Learning State Machines faster using Locality-Sensitive Hashing and an application in network-based threat detection

Learning State Machines faster using Locality-Sensitive Hashing and an application in network-based threat detection

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Rafail SKOULOS

Master of Engineering in Electrical and Computer Engineering,
National Technical University of Athens, Greece
born in Mytilene, Greece

Student number:	4847482	
Project duration:	November 1, 2019 – August 25, 2020	
Thesis committee:	Dr. ir. S.E. Sicco Verwer,	TU Delft, Supervisor
	Prof. dr. ir. R.L. Inald Lagendijk,	TU Delft
	Dr. ir. Mauricio Finavaro Aniche,	TU Delft

The work in the thesis was conducted together with APTA Technologies.

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

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

Contents

Abstract

The internet traffic is constantly rising nowadays due to the significant increase of the devices connected to the Internet. As a consequence, many cyber risks have arisen. Cybercriminals are trying to exploit the vulnerabilities of these devices to cause damage and gain profit. Monitoring the network traffic and detecting such threats has become essential in order to keep safe systems that are connected to the Internet. The powerful properties of state machines and the sequential nature of the network traffic data, makes them an interesting and promising solution for the implementation of an intrusion detection system.

The goal of this thesis is to implement a new state-merging heuristic which will speedup the state machine building procedure without a significant loss on the quality of the model, and use it to detect malicious host on network traffic data. The new state-merging heuristic is utilizing the Locality-sensitive Hashing concept to store the future traces of each state and simplify the consistency check for the merge of two states. The network traffic data used are in the NetFlow format, and they are encoded and converted into traces in order to build the state machine model and measure its performance. The state machine built is modeling a malicious behavior and used to classify other hosts.

We show that the models built can effectively detect the malicious hosts, with its performance being comparable to the one of a state-of-the-art model. At the same time, the time needed to build the model is much less when compared to the time needed by other state-merging heuristics.

Acknowledgments

Many people contributed to this thesis in their own way.

First of all, I would like to thank my supervisor Sicco Verwer, for his valuable guidance and and his willingness to help throughout the process. With his great ideas, he helped me to overcome all the problems I faced.

Of course I want to thank everyone from APTA Technologies. It was a great experience for me to be member of the team. I would like to pay my special regards to Christian Hammerschmidt who have willingly shared his precious time during the process to discuss about my project and to impart his knowledge.

Last but not least, I am grateful to my family and friends who have been standing by my side all these years. They kept me going on and this work would not have been possible without their support.

Rafail
Delft, August 2020

1

Introduction

Internet is a technology invented in the 1960s, which allows the communication between networks and devices worldwide. Since its invention, the number of people using it is constantly rising. People use it for a plethora of their daily activities, including communication with other people worldwide, social media usage, entertainment and education among others. It has been also proved vastly beneficial for many areas of the economy, including finance, retail, industry, healthcare, education, and science. Especially, the rise of the Internet of things (IoT) the last years has brought new possibilities and increased vastly the volume of the Internet traffic. As of April 2020, 4.54 billion among the 7.77 billion people on the world, are active Internet users[1]. In 2019, 167 petabytes of data were exchanged per month the through Internet, while the predicted traffic volume per month for 2022 is 333 petabytes [2].

However, except from its positive aspects, the increasing number of machines connected to the Internet has caused the rise of the cyber-attacks. As cyber-attacks are considered the actions performed by cyber-criminals whose goal is to harm the user. There are many types of cyber-threats. Some of the most common ones are:

- **Malware**: A malicious software which performs a malicious task on the device, like taking control of the system or stealing data.
- **Phishing**: Uses an email to trick the user to either reveal confidential information (e.g. passwords) or download a malware by clicking on a hyperlink included on the email.
- **Trojans**: Gets into the system looking like a normal software, and releases some malicious code on the host.
- **Denial of Service attack or Distributed Denial of Service Attack (DDoS)**: The attacker takes control of a big number of devices to invoke functions of the target system and make them unavailable to its intended users.

[1]https://www.statista.com/statistics/617136/digital-population-worldwide/
[2]https://www.statista.com/statistics/267202/global-data-volume-of-consumer-ip-traffic/

1

One of the biggest cyber-threats nowadays are botnets. A botnet is a number of Internet-connected devices, each of which is running one or more bots [3]. Bot is an infected device that executes the commands of a controller, named botmaster. Botnets can be used to perform large-scale attacks such as the aforementioned ones.

Considering the damage that can be caused to individual users, companies and governments by getting infected, the need of a method to detect and prevent cyber-attacks became crucial. An widely utilized method to prevent cyber-attacks on a network is the deployment of an Intrusion Detection System on it. Intrusion is defined as a set of actions aimed to compromise the security of computer and network components in terms of confidentiality, integrity and availability[1]. Intrusion Detection Systems monitor and analyze the network activities in order to detect the malicious host trying to attack the network. Many times these systems has a consulting role, meaning that they rise an alarm when they suspect that a host is malicious but the final decision for excluding this host from the network is taken by the administrator of the network.

Several techniques has been used to implement an IDS. The rapid development is Artificial Intelligence in the recent years, has lead to the development of many machine learning methods which are used to detect anomalies in the network traffic. Some of the most popular categories are the classification-based methods [2][3][4][5][6][7]. Also several statistical based methods, like PCA[8][9][10][11] and the clustering-based[12][13] can be used for the implementation of an intrusion detection system. Most of these methods, despite their high performance, are not interpretable, meaning that the reasoning behind the classification of a host as malicious cannot be presented or it is not understandable by the user. This is a major disadvantage as it makes it difficult to the network administrator to decide if the host was actually malicious or a false alarm has been risen.

An interesting method that can be used in order to detect malicious hosts is the State machines [14][15] [16]. State machines are usually used to build the communication profile of the malicious behavior and the to detect the anomalies in the network by comparing the profiles of the examined host with the communication profile built.

1.1 Motivation

State machines is an interesting technique used to create the communication profile (model) of a malicious host, mainly because the sequential nature of the network traffic data and the interpretability of the model built. One of the most famous algorithms used to learn state machines is the Blue-Fringe algorithm[17]. However one of its biggest disadvantages is that it require a significant amount of time to build the model, because a requirement for two states to be merged is that also their descendant states can be merged. That disadvantage becomes even more important when they are used to build an communication profile and detect malicious hosts in a streaming fashion.

In order to overcame this, we are using the Locality-sensitive hashing [18] (LSH) concept. LSH is a well-known hashing based algorithm which hashes similar items in the same bucket with high probability. The number of buckets is much smaller than the number of possible input items, so by using LSH we can cluster similar items together in a fast

[3]https://en.wikipedia.org/wiki/Botnet

and efficient way. There are many different kind LSH families, each one clustering the items according to a different metric, e.g.,their Euclidean distance.

Our proposed state-merging heuristic, hashes all the future traces for each state into buckets and then uses the discrete probability distribution extracted by these buckets when trying to decide when two states should me merged or not. An example of traces stored into LSH buckets and the distribution extracted from them, is presented in figure 1.1. The intuition behind this idea is that similar traces will be hashed on the same bucket when using the proper LSH family, so states with similar future traces will have similar distributions. Thus instead of comparing all the children states of these states when we need to decide whether to merge them or not, we can reduce significantly the time required by comparing only the aforementioned distributions.

Given the procedure we follow to encode the flows, described in section 6.1.3, we expect that the bigger the numerical difference between two symbols is, the more different the two flows represented by these symbols is. So the numerical difference between two symbols need to be taken into account when comparing two future traces. In addition, we also need to take into account the order to the symbols in the traces. For example, the traces <1,2,3> and <3,2,1> are different, although they contain the same symbols. Thus we should treat the traces as vectors and not as sets. So measuring the Jaccard similarity $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$ will not be accurate for our case. So we decided to proceed with LSH families that stores in the same bucket traces whose distance as vectors is small. The simplest distance metrics for two vectors are Manhattan, Euclidean and angular distance. Thus, we decided to p-stable distribution (with p=1 and p=2) and random hyperplane LSH families to capture the aforementioned distances between the traces.

1

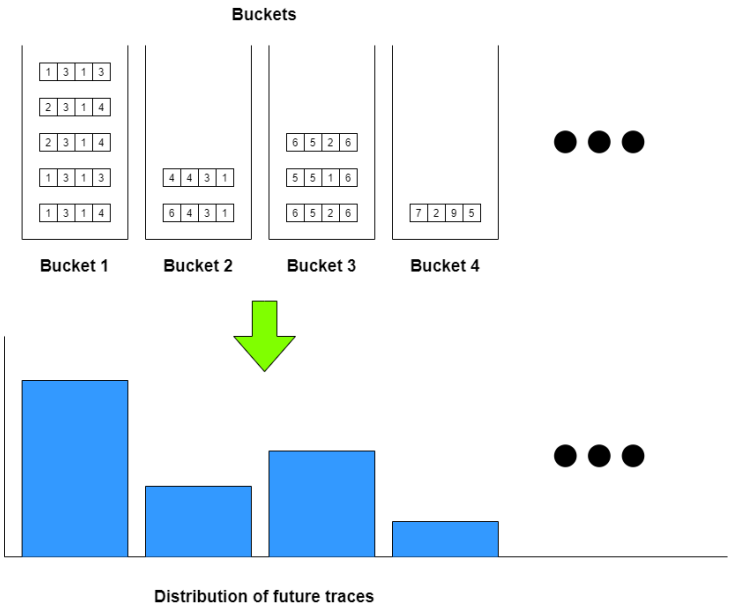


Figure 1.1: Example of extraction of the distribution of the future traces for a state from the LSH buckets

1.2 Research Questions

The objective of this work is to implement a new state-merging heuristic for state machines utilizing the LSH concept which will significantly speedup the learning procedure, and use it to learn state machine models from network traffic data that will be then used to detect malicious hosts. We use the concept of LSH to hash into buckets the future traces of each state, and compare their distribution as a consistency check for the merge of two nodes. The main question we want to answer is:

- How can I use the concept of LSH to create a fast state-merging heuristic that builds accurate state machines?

This question can be broken up in three parts. The first part is about the implementation of the proposed state-merging heuristic in a way that it will make the state-merging process faster without a significant loss on the ”quality” of the models built. The second part is about the LSH family that is the more efficient for this task. There are many LSH families, each of which is capturing different kind of similarities between the input items. For the reasons discussed section 1.1, the three LSH families we experimented with are p-stable distribution (with p=1 and p=2) and random hyperplane LSH families. Finally, the last part is about the performance of our method in building accurate models and how fast this procedure is.

To sum up, the research questions that we attempt to answer are the following:

1. How can I use the concept of LSH to create a fast state-merging heuristic that is accurate?

   (a) What should be stored to the LSH buckets?
   (b) What should be compared during the consistency check for two merges and how?
   (c) How will this heuristic make the state-merging process faster than using other heuristics ?

2. What kind of LSH family will be more efficient for the implementation of the heuristic?
   (a) LSH with 1-stable distribution which assigns to the same bucket traces with small Manhattan (l_1) distance?
   (b) LSH with 2-stable distribution which assigns to the same bucket traces with small Euclidean (l_2) distance?
   (c) LSH with random projections which assigns to the same bucket traces with small angular distance?

3. Which will be the implications of using this heuristic in terms of speed and performance?
   (a) What is the performance of a state machine build with the proposed heuristic on task of detecting malicious hosts on NetFlow data?
   (b) How it compares with the performance of a state-of-the-art model?
   (c) What is the performance of a state machine build with the proposed heuristic on learning distributions over strings and what how fast the building of the state machine is?

1.3 Contributions

In this thesis, we propose a new state-merging heuristic using the LSH hash-based algorithm to speedup the state-merging algorithm, and evaluate its runtime and its efficiency on detecting malicious hosts. We use the *flexfringe* tool [19] and integrate our heuristic to the Red-Blue fringe state-merging algorithm[17]. We hash into the LSH buckets the future traces for each state and compare the distribution extracted from the buckets when we need to decide if a merge is consistent. By hashing the future traces, we make an approximation about the future of each state so we avoid comparing their descendants states when trying to merge them. This significantly speedups the state-merging procedure. We also tried three different LSH families which capture different types of similarities between the traces. We evaluate the performance of state machines build with our heuristic on recognizing malicious host on CTU-13[20] dataset, by using three different evaluation methods, and we compare the best one with the performance of a state-of-the-art model. We also measured its runtime on the datasets of the PAUTOMAC competition[21].

Our experiments showed that by using the 1-stable distribution LSH family, we achieved the best performance on classifying malicious hosts on both single and multi scenario simulation, when compared to other methods like a trigram model or a state machine using a modified version of the alergia heuristic[22]. Regarding the runtime of out heuristic, when

1

compared to the alergia heuristic it achieved an speedup of 5.4, without an significant loss on the quality of the models produced. Hence, our main contributions are:

1. We introduced a new state-merging heuristic utilizing the LSH concept to measure the similarity of two states when deciding if they should be merged or not. To the best of our knowledge, there is no existing state-merging heuristic using the LSH concept.
2. We provided an empirical evaluation of this state-merging heuristic on building machines that detect malicious hosts. We showed that its performance is comparable with a state-of-the-art model.
3. We provided an empirical evaluation of its runtime, which according to our experiments is 5.4 times smaller than the runtime of a state machine using the alergia state-merging heuristic.
4. We provided a framework for fast detection of malicious hosts on NetFlow data.

1.4 Outline

The rest of this thesis is structured as follows: In chapter 2 the background information needed to understand the topic and the basic concepts of this thesis are provided, along with the description of the *flexfringe* tool used to build the models. In chapter 3, an overview of some previous studies on anomaly detection in network data and Locality-Sensitive Hashing is provided. Section 4.1 describes in detail the proposed state-merging heuristic and explains its implementation. Then,in chapter 5 the dataset used for our research is described and some data analysis is conducted. In chapter 6, the handling of the data, the evaluation the state machine model and the simulations we used for this evaluation are presented. Following, chapter 7 presents the experimental procedure followed along with its results and a discussion of them. Finally, in chapter 8 the conclusions that can be drawn from our work are presented, along with a reflection on the research questions, the limitations of our work and some ideas about future work.

2

Background

This chapter contains the background information required to understand the concepts, the term and the methods used in this thesis. Initially the concept of sequential data is described, as it the type of data used to on this thesis, followed by the definition and the description of the State Machines, which they are the main models used. Afterwards, the flexfringe software tool is described along with the basic concepts needed to understand its functionality. Following, the concept of Locality Sensitive Hashing is introduced and some additional informational are provided about the LSH families used on this thesis. Finally, we present the evaluations metrics used to evaluate the performance of the examined models on the host classification task.

2.1 Sequential Data

Sequential data is data that are ordered according to some of their characteristic. Time series are a well known type of sequential data, which consists of data points equally spaced in time. Other cases of sequential data are biological data from the DNA sequence, and data from text documents, where the order of the words is taken into account. According to V. Chandola et al. in [23], "a discrete/symbolic sequence is defined as a finite sequence of events, such that each event can be represented as a symbol belonging to a finite alphabet". Sequences contain subsequent values of a specific variable, hence are ideal for capturing the behavior of a variable and the structural dependencies between the sequential values.

The sequential data used in this project are Netflow Data, which consist of a sequence of network flows. These flows are characterized by several properties which are analyzed in a section 5.1.2.

2.1.1 Alphabet

The alphabet of a sequence is the set of all the possible letters of numbers that a symbol of a sequence can take. For example in the sequence <A,B,C,A>, the alphabet consists of all the different letters, that is [A,B,C].

2.1.2 Length

The length of a sequence is defined as the number of the symbols in a sequence. For the above defined sequence <A,B,C,A>, the length is 4. Except from the finite sequences, as the one mentioned before, there can be sequences with infinite length, such as the sequence of all the positive numbers $< 1, 2, 3, \ldots >$. However, in this project we will use only finite sequences as the number of flows in the NetFlow data is finite.

2.2 Trigram Model

Trigrams (3-grams) are continuous sequences of three events from a given sequence. Trigram can be used as a statistical model to assign probabilities to events of a sequence. Trigram model approximates the probability of a given events given all the previous events by calculating the conditional probability given the two previous events. The assumption that the probability of the occurrences of an events depends only on a limited number of previous events is known as Markov assumption. It can be formulated as:

$$P(E_n|E_1^{n-1}) \approx P(E_n|E_{n-1}, E_{n-2}) \tag{2.1}$$

Let's assume that we have three consecutive events E_1, E_2 and E_2. The conditional probability $P(E_3|E_2, E_1)$ can be computed by calculating the joint probability $P(E_1 \cup E_2 \cup E_3)$ and the marginal probability of the union of the two past events $P(E_1 \cup E_2)$. According to Kolmogorov's definition [24], the conditional probability $P(E_3|E_2, E_1)$ can be computed by the following formula:

$$P(E_3|E_2, E_1) = \frac{P(E_1 \cup E_2 \cup E_3)}{P(E_1 \cup E_2)}$$

When the trigram model is built, every distinct pair of consecutive events is a different state. For every of these states, there is a transition to another state according to the following symbol of the sequence, whose probability is calculated by using the formulas described above. An example of a trigram model is presented in figure 2.1.

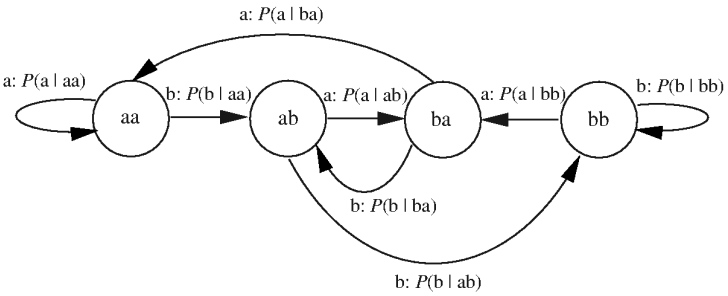


Figure 2.1: Trigram model [1]

[1]Source: http://www.phon.ox.ac.uk/jcoleman/old_SLP/Lecture_6/trigram-modelling.html

2.3 Finite State Machine

A Finite State Machine (FSM), also called Finite State Automata (FSA) or Deterministic Finite Automata (DFA), is a mathematical behavioral model representing a computer problem. It's components are states, transitions between the states and actions triggered by some transition. States represent the current situation after a sequence of actions have happened. FSM can be only in one state at a time. Transitions represent a change to the current state, while there is a condition for a transition to happen. Each transition can trigger an action. FSM is the type of model used in this project to predict whether host is malicious or benign.

The formal definition of State Machine as it is stated by Michael Sipser in the book "Introduction to the Theory of Computation" [25] is the following:

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

- Q is a finite set called the states,
- Σ is a finite state called the alphabet,
- $\delta : Qx\Sigma$ is the transition function,
- $q_0 \in Q$ is the start state and,
- $F \subseteq Q$is the set of accepted states.

An example of a state machine is the one on Figure 2.2. From this example we can derive that:

- The states are $Q = q_1, q_2, q_2$
- The alphabet is $\Sigma = 0, 1$
- The transition function δ is:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

- The start state is q_1
- The set of accepted states is $F = \{q_2\}$

The State Machine reads one character (symbol) of the input string at a time and moves to the next state according to the transition function, which in the provided figure means that follows the arrow of the current state with the corresponding symbol. For example, if we are on the initial state q_1 and and the incoming symbol is 1, then the next state will be q_2.

2

For a given input string (sequence) $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ of length n, the final state q_n is reached by following a sequence of transitions $(q_0, \delta(q_0, \sigma_1), (q_1, \delta(q_1, \sigma_2), q_n)$. If q_n is an accepting state, then we say that the given string is accepted, otherwise it is rejected. The set of all the accepting string by a State Machine M, is called the language L of the machine M.

State machines can be represented by a graph, as the one in Figure 2.2. Each node represents a state and each edge represents a transition. Each transition has a label which represents the symbol that caused this transition. For example, the transition from state q_1 to state q_2 has label 1. An arrow without a source state indicates the initial state, while the final state is depicted with a double circle. In our example, the initial state is q_1 and the final state is q_2.

State machines are used build a model representing the behavior of a system. The state machine of Figure 2.2 is modeling the following behavior:

$$A = \{w | w \; contains \; at \; least \; one \; 1 \\ and \; a \; even \; number \; of \; 0s \\ follow \; the \; last \; 1\}$$

In order to obtain the state machine which is modelling a specific system, a learning procedure should be followed. There are two main approaches for this learning procedure, the passive learning and the active learning.

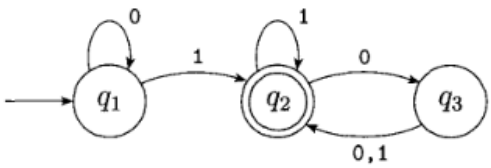


Figure 2.2: An example of a State Machine as provided in [25]

2.3.1 Passive Learning

Passive learning [26] technique is the most common algorithm for learning state machines. This algorithm tries to learn the simplest possible model from a set of traces extracted from sequential data, which should demonstrate the behavior of the system that will be modeled. These traces can contain both positive and negative traces. Positive traces are the ones which belong to the targeted language, so they should be accepted by the produces state machine. On the other hand, negative traces do not belong to the targeted language so the should be rejected by the state machine. However, is is common that the training data contain only positive data, which is the case in this project. The limitation of this method is that the finite number of traces which are provided as a training set may not capture the whole behavior of the modeled system. Conclusions can not be inferred from

unobserved behaviors, thus the modeled wont be able to model behaviors that are absent on the training data. The passive learning technique is demonstrated in Figure 2.3

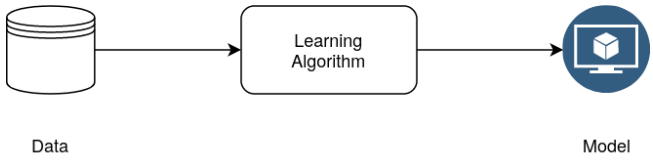


Figure 2.3: Passive learning

2.3.2 Active Learning

In 1987, Angluin proposed the L^* algorithm which implements an active learning method for learning state machines [27]. Most of the active learning algorithms proposed and developed since then are based in Angluin's approach. This approach is depicted in figure 2.4.

