timestamp of each alarm raised from the beginning of FTP Bruteforce attack. As time progress, the self-tuning rule generates alerts slightly faster than the static rule with user-defined thresholds.

### 6.2.3 Test 2.3 - Test the complete Intrusion Detection System using CICIDS 2017 dataset

In Test 2.3, self-tuning instances of signature-based rules were more accurate than static instances of signature-based rules as given in Table 6.3. It reveals that tuning thresholds in runtime improves the precision. As shown in Figure 6.3, Figure 6.4, and Figure 6.5 self-tuning FTP brute force detector raised alarms faster than the static signature-based rule without compromising the precision. However, self-tuning instances of DoS detector and Port Scan detector compromised the latency for precision. The latency versus precision tradeoff is depending on the loss function and the training data used to optimize the rule. However, considering targetted attacks, deployed signature-based rules have raised at least one alarm for each attack instances which claims 100% detection rate.

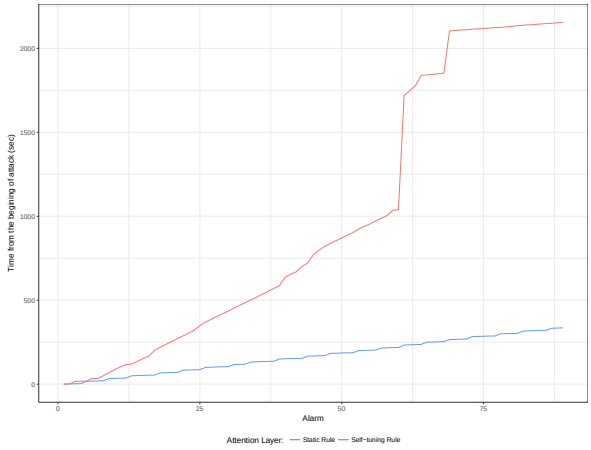



Figure 6.4: **Timestamp of Alarms Raised by HTTP Slow Header DoS Detector** – timestamp of each alarm raised from the beginning of HTTP Slow Header DoS attack. As time progress, the self-tuning rule generates alerts significantly later than the static rule with user-defined thresholds. However, the accuracy of self-tuning rule is slightly better than the accuracy of static rule.

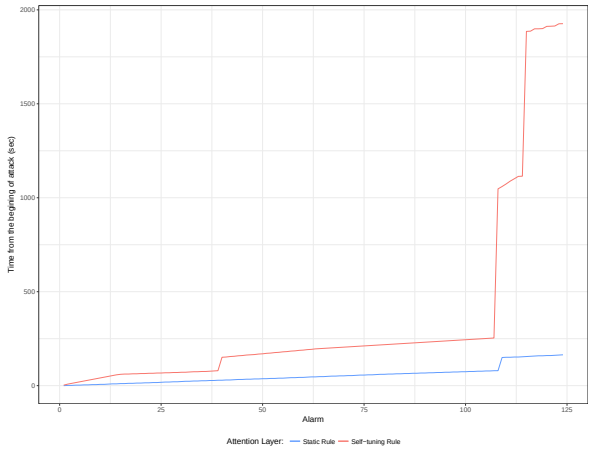



Figure 6.5: **Timestamp of Alarms Raised by Port Scan Detector** – timestamp of each alarm raised from the beginning of Port Scan probe. As time progress, the self-tuning rule generates alerts significantly later than the static rule with user-defined thresholds. However, the accuracy of self-tuning rule is significantly better than the accuracy of static rule.

Anomaly-based rules detected 76 unique anomalous sequences. Among them, 63 sequences were out of order TCP packets, and 5 sequences were anomalous GET requests. The remaining sequences were from HULK DoS, Web Brute Force, Cross-site scripting and Ares Botnet attacks. As discussed earlier in this chapter, out of order packets and anomalous GET requests are treated as potential anomalies. Therefore the proposed IDS claims 100% detection

rate with anomaly-based rules. Even though there was an instance of Distributed Denial of Service (DDoS) attack which involves a large volume of anomalous traffic, the anomaly-based detector was not able to detect it because of the definition of the sequence. Additional CEP rules are required to detect attacks which cannot be detected by the proposed anomaly-based detection.

