Learning State Machines from data streams and an application in network-based threat detection

Hans Schouten

Master Thesis

TU Delft

Learning State Machines from data streams and an application in network-based threat detection

by

Hans Schouten

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on Thursday December 6, 2018 at 14:30.

Student number:	4314891	
Project duration:	December 1, 2017 – December 6, 2018	
Thesis committee:	Dr. Ir. M.T.J. Spaan ,	TU Delft, Associate professor, Algorithmics Group
	Dr. Ir. Sicco Verwer,	TU Delft, Assistant professor, Cyber Security Group
	Mevr. N. Lokhorst,	KPN, Team leader, CISO Labs

An electronic version of this thesis is available at
`http://repository.tudelft.nl/`

TU Delft Delft University of Technology

Preface

This master thesis proposes an algorithm for learning State Machines on real-time data streams. The designed algorithm is applied to learn fingerprints of malicious network traffic, after which these fingerprints are evaluated in a detection method aimed to detect the presence of the malicious behaviour. This thesis research is performed at the Cyber Security Research Group of the Delft University of Technology, in collaboration with the CISO department of KPN.

I want to thank my supervisor Sicco Verwer, for his supervision and advice throughout the thesis project. I learned a lot from the knowledge you shared, especially on the topic of State Machines. I want to thank Nathalie Lokhorst and the other members of the KPN CISO team, for the help, insights and brainstorm sessions. I also want to thank my family and friends for all the kind support I received.

"I wish my work contributes to making the world a safer place"

Hans Schouten
Delft
2018-11-30

Abstract

Our increasingly interconnected society poses large risks in terms of cyber security. With network traffic volumes increasing and systems becoming more connected, maintaining visibility on IT networks is a challenging yet important task. In recent years the number of cyber threats have increased dramatically. Monitoring and threat detection are more essential than ever to stay in control in a growing threat landscape. The powerful properties of state machines and the similarities between network traffic and traces used to learn state machines makes this a promising approach. Current learning methods; however, maintain an intermediate data structure that is converted in a state machine after all data has been processed. The continuous nature of network traffic makes this conventional approach inapplicable. This study provides a solution by developing a method for learning State Machines on real-time data streams. The proposed algorithm, framework and implementation are generic and can be applied to any use case that benefits from learning state machines on data streams. This thesis explores one specific use case, which is the use of state machine fingerprints in network-based threat detection. A system is designed capable of learning state machines on real-time traffic channels. The proposed detection method is demonstrated to be highly effective in matching traffic from various malware types to pre-learned fingerprints. The work in this thesis forms a stepping stone to the development of a robust detection method, capable of detecting a variety of threats on network data with low false alarm rates.

Contents

1 Introduction

With the rise of the Internet of Things and an increasing amount and higher quality multimedia consumption, the volume of network traffic that is being generated is growing rapidly [65] [40]. An increase in cloud traffic world wide and rising ICT Development Indices for both well developed and developing countries contribute to this expansion [88]. The adoption of fiber-optic communication and 4G broadband cellular network technology allows data to be transferred at unprecedented speeds, which have never been achieved before. In the near future, the fifth generation of mobile technology and increasingly interconnected smart cities will demand even more networking capabilities [2].

These new technological advancements and an increasingly interconnected society poses large risks in terms of security. The number of cyber threats have increased dramatically in the last few years and is expected to continue its growth in the years to come. The Symantec Internet Security Threat Report shows that in 2017 attacks on Internet of Things devices have increased with 600%, the number of malware variants have doubled and the overall increase in reported vulnerabilities is 13% [83]. The Symantec report also indicates a growing number of more advanced adversaries performing targeted attacks on organizations, primarily for intelligence gathering. Another worrying topic is the growing interest in attacks on Industrial Control Systems and other critical infrastructure which form a vital part of our society.

One of the key processes for uncovering cyber risks is active monitoring and analysis of information security intelligence. The Global State of Information Security Survey of 2018 [67] shows that currently 48% of the organizations worldwide have adopted monitoring and analysis of their infrastructure. With an expanding and more complex cyber threat landscape, maintaining visibility on IT networks is essential and hence this percentage should grow rapidly in the coming years.

However, because of the increasing traffic volume, monitoring traffic for detecting malicious behaviour is becoming increasingly difficult. Especially at large corporations and Telecom providers, detecting malicious behaviour efficiently, effectively with low latency is a tough task. High sampling rates or data abstractions decrease the amount of useful patterns that can be found [55] [11]. On the other hand, processing unsampled datasets requires more and more powerful processing methods and advanced network equipment to cope with the higher data rates. The global growth in internet traffic, makes finding a suitable approach for network monitoring an even more challenging problem.

An organization facing these challenges is KPN, one of the largest Telecom providers in the Netherlands. In order to have good visibility on current threats on their infrastructure, the most powerful unsampled analysis approach is desired. However, the high data rates would put high demands on network equipment and would fill up disk space so rapidly that exploration of historic data for analysing past incidents will be very limited due to the resulting low retention time. This means that incidents which remain undetected for a period of time can no longer be investigated. A trade-off is made between sampling and the desired amount of retention time, while operating on the limits of hardware and software processing capabilities.

As a result of the processing limitations of the hardware and software when analysing unsampled datasets, organisations often opt for a solution in which high sampling rates are applied, accepting the consequence that less obvious anomalies will remain undetected. In recent years companies like Cisco, Plixer, Gigamon and FlowMon [66]; however, have introduced new hardware appliances announcing a next generation of network monitoring, which is the processing and analysis of unsampled network data. The ability to process unsampled data on a high scale at very high throughputs, introduces a great research opportunity since it allows for analysis methods on the exact se-

quences of network data, instead of randomly sampled data points. One promising technology that could be applied on these unsampled network streams in order to improve network based threat detection is State Machines.

1.1. State Machines

In recent years, at the Delft University of Technology, extensive research has been performed by the Intelligent Systems department on learning State Machines on various types of data in order to model the behaviour of complex systems. State Machines are mathematical models of computation, which can be used to describe systems or protocols. State Machines are learned on data sequences and hence have a similar learning input as what can be established with unsampled network traffic. This makes the powerful State Machine models an interesting candidate for network based threat detection by learning communication models on network data sequences to detect malicious behaviour. Figure 1.1 shows an example of such a State Machine model learned on traffic generated by the advanced banking Trojan Emotet. The graph shows specific combinations of traffic sequences that occurred in the malicious traffic.

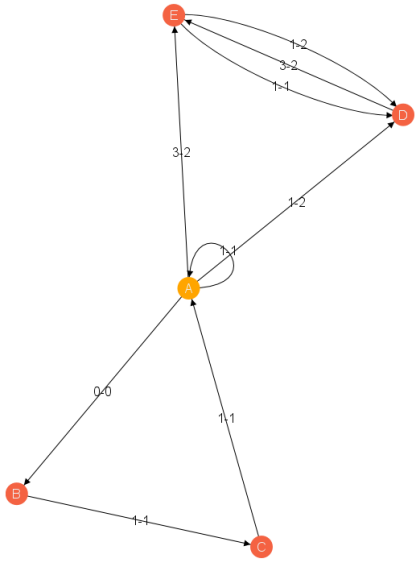


Figure 1.1: A State Machine learned on network data generated by the advanced Emotet banking Trojan

1.1.1. Use cases

State Machines have been applied to a broad range of use cases. In as early as 1978 State Machines were proposed as a method for software testing, offering a reliable approach for verifying that a piece of software meets its specification [17]. Also in high level system design State Machines have proven to be useful models, providing software architects with abstractions easing the design and analysis of complex hardware and software systems [9]. State Machines have been used to program wireless sensors, by defining all possible sensor states and corresponding actions to be taken in the form of a State Machine [45]. Another use case is described in a paper of 2006 [71], in which it is proposed the use of State Machines for intrusion detection on VoIP calls. In 2008 a group of researchers explained how State Machines could be used to create models of system logs to review a system's execution, control flow, data-flow and related statistics [84]. An entirely different use case is described in [43] by defining a method for constructing State Machines for recognizing human gestures by computer interfaces. Recent work at the Delft University of Technology and the Tongji University of Shanghai [102] shows how State Machine models could be applied to autonomous driving for learning car-following behaviour. This is just a glimpse of the possible applications that have been made with State Machines.

1.1.2. Powerful properties

From the large amount of research that is performed on State Machines and the variety of use cases it has offered solutions for, we can conclude that State Machines are very powerful models. By taking a closer look at the applications of State Machines, the following properties can be derived that enable State Machines to be applied to such a variety of use cases:

1. State Machines can form an exact model that can fully describe a system or protocol, to such an extend that we can conclude that all behaviour not captured in the model is a sign of unwanted, malicious or illegal be-

haviour. It can be used to check with certainty whether arbitrary sequences of events are allowed by the system or protocol that is modelled.

2. The State Machine's graph representation yields models that can be easily understood by humans. In contrast to for example deep neural networks, decisions based on State Machines can be easily explained by tracing routes through the graph.

1.1.3. Advantages of State Machines in network threat detection

In this thesis a link is made between the concept of State Machines and network based threat detection. The decision to apply State Machines to this use case can be motivated by considering that the properties mentioned in section 1.1.2 are advantageous for systems performing threat detection.

In network based threat detection there various types of malware that one would be able to detect. Each piece of malware executed on a victim's machine operates according to the implementation written by the malware developers, containing distinctive sequences of operations needed to complete the tasks it was created for. With the different objectives of various types of malware and benign applications comes differences in the behaviours of each piece of software. This is reflected in variations of the communication methods, amount of communication and targets that are communicated with, as each piece of software achieves different goals. Due to these variations in network communication and the powerful capabilities of State Machines in capturing and describing system behaviour, the use of State Machines might be a suitable approach worthwhile to further explore. Similar to how State Machines can learn powerful models on software call traces, it could possibly create equally powerful models on network traces.

A challenge in network based threat detection is that detection methods generate false positives. In order for threat detection to be reliable and effective, it is important to be able to evaluate why certain behaviour is flagged as malicious. With for example Neural Network based approaches [68], it is hard to verify the correctness of alerts as the underlying models are not easily interpretable by humans. This holds for a large number of machine learning techniques in which classifiers are learned on a range of traffic parameters. By learning State Machine models, on the other hand, one can visually compare models of different types of communication. This gives instant insights in how similar behaviour is. This is especially useful on verifying the correctness of detections or when a detection needs to be further examined.

1.1.4. State of the Art learning method

The powerful potentials of State Machines comes at a price: the problem of learning a DFA is very complex. Pitt and Warmuth proved in 1989 that it is NP-hard to find a DFA that is consistent with a given set of training strings and whose size is within a polynomial factor of the size of the smallest such DFA [64]. Kearns and Valiant proved in 1989 that predicting the output of a DFA can be as hard as breaking cryptosystems that are widely believed to be secure [46]. As a result, current methods used to learn State Machines are very computationally expensive and often require large amounts of memory resources.

The most common and accurate approaches of learning State Machines rely on Evidence-Driven State Merging (EDSM). It is a powerful technique that was at the core of the winning algorithms used in the Abbadingo competition [50] held in 1997. Although any approach was allowed in this competition, the two best performing algorithms relied on an EDSM technique. The competition gave rise to the Blue-Fringe algorithm. This algorithm has since then been extensively applied as State Machine learning method in among others 2005 [22], 2007 [97] [89] and 2008 [25] [49] [96].

Later in 2012 it formed the baseline algorithm of the StaMinA competition [98], which was in contrast to the Abbadingo competition targeting a specific use case, namely software model inference techniques. The competition was won with the DFASAT algorithm proposed by Sicco Verwer and Marijn Heule, as described in a paper published in 2013 [41]. At the core of the DFASAT algorithm still lies the greedy Blue-Fringe merging algorithm proposed during the Abbadingo competition. Also in recent years the Blue-Fringe strategy has been reused in various research, for example in 2016 [53] and in 2017 [62] [90].

Hence, the approach used in the Blue-Fringe strategy will be regarded as state of the art throughout this thesis.

1.2. Problem Statement

A large number of systems around the world are generating continuous streams of data. Examples are networking equipment, financial trading systems, applications that provide logging functionality, user activity on websites and the devices providing sensing capabilities like for instance weather sensors, medical sensors and sensors inside industrial equipment. State Machines have proven to be powerful in capturing characteristics of data sequences and condensing them into useful models. However, the continuous nature of a lot of today's data sources do not fit well with the way State Machine are used to be learned. Current state of the art State Machine learning methods require to extract a data dump, pre process the data and only then a State Machine learning method is applied. This limits the application of State Machines to use cases with data sets of pre-defined sizes. Furthermore, data streams are often non-stationary, which means behaviour changes over time [100]. In non-stationary environments, models can quickly become obsolete, hence in such environments obtaining a model as soon as possible is essential.

Low latencies are also desired in use cases where the learned models can trigger warnings that quickly should be acted up, for instance in network based threat detection. Current state of the art methods learn State Machines non-incrementally, which is a discouraged practise since it results in inefficient solutions due to being search intensive [69]. As a result of the above observations, it can be concluded that the current state of art approaches to learning State Machines suffer from the following drawbacks:

1. **In high throughput environments it requires large amounts of memory**
   In existing solutions, first all data should be stored before a State Machine learning method can be applied. In high throughput environments, storing all data before learning the State Machines will require extensive amounts of disc space.
2. **Obtaining low latencies is not possible**
   Current learning algorithms rely on all training data to be collected in advance before learning process starts. This means there will be a delay before the State Machine is produced. This delay is at least the data capture duration plus the State Machine learning period.
3. **Large demands on processing power at once**
   Since existing solutions starts the learning process after the data collection, instead of distributing the learning process over the entire period data has been captured, this puts high loads on the available processing power.
4. **The State Machines cannot be used during the learning process**
   Current state of the art State Machines learning methods do not learn State Machines incrementally. As a result, during the learning process an intermediate result of the State Machine can not already be put to use.
5. **Determining the right amount of training data in advance can be hard**
   The amount of training data used for learning State Machines using current methods is pre-determined, namely at the moment data is collected from a data source. This means that one should determine in advance how much data to use for training. This could result in the scenario of having selected an insufficient amount of training data.

The above mentioned issues impede the powerful State Machine models from being applied to high throughput data streams or complex non-stationary environments. Furthermore, use cases in which low latency is required or when State Machines should be learned iteratively over long periods of time are infeasible with current techniques.

The problem causing these challenges is the lack of an algorithm for learning State Machines on data streams in an incremental fashion. In order for researchers to effectively explore the possibility of applying State Machine models to additional novel use cases, one of which is network based threat detection, these challenges should be solved.

1.3. Proposed solution

In order to solve the above mentioned issues and to fit State Machine learning better in data streaming environments, a streaming variant of State Machine learning should be designed. The new method for learning State Machines proposed in this thesis solves each of the mentioned issues by having the following characteristics:

1. **In high throughput environments, streaming applications have no additional disc space requirements**
   In an environment with high throughput data streams an incremental learning approach does not require to store data elements to disc, but instead continuously learns a State Machine.
2. **Stream processing offers low latencies**
   In a stream processing application, elements of the stream are processed directly after they are consumed. The elements are instantly impacting the model, allowing for low latencies to be achieved.
3. **Computational resources are spread over the data collection period**
   In a streaming application, computations are performed during the consumption of stream elements instead of at the moment all elements or a batch of elements has been collected.
4. **State Machines can be used during the learning process**
   In an incremental learning approach on data streams, the intermediate results can be outputted for external use or being used in the streaming algorithm.
5. **Training set size can be determined while processing**
   In a stream processing application the amount of data used for learning a model can be determined on rules or heuristics. After the learned model has reached a certain level of maturity, the learning process could terminate. With learning algorithms on pre-determined batch sizes relevant data could be divided over two batches, whereas a streaming algorithm could learn on all relevant data.

The proposed learning method is applicable to network-based threat detection for constructing powerful models that can be used for real-time detection of malicious behaviour.

1.4. Research Objective

The objective of this thesis is to develop a method for learning State Machines on data streams with real-time performance. The proposed algorithm, framework and implementation should be generic allowing them to be applied to any use case benefiting from learning State Machines on data streams. In assessing the proposed method, one novel use case will be explored in this thesis. This use case is the application of State Machine fingerprints in network based threat detection for detecting malicious behaviour generated by malware.

1.4.1. Research Questions

The above mentioned research objective can be captured in a single research question, which will be answered throughout this thesis:

How can we develop an algorithm for learning State Machines on streaming data and apply this in a system capable of detecting the presence of malicious behaviour in real-time traffic channels?

Decomposition

In order to provide a structured answer to the research question, the main question is decomposed into four sub questions:

SQ I **How can we transform sequential algorithms into a streaming variant and implement them in Apache Flink?**
As section 2.2 explains in more detail, this thesis uses Apache Flink as data processing platform for implementing streaming algorithms. In section 2.1 a background into stream processing and commonly used techniques is provided. Later, in chapter 3, a number of popular algorithms are transformed into a streaming variant using the earlier described streaming techniques. This chapter explores the Apache Flink platform and finds the best approach of implementing streaming algorithms, answering sub question 1 and providing pre-work to answer sub question 2 and eventually the main research question.

SQ II **How can we learn State Machines on data streams?**
Chapter 4 describes the detailed design process of creating an algorithm that learns State Machines on data streams. The designed algorithm, its implementation and evaluation of correctness answers the second sub question.

SQ III **How can we detect the presence of a learned State Machine fingerprint in a traffic channel?**
Chapter 5 designs an method to apply the State Machine learning algorithm to learn fingerprints on labelled network data. The learned fingerprints are then used to detect the presence of the underlying behaviour on unlabelled data streams of network traffic.

SQ IV **How scalable and effective is the proposed method of using State Machine fingerprints for network based threat detection?**
In chapter 6 the performance of the developed system for network based threat detection is evaluated, both in terms of computation time and effectiveness. It evaluates whether malicious behaviour can be detected with real-time performance, which partly answers the main research question.

1.5. Methodology

In this thesis an algorithm, platform and implementation will be designed for learning State Machines on data streams. Hence, the appropriate research methodology is the methodology of design science as explained by Hevner et al. in Design Science in IS Research [60]. The work by Hevner et al. defines 7 guidelines that are recommended in performing Design-Science research.

1.5.1. Artefact

The result of design science in Information Systems research is a purposeful IT artefact in the form of a construct, a model, a method, or an instantiation. In Chapter 4, a method and implementation is carefully designed for learning State Machines on data streams. In chapter 5 a method is designed for using State Machines as models for detecting malicious behaviour on NetFlow streams.

1.5.2. Relevance

The second guideline dictates that the design of the artefact should be the response to an important and relevant problem. The problem initiating this thesis research is well-documented in Section 1.2.

1.5.3. Evaluation

The designed artefact should be thoroughly evaluated. This is described in Chapter 4 by validating the correctness of learned State Machines and the evaluation is described in Chapter 6 by assessing the system's computational performance and the effectiveness of the threat detection method.

1.5.4. Contributions

Effective design-science research must provide clear and verifiable contributions. Contributions made by this thesis research are stated in Section 1.6 and for verifiability the implementation details are provided in Chapter 4.

1.5.5. Research Quality

The last three guidelines are regarding research quality. These guidelines recommend the use of rigorous methods in the construction and evaluation of the designed artefact, an iterative design process and a research that is well adapted to the target audience. These guidelines are satisfied throughout this thesis research. Section 1.1.4 shows the related research highlighting a powerful method and its rigorous results, which is used as a basis of the method that is designed in this thesis. The iterative nature of the design process can be found in chapters 4 and 5. By explaining the required background knowledge in Chapter 2 and by describing the work on a high level as well as in its details, this thesis communicates the research to anyone of its target audience.

1.6. Contributions

This thesis will contribute to the scientific body of knowledge in various ways. The contributions can be grouped in three categories. First of all, this thesis contributes to the further maturation of the research being performed on State Machines, by proposing a method for learning State Machines on data streams. More specifically, this research makes the following contributions to State Machine research:

- **An algorithm for learning State Machines on data streams**
  Current state of the art State Machine learning algorithms learn the models in a resource intensive and delaying non-streaming form. This thesis designs an algorithm for learning State Machines in an incremental approach operating on continuous data stream.
- **A scalable framework for learning and analysing State Machines**
  The combination of tools together with the boilerplate code proposed in this thesis, form a scalable framework that can be used as a basis for creating and analysing State Machine learning methods as well as using these methods in practise and as part of further research.

Secondly, this thesis research contributes to the field of Cyber Security by exploring a novel method of learning fingerprints of malicious behaviour in network traffic. These contributions consist of:

- **A pipeline for high throughput NetFlow processing**
  There is an abundance of papers that claim to have used a setup that could process a specific amount of NetFlows data per second. However, most often it lacks details about or references to the actual code used. The point of scientific research is to have verifiable studies, so this thesis proposes a thoroughly documented stream processing pipeline which can be reused to verify results as well as a basis for other high throughput NetFlow processing applications and research.
- **A novel method for creating malware fingerprints**
  In this thesis a novel method is designed and evaluated for creating fingerprints of malicious network traffic using State Machines. The proposed fingerprints are interpretable by humans, which makes validation straightforward.
- **A novel method for detecting the presence of malicious behaviour in real-time traffic channels**
  This thesis presents a method for learning fingerprints on each traffic channel of a live network. It applies this capability in a novel way to determine whether fingerprints of malicious behaviour are present in each of the observed traffic channels.

And third, a more general additional contribution is made:

- **Streaming implementations**
  This thesis transforms several well-known sequential algorithms in a parallelized streaming form. For the selected algorithms very limited details on streaming implementations is present. This work presents streaming versions of: Heavy Hitters, Frequent Patterns, Frequent Sequences and Markov Chains.

1.7. Thesis Outline

To provide an answer to the research question, each of the sub questions should be answered. Before these questions can be addressed, first a body of background information will be provided in chapter 2. The second chapter explains the concepts and technologies necessary to understand the thesis. The first sub question is answered in chapter 3 by describing a number of sequential algorithms which are transformed into a streaming variant with an implementation in Apache Flink. The second sub question is answered in chapter 4 by describing the design process of an algorithm for learning State Machines on data streams. The third sub question is answered in chapter 5 by designing a method for learning State Machine fingerprints on malicious traffic and a method for detecting the presence of

these fingerprints in traffic channels. Chapter 6 provides an answer to the final sub question by analysing the performance of the proposed State Machine learning algorithm and implementation and by evaluating the effectiveness of the detection mechanism proposed in chapter 5. In chapter 7 the obtained results, designed artefacts and their limitations are discussed. The discussion is followed by future work in Chapter 8. The future work chapter mentions how additional research could extend the work put forward by this thesis. Finally, chapter 9 concludes the thesis by providing answers to the research questions.

2
Background

This chapter provides the background information required to understand the methods and technologies applied in this thesis. Since this thesis is about learning from data streams, first a high level overview of stream processing and the most popular stream processing methods is provided in section 2.1. Next, section 2.2 gives a background into the most common high scale data processing frameworks and the framework of choice in this thesis: Apache Flink. A popular memory efficient approach for obtaining summaries of stream elements is the use of sketches. This technology is explained in section 2.3. In section 2.4 NetFlow technology is explained, which is the data format used to learn network traffic models and to detect malicious behaviour on. The introduction already briefly touches upon State Machines. Section 2.5 contains the formal definition of State Machines which forms the basis of the models used in this thesis. Lastly, for validating the correctness of the proposed State Machine learning method and for detecting the presence of malicious behaviour in traffic channels, it is necessary to compare State Machine models. Section 2.6 introduces the state of the art methods of computing the similarity between State Machines.

2.1. Stream processing

Stream processing is a programming paradigm in which computations are performed on data directly as it is received by the system. Upon consuming an element of the stream, the system usually passes the element down a pipeline of operations that are performed on this single element. These operations can range from updating state, triggering an action, updating an aggregation or other statistics. The results of stream processing applications are for example an automatically controlling process, learning models or computing detailed statistics on the consumed data.

Counterparts of stream processing are batch processing and sequential processing. Sequential processing is the most common paradigm, in which a series of procedures are sequentially executed on a dataset. This processing method is essentially a for loop iterating over all data elements. Batch processing, as the name suggests, operates on batches of data that already have been stored over a period of time. On each batch a sequential processing approach can be applied. Usually batch processing applications processes large amounts of data with the aim to answer a specific question, as is often the case in big-data analytics [59]. Data is divided into batches due to limited disk and RAM resources on the machines performing the computations. Another common batch processing application is the execution of routinely scheduled tasks, for instance aggregating data into a useful statistic for a monthly report.

The advantage of stream processing compared to its two counterparts is that it can react to events instantly, without an additional delay of storing data at disk [74]. It can process large data volumes with less resources since it requires very limited storage capacity. Incrementally updating models as new streaming data comes in, fits better with the nature of timely data than periodically recomputing an entire dataset. This thesis focusses on the design of a method for detecting malicious behaviour on continuous flows of network data. In this use case, access to a solution as soon as possible is essential. Furthermore, a memory-efficient approach is desired to obtain a more scalable solution. Considering these requirements, stream processing is the paradigm of choice.

2.1.1. Streaming algorithms and techniques

For scalable processing on high throughput data streams a number of popular algorithms are described in literature. This section describes the most well-known algorithms and techniques, which form the basis of the methods used in this thesis.

Frequent Elements

One of the first streaming algorithms is the Boyer–Moore Majority Vote algorithm. This algorithm aims to find the element representing more than half of the dataset, with only very limited memory resources [58]. To get the majority element from a list, we normally would maintain a counter for each unique element. Eventually, if the element with

the highest count has a higher value than all elements combined divided by two, we have the majority element. Now suppose the number of possible elements is infinite. Maintaining a counter for each element is infeasible. The Majority Vote algorithm solves this challenge while maintaining only one element and one counter. Algorithm 1 shows the elegant solution.

Algorithm 1 Boyer–Moore Majority Vote

```
1:  m ← ∅
2:  i ← 0
3:
4:  for all element x do
5:      if i = 0 then
6:          m ← x
7:          i ← 1
8:      else if m = x then
9:          i ← i + 1
10:     else
11:         i ← i − 1
12:     end if
13: end for
14:
15: return m
```

The reason that majority element will always be returned is that it occurs more than 50% of the stream. This is often enough to decrease each counting attempt to zero. The Majority Vote is a special case of the Frequent Elements problem, which outputs a set of elements that occurs more frequently than a predefined fraction of the stream. The Frequent Elements problem is perhaps one of the most thoroughly studied streaming problems, with its first algorithm dating back to 1982 by J. Misra and D. Gries [57]. Although the Misra-Gries algorithm was one of the first solutions proposed for this problem, it is still a very popular approach. The algorithm is very similar to the Majority Vote approach, albeit with a set of elements and counters instead of just one.

Frequency Estimation

Recent stream processing problems are often similar to the Frequent Elements problem or rely on ideas and techniques developed in the process of solving this problem. One such related problem is Frequency Estimation, which does not try to find the k most frequent elements, but instead estimates the occurrence frequency of all unique items in the data stream. A very popular solution to this problem approximates the frequencies of elements using a Count-min Sketch [20]. Count-min sketches are a memory efficient data structure for storing frequencies and are still being studied and applied in very recent research [30]. The Count-min sketch is therefore a data structure that is applied for estimating frequencies in this thesis and will be explained in more detail in Section 2.3.

Flajolet-Martin Algorithm

Another well-known streaming problem is approximating the number of unique elements in a stream of unknown size. The researchers Flajolet and Martin proposed a solution [27] which hashes elements to uniformly distributed values. By storing the largest position of the least-significant bit of the previously hashed elements, the number of unique hashes can be estimated using the probability of an element occurring with the stored least-significant bit position. This solution has later become known as the Flajolet-Martin Algorithm or the Flajolet-Martin Sketch [3].

Bloom filters

A Bloom filter is yet another very important technique used in a large number of streaming applications and in other use cases where minimizing the memory footprint is essential [8]. This technique offers a solution to the problem of efficiently verifying that the current item of a stream has never been seen before. To become aware of the challenge in this problem, think for a minute about how you would confirm after seeing thousand email addresses whether the next email address did appear before. The naive solution again is to store each unique element. Now imagine that instead of thousand, the number of unique email addresses is one million. This would put high demands on memory resources. Bloom filters are a solution that can keep track of the seen items with only a small memory footprint. This is obtained by maintaining an array of binary values of length l. Next, k hash functions are defined. Each hash function maps input elements to an index between 0 and l. On storing an element, the positions for all k hashes are set to 1. On checking whether an element was seen before, one need to evaluate whether the value stored at the index of each hash function returns 1.

The reason why multiple hash functions are used, is because an unseen element could accidentally hash to the same index of an already stored element. This would result in a false positive. By using multiple hash functions, the chances of false positives will be reduced, although they are never eliminated. Note that since 1-values are never modified in the Bloom filter, once an element is stored it will always be recognized. Therefore, this data structure will never yield any false negatives. Figure 2.1 illustrates the Bloom filter's structure, the storing process of two elements and testing the occurrence of two elements. The image shows that element y_2 was actually never stored, but maps to the same positions as x_2. This is an example of a false positive.

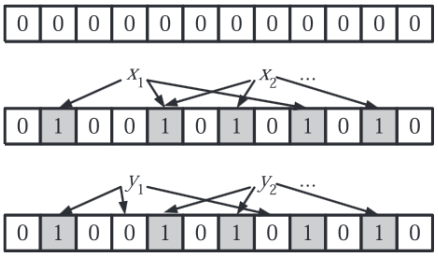


Figure 2.1: An example of a Bloom filter. The filter begins as an array of all zeros. Each item in the set x_i is hashed k times, with each hash yielding a bit location; these bits are set to 1. To check if an element y is in the set, hash it k times and check the corresponding bits. The element y_1 cannot be in the set, since a 0 is found at one of the bits. The element y_2 is either in the set or the filter has yielded a false positive. [12]

Reservoir Sampling

In another use case of data streams the goal is to find a fixed sized random sample of an unbounded stream. This is useful for applying non-streaming algorithms on a fixed sized subset of the streamed data, while preserving the distributions of the original unbounded stream. A popular algorithm solving this problem is reservoir sampling, which maintains a reservoir of elements each of which are replaced with a decreasing chance as more elements of the stream are consumed [94].

2.2. Scalable data processing frameworks

The previous section provides a background into stream processing and some popular techniques that inspire the decisions taken throughout the thesis. This thesis offers a solution for employing State Machines in environments that require large amounts of data to be processed, by proposing a method for learning State Machines on high throughput data streams. However, the algorithm in itself is only partly the solution. In order to test scalability and to apply the algorithm on a large scale in practical use cases, a robust implementation is required. To prevent shifting the focus too much towards creating fault tolerant high performance scalable solutions, while still delivering a useful contribution, it is decided to create implementations using a data processing platform. This section explores some well-known highly scalable data processing platforms in order to select a well-suited platform that can be used to implement, test and apply the algorithm proposed in this thesis.

2.2.1. Hadoop

Apache Hadoop is a very popular data processing framework, used in a large number of big data analytics scenarios [31]. Hadoop offers reliable, scalable, distributed computing based on the MapReduce programming model and handles files according to the Hadoop Distributed File System (HDFS).

Implementations using the MapReduce model consist of a chain of pairs of operations. Each operation pair consists of a map procedure followed by a reduce procedure. In short, the mapping operation prepares the data by redistributing it across the cluster to allow parallel computation. The reduce operation performs the actual computation on the mapped data. A visualisation of both phases is shown in figure 2.2.

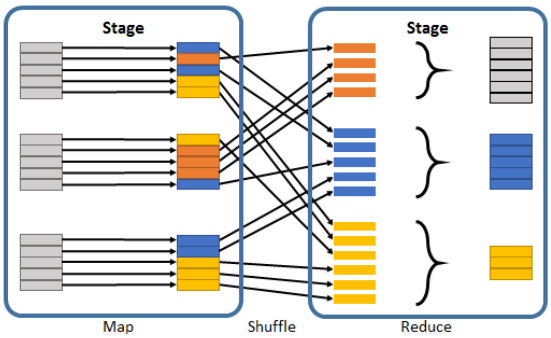


Figure 2.2: The Map and Reduce stages of a Hadoop job [56]

Mapping phase

The mapping procedure allows each node of a cluster to restructure its sequences of data elements into new sequences, for example by filtering or sorting. The mapping procedure outputs a data sequence which consists of key-value pairs. The key is used to redistribute data across different nodes. After a mapping procedure has been executed, the data is redistributed across the cluster by instructing all nodes to transmit tupples with a similar key to the same node. On receiving data, each node groups all incoming tupples into a number of new data sequences based on the key. As a result, after a mapping operation, a node has one or more sequences of data each containing all tupples with a certain key. The goal of a mapping operation is to transform data into key-value pairs that will result in meaningful distributed data groupings on which the next operations can be performed.

Reduce phase

On the resulting redistributed data sequences, a reduce procedure is performed in parallel across all nodes. A reduce procedure transforms a sequence of data elements into a smaller number of data elements. Usually, a reduce operation performs aggregations of data, like a sum operation. Multiple sets of map and reduce operations can be chained together, resulting in a pipeline of basic operations which are executed in parallel and distributed across a cluster of nodes. This yields an exact answer to a question that requires large amounts of data to be processed.

2.2.2. Apache Spark

Apache Spark [32] is another commonly used platform that relies on an approach similar to the MapReduce model of Hadoop. It also uses a series of mapping and reduce phases, in which data is distributed and then updated or aggregated. In contrast to Hadoop, Apache Spark processes data in memory instead of storing it all to disk on receiving data and after an operation has been completed. This makes the execution of Apache Spark much faster, especially in iterative processes like machine learning applications. However, since all data is stored in RAM, Hadoop can be a better candidate in case the volume of data is extremely large and latency is a less important requirement.

In recent versions of Apache Spark, Spark Streaming is introduced [54]. Spark Streaming utilizes Spark's fast in-memory processing capabilities, to read small chunks of data and perform the mapping and reducing procedures on these micro-batches. To summarize, the power of Apache Spark lies in its ability to have fast in-memory execution of iterative algorithms and in supporting near real-time processing using micro-batches.

Figure 2.3: High level overview of a Spark Streaming job with micro-batches [78]

2.2.3. Apache Storm

Apache Storm is a distributed stream processing platform [33]. In contrast to Hadoop and Apache Spark, it does not execute jobs on an entire dataset or on batches of data. Apache Storm, processes infinite streams using *topologies* instead of *jobs*. A *job* is a process that eventually stops when all data has been processed. In a *topology*, connections between nodes are established and each node performs a subtask on the incoming data stream and passes the results on to another node. An Apache Storm *topology* never terminates, but continues to process incoming data elements. The *topology* consists of Spouts, which are data sources, and Bolts, which are nodes responsible for a step of computation. A high level overview of the Apache Storm *topology* can be seen in Figure 2.4.

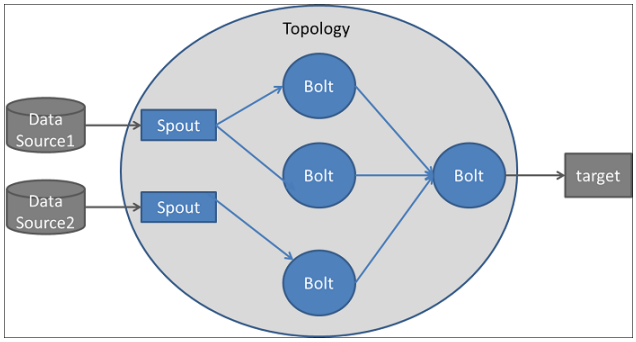


Figure 2.4: Example of an Apache Storm topology with Spouts as data sources and Bolts as steps of computation [80]

2.2.4. Apache Flink

Similar to Apache Storm, Apache Flink is a distributed stream processing platform [34]. In Flink, the data elements are processed directly as they are consumed, without writing elements to disk first. This allows Flink to to process high throughput streams with low latency. On a high level, a Flink job consists of one or more sources, a pipeline of transformations to perform on the data, followed by one or more sinks to export the results. As part of the transformations, Flink provides a Map and Reduce operation which have similar workings to the procedures provided by Apache Spark and Hadoop. However, it also supports numerous other operations, for example: Join, Split, Filter, Iterate and Fold [37]. These additional operators allow a large variety of tasks to be implemented with less intermediate steps than in previously mentioned frameworks. The operators allow for high level implementations instead of writing complex long chains of Map and Reduce operations. When a task is started the high level job is automatically transformed into an efficient execution plan optimized for the available cluster of nodes.

Apache Flink adopts a fault tolerance mechanism that is based on Chandy-Lamport distributed snapshots [14]. This popular mechanism allows for a global state to be maintained using stable checkpoints to which the cluster can revert in case of failures. A powerful property of Apache Flink is that it guarantees a strong exactly-once stream processing, meaning that each data element is exactly processed once, even in case of failures.

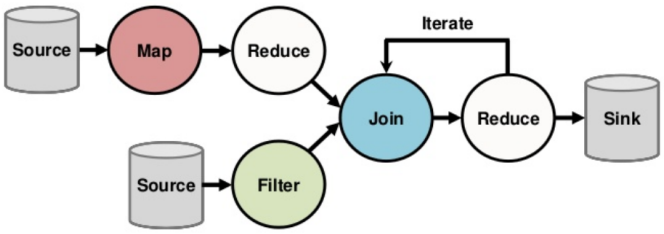


Figure 2.5: An example of an Apache Flink execution pipeline, illustrating the chains of operations performed on two incoming data streams [28]

2.2.5. Suitable framework

Although Hadoop and Apache Spark are very popular frameworks, they have a common drawback. Both platforms require data to be stored before it is processed. In case of Hadoop, the nature of the MapReduce programming model requires all data to be stored before the first mapping operation can be performed. Apache spark is a batch processing framework, which makes it in terms of data storage an improvement over the older Hadoop framework. Even though it offers streaming capabilities, in the core of Spark this is executed using micro-batches. This still put demands on disk requirements and introduces more latency than in true stream processing. Apache Storm and Apache Flink, on the other hand, both offer scalable distributed stream processing. Apache Flink offers exactly-once stream processing whereas Apache Storm guarantees at-least-once processing. This means that Apache Flink handles failures more efficiently. In terms of performance, a number of studies have been conducted comparing the throughput of Apache Storm and Apache Flink. One thorough study shows a 35x higher throughput of Apache Flink with Storm processing 400 thousand messages per second and Apache Flink an astonishing 15 million messages per second on similar hardware [13]. Also taking into account the more specialized set of elementary operators, Apache Flink is clearly the more superior platform.

