UNIVERSITY OF LIVERPOOL

DOCTORAL THESIS

Process Mining and Machine Learning for Intrusion Detection

Author:
Yinzheng Zhong

Supervisors:
Alexei Lisitsa
Yannis Goulermas

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Data Mining and Machine Learning Group
Department of Computer Science

May 31, 2023

Declaration of Authorship

I, Yinzheng Zhong, declare that this thesis titled, "Process Mining and Machine Learning for Intrusion Detection" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: 钟胤征

Date: 31 May 2023

“My brain is only a receiver, in the Universe there is a core from which we obtain knowledge, strength and inspiration. I have not penetrated into the secrets of this core, but I know that it exists.”

Nikola Tesla

UNIVERSITY OF LIVERPOOL

Abstract

School of Electrical Engineering, Electronics and Computer Science
Department of Computer Science

Doctor of Philosophy

Process Mining and Machine Learning for Intrusion Detection

by Yinzheng Zhong

With the increasing volume of internet traffic and the growth of the variety of internet services, the amount of cyber-attacks has increased vastly in recent years. Methods used to detect and prevent cyber-attacks are called intrusion detection systems. These systems prevent damage or compromise to the integrity, availability and confidentiality of infrastructures. However, the continuously increasing amount of data poses problems to the current intrusion detection methods. An intrusion detection system may suffer from a lack of efficiency, a lack of the ability to work with encrypted data and unable to find causal relationships between the cyber-attack and concurrent internet connections.

The thesis introduces a novel algorithm that is developed to address some of the existing issues of current intrusion detection systems. This technique takes advantage of process mining in the encoding of event data. Process mining is designed to discover the process model from the event log automatically and analyse the generated model. The performance of using process mining for intrusion detection has been verified and analysed at the early stage of this research. Then the process mining algorithm was modified with the combination of online processing capabilities. The resulting algorithm is a feature generator that takes the event log as the input and outputs a sequence of matrices that is suitable for machine learning and other processing.

The performance and efficiency of the feature generator have been verified with different datasets and machine learning algorithms. Results show that all the machine-learning algorithms that have been tested in classification yield accuracy that proves the generated feature can be used for intrusion detection. Verification has also been taken on anomaly detection approaches with various unsupervised machine learning algorithms, which further illustrate that the generated feature contains a higher abstraction of information of intrusions. The generation processing is efficient, and the processing speed is able to handle bandwidth in practical use.

Acknowledgements

I want to thank Dr Alexei Lisitsa, my first supervisor, for all of his suggestions and guidance throughout the research. I also want to thank Prof Yannis Goulermas, my second supervisor, for all of his help during my first year of research. I am incredibly grateful to the University of Liverpool for giving me access to the materials I needed for my research. I'd like to express my gratitude to Fluxicon for providing me with the licence to use the process mining software Disco. Finally, I would like to thank everyone who contributed to the open-source software that I have used during my research.

Contents

List of Figures

List of Tables

List of Abbreviations

ABOD	**A**ngle-**B**ased **O**utlier **D**etection
AUC	**A**rea **U**nder the **C**urve
BPM	**B**usiness **P**rocess **M**anagement
BPMN	**B**usiness **P**rocess **M**odel and **N**otation
CBLOF	**C**lustering-**b**ased **L**ocal **O**utlier **F**actor
CE	**C**ongestion **E**ncountered
CNN	**C**onvolutional **N**eural **N**etwork
COPOD	**Cop**ula-based **O**utlier **D**etection
DFG	**D**irectly-**F**ollows **G**raph
DPI	**D**eep **P**acket **I**nspection
DoS	**D**enial **o**f **S**ervice
DDoS	**D**istributed **D**enial **o**f **S**ervice
ECN	**E**xplicit **C**ongestion **N**otification
ECT	**E**CN-**C**apable **T**ransport
FEC	**F**orward **E**rror **C**orrection
FIFO	**F**irst **I**n, **F**irst **O**ut
FN	**F**alse **N**egatives
FP	**F**alse **P**ositives
EOS	**E**nd **o**f **S**equence
EOT	**E**nd **o**f **T**race
FM	**F**uzzy **M**iner
GUI	**G**raphical **U**ser **I**nterface
HBOS	**H**istogram-**b**ased **O**utlier **S**core
HIDS	**H**ost(-based) **I**ntrusion **D**etection **S**ystem
IDS	**I**ntrusion **D**etection **S**ystem
IForest	**I**solation **Forest**
IM	**I**nductive **M**iner
IPCA	**I**ncremental **P**rincipal **C**omponent **A**nalysis
ISN	**I**nitial **S**equence **N**umber
KNN	**K**-**N**earest **N**eighbours
LOF	**L**ocal **O**utlier **F**actor
LSTM	**L**ong **S**hort-**T**erm **M**emory
MLP	**M**ulti-**L**ayer **P**erceptions
MND	**M**ultivariate **N**ormal **D**istribution
MSL	**M**aximum **S**egment **L**ifetime
MSS	**M**aximum **S**egment **S**ize
MTU	**M**aximum **T**ransmission **U**nit
NIDS	**N**etwork(-based) **I**ntrusion **D**etection **S**ystem
NLP	**N**atural **L**anguage **P**rocessing
OOO	**O**ut **o**f **O**rder
PCA	**P**rincipal **C**omponent **A**nalysis
RNN	**R**ecurrent **N**eural **N**etworks

ROC	**R**eceiver **O**perating **C**haracteristic
SOS	**S**tart **o**f **S**equence
SOT	**S**tart **o**f **T**race
TCB	**T**ransmission **C**ontrol **B**lock
TCP	**T**ransmission **C**ontrol **P**rotocol
TET	**T**emporal **E**vent **T**able
TFGen	**T**ransition-based **F**eature **Gen**erator
TN	**T**rue **N**egatives
TP	**T**rue **P**ositives

Dedicated to my family, especially my parents and my wife, without whose constant support this thesis would not have been possible. . .

Chapter 1

Introduction

1.1 Overview

Process mining [1] is a method for discovering process models from event logs, analysing the generated process models, and updating the existing process models with newly discovered processes. A system for detecting intrusions checks for unauthorised behaviours or patterns that compromise the system's integrity and availability. Initially, the fundamental principles of process mining and intrusion detection will be introduced. Then, the thesis's motivation, scope, and structure will be presented.

1.2 Process Mining

Generally speaking, process mining is a two-step method. The first step is the process discovery step, during which event log data is used to construct process models. Analyzing the mind process models is the second step. Our generic feature generator derives its conceptual underpinnings from process mining algorithms. A process can be thought of as either a procedure or a series of actions followed in order to achieve a specific goal. There are procedures involved in every facet of our day-to-day lives, from the operations of well-established businesses to the running of private households. Both the production process for automobiles and the process for completing an order placed by a customer can be found in the industrial sector. When carried out in one's own home, the procedure is analogous to the series of actions required to cook a meal or switch on the television.

1.2.1 Business Process Management

Businesses tend to break down the process in a manageable way to decrease the chance of errors being created. Considering an online retail business selling groceries to local customers on its website. Usually, the way the orders are processed will be the following. The customer needs to add items to the shopping cart and go to checkout; at this stage, the order status will be Pending. Later the customer makes the payment, and the status of this order will be changed to Processing once the payment is confirmed. The warehouse then picks items for orders that are in the Processing stage, and the driver will be ready when the items are fully picked. The driver gets the order and marks the status as Delivering, and once the customer receives the order, the order status is Completed. A simple process model can be created to demonstrate the process in Fig. 1.1. The process starts from the left circle and then follows the direction of the arrows until the end place (right circle).

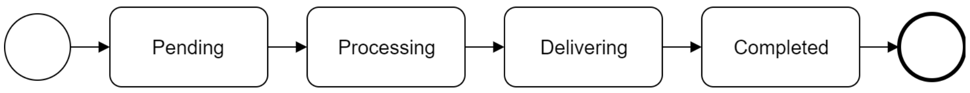


FIGURE 1.1: A simple business process model and notation (BPMN) for the online retailer example.

The business owner designs the model and instructs employees to adhere to it to prevent errors. Obviously, the employees do not wish to ship the goods before receiving payment. The idea behind breaking down a process is that it can be divided into sub-processes in a divide-and-conquer fashion. Also, a step can be a process; for instance, the Delivering step in order processing can be a process consisting of the following sub-steps: the driver receives the order, loads it onto the vehicle, drives to the destination, contacts the customer, and has the customer sign the delivery receipt.

1.2.2 Process Discovery

The order processing procedure depicted in Figure 1.1 is intended for a scenario in which everything runs smoothly and without exceptions; however, this is not the case in practice. The following are some common order processing exceptions.

1. Payment never received.
2. Request to cancel during processing/delivering.
3. Missing items during picking.
4. Item damaged during delivery.

To address these four concerns, it is necessary to modify the model so that it looks like Figure 1.2. Similarly, the process starts from the left circle. After the order has been submitted by the customer, the system will wait for the payment for a time period, and the payment will either be successful or timed out. In the case of a successful payment, the customer can amend the order before it is processed. The model is significantly more complicated than it was before, despite the fact that only four of the possible exceptions have been taken into consideration.

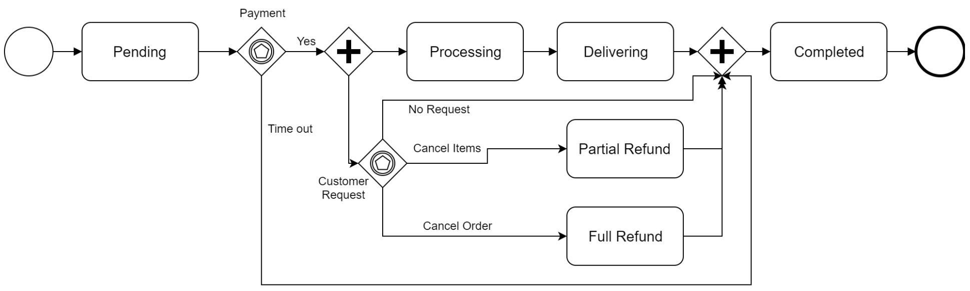


FIGURE 1.2: The updated BPMN for the online retailer example.

In practice, these order processing steps can be extremely complex, making the design of the model challenging. The model's complexity is also dependent on the level of detail and the manner in which each step is defined. On the basis of workflow patterns, it is proposed to construct the model using system event logs and process discovery techniques [2].

Even with a well-defined process model, humans still make mistakes, which is another obstacle. Human error is not entirely preventable, and sometimes people simply disregard the defined model; as a result, the process definition is permitted to evolve during practice, with certain constraints. The changes in the mined process model are discussed in [3] using various criteria. Techniques that generate the process models from event logs and update the pre-mined models with unforeseen traces are based on process discovery. Process discovery is a crucial aspect of process mining [4], and we will delve into the specifics in Chapter 2.

1.3 Intrusion Detection

1.3.1 Cybersecurity Concerns

The Cisco Virtual Network Index [5] estimates that the global Internet traffic has increased from 73.1 Exabytes per month in 2016 to approximately 235.7 Exabytes per month in 2021. This is due to the fact that the amount of data that is transferred through network cables has significantly increased each year. This equates to an increase of over 220 per cent in just five years or a growth rate of 26 per cent on an annual basis. There has also been a considerable amount of horizontal growth. A great number of services, such as finance, streaming, online gaming, and Internet video, have shifted from using conventional methods to utilising the Internet. Here, horizontal growth refers to the growth of the type of Internet services, whereas vertical growth refers to the data volume growth.

Cybersecurity becomes a significant concern as the value of information transferred online. For instance, the emerging threat exploits system vulnerabilities to steal valuable data. An estimated 33 billion personal records will be stolen in 2023, containing information such as addresses, credit card numbers, and identities [6]. In addition to the emerging threat, denial of service (DoS) attacks are a major issue-causing threat. Although this type of attack does not cause direct property damage, it renders services inoperable, resulting in a loss of revenue. Since 2016, the number of distributed denial of service (DDoS) attacks exceeding 1Gbps has increased by 150 per cent, as reported by [5] and [7]. In the fourth quarter of 2021, Cloudflare mitigated dozens of terabit-scale DDoS attacks. A few-minute DDoS attack has the potential to cause losses in the millions of dollars.

1.3.2 Intrusion Detection Systems

Utilizing an intrusion detection system, also known as an IDS, allows for the identification and classification of policy violations as well as attacks against the system's security. There are two main types of IDS: the network-based intrusion detection system (NIDS) and the host-based intrusion detection system (HIDS), and their distinction comes down to the initial purpose of the IDS.

The NIDS is typically installed on infrastructures such as routers and switches to monitor network activity and detect breaches. The NIDS solution is designed to monitor the entire network. It has access to all network traffic and takes decisions depending on the contents of packets or other analytical data. This broader perspective gives additional context for intrusion detection and the capability to detect widespread threats, yet, these systems lack access to the inner workings of the endpoints they defend. On the other hand, the HIDS is installed on a specific endpoint and is intended to defend it from both internal and external threats. It looks for file alterations, unusual network traffic, and any other suspicious behaviour on each unique system. A good example of HIDS is anti-virus software. This thesis will mostly examine network-based intrusion detection systems for detecting cyber attacks, such as DoS, botnets, and port scans.

Intrusion detection systems can also be categorised into two different types depending on the way they detect threats. These two types of IDSs are signature-based intrusion detection systems and anomaly-based intrusion detection systems. The signature-based IDS uses patterns to detect the intrusion or the machine learning algorithms that are trained with labelled attack data and used to classify whether there is an intrusion or not. An example of traditional signature-based well-known IDS is Snort [8], which detects intrusions based on predefined rules. The anomaly-based intrusion detection methods measure the similarity of giving activities to the known normal activities. The data can be identified as intrusions if the difference is larger than a threshold. Unlike signature-based intrusion detection systems, we are unaware of any anomaly-based intrusion detection systems running in a production environment or any predefined rules for the Snort framework that describes normal network behaviour. Nonetheless, the paper [9] introduces a variety of anomaly-based IDS techniques.

1.4 Problem Domains

There are a large number of existing techniques used in intrusion detection. Such as approaches that use data mining [10]–[12], machine learning [13]–[17] and other statistical methods [18]–[20] for the designing of intrusion detection systems. This section will discuss the problems that have been discovered in the area of IDS.

1.4.1 Level of Detection

Online or offline detection of network intrusion is available. Online detection monitors network activity in real-time in order to detect threats as quickly as possible. Offline detection typically examines the data logged and is executed manually by the administrator or at a predetermined interval. When discussing methods for detecting online intrusions, it is for this reason that we take into account detection at the packet level. A packet-level IDS analyse each incoming packet and will detect the threat as soon as a pattern appears; therefore, it can detect the intrusion before the connection terminates.

A large number of methods that employ commonly used datasets, regardless of whether they employ data mining or machine learning, are unable to detect intrusions at the packet level. It would appear that the datasets, like the well-known KDDCup'99 [21] and NSL-KDD [22] datasets, do not provide packet-level data that can be accessed in a straightforward manner. The NSL-KDD dataset is an improved version of the KDDCup'99 dataset, with enhancements such as the removal of duplicate records and the adjustment of the sizes of the train and test sets, respectively. The statistical values that are provided by these datasets are at the flow level. This means the data will not be generated until a connection is terminated or allowed to time out. For instance, the feature "Destination Bytes" indicates the total number of bytes transferred from the source to the destination in a single connection; obviously, this feature cannot be extracted before the connection closes or terminates. [23] provides specific information on all of the features that are a part of the KDD dataset. The concept of network flows will be introduced in Chapter 2.

1.4.2 Data Encryption

Some datasets include binary packet data in PCAP format in addition to feature sets that are formatted in a similar fashion to KDD datasets and are presented in CSV datasheet format. The CIC-IDS2017 dataset [24] and the CSE-CIC-IDS2018 dataset [25] are two examples. The packet data can be extracted from the binary format of these datasets; however, the difficulty in applying the packet data for classification is that if the extracted feature is from a single most recent packet, the information that can be obtained is insufficient. That is because a single packet may not correlate to attacks, and the majority of packets have encrypted or obfuscated content.

Some feature generation methods that are used in available datasets, such as ISCX 2012 [26] and the already mentioned datasets above, are based on flow-level analytical data, as the manner in which packets have been sent within a flow and the amount of data they have transferred are irrelevant to whether packets are encrypted or not. This is one of the solutions for intrusion detection with encrypted data.

1.4.3 Computational Cost

Packet-level detection is not always superior. As just discussed, techniques such as deep packet inspection (DPI) [27] that look for the contents or headers of packets do not work well if the packets are encrypted or obfuscated. Detection at the packet level also requires more resources than other methods. For example, some techniques use recurrent neural networks (RNN) as the detector by providing each packet as a time step [28]. If the time-series approaches are used, the dimension of the feature data may pose a problem with efficiency. Here is a comparison table (Table 1.1) that is obtained from [29]. The host-pair-level detection relies on the statistical data of all flows between two hosts at a given time, whereas flow-level detection is dependent on data generated from individual flows.

Levels	Pros	Cons
Packet-level	1. Real-time detection. 2. Allow pattern matching in payload content. 3. More information is extractable.	1. Slowest and not suitable for high-bandwidth networks. 2. Hard to apply on the encrypted payload.
Flow-level	1. Independent of the encrypted payload. 2. Faster than packet-level detection and are able to handle high-bandwidth networks. 3. Well-understood feature extraction methods	1. Some delay in detection. 2. Not the fasted way.
Host-pair-level	1. Provide global information on traffic between two hosts. 2. Good detection performance when used with other levels of data	1. Greater delay in detection.

TABLE 1.1: Pros & Cons for different levels of detection.

In general, detecting data at a higher level requires fewer computational resources because the total amount of data is smaller. Consequently, it is necessary to identify an algorithm that can be applied to packet-level detection and is efficient enough to be implemented onto consumer hardware. Additionally, we should attempt to address the concerns listed in Table 1.1.

1.4.4 Extensibility

When constructing something, one may consider whether or not it can be improved in the future. A similar principle can be applied to the design of intrusion detection systems, and it is anticipated that the system can be expanded or that additional research can be conducted on it. The majority of data mining and machine learning techniques employ similar extracted features and machine learning algorithms with modifications. The feature extractor may not be sufficiently general for use in other domains. CICFlowMeter [30] generates features based on properties such as the total number of forward packets, the total number of bytes transferred, and the number of particular flags in a flow, among others. This feature generator is inapplicable to HIDS that monitor system calls.

1.4.5 Concurrency Discovery Capability

Clearly, there exist methods that employ packet-level detection and are capable of handling encrypted data. For example, existing pattern-based IDSs examine audit trails for network data [8], [31]. In general, this type of IDS searches for specific activities or a sequence of activities, and then a determination is made based on the defined patterns.

Snort [8] employs a rule-based system that enables the creation of user-defined rules. Several options can be utilised during the development of rules, and some of them are listed below:

- content: Search the packet payload for the specified pattern.
- flags: Test the TCP flags for specified settings.
- ttl: Check the IP header's time-to-live (TTL) field.
- minfrag: Set the threshold value for IP fragment size.
- ack: Look for a specific TCP header acknowledgement number.
- msg: Sets the message to be sent when a packet generates an event.

A simple rule below will check all inbound traffic to the 10.10.1.0 subnet, if the payload of the content matches "/cgi-bin/phf", a message of "PHF probe!" will be sent. This rule does not appear to work on encrypted data; however, other attributes, such as flags and TTL, can be used to detect intrusions in encrypted data.

```
alert tcp any any -> 10.10.1.0/24 80 (content: "/cgi-bin/phf"; \
msg: "PHF probe!";)
```

The Variable-Length Audit Trail Pattern method proposed by Wespi et al. [31] is designed for HIDS. It translates system commands such as file-open and file-close into sequences of characters using the translation table. The translation table is essentially a mapping between system call names and specific keys. Using the Teiresias algorithm [32], a sequence of keys will be separated into subsequences of variable length. Comparing the subsequences to the training sequences in order to compute the boundary coverage. The algorithm will then identify the intrusion with the longest sequence of undetected events. Utilizing this algorithm with a newly designed translation table, it is possible to map packet headers to sequences of keys for NIDS; thus, the same technique can be extended to NIDS.

In [17], the damped incremental statistics is used to generate feature vectors at the packet level in real-time. This technique has been further explored in [33] at a later time with a different deep-learning model. The new value can be updated incrementally with the new packet data by keeping the previous statistical value. The previous values are also updated with the decay function, so older data have less weight. By implementing the 2-D statistics, statistical information between RX and TX is also extracted. Here, TX and RX are abbreviations for Transmit and Receive. The algorithm from [17] has a reasonable computational complexity so that the performance is not a key issue. By retaining statistical data, this algorithm is able to function with encrypted traffic.

There also exist numerous studies that employ audit trails for intrusion detection. Unfortunately, the majority of them are limited to a single network flow and disregard the global process structure. Therefore, they are unable to interpret parallelism information, which is crucial for detecting DDoS and botnet attacks. The global process structure, also called the global flow structure in this thesis, is the characteristic of all network flows observed on a cable, whereas the individual flow's characteristics are not disregarded. A further issue is that some of them do not work well with encrypted data. Data encryption is used extensively in modern Internet traffic. In addition, the majority of these techniques were published in the early 2000s or even the previous century, and there is a dearth of new research utilising these techniques.

1.5 Motivation

According to the study of existing intrusion detection methods mentioned in 1.4, three main issues are observed with some of the techniques used for intrusion detection. The first is the lack of ability to perform online detection efficiently; the second is that some packet-level detection techniques have difficulty applying to encrypted data; third, most techniques lack the ability to detect the global process structures. The presumption is to employ a technique that has not been widely explored in the field of NIDS, preferably one suitable for identifying the global process structure. In addition, we seek a method for extracting features that are not necessarily limited to numeric values; for instance, we can generate features based on relations or patterns across packets, which are not limited to a single packet or a single flow but also across different flows.

Process mining is a great technique that encodes global process structure and considers precedence relationships in model generation instead of numerical values. Also, considering the lack of existing research which utilises process mining for intrusion, process mining has the potential to be the alternative approach to intrusion detection. However, process mining is not designed to perform online execution. The motivation of this research is to cooperate with process mining and intrusion detection to test the performance of process mining-based intrusion detection and to modify process mining to perform online feature generation. This topic will be discussed in depth in Chapter 4.

1.6 Research Question

At the beginning of this research, there does not exist literature that covers the experiments of applying process mining to network intrusion detection. Due to the fact that this area is rarely explored, the following questions can be asked to aid the research.

1. How may process mining be applied to network intrusion detection?
2. How does process mining perform for network traffic data in anomaly detection?
3. How to execute process mining online, and can the technique be generalised?
4. How do the generalised algorithms function in different contexts, such as host-level intrusion detection?

1.7 Contributions

Some of the contributions of this thesis and the development of the process mining feature generation technique are listed below.

1. **Standardised method for transforming network packet data to event logs.**
   To combine process mining techniques with an intrusion detection system, it is necessary to convert network flows to event logs. The network packets are filtered, and only the required fields are exported and treated as attributes.
2. **Verification of the feasibility of process mining on network packet data.**
   Different process mining algorithms has been tested with the event logs that are converted from network traffic data. The generated process models have been compared and verified. The result provides valuable information in the development of the novel online feature generation algorithm.
3. **A novel algorithm for generating features for network intrusion detection systems.**
   A novel algorithm based on process mining is developed and evaluated for generating online features. The algorithm is initially evaluated using converted network data event logs. Then it is generalised and tested using the event log generated by the kernel call at the host level.
4. **A open-source Python package based on the feature generator.**
   The novel feature generator is implemented as a python package. The program reads the event logs and produce data for machine learning. This package aids in the future research and development of the feature generator.

1.8 Publications

1. **Zhong, Yinzheng, John Y. Goulermas, and Alexei Lisitsa. "Process Mining Algorithm for Online Intrusion Detection System." In proceeding of the International Conference on Software Testing, Machine Learning and Complex Process Analysis (2021)**
   In this paper, the technique of process mining in intrusion detection is presented. A novel process mining-inspired algorithm is proposed to be used to preprocess data in intrusion detection systems (IDS). The algorithm is designed to process the network packet data, and it works well in online mode for online intrusion detection. To test this algorithm, the CSE-CIC-IDS2018 dataset is used, which contains several common attacks. The packet data was

preprocessed with this algorithm and then fed into the detectors. The experiments use the algorithm with different machine learning (ML) models as classifiers to verify that the algorithm works as expected is reported. The content of this paper is presented in Chapter 4.

2. **Zhong, Yinzheng, and Alexei Lisitsa. "Can process mining help in anomaly-based intrusion detection?." arXiv preprint arXiv:2206.10379 (2022)**
   In this paper, the naive applications of process mining in network traffic comprehension, traffic anomaly detection, and intrusion detection are considered. The procedure of transforming packet data into an event log was standardised. Multiple process models are mined, and the process models mined with the inductive miner using ProM and the fuzzy miner using Disco are analysed. These two types of process models extracted from event logs of differing sizes are compared. The content of this paper is presented in Chapter 2 and Chapter 3.

3. **Zhong, Yinzheng, and Alexei Lisitsa. "Online Transition-Based Feature Generation for Anomaly Detection in Concurrent Data Streams." 9th International Conference on Information Systems Security and Privacy - ICISSP (2023), ISBN 978-989-758-624-8; ISSN 2184-4356, pages 576-582**
   In this paper, the transition-based feature generator (TFGen) technique is introduced, which reads general activity data with attributes and generates step-by-step generated data. The activity data may consist of network activity from packets, system calls from processes or classified activity from surveillance cameras. TFGen processes data online and will generate data with encoded historical data for each incoming activity with high computational efficiency. The input activities may concurrently originate from distinct traces or channels. The technique aims to address issues such as domain-independent applicability, the ability to discover global process structures, the encoding of time-series data, and online processing capability. The content of this paper is presented in Chapter 5.

1.9 Thesis Organisation

In Chapter 2, the detailed concepts of process mining and transmission control protocols will be discussed. These concepts are required to proceed to the next step, which is the application of process mining to network traffic data. The process mining techniques for intrusion detection systems will be examined in general, and the method of converting network traffic data to event logs will be proposed.

In Chapter 3, state diagrams are compared to process models that have been mined using different process mining algorithms. The result of the conformance checking based on process models and anomalous traces is not promising. However, there is a detailed explanation of why the naive approach of applying process mining to network data is not effective.

In Chapter 4, an algorithm for online process mining that is based on observations and studies from earlier chapters is proposed. This chapter begins by introducing the issues with existing network intrusion detection approaches and process mining, as well as the techniques used to address these problems. Then, the result using multiple machine learning techniques was verified and compared to a commonly used existing feature generator.

In the 5th and final chapter, the feature generator in a generic form is presented, including the development of a Python package that aids future research. Based on the fact that any standard event log can serve as the input of the technique, it is explored further with a host-level dataset, demonstrating that it has the potential to function with host-level data as well.

Chapter 2

Process Mining on Network Traffic Data

2.1 Overview

Following a naive approach to applying process mining to IDS, this chapter discusses which process models can be mined using process mining from the network traffic data. As a summary of the previous chapter, the observed problems are listed as well as the expectations for new feature generation techniques that solve the related problems from the problem domain section (Section 1.4) are shown in Table 2.1. In general, an algorithm with the characteristics listed in Table 2.1 is expected. The first two of these capabilities are provided by process mining. A large number of techniques have some of these capabilities, but not all of them.

No.	Capabilities	Importance
1	A generic approach.	Can be applied on different areas.
2	Discovering the global structure of the network traffic.	Discovers the concurrency from the data stream and may benefit DDoS detection.
3	Extract features at the packet level.	Needed for real-time detection.
4	Encoding the historical information.	For handling encrypted data and the behaviour of an entire flow.
5	Having a reasonable computation complexity.	Better performance for packet-level detections.

TABLE 2.1: Capabilities required for our algorithm.

The belief that process mining can be used for intrusion detection is derived from [34]. Similar concepts are also discussed in [35], [36]. According to [34], it is possible to discover process models, after which conformance checking can be used to identify anomalies. However, there is a lack of further research based on [34]–[36]. The intuition is that process mining is capable of identifying features of the global structure and can therefore detect parallelism patterns. Even though process mining is incapable of packet-level extraction, the experiment can still be conducted at the flow level. It is a relatively unexplored field, so understanding how process mining operates on network data is essential for the development of new techniques. The naive strategy will be based on flow-level detection, which is simple because each flow can be converted into a case in process mining. The efficiency of process mining and the challenges associated with its use for network intrusion detection needs to be evaluated.

The early experiments are concentrated on the transmission control protocol (TCP) because it is more prevalent in datasets and in general TCP is the most used protocol. Additionally, TCP packets within a flow adhere to a sequential order so that the flow can be interpreted as a process. TCP and process mining concepts pertinent to this thesis are introduced first, followed by a discussion of related works that employ process mining for anomaly detection in various fields. The method for converting network data to event logs is then standardised. Then, two well-known process mining algorithms, the inductive mining algorithm and the fuzzy mining algorithm are employed to mine process models from processed event logs.

2.2 Process Mining

2.2.1 Event Log

An event log is often known in similar forms such as workflow log, transaction log, audit trail, etc. [37]. An event log contains historical process executions that were recorded as streams of events. Each event refers to an activity that has been executed by any entity such as a system, personnel or other resources at some timestamp. Table 2.2 shows an example of a fraction of the event log that comes from the order processing logs of the online grocery store we mentioned in Chapter 1.

Case ID	Event ID	Timestamp	Properties			
			Stage	Handling	Supervisor	...
1	1	2022-02-23T10:48:55	Pending	System	Anne	...
1	2	2022-02-23T10:49:15	Processing	James	Sara	...
2	3	2022-02-23T10:49:23	Pending	System	Sara	...
1	4	2022-02-23T11:12:24	Delivering	Alex	Sara	...
2	5	2022-02-23T11:14:41	Processing	George	Sara	...
3	6	2022-02-23T11:27:39	Pending	System	Sara	...
2	7	2022-02-23T11:30:17	Delivering	Alex	Anne	...
1	8	2022-02-23T11:36:07	Complete	System	Anne	...
2	9	2022-02-23T11:42:40	Complete	System	Anne	...
4	10	2022-02-23T11:43:06	Pending	Anne	Sara	...
3	11	2022-02-23T11:45:18	Processing	George	Sara	...
4	12	2022-02-23T11:49:52	Processing	James	Sara	...
4	13	2022-02-23T12:01:05	Delivering	Alex	Anne	...
...	...	...	...	...	...	...

TABLE 2.2: A fragment of the event log.

The event log can be divided into cases; from the table, one can see that cases 1 through 4 correspond to four distinct order-placing customers. Each case is comprised of multiple events, each of which has a number of properties. For instance, one of the properties of the events is the order stage, whose values include Pending, Processing, Delivering, and Complete, recalling the example in Figure 1.1. According to the last two properties, properties such as the personnel involved are also present. These properties are known as attributes. Note that events occur at different timestamps; therefore, each event is unique even if some of its attributes have identical values. Some observable facts are listed in Table 2.2.

- An event log consists of more than one case.
- A case consists of one or more events.
- Each event must relate to one case.
- Events may have properties.
- Events are arranged chronologically and that affects mining outcome.

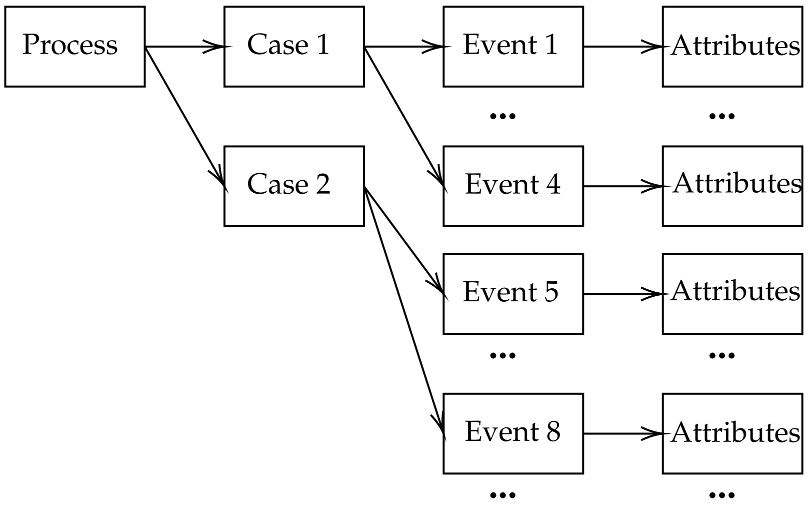


FIGURE 2.1: The structure of event logs.

Figure 2.1 illustrates the structure of an event log in a hierarchical manner. Based on the literature from [1], [38], [39], the formal definition of the event log will be examined now and the concept of traces and event classes will be introduced.

Definition 2.1 ($Seq()$)**.** *Giving a* set *A, $Seq(A)$ is the set of all finite sequences over A.*

For example, if $A = \{a, b, c\}$, $Seq(A) = \{\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle a, b\rangle, \langle b, a\rangle, \langle a, c\rangle, ..., \langle a, b, c\rangle, \langle a, c, b\rangle, \langle b, a, c\rangle ...\}$

Definition 2.2 (Event, case)**.** *Let $\mathcal{E}$ be the universe of* events*. A universe is a collection of all the entities to be considered in a given circumstance. A* case *$c \in Sec(\mathcal{E})$ is a* sequence *of events $e \in \mathcal{E}$. $\mathcal{C} = Sec(\mathcal{E})$ is the universe of cases.*

For example, $\langle a, b, c, d\rangle \in \mathcal{C}$ is a case that has 4 events.

Definition 2.3 (Trace)**.** *Each case $c \in \mathcal{C}$ has a respective* trace *$Trace(c) = t$ such that $t \in \mathcal{T}$ where $\mathcal{T}$ is the universe of traces, i.e., a case can be seen as an instance of a corresponding trace.*

Definition 2.4 (Event class)**.** *An* event class *ec maps one or more events to itself, i.e., an event is an instance of a corresponding event class. Let $EC(c)$ be the set of event classes observed in case c, and $ec(e)$ be the mapped event class of e where $e \in \mathcal{E}$, then $EC(t) = \{ec(e) | e \in Set(c)\}$.*

For example, the trace $\langle a, b, c, d, b, d\rangle$ has 6 events and 4 event classes a, b, c and d. i.e. $\mathcal{EC}(\langle a, b, c, d, b, d\rangle) = \{a, b, c, d\}$.

Definition 2.5 (Event log). *Let $\mathcal{L}$ be the universe of* event logs. *An* event log $L \in \mathcal{L}$ *is a finite multiset of observed traces.* $T(L)$ *is the set of observed traces in* L. *i.e.* $T(L) = \{t \in L\} \subseteq \mathcal{T}$. *On the other hand,* $\overline{T}(L)$ *is the unobserved traces in* L *such that* $\overline{T}(L) = \mathcal{T} - T(L)$. *Therefore, the observed event classes for* $\mathcal{L}$ *is* $\mathcal{EC}(\mathcal{L}) = \bigcup_{L \in \mathcal{L}} \mathcal{EC}(L)$ *according to Definition 2.4. Simply put, unobserved objects are potential objects that have not yet been seen and recorded, whereas observed objects are those that are present in the event log.*

According to the definition, the event log 2.1 can be demonstrated below. The superscripts are the number of cases derived from that trace (the base) or the number of instances of that trace.

$$L = \{\langle a,b,c\rangle^4, \langle b,c,d\rangle^3, \langle a,b,d\rangle^2, \langle b,c,d,e\rangle\} \tag{2.1}$$

From this event log, all properties about events, traces, event classes, cases, and event logs are listed here.

- **Traces**. We have 4 traces in total, i.e., $t_1 = \langle a,b,c\rangle$, $t_2 = \langle b,c,d\rangle$, $t_3 = \langle a,b,d\rangle$, and $t_4 = \langle b,c,d,e\rangle$.
- **Cases**. Each case has its respective trace. In this example, we have four cases of the first trace t_1, three cases of the second trace t_2, two cases of trace t_3 and 1 case of t_4; therefore, the number of cases in L is 10.
- **Event Classes**. Five event classes appears in L, where $EC(L) = \{a,b,c,d,e\}$ and $|EC(L)| = 5$.
- **Events**. Each event has its respective event class. As timestamp is not present in L, we assume the case will only start when the previous case finishes; hence, we rewrite the event log L in 2.2 by expending L, and we get 31 *events*.

$$\begin{aligned} L =&\langle a,b,c,a,b,c,a,b,c,a,b,c,b,c,d,b, \\ &c,d,b,c,d,a,b,d,a,b,d,b,c,d,e\rangle \end{aligned} \tag{2.2}$$

Some literature, such as [1], defines a trace as a sequence of activities rather than a sequence of event classes. Since the distinction between events and activities is unclear, so are the definitions of traces and cases. Therefore, the activity will not be defined in this thesis and only definitions 2.1 - 2.5 in the subsequent context are referred to be readers. These concepts may be confusing so here are more examples. A trace is, for a real-life instance, a predefined procedure for producing a toy. Numerous toys can be manufactured, so this process will be repeated numerous times. Each instance of the execution is a case. The trace consists of a series of event classes (event names or action names) and each execution of these classes is an event. An event log is essentially a finite set of different cases (executed traces).

2.2.2 Inductive Miner

The inductive miner (IM) [40]–[42] is a popular process mining algorithm that divides the event log into smaller sub-logs using the divide-and-conquer method. The inductive miner can detect alternatives, concurrency, and loops in event logs and much broader classes than the α-algorithm [43] can. Typically, IM generates process models in the form of workflow nets (WF-nets), i.e. Petri nets [44] with a single beginning and ending point. α-algorithm [43] is the first process discovering algorithm developed, which is limited to detecting sequential and concurrent patterns based on the directly-follows graph (DFG). The DFG will be introduced in this section later.

The algorithm of the inductive miner is demonstrated here. In order to produce the process model, the inductive miner will first discover the process tree, which consists of actions and operators $\oplus$. The WF-nets consist of the nodes of event classes and the silent action $\tau \notin EC$. Note that τ is a different symbol from the set of traces T and the trace universe $\mathcal{T}$. The operators connect sub-trees, and any process tree can be converted to equivalent WF-nets. There are four different operators,

- $\rightarrow$ denotes the sequential operator;
- $\wedge$ denotes the concurrent operator;
- $\times$ denotes the exclusive-selection operator;
- $\circlearrowleft$ denotes the loop operator.

Definition 2.6 (Process tree). *Let $\bigoplus = \{\rightarrow, \wedge, \times, \circlearrowleft\}$ be the set of operators and $\oplus \in \bigoplus$. Let $EC \subseteq \mathcal{EC}$ be a finite set of event classes where $\tau \notin EC$. Let Q be the process tree. Each operator allows the connection of multiple sub-trees.*

1. *If $a \in EC \cup \{\tau\}$, then $Q = a$ is a process tree.*
2. *If $Q_1, Q_2, ..., Q_{n-1}, Q_n$ $(n \geqq 1)$ are process trees and $\oplus = \{\rightarrow, \wedge, \times\}$, then $Q = \oplus(Q_1, Q_2, ..., Q_{n-1}, Q_n)$ is a process tree.*
3. *If $Q_1, Q_2, ..., Q_{n-1}, Q_n$ $(n \geqq 2)$ are process trees and $\oplus = \{\rightarrow, \wedge, \times \circlearrowleft\}$, then $Q = \oplus(Q_1, Q_2, ..., Q_{n-1}, Q_n)$ is a process tree.*

Basically, IM uses the directly-follows graph [45], and other components such as the dependency and frequency measurement components. The definition shows that the $\circlearrowleft$ operator requires at least two sub-trees.

Let us look at the trace $t = \langle a, b, c, b, a\rangle$ as an example. For better generality, it is assumed that each trace starts with a start event class ec^{start} and an end event class ec^{end}. Therefore, $t = \langle ec^{start}, a, b, c, b, a, ec^{end}\rangle$. ec^{start} is directly followed by a and a is directly followed by b. These relations are denoted as $ec^{start} \mapsto a$ and $a \mapsto b$ etc.

Definition 2.7 (Directly-follows graph). *Let L be an event log, the directly-follows graph of L is $G(L) = (EC(L), \mapsto_L, ec_L^{start}, ec_L^{end})$ where $EC(L)$ is the set of observed* event classes *in L.*

$$L = \{\langle a, b, c, d, e, g\rangle, \langle a, b, c, d, f, g\rangle, \langle a, c, d, b, f, g\rangle, \langle a, b, d, c, e, g\rangle, \langle a, d, c, b, f, g\rangle\} \tag{2.3}$$

Equation 2.3 is an event log that has 5 traces, 7 event classes (a, b, c, d, e, f, g) and 25 events. Based on this event log, the DFG is generated in Figure 2.2.

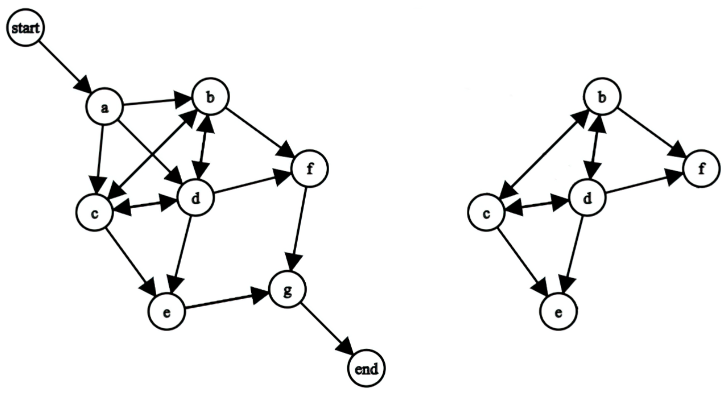


FIGURE 2.2: The DFG for event log 2.3

The left graph in Figure 2.2 is the initial graph. By ignoring ec_L^{start} and ec_L^{end} for now, the first event of all cases is a and the last event of all cases is g; therefore, event classes a and g form a sequential pattern (i.e. we must start with a and end with g). The IM recursively cut the DFG into sub-graphs, so in this case, a smaller graph by cutting off a and g is created. It forms the graph on the right as the result. Nodes b, c and d form the concurrent pattern, i.e., one can choose to go either b, c or d after a and without any constraint of the order. For example, $\langle a, b, c, d \rangle$ or $\langle a, c, d, b \rangle$. After the concurrent pattern, one can either choose to go to node f or e. Therefore, e and f can be cut off, and then the loop between c, b and b can be cut until no further cutting is possible. The last two steps are not shown in Figure 2.2.

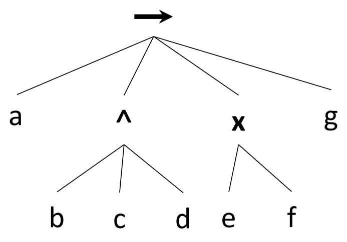


FIGURE 2.3: The process tree for event log 2.3

After the cutting process, the process tree can be created in Figure 2.3. This process tree consists of three operators: sequential, concurrent, and exclusive-selection operators. The concurrent operator connects b, c and d which are single-node subtrees; the exclusive-selection operator connects e and f and finally, the sequential operator connects all sub-trees. The equivalent process model is shown in Figure 2.4.

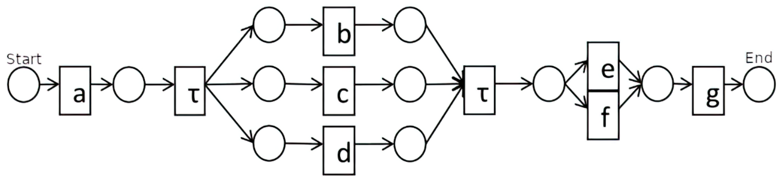


FIGURE 2.4: The WF-net for event log 2.3

Note in Fig. 2.4, τ has been used to connect the concurrent sub-tree with other trees because the concurrent tree cannot be directly connected to the exclusive-selection tree; therefore, τ actions are attached to both sides of the concurrent pattern. The inductive miner can generate a robust model from noisy and incomplete data, typically in the form of workflow nets, BPMN models, event-driven process chains (EPCs), or Yet Another Workflow Language (YAWL) models.

2.2.3 Fuzzy Miner

In the experiments, the fuzzy mining (FM) [46] process mining algorithm is also employed. By utilising the adaptive approach, which is a higher-level abstracted and aggregated model, the FM is able to solve process models that resemble a pile of spaghetti. Abstraction and aggregation are both made possible in FM by taking into account the significance and correlation of the various event classes. Maps intuitively serve as an example of such abstraction. A comparable illustration with maps of the University of Liverpool campus (Figure 2.5) is provided.

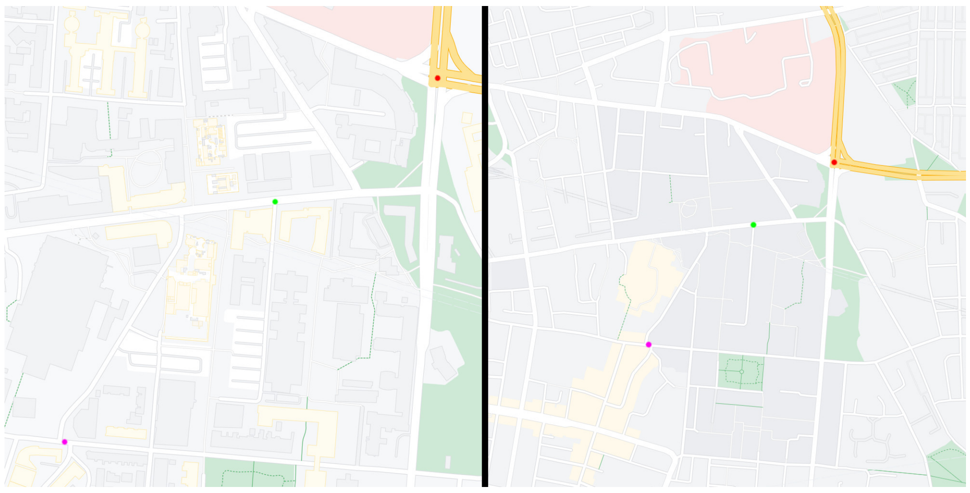


FIGURE 2.5: The campus maps. The map on the left has more details but less abstraction. The map on the right has better abstraction but fewer details.

These two maps allow the identification of layers such as buildings, parks, and roads. Depending on their functions, the buildings are typically painted in light yellow, pink, or grey. The parks are coloured green, whereas the roads are orange. Three identical points were marked on both sides of the map using three different colours to facilitate a comparison between them.

- **Aggregation:** The left-side map provides a detailed view of the shape of each individual building, while the right-side map displays a large grey area between the red and purple markers that denotes a cluster of campus buildings. This is called aggregation, and the purpose of such is to limit the amount of information displayed.
- **Abstraction:** Smaller roads are visible on the left-side map but are absent from the right-side map. For example, there are some roads that are parallel to each other between the purple and green markers on the left map; however, these roads cannot be seen on the right side. This is called abstraction, where insignificant information gets omitted.