Active learning requires a continuous interaction between a learner and a teacher for a model to be learned. The role of the teacher is usually played by an oracle. Active learning learns the targeted behavioral models by performing repeatedly experiments on it.

There are two types of queries that the learner executes, in order to learn a behavioral model. The first type are the membership queries where the learner executes queries consisting of strings from the input set and gets a response (yes/no) from the teacher indicating whether the proposed string belongs to the language or not. In this way, and with the training data, the leaner builds a model. The second type of queries are the equivalence queries, where the learning asks the teacher if the model leaned is equivalent with the targeted model. If so, the learning process is terminated. If the proposed model is wrong, the teacher returns a counterexample which is a string from the input data that is accepted by the targeted model and rejected by the proposed model and vice-versa. By analyzing the counterexample, the learner refines its observation table and construct an improved version of the model. This process is continued until the targeted model is obtained.

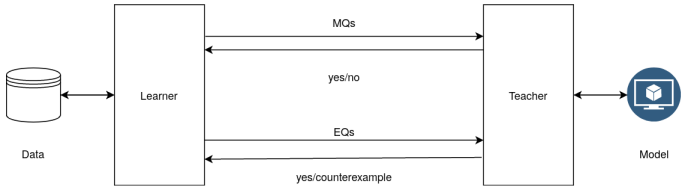


Figure 2.4: Active learning

2.4 Flexfringe overview

The tool used to learn the Finite State Machine (FSM) model is *flexfringe* [19]. It is an open-source software tool used to learn different variants of FSMs from traces with a specific format, as presented in section 2.4.1. Flexfringe is an evolution of DFASAT, which won the StaMiNa competition[28], [29], and had a very good performance on SPiCE competition [30].

It's core state-merging algorithm is the evidence-driven blue-fringe state-merging algorithm [31]. It was implemented by using a union-find data structure to store and undo merges, which makes it computationally efficient. However, this algorithm was adapted so that it can deal with different heuristics and model types, including probabilistic and non-probabilistic deterministic FSMs. Also, flexfringe was implemented in a way that is easy to be extended by the user by adding their own heuristic for the state-merging process. Furthermore, it contains many parameters that can be used to tune the model and adjust its functionality according to the requirements of the examined task. Last but not least, two important features of flexfringe is that it can make use of sink states, and it can also operate in streaming mode [32].

2.4.1 Input Format

The format of the inputs files for the *flexfringe* tool, is based on the Abbadingo competition format. In particular each input file contains a header line with two numbers, the number of traces (sequences) that the file contains and the number of different symbols over all the sequences. Each of the following lines is a sequence of elements separated by space, representing one sequence. More specifically, the first element is the numeric label of the sequence, the second is the length of the sequence and the following part is the symbol sequence. The elements of the sequence can contain any character except from space and backslash, while the may have data attached with them separated with backslash. An example of such a symbolic sequence can be seen in figure 2.5. For this project we train our model only with one type of sequences, so the first number of these line has no meaning. Furthermore, for the streaming mode, the header is omitted.

label length symbol1/data1 symbol2/data2 ... symboln/datan

Figure 2.5: Flexfringe input format

2.4.2 Output Format

While flexfringe is running and the model is being created, it outputs some the action executed along with its evidence score. This action can be either *extend (x)* or *merge (m)*. The score indicates how strong are the evidence for this action. An example of such an output can be seen in figure 2.6. This output can be very informative as it let user know the decision made while building the state machine model. Thus, it help the user during the reverse-engineering process where they can tune the model parameters or add extra data, in order to obtain a better model.

```
start greedy merging
x47 m21 m21 m13 m12 m10 m9 x12 m19
x23 m19 m9 m8 m5 m3 x20 m13 x14
m7 m6 m5 no more possible merges
```

Figure 2.6: Flexfringe output example

When the State Machine learning procedure ends and the final model is obtained, it is saved as a directed graph with labels for nodes and edges, in a Graphviz's dot format. Also the user has the option to adjust the output format, by modifying some of the print methods of the tool.

2.4.3 Merge Heuristics

As we described in section 2.4.5, heuristics are used to check the consistency of the examined merge action. *Flexfringe* contains an implementation of some of the well-know heuristics in the literature. In particular, it contains the implementation of the consistency checking and the other methods and structures which are essential for the functionality of the heuristic. The heuristics included to flexfringe, as explained by C. Hammerschmidt [22], are the following:

- **EDSM** [31]: used for DFA inference from both positive and negative data. The evidence score of the heuristic is the number of states merged, while the consistency check is based on the label of the data.
- **Overlap** [29]: it is based on the EDSM used for DFA inference in the Stamina competition [28]. The evidence score of the heuristic is based on the overlapping outgoing transitions, while the consistency check is based on the number of overlapping transitions.
- **KL-divergence** [33] : used for PDFA inference. The evidence score of the heuristic is the KL-divergence between the two examined states, while the consistency check is based on the distance between the distribution of the examined states, as computed by the KL-divergence.
- **Likelihood** [34]: used for PDFA inference. The consistency check selects a model according to its log-likelihood.
- **Alergia** [35]: used for PDFA inference. The evidence score of the heuristic is the number of merged states, while the consistency check uses the Hoeffding bound to measure the distance between distributions of the two examined states.
- **Modified Alergia**[22]: a modified version of alergia heuristic implemented by Sicco Verwer and used for the SPiCE competition [30]. The two major modifications were on the order the order the algorithm tries to merge the states and on the way of dealing with low frequency symbols. More details can be found in [22].
- **RSME** [22] used for regression automata inference. The consistency check uses a mean-squared-error penalized likelihood selection condition.

2.4.4 Create new merge Heuristics

An important advantage of *flexfringe* is the fact that it can be easily extended by adding a new merge heuristic. In order to do so, the user should implement their own function for the consistency checking between the examined states, and the evidence score for the quantification of the similarity of the two states. More information about the technical details of new merge heuristic creation are provided in [19].

2.4.5 State Merging

One of the most important process during the State Machine learning procedure is the state merging. The goal of state merging is to find the smallest possible FDA that is consistent with the input data. One of the most popular algorithms for state merging is the Evidence-Driven State Merging (EDSM) algorithm.

This algorithm begins by constructing a tree-shaped automaton called Augmented Prefix Tree Acceptor (APTA) from the input traces. Every input sequence is modeled from the APTA with states and transition labels between the states. For two sequences to share a path, mean that these two sequences have a common prefix. An example of an APTA with labeled data (positive and negative) is depicted in figure 2.7.

Afterwards, all the possible merges of two states are evaluated by computing its evidence score and the one with the best score is performed[17][31]. The role of the score metric is to calculate the similarity of the future behaviors of the two examined states. This procedure is repeated iteratively until no merges with high scores are possible. The score function is defined by the merge heuristic used. The decision of whether two state can be merged or not depends on the heuristic utilized. There are many different heuristics used in the literature, some of them are described in section 2.4.3. The threshold used to determine is a merge should be considered consistent or not is an important parameter of *flexfringe*. By setting it to a low value, ”bad” merges may be allowed, which usually results into a bad model. Thus, its value should be chosen wisely. Also the merge decision if affected by the type of the nodes, meaning positive or negative. However this does not apply to this project as we learn FSMs from only positive data.

During the state merging process the two states are merges into one. So all the input transitions of these states should point to the new merged state, while the outgoing transitions of both states should be maintained and assigned to the new merged state. This procedure is depicted in figure 2.8.

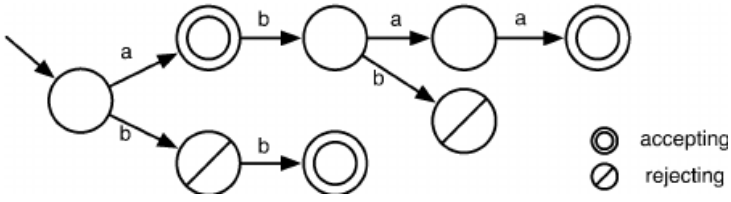


Figure 2.7: An augmented prefix tree acceptor for S = (S+ = { a, abaa, bb } , S− = { abb, b }) . The start state is the state with an arrow pointing to it from nowhere, Source: [36]

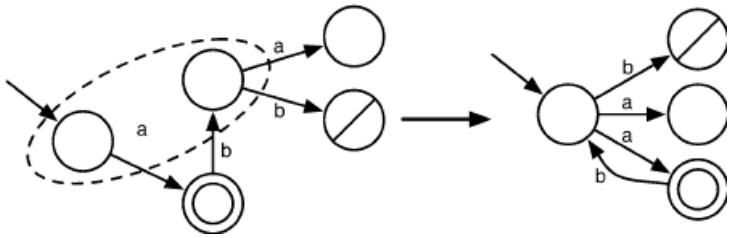


Figure 2.8: A merge of two states from the APTA from figure 2.7, Source: [36]

2.4.6 Red Blue Merging Algorithm

The state merging algorithm used by *flexfringe* is similar to the Red-Blue Fringe state merging algorithm, introduced in [17]. The main purpose of this method is to reduce the number of pairs which are possible to be merged. In order to do so, it follows the state merging procedure described in section 2.4.5, but has a restricted number of pairs that are tested for merge. In particular, it keeps two sets, one of red nodes and one of blue nodes, where red nodes are parts of the targeted DFA and blue nodes are candidate nodes for merging. Initially, the root node is colored red and all the children nodes blue. The remaining nodes are colored white. Then the following actions are performed:

1. Evaluate all the possible merges between red and blue nodes.
2. If there exists a blue node that cannot be merged with any of the red nodes, promote the shallowest such blue node to red. Then go to step 1
3. Otherwise, perform the merge with the highest score among the examined ones. Then go to step 1.
4. Halt, if there is no possible valid merge.

For the actions described above, it can be observed that once a node is colored red, then it never changes as it is assumed that it has been correctly identified by the previous iterations. An example of an APTA in an intermediate step of the Red Blue Merging process with its nodes colored, can be seen in figure 2.9.

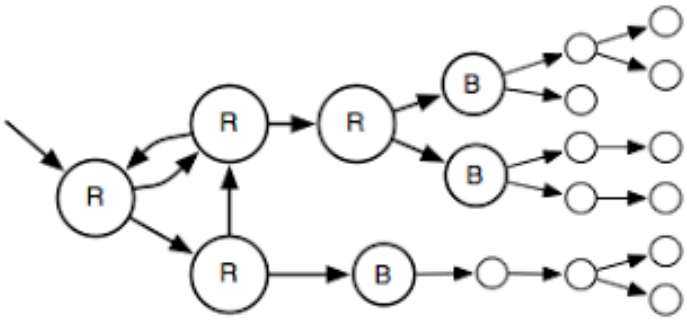


Figure 2.9: The Red Blue Merging Algorithm framework, Source: [36]

2.5 Neflow Data

NetFlow as a network protocol was introduced by Cisco in 1996. It is used to collect information about IP network traffic and monitor the network traffic activity. An overview of the basics of way the Netflow data are collected can be found in [37]. The main components of a NetFlow monitoring setup can be seen in figure 2.10 and are the following:

- **Flow exporter**: responsible for the aggregation of the packets into flows and their exportation to the flow collector.
- **Flow collector**: gets the flows from the flow exporter, stores and pre-processes them.
- **Analysis application**: analyzes the received flow data.

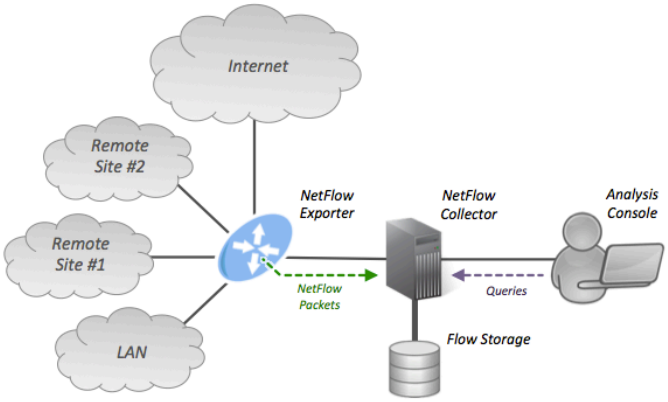


Figure 2.10: Main components of a NetFlow monitoring setup[2]

[2]Source: https://en.wikipedia.org/wiki/File:NetFlow_Architecture_2012.png

Netflows are sequences of packets passing on network interface, from a source host to a destination host. Each NetFlow is characterized by several properties, derived from the aggregation of packet-based features. These properties differ according to the NetFlow collector tool used to report the NetFlows. The most important ones, which are reported by most of the tools, can be found in section 5.1.2

Name	Description
Start Time	The timestamp of the first packet sent
Duration	The duration of flow
Source IP	IPv4 source address in the packet header
Destination IP	IPv4 destination address in the packet header
Source Port	Source port in the transport header
Destination Port	Destination port in the transport header
Protocol	The transaction protocol (TCP,UDP, ICMP, ...)
Packets	The total transaction packet count
Bytes	The total transaction bytes

Table 2.1: NetFlow data fields

Most of the projects analyzing network traffic are using NeTflow records instead of actual packet captures. The reasons behind this choice is that NetFlows does not capture the actual content of the message so they preserve their privacy. Thus, they are more frequently logged by the network operators. The only disadvantage of the usage of Netflows is that because of the high abstraction level, some information is lost.

2.6 Locality Sensitive Hashing

The idea of Locality-Sensitive Hashing (LSH) was introduced in 1998 by P. Indyk and R. Motwani [18]. It's initial purpose was to approximate the nearest neighbor algorithm in sublinear time. In contrast to Hashing, whose purpose is to hash different items in different buckets, LSH tries to hash similar items in the same bucket with high probability. Usually, the number of data objects is usually much larger than the number of buckets. In particular, the key idea of LSH is to use several hash functions such that objects that are close in the original feature space have high probability of collision after hashing. On the same way, objects that are not close in the original feature space will be assigned to different buckets with high probabilities. Collision means means that the two items will be assigned to the same bucket. In other words, LSH preserves the locality of the items by exploiting locality-sensitive hash functions. An example of a hash function that maps a set of objects to a smaller set of buckets is depicted in figure 2.11.

Let H be a family of hash functions mapping $\mathbb{R}^d$ to some universe U, and D a similarity measure. We choose a function h from H uniformly at random. The family H is called locality sensitive if it satisfies the following condition.

Definition 2.6.1 *(Locality-sensitive hashing) A family H is called (r_1, r_2, p_1, p_2)-sensitive if for any two points $p, q \in \mathbb{R}^d$ it applies that:*

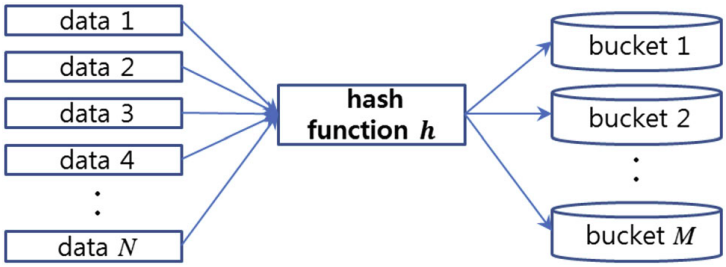


Figure 2.11: A hash function that maps objects into buckets. Source: [38]

- *if* $D(q,p) \geq r_1$ *then* $P_{r_H}[h(q) = h(p)] \geq p_1$,
- *if* $D(q,p) < r_2$ *then* $P_{r_H}[h(q) = h(p)] \leq p_2$.

In order for a locality-sensitive hash (LSH) family to be useful, it has to satisfy the inequalities $p_1 > p_2$ *and* $r_1 < r_2$.

Two representative techniques of locality sensitive hashing functions, which are the ones used on this project are presented in the following subsections.

2.6.1 P-stable distribution

A popular hashing technique used in Locality-Sensitive Hashing is the one based on p-stable distributions as introduced by Datar et al. [39]. According to [39], "a distribution D over $\mathbb{R}$ is called p-stable, if there exists $p \geq 0$ such that for any n real numbers $v_1, \ldots, v_n$ and i.i.d. variables $X_1, \ldots, X_n$ with distribution D, the random variable $\sum_i v_i \cdot X_i$ has the same distribution as the variable $(\sum_i |v_i|^p)^{1/p} \cdot X$, where X is a random variable with distribution D". It uses the l_p distance $\|x_i - x_j\|_p$, where x_i and x_j are two vectors, with p=1 or p=2, to decide the decide the bin that the input will be assigned to. This method uses a randomly selected vector x with entries selected at random from a p-stable distribution, to assign each data point to a bucket, Particularly, the data points are projected onto the vector and these projections are partitioned into a set of buckets (bins), which are intervals of equal length. This process is depicted in figure 2.12 The index of the bucket is decided by the following formula:

$$b = \left\lfloor \frac{\vec{x} \cdot \vec{u} + b}{w} \right\rfloor \tag{2.2}$$

where $\vec{x}$ is the random vector, $\vec{u}$ is the data point, w is the width of each bin, and b is a random variable uniformly distributed between 0 and w that makes the quantization error easier to analyze, with no loss in performance.

In particular, to hash the input under their Euclidean (l_2) distance, it chooses a 2-stable distribution, like the Gaussian distribution with density function $g(x) = \frac{1}{\sqrt{2*\pi}} e^{\frac{-x^2}{2}}$, to create the x vector and project the input vector into it. On the other hand, in order to hash the input under their Manhattan (l_1) distance, it chooses a 1-stable distribution, like the Cauchy distribution with density function $c(x) = \frac{1}{\pi} \frac{1}{1+x^2}$, to create the x vector.

By the definition 2.6.1 the biggest the difference between p_1 and p_2, the strongest the Locality-Sensitive hashing (LSH) family is. This can be achieved on the p-stable distributions LSH by performing more than one dot products in parallel. By performing k dot products, the ratio of the probabilities that points close in the initial feature space will be assigned to the same bucket after hashing increases, since $(\frac{p_1}{p_2})^k > \frac{p_1}{p_2}$.

In order to perform these k quantizations, we apply the formula equation (2.2) k times. On this way we obtain k bin assignments for each data point, with the intention that similar points will fall in the same bucket in all the quantizations.

The width w of the bucket, is an important parameter of this technique. By increasing the width w of the bucket, the number of data points falling into the same bucket will be increased, and vice-versa. Thus, there is a trade-off between the size of the LSH table and the number of points to consider when searching for similar data points. When the bucket size is large, there will be a small table with more points to consider for the final search due to the collisions. On the other hand, when the bin size is smaller, the table will be bigger but the final search will require less comparisons.

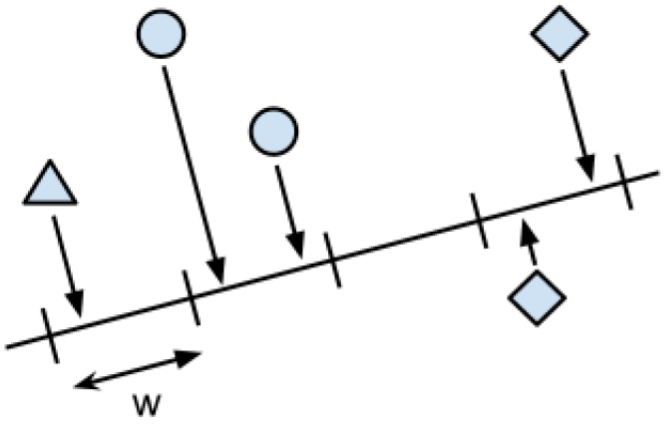


Figure 2.12: Random Projection LSH technique [40]

2.6.2 Random Hyperplanes

Another popular hashing technique used in Locality-Sensitive Hashing is the random-hyperplane method as as introduced by [41]. It uses the angle-based distance $\theta(x_i, x_j) = \arccos \frac{x_i^T \cdot x_j}{\|x_i\|_2 \|x_j\|_2}$, where x_i and x_j are two vectors, to decide the decide the bin that the input will be assigned to. Random hyperplanes method uses multiple randomly selected hyperplanes $H = \{h1, h2, \ldots, h_k\}$ to partition the data space into several parts. Each of these hyperplanes partitions the data space into 2 parts (sides). Then it assigns a label to each data point according to side of the hyperplane they belong, meaning that the value 1 is assigned to one side while 0 is assigned to the other side. By concatenating the bits generated for each hyperplane, each subspace is univocally labeled by a hash code. Each of

these subspaces (hash codes) is a bucket for the LSH. An example of this technique can be seen in figure 2.13. There we have three randomly selected hyperplanes h_1, h_2 and h_3, and the hash code (bucket) for each of the partitioned subspaces.

A hyperplane h_i in $\mathbb{R}^n$ space can be expressed:

$$h_i(x) = c_0^i + c_1^i x_1 + c_2^i x_2 + \ldots + c_n^i x_n = c_0^i + c^i \cdot x \tag{2.3}$$

where $c^i = (c_1^i, c_2^i, \ldots, c_n^i)$ is a vector in $\mathbb{R}^n$ whose components are sampled at random from a Gaussian distribution, like the standard Gaussian distribution $N(0,1)$. It has been shown that for two vectors x_i and x_j, the possibility that their hash value is the same is: $P(h(x_i) = h(x_j)) = 1 - \frac{\theta(x_i, x_j)}{\pi}$, where $\theta(x_i, x_j)$ is the angle between the two vectors x_i and x_j.