## 6.3 Phase 3 - Test the functionally auto-scaling ability of Wisdom Stream Processor

### 6.3.1 Test 3.1 - Compare the memory consumption of functionally auto-scaling deployment with manual deployment

In Test 3.1 manual deployment, the overall memory consumption of every "Wisdom" instances was between 450 - 500 Megabyte (MB) from the beginning to end (See Figure 6.6). In functionally auto-scaling deployment, Wisdom Manager started Port scanning detector from the beginning because there were packets matching Port scanning filter even in normal traffic (See Figure 6.7). FTP brute force detector and HTTP Slow Header DoS were started only after feeding packets containing those attacks. FTP brute force detector was stopped after the attack but HTTP Slow Header DoS detector was stopped after the actual attack and started a few times due to some matching packets in later traffic. However, those packets were not reported by DoS detector as attacks. According to these results, we can conclude that functionally auto-scaling deployment requires less amount of system resources than running rules all the time without compromising the accuracy.

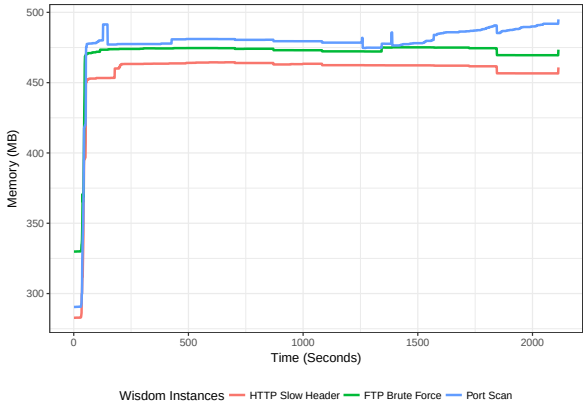



Figure 6.6: **Memory Consumption in Manual Deployment** – from the beginning, all "Wisdom" rules are actively checking for new packets.

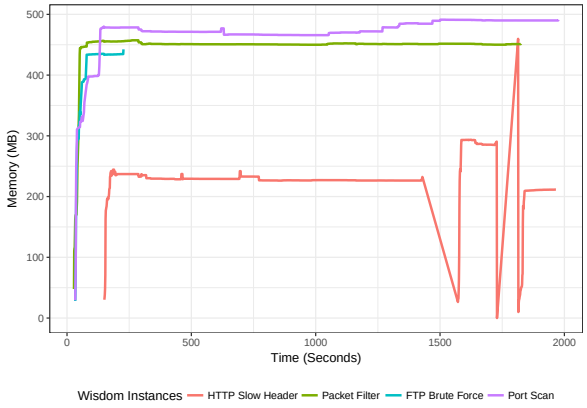



Figure 6.7: **Memory Consumption in Functionally Auto-scaling Deployment** – Packet Filter runs all the time, but other attack detectors run only if there is a possibility for those attacks.

# Chapter 7

# Conclusion

In this research, a novel MSeq2MSeq machine learning model, and an adaptive and functionally auto-scaling stream processor "Wisdom" were proposed to build an IDS for both anomaly-based and signature-based intrusion detection. The IDS developed using both machine learning and stream processing techniques was able to detect eight attacks with 100% detection rate. Above results are obtained by simulating a real-time network traffic by feeding streams of packets read from raw pcap files. In an attack which requires a lot of packets like DoS, the proposed IDS raises several alarms continuously. Therefore, the proposed IDS can be used to prevent network intrusions at their initial stage by modifying firewall rules with the first alarm generated by the IDS. Even though the MSeq2MSeq model has been used for packet prediction, it has a wide range of applications in other MSeq2MSeq problems like weather forecasting with multiple features and stock prediction. The "Wisdom" can be used as a general purpose stream processor with the ability to adapt itself, start new rules to add more features and stop unwanted rules to decrease resource consumption. Compared to signature-based rules, anomaly-based rules cannot differentiate a benign anomaly like an out of order packet from an attack. On the other hand, it is not always possible for a domain expert to come up with a signature to detect unknown attacks with false positive rates that are low enough for practical use. Therefore, combining both signature-based detection and anomaly-based detection takes the advantage of