- **Emphasis:** For better visualisation, objects on maps are highlighted. Buildings are coloured differently, and parks are coloured green; also highways are coloured orange to distinguish them from minor roads.

- **Customization:** Various types of maps serve distinct purposes. In addition to the maps depicted in Figure 2.5, topographic maps display the elevation of terrains, while geological maps illustrate the various types of rocks and materials in a specific area.

The processes of map generalisation and selection utilise the concepts discussed previously [47], [48]. There is no implication that spaghetti models are inadequate in every circumstance. Typically, these models contain additional information regarding the process they represent. When there is no need for extremely detailed data and a quick understanding of information is desired, it is beneficial to decrease the amount of data incorporated in models. Let us return for a moment to fuzzy mining and discuss the concepts of significance and correlation metrics used in fuzzy mining.

The *significance* is employed for abstracting event classes and binary precedence relations. On a graph, the event classes represent the nodes and the binary precedence relation represents the edges. During abstraction, insignificant nodes and edges are expected to be filtered out. Calculating the frequency of certain events or relationships is one example of measuring significance.

Correlation in FM only applies to binary precedence relations. It indicates the similarity between two event classes which share such a precedence relation. This can be done by comparing the attributes or event names between two relevant events. For example, there are three events that have three attributes each where precedence relations exist between any two of them. The first event is $(\mathcal{A}, \mathbb{B}, \mathsf{C})$, the second one is $(\mathcal{A}, \mathbb{B}, \mathsf{A})$ and the third event is $(\mathcal{D}, \mathbb{E}, \mathsf{F})$ where $\{\mathcal{A}, \dots, \mathcal{F}\}$, $\{\mathbb{A}, \dots, \mathbb{F}\}$ and $\{\mathsf{A}, \dots, \mathsf{F}\}$ are attributes. Typically, correlation is measured by counting the number of identical attributes. For instance, the correlation between the first and second events is 2; the correlation between the first and third events is 1. The first and second events exhibit a stronger correlation compared to the first and third events as they possess similar attributes. Higher correlation denotes higher importance of nodes.

One of the scenarios for the abstraction is that nodes with high significance will be preserved, nodes with low significance but high correlation will be aggregated, and nodes with low significance and correlation will be eliminated from the abstract model.

2.3 Transmission Control Protocol

The transmission control protocol, whose initial concept was introduced in [49], is a robust network communication protocol that is widely employed for Internet communication. The TCP segment is a layered protocol that runs on the Internet Protocol (IP) [50], and then applications at a higher level run on the TCP layer. Consequently, TCP is frequently referred to as TCP/IP. The RFC 793 document outlined the TCP implementation for IPv4 [51]. TCP is a connection-oriented protocol that transmits data over an IP network, and the transmitted data are ordered and include a checksum for error detection. Before data can be transmitted, the TCP connection must be established through a three-way handshake (or a four-way handshake for simultaneous opening). TCP facilities the following.

- **Basic Data Transfer**
  TCP is able to transmit streams of octets (bytes) between hosts in both ways where a certain number of octets are contained in a segment.

- **Reliability**
  Due to the lossy nature of the communication network, data transmission on the network is not reliable. TCP is capable of handling damaged data, data that was sent out of order, duplicate data and lost data. These are archived by using checksum, sequence ID and timestamps.

- **Flow Control**
  By indicating the acceptable data amount (window size) in ACK packets, the receiver has the ability to exercise control over the number of octets that are sent from the sender.

- **Multiplexing**
  TCP makes it possible for multiple connections to be established simultaneously between multiple hosts as well as between two hosts that share a single physical connection. Each host possesses a one-of-a-kind IP address as well as several ports.

- **Connection**
  An individual connection is denoted by its information, which includes sockets, sequence ID numbers, and flow control. In order to begin the data transmission, the connection must first be established.

- **Precedence & Security**
  At the application level, the precedence and security measures can be implemented in a variety of ways, including the utilisation of encryption on the payload, the implementation of additional error correction mechanisms, and the utilisation of higher-level protocols.

2.3.1 Packet Format

TCP/IP packets transmit via IP and consist of an IP header and an IP data segment. The TCP segment is the IP data, which is further subdivided into the TCP header and the payload data (Figure 2.6). Due to the fact that each TCP header contains 20 bytes of information, followed by a variable length of options that can range anywhere from 0 to 40 bytes, the total size of a TCP header can range anywhere from 20 bytes all the way up to 60 bytes. Both hosts have the ability to send a packet even if the options field is empty because it is not required. Table 2.3 provides an illustration of the format of the TCP header field. The table is organised with four primary columns, each of which represents an octet. These primary columns have been further subdivided into eight bits each.

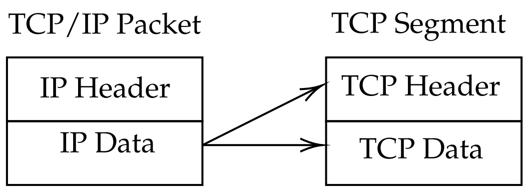


FIGURE 2.6: The TCP/IP packet.

	Octets	0								1								2								3							
Octets	Bits	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved bits			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (up to 40 bytes)																															
...	...																																
60	480																																

TABLE 2.3: The structure of a TCP header.

- **Source & Destination Ports**
  The port number is represented as a 16-bit binary value, which can take on any value between 0 and 65535.

- **Sequence Number**
  The sequence number allows one to determine which of the packets has already been received. This number is also helpful for identifying the packet that is missing or for reordering the packets that are out of order.

- **Acknowledgment Number**
  The ACK flag typically accompanies the acknowledgement number. In an ongoing communication, the receiver will set the number to indicate the expected arrival of the next packet. For instance, if a packet is received by the receiver with the sequence number set to 10, the receiver will respond with an acknowledgement number 11 along with the ACK flag set.

- **Data Offset**
  The total size of the header is indicated by the data offset using four bits, and it is expressed in terms of the size of words. A TCP header must have a minimum size of 20 bytes, which is equivalent to five words; in this circumstance, the data offset is set to 5. The number can go as high as 15, which is the maximum amount of data that the header can store (60 bytes). The data will be padded with 0-bits if necessary, where normally the options filed is padded to the 32-bit (4 byte) boundary.

- **Flags**
  Each TCP segment has 12 bits of flag information. The first 3 bits are reserved bits, and the rest 9 bits encode the status of 9 flags.
  NS flag is the nonce sum of ECN-Nonce. ECN stands for Explicit Congestion Notification, which is introduced in RFC 3168 document [52]. The CWR is the Congestion Window Reduced, and ECE is the ECN-echo flag.
  URG is set if data is present in the urgent pointer field of the header. ACK indicates an ongoing conversation when it is set, and the number in the acknowledgement number field is the expected number for the next packet.
  PSH is set by the sender, and the receiver is asked to process data from the buffer immediately even if the buffer is not filled.
  The RST is set when the connection must be reset due to any reason that makes the connection invalid.
  The synchronise flag (SYN) is used in the three-way handshake to synchronise the acknowledgement number, which we will look at later.

FIN is set when the packet is the last packet from the sender, i.e., the connection should be finished.

- **Window Size**
  A 16-bit binary value is stored in the window size field and is used for flow control. It gives an indication of how much free space there is in the receiver's buffers. This field will be communicated to the sender so that it can make any necessary adjustments to the rate at which it transmits packets to the receiver. If the window size is 0, the sender should refrain from sending any more packets.

- **Checksum**
  The 16-bit checksum is utilised to determine if the TCP header, payload, or IP header data contains an error.

- **Urgent Pointer**
  The urgent pointer field specifies the location of the urgent data within the data field. The sender is responsible for setting the urgent pointer, which is valid when the URG flag is set.

- **Options**
  The options field is a variable-length field that stores the data type and application payload. The data offset field determines the range of 0 to 40 bytes for the size of the payload. This field consists of three sections: the first octet, Option-kind, indicates the type of data; the second octet, Option-length, indicates the length of the options, including the lengths of the first two octets; and the final section, Option-data, contains the option's data. As padding is used, the length indication in Option-length could be shorter than the indication in the data offset field. Options are optional in the segment, but if they are present, Option-kind must be present and Option-presence length's depends on Option-kind.

2.3.2 TCP States

Prior to establishing a connection, the local user will listen on the port for any incoming connection requests. Assuming the user listening on the port is on the server side, the client TCP will send the connection request to the server and initiate the three-way handshake to establish the connection. Knowing the IP addresses and ports from the IP header and TCP header enables packet transfer. The tuple of address and port from a host is referred to as a socket, and a unique connection can be established by pairing with another socket from another host; therefore, multiple connections are possible between two hosts by using different ports.

Once the connection has been established, data transmission will commence. When the transmission is complete and the FIN flag is set, or when the connection is interrupted, the connection will close. The transmission control block (TCB) is a data structure that stores information about local and remote ports, buffer size, sequence numbers, and other variables. Continuous communication is dependent on this data structure. Typically, a connection will pass through multiple states from the beginning to the end. These states are LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, and TIME-WAIT, with CLOSED not actually being part of the connection itself. Figure 2.7 depicts the state diagram from the RFC 793 document [51]. Also, additional information for each state is presented below.

```
                              +---------+ ---------\      active OPEN
                              |  CLOSED |            \    -----------
                              +---------+<---------\   \   create TCB
                                |     ^              \   \  snd SYN
                   passive OPEN |     |   CLOSE        \   \
                   ------------ |     | ----------       \   \
                    create TCB  |     | delete TCB         \   \
                                V     |                      \   \
                              +---------+            CLOSE    |    \
                              |  LISTEN |          ---------- |     |
                              +---------+          delete TCB |     |
                   rcv SYN      |     |     SEND              |     |
                  -----------   |     |    -------            |     V
 +---------+      snd SYN,ACK  /       \   snd SYN          +---------+
 |         |<-----------------           ------------------>|         |
 |   SYN   |                    rcv SYN                     |   SYN   |
 |   RCVD  |<-----------------------------------------------|   SENT  |
 |         |                    snd ACK                     |         |
 |         |------------------           -------------------|         |
 +---------+   rcv ACK of SYN  \       /  rcv SYN,ACK       +---------+
   |           --------------   |     |   -----------
   |                  x         |     |     snd ACK
   |                            V     V
   |  CLOSE                   +---------+
   | -------                  |  ESTAB  |
   | snd FIN                  +---------+
   |                   CLOSE    |     |    rcv FIN
   V                  -------   |     |    -------
 +---------+          snd FIN  /       \   snd ACK          +---------+
 |  FIN    |<-----------------           ------------------>|  CLOSE  |
 | WAIT-1  |------------------                              |   WAIT  |
 +---------+          rcv FIN  \                            +---------+
   | rcv ACK of FIN   -------   |                            CLOSE  |
   | --------------   snd ACK   |                           ------- |
   V        x                   V                           snd FIN V
 +---------+                  +---------+                   +---------+
 |FINWAIT-2|                  | CLOSING |                   | LAST-ACK|
 +---------+                  +---------+                   +---------+
   |                rcv ACK of FIN |                 rcv ACK of FIN |
   |  rcv FIN       -------------- |    Timeout=2MSL -------------- |
   |  -------              x       V    ------------        x       V
    \ snd ACK                 +---------+delete TCB         +---------+
     ------------------------>|TIME WAIT|------------------>| CLOSED  |
                              +---------+                   +---------+
```

FIGURE 2.7: The state transition diagram from the RFC 793 document [51].

- **LISTEN (Server)** - The user on the server-side requests to listen to a port for any incoming connection request (SYN) from remote hosts.
- **SYN-SENT (Client)** - The client sends connection requests (SYN) and waits for a matching connection request (SYN) and the confirmation (ACK).
- **SYN-RECEIVED (Server)** - Received the connection request from the client TCP, so the local TCP also transmitted a matching connection request (SYN) and a connection request ACK to confirm the connection.
- **ESTABLISHED (Both)** - The connection is established and confirmed. This is the data transmission state.
- **FIN-WAIT-1 (Either)** - First connection termination request (FIN) sent from *A* and awaiting confirmation (ACK) from *B*.
- **FIN-WAIT-2 (Either)** - Side *A* has received the confirmation (ACK) from *B* of its first FIN sent now side *A* waits for the second *B* from *B*. This state can be "skipped" so side *A* can go from FIN-WAIT-1 to TIME-WAIT if side *B* sends the second FIN and ACK of the first FIN at once.
- **CLOSE-WAIT (Either)** - Side *B* received the first connection termination request (FIN) from *A* and now waiting for a connection termination request from the local user.

- **TIME-WAIT (Either)** - Side A received the second FIN form B, so it sends out the last ACK. Now side A waits for enough time to pass to be sure side B received the acknowledgement of the ACK of its second FIN.
- **LAST-ACK (Either)** - Side B ends out the second FIN and now waiting for the last ACK from A.
- **CLOSING (Both)** - Simultaneous close on both sides.
- **CLOSED (Both)** - No connection.

2.3.3 Connection Establishment

The TCP connection is established using the three-way handshake, and data transmission cannot begin until the connection is established. The server will initially verify that the service is active and listening on a port. This is the stage of passive-opening. The client will send the initial packet containing the local and remote addresses in the IP header, as well as the destination port and local port that the server is using for this specific service. The SYN flag is set to 1 in the TCP header, indicating a connection request. The initial sequence number (ISN) is initialised to the "random" number $\#_{seq}(C_1)$, where C represents the client.

The server receives the request and responds with the next packet that has ACK and SYN set, together with a new "random" sequence number $\#_{seq}(S_1)$ and an acknowledgement number $\#_{ack}(S_1) = \#_{seq}(C_1) + 1$ that is set to the next expected sequence number from the client. Obviously, the source and destination addresses and ports are swapped in this packet.

The client receives the server's confirmation and sends back a packet with the ACK flag set to notify the server of the received confirmation; at this point, the connection is established. In addition, the client sets $\#_{seq}(C_2) = \#_{seq}(C_1) + 1$ for the sequence number and $\#_{ack}(C_1) = \#_{seq}(S_1) + 1$ for the acknowledgement number. Figure 2.8 depicts this process.

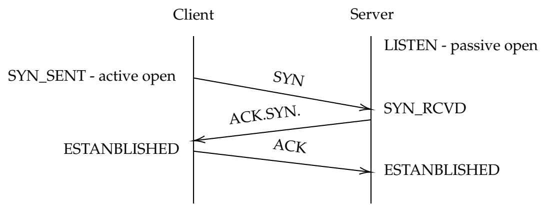


FIGURE 2.8: The three-way handshake.

The simultaneous open is a particular case of the connection establishment stage. Both hosts send connection requests simultaneously and return ACKs simultaneously, or we can say that the host sends the request before receiving the request from the other host. In this instance, both sides are active open.

This "random" ISN is not truly random; according to the RFC 793 document [51], the ISN is selected from a 32-bit counter whose value increases by one approximately every 4 microseconds. The purpose of this mechanism is to prevent confusion between a new packet and a delayed packet belonging to another connection. The full cycle of such a counter is approximately 4.55 hours. Additionally, SYN-marked packets carry the maximum segment size (MSS). The IP header and TCP header sizes are measured in MSS, and the greater the MSS, the greater the amount of data that can be transmitted in a single segment. The MSS field in SYN packets is used by both the client and server to declare the segment size they expect to receive. In the event that this option is missing, the default value of 536 bytes is assumed. The MSS can theoretically reach the maximum transmission unit (MTU) size [50], and a larger size may result in fragmentation.

2.3.4 Data Transmission

After the handshake, the connection continues based on the same principles of updating corresponding fields and enters the data transmission phase. Observing normal traffic patterns has revealed that the majority of bytes transferred through the cable are used for data transmission. The data is divided into smaller chunks that fit within an MSS-restricted segment. During data transmission, it is not necessary for recipients to acknowledge each and every packet they receive, and a single packet can sometimes be acknowledged twice. This may be associated with the network condition and host window sizes.

Assuming the client sends data to the server, Figure 2.9 illustrates typical data transmission procedures. The figure on the left depicts the client sending data to the server with packets 1 through 3 at the beginning, and the server acknowledging all three packets only once. Packet 4 precisely set the ACK number to $\#_{ack}(S_4) = \#_{seq}(C_3) + 1$. Occasionally, the server will acknowledge multiple consecutive packets, such as packets 8 and 9.

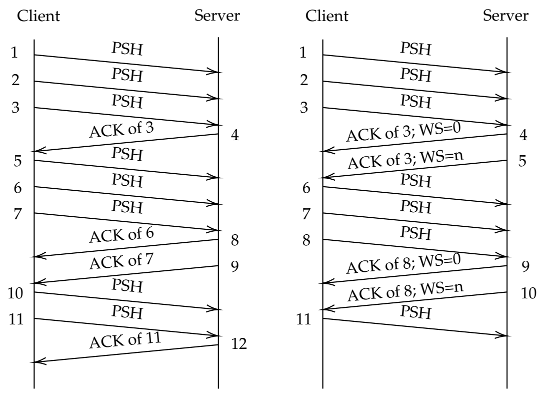


FIGURE 2.9: Data transmission.

Figure 2.9's right side depicts a faster client sending data to a slower server. Packet 4 informs the client via an ACK ($\#_{ack}(S_4) = \#_{seq}(C_3) + 1$) that its buffer is full with window size set to 0, so the sender stops PSH and awaits the next ACK. Later, packet 5 acknowledges the sender with the same acknowledgement number as packet 4, $\#_{ack}(S_5) = \#_{ack}(S_4)$, but with a larger window size than 0. After receiving this ACK, the sender resumes data transmission.

This sending process is being monitored using the sender's sliding window. The receiver and sender announce their window sizes in each packet they send; in this instance, the receiver's window size is relevant. The sender receives the window size from the receiver and then creates a sliding window with the announced size that encompasses the total amount of data the receiver expects the sender to transmit. According to the size announced by the receiver, the sliding window closes spaces on the left and opens more spaces on the right each time the receiver acknowledges certain packets.

Table 2.4, which is based on the left-hand side of Figure 2.9, depicts the sliding window highlighted in green to demonstrate this concept. It is assumed that the total amount of data to be transmitted is 8kB, that each packet can carry 1kB, and that the receiver's buffer size is 4kB. During the handshake, the receiver announced a 4kB window, so the sender set the sliding window to 4kB, which covers the first four packets. The receiver acknowledges all three packets while announcing the new 3kB window size in packet 4, so the sender closes the sliding window on the left, which the receiver has already acknowledged, and opens new slots on the right so that the sliding window's size matches the new announcement. The sender and receiver continue until data transmission is complete and the connection is severed.

	kB	1	2	3	4	5	6	7	8	
ACKs		Announced in Handshake								
4		Closed			Announced in 4					
8					Closed		Announced in 8			
9							Closed	Announced in 9		
12								Closed		Announced in 12

TABLE 2.4: The sliding window in data transmission.

2.3.5 Connection Termination

The TCP termination procedure is similar to the connection establishment procedure. The termination is normally done in four-way or three-way handshakes. Figure 2.10 shows these scenarios assuming the client is the initiator.

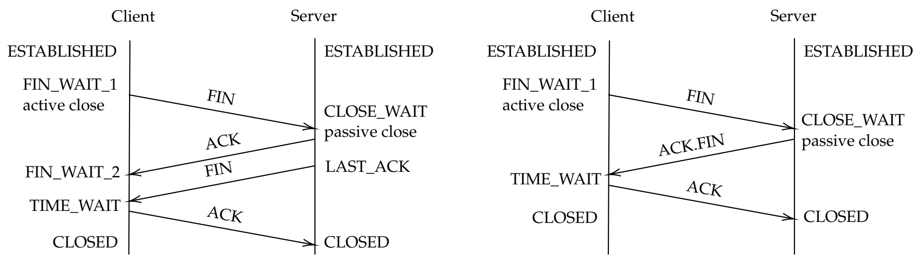


FIGURE 2.10: The connection termination stage. The four-way handshake is on the left and the three-way handshake is on the right.

The initiator will wait for 2MSL on the TIME_WAIT state. The MSL stands for maximum segment lifetime, which needs to be defined for each implementation. The MSL is the maximum amount of time the TCP segment is assumed to last during transmission. The reason for using MSL during termination is that the last ACK might be lost during transmission; therefore, the client waits long enough to make sure the receiver receives the ACK. If the server somehow did not receive the last ACK, it might resend the last FIN. The last ACK sent through the connection will be one trip, and the resend of FIN through the connection is another trip, so a round-trip is assumed to last as long as 2MSL. The 2MSL wait state prevents delayed packets in termination from being interpreted as a new connection.

The term "half-close" refers to a particular and unusual circumstance. For example, when the data transmission is complete, the client will initiate termination with the server by sending the FIN signal, and the server will respond by sending the ACK flag. The server is able to keep transmitting data to the client in an unbroken stream right up until the point when the server makes the decision to close the connection as well and sends the final FIN. [53] demonstrates how to use the half-close functionality that is available through the *rsh* command in Unix. The half-close scenario is discussed in RFC 5382 [54], despite the fact that it was not specified in RFC 793 [51]:

"The closing phase begins when both endpoints have terminated their half of the connection by sending a FIN packet."

The above statement can be understood as implying that when transitioning to the closing phase, it is expected that both parties will cease communication. Moreover, certain modern network implementations only allow for a brief time-out following the detection of the initial FIN, making the half-close feature impractical.

Another termination scenario is a simultaneous close, which occurs when both the server and client decide to terminate at the same time (Figure 2.11). This circumstance is comparable to the simultaneous open. The client and server simultaneously transmit FIN and enter the FIN_WAIT_1 state, then transition to TIME_WAIT upon receiving the ACK from the other side.

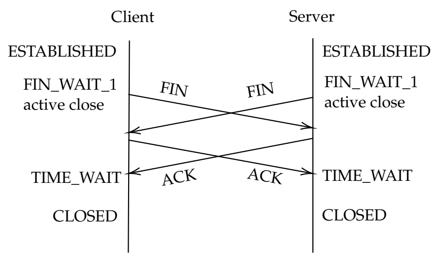


FIGURE 2.11: The simultaneous close.

2.3.6 Handling Packet Loss

When one or more packets fail to reach their destination or are significantly delayed during transmission, packet loss occurs. This may be the result of an unreliable network device, network congestion, or an attack involving packet-dropping [55]. Loss of packets has a direct effect on the network's throughput and delays the transfer of data. TCP is dependable in that it detects packet loss with time-out and retransmits the lost packet.

Every time the sender transmits a packet, a retransmission timer is initiated and a copy of the packet is placed in the retransmission queue. Each TCP segment in the queue has its own retransmission timer, which is currently counting down. If the ACK for a sent segment has not arrived when the timer expires, that packet is deemed lost. This can occur when either the data packet from the sender or the ACK of the received data packet is lost. In either case, the packet will be resent. Packet duplication is possible due to retransmission, and the receiver will simply discard the duplicated packet.

2.3.7 Handling Packet Out of Order

Out-of-order (OOO) packets are those that arrive at their destination in a different order than when they were sent. Typically, the sequence number is incremented by 1 for each packet sent in succession. OOO packets typically occur when packets are routed through a slower path, when packets are lost and resent by the sender, or when packets are routed through unreliable devices. Depending on the ACK number from the recipient, the OOO packets are either reassembled by the recipient or retransmitted by the sender.

2.3.8 Handling Damaged Data

The noise on the network wires and memory errors on network devices etc., can all cause packet data corruption. TCP is unable to recover data from corruption unless the corrupted packets are retransmitted. Corrupted packets are detected by the checksum in the TCP header. The checksum itself can be damaged, which cause false positive. All packets that have any damage are retransmitted as a result of lower throughput. Several higher-level implementations employ forward error correction (FEC) to recover data using redundancy [56]–[58], instead of retransmission, but these techniques are beyond the scope of our current discussion.

2.3.9 Handling Congestion

The explicit congestion notification (ECN) is introduced in the RFC 3168 document [52] as an extension of the earlier TCP/IP. For network devices, instead of dropping packets if congestion is experienced, they can mark the congestion in the ECN field in the IP header. The ECN field in the IP header is a 2-bit field that encodes four options,

- **Non-ECT**: The Non-ECN-Capable Transport when nothing is set in the ECN field (both bits are 0),
- **ECT(0)**: The ECN-Capable Transport,
- **ECT(1)**: The ECN-Capable Transport. Equivalent to ECT(0),
- **CE**: Congestion Encountered.

If the sender or the receiver supports ECN, they will set ECT(0) or ECT(1) in their IP header for outgoing packets. The congested router changes this field to CE in the IP header. Once the receiver received the packet marked with CE, the receiver set the ECE flag in the TCP header for all subsequent ACK packets to echo back the congestion mark. The sender will receive ECE from the receiver at some point and react by reducing the sending rate if any lost packet is detected. Once the sender reacts to the ECE, it will set the CWR flag in the TCP header, and the receiver stops sending ECE if CWR is received. Note that ECE stands for ECN-echo and CWR stands for congestion window reduced, which is mentioned in Section 2.3.1.

Later, the RFC 3540 document [59] specifies a new congestion signalling mechanism known as ECN-nonce, which is an extension of the current ECE and CWR. The ECN-nonce is utilised to prevent the receiver from concealing the congestion mark by not setting the ECE field from the returned ACK packet. As previously stated, the ECT can be either ECT(0) or ECT(1), corresponding to bit values 0 or 1, and it is set arbitrarily in the IP header. In addition, if congestion is encountered, the ECN field will be rewritten by the router, so the receiver will be unaware of the ECT value set by the sender. A flag in the TCP header, the nonce sum (NS) is a one-bit sum (XOR) of ECT values of both ends.

Assuming the sender sets ECT(1) in the IP header and there is no congestion, the receiver will receive ECT(1) from the sender because the router does not overwrite it. The receiver may set ECT to 0 (ECT(0)) and NS to 1, where 1 is the sum of ECT(1) set by the sender and ECT(0) set by the receiver; or the receiver may set ECT to 1 (ECT(1)) and NS to 0, where both sides set ECT to 1 so that the one-bit sum is 0. By checking the NS, the sender is aware that there is in fact no congestion. If congestion occurs, the receiver can either echo the ECE flag truthfully or conceal the congestion by attempting to guess the ECT value. The receiver cannot retrieve the ECT value set by the sender because the ECT has been overwritten; therefore, there is a 50 per cent chance that the NS will be incorrect. In other words, the sender can identify the issue and refuse to continue sending ECN-capable packets if the receiver incorrectly guesses the ECT value.

2.3.10 Handling Half-open

According to RFC793, a half-open TCP connection occurs when one side of an established connection has crashed and has not sent a notification (FIN) that the connection is closing. This is no longer a frequent practice. Today, half-open often refers to a situation in which the last ACK of the three-way handshake is not received from the initiator. The receiver will reset the half-open TCP connection.

2.4 Related Works

Prior to standardising the approach for converting network packet data into event logs, it is worth examining a few related studies that utilize process mining and conformance checking to identify anomalies. It is important to note that these studies are not exclusive to NIDS.

The technique introduced in [60] is used on industrial control systems (ICS) that identify anomalous behaviours and cyber-attacks using ICS data logs and the conformance checking analysis technique. The paper demonstrated the preprocessing of ICS data logs for PM and the result of anomaly detection. The inductive mining algorithm discovers their process model, and as a result, the average F-score for detecting attacks across two datasets is 0.81 with a recall of 0.71 and a precision of 0.93. A case is categorised as an anomaly if it does not produce a fitness score of 1.

On top of [34], papers [35], [36] provide expanded discussions on PM for IDS, discussing potential benefits of using PM for IDS, comparing different mining algorithms (such as the heuristic miner and the α-algorithm) that can be used for IDS, and displaying cyber-attacks detected in various industrial sectors, etc. Nonetheless, the precise method for implementing PM for IDS remains undetermined. Heuristic [61] miner improves the α-algorithm, which is capable of discovering short-loops patterns and skipping-activity patterns.

The paper [62] uses the Inductive Miner Infrequent algorithm to detect anomalous behaviours on the e-commerce platform. The data is processed based on the log recorded from ModSecurity, which is an open-source web application firewall, for the purpose of adapting PM. The events mainly consist of URLs and activated ModSecurity rules. The provided average fitness scores show that anomalies have lower scores, however, they do not have a clear measurement for F-score or accuracy.

The methodology described in [63] is based on PM and employs social network analysis metrics to identify anomalous behaviour. Instead of performing conformance testing, this method analyses social network metrics to identify anomalies. The log of normal traces is extracted from the process model using the workflow Petri net designer (WoPeD) process modelling tool [64]; the role-trace access control matrix, which is based on the normal traces and the role-activity access control matrix, is then created; and finally, the social network is constructed using the role-trace access control matrix and all logs. The experiment's F-score is greater than 0.92, per the outcome.

In general, PM applications in intrusion detection are not well established. There are discussions regarding the use of process mining for intrusion detection or for purposes other than intrusion detection. However, the experimental results of using process mining for intrusion detection are not well documented. This is the primary motivation for using process mining as an intrusion detection method in this thesis.

2.5 Data Preprocessing

2.5.1 Dataset

The dataset employed for process mining in this study is the CIC-IDS2017 dataset (abbreviated as IDS2017), as described in [24]. The IDS2017 dataset is the newest dataset of the time of the experiment which contains real-world benign and common attacks. The dataset consists of PCAP data and CSV data sheets. The PCAPs data is the binary packet data captured on the wire, while the CSV data sheets are extracted features from the captured PCAP data.

The B-profile system is used to generate the traffic that is included in the IDS2017 dataset [65]. The B-profile system makes use of profiles to generate both normal and attack traffic that is realistically modelled after human behaviour. This dataset is intended for use in performing intrusion detection. The purpose of the B-profile is to provide benign background traffic by extracting the abstract behaviour of a group of human users. It attempts to capture user-generated network events using machine learning and statistical analysis methods. Protocol request time, payload patterns, payload sizes, and protocol packet sizes are some of the encapsulating characteristics. B-Profiles can be used by an agent (CIC-BenignGenerator) or a human operator to create plausible benign events on the network. Organizations and academics can quickly produce realistic datasets using this method, doing away with the need to anonymise them.

The data was processed using the CICFlowMeter [30], which produces the CSV data that can be used in machine learning for intrusion detection after the traffic was generated and captured. The CICFlowMeter analyses each network flow and determines the statistical value for each flow, such as the total forward packet count and the count of FIN flags, which are characteristics akin to those of the KDD datasets previously mentioned. From Monday through Friday, IDS2017 generated and collected data using various profiles for each data set. The victims, attackers, and specifics of daily attacks are listed below.

1. **Monday 2017.07.03**
   Normal traffic only.

2. **Tuesday 2017.07.04**
   Brute force attacks including

   - FTP-Patator (09:20 – 10:20);
   - SSH-Patator (14:00 – 15:00).
     Attacker: 172.16.0.1
     Victim: 192.168.10.50

3. **Wednesday 2017.07.05**
   DoS/DDoS attacks including

   - DoS Slowloris (09:47 – 10:10);

- DoS Slowhttptest (10:14 – 10:35);
- DoS Hulk (10:43 – 11:00);
- DoS GoldenEye (11:10 – 11:23).
  Attacker: 172.16.0.1
  Victim: 192.168.10.50

Heartbleed attacks on port 444 (15:12 - 15:32).

Attacker: 172.16.0.1

Victim: 192.168.10.51

4. **Thursday 2017.07.06**
   Web attacks including

   - Force attacks (09:20 – 10:00);
   - XSS (10:15 – 10:35);
   - SQL Injection (10:40 – 10:42).
     Attacker: 172.16.0.1
     Victim: 192.168.10.50

   Infiltration including

   - Dropbox Download (14:19, 14:20-14:21, 14:33 -14:35);
     Attacker: 205.174.165.73
     Victim: 192.168.10.8
   - Cool Disk (14:53 – 15:00).
     Attacker: 205.174.165.73
     Victim: 192.168.10.25

   Port Scan + Nmap (15:04 – 15:45)

   Attacker: 192.168.10.8

   Victim: All IPs

5. **Friday 2017.07.07**
   Botnet ARES (10:02 – 11:02)

   Attacker: 205.174.165.73

   Victims: 192.168.10.15, 192.168.10.9, 192.168.10.14, 192.168.10.5, 192.168.10.8

   Port Scan (Multiple windows)

   Attacker: 172.16.0.1

   Victims: 192.168.10.50

   DDoS LOIT (15:56 – 16:16)

   Attacker: 172.16.0.1

   Victims: 192.168.10.50

The IP addresses listed above were obtained from local PCAP records. The first attack time and the last attack time are crucial; there are no benign or other types of attacks in between. Some attacks take place between the same hosts over a number of time periods so multiple time periods are provided. The infiltration via Dropbox download consists of two steps: first, the victim, who is running Metasploit, downloads the malicious file from Dropbox, and then the attacker uses the backdoor to run Nmap and port scan on the entire victim network.

2.5.2 Data Extraction

The required data is extracted directly from PCAP files instead of utilizing the preprocessed CSV data from the IDS2017 dataset. The reason for this is that the preprocessed CSV data sheets only provide flow-level statistical data, not packet-level data, and therefore cannot be used for process mining.

The naturally occurring sequential order of packets within a network flow enables the flow to be treated as a process. The attributes that can be utilized for process mining must be determined. The number of event classes and define their specific characteristics must be restricted. As with business process management (BPM), the first step is to define each atomic activity and subsequently record them in a precedence-based order. The recorded data can serve as the event log and is essential for process mining.

TCP flags are the most obvious field that can be converted into one of the attributes based on the structure of the TCP segment. Fields with arbitrary values, such as sequence numbers and IP addresses, cannot be used as attributes. Numerical information such as packet sizes and payload sizes cannot be used directly in the PM. This statement does not imply that these values cannot be converted to attributes; to convert numerical values to attributes, the value can be mapped into a discrete space. For instance, if the maximum size of a packet is 1500 Bytes, the packet size attribute can be defined with 10 buckets, and each bucket spans a range of 150 Bytes. However, adding a single attribute with 10 possible values multiplies the number of event classes by 10. In other words, the number of event classes will increase exponentially, and attribute sizes are difficult to regulate. At present, numerical values have been excluded from the event log due to their complexity and practicality.

Wireshark is a well-known, cross-platform, open-source network protocol analyzer with a graphical user interface (GUI). Wireshark offers features such as filtering, visualisation, and capturing, among others. Wireshark is used to divide and filter the PCAP dataset in the initial stage. All other packets are filtered out because only need TCP packets are needed. Using the filtering functionality based on the listed information of attacker IP addresses, victim IP addresses, and attack times, it is possible to divide the data into normal PCAP data and attack PCAP data. The example below uses Wireshark to filter out Botnet attack packets from Friday's data. One needs to highlight the packets in the GUI for the purpose of selecting the time period.

```
(((ip.src == 205.174.165.73) && (ip.dst == 192.168.10.15)) || \
((ip.dst == 205.174.165.73) && (ip.src == 192.168.10.15))|| \
((ip.src == 205.174.165.73) && (ip.dst == 192.168.10.9)) || \
((ip.dst == 205.174.165.73) && (ip.src == 192.168.10.9))|| \
((ip.src == 205.174.165.73) && (ip.dst == 192.168.10.14)) || \
((ip.dst == 205.174.165.73) && (ip.src == 192.168.10.14))|| \
((ip.src == 205.174.165.73) && (ip.dst == 192.168.10.5)) || \
((ip.dst == 205.174.165.73) && (ip.src == 192.168.10.5))|| \
((ip.src == 205.174.165.73) && (ip.dst == 192.168.10.8)) || \
((ip.dst == 205.174.165.73) && (ip.src == 192.168.10.8)))
```

The same procedure will be applied to all necessary files because IDS2017 contains a number of PCAP files that are related to various days and hosts. All PCAP files containing attacks are necessary, and some PCAP files containing only normal traffic are chosen from Monday's data. Due to the large size of the dataset, only a few PCAP files containing normal traffic are selected. The filtering process yields a number of new PCAP files that can be processed further. Case IDs and timestamps are required for process mining in addition to TCP flags. Each network flow must be identified based on the socket pairs in order to construct cases. The extraction of sockets, timestamps, and flags from each packet is necessary for this step. The required data is extracted into the CSV datasheet using TShark, a non-GUI version of Wireshark. Using TShark at this moment will be more efficient as all files can be processed in batch using the shell. CSV datasheets with the necessary feature columns are then obtained. Table 2.5 contains an example of a portion of the extracted data, and the example command line used in this step is shown below.

```
tshark -r '[dump PCAP path]' -T fields -J tcp -E separator=, \
-e frame.time -e ip.src -e ip.dst -e tcp.srcport -e tcp.dstport \
-e tcp.flags > '[output].csv'
```

...	...	...	...	...	...	...
Mar 2	2018 12:47:29.525978000 GMT	172.31.69.30	172.31.0.2			
Mar 2	2018 12:47:29.527139000 GMT	172.31.0.2	172.31.69.30			
Mar 2	2018 12:47:29.691028000 GMT	172.31.69.30	23.15.8.131	49684	80	0x000000c2
Mar 2	2018 12:47:29.711902000 GMT	23.15.8.131	172.31.69.30	80	49684	0x00000012
Mar 2	2018 12:47:29.711920000 GMT	172.31.69.30	23.15.8.131	49684	80	0x00000010
Mar 2	2018 12:47:29.712753000 GMT	172.31.69.30	23.15.8.131	49684	80	0x00000018
Mar 2	2018 12:47:29.716608000 GMT	172.31.69.30	172.31.0.2			
Mar 2	2018 12:47:29.716925000 GMT	172.31.69.30	172.31.0.2			
Mar 2	2018 12:47:29.718048000 GMT	172.31.0.2	172.31.69.30			
Mar 2	2018 12:47:29.718196000 GMT	172.31.0.2	172.31.69.30			
Mar 2	2018 12:47:29.719513000 GMT	172.31.69.30	13.33.81.58	49685	443	0x000000c2
Mar 2	2018 12:47:29.719681000 GMT	172.31.69.30	13.33.81.58	49686	443	0x000000c2
Mar 2	2018 12:47:29.725779000 GMT	172.31.69.30	172.31.0.2			
Mar 2	2018 12:47:29.726065000 GMT	172.31.0.2	172.31.69.30			
Mar 2	2018 12:47:29.726187000 GMT	172.31.69.30	67.227.156.183	49687	443	0x000000c2
Mar 2	2018 12:47:29.726679000 GMT	172.31.69.30	172.31.0.2			
Mar 2	2018 12:47:29.726895000 GMT	172.31.0.2	172.31.69.30			
Mar 2	2018 12:47:29.727225000 GMT	172.31.69.30	172.31.0.2			
Mar 2	2018 12:47:29.727483000 GMT	172.31.0.2	172.31.69.30			
Mar 2	2018 12:47:29.727627000 GMT	172.31.69.30	172.31.0.2			
Mar 2	2018 12:47:29.727993000 GMT	172.31.69.30	172.31.0.2			
Mar 2	2018 12:47:29.728242000 GMT	172.31.0.2	172.31.69.30			
Mar 2	2018 12:47:29.728487000 GMT	172.31.69.30	172.31.0.2			
Mar 2	2018 12:47:29.728629000 GMT	172.31.0.2	172.31.69.30			
Mar 2	2018 12:47:29.729349000 GMT	172.31.0.2	172.31.69.30			
Mar 2	2018 12:47:29.733559000 GMT	23.15.8.131	172.31.69.30	80	49684	0x00000010
Mar 2	2018 12:47:29.733782000 GMT	23.15.8.131	172.31.69.30	80	49684	0x00000018
Mar 2	2018 12:47:29.735968000 GMT	13.33.81.58	172.31.69.30	443	49685	0x00000012
Mar 2	2018 12:47:29.735994000 GMT	172.31.69.30	13.33.81.58	49685	443	0x00000010
Mar 2	2018 12:47:29.736145000 GMT	13.33.81.58	172.31.69.30	443	49686	0x00000012
Mar 2	2018 12:47:29.736162000 GMT	172.31.69.30	13.33.81.58	49686	443	0x00000010
Mar 2	2018 12:47:29.736638000 GMT	172.31.69.30	13.33.81.58	49685	443	0x00000018
Mar 2	2018 12:47:29.736776000 GMT	172.31.69.30	13.33.81.58	49686	443	0x00000018
Mar 2	2018 12:47:29.742864000 GMT	67.227.156.183	172.31.69.30	443	49687	0x00000012
...	...	...	...	...	...	...

TABLE 2.5: Data extracted with TShark.

Timestamps, source IP addresses, destination IP addresses, source ports, destination ports, and flags are displayed in the first to last columns. Some samples are removed from the datasheet due to the existence of missing information. Reconstructing cases and transforming datasheets into event logs is the next step, which will be discussed in the following section.

2.5.3 Event Log Construction

The event log requires at least timestamps and events that contain several attributes. The observed packets are in the sequence $P = \langle p_i \rangle_{i=1}^{n}$, where p_i is each individual packet. A packet can be considered as an event e in process mining where attributes form event classes. The hexadecimal flags are converted to human-readable strings, which makes the analysis easier, e.g., 0x000000c2 is equivalent to 0000 1100 0010 in binary, and it is 000.CWR.ECE.SYN according to the flag field in the TCP segment. The leading 000. is the reserved bits which are unnecessary.

The observed packets from the entire PCAP can also form a set of TCP flows $T = \{t_i\}_{i=1}^{m}$, where each flow t_i can be constructed according to the IP addresses and ports of two hosts (T can also be considered as the event log $L = T$ from the perspective of process mining). Please note that in process mining, a flow would correspond to a case, and both of these terms may be used in the future context interchangeably.

The start of a new TCP flow is defined as the reception of a packet with the SYN flag set but not the ACK flag (i.e., the first packet of the three-way handshake). In addition, the IP addresses and ports of two hosts are determined by this initial packet. To build each case, it is possible to search for SYN flags within the extracted data. Once a SYN flag is detected, a unique case ID is assigned to this new flow. Using the source and destination sockets contained in the initial packet, it is possible to track the flow to its final destination. For instance, the source IP is IP_{client} and the source port is $Port_{client}$, indicating that the packet originated from the client, and the destination IP is IP_{server} and the destination port is $Port_{server}$. The bidirectional flows may be reconstructed using the client socket $(IP_{client}, Port_{client})$ and the server socket $(IP_{server}, Port_{server})$. If a pair of sockets exist in either source or destination, packets are considered to belong to the same flow.

A TCP flow is deemed complete once a packet with either the FIN or RST flag is detected, and any incomplete flows are filtered out. The statement does not indicate that all other packet information is discarded after FIN or RST is set. Therefore, any ongoing packets during the termination stage after the first FIN or RST are retained. Incomplete flow occurs when the packet-capturing process is stopped before the flow is complete. The timestamps are formatted. At this point, the event log consists of case IDs, timestamps, and a solitary attribute for flags.

An extra attribute labelled S or C is included with each packet to maintain additional flow data for process mining. S denotes that the packet was sent from the server, while C denotes that it was sent from the client. This feature prevents a number of loops from taking place in process models, and the phenomenon will be discussed further later. The event log after reconstruction is depicted in Table 2.6. Keep in mind that Tables 2.5 and Table 2.6 were taken from two distinct data segments. Each row was assigned either the type of attack or normal based on the provided IP addresses of the attackers and victims. The IP addresses and ports are discarded from Table 2.6 as they are used to reconstruct the flows and are not needed in process mining.

Case_ID	Timestamp	Flags	Host
151043	2017/07/04 14:00:35.190179000	000.SYN.	C
151043	2017/07/04 14:00:35.222535000	000.ACK.SYN.	S
151043	2017/07/04 14:00:35.222586000	000.ACK.	C
151043	2017/07/04 14:00:35.237412000	000.ACK.PSH.	C
151043	2017/07/04 14:00:35.270301000	000.ACK.	S
151043	2017/07/04 14:00:35.270305000	000.ACK.	S
151043	2017/07/04 14:00:35.467062000	000.ACK.	S
151043	2017/07/04 14:00:35.467285000	000.ACK.PSH.	S
151043	2017/07/04 14:00:35.467347000	000.ACK.	C
151043	2017/07/04 14:00:35.467352000	000.ACK.PSH.	S
151043	2017/07/04 14:00:35.511515000	000.ACK.	C
151043	2017/07/04 14:00:45.461285000	000.ACK.	C
151043	2017/07/04 14:00:45.466144000	000.ACK.FIN.	S
151043	2017/07/04 14:00:45.466173000	000.ACK.	C
151043	2017/07/04 14:00:45.466198000	000.ACK.FIN.	C
151043	2017/07/04 14:00:45.493763000	000.ACK.	S
151043	2017/07/04 14:00:45.498542000	000.ACK.	S
281008	2017/07/05 17:03:45.498235000	000.SYN.	C
281008	2017/07/05 17:03:45.521284000	000.ACK.SYN.	S

281008	2017/07/05 17:03:45.521360000	000.ACK.	C
281008	2017/07/05 17:03:45.521561000	000.ACK.PSH.	C
281008	2017/07/05 17:03:45.544708000	000.ACK.	S
281008	2017/07/05 17:03:45.545025000	000.ACK.PSH.	S
281008	2017/07/05 17:03:45.545073000	000.ACK.	C
281008	2017/07/05 17:03:45.561805000	000.ACK.PSH.	C
281008	2017/07/05 17:03:45.624241000	000.ACK.	S
281008	2017/07/05 17:03:45.820846000	000.ACK.PSH.	C
281008	2017/07/05 17:03:45.843822000	000.ACK.	S
281008	2017/07/05 17:03:45.921777000	000.ACK.PSH.	S
281008	2017/07/05 17:03:45.964684000	000.ACK.	C
281008	2017/07/05 17:03:46.802581000	000.ACK.PSH.	C
281008	2017/07/05 17:03:46.825827000	000.ACK.	S
281008	2017/07/05 17:03:46.827025000	000.ACK.PSH.	S
281008	2017/07/05 17:03:46.827041000	000.ACK.	C
281008	2017/07/05 17:03:53.457141000	000.ACK.FIN.	C
281008	2017/07/05 17:03:53.480155000	000.ACK.PSH.	S
281008	2017/07/05 17:03:53.480156000	000.ACK.FIN.	S
281008	2017/07/05 17:03:53.480264000	000.RST.	C
...	...	...	...

TABLE 2.6: The constructed event log from packets.

There are several datasheets that need to be converted separately where each containing different attacks and normal traffic, as was previously mentioned. Upon conversion, the datasets containing attacks are segregated, and only the datasets comprising normal traffic are merged to generate a consolidated log.

