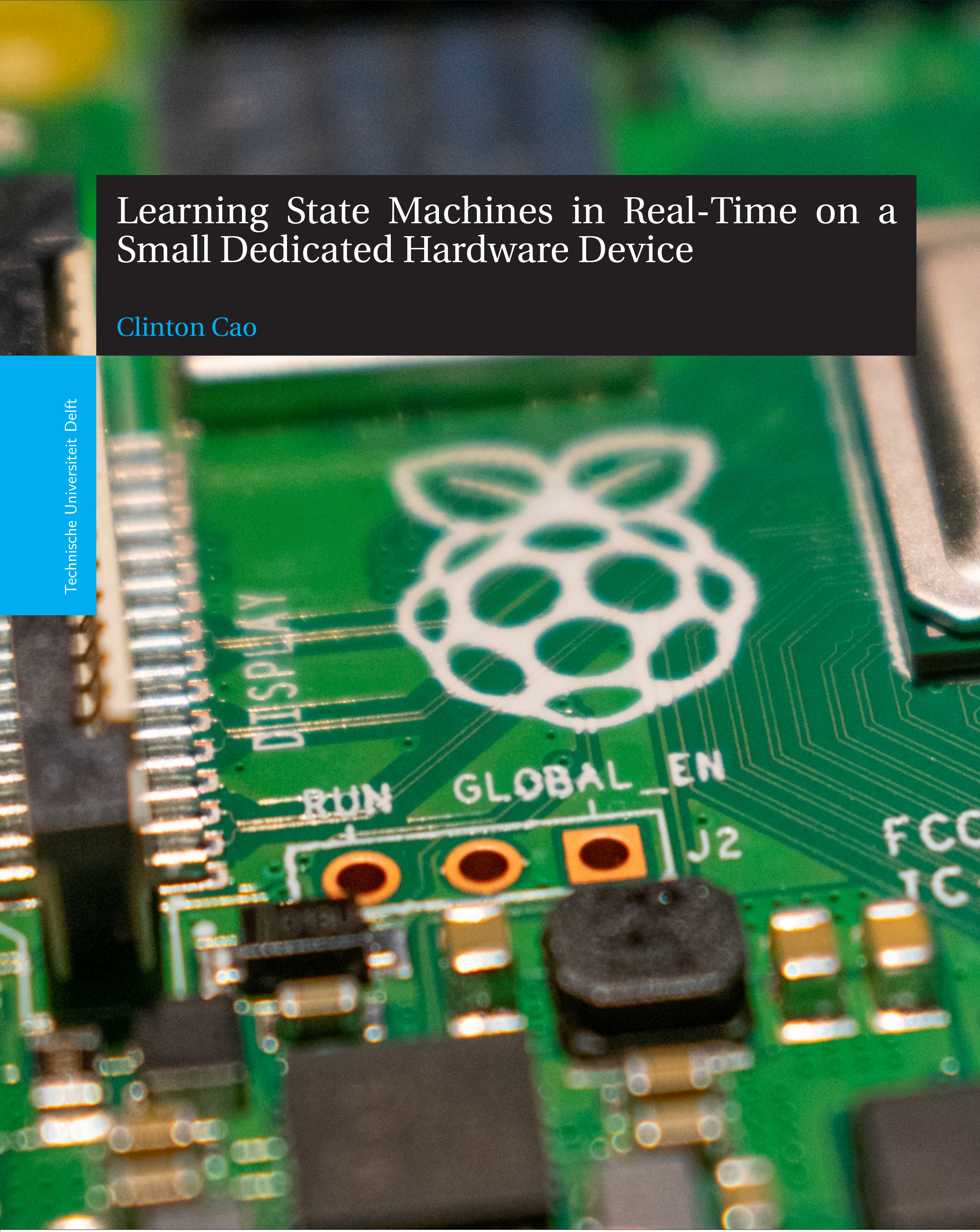


Learning State Machines in Real-Time on a Small Dedicated Hardware Device

Clinton Cao

Learning State Machines in Real-Time on a Small Dedicated Hardware Device

by

Clinton Cao

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on 23 November 2020 at 10:00.

Student number:	4349024	
Project duration:	October 1, 2019 – November 23, 2020	
Thesis committee:	Prof. dr. ir. R. L. Lagendijk,	TU Delft, Chair of Committee
	Dr. ir. S.E. Verwer,	TU Delft, Supervisor
	Dr. A. Panichella,	TU Delft, Committee Member

This research was conducted together with APTA Technologies.

Abstract

The Internet is a technology that was invented in the 1960s and was used only by a few users to do simple communications between computers. Fast forward to 2020, the Internet has become a technology that is being used by billions of users. It allows users to communicate with each other across the world and even allows users to access data without geographic restrictions. The Internet has made the lives of many people easier but it also comes with a price; many malicious users also want to have access to data. Therefore, it is needed to secure our networks to make sure that no attackers can exfiltrate data from a network. One way to do so is to use smart methods to detect anomalies in the network. Recently, a new method has been proposed to learn state machines in real-time from network traffic data. The state machines are then used for anomaly detection. This method was designed to be used on a larger system such as a desktop computer.

In this work, we investigated how we can use the newly proposed method to learn state machine in real-time on a smaller device. Smaller devices are cheaper and more mobile than larger systems but these have limited resource compared to the larger systems. Therefore, modifications would need to be made to the method for it to run efficiently on a smaller device. In this work, we propose to use the concepts of Locality Sensitive Hashing to improve the run-time of different parts of the method. We also attempted to reduce its memory footprint. In this work, we show the modifications that we have made and evaluated our modifications with different experiments that used both artificial and real-world data. From our results, it shows that we can use a smaller device to learn state machines in real-time and use these state machines for anomaly detection. Though our modifications have provided an improvement on parts of the method, there are still improvements that can be done.

Preface

Before you lies my thesis titled *"Learning State Machines in Real-Time on a Small Dedicated Hardware Device"*. This thesis was written for me to obtain my Master's degree at Delft University of Technology. Looking back at my whole journey of writing my thesis, it has taken a bit longer than I have expected. With the Netherlands also being affected by the Covid-19 pandemic, I had to adapt to a new working routine just like any other citizen of the country. Nonetheless, this journey was still a nice experience and I have learned much from it. I can now say that I have made it and I am very proud of it. Of course, I could have not made it to the end of this journey without the support and help of others. I would like to formally thank them here.

Firstly, I would like to thank my supervisor, Sicco, for his guidance throughout this whole journey. Thank you for helping me when I was stuck and did not know how to proceed further. Also, thank you for the critical feedback on my work and for motivating me to achieve the best that I can.

Secondly, I would like to thank my daily supervisor, Chris Hammerschmidt, for his guidance and for not getting annoyed from all the questions that I had asked him throughout this whole journey. Thank you for the brainstorm sessions, discussions, and feedback on my ideas. I would like to also take the opportunity here to thank all the colleagues from APTA Technologies. The coffee breaks were always fun and it was also fun to work with all of you.

Thirdly, I would like to thank Cas, Daniël, Felix, Jehan, Jorai, Maurits, Roy, Tim and Tristan for the fun coffee and lunch breaks. Though the pandemic has limited the number of breaks that we could have had with each other, it was always fun to have a chat with all of you during the breaks. Of course, the coffee and lunch breaks would not be fun without our beloved Sandra. I would like to thank you for the coffee breaks and also helping me with all the tasks that were related to the graduation process.

Finally, I would like to thank my friends and family for their support throughout my studies. Thank you all for the fun and laughs that we have had with each other, these were much needed. Also, thank you all for the mental support that you all have given me. This journey would not have been as fun as it was without every single one of you.

Clinton Cao
Delft, November 2020

Contents

1 INTRODUCTION

The Internet, a technology that was invented in the 1960s to allow computers to do simple communication with each other. Since then, the Internet has only been growing and it is being utilised by many users across the world. According to Statista, there are over 4.33 billion active users of the Internet as of July 2019 [1]. With this many users online, one can expect that many machines are connected using the Internet and that different types of data can be accessed at any time and anywhere. Take the cloud as an example, one can upload their photos onto the cloud and then access them without any geographic restrictions (granted that they have an internet connection). This eases the lives of many people as they do not have to be physically there to access the data, they just need to be connected to the internet and they can retrieve the data that they need. This does not only benefit the casual internet user but also corporations; employees of a company can access company data via the internet when they are on a business trip or when they are working from home.

Having the possibility to access data at any time and anywhere also comes with a price; some malicious users (attackers) also want to have access to the data and they can gain access to the data without any geographic restrictions. There are many motives why an attacker would like to have access to the data. Different studies have shown that the major motive of an attacker is financial gain [2–4].

There exist different types of methods that an attacker can use to get into the network to get access to the data. One of these methods is to send out phishing emails to trick individuals to click on a link. Once an individual has clicked on the link, they would be redirected to a site where they would need to sign in with their credentials or download a piece of malicious software. The attacker can then either log the credential that the individual has used or try to get into the network using the malicious software that was downloaded by the individual. They can then try to exfiltrate data once they have successfully gained access to the network.