Let s_x be the label of the bucket for a data point x in $\mathbb{R}^n$ space. As mentioned above, the label s_x is a bit string. For the data point x , the i_{th} bit $s_x[i]$ of the bit string label with respect to the hyperplane h_i is computed as follows:

$$s_x[i] = \begin{cases} 1 & \text{if } c_0^i + c^i \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

After the bucket label $s_x[i]$ has been computed for all the hyperplanes h_i, the bucket s_x where the data point belongs is decided. As in the random projection technique, an important parameter of the random hyperplanes method is the number of quantization bins. Here, the number of the buckets is decided by the number of random hyperplanes we are using. Specifically, by using k random hyperplanes, we can have at most 2^k buckets, as each hyperplane partitions the original space into two subspaces.

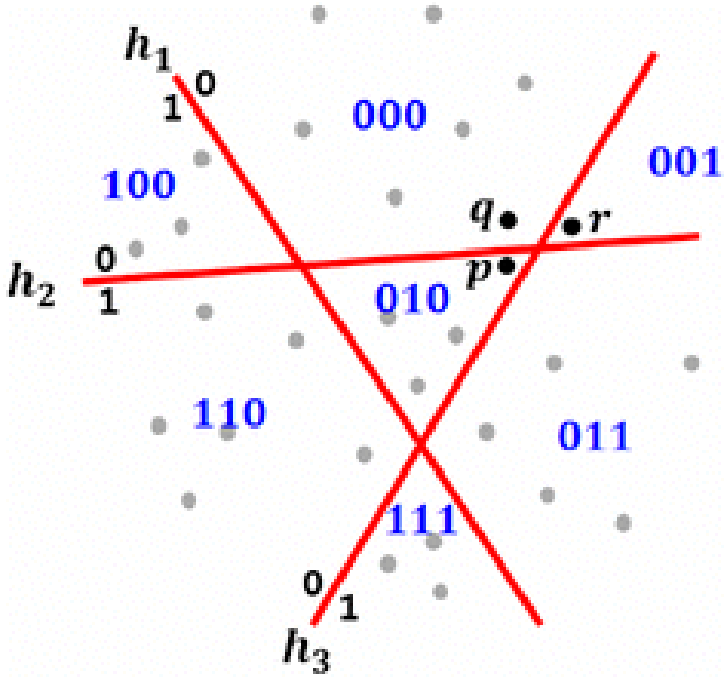


Figure 2.13: Random Hyperplane LSH technique [38]

2.7 Evaluation metrics

In order to evaluate the results of a binary classification task, the result of the classification task for an instance should be compared with its ground-truth (label). The outcome of the classifier is considered positive if it classifies the instance to one of the classes, and negative if it classifies it on the other class. Also, a prediction is considered as true if the predicted value for a sample is the same as the label, otherwise it is considered as false. Given that, we have four possible outcomes for the outcome of the classification process:

- True Positive (TP): The classifier assigns correctly the instance to the positive class.
- False Positive (FP): The classifier assigns the instance to the positive class, but the instance belongs to the negative class.
- True Negative (TN): The classifier assigns correctly the instance to the negative class.
- False Negative (FN): The classifier assigns the instance to the negative class, but the instance belongs to the positive class.

These possible outcomes of the evaluation process can be visualized in a confusion matrix, as depicted in figure 2.14. The ideal outcome will be the lack of false predictions. However, in most of the cases this is not possible. Instead, there is a trade-off between

the reliability and the sensitivity of the system. More specifically, for the purpose of this project where malicious hosts should be detected, it is important not to have false negatives (undetected malicious hosts) because the consequences of a cyber attack to a system can be very harmful. On the other hand, there are systems for which the reliability is more important, thus the number of false positives, also called false alarms, should be eliminated.

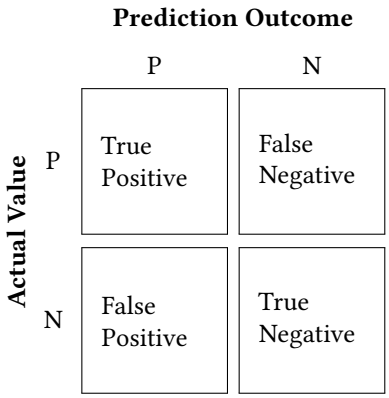


Figure 2.14: Confusion matrix

3

Related work

This chapter contains works related to the topic of this thesis, that is network anomaly detection. Initially some works are presented about the different techniques used to approach the anomaly detection topic. Afterwards, some more specific studies about the usage of the State Machines, which is the model used on this project, are introduced. Finally, there is a brief description of the works introduced several Locality Sensitive Hashing techniques and their usage.

3.1 Anomaly Detection Algorithms

There are many studies that focus on detecting anomalies in network data. As a result, there are several different kind of algorithms and techniques used to approach this task. There are several surveys[42][43][44][45] about a plethora of different network anomaly detection algorithms. These algorithms can be separated in the categories described below.

3.1.1 Classification-based

Classification is the task of predicting in which of the possible class a new observation belongs. The model used to perform the classification is trained on a dataset which contains observation from all the possible classes. Each of these observations has several attributes (features), which can be either numerical or categorical. It's classification performance is evaluated on an unseen dataset containing observations with the same features as the ones on the training dataset. The classification model can be linear of non-linear. In the task of network anomaly detection (intrusion detection), there usually many features and they are both numerical (e.g. bytes exchanged) and categorical (e.g. protocol). Two of the most popular classification-based techniques are discussed below.

Support vector machine (SVM)

Support vector machine [46] is a supervised machine learning technique, whose objective is to find a hyperplane that maximizes the margin between the data points of both classes. this process can be observed in figure 3.1. Hu et al. in [2] used Robust Support Vector Machine (RSVM) [47] to detect anomalies over noisy data, and compared it's performance

over a normal SVM. The presence of noisy data in the training set, leads in a highly non-linear decision boundary and thus introduces an overfitting problem on standard SVMs. RSVMs solve this issue by integrating an averaging technique which smoothens the decision boundary and controls the regularization. In addition, RSVMs are faster because they produce significantly less support vectors in comparison with the standard SVMs. When tested on noisy data, RSVM's attack detection rate was 81.8% with less than 3% false positive rate while SVM had an attack detection rate of 54.2% with less than 3% false positive rate.

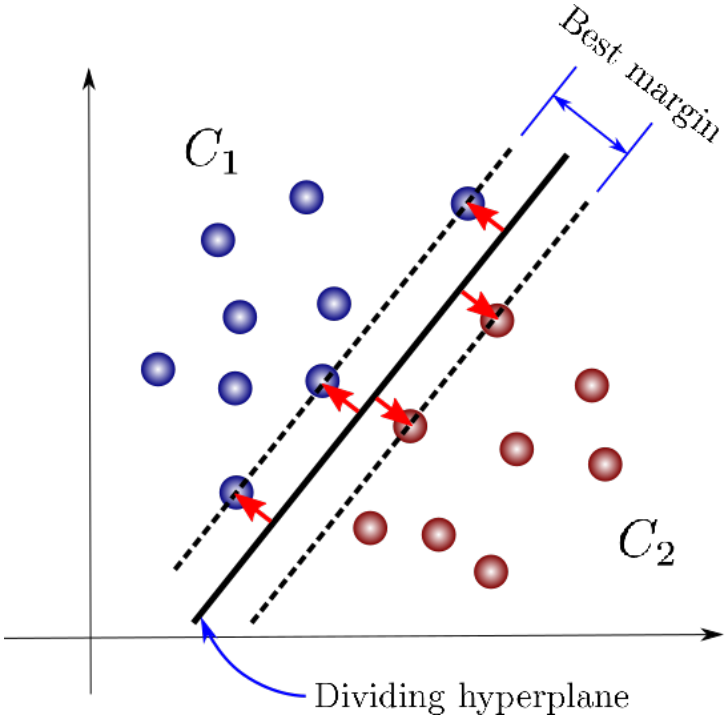


Figure 3.1: SVM [1]

However, B.Schölkopf et al. [3] adapted the SVM algorithm into a unsupervised machine learning algorithm, called One Class SVM. One Class SVM which is an one class classifier, gets data of one class and it's objective is to determine a function which is positive when the examined sample is inside a boundary defined by the training data and negative elsewhere. This process is depicted in figure 3.2. E. Eskin et al. in [4] used One Class SVM, among other unsupervised techniques, to detect anomalies in dataset of network connection records named KDD Cup[2]. The algorithm used tries to find hyperplanes which separate the data instances from their origins with the maximal margin, and subsequently solves an optimization problem to determine the best hyperplane. They achieved

[1]Source: https://miro.medium.com/max/1088/1*6U9NrruycDBsPOyivpn8UQ.png
[2]http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

a detection rate of 98% with a 10% false positive rate.

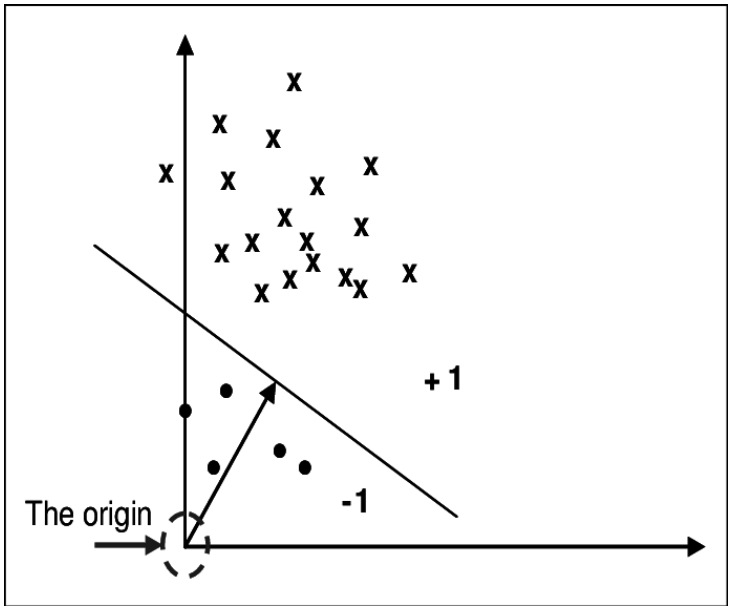


Figure 3.2: One class SVM [3]

Artificial Neural Network (ANN)

Artificial Neural Networks (ANNs) are computing techniques which consist of a mathematical model inspired by the neural structure of human's brain. ANNs imitate the way human brain learns, that is through experience. Their main components are neurons and connections between the neurons. They consist of an input layer, an output layer and a arbitrary number of layers between them. These layers are called hidden layers. If there is more than one hidden layer, then the it is called a deep neural network. Each layers contains neurons and the neurons between the layers are connected with a specific weight. The architecture of ANN with two hidden layers is demonstrated on figure 3.3. The most basic type of ANN is the Feed-forward Neural Network, while other types widely used are Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN).

In [5], B. Subba et al. built a ANN model in order to introduce an intelligent agent which is able to discriminate normal and abnormal audit records by identifying the underlying patterns, while is able to generalize on new and unseen records. The advantage of this technique is that a ANN is able to model complex non-linear hypotheses. The model built was a feed-forward neural network with a back propagation algorithm [6]. The data fed to the ANN were prepossessed and converted to numerical. After some experimentation, they achieved high performance in terms of accuracy and detection rate, 98.86%

[3]Source: https://www.researchgate.net/profile/Hany_Alashwal/publication/242572058/figure/fig1/AS:393295319584771@1470780319210/Classification-in-one-class-SVM.png

[4]Source: https://icdn6.digitaltrends.com/image/digitaltrends/artificial_neural_network_1-791x388.jpg

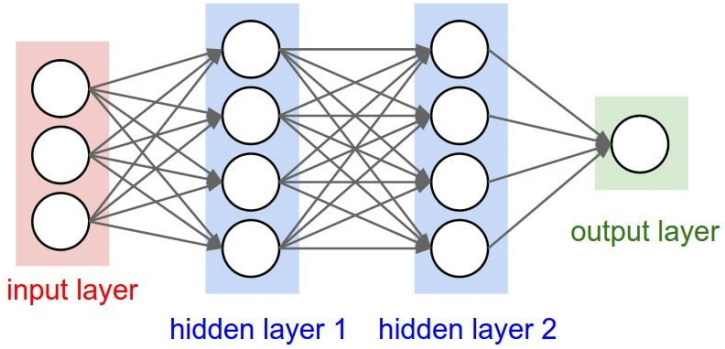


Figure 3.3: Artificial Neural Network Architecture [4]

and 95.77% correspondingly, while the computational overhead was much less that other methods like SVM.

Furthermore, J. Brown et al. in [7] used features of the application layer protocols, like HTTP, SMTP, FTP, to build an Evolutionary General Regression Neural Network (E-GRNN), in order to build a intrusion detection system that will be able to discriminate abnormal from normal behavior. They tested their system on a simulated network data obtained from the UNB ISCX Intrusion Detection Evaluation dataset [48]. Their system achieved a detection rate of 95.53% and a true positive rate of 2.11%.

3.1.2 Clustering and Outlier-based

Clustering is an unsupervised machine learning techniques which groups similar items in the same groups, called clusters [49]. Items in a cluster should be more similar to each other than items outside of this cluster. The similarity of two items is defined by a similarity measure, like Euclidean distance, and it differs according to the clustering algorithm used. Since it is unsupervised, it does not require labeled data. An example of clustering of 2-dimensional items in four clusters is demonstrated in figure 3.4a. Another concept that originate from clustering is outliers. As outliers are considered the data points whose distance (according to the similarity measure used) from the grouped data points is high. An example of outlier detection on 2-dimensional data points is demonstrated in figure 3.4b, where N_1 and N_2 are the two clusters formed and O_1 and O_2 are the two groups of outliers.

In [12], I. Syarif et al. investigated the performance of five clustering algorithms on detecting network intrusions on IDEVAL[5] dataset. The clustering algorithms evaluated were: k-Means, improved k-Means, k-Medoids, EM clustering and distance-based outlier detection. The best performance was achieved by the distance-based outlier detection algorithm with an accuracy of 80.15%. However the false positive rate was high (slightly above 20%) for all the five clustering algorithms.

A. Blaise et al. in [13] uses a clustering technique to detect malicious hosts on CTU 13 dataset. In particular, they start by characterizing a host behavior with attribute frequency

[5] https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-dataset

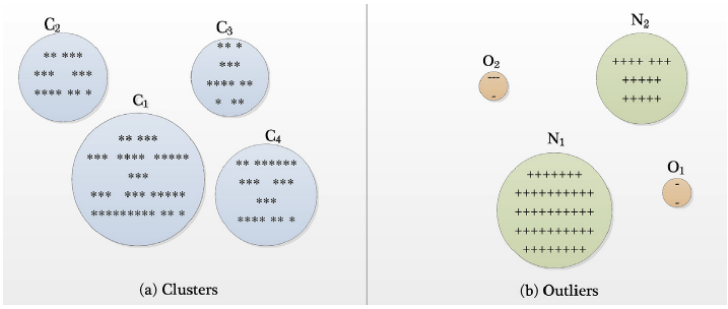


Figure 3.4: Clustering, Source: [44]

distribution signatures. By concatenating the signature of each attribute, they formed a vector with the signature of each host. Then they are learning behaviour of benign hosts and malicious hosts, by applying the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [50] clustering algorithm to the vectors representing each host. A cluster is characterized as malicious (bot) if it contains at least one bot, else it is characterized as benign. The two hyperparameters need to be tuned for this algorithms are the number of bins used in the attribute frequency distribution, and the density in the clustering algorithm. Finally, the classification of a new host is done by finding its closest cluster and assigning it its label. The distance between a new host and a cluster is calculated by the l_1 (Euclidean) distance between the signature of the new host and the centroid of the vector. They used some of the scenarios of the dataset to train the model, and the remaining scenarios to test it. The separation of the scenarios for the test and training set was done as recommended in [20]. Regarding the results, they outperform almost every other method they are comparing to, by achieving almost 100% accuracy.

3.1.3 Statistical-based

Statistical methods are widely used for anomaly detection. They fit a stochastic model to the training data, usually to the data points that represent the normal behavior, in order to capture the normal behavior of the system. Then they evaluate an event by whether or not belongs to the fitted model. As an anomaly is considered an event that has low probability to be generated by the assumed model. To decide whether an instance belongs to a model or not, usually a statistical inference test is performed, using either a threshold or a probability condition to recognize anomalies. Statistical techniques can be parametric or non-parametric. Parametric techniques assume knowledge of the underlying distribution and use the training data to estimates it's parameters, while non-parametric techniques do not assume any knowledge of the underlying distribution.

Principal Component Analysis (PCA)

Principal component analysis (PCA) [51] is a popular statistical technique for anomaly detection in network data. PCA is defined as a dimensionality reduction technique, which maps the n correlated features (variables) of a data set onto a new reduced set of k features, where $k<n$. These k variables are called principal components, and they are orthonormal vectors (define a k-subspace) which are uncorrelated. The greatest variance of the data

variables is captured by the first principal component, the second greatest variance by the second principal component, and so on.

A. Lakhina et al. [8] [9], were the first that used PCA for anomaly detection. They used PCA to build a anomaly detection system that separates the high-dimensional space of the network features into disjoint sub-spaces. Their intuition was that through PCA they will obtain a reduced subspace of k variables which captures the behavior of the normal traffic. Thus, the remaining subspace which is formed by the remaining variables $n-k$, corresponds to the anomalies. Finally, they projected every new traffic measurement onto both sub-spaces, and they used different thresholds to classify them as normal or anomalies.

Callegari et al. [10] build an network anomaly detection system based on the idea of anomaly detection with PCA ,as proposed by A. Lakhina et al. [8]. More specifically, they introduced a novel method for identifying the specific flow that is anomalous inside the aggregates. Also they applied, together with the entropy, the Kullback-Leibler divergence to construct the time-series from the aggregate flows, which had as a result better and more stable performance. The system was evaluated on a publicly available data-set of Netflow data collected in the Abilene/Internet2 Network[6], and the best detection rate achieved was 82%.

Finally, Y. Kanda et al. in [11], by combining sketches with random projections and PCA, they created a novel method to not only detect the anomaly in the network traffic, but also identify the source IP addresses associated with it in the backbone traces measured at as single link. They evaluated their method on a part of the data from the MAWI traffic repository from the WIDE project [52].

3.2 State Machines

As described in section 2.3, a Finite State Machine (FSM), also called Finite State Automata (FSA) or Deterministic Finite Automata (DFA), is a mathematical behavioral model representing a computer problem.

In [14], C. Hammerschmidt et al. utilizes Finite State Machines to learn communication profiles from NetFlow data in real-time, and uses them predict whether a host is benign or malicious. In order to convert the data to a format that can be used for FSAs, they initially encoded the numerical features by using percentiles as bin boundaries and the categorical ones by assigning a symbol to every unique value. Then, for every flow they concatenate the encoded value of the features in order to get the symbolic representation of this flow. Afterwards, they used a sliding window of fixed time over the flows to obtain the traces. In order to analyze the amount of data needed when learning the model in real time, they used as criteria the Hoeffding bound [53] and freshness, which is a criterion the introduced. They used the dfasat software package [54] with the Alergia and Overlap heuristics to learn the model, which was evaluated by calculating the acceptance rate and setting a threshold. The dataset used for the training and evaluation of the model was the CTU-13 Dataset [20]. The results indicated that by choosing the right values for the Hoeffding bound or the freshness criterion, they were able to recognize all the malicious hosts without any false positives.

[6] https://www.internet2.edu/products-services/advanced-networking/

G. Pellegrino et al. [15] used Probabilistic Deterministic Real Time Automata (PDRTA), which are a probabilistic version of Real Time Automata[55] including guards on the transition timings, to learn behavioral fingerprints from NetFlow data and used them to detect infected hosts. They also used the CTU-13 Dataset [20] for their experiments. In order to learn from the training data the used the RTI+ [34] state machine learning algorithm. They used their own algorithm to encode each flow and then they used a sliding window of fixed time over the flows to obtain the traces. Regarding the strategies used to determine if a new host should be classified as malicious or not according to the PDRTA learned from the training data, they used two strategies, which they called error based and fingerprint based. Finally, they tested their system in two different scenarios. On the first scenario they tried to detect an already known infection (single scenario), while on the second the they trained different models for different types of infection and then tried to detect unseen botnet families, by testing if any of the learned models can detect it. The results showed that in the single scenario they had very few false positives and false negatives. Regarding the multiple scenario evaluation, the system detected most of the infected hosts but also contained many false positives. In addition, they came to the conclusion that it is beneficial to use limited NetFLow data rather than a large amount of them.

M. Su [16] implemented a framework using finite state machines, applying frequent episode rules to detect Probe/Exploit (hacking) intrusions at real-time. These types of attack are difficult to be detected by a firewall or an anti-virus software. Attack episodes were mined from the log files, they were refined and the episode rules extracted the FSM model. The FSM built was used to monitor the connections on a specific port and raise an alarm whenever a predefined alarm state was reached. After the alarm was raised, the integrated real-time firewall cut-off the malicious connection. The attack data were collected by honeypots. Each attack (episode) consisted of a sequence of events. An sliding window was employed to identify the frequent events in an episode. The experiments, showed that the implemented system was able to successfully recognize complex intrusion episodes.

3.3 Locality-Sensitive Hashing (LSH)

The concept of Locality-Sensitive Hashing was analyzed in section 2.6. It's initial purpose was to approximate the nearest neighbor algorithm in sublinear time. J. Wang et al. [56] explain the concept pf LSH, demonstrate the different kind of LSH families that can be used for the nearest neighbor search task and the kind of similarity captured by each LSH family.

M. Slaney et al. [57] also deal with the nearest neighbor approximation problem. They describe the theory and the implementation of the random projections method, and discuss it implications in terms of accuracy and speed in applications like finding duplicate pages on the web and retrieving image and music.