For implementing the algorithms designed in Chapters 3, 4 and 5 Apache Flink will therefore be used as the underlying data processing platform.

2.3. Count-min Sketch

As briefly mentioned in section 2.1.1, Count-Min sketches are a commonly used technique for efficiently estimating the number of occurrences of encountered elements [20]. The naive approach of maintaining a counter for each element, would require increasing amounts of memory making it infeasible to count large numbers of elements. Count-Min sketches store counts in a compact data structure, based on Bloom filters. The data structure consists of a table with d rows and w columns.

On creating the data structure, all cells in the table are initialized with a zero. Each of the d rows is assigned a hash function h that is independent of the hash functions of other rows. Each hash function h maps an input x to an integer value $h(x) = i$, with i between 0 and w. The resulting number is an index in the table row that corresponds to h.

Once an element x is observed, it is hashed to $h(x) = i$ for all functions h belonging to each row d. All resulting row indices i for each d together form a series of cells that identifies the encountered element. At each of the row indices i, the value stored in the corresponding cell is increased by one.

For retrieving the count of an element x, it is hashed using each hash functions h to obtain all row indices $h(x) = i$. At all corresponding row indices the stored values are looked up from the Sketch. The minimum of all retrieved values is returned as the estimated count of element x. The storing or lookup operation is depicted in figure 2.6.

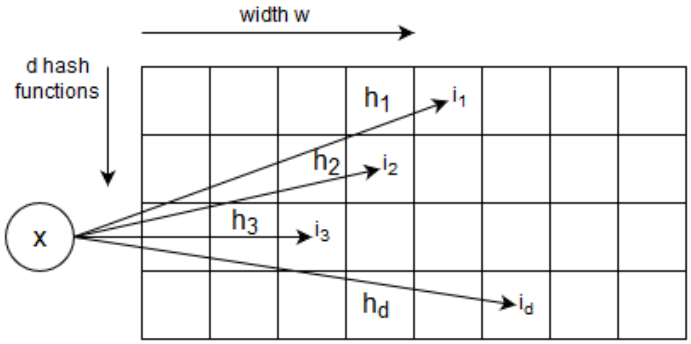


Figure 2.6: A Count-Min sketch with hash functions h_1 to h_d pointing to the cells containing all estimates of the count of item x. The minimum of all count estimates approximates the true count of element x.

The minimum of all retrieved values is used, since it could be the case that a hash function maps two different elements to the same row index. In such an event, the count stored at the cell is increased twice. However, when taking the minimum over the cells pointed at by multiple hash functions, this hash collision does not impact the finally retrieved value. By using multiple hash functions, the count is only imprecise if for a specific element all cells it is stored at have had a hash collision. Hence, the use of multiple hash functions in combination with computing a minimum on the retrieved counts, reduces the inaccuracy of the Count-Min sketch data structure. Another option for reducing the chance of hash collisions is increasing the width of the sketch. Of course by increasing both the number of hash functions (or sketch height) and the sketch width, the total memory footprint will increase. A trade-off is made between keeping the sketch size small and the chance of having counting errors.

The paper from Cormode G. published in 2009 [19] contains a detailed analysis of this trade-off. Suppose we would require the overestimation error for each element to be smaller than the factor ϵ with probability $1-\delta$. Then the sketch width should be of size $w = \lceil e/\epsilon \rceil$ with for e Euler's number. The number of hash functions required to obtain the $1-\delta$ certainty is given by $d = \lceil ln(1/\delta) \rceil$.

As a numeric example suppose we allow the overestimation to be within 5% with a probability of 99%. The corresponding ϵ is 0.05. Therefore, the sketch width should be $w = \lceil e/\epsilon \rceil = \lceil 2.71828/0.05 \rceil = 55$. To obtain a certainty of 99%, the corresponding δ equals $1-(99/100) = 0.01$. Hence, the number of hash functions to obtain a 99% certainty is given by $d = \lceil ln(1/\delta) \rceil = \lceil ln(1/0.01) \rceil = 5$.

2.4. NetFlow Technology

In 1996 Cisco introduced NetFlow technology with the aim to help make network traffic analysis easier and less resource intensive. Since then, NetFlow has become widely adopted and has become an important technique used by network engineers and analysts all over the world [42]. NetFlow defines a data type that summarizes packets exchanged between applications of two communicating end-points. Traffic from each source IP and source port towards each combination of destination IP and destination port is being aggregated into separate flow records, allowing a high level analysis on all communicating processes. A flow record summarizes a traffic channel into a

variety of statistics, of which the following selection is the most common: connection type, time of first packet, time of last packet, total number of bytes transferred and total number of packets transferred. Additionally, the NetFlow can contain network layer 3 header and routing information and level 4 TCP or UDP parameters depending on the collection setup.

NetFlows are created by routers and switches on packets that are being forwarded to the next hop towards their destinations. The resulting flow records, containing the packet summaries, are sent over UDP to a dedicated machine collecting NetFlows from one or more exporters throughout the network. The NetFlow collector stores the incoming NetFlows on disk and usually offers a web interface to analysts, enabling them to query the flow records gaining insights into the collected statistics. An overview of the common NetFlow collection architecture is shown in Figure 2.7.

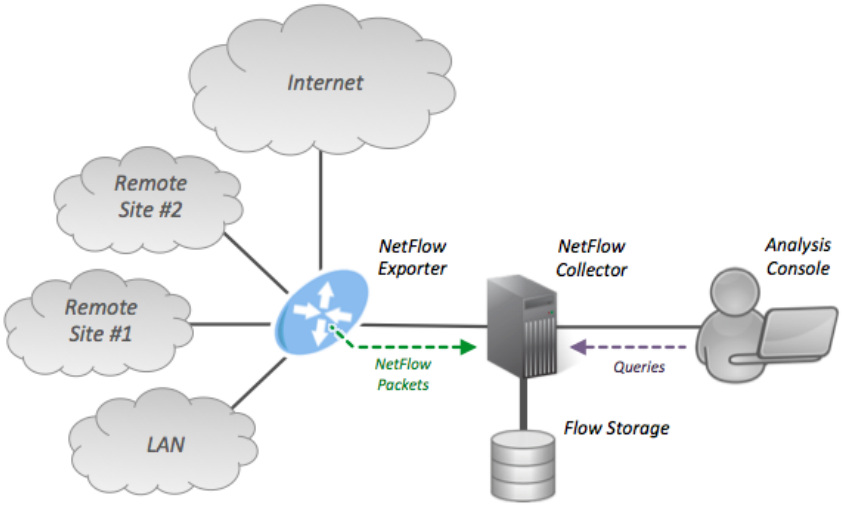


Figure 2.7: A common network architecture when using NetFlow technology. Routers and switches act as NetFlow exporters, summarizing network traffic channels and sending the resulting flow records to a common collector. The NetFlow collector combines flow records and offers an interface for analysts to query the data. [99]

2.4.1. Sampling

To cope with large data rates, especially at high speed backbone links, Cisco introduced a resource saving sampling method [16]. This sampling method uses only one in every n packets for computing the flow statistics. The first proposed methods selects every n^{th} packet or selects a random packet in windows of n packets. From the subset of packets that are aggregated into the flow record, the true statistics are approximated using the captured statistics and the sampling rate. This makes NetFlow collection possible on high speed routers, however it comes at a cost of less accurate statistics.

2.4.2. NetFlow v9

Since the first version of NetFlow, the technology has evolved intensively. The most recent version of Cisco's NetFlow technology is NetFlow v9, which supports summarizing packets of more protocols than the older NetFlow v5. For example Multi-cast, IPSec, and Multi Protocol Label Switching packets can be summarized in the latest version. NetFlow v9 is also more flexible by supporting to configure templates, defining what data will be captured and exported to the NetFlow collector.

2.4.3. IPFIX

IPFIX, short for IP Flow Information eXport, is strictly not a type of NetFlow, but it is based on NetFlow v9. It is an RFC standard [18] specifically meant to open up flow technology to a broader range of vendors. IPFIX is no longer focussed on collecting the information types supported by Cisco's hardware, but instead offers a more general approach and even more flexibility than NetFlow v9. The latest NetFlow version has 79 field types reserved for collecting various types of information. IPFIX on the other hand supports up to 238 field types and variable field lengths, allowing more diverse and non-fixed sized data, such as URLs.

2.5. State Machines

A State Machine, or Deterministic Finite Automaton (DFA), is a mathematical model of computation represented by a directed graph. The formal definition as put forward by Sipser in his book "Introduction to the Theory of Computation" [76] is the following:

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the *transition function*,
4. $q_0 \in Q$ is the *start state*, and
5. $F \subseteq Q$ is the *set of accept states*.

An example finite automata, or state machine, M_1 is shown below in figure 2.8. M_1 has three states, which are: q_1, q_2 and q_3. The start state is q_1, since it receives an arrow with no state attached. The transition function δ maps states in combination with a symbol from alphabet Σ to another state, resulting in transitions indicated by edges between the nodes in the graph. The labels of the transitions are the symbols with which a transitions to the targeted state is allowed. State q_2 is an accept state, indicated by the double circle around it. The state machine M_1 is called a Deterministic Finite Automata, since it has at each state for each symbol at most one transition. Also Nondeterministic Finite Automata (NFA) do exist, which can have multiple transitions per symbol departing from the same state. In this thesis NFA's are not considered, since during the evaluation of an NFA all possibilities should be evaluated, which makes visualisations of models less intuitive to read and introduces performance penalties in evaluation.

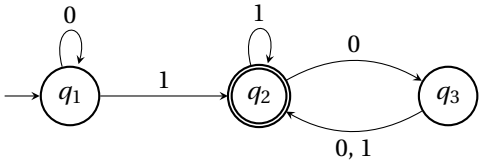


Figure 2.8: A State Machine M_1 consisting of three states

Besides DFA and NFA, another common type of State Machine is the Probabilistic Deterministic Finite Automata (PDFA). A PDFA is similar to the DFA, but includes the probabilities of observing each symbol $a \in \Sigma$ for all the states from Q. This addition results in the ability to derive for each sequence what the likelihood of occurrence is, by multiplying the probabilities for each symbol transition of the symbols in the sequence together. This enables testing the similarity of State Machines by comparing probability distributions. The comparison of State Machines will be useful later in this thesis report, for example in section 4.5.1. Hence, future mentions of the term State Machine in this research points to the PDFA type of State Machines.

2.6. State Machine similarity

In order to evaluate the correctness of State Machine learning methods or to compute the similarity between models, it is important to be able to compute the similarity of State Machines. One could try to visually compare State Machines or automate this by finding a metric for computing the similarity on State Machine model specifications. This is easy for identical State Machines; however, it is very hard to give a precise score when State Machines slightly differ. How bad is it if a State Machine contains one additional state, or if it misses one state? Which states are equal and are all states equally important? These questions can be avoided by considering a State Machine as a model that represents a certain formal language. Hence, two State Machines are similar if the languages they accept are similar. By tracing paths through each State Machines, one can obtain a lists of strings that contained in the language described by each model. A comparison of the lists of accepted strings using for example a Hamming Distance [44] results in a measure of similarity between the models. In case of PDFA's, more information is contained in the models. Besides describing all possible transitions, a PDFA also contains the probability for each state transition. A PDFA models a language together with a probability distribution that describes how likely it is for each string to appear. Therefore, PDFA similarity is often computed using the probability distributions of two models [5] [6] [93].

Entropy

The most common approaches of comparing PDFA's rely on the entropy of the probability distribution. In Information Theory, entropy is a measure of uncertainty of a random variable [21]. For example, a fair die has a higher

entropy than a fair coin, since it has more outcomes. For a discrete random variable X with alphabet A and probability mass function $p(x) = Pr\{X = x\}$ with $x \in A$, equation 2.1 shows the formula for the entropy of the probability distribution.

$$H(X) = -\sum_{x \in A} p(x)\ \log_2 p(x) \tag{2.1}$$

In the example of a fair die, all of its six sides have an equal probability of 1/6, thus giving an entropy of 2.585. The example of the fair coin, only has two outcomes with both a probability of 0.5, resulting in an entropy of 1.0.

By taking the logarithm with base 2, the resulting entropy can be seen as the minimum number of bits required to encode a single draw from the distribution. In other words, for a fair die, roughly 3 bits of information is needed to describe one throw. A fair coin toss contains only 1 bit of information.

2.6.1. Cross Entropy

The entropy of a distribution can be regarded as the information contained in a single sample, which is based on the characteristics of the underlying probability distribution. This can be useful for model comparisons, by computing the cross entropy between models. Cross entropy indicates the average number of bits required to encode a sample from the true model p with a coding scheme based on the derived model q. In case both models use the same alphabet A, the formula for cross entropy is given in equation 2.2.

$$H(p, q) = -\sum_{x \in A} p(x)\ \log_2 q(x) \tag{2.2}$$

For two perfect matching models, the cross entropy is equal to the entropy of the true model p. Any deviations in the probability distribution of q with respect to the true model, would result in an increase of the cross entropy. Hence, a lower cross entropy between samples of the true and induced model means a better generalization of the derived model to unseen samples. Therefore, the cross entropy can be used as a means to optimize learning methods for models that describe probability distributions [79].

2.6.2. Perplexity

The perplexity measure of a model is closely related to the cross entropy. It is computed by taking two to the power of the cross entropy, as formulated in equation 2.3. Therefore, an optimization that minimizes the cross entropy yields the same results as minimizing the perplexity. It is; however, considered as a more intuitive formulation. The perplexity equals to the possible number of outcomes if each of those outcomes was given the same probability. Instead of a fair die having a 2.585 bit entropy, it now has a perplexity of 6 which is equal to the number of sides of the die.

$$2^{H(p,q)} = 2^{-\sum_{x \in A} p(x) \log_2 q(x)} \tag{2.3}$$

Additionally, the power of two in the perplexity formula cancels out the logarithms, resulting in a linear scale. This makes it more intuitive for humans to interpret the significance of differences between values. As a result, the perplexity is a popular notation which is used to validate model correctness in various studies [24] [91] [86].

2.6.3. Kullback-Leibler divergence

As mentioned earlier, cross entropy indicates the average number of bits required to encode a sample from the true model p with a coding scheme based on the derived model q. As a result, the cross entropy of p and q can not be less than the entropy of the original model p, since that was the optimal number of bits required to represent all information in the samples of p. This makes the cross entropy a suitable statistic for the optimization of model q. However, since the cross entropy depends on the value of the entropy of the model p, it is hard to define a common threshold above which models can be regarded equal. One has to take into account the entropy of the true model p. This leads to the well-known Kullback-leibler divergence [48]. The Kullback-leibler divergence is the difference between the cross entropy and the entropy of the true model, as defined in equation 2.4. Note that this equation again only holds when both models use the same alphabet A.

$$D_{KL}(p \,||\, q) = H(p, q) - H(p) = \sum_{x \in A} p(x) \log_2 \frac{p(x)}{q(x)} \tag{2.4}$$

Using the Kullback-leibler divergence, it is possible to define a common threshold below which to regard model deviations as acceptable. Since we will need to compute PDFA similarities in chapters 4 and 5 and we need to define a maximum distance threshold later in chapter 5, this commonly used distance measure [86] [38] [24] will be the method of choice.

3

Apache Flink Streaming Implementations

This chapter explores the Apache Flink framework and defines an approach for implementing streaming algorithms. First, in section 3.1 the structure of an Apache Flink job and the supported streaming operators are discussed. In 3.2 a setup is described for running Apache Flink jobs. Sections 3.3 till 3.5 explore various well-known data processing problems in increasing complexity. The problems addressed are: Heavy Hitters, Frequent Patterns and Frequent Sequences. For each of these problems, existing attempts at solving the problem are listed. Next, streaming implementations are created using the Apache Flink framework, offering insights into the approach of how to tackle processing problems using Apache Flink. The implementations focus on solving each problem for network traffic analysis on NetFlow data. The problems increase in complexity and each problem demands other techniques. Finally, in section 3.6 the implementations and techniques are generalised into a basis that can be used for designing the main streaming implementation of this thesis which is described in chapter 4.

3.1. Apache Flink

Apache Flink provides a platform for executing transformations on distributed data collections, as explained in section 2.2.4. It offers a processing engine that can be accessed by Java or Scala applications via the Flink API [35]. With this API a regular program can be appended with commands that are executed using the Apache Flink engine, either on the same host or across a configured cluster.

3.1.1. Job Structure

A series of API calls together form a Flink Job, which are executed using lazy evaluation. This means that the API calls, like connecting with a data source or performing transformations, are not directly performed. Instead, these calls define the operations that will be performed, once the execution is explicitly triggered. The definition of an Apache Flink job has a fixed structure, consisting of the following five steps:

1. Obtain an execution environment
2. Connect to one or more data sources
3. Specify a pipeline of transformations that will be made on the data of each data source
4. Specify where to output the results of the computations
5. Trigger the program execution

Whether the job is executed locally or on a cluster, as well as other execution related settings, depend on the Execution Environment that is configured in the first step. Step 2 till 4 define the body of the Flink Job after which execution is triggered. The second and fourth step allow to connect with sources for consuming data elements and connections with sinks to output computation results.

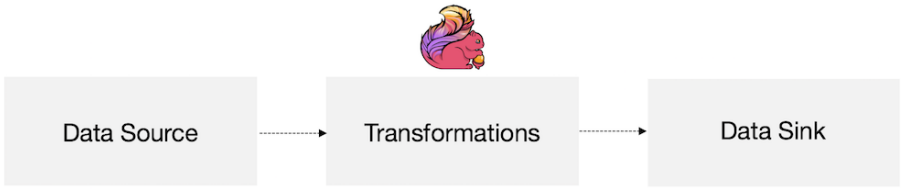


Figure 3.1: The high level structure of a Flink Job. Data of or more data sources follow a chain of transformations and are outputted to one or more data sinks. [36]

3.1.2. Sources and Sinks

The Apache Flink framework supports connection with a variety sources and sinks. Some popular supported connections are:

- Apache Kafka (source/sink)
- Amazon Kinesis Streams (source/sink)
- Elasticsearch (sink)
- Hadoop Distributed File System (sink)
- Twitter Streaming API (source)
- RabbitMQ (source/sink)

A connection with Apache Kafka in particular is useful as it offers a low-latency, high-throughput data pipeline, focussed on streaming applications with robust fault-tolerance [85]. It is easy to configure and can be deployed in self-controlled environments, in contrast to managed solutions like Amazon Kinesis. Furthermore, Apache Kafka can handle very high data rates, higher than for example RabbitMQ [23]. This makes Apache Kafka an excellent candidate for providing Apache Flink with streaming data in high-performance environments.

3.1.3. Streaming Operators

Streaming operators form the building blocks of each Flink Job, by transforming data from one form to another, closing in on the desired outcome. Apache Flink supports, as of writing, thirty-two different streaming operators [37]. The most important ones are described in this section.

Map

The Map operator takes one element and produces one element. It modifies each incoming element similarly resulting in a stream of modified elements. The following Map operation doubles the values of each incoming element:

```
DataStream<Integer> dataStream = //...
dataStream.map(new MapFunction<Integer, Integer>() {
  public Integer map(Integer value) {
    return 2 * value;
  }
});
```

FlatMap

The FlatMap operator takes one element and produces zero or more elements. The following FlatMap operation splits sentences into words:

```
dataStream.flatMap(new FlatMapFunction<String, String>() {
  public void flatMap(String value, Collector<String> out) {
    for(String word: value.split(" ")){
      out.collect(word);
    }
  }
});
```

Filter

The Filter operator evaluates a boolean function for each element and retains the elements for which the function returns true. The following Filter operation filters out all zero values:

```
dataStream.filter(new FilterFunction<Integer>() {
  public boolean filter(Integer value) {
    return value != 0;
  }
});
```

KeyBy

The KeyBy operator takes a single data stream and transforms it into a keyed data stream. A keyed stream divides data elements into partitions based on the values of the key field. Operations that are performed on a keyed stream are distributed across Flink nodes by assigning each instance a subset of keys with their corresponding streams. This allows meaningful parallel processing of subsets with similar data elements.

```
dataStream.keyBy("group") // Key by field "group"
dataStream.keyBy(0)       // Key by the first element of a Tuple
```

Reduce

The Reduce operation performs a "rolling" reduce on a keyed data stream. It continuously combines the incoming element with the last reduced value. As a result, the elements corresponding to each key of a keyed stream are reduced to a single value. The output of a reduce operation is a data stream containing the reduced values of each key of the keyed stream. The following Reduce operation creates a stream containing a sum for each key of a keyed stream:

```
keyedStream.reduce(new ReduceFunction<Integer>() {
  public Integer reduce(Integer value1, Integer value2) {
    return value1 + value2;
  }
});
```

TimeWindow

A TimeWindow operation can be defined on a keyed stream to execute the subsequent operations on chunks of data that arrived within a window of time. A time windows groups data in each key according to a selected characteristic (e.g., the data that arrived within the last 5 seconds). The following TimeWindow operation divides the keyed stream in chunks of 5 seconds.

```
dataStream.keyBy(0).timeWindow(Time.seconds(5));
```

WindowAll

A WindowAll operation is performed on a stream that is not partitioned into a keyed stream. All data elements that arrive within a certain time are forwarded to the subsequent operation. The following WindowAll operation divides all incoming data in chunks of 5 seconds:

```
dataStream.windowAll(Time.seconds(5));
```

3.2. Experimental Setup

This section describes a hardware and software setup for executing the streaming algorithms proposed in this thesis. As section 2.2.4 motivates, Apache Flink is a well-suited platform for creating scalable implementations. Section 3.1.2 argues why Apache Kafka is a good fit for providing Apache Flink with streaming data. Hence, both platforms are

combined, forming the core of the experimental setup. Apache Kafka accommodates a data pipeline of which data is consumed by Flink, but it also requires at least one data producer generating the streaming data. Since this thesis focusses on applications in network based threat detection, Apache Kafka is connected to a source of NetFlows. A high-level overview of this process is depicted in figure 3.2.

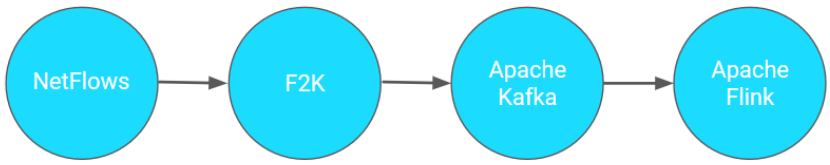


Figure 3.2: A high-level overview of the experimental setup. Raw NetFlows are sent from a NetFlow Collector to a host publishing NetFlows to Kafka. The tool vFlow reads the NetFlows from an Ethernet port, parses them into JSON format and publishes them via a Kafka Producer. Apache Kafka facilitates a data pipeline that forms a source for Apache Flink. Apache Flink executes a streaming algorithm on the incoming NetFlows.

3.2.1. Live Network Setup

As depicted in the overview in figure 3.2, the setup consists of 4 main parts. The next sections describe each individual part of the setup in more detail, when used in a live network.

1. NetFlow generation

The setup starts with a live network consisting of a series of routers and switches summarizing network traffic into NetFlow. The NetFlows are sent to a central Flow Collector, combining the NetFlow of the different sources. The Flow Collector transmits all incoming NetFlows to a host running Apache Kafka. In a small network, the Flow Collector can be directly connected with a single Kafka node. In high performance networks, the NetFlows transmitted by one or multiple Flow Collectors can be equally divided across a Kafka Cluster using a load balancer.

2. NetFlow to Kafka translator

On each Kafka node, the tool vFlow [72] is running, translating incoming NetFlows to JSON objects. vFlow reads NetFlows from the configured Ethernet port and publishes the resulting JSON objects to a Kafka Topic using a Kafka Producer. In a multi-node setup, each Kafka node publishes NetFlows to the same Kafka Topic. This allows Kafka to take care of the routing of the distributed NetFlows for further processing.

3. Apache Kafka

All NetFlows that are produced in the Topic across one or multiple Kafka nodes are routed by Apache Kafka to the right destinations. To provide scalability, the incoming NetFlows are divided over Partitions within the Topic. On nodes that require access to the NetFlows, a Kafka Consumer is created that registers for this topic. A Consumer receives all NetFlows, unless it registers for a specific Partition within the Topic. In that case, it receives only the NetFlows that were assigned to that partition. Therefore, in high-performance setups, a single Consumer should be used with a different Partition assigned to each host tasked with processing a subset of the NetFlows. In a less demanding setup, using a Consumer with a single partition operating on a single Kafka node is sufficient.

4. Apache Flink

Apache Flink runs across one or more hosts, depending on the required computational power. Each host in a Flink Cluster runs the Kafka Consumer with a specific Partition assigned. The Kafka Consumer is executed as a connector in the Flink framework, allowing Flink to process NetFlows directly after they are delivered by Kafka. The Flink Cluster is assigned a Flink Job which defines the streaming algorithm that will be executed on the incoming data elements.

3.2.2. Design Setup

Since a partly goal of this thesis is to design and implement streaming algorithms in a robust way that can be used in practise, the live network setup mentioned in section 3.2.1 proposes how to apply this work in practise. However, due to practical reasons, during the design process in this thesis a more compact setup is used.

NetFlow generation

In this chapter the live network is replaced by a NetFlow generator which acts as the live network by generating the flows that are consumed by vFlow. This offers more flexibility and control during the design process.

3.3. Heavy Hitters

A quite basic, yet very well-known, stream processing task is the Heavy Hitters problem. This problem aims to find a list of the N most often occurring instances in a data stream. Translated to network streams, this could mean computing a list of the top-N source IPs that have been transmitting the most amount of traffic. This statistic can be used in various use cases, like for example for detecting volumetric denial-of-service attacks.

3.3.1. Existing Approaches

The Heavy Hitters problem is an instance of the Frequent Elements problem as described in section 2.1.1. As briefly mentioned, the Misra-Gries algorithm [57] is one powerful approach for solving this problem. This algorithm approaches the problem by maintaining N elements with a corresponding counter. Once an element is consumed, the corresponding counter is increased. Or in case the element is not yet contained in the set, all counters are decreased. Once a counter reaches zero, the element is removed allowing for a new element to be added. On terminating the streaming algorithm, this approach results in a set of majority items, occurring more often than $\frac{1}{N}$ times the size of the stream. Another approach for finding the most frequent items is proposed by Yi et al. in 2013 [101]. Sliding windows are being used in a redesigned Space Saving algorithm, to find the most frequent items on streaming data in a memory efficient way. The proposed approach; however, is no distributed solution which restricts the scalability to only vertical scalable. Another approach by Ben-Basat et al. [7] works on distributed streams; however, requires two-way communication between each node and the coordinator for constantly updating a centrally stored intermediate result.

In the next section, a distributed approach is proposed that does yield an exact solution for the Heavy Hitters problem using time windows, without the need for continuously communicate intermediate results to a central coordinator node.

3.3.2. Proposed Streaming Approach

The existing approaches use memory efficient data structures preventing the need to store a counter for each IP address. However, this results in an approximation of the answer, which might not be necessary. Since for one of the thesis goals, applying State Machines in network threat detection, it is required to learn a State Machine for each IP address, the Heavy Hitters problem will be solved using an exact solution which stores a counter for each IP address. Maintaining a counter per IP address is a first step in testing the feasibility of storing data of a large number of hosts in a streaming application.

By maintaining a counter for each IP address, it is possible to compute the top-N Heavy Hitters in a single pass through all IP-count combinations. An efficient and exact approach is to maintain a fixed sized priority queue while traversing through all final counts. A priority queue is a data structure that maintains a list of items with a specific ordering. On adding an item it is checked whether the corresponding value, in this case the IP occurrence count, is higher than the lowest value in the queue. If that is the case, the current lowest element is removed and the new element is added. As a result, at any point in time, a priority queue of size N contains the N elements with highest value. Since a priority queue requires only a single pass over the counts, it uses O(1) checks to determine if an element should be added and it uses O(N) updates of the queue, makes it a very efficient, yet exact, data structure.

3.3.3. Apache Flink Implementation

The implementation uses the five steps of a typical Flink job structure as described in section 3.1.1. The data source is a Kafka Consumer that consumes any NetFlow data that is produced in the topic which the Consumer is assigned to. The final output of the algorithm is configured to be printed to the console. These second and fourth steps of the general Kafka job structure are repeated throughout the implementations in this chapter. What is different for each implementation is the third step, i.e. the actual pipeline of flink operators performed on the incoming NetFlow data.

Two operations should be performed on the incoming NetFlows, dividing the pipeline in two consecutive steps. The first step computes the total count of each source IP. The second step processes all total counts using the Priority Queue and outputs the top-N source IPs.

Step 1: distributed packet counting

Since a count per source IP is required, incoming NetFlows need to be divided into streams of flows with equal source IP to allow parallel counting. However, each incoming NetFlow JSON object contains a wrapper of meta data and one or more actual NetFlows inside. The NetFlows stored in a single JSON object can summarize packets of any number of host, so the inner NetFlows should first be extracted before it is possible to divide incoming flows into a stream per source IP. Taking into account the available streaming operators described in section 3.1.3, a *FlatMap* operation is the suitable candidate for mapping a single incoming NetFlow JSON object to separate objects for each NetFlow that is contained. After the *FlatMap* operation, the stream consists of all individual NetFlows. At this point, the NetFlows can be divided into a stream per source IP using the *keyBy* operator. The advantage of this is that each stream now contains all NetFlows needed to compute their total count, allowing parallel execution of the next steps. Since we

are dealing with potentially infinite streams, the computations are performed on regular time intervals. Each stream is split into chunks of 10 seconds using the *TimeWindow* operator. Next, on the stream of each source IP a *Reduce* operation is performed. This is a rolling operation with a single object that updates its state, i.e. the total count, as new NetFlows are consumed. The *Reduce* operation concludes the first step of the algorithm resulting in a stream containing the total counts for each source IP. A Java-like pseudocode of the operators used in the first step of the algorithm is shown below in Pseudocode 1.

```
DataStream<HeavyHitterNetFlow> hostFlowCounts = netflowStream
  .flatMap(in, out) {
    for (HeavyHitterNetFlow flow : in.dataset) {
      out.collect(flow);
    }
  })
  .keyBy("srcIP")
  .timeWindow(Time.seconds(10))
  .reduce(rollingCount, incomingNetflow) {
    rollingCount.addHitter(incomingNetflow);
    return rollingCount;
  });
```

Pseudocode 1: The first step of the Heavy Hitters implementation in Apache Flink. A *FlatMap* operator maps bundles of NetFlows into a stream of all individual NetFlows. With *keyBy* the NetFlows are divided into a stream for each source IP. Using a *TimeWindow* operator the subsequent step is performed for all flows that arrive within 10sec. The final step is a *Reduce* operator which counts the NetFlows of each source IP into a total count per host.

Step 2: computing the overall top-N sources

Step 1 yields a stream with a total count of each source IP. The second step should convert this stream into a top-N list. The stream resulting from step 1 receives a burst of elements as each 10 second time window expires. To convert results of each time window into a new top-N list, a similar window operation needs to be performed on the resulting stream from step 1. Since the second step only handles one value per IP address and needs to compare and store items in a single Priority Queue, it is not executed in parallel. Therefore, the pipeline starts with a *WindowAll* operator using a 10 second time window. This splits the entire stream into 10 seconds chunks, instead of being performed on distributed streams like is the case with the *TimeWindow* operator. The total counts of each 10 seconds chunk can be combined into a top-N list using a *Reduce* operation. In this operation a Priority Queue is maintained and for each incoming total count, the queue is updated if needed. At each point in time, the queue contains the top-N counts with corresponding IP addresses. After consuming the counts of all source IPs, the priority queue contains the exact solution to the heavy hitters problem. The top-N most frequently encountered source IPs are now outputted by returning all stored counts from the priority queue. The second step of the HeavyHitters algorithm is shown in Pseudocode 2.

```
DataStream<HeavyHitterNetFlow> topN = hostFlowCounts
  .windowAll(TumblingEventTimeWindows.of(Time.seconds(10)))
  .reduce(rollingTopN, newHostCount) {
    rollingTopN.updateQueue(newHostCount);
    return rollingTopN;
  });

topN.print();
```

Pseudocode 2: The second step of the Heavy Hitters implementation. The total count for each source IPs arriving in a 10 second time interval are combined into a single stream using the *WindowAll* operator. Next, a *Reduce* operation is performed, which combines all incoming total counts one by one keeping only the top-N counts using a Priority Queue. Finally, the contents of the Queue is printed.

3.4. Frequent Patterns

Counting frequent patterns is another interesting problem, offering insights into network traffic parameter combinations that often occur. Examples are communication protocols or port numbers that are often used by the hosts in a network. The applications of frequent pattern mining are more powerful than solutions to the heavy hitters problem, since it not only focuses on the number of connections, but rather on the characteristics of the connections that are made. The resulting statistics can be useful for understanding network behaviour or for detecting anomalies in network traffic.

3.4.1. Existing approaches

One of the first approaches to solving the frequent pattern problem was proposed in 1994 by the introduction of the Apriori algorithm [1]. This algorithm aims to detect frequent transactions in a transactional database or to find association rules in for example shopping basket data. The Apriori approach has; however, the drawback that all possible item combinations are first generated and then evaluated. This does not scale to large frequent itemsets, since the number of possible combinations grow exponentially. The popular FP-Growth method [39] proposes a solution to this problem, by introducing a compact FP-tree data structure containing counts of all encountered patterns, or in this use case flow parameter combinations. It consumes the transactions one by one without generating all possible parameter combinations. The downside of this solution is that it requires two passes over the data impeding applications in stream processing. In a recent paper [73] a single pass frequent pattern tree is proposed, which could be executed on data streams. The drawback of this newly proposed method is that this approach is difficult to parallelize since it requires a central tree structure. In the past decade, research is performed on a distributed FP-Growth variant: Parallel FP-Growth (PFP) [52]. This approach offers scalable frequent pattern mining capabilities based on MapReduce; however, this focusses on batch processing rather than real-time solutions. A real-time solution which is capable of computing frequent itemsets over sliding windows is proposed in the *Moment* system [15]. However, the system does rely on a centrally maintained state that and the paper does not describe how the execution could be parallelized.

In the next section, a distributed streaming approach is proposed that computes a top-N most frequent patterns using a Count-Min Sketch and a Priority Queue.

3.4.2. Proposed Streaming Approach

As explained in section 2.3, a popular data structure for efficiently storing a large number of frequencies in stream processing is the Count-Min Sketch [20]. This data structure consists of a bloom filter containing the counts of each event. However, since events are mapped to locations in the bloom filter using hash functions, one cannot trace back to which earlier consumed patterns the highest values stored in the sketch belong to. The standard Count-Min Sketch data structure is therefore only useful for updating and using the stored frequency of an item that is currently consumed from the stream. A trivial solution would be to store a set of all encountered patterns together with the Count-Min Sketch. The algorithm can then iterate over this list of observed patterns after termination and request for each pattern the frequency from the sketch. By using a Priority Queue, similar to the Heavy Hitters approach from section 3.3, the top-N most frequent patterns can be computed on the pattern frequencies returned from the sketch. We will call this approach: Trivial Frequent Patterns or TFP for short. The downside of this trivial solution is that maintaining a list of observed patterns defies the purpose of using a memory saving Count-Min Sketch, which was used to prevent maintaining a counter for each pattern. Therefore, there needs to be a solution that does not require a full list of patterns besides the Count-Min Sketch.

One solution is to keep track of the current most frequent patterns using a limited sized priority queue. This has a fixed memory footprint compared to an ever-growing set of all encountered patterns. To ensure that the performance of this approach is not worse than Trivial Frequent Patterns, a proof will be provided for this claim. Before we proof the claim, the distributed algorithm is described.

Variables

All threads of each node maintain the following variables:

- **Sketch**: a Count-Min Sketch which keeps track of the frequencies of encountered patterns
- **Q**: a priority queue of size N, with $N > 0$

Distributed Frequent Patterns

The algorithm consumes an unbounded stream of NetFlows. All NetFlows are distributed over the different threads based on the source IP contained in the incoming NetFlow, resulting in the NetFlows for a specific source IP to always be processed by the same streaming pipeline in the same thread.

The following steps are executed in each thread:
For each incoming NetFlow the algorithm generates a set of tuples and triples t of parameter values contained in the NetFlow. Each tuple or triple contains as first value the source IP address. Example tuples and triples are (sourceIP, destinationIP) and (sourceIP, sourcePort, destinationPort). For each t a hash h is computed and looked up in $Sketch$. The returned value v is increased with 1, resulting in v_{new}. Next, v is overwritten with v_{new} in $Sketch$ at location h.

If v is in Q, v is removed from Q and v_{new} is added.
Else if the size of Q is less than N, v_{new} is added to Q.
Else, the lowest value v_{qmin} is looked up from Q and if v_{qmin} is lower than v_{new}, v_{qmin} is removed from Q and v_{new} is added.

On termination of the system or at the end of a streaming window, all threads output their priority queue Q to one single thread at one of the nodes in the cluster. This thread iterates over the combinations stored in all queues and composes a final N-sized queue of the overall most frequent patterns.

Proof

This section provides a proof that the above described distributed implementation (Distributed Frequent Patterns or DFP) gives the N most frequent patterns, with no larger error than observed in the trivial single threaded approach (Trivial Frequent Patterns or TFP). The used error metric is the number of patterns from the optimal solution that do occur with the correct frequency in DFP or TFP.

The proof consists of two parts, the first part proofs that the use of a priority queue instead of a set of all encountered patterns does not impair the accuracy of the final set of most frequent patterns. The second part proves that a distributed variant does not perform worse than a single threaded variant.

Proof I

Proof by strong induction that a single threaded execution of Distributed Frequent Patterns (DFP) gives the N most frequent patterns, with no larger error than observed in the Trivial Frequent Patterns (TFP) approach.

On consuming a pattern p both approaches look up the frequency in $Sketch$ and increase the value by 1. The resulting frequency v is either the true frequency encountered so far or it will be a overestimation if all locations $h(f)$ for all hash functions h have been at least once the result of another $h(g)$ for some previously consumed pattern g.
In the TFP variant, the pattern p is added to the set S of encountered patterns. After all patterns have been consumed, the output of TFP is generated by iterating over each pattern in S and maintaining a top-N list of all frequencies looked up from $Sketch$.
In the DFP variant, the pattern p is added to Q with value v if the queue is not full yet or if the smallest value in the queue is less than v. After all patterns have been consumed, the output of DFP is Q.