Thus it is essential to make sure that no attackers can gain unauthorised access to the network. Being able to detect that an attacker is in the network, one can mitigate or even completely neutralise the damage that they can cause. A widely used solution for this problem is to deploy an intrusion detection system (IDS) on the network. An IDS is a system that is specifically configured to a given network and detect malicious or abnormal activities (anomalies). Once an anomaly is detected, an alert will be sent to the network administrator(s) for further analysis. An IDS, therefore, acts as the first line of defence against attackers that are trying to infiltrate the network.

Using an IDS does come with a price; some normal activities might be falsely flagged as an anomaly e.g. benign network traffic data is flagged to be malicious. This is known as a false positive. False positives can be costly as each alarm that is triggered by a false positive would require a network administrator for further analysis to verify whether it is indeed a false positive. With the growth of network traffic, analysing false positives becomes more troublesome as there is a wide variety of network traffic and much of it might be falsely flagged as anomalies. With the growing number of false positives, network administrators would have to put in more labour to analyse the false positives.

Additionally, not only does an IDS need to be able to detect that an attacker is in the network, but it should also trigger the alerts on time. The longer an attacker stays in a network, the more time the attacker has to find valuable data and exfiltrate the data from the network. Thus by triggering an alert as soon as an attacker is detected in the network, one can reduce the damage that can be done by an attacker.

Therefore when designing an IDS, one need to make sure that it does not produce too many false positives and that it creates the alerts on time when it has detected that an attacker has infiltrated the network.

1.1. DETECTING ANOMALIES USING EMBEDDED DEVICES

While a lot of research has been done on anomaly detection [5–11], most of the solutions are designed to work on larger systems such as a desktop computer. To deploy these solutions on an IDS, one would need to already have such a machine, and if not, they would need to purchase one. This brings an increase in the costs of deploying such an IDS. One way on how we could reduce the cost is to deploy an IDS on a smaller embedded device. These devices are usually much cheaper and more portable than a desktop computer.

Smaller embedded devices do have their limitations; they have a limited amount of resources available to be used. Simply deploying an IDS using the previously mentioned methods will not work out of the box on a small embedded device. Research has been carried out to assess whether it is possible to detect anomalies using an embedded device [12–24]. From these works, we see that different solutions have been proposed to be used to detect anomalies on an embedded device.

From these works, one interesting method is to learn state machine and use them to detect anomalies [21–25]. The state machines are learned from given input data and they are used to model the normal behaviour of a given system. When the system has executed a behaviour that was not captured in the state machine, this is an indication that the system is doing something unusual and this can be flagged as an anomaly. A formal definition of a state machine is given in Section 2.1.

1.2. PROBLEM STATEMENT

From the works where state machines are learned to detect anomalies, the work that was done by Schouten is particularly interesting to look at. Schouten proposed a highly effective method that can be used to learn state machines in real-time from a stream of network traffic data and these state machines can be used to detect anomalies on a network [25]. In comparison to the works mentioned in Section 1.1, this work uses a different algorithm to learn and construct the state machines; principles of the Blue-Fringe algorithm is used to learn the state machines [25]. This algorithm is further explained in Section 2.2.

This method, however, was designed to work on larger systems (e.g. desktops computers/laptops) and as mentioned before, there are benefits for using an embedded device to detect anomalies. In this work, we investigate how we can utilise this method to learn state machines in real-time on an embedded device. We hypothesise that the current implementation of the method might work out of the box on an embedded device but it will run into performance issues due to limited memory and processing power of the embedded device. Thus modifications would need to be made for it to work efficiently on an embedded device. Our hypothesis is based on the frameworks that are used to implement the current solution and because it was designed to work on a larger system that has more resource available. In this work, we specifically investigated how we can use the concepts of Locality Sensitive Hashing (LSH) [26] to make modifications on the method to improve its run-time. The concept of LSH is further explained in Section 2.4. Additionally, we will also investigate how we can use the Misra-Gries Summary algorithm to reduce memory consumption. A description of the algorithm is given in Section 2.3.2.

1.2.1. MOTIVATION OF USING LSH

In the current implementation of the method, the state-merging process would require us to go through all the states to find the most similar state. A visual representation of this process can be seen in Figure 1.1. When dealing with a large number of states, this would have an impact on the run-time of the learning process. To reduce the number of states that we need to evaluate in this process, we use LSH to hash each state into a bucket. The intuition behind this idea is that similar states would be hashed into the same bucket and when looking for the most similar state for a merge, we only need to search for a most similar state within a particular bucket. This approach can be seen as an approach to cluster similar states together and each bucket represents a cluster. A visual representation of this process can be seen in Figure. 1.2.

Additionally, the Kullback-Leibler divergence (KL-Divergence) is used as a metric to compute the similarity between two state machines. This metric measures how much one statistical distribution differs from another [27]. By using the concepts of LSH to not just only hash states but also to hash state machines, it provides us with a method to cluster similar state machines together. Thus we make an extension on the current method to not only detect anomalies but also cluster state machines. The intuition behind this idea is the same as the one for hashing states: similar state machines would be hashed into the same bucket and each bucket represents a cluster. In the current implementation, to see whether there are state machines that model the same behaviour (i.e. finding cluster of state machines), pair-wise comparisons would need to be done between all the state machines. When the number of state machine grows very large, this would impact the performance. With LSH, we only need one single pass over the state machines to find similar state machines and there is no need to do any pair-wise comparisons.

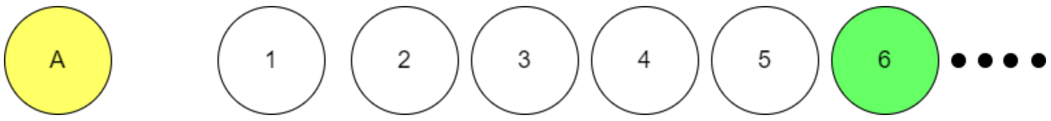


Figure 1.1: Finding the most similar state (marked in green) for a given state (marked in yellow) using the original state-merging method. In this example, we would have to iterate over 5 states before finding the most similar state.

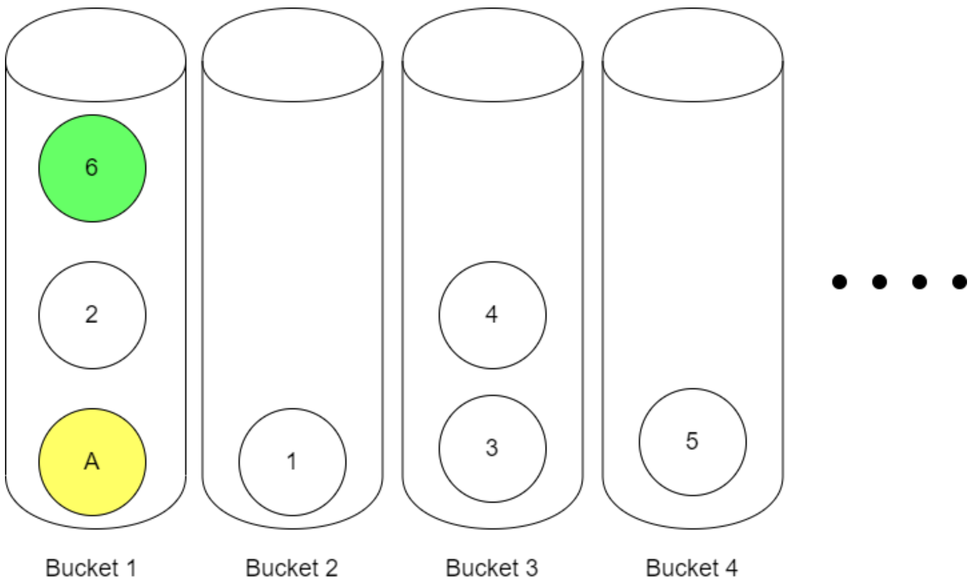


Figure 1.2: Finding the most similar state (marked in green) for a given state (marked in yellow) using LSH. In this example, we have to iterate over 1 state before finding the most similar state.

1.3. RESEARCH QUESTIONS

The objective of this research is to learn state machines in real-time from a stream of network traffic data on an embedded device. To investigate whether this is possible, we use the concepts of LSH to make optimisation on the original method so that it can be run efficiently on a smaller embedded device. This let us formulate the following research question (RQ):

RQ: How can we learn state machines in real-time from a stream of network traffic data on an embedded device?

This research question is further broken down into the following subquestions (SQ):

1. *What kind of modifications do we need to make to the original method that is proposed by Schouten?*
2. *How can we cluster states using LSH?*
3. *How much do the state machines, that are learned using the LSH approach, differ from the ones that are learned using the original method?*

4. *How can we cluster state machines using LSH?*

5. *How does LSH perform in comparison to KMeans Clustering?*

6. *How does the new version of the method, that utilises LSH, perform on a smaller embedded device?*

1.4. CONTRIBUTIONS

The answers to the questions that are listed in Section 1.3 would provide us with more insights on the following points:

- Firstly, we show whether it is possible to apply the method that was proposed by Schouten [25] on an embedded device, and thus also, to the best of our knowledge, be the first to show whether it is possible to learn state machines in real-time on an embedded device, using the Blue-Fringe algorithm. Being able to do so, it would also show that there is no need for large expensive systems to learn state machines from a stream of NetFlow data. Embedded devices are more mobile and cheaper when compared to the larger systems.

- Secondly, to our best knowledge, this is the first work that utilises the concepts of LSH to cluster states/state machines together. This work provides insights on how we have represented each state/state machines and how we have used this representation for clustering.