A. Andoni et al. [58] proposed the popular LSH method E2LSH, for the case where the objects are data points in the d-dimensional Euclidean space. According to their method, by calculating every bit in the codeword is calculated by a random linear projection and it is followed by a random threshold, the Hamming distance between codewords will asymptotically approach the Euclidean distance between the items. The performance of this method was proved near-optimal in the class of the locality-sensitive hashing algorithms.

L. Paulevé et al. [59] compare several families of space hashing functions used in conjunction with LSH algorithm in the task of searching for high-dimension SIFT descriptor [60]. The hashing families compared are random projections, lattice quantizers, k-means and hierarchical k-means. From this comparison, they concluded that unstructured quantizers as hash functions improves the accuracy of LSH with the random projection hash function as they fit the data distribution. They also compared two other querying mechanisms, namely multi-probe LSH [61] and query-adaptive LSH [62], with the one originally proposed in LSH, and conclude that there are trade-offs in terms of complexity, memory usage and recall.

4

Locality-Sensitive Hashing State-Merging Heuristic

This chapter contains the description of the proposed state-merging heuristic, which is one of the main contributions of that thesis. Initially,the functionality of this heuristic is described in detail, and then the implementation steps followed to integrate it into the flexfringe tool are presented.

4.1 Heuristic description

The main contribution of this project is the introduction of a new state-merging heuristic for DFA inference by using Locality-Sensitive Hashing (LSH) for the consistency check of the merge of two states. With this heuristic and some other modifications on the state-merging process that will be described below, we expect to build models much faster in comparison with the other heuristics used, but without a significant loss on the ”quality” of the model. An example of a state machine with 41 states built with the LSH state-merging heuristic is presented in figure 4.1.

4.1.1 Future traces distribution extraction

In order to implement our heuristic, we keep for every state of the APTA three different LSH tables where we hash all the future traces for the corresponding state. The future traces of a state are all the sequences of transition symbols from the examined state to all the other states reached from this state. For example, in the APTA presented in figure 2.7, the future traces will the ones presented in figure 4.3. The reason behind this choice is the assumption that by using an LSH structure, whose functionality was explained in section 2.6, and an appropriate hash function, similar traces will be assigned to the same bucket. In particular we store in the first table the traces of length less or equal than three, in the second the traces of length more than 3 and less or equal than 10, and in the last table we store the traces of length greater than 10. These settings are presented in table 4.1. We used three different tables in order to be able to have a representation of the short-term. mid-term and long-term future of each state, which will allow us to have a more accurate

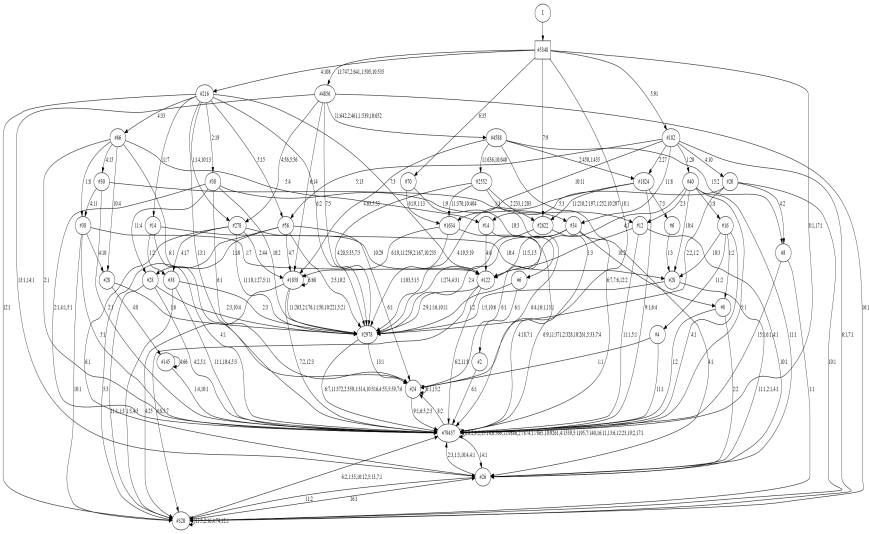


Figure 4.1: State machine with 41 states built with the LSH state-merging heuristic

representation of the future traces of a state. In order to avoid the keeping track of future traces that are very long and are not comparable with future traces with few symbols, we decided to hash traces that have 20 or less elements. Thus, after the APTA is created, the aforementioned procedure is followed for each state.

An important decision taken when designing the proposed heuristic was about the way we will handle traces of different length. As explained before, the LSH structure used to hash the future traces depends on the length of the traces. An important issue created is what actions should be followed when the length of a trace is between these bounds. In order to solve this issue, we decided to follow a simple and common practise, that is padding the vector with zeros (values of zero) until it reaches the next bound. An example can be found in figure figure 4.2, where a trace of length six is padded with four zeros. This trace it will be then hashed on the LSH structure used for the mid-term future.

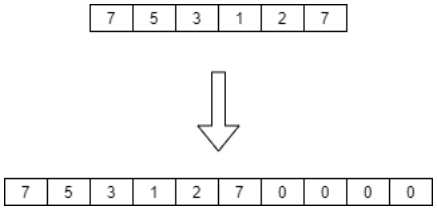


Figure 4.2: The trace is in the figure has length six, so it should be hashed using the mid-term LSH structure. However, because its length is less than ten we pad it with four zeros in the end, in order to reach the required length.

We expect this to affect the capability of LSH to assign similar traces to the same bin. However, we believe that future traces with different length should be treated as different.

By padding the vectors of future traces with zero values, we will have similar vectors with same length assigned to the same bucket. The similarity of two vectors is defined by the LSH family used. As explained before, with random hyperplane LSH the vectors in a bucket will have similar angular distance between them, while with 1-stable or 2-stable distribution LSH the vectors in a bucket will have similar Manhattan and Euclidean distance between them, respectively. When padding a vector with zero values, both the three distance measures will be big for a vector padded with zeros and another vector containing actual symbols. This difference increases when the number of zeros need to be padded in increased. Thus, traces with different length is highly possible to end up to a different bucket.

After the future traces has been hashed for each state and each of the the three LSH structures, the future traces distribution is extracted for each of them. In our case the number of possible buckets a trace can be assigned is countable, so by dividing the number of traces in each bucket with the total number of traces, we can extract a discrete probability distribution. This procedure is depicted in figure 4.4.

LSH table	Trace length
1	1-3
2	3-10
3	11-20

Table 4.1: The future traces stored in each of the three LSH tables

a, ab, aba, abaa, abb, b, bb

Figure 4.3: Future traces for APTA in figure 2.7

The hash families used for the LSH algorithm are random hyperplanes and p-stable distributions with p=1 and p=2. Each LSH family utilizes a different similarity measure to assign similar future traces to the same bin. The procedure followed to encode the flows (section 6.1.3), entails that the bigger the numerical difference between two symbols is, the more different the two flows represented by these symbols is. So the numerical difference between two symbols need to be taken into account when comparing two future traces. In addition, the order to the symbols in the traces plays an important role. For example, the traces <1,2,3> and <3,2,1> are different, although they contain the same symbols. Thus we should treat the traces as vectors and not as sets. So measuring the Jaccard similarity $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$ will not be accurate for our case. So we decided to proceed with LSH families that stores in the same bucket traces whose distance as vectors is small. Thus, we decided to p-stable distribution and random hyperplane LSH families to capture the aforementioned distances between the traces. The kind of similarities tries to capture with these LSH families are Manhattan (l_1) distance (1-stable distribution), Euclidean (l_2) distance (2-stable distribution) and angle-based distance (random projections), which are some of the simplest distance metrics for two vectors.

Each of the LSH families used has its own parameters need to be set. The most important of them, that exist in all the three families is the number of LSH buckets and the

number of hash functions, thus hash tables, used.

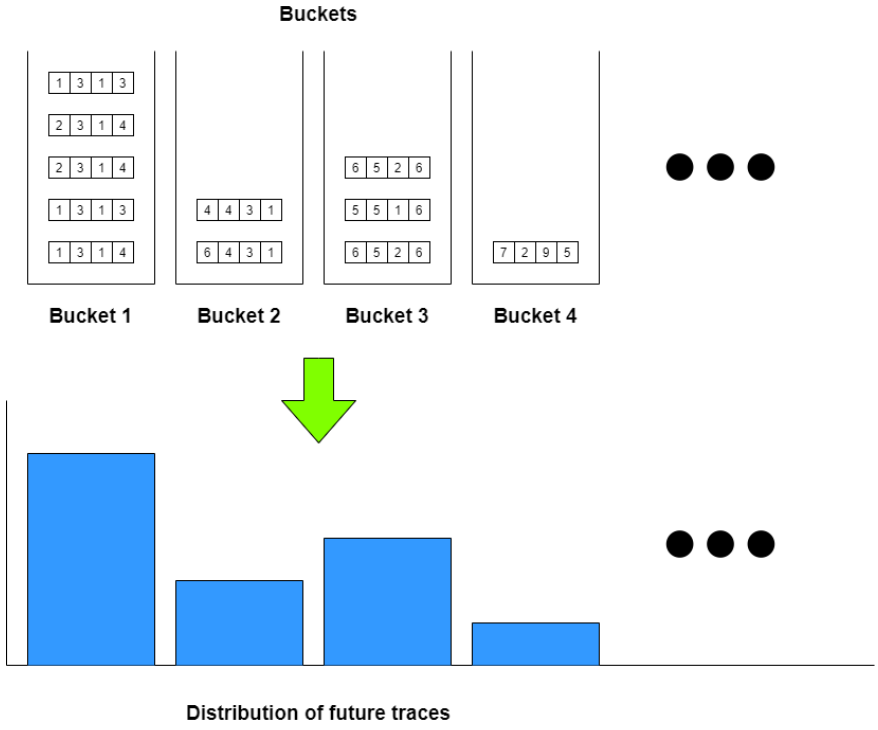


Figure 4.4: Example of extraction of the distribution of the future traces for a state from the LSH buckets

4.1.2 Future trace distribution update

An important step during the state-merging process, is what happens when two states are merged. In this case, we need to redefine the distribution of the future traces for the state resulting from the merge. In order to do so, for each bucket we add the elements of the two states and we extract the new distribution. We follow these steps for each of the three LSH structures. The opposite procedure is followed when we need to undo a merge of two states.

4.1.3 Merge consistency check

During the state machine learning process, the consistency check for the merge of two states is performed by comparing the three different distributions of the future traces coming for the two states, as they are extracted by the corresponding three LSH structures of each state. As discussed above, the distribution of the future traces for each state is extracted by measuring the percentage of traces assigned to each of the buckets of the LSH structure. After the distributions of short-term, mid-term and long-term future traces for the examined and the candidate for merge state are extracted, we compare the mea-

sure the similarity of the equivalent (for the same LSH structure) distributions by using the Kullback–Leibler (KL) divergence of the two distribution. Kullback-Leibler (KL) divergence [63], is a well known method measuring the similarity of two distributions are similar and it is defined as:

$$KL(P,Q) = \sum_{x} P(x)\log_2(P(x)/Q(x)) \tag{4.1}$$

where P and Q are the two distribution we want to compare. It should be mentioned that the lower the KL divergence value is, the more "similar" the two distributions are. Therefore, when we compare the distributions of two states, we consider a merge as consistent if the KL divergence value is lower than a user defined threshold. Otherwise, the merge is considered as inconsistent. Of course we follow this procedure for all the three different LSH structures to ensure that the distributions of short-term, mid-term and long-term future traces for the two states are similar. The condition for a merge of two states A and B to be considered as consistent is the following:

$$\begin{aligned} &KL\_divergence\_short\_term <= threshold \textbf{ and} \\ &KL\_divergence\_mid\_term <= threshold \textbf{ and} \\ &KL\_divergence\_long\_term <= threshold \end{aligned} \tag{4.2}$$

where KL_divergence_short_term, KL_divergence_mid_term and KL_divergence_long_term are the KL divergence value when comparing the aforementioned distributions of the two states for the three LSH structures. The threshold used on this decision is defined by the user and plays an important role to the state-merging process as it defines how similar two distributions should be in order to consider the merge of the states as consistent. In particular, the bigger the threshold value is, the more states the state machine will have.

The important speedup we expect to achieve from our proposed state-merging heuristic comes from the assumption that because we are comparing the distribution of the future traces in order to decide whether two states should be merged or not, there is no need to evaluate the consistency of the merging of their children states. This is an procedure followed from the Red-Blue state-merging algorithm, which is computationally expensive. In our case, we consider their merge as consistent and skip all the further checks for the merge consistency of the descendant nodes. However, this procedure makes an approximation about the consistency of the merge of the examined states, so we expect to produce slightly worse model than other heuristics which actually evaluate that consistency. For example in the merge depicted in figure 4.5, when trying to merge the nodes 1 and 2 following the Red-Blue fringe algorithm we should check the consistency of the merge of the nodes 3 and 4, so we need to check their consistency too. When using the proposed heuristic we avoid this and all the following consistency checks as we assume that we have already evaluated it by comparing the distribution of future traces.

4.1.4 Merge score calculation

Regarding the merge score used to decide the best merge between all the possible ones, we defined a score metric using the Kullback–Leibler (KL) divergence. In particular, as

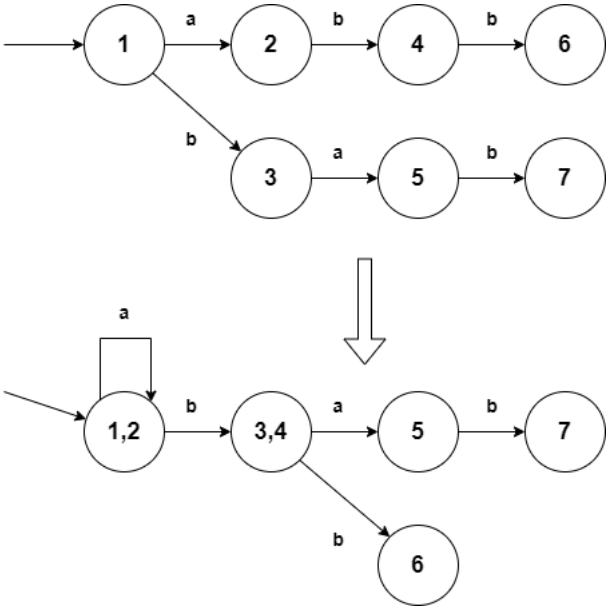


Figure 4.5: Example of the state-merging process

discussed in section 4.1.3, the smallest the value of KL divergence of the distribution of the future graces extracted by the LSH buckets is, the more similar the two distributions are. So we decided to use this value as a confidence score for the merge. However, the merge score should be high for ”good” merges and low for ”bad” merges. Given that, and the fact that the order of the KL divergence value can be very small, we decided to use the following metric as merge score:

$$\begin{aligned} merge\_score = & \frac{100000 - (KL\_divergence\_short\_term * 100000)}{3} \\ & + \frac{100000 - (KL\_divergence\_mid\_term * 100000)}{3} \\ & + \frac{100000 - KL\_divergence\_long\_term * 100000)}{3} \\ & = \frac{\begin{array}{l} 100000 * (3 - KL\_divergence\_short\_term \\ \quad - KL\_divergence\_mid\_term - KL\_divergence\_long\_term)) \end{array}}{3} \end{aligned} \tag{4.3}$$

where KL_divergence_short_term, KL_divergence_mid_term and KL_divergence_long_term are the KL divergence value when comparing the aforementioned distributions of the two states for the three LSH structures. We can easily see that the value of the merge score is the average value of three independent components $100000 - (KL\_divergence\_1 * 100000)$. Each of them represents one of the three different distributions extracted from one of the three LSH buckets, and can be easily observed that their value is inversely proportional to

the KL divergence score. Therefore, we achieved our goal as we have a score metric that is increases when the KL divergence decreases.

4.2 Implementation

In order to integrate our proposed state-merging heuristic as an evaluation class on *flexfringe* we had to implement several functionalities. In particular, as explained before, *flexfringe* has an evaluation class which is called during the state-merging process with the Red-Blue Fringe state merging algorithm. Every heuristic used during the state-merging process need to implement several functionalities (methods) which are essential for the merge of two states. The first type of methods need to implemented is about the data kept on each state:

- **Update**: Given the state that the current state is merge with, updates the data for the current state.
- **Undo**: Given the state that the current state was merge with, undoes the changed done by the merging of the two states.
- **Read from**: Called during the creation of the APTA and updates the data stored for this state.

The second type of methods need to implemented is about the data action taken during during the state-merging process:

- **Consistent**: Given two states, decides whether their merge is consistent.
- **Compute score** Given two nodes, computes the score for the merge of these two nodes. The higher the score is, the better the merge is considered.
- **Update score**: Given two states, updates the score for the examined merge.
- **Reset**: Used to reset some parameters used to during the state-merging process.

4.2.1 Future traces distribution extraction

As described above, these were the main functionalities of the evaluation class that we implemented in order to integrate my heuristic to the *flexfringe* tool. However, before describing the implementation of these methods, we need to describe an extra step taken before starting the state merging process. This was the traversal of the APTA in order to store the future traces for each state using the LSH structure. In particular, for each state of the APTA, we performed a Depth-first search (DFS) in order to find all the traces from the examined node to all the other nodes that can be reached by it. For each of these traces, I used the appropriate LSH structure according to its length as explained above, and stored it to the bucket of the LSH structure in which it belongs according to the hash function used. Finally, after we have stored all the traces, we compute the number of elements of each bucket for each of the three LSH structures and keep it on the data of this state.

4.2.2 Evaluation data

For each node of the APTA, some data should be kept in order to perform the state-merging process and build the state machine. For the implementation of the LSH heuristic, each node contains the following data:

- **Symbol occurrences**: All the symbols that occurs during the transitions from this state were stored, along with the number of occurrences of each symbol. It is used to calculate the probability of each transition for each state of the final model.
- **LSH counters**: Three counters were kept, one for each of the three LSH structures. Each counter contains the number of traces stored in each of the buckets of the LSH structure. There counter are used to extract the distribution if of the future traces for the state.

The data kept for each of the nodes, need to be updated during the state merging process of during the creation of the APTA. The function need to perform these updates were described above. The implementation of these functions is the following:

- **Update**: When two states are merged, the counter we keep about the buckets of the LSH structures of the new formatted node need to be updated. In order to do so, for each bucket we add the number of elements for the two states. This process is repeated for each of the three LSH structures. Also, another thing that needs to be updated during the merge of the merge of two states, is the symbol occurrences. This is done by adding the symbol occurrences of the two states and updating the counters accordingly. The pseudocode for its implementation is presented in alg. 1.
- **Undo**: When a merge of two nodes should be undone because it was decided that the merge is inconsistent, the update of the data need to be undone too. In order to do so, for each bucket we subtract the number of elements for the two states. This process is repeated for each of the three LSH structures. Also the symbol occurrences which were combined for the two states should return into their previous form. In order to do so we we subtract the symbol occurrences of the blue node from the symbol occurrences of the red node. The pseudocode for its implementation is presented in alg. 2.
- **Read from**: In order to store the symbols for the transitions of the node, along with their count, we update them while creating the APTA. In particular, for each symbol read we add one to the count of this symbols for current state. The pseudocode for its implementation is presented in alg. 3.

4.2.3 Evaluation function

There are also some data need to be stored for the evaluation function. This data are reset every time a new merge is tested. It is data required by the heuristic when evaluating if a merge is consistent or not. In the LSH state-merging heuristic, these are the following:

- **First merge**: During the state-merging process we keep a Boolean variable indicating whether this the first merge tried. This variable is used to make faster the

Algorithm 1: LSH data update

```
input  : a state S merged with the current state
output : Updates the symbol occurrences and the 3 counters
foreach symbol, count in other_state->symbol_occurrences do
    occurrences[symbol] ⟵ occurrences[symbol] + count
end
for i ⟵ 0 to number_of_buckets do
    short_term_counter[i] ⟵ short_term_counter[i] + S->short_term_counter[i];
    mid_term_counter[i] ⟵ mid_term_counter[i] + S->mid_term_counter[i];
    long_term_counter[i] ⟵ long_term_counter[i] + S->long_term_counter[i];
end
```

Algorithm 2: LSH data undo

```
input  : a state S merged with the current state
output : Updates the symbol occurrences and the 3 counters
foreach symbol, count in other_state->symbol_occurrences do
    occurrences[symbol] ⟵ occurrences[symbol] - count
end
for i ⟵ 0 to number_of_buckets do
    short_term_counter[i] ⟵ short_term_counter[i] - S->short_term_counter[i];
    mid_term_counter[i] ⟵ mid_term_counter[i] - S->mid_term_counter[i];
    long_term_counter[i] ⟵ long_term_counter[i] - S->long_term_counter[i];
end
```

merges of the descendant state of the initial nodes examined for merge. As it was discussed above, if the initial merge is consistent then we also consider as consistent the merge of their children nodes, so by using this variable we avoid the the unnecessary consistency checks.

- **Merge score**: A variable used to store the score of the merge, which is inversely proportional to the Kullback–Leibler (KL) divergence score of the distributions extracted by the LSH buckets.

Furthermore, we had to implement the basic functionalities presented above for the evaluation function, in order to decide about the consistency of a merge and compute the score of each merge according to our proposed method. The methods implemented are the following:

Algorithm 3: Read from

```
input  : The symbol s read
output : Updates the symbol occurrences for the current state
occurrences[symbol] ⟵ occurrences[symbol] + 1
```

- **Consistent**: Evaluates the consistency of a merge of states. First of all, if this is not the first merge done for this state, which means that their descendant states are tested for merge, it evaluates the merge as consistent without checking any other criteria. The reasons for this choice were explained above. If this is the first merge checked, then we evaluate the merge consistency with out standard criteria. Initially we extract the distribution of the future traces from the LSH buckets for each of the two examined nodes and each of the three LSH structures. Then we compute the three KL divergence similarity scores between the distribution of the future traces of each node, one for each LSH structure. Finally we consider the merge as consistent if each of the three KL divergence similarity scores is less or equal than the user defined threshold. The pseudocode for its implementation is presented in alg. 4. The function *get_distribution* computes the discrete probability distribution given the counter for each bucket.