Lemma I: if a pattern p occurs in the results of both DFP and TFP, the frequency of p in TFP is equal to or higher than the frequency of p listed in the result of DFP. This holds because from the last occurrence of p in DFP until the top-N list of TFP is composed, the frequency of p in $Sketch$ will either remain the same or increase due to hash collisions.

DFP will only perform worse than TFP if DFP has more overestimations than TFP. Each overestimated pattern o from the end result of DFP can either:

- *also appear in the top-N of TFP*
  If pattern o appears in both DFP and TFP, according to Lemma I, the value of o in TFP will be equal to or larger than the value of o in DFP. Since the frequency of o was an overestimation in DFP and the pattern is present in TFP with an equal or larger frequency, it must also be an overestimation in TFP.
- *not appear in the top-N of TFP*
  If pattern o is not included in the final result from TFP, this means the lowest value from the top-N of TFP is higher or equal to the frequency of o. All frequencies from the patterns p from positions o to N in the DFP are therefore lower than the lowest value in the top-N of TFP. Each of these patterns p are either:
  - *not contained in TFP*
    Values in DFP are less than or equal to values in TFP (Lemma I). Therefore, if on consuming a new pattern it becomes part of TFP, it either becomes part of DFP or it was already in DFP. So, if the pattern

p from DFP is not in TFP, this can only be the case if a hash collision increases a previously encountered pattern which became part of TFP and pushes p from the top-N. This means for p there should be an overestimated value in TFP.

- *contained in TFP*
  This means the frequency of p in TFP is higher than the frequency of p in DFP. Since a Count-Min Sketch never underestimates values, the frequency of p in DFP is either correct or overestimated. Since the value of p in TFP is higher than the value in DFP, it must have been overestimated in TFP.

Therefore, if an overestimation is present in DFP which is not present in TFP, one can conclude that, not only pattern o, but also all patterns below o in the top-N of DFP are overestimations in TFP.

So for each overestimation in DFP holds that it has at least one overestimation in TFP. Therefore a single threaded execution of DFP gives the N most frequent patterns, with no larger error than observed in the TFP approach.
Q.E.D.

Proof II
Proof that a distributed variant of DFP does not perform worse than a single threaded DFP variant.

In a distributed execution each thread on each of the nodes in the cluster performs the single threaded DFP variant. According to the algorithm specification, the NetFlows are distributed over the different threads based on the source IP contained in the incoming NetFlow, resulting in the NetFlows for a specific source IP to always be processed by the same streaming pipeline in the same thread. The patterns generated from each NetFlow all contain the source IP address as first value. Therefore, no two threads will ever count the same pattern. Hence, the top-N frequent patterns that are outputted by each thread will be disjoint sets of pattern frequencies. Since the distributed execution uses a Count-Min Sketch for each thread, the overestimations due to hash collisions in the distributed approach are less or equal to the overestimations for a single threaded execution, which uses a single Count-Min Sketch. So each thread outputs a disjoint subset of frequent patterns containing frequency values that are at least as accurate as the single threaded DFP variant. Therefore, the distributed DFP does not perform worse than a single threaded DFP.
Q.E.D.

Since it is proven that a single threaded DFP does not perform worse than TFP and it is proven that the distributed DFP does not perform worse than the single threaded DFP, one can conclude: The distributed implementation (DFP) gives the N most frequent patterns, with no larger error than observed in the trivial single threaded approach (TFP).

3.4.3. Apache Flink implementation

Similar to the Heavy Hitters implementation, the Frequent Pattern implementation should also consist of two consecutive steps. The counting of patterns can be executed in parallel, but the aggregation should be performed using one Priority Queue structure in a single thread.

Step 1: distributed pattern counting

Since NetFlow patterns of different sources can be contained within a single NetFlow JSON object, again a *FlatMap* operation is needed, creating a stream of all individual NetFlows. Next, a *KeyBy* operation groups all NetFlows over the different threads, based on their source IP. Using a *TimeWindow* operation prevents the algorithm from continuing forever. Instead, the frequent patterns are computed for regular intervals. In the last operation of step 1, a reduction function is executed on the streams of each source IP. In this reduction function, the NetFlow is consumed and a series of patterns are extracted from the flow. Example are: <sourceIP, destinationIP>, <sourceIP, sourcePort> and <sourceIP, destinationIP, sourcePort>. For each of the extracted patterns, the corresponding count is increased in the Count-Min Sketch. If one of the currently consumed patterns has become more frequent than the least frequent pattern stored in the priority queue, the queue will be updated to contain the new frequent pattern. The first step of the Frequent Pattern algorithm is shown in Pseudocode 3.

```
DataStream<FrequentPatternNetFlow> hostPatterns = netflowStream
.flatMap(in, out) {
  for (FrequentPatternNetFlow flow : in.dataset) {
    out.collect(flow);
  }
}
.keyBy("srcIP")
.timeWindow(Time.seconds(10))
.reduce(rollingCount, incomingNetflow) {
```

```
    rollingCount.addPatterns(incomingNetflow);
    return rollingCount;
  });
```

Pseudocode 3: The first step of the Frequent Patterns implementation in Apache Flink. A *FlatMap* operator maps bundles of NetFlows into a stream of all individual NetFlows. With a *keyBy* operation, the NetFlows are divided into a stream for each source IP. Using the *TimeWindow* operator, the subsequent step is performed for all flows that arrive within 10 seconds. The final step is a *Reduce* operator which updates the Count-Min Sketch and a Priority Queue of the current top-N patterns per host.

Step 2: computing the overall top-N patterns

The result of the first step is a stream with for each host the top-N patterns that have been observed. The second step should convert this into an overall top-N of patterns amongst all hosts. The stream resulting from step 1 receives a burst of elements as each time window expires and the reduction operation finishes. To convert results of each time window of step 1 into an overall top-N list, a similar sized time window should be applied in step 2. Similar to the Heavy Hitters implementation, this requires a *WindowAll* operation, since the computation needs to be performed on all data in a single stream. The last operation is a *Reduce* operation, which processes the top-N patterns for each host within the current time window into an overall top-N. This is done by maintaining an overall priority queue that is updated on processing a pattern from the top-N of a certain host that is more frequent than the least frequent pattern of the current overall top-N. All operations of the second step of the frequent patterns algorithm are shown in Pseudocode 4.

```
DataStream<FrequentPatternNetFlow> topN = hostPatterns
  .windowAll(TumblingEventTimeWindows.of(Time.seconds(10)))
  .reduce(rollingTopN, newHostTopN) {
    rollingTopN.mergeTopN(newHostTopN);
    return rollingTopN;
  });

topN.print();
```

Pseudocode 4: The second step of the Frequent Patterns implementation. The top-N patterns for each source IPs, arriving in a 10 second time interval, are combined into a single stream using the *WindowAll* operator. Next, a *Reduce* operation is performed, which combines the incoming Priority Queues one by one. A local Priority Queue is updated with patterns of the incoming queue in case it contains patterns that are more frequent. Eventually, the contents of the final queue is printed.

3.5. Frequent Sequences

Learning Frequent Sequences is another useful technique for better understanding network behaviour and can also be employed to detect anomalies in network traffic. In contrast to frequent patterns it does not highlight the frequency of traffic parameter combinations, instead it focuses on how likely it is to encounter a sequence of traffic patterns. A traffic sequence can be represented as a transition from a current state to a next state. The possible states can be defined in various ways, for example based on the number of packets or bytes that are transferred. These states can be assigned to individual hosts, or to traffic channels using combinations of IP addresses. This section introduces a scalable implementation that learns frequent traffic sequences using Markov Chains on traffic channels. This problem has characteristics very similar to learning State Machines, so this problem is the last related problem being explored, forming the basis for solving the State Machine learning problem.

3.5.1. Existing approaches

For mining frequent sequences a similar approach can be used as with mining frequent patterns. Instead of storing the encountered NetFlow parameter combinations in the FP Tree, each NetFlow will be assigned a classification and the patterns of subsequent classifications are stored in the FP Tree. This approach is proposed in the PrefixSpan algorithm [61]. Other approaches for mining frequent sequences are generating and counting n-grams [82] and learning markov chains [47]. Of the later, only few examples are presented in literature. However, Markov Chains is a powerful approach requiring only few operations per encountered element and is relatively easy to execute in parallel. Therefore, this section explores a scalable implementation that learns Markov Chains for mining frequent patterns.

3.5.2. Proposed Streaming Approach

As mentioned before, Markov Chains represent sequences as state transitions. Therefore, there needs to be a mapping of NetFlows to states in order to compute the frequency of transitioning from one state to another state. The appropriate way to map NetFlows to states depends on the use case. In this example, the flows are mapped into three categories based on the total number of bytes transferred in the packets summarized by the flow. The categories are LOW, MEDIUM and HIGH, using the thresholds: 200 bytes, 500 bytes and more than 500 bytes respectively. These values are selected for testing purposes. Subsequent elements should be mapped to the corresponding type of state, to obtain sequences of states. For small numbers of possible state combinations, like in this example, it is feasible to maintain a counter for each occurring transition. By counting all occurring state transitions in a specific time window, the frequency of each transition can be computed. Using a matrix with all possible states on the horizontal and vertical axis, the frequencies for each transition can be added to the corresponding cells. This results in a Markov Chain solving the frequent sequences problem by offering insights into how frequent each transition (i.e. sequence) is.

3.5.3. Apache Flink implementation

similar to the previous two implementations, the incoming bundled NetFlows should first be split up into individual NetFlows before they can be distributed across nodes or threads. This is again realised using a *FlatMap* operation. In preparation to learning State Machines, this example focusses on traffic channels instead of single hosts, so a *KeyBy* operation on IP pairs is performed next. To obtain results in regular intervals, instead of at termination of the stream, again a *TimeWindow* operator is applied. In the final operation a *Reduce* is performed, which learns the Markov Chain on elements of the intervals of the distributed streams per IP pair. In the *Reduce* operation, NetFlows of an IP pair are consumed one by one by an object that accumulates all data of this IP pair. This object should maintain the state corresponding to the previous NetFlow in order to observe transitions from one state to another. All occurring transitions need to be counted. By low number of possible states, the number of possible transitions are limited and hence the transitions can be simply counted using a HashSet. On printing the results, the HashSet of counts can be transformed into a matrix of frequencies resulting in a matrix representation of a Markov Chain. The frequent sequence implementation is shown in Pseudocode 5.

```
DataStream<FrequentSequenceNetFlow> hostSequences = netFlowStream
  .flatMap(in, out) {
    for (FrequentSequenceNetFlow flow : in.dataset) {
      out.collect(flow);
    }
  }
  .keyBy("IPPair")
  .timeWindow(Time.seconds(10))
  .reduce(rollingCount, newNetflow) {
    rollingCount.processFlow(newNetflow);
    return rollingCount;
  });

hostSequences.print();
```

Pseudocode 5: A frequent sequence implementation in Apache Flink. A *FlatMap* operator maps bundles of NetFlows into a stream of all individual NetFlows. A *KeyBy* on IP pairs divides the stream into separate streams for each pair of IP addresses. Using the *TimeWindow* operation, the incoming data is divided in chunks of 10 seconds. A *Reduce* operation is applied to each chunk in parallel, mapping the NetFlow to a state transition and increasing the count. Next, the Markov Chain for each host is printed.

3.6. Generalized Job Structure

From exploring the data mining problems throughout this chapter and by designing streaming solutions to solve them, a basis can be formed for solving the main problem of this thesis. Learning State Machines on data streams is especially similar to the Markov Chain problem, as both map elements to states and observe state transitions. The implementations throughout this chapter show a number of similarities in the approaches to implement the algorithms using Apache Flink, which can be generalized into a generic job structure. When processing NetFlows it is important to first generate a single stream consisting of all individual flows, using the *FlatMap* operator. In order to distribute workload across nodes in a cluster or to different threads on a node, a *KeyBy* operation should be applied. This divides a single stream into meaningful parallel streams, based on a provided key. In the NetFlow case, this can be for example a single source IP address or IP pairs. Next, a *TimeWindow* needs to be applied to specify over which finite interval the data should be combined. Finally, a *Reduce* operation performs the actual algorithm steps on the parallel streams of similar elements. The final operation is outputting the results to console or any Flink Sink of choice.

In the next chapter an algorithm is designed for learning State Machines on data streams. To ensure the algorithm is scalable and performs well in high throughput environments, the algorithm will be implemented in Apache Flink based on the general approach that is formed in this chapter.

4

Learning State Machines on data streams

This chapter describes the design of a method for learning State Machines on data streams. It describes in detail, to our knowledge, the first streaming algorithm for learning State Machines as well as the various design decisions made in creating the algorithm. The algorithm is based on the approach taken in the Blue-Fringe algorithm, which will be explained first in section 4.1. Section 4.4 describes the approach to evaluate the correctness of the method, which will be used throughout the iterative design process. Next, in section **??** the experimental setup that has been defined earlier in section 3.2 is revised to allow learning State Machines on streams of training and test sequences. The chapter continues with a high level design of the algorithm in section 4.2, designing the overall approach of the streaming algorithm. Section 4.3 implements the algorithm in Apache Flink, which will be used to test design decisions and settings in the further development of the algorithm in section 4.5. The chapter concludes in section 4.6 with a validation of the correctness of the final algorithm.

4.1. Original Blue-Fringe

As mentioned before in section 1.1.4, the current State of Art in State Machine learning methods rely on the principles of the Blue-Fringe algorithm. The streaming algorithm that will be designed in this study will therefore adopt principles of this approach.

The original Blue-Fringe algorithm [50] starts by creating a Prefix Tree Acceptor on all input sequences. A Prefix Tree Acceptor is the smallest tree structure that captures all sequences that are accepted by the language it describes [95]. Each node in the tree represents the prefix of the full sequences that are stored in the leaves of the tree. An example of a Prefix Tree is illustrated in figure 4.1. By regarding the leaves as accept states for all possible accepted sequences in the model, the Prefix Tree can be called a Prefix Tree Acceptor.

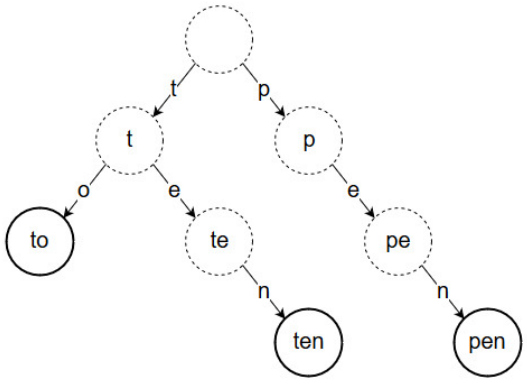


Figure 4.1: An example Prefix Tree Acceptor with each state representing the prefix of elements stored in the leaves. The tree contains the elements *to*, *ten* and *pen*. Solid lines indicate accepting states.

In the Blue Fringe algorithm, after a Prefix Tree Acceptor (PTA) has been constructed on all data sequences, the PTA is then reduced into a DFA by merging nodes in the tree. This is done using a strategy named Red-Blue. This strategy decides which states of the PTA should be merged. In the Red-Blue strategy, all red states are considered definitive

states belonging to the final solution. All blue states are connected to red states, but are still under evaluation and not yet part of the final solution. In the first step of the Red-Blue strategy, the root of the PTA is colored red. Its direct children are colored blue and all other nodes remain uncolored. The second step is an iterative step in which all blue states are evaluated. Each blue node is compared with all red nodes by computing a score of how similar the sub trees are. If there exists a blue node that cannot be merged with any red node, the blue node is colored red. Else if no blue node can be promoted, the highest scoring red/blue merge is performed. If a change has been made to the DFA, all uncolored states connected to red states are again colored blue. This concludes one cycle of the iterative step, which is now performed again. The iterative step is repeated, until all blue nodes are either merged with a red node or promoted to become a red node itself. Hence, the final result is a DFA containing only red nodes. This original Blue-Fringe approach is described in algorithm 2, based on the work from Sebban et al. [70].

Algorithm 2 Blue-Fringe

```
1:  data ← all data elements
2:  A ← generatePTA(data)
3:
4:  while blue state exists do
5:      P ← ∅
6:      M ← ∅
7:      for all blue state B do
8:          no_merge_found ← true
9:          for all red state R do
10:             if mergable(A, B, R) then
11:                 no_merge_found ← false
12:                 M ← M ∪ {(B, R)}
13:             end if
14:         end for
15:         if no_merge_found then
16:             P ← P ∪ {B}
17:         end if
18:     end for
19:     if P ≠ ∅ then
20:         A ← do_best_promotion(A, P)
21:     else
22:         A ← do_best_merge(A, M)
23:     end if
24: end while
25:
26: return A
```

4.1.1. Limitations

The original Blue-Fringe algorithm has a number of limitations preventing it from being employed to learn State Machines on infinite data streams. The most obvious one is that the PTA needs to be fully constructed before the Red-Blue strategy is applied. On data streams; however, there is no pre-determined last element, so on an infinite data stream the Red-Blue strategy will never be applied. Also, creating a full PTA before merging the states results in large memory footprints since it stores all unique occurring prefixes. In the worst case this equals all possible symbol combinations. Furthermore, creating a full PTA makes the Red-Blue merging strategy a computationally expensive task as each node should be compared with all red nodes during each iterative step of the Red-Blue strategy. This confines the scalability of the algorithm.

The algorithm proposed in the next sections removes the step of learning a full Prefix Tree Acceptor. It uses approximations to reduce the memory footprint and computation complexity. It also learns a solution iteratively, allowing intermediate results, desirable when learning on infinite streams.

4.2. High level design

The algorithm that is proposed in this thesis is a distributed streaming State Machine learning algorithm based on the concepts of the Blue-Fringe algorithm. Since the proposed algorithm borrows principles of the Blue-Fringe algorithm, it will be called the Streaming Blue-Fringe. In the previous section we saw that the original Blue-Fringe first computes a Prefix Tree Acceptor (PTA) on all data elements. Since this is not feasible on infinite data streams, the

Streaming Blue-Fringe will not create this data structure. Instead, an iterative learning approach is proposed.

Iterative learning

In order to learn State Machines on infinite data streams, while adopting the powerful Red-Blue strategy, it is essential to continuously perform the merging strategy. This is important because in data streams there is no pre-determined end of the stream. Hence, the algorithm will build the State Machine from the bottom up instead of storing all data in an intermediate structure and reducing it into a State Machine. Since each State Machine has at least one state, it will start by creating a single root state. Based on the symbols that are consumed state transitions to additional states can be added from the root state. This gradually creates an entire State Machine. However, since we decided not to store the entire stream in a PTA, the algorithm does not know directly on adding a state whether it can be merged. Therefore, each state should have counted a certain amount of elements that have passed through the state. By storing these elements we have a summary of the data passed through the state. In section 2.3 we have seen that Count-Min sketches are an excellent solution for counting large number of elements efficiently. Hence, they will be used to maintain a summary of the elements in each state. This summary can then be used to compare different states, in order to decide which states to merge. To achieve this, similarly to in the Blue-Fringe, red and blue states are considered. The red states form definitive state and blue states are currently accumulating data. To keep the learning process efficient, around a red state only one layer of blue states can be added. With this in place, in order to start the learning process the algorithm should start with a red-colored root state. Now data has to be consumed by the algorithm.

Sequences

State Machines are usually learned on a set of finite strings, with each of these strings composed of symbols. In data streams; however, there are no finite strings. The stream forms one infinite string of symbols. Therefore, in the Streaming Blue-Fringe needs a way to divide this infinite stream of symbols into strings or sequences in order to learn a State Machine. One common approach is to inject stop symbols into the stream instructing the algorithm to combine the previous symbols as one sequence and to start collecting new symbols for constructing the next sequence. This approach is useful when you have control over the stream and have meaningful bundles of streaming data. However, in real-world applications, such as the use case described in chapter 5, this is often not the case. Hence, another approach is required. An alternative solution is the use of sliding windows. This divides the stream into equally-sized strings, forming the symbol sequences that can be used in the learning algorithm.

Futures

With an infinite stream of symbols now split into fixed-sized sequences, the next step is to process the sequences one by one. In order to accumulate data in each of the states, which is needed to determine the merge or coloring decision, the sequences should traverse through the State Machine. The root state forms the entry point of the State Machine, so each sequence starts in the root state. The root state now stores the sequence in its Count-Min sketch. Next, the sequence should transition to a neighbouring state. However, not each state should count a sequence. The sequence should make transitions based on the symbols stored in the sequence. Since the sequence originates from a sliding window, the subsequent sequence will not contain the first symbol but it would contain the second and subsequent symbols. Hence, a sliding window sequence captures a current symbol and a series of future symbols that describe the prefix of the next sequence. If the algorithm uses the first symbol to determine to which state a transition should be made. Then the next sequence (starting with the second symbol of the current sequence) could continue the path through the State Machine this sequence has started. By repeating this process, each state receives elements that describe the path that the next sequences would take. Hence, the sequences can be called futures.

Instances

In the current version, each future follows a transition in the State Machine or creates a transition if it is at a red state and no transition with the current symbol exists. However, when starting to learn a State Machine, only one transition can be made. Then the next future ends up in a blue state and cannot move further. Therefore, each new future starts not only were the previous future left off, but also in the root state. As a result of this addition the process of traversing the State Machine and counting passing futures can continue. As a consequence of this decision, a future can start in multiple states. Each state a future starts in, is called an instance.

State merging

As mentioned before, in blue states futures are counted until a certain threshold. This threshold is called the significance boundary. Once this threshold is reached, the sketch is considered to have accumulated enough data to allow statistically accurate comparisons against other sketches. Which value forms an effective threshold will be determined through extermination, later in section 4.5.2. In the sketch comparisons, the sketch of the blue state is compared against the sketches of all current red states. A method suitable for sketch comparisons is considered later in section 4.5.1. If two sketches are below a certain threshold, the similarity threshold, the blue state can be merged with the corresponding red state. In case multiple red states match with the blue state, the merge with the highest similarity is performed.

4.2.1. Pseudocode

With the high level design created in the previous section, this section focusses on the implementation of the Streaming Blue Fringe. A pseudocode of the most important parts of the learning method is presented in Algorithm 3. An overview of the concepts and variables used in the algorithm are listed in table 4.1. This section will now walk through the algorithm step by step.

First three variables are initialise based on input arguments that need to be supplied to the algorithm. These are *future_size*, *similarity_threshold* and *significance_boundary*, as defined in table 4.1. Next, a number of additional variables are initialised. A variable for maintaining the current *symbol* and *future* queue are created. The *future* queue stores the sliding window containing the current sequence of symbols. The lists *instances* and *reds* contain respectively the states at which the current sequence will be evaluated and the red-colored states. The learning process always starts with a root state, so it is created, colored red and added to *reds*.

After instantiation of all variables, the *Consume* procedure is defined. This procedure is called for each element in the stream. It updates the current sliding window sequence in the first two operations. Next, a check is performed to confirm that already enough elements are consumed to start the learning process. If that is the case, one step of the learning method can be executed. A single step of the learning algorithm is described in the *evaluateInstances* procedure.

The *evaluateInstances* procedure evaluates the current future in each instance. The list of instances contains all states that the previously consumed symbol did transition to, together with the root state which is added in line 17. The lines 24 till 41 evaluates each of the instances. First, in line 25, the Count-Min sketch of the current instance is updated, by increasing the counts for each hash function of the current future. Next, a check is performed to see whether the current instance is a blue state that has accumulated enough data. If that is the case, the most similar state is computed using the *mostSimilar* procedure. If no red state is similar, the instance should be colored red and added to the *reds* list. If a similar state exists, the current blue instance is merged with the similar red state in line 32. Line 33 ensures that after a merge, the evaluation continues in the merged state. In line 35, the current model is published in order to give access to intermediate solutions. Line 37 is reached for all instances and checks whether the path through the State Machine has become too long. This prevents a too large number of instances as a result of loops in the State Machine. In case the maximum path length is not reached yet, lines 38 and 39 prepare the next instance by making a transition based on the current symbol. Finally, line 42 overwrites the current list of instances with the new list, finishing one evaluation.

Concept	**Explanation**
Red state	A state that is part of the final State Machine.
Blue state	A temporary state that is not yet part of the final State Machine. It has not seen enough sequences to determine whether it should be an actual state or be merged with an existing red state.
Future	A sequence (or sliding window) of symbols that indicate the transitions made from the state it is currently evaluated in.
Future size	The length of the sliding window of symbols.
Instance	A state in which the next future will be evaluated.
Evaluation	A future is evaluated in a state. This means counting the sequence in the sketch and promoting the state if its sketch has become significant.
State promotion	Coloring a blue state red or merging the state with an existing red state.
Significant sketch	A sketch is significant if it has performed more updates or insertions than the significance boundary.
Significance boundary	The number of updates or insertions that need to be performed in a sketch before the state can be promoted.
Similarity threshold	The lower bound for the similarity between the sketches of two states before they are allowed to be merged.

Table 4.1: Explanations of various variables or concepts used in the Streaming Blue-Fringe algorithm

Algorithm 3 Streaming Blue Fringe

```
1: future_size ← Args.future_size                                  ▷ Initialise the future size
2: similarity_threshold ← Args.similarity_threshold                ▷ Initialise the similarity threshold
3: significant ← Args.significance_boundary                        ▷ Initialise significance boundary
4:
5: symbol ← NULL
6: future ← new Queue(future_size)                                 ▷ FIFO queue of fixed size
7: instances ← new List<State>
8: reds ← new List<State>
9: root ← new State
10: root.colorRed()
11: reds.add(root)
12:
13: procedure CONSUME(element)                                     ▷ Called on consuming a new element
14:     symbol ← getSymbol(element)
15:     future.enqueue(symbol)
16:     if future.size = future_size then
17:         instances.add(root)
18:         evaluateInstances()
19:     end if
20: end procedure
21:
22: procedure EVALUATEINSTANCES
23:     new_instances ← new List<State>
24:     for all instances do
25:         instance.increaseFrequency(future)                     ▷ Increase counters in sketch of this instance
26:         if instance.isBlue() && instance.updatesCount() = significant then
27:             similar ← mostSimilar(instance, reds)
28:             if similar = NULL then
29:                 instance.colorRed()
30:                 reds.add(instance)
31:             else
32:                 instance.merge(similar)
33:                 instance ← similar
34:             end if
35:             root.outputPDFA()                                  ▷ Output the current version of the model
36:         end if
37:         if instance.maxStepsReached() = False then             ▷ Prevent infinite loops
38:             next ← instance.getState(symbol)                   ▷ Get the state in direction of the symbol
39:             new_instances.add(next)
40:         end if
41:     end for
42:     instances ← new_instances
43: end procedure
44:
45: procedure MOSTSIMILAR(state, reds)
46:     highest_similarity ← 0
47:     most_similar ← NULL
48:     for all reds do
49:         similarity ← computeSimilarity(state, red)             ▷ Compute similarity between sketches
50:         if similarity > highest_similarity then
51:             highest_similarity ← similarity
52:             most_similar ← red
53:         end if
54:     end for
55:     if highest_similarity > similarity_threshold then
56:         return most_similar
57:     end if
58:     return NULL
59: end procedure
```

4.3. Apache Flink implementation

The state maintained by the learning algorithm is very complex, as it contains various statistics and the current PDFA structure which are being updated constantly. Furthermore, during the learning process often loops are introduced into the DFA, which would change the execution flow of a distributed algorithm. Therefore, it is decided not to attempt redistributing the state of the State Machine learning process. Instead, the Apache Flink job is implemented in such a way that only different State Machines can be learned distributed across a cluster. As a result, the actual Flink Job managing the complex learning process is quite straightforward.

The Apache Flink job starts with a *KeyBy* operation dividing the elements according to a specified grouping of the elements, which differs depending on the use case. In this example, each incoming element is expected to have a *"groupKey"* attribute, which is used to divide elements into different groups. All resulting streams containing the elements of a group are distributed across nodes of a cluster and threads within a node. Next, on element streams of each group a separate State Machine is learned in parallel, by performing the *Reduce* method on each substream. The steps of the Apache Flink job are illustrated in pseudocode 6.

```
DataStream<StateMachineElement> groupSequences = stream
.keyBy("groupKey")
.reduce(streamingBlueFringe, incomingElement) {
  streamingBlueFringe.consumeElement(incomingElement);
  return streamingBlueFringe;
});
```

Pseudocode 6: An Apache Flink job implementation of the Streaming Blue-Fringe. A *KeyBy* on a key of choice, divides the stream into separate streams each of which will yield a State Machine. A *Reduce* operation is applied to each sub stream in parallel, learning the State Machine on the stream of that group.

4.3.1. Reduce function

Inside the *Reduce* function is where the actual State Machine learning is performed. The steps described in algorithm 3 are implemented in Java and the *consume* method is called for each incoming streaming element. After consuming an element, if it is detected that a transition or the number of states of the current PDFA has changed, the PDFA is stored to disk. As a result, each step of the State Machine learning process is saved on disk. This is important, since it is unknown if the State Machine will receive any additional elements as we are handling a stream of unknown size. Furthermore, this allows the use of intermediate results during the learning process.

4.4. Validation approach

With the high level design completed and the algorithm implementation details determined, we need to focus on an approach for validating the correctness of the leaning method. This is essential for verifying the correctness of both the algorithm and the implementation. Furthermore, it can shed light on possible drawbacks of the used approach, for example regarding the approximations used in the Count-Min sketches.

In 2012 the PAutomaC Probabilistic Automaton Learning Competition was held [91]. The goal of the PAutomaC challenge was to provide an overview of which probabilistic automaton learning techniques works best in which setting and to stimulate the development of new techniques for learning string distributions. The competition consisted of 48 problems in which participants received artificial datasets [26] with training data on which models should be learned. The learned models are tested against a test set for each of the problems. The test and training sets have been generated using the PAutomaC model generator written in Python, which is explained in detail in this paper [92]. The PAutomaC model generator formed the core of this successful competition, hence can be regarded as a reliable tool for model generation. Therefore, it is decided to use models generated by the PAutomaC generator as a ground truth in validating the correctness of models learned by the designed streaming algorithm.

4.4.1. Datasets

The PAutomaC model generator produces a model file as well as a training- and testset with sequences that match with the model. The training- and testset have the following format. The first line contains the total number of lines and the alphabet size, divided by a space. Each of the subsequent lines represent a training or test sequence starting with the number of elements in the sequence, followed by each of the sequence elements. The sequence elements are again divided by a space. This is an example of a portion of training data:

```
50000 2
31 1 0 0 1 1 0 0 0 1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 1 1 0 0 1 0 1 0
3 1 0 1
3 1 0 0
2 0 0
2 1 0
2 1 0
8 0 1 1 0 0 1 1 0
7 1 0 1 0 0 0 0
18 0 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 0
4 1 0 0 0
2 1 0
2 1 0
...
```

To verify the correctness of a learned model, a smaller series of test sequences is generated. The following example is a portion of a test set, in the same format as the training set.

```
2601 4
2 1 2
22 1 2 2 1 1 2 3 2 1 1 2 2 1 3 2 3 1 0 3 2 3 0
6 0 2 1 1 3 0
5 0 2 1 1 2
6 1 2 1 3 1 0
11 3 2 3 3 1 1 0 2 1 1 2
0
5 3 3 1 0 3
4 1 2 2 1
12 1 2 3 2 2 0 0 3 3 3 1 1
22 0 3 0 3 3 3 2 0 2 1 2 1 0 1 2 2 2 0 1 3 3 3
8 0 2 3 3 2 1 2 3
...
```

With each test sequence a probability is given that defines how likely it is for the sequence to occur in the model. These probabilities allow for models that take into account not only whether a sequence fits into the model, but also how likely it is to occur. This allows for validating more powerful models containing more information, i.e. PDFA's instead of DFA's (explained in section 2.5). The probabilities belonging to the test sequences are stored in a solution file with the following format. First the number of probabilities is defined, which equals the number of test sequences. Next, for each sequence the probability is given. The following is an example of the solution file belonging to the above given test file:

```
2601
0.0949664425738
4.56916633809e-10
0.000651491956811
0.00712079113499
0.000783532644979
1.40255660178e-06
0.306137685948
0.000121759919508
0.0183582987577
2.61926288185e-07
8.18568718192e-15
1.04687209632e-05
...
```

The final file is the model file, containing a definition of the model that was used to generate the training and test sequences. It describes the initial states with for each state the probability of a sequence starting in this state. Next, the final states are given, which are accept states in which a sequence is allowed to end. Next, all pairs of state,symbol-combinations are given, describing all symbols encountered in each state together with the probability of a transition with the symbol in this state. Finally, all triples of state,symbol,state-combinations are given, describing the target state of a state transition. Since the model files should be able to represent NFA's, for each transition from a state with a symbol the corresponding probability is given. Since the method designed in this work focusses on PDFA's, the resulting state of a transition is deterministic yielding 1.0 for each transition probability. This is a example of a model specification:

```
I: (state)
(2) 1.0
F: (state)
(1) 0.26977308315
(2) 0.0604117825127
S: (state,symbol)
(0,0) 1.0
(1,0) 0.517967381019
(1,1) 0.482032618981
(2,0) 0.527213405834
(2,1) 0.472786594166
T: (state,symbol,state)
(0,0,1) 1.0
(1,0,1) 1.0
(1,1,2) 1.0
(2,0,1) 1.0
(2,1,0) 1.0
```

4.4.2. Generated datasets

For steering decisions throughout the design process and to validate the final performance, two groups of datasets are created. A design dataset is constructed that consists of 15 different models, generated by PAutomaC. 15 different models are used in order to test design decisions with models of different sizes and complexities. This design dataset is used in section 4.5. Also a validation set is created, containing datasets of 10 different models. This set is used in section 4.6 to evaluate the performance of the final algorithm on another dataset than is used during the design. This prevents optimization of the method to one specific set of models.

Design dataset

For steering decisions, a dataset of 15 models are generated that capture a range of different characteristics. Among the dataset are models with alphabet sizes between 4 and 10 and number of states between 4 and 18, in order to test design decisions on basic and complex models. Furthermore, the transition sparsity and symbol sparsity varies using the bounds described in [91]. This helps to generate both models that are quite sparse in the number of connections, as well as models that have more intertwined connections between its states. Table 4.2 describes the dataset used in the design process.

Label	Number of states	Alphabet size	Transition sparsity	Symbol sparsity
Set 1	4	4	0.248	0.430
Set 2	5	4	0.200	0.312
Set 3	6	4	0.199	0.311
Set 4	7	4	0.179	0.789
Set 5	8	5	0.135	0.585
Set 6	9	5	0.132	0.612
Set 7	10	5	0.143	0.542
Set 8	11	6	0.191	0.348
Set 9	12	6	0.198	0.485
Set 10	13	6	0.181	0.570
Set 11	14	6	0.124	0.222
Set 12	15	7	0.070	0.617
Set 13	16	8	0.105	0.749
Set 14	17	9	0.197	0.616
Set 15	18	10	0.159	0.412

Table 4.2: Characteristics of the models used for evaluating design decisions

Validation dataset

For validation of the final learning method, a dataset of 10 models have been generated. The number of states range between 4 and 22 and the alphabet size ranges from 5 and 15, in order to evaluate the learning method on a large variety of models. Similarly to the design dataset, the transition and symbol sparsity is also varied in order to capture all kinds of models. The characteristics of the models used for evaluation are presented in table 4.3.

Label	Number of states	Alphabet size	Transition sparsity	Symbol sparsity
Set 1	4	5	0.210	0.622
Set 2	6	6	0.179	0.440
Set 3	8	7	0.152	0.656
Set 4	10	8	0.142	0.607
Set 5	12	9	0.091	0.438
Set 6	14	10	0.188	0.665
Set 7	16	11	0.156	0.263
Set 8	18	12	0.171	0.717
Set 9	20	13	0.065	0.339
Set 10	22	14	0.060	0.650

Table 4.3: Characteristics of the models used for validating the designed learning method

4.4.3. Performance computation

In order to validate the correctness of the learning process, models will be learned with training data before they are evaluated against a test dataset. In order to evaluate the performance of a learning method, the Kullback-Leibler divergence measure will be used. The precise workings of this KL divergence measure has been explained earlier in section 2.6. In short, it describes how much more bits are required to describe samples from a true distribution using the modelled distribution as represented by the learned PDFA. So, it is a measure of how much a derived model diverges from the true model.

Each of the by PAutomaC generated models contains a test set with a number of sequences which are passed through the learned model, together with a solution file describing the probabilities of each test sequence. Since each state transition contains its chance of occurring, the total probability for each test sequence can be computed using the learned model. Using the list of probabilities from the solution file and the probabilities of the sequences in the learned model, the KL divergence can be computed. At least almost, because first one problem needs to be addressed.

Smoothing

By taking a closer look at the formula for KL divergence in equation 2.4, one can note that the divergence becomes undefined if a sample from the true model is not accepted by the derived model. To cope with this problem, it is common to smoothen the probabilities by giving a very small chance for samples that are actually of zero probability [63]. In this thesis we will assign each unaccepted sequence a small probability of 10^{-15}. Next, all probabilities are normalized to ensure the sum of probabilities remains 1.

Combined performance

Since 15 models are used throughout the design process and 10 models are used to validate the final learning method, the performance of each model within a set needs to be combined into one performance score. Since we decided that the performance on basic models is equally important as the performance on the more complex models, the KL divergence on all models will be averaged. This yields a combined performance score on all models. The smaller the score, the closer all learned models are to the actual models and hence, the better the learning method.

4.5. Iterative design steps

Within the high level design are still a number of components for which the optimal strategy needs to be determined by experimentation. For example an appropriate method for comparing states, the state similarity threshold, the future size, the significance boundary and the sketch size. For each decision an initial guess can be made, but the best setting should be decided based on performance while testing the algorithm. In this section iteratively each decision is determined by evaluation of the different options. First of all, the method of state similarity computation is determined in section 4.5.1. In this section four similarity metrics are considered, of which three are tested together with various similarity thresholds and for different future sizes. Next, different significance boundaries are tested. In the initial guess, the significance boundary is set to 500 sequences in order to make sure that enough data has been accumulated to get a statistically accurate sketch comparison. In section 4.5.2 this is reduced in order to be able to learn state machines on smaller datasets. Finally, the size of the Count-Min Sketch is determined. The initial settings are a width of 500 with 50 hash functions, in order to minimize the chance of hash collisions while making the other decisions. In section 4.5.3 this is reduced in order to make the learning method more memory efficient.