- Finally, we provide empirical evaluations on how well LSH performs in the creation and clustering of state machines. The clustering performance of LSH is compared to the clustering performance of KMeans Clustering. This provides insights on how well LSH performs in comparison to a well-known clustering algorithm.

1.5. THESIS OUTLINE

The rest of this thesis is structured as follows: Chapter 2 we provide necessary background information that is needed to understand the concepts and terms that are explained in this thesis. Chapter 3 describes the methodology of how we approached this research. Chapter 4 and Chapter 5 present the results of the experiments where we clustered states and state machines using LSH, respectively. Chapter 6 presents the results of the experiment that we have run on an embedded device. In Chapter 7, we discuss the limitations of our work. Finally, in Chapter 8, we conclude our work and list some future work.

2 BACKGROUND

In this chapter, we provide a detailed description of the terms and concepts that are used in this thesis. These terms and concepts are important for the understanding of the work that is done in this thesis. First, we provide the formal definition of a state machine in Section 2.1. Then in Section 2.2, we describe the Blue-Fringe algorithm. A description of the streaming paradigm is given in Section 2.3. Section 2.4 describes the concept of Locality Sensitive Hashing. Finally, Section 2.5 provides some related work. The reader can feel free to skip or skim through this chapter if they are already familiar with the concepts that are explained in this chapter.

2.1. STATE MACHINES

A state machine (formally known as a finite-state automaton or a finite-state machine) is a mathematical model that is used to describe a system and its behaviour. Formally, a state machine is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ [28], where:

- Q is the finite set of states.
- Σ is a finite set of symbols that are used as input.
- δ is a function that describes how the machine will between states based on the given input.
- q_0 is the start state of the machine, where $q_0 \in Q$
- F is a set of final states of the machine, where $F \subseteq Q$.

An example of a state machine is given in Figure 2.1, where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, $F = \{q_2\}$ and δ is defined as a transition table that is shown in Table 2.1. The state transition table provides an overview of all possible transitions that can occur for a given state [28]. The state machine reads the input one character(symbol) at a time (in this case it can be either a 0 or a 1) and based on the character that it has read, it switches to the corresponding state.

For example for the string 0110, it switches from q_0 to q_1. Then it reads the second character and it stays in q_1. For the third character, it stays in q_1 again. Finally, it reads the last character and it switches from q_1 to q_2. Because we ended at q_2 after reading the entire input, we say that the state machine has accepted the input string.

This input string belongs to a set of strings that are accepted by the state machine. We call this set the language of the state machine. In this example, the language of the state machine is 01*0, where the * denotes that there are zero or more consecutive ones in the string.

2.1.1. USING STATE MACHINES FOR ANOMALY DETECTION

Knowing the language of the state machine, we know what kind of strings will not be accepted by the state machine. If a string that is not part of the language still causes the state machine to end up in one of the final states, then we know that this is an abnormal behaviour of the state machine. This is a useful property that can be utilised for anomaly detection. As shown in [7, 11, 23, 24], different types of input data can be used

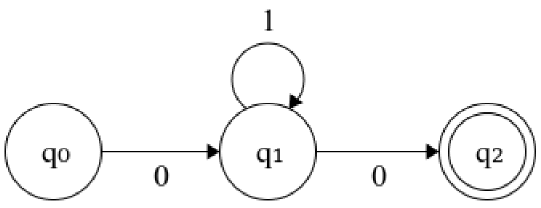


Figure 2.1: An example of a state machine that accepts strings from the language of 01*0.

	0	1
q_0	q_1	–
q_1	q_2	q_1
q_2	–	–

Table 2.1: Transition function, represented as a transition table of example state machine that is shown in Figure 2.1.

to learn state machines. Thus we can use the input data to build the language of the state machine. A string from the language represents the set of actions that are executed by a system. If the system has executed a set of actions that are not modelled in the state machine, this implies that we have found a string that is not in the language but it is still accepted by the state machine. This should be flagged as unusual behaviour (anomaly) of the system.

However, there is a drawback in using state machines for anomaly detection. In 1989, Pitt and Warmuth have proved that finding a state machine with the minimum amount of states and that is consistent with the given training samples, is an NP-hard problem [29]. Therefore, when learning the state machines, we need to be approximate the state machines. But approximating the state machines, it means that they are not complete and might not capture all normal behaviours of a system. This consequently means that normal behaviour might be flagged as malicious behaviour (false positive). The Blue-Fringe algorithm [30] was proposed for the learning of state machines and it works well on randomly generated problems. The algorithm is further explained in Section 2.2.

2.2. BLUE-FRINGE ALGORITHM

The Blue-Fringe algorithm starts by creating a prefix tree acceptor from the training data [30]. A prefix tree acceptor (PTA) is a state machine that has the form of a tree. The prefixes in the training data are used to create the smallest possible state machine [31]. A simple example of a PTA is shown in Figure 2.2. In this example, the PTA accepts the string *ac, ab, ca* and *cc* and they are stored in the leaf nodes of the tree. The prefixes of each string are stored in the ancestor nodes. Once the PTA has been created, the algorithm colours the nodes of the tree [30]:

- the root node gets coloured red.
- the children of root node get coloured blue.
- all other remaining nodes are coloured white.

After colouring the nodes in the tree, the algorithm starts to merge states to form a smaller state machine. The merging of the states is done with the help of the colours; a blue node is merged with a red node. A score is given for the merge of a pair and it is used to evaluate the quality of the merge. The score of a merge is given based on three criteria [30]. The merging of the states is an iterative process that consists of the following actions:

1. Compare each blue node with all red nodes and compute the candidates for merging
2. Compute the score for the candidate and evaluate the merges
3. If no blue node can be merged with any red node, then promote a blue node to a red node and go to step 1.

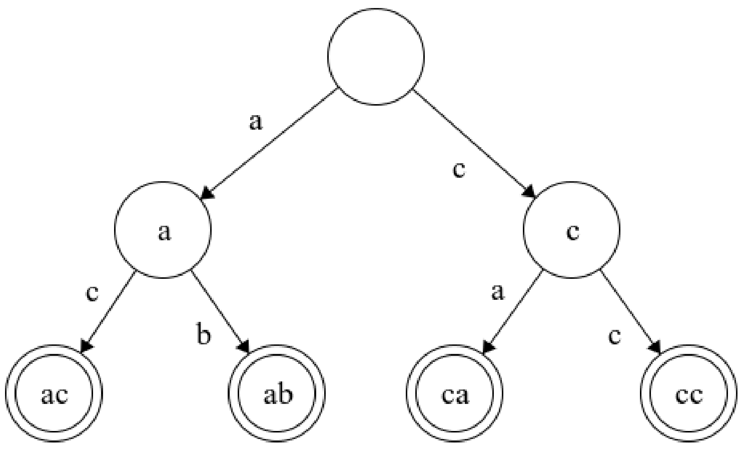


Figure 2.2: An example PTA that accepts the strings *ac, ab, ca* and *cc.*

4. If no blue node can be promoted, then merge the highest scoring candidate and update the colours of the nodes. Then go back to step 1.
5. Output the final state machine.

2.2.1. ADAPTED VERSION OF THE BLUE-FRINGE ALGORITHM

Schouten has pointed out that the Blue-Fringe algorithm has a high memory footprint and high computation complexity. This causes scalability issues you want to learn state machines from a stream of data [25]. Thus an adapted version of the Blue-Fringe algorithm has been proposed by Schouten, namely the *Streaming Blue-Fringe* algorithm. The proposed algorithm learns iteratively from the data stream but it does not learn the full PTA as you would in the Blue-Fringe algorithm. Additionally, approximations are used to set a halting point for the learning process [25].

2.3. THE STREAMING PARADIGM

There are two methods on how one can process data. The first method is to gather data and store the data in a (giant) batch, and this will be used later for processing. Because it is gathered in a batch, the data is stationary and will not change. Additionally, the first and last element of the batch is known. Because the data does not change, it can be reused multiple times for data processing.

The second method for processing data is called stream processing. In contrary to the first method, the data comes in a continuous stream and we do not know when we will reach the end of the stream. Thus it is hard to know which element is the last element of the data stream. Due to the aforementioned properties, we cannot store the entire stream on disk; we do not know how much space we need to store the entire stream and adding more disk space would also increase the cost. Additionally, we have a limited amount of time to process each element of the stream. If we do not process each element from the stream as quickly as possible, the element might be lost and we might miss important information [26].

One way to solve these issues of stream processing is to sample elements from the stream to create a summary or an approximation that best describe the whole stream. This summary or approximation can then be used to answer a particular question(s) [26]. The drawback of this method is that the answers are not exact and therefore we need to use algorithms that have high accuracy.

2.3.1. SAMPLING METHODS

As mentioned before, one needs to make sure that they sampled elements from the stream that best describe the stream. Several methods exist for sampling elements from a stream. We will describe three existing methods.

RESERVOIR SAMPLING

With reservoir sampling, a sample of size k it selected from a stream with an unknown length n. Usually, the first k element of the stream is selected and these are stored in a reservoir with size k. For each consecutive element $i > k$ of the stream, sample the element with probability k/i. If this element is sampled, then randomly replace one of the previously sampled elements that are stored in the reservoir [32]. At the end of the stream, the elements that are in the reservoir will be used as the sample that represents the stream.