- **Compute merge score**: Computes the score of the merge, which is used to choose the best between all the possible merges of the Red-Blue fringe state-merging algorithm. If this is not the first merge done for the initial red and blue pair of nodes, then we already know the score of the merge. Otherwise, we need to compute it. Initially we extract the distribution of the future traces from the LSH buckets for each of the two examined nodes and each of the three LSH structures. Then we compute the three KL divergence similarity scores between the distribution of the future traces of each node, one for each LSH structure. Finally we compute the score of the merge with the following formula:

$$merge\_score = \frac{100000 * (3 - kl\_divergence\_short\_term - kl\_divergence\_mid\_term - kl\_divergence\_long\_term))}{3} \tag{4.4}$$

  where kl_divergence_short_term, kl_divergence_mid_term, and kl_divergence_long_term are the KL divergence scores for the first, second and third LSH structure respectively. The pseudocode for its implementation is presented in alg. 5.

- **Update merge score**: The score is only computed on the first merge and it is not updated during the merge of the descendant states. Thus, this function is not used in our case.

- **Reset**: During the reset of the data of the evaluation function, the merge score is set to and the Boolean variable *first_merge* is set to True, because the next time the evaluation function will be used, we we start examining from the merge of the initial nodes. The pseudocode for its implementation is presented in alg. 6.

Algorithm 4: Consistent

input : The two nodes n1 and n2, whose merge consistency is checked
output : A Boolean variable indicating if the merge is consistent or not
if *first_merge==false* **then**
  return true
end
first_merge ⟵ false
n1_short_term_distribution ⟵ **get_distribution**(n1->short_term_counter)
n2_short_term_distribution ⟵ **get_distribution**(n2->short_term_counter)
n1_mid_term_distribution ⟵ **get_distribution**(n1->mid_term_counter)
n2_mid_term_distribution ⟵ **get_distribution**(n2->mid_term_counter)
n1_long_term_distribution ⟵ **get_distribution**(n1->long_term_counter)
n2_long_term_distribution ⟵ **get_distribution**(n2->long_term_counter)
kl_divergence_short_term ⟵
 compute_kl_divergence(n1_short_term_distribution,
 n2_short_term_distribution)
kl_divergence_mid_term ⟵ compute_kl_divergence(n1_mid_term_distribution,
 n2_mid_term_distribution)
kl_divergence_long_term ⟵ compute_kl_divergence(n1_long_term_distribution,
 n2_long_term_distribution)
if *kl_divergence_short_term > LSH_KL_THRESHOLD or kl_divergence_mid_term > LSH_KL_THRESHOLD or kl_divergence_long_term > LSH_KL_THRESHOLD* **then**
  return false
else
  return true
end

Algorithm 5: Compute merge score

```
input  : The two nodes n1 and n2, whose merge score we want to compute
output : The score of the merge of the two nodes
if first_merge==false then
    return true
end
first_merge ⟵ false
n1_short_term_distribution ⟵ get_distribution(n1->short_term_counter)
n2_short_term_distribution ⟵ get_distribution(n2->short_term_counter)
n1_mid_term_distribution ⟵ get_distribution(n1->mid_term_counter)
n2_mid_term_distribution ⟵ get_distribution(n2->mid_term_counter)
n1_long_term_distribution ⟵ get_distribution(n1->long_term_counter)
n2_long_term_distribution ⟵ get_distribution(n2->long_term_counter)
kl_divergence_short_term ⟵
  compute_kl_divergence(n1_short_term_distribution,
  n2_short_term_distribution)
kl_divergence_mid_term ⟵ compute_kl_divergence(n1_mid_term_distribution,
  n2_mid_term_distribution)
kl_divergence_long_term ⟵ compute_kl_divergence(n1_long_term_distribution,
  n2_long_term_distribution)
kl_score ⟵ 10 *(3 - kl_divergence_short_term - kl_divergence_mid_term -
  kl_divergence_mid_term) / 3
return kl_score
```

Algorithm 6: Reset

```
output : Resets the value of the variables of evaluation function
kl_score ⟵ 0
first_merge ⟵ true
```

5

Data Exploration

This chapter contains the description of the two datasets used to measure the efficiency of our proposed heuristic in building state machines to detect malicious host and to measure its runtime efficiency. Initially there is a description of the CTU-13 dataset and its features. An analysis on the features values is conducted to justify the choice of the features used for the encoding of the flows and the choice of sliding window technique. Also, some graphs about the number of flows per host are presented. Finally the datasets of the PAUTOMAC competition are described, along with their generation procedure and some statistics about them.

5.1 CTU-13 dataset

5.1.1 Data Description

The dataset we used to evaluate the performance of the proposed state-merging heuristic is the CTU-13 dataset[20]. More specifically, we used the bidirectional NetFlow format of this dataset, as the author of CTU 13 suggest, because it has better labels and better quality of data in comparison with the unidirectional dataset. CTU-13 is a publicly available labeled dataset with network traffic that was captured from a large network in the CTU University, Czech Republic, in 2011. The dataset is organized in 13 scenarios, each of which consists of malicious, benign and background NetFlows. The malicious NetFlows of each scenario comes from a specific type of botnet which used several protocols and performed different actions[1]. More details on the creation of the dataset, like the design of the botnet scenarios, the network topology, the dataset preprocessing and labeling, can be found in [20].

As mentioned above, each flow was labeled as normal, botnet or background. The distribution of the flow labels for each of the 13 scenarios can be seen in table 5.1. As it can be noticed, in all the scenarios the majority of the flows (above 90% in almost every scenario) are labeled as background flows. Regarding, the flows labeled as botnet or normal, the situation differs according to the examined scenario, with some scenario having more

[1]https://www.stratosphereips.org/datasets-ctu13

botnet flows while other have more normal ones. It should be mentioned that the background flows were removed for the purpose of this project since no safe conclusion can be drawn about whether they come from a malicious or a benign host. Also the duration of recording of each scenarios along with the number of packets exchanged in it (after the background flows were removed) can be observed in table 5.2.

Scenario	Background	Botnet	Normal
1	2753288 (97.47%)	40961 (1.45%)	30387 (1.08%)
2	1778061 (98.34%)	20941 (1.16%)	9120 (0.50%)
3	4566929 (96.95%)	26759 (0.57%)	116950 (2.48%)
4	1093228 (97.52%)	2580 (0.23%)	25268 (2.25%)
5	124252 (95.7%)	901 (0.69%)	4679 (3.60%)
6	546795 (97.83%)	4630 (0.83%)	7494 (1.34%)
7	112337 (98.47%)	63 (0.06%)	1677 (1.47%)
8	2875281 (97.33%)	6127 (0.21%)	72822 (2.47%)
9	1872554 (89.70%)	184987 (8.86%)	29967 (1.44%)
10	1187592 (90.67%)	106352 (8.12%)	15847 (1.21%)
11	96369 (89.85%)	8164 (7.61%)	2718 (2.53%)
12	315675 (96.99%)	2143 (0.66%)	7653 (2.35%)
13	1853207 (96.26%)	40003 (2.08%)	31939 (1.66%)

Table 5.1: Distribution of the flow labels in each scenario of the dataset

Scenario	Duration(hrs)	# Packets
1	6.12	1115476
2	4.19	247622
3	66.82	1025399
4	4.19	1544016
5	0.5	290674
6	2.15	241698
7	0.35	53340
8	19.47	1653553
9	5.19	2742699
10	4.76	3790613
11	0.27	84970
12	1.22	186436
13	16.37	688591

Table 5.2: Number of packets exchanged in each scenario and its duration

Furthermore, as it was mentioned before, the network traffic for each scenario contains malicious flows from a specific type of botnet. The types of botnets used were Neris,Rbot, Virut, Menti, Sogou, Murlo and NSIS.ay. The type of botnet used to create the malicious traffic for each scenario can be noticed in table 5.3. This table also contains the number

of botnet and normal hosts per scenario. Given that we evaluate the performance of the proposed state-merging heuristic by classifying hosts as malicious or benign, the number of malicious and benign hosts on each scenarios plays an important role. As it is explained in chapter 6, the number of the malicious hosts and the type of botnet are very important for the way that we split the dataset in a training and testing dataset. As it can observed from table 5.3, all the scenarios contains five or six benign hosts. When it comes to the malicious hosts, all scenarios expect 9, 10, 11 and 12 contain only one malicious hosts while scenarios 11 and 12 contain three malicious hosts and scenarios 9 and 10 contain ten malicious hosts.

Scenario	Botnet	Normal	Bot
1	1	19	Neris
2	1	14	Neris
3	1	24	Rbot
4	1	22	Rbot
5	1	22	Virut
6	1	16	Menti
7	1	13	Sogou
8	1	23	Murlo
9	10	25	Neris
10	10	18	Rbot
11	3	15	Rbot
12	3	20	NSIS.ay
13	1	24	Virut

Table 5.3: Number of botnets and normal hosts, and type of botnet for each scenario

Table 5.4 contains the main characteristics of the scenarios and their behaviors. Especially, it describes the protocols used(IRC, P2P or HTTP), and if they sent SPAM, did Click-Fraud, port scanned, did distributed denial-of-service (DDoS) attacks, used Fast-Flux techniques or if they were custom compiled.

5.1.2 NetFlow features

Each flow is is described by some features. The most important of them can be seen in table 5.5. It should be mentioned that the time difference and destination bytes were not present in the dataset, but they were easily extracted by the other features. The features start time, source IP and destination IP were not used as features for the encoding of the flows (section 6.1.3). Instead, they were only used to group together the flows of one host. Also, source and destination port features, contained many missing values, so we decided not to use them.

5.1.3 NetFlow features used for encoding

The procedure followed to convert a NetFlow in to a discrete symbol is explained in section 6.1.3. However, as it can be clearly noticed, the choice of the features used for the flow encoding should be chosen by us.

Id	IRC	SPAM	CF	PS	DDoS	FF	P2P	US	HTTP	Note
1	✓	✓	✓							
2	✓	✓	✓							
3	✓			✓				✓		
4	✓				✓			✓		UDP and ICMP DDoS
5		✓		✓					✓	Scan web proxies
6				✓						Proprietary C&C. RDP
7									✓	Chinese hosts
8				✓						Proprietary C&C. Net-BIOS, STUN
9	✓	✓	✓	✓						
10	✓				✓			✓		UDP DDoS
11	✓				✓			✓		UDP DDoS
12							✓			Synchronization
13		✓		✓					✓	Captcha. Web mail.

Table 5.4: Characteristics of botnet scenarios

Name	Description
Start Time	The timestamp of the first packet sent
Duration	The duration of flow
Source IP	IPv4 source address in the packet header
Destination IP	IPv4 destination address in the packet header
Source Port	Source port in the transport header
Destination Port	Destination port in the transport header
Protocol	The transaction protocol (TCP,UDP, ICMP, ...)
Packets	The total transaction packet count
Source Bytes	The number of bytes sent by the source IP
Destination Bytes	The number of bytes sent by the destination IP
Time difference	The time difference (ms) between this flow and the next flow

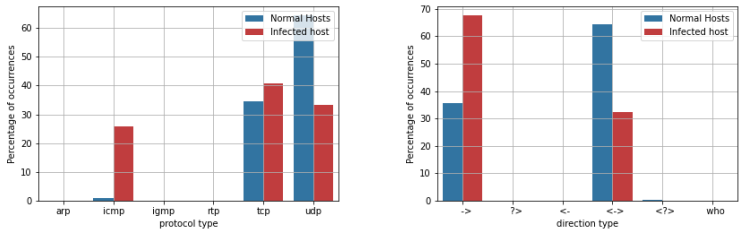
Table 5.5: NetFlow data features

In order to choose the aforementioned features, we investigated the behavior of the malicious and benign hosts. After discretizing the categorical features, the statistics presented in table 5.6 were calculated, so that an understanding on the behaviour between the infected host and the normal ones could be acquired.

From table 5.6, it can be seen that all the features presented can be indicative on the type of each NetFlow, since they demonstrate quite different behaviour between the infected and the normal hosts (mostly in term of their mean values). We decided to exclude the source and destination bytes features, as we can use total bytes features instead of these. Thus, we used the protocol, direction, duration, packets, total bytes and time difference features to encode each flow of the dataset. To further highlight the difference on the behaviour between infected and normal hosts on the two categorical features (direction and protocol), the plots presented in figure 5.1 were created.

	Host Type	mean	std	min	25%	50%	75%	max
Protocol	infected	1.1	1.17	0	0	1	2	4
	normal	0.56	0.53	0	0	1	1	4
Direction	infected	1.6	0.56	0	1	2	2	2
	normal	1.11	0.69	0	1	1	2	5
Duration	infected	38.42	227.16	0	0	0.07	2.98	3599.94
	normal	21.63	188.41	0	0	0	0.03	3600
Packets	infected	12.97	857.23	1	1	2	3	176254
	normal	18.51	436.75	1	2	2	6	68953
Source bytes	infected	7369.03	682737.1	60	74	186	1066	138738600
	normal	1080.93	94488.98	0	74	81	272	54929440
Destination Bytes	infected	1997.36	61885.39	0	0	0	202	17196410
	normal	12121.58	416735.9	0	149	149	326	68378090
Total Bytes	infected	9366.41	685662.6	60	186	365	1066	138738600
	normal	13202.51	436370.1	60	261	400	453	69267540
Time Difference	infected	291.50	813.36	0	0	11	99	21392
	normal	956.59	2488.94	0	0	1	141	20000

Table 5.6: Statistics derived from the infected and the normal hosts



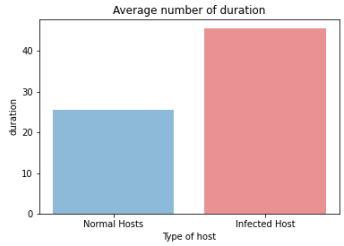
(a) Percentage of protocol types present in the flows of all scenarios (b) Percentage of direction types present in the flows

Figure 5.1: Visualization of differences in the behaviour between the infected and the normal host for the categorical features: protocol and direction

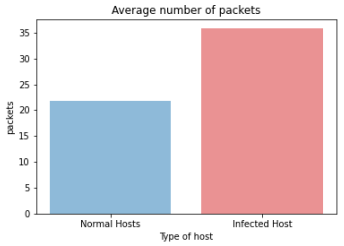
5.1.4 Distribution of the number of flows over time

As it is described in section 6.1.4, in order to use this dataset to infer a state machine model, we should first encode each flow and then use a sliding window technique to obtain sequences of symbols. When using a window of fixed time, it is important that the number of flows does not change very much over time, in order for all the windows to have similar length.

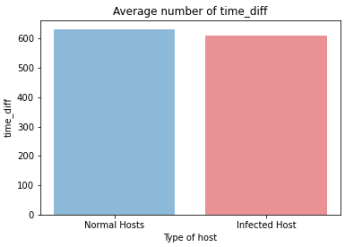
Figure 5.3 and figure 5.4 depicts the distribution of the number of flows over time for each of the 13 scenarios. As we can observe from these figures, in all the scenarios there are some time periods with many flows (spikes on the plot) and some other periods with a very small number of flows. This causes a significant problem during the window extraction process because regardless of the window time range, there would be some windows with many flows and some others with very few flows. That is the reason why we chose to windows of fixed number of flows, as described in section 6.1.4.



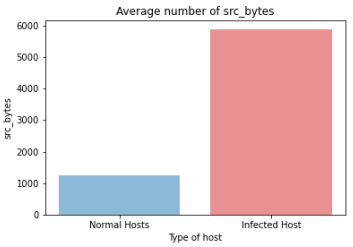
(a) Average flow duration



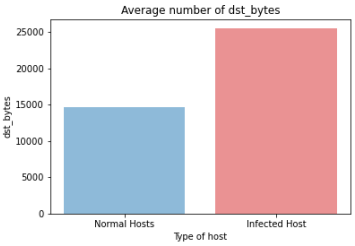
(b) Average number of packets transmitted



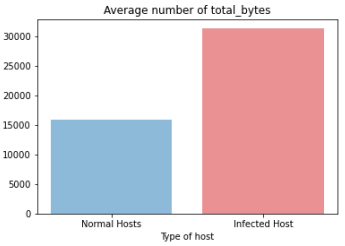
(c) Average time difference (in ms) between two consecutive flows of a host



(d) Average number of bytes received



(e) Average number of bytes transmitted

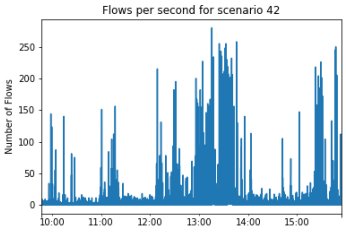


(f) Average number of bytes exchanged

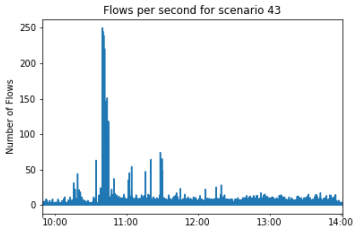
Figure 5.2: Visualization of differences in the behaviour between the infected and the normal host for the numerical features: duration, time difference, packets, source bytes, destination bytes and total bytes.

5.1.5 Number of flows per host

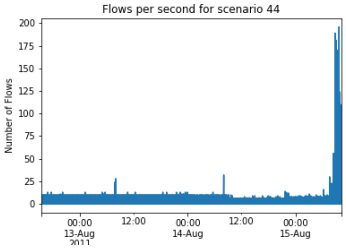
An important parameter in every machine learning model is the amount of data. More specifically, in our project it is important to have many data for each host we want to classify, in order to capture its behavior and either use it to build a model or to evaluate it. figure 5.5 depicts the distribution of the number of flows for the benign and the malicious hosts of the whole dataset. As it can be easily noticed, most of the normal hosts have a small number of flows while the malicious hosts usually have an adequate number of flows. This means that there is a high probability that some normal hosts will be missclassified as malicious because there would be not enough data to make a correct judgment.



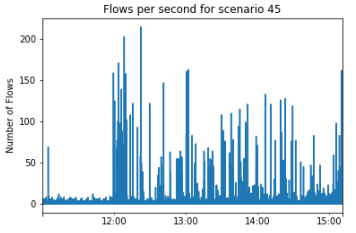
(a) number of flows per second for scenario 1



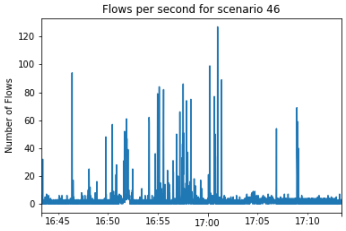
(b) number of flows per second for scenario 2



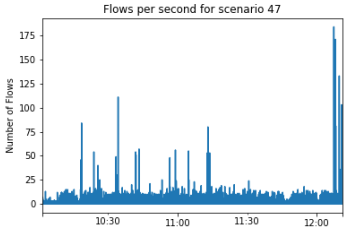
(c) number of flows per second for scenario 3



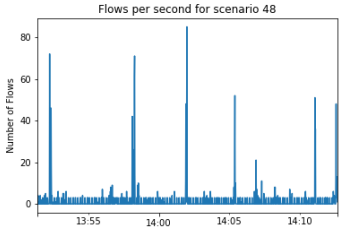
(d) number of flows per second for scenario 4



(e) number of flows per second for scenario 5

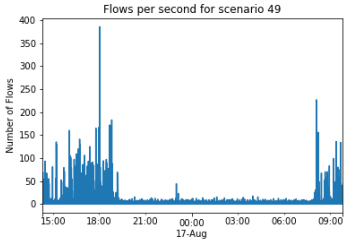


(f) number of flows per second for scenario 6

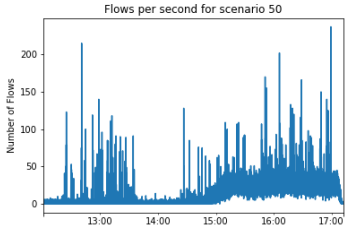


(g) number of flows per second for scenario 7

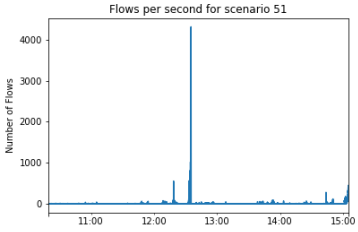
Figure 5.3: Visualization of the number of flows per second for scenarios 1-7



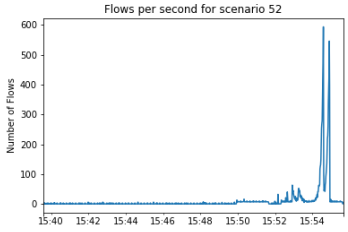
(a) number of flows per second for scenario 8



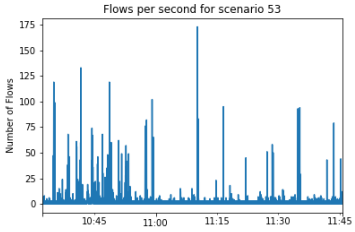
(b) number of flows per second for scenario 9



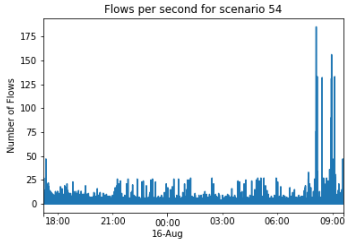
(c) number of flows per second for scenario 10



(d) number of flows per second for scenario 11

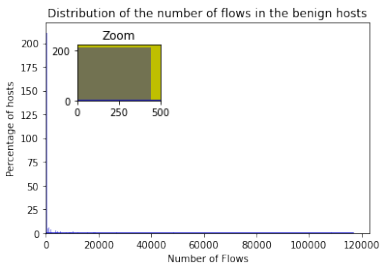


(e) number of flows per second for scenario 12

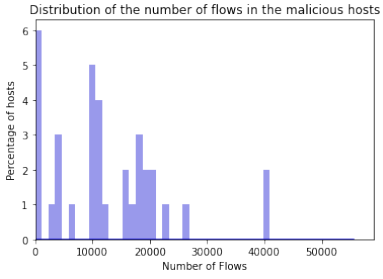


(f) number of flows per second for scenario 13

Figure 5.4: Visualization of the number of flows per second for scenarios 8-13



(a) Benign hosts



(b) Malicious hosts

Figure 5.5: Distribution of the number of flows for the benign and the malicious hosts

5.2 PAUTOMAC competition dataset

Another dataset used in this project is the one from the PAUTOMAC competition[21]. The description of this competition and the reason for using this data are described in section 6.4.