4.5.1. State similarity metric

In each state the Count-Min Sketch is maintained, approximating the frequencies of all sequences passing through a state. Once the state has accumulated enough data to get a statistically accurate comparison of the sketches, a state can be evaluated by comparing its sketch with sketches of red-colored states. The sketches are represented as a matrix of values that are either zero or larger than zero. For comparing sketches, a large number of comparison methods are available. Three methods are considered, of which two are compared in terms of performance. It can be seen from the comparisons, that the state similarity metric does not have a very large impact on the performance. The following metric are evaluated:

1. **KL divergence**
   Since the values in a sketch represents a distribution, the KL divergence that is used to compare State Machines can also be applied to compare single states within a State Machine.
2. **SSdeep**
   SSdeep is a fuzzy hashing that tries to match inputs that have homologies. Such inputs have sequences of identical bytes in the same order, although bytes in between these sequences may be different in both content and length. It could be a solution since it is popular in similarity hashing and works on binary formats, which sketches usually are. However, after closer inspection it performs identical matches on sub sequences whereas sketches can be very similar while having small differences in their values. The SSdeep hashing does not take into account close similarities in distributions where the values are similar but not identical matches. It only regards exact matches and thus is omitted from testing.
3. **Cosine Similarity**
   A popular method that does not perform identical sub sequence matches, but instead takes into account the distribution of values is a cosine similarity. Sketches can be compared by computing a cosine similarity on their vector representations. This computes the angle between two vectors. The smaller the angle, the more similar both vectors are. This is insusceptible to the issue of low similarities where the vector values slightly differ. A slightly different value for the same index in two vectors means the angle is not equal, but it is still very close.

Selection of similarity method

The KL divergence and Cosine Similarity metric are both added as state similarity metrics in the Apache Flink implementation. Both state similarity metrics are tested individually on all design datasets listed in table 4.2. In each run a different combination of similarity threshold and future size is used. In a run for a given parameter combination, all

15 models are learned and the KL divergences is calculated between each learned models and its ground truth. By averaging the KL divergence for each of the 15 learned models, an average KL divergence performance is obtained for each parameter combination.

Figure 4.2 shows the average KL divergence compared for different Cosine Similarity thresholds on the horizontal axis. Each line indicates another future size. As explained in section 2.6, a lower KL divergence means a better fit between the learned models and the actual models. From figure 4.2 one can conclude that a higher future size yields a higher average KL divergence between the 15 learned models and the ground truth from the datasets of table 4.2. The best performing combination seems to be a Cosine Similarity threshold of 0.95 using a future size of 2.

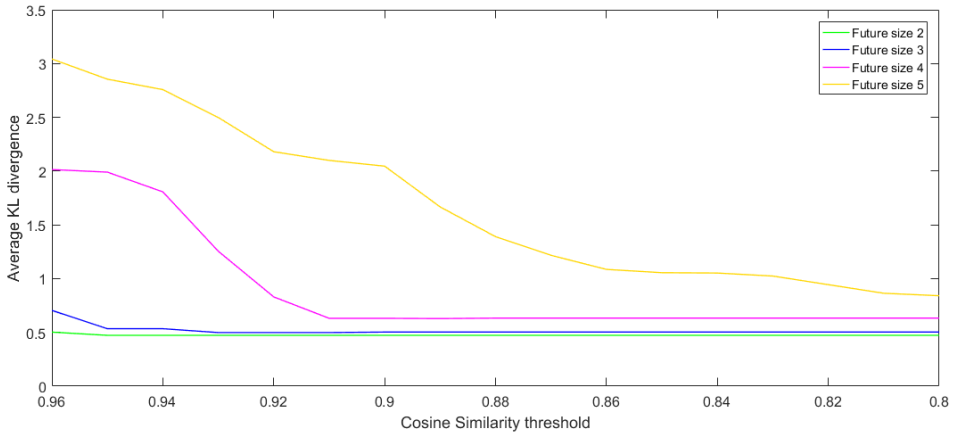


Figure 4.2: The average KL divergence indicating the performance on the set of design models for different Cosine Similarity thresholds and for different future sizes. A Cosine Similarity threshold of 0.95 with a future size of 2 yields the best performance.

Figure 4.3 shows the average KL divergence performance for different KL divergence thresholds and different future sizes. The experiments are performed similarly to the steps taken in obtaining the results of figure 4.2. This graph shows that a KL divergence threshold of 0.15 with a future size of 2 yields the best performance. In terms of average KL divergence, there is little difference between using a Cosine Similarity measure and using the KL divergence for sketch comparisons. However, since a Cosine Similarity does not require smoothing, it is more straightforward and hence the metric of choice.

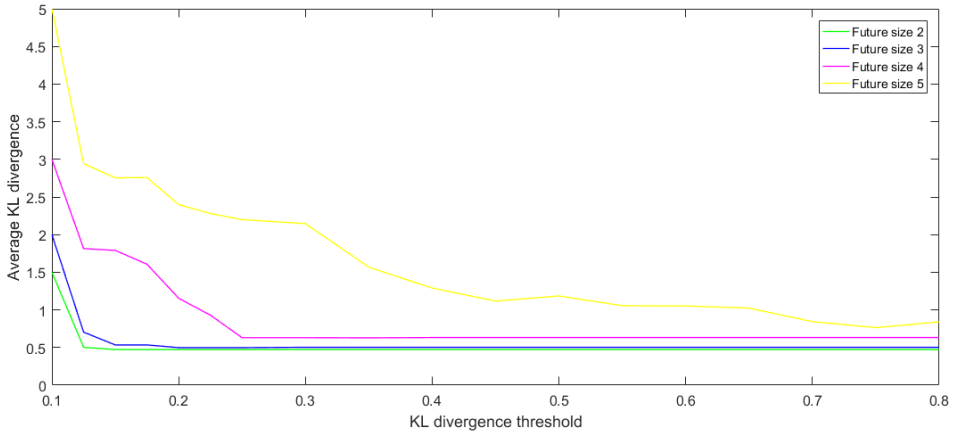


Figure 4.3: The average KL divergence indicating the performance on the set of design models for different KL divergence thresholds used for state merging. Each line shows the performance for a different future size. A KL divergence threshold of 0.15 with a future size of 2 yields the best performance.

4.5.2. Minimizing the significance boundary

The significance boundary is the number of futures that should pass through a state, before the state is either promoted to become a red state or merged with an existing red state. A higher significance boundary yields better performance since more sequences gives a more accurate approximation of the distribution of sequences passing through the state. More accurate approximations of distribution gives a more accurate computation of state similarity. However, a larger significance boundary means more samples are needed before a state becomes definitive. This means longer learning time and less detailed State Machine if little samples are being available. Therefore, the significance boundary should be as small as possible while still allowing accurate state comparisons. First of all we confirm that for small significance boundaries, still a future size of 2 outperforms larger future sizes. The performance for various future sizes with a significance boundary of 50 is shown in figure 4.4. Therefore, further tests on significance boundary will be only be performed with a future size of 2.

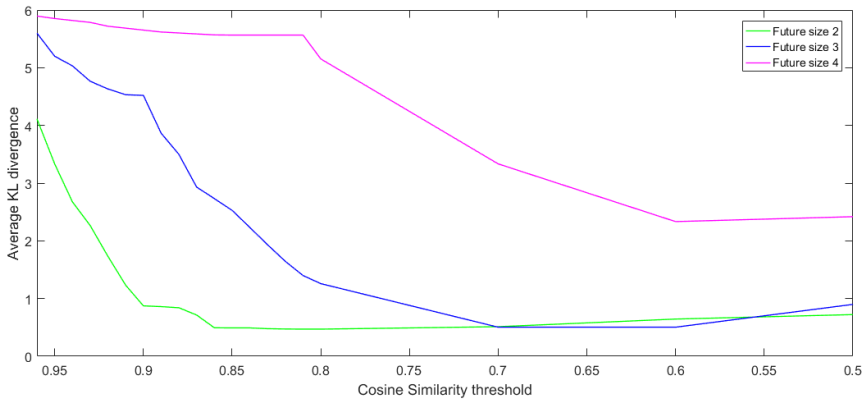


Figure 4.4: The average KL divergence indicating the performance over different Cosine Similarity thresholds. Each line represents the relation for a different future size. A Cosine Similarity threshold of 0.83 with a future size of 2 yields the best performance. This graph indicates that also for low significance boundaries, in this case 50, the future size of 2 gives the best performance.

Next, the average KL divergence is computed over the 15 design models from table 4.2 for different significance boundaries. Because figures 4.2 and 4.4 show that the optimal Cosine Similarity threshold varies when the significance boundary changes, it is decided to plot the performance for different Cosine Similarity values. Each line will represent another threshold. The results are depicted in figure 4.5. Note that the 15 datasets used in this experiment have alphabet sizes ranging from 4 till 10. Hence, the average KL divergence performance captures the performance for multiple alphabet sizes.

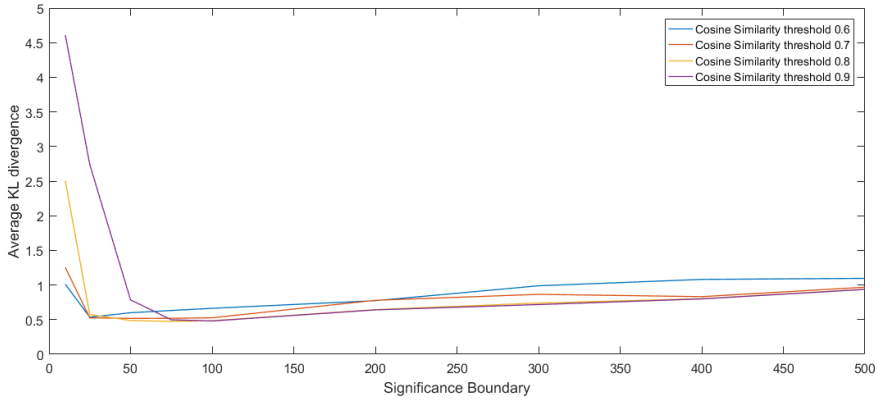


Figure 4.5: The average KL divergence indicating the performance over different significance boundaries. Each line represents the relation for a different Cosine Similarity threshold. The graph indicates that around 50 is the optimal significance boundary and that a lower threshold allows a lower significance boundary.

Counterintuitively, this graph shows that a larger boundary yields a slightly worse performance. This could be explained by the possibility that the datasets contained too little training data to learn a full model when using a high significance boundary. However, in general one can say that the significance boundary does not have a large impact on performance, unless it is too small. In that case there is not enough data stored in the sketches, to make a statistically accurate comparison. This is shown by the steep lines on the left side of the graph. The best performance turns out to be around a significance boundary of 50. Another observation one can make from the graph, is that for lower significance boundaries a lower Cosine Similarity threshold performs better. Overall this graph shows positive results, because it is desirable to choose a low significance boundary. This means you need less training data in order to train a model, which is beneficial if only small amounts of data are available.

4.5.3. Minimizing sketch size

Each state contains a Count-Min Sketch that approximates the number of occurrences for each sequence in a memory efficient way. The Count-Min Sketch consists of a number of hash functions with for each function a vector of a specific length. With a larger number of hash functions and a larger vector length, the chance of hash collisions reduces resulting in more accurate sequence counts. However, this increases the memory footprint, so we would like to minimize the sketch size. Figure 4.6 shows how the performance varies over different sketch sizes. On the horizontal axis, the sketch width is shown. Each line represents another sketch depth.

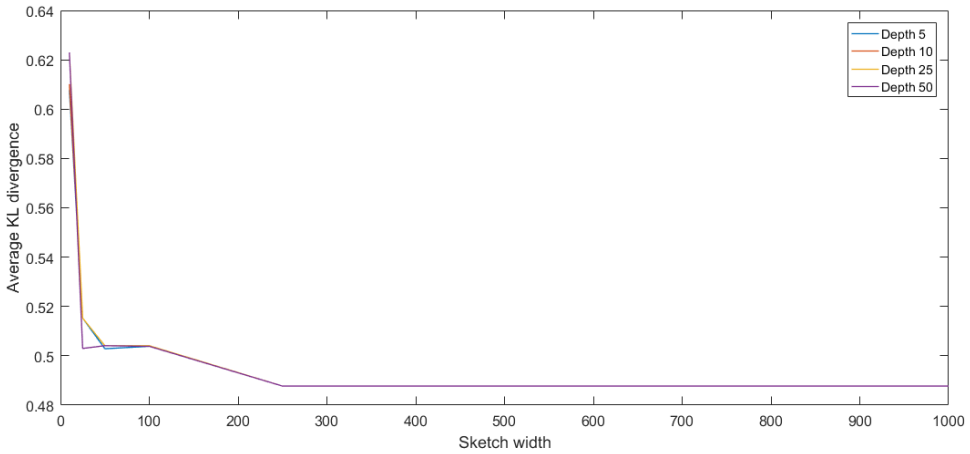


Figure 4.6: The performance for different Count-Min sketch widths and depths. Sketches with a width larger than 25 are usable.

This graph gives a general idea of the relation; however, distinctions between sketch depths are hard to read from the graph. In table 4.4 the values from the previous graph are represented in a more readable format. It is important to note that the actual sketch size in memory equals to the sketch width times the number of hash functions, i.e. the sketch depth. With this in mind, a sketch width of 250 and a depth of 5 is the smallest sketch that still gives an 0.488 average KL divergence.

	50	**25**	**10**	**5**
1000	0.488	0.488	0.488	0.488
750	0.488	0.488	0.488	0.488
500	0.488	0.488	0.488	0.488
250	0.488	0.488	0.488	0.488
100	0.504	0.504	0.504	0.503
50	0.504	0.504	0.504	0.503
25	0.503	0.515	0.515	0.515
10	0.623	0.623	0.610	0.608

Table 4.4: The average KL divergence performance for different sketch sizes. Each column represents a sketch depth and each row represents a sketch width. A smaller value indicates a better performance.

4.5.4. Final design

To summarize the findings of section 4.5, we first saw that a cosine similarity could obtain the same levels of performance as the KL divergence state similarity metric. Since the KL divergence cannot handle zero probabilities and thus requires smoothing of the vectors, the Cosine Similarity is a little more efficient to calculate. Hence, the Cosine Similarity is selected for comparing state similarity. Figures 4.2, 4.4 and 4.3 show that a future size of 2 gives the best overall performance. Figure 4.5 shows that significance boundary of 50 is a safe and low boundary, which performs optimal with a Cosine Similarity threshold of 0.8. Finally, table 4.4 showed that a sketch size of 250 with a depth of 5 is the smallest sketch with still an optimal performance. Table 4.5 gives an overview of the optimal settings that were obtained.

Setting	Value
State similarity metric	Cosine Similarity
Similarity threshold	0.8
Future size	2
Significance boundary	50
Sketch width	250
Sketch depth	5

Table 4.5: The best performing settings for the Streaming Blue-Fringe.

Note that the initial decision to use Count-Min sketches was based on the advantages in approximating counts of large numbers of elements. Count-Min sketches use a highly efficient data structure that requires memory resources of a fixed size, independent of how many elements are counted. The experiment results in this section showed; however, that in practise a future size of 2 already captures enough information to learn accurate models. As a result, for small alphabet sizes the number of unique elements stored in the sketches will remain low. Hence, it is expected that the use of Count-Min sketches is not essential to efficiently learn accurate models. One could also experiment with maintaining Prefix Tree Acceptors in each state instead. Nonetheless, it is decided to continue using Count-Min sketches in this study. The motivation for this decision is the ease of sketch comparisons and to ensure the method stays robust, also for larger alphabets.

4.6. Validating the learning process

The previous section tested various options, using a set of 15 generated models. In order to prevent overfitting, in this section the final performance will be measured using the validation dataset as shown in table 4.3. For each model the KL divergence is computed. The resulting KL divergences are shown in table 4.6. The results show that the average KL divergence is similar to the optimal performance from the previous section. This shows that the previously determined settings work good in general, since it also performs well on new data.

Validation model	KL divergence
Model 1	0.250
Model 2	0.166
Model 3	0.297
Model 4	0.960
Model 5	0.970
Model 6	0.380
Model 7	0.608
Model 8	0.543
Model 9	0.546
Model 10	0.410
Average	*0.513*

Table 4.6: The KL divergence for each model from the validation set, following by the average KL divergence over all models.

4.6.1. Learning steps

Next, we will have a look at the different steps of the learning process. In order to limit the number of steps to visualise, an additional model is made existing of only 3 states. Also, the white states are hidden from the visualisation to make it more readable. The Streaming Blue-Fringe is executed on the training set of this model and visualisations are created during the learning process. Once a state is colored red, merged or a new blue state is added, a visualisation of the current PDFA is written to disk. The visualisations created at each of the learning steps is shown in figure 4.7. In figure 4.7 h the correct model generated by the PAutomaC tool is visualised. By visually comparing 4.7 g and 4.7 h, one can conclude that the Streaming Blue-Fringe has learned a model identical to the true model. The different visualisations correctly show the different steps of the learning process.

Step I In 4.7a, the first blue state is created starting from root state A with symbol 0.

Step II In 4.7b the second blue state is created. The transition to this state also starts at the root state, but now with symbol 1.

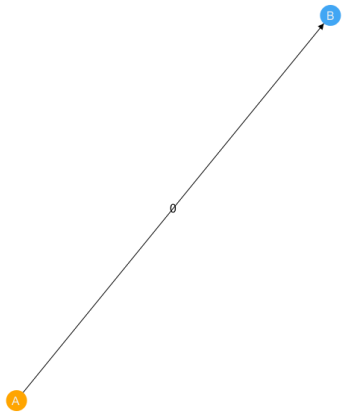
Step III Figure 4.7c captures the moment the state with symbol C is promoted to become a red state. Directly after promotion, a new blue state is created with a transition of symbol 0.

Step IV 4.7d shows the moment state B reaches its significance boundary. Its sketch is not similar to the root state and hence state B becomes a red state. Next, two blue states are created for symbols 0 and 1 that pass through state B.

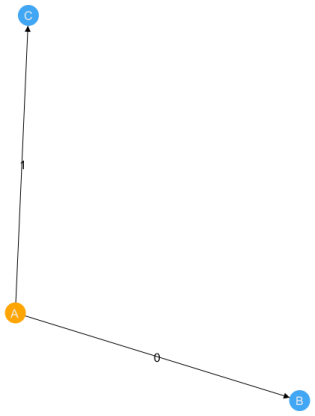
Step V 4.7e shows the moment that state D reaches its significance boundary. The sketch of state D is identical to the sketch of state B, as shown in figure 4.8. As a result, the blue state D merges with state B, resulting in the model of figure 4.7e.

Step VI In figure 4.7f, state F reaches its significance boundary and has a high similarity with the root state A. This results in a merge of state F with root state A. As a result, state B has a transition to state A.

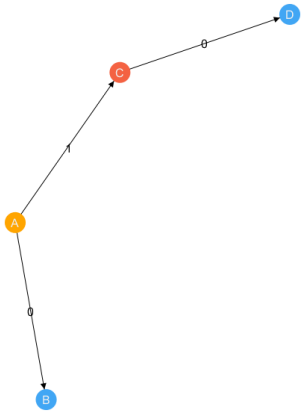
Step VII Finally, in figure 4.7g, state E is merged with state B.



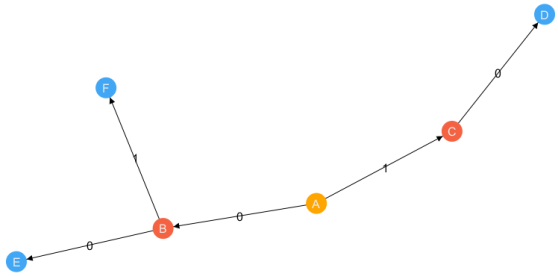
(a) Step I



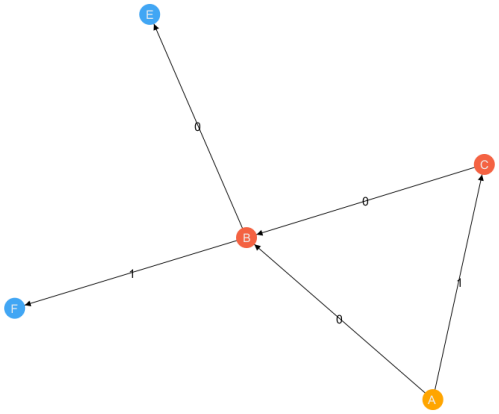
(b) Step II



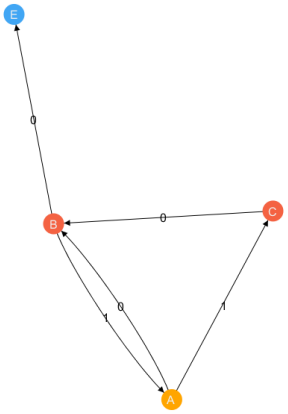
(c) Step III



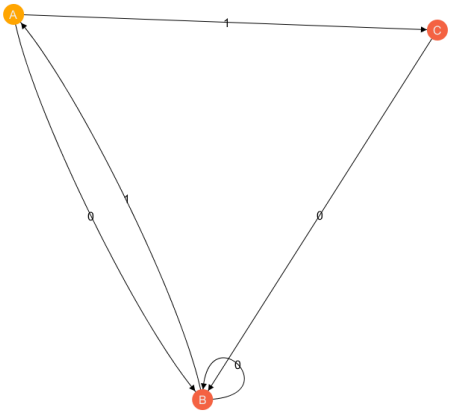
(d) Step IV



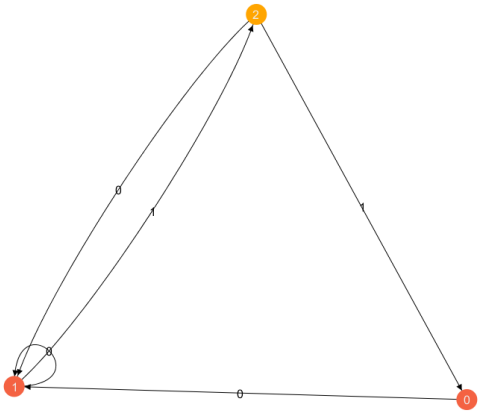
(e) Step V



(f) Step VI



(g) Step VII



(h) PAutomaC model

Figure 4.7: The steps of learning a State Machine with three states.

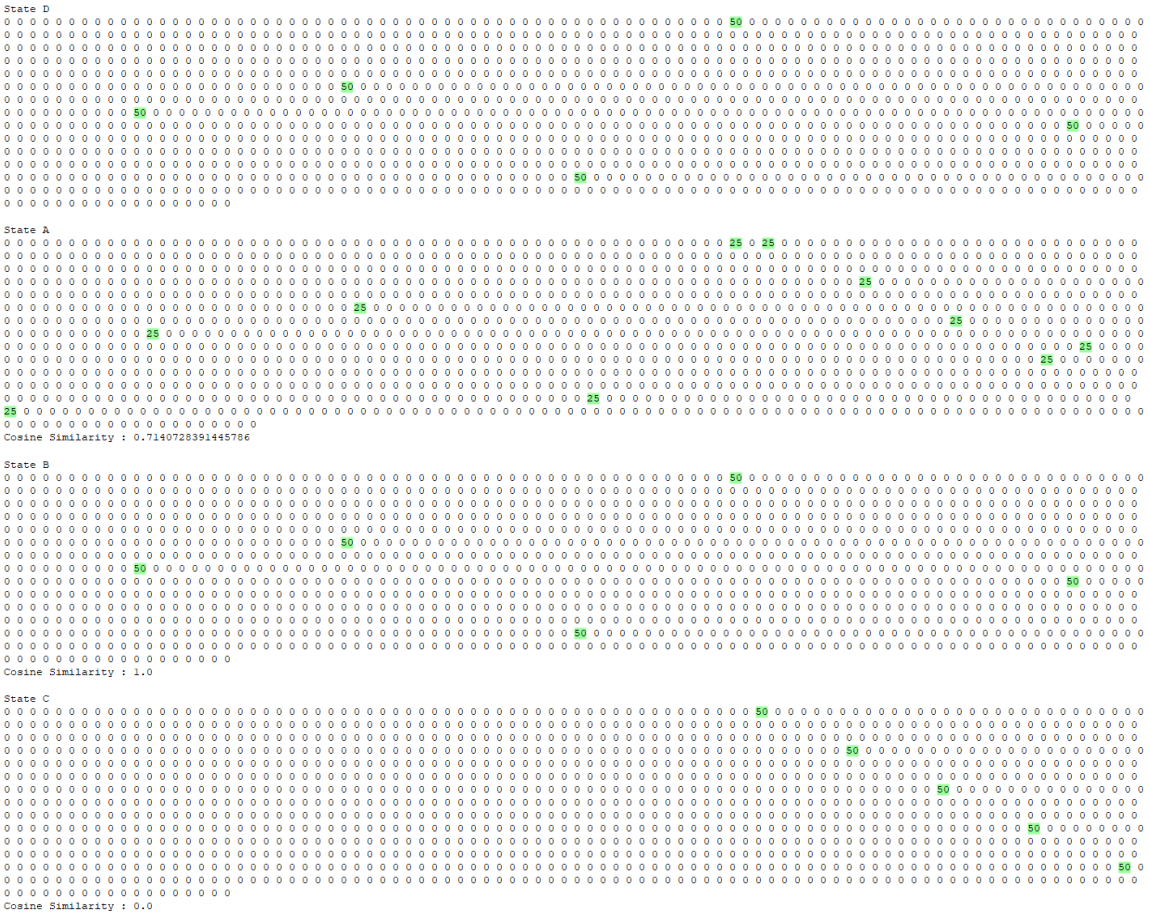


Figure 4.8: Vector representations of the sketches as state D reaches its significance boundary in figure 4.7 d. First the sketch of blue state D is shown, followed by the sketches of red states A, B and C. State D shows a perfect match with the sketch of state B.

4.6.2. Visual comparison of validation models

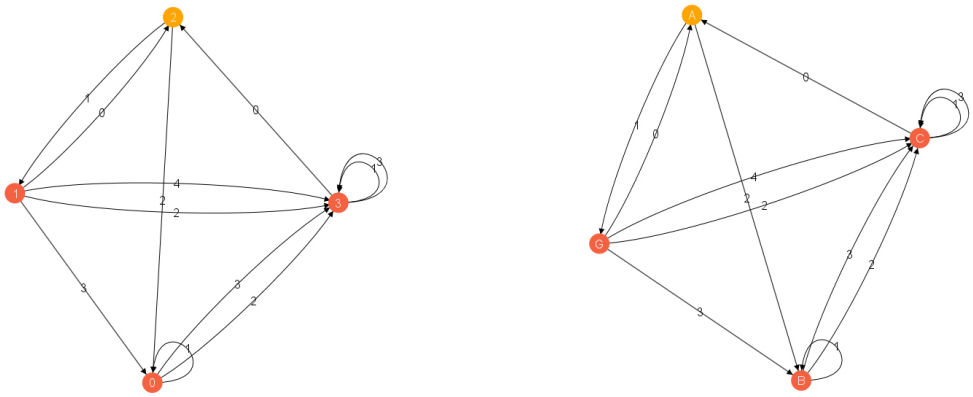
The steps of figure 4.7 show the learning process of a small State Machine. To validate the correctness of the learning method, it is also important to inspect the more complex models. The following figures show all models from the validation set, together with model that is learned with the Streaming Blue-Fringe. The first four learned models are identical to the true PAutomaC model.

Figure 4.13 shows a small deviation between the generated model 5 and the learned model. It appears that an additional state merge has been performed between states 7 and 10 of the PAutomaC model. After closer inspection, the most often occurring transition from state 7 was a transition to state 10 with symbol 0. Since state 10 contains a self-loop with symbol 0, that often occurred, the futures from states 7 and 10 are quite similar. Hence, the algorithm merged the two states. Although this is a visual difference, since the algorithm optimizes for the probability distribution between State Machines, this can happen during the learning process.

Model 6, which is displayed in figure 4.14, is visually very similar to the true model. However, the root state in the PAutomaC model shows a self loop with symbol 4, which is not present in the learned model. After inspection of the root state, the reason was that symbol 4 only occurred 24 times. Because the significance boundary is 50, this state had not yet become part of the State Machine. Although this shows a visual difference, it hardly impacts the performance as a transition with symbol 4 occurs very rarely. Therefore the probability in the PAutomaC model is very low, which matches with the near-zero probability assigned to traces starting with symbol 4 in the learned model.

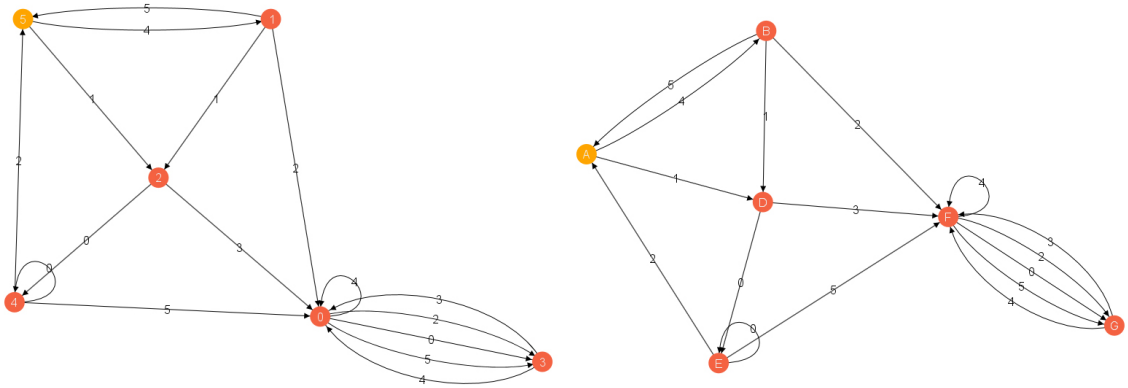
Figure 4.15 shows quite some differences between the true model 7 and the corresponding learned model. By manual inspection of the blue states during the learning process, it was the case that a number of states visible in the PAutomaC model, received too little traces. Examples are the transition from state 0 to 7 and the transition from state 4 to 7. Furthermore, after closer inspection of figure 4.15 a, one can conclude that state 9 and 13 are not reachable from the root state. Therefore, there do not exist traces to these states in the training set and hence they could not become part of the learned model. This is probably a result of the low symbol sparsity of model 7.

Finally, models 8 till 10 are quite complex, which makes them very hard to manually analyse. By looking at the general structure they look very similar. The number of self-loops in both the true and the learned models are similar. There are some states missing in the learned model. This is either due to unreachable states, like for example state 3 in model 9, or due to additional state merges. Overall, the complex models look similar to their PAutomaC counterpart.



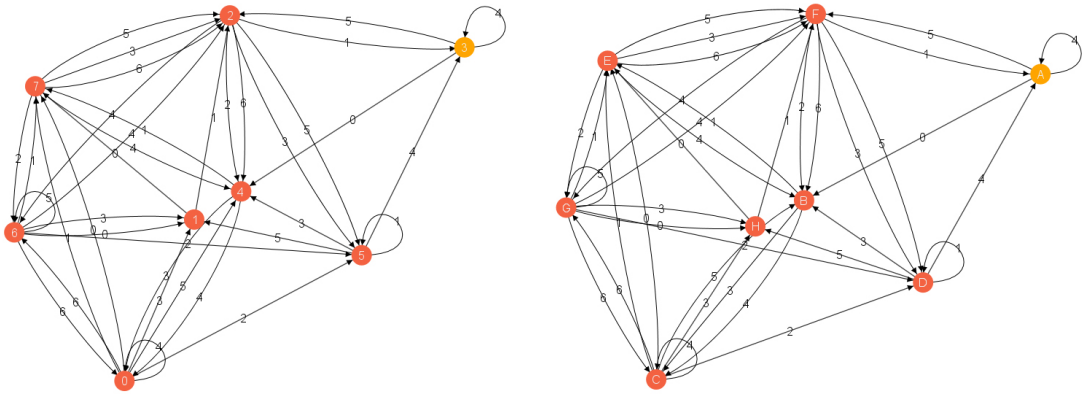
(a) PAutomaC model **(b)** Learned model

Figure 4.9: Visual comparison of validation model 1



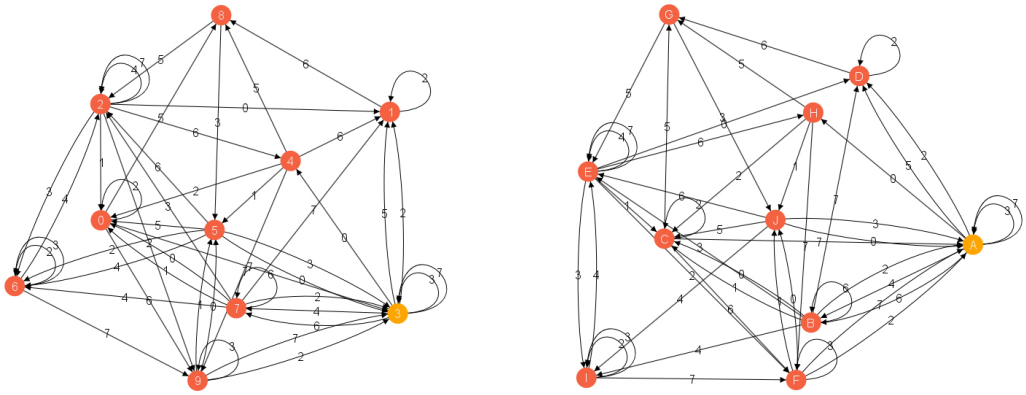
(a) PAutomaC model **(b)** Learned model

Figure 4.10: Visual comparison of validation model 2



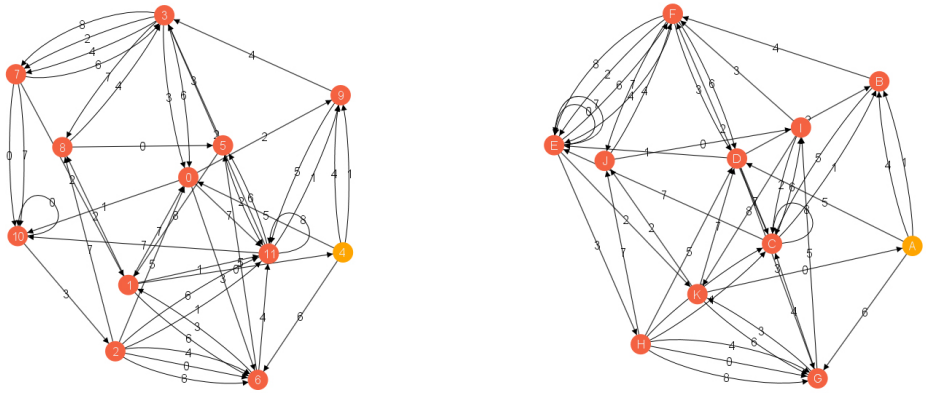
(a) PAutomaC model **(b)** Learned model

Figure 4.11: Visual comparison of validation model 3



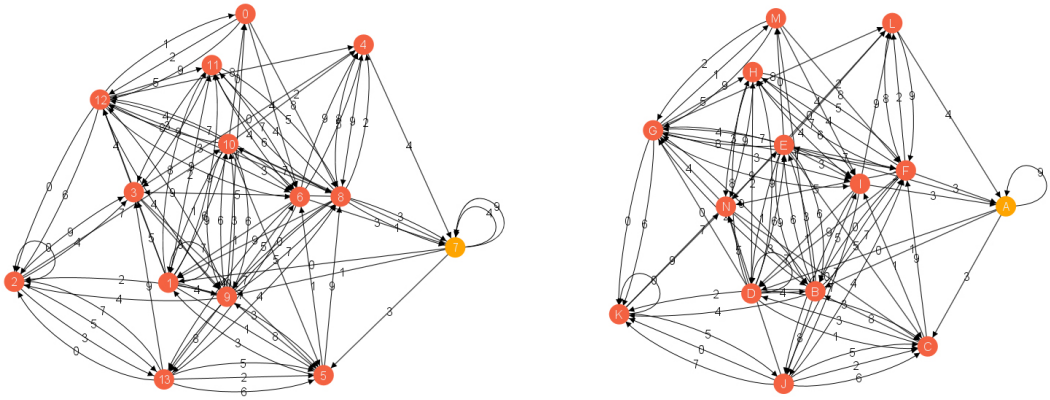
(a) PAutomaC model **(b)** Learned model

Figure 4.12: Visual comparison of validation model 4



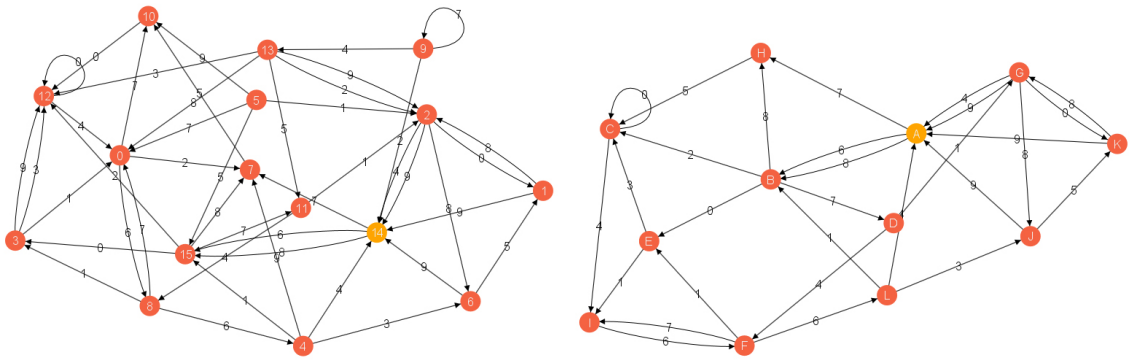
(a) PAutomaC model **(b)** Learned model

Figure 4.13: Visual comparison of validation model 5



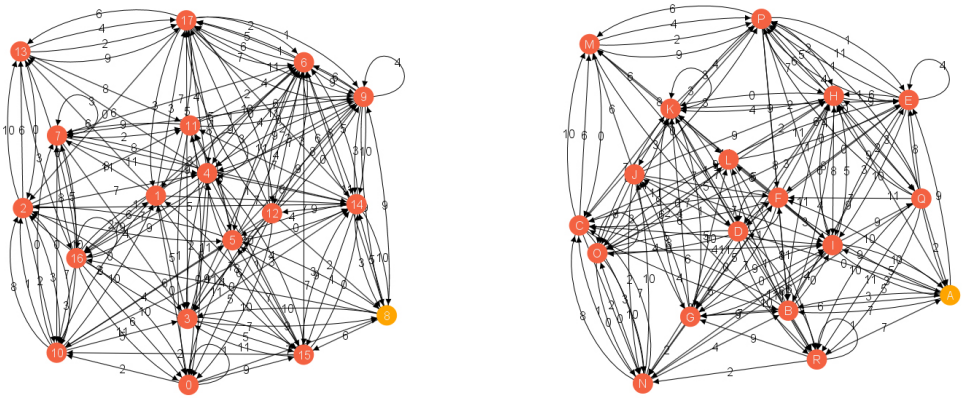
(a) PAutomaC model **(b)** Learned model

Figure 4.14: Visual comparison of validation model 6



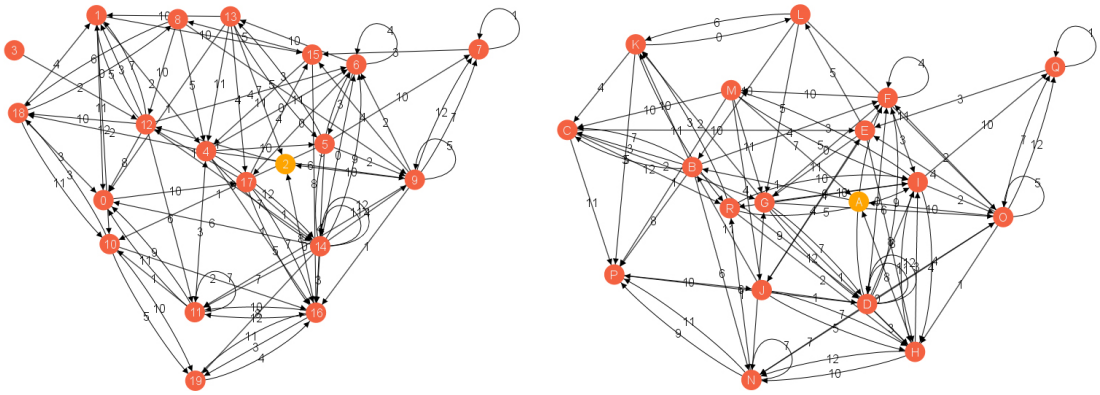
(a) PAutomaC model **(b)** Learned model

Figure 4.15: Visual comparison of validation model 7



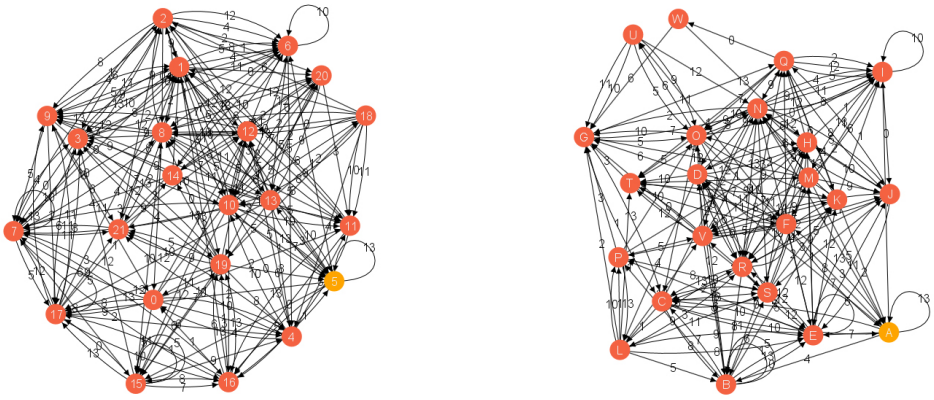
(a) PAutomaC model **(b)** Learned model

Figure 4.16: Visual comparison of validation model 8



(a) PAutomaC model **(b)** Learned model

Figure 4.17: Visual comparison of validation model 9



(a) PAutomaC model

(b) Learned model

Figure 4.18: Visual comparison of validation model 10

5

The application of State Machines for detecting malicious behaviour

In previous chapters, background knowledge is collected after which an algorithm is designed and implemented for learning State Machines on data streams. This chapter explores one application of this technology in network based threat detection. In section 5.1 a modified experimental setup is described, tailored for the threat detection use case. Section 5.2 develops a method for learning State Machine fingerprints on network traffic streams using the system developed in the previous chapter. The learned fingerprints represent malicious behaviour, for which it is valuable to organizations and society that we are able to detect their presence. In section 5.3 a detection mechanism is developed aimed at detecting the presence of the behaviour captured in the State Machine fingerprints. The performance in terms of effectiveness and scalability of the approach proposed in this chapter will be evaluated in chapter 6.