MIN-WISE SAMPLING

In Min-Wise Sampling, each element in the stream gets a tag. Each tag is just a random number from the interval of $[0,1]$. From the stream, only k elements with the smallest tag are kept [33, 34]. These k element will be used as the sample that represents the stream.

CISCO SAMPLED NETFLOW

The solution that was designed by Schouten uses Netflow data to learn state machines, this means that a different type of sampling method would need to be used. Cisco has developed a sampling method for this kind of use case [35]. The method randomly selects one packet out of every n packets. The selected packets are then aggregated and these are used for approximating the actual statistics of the NetFlow data [25].

2.3.2. OTHER EXISTING STREAMING ALGORITHMS

Besides the sampling methods, there are other existing streaming algorithms, that were developed to solve a particular problem and we will describe three algorithms. These algorithms were designed to solve some of the most known problems in stream processing. From these three algorithms, the last algorithm is used by Schouten to count the sequences that have occurred in a particular state [25].

FLAJOLET-MARTIN ALGORITHM

One well-known in problem in stream processing is finding the number of distinct elements in the stream. The Flajolet-Martin algorithm is an algorithm that can be used to estimate the number of distinct elements in the stream with just a single pass over the stream [36]. The algorithm first hashes each element into a n bit value string. The algorithm keeps track of a bit string that has the largest number of trailing zeroes in the bit string. Take the bit string 1001000111100000 as an example. In this example, there are 5 trailing zeroes. Let k denote the maximum number of trailing zeroes seen from the bit strings. At the end of the stream, the number of distinct elements is estimated using the following formula: $number\ of\ distict\ elements = 2^k$

MISRA-GRIES SUMMARY

Another well-known problem in stream processing is finding the most frequent elements from a stream with an unknown size of n. The Misra-Gries Summary is an algorithm that was designed to solve this problem [37]. The algorithm uses associative arrays to store the k most frequent elements and their count. For each element in the stream, the algorithm checks whether it has seen this element before. If the algorithm has seen this element, then increment its count in the array that store the count of the element. If the algorithm has not seen the element before and the array is not full, then store it in the array and set its count to 1. If the algorithm has not seen the element before and the array is full, then decrement the count of every element in the array and if the count of an element has reached to 0, then remove it from the array. At the end of the stream, the algorithm returns the elements that have a frequency of more than $\frac{n}{k}$ [37].

COUNT-MIN SKETCH

Count-Min Sketch is another well-known algorithm to efficiently estimate the frequency of the elements [38]. The algorithm uses a two-dimensional array to store the count of the elements. This two-dimensional array can be seen as a matrix of size $d \times w$. Initially, each cell in the matrix is set to zero. Additionally, there are d hash functions, each belonging to a particular row. These hash functions are used to update the count in the cells. Each hash function takes an element from the stream as the input and hashes this element to a number that is between 0 and w. This number is the index at where an update will occur in the row. The corresponding entry of the row is then incremented by one. An example visualisation of this operation can be seen in Figure 2.3. To get the frequency estimation of an element, one can ask the algorithm to give the count for that particular element; the element is hashed using the hash functions. The values of the corresponding cells are then retrieved and the lowest count will be used as the frequency estimation for the element. Count-min Sketch is an essential algorithm for the work of Schouten, as it is used to compute the most frequent patterns from the NetFlow data [25].

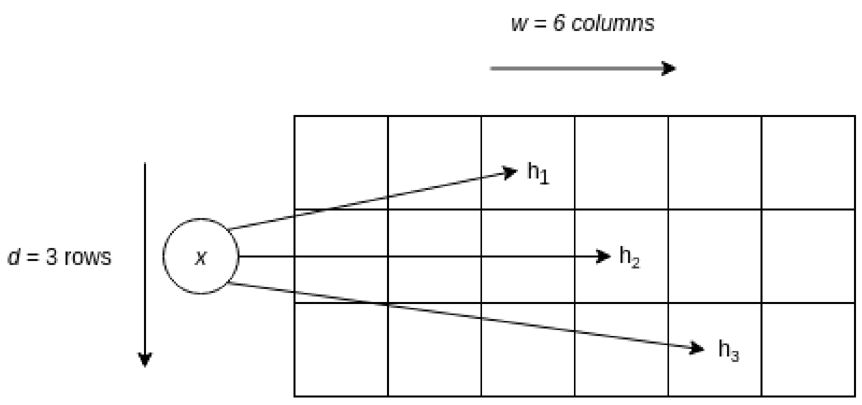


Figure 2.3: The update operation of Count-Min Sketch.

2.3.3. EXISTING DATA PROCESSING & STREAMING FRAMEWORKS

One known problem with processing data today is the fact that data comes in a large volume. One would need to use scalable solutions to process data. Several frameworks exist that can be used to process data. We will mention a few of these frameworks.

APACHE HADOOP

Apache Hadoop is a well-known framework for processing a large volume of data across a cluster of computers using the MapReduce programming model [39]. The framework uses a storage system, that is known as Hadoop Distributed File System (HDFS), for storing data and distributing data across the cluster of computers. Processing the data is done in two different phases: the Mapping Phase and the Reduce phase. During the mapping phase, all data are split into blocks and they are prepared before they are sent to the cluster of machines. When the splitting and preparations are done, the blocks are shuffled and they are distributed over the cluster of machines. During the reduce phase, each machine in the cluster aggregates the blocks and outputs the results. The results of each machine can then be aggregated again. The mapping and reduce phases can be seen in Figure 2.4.

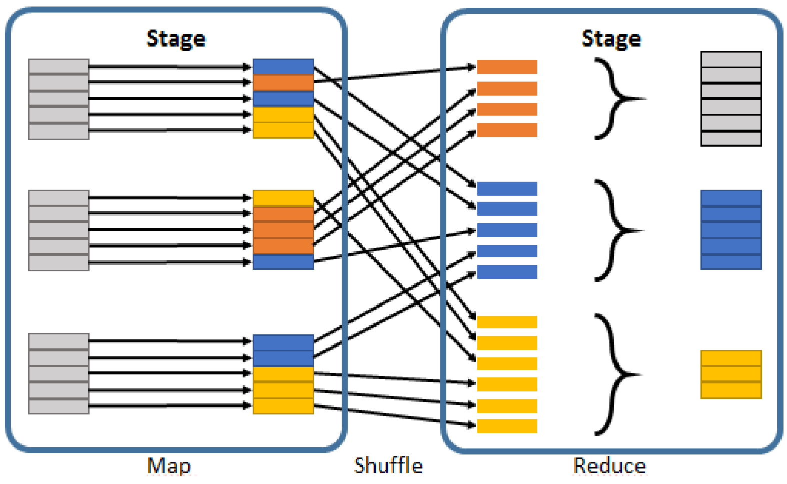


Figure 2.4: The Mapping phase and Reduce Phase of Hadoop [25].

APACHE FLINK

Another known data processing framework is Apache Flink. In contrast to Hadoop, Flink provides both support for stream processing and batch processing, whereas Hadoop only supports batch processing [40]. In Flink, the input data is fed to the source nodes and the results come out of the sink node. In between the source nodes and sink node, different operations can be performed on the data to get the desired results in the sink node [25]. An example of this whole process is shown in Figure 2.5. Flink has a built-in fault tolerance mechanism, where a global state is maintained with multiple checkpoints. In case of a failure, the machines in the cluster can switch back to the last checkpoint and continue with the job[25]. One important property of Flink is that each element of the stream gets processed exactly once, even in the case of failures [25]. This is important for the correctness of the results.

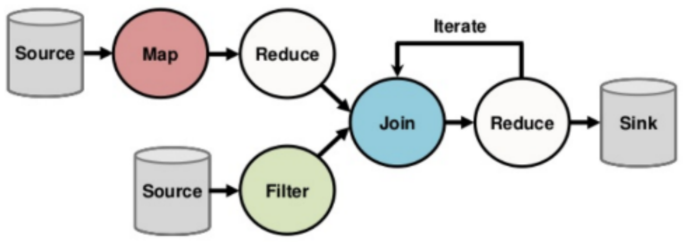


Figure 2.5: An example of Flink processing two streams of data [25].

The method that was designed by Schouten utilises Apache Flink for learning and creating the state machines [25]. Flink was chosen based on the following points [25]: firstly, Flink can a stream of data without having to store it on disk. Secondly, the property of processing each element of the stream exactly once (even in case of failures). Finally, Flink achieves a higher performance in comparison to other data processing frameworks.

APACHE KAFKA

Apache Kafka is a framework that is developed for distributed stream processing [41]. It acts as a messaging system, where it gathers the messages that are published by the publishers and it forwards them to the corresponding topics. To get the messages that are published to a particular topic, one needs to subscribe to that topic. In the method that was designed by Schouten, NetFlow data are sent to the Kafka and it redistributes the data to the corresponding channel in which Flink is connected to [25]. Kafka was chosen due to the fact it has low latency, it is fault-tolerant, easy to configure and Kafka can be deployed in self-controlled environments [25]. Using Kafka does have a drawback. As mentioned in [15], a Java Virtual Machine (JVM) is needed to run Kafka as it is written in Java. As we are trying to apply Schouten's method on embedded devices, the devices must have enough memory to run a JVM. Additionally, it was reported that Kafka has a higher latency in comparison to two other frameworks but it is consistent [15].