2.6 Summary

This chapter recalls the first research question. This chapter aims to demonstrate an understanding of process mining and transmission control protocol before developing a method for translating network packets into event logs for future research. Process mining concepts such as event logs, DFGs, and process model abstractions were introduced in depth. DFG is fundamental to the majority of process-discovery methods, including the method that will be presented in this thesis. Internet communication relies heavily on the transmission control protocol. Relevant TCP concepts, including the TCP stages, flags, and error handling, are introduced in depth. To better comprehend how data are converted into event logs, relevant studies are examined. The IDS 2017 dataset was chosen for the experiment because it is the most recent available dataset at the time of the experiment and contains unprocessed PCAP data. After preparation, Tshark and Wireshake are used to extract and filter packet data; the packets are then processed and stored in CSV datasheets. CSV datasheets are equivalent to standard event logs and can be utilised directly for process mining.

Chapter 3

Mined Process Models

Chapter 2 discussed how to create event logs from network packets. Case ID, timestamps, header flags, and S/C labels are the data columns used in the subsequent experiment. These columns were chosen because they are not numerical and process mining can extract transitional information from the data.

Every parameter was retained as default and did not apply any abstraction in both ProM [66] and Disco [67] in process mining. Thus, the fuzzy model will be equivalent to the initial process model or DFG. However, the initial model will still be referred to as fuzzy in a later context. The primary reason for not using abstraction is that it is more important to comprehend the problems than to examine various mining hyperparameters. In other words, the intention is to determine how network packets behave in real-world scenarios and how closely they match the state diagrams. Secondly, the aim is to mine authentic models from the event logs provided, as rare cases are not always anomalies. Although [45] mentions that lowering the thresholds will cause the process models to resemble DFGs and that there are limitations to this, the decision is to keep these rare traces because they are known to be normal traces. Thirdly, although process mining is capable of discovering the global data structure, it cannot perform real-time detection with the fuzzy miner or inductive miner. The initial goal of our research is to design a packet-level intrusion detection system, but this naive approach may fail.

This chapter shows the experiment of using ProM and Disco to mine several process models using normal cases. The process table, which is the Petri net's equivalent representation, is then presented. The process table is a novel way of representing the process model for better readability. The findings from these process model studies are presented and contrasted. Finally, the normal inductive model is used to check conformance for anomaly-based intrusion detection. It must be noted that a typo exists in some charts in the following sections where ECE has been written as ECN. Note that the correct spelling is ECE in all cases.

3.1 Inductive Miner - ProM

ProM is an open-source process mining framework that is written in Java and provides a graphical user interface. The framework allows plugins that provide various functionalities to be built such as process discovery algorithms, exporting tools and data filtering tools [68]–[70]. The default event log format for ProM is XES files which are similar to the XML format; however, the import plugin allows event logs with CSV or XML format to be imported. During process mining, the attribute strings of events will be concatenated with the pipe symbol "|" to create the event classes. For example, the flag attribute for an event is 000.SYN. and the host attribute is C, the event class name for this event will be 000.SYN.|C.

First, for comparisons, several process models are mined with normal cases only. Nine models are derived from normal datasets of varying sizes. One of the process models is mined from the complete log, while the others are mined from sub-logs of the complete log. Cases for the sub-logs are selected at random from the complete log. For each size, two models are mined from two distinct sub-logs. In this section, the focus is on displaying each of these models and discussing the observations made from them. The following is a list of all process models mined using an inductive miner.

1. Process model mined with 5 traces 1^{st}.
2. Process model mined with 5 traces 2^{rd}.
3. Process model mined with 100 traces 1^{st}.
4. Process model mined with 100 traces 2^{rd}.
5. Process model mined with 20k traces 1^{st}.
6. Process model mined with 20k traces 2^{rd}.
7. Process model mined with 100k traces 1^{st}.
8. Process model mined with 100k traces 2^{rd}.
9. Process model mined with the complete-log.

The process models mined with ProM are in the form of Petri nets. An illustration of a process model discovered using the inductive miner is shown in Figure 3.1. The Petri net consists of three different types of components: places, actions, and silent actions τ. Tokens are kept in *places*, which are represented in the Petri net as circles. Rectangles with corresponding event class names are used to represent *actions*. *Silent actions* can consume and fire tokens without evoking any actual events; they are represented as rectangles without event class names.

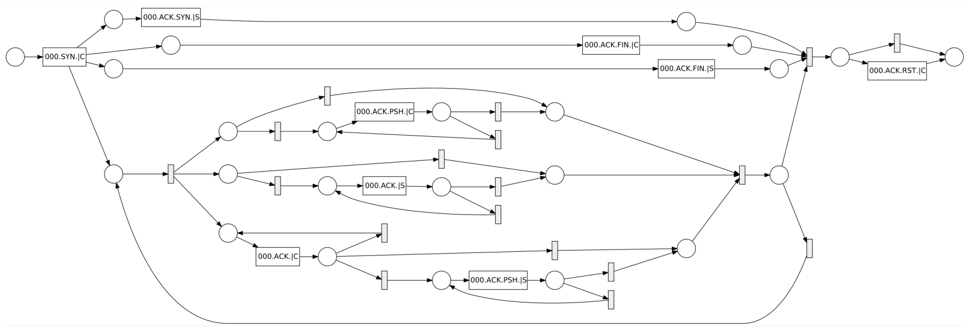


FIGURE 3.1: process model example.

The left-most place is a special place that is not connected to any output of any action. It stores the initial token that the Petri net can use to start replaying. The initial token can be consumed by action 000.SYN|C, and the action will produce four tokens for four places that are connected to its outputs. The newly produced tokens can then be consumed by other actions. In each step, an action can only consume one token from a place and produce one token in a place. If the action has multiple inputs or outputs places, it will consume or produce one and only one token for each connected place. The action can not fire if there is not one token in all input places, i.e., if an action has two input places, each place needs to have one token for this action to fire. A place can hold multiple tokens, and the action may fire multiple times. The right-most place is also a special place that stores the end token.

For enhanced readability, colouring and numbering are applied to process models and key locations. Actions in the connection-establishing stage are highlighted in yellow, while those in the established (data transmission) stage are highlighted in green; actions in the termination stage remain uncoloured. Typically, the inductive models feature five distinct patterns, as shown in Figure 3.2. Unique IDs are allocated to each location and silent action. In the subsequent content, places are denoted with the letter P, such as P1 for a place with ID 1, and silent actions with the letter S, like S1 for a silent action with ID 1.

- **Loops**
  Loops enable a single token to act as input to the Petri net and initiate multiple executions of an action. As observed in the image, 000.ACK.|C can consume a token from place P8 and generate a token at P11. Nevertheless, silent action S12 possesses the capability to 'transfer' the token back to P8, making it available for use by 000.ACK.|C once more.

- **Short-cuts**
  Through the use of a shortcut, a particular action in the model can be circumvented, meaning that it is no longer necessary for the action to fire when the token is replayed. Within the context of this illustration, 000.ECE.ACK.SYN.|S will be circumvented in the event that the silent action S2 uses up the token located in P2. This pattern typically appears when there are two traces in the event log that are very similar to one another, but only one of the traces contains the event class 000.ECE.ACK.SYN.|S.

- **Sequences**
  If two actions are to occur in a process model, the second action must fire only after the first action has fired, and not the other way around. This is what is meant by the concept of sequence. For example, 000.ECE.ACK.SYN.|S will not be able to fire until after 000.CWR.ECE.SYN.|C has already fired.

- **Concurrences**
  Concurrency does not mean that all of the involved actions are executed at the same time in the event log; rather, it means that these actions can fire in any order. In the scenario depicted in Figure 3.2, the order in which the actions 000.ACK.PSH|S, 000.ACK.FIN.|S, and 000.ACK.PSH.FIN.|S are fired does not make a difference in the model representation as long as each of these actions is executed before S41 is executed. It is possible for the sub-sequence of a flow to be written as either $\langle 000.ACK.PSH|S, 000.ACK.PSH.FIN.|S, 000.ACK.FIN.|S\rangle$ or $\langle 000.ACK.PSH|S, 000.ACK.PSH.FIN.|S, 000.ACK.FIN.|S.\rangle$ The reason for concurrent events was discussed earlier in Chapter 2.

- **Exclusive-selection**
  Under the exclusive-selection pattern, only one action can be selected at a time. Because P6 only has one token that can be consumed once in the chart, either 000.ACK.PSH.|C, 000.ACK.SYN.|S, or 000.SYN.|C could fire.

An action might appear in multiple patterns due to the existence of nested patterns. For example, 000.ACK.|S is included within the loop, shortcut, sequence, and concurrence patterns in Figure 3.1. To enhance the visualisation of nested structures,

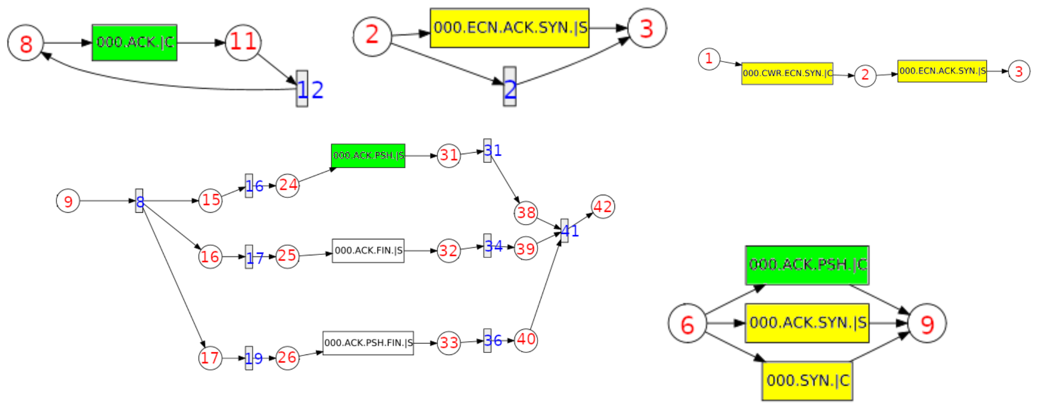


FIGURE 3.2: Inductive model pattern examples. Loops: first row first figure; short-cuts: first row second figure; sequences: first row third figure; concurrences: second row first figure; exclusive-selection: second row second figure.

a type of process model called Process Table has been developed, derived from Petri nets.

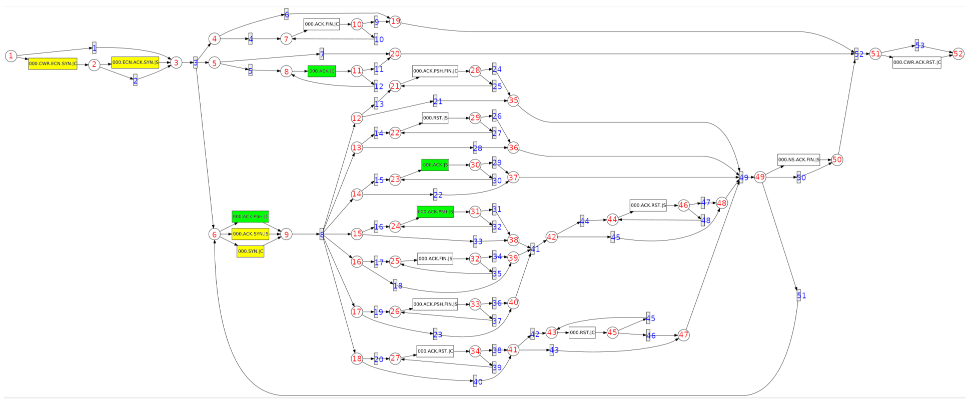


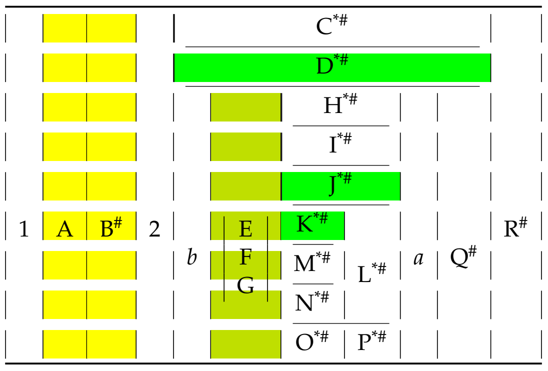
FIGURE 3.3: process model example (Appendices figure A.1).

				000.ACK.FIN.|C$^{*\#}$						
				000.ACK.|C$^{*\#}$						
						000.ACK.PSH.FIN.|C*				
						000.RST.|$^{*\#}$				
						000.ACK.|$^{*\#}$				
1	000.CWR.ECE.SYN.|C	000.ECE.ACK.SYN.|S$^{\#}$	2		000.ACK.PSH.|C	000.ACK.PSH.|S$^{*\#}$	000.ACK.RST.|S$^{*\#}$			000.CWR.ACK.RST.|C$^{\#}$
				b	000.ACK.SYN.|S	000.ACK.FIN.|$^{*\#}$		*a*	000.NS.ACK.FIN.|$^{\#}$	
					000.SYN.|C	000.ACK.PSH.FIN.|S$^{*\#}$				
						000.ACK.RST.|C$^{*\#}$	000.RST.|C$^{*\#}$			

TABLE 3.1: the overview of the process model example.

Transitions and patterns equivalent to Figure 3.3 are consistently represented in process table 3.1. Process table 3.2 serves as a reference to process table 3.1, with all event-class names replaced by characters in process table 3.2 to facilitate a clearer explanation. The interpretation of a process table will be discussed next.

Event classes in a row adhere to the sequential pattern, while those in a column follow the concurrent pattern. For example, it is evident that event classes C, D, and the set of event classes beneath D follow the concurrent pattern. This set of event classes, including b, E, F, . . . , O, P, and a, is nestled within the concurrent pattern.



C*#
D*#
H*#
I*#
J*#
1
A
B#
2
E
F
G
K*#
b
M*#
L*#
a
Q#
R#
N*#
O*#
P*#

TABLE 3.2: the reference of table 3.2.

A and B are positioned in the same row but in different columns since they are part of a sequential pattern. The '*' symbol denotes an action with a loop pattern, such as C and D, while the '#' symbol denotes a skippable action, like B and C. In this instance, a shortcut spanning A and B is represented by numbers 1 and 2, which jointly symbolise a silent action. This shortcut begins before A and ends after B, with its locations corresponding to P1 and P3 in figure 3.1. Another silent action connects a and b, indicating a loop that corresponds to locations P6 and 49 in figure 3.1. Within the process table, the exclusive-selection pattern is displayed as a single cell containing multiple actions, such as E, F, and G. Tables incorporating rare cases will be addressed when they occur in section 3.1.5, as some of these rare instances were not observed in this process model.

If an action to the left of the vertical border fires, all actions on the right side must either fire or be bypassed. For instance, when B fires, C, D, and the nested pattern should all fire or bypass. Clearly, C and D can be bypassed, necessitating the firing of either E, F, or G from the exclusive-selection pattern. The action on the right side of the vertical border can only be triggered when all actions on the left side have been fired or bypassed.

Let us examine the concepts of missing tokens and remaining tokens. The missing token indicates that some events during a token replay on a case do not have the necessary token to consume. A case that has been perfectly executed will have the exact number of tokens to consume and will only leave one end token at the end place after the replay. A token is a remaining token if it remains unconsumed in any place aside from the end place. The case below will cause a missing token and a remaining token.

- $\langle A, A, E \rangle$

Event A consumes the initial token and produces a token at the input place of node B. The second A event has no token to consume, so there is a missing token. Second A produces another token at the input place of node B, but E will only consume one token. In summary, this replay results in one missing token and one remaining token. The cases below are those which perfectly comply with the process model.

- $\langle A, B, C, C, C, E, H, J, I, D, K, N, M, L, O, P, Q, R \rangle$
- $\langle A, B, F, M, L, Q, R \rangle$
- $\langle E, E, E, E, E \rangle$

In the following sections, the process models for various setups and the event classes observed in each setup will be demonstrated. Traces that can be perfectly replayed on the process model will be shown, reflecting the generality and accuracy of the models. These traces are not actual traces from event logs but rather synthetic traces created by observing each process model.

The concept of accuracy states that the process model extracted from the event log should allow fewer unobserved traces in the event log to be replayed perfectly on this model. In other words, an accurate process model closely resembles the observed traces from the event log in its representation of patterns. On the other hand, generalisation can be the opposite of accuracy. A general process model permits the flawless replay of additional unanticipated traces. If a set of normal cases is used to mine a model, a model that well-establishes the normal behaviours can be anticipated, resulting in a high true-positive rate for anomaly detection. However, the false-positive rate might be high because normal traces may not be observed in the event log. The benefit of a general model is that it can accommodate unanticipated normal traces, thereby reducing the false-positive rate. Nevertheless, there is a high likelihood that anomalies will be perfectly replayed on the model, resulting in a high false-negative rate. The balance between a process model's accuracy and generality must be determined based on its application, which will be discussed in greater detail later.

3.1.1 Process Models Mined with 5 Cases

Five cases were used to mine the model in Figure 3.4. This is a case with relatively a small number of cases to test how the number of cases affects the generality of the process model and how generality affects the model's accuracy. Nine of the event classes listed in Table 3.3 are observed with this configuration. The process table is presented in Table 3.4

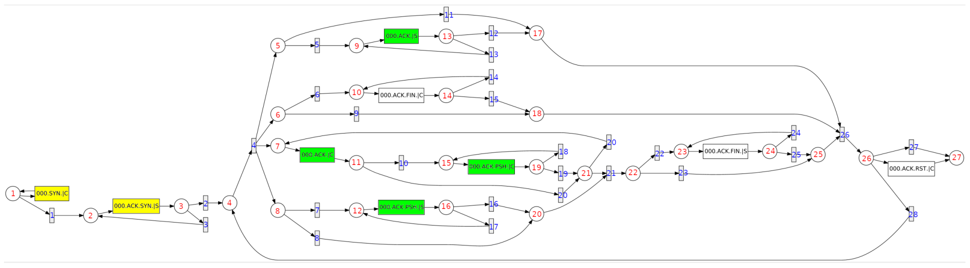


FIGURE 3.4: First model mined with 5 traces (Appendices figure A.2).

000.SYN.|C	000.ACK.SYN.|S	000.ACK.|S	000.ACK.FIN.|C
000.ACK.|C	000.ACK.PSH.|S	000.ACK.PSH.|C	000.ACK.FIN.|S
000.ACK.RST.|C	|	|	|

TABLE 3.3: List of *event classes* in the first model mined with 5 traces.

			000.ACK.|S$^{*\#}$						
			000.ACK.FIN|C$^{*\#}$						
000.SYN.|C*	000.ACK.SYN.|S*	2	*b*	000.ACK.|C	000.ACK.PSH.|C$^{*\#}$	*a*	000.ACK.FIN.|S$^{*\#}$	1	000.ACK.RST.|C$^{\#}$
			000.ACK.PSH.|S$^{*\#}$						

TABLE 3.4: The process table of the first model mined with 5 traces.

The following (Trace 3.1) is a list of the shortest traces that can be replayed on this process model. In the sections that follow, we notate the sequence using → for easier reading.

$$START \rightarrow 000.SYN.|C \rightarrow 000.ACK.SYN.|S \rightarrow 000.ACK.|C \rightarrow END \tag{3.1}$$

The shortest trace appears to be a valid TCP flow at the beginning, as it follows the three-way handshake connection establishment; however, the connection terminates without FIN or RST, which is not valid and should not occur. The accuracy of the process model is evaluated by analyzing the shortest trace. Here is a potential trace (Trace 3.2) involving all observed event classes that can be replayed perfectly on the process model.

$$\begin{aligned} &START \rightarrow 000.SYN.|C \rightarrow 000.ACK.SYN.|S \rightarrow 000.ACK.|S \rightarrow 000.ACK.|C \\ &\rightarrow 000.ACK.|S \rightarrow 000.ACK.FIN|C \rightarrow 000.ACK.PSH.|C \rightarrow 000.ACK.FIN.|S \\ &\rightarrow 000.ACK.RST.|C \rightarrow END \end{aligned} \tag{3.2}$$

The model seems general enough to allow all event classes to appear in one trace, however, this might allow illegal traces to be replayed on the process model. According to the above trace, the three-way handshake is not a proper handshake as there is an extra ACK packet sent from the server, which could happen in real life, but the data transmission stage and the termination stage are mixed together, so these stages are invalid.

The second process model, mined with 5 cases, can be seen in Figure 3.5. From these 5 cases, 9 distinct event classes were observed, as listed in Table 3.5. These nine observed event classes correspond precisely with the first model. The corresponding process table is shown in Table 3.6.

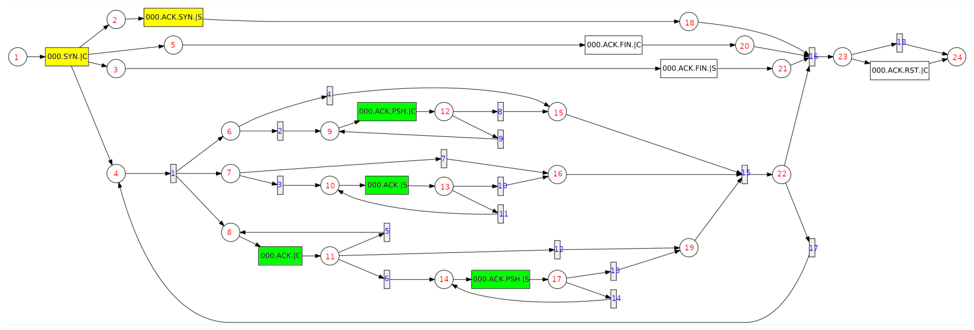


FIGURE 3.5: Second process model mined with 5 traces (Appendices figure A.3).

000.SYN.|C	000.ACK.SYN.|S	000.ACK.PSH.|C	000.ACK.|S
000.ACK.|C	000.ACK.PSH.|S	000.ACK.FIN.|C	000.ACK.FIN.|S
000.ACK.RST.|C	|	|	|

TABLE 3.5: List of *event classes* in the second process model mined with 5 traces.

The shortest trace that is replayable by the second process model is shown below.

$$\begin{aligned} &START \rightarrow 000.SYN.|C \rightarrow 000.ACK.SYN.|S \rightarrow 000.ACK.FIN.|S \rightarrow \\ &000.ACK.FIN.|C \rightarrow 000.ACK.|C \rightarrow 000.ACK.|C \rightarrow END \end{aligned} \tag{3.3}$$

🟨		000.ACK.SYN.|S			
🟨		000.ACK.FIN.|C			
🟨		000.ACK.FIN.|S			
000.SYN.|C		000.ACK.PSH.|C$^{*\#}$			000.ACK.RST.|C$^{\#}$
🟨	*b*	000.ACK.|S$^{*\#}$		*a*	
🟨		000.ACK.|C*	000.ACK.PSH.|S$^{*\#}$		

TABLE 3.6: The process table of the second model with 5 traces.

Trace 3.3 is less precise than Trace 3.1 for the preceding model. Trace 3.1 has an accurate connection establishment stage, whereas Trace 3.3 has no valid stages, similar to Trace 3.2 in the first model. Below is depicted a possible trace that includes all observed event classes.

$$\begin{aligned}&START \rightarrow 000.SYN.|C \rightarrow 000.ACK.SYN.|S \rightarrow 000.ACK.PSH.|C \rightarrow \\ &000.ACK.FIN.|S \rightarrow 000.ACK.FIN.|C \rightarrow 000.ACK.|C \rightarrow 000.ACK.|S \rightarrow \\ &000.ACK.PSH.|S \rightarrow 000.ACK.RST.|C \rightarrow END\end{aligned} \tag{3.4}$$

Although Trace 3.4 lacks a valid stage, it can still be replayed on the process model as a "normal" trace that is comparable to Trace 3.2.

The construction of the process table using patterns from the process model will be illustrated in the following section, where 100 cases are mined. As the next section provides superior examples, the demonstration will take place there rather than in the current section. Additionally, the usage of the process table to generate synthetic traces, such as 3.3 and 3.4, is demonstrated.

3.1.2 Process Models Mined with 100 Traces

The process model in Figure 3.6 was obtained from 100 cases in which the event log contained the 12 different event names listed in Table 3.7. In contrast to the previous example, some classes that are more uncommon are observed, such as 000.ACK.RST.|S, 000.RST.|C, and 000.RST.|S. Let us review the creation of the process table. The specifics of patterns obtained from Figure 3.6 are listed below. A process table (Table 3.8) is constructed, drawing on the observations mentioned earlier, which assists in visualising and pinpointing all potential traces capable of being perfectly replayed on the process model.

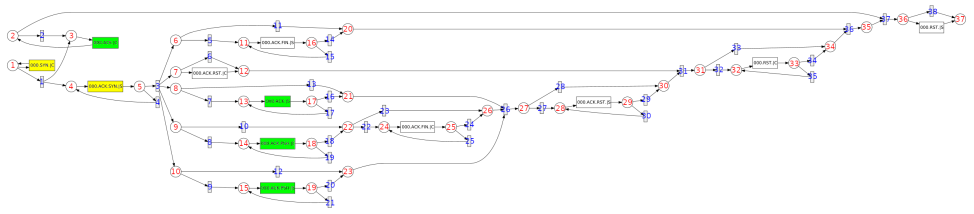


FIGURE 3.6: First process model mined with 100 traces (Appendices figure A.4).

000.SYN.|C	000.ACK.|C	000.ACK.SYN.|S	000.ACK.RST.|C
000.ACK.FIN.|S	000.ACK.|S	000.ACK.PSH.|C	000.ACK.PSH.|S
000.ACK.FIN.|C	000.ACK.RST.|S	000.RST.|C	000.RST.|S

TABLE 3.7: List of *event classes* in the first model with 100 traces.

- 000.ACK.|C and all actions between places P4 and P35 are concurrent. 000.RS T.|S is the next action in a sequential pattern.
- 000.ACK.FIN|S and all actions between P5 and P35 are in a concurrent pattern.
- 000.ACK.|S, all actions between P9 and P26, and 000.ACK.PSH.|S are in a concurrent pattern. 000.ACK.PSH.|C is followed by 000.ACK.FIN.|C are sequencial.
- 000.ACK.RST.|S and 000.RST.|C are in a sequential pattern.

	000.ACK.|C*					
		000.ACK.FIN.|S$^{*\#}$				
		000.ACK.RST.|C$^{\#}$				
000.SYN.|C*	000.ACK.SYN.|S*	000.ACK.|S$^{*\#}$		000.ACK.RST.|S$^{*\#}$	000.RST.|C$^{*\#}$	000.RST.|S$^{\#}$
		000.ACK.PSH.|C$^{*\#}$	000.ACK.FIN.|C$^{*\#}$			
		000.ACK.PSH.|S$^{*\#}$				

TABLE 3.8: The process table of the first model mined with 100 traces.

All potential traces are clearly displayed in Process Table 3.8. Since the calculation of the missing tokens and remaining tokens are not necessary, using a process table to identify potential traces is easier. Additionally, it provides a clearer picture of which event classes can repeat or be bypassed. As a result, it's simple to make a potential trace like the one shown below (Trace 3.5). In this trace 000.SYN.|C has repeated once and 000.ACK.RST.|C, 000.ACK.PSH.|C and 000.RST.|C were bypassed; all other event classes appeared once. This trace might not be possible to happen in real life, but it can be replayed on this process model. The shortest possible trace that is can be replayed on the process model is shown below. To create the shortest trace, it is necessary to determine which event classes cannot be bypassed (without '#'). Because 000.ACK.|C and 000.ACK.SYN|S are in parallel, Trace 3.6 is close to being a three-way handshake. The handshake is invalid if 000.ACK.|C is executed prior to 000.ACK.SYN.|S. Now, let's examine a possible trace (Trace 3.7) containing all observed event classes.

$$\begin{aligned}&START \rightarrow 000.SYN.|C \rightarrow 000.SYN.|C \rightarrow 000.ACK.|C \rightarrow 000.ACK.SYN.|S\\&\rightarrow 000.ACK.PSH.|S \rightarrow 000.ACK.|S \rightarrow 000.ACK.FIN.|C \rightarrow 000.ACK.RST.|S\\&\rightarrow 000.ACK.RST.|C \rightarrow 000.RST.|S \rightarrow END\end{aligned} \tag{3.5}$$

$$START \rightarrow 000.SYN.|C \rightarrow 000.ACK.|C \rightarrow 000.ACK.SYN.|S \rightarrow END \tag{3.6}$$

$$\begin{aligned}&START \rightarrow 000.SYN.|C \rightarrow 000.ACK.|C \rightarrow 000.ACK.SYN.|S \rightarrow 000.ACK.|S\\&\rightarrow 000.ACK.PSH.|C \rightarrow 000.ACK.PSH.|S \rightarrow 000.ACK.FIN.|C \rightarrow 000.ACK.RST.|S\\&\rightarrow 000.ACK.RST.|C \rightarrow 000.RST.|C \rightarrow 000.ACK.FIN.|S \rightarrow 000.RST.|S \rightarrow END\end{aligned} \tag{3.7}$$

Once again, the inductive miner has a high generality, but the trace is invalid. However, the termination stage is better in contrast to earlier examples because the data transmission packets are not mixed in this stage.

Figure 3.7 shows the second model that is mined from 100 cases with 13 different event classes observed from these 100 cases. The observed event classes are shown in Table 3.9. 000.ACK.PSH.FIN|S is the newly observed event class. The corresponding process table is shown in Table 3.10

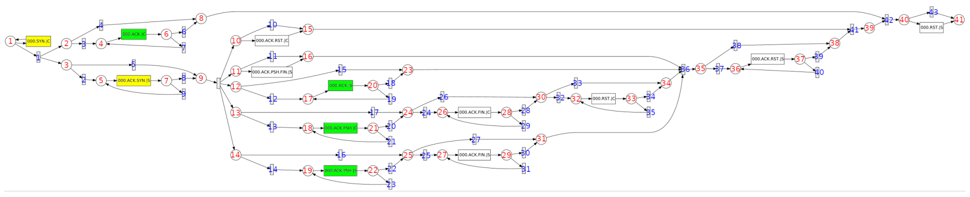


FIGURE 3.7: Second process model mined with 100 traces (Appendices figure A.5).

000.SYN.|C	000.ACK.|C	000.ACK.SYN.|S	000.ACK.RST.|C
000.ACK.PSH.FIN|S	000.ACK.|S	000.ACK.PSH.|C	000.ACK.PSH.|S
000.ACK.FIN.|C	000.ACK.FIN.|S	000.RST.|C	000.ACK.RST.|S
000.RST.|S	|	|	|

TABLE 3.9: List of *event classes* in the second process model mined with 100 traces.

000.SYN.|C*	000.ACK.|C$^{*\#}$					
	000.ACK.SYN.|S$^{*\#}$	000.ACK.RST.|C$^{\#}$				000.RST.|S$^{\#}$
		000.ACK.PSH.FIN|S$^{\#}$			000.ACK.RST.|S$^{*\#}$	
		000.ACK.|S$^{*\#}$				
		000.ACK.PSH.|C$^{*\#}$	000.ACK.FIN.|C$^{*\#}$	000.RST.|C$^{*\#}$		
		000.ACK.PSH.|S$^{*\#}$	000.ACK.FIN.|S$^{*\#}$			

TABLE 3.10: The process table of the second model mined with 100 traces.

The shortest possible trace that can be replayed on the process model is shown in Trace 3.8, and the possible trace with all observed event classes are shown in Trace 3.9. As the size of the event log grows larger, the accuracy of process models decreases and the generality increases.

$$START \rightarrow 000.SYN.|C \rightarrow END \qquad (3.8)$$

$$\begin{aligned}&START \rightarrow 000.SYN.|C \rightarrow 000.ACK.SYN.|S \rightarrow 000.ACK.FIN.|S \rightarrow \\ &000.ACK.PSH.|S \rightarrow 000.ACK.|S \rightarrow 000.ACK.RST.|C \rightarrow 000.ACK.PSH.|C \rightarrow \\ &000.ACK.FIN.|C \rightarrow 000.ACK.RST.|S \rightarrow 000.RST.|C \rightarrow 000.ACK.|C \rightarrow \\ &000.RST.|S \rightarrow END\end{aligned} \qquad (3.9)$$

3.1.3 Process Models Mined with 20k Traces

The process model depicted in Figure 3.8 was extracted from 20,000 cases and includes event classes not observed in previous models, such as 000.CWR.ECE.SYN.|C and 000.CWR.ACK.PSH.|C. Table 3.11 displays 19 observed event classes from the event log.

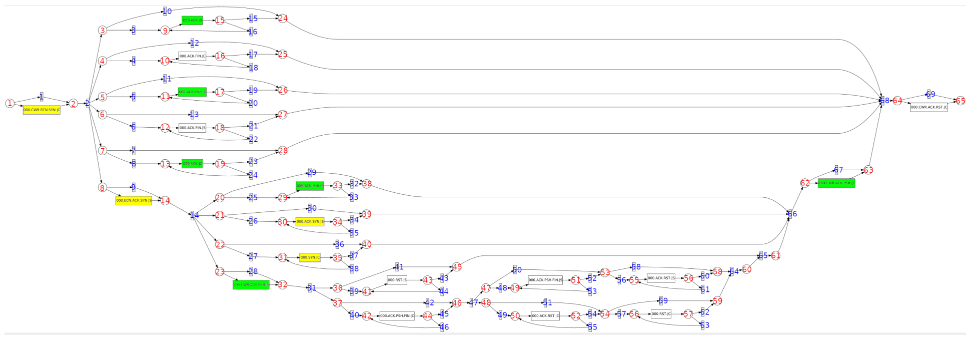


FIGURE 3.8: First process model mined with 20k traces (Appendices figure A.6).

000.CWR.ECE.SYN.|C	000.ACK.|S	000.ACK.FIN.|C	000.ACK.PSH.|C
000.ACK.FIN.|S	000.ACK.|C	000.ECE.ACK.SYN.|S	000.ACK.PSH.|S
000.ACK.SYN.|S	000.SYN.|C	000.CWR.ACK.PSH.|S	000.RST.|S
000.ACK.PSH.FIN.|C	000.ACK.PSH.FIN|S	000.ACK.RST.|C	000.ACK.RST.|S
000.RST.|C	000.CWR.ACK.PSH.|C	000.CWR.ACK.RST.|C	|

TABLE 3.11: List of event classes in the first model mined with 20k traces.

	000.ACK.|S$^{*\#}$						
	000.ACK.FIN.|C$^{*\#}$						
	000.ACK.PSH.|C$^{*\#}$						
	000.ACK.FIN.|S$^{*\#}$						
	000.ACK.|C$^{*\#}$						
		000.ACK.PSH.|S$^{*\#}$					
000.CWR.ECE.SYN.|C$^{\#}$		000.ACK.SYN.|S$^{*\#}$					000.CWR.ACK.RST.|C$^{*\#}$
		000.SYN.|C$^{*\#}$					
	000.ECE.ACK.SYN.|S$^{*\#}$		000.RST.|S$^{*\#}$			000.CWR.ACK.PSH.|C$^{\#}$	
		000.CWR.ACK.PSH.|S$^{\#}$	000.ACK.PSH.FIN.|C$^{*\#}$	000.ACK.PSH.FIN|S$^{*\#}$	000.ACK.RST.|S$^{*\#}$		
				000.ACK.RST.|C$^{*\#}$	000.RST.|C$^{*\#}$		

TABLE 3.12: the process model of the first model mined with 20k traces.

Since all event classes can be bypassed, as shown by the process table in Table 3.12, the shortest trace (Trace 3.10) that can be replayed by the process model is one that contains only the START and END tokens. The accuracy of this process model is much worse than any previous examples. A potential trace (Trace 3.11) that includes every event class that was discovered. The trace is sufficiently broad to encompass all observed event classes, but a real-world connection cannot possibly behave in this manner.

$$START \rightarrow END \qquad (3.10)$$

$$\begin{aligned}
&START \rightarrow 000.CWR.ECE.SYN.|C \rightarrow 000.ECE.ACK.SYN.|S \rightarrow 000.ACK.|S \rightarrow \\
&000.ACK.FIN.|C \rightarrow 000.ACK.FIN.|S \rightarrow 000.ACK.PSH.|C \rightarrow 000.ACK.|C \rightarrow \\
&000.CWR.ACK.PSH.|S \rightarrow 000.SYN.|C \rightarrow 000.RST.|S \rightarrow 000.ACK.PSH.FIN.|C \rightarrow \\
&000.ACK.PSH.FIN.|S \rightarrow 000.ACK.RST.|C \rightarrow 000.RST.|C \rightarrow 000.ACK.RST.|S \rightarrow \\
&000.ACK.SYN.|S \rightarrow 000.ACK.PSH.|S \rightarrow 000.CWR.ACK.PSH.|C \rightarrow \\
&000.CWR.ACK.RST.|C \rightarrow END
\end{aligned} \tag{3.11}$$

Figure 3.9 is the second process model mined from 20,000 traces. Similarly, these 20,000 traces contain 19 different event classes that are shown in Table 3.13.

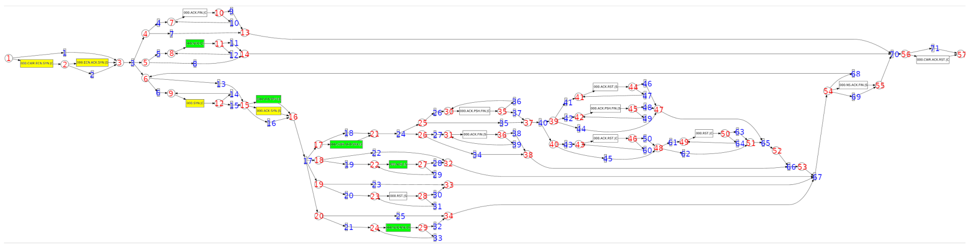


FIGURE 3.9: second process model mined with 20k traces (Appendices figure A.7).

000.CWR.ECE.SYN.|C	000.ECE.ACK.SYN.|S	000.ACK.FIN.|C	000.ACK.|C
000.SYN.|C	000.ACK.PSH.|C	000.ACK.SYN.|S	000.CWR.ACK.PSH.|C
000.ACK.|S	000.RST.|S	000.ACK.PSH.|S	000.ACK.PSH.FIN.|C
000.ACK.FIN.|S	000.ACK.RST.|S	000.ACK.PSH.FIN.|S	000.ACK.RST.|C
000.RST.|C	000.NS.ACK.FIN.|S	000.CWR.ACK.RST.|C	|

TABLE 3.13: List of *event classes* in the second model mined with 20k traces.

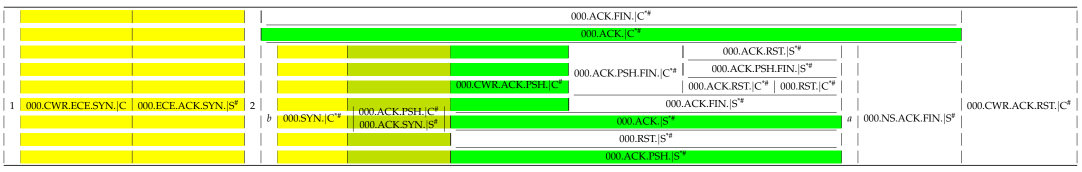


TABLE 3.14: The process table of the second model mined with 20k traces.

Table 3.14 presents, as always, the process table of this process model. The event classes 000.ACK.PSH.|C and 000.ACK.SYN.|S are involved in an exclusive-selection pattern that did not exist in previously mined models. The shortest trace is identical to Trace 3.10, which contains only the START and END tokens. In the following subsections, all process models mined with more cases have the same issue, so they will not be discussed again. It is impossible to create a trace that contains all event classes due to the exclusive-selection pattern. Trace 3.12 includes all event classes with the exception of 000.ACK.PSH.|C and 000.ACK.SYN.|S, which are exclusive. 000.ACK.SYN.|S is associated with the connection establishment stage, whereas 000.ACK.PSH.|C is associated with the data transmission stage. Therefore, the existence of a selection pattern between these two event classes is not possible. The occurrence of the 'selection' pattern could be due to the fact that some traces are ECN-capable and have ECE or CWR flags set during the handshake, which overlaps some event classes before and after the selection pattern.

$$START \rightarrow 000.CWR.ECE.SYN.|C \rightarrow 000.ECE.ACK.SYN.|S \rightarrow 000.ACK.|C \rightarrow 000.ACK.FIN.|C \rightarrow 000.SYN.|C \rightarrow 000.ACK.PSH.|C\ (000.ACK.SYN.|S) \rightarrow 000.CWR.ACK.PSH.|C \rightarrow 000.ACK.|S \rightarrow 000.ACK.PSH.|S \rightarrow 000.ACK.PSH.FIN.|C \rightarrow 000.ACK.RST.|S \rightarrow 000.ACK.RST.|C \rightarrow 000.ACK.FIN.|S \rightarrow 000.RST.|C \rightarrow 000.ACK.PSH.FIN.|S \rightarrow 000.RST.|S \rightarrow 000.NS.ACK.FIN.|S \rightarrow 000.CWR.ACK.RST.|C \rightarrow END \tag{3.12}$$

3.1.4 Process Models Mined with 100k Traces

Figure 3.10 depicts the model extracted from 100k cases, which includes 23 event names listed in Table 3.15. Also shown in Table 3.16 is the process table.

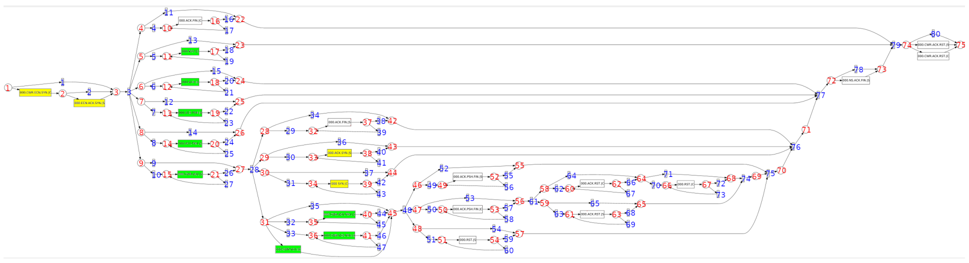


FIGURE 3.10: First model mined with 100k traces (Appendices figure A.8).

000.CWR.ECE.SYN.|C	000.ECE.ACK.SYN.|S	000.ACK.FIN.|C	000.ACK.|C
000.ACK.|S	000.ACK.PSH.|S	000.ACK.PSH.|C	000.CWR.ACK.|S
000.ACK.FIN.|S	000.ACK.SYN.|S	000.SYN.|C	000.CWR.ACK.PSH.|C
000.CWR.ACK.PSH.|S	000.CWR.ACK.|C	000.ACK.PSH.FIN.|S	000.ACK.PSH.FIN.|C
000.RST.|S	000.ACK.RST.|C	000.ACK.RST.|S	000.RST.|C
000.NS.ACK.FIN.|S	000.CWR.ACK.RST.|S	000.CWR.ACK.RST.|C	|

TABLE 3.15: List of *event classes* in the first model mined with 100k traces.

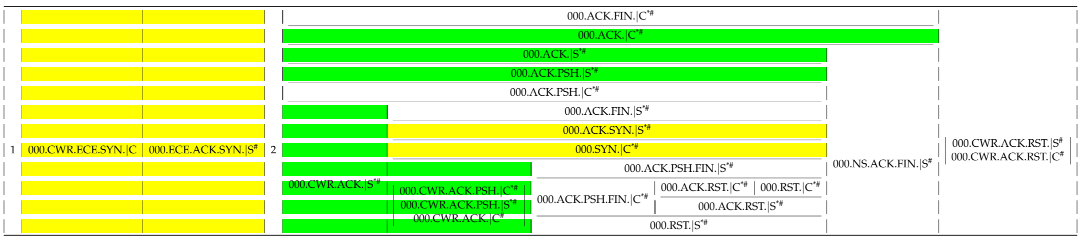


TABLE 3.16: The process table of the first model mined with 100k traces.

Despite Trace 3.13's apparent generality, the trace is invalid at all stages of a TCP connection. This model contains two patterns of exclusive selection. The first exclusive-selection pattern consists of 000.CWR.ACK.PSH.|C, 000.CWR.ACK.PSH.|S, and 000.CWR.ACK.|C, whereas the second exclusive selection pattern consists of 000.CWR.ACK.RST.|S and 000.CWR.ACK.|C; these event classes can occur alternatively within the stage, which is more accurate than the second model mined with 20k traces. All event classes may be bypassed, so the shortest trace is unavailable.

$$
\begin{aligned}
&START \rightarrow 000.CWR.ECE.SYN.|C \rightarrow 000.ECE.ACK.SYN.|S \rightarrow 000.CWR.ACK.|S \\
&\rightarrow 000.ACK.FIN.|C \rightarrow 000.ACK.FIN.|S \rightarrow 000.SYN.|C \rightarrow 000.ACK.SYN.|S \\
&\rightarrow 000.CWR.ACK.PSH.|C\ (000.CWR.ACK.PSH.|S, 000.CWR.ACK.|C) \\
&\rightarrow 000.ACK.PSH.FIN.|C \rightarrow 000.ACK.PSH.FIN.|S \rightarrow 000.ACK.RST.|C \\
&\rightarrow 000.RST.|C \rightarrow 000.ACK.RST.|S \rightarrow 000.RST.|S \rightarrow 000.ACK.|C \\
&\rightarrow 000.ACK.|S \rightarrow 000.ACK.PSH.|S \rightarrow 000.ACK.PSH.|C \rightarrow 000.NS.ACK.FIN.|S \\
&\rightarrow 000.CWR.ACK.RST.|S\ (000.CWR.ACK.RST.|C) \rightarrow END
\end{aligned}
\tag{3.13}
$$

3.1.5 Other Inductive Models

Both the second model mined with 100k cases and the final model mined with the complete set contain 23 event classes and are similar. In other words, both event logs containing 100k cases cover all observable event classes in the dataset. Because of the similarity of the process models extracted from the final three setups, the list of observed event classes will not be displayed. The inductive models are depicted in Appendix Figures A.9 and A.10. The process tables are displayed in 3.17 and 3.18.

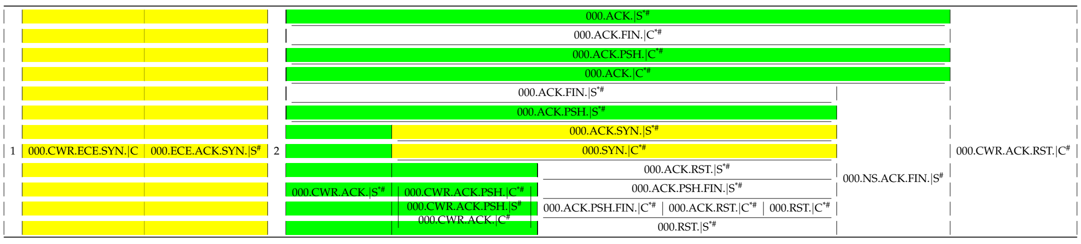


TABLE 3.17: The process table of the second model with 100k traces.