5.1. Experimental setup

Section 3.2 introduced a hardware and software setup used to develop and execute various Apache Flink jobs on NetFlow streams. Until now, modified setups were used employing a NetFlow generator and the PAutomaC sequence generator to gain more control over the design process. In this chapter, the setup is applied on NetFlow captures of malicious data and to real-world NetFlow data. Since it was not feasible to gain access to a host that is directly connected to a live Flow Collector, a machine is introduced that mimics the actions of a flow collector. A dump of one year of unsampled NetFlow data is obtained which was collected at the EEMCS Faculty of the TU Delft. This data is replayed using nfreplay from a host running Kali Linux towards the machine running the tool vFlow and the Apache Flink job.

5.1.1. Real-world datasets

In this chapter two NetFlow datasets will be used. The first is a sampled dataset collected by KPN, one of the largest Telecom providers of the Netherlands. This dataset contains anonymized NetFlows of customer traffic from their international network. The traffic captured in these flows has been summarized at one of four collectors. So a roughly one fourth portion of the international KPN traffic is present in the data. The dataset is sampled using a sampling rate of 1 in 8190 packets, further reducing the amount of captured packets. However, still a massive dataset remains containing rough statistics of the vast amounts traffic flowing in and out of the Netherlands.

The other dataset is an anonymized unsampled dataset collected by the Delft University of Technology. The data has been collected on the Eduroam network of the EEMCS faculty building. The Eduroam network is a network which is daily used by a large number of researchers and students. It therefore contains a large variety of different traffic types. This dataset is unsampled, so all packets have been summarized in creating the netflows present in the data.

Since the method proposed in this thesis learns behaviour using sequences of events, it is important to process all network data. If data is missing or out of order, it is not possible to detect the presence of any pre-learned behaviour. The highly sampled dataset of KPN, makes it therefore unsuitable for detecting attacks using the proposed method. It is; however, a useful dataset for evaluating the throughput of the final solution. This will be further analysed in chapter 6. The data of the Eduroam network is unsampled, but contains a number of errors as will be explained in section 7.1.2. Therefore, this data is also not suitable for evaluation of the detection performance of the final system. However, it is still usable for evaluation of the runtime performance as will be examined in section 6.1.3. Also, it can be used to gain insights into packet statistics of average network traffic. That is what the Eduroam data is

used for in this chapter. An overview of the characteristics of both datasets is depicted in table 5.1.

	TU Delft Eduroam	KPN International
First flow	2017-10-01	2018-04-01
Last flow	2018-06-30	2018-04-31
Reporting interval	5min	5min
Sampling rate	1:1	1:8190
Average flows/sec	267	91k

Table 5.1: Various statistics of both real-world datasets accessible in this thesis

5.2. Learning fingerprints

In order to detect the presence of malicious behaviour in traffic channels described by NetFlow data, first fingerprints should be learned of known malicious behaviour. In order to learn these fingerprints, the incoming NetFlows should be mapped to symbols, forming the sequences used to learn the State Machine. In section 5.2.1 a mapping is defined which maps NetFlows to individual symbols. Next, network traffic of known malicious behaviour is required, to be able to learn fingerprints. A great source of samples of malicious network traffic is provided by the Stratosphere IPS project [81]. In section 5.2.2 network traffic of various types of malware are explored. Section 5.2.3 describes the learning process on the malicious network traffic and gives an overview of the fingerprints that are learned.

5.2.1. NetFlow to symbol mapping

In the previous chapter, while designing the learning method, the datasets generated by PAutomaC prescribed the alphabet to be used for mapping subsequent elements to sequences. In order to learn State Machines on NetFlow streams, a mapping needs to be defined for mapping NetFlows to symbols. NetFlows can describe an almost infinite amount of parameter combinations, so mapping them into a finite set of symbols is essential. In creating a mapping it is important to use a number of possible mappings that is not too large. Otherwise a small deviation in a NetFlow parameter will result in different symbols and hence in a completely different State Machine, yielding false negatives. On the other hand, the alphabet should also not be too small, to prevent false positives as a result of equal State Machines for different types of behaviour. Hence, it is important to find the right balance in the number of symbols. In order to find a good mapping, it is important to get an equal spread of the symbols over the possible parameters values. For instance, if one symbol covers 90% of the data, the system would not be able to capture fine differences within this 90%. Therefore, each of the symbols should cover an equal portion of the variance within the dataset, to get the most diverse State Machines with a given number of symbols.

Dataset for parameter statistics

In order to find the best spread of symbols over all possible parameter values, a portion of data is analysed. Since the Eduroam data describes traffic of a large number of hosts, it is expected to contain data of a large range of different types of traffic. Therefore, the Eduroam dataset will be used for the analysis of NetFlow parameters. In order to find the best mapping of NetFlow parameters to symbols, all traffic captured during one working day is analysed. More than 14 million NetFlows were processed, of which one in thousand is kept in order to reduce the number of points to use in the analysis. Roughly 14200 parameter combinations remained after the random sampling of flows from an entire day of network traffic.

Parameter selection

As mentioned in section 2.4, a NetFlow usually has the following parameters:

1. Arrival time of first packet
2. Arrival time of last packet
3. Total number of bytes transferred
4. Total number of packets transferred

Using the arrival time of the first and last packet in the flow, the flow duration can be computed. By subtracting the time of the last packet from the arrival time of the first packet of the subsequent flow, the flow inter-arrival time can be computed. Both duration and inter-arrival time are interesting derived parameters, capturing information about the rate at which the packets flowed and the delays within the communication. However, as mentioned in section 5.1.1 the available unsampled Eduroam traffic does not contain correct timestamps, so these parameters will not be further explored. Hence, in this thesis the focus is on the number of bytes and the number of packets captured in a flow.

In order to get an understanding of the distribution of both parameters in a real-world dataset, both parameters are plotted in a scatterplot in figure 5.1 using the 14200 data points derived from one working day of traffic. The figure shows all data points concentrated in a diagonal area. This is to be expected, since the number of bytes transferred in a flow grows as more packets are added. It is impossible to have a NetFlow with a large number of packets and only a small number of bytes transferred, since each packet has a minimum size. Also, a small number of packets cannot have a large total size since the size of a single network packet is limited to 1500 bytes. The correlation between number of packets transferred and the total size of the packets in the NetFlow equals 0.93, confirming a strong positive linear dependence.

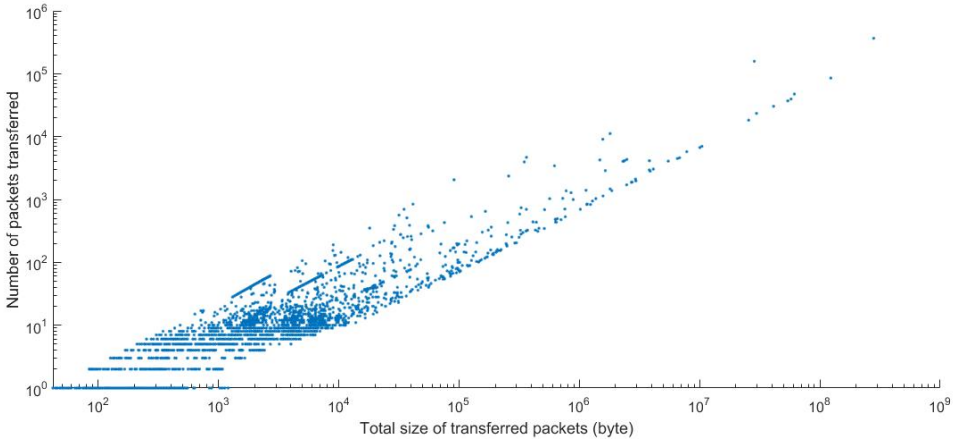


Figure 5.1: The relation between the number of packets in a NetFlow versus the total size of all packets transferred.

The real piece of information stored in the number of bytes transferred, is the average byte size of a packet. To extract this information, the number of bytes transferred is divided by the number of packets transferred. The relation between the number of packets transferred and the average packet size of the packets within the flow is shown in figure 5.2. The graph shows a much more randomly looking distribution of parameter combinations. The average packet size is ranging from 0 to 1500 bytes as expected and no clear relation is visible between the number of packets and the average size. This is confirmed by computing the correlation, which turns out to be only 0.039.

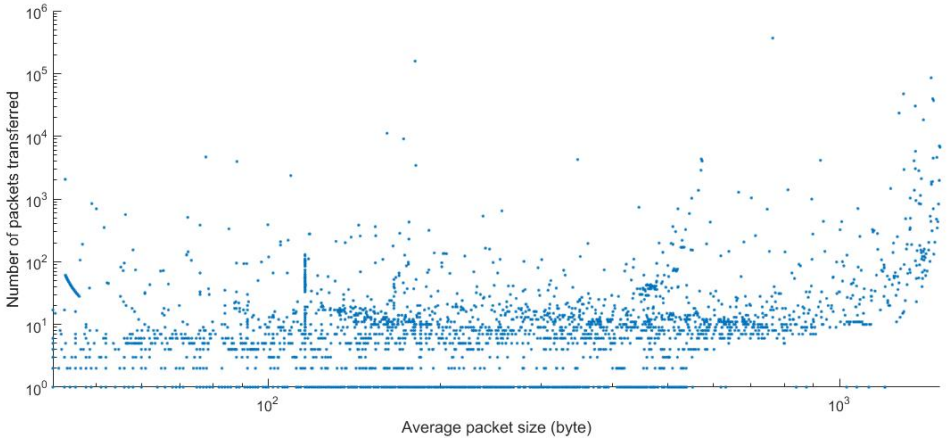


Figure 5.2: The relation between the number of packets in a NetFlow versus the average packet size.

We now have two interesting independent NetFlow parameters:

1. **Number of packets transferred**, indicating the amount of communication that took place
2. **Average packet size**, which varies for different types of traffic

Boundary value selection

In order to create symbols with equal occurrence frequency, the data is ordered and divided into percentiles. The data is ordered once for the *Number of packets transferred* and once for the *Average packet size*. The resulting values for a subset of percentiles is shown in table 5.2.

Percentile	Number of packets transferred	Average packet size
10th	1	46
20th	1	77
25th	1	94
30th	1	116
33th	1	116
40th	3	148
50th	4	161
60th	5	221
66th	8	311
70th	9	369
75th	9	382
80th	10	413
90th	15	628

Table 5.2: The number of packets transferred and the average packet size for different NetFlow percentiles.

It is interesting to see that although the previous two graphs showed the occurrence of NetFlows containing more than 100 packets, the vast majority of NetFlows summarized connections of 15 packets or less. The table also reveals that it is not possible to divide the data in more than 3 equally occurring parts, using the *number of packets transferred* in a flow as boundary value. Hence, it is decided to use the 33th and 66th percentile as boundary value for the *number of packets transferred* parameter.

Since the *average packet size* parameter is independent of the *number of packets transferred* parameter, it is possible to also select equally spaced boundary values from the table for this parameter. For the *average packet size* parameter, different groupings are possible. In order to find the best number of groupings, which is not too general and not too specific, the performance of different divisions could be tested. Models can be learned from different days of the same malware execution. If we assume that the malware exhibits the same behaviour on each day, the models should be similar. By comparing the kl divergences between models of different days for different *average packet sizes*, it is possible to find the best *average packet size* mapping that would result in models of equal behaviour having a close divergence. However, due to time constraints it is decided not to perform this experiment and make an educated guess instead. It is assumed that the percentiles 20, 40, 60 and 80, is a division that is not too susceptible to noise while still being able to capture enough differences in types of behaviour.

Resulting symbol alphabet

From the parameter and boundary value selection, an alphabet can be constructed containing a set of symbols that describe equally frequent NetFlow parameter combinations. We defined three equally occurring groups based on the number of packets transferred and defined five groups based on the average packet size. By combining both parameters, this yields 15 nearly equally probable groups, since both parameters are independent. The groups could be assigned a character of the alphabet; however this hides the information about the group. Hence, it is decided to map the groups of each parameter to an integer value and then concatenate the parameter values. The resulting fifteen mappings of NetFlows to symbols are shown in table 5.3.

Symbol	Average packet size		Packets transferred	
	Upper bound	Lower bound	Upper bound	Lower bound
0-0	77	0	1	0
1-0	148	78	1	0
2-0	221	149	1	0
3-0	413	222	1	0
4-0	∞	414	1	0
0-1	77	0	8	2
1-1	148	78	8	2
2-1	221	149	8	2
3-1	413	222	8	2
4-1	∞	414	8	2
0-2	77	0	∞	9
1-2	148	78	∞	9
2-2	221	149	∞	9
3-2	413	222	∞	9
4-2	∞	414	∞	9

Table 5.3: Mapping from NetFlow parameters to symbols

5.2.2. Datasets of malicious behaviour

The Stratosphere IPS Project [81] contains network traffic of over 300 malware instances, recorded between 2013 and 2018. The packet captures originate from the related Malware Capture Facility Project [29]. In this project, honeypots are employed to execute binaries of malware in controlled environments. Each analysed type of malware is allowed to run for a number of days, while all the traffic flowing in and out of the machine is captured in pcap files. The pcap files are bundled together with NetFlow statistics and reports created by various analysis tools and published on the Stratosphere IPS website [81].

Selecting malicious traffic samples

From the malicious traffic samples made available on the Stratosphere IPS website, a selection is made to create multiple fingerprints of malicious behaviour. Two datasets are constructed, one set of samples which will be used to define a detection threshold in section 5.3.3 and one set which will be used to evaluate the detection method in chapter 6.

In order to define a detection threshold, fingerprints of different days of executing the same malware sample are compared. It is assumed that malware behaves in a similar way on different days. With this assumption, a detection threshold is considered good if a fingerprint of day one detects the behaviour of all later days. The search for the right threshold is performed on a random selection of recent malware samples as described in table 5.4.

CTU ID	Malware name	MD5 hash	Execution duration
168-2	Andromeda Botnet	be8797e324da219fedf06732347c4993	8 days
210-1	WisdomEyes Trojan	1aa8d5ca763ef73bb48afd5aab4566bb	4 days
214-1	Locky Malware	e7aad826559c8448cd8ba9f53f401182	15 days
342-1	BitCoinMiner	ad4c296849b12786e6b4edc8b271b3d9	36 days

Table 5.4: Malware samples used to define the detection threshold

Once a detection method is created and a detection threshold has been determined in section 5.3, the detection method can be evaluated. For this an additional set of traffic samples has been selected, which will be described later in section 6.2.

5.2.3. Learning fingerprints of malicious behaviour

Using the experimental setup described in section 5.1 and the NetFlow to symbol mapping from section 5.2.1, the Streaming Blue-Fringe is executed on NetFlow recordings of different types of malware. It is decided to learn fingerprints on only UDP and TCP traffic, since these are the most common transport layer protocols used in command and control channels. Since hosts usually exhibit multiple types of behaviour simultaneously, traffic is divided based

on the pairs of sourceIP and destinationIP. Each unique sourceIP,destinationIP combination will result in a separate State Machine. This ensures that the presence of benign traffic does not impair the ability to detect malicious traffic. For example a user that is browsing the web, communicates over multiple benign traffic channels while in the background a connection to a command and control channel is active. Each of these connections would result in a different State Machine, enabling the detection of the malicious behaviour regardless how much normal traffic is generated.

Figure 5.3 shows an overview of the learning process. Malware samples are feeded into Apache Kafka. In this study, the samples are read from disk, but Kafka can also be connected to any other source. For example to one or more NetFlow collectors in a honeynet. Apache Kafka publishes all incoming NetFlows into a Kafka topic. This topic is read by Apache Flink. Next a two-part Apache Flink job is executed, which will be shown in more detail later in section 5.3.1. Essentially, the first step is a mapping operation that maps the stream of NetFlows into a separate stream for each traffic channel (i.e. for each IP pair). The second step performs the actual Streaming Blue-Fringe and learns a State Machine on each traffic channel. The result is a model for each traffic channel, on which manual selection is applied to filter out non-malicious traffic channels.

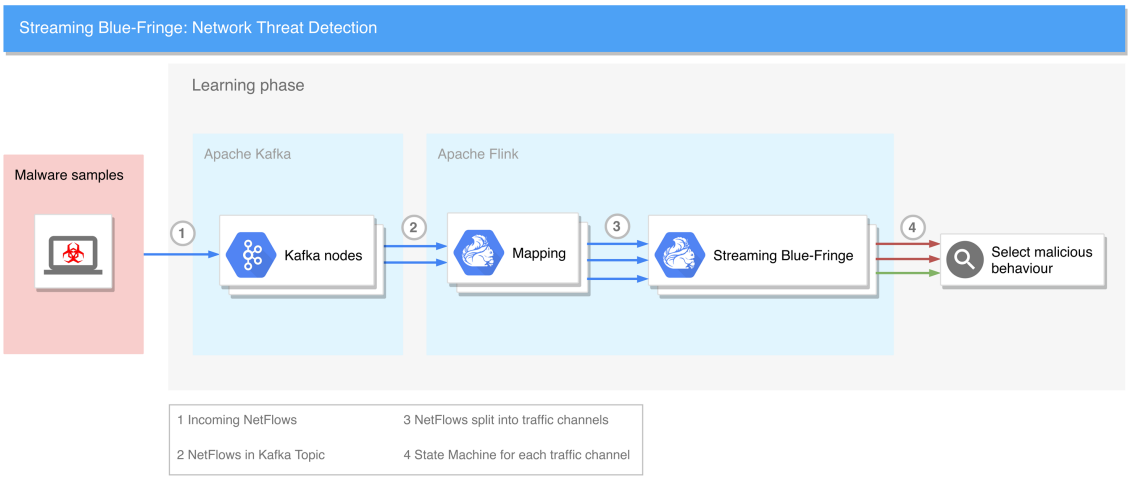


Figure 5.3: An overview of the learning phase of applying the Streaming Blue-Fringe to network-based threat detection.

Fingerprints are learned by replaying the netflows captured from the malware samples presented in table 5.4. The data is replayed for one sample at a time and only for the first day of traffic. The traffic of only the first day is used in order to be able to define an appropriate detection threshold in section 5.3.3 on the traffic of the subsequent days. Since each sample contains connections with multiple hosts, manual inspection of the packet capture files was performed to find the IP address of the command and control server. The IP addresses were verified by searching for malicious connections to this host on the Hybrid Analysis website [4]. The resulting fingerprints of the first day of execution of each type of malware from table 5.4 are described in figures 5.4 to 5.7.

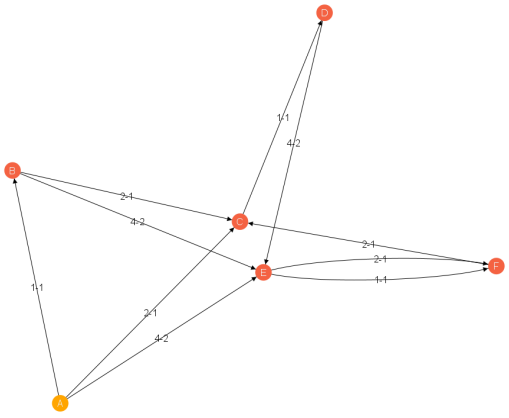


Figure 5.4: Fingerprint of Command & Control traffic from the first day of Andromeda Botnet traffic (168-2)

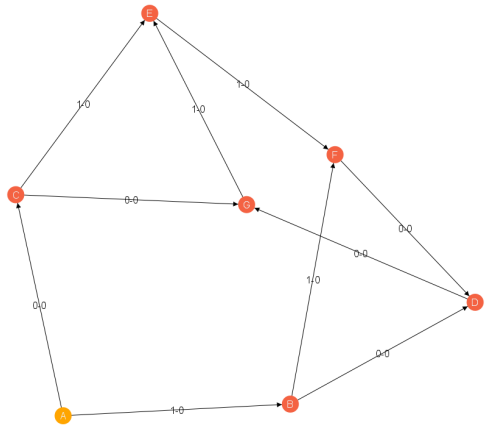


Figure 5.5: Fingerprint of Command & Control traffic from the first day of WisdomEyes Trojan traffic (210-1)

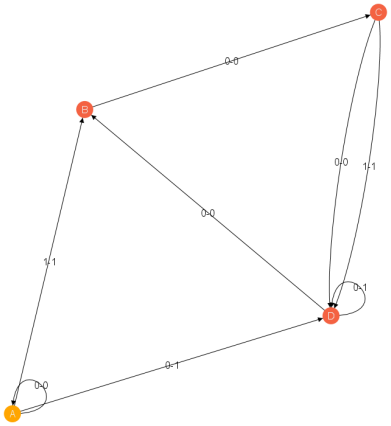


Figure 5.6: Fingerprint of Command & Control traffic from the first day of Locky Malware traffic (214-1)

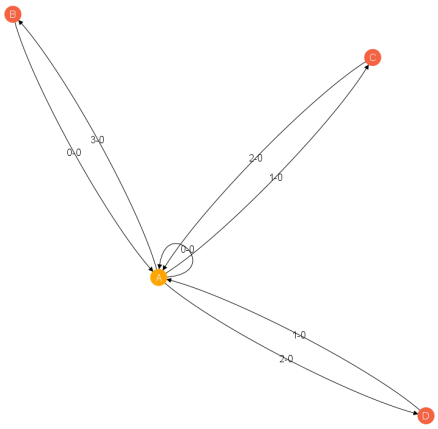


Figure 5.7: Fingerprint of Command & Control traffic from the first day of BitCoinMiner Trojan traffic (342-1)

5.3. Detection mechanism

For detecting malicious behaviour, the real-time learning method from chapter 4 can be executed with either a connection to a real-world network or on replayed data. During the learning process, State Machines need to be regularly compared with known malicious behaviour. Similar to section 5.2.3, traffic channels are defined by pairs of source IP and destination IP. This ensures detection on similar subsets of traffic as was used when learning fingerprints of known malicious behaviour.

Figure 5.8 shows an overview of how the results of the learning phase from figure 5.3 can be combined into a detection phase. The pre-selected fingerprints of malicious behaviour are now used as an input in the detection phase. In the bottom left, the detection phase receives NetFlows from a NetFlow source. In practise this is a NetFlow collector, but in the controlled environment of this thesis the source is a replayed stream of NetFlows. Next, similar to the learning phase, the flows propagate via a Kafka pipeline and a Flink mapping operation into separate streams for each traffic channel. Next, the Streaming Blue-Fringe learns State Machines for each traffic channel. This is where the learning and detection phase differ. The detection phase executes the Streaming Blue-Fringe in detection mode, which means State Machines are on regular intervals compared against the stored fingerprints. How this is achieved, is described in section 5.3.2. In the bottom right, the traffic channels that do match with pre-learned fingerprints are marked as detected threats. The detected threats can outputted via a range of methods. In this thesis, the detections are written to the console and corresponding models are stored on disk.

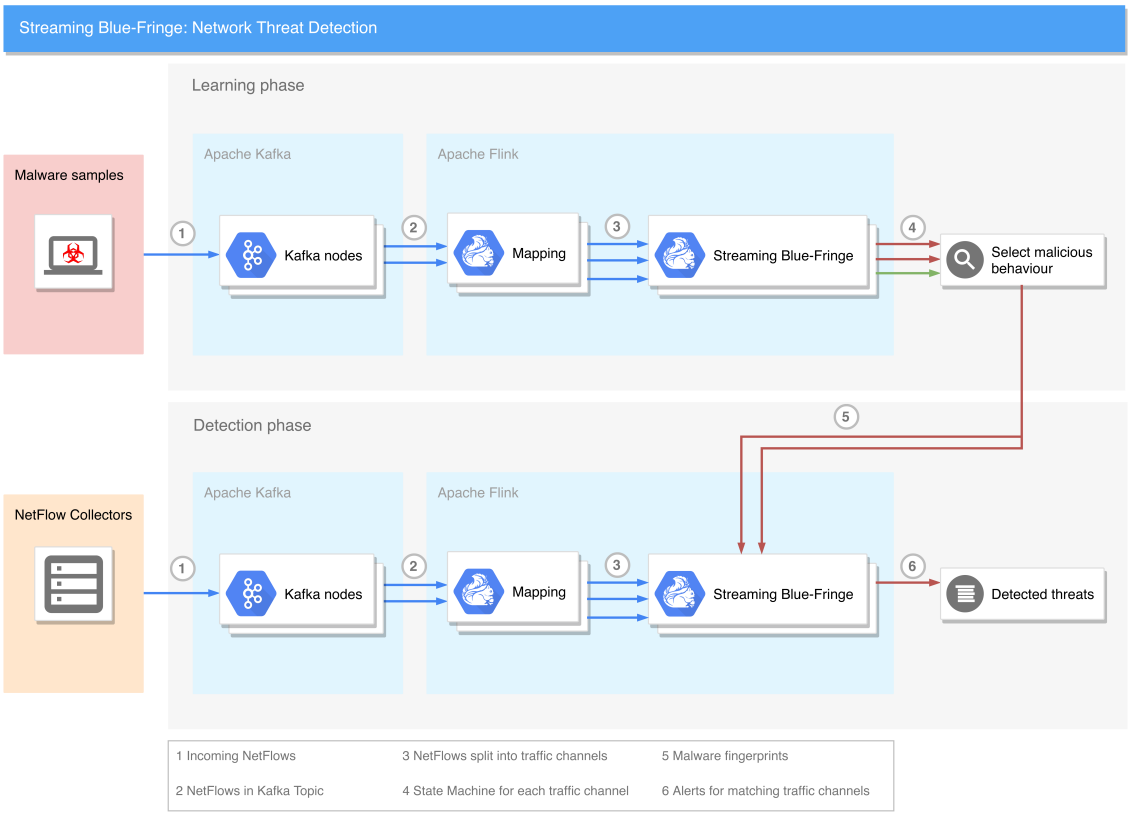


Figure 5.8: An overview of the learning and detection phase of applying the Streaming Blue-Fringe to network-based threat detection.

5.3.1. Apache Flink implementation

Due to its similarity with the other problems described in chapter 3, the implementation of the Streaming Blue-Fringe adopts the generalized structure as defined in section 3.6. First, the incoming NetFlows should be split up into individual NetFlows before they can be distributed across nodes or threads. The *FlatMap* operation performs this task, resulting in a stream of all individual NetFlows. Since for each traffic channel a separate State Machine is learned, the learning process can be performed using a single *Reduce* operation. This is the case since data of one traffic channel is never needed in another Streaming Blue-Fringe instance. After mapping NetFlows into a stream per traffic channel, data does not need further redistribution.

```
DataStream<StateMachineNetFlow> hostSequences = netflowStream
.flatMap(in, out) {
  for (StateMachineNetFlow flow : in.dataset) {
    out.collect(flow);
  }
}
.keyBy("IPPair")
.reduce(streamingBlueFringe, incomingNetflow) {
  streamingBlueFringe.consumeNetFlow(incomingNetflow);
  return streamingBlueFringe;
});
```

Pseudocode 7: An Apache Flink job implementation of the Streaming Blue-Fringe. A *FlatMap* operator maps bundles of NetFlows into a stream of all individual NetFlows. A *KeyBy* on IP pair divides the stream into separate streams for each source,destination pair. A *Reduce* operation is applied to each channel in parallel, learning the State Machine on the streams of NetFlows.

In the call to the *consumeNetFLow()* method, the State Machine of the corresponding traffic channel is updated using an implementation of the pseudocode defined in algorithm 3.

5.3.2. Similarity evaluation

To detect the presence of malicious behaviour the State Machines that are currently being learned should be regularly compared to the pre-learned fingerprints. Since consuming a single NetFlow has a very small impact on the State Machine, it is not necessary to compare State Machines on each incoming NetFlow. The largest changes in the State Machines occur once a blue state is promoted to become a red state or merged with an existing red state. Once this happens, the probability distribution of the sequences accepted by the State Machine changes significantly by allowing additional sequences. Hence, it is decided to compare the State Machine of a traffic channel with malicious fingerprints once a blue state is promoted or merged with an existing red state.

During the setup of the Apache Flink job, each node loads all fingerprints in memory. Once a traffic channel promotes or merges a state, the current State Machine is compared with all locally loaded fingerprints. The comparison is performed by generating up to 1000 unique random sequences with corresponding probability from the current State Machine. Each fingerprint is evaluated by passing all sequences through the fingerprint and assigning the corresponding probability or 10^{-15} if the sequence is not accepted by the fingerprint. A near-zero probability is used in order to smoothen the probabilities as explained in section 4.4.3. Next, for each fingerprint the occurrence probabilities of all sequences are normalized to ensure the sum of probabilities remains 1 after smoothing. Now a list of probabilities is established for the current State Machine and for each fingerprint. For each fingerprint the KL divergence is computed using equation 2.4. This yields a measure of dissimilarity between the current State Machine and each fingerprint. By comparing the distances with a predetermined threshold value, it can be determined whether the current State Machine is similar to one or more of the fingerprints. In order to optimize the number of false and true positives, an appropriate value for the detection threshold is explored in the next section.

5.3.3. Detection threshold

The detection threshold is used to determined whether a KL divergence between State Machines is small enough to consider the models equal. If the threshold is too high, this increases the chance of false positives. If the threshold is too low, malicious behaviour that is slightly different than the fingerprint will remain undetected. Therefore, it is important to determine an appropriate detection threshold value. Initially, the threshold will be set to 1.0, based on the largest KL divergences of the PAutomaC models shown in table 4.6. In this section, the threshold is optimized using the four fingerprints created in section 5.2.3. It is decided to use data of each malware sample over different days, in order to ascertain that we base the threshold on variance in behaviour of individual malware samples. When looking at different samples, the malware could have been altered. In the worst-case this means a completely different communication protocol, which would result in a very broad detection threshold and thus high chance of false positives.

Andromeda Botnet

The first malware sample is from the Andromeda Botnet containing Command & Control channel traffic, modelled earlier in figure 5.4. The fingerprint is kept in memory while learning State Machines on each day of network data

separately. On each promoted or merged blue state in one of the State Machines, the corresponding model is compared against the fingerprint. All detections with a KL divergence lower than 1.0 are presented in table 5.5. On day 6 and 8 there was no detection. Manual inspection revealed there was no traffic with any IP addresses flagged as malicious during these days. From the results it can be concluded that the traffic on all other days shows Command & Control traffic very similar to the first day.

Day	Source IP	Destination IP	KL Divergence
2	192.168.1.110	108.167.137.38	0.00221
3	192.168.1.110	108.167.137.38	0.00572
4	192.168.1.110	108.167.137.38	0.00172
5	192.168.1.110	108.167.137.38	5.52×10^{-4}
6	no malicious traffic	no malicious traffic	no malicious traffic
7	192.168.1.110	108.167.137.38	7.38×10^{-4}
8	no malicious traffic	no malicious traffic	no malicious traffic

Table 5.5: Detections of the Andromeda Botnet (168-2) using a fingerprint constructed from traffic of day 1

WisdomEyes Trojan

Manual inspection of the WisdomEyes Trojan sample shows a connection to a Command & Control server with IP address 92.51.156.90. On traffic to this server the model from figure 5.5 has been learned. The KL divergence with the models learned on days 2 till 4 are presented in table 5.6. The models show a low divergence, with the largest KL divergence at 0.0620.

Day	Source IP	Destination IP	KL Divergence
2	192.168.1.119	92.51.156.90	1.14×10^{-4}
3	192.168.1.119	92.51.156.90	0.0102
4	192.168.1.119	92.51.156.90	0.0620

Table 5.6: Detections of the WisdomEyes Trojan (210-1) using a fingerprint constructed from traffic of day 1

Locky Malware

From inspection of the PCAP can be concluded that the Locky Malware (214-1) connects with 3 Command & Control channels on each day. The IP addresses were 91.121.97.170, 185.46.11.239 and 188.138.88.184, which are Locky Command & Control servers according to Ransomware Tracker [87]. The model from figure 5.6 was learned on the traffic channel to the server with IP address 185.46.11.239. When running the detection mechanism using the Locky fingerprint, all three Command & Control channels were detected during each day of execution. The resulting KL Divergence values are presented in table 5.5. The highest KL divergence across all days is 0.319.

Day	Source IP	Destination IP	KL Divergence
2	192.168.1.122	91.121.97.170	0.301
2	192.168.1.122	185.46.11.239	0.293
2	192.168.1.122	188.138.88.184	0.209
3	192.168.1.122	91.121.97.170	0.229
3	192.168.1.122	185.46.11.239	0.270
3	192.168.1.122	188.138.88.184	0.224
4	192.168.1.122	91.121.97.170	0.230
4	192.168.1.122	185.46.11.239	0.232
4	192.168.1.122	188.138.88.184	0.260
5	192.168.1.122	91.121.97.170	0.174
5	192.168.1.122	185.46.11.239	0.165
5	192.168.1.122	188.138.88.184	0.181
6	192.168.1.122	91.121.97.170	0.158
6	192.168.1.122	185.46.11.239	0.165
6	192.168.1.122	188.138.88.184	0.158
7	192.168.1.122	91.121.97.170	0.319
7	192.168.1.122	185.46.11.239	0.277
7	192.168.1.122	188.138.88.184	0.187
8	192.168.1.122	91.121.97.170	0.199
8	192.168.1.122	185.46.11.239	0.253
8	192.168.1.122	188.138.88.184	0.259
9	192.168.1.122	91.121.97.170	0.255
9	192.168.1.122	185.46.11.239	0.294
9	192.168.1.122	188.138.88.184	0.200
10	192.168.1.122	91.121.97.170	0.253
10	192.168.1.122	185.46.11.239	0.251
10	192.168.1.122	188.138.88.184	0.253
11	192.168.1.122	91.121.97.170	0.188
11	192.168.1.122	185.46.11.239	0.250
11	192.168.1.122	188.138.88.184	0.251
12	192.168.1.122	91.121.97.170	0.190
12	192.168.1.122	185.46.11.239	0.188
12	192.168.1.122	188.138.88.184	0.226
13	192.168.1.122	91.121.97.170	0.253
13	192.168.1.122	185.46.11.239	0.252
13	192.168.1.122	188.138.88.184	0.252
14	192.168.1.122	91.121.97.170	0.245
14	192.168.1.122	185.46.11.239	0.242
14	192.168.1.122	188.138.88.184	0.243
15	192.168.1.122	91.121.97.170	0.241
15	192.168.1.122	185.46.11.239	0.230
15	192.168.1.122	188.138.88.184	0.231

Table 5.7: Detections of the Locky Malware (214-1) using a fingerprint constructed from traffic of day 1

BitCoinMiner Trojan

Inspection of all IP addresses contacted by the infected host in the BitCoinMiner Trojan sample (342-1), reveals one Command & Control channel with the IP address 5.187.7.235. The traffic channel towards this host was depicted earlier, in figure 5.7. Table 5.8 shows a KL divergence commonly around 0.1 and 0.2. On days 11 and 18, no detections were made. Manual inspection revealed that the behaviour of the malware on these days was completely different. The State Machines of days 11 and 18 do show very little resemblance to the State Machines of other days. Why the behaviour deviated, was not further investigated. From the table can be concluded that the highest KL divergence across all days is 0.311.