signature-based rules without missing unknown attacks. Another advantage of using stream processor as an IDS is the ability to utilize external resources like system logs and knowledge about hosted services and legitimate users. However, none of them were used in this research because none of the publicly available datasets provide such additional information.

Compared to popular IDS's like Snort, the proposed IDS offers functionally auto-scaling deployment, SQL like query, anomaly-based intrusion detection, and self-tuning rules. The ability of the proposed IDS in processing events received from different sources makes it suitable to detect attacks with high accuracy. Among the four challenges raised by Sommer and Paxon, the proposed IDS has addressed the first three problems. The last challenge: "lack of appropriate public datasets" still affects the research because the accuracy of the machine learning model is depending on the quality of training data. Furthermore, the CICIDS 2017 dataset does not contain multiple occurrences of the same attack at different times which limits testing the long-term benefit of self-tuning rules. However, creating an ideal dataset is beyond the scope of this research. Training the machine learning model using more data will increase the accuracy of the anomaly-based detector. Similarly, writing new signature-based rules for every known attack will make the signature-based detector to detect all those attacks. As a future direction, I recommend deploying the proposed IDS in a honeynet for a long period with new self-tuning signature-based rules to test the long-term effect of the self-tuning IDS. A honeynet can also be used to test advanced signature-based rules using additional information sources like application logs and to continuously train the MSeq2MSeq model using the traffic received by the honeynet.

# Bibliography


[1] J. Seidl, "Goldeneye layer 7 (keepalive+nocache) dos test tool," may 2018. [Online]. Available: https://github.com/jseidl/GoldenEye

[2] B. Shteiman, "Hulk - http unbearable load king packet storm," may 2018. [Online]. Available: https://packetstormsecurity.com/files/112856/HULK-Http-Unbearable-Load-King.html

[3] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," Technical report, Tech. Rep., 2000.

[4] M. Sabhnani and G. Serpen, "Application of machine learning algorithms to kdd intrusion detection dataset within misuse detection context." 01 2003, pp. 209–215.

[5] A. Ahmed, A. Lisitsa, and C. Dixon, "A misuse-based network intrusion detection system using temporal logic and stream processing," in *2011 5th International Conference on Network and System Security*, Sept 2011, pp. 1–8.

[6] L. Bontemps, V. L. Cao, J. McDermott, and N. Le-Khac, "Collective anomaly detection based on long short term memory recurrent neural network," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1703.09752

[7] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 305–316.

[8] Cisco, “Snort - network intrusion detection & prevention system,” may 2018. [Online]. Available: https://www.snort.org/

[9] M. Ficco and L. Romano, “A generic intrusion detection and diagnoser system based on complex event processing,” in *2011 First International Conference on Data Compression, Communications and Processing*, June 2011, pp. 275–284.

[10] M. Blount, M. R. Ebling, J. M. Eklund, A. G. James, C. McGregor, N. Percival, K. Smith, and D. Sow, “Real-time analysis for intensive care: Development and deployment of the artemis analytic system,” *IEEE Engineering in Medicine and Biology Magazine*, vol. 29, no. 2, pp. 110–118, March 2010.

[11] S. Nielsen, C. Chambers, and J. Farr, “Fleet management systems and methods for complex event processing of vehicle-related information via local and remote complex event processing engines,” jun 2013, uS Patent 8,473,148.