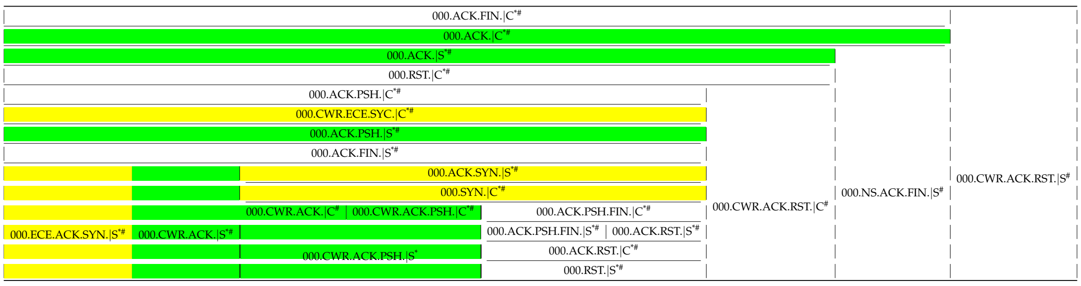


TABLE 3.18: The process table of the model mined with all traces.

The second model mined with 100k cases follows the same pattern of exclusive selection as the first model mined with 100k cases. The event classes 000.CWR.ACK.|C, 000.CWR.ACK.PSH.|C, and 000.CWR.ACK.PSH.|S have the combined patterns of exclusive-selection pattern and sequence pattern for the model mined with all cases; therefore, the model could only fire 00.CWR.ACK.|C followed by 000.CWR.ACK.PSH.|C, or fire 000.CWR.ACK.PSH.|S only. In table 3.18, it appears as a cell with three actions and a border separating 000.CWR.ACK.|C and 000.CWR.ACK.PSH.|C. In both process models, all event classes can be bypassed, so short traces are unavailable. Traces with all event classes will not be displayed because their characteristics are comparable to Trace 3.13.

3.2 Fuzzy Miner - Disco

Disco is a proprietary closed-source process mining tool. It uses directed graphs instead of Petri nets as process models. Due to the restrictions of Disco for academic purposes, the case limit for all event logs has been set to 50,000. Consequently, there are only eight configurations for process mining with Disco, which are listed below. The first six setups utilise the same event logs as the first six ProM setups. Event logs with 50,000 traces are sub-logs of the full-log.

1. First process model with 5 traces.
2. Second process model with 5 traces.
3. First process model with 100 traces.
4. Second process model with 100 traces.
5. First process model with 20k traces.
6. Second process model with 20k traces.
7. First process model with 50k traces.
8. Second process model with 50k traces.

Let us begin by examining an example of a process model mined with Disco. Figure 3.11 is a model of a process mined from 100 cases using Disco. If a link is shown between actions A and B in the process model, there must be a relation in the event log where *A* is followed by *B*, i.e., every relation is recorded in the process model assuming the model is the initial model. Adjacency matrix may be used to represent the directed graph. Table 3.19 displays the number of events/nodes and the number of edges.

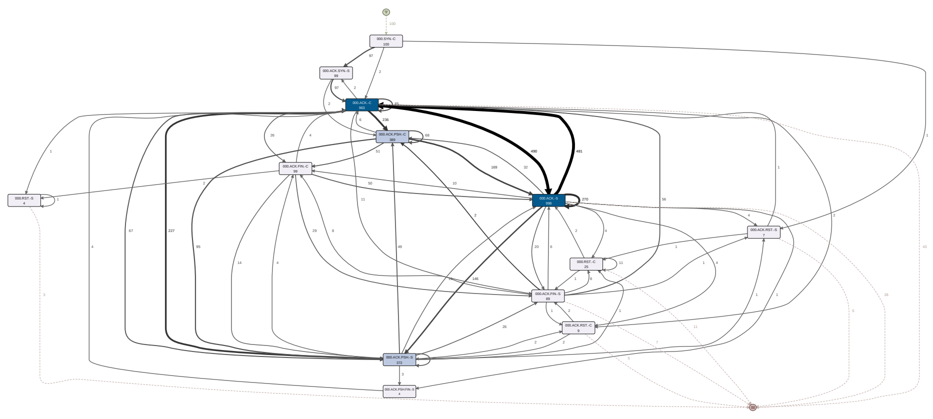


FIGURE 3.11: An example of fuzzy model (Appendices figure A.4).

0. 000.SYN.|C	1. 000.ACK.SYN.|S	2. 000.ACK.|C	3. 000.ACK.PSH.|C
4. 000.ACK.|S	5. 000.ACK.PSH.|S	6. 000.ACK.FIN.|C	7. 000.ACK.FIN.|S
8. 000.ACK.RST.|C	9. 000.ACK.PSH.FIN.|S	10. 000.RST.|C	11. 000.ACK.RST.|S
12. 000.RST.|S	13. START	14. END	|

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
100	99	963	389	998	372	99	89	9	4	25	7	4	100	100

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	97	2	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	97	2	0	0	0	0	0	0	0	0	0	0	0
2	0	2	85	236	490	67	26	11	2	0	0	0	1	0	43
3	0	0	6	68	169	95	51	0	0	0	0	0	0	0	0
4	0	0	481	32	270	146	10	20	4	1	4	4	0	0	26
5	0	0	227	49	11	48	4	26	2	3	1	1	0	0	0
6	0	0	4	0	50	14	0	29	0	0	0	0	2	0	0
7	0	0	56	2	6	0	8	0	1	0	8	1	0	0	7
8	0	0	0	0	0	2	0	2	0	0	0	0	0	0	5
9	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	2	0	0	1	0	0	11	0	0	0	11
11	0	0	1	0	0	0	0	0	0	0	1	0	0	0	5
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	3
13	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

TABLE 3.19: Event classes (top), event class count (middle) and adjacency matrix (bottom) of the process model.

The top table of Table 3.19 displays the ID number for each event class used in the tables that follow. The table in the middle represents the weight of nodes, or the counts of events with respect to the event class IDs. Here, the adjacency matrix displays all edge weights under each ID; for instance, the relation (000.SYN.|C, 000.ACK.SYN.|S) occurs 97 times in this event log, per row 0 column 1.

3.2.1 Process Models Mined with 5 Traces

Figure 3.12 depicts the first process model mined with 5 traces, and Table 3.20 contains all pertinent information. As some of the event logs utilised in the fuzzy mining setups are identical to those employed in the inductive mining setup, the observed event classes will not be displayed.

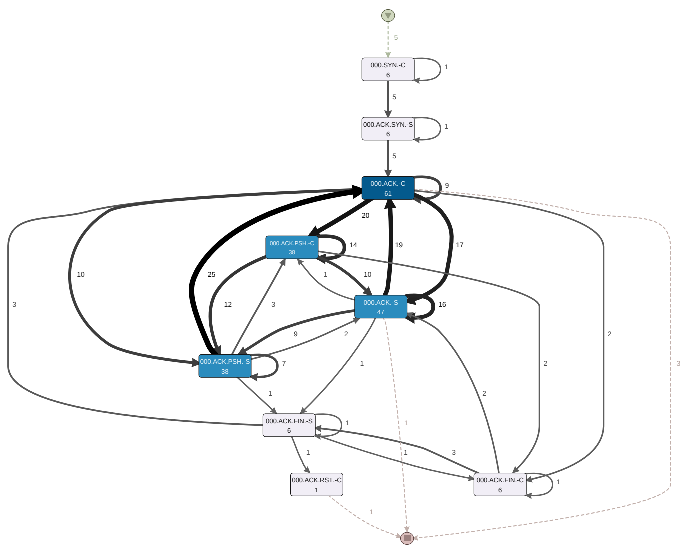


FIGURE 3.12: Process models mined with 5 traces.

0. 000.SYN.|C	1. 000.ACK.SYN.|S	2. 000.ACK.|C	3. 000.ACK.PSH.|C
4. 000.ACK.|S	5. 000.ACK.PSH.|S	6. 000.ACK.FIN.|C	7. 000.ACK.FIN.|S
8. 000.ACK.RST.|C	9. START	10. END	|

0	1	2	3	4	5	6	7	8	9	10
6	6	61	38	47	38	6	6	1	5	5

	0	1	2	3	4	5	6	7	8	9	10
0	1	5	0	0	0	0	0	0	0	0	0
1	0	1	5	0	0	0	0	0	0	0	0
2	0	0	9	20	17	10	2	0	0	0	3
3	0	0	0	14	10	12	2	0	0	0	0
4	0	0	19	1	16	9	0	1	0	0	1
5	0	0	25	3	2	7	0	1	0	0	0
6	0	0	0	0	2	0	1	3	0	0	0
7	0	0	3	0	0	0	1	1	1	0	0
8	0	0	0	0	0	0	0	0	0	0	1
9	5	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0

TABLE 3.20: Event classes (top left), event class count (top right) and adjacency matrix (bottom) of the process model.

A quick comparison of the fuzzy model with the inductive model mined from the same event log is made. The shortest trace complying with the fuzzy model is identical to Trace 3.1, as previously demonstrated with the inductive model; however, the fuzzy model strictly adheres to the three-way handshake. In other words, improper handshakes are not permitted by the fuzzy model, regardless of the data transmission stage and termination stage configurations. It is observed that 000.ACK.|C is the third packet in Trace 3.2, which is correct as 000.ACK.|C cannot be skipped. In fact, any event classes listed in the second column (parallel pattern) of process table 3.4, such as 000.ACK.|C or 000.ACK.FIN.|C, could be the third packet from the inductive model. The fuzzy model proves to be more accurate in this situation.

$$
\begin{aligned}
&START \rightarrow 000.SYN.|C \rightarrow 000.ACK.SYN.|S \rightarrow 000.ACK.|C \rightarrow 000.ACK.FIN|C \\
&\rightarrow 000.ACK.|S \rightarrow 000.ACK.PSH.|C \rightarrow 000.ACK.PSH.|S \rightarrow 000.ACK.FIN.|S \\
&\rightarrow 000.ACK.RST.|C \rightarrow END
\end{aligned} \tag{3.14}
$$

Trace 3.14 includes all observed event classes and has mixed established and termination stages due to the occurrence of 000.ACK.|C at both stages. Compared to the inductive model, this model clearly imposes stricter constraints on how packets should behave.

Figure 3.13 illustrates the second process model comprising five cases. Moving forward, only the process model itself will be displayed, as it offers improved readability compared to the table of counts and adjacency matrices. The event log contains a case in which the three-way handshake is incorrectly recorded as Sequence 3.15. The cause of the phenomenon is unknown, but it may occur during the packet-capturing phase. The process model demonstrates that it supports the standard handshake and Sequence 3.15, both of which are more accurate than the model mined with an inductive miner using the same event log.

$$000.SYN.|C \rightarrow 000.ACK.|C \rightarrow 000.ACK.SYN.|S \tag{3.15}$$

3.2.2 Other Process Models

Figure 3.14 shows the first fuzzy model mined with 100 cases. 2As more cases are mined, more event classes are observed, and more relations are observed in the event log. Some relations appear only once or twice, such as (000.ACK.|C, 000.ACK.RST.|C), indicating that the connection can be reset immediately after the

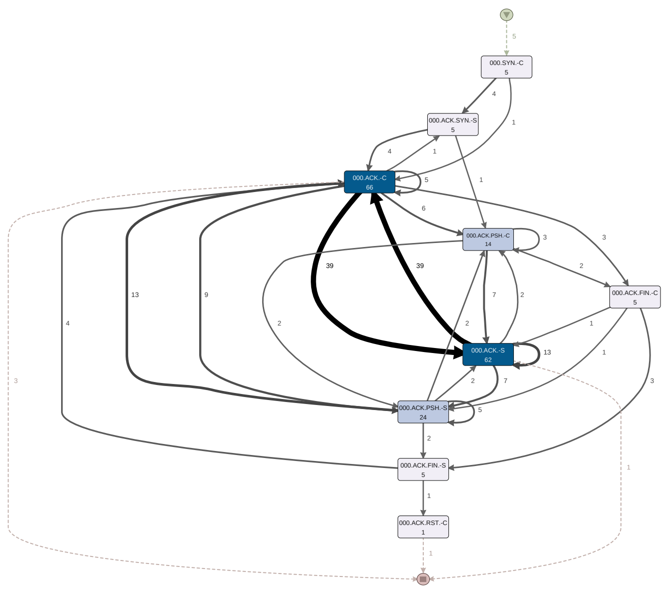


FIGURE 3.13: Second process model mined with 5 traces.

establishing stage. The case referenced in Sequence 3.15 is still present in this process model, but the case ID in the event log has changed. In the second event log containing five traces, the case ID is 154987; however, the case ID in this event log is 110821. That indicates the phenomenon occurred multiple times in the event log.

Due to the similarity between models, further discussion of other configurations will be omitted. The process models can be found in Appendix A. Sequence 3.15 occurred 611 times with a maximum of 50k traces. Evidently, this sequence emerges solely when examining the fuzzy model. Even in the presence of ECN-capable connections, fuzzy models disallow improper handshakes, highlighting the high accuracy of these models.

3.3 State Diagrams

The RFC TCP state transition diagram describes a TCP flow in general terms. The diagram depicts three phases of a TCP connection: the establishing, established, and closing phases. The diagram was previously presented in Figure 2.7. In this diagram, the establishing stage is represented by the three-way handshake, but the established stage is left undefined.

The information displayed in the state diagram from Bishop et al. [71] more closely resembles the process models, as the analysis is based on real traffic data observed through system calls. In this diagram, transitions with RST are coloured orange; those with SYN are coloured green; those with FIN set are coloured blue; and the rest are coloured black. Flags in the diagram, such as 'Arsf', have specific meanings: 'A' represents ACK set, while 'a' indicates ACK clear; 'R' denotes RST set, and 'r' signifies RST clear; 'S' corresponds to SYN set, and 's' means SYN clear; 'F' stands for FIN set, and 'f' refers to FIN clear. The transition rules of Figure 3.15 from [71] are detailed below.

- **close_3** Successful abortive close of a synchronised connection.

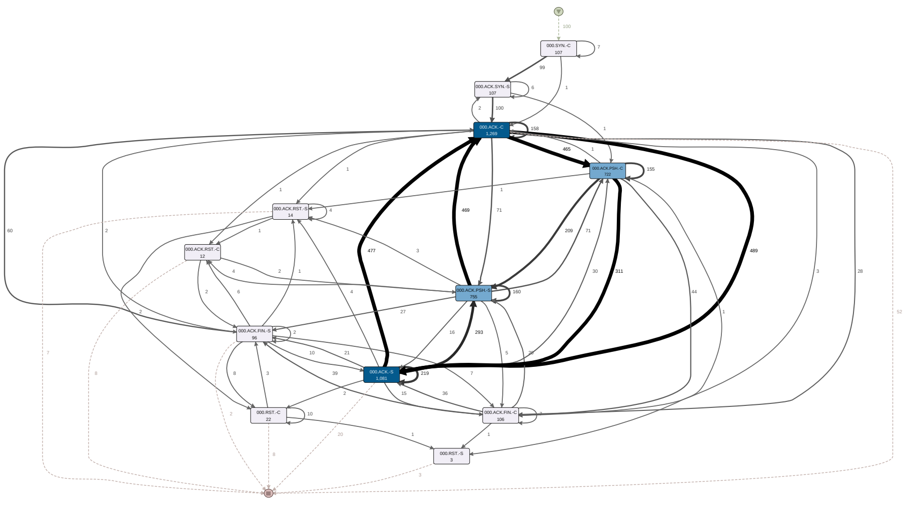


FIGURE 3.14: First process model mined with 100 traces.

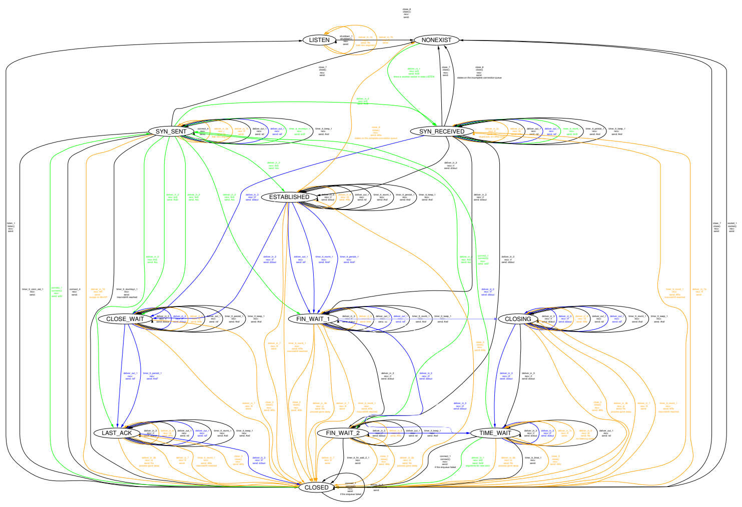


FIGURE 3.15: State diagram by Bishop et al. [71].

- **close_7** Closed the last file descriptor successfully for a connection in the CLOSED, SYN_SENT or SYN_RECEIVED states.
- **close_8** Closed the last file descriptor successfully for a listening TCP connection.
- **connect_1** Establishing connection by creating a SYN.
- **connect_4** Connection has pending error.
- **deliver_in_1** Passive open when receiving SYN, then send SYN.ACK.
- **deliver_in_1b** A listening connection receives and drops a bad datagram, then it either send back a RST segment or drop it.
- **deliver_in_2** Completion of active open in SYN SENT state by receiving SYN.ACK and send ACK, or simultaneous open in SYN SENT state by receiving SYN and send SYN.ACK.

- **deliver_in_2a** Receive a bad or boring datagram, then the RST is sent back or the packet is ignored if the state is in SYN_SENT connection.
- **deliver_in_3** Receive data, FINs, and ACKs in a connected state.
- **deliver_in_3b** Receive data after process has gone away.
- **deliver_in_3c** Receive stupid ACK or LAND DoS in SYN_RECEIVED state.
- **deliver_in_6** Receive and drop (silently) a sane segment that matches a CLOSED connection.
- **deliver_in_7** Receive RST and terminate non-CLOSED; LISTEN; SYN_SENT; SYN_RECEIVED; TIME_WAIT connection.
- **deliver_in_7a** Receive RST and terminate SYN_RECEIVED connection.
- **deliver_in_7b** Receive RST and ignore for LISTEN socket.
- **deliver_in_7c** Receive RST and ignore for SYN_SENT or TIME_WAIT connection.
- **deliver_in_7d** Receive RST and terminate SYN_SENT connection.
- **deliver_in_8** Receive SYN in non-CLOSED; LISTEN; SYN SENT; TIME WAIT state.
- **deliver_in_9** Receive SYN in TIME_WAIT state if there is no matching LISTEN socket or sequence number has not increased.
- **deliver_out_1** Common case TCP output.
- **listen_1** Successfully put connection in LISTEN state.
- **listen_1c** Successfully put connection in the LISTEN state from any non-CLOSED; LISTEN state on FreeBSD.
- **shutdown_1** Shut down read or write half of TCP connection.
- **connection_1** Successfully return a new file descriptor for a fresh connection.
- **timer_tt_2msl_1** 2MSL timer expires.
- **timer_tt_conn_est_1** Connection establishment timer expires.
- **timer_tt_fin_wait_2_1** FIN_WAIT 2 timer expires.
- **timer_tt_keep_1** Keepalive timer expires.
- **timer_tt_persist_1** Persist timer expires.
- **timer_tt_rexmt_1** Retransmit timer expires.
- **timer_tt_rexmtsyn_1** SYN retransmit timer expires.

	SYN.	ACK.	ACK.SYN.	ACK.RST.	FIN.	DATA	ACK.FIN.	RST.	START	END
SYN.	1	1	1	1				1		1
ACK.						1		1		
ACK.SYN.	1	1	1	1	1					
ACK.RST.										1
FIN.	1	1			1		1	1		
DATA	1	1		1	1		1	1		
ACK.FIN.	1	1			1					
RST.									1	1
START	1							1	1	
END										

TABLE 3.21: The adjacency matrix of state diagram by Bishop et al.

To facilitate the observation of traces in the manner of a 'process model', an adjacency matrix was created, as shown in Table 3.21, based on the diagram and transition rules previously discussed.

Compared to the RFC diagram, this state diagram has much better details. It includes more information about the transitions with RST and additional potential traces. Table 3.21 makes this clear, and it is similar to the process models discovered using Disco in that START cannot lead directly to PSH or END. The flags S and C, which designate whether a packet was sent from the server or the client, are included in the event names in our process models. It's unclear whether a packet is sent from the client or server because the state diagram maintains the states of the RFC TCP state diagram. Additionally, the data transmission stage does not include the PSH flag. Although the diagram has been converted to an adjacency matrix in order to compare it to process models, it is still difficult to compare because the state diagram emphasises states rather than packets. In addition, whereas our event logs created from PCAPs lack state information, the state diagram monitors connections at the system-call level and can determine the precise current state of a connection.

3.4 Model Comparisons

3.4.1 Process Models

Nine ProM-mined process models were displayed in section 3.1. Better accuracy is offered by process models that were mined from fewer cases (typically fewer than 20,000 cases). All process models that were mined for inductive models using more than 20,000 traces can have all actions bypassed. The generality of the process models mined with ProM results in more silent actions and parallel actions being added when different cases are mined, which increases the likelihood that an action will be bypassed or placed in the wrong order. Silent actions have the benefit of allowing a process model with a small size to better fit unobserved traces (traces not included in the event log). The drawback is that process models will deviate from the original dataset's accuracy, allowing traces that are not feasible in reality to obtain perfect fitness from the process model. Later on, the concept behind the fitness calculation will be covered.

For example, the trace below has the perfect fitness in all inductive models that are mined with more than 20,000 cases.

$$\text{START} \rightarrow \text{000.ACK.PSH.}|\text{S} \rightarrow \text{END.}$$

Since data transmission cannot begin before a connection has been made, this trace is not feasible in real life. These traces were regarded as incomplete traces and were filtered out, so they can not appear in the event log. Process models that were extracted from fewer traces, however, do not have this problem because they are more accurate. A more extreme example, such as

$$\text{START} \rightarrow \text{END},$$

is also an inappropriate trace.

A more favourable trade-off between accuracy and generality can be achieved with fuzzy models. If a specific edge exists in the process model, there must be a trace containing that particular relation. Although inductive models lack this property, it does not mean that process models discovered using Disco have no generality. The rare three-way handshake case observed in section 3.2.1 demonstrates generality, as it allows bypassing 000.ACK.SYN.|S by adding a path between 000.SYN.|C and 000.ACK.|C for these types of handshakes. The shortcut was not intentionally added to the process model like the inductive miner does; instead, it emerges naturally as the complexity of the paths increases.

Fuzzy models are not always superior, however. In business processes, predefined procedures or actions are typically executed according to rules and constraints. Additionally, extra attributes can be defined in business process logs, thereby improving the quality of the discovered inductive model. In the case of network packet data process mining, fuzzy models seem more applicable.

3.4.2 Process Models and State Diagrams

A comparison of the generality levels of various discussed models is presented in Table 3.22. A general model should allow normal, unobserved behaviours to pass conformance tests. Only rare normal flows present in actual network traffic data are compared. The examples illustrate some infrequent flows. Table columns display models, such as an inductive model mined with five traces, a fuzzy model mined with one hundred traces, and so on. The row shows rare cases expected to be observed in the models. If the case is observed in any process model or diagram, the cell is marked Y(es); otherwise, the cell is marked N(o). Following the inductive model in terms of generality is the fuzzy model. This outcome is anticipated due to the algorithmic design, where an inductive miner explores patterns such as concurrencies, resulting in numerous possible traces not present in the event log. In contrast, RFC diagrams are general descriptions of TCP states, with both diagrams limited to predefined stages like LISTEN, SYN-SENT, and ESTABLISHED, among others.

	Ind. 5t	Fuzzy 5t	Ind. 100t	Fuzzy 100t	Ind. 20kt	Fuzzy 20k		RFC Diag.	Bishop
PSH in established stage.	Y	Y	Y	Y	Y	Y		N	N
Duplicate pakages.	Y	Y	Y	Y	Y	Y		N	N
Reset during handskake.	N	N	Y	Y	Y	Y		N	Y
Send of receive SYN after estanblished.	N	N	N	N	Y	Y		N	N
Reset during closing	Y	N	Y	N	Y	Y		N	Y

TABLE 3.22: Generality Comparison.

	Ind. 5t	Fuzzy 5t	Ind. 100t	Fuzzy 100t	Ind. 20kt	Fuzzy 20k		RFC Diag.	Bishop
Data transmission without handshake	N	N	Y	N	Y	N		N	N
RST or FIN before handshake	N	N	Y	N	Y	N		N	N
Loop through Close and Established stages	N	N	Y	Y	Y	Y		N	N
Sending same packets infinitely without response from the other side	Y	Y	Y	Y	Y	Y		N	N

TABLE 3.23: Accuracy Comparison.

We contrast accuracy in Table 3.23. An accurate model that depicts normal behaviours should not allow abnormal traces to pass the compliance checking. We assume a trace is abnormal based on typical use cases because the behaviour of a flow can vary depending on the implementation of an application. In contrast to Table 3.22, we DO NOT anticipate any of the cases in Table 3.23 to fit any process model or diagram. We achieve greater accuracy with fuzzy models, and the diagrams are the most accurate. Cases involving infinite loops are a common issue with process models. Because diagrams adhere to predefined stages, looping through the close stage and the established stage cannot occur. Due to the fact that ACK typically occurs after receiving any packet, a link will be created between this packet and the ACK packet, causing loops between stages. For example, ACKs occur in both established and termination stages, and this is the cause of the common issue where packets from these two stages are mixed together in traces such as Trace 3.2 and Trace 3.4. The problem occurs in both inductive models and fuzzy models.

Accuracy is compared in Table 3.23. An accurate model depicting normal behaviours should not allow abnormal traces to pass the conformance checking. A trace is assumed to be abnormal based on typical use cases, as the behaviour of a flow can vary depending on an application's implementation. In contrast to Table 3.22, none of the cases in Table 3.23 are expected to fit any process model or diagram. Fuzzy models offer greater accuracy, while diagrams are the most accurate. Cases involving infinite loops are a prevalent issue with process models. As diagrams adhere to predefined stages, looping through the close stage and the established stage cannot occur. Since ACK generally occurs after receiving any packet, a link is created between this packet and the ACK packet, leading to loops between stages. For instance, ACKs occur in both established and termination stages, which causes the common issue where packets from these two stages mix together in traces such as Trace 3.2 and Trace 3.4. This problem occurs in both inductive models and fuzzy models.

On the basis of this observation, the question can be posed: what exactly can process mining extract from event logs? Clearly, process mining relies heavily on the identification of transitions in the event log. In our case, without prior knowledge of the TCP protocol and pre-processing the packet data before adding them to the event log, the information of the sender (C/S) and the stages will be lost, resulting in a greater number of loops and a greater degree of analysis difficulty. Tracking the flow and identifying the state of each packet is a further enhancement to our current event log construction procedure, as it provides us with one more attribute that prevents looping.

3.5 Conformance Checking

The effectiveness of process mining in detecting anomalies in network data is now examined. A process model will be mined following [34], and conformance checking will be applied to the traces to be audited. Initially, the inductive miner is used to extract the process model from a dataset consisting solely of normal network traffic. Then, using the fitness score measurement, attempts are made to identify anomalous traces that do not conform to the normal process model. Fitness is measured through conformance checking by replaying traces on process models, with the calculations described as:

$$fitness(t,N) = \frac{1}{2}\left(1-\frac{m}{c}\right)+\frac{1}{2}\left(1-\frac{r}{p}\right) \tag{3.16}$$

and

$$fitness(L,N) = \frac{1}{2}\left(1-\frac{\Sigma_{t\in L}L(t)\times m_{N,t}}{\Sigma_{t\in L}L(t)\times c_{N,t}}\right)+\frac{1}{2}\left(1-\frac{\Sigma_{t\in L}L(t)\times r_{N,t}}{\Sigma_{t\in L}L(t)\times p_{N,t}}\right), \tag{3.17}$$

where Equation 3.16 represents conformance checking on the trace level, and Equation 3.17 is on the log level. In this context, t represents the trace, N is the process model, and L is the event log. m is the number of missing tokens, and r is the number of remaining tokens. The concepts of missing tokens and remaining tokens were introduced in Section 3.1. The consumed tokens, c, are the tokens consumed by firing all actions, and the produced tokens, p, are the tokens produced (fired) by all actions. When calculating log-level fitness, it essentially counts m, c, r, and p throughout the entire event log, not just for a single trace as in trace-level fitness calculations. [72] provides details on how conformance checking is conducted.

Using the inductive models mined with a 20k event log (the same event log from setup 5), conformance checking is performed, and the fitness score is calculated with ProM. A score of 1 indicates a perfect alignment of all cases with the process model; otherwise, the score will be less than 1. The average fitness scores (log-level fitness scores) for multiple setups are shown in Table 3.24. Conformance checking is performed with various categories, including another normal log and all attacks. The normal column represents the fitness score of an additional subset, distinct from the one used to generate the model. All preprocessed data will be accessible on Zenodo[1].

Normal	BruteForce	DoS	Heartbleed	CoolDisk	Dropbox	PortscanNmap	Web	Botnet_ARES	Port_Scan_DDos
1	1	1	1	1	1	1	1	1	1

TABLE 3.24: Fitness scores of conformance checking.

Since all cases display a perfect model fit, identifying anomalies becomes impossible. Some related works mentioned in Section 2.4 from Chapter 2 achieve good accuracy using conformance checking; however, the limitation of these related works lies in mining the normal model from only dozens or hundreds of cases, leading to a relatively accurate model. Nevertheless, network flows can be considerably more complex; thus, mining a process model from deviated flows will result in a much more general model. As long as the flows are being routed on the network, whether they are attacks or normal traffic, they always comply with TCP, and attacks typically do not exploit the protocol itself. The belief is that the frequency distribution of transitions is more critical for anomaly detection in IDS. An anomaly may not be regarded as a specific transitional change but rather as the global frequency structure.

3.6 Summary

This chapter recalls the second research question. It has shown several process models mined with Disco and ProM with explanations of the observations. Also, a novel method of presenting process models in process tables was shown. The focus of this chapter is to compare process models and state diagrams within the context of network packet data process mining and investigates the efficacy of conformance checking in detecting network data anomalies.

Process models mined with ProM, particularly inductive models, are susceptible to generality issues, which can result in actions being bypassed and infeasible traces being permitted. Fuzzy models provide a better balance between precision and generalisation. However, fuzzy models may continue to struggle with accuracy issues arising from infinite loops in process models. When compared to state diagrams such as RFC diagrams, which adhere to predefined TCP stages, both inductive and

[1]https://doi.org/10.5281/zenodo.6646875

fuzzy models struggle to achieve the optimal balance between accuracy and generality. To examine the efficacy of process mining in detecting anomalies in network data, a process model is extracted from a dataset containing normal network traffic using the inductive miner. In the context of network packet data process mining, fuzzy models appear more applicable than inductive models. As both normal traffic and attacks conform to the TCP protocol, conformance checking is limited in its ability to detect anomalies.

Chapter 4

Online Process Mining Algorithm

In Chapter 3, issues encountered so far are found and discussed, leading to the belief that utilising PM and conformance checking directly for network intrusion detection proves to be inefficient. Upon comparing models and observing their behaviour, it becomes apparent that detecting anomalies is challenging due to the generality of the process model. In addition, an explanation is provided as to why employing PM for network traffic anomaly detection is ineffective. This chapter proceeds to review and introduce a novel algorithm capable of performing packet-level detection, inspired by process mining.

4.1 Existing Problem

4.1.1 Existing NIDS Approaches

In Chapter 1, a few NIDS-related techniques were mentioned, with more examples to be provided in this section. NIDS can be broadly categorised into two groups: online NIDS and offline NIDS. A significant amount of research is being conducted in the area of online approaches, which are the methods of interest. Subsequently, flow-level and packet-level NIDS are categorised from the online NIDS. Disregarding the requirement for computational resources, packet-level NIDS undoubtedly emerges as the optimal method for delivering genuine online detection.

The majority of packet-level NIDS techniques, however, have the following shortcomings. Systems that operate at the packet level typically lack the ability to decrypt encrypted data. Examples include the previously introduced Snort [8], which uses predefined rules to check for intrusions, and the case-based agent [73], which uses case-based reasoning on packet XML data. Another example is the technique [74], which converts bytes of packets to grayscale images before using a hierarchical network structure for classification. In order to detect encrypted traffic, it is necessary to analyse the time series data. Some techniques use recurrent neural networks for intrusion detection, for example, [28] uses Long short-term memory (LSTM) to categorise a time series of raw packets. For smaller network devices, the resources required to run and train LSTM may not always be readily available. Employing flow-level data serves as an alternative approach; however, this level of detection does not align with the desired outcome that this research is looking for.

Certain methods update the analytical values for temporal historical packets with analytical data, eliminating the need for recurrent neural networks or time-series packet data as input. The analytical data can be used as input for classifiers not designed for time-series data, such as multi-layer perceptrons (MLP), which prove to be more effective as they already encode historical data at a higher level of abstraction. An excellent example is given in [17], which employs damped incremental statistics for feature extraction.

Packet-level IDS with the use of historical information addresses the issue of encrypted data. However, a challenge arises that may result in poor performance for attacks such as DDoS attacks. A DDoS attack transpires due to the volume of connections rather than a single connection. Each individual connection may appear completely normal, so if the focus is solely on the information provided by a single connection, the attack will not be identifiable. This problem can be resolved by encoding the global flow information. [17] proposed using 2D-statistics to analyse the TX and RX traffic of a connection, asserting that the computational complexity is $O(1)$. It is believed that, given a wire with n packets passing through, the complexity of the damped incremental statistics method is $O(n)$. The 2D-statistics are between TX and RX traffic. Modifying the algorithm to perform the 2D-statistics between flows should result in a complexity of $O(mn)$, assuming there are m flows in the wire.

4.1.2 Process Mining

Process mining is intended for business process model discovery and analysis, and is capable of encoding the global process structure. The capacity to observe global process structure is considered crucial for detecting attacks such as botnet, DoS/DDoS, and brute force. The activities are recorded in the event log that can be used for process mining in the future. The collection of these activities could take days or even weeks, and the event log is then used to determine the process model. This may be appropriate for offline intrusion detection, but it cannot be used traditionally for online anomaly detection. Online conformance checking is available in later research [75] but the anomaly detection is limited to conformance checking. Tests were conducted in Chapter 3 as a naive approach to applying process mining to network data and attempting anomaly detection with conformance checking. The outcomes proved to be less than promising.

From the last chapter, it is observed that fuzzy models handle the trade-off between accuracy and generality more effectively, and process models mined with larger event logs typically permit illegal traces to pass conformance checking. However, regardless of the event log size, using process mining directly for intrusion detection does not offer a truly online solution, and even the offline performance might not be promising. In general, smaller models exhibit greater accuracy. Since an offline approach does not align with the desired outcome, further testing of offline NIDS with conventional process mining has not been pursued.

Performance poses another problem. The inductive miner requires more resources than the fuzzy miner, and even if a sliding window can be maintained that shifts for every incoming packet p to achieve online process mining and merely mine process models from a relatively small sliding window, performance will be a drawback for the inductive miner. Additionally, compared to the graph produced by the fuzzy miner, the Petri net is more difficult to convert to other formats that can be used by classifiers and anomaly detectors. Furthermore, another potential issue with this online process mining approach using the sliding window is identified, regardless of the performance concern. Assuming the size of the sliding window is 500, flows with more than 500 packets will not be fully covered by the sliding window. Mining a process model with an incomplete trace is not expected, and incomplete traces are treated as outliers during data cleaning.

4.2 Dataset

A more recent CSE-CIC-IDS2018 dataset [25] has become available during the development of the new method. This dataset is comparable to the IDS2017 dataset, but it encompasses more data and updated attack categories. B-profiles and M-profiles are utilised to generate network traffic data. Unlike the local computers used to produce the IDS2017 data, the IDS2018 dataset is generated within the AWS cloud environment.

B-profile was previously mentioned in Section 2.5.1 in the context of how it creates benign traffic data that appears to be natural. On the other hand, attack data was produced using the M-profile system. When describing a cyber-attack scenario, the M-profile makes an effort to be as specific as possible. Compilers and autonomous agents would be used to analyse and run these scenarios. Based on the implemented network topology, a scenario is built for each attack, and the attack is carried out from one or more machines outside the target network. Various sets of operating systems are installed, including Ubuntu, Microsoft Windows servers from 2012 and 2016, and Microsoft Windows operating systems (Windows 8.1 and Windows 10). In the M-profile, seven attack scenarios have been implemented. The specifications of the dataset are listed below.

1. **Wednesday 2018.02.14**
   Brute Force attacks including
   - FTP (10:32 – 12:09);
     Attacker: 18.221.219.4
     Victim: 18.217.21.148
   - SSH (14:01 – 15:31).
     Attacker: 13.58.98.64
     Victim: 18.217.21.148
2. **Thursday 2018.02.15**
   DoS attacks including
   - GoldenEye (09:26 – 10:09);
     Attacker: 18.219.211.138
     Victim: 18.217.21.148
   - Slowloris (10:59 – 11:40).
     Attacker: 18.217.165.70
     Victim: 18.217.21.148
3. **Friday 2018.02.16**
   DoS attacks including
   - SlowHTTPTest (10:12 – 11:08).
     Attacker: 18.219.193.20
     Victim: 18.217.21.148
   - Hulk (13:45 – 14:19).
     Attacker: 18.219.193.20
     Victim: 18.217.21.148
4. **Tuesday 2018.02.20**
   DDoS attacks including

- LOIC-HTTP (10:12 – 11:17);
- LOIC-UDP (13:13 – 13:32).
  Attacker: 18.218.115.60, 18.219.9.1, 18.219.32.43, 18.218.55.126, 52.14.136.135, 18.219.5.43, 18.216.200.189, 18.218.229.235, 18.218.11.51, 18.216.24.42
  Victim: 18.217.21.148

5. **Wednesday 2018.02.21**
   DDoS attacks including
   - LOIC-UDP (10:09 – 10:43);
   - DDOS-HOIC (14:05 – 15:05).
     Attacker: 18.218.115.60, 18.219.9.1, 18.219.32.43, 18.218.55.126, 52.14.136.135, 18.219.5.43, 18.216.200.189, 18.218.229.235, 18.218.11.51, 18.216.24.42
     Victim: 18.218.83.150
6. **Thursday 2018.02.22**
   Brute Force attacks including
   - Web (10:17 – 11:24);
   - XSS (13:50 – 14:29).
     Attacker: 18.218.115.60
     Victim: 18.218.83.150

   SQL Injection attacks (16:15 - 16:29).

   Attacker: 18.218.115.60

   Victim: 18.218.83.150
7. **Friday 2018.02.23**
   Brute Force attacks including
   - Web (10:03 – 11:03);
   - XSS (13:00 – 14:10).
     Attacker: 18.218.115.60
     Victim: 18.218.83.150

   SQL Injection attacks (15:05 - 15:18).

   Attacker: 18.218.115.60

   Victim: 18.218.83.150
8. **Wednesday 2018.02.28**
   Infiltration attacks (10:50 - 12:05, 13:42 - 14:40).

   Attacker: 13.58.225.34

   Victim: 18.221.148.137
9. **Thursday 2018.03.01**
   Infiltration attacks (9:57 - 10:55, 14:00 - 15:37, 14:00 - 15:37).

   Attacker: 13.58.225.34

   Victim: 18.216.254.154

10. **Friday 2018.03.02**
    Botnet attacks (10:11 - 11:34, 14:24 - 15:55).

    Attacker: 18.219.211.138

    Victim: 18.217.218.111, 18.222.10.237, 18.222.86.193, 18.222.62.221, 13.59.9.106, 18.222.102.2, 18.219.212.0, 18.216.105.13, 18.219.163.126, 18.216.164.12

4.3 Algorithm Design

In general, the aim is to harness the global flow discovery potential of process mining in an online manner. As mentioned previously, process mining analyses the relationships between packets in flows and encodes the global flow structure into the process model rather than analysing the flows themselves. Before discussing the algorithm, it is necessary to revisit the definitions of transitions and event classes, which may differ slightly from those used in conventional process mining.

4.3.1 Definition

Given a sequence of processed packets, P, a *transition* in P is defined as a pair of consecutive packets (p_i, p_j) within a flow in P. The *transition* was previously referred to as the precedence relation, and P can be treated as the event log. The processed packets are those that have been captured and had unnecessary data removed, such as numerical values. All packets will be referred to as processed packets in the subsequent context.

A *trace* is a series of packets that belong to the same TCP flow.

Here is an example, a series of packets $P = \langle p_1, p_2, p_3, p_4, p_5 \rangle$ is given with two traces, t_1 and t_2. Flow $t_1 = \langle p_1, p_3, p_5 \rangle$ and flow $t_2 = \langle p_2, p_4 \rangle$. Two transitions arise for t_1: (p_1, p_3) and (p_3, p_5), while a single transition for t_2: (p_2, p_4). Although packets p_1 and p_2 are consecutive, they are not considered a transition since they belong to separate flows.

An *event class* $ec(p)$ of a packet p is the concatenation of enabled flags of a packet followed by an indicator. e.g. 000.SYN.|C, where the last character is an indicator that indicates either the packet is sent from the client or the server. In this case, C indicates the packet is sent from the client. Although every packet is unique (similar to an event), every packet has its corresponding *event class*. Examples of event classes were explored in Chapter 3.

A *transition type* (p_i, p_j) consists of a pair of corresponding event classes $(ec(p_i), ec(p_j))$. Transition types may also be referred to as *relations*. For instance, (000-.SYN.|C, 000.ACK.SYN.|S) represents a *relation* signifying that a packet with the SYN flag enabled is succeeded by a consecutive packet with both ACK and SYN flags enabled. Frequencies of transitions are measured under transition types. For instance, a count of 5 for (000-.SYN.|C, 000.ACK.SYN.|S) indicates the occurrence of 5 transitions relating to 10 packets sharing the same transition type.

4.3.2 Sliding Window

The dataset is analysed, revealing that the IDS2018 dataset of normal traffic data comprises 23 possible event classes that can be observed. It is assumed that the majority of flag combinations are encompassed by these 23 event classes. For default rear case handling, all other packets with flag combinations not observed in the dataset can be categorised as OTHERS.

In total, there are 26 event classes, consisting of 23 event classes from the observation and 3 default classes (START, END, and OTHERS). START and END serve as two tokens marking the beginning and end of a trace. Consequently, assuming every class can be paired with other classes, there will be $26^2 = 676$ possible relations. This finite set of 26 event classes can be found in Table 4.1.

000.SYN.|C	000.ACK.SYN.|S	000.ACK.|C	000.ACK.PSH.|C
000.ACK.PSH.|S	000.ACK.FIN.|C	000.ACK.|S	000.ACK.FIN.|S
000.ACK.RST.|C	000.ACK.RST.|S	000.RST.|S	000.ACK.PSH.FIN.|S
000.RST.|C	000.CWR.ECE.SYN.|C	000.ECE.ACK.SYN.|S	000.NS.ACK.FIN.|S
000.ACK.PSH.FIN.|C	000.CWR.ACK.PSH.|C	000.CWR.ACK.|C	000.CWR.ACK.|S
000.CWR.ACK.PSH.|S	000.CWR.ACK.RST.|S	000.CWR.ACK.RST.|C	START
END	OTHERS		

TABLE 4.1: Possible event classes used.

Our proposed online algorithm operates as follows. Given a sequence of packets P, the algorithm outputs the sequence (stream) of frequencies of relations observed in the last l packets (for some l), organised in the form of an adjacency matrix (26x26). Here, the frequencies of relations observed in the last l packets are process models. In other words, the algorithm outputs a series of adjacency matrices.

To decrease computational complexity in the experiments, a sliding window and a cap of 500 packets l were employed to calculate the frequency of transitions. This constraint on window size is based on numerous experiments and can be substituted with any other number. The only hyperparameter in this algorithm is the size of the sliding window, which will be discussed later. Preferring models with higher accuracy, a small starting number of events was chosen, with 500 being deemed suitable. This decision stems from the analysis of over 4 million events from typical network traffic, revealing an average of 94 events per trace. Consequently, 500 events usually encompass 5 traces, which is considered a suitable starting point. Process models discovered using 5 traces have been examined, and they demonstrate higher accuracy over generality. In theory, during attacks such as DDoS, 500 events could cover more cases due to a larger number of connections and relatively fewer packets transmitted per connection. Conversely, activities like file transferring may be covered by fewer cases.

The sliding window starts from p_1 and covers $\langle p_i \rangle_{i=1}^{l} = \langle p_1, p_2, p_3 \dots p_{l-1}, p_l \rangle$, and the frequency of transitions A' is calculated as A/l, then the window will be shifted one step further which covers $\langle p_i \rangle_{i=2}^{l+1}$. This process results in a sequence $\langle A'_i \rangle_{i=1}^{n-l+1}$ and each A'_i is a snapshot of a process model with l events. Here, n is the total number of packets. In process mining, the events are instances of event classes. The process of producing A'_i is shown in Figure 4.1.

Labelled data is needed for training and testing, so labels from packets are passed to A'. If an incoming packet originates from a flow labelled as a specific attack, the final A'_i will be labelled as that type of attack, such as DDoS or Brute Force. For example, P_{500} and P_n are packets that come from anomalous flows and the corresponding output A_1 and $A_{n-500+1}$ are also labelled.

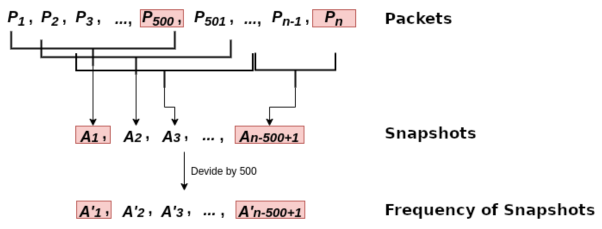

P1, P2, P3, ..., P500, P501, ..., Pn-1, Pn
Packets
A1, A2, A3, ..., An-500+1
Snapshots
Devide by 500
A'1, A'2, A'3, ..., A'n-500+1
Frequency of Snapshots

FIGURE 4.1: Diagram of the sliding window. Packets P_{500} and P_n belong to traces that are marked as attacks, therefore, A_1 and $A_{n-500+1}$ are also labelled as attacks for training classifiers.