Initially there were two types of data used to build the models and evaluate them, namely : artificially generated and real-world data. However, because there was not an unbiased way to evaluate the results from the real-world data, they finally used only the artificial data.

5.2.1 Artificial data generation

The generation of the artificial data began with the building of a random probabilistic automaton with 5-75 states and an alphabet of 4-24 symbols. Then a number of state-symbol pairs (between 20% and 80% of all possible) was generated from the aforementioned automaton. More specifically, they started by choosing an initial state at random and then they chose a symbol from the symbol set not been previously selected. Following, one transition was generated from from every state-symbol pair to to a randomly selected targets, and a percentage between 0% and 20% of all the possible transitions were selected without replacement.

From each random probabilistic automaton, one train and one test set was constructed. The train sets size was 100000 with probability 25%, otherwise it was 20000. Regarding the testing set, it contained 1000 unique strings. On both the training and testing sets, the length of each string was between 5 and 50 symbols. Finally 16 such machines and couples of training and testing dataset were produced. Almost the same procedure followed also for 16 Hidden Markov Models (HMM) and Probabilistic Deterministic Finite Automata (PDFA), resulting in 48 pairs of training and testing sets. A more detailed description of the generation of the artificial data can be found on [64].

5.2.2 Data Description

Tables 5.7 and 5.8 presents some statistics about number of traces, the number of symbols and the average length of the traces for each of the 48 artificial training and testing sets of PAUTOMAC competitions, accordingly. The number of traces in the training sets was expected because as mentioned above, its size was 100000 with probability 25%, otherwise it was 20000. The number of traces in the testing sets was fixed (1000) as expected. Regarding the number of symbols, it was expected that the training and testings sets for each scenario will have exactly the same number of symbols. Finally, it can be easily noticed that the length of the traces in the testing sets is in average significantly bigger than the one of the training sets.

	mean	**std**	**median**	**min**	**max**
# Traces	41666.667	35551.215	20000	20000	100000
# Symbols	11.625	5.943	10	4	23
Trace length	10.707	5.353	8.089	4.939	26.669

Table 5.7: Statistics for the number number of traces, the number of symbols and the average length of the traces for all the artificial training sets of PAUTOMAC competition

	mean	**std**	**median**	**min**	**max**
# Traces	1000	0	1000	1000	1000
# Symbols	11.625	5.943	10	4	23
Trace length	14.691	5.238	13.421	8.323	29.1

Table 5.8: Statistics for the number number of traces, the number of symbols and the average length of the traces for all the artificial testing sets of PAUTOMAC competition

6

Methodology

This chapter contains the description of the steps followed in order to classify a malicious host. Firstly, the discretization and encoding the flows of the dataset is presented, along with the procedure followed to extract the traces needed to build the state machine. Then the three methods for recognizing a host as infected are discussed. Following, the two types of simulations, namely single-scenario and multi-scenario, used to evaluate the different models on the task of recognizing malicious hosts are described. Finally the procedure followed to measure the runtime efficiency of the proposed heuristic is described.

6.1 Data

As it was referred in the previous chapters, the dataset used in this project, namely CTU 13, consists of NetFlow data. Therefore, every scenario of this dataset contains a specific number of flows with some attributes.

An initial step for each scenario, was the removal of the background flows. As said in previous chapter, the background traffic is real traffic from other participants in the network. In our case we want to classify a host as malicious or benign so we decided to remove these flows because they do not match to our use-case.

Also, we have to add manually a new flow attribute, which derives from the existent attributes. This feature was the amount of data sent. Since we have the total amount of data exchanged for each flow and amount of data received for the source IP of this flow record, we extracted this feature by subtracting these two values.

6.1.1 Data split

For each scenario we divided the data in three disjoint datasets, following the strategy used in [15]:

- Configuration dataset: We randomly selected at least 30% of the benign flows of the examined scenario. More specifically, all the flows of a host are collected until the number of flows reach or pass the 30% of the benign flows of the examined scenario. This dataset was used to calculate the percentiles for the discretization features and for the tuning of the evaluation methods.

- Training dataset: We randomly selected all the flows from one malicious host of the examined scenario. These flows were used to build the State Machine model
- Evaluation Dataset: It consists of all the remaining flows from both malicious and benign hosts, and it was used to evaluated the derived model.

From the three datasets described above, it is noticeable that is possible that the evaluation dataset can be bigger that the training dataset. This does not agree with the common way the dataset is divided in machine learning tasks, where the biggest part of the dataset is used for training purposed. However, in this case the flows from one host are enough to infer the model so there is no need to used more flows.

6.1.2 Data Discretization

Initially, each feature of the flow should be discretized. Discretization is a dimensionality technique which diminishes data from a large domain of numerical values to a small domain of categorical values. For the categorical features this process is simple as the discretization is done by assigning a progressive non-negative number to every possible value of the corresponding feature. For example, for the protocol type feature, we assign 0 if the protocol is UDP, 1 if it is UDP, 2 if it is ICMP, etc.

When it comes to the numerical features, the discretization was accomplished by using percentiles to cluster them. In order to decide the optimal number of clusters, the ELBOW method was used [65]. In order to apply the ELBOW method we start by applying the K-means clustering for a specific range of values for k, which is the number of expected clusters. In our case, the k parameter was between 1 and 10, inclusive. For each value of the k parameters we calculate and plot the inertia value. Inertia is the the sum of squared distances of samples to their closest cluster center. An sample output for this visualization is depicted in figure 6.1. The optimal number of clusters is determined by choosing the value of k in the plot, after which the inertia value start decreasing in a linear fashion. In figure 6.1, we can conclude that the optimal number of clusters is 3. This point of the plot has the shape of an "elbow", hence the name of the method.

After replacing each feature value by the cluster it belongs, a progressive non-negative number is assigned to every value (cluster). For example, if for some feature the ELBOW method suggests that there are 4 clusters then we assign 0 if the value is before the 25th percentile, 1 if the value is between the 25th and 50th percentile, 2 if the value is between the 50th and 75th percentile, and 3 if the value is after the 75th percentile.

6.1.3 NetFlow Encoding

In order to convert the Netflow data, where each flow consists of several features, to a symbolic sequence which can be used as a input for State machine, we should encode them. That means that every flow from the NetFlow data should be converted to a discrete symbol. In order to do so, we follow the Algorithm 7 [15], where M_i denotes the attribute mapping for feature i as has been computed with the aforementioned technique and $|M_i|$ the number of values for feature f_i.

An example that demonstrates the process followed in alg. 7 is the following. Let's assume that each NetFlow has only 2 attributes: protocol and bytes. Regarding, the protocol let assume that the only values observed are TCP and UDP. The mapping described

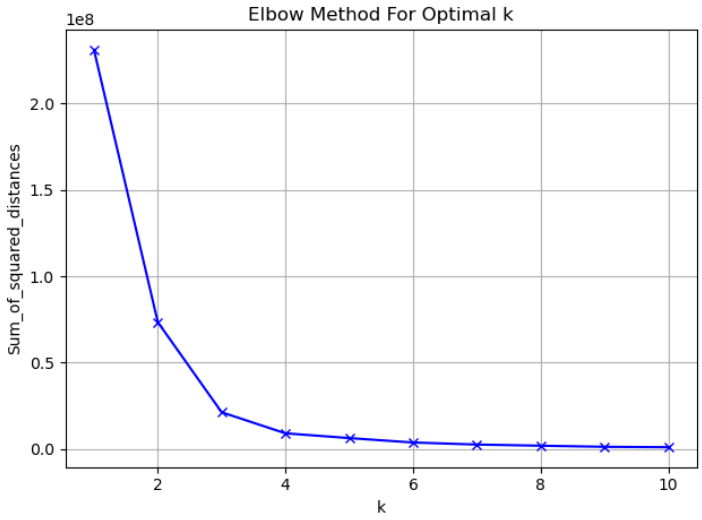


Figure 6.1: Sample visualisation of ELBOW method

above will assign 0 to the former and 1 to the latter. When it comes to the bytes attribute, let assume that the values observed are: $\{20, 35, 35, 40, 90\}$ and assume we are interested in the 20th and 60th percentiles. The first step is to find out the ordinal ranks of these values in the list of the observed values, using the formula: $r(p) = \left\lceil \frac{p}{100} \times N \right\rceil$, where p is the required percentile, and N is the collection size. Therefore $r(20) = \left\lceil \frac{20}{100} \times 5 \right\rceil = 1$ and $r(60) = \left\lceil \frac{60}{100} \times 5 \right\rceil = 3$. Percentiles are the values of the observed list that corresponds to the ordinal ranks. Therefore, the 20th percentile is 20 and the 60th percentile is 35. Thus, the mapping for the bytes value is $M_{bytes}(v) = \{0 if\, v \leq 20, 1 if 1 < v \leq 35, 2 \quad else\}$. where v is the value of the bytes attribute for the examined NetFlow. Given the mappings for the protocol and bytes features, the symbol for the $\langle UDP, 25 \rangle$ is: $1 \times \frac{6}{2} + 1 \times \frac{3}{3} = 4$ while the one for the instance $\langle TCP, 100 \rangle$ is $1 \times \frac{6}{2} + 2 \times \frac{3}{3} = 5$.

6.1.4 Sliding Windows

State machines are sequential models, thus their input should be a sequence of symbols. This means that after we have discretized all the features and converted each flow to a discrete symbol, we should process them to obtain sequences of events. Theses sequences were obtained by applying the sliding window technique.

Before describing the sliding window technique utilized, it should be mentioned that since we are modeling the behavior on host level, before applying the sliding window technique we grouped the NetFlows based on their source IP.

Firstly, as mentioned above, we group the Netflows of the examined scenario based on their source IP. Then, we should apply a window to obtain sequential events. This window can be either a time window, which means that all NetFlows with a time range

Algorithm 7: Netflow encoding

input : a Netflow $n = \langle f_0, f_1, \ldots, f_k \rangle$ with k features
input : an attribute mapping M_i for $i = 0, 1, \ldots, k$
output : an integer code for the NetFlow n
code $\longleftarrow 0$;
spaceSize $\longleftarrow \prod_{i=0}^{k} |M_i|$;
for $i \longleftarrow 0$ **to** k **do**
    code $\longleftarrow$ code $+ M_i(f_i) \times \frac{spaceSize}{|M_i|}$;
    spaceSize $\longleftarrow \frac{spaceSize}{|M_i|}$;
end
return code;

will be assigned to the same sequence, or a fixed length window which means that all sequences will contain a fixed number of windows. On the former case the start of the next window is incremented by a specific time period smaller than the window range, while on the latter case the start of the next window is incremented by a specific number of flows, again smaller than the length of the window. This difference between the beginnings of successive windows is called stride.

Our first attempt was to apply a time window of fixed time. However, the traffic volume didn't have a relatively stable ratio, meaning for a small number of time periods there was a great volume of NetFlows while for the remaining time the traffic was significantly smaller. These "spikes" on the traffic volume, led to obtaining many sequences with few symbols (NetFlows) and few sequences with many NetFlows. As it can be understood, there traces are not suitable for learning a state machine.

In order to solve this issue, we tried to customize the time range of the window based on the dataset I had. Especially, the median value of the time difference between successive flows was extracted from the configuration dataset. Then the time range of the windows was set to $n^*median$ value, so as every window to contain approximately n symbols (flows). Although the number of flows was more balanced than before, they keep being inappropriate for learning a state machine model.

Finally, the approach that was finally chosen is the one with windows of fixed length. The window size chosen was 20 flows with a stride size of 10 flows. The length of 20 was chosen because we believe that this length is big enough to capture the behavior of the host. Also this helps to produce "shorter" state machines, thus easier to learn. By applying this technique we obtained sequences of equal length which is a much more suitable format than the aforementioned for learning a state machine. The disadvantage of this method is that it may contain in the same sequence flows that are whose time difference is very much, therefore they are not related.

6.2 Recognizing a Host as Infected

6.2.1 Acceptance ratio-based

This method of recognizing a host as infected is based on the acceptance ratio of the windows extracted for a host and the communication profile build, as described in [14]. In particular after building the communication profile for the infected host(s), we measure the acceptance ratio of each examined host based on its windows. The acceptance ratio is given by:

$$acceptance\_ratio = \frac{\#\ accepted\ windows}{\#\ rejected\ windows} \tag{6.1}$$

Given that the communication profile is capturing the behavior of a malicious host, a host is classified as malicious if the acceptance ratio value after t windows is bigger than a threshold t. A preliminary analysis showed that the number of flows n should be 25. Regarding the classification threshold t, it was decided by measuring the maximum acceptance ratio of the hosts in the configuration dataset after 25 flows has been examined, and obtaining their average value. So this value differs according to the examined scenario.

6.2.2 Error-based

Symptoms are the states of the state machine models. The error-based method, inspired by [15], for recognizing a host as infected is based on the counts of the symptoms on the modeled host and on the candidate host. Let's assume that C in the candidate host and M is the malicious modeled host. For each candidate host we measure the error with the following formula:

$$S = \frac{\sum_i |Error_i^M - Error_i^M|}{\sum_i (Error_i^M + Error_i^M)} \tag{6.2}$$

where $Error_i^M$ and $Error_i^M$ is the error on each symptom (state) i for the candidate and malicious host respectively. Given that the communication profile is capturing the behavior of a malicious host, a candidate host is classified as malicious if the error is below a threshold t. This threshold is decided measuring the same error for the hosts of the configuration dataset, and it differs per scenario.

6.2.3 Fingerprint-based

The fingerprint-based method [15] is based on the discrimination of the symptoms that happen when a host is malicious but never when it is a benign one. In order, to measure these occurrences of the specific symptom, we use again the configuration dataset to find the symptoms that never occurs in the benign hosts. We also use the modeled host to investigate which of these symptoms happen in the malicious hosts. Finally, when examining the candidate host, we classify it as malicious if there exists a symptom i such that $Counts_i^B = 0$, $Counts_i^M > 0$, and $Counts_i^C > 0$, where B is the benign host. Otherwise, the candidate host is classified as benign.

6.3 Single vs. Multi scenario simulation

In order to evaluate the classification performance of our state machine inferring technique, we experimented with two different types of experiments, inspired by [20]. The structure of these experiments is discussed below.

6.3.1 Single scenario simulation

The simpler one is the single scenario simulation. The goal of this experiment is to evaluate our model on detecting known threats. For example, a state machine is inferred based on NetFlow records from a host performing DDOS attacks, and then its performance is evaluated on the remaining hosts, among which there are other hosts performing DDOS attacks. The procedure followed in the single scenario simulation is the following:

For each scenario:

- Spit the dataset in configuration, training and evaluation sets.
- For each dataset, group the NetFlows by their source IP. Then apply data discretization, flow encoding and sliding windows to obtain the windows for each host.
- Build a communication profile using the malicious host on the training dataset.
- Use the configuration dataset to tune the evaluation method described in section 6.2.
- Evaluate the performance of the communication profile on the evaluation dataset.

The whole procedure is presented in figure 6.2. It is important to mention that the scenarios used on this simulation were the scenarios 9, 10, 11 and 12, as they were the only scenarios containing more than one malicious hosts. This was necessary as we need one malicious host to build the communication profile, but we also need other malicious hosts to evaluate its performance. Therefore, there is no meaning on using the other scenarios, we would end up evaluating the classification performance of the model only on benign hosts.

6.3.2 Multi scenario simulation

The multi scenario simulation is a much more challenging experiment, because the infection models built have to detect unseen threats. More specifically, the following procedure is being followed:

- Scenarios 3, 4, 5, 7, 10, 11, 12, and 13 are used for as training sets, with a part of them being used as configuration dataset. With the procedure described in section 6.1.1, at least 30% of each of these scenarios are is used as configuration dataset. On the other hand, scenarios 1, 2, 6, 8, and 9 are used as evaluation datasets.
- For each dataset and scenario, group the NetFlows by their source IP. Then apply data discretization, flow encoding and sliding windows to obtain the traces for each host.
- Build a communication profile for every malicious host on the training dataset

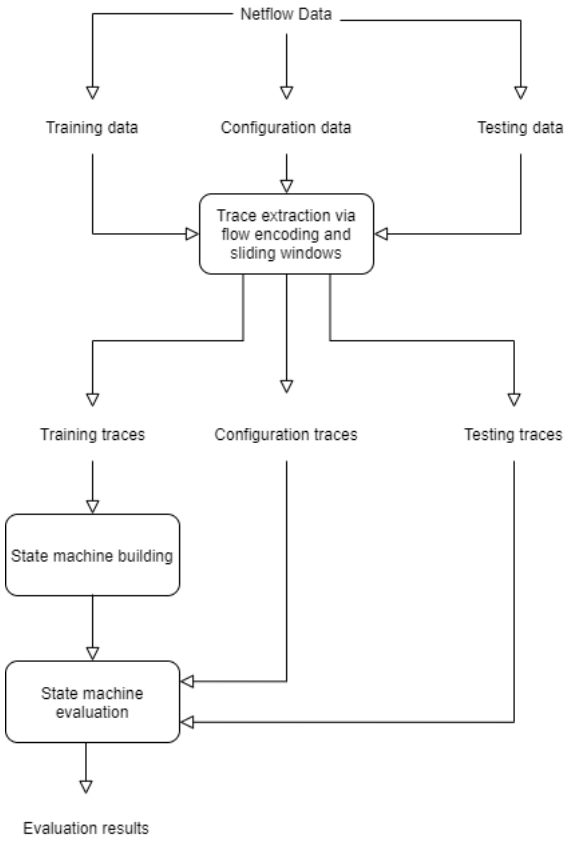


Figure 6.2: Pipeline of the single-scenario simulation

- Use the configuration dataset to tune the evaluation methods described in section 6.2.

- Evaluate the performance of the communication profile on the evaluation datasets. In this simulation we have multiple models for the malicious hosts so consider a candidate host as malicious if any of the communication profiles built recognizes it as one.

As explained before, the separation of the scenarios into training and evaluation set was done in a way that will ensure that the two datasets will include different types of bots. As can be observed in chapter 5, the scenarios of the training dataset contain Rbot, Virut, Sogou and NSIS.ay bots, while the scenarios of the evaluation dataset contain Neris, Menti and Murlo bots.

6.4 Measuring runtime and model quality

In order to measure the performance of our proposed state-merging heuristic in terms of runtime, we used the dataset provided by the PAUTOMAC probabilistic automaton learning competition [21]. The decision to use this dataset instead of the CTU-13 dataset, was done because the number of windows extracted for each host of the CTU-13 dataset was not big enough, so the models were built very fast. As a result, measuring the time efficiency of our method there would not provide reliable results. Therefore we measured the runtime of flexfringe for building the model when using our heuristic and compared it with the runtime of flexfringe for building the model when using the modified alergia heuristic.

The PAUTOMAC probabilistic automaton learning competition is an on-line challenge that took place in 2012. As described in [64], "the goal of PAUTOMAC was to provide an overview of which probabilistic automaton learning techniques work best in which setting and to stimulate the development of new techniques for learning distributions over strings". As described in section 5.2, the data were provided in the form of sequences of symbols for this competition, so there was no need to follow the procedure followed for the CTU-13 dataset.

Except from the runtime and the number of states that we measure for every test set provided for this competition, we decided to measure also the efficiency of the our model on learning the distribution over the given strings. Therefore, we calculated the score used on this competition and compared with the one achieved with the modified version of the alergia heuristic.

The evaluation measure use on the PAUTOMAC was based on the perplexity score. Specifically, for a test set S the perplexity score achieved is calculated by the following formula:

$$Score(C, S) = 2^{-\sum_{x \in S} P_T(x) * \log_2(P_C(x)} \tag{6.3}$$

where $P_T(x)$ is the normalized probability of the trace x in the target model and $P_C(x)$ is the normalized candidate probability for the trace x of the examined model.

7

Experiments

This chapter contains the description of the experiments conducted in order to answer the second and third research questions of thesis. Initially, we describe the experimental procedure and the hyper-parameter tuning for the models used in the experiments. Then we present and discuss the results of the different models in the single-scenario and multi-scenario simulation of detecting malicious botnets hosts on the CTU-13 dataset. We compare the performance of the methods using the proposed state-merging heuristic with other models and comment about the performance about of the different LSH families. Following we compare the performance of our method on the multi-scenario simulation with the one achieved by a state-of-the-art work. Finally, the runtime and performance of out proposed heuristic on building state machines is presents and its results are compared with the results of the modified alergia heuristic.

7.1 Experimental Configuration

The core of this project, which is the *flexfringe* tool modified by adding the new state merging method with the usage of LSH to compare the future traces of each state, was implemented in C++. However, all the other parts of the experimental procedure were implemented in Python 3.8. Especially, Python 3.8 was used for the data discretization and flow encoding, as well as the conduction of the experiments and their evaluation, including the visualization of the results.