Day	Source IP	Destination IP	KL Divergence
2	192.168.1.126	5.187.7.235	0.00524
3	192.168.1.126	5.187.7.235	0.0416
4	192.168.1.126	5.187.7.235	0.0386
5	192.168.1.126	5.187.7.235	0.00413
6	192.168.1.126	5.187.7.235	0.379
7	192.168.1.126	5.187.7.235	0.198
8	192.168.1.126	5.187.7.235	0.169
9	192.168.1.126	5.187.7.235	0.230
10	192.168.1.126	5.187.7.235	0.0300
11	different behaviour	different behaviour	different behaviour
12	192.168.1.126	5.187.7.235	0.00479
13	192.168.1.126	5.187.7.235	0.0670
14	192.168.1.126	5.187.7.235	0.0159
15	192.168.1.126	5.187.7.235	0.0263
16	192.168.1.126	5.187.7.235	0.200
17	192.168.1.126	5.187.7.235	0.00602
18	different behaviour	different behaviour	different behaviour
19	192.168.1.126	5.187.7.235	0.119
20	192.168.1.126	5.187.7.235	0.0140
21	192.168.1.126	5.187.7.235	0.0336
22	192.168.1.126	5.187.7.235	0.251
23	192.168.1.126	5.187.7.235	0.116
24	192.168.1.126	5.187.7.235	0.292
25	192.168.1.126	5.187.7.235	0.249
26	192.168.1.126	5.187.7.235	0.0276
27	192.168.1.126	5.187.7.235	0.0124
28	192.168.1.126	5.187.7.235	0.256
29	192.168.1.126	5.187.7.235	0.311
30	192.168.1.126	5.187.7.235	0.0642
31	192.168.1.126	5.187.7.235	0.0212
32	192.168.1.126	5.187.7.235	0.0165
33	192.168.1.126	5.187.7.235	0.122
34	192.168.1.126	5.187.7.235	0.0163
35	192.168.1.126	5.187.7.235	0.0236
36	192.168.1.126	5.187.7.235	0.212

Table 5.8: Detections of the BitCoinMiner Trojan (342-1) using a fingerprint constructed from traffic of day 1

Threshold value

Based on the performance of the previous four malware samples, the largest deviation appears to be 0.319. The tests on these four samples give a good indication of what deviations can be expected in malware behaviour. Since four samples is not a very accurate sample size, it is decided to take a safe threshold value of 0.5. The next chapter will use the detection method proposed in this chapter, using the detection threshold of 0.5, to evaluate the effectiveness and scalability of the Streaming Blue-Fringe when applied to network-based thread detection.

6

Effectiveness and scalability

The objective of this thesis is to research and develop a method for learning State Machines on data streams with real-time performance. This chapter assesses the performance aspect of the learning method designed earlier in chapter 4. The performance is evaluated by analysing the time complexity of the learning algorithm and by replaying real-world data streams as described in section 6.1. Furthermore, this chapter evaluates the effectiveness of applying State Machines to the network based threat detection use case. The previous chapter proposed a method for detecting malicious behaviour on NetFlow data streams. The evaluation of the effectiveness of this method using both known malicious data and traffic of normal behaviour is described in section 6.2.

6.1. Scalability

In order to obtain real-time learning and detection performance on small datasets as well as on large real-world networks, a scalable solution is essential. The scalability the proposed learning and detection method can be evaluated using the two real-world datasets from the networks of TU Delft and KPN. These large networks are processing millions of flows each day, so testing on these datasets yields interesting insights into how feasible it is to apply the proposed methods on a large scale in practise. But first of all, this section focusses on evaluating the algorithm without the Cyber Security use case. In paragraph 6.1.1 and 6.1.2 the theoretic performance of the Streaming Blue-Fringe is compared against the original Blue-Fringe algorithm. Next, in paragraph 6.1.3 the Streaming Blue-Fringe is evaluated against TU Delft and KPN data in order to evaluate the throughput and memory performance when learning State Machines on network data.

6.1.1. Time complexity

According to literature [50] the time complexity of the original Blue-Fringe algorithm is $O(PS^3)$, with for P the size of the initial Prefix Tree Acceptor and for S the number of states of the resulting State Machine. In the worst case P equals the total number of data elements E, resulting in a worst case performance of $O(ES^3)$.

Streaming Blue-Fringe

In the Streaming Blue-Fringe, sliding window sequences of a fixed length are constructed based on the stream of elements. There are not more sequences evaluated than there are elements in the original stream. Each sequence starts in the root state and follows a path l through the current State Machine of a maximum length in order to prevent infinite loops. During each step of this path, with $O(1)$ a new state is added if none exists and the sketch of the encountered state is updated with an $O(1)$ operation. Hence, the algorithm so far has a run-time that is linear in terms of the stream length. However, one additional operation is performed during the algorithm. Once a blue state's sketch has accumulated enough sequences to reach the significance boundary it is compared with all currently existing red states and either merged or promoted. During the processing of each stream element, at most l states can reach the significance boundary. Since l is constant, the worst case performance is in the order of the number of stream elements times the cost of the Red-Blue merging strategy. During the merging strategy, the candidate state is compared against all currently existing red states. The complexity of the comparison of two states is constant and does not depend on the size of the current State Machine as it involves a sketch comparison. Since red states are never removed, the number of comparisons during each of the Red-Blue merging strategies is bound by the final number of states S. Hence, the time complexity of the Streaming Blue-Fringe is $O(ES)$. This is two orders of magnitude more efficient than the original Blue-Fringe.

6.1.2. Space complexity

In the original Blue-Fringe, first the initial Prefix Tree Acceptor is learned. When the Red-Blue strategy is performed on the PTA, the memory required shrinks from storing the PTA to the point that only the final DFA needs to be kept in memory. The space complexity of the original Blue-Fringe is thus the space required to store the PTA of size P. In the worst case given an alphabet Σ and a future size f, the PTA describes $|\Sigma|^f$ data elements. The resulting DFA, with size S, is always smaller or equal to the size of the PTA. Hence, equation 6.1 describes the relation between the number of possible uniquely stored elements, the PTA size and the final number of states in the DFA. The space complexity of the original Blue-Fringe is $O(P)$, which is in the worst case equal to $O(|\Sigma|^f)$.

$$O(|\Sigma|^f) \geq O(P) \geq O(S) \tag{6.1}$$

Streaming Blue-Fringe

In the Streaming Blue-Fringe the solution is gradually build, starting with only the root state. For each state that is maintained, a constant amount of memory is required which is equal to the size of the sketch. On adding a new state, a transition to this state is added. In case states are promoted, the state and its inwards transition become part of the final State Machine. In case a state is merged, the inward transition is pointed to another state. Hence, the number of transitions never decreases during the learning process and reflects the total number of state that have existed during the learning process. The maximum number of states that can have existed is bounded by the number of states S of the final State Machine times the alphabet size Σ. This is because for a PDFA each state can have at most a transition for each symbol of the alphabet. Since the alphabet size is constant, the number of states that have existed during the learning process is of the order S. Hence, the space complexity of the Streaming Blue-Fringe is of the order $O(S)$. Compared with the original Blue-Fringe, this space complexity occurs only in the optimal case when the PTA is already equal to the final DFA. In all other cases, the memory complexity is larger than $O(S)$ and can even be $O(|\Sigma|^f)$, which means it stores all possible length f combinations from Σ. Therefore, the Streaming Blue-Fringe has a more efficient space complexity.

6.1.3. Performance on real-world data

The previous two sections explained that the theoretic performance of the Streaming Blue-Fringe algorithm is more efficient in terms of time and space complexity compared with the original Blue-Fringe. This section will look at the runtime performance of the Streaming Blue-Fringe when applied to the threat detection use case. In this comparison the maximum throughput is tested and the CPU usage and memory consumptions are monitored over time. All experiments in this section are performed on a desktop computer running Windows 10 with 16GB RAM and an i5 6600k processor. The data is replayed towards the main computer using nfreplay from a laptop running Kali Linux.

Throughput

In order to test the maximum throughput of the Streaming Blue-Fringe in the described setup, it is decided to fill up the buffer of Apache Kafka with flows before starting the Apache Flink job. This ignores possible speed bottlenecks in network infrastructure and in the process of replaying the NetFlow data, while assessing the throughput of the Apache Flink job running the Streaming Blue-Fringe. NetFlows are replayed from the EEMCS Eduroam network from the TU Delft, as listed in table 5.1. Figure 6.1 shows the number of flows processed over time.

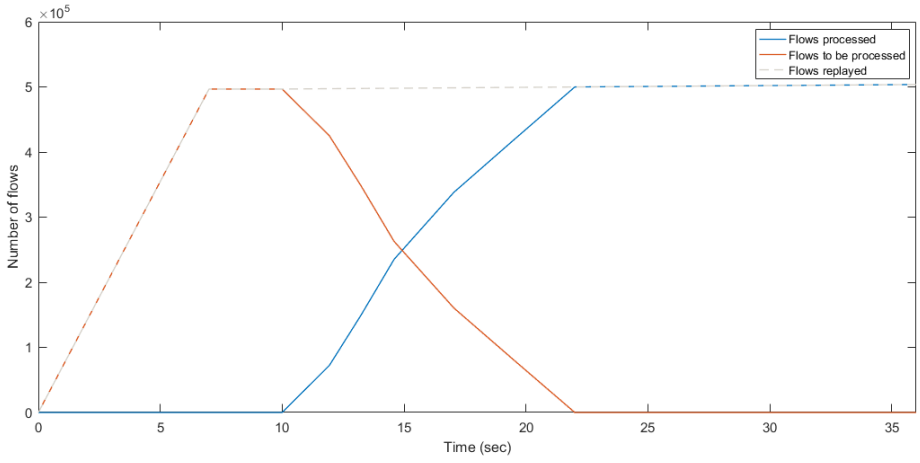


Figure 6.1: A test of the maximum throughput of the Streaming Blue-Fringe. During the first 7 seconds, ~500.000 flows are replayed. At $t = 10$ the Streaming Blue-Fringe becomes active and from $t = 10$ till $t = 22$ the entire backlog of data is processed. The maximum throughput is 41.667 flows/sec.

First, 30 minutes of Eduroam NetFlow data containing roughly 500.000 NetFlows is replayed in 7 seconds, received by the vFlow tool on the desktop PC and stored in the Apache Kafka buffer. After the 7 seconds of replaying at high speed, nfreplay is started with instructions to replay NetFlows in real-time speed. 3 seconds after the burst of high speed NetFlow traffic, the Apache Flink job containing the Streaming Blue-Fringe is started. Immediately, the backlog of flows stored in Kafka is reduced as the Streaming Blue-Fringe starts processing the 30 minutes of network data. During the next 12 seconds, the entire backlog of ~500.000 NetFlows together with all newly incoming flows is processed. In subsequent seconds, the Streaming Blue-Fringe continues processing the real-time incoming flows. From the test can be concluded that the Streaming Blue-Fringe, with the hardware setup, is able to process around 500.000 flows in roughly 12 seconds. This yields a average maximum achievable throughput of 41.667 flows/sec. Compared with the rate at which NetFlows are generated in the KPN dataset this throughput is not sufficient. Table 5.1 showed that the datasample from KPN produces data at around 91.000 flows/sec. However, given the scalable nature of the implementation in Apache Flink, with more nodes this should be easily achievable. When compared with the data rates of the Eduroam traffic from EEMCS, the achieved throughput of 41.667 flows/sec is much higher than the 267 flow/sec generated by the Eduroam traffic. As a result we can conclude that in terms of throughput, the proposed setup is capable of running in smaller organizational networks with commodity hardware. If more resources are needed, multiple Apache Flink nodes can be used.

Memory consumption

Besides the maximum throughput that can be achieved it is also important to know how much resources the solution consumes over time. Since a State Machine is learned for each IP pair combination, the number of State Machines that are kept in memory grows linearly with the number of connections. For large amounts of connections, this can put high demands on the RAM that is available. In order to handle the available resources efficiently, it is decided to clear the State Machine once it has accumulated more states than present in the largest fingerprint. At this point the traffic channel is assumed to either exhibit different behaviour or is already matched against the fingerprints. Throughout the test, the fingerprints of the malware samples from table 6.2 are loaded and used. In order to test the memory consumption and required computation power over time, one day of Eduroam traffic is replayed from 9am till 22pm. The data is replayed at a rate of 1 minute of NetFlow data per second and each 5 seconds the current CPU usage and memory consumption are collected. The resulting values are plotted in figure 6.2. From the graph one can read that the amount of memory increases over time until around 5pm. Probably around that time no more additional devices are connected to the network and more and more State Machines are cleared as they reach more states than the largest fingerprint. Also the CPU usage increases over time. The increase of CPU usage over time could be explained by the following reasons:

1. As State Machines grow more complex, the Red-Blue strategy compares against more red states.
2. Over time more State Machines have become large enough to compare against attack fingerprints.
3. The Java Garbage Collector gets more priority to free memory as more memory is consumed.

It is important to note that the actual CPU usage in real-time performance is less than showed in figure 6.2. This is because during the performance test, data is replayed at 60 times the normal rate.

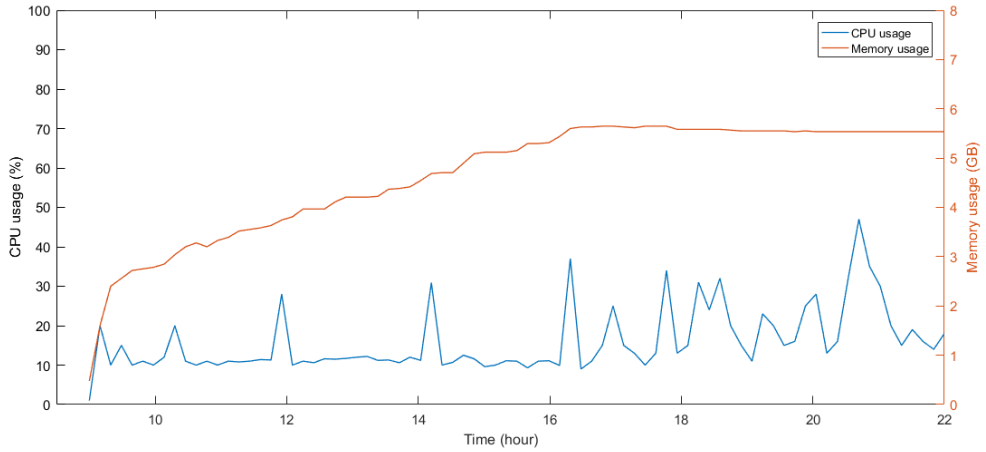


Figure 6.2: The memory consumption and CPU usage when replaying an entire day of Eduroam NetFlow data. The data has been replayed with a rate of 1 minute per second.

Performance conclusion

As displayed in table 5.1, the Eduroam network generates on average 267 flows/sec. The tested setup was able to gain a throughput of 41.667 flows/sec. So, for small and medium sized organizations the Streaming Blue-Fringe in combination with commodity hardware is perfectly able to process the generated NetFlow data in real-time. The much larger KPN network generates on average 91.000 flows/sec. This is more than the used setup is able to process. However, when switching from commodity hardware to a server with more CPU cores and RAM this performance should be perfectly possible to achieve. Moreover, the use of Apache Flink allows for horizontal scalability in which case multiple servers can take parts of the traffic load. This allows to further scale the solution to use cases in which the network cards of single servers are not fast enough to handle all data.

6.2. Effectiveness

In order to determine how effective the detection method is, it is tested on both known malicious traffic and normal traffic in which no attacks are present. A robust method would be able to detect the presence of the malicious behaviour with very little or no alarms on the normal data. Also, the alarms on malicious data should be only for traffic channels that contain the Command & Control traffic. In the malicious samples, other traffic channels are present as a result of other applications and background services that are part of the operating system. In an effective method, these non-malicious traffic channels should not be flagged as malicious.

6.2.1. Normal data

The Stratosphere datasets present a collection of normal traffic samples, capturing non-malicious behaviour. The samples contain different types of normal behaviour and are of a varying duration. It usually takes one or more hours of traffic before the learned State Machine grows big enough to be matched against earlier learned fingerprints. The normal datasets are; however, mostly of short duration compared to the malicious samples. These short normal samples might not contain enough traffic to learn State Machines to the size of the fingerprints. Hence, testing against short normal samples could result in an incorrect sense of good performance. Therefore, only the sets of normal traffic that have more than one hour of traffic are considered in evaluating the effectiveness of the detection method. The normal traffic samples that are selected for evaluation are presented in table 6.1.

CTU ID	Description	Execution duration
Normal-4	DNS traffic	1.9 hours
Normal-5	Computer on university network	4.9 hours
Normal-6	Computer on university network	4.5 hours
Normal-20	Web browsing & background traffic	4 hours
Normal-22	Web browsing & background traffic	1.5 hours
Normal-23	Web browsing & background traffic	2.2 hours
Normal-24	Web browsing & background traffic	4.3 hours
Normal-25	Web browsing & background traffic	1.5 hours
Normal-26	Web browsing & background traffic	6.5 hours
Normal-27	Web browsing & background traffic	4 hours
Normal-29	Web browsing & background traffic	2 hours
Normal-30	Web browsing & background traffic	4 hours
Normal-31	Web browsing & background traffic	3.8 hours
Normal-32	Web browsing & background traffic	4 hours

Table 6.1: Benign traffic samples used to evaluate the number of false alarms.

6.2.2. Malicious data

For creating and testing fingerprints of malicious behaviour, samples from the Stratosphere dataset are selected. Among the Stratosphere samples are numerous malware types that are executed only once or with very large time gaps. At least two samples are needed, one for learning a fingerprint and one for testing. Furthermore, the time gap between the captured samples should not be too large as malware gets regularly updated in an attempt to remain undetected. It is assumed that within one month the behaviour remains more or less equal, to have meaningful comparisons. Also, all samples should have different hash values to avoid biases due to learning and testing on the exactly the same binary. A last prerequisite is that samples are not reused from last chapter, to avoid testing on data that was used to optimize the detection method. In table 6.2 four malware types are shown, that fit these constraints. For each malware type, the first sample will be used as a fingerprint for evaluating the detection mechanism on traffic channels of the subsequent samples.

CTU ID	Malware name	MD5 hash	Execution duration	Fingerprint
170-1	Necurse	1db5333a57f56c4b80bc213ed7675793	5 days	Yes
176-1	Necurse	943a641f4336f919a14bb10cad6daa5e	4 days	No
228-1	Dridex	81e94ac247fecb32add3a666d11beb9e	3 days	Yes
246-1	Dridex	3635ac6099baedae893b3991f730652c	33 days	No
248-1	Dridex	f1d06663a626a7ad7a882f1ddf3734fd	33 days	No
249-1	Dridex	af07a28f2cf91bbf57fd5023ee21b336	23 days	No
264-1	Emotet	28140bd636324bad2f0e8394f3e7f723	19 days	Yes
268-1	Emotet	ac765e9809de73f444cd2cce04256dac	16 days	No
269-1	Emotet	3988863fb18686dc6657245afddb597d	16 days	No
271-1	Emotet	a2350072233e3547a07a2b38509e8711	16 days	No
272-1	Emotet	8a5d3cada819fe7fd9db67d8c0af120e	16 days	No
276-1	Emotet	fa0cea9b855b83dc6a9f8d931882efd2	20 days	No
279-1	Emotet	402d735e59d191b2bde2f5f094688de5	10 days	No
324-1	Trickbot	3d5eeaa64da02d7066e5f57c25368757	4 days	Yes
325-1	Trickbot	011517b0b3c6a79d740033df71120392	33 days	No
327-1	Trickbot	2b48789d9272700de5405bf9a9c05204	34 days	No

Table 6.2: Malware traffic samples used to evaluate the detection performance.

Fingerprints

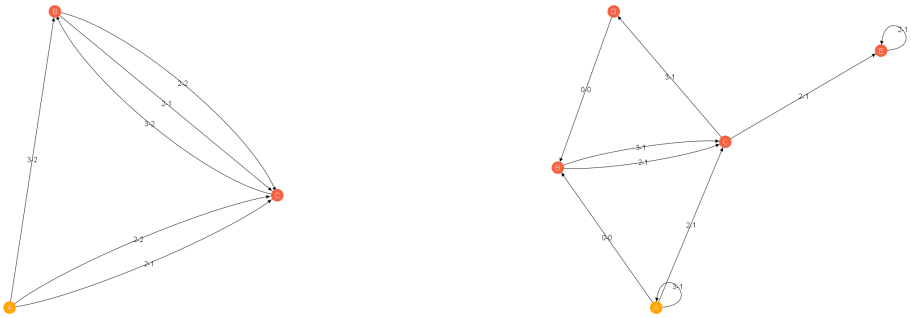
For each of the malware samples labelled as fingerprint in table 6.2, a State Machine is learned for all traffic channels they contain. During the learning process, once a new state is added to a State Machine, the current version of this State Machine is stored to disk. This results in a sequence of State Machines for each traffic channel of a particular malware sample.

In the current learning method not all State Machines converge into a compact model. Some State Machines keep expanding and continue to grow as more data elements are consumed. The traffic channels that do converge into a fixed-sized model, require usually half a day or more network data before the model has reached its final size. Since the selected normal traffic samples from table 6.1 contain traffic of only a few hours, the fingerprints are selected after learning State Machines on network data with durations between 2 hours and one day of network data.

State Machines are learned for each traffic channel and not all of these channels contain malicious behaviour. Therefore, a selection needs to be made to determine which State Machines should be used as fingerprints. To be absolutely certain that a fingerprints model malicious traffic and not benign background behaviour, only traffic channels to Command & Control servers is used. In order to select only Command & Control traffic, first all State Machines are discarded that do not contain traffic from the infected host towards an external IP address. Next, the external IP addresses are looked up in online to check whether it is a known Command & Control server. The following heuristic is manually executed to check whether an IP address can be regarded malicious:

1. The IP address is looked up on ipvoid.com, which checks the IP address against 96 blacklists.
2. The IP address is looked up on feodotracker.abuse.ch, which contains over 1900 known Command & Control servers of Feodo and related malware like Dridex and Emotet.
3. The IP address is looked up on cymon, which is an Open Source Intelligence platform.
4. The IP address is looked up on tracker.h3x.eu, which lists known Command & Control channels of various types of malware. For finding Trickbot servers, this source is especially valuable.
5. The IP address is looked up on virustotal to see whether malware samples have communicated with this server.

After discarding all State Machines of traffic channels other than of Command & Control traffic it becomes apparent that, similar to the findings of section 5.3.3, most malware samples do not communicate with a single Command & Control server. The studied malware variants usually switch to different Command & Control servers throughout their multi-day execution period. Not all of these servers receive equal amounts of traffic and as a result some of the State Machines remain very small. These small State Machines have high probability of introducing false positives, so they are not used. The selected fingerprints are chosen from the more complex State Machines. One fingerprint of each of the malware types from table 6.2 are showed in figure 6.3. On average three State Machines are used for each of the malware types. The State Machines per malware type look quite similar, but not identical.



(a) Dridex fingerprint

(b) Emotet fingerprint

(c) Necurse fingerprint

(d) Trickbot fingerprint

Figure 6.3: Examples of a fingerprint for each type of malware used to evaluate the detection performance.

6.2.3. Detection performance

In order to compute the detection performance, for each malicious sample a list of contacted malicious IP addresses is composed using the heuristic of section 6.2.2. The performance is determined by running the detection algorithm on all samples from table 6.1 and on all samples of table 6.2 that were not used for learning the fingerprints. In total 11 fingerprints resulted from the four malware types, which are all loaded in memory of the Streaming Blue-Fringe. Subsequently, the network traffic of each sample is replayed one by one to the machine running the detection Job. The detection job outputs the State Machine for each channel on which a detection is made, together with the fingerprint that matched with the traffic channel's State Machine. With these results and the known malicious IP's contacted by each of the samples, the performance can be computed.

In order to interpret the significance of the resulting performance, it is compared against a commonly used baseline: a blacklisting approach. The blacklisting approach constructs a list of IP addresses based on the contacted Command & Control servers of the samples used during learning. For each test sample, the number of traffic channels is counted for which the IP address was blacklisted during the training phase.

The described Streaming Blue-Fringe detection and blacklisting approach is now performed for each of the traffic types and oen by one described below.

Necurse

For the Necurse malware, two samples were selected which were both executed in December 2016. The first sample (170-1) was used as a learning sample resulting in three fingerprints. The Command & Control servers contacted during learning are showed in the first row of table 6.3. The table reveals that the evaluation sample (176-1) also contacted three Command & Control servers, of which one server was different. Hence the blacklisting approach detects two out of three channels correctly. On running the Streaming Blue-Fringe in detection mode, all three Command & Control servers are detected without any additional detections. Hence the number of false positives of the Streaming Blue-Fringe is zero. The blacklisting approach can only detect one of the previously seen malicious IP's, so it would never result in false positives.

A summary of the findings on a per traffic channel level is listed in table 6.4. In total 4028 traffic channels are present in the 176-1 sample of which three channels are malicious. The Streaming Blue-Fringe yields a perfect performance whereas the blacklisting approach missed one traffic channel.

Since in practise it is sufficient to detect at least one Command & Control channel to prove that a host is infected, table 6.5 shows the host-level results. Both the Streaming Blue-Fringe and blacklisting approach detected at least one

of the malicious traffic channels from the infected host 10.0.2.106. All other 63 hosts remained correctly undetected, so both methods received optimal detection performance on sample 176-1 when using 170-1 for training.

CTU ID	C&C servers contacted	Blacklist detections	SBF detections	True Positives
170-1 (fingerprint)	91.200.14.80 91.219.31.20 51.254.240.164	Not applicable	Not applicable	Not applicable
176-1	91.200.14.80 164.132.138.50 51.254.240.164	91.200.14.80 51.254.240.164	91.200.14.80 164.132.138.50 51.254.240.164	Blacklist: 2 / 3 SBF: 3 / 3

Table 6.3: Command & Controls channels in the Necurse fingerprinted sample and detections on the test sample.

	True Positives	False Positives	True Negatives	False Negatives
Blacklisting	2	0	4025	1
Streaming Blue-Fringe	3	0	4025	0

Table 6.4: Total performance of the Streaming Blue-Fringe and blacklisting baseline on the traffic channels of sample 176-1.

	True Positives	False Positives	True Negatives	False Negatives
Blacklisting	1	0	63	0
Streaming Blue-Fringe	1	0	63	0

Table 6.5: Performance of the Streaming Blue-Fringe and blacklisting baseline on host level. Only one infected host was present in sample 176-1, of which at least one channel was detected by both methods.

Dridex

Of the Dridex malware four samples were selected: 228-1, 246-1, 248-1 and 249-1. The first sample, containing only 3 days of traffic, was used to create the Dridex fingerprints. The training sample described traffic channels with three malicious hosts, each of which contained enough data to learn a usable fingerprint. With these three fingerprints the Streaming Blue-Fringe is capable of detecting Command & Control channels across each of the test samples.

Table 6.7 shows the performance for both detection methods on a per channel level. Similar to the previous malware type, no misdetections have been made. The Streaming Blue-Fringe has detected six out of eight malicious traffic channels. None of the malicious server addresses from the sample 228-1 can be found in later samples, resulting in a blacklisting performance of zero out of eight. Hence, for this malware sample the Streaming Blue-Fringe shows a clear advantage over blacklisting. Although the IP addresses of the contacted Command & Control servers do not match, the network behaviour across samples proves to be very similar.

On a traffic channel level, the Streaming Blue-Fringe missed two channels. However, when viewed on a host-level, the performance is optimal for the Streaming Blue-Fringe as can be seen in table 6.7. The blacklisting approach did not detect any of the infected hosts.

CTU ID	C&C servers contacted	Blacklist detections	SBF detections	True Positives
228-1 (fingerprint)	8.8.247.90 136.243.209.34 198.167.136.139	Not applicable	Not applicable	Not applicable
246-1	109.74.9.119 203.153.165.21	None	109.74.9.119	Blacklist: 0 / 2 SBF: 1 / 2
248-1	59.125.50.132 89.163.220.168 95.158.148.249	None	59.125.50.132 89.163.220.168 95.158.148.249	Blacklist: 0 / 3 SBF: 3 / 3
249-1	64.250.115.129 217.182.53.102 216.66.0.143	None	64.250.115.129 217.182.53.102	Blacklist: 0 / 3 SBF: 2 / 3

Table 6.6: Command & Controls channels in the Dridex fingerprinted sample and detections on the test samples.

	True Positives	False Positives	True Negatives	False Negatives
Blacklisting	0	0	51	8
Streaming Blue-Fringe	6	0	51	2

Table 6.7: Total performance of the Streaming Blue-Fringe and blacklisting baseline on the traffic channels of all Dridex samples.

	True Positives	False Positives	True Negatives	False Negatives
Blacklisting	0	0	25	3
Streaming Blue-Fringe	3	0	25	0

Table 6.8: Total performance of the Streaming Blue-Fringe and blacklisting baseline on the malicious hosts of all Dridex samples.

Emotet

The Emotet training sample 264-1 connects with three Command & Control servers of which two channels are used to form fingerprints. The traffic to server 178.79.172.45 contains very little flows and as a result the corresponding State Machine reached only two states. The traffic of the two other malicious channels was modelled and used to detect malicious behaviour in the other samples. The tested range of Emotet samples contain connections to a large number of Command & Control servers as is listed in table 6.9.

CTU ID	C&C servers contacted	Blacklist detections	SBF detections	True Positives
264-1 (fingerprint)	87.106.77.193 103.4.18.170 178.79.172.45	Not applicable	Not applicable	Not applicable
268-1	45.79.186.178 87.106.77.193 103.4.18.170 172.93.54.93 178.79.172.45 192.155.83.86	87.106.77.193 103.4.18.170 178.79.172.45	87.106.77.193 103.4.18.170	Blacklist: 3 / 6 SBF: 2 / 6
269-1	45.79.186.178 87.106.77.193 103.4.18.170 172.93.54.93 178.79.172.45 192.155.83.86	87.106.77.193 103.4.18.170 178.79.172.45	87.106.77.193 103.4.18.170	Blacklist: 3 / 6 SBF: 2 / 6
271-1	45.79.186.178 87.106.77.193 103.4.18.170 172.93.54.93 178.79.172.45 192.155.83.86	87.106.77.193 103.4.18.170 178.79.172.45	87.106.77.193 103.4.18.170	Blacklist: 3 / 6 SBF: 2 / 6
272-1	45.79.186.178 87.106.77.193 103.4.18.170 172.93.54.93 178.79.172.45 192.155.83.86	87.106.77.193 103.4.18.170 178.79.172.45	87.106.77.193 103.4.18.170	Blacklist: 3 / 6 SBF: 2 / 6
276-1	45.79.186.178 85.25.119.91 87.106.77.193 103.4.18.170 109.169.66.107 139.59.33.202 178.79.172.45 192.155.83.86	87.106.77.193 103.4.18.170 178.79.172.45	87.106.77.193 103.4.18.170 109.169.66.107	Blacklist: 3 / 8 SBF: 3 / 8
279-1	89.231.13.18 89.231.13.27 89.231.13.33 212.24.109.200 212.24.109.218 212.24.110.1 212.24.110.154 212.24.110.190	None	None	Blacklist: 0 / 8 SBF: 0 / 8

Table 6.9: Command & Controls channels in the Emotet fingerprinted sample and detections on the test samples.

The first five tested samples contained traffic to all blacklisted servers, yielding three out of six and once a three out of eight detection performance. The Streaming Blue-Fringe again struggled to learn matchable State Machines for server 178.79.172.45. Also the newly contacted Command & Control servers in the first five test samples remained undetected. Manual inspection revealed that also these traffic channels contained very little traffic which resulted in very small and unmatched State Machines. The last test sample (279-1) contained no overlap in terms of contacted servers. The State Machines learned from this sample show similar State Machines as the fingerprints, but with different symbols on the transitions. The first part of each symbol, indicating the average packet size, was larger compared to the fingerprints. This shows that while the malware still uses the same client-server communication protocol, data has been downloaded or exfiltrated at a higher rate. An interesting finding in table 6.9 is that in sample 276-1 the Streaming Blue-Fringe correctly detected traffic to the Command & Control server 109.168.66.107. This host was not blacklisted, confirming yet again that the Streaming Blue-Fringe is capable of detecting similar behaviour to newly encountered hosts.

When combining the channel-level results of all traffic samples, the results of table 6.11 are obtained. As the channels to server 178.79.172.45 remained undetected and the Streaming Blue-Fringe did only manage to obtain one additional detection, the blacklisting received a higher performance. Both methods again resulted in 0 false positives.

	True Positives	False Positives	True Negatives	False Negatives
Blacklisting	15	0	144	25
Streaming Blue-Fringe	11	0	144	29

Table 6.10: Total performance of the Streaming Blue-Fringe and blacklisting baseline on the traffic channels of all Emotet samples.

Although the Streaming Blue-Fringe detected less malicious traffic channels compared to the baseline approach, table 6.11 shows that the host-level performance is identical. Both methods detected at least one Command & Control traffic channel of the malicious host in each of the first five samples. The infected host in the last sample remained fully undetected. The absence of any false positives is reflected in a total of 72 correctly undetected hosts.

	True Positives	False Positives	True Negatives	False Negatives
Blacklisting	5	0	72	1
Streaming Blue-Fringe	5	0	72	1

Table 6.11: Total performance of the Streaming Blue-Fringe and blacklisting baseline on the malicious hosts of all Emotet samples.

Trickbot

The Trickbot samples connect to the largest number of Command & Control servers of all studied malware types. The test samples 325-1 and 327-1 connect to 18 and 21 malicious servers respectively. The training sample; however, shows only three connections which are listed in the first row of table 6.12. This is explained by the significantly shorter execution time of only four days compared to 33 and 34 days for the other samples. As a result the blacklisting approach learns only three Command & Control servers. These servers are not found in any of the test samples, resulting in the worst-case performance of zero detections.

CTU ID	C&C servers contacted	Blacklist detections	SBF detections	True Positives
324-1 (fingerprint)	95.154.199.237 95.213.195.169 141.255.167.124	Not applicable	Not applicable	Not applicable
325-1	5.200.55.47 37.60.177.19 79.106.41.9 80.87.198.204 82.146.48.44 82.146.48.241 82.202.236.84 94.250.253.142 94.250.255.50 95.154.199.136 179.43.160.45 194.87.93.30 194.87.93.84 194.87.94.225 194.87.146.14 195.62.53.88 195.88.209.128 195.133.147.140	None	37.60.177.19 80.87.198.204 82.146.48.44 82.146.48.241 82.202.236.84 94.250.253.142 94.250.255.50 95.154.199.136 179.43.160.45 194.87.93.30 194.87.93.84 194.87.94.225 194.87.146.14 195.62.53.88 195.88.209.128 195.133.147.140	Blacklist: 0 / 18 SBF: 16 / 18
327-1	80.87.198.204 80.87.199.190 82.146.48.44 82.146.48.241 82.202.226.138 92.53.66.199 92.53.91.20 94.250.253.142 95.154.199.120 95.154.199.136 95.213.191.30 177.251.27.6 194.87.93.30 194.87.93.84 194.87.94.225 194.87.103.78 194.87.146.14 194.87.239.201 95.62.53.88 195.88.209.128 195.133.147.140	None	80.87.198.204 80.87.199.190 82.146.48.44 82.146.48.241 92.53.66.199 94.250.253.142 95.213.191.30 194.87.93.30 194.87.93.84 194.87.94.225 194.87.103.78 194.87.146.14 194.87.239.201 95.62.53.88 195.88.209.128 195.133.147.140	Blacklist: 0 / 21 SBF: 16 / 21

Table 6.12: Command & Controls channels in the Trickbot fingerprinted sample and detections on the test sample.

The second test sample (327-1) contacts about 50% new servers, so even a longer duration training sample would most likely not yield a good performance for the blacklisting method. The Streaming Blue-Fringe, on the other hand, is clearly not impaired by the significantly shorter amount of training data and the high percentage of newly contacted servers. On both test samples, 16 channels are correctly identified. This is respectively 89% and 76% of all malicious channels present in the first and second test sample.

	True Positives	False Positives	True Negatives	False Negatives
Blacklisting	0	0	306	39
Streaming Blue-Fringe	32	0	306	7

Table 6.13: Total performance of the Streaming Blue-Fringe and blacklisting baseline on the traffic channels of all Trickbot samples.

Table 6.13 shows a combined overview of the channel-level performance of both methods. Both detections methods have zero false positives, but the Streaming Blue-Fringe shows a clear advantage by detecting 32 out of 39 Command & Control channels.

When inspecting the results at a host-level, the Streaming Blue-Fringe reaches the optimal performance, as showed in table 6.14.

	True Positives	False Positives	True Negatives	False Negatives
Blacklisting	0	0	184	2
Streaming Blue-Fringe	2	0	184	0

Table 6.14: Total performance of the Streaming Blue-Fringe and blacklisting baseline on the malicious hosts of all Trickbot samples.