4.3.3 Temporal Event Table

The temporal event table (TET) is a data structure employed to address the information loss of the initial transition of each window. Each trace begins with an initial token SOT (start of trace). Assuming a trace with 1000 events exists and the sliding window size is only 500, a DFG is generated for each window. Since the window is not large enough to encompass the entire trace, assigning the SOT token becomes uncertain. Clearly, this token cannot be assigned for each window; if it is simply removed, the information about the start of a trace is lost. The same issue applies to the EOT (end of trace) token. To better illustrate the concept, consider a smaller-scale experiment with an event log of 10 events involved in 2 traces. Trace $\Diamond = \langle a_\Diamond, b_\Diamond, c_\Diamond, d_\Diamond, c_\Diamond, e_\Diamond \rangle$ and trace $\Box = \langle a_\Box, b_\Box, a_\Box, e_\Box \rangle$ are the two traces. With 5 event classes and a sliding window size of 3, the process is demonstrated in Figure 4.2.

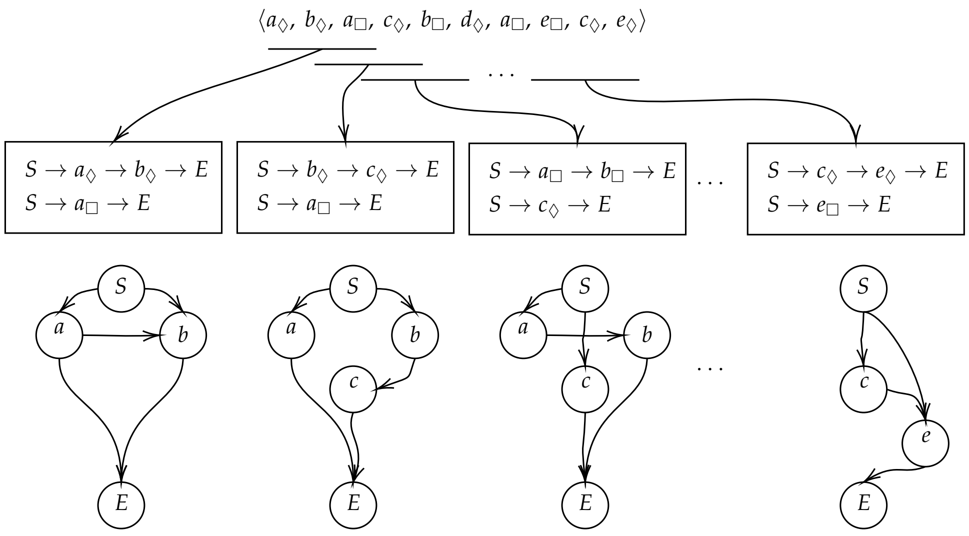

$\langle a_\Diamond, b_\Diamond, a_\Box, c_\Diamond, b_\Box, d_\Diamond, a_\Box, e_\Box, c_\Diamond, e_\Diamond \rangle$
...
$S \to a_\Diamond \to b_\Diamond \to E$
$S \to a_\Box \to E$
$S \to b_\Diamond \to c_\Diamond \to E$
$S \to a_\Box \to E$
$S \to a_\Box \to b_\Box \to E$
$S \to c_\Diamond \to E$
...
$S \to c_\Diamond \to e_\Diamond \to E$
$S \to e_\Box \to E$
S
a
b
c
e
E
...

FIGURE 4.2: Online process mining without the TET. S stands for the SOT token and E stands for the EOT token. Same event classes will likely occur many times within the window which causes the weights of corresponding nodes or edges to increase, however, the weights are not reflected in this figure.

The solution here is that the last event class that is outside the window of each flow t_i was kept in the TET, so the SOT and EOT tokens will only be set at the beginning and of a particular TCP flow instead of each sliding window. As the last event class is known, the relation can be mined even if the window has already passed the previous event. The process with the TET is shown in Figure 4.3.

In other words, the original process mining takes into account the entire event log P, and P generates a single relatively large adjacency matrix A. However, this is unsuitable for online processing; consequently, the last event class of a trace is retained temporarily until the trace ends, and process model generation is restricted to only use l packets.

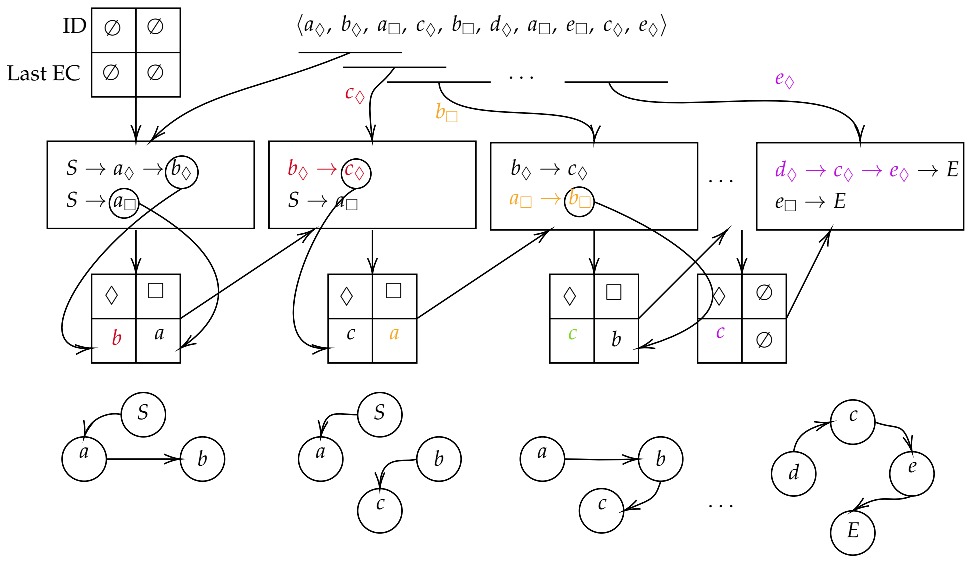

ID
Last EC
$\langle a_\Diamond, b_\Diamond, a_\Box, c_\Diamond, b_\Box, d_\Diamond, a_\Box, e_\Box, c_\Diamond, e_\Diamond \rangle$
$S \to a_\Diamond \to b_\Diamond$
$S \to a_\Box$
$b_\Diamond \to c_\Diamond$
$S \to a_\Box$
$b_\Diamond \to c_\Diamond$
$a_\Box \to b_\Box$
$d_\Diamond \to c_\Diamond \to e_\Diamond \to E$
$e_\Box \to E$
$c_\Diamond$
$b_\Box$
$e_\Diamond$
S
a
b
c
d
e
E

FIGURE 4.3: Online process mining with the TET. S stands for the START token and E stands for the END token. Same event classes will likely occur many times within the window which causes the weights of corresponding nodes or edges to increase, however, the weights are not reflected in this figure.

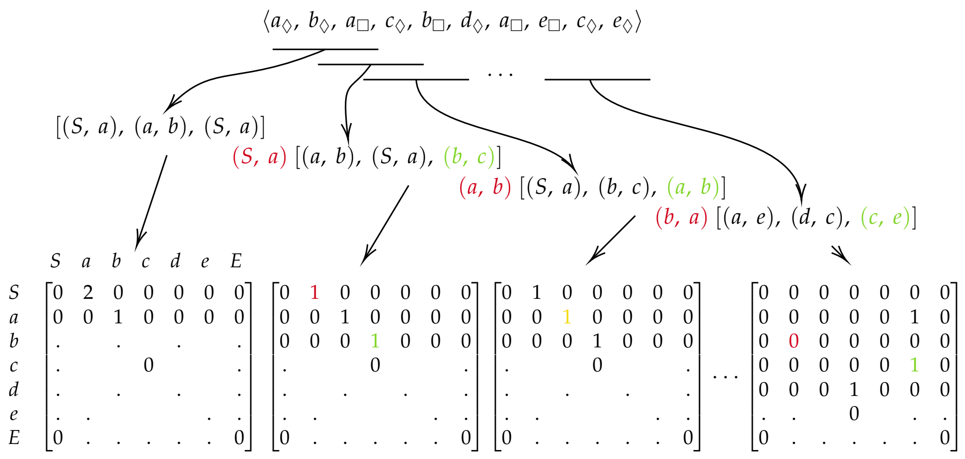
The TET is a hash table that contains a key entry which is normally the case ID, and the event class is stored under the corresponding case ID. In Figure 4.3, the TET is initialised with the first window, and the Last EC value is updated according to the latest packet. For each incoming packet, the system uses the case ID from the incoming packet as a key to retrieve the previous event. For example, $c_\Diamond$ has case ID $\Diamond$ and it exists in TET where the event class record in TET for $\Diamond$ is b; the system creates the relation (b, c) and then updates the record in TET to c. The values will be empty when the traces reach the end, and the key/ID in the TET for this trace will be destroyed. TET collaborates with the transition buffer data structure, which is going to be introduced in the next section.

One could argue that it would be simpler to only include the SOT token before specific event classes are detected, or to simply insert the token before the trace. Indeed, it is always required for identifying a new trace; however, the primary reason for using TET is that it is necessary to track the case regardless of the method employed, and searching for the event with the same case ID in the window is inefficient. Furthermore, if a window is too small to cover any event of a case in between, the track of this case will be lost. This may happen when a connection pauses packet transferring at some point, and other connections send more than 500 packets before this connection continues. Second, the TET is extensible, allowing us to retain older events for each case and, potentially, all historical transitions of a flow. In addition, some flows may be completely outside of the current window within a given period of time, making it imperative to record the last event class name for such flows.

Another argument is that the SOT and END tokens can be removed. The purpose of using the token is similar to the use of the start of sequence (SOS) and end of the sequence (EOS) tokens in natural language processing (NLP) [76]–[78]. As the flows always begin with SYN and end with FIN or RST, which are equivalent to SOS and EOS tokens, they can be removed for network data. However, in order to maintain consistency, the SOT tokens are retained. The END token is omitted for the purpose of implementation simplicity, but it is recommended to include the END token when working with non-NIDS systems whose traces end with a wider variety of event classes.

4.3.4 Transition Queue

The frequency of the transitions needs to be updated for each new packet, with calculations based solely on the last l packets. The aim is to measure the frequency by counting the incoming relations into an adjacency matrix A. The transition queue, also referred to as the transition buffer, is an l-sized list that stores historical transitions from multiple cases and adheres to the FIFO (first in, first out) principle. Here, l represents the sliding window's size. Utilising the transition queue allows for the management and identification of obsolete transitions that should not be included in the current process model, as opposed to employing resource-intensive mining process models for each sliding window. Figure 4.4 illustrates the structure of the transition queue.



$\langle a_\diamond, b_\diamond, a_\square, c_\diamond, b_\square, d_\diamond, a_\square, e_\square, c_\diamond, e_\diamond \rangle$
$[(S, a), (a, b), (S, a)]$
$(S, a)\ [(a, b), (S, a), (b, c)]$
$(a, b)\ [(S, a), (b, c), (a, b)]$
$(b, a)\ [(a, e), (d, c), (c, e)]$
S a b c d e E

FIGURE 4.4: The transition queue. The count of transition (edge weight) decrease by 1 when the transition go out the queue (red colour), the count increase by 1 otherwise (green colour). Yellow colour means the count is increased and decreased at the same time.

Similar to the TET, the transition buffer will be initialised with the first sliding window. For each incoming event, the new transition that is created based on the retrieved value from TET will be pushed into the queue and the adjacency matrix's count for this transition will increase by 1. In the interim, the oldest transition will be removed from the queue and its transiting count in the adjacency matrix will be reduced by 1. Using this method, adjacency matrices A are generated dynamically based on the previous matrix. Close inspection reveals that the adjacency matrices do not correspond to the graphs shown in Figure 4.3. The matrices have one additional transition compared to the graphs in Figure 4.3, which is the leftmost transition within the queue. These additional transitions are perfectly acceptable, so their retention in the experiment is justified.

4.3.5 Pseudocode

NIDS utilises Algorithm 1 as the pseudocode for the online process mining technique. The pseudocode is divided into two sections: the first focuses primarily on the initialisation phase, while the second addresses the online processing of each packet. A brief examination of the pseudocode will now be conducted.

The first part starts from line 1 to line 21. The input data P is a list of packets, and the parameter defines the window size that will be used. For the production environment, the packets are captured in real-time. For initialisation in the production environment, the processor will wait until the first l packets are collected, and later packets will be processed online. There are two outputs, a list of the transition matrices and a list of passed-through labels. The list of labels is used for training and testing the classifiers only, and it is not needed in the online detection phase after the classifiers or anomaly detectors are trained. The key used in TET is essentially the socket information, but the case ID is also applicable when using this algorithm offline for training and testing dataset generation. The *current_flags* variable is the current event class name.

Algorithm 1: pseudocode of packet preprocessing.

```
1  in $P = [n]$;                                                    /* load $n$ packets */
2  param $l = 500$;                                            /* define the window size */
3  $A = [26\ by\ 26]$;                                       /* a $26 * 26$ adjacency matrix */
4  $list\_A' = [\,]$;                                             /* initialise list of $A'_i$ */
5  $list\_attacks = [\,]$;                                     /* initialise list of attacks */
   /* a dictionary where the key is the concatenation of IPs and Ports
      ($''IP_1 : PORT_1|IP_1 : PORT_1''$) of hosts, and the value is the flags of the previous packet
      */
6  $tet = \{\}$;
7  $buffer = [l]$;                          /* an FIFO buffer that keeps the last $l$ transitions */
   /* initialise with first $l$ packets                                                    */
8  for $i = 1$ to $l$ do
       /* check if the packet belong to any existing flow                                  */
9      if $''IP_1 : PORT_1|IP_2 : PORT_2''$ in $tet.key()$ or $''IP_2 : PORT_2|IP_1 : PORT_1''$ in $tet.key()$ then
           /* count the transition into $A$                                                 */
10         $A[tet[''IP_1 : PORT_1|IP_2 : PORT_2''], current\_flags] += 1$;
           /* update the state of the flow to the current flags into the dict               */
11         $tet[''IP_1 : PORT_1|IP_2 : PORT_2''] = current\_flags$;
12         push $current\_flags$ into $buffer$;
           /* check whether TCP flow terminates                                             */
13         if $''FIN''$ in $current\_flags$ or $''RST''$ in $current\_flags$ then
14             remove key $''IP_1 : PORT_1|IP_2 : PORT_2''$ from $tet$;
15         end
       /* check if new TCP flow starts                                                     */
16     else if $''SYN''$ in $current\_flags$ and $''ACK''$ not in $current\_flags$ then
17         $A[tet[''START''], current\_flags] += 1$;
18         push $current\_flags$ into $buffer$;        /* push the transition (event) into buffer */
19     end
20 end
21 append $A/l$ to $list\_A'$;             /* append the frequency of transitions into the list */
```

After the initialisation, the first transition frequency matrix A'_1 and the first label of P_l are available. The transition frequency matrices are basically the normalised weights of edges of the process models and the maximum possible frequency of a transition is 1. Following a similar principle, the algorithm now updates all data structures and list for each incoming packet with a computational complexity of $O(n)$. Ultimately, the outputs are obtained and prepared for use in machine learning for intrusion detection. These outputs are also suitable for alternative techniques, such as signal processing.

```
for $i = l + 1$ to $n$ do
    if $"IP_1 : PORT_1|IP_2 : PORT_2"$ in $tet.key()$ or $"IP_2 : PORT_2|IP_1 : PORT_1"$ in $tet.key()$ then
        $A[$pop $buffer] -= 1$;        /* sub 1 for transition that went outside the window */
        $A[tet["IP_1 : PORT_1|IP_2 : PORT_2"], current\_flags] += 1$;
        $tet["IP_1 : PORT_1|IP_2 : PORT_2"] = current\_flags$;
        push $current\_flags$ into $buffer$;
        /* check whether TCP flow terminates                                    */
        if $"FIN"$ in $current\_flags$ or $"RST"$ in $current\_flags$ then
            remove key $"IP_1 : PORT_1|IP_2 : PORT_2"$ from $tet$;
        end
    /* check if new TCP flow starts                                             */
    else if $"SYN"$ in $current\_flags$ and $"ACK"$ not in $current\_flags$ then
        $A[$pop $buffer] -= 1$;
        $A[tet["START"], current\_flags] += 1$;
        push $current\_flags$ into $buffer$;
    end
    append $A/l$ to $list\_A'$;
    /* Attack IP is from the labelled data                                      */
    if $Attack\ IP$ in $current\_flags$ then
        append $i$ to $list\_attacks$
    end
end
Output: $list\_A'$;
Output: $list\_attacks$;
```

4.4 Feature Generation

4.4.1 Processing

Before employing the online feature generation algorithm, a discussion on data preparation is necessary. The event log construction procedure remains identical to the method standardised in Chapter 2. There are 14 different types of network traffic that utilise TCP, including normal traffic and 13 distinct types of attacks. For the IDS 2018 dataset, normal data can only be separated from each day, unlike the IDS 2017 dataset, which designated a specific day for collecting normal data. Event logs are generated for several normal PCAP files before merging them into a larger event log. The PCAP files, specifically chosen due to containing only regular data, are listed below.

- capPC1-172.31.64.121.pcap
- capPC1-172.31.65.117.pcap
- capPC1-172.31.66.90.pcap
- capPC1-172.31.66.111.pcap
- capWIN-J6GMIG1DQE5-172.31.65.104.pcap
- capWIN-J6GMIG1DQE5-172.31.65.115.pcap
- capWIN-J6GMIG1DQE5-172.31.67.62.pcap
- capWIN-J6GMIG1DQE5-172.31.67.109.pcap
- UCAP172.31.69.25-part1.pcap

The rest of the event log preparations are similar. Unique IDs are assigned to each category, and such IDs will be used as the label of the data for identification purposes. The list of different types of attacks and their IDs are listed below.

0. Normal
1. FTP-BruteForce
2. SSH-Bruteforce
3. DoS-GoldenEye
4. DoS-Slowloris
5. DoS-SlowHTTPTest
6. DoS-Hulk
7. DDoS-LOIC-HTTP
8. DDoS-HOIC
9. Brute Force-Web
10. Brute Force-XSS
11. SQL Injection
12. Infiltration
13. Botnet

4.4.2 Generated Features

Following the preparation, each event log undergoes processing with the online feature generation algorithm. Prior to the classification step involving machine learning, it is worthwhile to determine if any observations can be made based only on the frequencies of transitions. The direct output of the algorithm, without additional processing, can be viewed as the fluctuation of frequencies.

77 out of 676 transition types are given in Figure 4.5. There are 10 transition types in the first chart and 10 transition types in the second chart, and so on. Upon closer inspection, a red bar can be observed on the x-axis, signifying the occurrence of an attack. Roughly the attack happens between 50,000th to 310,000th packets. The transition (a, b) is written as a-b in the charts.

The change in frequency during the attack is easily discernible in the majority of the charts, or it could be said that it is visible across most of the transitions. During the attack, there begins the stabilisation in the transition frequency patterns seen in the first three charts. When looking at these three charts, it is not immediately clear whether the frequency has become lower or higher on average. Charts 4 and 6 show lower frequencies that are stable throughout the attack, whereas chart 7 shows higher frequencies that are present throughout the attack. It is a sign that correlated information has been extracted, which is important for the classification step that comes after this.

4.5 Signature-based Detection

4.5.1 Algorithms

- **Multi-layer perceptions (MLP)**
  A multi-layer perceptron, or MLP, is a type of artificial neural network (ANN)

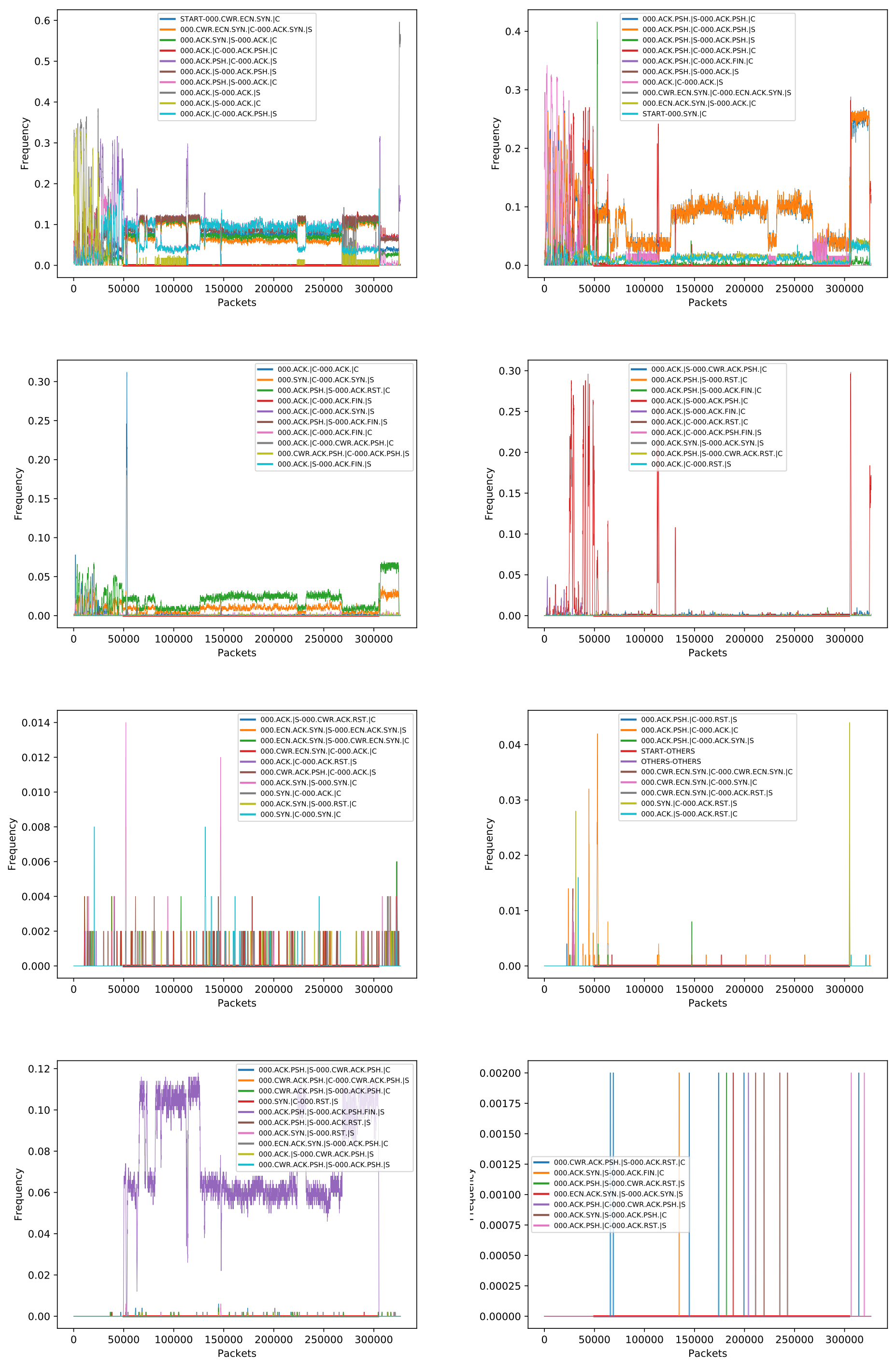


FIGURE 4.5: The chart show frequency fluctuation under Botnet attacks.

made up of numerous perceptrons due to its multiple layers. It comprises an input layer that receives the signal, an output layer that predicts based on the input, and any number of hidden layers that function as the MLP's computational units. The input layer processes the signal. MLPs with just one hidden layer can approximate any continuous function. Multilayer perceptrons are commonly employed in supervised learning problems, where the objective is to teach perceptrons to predict correlations in labelled data sets.

Since MLP is not designed for time-series prediction, each A' is trained individually. The 26^2-sized A' matrices are flattened into 676-dimensional vectors. For the MLP, four layers of neural networks are used: the first layer has 676 neurons; the second and third layers have 128 neurons each; and the final layer contains 2 neurons with the Softmax function for 2-class classification. For multi-class classification, the last layer is modified to include 14 neurons. Softmax ensures that the output ranges between 0 and 1, and that the sum of the probability distribution equals 1.

- **Long short-term memory (LSTM)** [79]
  Long Short-Term Memory (LSTM) networks are a form of recurrent neural network (RNN) with the ability to learn order dependency in time-series prediction tasks. The fact that long short-term memory (LSTM) is one of the first implementations to overcome the technical challenges and deliver on the promise of recurrent neural networks may be one of the contributing factors to the success of LSTMs. These challenges are known as the vanishing gradients problem and the expanding gradients problem, and they are ones that have been encountered by earlier RNNs.

  LSTM is a recurrent neural network model suitable for time-series data, in this case, classifying the A' matrices in a time-series manner. Utilising 2 layers of LSTMs, each with 128 units, the first layer has an input dimension of $(676, s)$. The number 676 represents the dimension of the flattened A', and s refers to the time steps. The final layer is a fully-connected layer with 2 neurons for output, utilising the Softmax function. Likewise, the last layer is modified to include 14 units for multi-class classification. Experiments are conducted with different time step values, $s \in \{50, 100, 250\}$.

- **Convolutional neural network (CNN)** [80]
  Convolutional Neural Network has produced ground-breaking breakthroughs in various pattern recognition-related domains, including image processing and speech recognition. The most advantageous element of CNNs is their ability to reduce the number of parameters in ANN. This success has led both academics and developers to use bigger models to perform complex tasks, which was not achievable with traditional ANNs; The most significant assumption regarding issues handled by CNN is that they should not have spatially dependent characteristics. In a face detection application, for instance, there is no need to consider the location of faces within the photos. The primary challenge lies in detecting them regardless of their position in the provided photographs [81].

  Since CNN is designed for image recognition, it naturally accepts 2D-tensor inputs. Thus, matrices A' are fed as single-channel images, with the first convolution layer having an input shape of $(26, 26, 1)$ and 16 units, followed by a max-pooling layer. The second convolution layer has 32 units and is also followed by a max-pooling layer. Then the output from the max-pooling layer is flattened and connected to a fully connected dense layer with 32 neurons. The setup for the layer of output is the same as the previous models.

- **K-nearest Neighbours (KNN)** [82]
  KNN is a non-parametric supervised learning approach and one of the simplest machine learning algorithms that assess the similarity between new data and existing instances and classifies the new case into the category that is most similar to existing categories. KNN saves the dataset, and at the time of classification, it calculates the distance between the stored dataset and the new data. It then selects k nearest neighbours from the stored dataset, which may be used to identify the category of the new data based on the categories of the neighbours. KNN is typically employed for classification issues, although it may also be applied to regression. Inputs are flattened into 676-dimensional vectors, as in the previous models, and a k value of 5 is used for both binary classification and multi-class classification.

The rationale for using these classifiers is not to test and compare which classification model performs the best but to demonstrate that the output from the online process mining technique is versatile enough for various types of classifiers. The selection of hyper-parameters, including the number of neurons, layers of perceptions, training epochs, and so on, is based on similar considerations. Other hyper-parameter selections have been tested, and the ones listed above provide relatively better trade-offs between training speed, performance, and hardware limitations in the specific experimental setups. Other hyper-parameter selections have not been listed but one can tune the hyper-parameters based on the provided selections according to their specific environment.

The dataset is split into 80% training data and 20% testing data. All classifiers have been trained using the same datasets. Excluding KNN, all neural networks are trained for 10 epochs with a batch size of 64. The loss function used for all neural networks is Categorical Cross Entropy Loss, and the optimiser employed is Adam (Adaptive Moment Estimation) [83]. Adam excels in various tasks and architectural types by combining the benefits of RMSProp and AdaGrad (Adaptive Gradient Algorithm and Adaptive Gradient Algorithm, respectively). Class weights have also been applied to unbalanced datasets.

4.5.2 Binary Classification Results

All classifiers are trained through a 5-fold cross-validation procedure and the results are listed below. The numbers of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) in the confusion matrices are the sums of numbers from different folds. These metrics will be used to calculate the F1-scores (sometimes also called F-score). The F1-Score is comprised of both Precision and Recall. Precision and Recall are the two most prevalent measures that take class imbalance into consideration. The F1-score is calculated as follows (Equation 4.1 - 4.3):

$$Precision = \frac{TP}{TP + FP}, \tag{4.1}$$

$$Recall = \frac{TP}{TP + FN}, \tag{4.2}$$

$$F1\ score = 2 \times \frac{Precision * Recall}{Precision + Recall}\ . \tag{4.3}$$

It must be mentioned that only the F1-score is directly present in the subsequent context. The rationale for this is twofold: firstly, the F1-score is a metric that encapsulates the performance of a classification model by taking into account both precision and recall. Calculated as the harmonic mean of precision and recall, it ranges from 0 to 1, with higher values signifying superior performance [84]. Secondly, this approach simplifies the outcome, allowing the reader to more easily perceive the performance difference. As the experiments aim to demonstrate that the feature generation functions as anticipated rather than measuring the performance of various machine learning models, displaying solely the F1-score can be a more suitable choice. All confusion matrices are included, thus other metrics can always be calculated based on these matrices.

- FTP BruteForce (Table 4.2 and Table 4.3)
  All classifiers performed admirably and returned high true positive and true negative rates for FTP Brute-force attacks. The results for all classifiers are close to 1 on the F-score.

MLP		Actual	
		Attack	Normal
Classifier	Attack	385966	0
	Normal	754	386720

LSTM-50		Actual	
		Attack	Normal
Classifier	Attack	384723	0
	Normal	1836	386561

LSTM-100		Actual	
		Attack	Normal
Classifier	Attack	385292	0
	Normal	1269	386559

LSTM-250		Actual	
		Attack	Normal
Classifier	Attack	385513	328
	Normal	1051	386228

KNN		Actual	
		Attack	Normal
Classifier	Attack	385988	0
	Normal	732	386720

CNN		Actual	
		Attack	Normal
Classifier	Attack	385985	0
	Normal	735	386720

TABLE 4.2: Confusion matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.9990	0.9976	0.9984	0.9982	0.9991	0.9990

TABLE 4.3: F1-score for each classifier.

- SSH-BruteForce (Table 4.4 and Table 4.5)
  The scores for SSH BruteForce attacks across all classifiers are lower than those for FTP BruteForce dangers, but they are still quite good. The fact that LSTM-250 has a lower score than other classifiers is probably attributable to the challenges associated with training longer LSTMs.

MLP		Actual	
		Attack	Normal
Classifier	Attack	476774	27
	Normal	23097	500102

LSTM-50		Actual	
		Attack	Normal
Classifier	Attack	476767	1
	Normal	23093	499819

LSTM-100		Actual	
		Attack	Normal
Classifier	Attack	476894	60
	Normal	23067	499659

LSTM-250		Actual	
		Attack	Normal
Classifier	Attack	481756	99804
	Normal	18482	399638

KNN		Actual	
		Attack	Normal
Classifier	Attack	478656	5027
	Normal	21293	495024

CNN		Actual	
		Attack	Normal
Classifier	Attack	477156	45
	Normal	23004	499795

TABLE 4.4: Confusions matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.9763	0.9764	0.9763	0.8907	0.9732	0.9764

TABLE 4.5: F1-score for each classifier.

- DoS GoldenEye (Table 4.6 and Table 4.8)
  Except for the LSTM-100 classifier, all classifiers performed well for the DoS GoldenEye attack, and KNN is clearly the best.

MLP		Actual	
		Attack	Normal
Classifier	Attack	192862	2303
	Normal	30681	221239

LSTM-50		Actual	
		Attack	Normal
Classifier	Attack	214492	5334
	Normal	8863	218031

LSTM-100		Actual	
		Attack	Normal
Classifier	Attack	213057	135009
	Normal	10309	88345

LSTM-250		Actual	
		Attack	Normal
Classifier	Attack	207091	5617
	Normal	16277	217735

KNN		Actual	
		Attack	Normal
Classifier	Attack	221243	5769
	Normal	2300	217773

CNN		Actual	
		Attack	Normal
Classifier	Attack	208511	9966
	Normal	15032	213576

TABLE 4.6: Confusion matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.9212	0.9680	0.7457	0.9498	0.9821	0.9434

TABLE 4.7: F1-score for each classifier.

- DoS-Slowloris (Table 4.8 and Table 4.9)
  For DoS Slowloris attacks, all instances have high scores that are near 1; nevertheless, LSTM classifiers do less well than other classifiers, with LSTM-250 performing the worst.

MLP		Actual	
		Attack	Normal
Classifier	Attack	103875	340
	Normal	812	104348

LSTM-50		Actual	
		Attack	Normal
Classifier	Attack	92423	20992
	Normal	11891	83334

LSTM-100		Actual	
		Attack	Normal
Classifier	Attack	92927	21082
	Normal	11390	83241

LSTM-250		Actual	
		Attack	Normal
Classifier	Attack	92959	42006
	Normal	11362	62313

KNN		Actual	
		Attack	Normal
Classifier	Attack	103977	406
	Normal	710	104282

CNN		Actual	
		Attack	Normal
Classifier	Attack	103927	331
	Normal	761	104356

TABLE 4.8: Confusion matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.9945	0.8490	0.8513	0.7770	0.9947	0.9948

TABLE 4.9: F1-score for each classifier.

- DoS-SlowHTTPTest (Table 4.10 and Table 4.11)
  DoS SlowHTTPTest attacks are easily detected by all classifiers. Despite the fact that LSTMs received slightly lower scores, all classifiers have similar accuracy.

MLP	Actual		
		Attack	Normal
Classifier	Attack	210417	47
	Normal	683	211053

LSTM-50	Actual		
		Attack	Normal
Classifier	Attack	202544	0
	Normal	8343	210873

LSTM-100	Actual		
		Attack	Normal
Classifier	Attack	208409	181
	Normal	2478	210692

LSTM-250	Actual		
		Attack	Normal
Classifier	Attack	202739	0
	Normal	8149	210872

KNN	Actual		
		Attack	Normal
Classifier	Attack	210474	3
	Normal	626	211097

CNN	Actual		
		Attack	Normal
Classifier	Attack	210434	3
	Normal	666	211097

TABLE 4.10: Confusion matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.9983	0.9798	0.9937	0.9803	0.9985	0.9984

TABLE 4.11: F1-score for each classifier.

- DoS-Hulk (Table 4.12 and Table 4.13)
  Detecting the DoS Hulk attacks is not an easy task for all classifiers, especially for the LSTM-50 classifier. Although the results are not impressive, these classifiers still manage to detect some attacks.

MLP	Actual		
		Attack	Normal
Classifier	Attack	287896	115
	Normal	211924	500065

LSTM-50	Actual		
		Attack	Normal
Classifier	Attack	456604	400543
	Normal	42787	99746

LSTM-100	Actual		
		Attack	Normal
Classifier	Attack	326995	19125
	Normal	172774	480786

LSTM-250	Actual		
		Attack	Normal
Classifier	Attack	352932	112292
	Normal	147080	387376

KNN	Actual		
		Attack	Normal
Classifier	Attack	354493	105628
	Normal	145962	393917

CNN	Actual		
		Attack	Normal
Classifier	Attack	288750	426
	Normal	211602	499222

TABLE 4.12: Confusion matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.7309	0.6732	0.7731	0.7313	0.7381	0.7314

TABLE 4.13: F1-score for each classifier.

- DDoS-LOIC-HTTP (Table 4.14 and Table 4.15)
  MLP produces a high F1-score for DDoS LOIC HTTP attacks, whereas other classifiers are less competitive. LSTMs have the lowest scores among classifiers.

MLP	Actual		
		Attack	Normal
Classifier	Attack	360232	1150
	Normal	140054	498564

LSTM-50	Actual		
		Attack	Normal
Classifier	Attack	432556	204700
	Normal	67546	294878

LSTM-100	Actual		
		Attack	Normal
Classifier	Attack	443068	300217
	Normal	56602	199793

LSTM-250	Actual		
		Attack	Normal
Classifier	Attack	414506	199542
	Normal	85274	300358

KNN	Actual		
		Attack	Normal
Classifier	Attack	404745	64547
	Normal	95062	435646

CNN	Actual		
		Attack	Normal
Classifier	Attack	366844	5945
	Normal	133173	494038

TABLE 4.14: Confusion matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.9968	0.7606	0.7129	0.7443	0.8353	0.8406

TABLE 4.15: F1-score for each classifier.

- DDOS-HOIC (Table 4.16 and Table 4.17)
  Similarly, for DDoS HOIc attacks, MLP has a relatively high F1-score, whereas other classifiers have similar scores.

MLP		Actual	
		Attack	Normal
Classifier	Attack	294767	9528
	Normal	205367	490338

LSTM-50		Actual	
		Attack	Normal
Classifier	Attack	328087	100441
	Normal	171242	399910

LSTM-100		Actual	
		Attack	Normal
Classifier	Attack	342524	21977
	Normal	157666	477513

LSTM-250		Actual	
		Attack	Normal
Classifier	Attack	371456	199809
	Normal	128516	299899

KNN		Actual	
		Attack	Normal
Classifier	Attack	388147	97060
	Normal	111925	402868

CNN		Actual	
		Attack	Normal
Classifier	Attack	319166	25677
	Normal	180443	474714

TABLE 4.16: Confusion matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.9687	0.7072	0.7922	0.6935	0.7879	0.7559

TABLE 4.17: F1-score for each classifier.

- Brute Force-Web (Table 4.18 and Table 4.19)
  All classifiers achieve high F1-scores for Web BruteForce attacks, but MLP is superior to other classifiers. LSTMs scored marginally lower than other types.

MLP		Actual	
		Attack	Normal
Classifier	Attack	32474	123
	Normal	1748	34100

LSTM-50		Actual	
		Attack	Normal
Classifier	Attack	31688	492
	Normal	2231	33429

LSTM-100		Actual	
		Attack	Normal
Classifier	Attack	32207	836
	Normal	1700	33097

LSTM-250		Actual	
		Attack	Normal
Classifier	Attack	32033	561
	Normal	1897	33349

KNN		Actual	
		Attack	Normal
Classifier	Attack	33112	318
	Normal	1110	33905

CNN		Actual	
		Attack	Normal
Classifier	Attack	32594	103
	Normal	1629	34119

TABLE 4.18: Confusion matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.9962	0.9588	0.9621	0.9631	0.9789	0.9741

TABLE 4.19: F1-score for each classifier.

- Brute Force-XSS (Table 4.20 and Table 4.21)
  Similar to Web BruteForce attacks, all classifiers achieve high F1-scores for XSS Brute Force attacks, with MLP outperforming the competition.

MLP		Actual	
		Attack	Normal
Classifier	Attack	32691	48
	Normal	1312	33954

LSTM-50		Actual	
		Attack	Normal
Classifier	Attack	32001	530
	Normal	1613	33056

LSTM-100		Actual	
		Attack	Normal
Classifier	Attack	32230	802
	Normal	1373	32795

LSTM-250		Actual	
		Attack	Normal
Classifier	Attack	32219	49
	Normal	1350	33582

KNN		Actual	
		Attack	Normal
Classifier	Attack	33310	201
	Normal	691	33803

CNN		Actual	
		Attack	Normal
Classifier	Attack	32900	55
	Normal	1102	33948

TABLE 4.20: Confusion matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.9985	0.9676	0.9674	0.9788	0.9868	0.9827

TABLE 4.21: F1-score for each classifier.

- SQL Injection
  The SQL Injection attack has been skipped due to insufficient data.
- Infiltration
  The Infiltration attack has been skipped owing to insufficient data.
- Botnet (Table 4.22 and Table 4.23)
  The LSTM-50 classifier got a relatively high score for Botnet attacks followed by the second place which is the LSTM-100 classifier.

MLP		Actual	
		Attack	Normal
Classifier	Attack	392692	88105
	Normal	107351	411852

LSTM-50		Actual	
		Attack	Normal
Classifier	Attack	480199	16110
	Normal	19320	484051

LSTM-100		Actual	
		Attack	Normal
Classifier	Attack	440010	23898
	Normal	59247	476525

LSTM-250		Actual	
		Attack	Normal
Classifier	Attack	413913	82479
	Normal	86073	417215

KNN		Actual	
		Attack	Normal
Classifier	Attack	439268	82892
	Normal	42148	398522

CNN		Actual	
		Attack	Normal
Classifier	Attack	323537	54563
	Normal	48777	317748

TABLE 4.22: Confusion matrices for each classifier.

MLP	LSTM-50	LSTM-100	LSTM-250	KNN	CNN
0.8168	0.9644	0.9137	0.8308	0.8754	0.8623

TABLE 4.23: F1-score for each classifier.

Table 4.24 is the direct comparison of F1-scores gathered from the results above.

Attack Type	**MLP**	**LSTM-50**	**LSTM-100**	**LSTM-250**	**KNN**	**CNN**
FTP-BruteForce	0.9990	0.9976	0.9984	0.9982	**0.9991**	0.9990
SSH-Bruteforce	0.9763	0.9764	0.9763	0.8907	0.9732	**0.9764**
DoS-GoldenEye	0.9212	0.9680	0.7457	0.9498	**0.9821**	0.9434
DoS-Slowloris	0.9945	0.8490	0.8513	0.7770	0.9947	**0.9948**
DoS-SlowHTTP	0.9983	0.9798	0.9937	0.9803	**0.9985**	0.9984
DoS-Hulk	0.7309	0.6732	**0.7731**	0.7313	0.7381	0.7314
DDoS-LOIC-HTTP	**0.9968**	0.7606	0.7129	0.7443	0.8353	0.8406
DDoS-HOIC	**0.9687**	0.7072	0.7922	0.6935	0.7879	0.7559
BruteForce-Web	**0.9962**	0.9588	0.9621	0.9631	0.9789	0.9741
BruteForce-XSS	**0.9985**	0.9676	0.9674	0.9788	0.9868	0.9827
Botnet	0.8168	**0.9644**	0.9137	0.8308	0.8754	0.8623

TABLE 4.24: The F1-scores for binary classification.

4.5.3 Multi-class Classification Results

Multi-class classification is performed for all categories, which is a more challenging task, and the accuracy of the F-score for multi-class classification is measured. The calculation of F-score for multi-class classification is similar to the calculation of the binary-classification. The number of TP is retrieved from the diagonal line of the confusion matrix, the number of FP is the sum of the rest of the row and the FN is the sum of the rest of the column. The calculation of the F-score for multi-class classification is shown below.

$$Precision_c = \frac{TP_c}{TP_c + FP_c}, \tag{4.4}$$

$$Recall_c = \frac{TP_c}{TP_c + FN_c} \tag{4.5}$$

and

$$F1\ score_c = 2 \times \frac{Precision_c * Recall_c}{Precision_c + Recall_c}, \tag{4.6}$$

where c is the class. Confusion matrices for multi-class classification are shown in Tables 4.25 - 4.30.

	MLP													
	Normal	FTP-BF	SSH-BF	DoS-GoldenEye	DoS-Slowloris	DoS-SlowHTTPTest	DoS-Hulk	DDoS-LOIC-HTTP	DDOS-HOIC	BF-Web	BF-XSS	SQL Injection	Infiltration	Botnet
Normal	1374320	8	0	1080	242	31	1001	231	411	165	19	0	195	1668
FTP-BF	21015	386626	16	0	0	0	0	10	0	0	6	0	0	0
SSH-BF	26512	0	570004	0	0	0	0	0	0	0	0	0	0	1
DoS-GoldenEye	32799	0	26	222420	0	0	238	0	0	0	0	0	0	0
DoS-Slowloris	3248	0	0	5	104370	0	41	0	0	0	0	0	3	0
DoS-SlowHTTPTest	43111	0	0	0	4	210947	0	0	0	0	0	0	0	0
DoS-Hulk	297411	0	0	24	0	0	700247	0	0	0	0	0	0	0
DDoS-LOIC-HTTP	221011	0	0	0	0	0	17	777736	0	0	0	0	0	0
DDoS-HOIC	308941	0	0	0	0	0	0	0	699064	0	0	0	0	0
BF-Web	2125	0	0	0	0	0	0	0	0	33954	0	0	0	1
BF-XSS	2352	52	78	0	0	0	24	0	0	18	33946	0	0	0
SQL Injection	3936	34	34	0	0	0	0	28	0	86	33	445	0	0
Infiltration	890174	0	0	12	73	122	213	0	12	0	0	0	4216	417
Botnet	361679	0	0	2	0	0	0	0	4	0	0	0	0	333826

TABLE 4.25: Confusion table for MLP multi-class classification.

	LSTM-50													
	Normal	FTP-BF	SSH-BF	DoS-GoldenEye	DoS-Slowloris	DoS-SlowHTTPTest	DoS-Hulk	DDoS-LOIC-HTTP	DDOS-HOIC	BF-Web	BF-XSS	SQL Injection	Infiltration	Botnet
Normal	674007	88	351	538	0	89	55	110	131	367	126	292	1643	3260
FTP-BF	0	0	0	0	0	0	0	0	0	0	1	0	0	0
SSH-BF	31172	0	569564	0	0	0	0	0	0	33677	26629	0	0	0
DoS-GoldenEye	34172	0	0	222848	0	0	93	76	0	22	0	14	0	0
DoS-Slowloris	651153	0	0	52	104672	72	0	1	0	26	0	6	2758	0
DoS-SlowHTTPTest	25097	386536	67	6	0	210894	0	31	0	49	507	71	8	0
DoS-Hulk	298106	0	0	41	0	0	701452	176	126	5	0	42	0	0
DDoS-LOIC-HTTP	220851	0	0	0	0	0	0	777403	0	0	0	0	0	6
DDoS-HOIC	299899	0	0	0	0	0	0	38	699043	0	0	0	0	48
BF-Web	329	0	0	0	0	0	0	0	0	69	6727	0	0	0
BF-XSS	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SQL Injection	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Infiltration	735	0	0	0	1	0	0	0	0	0	0	0	5	0
Botnet	357522	0	0	0	0	0	0	0	0	0	0	0	0	332525

TABLE 4.26: Confusion table for LSTM-50 multi-class classification.

	LSTM-100													
	Normal	FTP-BF	SSH-BF	DoS-GoldenEye	DoS-Slowloris	DoS-SlowHTTPTest	DoS-Hulk	DDoS-LOIC-HTTP	DDOS-HOIC	BF-Web	BF-XSS	SQL Injection	Infiltration	Botnet
Normal	960902	0	37	414	371	206	463	64	4	310	0	197	1785	1248
FTP-BF	143	0	2	0	0	120	0	0	0	0	3	0	0	0
SSH-BF	30620	0	569654	0	0	0	0	0	0	33701	33509	4	1	0
DoS-GoldenEye	35503	0	0	222642	0	0	1857	0	11	0	0	5	1	0
DoS-Slowloris	8594	0	20	22	104100	7	0	0	0	10	4	25	19	0
DoS-SlowHTTPTest	26222	386706	98	0	0	210665	0	101	0	75	487	140	24	0
DoS-Hulk	301426	0	0	322	0	0	699285	204	138	0	0	9	4	0
DDoS-LOIC-HTTP	416058	0	0	0	0	0	27	622367	558838	0	0	0	0	2
DDoS-HOIC	104840	0	0	0	0	0	41	155186	140383	0	0	0	0	0
BF-Web	1	0	0	0	0	0	0	0	0	0	0	0	0	0
BF-XSS	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SQL Injection	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Infiltration	315839	0	9	132	212	90	0	37	0	26	0	32	2578	0
Botnet	392347	0	201	0	0	0	0	0	0	99	0	0	1	334650

TABLE 4.27: Confusion table for LSTM-100 multi-class classification.