2.4. LOCALITY SENSITIVE HASHING (LSH)

Locality Sensitive Hashing is a technique that is used to hash similar items into the same bucket [26]. The intuition behind this technique is that similar items are more likely to be hashed into the same bucket than dissimilar items. Figure 2.6 provides an example of items being to hashed their corresponding buckets by LSH. To hash the items, one or multiple hash function can be used to create the hash for the items. Unlike cryptographic hash functions, the goal of the hash functions that are used for LSH is not to minimise the number of collisions but rather to maximise the number of collisions. LSH is a technique that can also be used to cluster items or do a nearest-neighbour search.

By using hash functions that maximises the number of collisions, it increases the probability that similar items are more likely to be hashed to the same buckets. While this is the property that we want to have with LSH, it also means that there is a probability that items are put into buckets in which they are not similar to any of the other items within the same bucket. These are considered as false positives. Additionally, there is a probability that similar items might get different hashes and be put into different buckets. These are then considered as false negatives. An example illustration of a false positive and a false negative is show in Figure 2.7 and Figure 2.8 respectively. In these two examples, we can consider the first two vectors (v_1 and v_2) similar based on their Euclidean distance. Similar can be said for vectors v_3 and v_4. The hash for each vector v_i (where $i = 1,2,3,4$) is computed as follow: for each random vector, the dot product is computed between the random vector and the vector v_i. If the dot product is positive, then we get a 1 else we get a 0.

The number of buckets that are used in LSH also have an effect on how the number of collisions that can occur. If the number of buckets is equal to the number of items or even larger, then the probability for a collision is very low as each item can be hashed to its own bucket. If the number of buckets is too small, then many collisions might occur but this again could lead to the problem of getting many false positives. One way to find the right number of buckets to use is to do an empirical evaluation and see which number of buckets would give the lowest number of false positives.

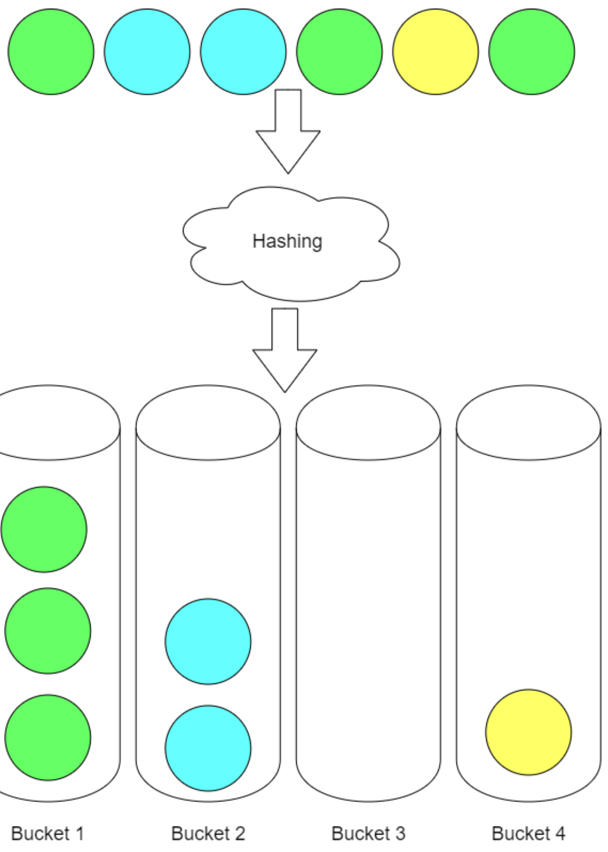


Figure 2.6: An example of items being hashed to their corresponding buckets by LSH.

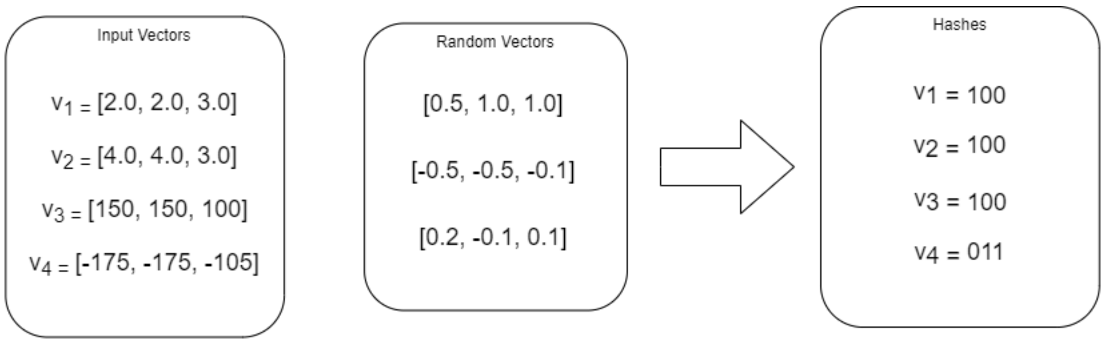


Figure 2.7: An example of a false positive in LSH. Vector v_1 is similar to vector v_2 and vector v_3 is similar to vector v_4. We expect that vectors v_1 and v_2 should get a same hash, and vectors v_3 and v_4 should get the same hash. From the final hashing results, we see that vectors v_1, v_2 and v_3 got the same hash but vector v_3 is not similar to v_1 nor v_2. Therefore vector v_3 is a false positive.

2.4.1. LSH HASHING METHODS

There exist different hashing methods that can be used for LSH [26]. We provide a description of three existing methods in the following paragraphs.

MINHASHING

MinHashing is a method that can be used to compute the similarity between two sets of elements/items. There is a universal set that contains all the elements/item that can occur and there exist several subsets of

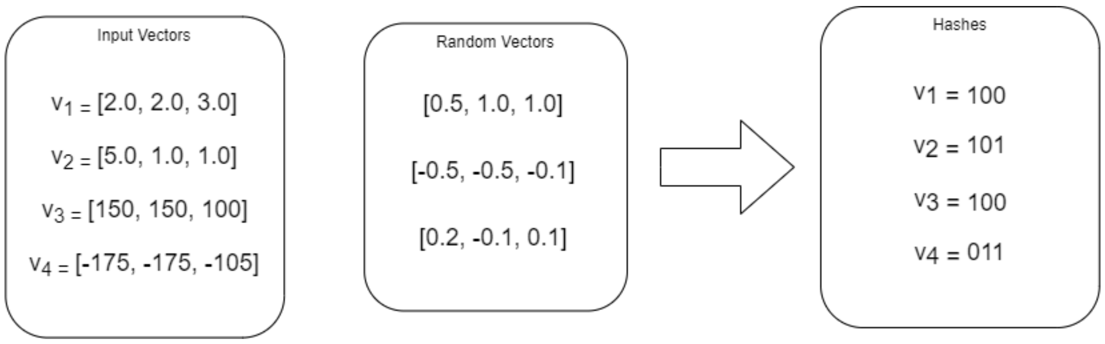


Figure 2.8: An example of a false negative in LSH. Again here vector v_1 is similar to vector v_2, and vector v_3 is similar to vector v_4. We expect to see that the vectors that are similar to each other should get the same hash. From the final hashing results, we see that vectors v_2 and v_3 a different hash than the one that we expected them to get. Vectors v_2 and v_3 are false negatives.

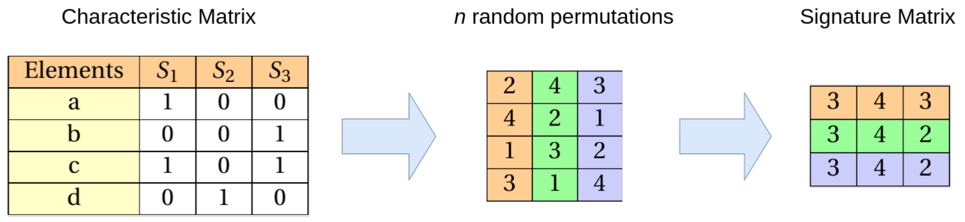
this universal set. For this method, two different matrices would need to be constructed [26].

The first matrix (also known as the characteristic matrix) defines which elements exist within a particular subset. Each row of this matrix represents an element from the universal set and each column represents a subset. A 1 would be put in a cell of the characteristic matrix if the element exists in the corresponding subset. A 0 would be put in a cell if the element does not exist in a particular subset. An example of a characteristic matrix is shown in Figure 2.9.

Elements	S_1	S_2	S_3
a	1	0	0
b	0	0	1
c	1	0	1
d	0	1	0

Figure 2.9: An example characteristic table for MinHashing. In this example, there is a universal set $U = \{a, b, c, d\}$ and three subsets, namely $S_1 = \{a, c\}, S_2 = \{d\}$ and $S_3 = \{b, c\}$.

The second matrix (also known as the signature matrix) is constructed as follow: First, n permutations is computed on the rows of the characteristic matrix. Then the entries of each row of the signature matrix are computed by following the order of a permutation and finding the first 1 that occurs in a particular column. An example of this process is shown in Figure 2.10.



Elements	S_1	S_2	S_3
a	1	0	0
b	0	0	1
c	1	0	1
d	0	1	0

2	4	3
4	2	1
1	3	2
3	1	4

3	4	3
3	4	2
3	4	2

Figure 2.10: An example characteristic table for MinHashing. In this example, there is a universal set $U = \{a, b, c, d\}$ and three subsets, namely $S_1 = \{a, c\}, S_2 = \{d\}$ and $S_3 = \{b, c\}$.