[12] J. Hazra, K. Das, D. P. Seetharam, and A. Singhee, “Stream computing based synchrophasor application for power grids,” in *Proceedings of the First International Workshop on High Performance Computing, Networking and Analytics for the Power Grid*, ser. HiPCNA-PG ’11. New York, NY, USA: ACM, 2011, pp. 43–50. [Online]. Available: http://doi.acm.org/10.1145/2096123.2096134

[13] The Apache Software Foundation, “Apache flink: Scalable stream and batch data processing,” may 2018. [Online]. Available: https://flink.apache.org/

[14] EsperTech Inc, “Esper - espertech,” apr 2018. [Online]. Available: http://www.espertech.com/esper/

[15] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, “Siddhi: A second look at complex event processing architectures,” in *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*,

ser. GCE '11. New York, NY, USA: ACM, 2011, pp. 43–50. [Online]. Available: http://doi.acm.org/10.1145/2110486.2110493

[16] WSO2, "Siddhi query guide - siddhi," may 2018. [Online]. Available: https://wso2.github.io/siddhi/documentation/siddhi-4.0

[17] R. Bhargavi, R. Pathak, and V. Vaidehi, "Dynamic complex event processing - adaptive rule engine," in *2013 International Conference on Recent Trends in Information Technology (ICRTIT)*, July 2013, pp. 189–194.

[18] Y. Turchin, A. Gal, and S. Wasserkrug, "Tuning complex event processing rules using the prediction-correction paradigm," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '09. New York, NY, USA: ACM, 2009, pp. 10:1–10:12. [Online]. Available: http://doi.acm.org/10.1145/1619258.1619272

[19] R. Mousheimish, Y. Taher, and K. Zeitouni, "Automatic learning of predictive cep rules: Bridging the gap between data mining and complex event processing," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '17. New York, NY, USA: ACM, 2017, pp. 158–169. [Online]. Available: http://doi.acm.org/10.1145/3093742.3093917

[20] A. Margara, G. Cugola, and G. Tamburrelli, "Learning from the past: Automated rule generation for complex event processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: ACM, 2014, pp. 47–58. [Online]. Available: http://doi.acm.org/10.1145/2611286.2611289

[21] O.-J. Lee and J. E. Jung, "Sequence clustering-based automated rule generation for adaptive complex event processing," *Future Generation Computer Systems*, vol. 66, pp.

100 – 109, 2017. [Online]. Available: {http://www.sciencedirect.com/science/article/pii/S0167739X16300243}

[22] N. Mehdiyev, J. Krumeich, D. Werth, and P. Loos, "Determination of event patterns for complex event processing using fuzzy unordered rule induction algorithm with multi-objective evolutionary feature subset selection," in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, Jan 2016, pp. 1719–1728.

[23] G. Combs, "Wireshark  go deep." may 2018. [Online]. Available: https://www.wireshark.org/

[24] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. NIPS*, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[25] G. Loganathan, J. Samarabandu, and X. Wang, "Sequence to sequence pattern learning algorithm for real-time anomaly detection in network traffic," in *2018 IEEE Canadian Conference on Electrical & Computer Engineering (CCECE) (CCECE 2018)*, Quebec City, Canada, May 2018.

[26] G. Beni and J. Wang, "Swarm intelligence in cellular robotic systems," in *Robots and Biological Systems: Towards a New Bionics?*, P. Dario, G. Sandini, and P. Aebischer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 703–712.

[27] G. Loganathan, J. Samarabandu, and X. Wang, "Real-time intrusion detection in network traffic using adaptive and auto-scaling stream processor," in *2018 IEEE Global Communications Conference: Communication & Information System Security (Globecom2018 CISS) - In Press*, Abu Dhabi, United Arab Emirates, Dec. 2018.

[28] Google, "grpc," https://www.grpc.io, jun 2018.