	LSTM-250													
	Normal	FTP-BF	SSH-BF	DoS-GoldenEye	DoS-Slowloris	DoS-SlowHTTPTest	DoS-Hulk	DDoS-LOIC-HTTP	DDOS-HOIC	BF-Web	BF-XSS	SQL Injection	Infiltration	Botnet
Normal	492931	77492	92	495	2671	146	571	216	33	94	190	76	956	212
FTP-BF	1815	0	0	0	0	18	0	0	0	8	2	12	4	0
SSH-BF	30024	0	569600	0	0	0	0	0	0	33620	33312	3	0	0
DoS-GoldenEye	48761	0	0	221809	15771	0	2591	4	7	1	0	0	102	108
DoS-Slowloris	11577	0	0	73	83169	0	220	0	0	0	0	0	69	79
DoS-SlowHTTPTest	28232	309075	11	146	70	210808	0	83	0	99	437	200	32	0
DoS-Hulk	430455	0	0	838	2351	17	697558	140	18	1	0	1	209	0
DDoS-LOIC-HTTP	210760	0	0	0	0	0	15	756088	21	0	0	0	0	1
DDoS-HOIC	310564	0	0	6	626	0	278	21200	699027	0	0	0	0	0
BF-Web	5	0	0	0	0	0	0	0	0	1	1	0	0	0
BF-XSS	8	0	1	0	0	0	0	0	0	4	2	0	0	0
SQL Injection	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Infiltration	58022	0	0	0	0	6	0	0	0	37	0	9	564	0
Botnet	965884	0	8	93	2	27	133	0	0	344	21	28	2476	335383

TABLE 4.28: Confusion table for LSTM-250 multi-class classification.

KNN														
	Normal	FTP-BF	SSH-BF	DoS-GoldenEye	DoS-Slowloris	DoS-SlowHTTPTest	DoS-Hulk	DDoS-LOIC-HTTP	DDOS-HOIC	BF-Web	BF-XSS	SQL Injection	Infiltration	Botnet
Normal	378355	1	445	1391	55	7	17015	11579	16785	38	25	25	372	17733
FTP-BF	94	50280	0	0	0	0	0	0	0	0	0	0	0	0
SSH-BF	3303	0	74150	0	0	0	0	0	0	0	0	0	0	0
DoS-GoldenEye	815	0	0	27847	0	0	5	0	0	0	0	0	0	0
DoS-Slowloris	122	0	0	0	13440	0	0	0	0	0	0	0	0	0
DoS-SlowHTTPTest	95	0	0	0	0	27332	0	0	0	0	0	0	0	0
DoS-Hulk	28209	0	0	0	0	0	75186	0	0	0	0	0	0	0
DDoS-LOIC-HTTP	20458	0	0	0	0	0	0	89361	0	0	0	0	0	0
DDoS-HOIC	21807	0	0	0	0	0	0	0	74191	0	0	0	0	0
BF -Web	210	0	3	0	0	0	0	0	0	4447	0	0	0	0
BF-XSS	112	0	0	0	0	0	0	0	0	0	4423	0	0	0
SQL Injection	18	0	0	0	0	0	0	0	0	0	0	21	0	0
Infiltration	49	0	0	0	0	0	0	0	0	0	0	0	161	0
Botnet	13614	0	0	0	0	0	0	0	0	0	0	0	0	26421

TABLE 4.29: Confusion table for KNN multi-class classification.

CNN														
	Normal	FTP-BF	SSH-BF	DoS-GoldenEye	DoS-Slowloris	DoS-SlowHTTPTest	DoS-Hulk	DDoS-LOIC-HTTP	DDOS-HOIC	BF-Web	BF-XSS	SQL Injection	Infiltration	Botnet
Normal	1712106	0	38	101	47	11	111	197	188	3	6	0	216	1138
FTP-BF	20397	386719	3	0	0	0	0	0	0	0	0	0	0	0
SSH-BF	26334	0	570093	0	0	0	0	0	0	0	0	0	0	0
DoS-GoldenEye	34052	0	0	223442	0	0	694	0	0	0	0	0	0	0
DoS-Slowloris	3326	0	0	0	104568	0	11	0	0	0	0	0	0	0
DoS-SlowHTTPTest	54916	0	0	0	11	211089	0	0	0	0	0	0	0	0
DoS-Hulk	297756	0	0	0	0	0	700954	0	0	0	0	0	0	0
DDoS-LOIC-HTTP	220901	0	0	0	0	0	0	777771	0	0	0	0	0	0
DDoS-HOIC	306137	0	0	0	7	0	0	4	699262	0	0	0	3	0
BF-Web	2180	0	16	0	0	0	0	11	2	34220	0	0	0	0
BF-XSS	1684	0	0	0	0	0	0	0	0	0	33985	0	0	0
SQL Injection	2132	0	9	0	0	0	0	23	3	0	13	445	0	0
Infiltration	553680	0	0	0	56	0	11	0	0	0	0	0	4195	35
Botnet	353033	0	0	0	0	0	0	0	36	0	0	0	0	334739

TABLE 4.30: Confusion table for CNN multi-class classification.

Attack Type	**MLP**	**LSTM-50**	**LSTM-100**	**LSTM-250**	**KNN**	**CNN**
Normal	0.5533	0.4117	0.5401	0.3115	**0.8306**	0.6457
FTP-BruteForce	0.9734	NaN	NaN	NaN	**0.9991**	0.9743
SSH-Bruteforce	0.9772	0.9254	0.9206	0.9215	0.9753	**0.9774**
DoS-GoldenEye	0.9287	0.9219	0.9209	0.9215	**0.9618**	0.9277
DoS-Slowloris	0.9830	0.2433	0.9573	0.8323	**0.9935**	0.9837
DoS-SlowHTTP	0.9070	0.5056	0.5042	0.5546	**0.9981**	0.8844
DoS-Hulk	0.8241	0.8245	0.8212	0.7611	0.7688	**0.8244**
DDoS-LOIC-HTTP	**0.8755**	0.8754	0.524	0.8668	0.8480	**0.8755**
DDOS-HOIC	0.8188	0.8232	0.2553	0.8078	**0.8253**	0.8203
BruteForce-Web	0.9659	0.0334	NaN	0	0.9679	**0.9687**
BruteForce-XSS	0.9634	NaN	NaN	0	**0.9847**	0.9756
SQL-Injection	0.1765	NaN	NaN	NaN	**0.4941**	0.2908
Infiltration	0.0898	0.0417	0.142	0.0179	**0.4334**	0.0149
Botnet	0.6473	0.6465	0.6295	0.4089	0.6277	**0.6540**

TABLE 4.31: The F1-score for multi-class classifications.

According to Table 4.31, KNN and CNN produce favourable outcomes. The division by 0 results in the existence of NaNs in the table. When the number of TP is equal to zero, both precision and recall will be equal to zero, and the F-score cannot be calculated. BruteForce-Web attack detection with LSTM-250 yields an F-score of 0 because, despite the fact that the number of TP is not zero, the F-score is too low to be displayed with four decimal places. Even if the results are randomly categorised, the F-score will be higher, so the NaN and 0 F-scores may not make sense. However, the classifier may not be able to differentiate the data from other categories, causing the classifiers to incorrectly categorise all positive data into wrong categories, and leave with 0 TP. For some types of attacks, LSTMs are unable to produce competitive results. It is believed that the difficulty of LSTM training plays a role. Moreover, it is worth noting that RNNs may not be necessary for the processed data, as the data from the algorithm already encodes time-series information.

In addition, yielding a high accuracy is not always a good indication. DDoS-LOIC and DDoS-HOIC are two different but similar DDoS attack setups. LOIC stands for Low Orbit Ion Cannon, a network stress test tool developed by Praetox Technologies. LOIC can be utilised by DDoS offenders to flood target systems with bogus TCP, UDP, and HTTP GET requests. HIOC on the other hand, stands for High Orbit Ion Cannon, which is a replacement of LOIC for DoS/DDoS attacks. It allows a single user to have larger concurrent requests and it obfuscates attacker IP addresses. Let us consider Table 4.25 as an example. HOIC and LOIC attacks are clearly identifiable by MLP although both of them are DDoS attacks under HTTP protocol. The reason is that all connections are identical with the same tool and the classifiers can identify the characteristics of the attacks easily. However, by modifying the request; i.e., using HOIC instead of LOIC, the classifiers will not be able to detect the attack completely. This illustrates the case of sub-optimal generality for classification thus anomaly-based intrusion detection methods are required for detecting the evolving attacks.

4.6 Anomaly-based Detection

Anomaly detection in data analysis typically refers to the discovery of uncommon observations of patterns that differ considerably from the majority of the data and do not adhere to a well-defined concept of normal behaviour. Anomaly detection is also known as outlier detection or novelty detection. [85] introduced the applications of anomaly detection in many areas such as intrusion detection, fraud detection and industrial damage detection. The paper also introduced several machine learning techniques used in anomaly detection.

The measurement is called the outlier factor or outlier score in the following context. Higher outlier factors mean the data have a larger difference than the normal data, in other words, the data is more likely to be an anomaly. The training dataset has normal data only and the testing dataset contains a mixture of normal and attack data.

4.6.1 Anomaly Detectors

- **Multivariate Normal Distribution (MND)**
  Assuming the normal data are normally distributed variables and they are used to build the MND and the probability distribution presents at a high dimension. The outlier factors are calculated as the inverse of probability densities of the testing data according to the distribution of normal data.

  The data are flattened into multi-dimensional vectors for MND and all outlier detectors below.

- **Copula-Based Outlier Detection (COPOD)**
  COPOD is inspired by copulas for modelling the distribution of multivariate data. Copula allows the joint distribution to be defined using only marginals, which enables greater modelling flexibility for high-dimensional data. COPOD first generates an empirical copula and then utilises it to predict the tail probability of each data point in order to identify its levels of outlierness. The details of COPOD can be found in [86].

- **Autoencoder**
  Autoencoders are a kind of unsupervised learning technology that uses neural networks to learn representations. A bottleneck in the network imposes a compressed knowledge representation of the original input, according to the neural network architecture. The neural networks have narrower layers between the input and output. This compression and subsequent reconstruction would be very difficult if the input characteristics were all independent of one another, and this property is important for anomaly detection.

When training the autoencoder, the output data is exactly the same as the input data, enabling the autoencoder to learn how to reconstruct the input data during training. The autoencoder is trained with normal data only. Assuming the normal data exhibit similar characteristics, the outputs will resemble the input and have small errors compared to the input. In other words, a larger error suggests that the data is more likely to be anomalous.

- **Angle-Based Outlier Detection (ABOD)**
  Angle-based outlier detection (ABOD) combats the "curse of dimensionality" by evaluating the broadness of a point's angle spectrum as an outlier component in addition to proximity-based measurements. For a given point within a cluster, the angles between this point and other points vary considerably. At the cluster's border, the variance of the angles will diminish and the variance will be smaller for outliers. The details of ABOD can be found in [87].

- **Clustering-Based Local Outlier Factor (CBLOF)**
  The clustering-based local outlier factor (CBLOF) assumes that normal data objects belong to big, dense clusters, whereas outliers belong to small, sparse clusters or do not belong to any clusters. Approaches based on clustering identify outliers by determining the link between items and clusters. An item is an outlier if it does not belong to any cluster, has a great distance from the cluster, or is part of a small or sparse cluster [88].

- **Histogram-Based Outlier Score (HBOS)**
  The histogram-based outlier score tends to improve the computation speed by sacrificing some accuracy. HBOS assumes feature independence, making it significantly faster than multivariate techniques [89].

- **Isolation Forest (IForest)**
  Isolation Forest (IForest) finds anomalies by isolating them rather than by capturing normal points [90]. Isolation forest is a novel technique that explicitly isolates anomalies using binary trees, demonstrating a new prospect for a speedier anomaly detector that directly targets abnormalities without profiling all normal instances. Using recursive random partitioning, it is anticipated that anomalies are simpler to separate than normal data points. The metric of outlierness is the number of steps required to isolate a data point.

- **K-nearest Neighbors (KNN)**
  K-nearest Neighbors (KNN) has been used in supervised learning previously, and it is also used in anomaly-based detection. Instead of measuring the distance between the given testing data with other labelled data points, the presented data point is only compared with the normal data. The distance to the normal data will be the outlier factor.

- **Local Outlier Factor (LOF)**
  Local Outlier Factor (LOF) is an unsupervised anomaly detection technique that calculates the local density deviation of a particular data point relative to its neighbours. Outliers are samples that have a significantly lower density than their neighbours [91].

- **Principal Component Analysis (PCA)**
  PCA is a dimensionality reduction approach that decreases the number of dimensions in a dataset without losing a large amount of information. The

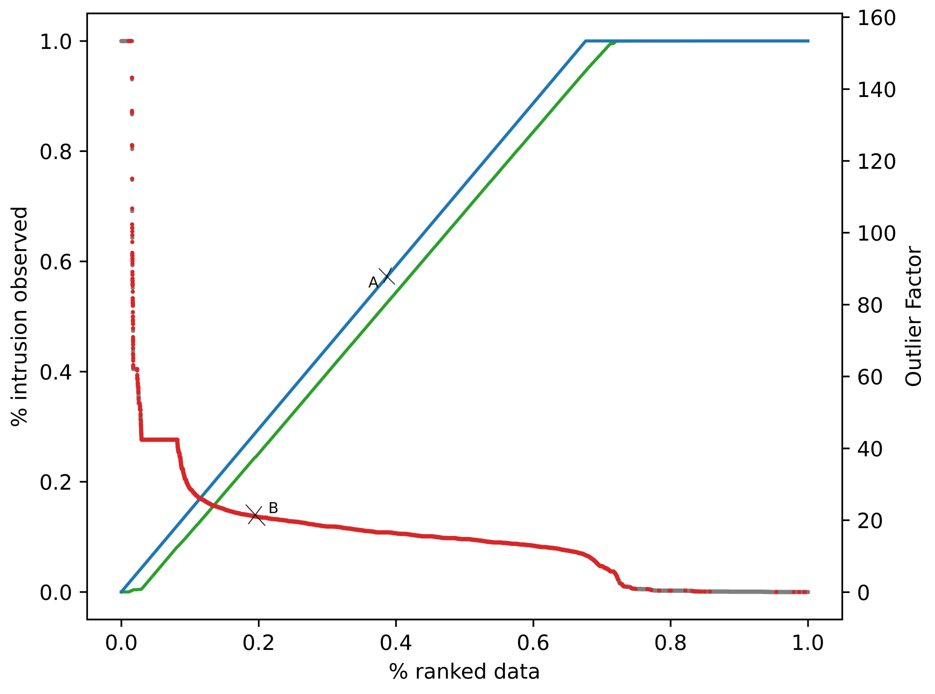


FIGURE 4.6: Example figure for anomaly-based intrusion detection.

essence of PCA-based anomaly detection is that an anomalous sample should have more reconstruction error than a normal sample. In other words, the reconstruction error of an anomalous sample that is compressed and subsequently decompressed with PCA should be greater than when the same operation is done to a normal sample [92].

First, the performance of each anomaly detector is analysed based on the distribution of outlier factors. For each anomaly detector, the analyses are carried out on each type of attack. In real-world applications, attacks are not labelled, so the anomaly detector will not indicate the type of attacks. The reason for analysing each attack separately is to understand the difficulty of detecting each attack. Figure 4.6 provides an example of how the analyses are performed.

Initially, the data are ranked based on the outlier score in descending order, meaning that the data with the highest outlier score are expected to be ranked highest. Next, the top-ranked data are compared to their actual labels to verify if they are indeed intrusions. On the chart, the blue curve represents the expected result that should be observed with a perfect detector, whereas the green curve represents the actual result. Both the blue and green curves are connected to the left y-axis. The dots represent data points, which are distributed on the chart based on their outlier factors; the red dots represent actual intrusions, while the grey dots represent normal data. The dots correspond to the right y-axis. Both curves and points share the x-axis.

The x-axis represents the position of the data in the ranked output. For instance, $x = 0.4$ implies that the data point is situated at roughly the 40% mark of the ranked data. Consider a dataset $\{a, b, c\}$ with outlier factors $(20, 10, 30)$. Once ranked, the sequence becomes $\langle c, a, b \rangle$, placing 'a' at the 50% location on the x-axis. The left y-axis y_l represents the proportion of observed intrusions within the top x per cent of ranked data. Figure 4.6's data point 'A' indicates that, for the top 40 per cent of data, approximately 60 per cent of intrusions are expected to be ranked at the top (blue curve), whereas only about 55 per cent of actual intrusions have been observed to be ranked at the top (green curve). Therefore, approximately 5 per cent of normal data have been improperly ranked. In addition, the turning point of the blue curve indicates that this dataset contains approximately 67% of intrusions, as the blue curve represents the expected result and all intrusions are expected to be ranked at the top. The turning point on both green and blue curves encompasses 100 per cent of intrusions. The right y-axis y_r represents the outlier factor for a specific data point (dots). For instance, point 'B' in Figure 4.6 is an anomaly (red dot) that occurs at a 20 per cent location of the ranked data and has an outlier factor of 20.

The chart may be somewhat challenging to interpret; however, to measure performance, one can simply assess the proximity of the green curve to the blue curve. This chart is favoured over the receiver operating characteristic (ROC) curve, as it offers insights into the extent of intrusion in the dataset and its distribution after the application of anomaly detectors. The ROC curve will be generated for performance measurements in a subsequent context. Now, it is time to explore some examples of performance for a range of attack types paired with various detector types. With over 100 charts, showcasing them all is unfeasible; thus, a selection of representative examples will be provided as illustrations.

4.6.2 Results

- FTP-BruteForce
  Apart from COPOD, all detectors yield favourable and similar outcomes. As evidenced in Figure 4.7's KNN chart, the green and blue curves are in close proximity to each other. In addition, red dots are ranked highest. Normal data have been prioritised for COPOD, and the result is not encouraging.

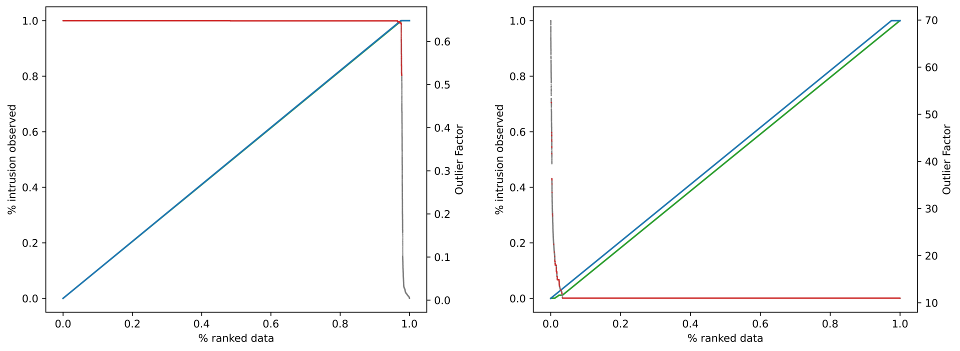


FIGURE 4.7: KNN detector on the left and COPOD detector on the right.

- SSH-Bruteforce
  All anomaly detectors show the difficulty of detecting SSH-Bruteforce attacks. All detectors yield identical results and random data ordering. IForest is slightly better than others, as the green curve begins closer to the blue curve. Other charts depict a random ranking, which causes the green curve to overlap the diagonal line. Some results are shown in Figure 4.8.

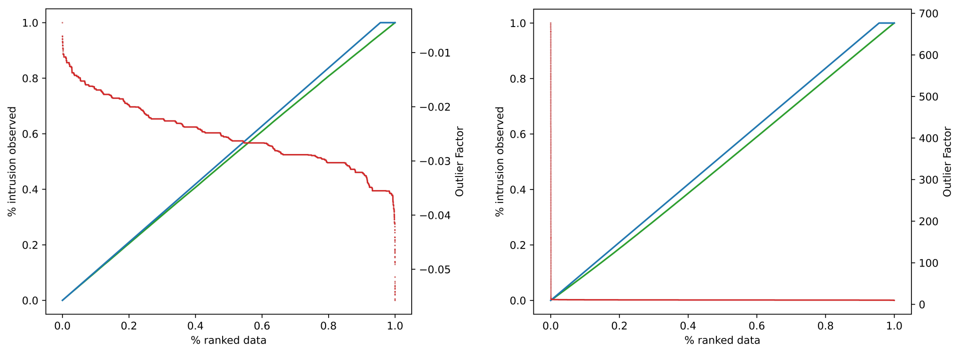


FIGURE 4.8: IForest detector on the left and AutoEncoder detector on the right.

- DoS-GoldenEye
  All detectors can distinguish a small number of normal data, but the remaining data are harder to be identified. In other words, the last 3% (located at $(x = 1, y_l \approx 0.97)$ on the charts) of the entire ranked data are normal data only. This indicates that the detection system will produce a higher rate of false positives for anomaly detection if the threshold is set to the beginning of the last 3% of normal data. The threshold for HBOS will be approximately 660 in this case. However, 660 is not necessarily the threshold for the best F-score. Some results are shown in Figure 4.9.

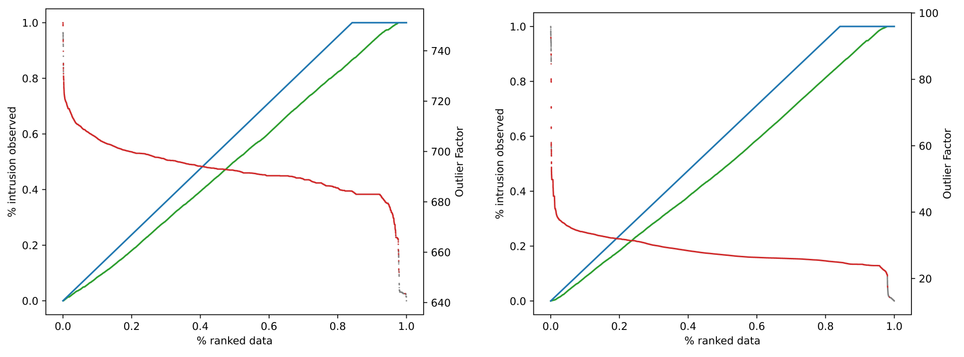


FIGURE 4.9: HBOS detector on the left and COPOD detector on the right.

- DoS-Slowloris
  The only anomaly detector that performs poorly is the LOF detector. IForest performs the best, followed by HBOS and MND which perform marginally below par. PCA, KNN, COPOD, and CBLOF have comparable performance, which, while not as good as HBOS and MND, is still quite good. ABOD performs worse than PCA and other detectors with comparable performance, and it clearly ranked approximately 5 per cent of normal data at the top. Some results are shown in Figure 4.10.

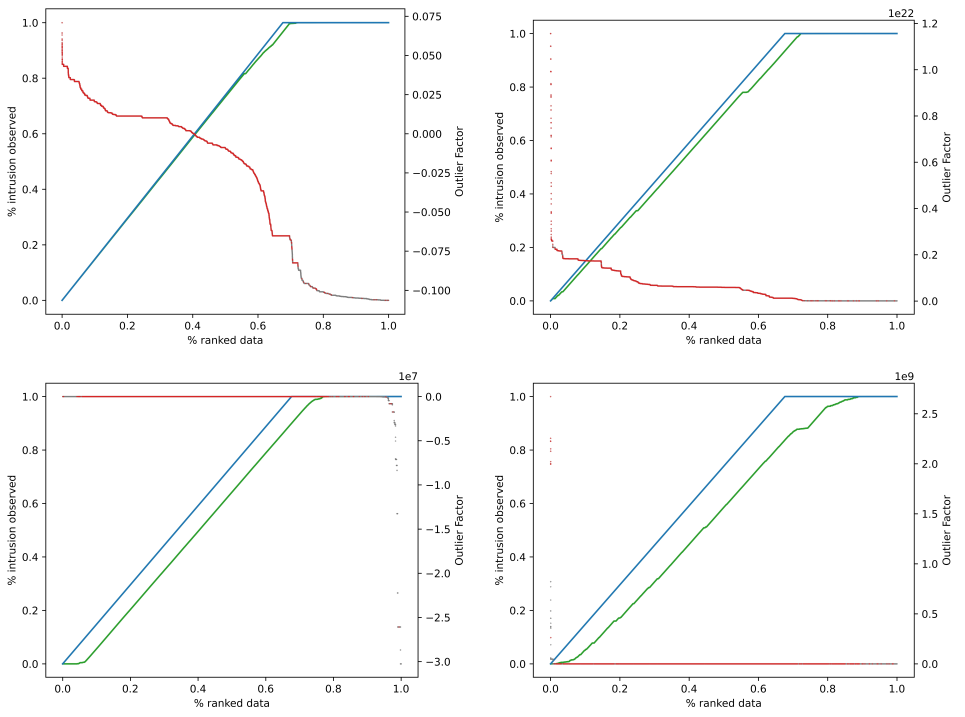


FIGURE 4.10: IForest detector the first, PCA detector the second, ABOD detector the third and LOF detector the fourth.

- DoS-SlowHTTPTest
  DoS-SlowHTTPTest attack detection has a large performance margin for detectors. ABOD, Autoencoder, CBLOF, KNN, LOF, and PCA produce results that are close to ideal, whereas COPOD, HBOS, and IForest rank almost all normal data at the top, which is inferior to the random ranking (Figure 4.11).

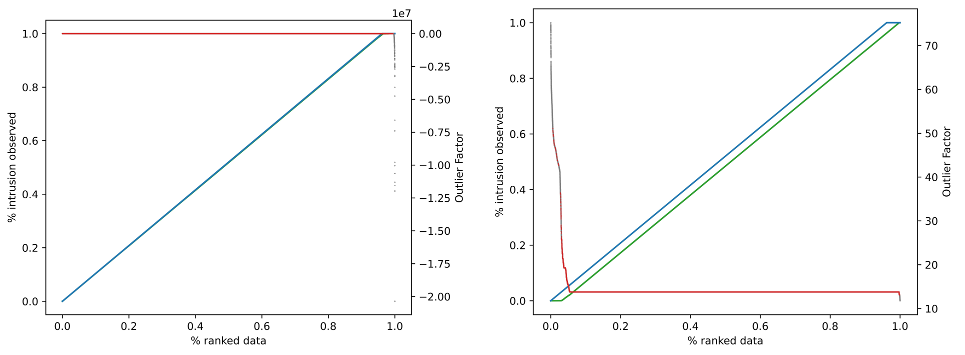


FIGURE 4.11: ABOD detector on the left and COPOD detector on the right.

- DoS-Hulk, DDoS-LOIC-HTTP and DDOS-HOIC
  DoS-Hulk, DDoS-LOIC-HTTP and DDOS-HOIC anomaly detection is the most difficult task so far. All detectors failed to produce a promising result, and all resulting charts are nearly identical. Results that are better than others are shown in Figure 4.12.

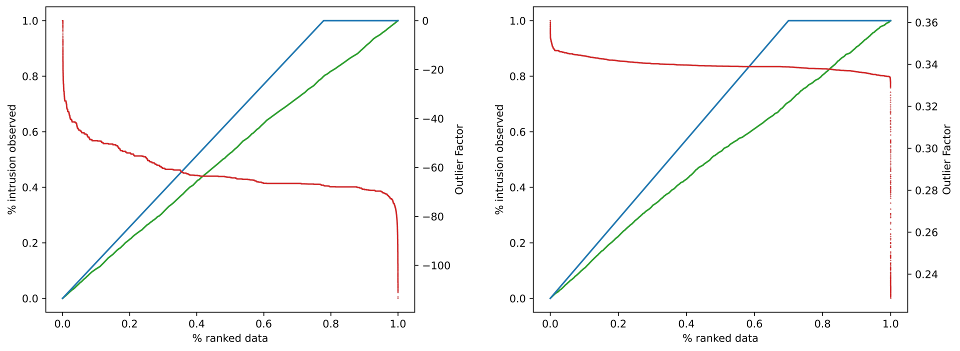


FIGURE 4.12: DDoS-LOIC-HTTP detection with ABOD on the left and DDOS-HOIC detection with CBLOF on the right.

- BruteForce-Web and BruteForce-XSS
  LOF is the most effective BruteForce-Web attack detector, though false negatives are to be expected. KNN is inferior to LOF, and other detectors perform poorly with normal data typically ranking highest. The results are similar to BruteForce-XSS attacks. Some results are shown in Figure 4.13.

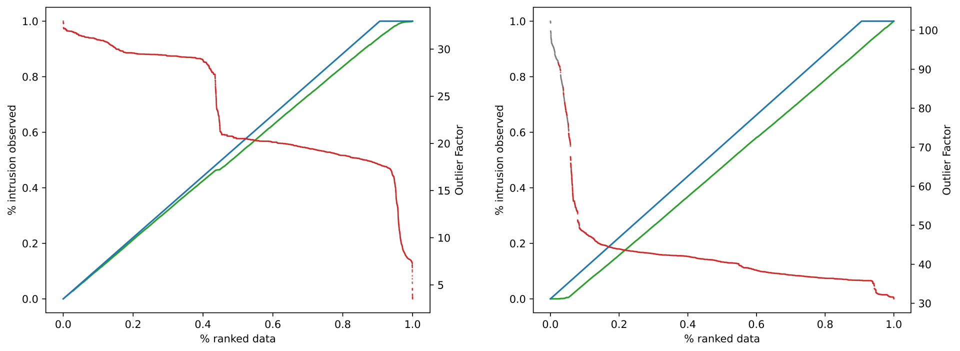


FIGURE 4.13: LOF detector on the left and COPOD detector on the right.

- SQL Injection
  Autoencoder, MND and PCA perform better than other detectors. All detectors ranked a part of the intrusions to the top, except for IForest, COPOD and HBOS, which perform poorly. Some results are shown in Figure 4.14.

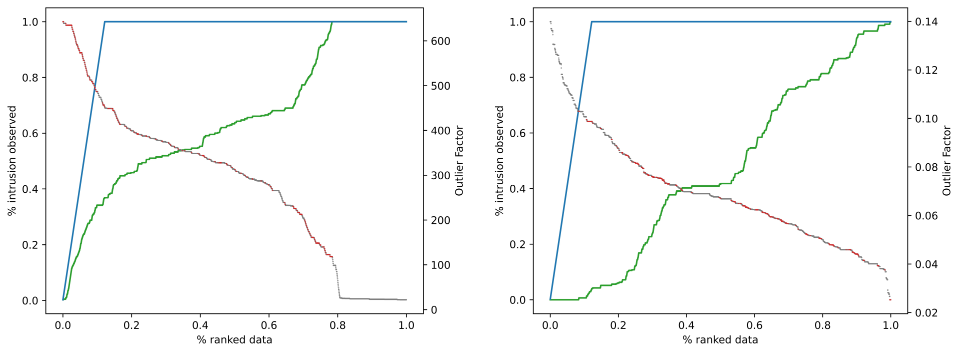


FIGURE 4.14: Autoencoder detector on the left and IForest detector on the right.

- Infiltration
  Infiltration attacks are the most difficult to detect compared to other types of attacks. None of the detectors are able to potentially differentiate between attacks and normal data, and some normal data have higher outlier factors than the intrusions. In general, the green curves in all graphs are below the diagonal lines; therefore, none of the detectors performs better random data ranking. Some results are shown in Figure 4.15.

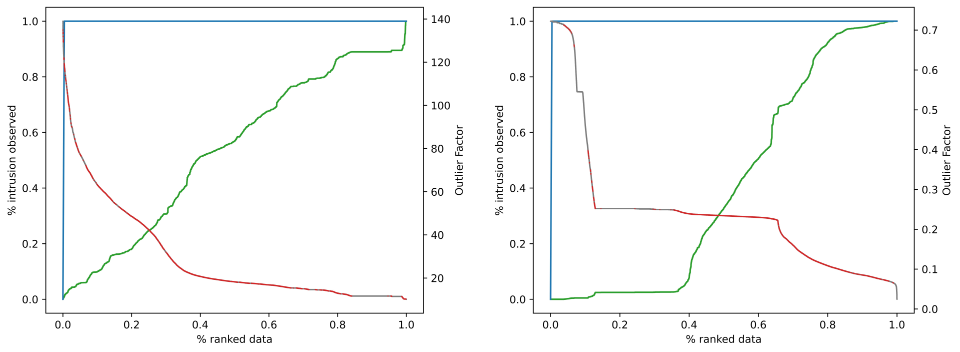


FIGURE 4.15: COPOD detector on the left and CBLOF detector on the right.

- Botnet
  Autoencoder, PCA, and MND are the most effective detectors, whereas other detectors are marginally less effective. COPOD has not produced a promising outcome and is inferior to random selection. Some results are shown in Figure 4.16.

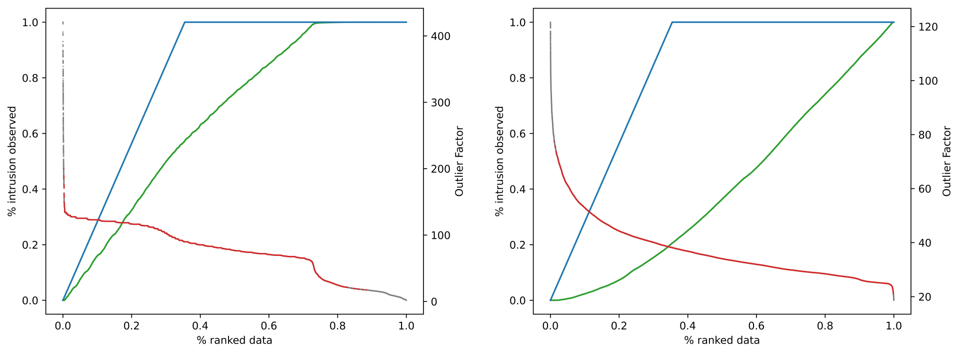


FIGURE 4.16: Autoencoder detector on the left and COPOD detector on the right.

4.7 Comparison between CICFlowMeter and Ours

A comparison is made between the performance of binary-classification and anomaly detection, using preprocessed data with the online process mining technique and the CICFlowMeter. The binary-classification result is sourced from [14], where a CNN serves as the classifier. To ensure a fair comparison, results produced by CNN will also be assessed. The first two columns in Table 4.32 display the F-score of classification with CNN. It should be noted that the results from Kim et al. exhibit lower precision, with only two decimal places. Though some classes have high F-scores with the CICFlowMeter, sizeable margins exist between certain classes. In conclusion, the technique achieves an average F-score of 0.91, while the CICFlowMeter attains 0.82.

Attack Type	**Ours (CNN)**	**CICFM (CNN)**	**Ours (Mixed)**
FTP-BruteForce	**0.9990**	0.98	**0.9991**
SSH-Bruteforce	**0.9764**	0.96	**0.9764**
DoS-GoldenEye	**0.9434**	0.47	**0.9821**
DoS-Slowloris	**0.9948**	0.66	**0.9948**
DoS-SlowHTTP	**0.9984**	**1**	**0.9985**
DoS-Hulk	0.7314	**1**	0.7731
DDoS-LOIC-HTTP	0.8406	**1**	**0.9968**
DDOS-HOIC	0.7559	**1**	0.9687
BruteForce-Web	**0.9741**	0.3	**0.9962**
BruteForce-XSS	**0.9827**	0.65	**0.9985**
Botnet	0.8623	**1**	0.9644

TABLE 4.32: Comparison between CICFlowMeter and our technique.

The last column compiles the best scores from various classifiers employing our technique, resulting in an average F-score of 0.97, which is highly competitive. Now, attention turns to the comparison for anomaly detection. The CICFlowMeter processes PCAP files and generates statistical data, which is then filtered to retain only TCP traffic. Moreover, only the following data columns are kept:

'Protocol', 'Flow Duration', 'Tot Fwd Pkts', 'Tot Bwd Pkts', 'TotLen Fwd Pkts', 'TotLen Bwd Pkts', 'Fwd Pkt Len Max', 'Fwd Pkt Len Min', 'Fwd Pkt Len Mean', 'Fwd Pkt Len Std', 'Bwd Pkt Len Max', 'Bwd Pkt Len Min', 'Bwd Pkt Len Mean', 'Bwd Pkt Len Std', 'Flow Byts/s', 'Flow Pkts/s', 'Flow IAT Mean', 'Flow IAT Std', 'Flow IAT Max', 'Flow IAT Min', 'Fwd IAT Tot', 'Fwd IAT Mean', 'Fwd IAT Std', 'Fwd IAT Max', 'Fwd IAT Min', 'Bwd IAT Tot', 'Bwd IAT Mean', 'Bwd IAT Std', 'Bwd IAT Max', 'Bwd IAT Min', 'Fwd PSH Flags', 'Bwd PSH Flags', 'Fwd URG Flags', 'Bwd URG Flags', 'Fwd Header Len', 'Bwd Header Len', 'Fwd Pkts/s', 'Bwd Pkts/s', 'Pkt Len Min', 'Pkt Len Max', 'Pkt Len Mean', 'Pkt Len Std', 'Pkt Len Var', 'FIN Flag Cnt', 'SYN Flag Cnt', 'RST Flag Cnt', 'PSH Flag Cnt', 'ACK Flag Cnt', 'URG Flag Cnt', 'CWE Flag Count', 'ECE Flag Cnt', 'Down/Up Ratio', 'Pkt Size Avg', 'Fwd Seg Size Avg', 'Bwd Seg Size Avg', 'Fwd Byts/b Avg', 'Fwd Pkts/b Avg', 'Fwd Blk Rate Avg', 'Bwd Byts/b Avg', 'Bwd Pkts/b Avg', 'Bwd Blk Rate Avg', 'Subflow Fwd Pkts', 'Subflow Fwd Byts', 'Subflow Bwd Pkts', 'Subflow Bwd Byts', 'Init Fwd Win Byts', 'Init Bwd Win Byts', 'Fwd Act Data Pkts', 'Fwd Seg Size Min', 'Active Mean', 'Active Std', 'Active Max' and 'Active Min'

For information about the data columns, one can refer to the Intrusion detection evaluation dataset (ISCXIDS2012) [26]. Data was labelled according to sockets and normalised column-wise. The same anomaly detectors were employed for the preprocessor on the data generated by the CICFlowMeter. Figure 4.17 illustrates the comparison of the ROC for the performance of both techniques. Despite the challenges outlined in the previous section, the overall performance of the preprocessor in anomaly detection surpasses that of the CICFlowMeter by a significant margin. Performance issues concerning certain types of attacks do not necessarily suggest the technique is ineffective; rather, they highlight the inherently difficult nature of anomaly detection.

From Figure 4.17, it is evident that the best result for online process mining, 0.63, is achieved by the Autoencoder and LOF. Conversely, the CICFlowMeter's top result, 0.42, is produced by the LOF detector. Some detectors yield 0 AOC, which can be attributed to the high dimensionality of the data and certain data points having

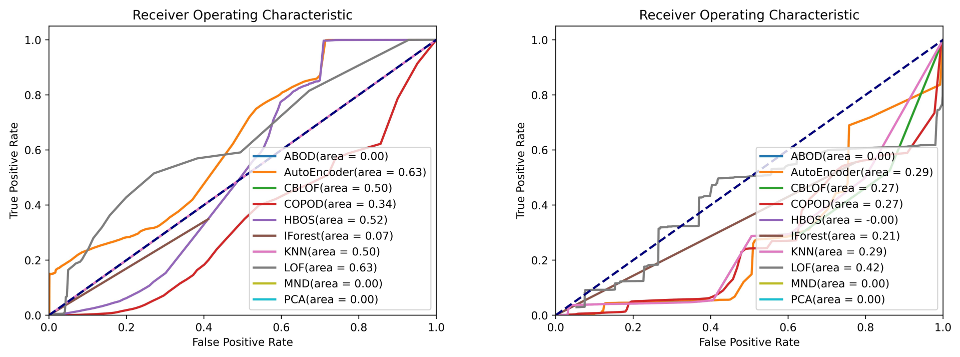


FIGURE 4.17: The receiver operating characteristic (ROC) for anomaly-based intrusion detection setup. The first chart shows the performance of our preprocessor, whereas the second chart shows the performance of CICFlowMeter.

excessively large or infinite outlier factors. Consequently, it becomes impossible to identify a discrete space for thresholds. For instance, MND creates a particularly steep and spike-like distribution where the density difference between the centre and outer area is substantial. Moreover, the accuracy measurement of our technique is based on the packet level. In other words, it is the measurement of how many packets are classified as anomalies. A connection that is an intrusion will likely have some of its packets classified as normal. Hence, if a connection is classified as an intrusion when at least one of its packets is deemed an anomaly, the resulting score could be higher.

4.8 Summary

This chapter recalls the third research question. The chapter discussed the limitations of existing Network Intrusion Detection Systems (NIDS) and process mining techniques in detecting network attacks. NIDS can be categorized into online and offline approaches, with packet-level and flow-level NIDS being subcategories of online NIDS. Researchers have explored methods such as encoding global flow information and using historical information to overcome these challenges. However, process mining faces its own limitations when applied to online anomaly detection, and the performance of some mining techniques is a concern.

The feature generator algorithm was discussed in detail with the pseudocode provided. The chapter introduces new definitions for transitions and event classes to better adapt to the network packet environment. The algorithm was then used for two types of detection: signature-based detection and anomaly-based detection. Performance was measured to show that the feature generator works as expected across various machine-learning algorithms. Finally, the result has been compared with CICFlowMeter, which is a widely used feature generator for network packet data.

Chapter 5

TFGen: A General Approach

5.1 Online Feature Generation on Concurrent Data Streams

The feature generation algorithm has been presented in Chapter 4. A more detailed, accurate, and general version of the algorithm, applicable to various types of streams, is provided here.

Given a sequence of events, P, a *transition* in P is defined as a pair of consecutive events (p_i, p_j) within the same case. This transition is referred to as the precedence relation, and P can be treated as the event log.

A *trace* consists of a series of event names, while a *case* is an executed instance of a trace. A trace, for instance, can be a predefined procedure or process for producing a specific type of medication. This medication can be manufactured multiple times, resulting in numerous cases. The event log contains logs from the production of various types of medications. In this thesis, the case is also called a flow.

For example, consider a series of events $P = \langle p_1, p_2, p_3, p_4, p_5 \rangle$ with two flows t_1 and t_2, where flow $t_1 = \langle p_1, p_3, p_5 \rangle$ and flow $t_2 = \langle p_2, p_4 \rangle$. There will be two transitions for t_1: (p_1, p_3) and (p_3, p_5); and one transition for t_2: (p_2, p_4). Although p_1 and p_2 are consecutive events, they are not considered a transition since they belong to different flows.

An *event class* $ec(p)$ is the name of an event p, typically a concatenated string of non-numerical attribute data. In the context of network traffic, it represents the enabled flags of packets and some indicators (Chapter 2). An event is the executed instance of an event class.

The primary distinction between TFGen and traditional process mining lies in the focus. Process mining concentrates on process model generation and analytical methods like conformance checking on process models, whereas TFGen is designed for online processing and feature generation for machine learning.

The TFGen implementation as a Python package can be found on Github[1]. This implementation can process around 80,000 events per second using a single thread of an Intel Core i5-12600K processor[2]. The package was developed to enhance the extensibility of the current feature generation method, which is advantageous for future research.

5.2 Documentation

5.2.1 Installation

Install TFGen package via PyPI.

[1]https://github.com/yinzheng-zhong/TFGen

[2]Using the provided NIDS dataset on Github

```
pip install tfgen     # normal install
pip install --upgrade tfgen  # update tfgen
```

5.2.2 How to Use

First, the observable event classes must be obtained, which can be achieved using the method "get_observable_ec()". The generated features may vary if the event classes or their order change, so it is recommended to save the observable event classes for future use. The first parameter of the method should be a dataframe, list, or array of attributes. This step can be skipped if the observable event classes are already available. The datasets to be used for the subsequent code examples can be found in release v0.2.1[3] on GitHub.

```
from tfgen.observe_event_classes import get_observable_ec

data_for_ec = pd.read_csv('test_data_for_ec.csv')

# Flags and S/C are the attributes
ec = get_observable_ec(data_for_ec[['Flags', 'S/C']])
```

At this point, the creation of the TFGen object can commence. The first parameter is a list of observable event classes obtained in the previous step, while the second parameter is the window size.

```
from tfgen import TFGen
tfgen = TFGen(ec, window_size=500)
```

Now, the process of loading the event log data to create features is underway. Prior to sending the data to TFGen, ensure that it is arranged in chronological order. Marking the end of each case with an EOT (End of Transmission) is necessary, and this must be done for each attribute. Without EOT, the TET (Temporal Event Trace) will continue to expand. This might resemble the following table (Table 5.1).

Case_ID	Flags	S/C
...	...	...
13	000.ACK.FIN.	C
13	000.ACK.	S
14	000.SYN.	C
13	000.ACK.RST.	S
13	EOT	EOT
14	000.ACK.SYN.	S
...	...	...

TABLE 5.1: Example of EOT in event logs.

```
data_for_feature = pd.read_csv('test_data_with_eot.csv')
```

Either the offline dataset or the dataset loaded in online streaming mode are options. The following is the procedure to load the dataset in offline mode:

```
tfgen.load_from_dataframe(data_for_feature,
                          case_id_col='Case_ID',
                          attributes_cols=['Flags', 'S/C'])
output = tfgen.get_output_list()  # this will return a list of data.
```

[3]https://github.com/yinzheng-zhong/TFGen/releases/tag/v0.2.1

Note that the output is a list (or other iterable) of tuples where each tuple contains two variables (case_id, transition_table). The case_id comes from the last processed event, and it can be used for labelling the data for supervised learning or validation. "get_output_list()" can only be used in offline mode.

The following example uses the generator as input for online streaming mode.

```
# replace this generator with the actual generator
def replace_with_the_actual_generator():
    while True:
        for rows in data_for_feature.values:
            case_id = rows[0]
            event_attrs = rows[[2, 3]]

            # event_attr is an iterable with multiple attributes.
            yield case_id, event_attrs

# Use the generator as an input for online streaming.
tfgen.load_from_generator(replace_with_the_actual_generator)

# this will return a generator as the output.
out = tfgen.get_output_generator()
```

Only the input methods "load_from_dataframe()" or "load_from_generator()" can be used with the output method "get_output_generator()".