The signature matrix can then be further divided into b number of bands, where each band has its own hash function that is used to hash the signatures into different buckets [26].

HAMMING DISTANCE

The Hamming distance can be used to compute how similar two elements are based on their binary representation. To do so, one would need to have a hash function that maps each element from one space to the binary space i.e. each element will be transformed into a binary vector [26]. With this binary representation, one can compute the Hamming distance between elements. When the Hamming distance between two binary vectors is zero, then these two elements are the same. Let us do a simple example of computing the Hamming distances between four items. First, we use a simple hash function that takes the decimal value of an item and compute its binary value. This is show in Figure 2.11. Then we compute the Hamming distances between the four items and the results are shown in Table 2.2. When we look at the hamming distances between the four items, we see that X_1 is similar to X_2 and X_3 is similar to X_4.

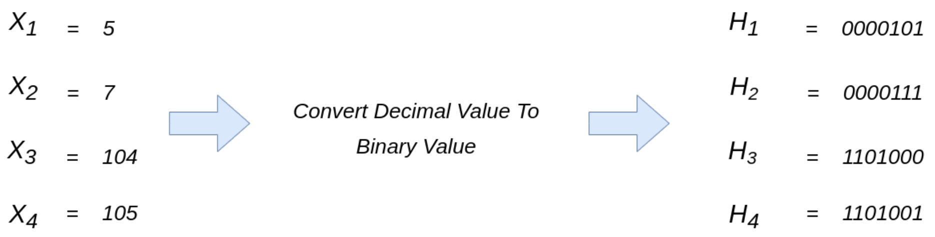


Figure 2.11: Example of hamming distance: computing the binary representation of the items.

Binary Values	0000101	0000111	1101000	1101001
0000101	0	1	5	4
0000111	1	0	6	5
1101000	5	6	0	1
1101001	4	5	1	0

Table 2.2: Hamming distances between the four items.

RANDOM HYPERPLANES

Random Hyperplanes (also known as Random Projections) is a method in which the space where the elements are located, is separated into different regions by drawing hyperplanes through that space [26]. The intuition behind this method is the similar items will be put into the same region and dissimilar items will be put into different regions. A visual representation of this of a space that is separated by random hyperplanes is shown in Figure 2.12

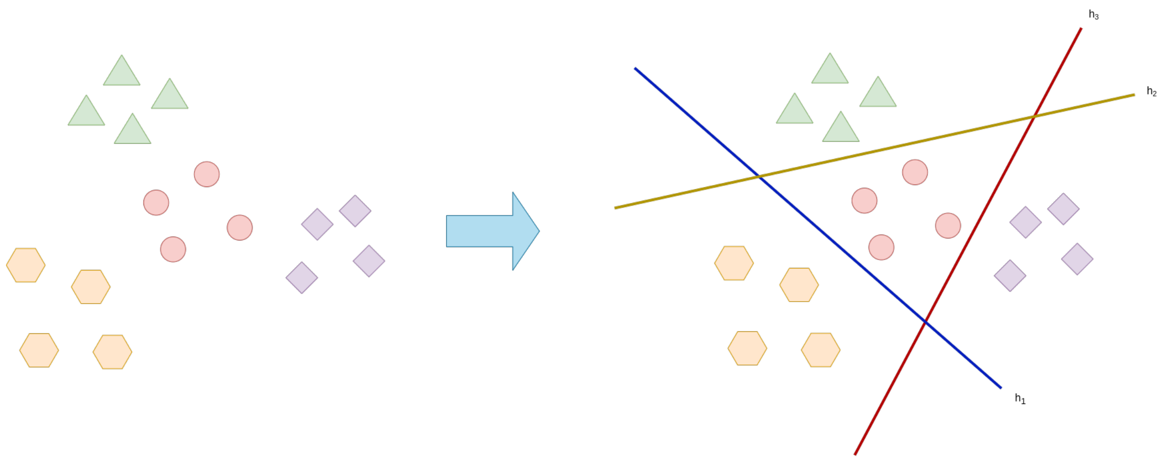


Figure 2.12: An example of a space that is separated into different regions by three different hyperplanes.

For an item to be put within a particular region of the space, we would need to compute the hash of the item. Before the start of the computation, we define a list of vectors, denoted as $L_{vectors}$. These vectors contain random numerical values that can be either negative or positive. This list of vectors can be seen as a matrix, where each row represents one vector. The vectors are created only once and will not be changed. The hash of an item is then computed as follow:

- The item is transformed into a vector v_{item} that contains numerical values.
- The dot product is computed between v_{item} and each random vector in the $L_{vectors}$.

- If the dot product between v_{item} is negative, a 0 is returned. If the dot product is zero or higher, then a 1 is returned.
- The zeroes and ones (that were returned from the dot product computations) are then concatenated to form one single bitstring, which is used to represent the hash of the item [26] and to determine in which bucket the item will be put into.

An small example of the hashing process is shown in Figure 2.13. In this example $L_{vectors} = \{\{-1,1,-2,2\},\{1,5,3,7\},\{-1,-2,3,-8\},\{0,5,-1,3\}\}$ and $v_{item} = \{1,6,2,2\}$. Based on the dot product values, we get the 1101 as the hash for the item.

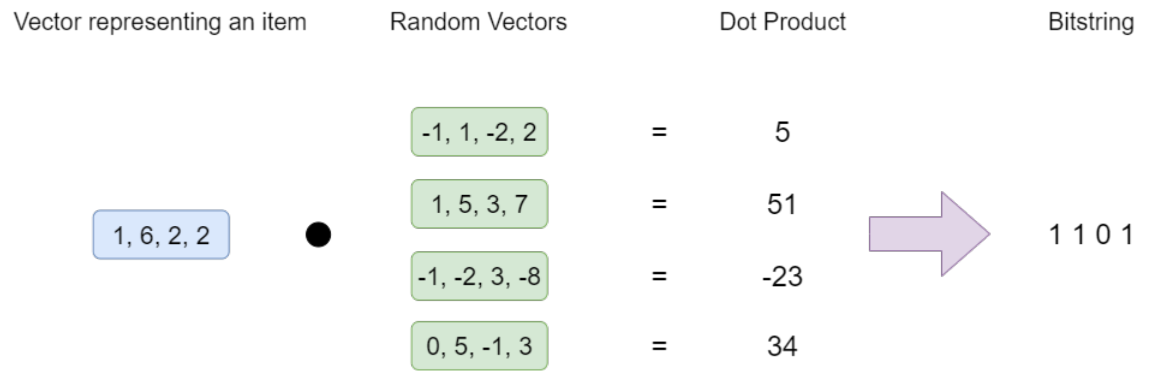


Figure 2.13: A small example of the hashing process that is done in the Random Hyperplanes method.

2.5. RELATED WORK

When it comes to the design of an anomaly detection method, there is no best way to do it. As mentioned in Chapter 1, two important points should be taken into account when designing an anomaly detection method: the method should not produce a lot of false positives and an alert should be triggered as soon as an intrusion has been detected. Several solutions have been proposed, each using a different method to detect anomalies. We discuss a few of these solutions in this section.

2.5.1. SOLUTION NOT UTILISING STATE MACHINES

BREURKES ET AL.

Breurkes et al. have proposed a solution that detects anomalies by using two main methods: comparing the distribution of log data and checking whether there is unusual behaviour in the Markov models that were created for the processes [5]. The intuition behind these two methods is that the distributions of the log data and Markov models of the processes should be similar when the system is behaving as it should be. When a large difference is detected in the distributions or models, then this might be a signal that the system is doing something unusual. This work differs from ours as no state machines are learned from the log data, Markov models are learned instead.

KORTEBI ET AL

Kortebi et al. have proposed another solution to detect anomalies using what they call the "FlowMon Traffic Monitoring" approach. In this approach, network packets are collected and then the statistics of the IP flows are calculated. These statistics are used later for anomaly detection [6]. The solution is more efficient when compared to methods that monitor traffic based on full packet capture. Additionally, the solution is also compatible with encrypted traffic. It was shown through an experiment that the solution can be used on a home network and that it could bring benefits to different actors: users of the home network and Internet Service Provider (ISP). This work also differs from ours as no state machines are learned from the network packets.

2.5.2. Solutions Utilising State Machines

Pellegrino et al

Pellegrino et al. have proposed their solution, BASTA, which uses probabilistic real-time automata to create fingerprints from network traffic streams of the host [8]. In this solution, the focus in on the timed events because of time information is an important property that can be used to characterise network traffic. Two strategies are used for finding the same infection on a host: error-based and fingerprint-based. In the error based approach, a new host is compared with a known malicious host and it is checked how many symptoms does a new host have in common with a malicious host. In the fingerprint approach, uses a known malicious dataset to create fingerprints of the distinguishing malicious symptoms. It is then checked at a new host whether they have these particular symptoms. From the experiments, it was found that the error-based method works better on a dataset that does not have any noise and the fingerprint-based method works better on a noisy dataset. Additionally, it was shown that BASTA has a high performance, even in experiments with a difficult setting. Though this work does learn state machines from a stream of data, it does not use Blue-Fringe algorithm to learn the state machine and the machines are not learned on an embedded device.