[29] The Apache Software Foundation, "Apache kafka," apr 2018. [Online]. Available: https://kafka.apache.org/

[30] S. Jayasekara, S. Kannangara, T. Dahanayakage, I. Ranawaka, S. Perera, and V. Nanayakkara, "Wihidum: Distributed complex event processing," *Journal of Parallel and Distributed Computing*, vol. 79-80, pp. 42 – 51, 2015, special Issue on Scalable Systems for Big Data Management and Analytics. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731515000519

[31] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. " O'Reilly Media, Inc.", 2016.

[32] I. Sharafaldin, A. Habibi Lashkari, and A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," 01 2018, pp. 108–116.

[33] Sweet Software, "Ares," may 2018. [Online]. Available: https://github.com/sweetsoftware/Ares

[34] A. González, P. Lobato, M. A. Lopez, and O. C. M. B. Duarte, "An accurate threat detection system through real-time stream processing," in *SemanticScholar*, 2016.

[35] Bro Project Team, "The bro network security monitor," may 2018. [Online]. Available: https://www.bro.org/

[36] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014, pp. 1724–1734. [Online]. Available: http://www.aclweb.org/anthology/D14-1179

[37] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014. [Online]. Available: http://arxiv.org/abs/1409.0473

[38] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2015, pp. 1412–1421. [Online]. Available: http://aclweb.org/anthology/D15-1166

[39] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2016, pp. 1480–1489. [Online]. Available: http://www.aclweb.org/anthology/N16-1174

[40] J. Gehring, M. Auli, D. Grangier, and Y. N. Dauphin, "A Convolutional Encoder Model for Neural Machine Translation," *ArXiv e-prints*, 2016.

[41] L. Yao, A. Torabi, K. Cho, N. Ballas, C. Pal, H. Larochelle, and A. Courville, "Describing videos by exploiting temporal structure," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 4507–4515.

[42] N. Srivastava, E. Mansimov, and R. Salakhutdinov, "Unsupervised learning of video representations using lstms," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 843–852. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045209

[43] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff, "Lstm-based encoder-decoder for multi-sensor anomaly detection," vol. abs/1607.00148, 2016. [Online]. Available: http://arxiv.org/abs/1607.00148

[44] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long short term memory networks for anomaly detection in time series," in *ESANN, 23rd European Symposium on Artificial Neural Networks, Compu- tational Intelligence and Machine Learning*, 2015.

[45] M.-T. Luong, "Neural Machine Translation," no. December, p. 156, 2016. [Online]. Available: https://github.com/lmthang/thesis

[46] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz, "Boa: The bayesian optimization algorithm," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, ser. GECCO'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 525–532. [Online]. Available: http://dl.acm.org/citation.cfm?id=2933923.2933973

[47] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, Jan 2016.

[48] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 2951–2959. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999325.2999464

[49] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," *CoRR*, vol. abs/1606.06543, 2016. [Online]. Available: http://arxiv.org/abs/1606.06543

[50] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, Nov 1995, pp. 1942–1948 vol.4.

[51] F. H. F. Leung, H. K. Lam, S. H. Ling, and P. K. S. Tam, "Tuning of the structure and parameters of a neural network using an improved genetic algorithm," *IEEE Transactions on Neural Networks*, vol. 14, no. 1, pp. 79–88, Jan 2003.

[52] Z.-L. Gaing, "A particle swarm optimization approach for optimum design of pid controller in avr system," *IEEE Transactions on Energy Conversion*, vol. 19, no. 2, pp. 384–391, June 2004.

[53] R. Hassan, B. Cohanim, O. De Weck, and G. Venter, "A comparison of particle swarm optimization and the genetic algorithm," in *46th AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference*, 2005, p. 1897.

[54] J. W. Ronald and Z. David, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.

[55] D. Harris and S. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[56] N. Tripathi, N. Hubballi, and Y. Singh, "How secure are web servers? an empirical study of slow http dos attacks and detection," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*, Aug 2016, pp. 454–463.