The data can be entered one at a time into TFGen. Due to the fact that TFGen requires several events to initialise, the output is not guaranteed. To use this method, one should handle the InitialisingException exception.

```
from tfgen import InitialisingException

data_for_feature_array = data_for_feature.values
for sample in data_for_feature_array:
    case_id = sample[0]
    event_attrs = sample[[2, 3]]

    # tfgen.load_next(<your data sample>).
    # The sample is a tuple of (case_id, event_attrs)
    # and event_attrs is an iterable with multiple attributes.
    tfgen.load_next(case_id, event_attrs)
    try:
        print(tfgen.get_output_next())
    except InitialisingException:
        continue
```

The output method "get_output_next()" is compatible with all input methods.

5.2.3 Methods

Currently, the "Classic" and the "ClassicLargeSparse" methods for feature generation are available. The "Classic" method is employed by default. The "ClassicLargeSparse" method can be used to output Scipy sparse matrices for event logs that contain a larger number of event classes.

```
from tfgen import TFGen

tfgen = TFGen(ec, window_size=500, method=TFGen.ClassicLargeSparse)
```

5.2.4 Implementing New Methods

By deriving from the "BaseMethod" located in "tfgen/methods/base method.py," one can extend the existing methods by creating new method classes. All classes must be placed under "tfgen/methods/" directory. The next event sample must be obtained using method "self.get_next_data()," and the generated feature must be sent to the output using method "self.send_data()". "self.finished" will become "True" if the input stream reaches the end.

```
from tfgen.methods.base_method import BaseMethod

class NewMethod(BaseMethod):
def __init__(self, ec_lookup_table, window_size,
            input_stream, output_stream):
    super().__init__(ec_lookup_table, window_size,
                    input_stream, output_stream)

# entry
def start_processing(self):
    while True:
        # event is a tuple of (case_id, event_attrs)
        event = self.get_next_data()
        # do something
        self.send_data(processed_data)
        if self.finished:
            break
```

Then include the new method in the "TFGen" class found in "tfgen/tfgen.py". Two locations are required to be modified.

```
class TFGen:
    METHOD_CLASSIC = 101
    METHOD_CLASSIC_LARGE_SPARSE = 102
    METHOD_NEW_METHOD = 103  # The first location. New method class

    def _select_method(self, method):
    if method == TFGen.METHOD_CLASSIC:
        return Classic(
            self.ec_lookup, self.window_size,
            self.input_stream, self.output_stream
            )
    elif method == TFGen.METHOD_CLASSIC_LARGE_SPARSE:
        return ClassicLargeSparse(
            self.ec_lookup, self.window_size,
            self.input_stream, self.output_stream
        )
    # The second location. The instance to the new method class
    elif method == TFGen.METHOD_NEW_METHOD:
        return NewMethod(
            self.ec_lookup, self.window_size,
            self.input_stream, self.output_stream
        )
    else:
        raise Exception("Method␣not␣supported")
```

5.3 Future Research

5.3.1 Applications

Our initial experiments show that TFGen is quite competitive and outperforms CICFlowMeter at almost all comparable tasks. Further improvements obviously can be done. Giving the generality and flexibility of the algorithm, it may be applicable to the domain of transition-based problems or data that can be mapped to discrete space. Here are some instances.

- It is applicable to HIDS based on system calls or kernel operations [93]–[95]. The calls are provided as events, and each process will generate a distinct case. With a larger-scaled environment, agents can be deployed on multiple systems for concurrent data collection from a cluster of hosts; TFGen is ideally applicable as long as cases can be modelled and the lengths of cases are finite.
- Computer vision and sensor-based security systems [96], [97] that detect and monitor a series of activities for health and safety measurement. Instead of using the current approaches, the series of classified activities can be modelled as cases where each case can be produced by specific personnel. Each case consists of events that have several attributes such as gesture, department and gender. The benefit could be better overall performance, and the behaviours of multiple personnel are encoded.
- Anomaly detection in the operation of critical infrastructures such as power grid, water and wastewater systems, transportation systems [98], where TFGen can be used in conjunction with numerical sensor readings to encode time-series activity data. For example, sensor data such as the status of valves or the pressure in water systems can be treated as attributes of events, which can then be sent to a centralised processor for feature generation.

These applications are based on the hypothesis that TFGen supports all standard event logs as long as processes can be converted to an event log, and the performance and practicability of using TFGen in these areas can be open research questions for future work.

5.3.2 TFGen Applied on HIDS

To demonstrate the generality of TFGen, a brief experiment is conducted to assess the performance of HIDS using the dataset available at [99]. Two datasets are provided: one containing API calls captured by Cuckoo Sandbox, and another containing kernel calls captured from the custom kernel hook [100]. Cuckoo Sandbox is a tool designed to analyse applications within an isolated environment.

Each dataset has a few sub-datasets that contain captures of different applications, and each sub-datasets contains several folders containing each process named by IDs (e.g. 1, 2 and 3). Since the dataset does not contain a native event log, event logs must be reconstructed using timestamps and function call attributes. All method call attributes from all processes must be merged and sorted according to their timestamps. The kernel set does not provide a global timestamp in each call, therefore, this dataset is not ideal for constructing the event log. Each process folder in the Cuckoo dataset contains a report JSON file which provides API call information. The attributes "category", "status", and "api" were selected from the calls. Table 5.2 displays a portion of the event log.

The event log generated from the "Hippo" sub-dataset, containing only normal processes, serves as training material. The first 200,000 cases are utilised for training, and more than 300 event classes are observed, a significantly higher number than that in the NIDS dataset. A test is conducted using one million cases. Due to dataset constraints, the event log merges 500,000 cases from the Hippo sub-dataset and 500,000 cases from the Virus sub-dataset. A transition matrix exceeding a size of 300^2 is produced by 300 event classes, which is too large to process; considering the testing dataset contains one million events, maintaining the dataset in memory proves challenging. Even with online training, the dimensionality remains too high.

case_id	status	api	category
hippo_0	1	SetErrorMode	system
hippo_0	1	LdrGetDllHandle	system
hippo_0	0	LdrGetProcedureAddress	system
hippo_0	1	GetSystemDirectoryW	file
hippo_0	1	NtClose	system
hippo_0	1	NtOpenKey	registry

TABLE 5.2: A fraction of the event log.

It should be noted that when "api" serves as an attribute, "category" becomes redundant, so only "status" and "api" are required only as attributes at this point.

Several methods are employed to reduce the dimension of the output. The first approach generates transition matrices based on a limited number of the most occurring (frequent) event classes. Any event class (infrequent) not observed or included is categorised as the default event class "Other". Event classes falling within the limited range are called visible event classes, while others are referred to as hidden event classes. The second method applies incremental principal component analysis (IPCA) [101] to the output data. IPCA often supersedes principal component analysis (PCA) when the dataset for decomposition is too large for memory storage. A low-rank approximation of the input data is constructed by IPCA, using a memory quantity independent of the number of input data samples. Modifying the batch size allows for control over memory utilisation, which still depends on the input data's characteristics. In all instances, the IPCA batch size is set to 2000. The reason for selecting a 2000 batch size is the same as the selection of hyper-parameters of the machine learning models in Chapter 4. The third method combines the first two, with all setups detailed below.

- t0_ipca1000: Not limiting the number of event classes and using IPCA with 1000 components.
- t0_ipca100: Not limiting the number of event classes and using IPCA with 100 components.
- t100_ipca0: Limiting the visible event classes to 100 without using IPCA.
- t50_ipca0: Limiting the visible event classes to 50 without using IPCA.
- t25_ipca0: Limiting the visible event classes to 25 without using IPCA.
- t10_ipca0: Limiting the visible event classes to 10 without using IPCA.
- t100_ipca1000: Limiting the visible event classes to 100 and using IPCA with 1000 components.
- t50_ipca100: Limiting the visible event classes to 50 and using IPCA with 100 components.

Before IPCA is applied, the matrices are first flattened, and the outputs are n-dimensional vectors, where n is the number of components. When IPCA is not applied, the outputs remain as matrices; for instance, the configuration t50_ipca0 generates 53×53 matrices (50 event classes and 3 default tokens, SOT, EOT and Other). Autoencoder is utilised for configurations that employ IPCA, whereas convolutional autoencoder is utilised for configurations that do not employ IPCA. The convolutional autoencoder is designed for image processing which uses convolutional layers instead of dense layers. In the case of t50_ipca0, the shape of the input tensors will be $1 \times 53 \times 53$ where 1 is the image channel number.

The autoencoder has 9 layers, except the first two layers which have a number of neurons equal to the input dimension, each layer behind has half the amount of neurons of the previous layer. The 5th layer is the middle layer and the following decoder layers are symmetric to the first 4 layers. The convolutional autoencoder has 7 layers, with 3 layers of convolutional layers, a max-pooling layer in the middle and 3 transposed convolution operators as the decoder layers.

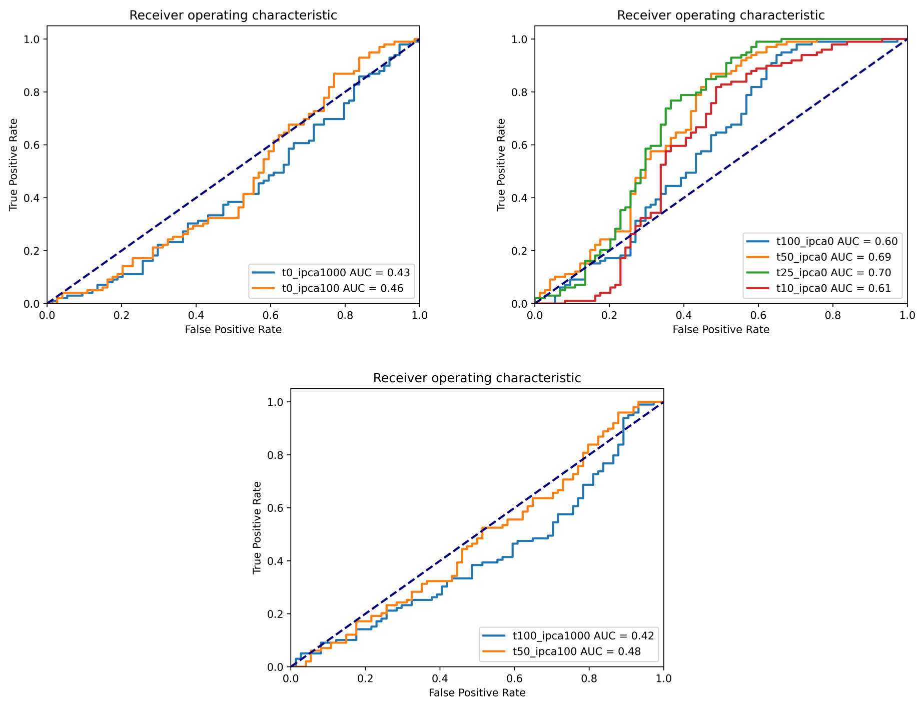


FIGURE 5.1: ROC-AUC of case-level detection. Top left: Using IPCA only. Top right: Using limited event classes only. Bottom: using IPCA and limited event classes combined.

The measurement is conducted at the case level, with the outlier factor of a case determined by the event with the highest outlier factor. Figure 5.1 shows the ROC and AUC (area under the curve) of all instances. The results for feature reduction using IPCA only (top-left chart) are not promising as the AUC is less than 0.5, however, reducing dimension by limiting the number of event classes gives better results compared to using IPCA (top-right chart). Also, setups t50_ipca0 and t25_ipca0 yield the best results. While limiting the number of event classes has a clear performance impact, adding extra IPCA reduction on top of limited event classes seems to worsen the performance (bottom chart).

The processing speed and AUC of all instances are documented in Table 5.3. It becomes evident that employing 25 and 50 visible event classes yields better results, while the processing speed remains relatively optimal. Utilising a larger number of visible event classes results in a decreased feature generation speed, which appears to not maintain the $O(1)$ complexity. This can be attributed to the memory copy bottleneck of large matrices and other overheads in offline dataset generation, issues that should not arise in online processing. Additionally, IPCA significantly slows down processing without enhancing accuracy.

Setting	Best AUC	Feature Gen. Speed (events/s)	Inference Speed (events/s)
t0_ipca1000	0.43	26.29	67667.61
t0_ipca100	0.46	35.38	84521.45
t100_ipca0	0.60	34843.18	9481.63
t50_ipca0	0.69	37008.24	28341.88
t25_ipca0	0.70	74243.74	48003.98
t10_ipca0	0.61	80645.42	52056.80
t100_ipca1000	0.42	134.81	65358.06
t50_ipca100	0.48	331.80	84199.91

TABLE 5.3: Processing speed and performance comparison.

Table 5.3 documents the processing speed and AUC for each instance. It is not difficult to see that using 25 and 50 visible event classes yields the best results and processing speed. When using a greater number of visible event classes, the rate of feature generation decreases, suggesting that the $O(1)$ complexity is not maintained; however, this is due to the memory copying of large matrices and other overheads that occur during offline dataset generation, which should not occur during online processing. In addition, IPCA significantly slows down processing without improving accuracy.

Further investigation into TFGen's capabilities is conducted by removing the "status" attribute and substituting it with API calls. This approach reduces the number of observable event classes to around 200. Figure 5.2 displays the results, which resemble those obtained when using both API call names and status as attributes. This may indicate that the significance of the status attribute in this experiment is reduced. All setups are detailed below. As the processing speed is similar to that of the previous set of setups, it will not be demonstrated again.

- api_t0_ipca1000: Not limiting the number of event classes and using IPCA with 1000 components.
- api_t0_ipca100: Not limiting the number of event classes and using IPCA with 100 components.
- api_t100_ipca0: Limiting the visible event classes to 100 without using IPCA.
- api_t50_ipca0: Limiting the visible event classes to 50 without using IPCA.
- api_t25_ipca0: Limiting the visible event classes to 25 without using IPCA.
- api_t10_ipca0: Limiting the visible event classes to 10 without using IPCA.
- api_t100_ipca1000: Limiting the visible event classes to 100 and using IPCA with 1000 components.
- api_t50_ipca100: Limiting the visible event classes to 50 and using IPCA with 100 components.

Besides using only API calls, another feature reduction method is employed, involving a simple linear transformation on an all-one vector $\mathbf{e} = \mathbf{1}^n$, where n represents the number of event classes. Given the $n \times n$ transition matrix A, the feature vector $\mathbf{x} = A\mathbf{e}$ is obtained. In certain instances, this method takes the place of IPCA, with setups detailed below.

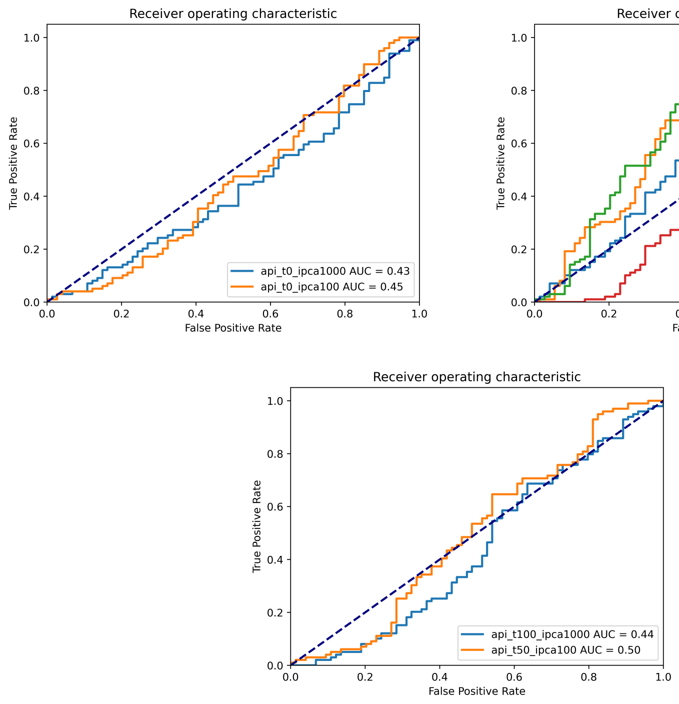


FIGURE 5.2: ROC-AUC of detection performance when using API call names only. Top left: Using IPCA only. Top right: Using limited event classes only. Bottom: using IPCA and limited event classes combined.

- api_t0_lt: Not limiting the number of event classes and using linear transformation for feature reduction.
- api_t100_lt: Limiting the visible event classes to 100 and using linear transformation for feature reduction.
- api_t50_lt: Limiting the visible event classes to 50 and using linear transformation for feature reduction.
- api_t25_lt: Limiting the visible event classes to 25 and using linear transformation for feature reduction.

Compared to IPCA, this approach is considerably easier and quicker. In comparison to less than 400 events/s when utilising IPCA with setup api_t50_ipca100, the feature generation rate is over 34,000 events/s with setup api_t100_lt. The performance is displayed in Figure 5.3 and Table 5.4. For settings that do not restrict the event classes, limit visible event classes to 100, and limit visible event classes to 50, there is a noticeable speed improvement.

Setting	Best AUC	Feature Gen. Speed (events/s)	Inference Speed (events/s)
api_t0_lt	0.64	16344.79	84504.82
api_t100_lt	0.76	34940.01	78582.83
api_t50_lt	0.74	35205.75	83802.39
api_t25_lt	0.60	74669.47	82856.19

TABLE 5.4: Processing speed and performance comparison for linear transformation method.

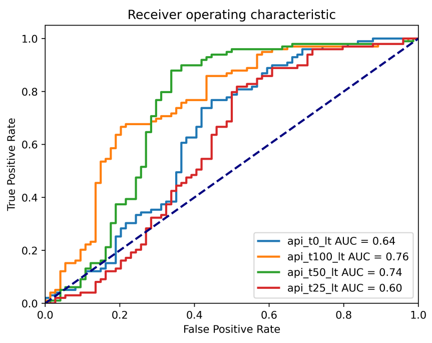


FIGURE 5.3: ROC-AUC of detection performance when using API call names only and reducing feature with the linear transformation.

In a concluding test, an experiment using category as the sole attribute was conducted, offering a higher level of process abstraction. Thus, only sixteen event classes are observed. Due to the reduced number of event classes, no additional dimension reduction is performed, and both autoencoder and convolutional autoencoder are employed. Figure 5.4 (left) depicts the experimental outcome. The result is not promising and in this instance, the autoencoder produces slightly superior results to the convolutional autoencoder. This does not imply that autoencoders are always superior to convolutional autoencoders. Based on two prior setups, Figure 5.4 (right) depicts a verification using an autoencoder rather than a convolutional autoencoder. As can be seen, the autoencoder is not superior to the convolutional autoencoder. Therefore, the selection between anomaly detectors may relate to the problem itself.

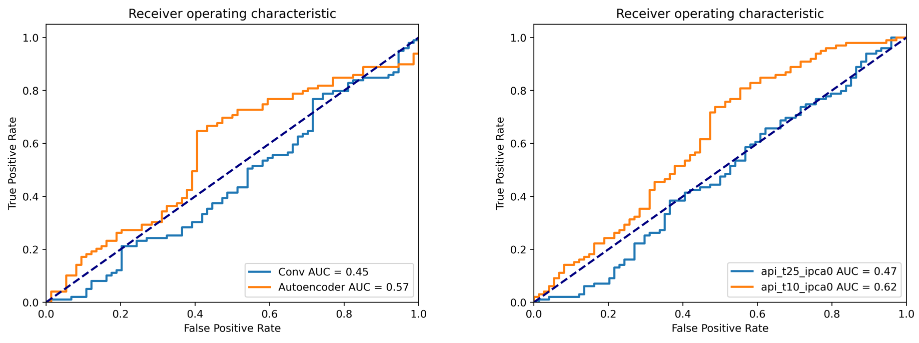


FIGURE 5.4: Left: ROC-AUC of detection performance when using category attribute only. Right: Using limited event classes only.

From the results of these HIDS experiments, it has been deduced that using fewer visible event classes can produce superior outcomes in certain situations. The rationale could be that the frequent event classes in normal cases indicate normal behaviour, while the less significant hidden event classes are grouped as the default event class. In such a scenario, the hidden event classes are consolidated, and their importance is reduced.

5.4 Known Issues

These are some disadvantages of TFGen, for instance, a smaller window size may result in sparse matrices, necessitating an adjustment of the window size based on the problem and anomaly detector. When event classes contain numerous attributes, such as using the HIDS dataset, the output matrices may be of high dimension. Using dimension reduction techniques such as IPCA or a list of limited event classes is possible; however, this may require extra computational resources.

Further examinations are necessary, including conducting intrusion detection using native host-level event logs with TFGen. As mentioned earlier, the event log utilised is created from the Cuckoo log and does not incorporate function calls from sub-processes. In other words, the constructed event log might not provide as comprehensive a global observation of events when compared to the native event log.

5.5 Summary

This chapter recalls the last research question. A generalised version of the online process mining algorithm was provided to extend the algorithm to other areas that can potentially transform their data into event logs. To help future research, a Python implementation of this technique has been developed. This implementation is able to efficiently convert standard event log data into transition matrices. For testing, HIDS data was used, which shows the algorithm works as expected and a smaller number of visible event classes leads to superior performance and higher efficiency. A few examples of application areas using this feature generator were given.

At the point of this thesis, several datasets have been released by Canadian Institute for Cybersecurity[4]. These datasets include new DDoS-specific datasets, IoT (Internet of Things) datasets, host-level datasets and more. Future research can focus on newly released datasets and issues described in the previous section. For instance, employing process mining techniques such as aggregation used in fuzzy mining, to minimize the dimension; or other feature reduction strategies to address the dimensionality problem. In addition, research can be extended to areas outside the domain of intrusion detection, such as anomaly detection in critical infrastructures and security.

[4]https://www.unb.ca/cic/datasets/index.html

Bibliography

[1] W. M. Van der Aalst, *Process mining: data science in action*. Springer, 2016.

[2] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and parallel databases*, vol. 14, no. 1, pp. 5–51, 2003.

[3] W. M. van der Aalst and S. Jablonski, "Dealing with workflow change: Identification of issues and solutions," *Computer systems science and engineering*, vol. 15, no. 5, pp. 267–276, 2000.

[4] W. Van Der Aalst, *Process mining: discovery, conformance and enhancement of business processes*. Springer, 2011, vol. 2.

[5] Cisco, *Global 2021 forecast highlights*. [Online]. Available: https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2021_Forecast_Highlights.pdf.

[6] norton, *10 cyber security facts and statistics for 2018*. [Online]. Available: https://us.norton.com/internetsecurity-emerging-threats-10-facts-about-todays-cybersecurity-landscape-that-you-should-know.html.

[7] O. Yoachimik and V. Ganti, *Ddos attack trends for q4 2021*, 2022. [Online]. Available: https://blog.cloudflare.com/ddos-attack-trends-for-2021-q4/.

[8] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks.," in *Lisa*, vol. 99, 1999, pp. 229–238.

[9] V. Jyothsna, R. Prasad, and K. M. Prasad, "A review of anomaly based intrusion detection systems," *International Journal of Computer Applications*, vol. 28, no. 7, pp. 26–35, 2011.

[10] W. Lee and S. Stolfo, "Data mining approaches for intrusion detection," 1998.

[11] G. M. Borkar, L. H. Patil, D. Dalgade, and A. Hutke, "A novel clustering approach and adaptive svm classifier for intrusion detection in wsn: A data mining concept," *Sustainable Computing: Informatics and Systems*, vol. 23, pp. 120–135, 2019.

[12] N. Ashraf, W. Ahmad, and R. Ashraf, "A comparative study of data mining algorithms for high detection rate in intrusion detection system," *Annals of Emerging Technologies in Computing (AETiC), Print ISSN*, pp. 2516–0281, 2018.

[13] A. F. M. Agarap, "A neural network architecture combining gated recurrent unit (gru) and support vector machine (svm) for intrusion detection in network traffic data," in *Proceedings of the 2018 10th International Conference on Machine Learning and Computing*, ACM, 2018, pp. 26–30.

[14] J. Kim, Y. Shin, E. Choi, *et al.*, "An intrusion detection model based on a convolutional neural network," *Journal of Multimedia Information System*, vol. 6, no. 4, pp. 165–172, 2019.

[15] Y.-F. Hsu, Z. He, Y. Tarutani, and M. Matsuoka, "Toward an online network intrusion detection system based on ensemble learning," in *2019 IEEE 12th international conference on cloud computing (CLOUD)*, IEEE, 2019, pp. 174–178.

[16] S. Roshan, Y. Miche, A. Akusok, and A. Lendasse, "Adaptive and online network intrusion detection system using clustering and extreme learning machines," *Journal of the Franklin Institute*, vol. 355, no. 4, pp. 1752–1779, 2018.

[17] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," *arXiv preprint arXiv:1802.09089*, 2018.

[18] R. Vijayasarathy, S. V. Raghavan, and B. Ravindran, "A system approach to network modeling for ddos detection using a naive bayesian classifier," in *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*, IEEE, 2011, pp. 1–10.

[19] W. Lee and D. Xiang, "Information-theoretic measures for anomaly detection," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, IEEE, 2000, pp. 130–143.

[20] J. David and C. Thomas, "Ddos attack detection using fast entropy approach on flow-based network traffic," *Procedia Computer Science*, vol. 50, pp. 30–36, 2015.

[21] 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*, Ieee, 2009, pp. 1–6.

[22] UNB, *Nsl-kdd dataset*. [Online]. Available: `https://www.unb.ca/cic/datasets/nsl.html`.

[23] L Dhanabal and S. Shantharajah, "A study on nsl-kdd dataset for intrusion detection system based on classification algorithms," *International journal of advanced research in computer and communication engineering*, vol. 4, no. 6, pp. 446–452, 2015.

[24] UNB, *Cic-ids2017 dataset*. [Online]. Available: `https://www.unb.ca/cic/datasets/ids-2017.html`.

[25] UNB, *Cse-cic-ids2018 dataset*. [Online]. Available: `https://www.unb.ca/cic/datasets/ids-2018.html`.

[26] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *computers & security*, vol. 31, no. 3, pp. 357–374, 2012.

[27] T. AbuHmed, A. Mohaisen, and D. Nyang, "A survey on deep packet inspection for intrusion detection systems," *arXiv preprint arXiv:0803.0037*, 2008.

[28] R.-H. Hwang, M.-C. Peng, V.-L. Nguyen, and Y.-L. Chang, "An lstm-based deep learning approach for classifying malicious traffic at the packet level," *Applied Sciences*, vol. 9, no. 16, p. 3414, 2019.

[29] X. Jing, Z. Yan, and W. Pedrycz, "Security data collection and data analytics in the internet: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 586–618, 2018.

[30] A. H. Lashkari, G. Draper-Gil, M. S. I. Mamun, and A. A. Ghorbani, "Characterization of tor traffic using time based features.," in *ICISSp*, 2017, pp. 253–262.

[31] A. Wespi, M. Dacier, and H. Debar, “Intrusion detection using variable-length audit trail patterns,” in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2000, pp. 110–129.

[32] I. Rigoutsos and A. Floratos, “Combinatorial pattern discovery in biological sequences: The teiresias algorithm.,” *Bioinformatics (Oxford, England)*, vol. 14, no. 1, pp. 55–67, 1998.

[33] Y. Zhong, W. Chen, Z. Wang, *et al.*, “Helad: A novel network anomaly detection model based on heterogeneous ensemble learning,” *Computer Networks*, vol. 169, p. 107 049, 2020.

[34] W. M. Van der Aalst and A. K. A. de Medeiros, “Process mining and security: Detecting anomalous process executions and checking process conformance,” *Electronic Notes in Theoretical Computer Science*, vol. 121, pp. 3–21, 2005.

[35] V. P. Mishra and B. Shukla, “Process mining in intrusion detection-the need of current digital world,” in *International Conference on Advanced Informatics for Computing Research*, Springer, 2017, pp. 238–246.

[36] V. P. Mishra, J. Dsouza, and L. Elizabeth, “Analysis and comparison of process mining algorithms with application of process mining in intrusion detection system,” in *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, IEEE, 2018, pp. 613–617.

[37] W. M. Van der Aalst, B. F. Van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. Weijters, “Workflow mining: A survey of issues and approaches,” *Data & knowledge engineering*, vol. 47, no. 2, pp. 237–267, 2003.

[38] C. W. Günther, “Process mining in flexible environments,” 2009.

[39] W. M. van der Aalst, “Relating process models and event logs - 21 conformance propositions,” in *ATAED@Petri Nets/ACSD*, 2018.

[40] S. J. Leemans, D. Fahland, and W. M. van der Aalst, “Discovering block-structured process models from event logs-a constructive approach,” in *International conference on applications and theory of Petri nets and concurrency*, Springer, 2013, pp. 311–329.

[41] S. J. Leemans, D. Fahland, and W. M. van der Aalst, “Discovering block-structured process models from incomplete event logs,” in *International conference on applications and theory of petri nets and concurrency*, Springer, 2014, pp. 91–110.

[42] S. J. Leemans, D. Fahland, and W. M. van der Aalst, “Scalable process discovery with guarantees,” in *Enterprise, Business-Process and Information Systems Modeling*, Springer, 2015, pp. 85–101.

[43] W. Van der Aalst, T. Weijters, and L. Maruster, “Workflow mining: Discovering process models from event logs,” *IEEE transactions on knowledge and data engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.

[44] W. Reisig, *A primer in Petri net design*. Springer Science & Business Media, 2012.

[45] W. M. van der Aalst, *A practitioner's guide to process mining: Limitations of the directly-follows graph*, 2019.

[46] C. W. Günther and W. M. Van Der Aalst, "Fuzzy mining–adaptive process simplification based on multi-perspective metrics," in *International conference on business process management*, Springer, 2007, pp. 328–343.

[47] F. Töpfer and W. Pillewizer, "The principles of selection," *The Cartographic Journal*, vol. 3, no. 1, pp. 10–16, 1966.

[48] B. P. Buttenfield and R. B. McMaster, *Map Generalization: Making rules for knowledge representation*. Longman Scientific & Technical London, 1991.

[49] V. Cerf and R. Kahn, "A protocol for packet network intercommunication," *IEEE Transactions on communications*, vol. 22, no. 5, pp. 637–648, 1974.

[50] J. Postel, *Rfc0791: Internet protocol*, 1981.

[51] J. Postel *et al.*, "Transmission control protocol," 1981.

[52] K. Ramakrishnan, S. Floyd, D. Black, *et al.*, "The addition of explicit congestion notification (ecn) to ip," 2001.

[53] W. R. Stevens, *TCP/IP illustrated vol. I: the protocols*. Pearson Education India, 1993.

[54] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh, "Nat behavioral requirements for tcp," RFC 5382 (Best Current Practice), Tech. Rep., 2008.

[55] X. Zhang, S. F. Wu, Z. Fu, and T.-L. Wu, "Malicious packet dropping: How it might impact the tcp performance and how we can detect it," in *Proceedings 2000 International Conference on Network Protocols*, IEEE, 2000, pp. 263–272.

[56] H. Lundqvist and G. Karlsson, "Tcp with end-to-end fec," in *International Zurich Seminar on Communications, 2004*, IEEE, 2004, pp. 152–155.

[57] A Chockalingam, M. Zorzi, and V. Tralli, "Wireless tcp performance with link layer fec/arq," in *1999 IEEE International Conference on Communications (Cat. No. 99CH36311)*, IEEE, vol. 2, 1999, pp. 1212–1216.

[58] C. Barakat and E. Altman, "Bandwidth tradeoff between tcp and link-level fec," *Computer networks*, vol. 39, no. 2, pp. 133–150, 2002.

[59] N. Spring, D. Wetherall, and D. Ely, "Robust explicit congestion notification (ecn) signaling with nonces," RFC 3540, June, Tech. Rep., 2003.

[60] D. Myers, S. Suriadi, K. Radke, and E. Foo, "Anomaly detection for industrial control systems using process mining," *Computers & Security*, vol. 78, pp. 103–125, 2018.

[61] A. Weijters, W. M. van Der Aalst, and A. A. De Medeiros, "Process mining with the heuristics miner-algorithm," *Technische Universiteit Eindhoven, Tech. Rep. WP*, vol. 166, no. July 2017, pp. 1–34, 2006.

[62] M. Bruno, P. Ibañez, T. Techera, D. Calegari, and G. Betarte, "Exploring the application of process mining techniques to improve web application security," in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021, pp. 1–10.

[63] M. Ebrahim and S. A. H. Golpayegani, "Anomaly detection in business processes logs using social network analysis," *Journal of Computer Virology and Hacking Techniques*, vol. 18, no. 2, pp. 127–139, 2022.

[64] A Eckleder and T Freytag, "Woped a tool for teaching, analyzing and visualizing workflow nets," *Petri Net Newsletter*, vol. 75, pp. 3–8, 2008.

[65] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization.," *ICISSp*, vol. 1, pp. 108–116, 2018.

[66] H. Verbeek, J. Buijs, B. Van Dongen, and W. M. van der Aalst, "Prom 6: The process mining toolkit," *Proc. of BPM Demonstration Track*, vol. 615, pp. 34–39, 2010.

[67] C. W. Günther and A. Rozinat, "Disco: Discover your processes," in *BPM*, 2012.

[68] W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, *et al.*, "Prom 4.0: Comprehensive support for real process analysis," in *Petri Nets and Other Models of Concurrency – ICATPN 2007*, J. Kleijn and A. Yakovlev, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 484–494, ISBN: 978-3-540-73094-1.

[69] B. F. Van Dongen, A. K. A. de Medeiros, H. Verbeek, A. Weijters, and W. M. van Der Aalst, "The prom framework: A new era in process mining tool support," in *International conference on application and theory of petri nets*, Springer, 2005, pp. 444–454.

[70] H. Verbeek, B. F. van Dongen, J. Mendling, and W. M. van der Aalst, "Interoperability in the prom framework.," *EMOI-INTEROP*, vol. 6, 2006.

[71] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Tcp, udp, and sockets: Rigorous and experimentally-validated behavioural specification: Volume 1: Overview," University of Cambridge, Computer Laboratory, Tech. Rep., 2005.

[72] A. Rozinat and W. M. Van der Aalst, "Conformance checking of processes based on monitoring real behavior," *Information Systems*, vol. 33, no. 1, pp. 64–95, 2008.

[73] D. Schwartz, S Stoecklin, and E Yilmaz, "Case-based agents for packet-level intrusion detection in ad hoc networks," in *Proc, of the 17th Int. Symp. on Computer and Information Sciences*, vol. 7, 2002, p. 59.

[74] B. Wang, Y. Su, M. Zhang, and J. Nie, "A deep hierarchical network for packet-level malicious traffic detection," *IEEE Access*, vol. 8, pp. 201 728–201 740, 2020.

[75] S. J. van Zelst, A. Bolt, M. Hassani, B. F. van Dongen, and W. M. van der Aalst, "Online conformance checking: Relating event streams to process models using prefix-alignments," *International Journal of Data Science and Analytics*, vol. 8, pp. 269–284, 2019.

[76] L. Dong, N. Yang, W. Wang, *et al.*, "Unified language model pre-training for natural language understanding and generation," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[77] J. Hirschberg and C. D. Manning, "Advances in natural language processing," *Science*, vol. 349, no. 6245, pp. 261–266, 2015.

[78] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *ieee Computational intelligenCe magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[79] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[80] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[81] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 international conference on engineering and technology (ICET)*, Ieee, 2017, pp. 1–6.

[82] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[83] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[84] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information processing & management*, vol. 45, no. 4, pp. 427–437, 2009.

[85] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[86] Z. Li, Y. Zhao, N. Botta, C. Ionescu, and X. Hu, "Copod: Copula-based outlier detection," in *2020 IEEE International Conference on Data Mining (ICDM)*, IEEE, 2020, pp. 1118–1123.

[87] H.-P. Kriegel, M. Schubert, and A. Zimek, "Angle-based outlier detection in high-dimensional data," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2008, pp. 444–452.

[88] Z. He, X. Xu, and S. Deng, "Discovering cluster-based local outliers," *Pattern recognition letters*, vol. 24, no. 9-10, pp. 1641–1650, 2003.

[89] M. Goldstein and A. Dengel, "Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm," *KI-2012: poster and demo track*, vol. 9, 2012.

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

[91] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: Identifying density-based local outliers," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 93–104.

[92] M.-L. Shyu, S.-C. Chen, K. Sarinnapakorn, and L. Chang, "A novel anomaly detection scheme based on principal component classifier," Miami Univ Coral Gables Fl Dept of Electrical and Computer Engineering, Tech. Rep., 2003.

[93] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen, "Host-based intrusion detection system with system calls: Review and future trends," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.

[94] J. Byrnes, T. Hoang, N. N. Mehta, and Y. Cheng, "A modern implementation of system call sequence based host-based intrusion detection systems," in *2020 Second IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, IEEE, 2020, pp. 218–225.

[95] M. Kadar, S. Tverdyshev, and G. Fohler, "System calls instrumentation for intrusion detection in embedded mixed-criticality systems," in *4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[96] L. Ding, W. Fang, H. Luo, P. E. Love, B. Zhong, and X. Ouyang, "A deep hybrid learning model to detect unsafe behavior: Integrating convolution neural networks and long short-term memory," *Automation in Construction*, vol. 86, pp. 118–124, 2018, ISSN: 0926-5805.

[97] Z. Luo, J.-T. Hsieh, N. Balachandar, *et al.*, "Computer vision-based descriptive analytics of seniors' daily activities for long-term health monitoring," *Machine Learning for Healthcare (MLHC)*, vol. 2, p. 1, 2018.

[98] M. Gauthama Raman, N. Somu, and A. P. Mathur, "Anomaly detection in critical infrastructure using probabilistic neural network," in *International Conference on Applications and Techniques in Information Security*, Springer, 2019, pp. 129–141.

[99] M. Nunes, *Dynamic Malware Analysis kernel and user-level calls*, version 1.0, Mar. 2018. DOI: 10.5281/zenodo.1203289. [Online]. Available: https://doi.org/10.5281/zenodo.1203289.

[100] M. Nunes, P. Burnap, O. Rana, P. Reinecke, and K. Lloyd, "Getting to the root of the problem: A detailed comparison of kernel and user level data for dynamic malware analysis," *Journal of Information Security and Applications*, vol. 48, p. 102 365, 2019.

[101] D. A. Ross, J. Lim, R.-S. Lin, and M.-H. Yang, "Incremental learning for robust visual tracking," *International journal of computer vision*, vol. 77, no. 1, pp. 125–141, 2008.

Appendix A

Figures

A.1 Mined Models using ProM

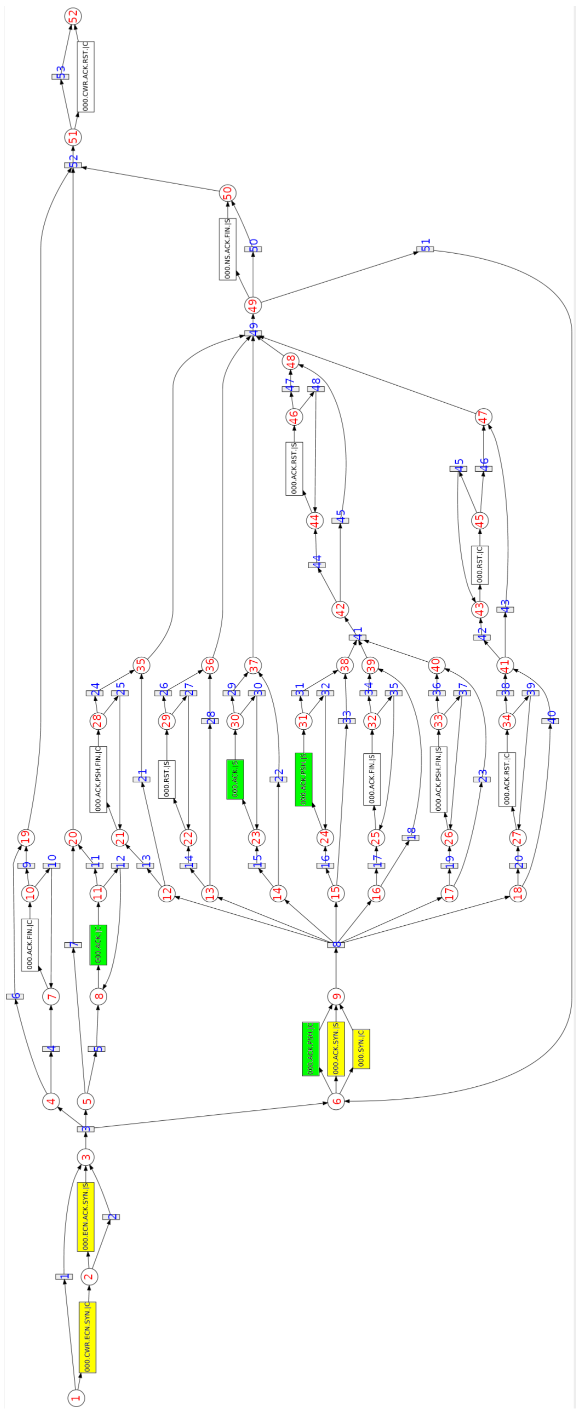


FIGURE A.1: process model example.

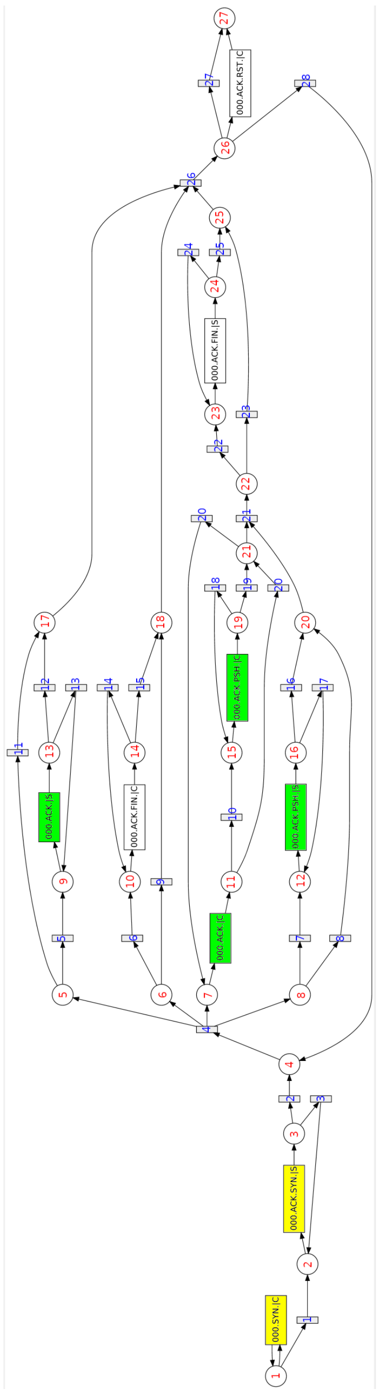


FIGURE A.2: first process model with 5 cases.

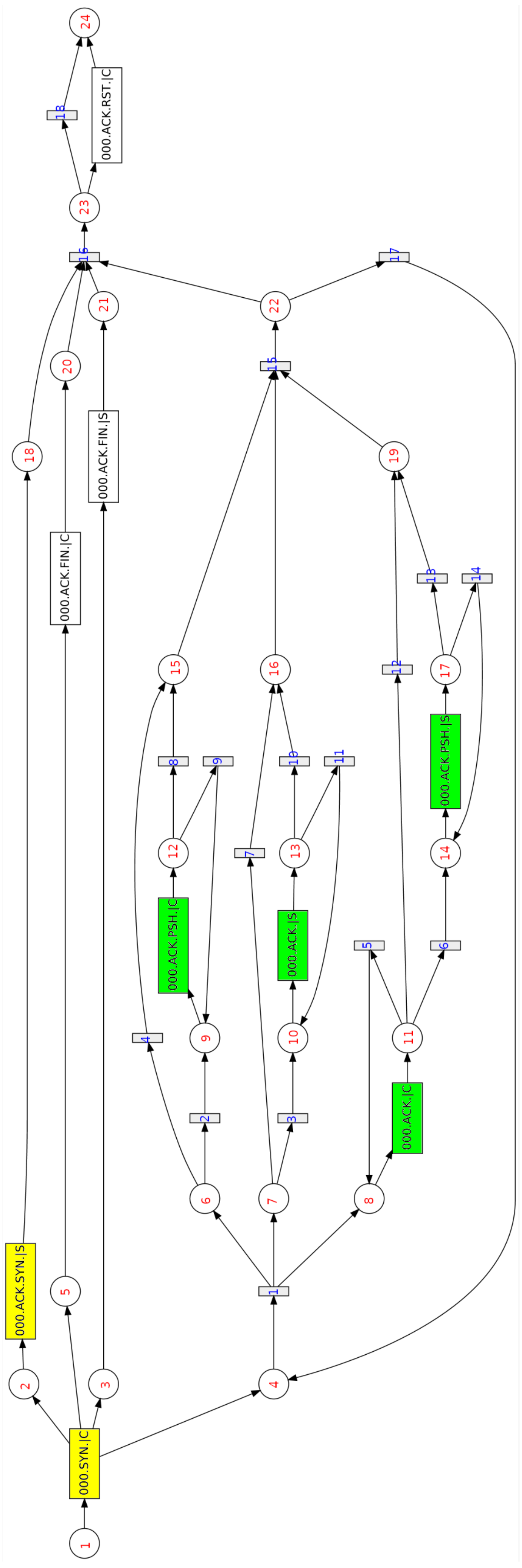


FIGURE A.3: second process model with 5 cases.

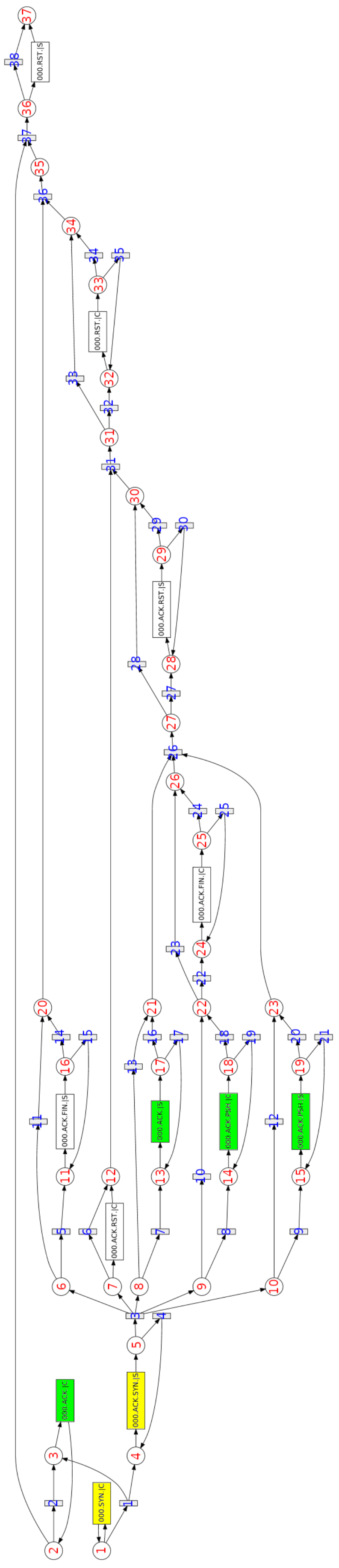


FIGURE A.4: first process model with 100 cases.