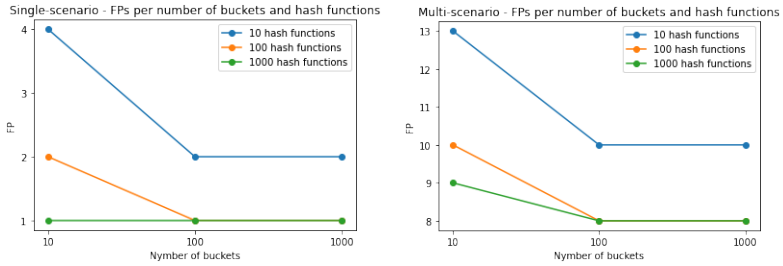
The experiments were sun on a local machine with 8GB RAM and Intel Core i5 8550U at 1.8 GHz.

7.2 Hyper-parameter Tuning

The most important hyperparameters need to be tuned for the LSH concept are the number of bins and the number of hash functions used. When the number of bins is small, there is a high probability that some non-similar vectors will end up on the same bucket. On the other hand, if the number of buckets is very big we will end up assigning on the same bucket only same vectors. This will have no difference than a simple counter of different vectors, thus it will be computationally expensive and it will require much memory.

Regarding the number of hash functions used, we can say that with more hash function we will have more stable results. This happens because, a possible error being done by one hash functions who assigned one vector to the ”wrong” bin, will not be done by all the other hash functions. So the distribution of future traces we extract by combining the buckets of each hash function will be more accurate. However, as for the number of buckets, the more increase on the number of hash functions entails increase in the computational cost of the LSH structure.

In order to tune these hyperparameters, we experimented with different values for the number of buckets and the number of hash functions. The results from the hyperparameter tuning for the error-based evaluation method and the 1-stable distribution LSH family are presented in figure 7.1.Figure 7.1a contains the results for the single scenario simulation, while figure 7.1b contains the results for the multi-scenario simulation. Given that there was no undetected malicious host (false negative), we measure the number of false positives for each configuration. It should be mentioned that for both simulation we count the number of false positives for all the scenarios aggregated and not individually. As we can observe, the behavior for both the simulations is the same. More specifically, when using 10 hash functions or 10 buckets the number of false positives reaches its maximum value in comparison with the other configurations. When increasing either of them, the number of false positives decreases with the minimum value for both cased reached for 100 buckets and 100 hash functions. Finally, when we increase either of these two hyperparameters, we observe that the number of false positives does not decrease. Thus there is no reason to increase the computational cost and the runtime of the system given that there is no improvement in the results. The behavior of the other two LSH families (2-stable distribution and random hyperplanes) is similar with the one described above. Therefore, for the experiments presented in the following sections we used 100 hash functions and 100 buckets.



(a) False positive predictions for single-scenario simulation and various values for the number of buckets and hash functions

(b) False positive predictions for single-scenario simulation and various values for the number of buckets and hash functions

Figure 7.1: False positive predictions for single-scenario and multi-scenario simulations and various values for the number of buckets and hash functions

Another important parameter is the threshold used when comparing the KL divergence of two distributions to decide if a merge is consistent. This parameter significantly affects the states machines built, because the higher the threshold is, the more merges will be allowed. Thus, a model with less states will be built. After some preliminary analysis

with the value of this threshold, its value was set to 0.01.

7.3 Experimental procedure

The first part of the experiments is about the efficiency of our proposed method on detecting the malicious hosts of the CTU-13 dataset. As was described in chapter 6, we evaluate our method for two types of simulation, namely single scenario and multi scenario. In addition, we compare the performance of our best model in the host classification task with the one of a state-of-the art model using the same configuration about the separation for the datasets into test and training dataset. Initially, the dataset is separated into training, configuration and testing datasets according to the type of simulation used. We discretized and encoded each flow before we group the flows in host level. Then, the sliding window method was used to extract sequential traces for each host. Afterwards, we build the the models using the training datasets, we tuned the evaluation methods by using the configuration dataset, and finally evaluated the models on the testing dataset. In order to build the model using our method, we used *flexfringe* with the custom state-merging heuristic and experimented with three different LSH families, namely random hyperplanes, 1-stable distribution and 2-stable distribution. Furthermore, we compared the aforementioned models with a trigram baseline model and a models built with *flexfringe* using the modified version of alergia as a state-merging heuristic. When reporting the results, we use the following abbreviations:

- **3gram**: The trigram model.
- **ALERGIA**: The state machine model built with the modified alergia state-merging heuristic.
- **RH**: The state machine model built with the proposed state-merging heuristic using the random hyperplanes LSH family.
- **1-SD**: The state machine model built with the proposed state-merging heuristic using the 1-stable distribution LSH family.
- **2-SD**: The state machine model built with the proposed state-merging heuristic using the 2-stable distribution LSH family.

The second part of the experiments, is mainly about measuring the runtime of our proposed state-merging heuristic. In particular we used the data provided by the PAUTOMAC competition and measured the runtime, the number of states of the created model and the perplexity score of the model. We used the LSH family that provided the best results in the classification task to build the model with our proposed state-merging heuristic, and we compared its performance with a model built using the modified version of alergia as a state-merging heuristic.

7.4 Single Scenario

On this simulation, we build a model from a malicious host of each scenario and evaluate its performance on classifying the other hosts of the same scenario. As was explained in

section 6.3.1, we are using only the scenarios 9,10,11 and 12 as they are the only ones that contain more than one malicious host. Also, we have used three different methods that, given the malicious host model and the traces for the examined host, decides whether a host is malicious or not. These are the error-based, fingerprint-based and the acceptance ratio-based. Since we are using a model built from a specific type of botnet to detect other botnets of the same type, we expect that we will be able to recognize the most of the malicious hosts successfully.

The results for the error-based evaluation method can be seen in table 7.1. It can be easily observed that the trigram model detects all the malicious hosts for all the four scenarios while having very few false positives, one for scenario 9 and 12. On the other hand the state machine build with the alergia state-merging heuristic, although raising few false alarms, is not so sensitive as it fails to recognize two malicious hosts for scenario 9 and one for scenario 11. When it comes to the state machine built using our proposed state-merging heuristic, for all the three LSH families it detects all the botnets of the three first scenarios (9, 10 and 11) with few positive alarms. Regarding scenario 12, LSH with 1-stable distribution has no false positives while the LSH with random hyperplanes and 2-stable distribution raise one false alarm each.

In general, we can say that the best performance is achieved by the trigram baseline method and state machine with our proposed state-merging heuristic and 1-stable distribution LSH family. Both these two models, had only one false prediction, in scenario 9 where they raised a false alarm. Also it is worth mentioning that the error-based evaluation method seems to be an effective method for deciding if a host is malicious, as for the majority of scenarios and models almost all the predictions are correct.

Table 7.2 shows the results of the fingerprint-based evaluation method. As it can be noticed there, for scenario 9 all methods classify correctly all the malicious hosts, but they are raising a few false alarms. For scenario 10, all methods except for the state machine with our state merging heuristic using LSH with 1-stable distribution, do not successfully recognize all the botnet hosts. As for scenario 11, all methods achieve perfect score except for the trigram model the state machine with our state merging heuristic using LSH with random hyperplane, which fail to recognize on botnet host. Finally, in scenario 12 all only the state machine with alergia heuristic and the one using LSH with random hyperplanes fails to recognize all the malicious hosts.

Overall, again the trigram model and the the state machine with our state merging heuristic using LSH with 1-stable distribution or 2-stable distribution, achieves the best results. However, it is clear that the overall results achieved with the the fingerprint-based evaluation method are worse than the ones of the fingerprint-based evaluation method.

The results for the acceptance ratio-based evaluation method can be seen in table 7.3. It is clear that the results for this evaluation method are much worse that the previous ones for all the different models. In particular, in scenarios 9 and 12 all methods fail to recognize all the botnet hosts. On the other scenario, we have some models with 100% while some other fail to recognize many malicious hosts.

We can conclude that this evaluation method is quite unstable, as in half the scenarios it misclassifies almost all the malicious hosts as benign. Furthermore, it has a clear tendency towards the negative class, thus we have few false positives but also many undetected malicious hosts. In comparison with the other two evaluation methods, we can conclude

	Model	TP	FP	TN	FN
Scenario 9	3gram	9	1	22	0
	ALERGIA	7	1	22	2
	RH	9	2	21	0
	1-SD	9	1	22	0
	2-SD	9	1	22	0
Scenario 10	3gram	9	0	15	0
	ALERGIA	9	1	14	0
	RH	9	0	15	0
	1-SD	9	0	15	0
	2-SD	9	1	14	0
Scenario 11	3gram	2	0	12	0
	ALERGIA	1	0	21	1
	RH	2	0	12	0
	1-SD	2	0	12	0
	2-SD	2	0	12	0
Scenario 12	3gram	2	0	17	0
	ALERGIA	2	1	16	0
	RH	2	1	16	0
	1-SD	2	0	17	0
	2-SD	2	1	16	0

Table 7.1: Evaluation results for all scenarios and models, using the error-based evaluation method

that this is the one providing the worst results.

All in all, for the single scenario simulation we can reach to the conclusion that the error-based evaluation method has provided the best results, meaning that in for most of the scenarios and models, the accuracy was almost perfect. Also, it provided the more stable results, as the performance of each method didn't differ significantly per scenario. Regarding the models evaluated, the trigram model method and the state machine with the LSH with 1-stable distribution state-merging heuristic have achieved the best results in the majority of the cases. Especially, their performance was almost perfect as they had only one false prediction when the misclassified a benign host. It is impressive that our state-merging heuristic using LSH with 1-stable distribution has better performance than the alergia heuristic given that it makes many approximations when during the state-merging process, so we expected an from alergia to have same or better performance. This may be caused by the fact that the behavior of the hosts in CTU-13 is not so complicated, so our method achieves better results by inferring "simpler" models. In the single-scenario simulation, we expected that the results will be almost perfect because the models are trained on the same type of botnet with the ones they are trying to detect.

	Model	TP	FP	TN	FN
Scenario 9	3gram	9	3	20	0
	ALERGIA	9	3	20	0
	RH	9	5	18	0
	1-SD	9	3	20	0
	2-SD	9	3	20	0
Scenario 10	3gram	8	2	13	1
	ALERGIA	8	0	15	1
	RH	8	2	13	1
	1-SD	9	2	13	0
	2-SD	8	0	13	1
Scenario 11	3gram	1	0	12	1
	ALERGIA	1	0	12	0
	RH	1	0	12	1
	1-SD	2	0	12	0
	2-SD	2	0	12	0
Scenario 12	3gram	2	1	16	0
	ALERGIA	0	1	16	2
	RH	1	1	16	1
	1-SD	2	1	16	0
	2-SD	2	1	16	0

Table 7.2: Evaluation results for all scenarios and models, using the fingerprinting-based evaluation method

	Model	TP	FP	TN	FN
Scenario 9	3gram	0	0	23	9
	ALERGIA	0	1	22	9
	RH	0	0	23	9
	1-SD	0	0	23	9
	2-SD	0	0	23	9
Scenario 10	3gram	9	0	15	0
	ALERGIA	9	0	15	0
	RH	3	0	15	6
	1-SD	6	0	15	3
	2-SD	8	0	15	1
Scenario 11	3gram	1	0	12	1
	ALERGIA	1	0	12	1
	RH	1	0	12	1
	1-SD	2	0	12	0
	2-SD	2	0	12	0
Scenario 12	3gram	0	1	16	2
	ALERGIA	0	0	17	2
	RH	0	0	17	1
	1-SD	0	0	16	2
	2-SD	0	1	16	2

Table 7.3: Evaluation results for all scenarios and models, using the acceptance ratio-based evaluation method

7.5 Multi Scenario

On this simulation, we build a model for every malicious host in scenarios 3, 4, 5, 7, 10, 11, 12 and 13, and evaluate its performance of the examined method on classifying the hosts of scenarios 1, 2, 6, 8 and 9. If any of the models marks a host as malicious, then it is classified as malicious. As was explained in section 6.3.2, we split the scenarios on the aforementioned way, so as the scenarios in the training set contain different botnet families than the evaluation set. Since we use different botnets families to build the models than the botnet families we are trying to detect, we expect that the classification results will be much more worse than the ones of the single scenario simulation. Our goal is to build a system that will be able to detect all the botnets, while it minimizes the number of false alarms. Also, we have used the three evaluation methods that we used on the single scenario simulation.

Starting from the result of the error-based evaluation method presented in table 7.4, we can observe that in all the scenarios and all the methods, there are no undetected malicious hosts but there are many false positives. This behavior was expected because our detection system focus more on the sensitivity than the reliability, because for a host to be classified as benign it should be classified as benign by all of the models. Also, there is no significant difference in the performance of the different models. However, we can observe that for all the scenarios, the performance of the state machine using LSH with 1-stable distribution as a state-merging heuristic has better or at least equal performance with the other models. Thus, by taking into account all the scenarios, this is the best performing model.

Regarding the results of the fingerprint-based evaluation method presented in table 7.5, is can be easily noticed that as in the error-based evaluation method discussed above, all the botnet hosts are recognized. However, in that case the number of false alarms raised is much bigger. This was an expected behavior because of the fact that the fingerprinting evaluation method tends to be more sensitive on the way it classifies a host as malicious.

Finally, the results of the acceptance ratio based evaluation method as presented in table 7.6. We can see that contrary to the two other evaluation methods, this ones fails to detect some of the malicious hosts. When it comes to the performance of the different models, we can observe that for all the scenarios, the performance of the state machine using LSH with 1-stable distribution as a state-merging heuristic has better or at least equal performance with the other models.

All in all, we can conclude that the error-based has provided the best results for almost all the models, with no undetected malicious host and a few false alarms. The fingerprinting evaluation method produced more false positives as expected, while the acceptance rate one failed to recognize many of the botnets. Regarding the different models used to build the communication profile of the hosts, the state machines using LSH as a state-merging heuristic has provided equally good or even better classification results, which means that the approximations made by our method when merging two states are accurate. Especially the one using the 1-stable distribution LSH family, has the best overall results in all the scenarios.

In the multi-scenario simulation, we expected that the number of false positives would be higher that the ones in the single-scenario simulation because the models are trained in different types of botnets than the botnets they are trying to recognize. The fact that they succeed to recognize all the malicious hosts, without few false alarms is very impressive.

	Model	TP	FP	TN	FN
Scenario 1	3gram	1	5	14	0
	ALERGIA	1	2	17	0
	RH	1	5	14	0
	1-SD	1	2	17	0
	2-SD	1	4	15	0
Scenario 2	3gram	1	0	14	0
	ALERGIA	1	0	14	0
	RH	1	1	13	0
	1-SD	1	0	14	0
	2-SD	1	0	14	0
Scenario 6	3gram	1	2	14	0
	ALERGIA	1	3	13	0
	RH	1	2	14	0
	1-SD	1	1	15	0
	2-SD	1	1	15	0
Scenario 8	3gram	1	2	21	0
	ALERGIA	1	4	19	0
	RH	1	2	21	0
	1-SD	1	2	21	0
	2-SD	1	3	20	0
Scenario 9	3gram	10	3	22	0
	ALERGIA	10	5	20	0
	RH	10	5	20	0
	1-SD	10	3	22	0
	2-SD	10	3	22	0

Table 7.4: Evaluation results for all scenarios and models, using the error-based evaluation method

	Model	TP	FP	TN	FN
Scenario 1	3gram	1	6	13	0
	ALERGIA	1	7	12	0
	RH	1	8	11	0
	1-SD	1	4	15	0
	2-SD	1	5	14	0
Scenario 2	3gram	1	3	11	0
	ALERGIA	1	1	13	0
	RH	1	5	9	0
	1-SD	1	1	13	0
	2-SD	1	2	12	0
Scenario 6	3gram	1	4	12	0
	ALERGIA	1	3	13	0
	RH	1	3	13	0
	1-SD	1	2	14	0
	2-SD	1	3	13	0
Scenario 8	3gram	1	6	17	0
	ALERGIA	1	7	16	0
	RH	1	8	15	0
	1-SD	1	4	19	0
	2-SD	1	6	17	0
Scenario 9	3gram	10	5	20	0
	ALERGIA	10	6	19	0
	RH	10	6	19	0
	1-SD	10	5	20	0
	2-SD	10	6	19	0

Table 7.5: Evaluation results for all scenarios and models, using the fingerprinting-based evaluation method

	Model	TP	FP	TN	FN
Scenario 1	3gram	1	5	14	0
	ALERGIA	1	6	13	0
	RH	0	1	18	1
	1-SD	1	4	15	0
	2-SD	0	2	17	1
Scenario 2	3gram	1	0	14	0
	ALERGIA	1	0	14	0
	RH	1	1	13	1
	1-SD	1	1	13	0
	2-SD	1	1	13	0
Scenario 6	3gram	1	2	14	0
	ALERGIA	1	2	14	0
	RH	1	2	14	0
	1-SD	1	1	15	0
	2-SD	1	1	15	0
Scenario 8	3gram	1	4	19	0
	ALERGIA	1	5	18	0
	RH	1	5	18	0
	1-SD	1	4	19	0
	2-SD	1	4	19	0
Scenario 9	3gram	0	2	23	10
	ALERGIA	0	2	23	10
	RH	0	3	22	10
	1-SD	0	2	23	10
	2-SD	1	3	22	9

Table 7.6: Evaluation results for all scenarios and models, using the acceptance ratio-based evaluation method

7.6 Comparison to state-of-the-art detection techniques

As discussed in chapter 3, A. Blaise et al. in [13] built a system called BotFP (Bot Finger-Printing) a clustering technique to detect malicious hosts on CTU 13 dataset. They extract a vector representation of each host by extracting an attribute frequency distribution signature for every attribute of its flows and concatenating them. Then, they are learning the behaviour of benign hosts and malicious hosts by creating clusters using the DBSCAN clustering algorithm, where each cluster is indicated as malicious if it contains at least one bot host. Finally, the classification of a new host is done by finding its closest cluster, by examining its distance to the centroid of each cluster, and assigning it the label of this cluster.

This work performs the classification on host level and follows the same procedure for splitting the dataset in training and test dataset, that is the one proposed in [20]. Thus, their results can be compared with our results from the multi-scenario simulation. However there are two important differences that we should take into account during this comparison. The first is that while we build state machines from only malicious hosts, their method used the data from both malicious and benign hosts to create the clusters. The second difference is that they use the unidirectional version of the dataset, while we use the bidirectional one. An important difference between these two versions is that the are labeled differently. In particular, in the description of the dataset is mentioned that the bidirectional flows have better labels and better quality of data, so we should use these files for our research. The outcome of the usage of different versions of the dataset, is that in their case the number of benign hosts is much bigger in comparison with the version we used.

By keeping these differences in mind, we are going to compare the results of our system with the ones presented in the table 7.7. For our results, we are using the error-based evaluation method and the state machine build with the LSH state-merging heuristic using the 1-stable distribution, as this is the combination had the best performance.

The results of both method are presented in table 7.7. First of all we can observe that both models recognize correctly all the malicious hosts, as there are no false negative predictions. Regrading the false positive predictions, both methods have a few of them with my model outperforming in scenarios 1 and 8, and the BotFP outperforming in scenarios 6 and 9. In total, our model had 8 false alarms, while the false alarms of BotFp were 9. However, as discussed before, the BotFP model examines much more benign hosts, which can be easily seen from the number of true negative predictions, thus the comparison is not so fair. With more benign hosts examined, we expect that our model will more false alarms, so it performance will be worse than the BotFP model on this task. This is depicted in the accuracy of the classification, where BotFp achieves slightly better accuracy in all the scenarios, except for scenario 2 where both achieve 100% accuracy. However, this behavior was expected because we anticipated a loss in the accuracy of the model due to the approximations done to achieve a speedup. Also the fact that BotFP learns from both benign and malicious hosts, gives it an advantage over our model.

	Model	TP	FP	TN	FN	Accuracy
Scenario 1	My method	1	2	17	0	0.9
	BotFP	1	3	163	0	0.98
Scenario 2	My method	1	0	14	0	1
	BotFP	1	0	131	0	1
Scenario 6	My method	1	1	15	0	0.94
	BotFP	1	0	111	0	0.1
Scenario 8	My method	1	2	21	0	0.92
	BotFP	1	5	165	0	0.97
Scenario 9	My method	10	3	22	0	0.91
	BotFP	10	1	133	0	0.99

Table 7.7: Comparison of the classification results between BotFP and a state machine build with the LSH state-merging heuristic using the 1-stable distribution

7.7 Runtime efficiency

As discussed in the introduction of this chapter, the second part of the experiments conducted was about measuring the runtime of *flexfringe* with our proposed state-merging heuristic. More specifically, we used the 48 datasets used on the PAUTOMAC competition to measure the runtime of our method along with the quality of the models built. In order to do so we compared the perplexity score of our model and we compared to the perplexity score of the target model, which is the one used to build the dataset. The LSH famlily used for our state-merging heuristic was the LSH with 1-stable distribution, as it was the one achieved the best results in terms of classification performance, especially on the single scenario simulation. In order to measure the speedup achieved with the state-merging heuristic we propose, we compared it with the modified version of alergia heuristic discussed in section 2.4.3, as alergia is a well-known and fast state-merging heuristic. Also a team who took part in the PAUTOMAC competition with an alergia-based method achieved on of the best scores.