[57] P. Bonami, M. Kilinç, and J. Linderoth, "Algorithms and software for convex mixed integer nonlinear programs," in *Mixed Integer Nonlinear Programming*, J. Lee and S. Leyffer, Eds. New York, NY: Springer New York, 2012, pp. 1–39.

[58] R. Hemmecke, M. Köppe, J. Lee, and R. Weismantel, *Nonlinear integer programming*, 2010.

[59] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, "The 1999 darpa off-line intrusion detection evaluation," *Comput. Netw.*, vol. 34, no. 4, pp. 579–595, 2000. [Online]. Available: http://dx.doi.org/10.1016/S1389-1286(00)00139-0

[60] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the kdd cup 99 data set," in *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, July 2009, pp. 1–6.

[61] N. Moustafa and J. Slay, "Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set)," in *2015 Military Communications and Information Systems Conference (MilCIS)*, Nov 2015, pp. 1–6.

[62] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *ECOOP' 93 — Object-Oriented Programming*, O. M. Nierstrasz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 406–431.

[63] Oracle, "Java + you," apr 2018. [Online]. Available: https://www.java.com/en/

[64] Google Inc, "Tensorflow," jun 2018. [Online]. Available: https://www.tensorflow.org

[65] Raspberry Pi Foundation, "Raspberry pi - teach, learn, and make with raspberry pi," apr 2018. [Online]. Available: https://www.raspberrypi.org/

# Appendix A

# Signature-based Wisdom Rules

## A.1 HTTP SlowHeader Detector

```
@app(name='HTTPSlowHeaderDetector', version='1.0.0')

@source(type='kafka', bootstrap='localhost:9002')
def stream PacketStream;

@sink(type='console')
@sink(type='file.text', path='/tmp/dos_attack.txt')
def stream DoSAttackStream;

@config(trainable=true, minimum=100, maximum=60000, step=-1)
def variable time_threshold = 1189;

@config(trainable=true, minimum=3, maximum=1000, step=1)
def variable count_threshold = 3;

from PacketStream
    filter 'http' == app_protocol and
        dst_port == 80 and '\r\n\r\n' in data
        and 'Keep-Alive: \\d+' matches data
    partition by dst_ip
    window.externalTimeBatch('timestamp', $time_threshold)
    aggregate count() as no_of_packets
    filter no_of_packets >= $count_threshold
    select src_ip, dst_ip, timestamp
insert into DoSAttackStream;
```

## A.2 FTP Brute Force Detector

```
@app(name='FTPBruteForceDetector', version='1.0.0')

@source(type='kafka', bootstrap='localhost:9002')
def stream PacketStream;

@sink(type='console')
@sink(type='file.text', path='/tmp/ftp_brute_force_attack.txt')
def stream FTPBruteForceAttack;

@config(trainable=true, minimum=100, maximum=60000, step=-1)
def variable time_threshold = 3220;

@config(trainable=true, minimum=3, maximum=1000, step=1)
def variable count_threshold = 7;

from PacketStream
    filter 'FTP[CONTROL]' == app_protocol and '530 Login incorrect'
       in data
    partition by dst_ip
    window.externalTimeBatch('timestamp', $time_threshold)
    aggregate count() as no_of_packets
    filter no_of_packets >= $count_threshold
    select src_ip, dst_ip, no_of_packets, timestamp
insert into FTPBruteForceAttack;
```

## A.3 Port Scan Detector

```
@app(name='PortScanDetector', version='1.0.0')

@source(type='kafka', bootstrap='localhost:9002')
def stream PacketStream;

@sink(type='console')
@sink(type='file.text', path='/tmp/port_scan_probe.txt')
def stream PortScanProbe;

@config(trainable=true, minimum=100, maximum=60000, step=-1)
def variable time_threshold = 108;

@config(trainable=true, minimum=3, maximum=1000, step=1)
def variable count_threshold = 9;

from PacketStream
    filter syn == true and ack == false
    partition by src_ip + dst_ip
    window.unique:externalTimeBatch('dst_port', 'timestamp',
       $time_threshold)
    aggregate count() as no_of_packets
    filter no_of_packets >= $count_threshold
    select src_ip, dst_ip, no_of_packets, timestamp
insert into PortScanProbe;
```