Normal data

Table 6.1 lists a series of normal traffic samples, capturing typical user behaviour. The samples contain DNS traffic and browsing behaviour to a large number of the Alexa top 1,000 websites. Also traffic is present of various applications and services running in the background while collecting the normal behaviour.

The fingerprints of all four malware types used for testing are again loaded into memory of the Streaming Blue-Fringe. The normal samples are replayed one by one in order to find possible false positives. Manual inspection of the learned State Machines show that even though the amount of normal traffic is in the order of hours, the State Machines are complex enough to allow comparisons with the pre-learned fingerprints. After replaying all normal traffic samples, the result is very positive. Similar to running the Streaming Blue-Fringe on the malware samples, not any erroneous detection is made on all fourteen normal samples.

Combined results

On combining the channel-level performance of all four malware types for both detection methods, the results of table 6.15 can be obtained. The table does not contain the performance in terms of false positives, since this was 0% for both detection methods. The percentages are based on the total number of correctly detected Command & Control channels divided by the total number of Command & Control channels. The bottom row describes the average performance over all malware types. It is decided to use equal weights, even though not all types contained an equal number of test samples. The intuition behind this decision is that each malware types seems to perform equal across all its samples. Hence, the number of tested samples does not have a large impact on the precision. Moreover, a single badly performing and abundant malware type should not have a large negative impact on the overall performance as the ability to detect each malware type is considered equally important.

The averages presented in the channel-level overview show that the Streaming Blue-Fringe performs significantly better than the baseline method.

	Correct blacklist detections	Correct SBF detections
Necurse	66.7%	100%
Dridex	0%	75%
Emotet	37.5%	27.5%
Trickbot	0%	82%
Average	26%	71%

Table 6.15: The percentages of correct detections for both detection methods over all malware types. The percentages are based on the number of correctly detected malicious traffic channels.

When summarizing the detection performance on a host-level, the performance of both detection methods increase significantly. Table 6.16 shows that the Streaming Blue-Fringe reaches an exceptional 95.8% detection performance, whereas the baseline method does not manage to detect half of all infected hosts.

	Correct blacklist detections	Correct SBF detections
Necurse	100%	100%
Dridex	0%	100%
Emotet	83.3%	83.3%
Trickbot	0%	100%
Average	45.8%	95.8%

Table 6.16: The percentages of correct detections for both detection methods over all malware types. The percentages are based on the number of correctly identified infected hosts.

6.2.4. Analysis of false negatives

The results of section 6.2.3 show that in some samples a number of Command & Control channels are not detected. Studying why these samples remain undetected is important to gain a better understanding of possible drawbacks of the detection method and how the method could be improved upon.

Dridex

At first we will have a look at the Dridex malware samples for which two traffic channels remained undetected. The three models learned from the malicious traffic in training sample 228 all are very similar to the fingerprint displayed in figure 6.3a. This figure shows the model learned from traffic to 8.8.247.90, of which a packet trace is listed in figure 6.4. The trace shows that a TLS connection is set-up, after which data is transferred between the infected host and the Command & Control server.

ip.addr == 8.8.247.90

No.	Time	Source	Destination	Protocol	Length	Info
1490	697.327350	192.168.1.123	8.8.247.90	TCP	66	49158 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
1491	697.327419	8.8.247.90	192.168.1.123	TCP	66	443 → 49158 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
1492	697.327506	192.168.1.123	8.8.247.90	TCP	54	49158 → 443 [ACK] Seq=1 Ack=1 Win=65700 Len=0
1493	697.447724	192.168.1.123	8.8.247.90	TLSv1.2	182	Client Hello
1494	697.447844	8.8.247.90	192.168.1.123	TCP	54	443 → 49158 [ACK] Seq=1 Ack=129 Win=30336 Len=0
1496	697.998442	8.8.247.90	192.168.1.123	TLSv1.2	1514	Server Hello
1497	697.998484	8.8.247.90	192.168.1.123	TLSv1.2	733	Certificate, Server Key Exchange, Server Hello Done
1498	697.998595	192.168.1.123	8.8.247.90	TCP	54	49158 → 443 [ACK] Seq=129 Ack=2140 Win=65700 Len=0
1507	698.328775	192.168.1.123	8.8.247.90	TLSv1.2	220	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
1508	698.328922	8.8.247.90	192.168.1.123	TCP	54	443 → 49158 [ACK] Seq=2140 Ack=295 Win=31360 Len=0
1509	698.329576	8.8.247.90	192.168.1.123	TLSv1.2	145	Change Cipher Spec, Encrypted Handshake Message
1511	698.517849	192.168.1.123	8.8.247.90	TLSv1.2	699	Application Data
1512	698.554988	8.8.247.90	192.168.1.123	TCP	54	443 → 49158 [ACK] Seq=2231 Ack=940 Win=32640 Len=0
1515	699.250638	8.8.247.90	192.168.1.123	TLSv1.2	283	Application Data
1516	699.250947	8.8.247.90	192.168.1.123	TLSv1.2	192	Application Data, Encrypted Alert
1517	699.251097	192.168.1.123	8.8.247.90	TCP	54	49158 → 443 [ACK] Seq=940 Ack=2599 Win=65240 Len=0
1518	699.271854	192.168.1.123	8.8.247.90	TCP	54	49158 → 443 [FIN, ACK] Seq=940 Ack=2599 Win=65240 Len=0
1519	699.271967	8.8.247.90	192.168.1.123	TCP	54	443 → 49158 [ACK] Seq=2599 Ack=941 Win=32640 Len=0

Figure 6.4: A packet trace of traffic to Dridex C&C server 8.8.247.90 captured in sample 228-1. The trace shows communication over TLSv1.2.

In order to examine why two traffic channels remained undetected, the packet traces and learned models of these channels are analysed. The first undetected Command & Control traffic was a traffic channel to server 203.153.165.21 by sample 246-1. Figure 6.5 shows a packet trace for this traffic channel. The trace reveals traffic over TLS which looks similar to the trace of figure 6.4. Throughout communication attempts within each sample, the handshake resulted in consistent packet sizes. However, the packet sizes differ between the fingerprinted channels and the channel to server 203.153.165.21. Closer inspection shows that traffic to 203.153.165.21 uses TLSv1 compared to TLSv1.2 used in the fingerprinted Dridex communication. The traffic to 203.153.165.21 shows a smaller Cipher Spec and a different connection breakdown. These differences are expected to result from the different TLS versions, although this was not further analysed. By scanning through the PCAP of traffic to server 203.153.165.21, another finding was made. Traffic to server 203.153.165.21 shows chains of packets being retransmitted and packets arriving out of order. The fingerprinted traffic only occasionally shows a single packet retransmission. Both the difference in retransmissions and the difference in handshake caused flows to server 203.153.165.21 to map to different symbols compared to the fingerprints. This becomes visible by visualising the model learned on traffic to server 203.153.165.21 in figure 6.6. This model shows a larger variation in the number of state transitions and differences in transition symbols.

ip.addr == 203.153.165.21 && not icmp

No.	Time	Source	Destination	Protocol	Length	Info
34	166.015245	192.168.1.110	203.153.165.21	TCP	66	49191 → 8343 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
38	166.394969	203.153.165.21	192.168.1.110	TCP	66	8343 → 49191 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460 SACK_PERM=1 WS=128
39	166.395118	192.168.1.110	203.153.165.21	TCP	54	49191 → 8343 [ACK] Seq=1 Ack=1 Win=65700 Len=0
42	166.435558	192.168.1.110	203.153.165.21	TLSv1	182	Client Hello
43	166.815192	203.153.165.21	192.168.1.110	TCP	60	8343 → 49191 [ACK] Seq=1 Ack=129 Win=15744 Len=0
44	166.816084	203.153.165.21	192.168.1.110	TLSv1	1188	Server Hello, Certificate, Server Hello Done
45	166.824574	192.168.1.110	203.153.165.21	TLSv1	380	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
47	167.216661	203.153.165.21	192.168.1.110	TLSv1	113	Change Cipher Spec, Encrypted Handshake Message
49	167.417760	192.168.1.110	203.153.165.21	TCP	54	49191 → 8343 [ACK] Seq=455 Ack=1194 Win=64504 Len=0
114	171.269223	192.168.1.110	203.153.165.21	TLSv1	731	Application Data
116	171.689530	203.153.165.21	192.168.1.110	TCP	60	8343 → 49191 [ACK] Seq=1194 Ack=1132 Win=18176 Len=0
119	174.471869	203.153.165.21	192.168.1.110	TLSv1	267	Application Data
120	174.471903	203.153.165.21	192.168.1.110	TCP	60	8343 → 49191 [FIN, ACK] Seq=1407 Ack=1132 Win=18176 Len=0
121	174.472074	192.168.1.110	203.153.165.21	TCP	54	49191 → 8343 [ACK] Seq=1132 Ack=1408 Win=64292 Len=0
122	174.489021	192.168.1.110	203.153.165.21	TCP	54	49191 → 8343 [FIN, ACK] Seq=1132 Ack=1408 Win=64292 Len=0
123	174.868604	203.153.165.21	192.168.1.110	TCP	60	8343 → 49191 [ACK] Seq=1408 Ack=1133 Win=18176 Len=0

Figure 6.5: A packet trace of undetected traffic to Dridex C&C server 203.153.165.21 captured in sample 246-1. The trace shows communication over TLSv1.

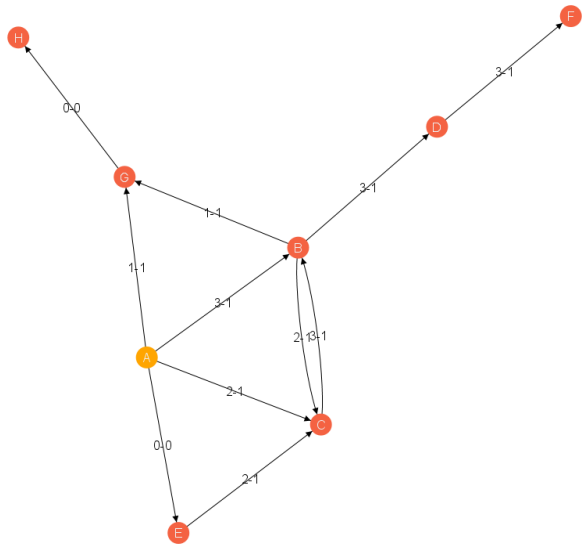


Figure 6.6: The model learned from undetected traffic to Dridex C&C server 203.153.165.21 captured in sample 246-1.

The other Dridex traffic channel that remained undetected was traffic to 216.66.0.143 present in sample 249-1. Figure 6.7 shows a packet trace for the undetected traffic channel to server 216.66.0.143. The trace shows TCP traffic between the infected host and the Command & Control server. The packet trace clearly deviates from the trace shown in figure 6.4 describing TLS traffic. The model learned from the traffic channel to server 216.66.0.143 is visualised in figure 6.8. As one would expect, the difference in communication resulted in a different model when compared to the fingerprint of figure 6.3a. From this examination one can conclude that the Dridex instance in sample 249-1 showed network behaviour which was not present and different from the training sample. The difference in behaviour resulted in a different model and hence no detection was made.

ip.addr == 216.66.0.143 && not icmp

No.	Time	Source	Destination	Protocol	Length	Info
146	441.643096	192.168.1.130	216.66.0.143	TCP	66	49161 → 5353 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
147	441.750654	216.66.0.143	192.168.1.130	TCP	66	5353 → 49161 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
148	441.750833	192.168.1.130	216.66.0.143	TCP	54	49161 → 5353 [ACK] Seq=1 Ack=1 Win=65700 Len=0
151	441.751596	192.168.1.130	216.66.0.143	TCP	182	49161 → 5353 [PSH, ACK] Seq=1 Ack=1 Win=65700 Len=128 [TCP segment of a reassembled PDU]
152	441.858677	216.66.0.143	192.168.1.130	TCP	60	5353 → 49161 [ACK] Seq=1 Ack=129 Win=30336 Len=0
153	441.859202	216.66.0.143	192.168.1.130	TCP	1060	5353 → 49161 [PSH, ACK] Seq=1 Ack=129 Win=30336 Len=1006 [TCP segment of a reassembled PDU]
154	441.859936	192.168.1.130	216.66.0.143	TCP	412	49161 → 5353 [PSH, ACK] Seq=129 Ack=1007 Win=64692 Len=358 [TCP segment of a reassembled PDU]
156	441.973323	216.66.0.143	192.168.1.130	TCP	145	5353 → 49161 [PSH, ACK] Seq=1007 Ack=487 Win=31360 Len=91 [TCP segment of a reassembled PDU]
157	441.975202	192.168.1.130	216.66.0.143	TCP	715	49161 → 5353 [PSH, ACK] Seq=487 Ack=1098 Win=64600 Len=661 [TCP segment of a reassembled PDU]
158	442.121882	216.66.0.143	192.168.1.130	TCP	60	5353 → 49161 [ACK] Seq=1098 Ack=1148 Win=32768 Len=0
159	442.521668	216.66.0.143	192.168.1.130	TCP	299	5353 → 49161 [PSH, ACK] Seq=1098 Ack=1148 Win=32768 Len=245 [TCP segment of a reassembled PDU]
160	442.521698	216.66.0.143	192.168.1.130	TCP	60	5353 → 49161 [FIN, ACK] Seq=1343 Ack=1148 Win=32768 Len=0
161	442.521864	192.168.1.130	216.66.0.143	TCP	54	49161 → 5353 [ACK] Seq=1148 Ack=1344 Win=64356 Len=0
162	442.522095	192.168.1.130	216.66.0.143	TCP	54	49161 → 5353 [FIN, ACK] Seq=1148 Ack=1344 Win=64356 Len=0
163	442.629474	216.66.0.143	192.168.1.130	TCP	60	5353 → 49161 [ACK] Seq=1344 Ack=1149 Win=32768 Len=0

Figure 6.7: A packet trace of undetected traffic to Dridex C&C server 216.66.0.143 captured in sample 249-1. The trace shows TCP traffic between the infected host and C&C server.

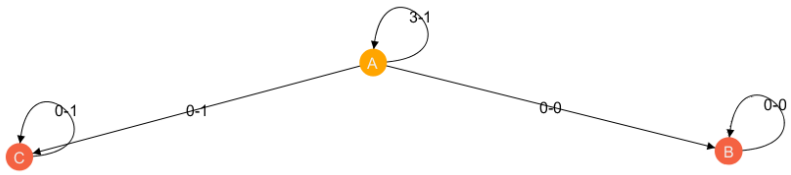


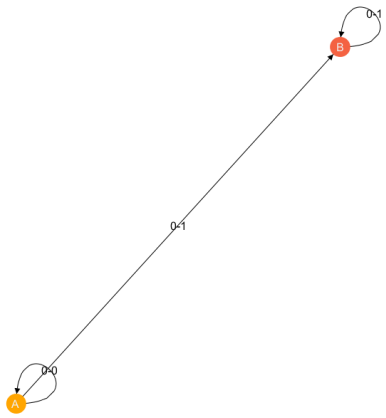
Figure 6.8: The model learned from undetected traffic to Dridex C&C server 216.66.0.143 captured in sample 249-1.

The causes of false negatives in the Dridex samples can be summarized by:

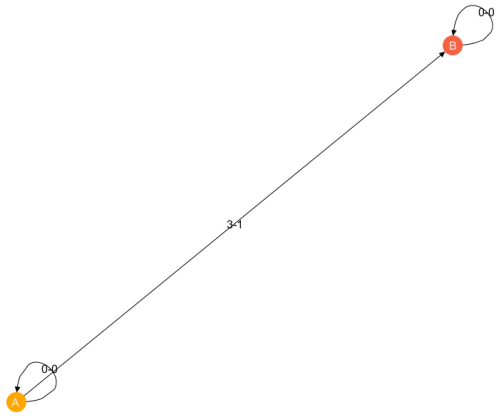
- Problems in the connection, resulting in chains of retransmissions.
- A difference in TLS protocol version.
- The use of an additional communication protocol, unseen in the training sample.

Emotet

We will now have a detailed look at the Emotet malware samples. From the Emotet samples, a large portion of the malicious traffic channels remained undetected. In order to determine what causes this moderate performance, the models of the traffic channels across all Emotet samples are visualised. Manual inspection revealed that all undetected malicious traffic channels from each sample, except 279-1, shows similar behaviour. On comparing these undetected models with the State Machines learned on the training data, an interesting discovery is made. The training sample 264-1 contained only three traffic channels to Command & Control servers. From these channels, traffic to server 178.79.172.45 resulted in a State Machine consisting of only 2 states. The corresponding model is visualised in figure 6.9a. Since this trivial model can only produce a small number of unique traces, using this model as fingerprint was expected to have high chances of introducing false positives. Therefore, it was decided to not include this model as one of the Emotet fingerprints. However, it turns out that 21 out of 29 undetected channels resulted in a similar two-state model. An example is traffic to the undetected server 45.79.186.178 from sample 268-1. The corresponding model is visualised in figure 6.9b.



(a) Model of traffic in sample 264-1 to server 178.79.172.45



(b) Model of traffic in sample 268-1 to server 45.79.186.178

Figure 6.9: Difference between the not selected Emotet traffic channel of training sample 264-1 and an undetected traffic channel from sample 268-1. The models show identical structure, although the symbols for state transitions differ.

By taking a closer look at the models of figure 6.9, one can see that, although the structure is identical, both models have different state transitions. To determine the cause of this deviation, the PCAP files are examined. The packet trace in figure 6.10 reveals that the server 178.79.172.45 was not reachable in the training sample. The packet trace to server 45.79.186.178 displayed in figure 6.11 do show responses. Since in this case the server did respond to the initial SYN packet, the client performed its HTTP GET request. Because of the HTTP GET request, NetFlows to 45.79.186.178 yield a higher average packet size. This resulted in a state transition with symbol 3-1 compared to a transition with symbol 0-1 for the State Machine of the training sample. From the 21 traffic channels with similar

models to the not selected traffic channel from the training sample, 2 were identical to figure 6.9a and 19 were equal to figure 6.9b.

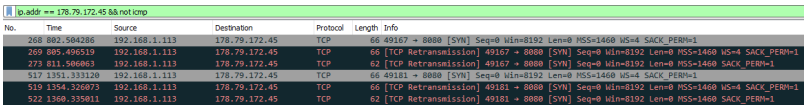


Figure 6.10: A packet trace of traffic to Emotet C&C server 178.79.172.45 captured in sample 264-1. The trace shows that the server is unreachable.

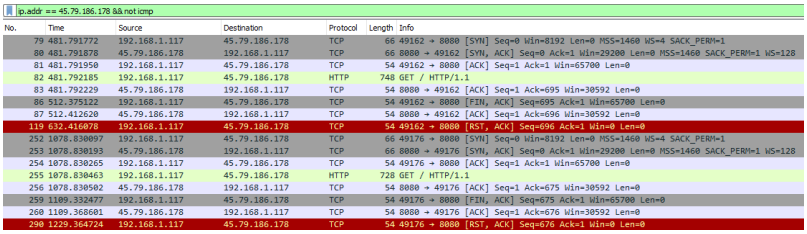


Figure 6.11: A packet trace of traffic to Emotet C&C server 45.79.186.178 captured in sample 268-1. The trace shows a successful TCP handshake after which the client performs an HTTP GET request.

From this examination, one can conclude that the majority of undetected Emotet channels yield models regarded too trivial to be used as a robust fingerprint. Including one of these models as fingerprint would have increased the detection rate, possibly at the expense of some false positives. In this specific example; however, including the model to server 178.79.172.45 from sample 264-1 would only have slightly improved performance, as it was modelling failing connections.

Not selecting a fingerprint matching with these smaller models explains the moderate performance of samples 268-1 till 276-1. Sample 279-1; however, requires special examination. Sample 279-1 resulted in models that look substantially different from all models of other samples. One example is the traffic channel to server 212.24.110.1 as visualised in figure 6.12. This looks different to both the selected fingerprint in figure 6.3b and the models of figure 6.9. All Emotet channels of sample 279-1 show a model similar to figure 6.12. Again a PCAP examination is performed to find the cause of the model differences. Figure 6.13 lists the packet trace of a typical connection to 212.24.110.1 as captured in sample 279-1. Figure 6.14 lists a packet trace for a connection to server 103.4.18.170 used in creating the fingerprint. The difference is very clear, the communication in sample 279-1 is performed over TLS. This results in different amounts of packets and with different sizes compared to the fingerprinted traffic channel. This explains why the sample 279-1 resulted in zero detections. Probably a new version of Emotet was spreading which exclusively operates over TLS.

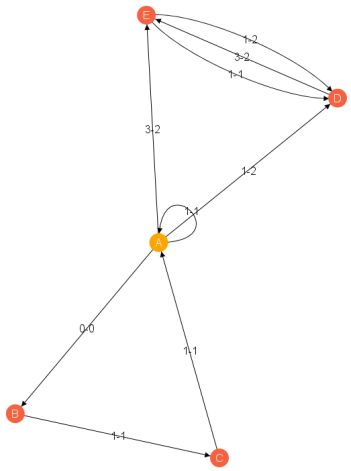


Figure 6.12: The model learned from traffic to Emotet C&C server 212.24.110.1 captured in sample 279-1.

ip.addr == 212.24.110.1

No.	Time	Source	Destination	Protocol	Length	Info
1040	3202.501626	192.168.1.130	212.24.110.1	TCP	66	49189 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
1041	3202.512717	212.24.110.1	192.168.1.130	TCP	66	443 → 49189 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
1042	3202.512858	192.168.1.130	212.24.110.1	TCP	54	49189 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
1043	3202.827611	192.168.1.130	212.24.110.1	TLSv1	144	Client Hello
1044	3202.827828	212.24.110.1	192.168.1.130	TCP	54	443 → 49189 [ACK] Seq=1 Ack=91 Win=29312 Len=0
1062	3330.078983	212.24.110.1	192.168.1.130	TLSv1	1514	Server Hello
1063	3330.079069	212.24.110.1	192.168.1.130	TLSv1	705	Certificate, Server Key Exchange, Server Hello Done
1064	3330.079182	192.168.1.130	212.24.110.1	TCP	54	49189 → 443 [ACK] Seq=91 Ack=2112 Win=65536 Len=0
1065	3330.099855	192.168.1.130	212.24.110.1	TLSv1	188	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
1066	3330.099947	212.24.110.1	192.168.1.130	TCP	54	443 → 49189 [ACK] Seq=2112 Ack=225 Win=30336 Len=0
1067	3330.100414	212.24.110.1	192.168.1.130	TLSv1	113	Change Cipher Spec, Encrypted Handshake Message
1068	3330.101347	192.168.1.130	212.24.110.1	TLSv1	331	Application Data
1069	3330.101727	212.24.110.1	192.168.1.130	TLSv1	608	Application Data, Application Data
1070	3330.101903	192.168.1.130	212.24.110.1	TLSv1	91	Encrypted Alert
1071	3330.102039	192.168.1.130	212.24.110.1	TCP	54	49189 → 443 [FIN, ACK] Seq=539 Ack=2725 Win=65024 Len=0
1072	3330.102083	212.24.110.1	192.168.1.130	TLSv1	91	Encrypted Alert
1073	3330.102370	212.24.110.1	192.168.1.130	TCP	54	443 → 49189 [FIN, ACK] Seq=2762 Ack=540 Win=31360 Len=0
1074	3330.102461	192.168.1.130	212.24.110.1	TCP	54	49189 → 443 [ACK] Seq=540 Ack=2763 Win=65024 Len=0

Figure 6.13: A packet trace of traffic to Emotet C&C server 212.24.110.1 captured in sample 279-1. The trace shows communication over TLSv1.

ip.addr == 103.4.18.170 && not icmp

No.	Time	Source	Destination	Protocol	Length	Info
159	580.825536	192.168.1.113	103.4.18.170	TCP	66	49161 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
160	581.164603	103.4.18.170	192.168.1.113	TCP	66	443 → 49161 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460 SACK_PERM=1 WS=128
161	581.164814	192.168.1.113	103.4.18.170	TCP	54	49161 → 443 [ACK] Seq=1 Ack=1 Win=65700 Len=0
163	581.165177	192.168.1.113	103.4.18.170	HTTP	746	GET / HTTP/1.1
164	581.504004	103.4.18.170	192.168.1.113	TCP	60	443 → 49161 [ACK] Seq=1 Ack=693 Win=16000 Len=0
165	581.504528	103.4.18.170	192.168.1.113	HTTP	616	HTTP/1.1 400 Bad Request (text/html)
166	581.504542	103.4.18.170	192.168.1.113	TCP	60	443 → 49161 [FIN, ACK] Seq=563 Ack=693 Win=16000 Len=0
167	581.504747	192.168.1.113	103.4.18.170	TCP	54	49161 → 443 [ACK] Seq=693 Ack=564 Win=65136 Len=0
168	581.504922	192.168.1.113	103.4.18.170	TCP	54	49161 → 443 [FIN, ACK] Seq=693 Ack=564 Win=65136 Len=0
169	581.843802	103.4.18.170	192.168.1.113	TCP	60	443 → 49161 [ACK] Seq=564 Ack=694 Win=16000 Len=0
394	1129.805448	192.168.1.113	103.4.18.170	TCP	66	49175 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
395	1130.131750	103.4.18.170	192.168.1.113	TCP	66	443 → 49175 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460 SACK_PERM=1 WS=128
396	1130.131949	192.168.1.113	103.4.18.170	TCP	54	49175 → 443 [ACK] Seq=1 Ack=1 Win=65700 Len=0
398	1130.132269	192.168.1.113	103.4.18.170	HTTP	726	GET / HTTP/1.1
399	1130.458319	103.4.18.170	192.168.1.113	TCP	60	443 → 49175 [ACK] Seq=1 Ack=673 Win=16000 Len=0
400	1130.458514	103.4.18.170	192.168.1.113	HTTP	616	HTTP/1.1 400 Bad Request (text/html)
401	1130.458535	103.4.18.170	192.168.1.113	TCP	60	443 → 49175 [FIN, ACK] Seq=563 Ack=673 Win=16000 Len=0
402	1130.458656	192.168.1.113	103.4.18.170	TCP	54	49175 → 443 [ACK] Seq=673 Ack=564 Win=65136 Len=0
403	1130.458798	192.168.1.113	103.4.18.170	TCP	54	49175 → 443 [FIN, ACK] Seq=673 Ack=564 Win=65136 Len=0
404	1130.784727	103.4.18.170	192.168.1.113	TCP	60	443 → 49175 [ACK] Seq=564 Ack=674 Win=16000 Len=0

Figure 6.14: A packet trace of traffic to Emotet C&C server 103.4.18.170 captured in sample 264-1. The trace shows HTTP traffic to port 443.

The causes of false negatives in the Emotet samples can be summarized by:

- The decision to exclude a small model as fingerprint because it is expected to result in false positives.
- The training sample contains communication attempts in which the server is unreachable.
- A newer version of the malware communicates over TLS, whereas the training sample did not.

Trickbot

The final malware type with false negatives is Trickbot. The Streaming Blue-Fringe is again used to learn models of all traffic channels after which the undetected channels are selected and manually inspected. Of the seven undetected traffic channels, two models turned out to be identical to the model of figure 6.9a. This figure illustrated a model of a failing TCP handshake. After inspection of the PCAP's, it was clear that these two undetected channels show identical behaviour to the trace of figure 6.10. The two models indeed described failing TCP handshakes as a result of an unreachable server.

The other five undetected channels show models very similar to the learned fingerprints. All five models turned out to have the same cause for being undetected so only one traffic channel will be further examined. Figure 6.15 illustrates the model learned from undetected traffic to server 92.53.91.20 captured in sample 327-1. The model has similar transitions and the structure is similar to the fingerprint of figure 6.3d. The major difference is that the traffic to 92.53.91.20 resulted in transitions with symbol 2-1 which are not present in the fingerprinted channels. To recall from table 5.3, symbol 2-1 means flows with an average packet size between 149 and 221 bytes. In order to get an average packet size between 149 and 221 bytes, at least one packet must be equal or larger than 149 bytes. A filter query on the PCAP of sample 327-1 results in zero packets with a size greater than or equal to 149 bytes. This shifts the focus from packet inspection to analysing the flows that summarized the traffic.

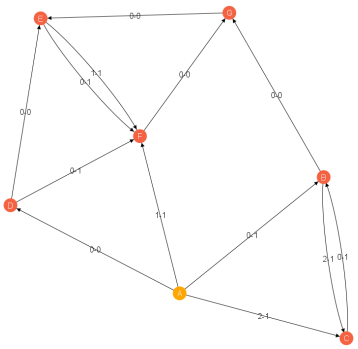


Figure 6.15: The model learned from traffic to Trickbot C&C server 92.53.91.20 captured in sample 327-1.

Table 6.17 lists the first 10 NetFlows from connections to server 95.213.195.169 used to create the Trickbot fingerprint of figure 6.3d. The table shows flows originating from both sides of the connection. Since the models are learned from traffic from the client to the server, the third and seventh line can be ignored. The highest average byte size in the resulting flows is the first flow with an average of 88 bytes.

Time	Protocol	Source	Destination	Packets transferred	Bytes transferred
01:09:23.925052	TCP	192.168.1.123	95.213.195.169	3	264
01:09:23.925171	TCP	192.168.1.123	95.213.195.169	2	120
01:11:33.495297	TCP	95.213.195.169	192.168.1.123	2	597
01:11:33.495672	TCP	192.168.1.123	95.213.195.169	2	108
01:11:33.496114	TCP	192.168.1.123	95.213.195.169	3	226
01:11:33.496187	TCP	192.168.1.123	95.213.195.169	2	116
01:13:44.565341	TCP	95.213.195.169	192.168.1.123	2	597
01:13:44.565737	TCP	192.168.1.123	95.213.195.169	2	108
01:14:05.569950	TCP	192.168.1.123	95.213.195.169	3	230
01:14:05.570040	TCP	192.168.1.123	95.213.195.169	2	120

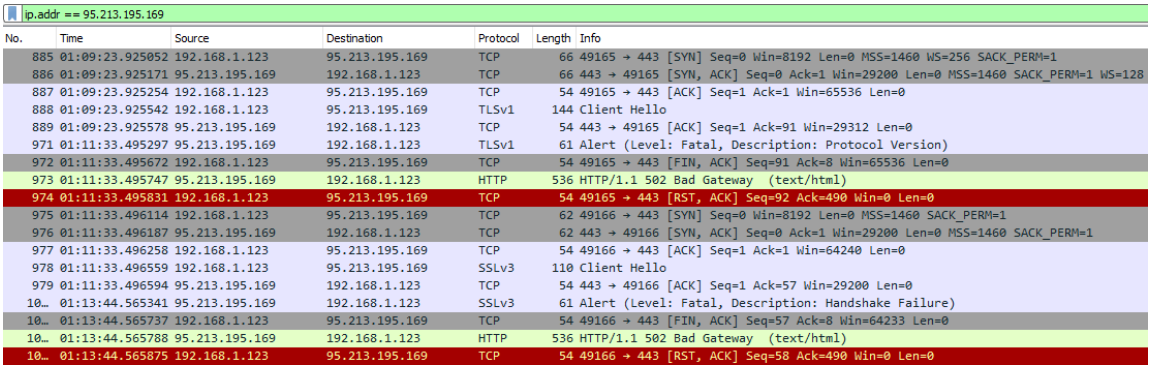
Table 6.17: NetFlows of communication with Trickbot C&C server 95.213.195.169 captured in sample 324-1.

Table 6.18 lists the first 10 NetFlows from connections to server 92.53.91.20 which remained undetected in sample 327-1. It becomes apparent that only flows from the infected host to the server are collected. The table also shows that each of the flows contains more packets. Furthermore, the highest average byte size is 177 bytes, which is achieved by for example the second flow. This table shows clear differences with the flows used to create the fingerprint. This explains the differences between the fingerprints and the model of figure 6.15.

Time	Protocol	Source	Destination	Packets transferred	Bytes transferred
03:37:31.988842	TCP	192.168.1.121	92.53.91.20	5	372
03:37:31.988980	TCP	192.168.1.121	92.53.91.20	4	709
03:37:32.043521	TCP	192.168.1.121	92.53.91.20	5	338
03:37:32.043609	TCP	192.168.1.121	92.53.91.20	4	709
03:37:53.099354	TCP	192.168.1.121	92.53.91.20	5	338
03:37:53.099476	TCP	192.168.1.121	92.53.91.20	4	709
03:37:53.151797	TCP	192.168.1.121	92.53.91.20	5	372
03:37:53.151852	TCP	192.168.1.121	92.53.91.20	4	709
03:38:14.219721	TCP	192.168.1.121	92.53.91.20	5	372
03:38:14.219809	TCP	192.168.1.121	92.53.91.20	4	709

Table 6.18: NetFlows of communication with Trickbot C&C server 92.53.91.20 captured in sample 327-1.

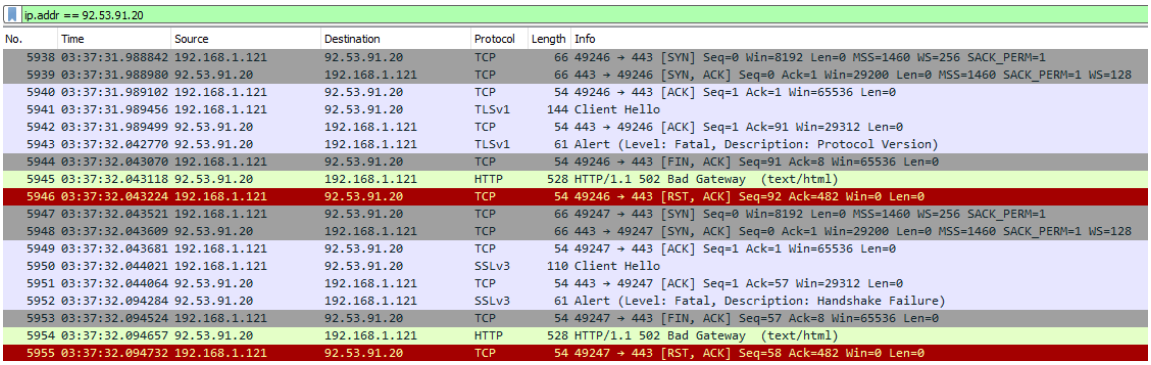
However, as mentioned earlier, the packet trace to server 92.53.91.20 did not contain any packets larger than 149 bytes. In fact, the packet traces of traffic to server 95.213.195.169 and 92.53.91.20 are nearly identical. This is illustrated in figures 6.16 and 6.17. Both traces show almost identical packet sizes from the infected host to the Command and Control servers. The packet sizes of figure 6.16 matches with the flows listed in table 6.17. The flows listed in table 6.18 do not match with the recorded traffic. Both the packet sizes and the number of packets are too high. From this observation, we conclude that the collected flows contain errors and do not describe the actual packet traces. Similar to this example, all other four undetected channels suffered from this issue.



ip.addr == 95.213.195.169

No.	Time	Source	Destination	Protocol	Length	Info
885	01:09:23.925052	192.168.1.123	95.213.195.169	TCP	66	49165 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
886	01:09:23.925171	95.213.195.169	192.168.1.123	TCP	66	443 → 49165 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
887	01:09:23.925254	192.168.1.123	95.213.195.169	TCP	54	49165 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
888	01:09:23.925542	192.168.1.123	95.213.195.169	TLSv1	144	Client Hello
889	01:09:23.925578	95.213.195.169	192.168.1.123	TCP	54	443 → 49165 [ACK] Seq=1 Ack=91 Win=29312 Len=0
971	01:11:33.495297	95.213.195.169	192.168.1.123	TLSv1	61	Alert (Level: Fatal, Description: Protocol Version)
972	01:11:33.495672	192.168.1.123	95.213.195.169	TCP	54	49165 → 443 [FIN, ACK] Seq=91 Ack=8 Win=65536 Len=0
973	01:11:33.495747	95.213.195.169	192.168.1.123	HTTP	536	HTTP/1.1 502 Bad Gateway (text/html)
974	01:11:33.495831	192.168.1.123	95.213.195.169	TCP	54	49165 → 443 [RST, ACK] Seq=92 Ack=490 Win=0 Len=0
975	01:11:33.496114	192.168.1.123	95.213.195.169	TCP	62	49166 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 SACK_PERM=1
976	01:11:33.496187	95.213.195.169	192.168.1.123	TCP	62	443 → 49166 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1
977	01:11:33.496258	192.168.1.123	95.213.195.169	TCP	54	49166 → 443 [ACK] Seq=1 Ack=1 Win=64240 Len=0
978	01:11:33.496559	192.168.1.123	95.213.195.169	SSLv3	110	Client Hello
979	01:11:33.496594	95.213.195.169	192.168.1.123	TCP	54	443 → 49166 [ACK] Seq=1 Ack=57 Win=29200 Len=0
10…	01:13:44.565341	95.213.195.169	192.168.1.123	SSLv3	61	Alert (Level: Fatal, Description: Handshake Failure)
10…	01:13:44.565737	192.168.1.123	95.213.195.169	TCP	54	49166 → 443 [FIN, ACK] Seq=57 Ack=8 Win=64233 Len=0
10…	01:13:44.565788	95.213.195.169	192.168.1.123	HTTP	536	HTTP/1.1 502 Bad Gateway (text/html)
10…	01:13:44.565875	192.168.1.123	95.213.195.169	TCP	54	49166 → 443 [RST, ACK] Seq=58 Ack=490 Win=0 Len=0

Figure 6.16: A packet trace of traffic to Trickbot C&C server 95.213.195.169 captured in sample 324-1.



ip.addr == 92.53.91.20

No.	Time	Source	Destination	Protocol	Length	Info
5938	03:37:31.988842	192.168.1.121	92.53.91.20	TCP	66	49246 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
5939	03:37:31.988980	92.53.91.20	192.168.1.121	TCP	66	443 → 49246 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
5940	03:37:31.989102	192.168.1.121	92.53.91.20	TCP	54	49246 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
5941	03:37:31.989456	192.168.1.121	92.53.91.20	TLSv1	144	Client Hello
5942	03:37:31.989499	92.53.91.20	192.168.1.121	TCP	54	443 → 49246 [ACK] Seq=1 Ack=91 Win=29312 Len=0
5943	03:37:32.042770	92.53.91.20	192.168.1.121	TLSv1	61	Alert (Level: Fatal, Description: Protocol Version)
5944	03:37:32.043070	192.168.1.121	92.53.91.20	TCP	54	49246 → 443 [FIN, ACK] Seq=91 Ack=8 Win=65536 Len=0
5945	03:37:32.043118	92.53.91.20	192.168.1.121	HTTP	528	HTTP/1.1 502 Bad Gateway (text/html)
5946	03:37:32.043224	192.168.1.121	92.53.91.20	TCP	54	49246 → 443 [RST, ACK] Seq=92 Ack=482 Win=0 Len=0
5947	03:37:32.043521	192.168.1.121	92.53.91.20	TCP	66	49247 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
5948	03:37:32.043609	92.53.91.20	192.168.1.121	TCP	66	443 → 49247 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
5949	03:37:32.043681	192.168.1.121	92.53.91.20	TCP	54	49247 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
5950	03:37:32.044021	192.168.1.121	92.53.91.20	SSLv3	110	Client Hello
5951	03:37:32.044064	92.53.91.20	192.168.1.121	TCP	54	443 → 49247 [ACK] Seq=1 Ack=57 Win=29312 Len=0
5952	03:37:32.094284	92.53.91.20	192.168.1.121	SSLv3	61	Alert (Level: Fatal, Description: Handshake Failure)
5953	03:37:32.094524	192.168.1.121	92.53.91.20	TCP	54	49247 → 443 [FIN, ACK] Seq=57 Ack=8 Win=65536 Len=0
5954	03:37:32.094657	92.53.91.20	192.168.1.121	HTTP	528	HTTP/1.1 502 Bad Gateway (text/html)
5955	03:37:32.094732	192.168.1.121	92.53.91.20	TCP	54	49247 → 443 [RST, ACK] Seq=58 Ack=482 Win=0 Len=0

Figure 6.17: A packet trace of traffic to Trickbot C&C server 95.53.91.20 captured in sample 327-1.