Umadevi et al

Umadevi et al. have proposed a solution that utilises timed automata to detect vulnerabilities in Cyber Physical Systems (CPS) [11]. A CPS is a distributed system that consists of multiple processes. Each process reports their state to the set of control devices. Based on the state of the processes, the control devices decide on the action that needs to be executed. In this work, the authors have put their focus on the Secured water treatment (SWaT) system. The system is divided into six stages and for each state, a Non-deterministic Timed Automaton (NTA) is learned. These NTAs model the interactions between the different modules (within a particular stage), the sensors and actuators. What is different in this research in comparison to other works, is that automaton is not used to detect anomalies in the system but to detect possible vulnerabilities in a system. It seems that this solution is more of a preventive solution than a reactive solution. This work differs from ours as the method that was designed by Schouten is a reactive solution.

Schouten

As mentioned before, Schouten has proposed a new method that learns state machines in real-time from a stream of network traffic data and these state machines are used to detect anomalies in a network [25]. This method uses an adapted version of the Blue-Fringe algorithm to learn the smallest state machines. In contrast to the original Blue-Fringe algorithm, a full PTA is not learned before the merging occurs; sketches and approximations are used to create a PTA and stop the learning process. The solution is implemented using Apache Flink and Apache Kafka to process the stream of network traffic data and learn the corresponding state machines. The work that is done by Schouten forms the basis for our work.

Solutions on Embedded Devices

Ezeme et al

Ezeme et al. have proposed a solution that can detect anomalies on real-time embedded systems [14]. This solution is designed to not just be used on the application layer but also other layers of the system. It uses a model that creates a hierarchy in which temporal relationships is learned among events that occur in the system. Additionally, the impact that each feature has on other features. With this information, the model can then filter out irrelevant information. The model that is learned can then be used to detect anomalies that occur within the system. The events can be gathered from the execution traces of the system. The solution uses a semi-supervised learning method that is based on the closed world approach. Such learning method can be used because you know exactly what the standard behaviour is of the system. Though it has been shown that this solution has better performance in comparison to two other solutions, there is an assumption that one would need to have a system with well-defined behaviour. It is questionable what is considered as a system that has a well-defined behaviour. This work differs from ours as we do not create a hierarchy from the NetFlow data. Also, no state machines are learned using this method.

Sforzin et al

Sforzin et al. have investigated in their work whether it is possible to run an IDS on an embedded device [17]. The goal is not to detect intrusions or anomalies that occur on the device but rather on a network. Thus the device monitors the network and checks whether an attacker has infiltrated the network. The authors particularly selected the Raspberry Pi 2 Model B for their research and installed a well-known open-source

IDS, Snort [42]. The performance is evaluated based on four different metrics: average CPU usage, average RAM usage, packet capture rate and the number of alerts generated by the Snort. Due to the wide variety of network traffic, multiple trace files were used to represent different types of network traffic. Additionally, Snort is run using different rulesets to observe the influence of the rulesets on the performance of the device. From the results of the experiments, the performance does drop as the more rules are used and more packets are replayed but the memory usage of the device never reached to 100%. This work does not learn any state machines from the data but it does show that it is possible to run an IDS on a Raspberry Pi 2 Model B.

NYKVIST & LARSSON

This work is somewhat similar to the work that is done in this thesis. Nykvist and Larsson have proposed an anomaly detection method that utilises pattern matching algorithms: Aho-Corasick algorithm [43] (AC-algorithm) and Knuth-Morris-Pratt algorithm [44] (KMP-algorithm). Additionally, R-trees [45] are used to filter on particular rules in a given ruleset. Initially, all the rules from a ruleset are parsed and put into an R-tree. This makes searching for a particular rule quicker. Then state-machines are created for each of the algorithms if there is a rule that has the "content" parameter. The state-machines are used for pattern matching between packets. When a match has been found, an alert is triggered. The authors did a performance analysis by running the two algorithms and Snort on two different devices: Raspberry Pi 3 Model B+ and WiFi Pineapple [46]. From the results, they have found the following: First of all, the AC algorithm has a better performance on large rulesets. Second of all, it was shown that the Wifi-Pineapple is not suitable to act as an IDS due to the limited maximum throughput of the device. The device starts dropping packets once it has reached the limit. Finally, when comparing the two algorithms to Snort the two algorithms performs better than Snort in specific conditions. One thing should be noted from this work; though it is similar to the work that is done in this thesis, this solution does not learn state machines from a stream of network traffic data. The state-machine models are created based on the rules from a given ruleset. This sets our work apart from the work that is done by Nykvist and Larsson.

3
Methodology

This chapter describes our approach on how we will utilise LSH for the learning of state machines and the clustering of states/state-machines. We first describe in Section 3.1 the hashing method that we will select to use in our LSH approach. Then we describe in Section 3.2 how we will utilise LSH to hash the states, that are derived from the stream of network traffic data, into buckets that correspond to their hashes. These buckets are then used for the state-merging process. In Section 3.3, we describe the approach on how we evaluate the usage of LSH for the state-merging process. Section 3.4 describes how we will utilise LSH to hash the state machines into buckets. Each of these buckets can be seen as a cluster of state machines and can be used for finding similar state machines. In Section 3.5, we describe our approach on how we will evaluate the usage of LSH for the clustering of state machines. Finally in Section 3.6, we describe how we will evaluate whether it is possible to learn state machines in real-time on an embedded device.

3.1. Selecting the Hashing Method For Our LSH Approach

Before we can start hashing the states/state machines into buckets, we need to select one hashing method that we will use in our LSH approach. From the three different hashing methods that we have described in Subsection 2.4.1, we have decided to use the random hyperplanes as the method for hashing of the states and state machines. Our choice was made based on the following two reasons:

1. Using the MinHashing method requires us to store two different matrices and it also requires us to store the n permutations on the rows of the characteristic matrix. Let us first analyse the space complexity of the MinHashing method. Let e denote the number of elements in the universal set U and s denote the number of subsets that can be created from the universal set U. The size of the characteristic matrix is equal to $e \times s$, which means that the space complexity of the characteristic matrix is $O(e \times s)$. The n random permutations can be seen as a list of n vectors and each vector has the length that is equal to e. Thus the space complexity of the n random permutations is $O(n \times e)$. The size of the signature matrix is equal to $n \times s$ and thus the space complexity of the signature matrix is $O(n \times s)$. The total space complexity of the MinHashing method is then $O(e \cdot s + e \cdot n + n \cdot s)$.

   For the Random Hyperplanes method, we only need to store a list of n vectors, where each vector has a length of e. Thus the space complexity of the Random Hyperplanes method equals to $O(e \times n)$. Essentially, we will only need to store one single matrix for the Random Hyperplanes method. As we will be learning state machines on an embedded device, there is less space available for us to use. We have to reduce the space that is used by our method and therefore we have decided to not use the MinHashing method as it uses more space than the Random Hyperplanes approach.

2. In comparison to MinHashing, there is no need to keep track of different matrices if we use the Hamming Distance method. We only need one or multiple hash functions that can map each state/state machine to a bitstring and then use these bitstrings to compute the similarity. Though we do not have to keep track of different matrices using this method, we still would have to do the pair-wise comparisons of the bitstrings to find the two most similar states/state machines. Recall that the pair-wise comparisons approach has the time complexity of $O(n^2)$. This pair-wise comparison of bitstrings is no different than the original method for finding a state for a merge. Our goal for using LSH is to avoid the

pair-wise comparisons that need to be done to find a state for a merge. Thus we have decided to not use the Hamming Distance method as the hashing method in our LSH approach.

3.2. USING LSH TO HASH STATES

As elaborated in Subsection 1.2.1 and illustrated on Figure 1.1, when looking for a most similar state for a merge we would need to traverse through all states to find a most similar state. The pseudocode of this process is shown in Algorithm 1. As this operation has a quadratic run-time, it will have an impact on the performance when the number of states is very large. Our approach is to use LSH to hash similar states into the same buckets. With this approach, there is no need to traverse through all states to find a most similar state for a merge. The pseudocode for finding a most similar state for a merge using LSH is shown in Algorithm 2. The intuition is that similar states would get the same hash and be put within the same bucket. Figure 3.1 provides a high-level illustration on which part of the original method was modified to our approach of using LSH to merge states.

Algorithm 1: Finding a most similar state (original method)

```
Input: A state $s$ and a list $L$ that contains all states besides $s$
mostSimilarState = null
mostSim = 0
for each state $o$ in $L$ do
    similarity = computeSimilarity($s$, $o$)
    if similarity > mostSim then
        mostSimilarState = $o$
        mostSim = similarity
    end
end
if mostSim > COSINE_SIMILARITY then
    return mostSimilarState
end
return null
```

Algorithm 2: Finding a most similar state (LSH)

```
Input: A state $s$ and the buckets $B$
mostSimilarState = null
mostSim = 0
bucket = getBucket($s$, $B$)
for each state $o$ in bucket do
    similarity = computeSimilarity($s$, $o$)
    if similarity > mostSim then
        mostSimilarState = $o$ mostSim = similarity
    end
end
if mostSim > COSINE_SIMILARITY then
    return mostSimilarState
end
return null
```

3.2.1. STATE REPRESENTATION FOR HASHING

As mentioned in the description of Random Hyperplanes, we need a vector representation of the state to be able to hash the state. The question is how we can transform a state into a vector that contains numerical values. Our answer to this question lies in the current implementation of a state. Each state contains a Count-Min sketch vector, which contains an approximate count of the sequences that have occurred in that state. This vector is used to compare different states and check whether two states can be merged based on their Count-Min sketch vector [25].