## A.4 SQL Injection Detector

```
@app(name='SQLInjectionDetector', version='1.0.0')

@source(type='kafka', bootstrap='localhost:9002')
def stream PacketStream;

@sink(type='console')
@sink(type='file.text', path='/tmp/sql_injection_probe.txt')
def stream SQLInjectionAttack;

from PacketStream
    filter app_protocol == 'HTTP'
        and '/dv/vulnerabilities/sqli/' in data
    select src_ip, dst_ip, timestamp
insert into SQLInjectionAttack;
```

# Appendix B

# Anomaly-based Wisdom Rules

## B.1 TCP Packets Filter

```
@app(name='TCPPacketsFilter', version='1.0.0', playback='timestamp')

@source(type='kafka', bootstrap='localhost:9002')
def stream PacketStream;

@sink(type='kafka', bootstrap='localhost:9002', topic='TCPStream')
def stream TCPStream;

from PacketStream
    filter transport_layer == 'TCP' and
        not (src_port == 443 or dst_port == 443)
    select highest_layer, transport_layer, src_ip, src_port, dst_ip,
        dst_port, ip_flag, transport_flag, timestamp
    partition by src_ip + dst_ip
    map copy('src_ip') as srcIp, copy('dst_ip') as dstIp
    window.idleTimeLengthBatch(time.sec(1), 1000)
    limit 320
    aggregate collect('src_ip') as src_ip,
        collect('dst_ip') as dst_ip, collect('ip_flag') as ip_flag,
        collect('transport_flag') as transport_flag,
        collect('transport_layer') as transport_layer,
        collect('highest_layer') as highest_layer
    select src_ip, dst_ip, highest_layer, transport_layer, src_ip,
       dst_ip, ip_flag, transport_flag, srcIp, dstIp, timestamp
    map len('ip_flag') as no_of_packets
insert into TCPStream;
```

## B.2 UDP Packets Filter

```
@app(name='UDPPacketsFilter', version='1.0.0')

@source(type='kafka', bootstrap='localhost:9002')
def stream PacketStream;

@sink(type='kafka', bootstrap='localhost:9002', topic='UDPStream')
def stream UDPStream;

from PacketStream
    filter transport_layer == 'UDP'
    select highest_layer, transport_layer, src_ip, src_port, dst_ip,
        dst_port, ip_flag, transport_flag, timestamp
    partition by src_ip + dst_ip
    map copy('src_ip') as srcIp, copy('dst_ip') as dstIp
    window.idleTimeLengthBatch(time.sec(1), 1000)
    limit 320
    aggregate collect('src_ip') as src_ip,
        collect('dst_ip') as dst_ip, collect('ip_flag') as ip_flag,
        collect('transport_flag') as transport_flag,
        collect('transport_layer') as transport_layer,
        collect('highest_layer') as highest_layer
    select src_ip, dst_ip, highest_layer, transport_layer, src_ip,
       dst_ip, ip_flag, transport_flag, srcIp, dstIp, timestamp
    map len('ip_flag') as no_of_packets
insert into UDPStream;
```