The results for each of the 48 datasets of the PAUTOMAC competition are presented in table 7.8, while some statistics about the performance of the two aforementioned methods are presented in table 7.9. These two tables show the runtime, the number states of the models and the perplexity score along with its difference from the optimal one. As it can be observed from table 7.9 the average runtime of our heuristic much smaller than the one of alergia heuristic. In particular, we achieved a speedup of $\frac{650.75}{120.5} = 5.4$. When the average difference of the achieved perplexity score and the optimal one for the two heuristics is examined, we can see that our heuristic's average error is bigger and especially 74% more bigger than the error of alergia heuristic. However, we expected this trade-off between the runtime and the error of the model, as our heuristic makes approximations about the children nodes of the nodes that are candidate for merge in order to achieve this significant speedup. Finally, regarding the number of states of the state machines built with these two heuristics, that the alergia heuristic tends to produce models with more states than the models built with our heuristic. Especially the mean number of states for the alergia heuristic is very high, but this happens because for some few datasets a model with too

many states was produced. This becomes clear when we examine the median value of the number of the states for alergia, which is much smaller than the mean value, but two times bigger than the one of our heuristic.

Set	Method	Runtime (s)	States	Perplexity score	Optimal Perplexity score
1	LSH	42	32	48.403	29.898
	ALERGIA	114	65	38.663	
2	LSH	48	43	201.130	168.331
	ALERGIA	93	48	172.654	
3	LSH	12	29	85.956	49.965
	ALERGIA	51	64	65.368	
4	LSH	10	23	145.515	80.818
	ALERGIA	18	109	134.219	
5	LSH	3	17	76.933	33.235
	ALERGIA	10	14	57.903	
6	LSH	30	39	148.041	66.985
	ALERGIA	132	86	122.841	
7	LSH	8	44	82.611	51.224
	ALERGIA	23	21	66.775	
8	LSH	126	54	191.256	81.375
	ALERGIA	291	252	167.945	
9	LSH	6	10	46.970	20.840
	ALERGIA	8	46	32.588	
10	LSH	143	50	57.279	33.303
	ALERGIA	1093	135	47.024	
11	LSH	553	78	88.409	31.811
	ALERGIA	765	120	67.647	
12	LSH	132	50	46.190	21.655
	ALERGIA	203	117	28.259	
13	LSH	31	45	104.436	62.806
	ALERGIA	41	180	86.840	
14	LSH	8	36	131.557	116.792
	ALERGIA	11	40	118.713	
15	LSH	117	41	80.947	44.242
	ALERGIA	1623	122	62.178	
16	LSH	145	45	52.791	30.711
	ALERGIA	657	173	37.190	
17	LSH	146	50	98.275	47.311
	ALERGIA	760	119	70.749	
18	LSH	55	105	81.449	57.329
	ALERGIA	87	731532	67.850	
19	LSH	175	58	26.101	17.879
	ALERGIA	183	175262	24.189	
20	LSH	227	38	141.510	90.971
	ALERGIA	870	67	117.268	

21	LSH ALERGIA	610 3320	61 95	100.072 79.031	30.519
22	LSH ALERGIA	37 54	101 1322137	42.082 35.706	25.982
23	LSH ALERGIA	119 1332	41 179	28.696 26.119	18.408
24	LSH ALERGIA	11 15	36 21	72.560 59.528	38.729
25	LSH ALERGIA	56 154	3372 72	97.362 84.648	65.735
26	LSH ALERGIA	17 23	46 150	140.900 122.501	80.743
27	LSH ALERGIA	138 158	48 66	78.804 63.875	42.427
28	LSH ALERGIA	7 10	34 41	100.122 84.966	52.744
29	LSH ALERGIA	7 6	40 108	45.104 35.412	24.031
30	LSH ALERGIA	23 121	31 102	37.603 29.360	22.926
31	LSH ALERGIA	8 10	40 59	55.705 53.350	41.214
32	LSH ALERGIA	40 59	42 134	53.351 42.985	32.613
33	LSH ALERGIA	18 20	40 18	46.526 32.040	31.865
34	LSH ALERGIA	819 13545	74 145	52.854 41.253	19.955
35	LSH ALERGIA	140 135	97 108	55.210 41.130	33.777
36	LSH ALERGIA	377 789	37 86	46.083 41.658	37.986
37	LSH ALERGIA	45 87	32 27	22.1023 21.928	20.980
38	LSH ALERGIA	47 56	32 11	31.179 30.331	21.446
39	LSH ALERGIA	45 80	39 40	11.478 10.659	10.002
40	LSH ALERGIA	117 498	52 114	10.873 10.250	8.201
41	LSH ALERGIA	173 181	32 31	15.717 15.193	13.912
42	LSH	8	26	18.862	16.004

	ALERGIA	17	22	17.035	
43	LSH	4	29	35.725	32.637
	ALERGIA	6	12	35.016	
44	LSH	97	33	13.287	11.709
	ALERGIA	213	24	12.897	
45	LSH	8	45	25.622	24.042
	ALERGIA	15	16	25.303	
46	LSH	459	61	16.144	11.982
	ALERGIA	2494	69	15.512	
47	LSH	97	84	6.813	4.119
	ALERGIA	150	343776	5.331	
48	LSH	226	63	11.345	8.036
	ALERGIA	655	43	8.860	

Table 7.8: Comparison of LSH and ALERGIA state merging heuristics for all the problem instances used in PAUTOMAC probabilistic automaton learning competition.

	Heuristic	**mean**	**std**	**median**	**min**	**max**
Runtime (seconds)	LSH	120.5	169.98	50.5	4	819
	ALERGIA	650.75	1989.85	117.5	6	13545
States	LSH	46.25	19.93	41	10	105
	ALERGIA	53672.45	218603.96	79	11	1322137
Error	LSH	27.18	23.52	21.76	1.22	109.88
	ALERGIA	15.59	17.42	10.05	0.17	86.57

Table 7.9: Cumulative statistics for the comparison of runtime, number of states and KL divergence score difference from the optimal one, for LSH and ALERGIA state merging heuristics.

7.8 Discussion

Regarding the single scenario simulation, we can reach to the conclusion that the error-based evaluation method has provided the best results, meaning that in for most of the scenarios and models, the accuracy was almost perfect. Also, it provided the more stable results, as the performance of each method didn't differ significantly per scenario. Regarding the models evaluated, the trigram model method and the state machine with the LSH with 1-stable distribution state-merging heuristic have achieved the best results in the majority of the cases. Especially, their performance was almost perfect as they had only one false prediction when the misclassified a benign host. It is impressive that our state-merging heuristic using LSH with 1-stable distribution has better performance than the alergia heuristic given that it makes many approximations when during the state-merging process, so we expected an from alergia to have same or better performance. This may be caused by the fact that the behavior of the hosts in CTU-13 is not so complicated, so our method achieves better results by inferring "simpler" models. In the single-scenario simulation, we expected that the results will be almost perfect because the models are trained on the same type of botnet with the ones they are trying to detect.

When it comes to the multi-scenario simulation, the error-based has provided again the best results for almost all the models, with no undetected malicious host and a few false alarms. The fingerprinting evaluation method produced more false positives as expected, while the acceptance rate one failed to recognize many of the botnets. Regarding the different models used to build the communication profile of the hosts, the state machines using LSH as a state-merging heuristic has provided equally good or even better classification results, which means that the approximations made by our method when merging two states are accurate. Especially the one using the 1-stable distribution LSH family, has the best overall results in all the scenarios. On this simulation, we expected that the number of false positives would be higher that the ones in the single-scenario simulation because the models are trained in different types of botnets than the botnets they are trying to recognize. The fact that they succeed to recognize all the malicious hosts, without few false alarms is impressive.

When we compare the performance of the state machine built with the LSH state-merging heuristic using the 1-stable distribution LSH, and the one of the state-of-the -art model BotFP, we observe that both achieve high accuracy with the one of BotFP being slightly higher. However, the fact that in the work for BotFP were used more benign hosts than in our project makes the comparison more difficult. Also the fact that BotFP learns from both benign and malicious hosts, gives it an advantage over our model.

Finally, regarding the experiments conducted about the runtime of the proposed heuristic and the quality built, we can conclude that the proposed state-merging heuristic significantly reduces the runtime of the state machine learning procedure when comparing it to the alergia heuristic, as it achieves a speedup of 5.4. However, its performance on capturing the underlying distribution of the target machine is slightly worse. This means that the proposed heuristic utilizing the LSH concept can be very beneficial in concepts where the speed of the model inference is more important than the small possible loss on the ”quality” of the model. Such a concept can be the one of learning state machine models in real-time. We also observed by the experiments in chapters 7.4 and 7.5 that this loss on the ”quality” of the model did not affect the performance on detecting malicious hosts. On the other hand, in tasks where the objective is to learn the best possible models without any concern about the time needed, other methods can be preferred.

8

Conclusion

The goal of this thesis is to introduce a new state-merging heuristic which will be faster than the existing ones and it can be used in a streaming fashion, but also efficient. We also use this heuristic to build state machines and use them to detect malicious hosts in network traffic data.

We started by implementing the proposed heuristic and integrating it to the *flexfringe tool*. Also, we experimented with three different LSH families in order to find out which one performs better. We tested our method on the task of detecting malicious hosts on the CTU-13 dataset. Two different types of simulation were used and three different methods of recognizing a host as malicious were tested. The performance of the state machine built with our proposed heuristic was compared with the performance of a state-of-the-art model. Furthermore, we used the dataset provided by the PAUTOMAC competition to compare the time needed to build a state machine with the proposed state-merging heuristic with the one when using the alergia heuristic. We also compared the quality of the aforementioned models using the perplexity score.

The results showed that the state machines built with our state-merging heuristic had similar classification results with trigram models or states machines built with the alergia heuristic. Also its results were comparable with the state-of-the-art model. Furthermore, the runtime was significantly smaller for with our proposed heuristic when comparing it with the alergia heuristic, while the quality of the model built was slightly worse.

8.1 Reflection on Research Questions

1. **How can I use the concept of LSH to create a fast state-merging heuristic that is accurate?**
   The whole procedure of how to use an LSH structure to store the future traces of each state and compare their distributions during the state-merging process to decide about the consistency of a merge was presented in section 4.1. By using this heuristic, we make the assumption that there no need to check the consistency of the merge of the descendant states (as done usually), as we had already examined the similarity of their future traces by comparing the distributions extracted by the LSH

table. By exploiting the properties of LSH and skipping these consistency checks we achieved a significant speedup without a significant loss in the model's quality.

2. **What kind of LSH family will be more efficient for the implementation of the heuristic?** Three different LSH families were examined, namely random hyperplanes, 1-stable distributions and 2-stable distributions. There are used to assign the vectors to a bin according to their angle-based, Manhattan and Euclidean distance respectively. Judging by the results of our experiments, we can conclude that LSH with 1-stable distribution was the one achieving the best results, especially on the single scenario simulation. This mean that the Euclidean distance performed the best on capturing the similarities between the future traces. However, its performance did not differ much from the performance of the other two LSH families.

3. **Which will be the implications of using this heuristic in terms of speed and performance?** Regarding the performance on detecting malicious hosts, the results of our state-merging heuristic were same or better than the results achieved by using the alergia heuristic or a trigram baseline model. Also they were comparable with the results of the state-of-the-art model. When it comes to the experiments we did with the PAUTOMAC dataset, we achieved as speedup of 5.4, but the quality (measured with the perplexity score) of the model built was slightly worse. This was an expected behavior, because of the assumptions made during the state-merging process. Overall, the the speedup achieved was significant. Regarding its performance, it achieved same or better results than the other method on the classification task, while the quality of the produced model, as measured in the PAUTOMAC competition datasets, was slightly worse than the one achieved using the alergia state-merging heuristic. Thus, as described in section 7.7, its usage can be beneficial for some type of tasks whose biggest priority is the runtime of the model production process.

8.2 Limitations

The work done on this projects has some limitations:

First of all, the heuristic proposed is using an LSH structure to store the future traces and compares the distributions extracted from them in order to decide whether two states can be merged or not. The buckets of the LSH structure will contain traces that are similar but not necessarily the same. Thus we compare an approximation of the real distribution of the future traces. Also the fact that we don't check the merge the consistency of the merges of the children states when following the procedure of Red-Blue fringe algorithm, is also an important approximation done in order to achieve a speedup. These approximations done during the state-merging process of the Red-Blue fringe algorithm, can lead to incorrect merges, thus non accurate models. Of course the accuracy of these approximations is affected mainly by the number of buckets used for the LSH structure. The more buckets we use, the more accurate the approximation should be. However, there is a clear trade-off between the accuracy of the approximation and the computational cost of the state-merging process, meaning that more buckets will approximate more accurately the real distribution of the future traces but they are more computationally expensive.

Another important limitation is the zero-padding done to the traces we store to the LSH structure. In particular, because we have traces with different length, we are padding them with zero values at the end until they reach the desired length. This is done in order to be able to extract their hash value and assign them to an LSH bucket. We expect that this padding will slightly affect the the capability of locality-sensitive hash functions to assign similar vectors in the same bucket. As a result, the distribution of the future traces extracted by the LSH buckets will be less accurate. However, as discussed in section 4.1.1, we expect that for the hash families we use, the loss in the accuracy won't be so high.

Also, as explained in section 5.1.5, in some scenarios of the CTU 13 dataset, there some hosts, especially normal hosts, that contain a very small amount of flows. This create an important issue during the host classification task, as the lack of a satisfying number of flows impairs the classification process of a host. As a result, we expect that some errors on the classification will occur because of the lack of data for the examined host.

8.3 Future work

On this project we used the LSH structure to hash the future traces for each state and then use the distribution extracted from them to check whether their merge is consistent. An improvement on this procedure would be to use some extra LSH structure to store also the past traces of each state, and then use their distribution for the consistency check of the merge. By using the past traces the comparison would be more robust, as we would have a better indication about the similarity of two states.

Furthermore, future work can apply different on the encoding of the flows and mainly on the extraction of the windows from the encoded flows of a host. In particular, we are using windows of fixed length to extract these sequences of events. The issue with that method is that one specific sequence may include events whose time difference is big. The other method of extracting these sequences id to apply a sliding window of fixed time. However, this will not work properly because of the spikes in the distribution of the number of flows over time, as shown in section 5.1.4. This can be solved by adjusting the time period of each window, but checking that the time difference of the flows is not big and also that the flows belong to the same connection. This would lead to sequences of flows which are actually meaningful.

Another interesting future work would be to experiment with different similarity measures when comparing two distributions to decide if a merge is consistent. There are many different similarity measures belonging to different families presented in [66]. This can possibly lead to a better judgement about the consistency of a merge, hence to better state merges.

Bibliography

References

[1] Richard Heady, George Luger, Arthur Maccabe, and Mark Servilla. The architecture of a network level intrusion detection system. Technical report, Los Alamos National Lab., NM (United States); New Mexico Univ., Albuquerque ..., 1990.

[2] Wenjie Hu, Yihua Liao, and V Rao Vemuri. Robust anomaly detection using support vector machines. In *Proceedings of the international conference on machine learning*, pages 282–289. Citeseer, 2003.

[3] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471, 2001.

[4] Eleazar Eskin, Andrew Arnold, Michael Prerau, Leonid Portnoy, and Sal Stolfo. A geometric framework for unsupervised anomaly detection. In *Applications of data mining in computer security*, pages 77–101. Springer, 2002.

[5] Basant Subba, Santosh Biswas, and Sushanta Karmakar. A neural network based system for intrusion detection and attack classification. In *2016 Twenty Second National Conference on Communication (NCC)*, pages 1–6. IEEE, 2016.

[6] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[7] James Brown, Mohd Anwar, and Gerry Dozier. An evolutionary general regression neural network classifier for intrusion detection. In *2016 25th International conference on computer communication and networks (ICCCN)*, pages 1–5. IEEE, 2016.

[8] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. *ACM SIGCOMM computer communication review*, 34(4):219–230, 2004.

[9] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining anomalies using traffic feature distributions. *ACM SIGCOMM computer communication review*, 35(4):217–228, 2005.

[10] Christian Callegari, Loris Gazzarrini, Michele Pagano, and Teresa Pepe. A novel pca-based network anomaly detection. pages 1–5, 01 2011.

[11] Yoshiki Kanda, Kensuke Fukuda, and Toshiharu Sugawara. Evaluation of anomaly detection based on sketch and pca. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–5. IEEE, 2010.

[12] Iwan Syarif, Adam Prugel-Bennett, and Gary Wills. Unsupervised clustering approach for network anomaly detection. In *International conference on networked digital technologies*, pages 135–145. Springer, 2012.

[13] Agathe Blaise, Mathieu Bouet, Vania Conan, and Stefano Secci. Botfp: Fingerprints clustering for bot detection. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE, 2020.

[14] Christian Hammerschmidt, Samuel Marchal, Radu State, Gaetano Pellegrino, and Sicco Verwer. Efficient learning of communication profiles from ip flow records. In *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, pages 559–562. IEEE, 2016.

[15] Gaetano Pellegrino, Qin Lin, Christian Hammerschmidt, and Sicco Verwer. Learning behavioral fingerprints from netflows using timed automata. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 308–316. IEEE, 2017.

[16] Ming-Yang Su. Discovery and prevention of attack episodes by frequent episodes mining and finite state machines. *Journal of Network and Computer Applications*, 33(2):156–167, 2010.

[17] 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.

[18] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

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

[20] Sebastian Garcia, Martin Grill, Jan Stiborek, and Alejandro Zunino. An empirical comparison of botnet detection methods. *computers & security*, 45:100–123, 2014.

[21] Sicco Verwer, Rémi Eyraud, and Colin Higuera. Results of the pautomac probabilistic automaton learning competition. In *International Conference on Grammatical Inference*, pages 243–248, 2012.

[22] Christian Hammerschmidt. *Learning Finite Automata via Flexible State-Merging and Applications in Networking*. PhD thesis, University of Luxembourg, Luxembourg, 2017.

[23] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection for discrete sequences: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 24(5):823–839, 2010.

[24] Nikolaevich Kolmogorov, Andreĭ and Albert T Bharucha-Reid. *Foundations of the theory of probability: Second English Edition.* Courier Dover Publications, 2018.

[25] M Sipser. Introduction to the theory of computation, pws pub. *Co., Boston*, 1997.

[26] Alan W Biermann and Jerome A Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers*, 100(6):592–597, 1972.

[27] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[28] 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.

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

[30] Borja Balle, Rémi Eyraud, Franco M Luque, Ariadna Quattoni, and Sicco Verwer. Results of the sequence prediction challenge (spice): a competition on learning the next symbol in a sequence. In *International Conference on Grammatical Inference*, pages 132–136, 2017.

[31] K Lang. Evidence driven state merging with search. *Rapport technique TR98–139, NECI*, 31, 1998.

[32] Borja Balle, Jorge Castro, and Ricard Gavalda. Adaptively learning probabilistic deterministic automata from data streams. *Machine learning*, 96(1-2):99–127, 2014.

[33] 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.

[34] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In *International Colloquium on Grammatical Inference*, pages 203–216. Springer, 2010.

[35] Rafael C Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In *International Colloquium on Grammatical Inference*, pages 139–152. Springer, 1994.

[36] 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.

[37] 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.

[38] Kyung Mi Lee, Yoon-Su Jeong, Sang Ho Lee, and Keon Myung Lee. Bucket-size balancing locality sensitive hashing using the map reduce paradigm. *Cluster Computing*, 22(1):1959–1971, 2019.

[39] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[40] M Visser. Feature fusion for efficient content-based video retrieval. 2013.

[41] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[42] Monowar H Bhuyan, Dhruba Kumar Bhattacharyya, and Jugal K Kalita. Network anomaly detection: methods, systems and tools. *Ieee communications surveys & tutorials*, 16(1):303–336, 2013.

[43] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31, 2016.

[44] Nour Moustafa, Jiankun Hu, and Jill Slay. A holistic review of network anomaly detection systems: A comprehensive survey. *Journal of Network and Computer Applications*, 128:33–55, 2019.

[45] Gilberto Fernandes, Joel JPC Rodrigues, Luiz Fernando Carvalho, Jalal F Al-Muhtadi, and Mario Lemes Proença. A comprehensive survey on network anomaly detection. *Telecommunication Systems*, 70(3):447–489, 2019.

[46] Vladimir Vapnik. The support vector method of function estimation. In *Nonlinear Modeling*, pages 55–85. Springer, 1998.

[47] Qing Song, Wenjie Hu, and Wenfang Xie. Robust support vector machine with bullet hole image classification. *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, 32(4):440–448, 2002.

[48] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *computers & security*, 31(3):357–374, 2012.

[49] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[50] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[51] Ian T Jolliffe. Principal components in regression analysis. In *Principal component analysis*, pages 129–155. Springer, 1986.

[52] CSL Sony and Kenjiro Cho. Traffic data repository at the wide project. In *Proceedings of USENIX 2000 Annual Technical Conference: FREENIX Track*, pages 263–270, 2000.

[53] Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. *Concentration inequalities: A nonasymptotic theory of independence*. Oxford university press, 2013.

[54] Neil Walkinshaw, Kirill Bogdanov, Christophe Damas, Bernard Lambeau, and Pierre Dupont. A framework for the competitive evaluation of model inference techniques. In *Proceedings of the First International Workshop on Model Inference In Testing*, pages 1–9, 2010.

[55] Catalin Dima. Real-time automata. *Journal of Automata, Languages and Combinatorics*, 6(1):3–24, 2001.

[56] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*, 2014.

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

[58] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th annual IEEE symposium on foundations of computer science (FOCS'06)*, pages 459–468. IEEE, 2006.

[59] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern recognition letters*, 31(11):1348–1358, 2010.

[60] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[61] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, pages 950–961, 2007.

[62] Hervé Jégou, Laurent Amsaleg, Cordelia Schmid, and Patrick Gros. Query adaptative locality sensitive hashing. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 825–828. IEEE, 2008.

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

[64] 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.

[65] Cyril Goutte, Peter Toft, Egill Rostrup, Finn A Nielsen, and Lars Kai Hansen. On clustering fmri time series. *NeuroImage*, 9(3):298–310, 1999.

[66] Sung-Hyuk Cha. Comprehensive survey on distance/similarity measures between probability density functions. *City*, 1(2):1, 2007.