The causes of false negatives in the Trickbot samples can be summarized by:

- Samples 325-1 and 327-1 contained connections to unreachable C&C servers. For this behaviour no model was learned.
- The flows collected from some traffic channels do not match with the actual traffic. It is assumed that errors occurred during the flow collection process.

7 Discussion

The research conducted throughout this thesis has produced various interesting findings, contributing to the body of knowledge with regards to State Machines and network-based threat detection. The proposed Streaming Blue-Fringe algorithm enables the learning of Probabilistic Deterministic Finite Automata on real-time data streams. This solves a number of challenges in current State Machine learning methods, like the ability to achieve low latencies, require less resources and obtain access to meaningful intermediate results. The ability to learn State Machines on data streams offers a range of new research opportunities for the application of State Machines to use cases where real-time learning is required or beneficial. This thesis focussed on the application of State Machines to one of these use cases, namely the detection of network-based cyber threats. The results of both the real-time learning method and the Cyber Security use case are discussed in section 7.2. Since the amount of allocated time and the resources available during the study were constrained, this work faces a number of limitations. These factors have impacted the proposed solutions as well as the evaluation of the findings. In order to take into account the limitations while interpreting the results and to provide ways to further improve the current study first the limitations of this work are discussed in section 7.1.

7.1. Limitations

The work presented in this thesis has a number of limitations. Firstly, section 7.1.1 explains that the proposed State Machine learning method uses approximations, which could lead to learning an incorrect model. Section 7.1.2 highlights the fact that no proper unsampled real-world dataset was available during this study. This limits the evaluation of the proposed threat detection method. A final limitation that is considered, is the limited ability to learn models when a traffic channel with a small amount of traffic is encountered. This limitation is discussed in section 7.1.3.

7.1.1. Approximations while learning State Machines

The proposed Streaming Blue-Fringe uses Count-Min sketches for maintaining the occurrence frequencies of sequences of futures states in each state. On storing a sequence in the Count-Min sketch, hash collisions with other stored sequences could occur. This would result in overestimations of the frequencies. In the worst-case, this leads to states being merged that are not similar resulting in incorrect models. This frequency approximation impairs the learning accuracy, in case the sketch size is too small or when a large variety of sequences are stored. This is illustrated in section 4.5.3. On the other hand, using approximations is what improves runtime complexity of the algorithm as explained in section 6.1. There is a trade-off between opting for a higher accuracy or a less resource intensive solution. No approximations results in a higher accuracy, yet requires more resources when learning the models. Since real-time performance was an important driver of the algorithm design, it was decided to use approximations.

7.1.2. No access to unsampled real-world datasets

In the explored application of the Streaming Blue-Fringe, NetFlow traffic was used instead of network packets in order to obtain a less resource intensive threat detection method. To make collection and processing even more efficient, NetFlows are often sampled. However, when sampling is applied to the NetFlows collection process this yields to incorrect packet and byte counts. Furthermore, large portions of the NetFlows are discarded. Since the detection method proposed in this work aims to detect malicious behaviour based on packet statistics of traffic sequences, sampling would affect the traffic sequences. When sampling is applied, two identical network streams will most likely have different sequences with different packet count and size statistics. Hence, it is important to have access to unsampled datasets.

During the thesis only access was obtained to a sampled dataset from KPN and an unsampled dataset from the TU Delft. The TU Delft dataset; however, contained a number of errors. All dates were set to January 1 1970, as the NetFlow collecting routers did not set timestamps on creating a NetFlow. Furthermore, a large portion of flow durations were negative and some NetFlows contained timestamps that were slightly later than the other timestamps. As a result, the order in which the flows are being sent using *nfreplay* was not the correct ordering that is obtained when displaying all flows using *nfdump*. A significant amount of time was spent on getting the limited amount of real-world data to a usable state. As the number of faced problems mounted, it was decided to omit evaluation on real-world data and solely focus on the Stratosphere IPS datasets.

7.1.3. Challenges on traffic channels with small amounts of traffic

The Streaming Blue-Fringe requires each of the states to process more elements than the significance boundary, before the state is promoted or merged. As a consequence, on traffic channels with small amounts of traffic the State Machines remain very small. State Machines with a low number of states and without loops are not suitable for tracing paths that can be compared and matched with other State Machines. These State Machines cannot be used to generate sequences to be compared with State Machine fingerprints. Hence, traffic channels with little traffic can not be matched against fingerprints of known malicious behaviour. This is a limitation in the proposed learning and detection method. Not only the proposed Streaming Blue-Fringe experiences this issue, it is a more general problem for statistical approaches. There needs to a certain amount of data available, before a statistic gets enough significance to be able to draw a correct conclusion. A rule-based approach would not have had this issue. However, this has its own series of challenges, for example the generation usually requires domain experts and rule-based approaches are not robust against variations in behaviour. This makes the proposed approach, although it has its limitations, still worthwhile to be applied.

7.2. Results

Although the performed study is constrained by the previously mentioned limitations, it has put forward a number of interesting results. This section enumerates the key findings and discusses their significance and possible implications. First the results of the proposed method for learning State Machines is discussed in section 7.2.1. Next, section 7.2.2 examines the findings of employing the Streaming Blue-Fringe for network-based threat detection.

7.2.1. Streaming Blue-Fringe

Even though the original Blue-Fringe algorithm was already proposed in 1998 [50], the general approach as well as the Red-Blue strategy at its core still prove to be powerful in recent studies [53] [62] [90]. Section 1.2 and more specifically section 4.1.1 show the limitations of using the Blue-Fringe for a specific set of problems that require learning State Machines on real-time data streams. Chapter 4 overcomes these limitations by designing the Streaming Blue-Fringe algorithm. This algorithm borrows the Red-Blue merging strategy, but it does not start by learning the full Prefix Tree Acceptor. Instead, the State Machines are learned iteratively while performing the state comparisons based on fixed-sized futures stored in Count-Min sketches. The use of Count-Min sketches yields a space complexity in the order of the number of states, as explained in section 6.1, instead of the number of stream elements which is the worst case complexity of the original Blue-Fringe. Also, the sketch comparisons and iterative learning approach significantly improved the time complexity.

Since these improvements resulted from the use of approximations, the correctness of the solutions can not be guaranteed as explained in section 7.1.1. This is confirmed by the visual comparisons of the validation models illustrated in section 4.6.2. The deviations; however, mostly occur in the more complex models with larger numbers of interconnected states. Furthermore, one could argue that these visual differences are only of small importance. The noticed deviations are mostly the result of uncolored states receiving too little sequences to become part of the final State Machine. The corresponding traces rejected by the State Machine are very uncommon in the training data and hence not regular behaviour of the modelled process. Even if the behaviour missing in the learned model is an indicator of the modelled process, it can hardly be used for matching as the statistical significance of the observed behaviour is low. Matching on low occurring details is susceptible to minor deviations in behaviour which would increase chances of false positives.

From the consistent low Kullback–Leibler divergences across validation samples in table 4.6 and the high visual similarities between models in section 4.6.2, one can conclude that the proposed algorithm is capable of learning accurate State Machine models. The improvements in terms of memory and run-time performance as well as the iterative learning approach makes the Streaming Blue-Fringe a very good candidate for modelling systems in real-time. The proposed learning method can be applied by system engineers in real-world applications and by researchers in future studies.

7.2.2. Network Threat Detection

The proposed Streaming Blue-Fringe is a general State Machine learning method, which could be applied to numerous use cases. The use case driving the motivation to design the algorithm in this thesis was the idea that the use of State Machines could offer a powerful approach for detecting advanced threats in computer networks. Chapter 5 has focused on applying the algorithm into a system for learning models of malicious network behaviour. This required a mapping of NetFlow parameters to symbols used in the learning method. Analysis of an entire day of traffic from the EEMCS faculty of Delft University of Technology resulted in fifteen groups based on the number of packets transferred and the average packet size. Four malware samples were selected and fingerprints are learned in section 5.2.3. The resulting fingerprints are all different, indicating that the learning method and proposed symbol mapping shows potential for being able to distinguish behaviour of different malware types. Section 5.3.3 shows that models learned on the first day of traffic proved to be very similar to the models learned on subsequent days. The Kullback–Leibler divergences between the models learned on training and validation traffic show that a detection threshold can be defined for discriminating between similar and dissimilar network behaviour. It was decided that traffic channels with a Kullback–Leibler divergence below 0.5 can be considered as exhibiting similar behaviour. Due to time constraints, only four samples were analysed. The resulting detection threshold is therefore not regarded as very precise. Future studies could assess more malware types and improve upon this.

The detection method developed in chapter 5 is evaluated in chapter 6. Fingerprints were learned on four newly selected malware types, ensuring an unbiased evaluation of the detection capability. Visual comparison of the fingerprints of chapters 5 and 6 further affirms that network traffic of different malware types show significant diversity in behaviour which could be detected via model comparisons. The Streaming Blue-Fringe's detection approach is compared against a common baseline, that is blacklisting of known malicious servers. The results of section 6.2.3 show that nearly all infected hosts are detected. Also the majority of malicious traffic channels are being detected. On both host- and channel-level, the Streaming Blue-Fringe detection approach significantly outperforms the blacklisting baseline approach. Again, due to practical reasons, the number of tested samples is not very high. Hence, the obtained performance values could vary on evaluating additional samples. However, the obtained results clearly demonstrate that the use of State Machines is indeed a very promising solution for detecting malware indicators or other unwanted behaviour in network traffic. There is no reason to doubt that the proposed detection method will also achieve good performance on additional malware samples. Especially considering that only two NetFlow parameters are currently being used to map NetFlows to symbols. The current results already show a very powerful detection method, obtaining zero false positives. Nonetheless, It is reasonable to expect that with more fingerprints and more processed traffic, eventually false positives would start to appear. By taking into account more NetFlow parameters, e.g. the NetFlow inter-arrival time, the number of unique State Machines vastly increases. This would decrease the chances for false positives. Besides, the detection process could be altered to only consider a fingerprint as matched if multiple steps throughout the learning process are matching. These are just two examples of, even if false positives would appear, techniques can be applied to reduce them.

Analysis of false negatives

Although false positives were not present among the evaluated malware types, some false negatives did occur. In total 38 out of 90 Command & Control traffic channels remained undetected. Section 6.2.3 in detail examined the causes for false positives across each of the malware types.

One of the most common causes of false negatives is due to unreachable servers in either the learning or detection phase. If during the learning phase a server is unreachable, the resulting State Machine is not representative for behaviour that contain actual successful connections. Similarly, a model learned on a successful connection often does not match with channels to unreachable servers. One way to combat the problem of unreachable servers, is to realize that these connections do not perform any malicious operations like extracting sensitive information. A malware infection that fails to communicate to the outside world is very limited in the harm it can do. In case it does have some traffic channels with successful connections, detecting these channels would suffice to detect that a host is infected. In case the failing connections are present in the learning phase, this is more disadvantageous. Fingerprints of failing connections could result in false positives by possible matches with failing connections of benign services. Furthermore, a fingerprint on a failing connection could result in false negatives by not matching with traffic to active Command & Control servers. Hence, the main problem is to find a method that prevents learning fingerprints on traffic channels with failing connections. Luckily solving this is straightforward. One could create heuristics that detect failing connections on packet level easily by observing the TCP flags. Depending on the NetFlow collection method, these flags could also be accessed on a NetFlow level.

Another portion of the false negatives was caused by not using the smaller models as fingerprint. Some traffic channels only contain small amounts of traffic, impeding the Streaming Blue-Fringe from learning models complex enough to describe a large variety of unique sequences. In this work it was assumed that models with less than 6 unique sequences would contain too little information. Trivial models that accept a low number of unique sequences are expected to match with non-malicious traffic channels as their State Machines are still starting to form. This would result in false positives. Hence, a trade-off is made between obtaining additional detection performance and introducing false positives. According to our best knowledge there is no easy solution that will reduce the number of

false negatives without introducing false positives. An approach that eliminates the trade-off would be able to learn meaningful models on small amounts of traffic or use small models without introducing false positives. In order to improve learning on small amounts of data, the impact of a lower significance boundary can be further studied. This boundary dictates the amount of traffic sequences required before a state is merged or promoted, as explained in section 4.5.2. A lower boundary means the State Machine learns faster, but it will take decisions on data that is less statistically accurate.

A final portion of false negatives was caused by behavioural differences between malware samples. Robustness against evolving malware is an important yet challenging aspect in developing effective threat detection methods. In literature it is described that most types of malware are obfuscated and packed resulting in unique binaries with very little similarities [75]. An updated malware type could have a largely different binary as a means to prevent detection by binary signatures. In terms of communication protocols, which would affect network based detection methods, malware remains relatively constant. A study shows that a large number of malware samples use the same network infrastructure for at least a year [51]. Another study reveals that malware communication protocols show large similarities across different samples within malware families [10]. It could be the case that communication protocols in some of the undetected channels did not evolve, but instead showed variance in the same type of behaviour. The symbol mapping from section 5.2.1 was based on general traffic statistics, but not optimized to capture differences in malware behaviour per se. Possibly by expanding the boundaries used to map NetFlows to symbols, the learned models are less susceptible to variance in behaviour within different samples of a specific malware type.

8 Future Work

In this chapter a number of open questions, remarks and ideas for improvement are described. Future studies with regards to learning State Machines on data streams can use this as a handhold to further improve the proposed technique. This chapter also gives insights into alterations that could be explored with regards to using State Machines for threat detection. Section 8.1 describes changes to the algorithm or the Apache Flink implementation that could be explored. Section 8.2 describes possible improvements on learning fingerprints of malicious traffic, as well as ideas for improving the detection mechanism.

8.1. Algorithm Improvements

This work proposed an algorithm for learning State Machines on data streams. After a rigorous design, implementation and testing of the method still a number of improvements could be made. Due to time constraints, not all ideas could be tested.

8.1.1. Locality Sensitive Hashing

Once a state reaches its significance boundary, the candidate state is compared against all current red states. As a State Machine accumulates more states over time, this comparison will take an increasing amount of time. This could be improved by applying Locality Sensitive Hashing to the state sketches [77]. If sketches can be hashed to bins based on their distribution, it should be sufficient to only compute the more accurate comparison on states within the same bin.

8.1.2. Distributed learning

The current algorithm is proven to be a scalable approach, which is able to learn State Machines on data streams. With the application of threat detection in mind, which requires to learn a large number of State Machines, the current setup is designed to execute multiple instances of the algorithm in a distributed way. If the use case is to learn a single and very large State Machine, it would be advantageous to be able to distribute the learning process of a single State Machine. This is expected to be quite a challenge, as a distributed state should be maintained and controlled. An approach could be to map sequences to different threads based on their first symbols. This way the learning process of different branches in the State Machine will be distributed. However, it is challenging as at some point in time it could be necessary to merge states from different parts of the distributed execution. Although this will be a challenge, the result would be really powerful as a single very large and complex process could be modelled in real-time.

8.2. Threat detection improvements

The proposed algorithm to learn State Machines has been applied to network based threat detection on NetFlow traffic. Tests using data from the Stratosphere IPS shows that this is a promising technique. Unfortunately, it was not possible to obtain correct unsampled data of a real-world network. In future work the method could be tested in a real-world scenario. Apart from that, a number of ideas can be explored.

8.2.1. Reset traffic channels

In the current learning method, the system keeps learning the State Machine on each traffic channel indefinitely. As a consequence, memory resources that are claimed will never be freed. When learning in high throughput environments on multiple days, this could result in eventually running out of memory. Furthermore, the behaviour of hosts

can change and in live networks IP addresses can get reassigned to other machines. Therefore, in future work it is important to focus on how to be able to run the system for extended periods of time. One naive solution is to reset the State Machine for each traffic channel at the end of the day. Future studies could focus on this and come up with additional techniques. It would be very interesting to be able detect changes over time and update states in the Streaming Blue-Fringe, even if they are already colored red.

8.2.2. Changing perspective

Currently State Machines are learned on IP pairs, as it is expected that traffic channels are more or less independent. Future studies can look deeper into learning State Machines for each host or on bi-directional connections. Learning a single State Machine on bi-directional connections could be more robust since input and output behaviour is considered instead of only the traffic that is being sent.

8.2.3. Reservoir Sampling

In the proposed approach, State Machines are learned after which the distributions of the languages represented by State Machines are compared using the KL divergence. Since distributions are compared, this raised the idea that traffic could be compared in similar ways, but without the State Machine learning process. One such approach is to use Reservoir Sampling to sample for each connection the occurring traffic sequences. The sampled sequences can be compared with traffic sequences sampled from malicious traffic using a similar KL divergence approach. It would be interesting to know how such approach compares to actually learning State Machines. It is expected that State Machines have an edge, since it could create transitions that predict unseen traffic sequences.

8.2.4. Combine State Machines with Honeynets

The proposed learning method could be combined with Honeynets or other machines that execute malware in a controlled environment. In theory this makes it possible to learn fingerprints on new types of malware as soon as they spread through the wild. Since the learning process is automated and operates on data streams, in short notice a fingerprints is learned and ready to be applied for detection. An open question; however, is how to prevent the number of fingerprints of growing too large which makes detection slower and increases the chances of false positives.

9 Conclusion

Our increasingly interconnected society poses large risks in terms of cyber security. A growing portion of organizations and critical infrastructure is becoming heavily reliant on the security of hardware, software and networking protocols. In this growing threat landscape an increasing interest is observed for malicious actors to find and abuse vulnerable systems. In order to maintain visibility in our complex and data-rich networks, monitoring for any prevalent threats is an important task. Newly emerging hardware, capable of processing unsampled NetFlows, allows threat detection approaches on exact traffic sequences instead of random datapoints. In this thesis the use of State Machines is considered a very promising approach to threat detection. They could provide a method for capturing known malicious behaviour in powerful and easy to understand models. Learning state machines on traffic channels enables regular comparison of observed communication with pre-learned malicious behaviour. However, existing methods for learning state machines are not designed to learn models on infinite streams. Instead they are developed for learning a single state machine on a fixed-sized dataset. In order to effectively learn state machines in high throughput environments, a streaming algorithm should be designed. Access to an efficient streaming algorithm would enable exploring applications of state machines to numerous streaming use cases. One of which is the use of state machines to detect cyber threats.

This thesis studied the application of State Machines for network based threat detection. The objective of this thesis was to develop a method for learning State Machines on data streams with real-time performance. This objective has been captured in the following research question:

How can we develop an algorithm for learning State Machines on streaming data and apply this in a system capable of detecting the presence of malicious behaviour in real-time traffic channels?

In order to provide a structured answer to the research question, the main question was decomposed into four sub questions. By answering each of the sub questions, results are obtained that contribute to answering the main research question.

SQ I: How can we transform sequential algorithms into a streaming variant and implement them in Apache Flink?

Apache Flink has been selected as a robust framework for implementing and executing scalable streaming algorithms. Throughout chapter 3 streaming algorithms for various well-known data science problems have been studied and implemented. This resulted in a general approach that forms a starting point for developing and implementing streaming algorithms in Apache Flink.

SQ II: How can we learn State Machines on data streams?

The Blue-Fringe algorithm is a state of the art method for learning State Machines. Principles of the Blue-Fringe are used for developing a streaming algorithm capable of learning state machines on data streams. The resulting algorithm is called the Streaming Blue-Fringe. This algorithm, which is designed in chapter 4, learns a state machine by starting with a single state. During the learning process it gradually adds more states. This results in an iterative algorithm with access to intermediate solutions. In the Streaming Blue Fringe, the need to store all unique data elements in a Prefix Tree Acceptor before learning the State Machine is omitted. Instead, each state maintains a memory efficient Count-Min sketch. Each sketch approximates the frequencies of sequences that pass through the corresponding state. The benefit is a fixed-sized memory footprint that only depends on the size of the current State Machine. The learning method is evaluated by comparing models learned on generated data sequences, with the ground truth models used to generate the data. Visual comparison shows exact matches for small models and high similarities in complex models with large number of transitions.

SQ III: How can we detect the presence of a learned State Machine fingerprint in a traffic channel?
To be able to use State Machines for detecting network based threats, first models should be learned of known malicious traffic. Since State Machines are learned on sequences of symbols, a mapping should be made that maps network traffic to symbols. In section 5.2.1 fifteen symbols are defined by dividing NetFlows into equally sized groups based on two parameters. The selected NetFlow parameters are the average packet size and the number of packets transferred. Mapping a stream of NetFlows to symbols based on these parameters results in sequences. Using these sequences, State Machines can be learned for each IP pair. Since each connections is either benign or malicious, using IP pairs considers each connection individually. Probabilistic Deterministic Finite Automata can be describe a probability distributions of the accepted sequences. A traffic channel can be compared against a fingerprint by comparing probability distributions of the models. First, a set of traces and corresponding probability is generated from a fingerprint model. For the same set, the probabilities are computed for the current model of a traffic channel. Computing the Kullback–Leibler divergence between the resulting discrete probability distributions, results in a measure of how different the traffic channel is from the fingerprint. If the KL-divergence is above a determined threshold, one can conclude that the malicious traffic is detected in the traffic channel.

SQ IV: How scalable and effective is the proposed method of using State Machine fingerprints for network based threat detection?
Chapter 6 has studied the scalability of the proposed Streaming Blue-Fringe. Although the algorithm was not executed on a cluster, it has been successfully implemented and executed in Apache Flink, which offers a robust framework for distributed stream processing with horizontal scalability. Execution on commodity hardware shows the proposed method can easily process NetFlows of an entire day of traffic from the TU Delft's EEMCS faculty in only a few hours. The peak throughput was measured to be over 40.000 NetFlows per second, whereas the live network generates on average 267 NetFlows per second. Compared to the original Blue-Fringe algorithm, the worst-case runtime and memory performance of the Streaming Blue-Fringe is proven to be more efficient. The evaluation in section 6.2 determined the detection performance on 90 Command & Control channels spread over 12 malware samples of 4 different malware types. Averaged over the malware types, 71% percent of the malicious traffic channels were detected. On a host level, 95.8% of the infected hosts were successfully detected. Compared to a blacklisting baseline with only 26% channel detections and 45.8% host detections, the proposed method is far superior. Furthermore, no false positives were detected on the hundreds of benign traffic channels across all malware samples and in the 14 evaluated non-malicious datasets. Learning State Machines captured behaviour unique to each malware type. This modelled behaviour can be successfully detected on new network traces, showing that the proposed detection method is highly effective.

How can we develop an algorithm for learning State Machines on streaming data and apply this in a system capable of detecting the presence of malicious behaviour in real-time traffic channels?
The developed Streaming Blue-Fringe is an efficient algorithm for learning accurate State Machines on streaming data. This algorithm has been successfully applied for learning State Machines of network traffic to malware Command & Control servers. The resulting models are used as fingerprints in a scalable system developed to detect malicious behaviour on live networks. The system proves capable of learning State Machines on real-time traffic channels. Evaluation of the proposed detection method yields excellent performance without any false positives. The results in this thesis show that the use of state machines is a viable approach for network-based threat detection.

This thesis research has taken the first steps to a novel, robust threat detection method. Future studies could expand on this work by studying individual components like the symbol mapping, sequence length and model comparisons in more detail. Additional research could further improve the learning and detection methods. Eventually, through practical applications, these techniques could give cyber defense a powerful tool against its adversaries. Techniques that contribute to making our cyber world a safer place.

Bibliography

[1] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[2] Vito Albino, Umberto Berardi, and Rosa Maria Dangelico. Smart cities: Definitions, dimensions, performance, and initiatives. *Journal of Urban Technology*, 22(1):3–21, 2015.

[3] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.

[4] Hybrid Analysis. Hybrid analysis, 2018. URL `https://www.hybrid-analysis.com`.

[5] Borja Balle, Jorge Castro, and Ricard Gavalda. Learning pdfa with asynchronous transitions. In *International Colloquium on Grammatical Inference*, pages 271–275. Springer, 2010.

[6] Borja Balle, Jorge Castro, and Ricard Gavalda. A lower bound for learning distributions generated by probabilistic automata. In *International Conference on Algorithmic Learning Theory*, pages 179–193. Springer, 2010.

[7] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *INFOCOM*, pages 1–9, 2016.

[8] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13 (7):422–426, 1970.

[9] Egon Börger. High level system design and analysis using abstract state machines. In *International Workshop on Current Trends in Applied Formal Methods*, pages 1–43. Springer, 1998.

[10] Amine Boukhtouta, Nour-Eddine Lakhdari, and Mourad Debbabi. Inferring malware family through application protocol sequences signature. In *New Technologies, Mobility and Security (NTMS), 2014 6th International Conference on*, pages 1–5. IEEE, 2014.

[11] Daniela Brauckhoff, Bernhard Tellenbach, Arno Wagner, Martin May, and Anukool Lakhina. Impact of packet sampling on anomaly detection metrics. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 159–164. ACM, 2006.

[12] Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*. Citeseer, 2002.

[13] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[14] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[15] Yun Chi, Haixun Wang, Philip S Yu, and Richard R Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on*, pages 59–66. IEEE, 2004.

[16] Baek-Young Choi and Supratik Bhattacharyya. On the accuracy and overhead of cisco sampled netflow. In *Proceedings of ACM SIGMETRICS Workshop on Large Scale Network Inference (LSNI)*, pages 1–6, 2005.

[17] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3):178–187, 1978.

[18] Benoit Claise. Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information. Technical report, 2008.

[19] Graham Cormode. Count-min sketch. In *Encyclopedia of Database Systems*, pages 511–516. Springer, 2009.

[20] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[21] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[22] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel Van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[23] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 227–238. ACM, 2017.

[24] Pierre Dupont and Lin Chase. Using symbol clustering to improve probabilistic automaton inference. In *International Colloquium on Grammatical Inference*, pages 232–243. Springer, 1998.

[25] Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde. The qsm algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22(1-2):77–115, 2008.

[26] Adriaans P. et al. Pautomac probabilistic automaton learning competition, 2012. URL `http://ai.cs.umbc.edu/icgi2012/challenge/Pautomac/index.php`.

[27] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[28] Gyula Fóra. Real-time data processing with apache flink, 2018. URL `https://www.slideshare.net/GyulaFra/flink-streaming-budapestdata`.

[29] S. Garcia. Malware capture facility project, 2013. URL `https://mcfp.weebly.com/`.

[30] Mayank Goswami, Dzejla Medjedovic, Emina Mekic, and Prashant Pandey. Buffered count-min sketch on ssd: Theory and experiments. *arXiv preprint arXiv:1804.10673*, 2018.

[31] The Apache Software Group. Apache hadoop framework, 2008. URL `https://hadoop.apache.org/`.

[32] The Apache Software Group. Apache spark framework, 2012. URL `https://spark.apache.org/`.

[33] The Apache Software Group. Apache storm framework, 2014. URL `https://storm.apache.org/`.

[34] The Apache Software Group. Apache flink framework, 2015. URL `https://flink.apache.org/`.

[35] The Apache Software Group. Apache flink api concepts, 2017. URL `https://ci.apache.org/projects/flink/flink-docs-master/dev/api_concepts.html`.

[36] The Apache Software Group. Apache flink introduction, 2017. URL `https://flink.apache.org/introduction.html`.

[37] The Apache Software Group. Apache flink stream operators, 2018. URL `https://ci.apache.org/projects/flink/flink-docs-master/dev/stream/operators/`.

[38] Shobhit Gupta, John A Stamatoyannopoulos, Timothy L Bailey, and William Stafford Noble. Quantifying similarity between motifs. *Genome biology*, 8(2):R24, 2007.

[39] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM sigmod record*, volume 29, pages 1–12. ACM, 2000.

[40] Mike Hazas, Janine Morley, Oliver Bates, and Adrian Friday. Are there limits to growth in data traffic?: on time use, data generation and speed. In *Proceedings of the second workshop on computing within limits*, page 14. ACM, 2016.

[41] Marijn JH Heule and Sicco Verwer. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, 18(4):825–856, 2013.

[42] Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto, and Aiko Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, 16(4):2037–2064, 2014.

[43] Pengyu Hong, Matthew Turk, and Thomas S Huang. Constructing finite state machines for fast gesture recognition. In *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, volume 3, pages 691–694. IEEE, 2000.

[44] L Kari, S KONSTANTlNIDlS, S Perron, G WOZNIAKI, and J KU. Computing the hamming distance of a regular l language in quadratic tirne˜. 2004.

[45] Oliver Kasten and Kay Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 7. IEEE Press, 2005.

[46] Michael Kearns and Leslie Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM (JACM)*, 41(1):67–95, 1994.

[47] Maciej Korczyński and Andrzej Duda. Markov chain fingerprinting to classify encrypted traffic. In *Infocom, 2014 Proceedings IEEE*, pages 781–789. IEEE, 2014.

[48] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[49] Bernard Lambeau, Christophe Damas, and Pierre Dupont. State-merging dfa induction algorithms with mandatory merge constraints. In *International Colloquium on Grammatical Inference*, pages 139–153. Springer, 2008.

[50] Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference*, pages 1–12. Springer, 1998.

[51] Chaz Lever, Platon Kotzias, Davide Balzarotti, Juan Caballero, and Manos Antonakakis. A lustrum of malware network communication: Evolution and insights. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 788–804. IEEE, 2017.

[52] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y Chang. Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems*, pages 107–114. ACM, 2008.

[53] Qin Lin, Christian Hammerschmidt, Gaetano Pellegrino, and Sicco Verwer. Short-term time series forecasting with regression automata. 2016.

[54] Altti Ilari Maarala, Mika Rautiainen, Miikka Salmi, Susanna Pirttikangas, and Jukka Riekki. Low latency analytics for streaming traffic data with apache spark. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2855–2858. IEEE, 2015.

[55] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, and Hui Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 165–176. ACM, 2006.

[56] Make42. Mapping and reduce stages of a spark job, 2018. URL `https://stackoverflow.com/questions/37528047/how-are-stages-split-into-tasks-in-spark/37529310`.

[57] Jayadev Misra and David Gries. Finding repeated elements. Technical report, Cornell University, 1982.

[58] J Strother Moore. A fast majority vote algorithm. *Automated Reasoning: Essays in Honor of Woody Bledsoe*, 1981.

[59] Mike Olson. Hadoop: Scalable, flexible data storage and analysis. *IQT Quart*, 1(3):14–18, 2010.

[60] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[61] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *icccn*, page 0215. IEEE, 2001.

[62] Gaetano Pellegrino, Christian Hammerschmidt, Qin Lin, and Sicco Verwer. Learning deterministic finite automata from infinite alphabets. In *International Conference on Grammatical Inference*, pages 120–131, 2017.

[63] Carol Peters. *Advances in Multilingual and Multimodal Information Retrieval: 8th Workshop of the Cross-Language Evaluation Forum, CLEF 2007, Budapest, Hungary, September 19-21, 2007, Revised Selected Papers*, volume 5152. Springer Science & Business Media, 2008.

[64] L PITT. The minimum dfa consistency problem cannot be aproximated within any polynomial. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*. ACM Press, 1989.

[65] Jacob Poushter et al. Smartphone ownership and internet usage continues to climb in emerging economies. *Pew Research Center*, 22:1–44, 2016.

[66] Viktor Puš, Lukáš Kekely, Jan Kucera, and Denis Matoušek. Live demonstration of application layer traffic monitoring at 100 gbps.

[67] CIO PwC and CSO. The global state of information security survey 2018, 2018. URL `https://www.pwc.com/sg/en/publications/assets/gsiss-2018.pdf`.

[68] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*, pages 11–20. IEEE, 2015.

[69] Jeffrey C Schlimmer and Douglas Fisher. A case study of incremental concept induction. In *AAAI*, volume 86, pages 496–501, 1986.

[70] Marc Sebban, Jean-Christophe Janodet, and Frédéric Tantini. Blue: a blue-fringe procedure for learning dfa with noisy data. In *Proceedings of Genetic and Evolutionary Computation Conference*, 2004.

[71] Hemant Sengar, Duminda Wijesekera, Haining Wang, and Sushil Jajodia. Voip intrusion detection through interacting protocol state machines. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 393–402. IEEE, 2006.

[72] Verizon Digital Media Services. Enterprise network flow collector (ipfix, sflow, netflow), 2017. URL `https://github.com/VerizonDigital/vflow`.

[73] Nima Shahbazi, Rohollah Soltani, Jarek Gryz, and Aijun An. Building fp-tree on the fly: Single-pass frequent itemset mining. In *Machine Learning and Data Mining in Pattern Recognition*, pages 387–400. Springer, 2016.

[74] Saeed Shahrivari. Beyond batch processing: towards real-time and streaming big data. *Computers*, 3(4):117–129, 2014.

[75] Anshuman Singh, Andrew Walenstein, and Arun Lakhotia. Tracking concept drift in malware families. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, pages 81–92. ACM, 2012.

[76] Michael Sipser. *Introduction to the Theory of Computation, Second Edition*. Thomson Course Technology, 2006.

[77] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal processing magazine*, 25(2):128–131, 2008.

[78] Apache Spark. Apache spark streaming programming guide, 2018. URL `https://spark.apache.org/docs/1.2.2/streaming-programming-guide.html`.

[79] Andreas Stolcke and Stephen Omohundro. Hidden markov model induction by bayesian model merging. In *Advances in neural information processing systems*, pages 11–18, 1993.

[80] Apache Storm. Apache spark architecture, 2018. URL `https://dzone.com/articles/apache-storm-architecture`.

[81] Stratosphere. Stratosphere ips datasets, 2018. URL `https://www.stratosphereips.org/datasets-overview/`.

[82] Zhong Su, Qiang Yang, Ye Lu, and Hongjiang Zhang. Whatnext: A prediction system for web requests using n-gram sequence models. In *wise*, page 0214. IEEE, 2000.

[83] Symantec. Internet security threat report volume 23, 2018. URL `https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf`.

[84] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Salsa: Analyzing logs as state machines. *WASL*, 8:6–6, 2008.

[85] KMM Thein. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483, 2014.

[86] Franck Thollard, Pierre Dupont, Colin de la Higuera, et al. Probabilistic dfa inference using kullback-leibler divergence and minimality. In *ICML*, pages 975–982, 2000.

[87] Ransomware Tracker. Ransomware tracker, 2018. URL `https://ransomwaretracker.abuse.ch`.

[88] International Telecommunication Union. Measuring the information society report 2017, 2017. URL `https://www.itu.int/en/ITU-D/Statistics/Documents/publications/misr2017/MISR2017_Volume1.pdf`.

[89] SE Verwer, MM De Weerdt, and Cees Witteveen. An algorithm for learning real-time automata. In *Benelearn 2007: Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands, Amsterdam, The Netherlands, 14-15 May 2007*, 2007.

[90] Sicco Verwer and Christian A Hammerschmidt. flexfringe: a passive automaton learning package. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 638–642. IEEE, 2017.

[91] Sicco Verwer, Rémi Eyraud, and Colin Higuera. Results of the pautomac probabilistic automaton learning competition. 2012.

[92] Sicco Verwer, Rémi Eyraud, and Colin De La Higuera. Pautomac: a probabilistic automata and hidden markov models learning competition. *Machine learning*, 96(1-2):129–154, 2014.

[93] Enrique Vidal, Franck Thollard, Colin De La Higuera, Francisco Casacuberta, and Rafael C Carrasco. Probabilistic finite-state machines-part i. *IEEE transactions on pattern analysis and machine intelligence*, 27(7): 1013–1025, 2005.

[94] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11 (1):37–57, 1985.

[95] Wieczorek W. *Grammatical Inference*. Springer, 2017.

[96] Neil Walkinshaw and Kirill Bogdanov. Inferring finite-state models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 248–257. IEEE Computer Society, 2008.

[97] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 209–218. IEEE, 2007.

[98] Neil Walkinshaw, Bernard Lambeau, Christophe Damas, Kirill Bogdanov, and Pierre Dupont. Stamina: a competition to encourage the development and assessment of software model inference techniques. *Empirical software engineering*, 18(4):791–824, 2013.

[99] Wikipedia. Netflow architecture, 2018. URL `https://en.wikipedia.org/wiki/NetFlow#/media/File:NetFlow_Architecture_2012.png`.

[100] Kenji Yamanishi and Jun-ichi Takeuchi. A unifying framework for detecting outliers and change points from non-stationary time series data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 676–681. ACM, 2002.

[101] Ke Yi and Qin Zhang. Optimal tracking of distributed heavy hitters and quantiles. *Algorithmica*, 65(1):206–223, 2013.

[102] Yihuan Zhang, Qin Lin, Jun Wang, and Sicco Verwer. Car-following behavior model learning using timed automata. *IFAC-PapersOnLine*, 50(1):2353–2358, 2017.