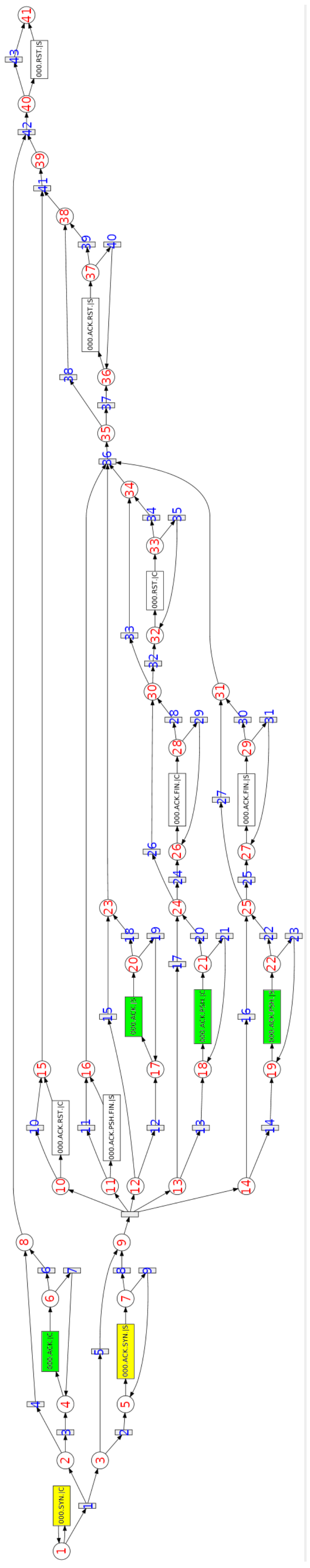


FIGURE A.5: second process model with 100 cases.

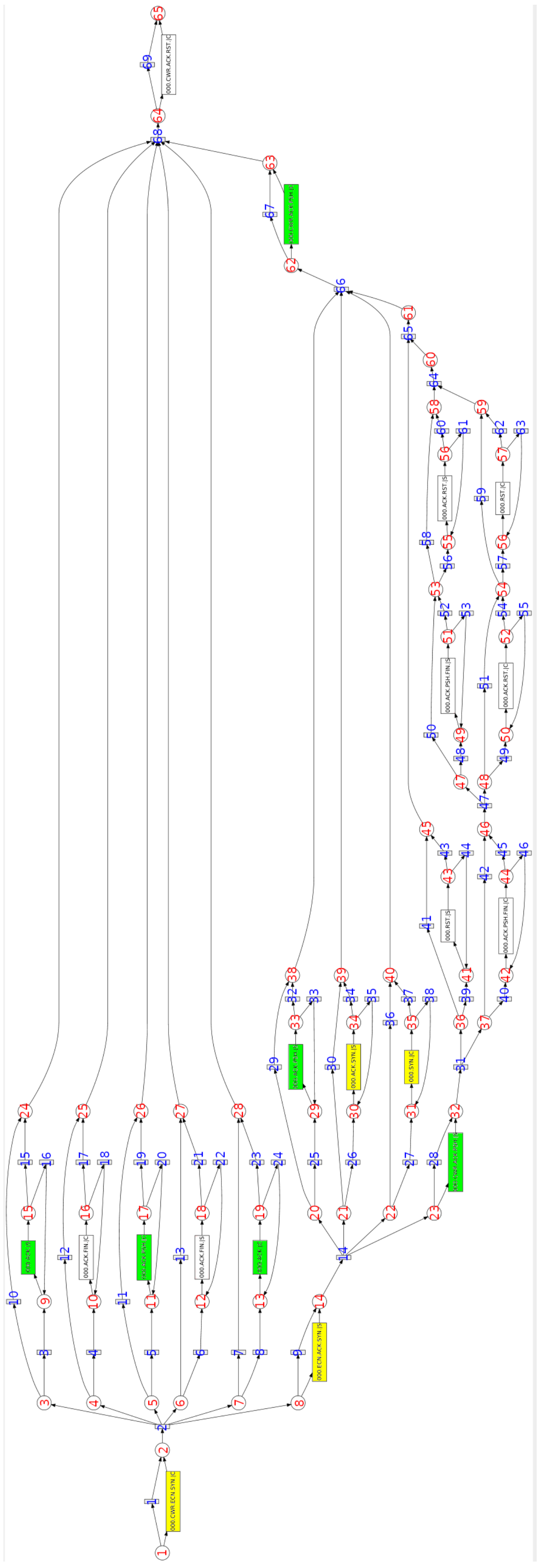


FIGURE A.6: first process model with 20k cases.

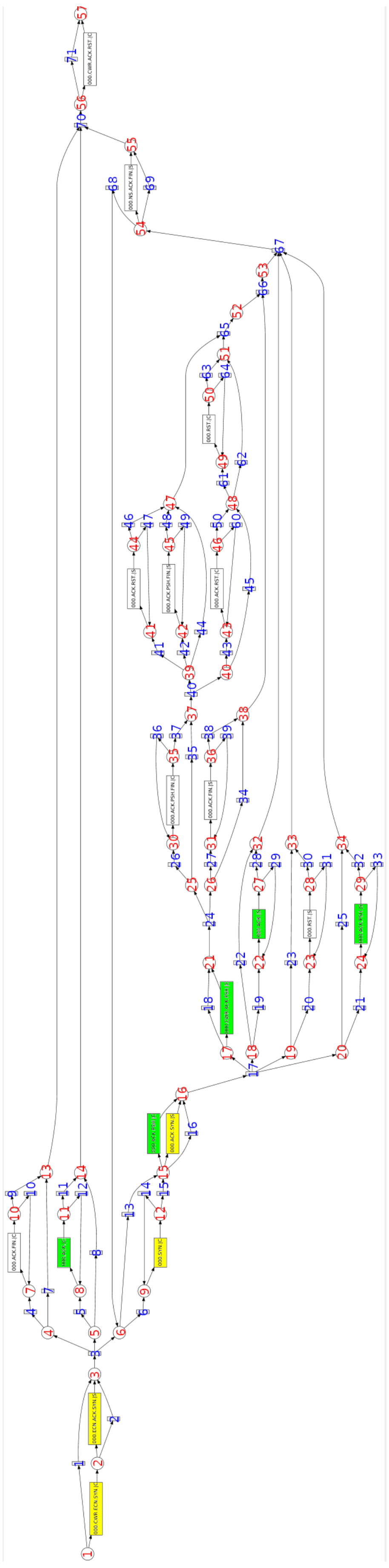


FIGURE A.7: second process model with 20k cases.

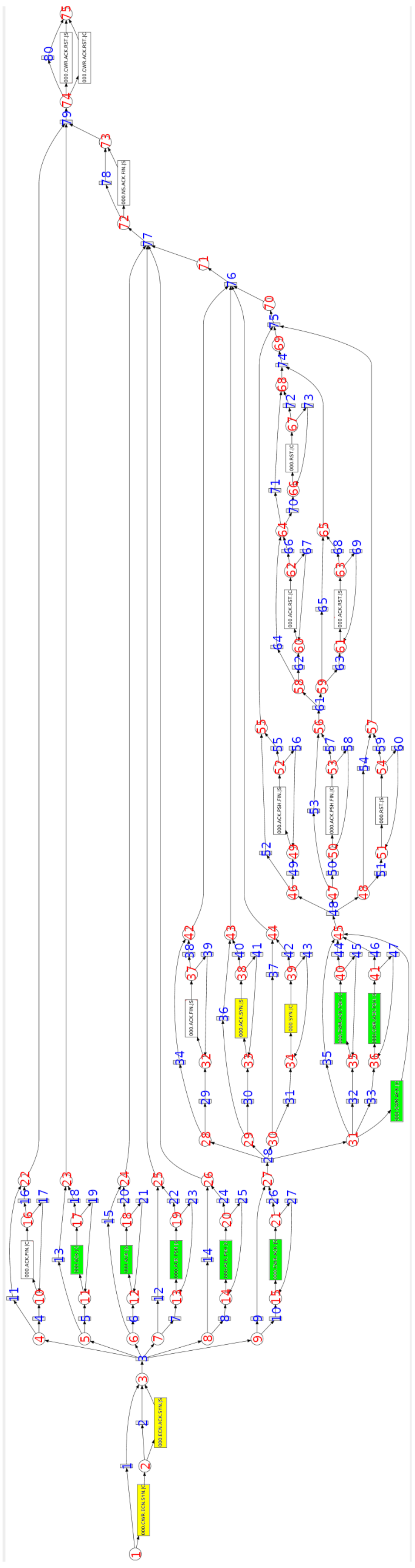


FIGURE A.8: first process model with 100k cases.

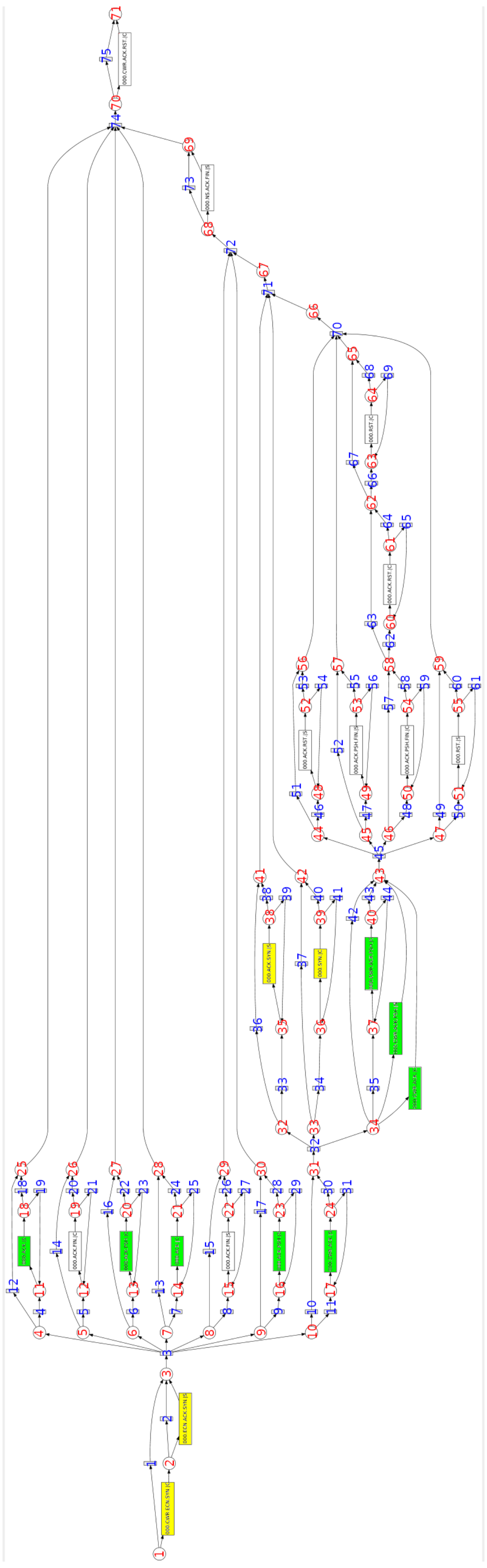


FIGURE A.9: second process model with 100k cases.

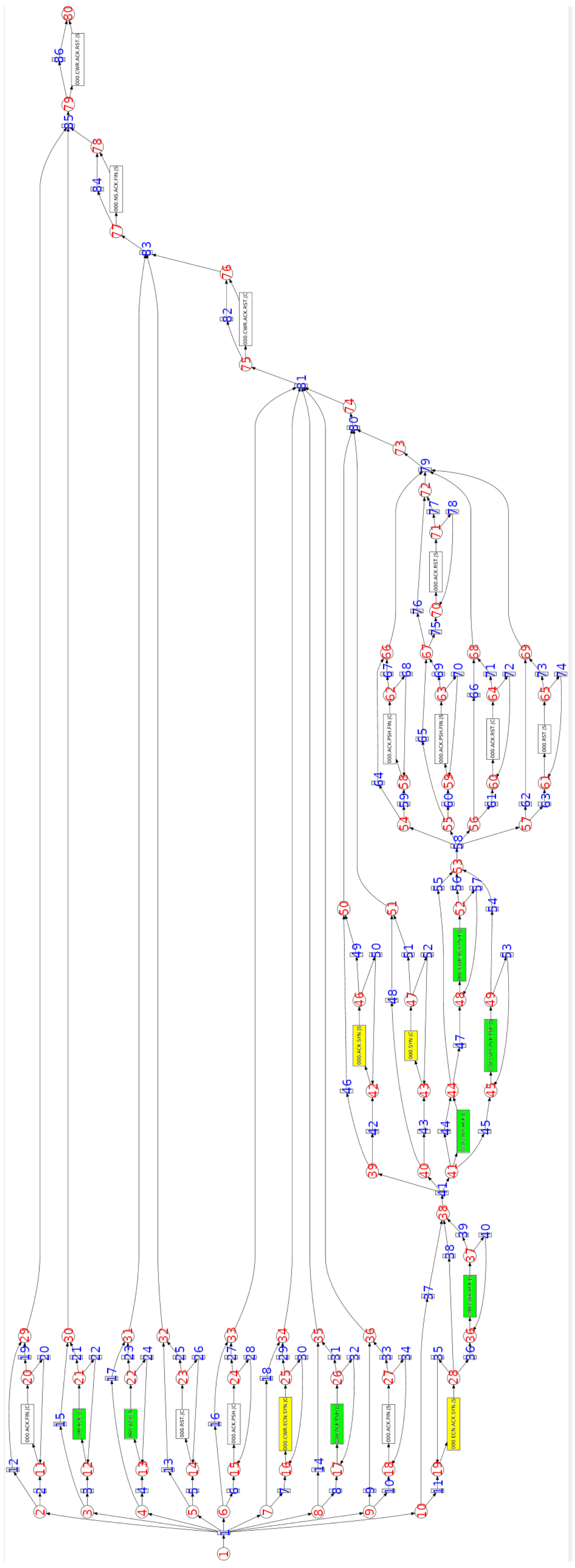


FIGURE A.10: process model with all cases.

A.2 Mined Models using Disco

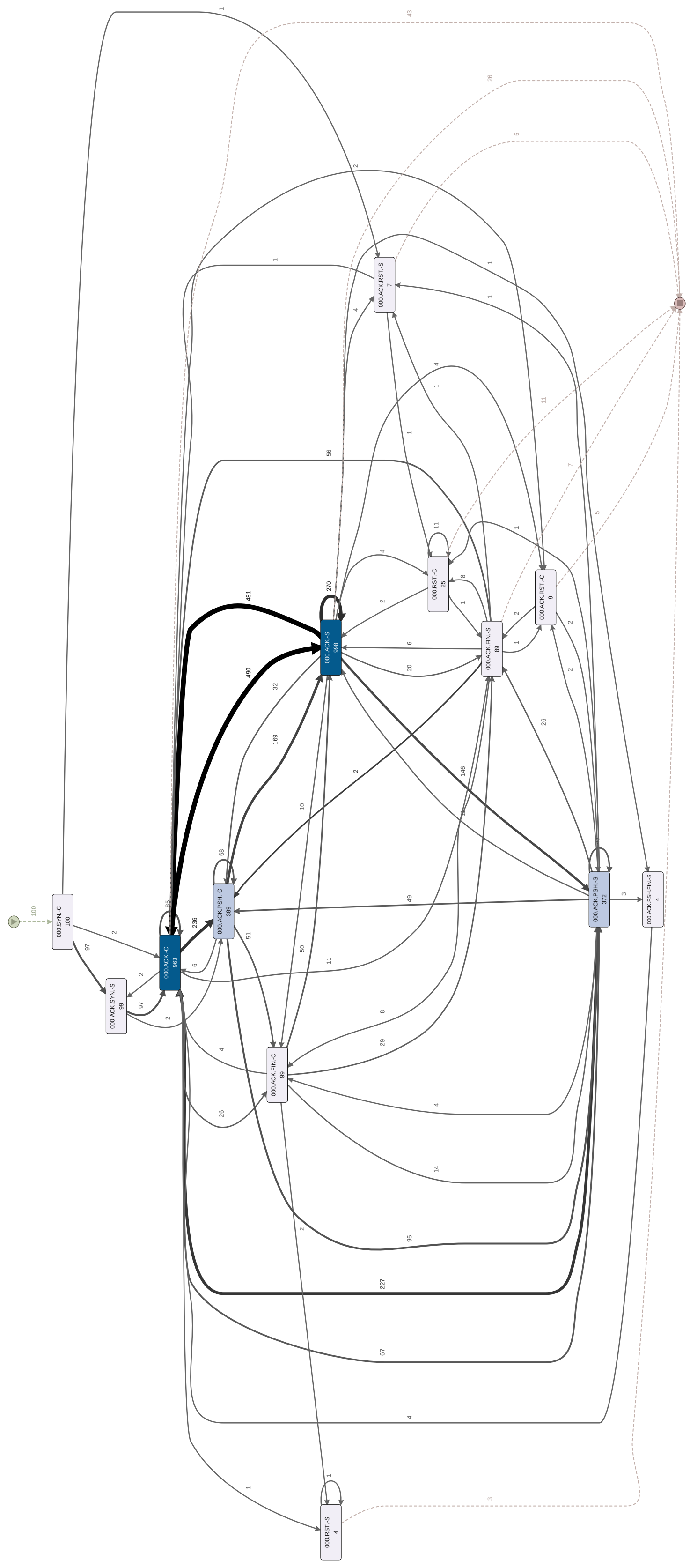


FIGURE A.11: example process model.

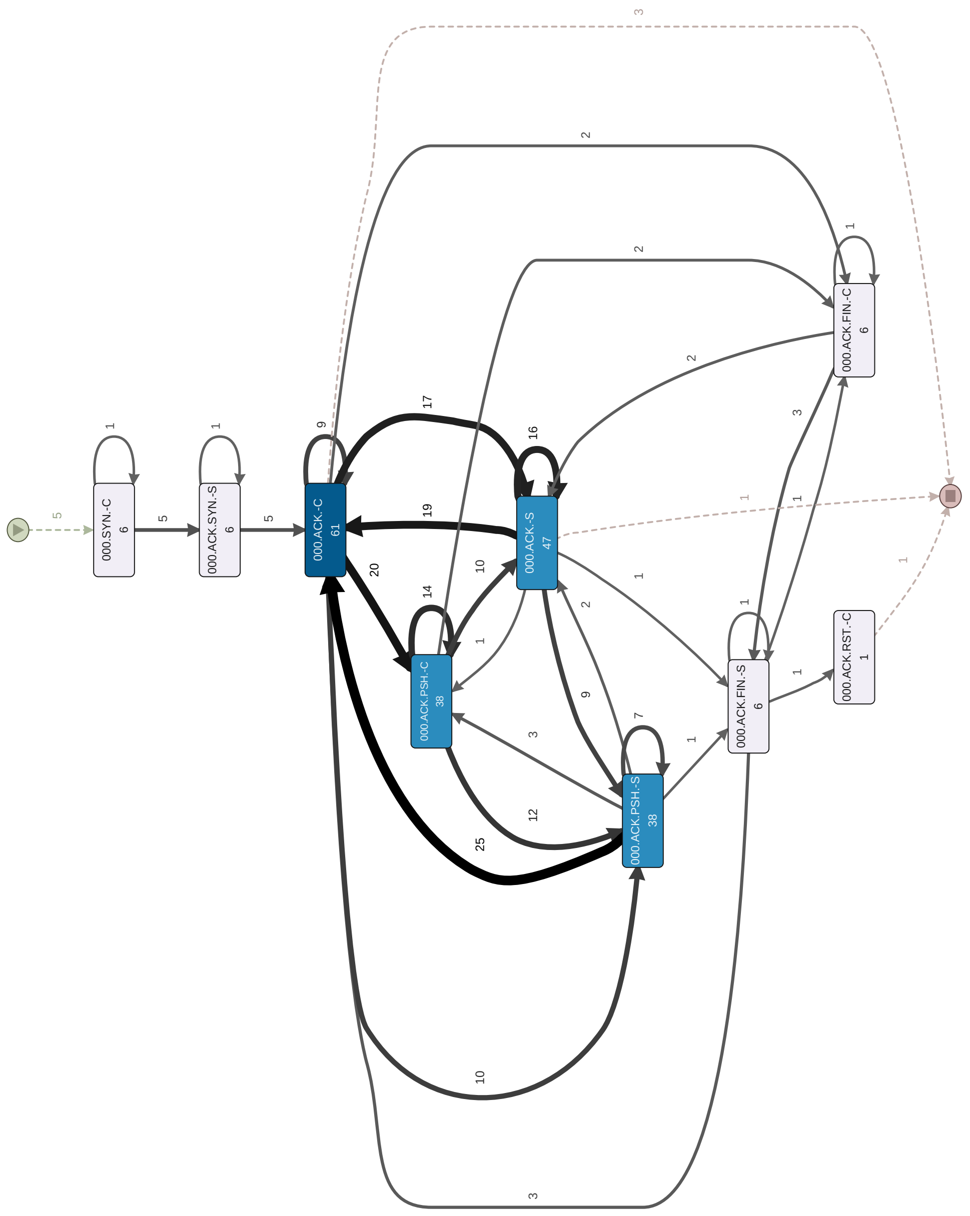


FIGURE A.12: first process model with 5 cases.

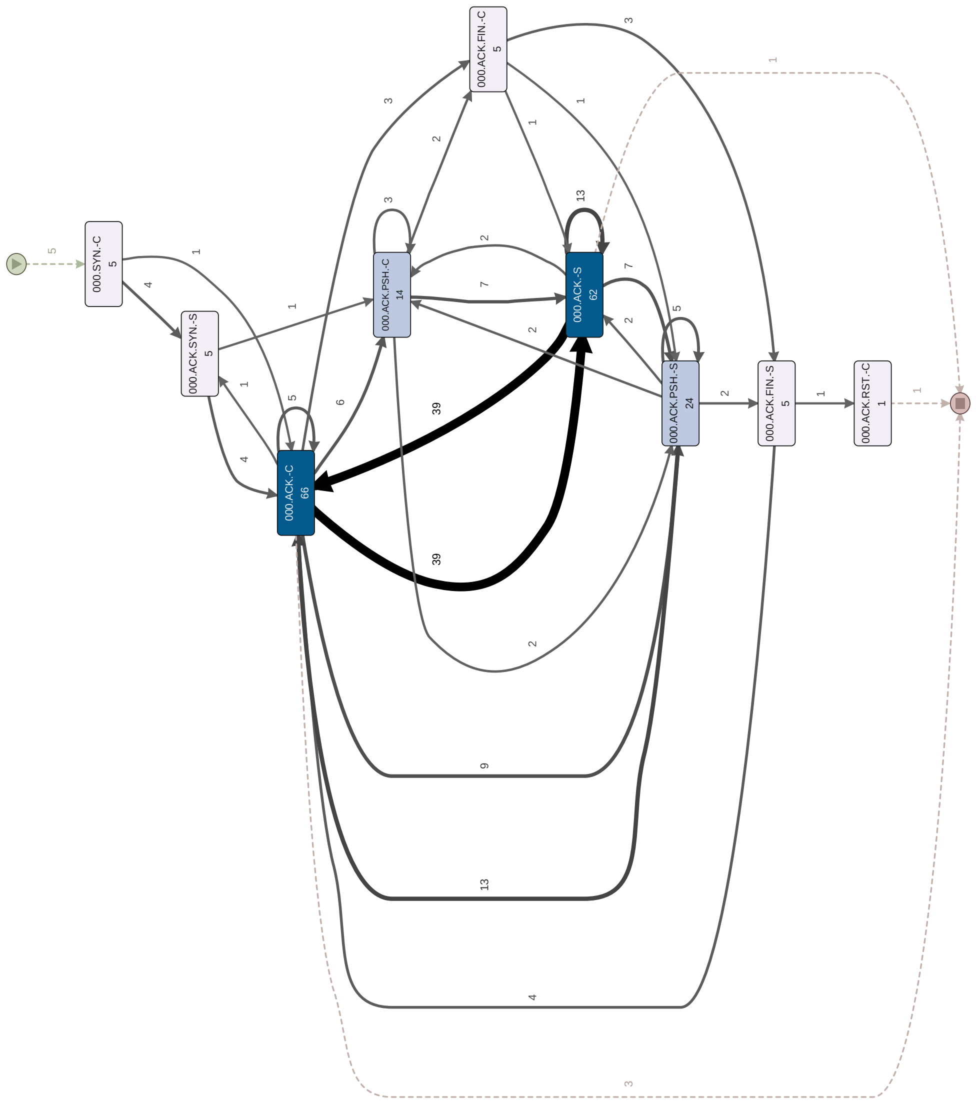


FIGURE A.13: second process model with 5 cases.

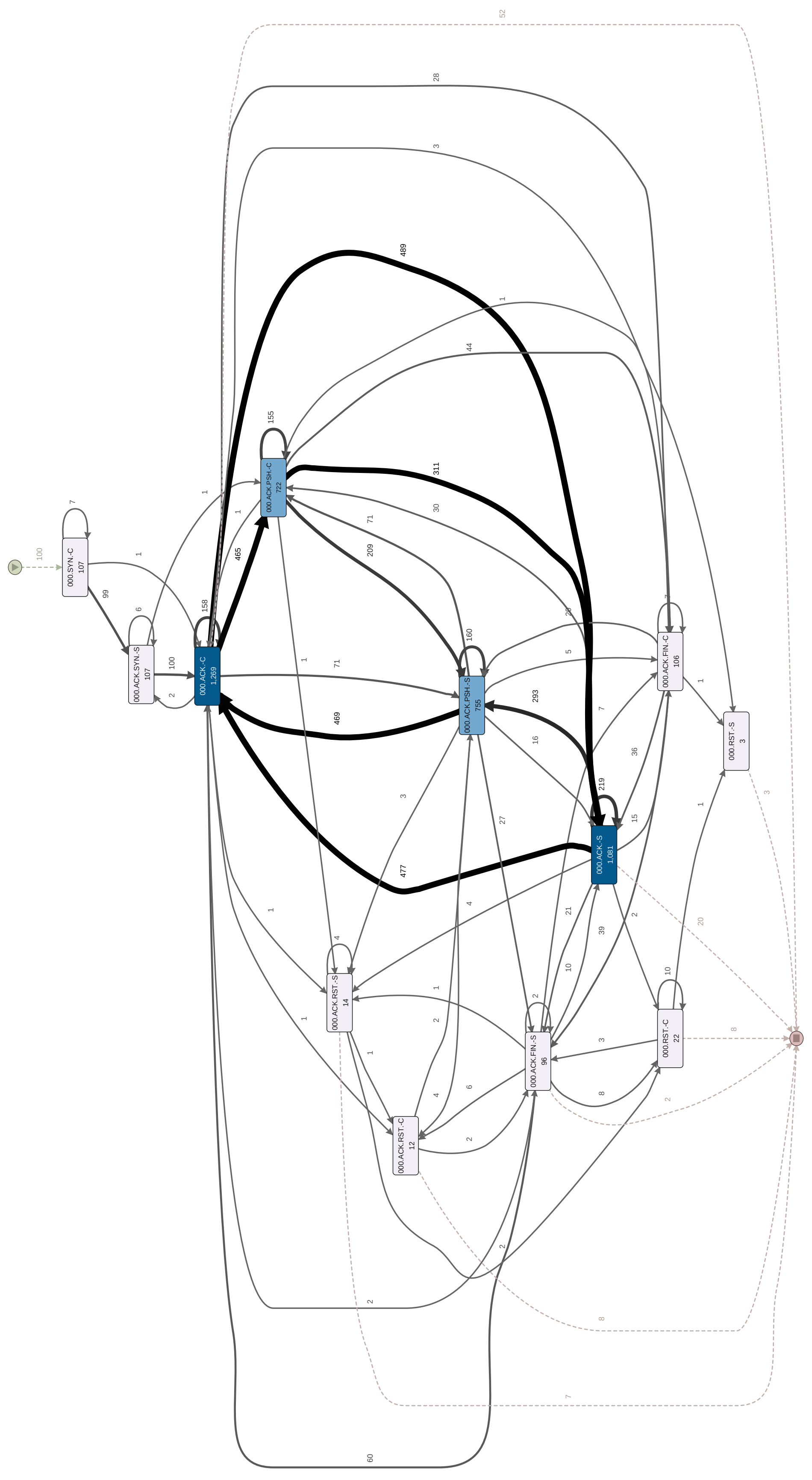


FIGURE A.14: first process model with 100 cases.

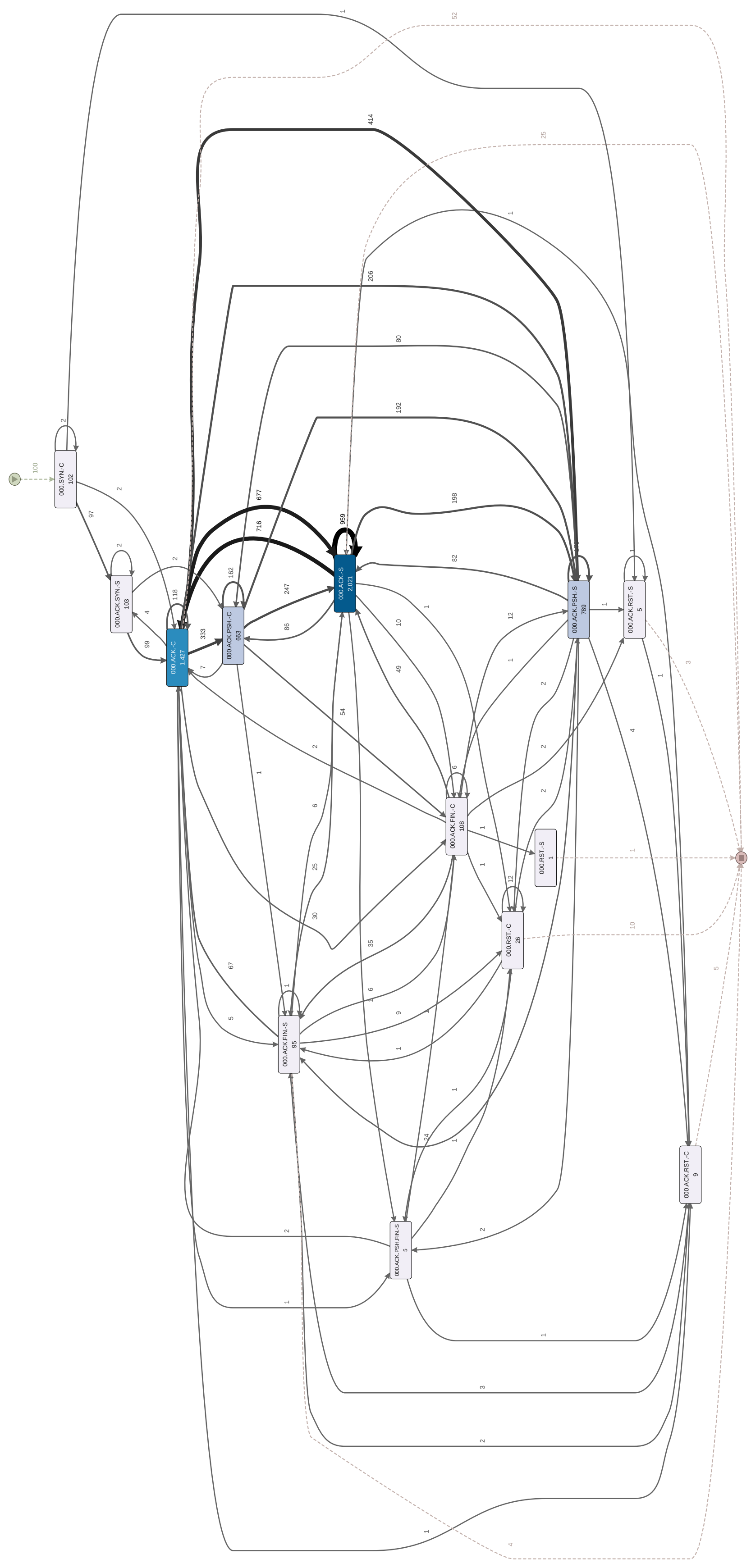


FIGURE A.15: second process model with 100 cases.

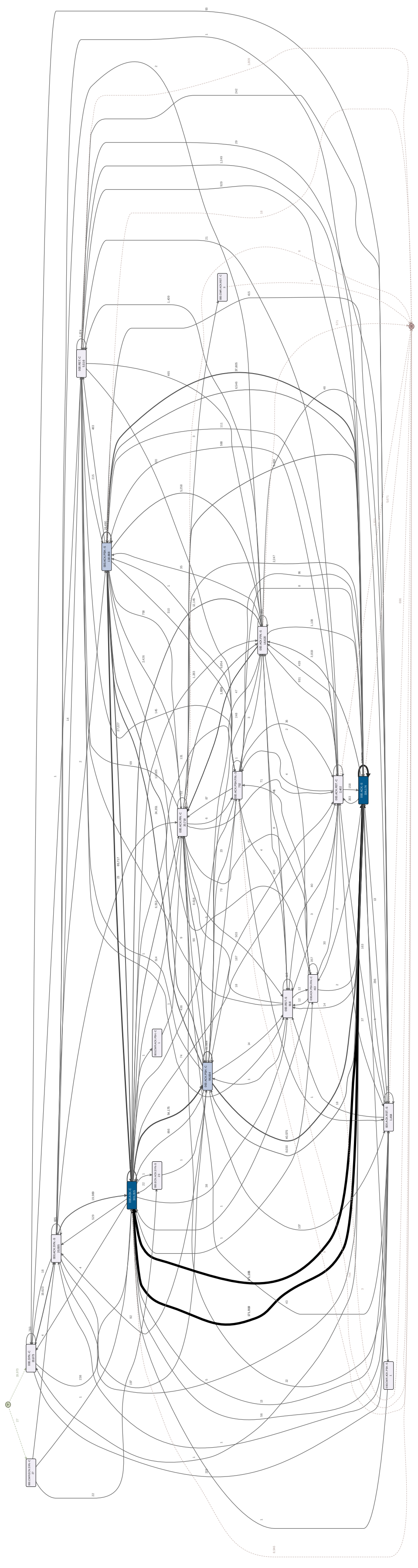


FIGURE A.16: first process model with 20k cases.

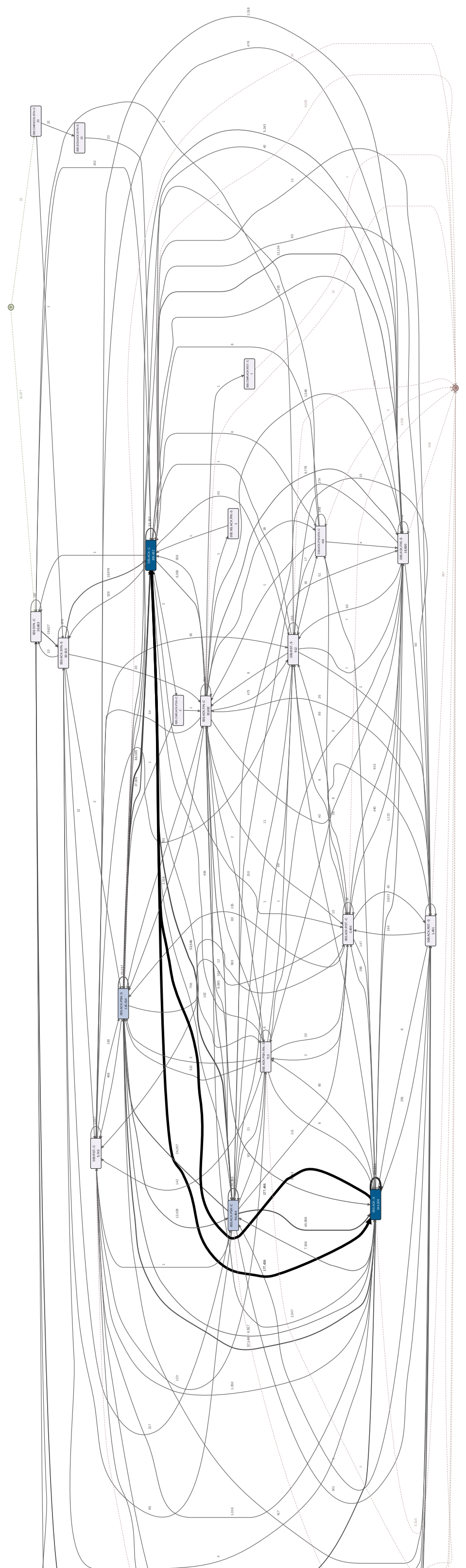


FIGURE A.17: second process model with 20k cases.

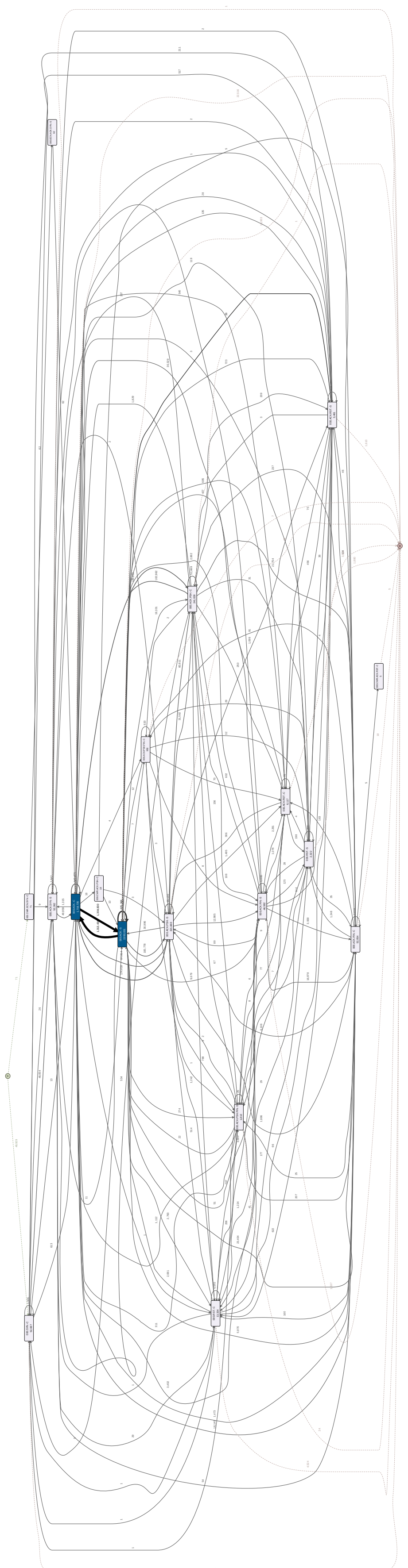


FIGURE A.18: first process model with 50k cases.

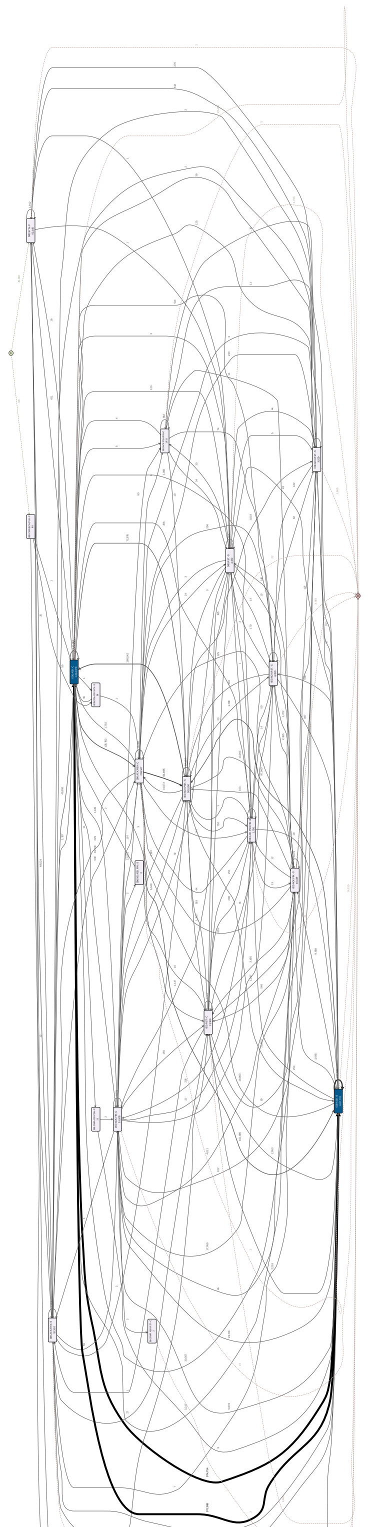


FIGURE A.19: second process model with 50k cases.

A.3 State Diagrams

```
                              +---------+ ---------\      active OPEN
                              |  CLOSED |            \    -----------
                              +---------+<---------\   \   create TCB
                                |     ^              \   \  snd SYN
                   passive OPEN |     |   CLOSE        \   \
                   ------------ |     | ----------       \   \
                    create TCB  |     | delete TCB         \   \
                                V     |                      \   \
                              +---------+            CLOSE    |    \
                              |  LISTEN |          ---------- |     |
                              +---------+          delete TCB |     |
                   rcv SYN      |     |     SEND              |     |
                  -----------   |     |    -------            |     V
 +---------+      snd SYN,ACK  /       \   snd SYN          +---------+
 |         |<-----------------           ------------------>|         |
 |   SYN   |                    rcv SYN                     |   SYN   |
 |   RCVD  |<-----------------------------------------------|   SENT  |
 |         |                    snd ACK                     |         |
 |         |------------------           -------------------|         |
 +---------+   rcv ACK of SYN  \       /  rcv SYN,ACK       +---------+
   |           --------------   |     |   -----------
   |                  x         |     |     snd ACK
   |                            V     V
   |  CLOSE                   +---------+
   | -------                  |  ESTAB  |
   | snd FIN                  +---------+
   |                   CLOSE    |     |    rcv FIN
   V                  -------   |     |    -------
 +---------+          snd FIN  /       \   snd ACK          +---------+
 |  FIN    |<-----------------           ------------------>|  CLOSE  |
 | WAIT-1  |------------------                              |   WAIT  |
 +---------+          rcv FIN  \                            +---------+
   | rcv ACK of FIN   -------   |                            CLOSE  |
   | --------------   snd ACK   |                           ------- |
   V        x                   V                           snd FIN V
 +---------+                  +---------+                   +---------+
 |FINWAIT-2|                  | CLOSING |                   | LAST-ACK|
 +---------+                  +---------+                   +---------+
   |                rcv ACK of FIN |                 rcv ACK of FIN |
   |  rcv FIN       -------------- |    Timeout=2MSL -------------- |
   |  -------              x       V    ------------        x       V
    \ snd ACK                 +---------+delete TCB         +---------+
     ------------------------>|TIME WAIT|------------------>| CLOSED  |
                              +---------+                   +---------+
```

FIGURE A.20: RFC TCP state transition diagram [51].

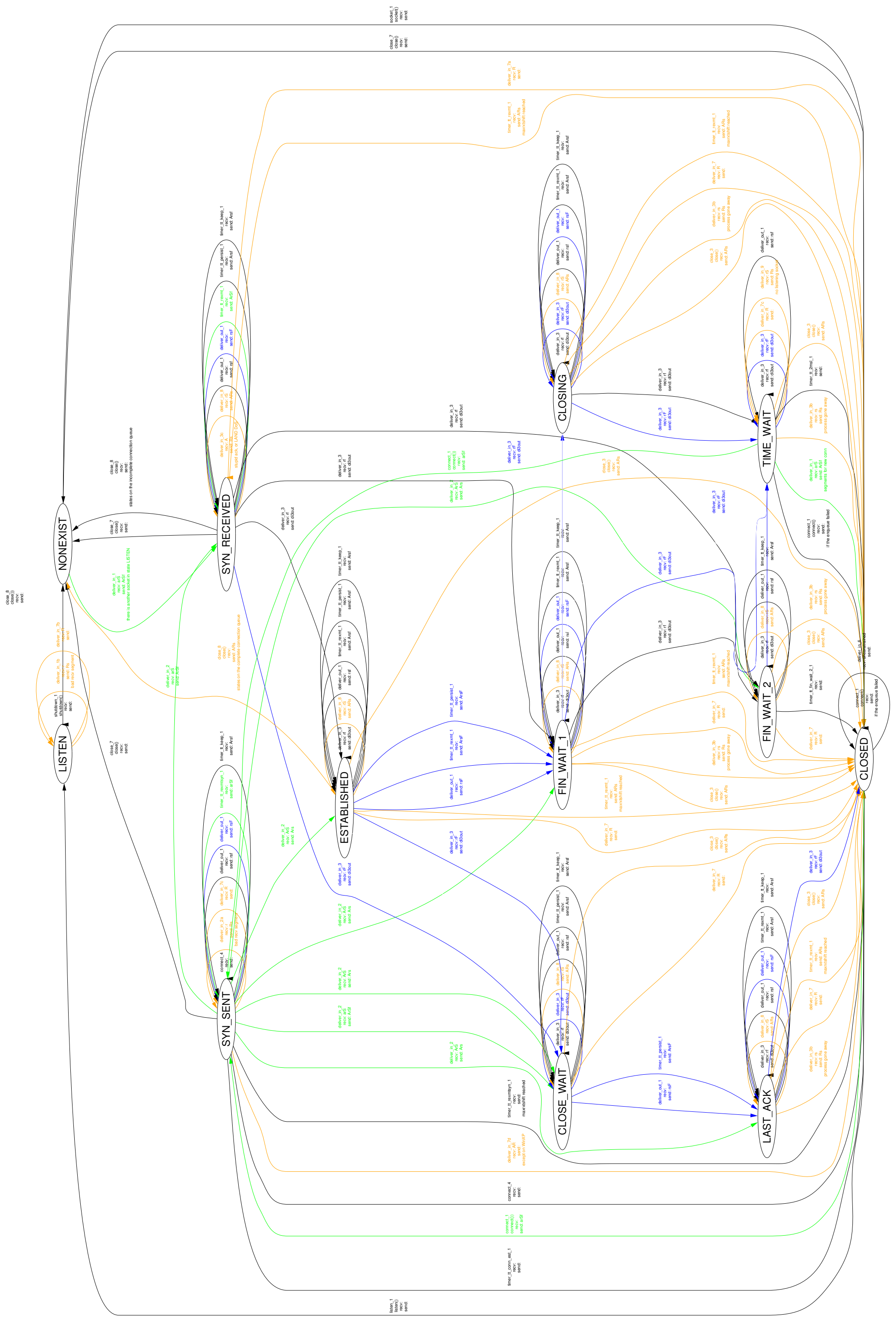


FIGURE A.21: state diagram by Bishop et al. [71].