## B.3 TCP Bucket Connector

```
@app(name='TCPBucketConnector', version='1.0.0')

@source(type='kafka', bootstrap='localhost:9002', topic='TCPStream')
def stream TCPStream;

@sink(type='kafka', bootstrap='localhost:9002', topic='
   ProcessedStream')
def stream ProcessedStream;

from TCPStream
    filter no_of_packets > 3 and no_of_packets <= 10
    map grpc('localhost:9001', 'accuracy') as accuracy
insert into ProcessedStream;

from TCPStream
    filter no_of_packets > 10 and no_of_packets <= 20
    map grpc('localhost:9002', 'accuracy') as accuracy
insert into ProcessedStream;

from TCPStream
    filter no_of_packets > 20 and no_of_packets <= 40
    map grpc('localhost:9003', 'accuracy') as accuracy
insert into ProcessedStream;

from TCPStream
    filter no_of_packets > 40 and no_of_packets <= 80
    map grpc('localhost:9004', 'accuracy') as accuracy
insert into ProcessedStream;

from TCPStream
    filter no_of_packets > 80 and no_of_packets <= 160
    map grpc('localhost:9005', 'accuracy') as accuracy
insert into ProcessedStream;

from TCPStream
    filter no_of_packets > 160 and no_of_packets <= 320
    map grpc('localhost:9006', 'accuracy') as accuracy
insert into ProcessedStream;
```

## B.4 UDP Bucket Connector

```
@app(name='UDPBucketConnector', version='1.0.0')

@source(type='kafka', bootstrap='localhost:9002', topic='UDPStream')
def stream UDPStream;

@sink(type='kafka', bootstrap='localhost:9002', topic='
   ProcessedStream')
def stream ProcessedStream;

from UDPStream
    filter no_of_packets > 3 and no_of_packets <= 10
    map grpc('localhost:9101', 'accuracy') as accuracy
insert into ProcessedStream;

from UDPStream
    filter no_of_packets > 10 and no_of_packets <= 20
    map grpc('localhost:9102', 'accuracy') as accuracy
insert into ProcessedStream;

from UDPStream
    filter no_of_packets > 20 and no_of_packets <= 40
    map grpc('localhost:9103', 'accuracy') as accuracy
insert into ProcessedStream;
```

## B.5 Processed Stream Filter

```
@app(name='ProcessedStreamFilter', version='1.0.0')

@source(type='kafka', bootstrap='localhost:9002', topic='
   ProcessedStream')
def stream ProcessedStream;

@sink(type='console')
@sink(type='file.text', path='/tmp/anomalous_attacks.txt')
def stream AttackStream;

from ProcessedStream
    filter accuracy > 0 and accuracy < 0.125
    select srcIp, dstIp, timestamp, no_of_packets, accuracy
insert into AttackStream;
```

# Curriculum Vitae

| | |
|---|---|
| **Name:** | Gobinath Loganathan |
| **Post-Secondary Education and Degrees:** | University of Moratuwa<br>Moratuwa, Sri Lanka<br>2011 - 2016 BSc.Eng<br><br>University of Western Ontario<br>London, ON<br>2017 - 2018 MESc. |
| **Honours and Awards:** | runner-up for the best paper in<br>CCECE 2018 Conference |
| **Related Work Experience:** | Teaching Assistant<br>The University of Western Ontario<br>2017 - 2018<br><br>Software Engineer<br>WSO2<br>2016 - 2017 |

**Publications:**

G. Loganathan, J. Samarabandu, and X. Wang, "Real-time Intrusion Detection in Network Traffic Using Adaptive and Auto-scaling Stream Processor," in *2018 IEEE Global Communications Conference: Communication & Information System Security (Globecom2018 CISS) - In Press*, Abu Dhabi, United Arab Emirates, Dec. 2018.

G. Loganathan, J. Samarabandu, and X. Wang, "Sequence to sequence pattern learning algorithm for real-time anomaly detection in network traffic," in *2018 IEEE Canadian Conference on Electrical & Computer Engineering (CCECE) (CCECE 2018)*, Quebec City, Canada, May 2018.

G. Loganathan, V. Balachandrasarma, A. Anton Yogarajah, S. R. Dharmasena, M. Walpola and S. Perera, "An ORM Based Context Model for Context-Aware Computing," in *5th EAI International Conference on Context-Aware Systems and Applications (ICCASA 2016)*, Ho Chi Minh City, Vietnam, Nov 2016.