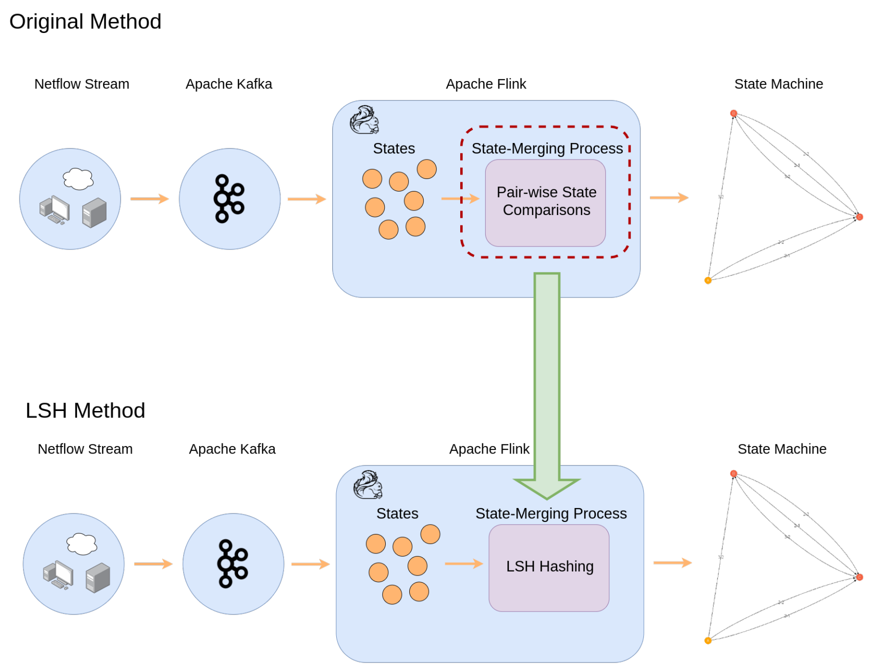


Figure 3.1: A high-Level illustration that shows which part of the original method was modified to the usage of LSH. In the state-merging process of the original method, pair-wise comparisons are done to find the most similar state for a merge. In our approach, LSH is used to hash the states into different buckets and these are used to find the most similar state for a merge.

In the current implementation, the Count-Min sketch vectors of the states are used to compute the cosine similarity between states [25]. The state that has the highest similarity value will then be used for a merge. As the Count-Min sketch vector of a state contains numerical values and they can be used to decide whether two states should be merged, it seemed natural to also use the Count-Min sketch vectors as the vector representation of a state. The Count-Min sketch vector of a state is then used to compute the bitstring(hash) of the state.

3.3. EVALUATING THE HASHING OF STATES

To assess whether LSH works well for the clustering and eventually the merging of states, we have set up three different experiments. A description of each experiment is given in the following subsections.

3.3.1. ASSESSING THE ITEMS IN THE BUCKETS

In this experiment, we want to evaluate how well the similar states are clustered together in the buckets. For this experiment, we randomly generate vectors which will be used to represent the Count-Min sketch vectors of the states. Thus in this experiment, no states from actual state machines will be used. We hash all of these vectors using LSH into buckets and each bucket is used to represent a cluster of states. But how do we know whether the state is clustered correctly? Our answer to this question is to compare the clustering results of LSH to two other methods of clustering: KMeans Clustering and Random Clustering. In KMeans Clustering, centroids are used in order to assign items to their cluster. Items are assigned to a cluster with the closest centroid. In Random Clustering, each vector will be randomly assigned to a bucket and each bucket has an equal probability of being chosen. In this experiment, we assume that KMeans clustering will do a better job at clustering the vectors than our LSH method. The results of KMeans clustering thus serve as the baseline of this experiment. The results of Random clustering are used to assess whether LSH is just as bad or worse than random guessing.

VISUALLY COMPARING THE CLUSTERING RESULTS

To compare the clustering results between the three different clustering methods, we will visually compare the clustering results. This can be done by computing the similarities between the vectors that we have generated. As the original method uses the cosine similarity to find a most similar state for a merge, we will also use cosine similarity to compute the similarities between the vectors.

Knowing the similarities between the vectors, we can construct a similarity matrix. This matrix can be used to produce a heatmap in which it can show the relations between the vectors. To see the clusters better in the heatmap, we will sort the rows and columns based on the cluster-ID. Doing so, it allows us to group the vectors that that are put into the same cluster together and we can visually assess whether vectors are put into the wrong clusters. The procedure for creating the similarity matrix and eventually the heatmap is shown in Algorithm 3

Algorithm 3: Computing the similarity matrix and creation of the heatmap

```
Input: A list $L$ containing all vectors
similarity_matrix = create a matrix of size $\|L\| \times \|L\|$
// Compute the similarity between the vectors and store it in a matrix
for each vector $v_1$ in $L$ do
    for each vector $v_2$ in $L$ do
        if $v_1 = v_2$ then
            similarity_matrix[ $v_1$][$v_1$] = 0.0
        else
            similarity_matrix[ $v_1$][$v_2$] = cosineSimilarity($v_1$, $v_2$)
        end
    end
end
// Sort the matrix for the creation of the heatmap
sortRows(similarity_matrix)
sortColumns(similarity_matrix)
// Create the heatmap using the sorted matrix
createHeatMap(similarity_matrix)
```

METRICS TO ASSESS THE QUALITY OF THE CLUSTERS

Besides visually comparing the clustering results between the methods, we also want to assess and compare the quality of the clusters between the methods; only looking at the heatmap does not tell us enough regarding the quality of the clusters. To assess the quality of the clusters, we need metrics that can tell us about the quality of the clusters that were formed by the methods. For this experiment, we have chosen two metrics that we will use the assess the quality of the clusters. We describe each metric in the following paragraphs.

As we will cluster the vectors using multiple numbers of clusters and we do not know which is the best label to give to each vector, we have chosen to use the Silhouette Coefficient score. This is a metric that is used to assess whether the clusters are well defined and it should be used when the ground truth labels of the clusters are not known. The score tells us how well each item fits within the cluster that it has been assigned to and how different it is to the other items from the other clusters. The higher the score, the better the clusters are defined. The highest score that can be achieved for this metric is 1 and the lowest -1. A score that is close to 1 means that the clusters are well defined and score that is close to -1 means that the items are incorrectly clustered. A score that is close to zero means that there are overlapping clusters [47]. Let i be an item from a cluster c. The score $s(i)$ is computed using the formula that is shown in equation 3.1, where $a(i)$ and $b(i)$ are defined as follows [47]:

- $a(i)$ is the average distance between an item i and all other items within the same cluster c.
- $b(i)$ is the average distance between an item i and all other items that are from the closest neighbouring cluster c', where $c \neq c'$.

$$s(i) = \frac{b(i) - a(i)}{max(a(i), b(i))} \tag{3.1}$$

Additionally, we will also compute the average accuracy of each clustering method by checking whether the closest neighbour of a given vector is also put within the same cluster as the given vector. We have chosen to compute the accuracy in this manner as our goal for using LSH is to quickly find a most similar state for a merge. This is essentially looking for the closest neighbour. The pseudocode for calculating the accuracy of the clustering method is shown in Algorithm 4.

Algorithm 4: Computing the accuracy for a clustering method

```
Input: A list $L$ containing all vectors
correct = 0
wrong = 0
for each vector $v$ in $L$ do
    closestVector = getClosestVector($v$)
    cluster = getVectorCluster($v$)
    if cluster contains closestVector then
        correct++
    else
        wrong++
    end
end
return correct ÷ (correct + wrong)
```

3.3.2. Assessing the State Machines Learned Using LSH

In this experiment, we want to evaluate the difference between the state machines that are learned using the original method that was written by Schouten and the state machines that are learned using our LSH method. Thus the state machines that are learned using Schouten's method serves as the ground truth for this experiment. To compare how much our state machines differ from the ground truth state machines, we compute the difference between the state machines. To be able to compute the difference between the state machines, we need metrics that can tell us how much a pair of state machines differ from each other. In the following paragraph, we provide a description of the metrics that we have chosen to use to compute the difference between the state machines.

Metrics to Assess the Difference Between State Machines

We have chosen to use the Kullback-Leibler divergence (KL-Divergence) and Perplexity as the metrics to measure the difference between the state machines. KL-Divergence is a metric that measures how much one model differs from another [25, 27]. The formula is shown in equation 3.2, where p is the true model, q is the predicted model and A is the alphabet that is shared by both models. The KL-Divergence returns values from $[0,\infty)$. The larger the value, the larger the difference between the two models. A value of zero means that the two models are identical.

$$KL-Divergence(p,q) = \sum_{x\in A} p(x)\cdot log_2 \frac{p(x)}{q(x)} \tag{3.2}$$

Perplexity is a metric that measures how similar two models are to each other. The formula is shown in equation 3.3, where again p is the true model and q is the predicted model and A is the alphabet that is shared between the two models [25]. A lower perplexity value means a higher similarity between the two models.

$$Perplexity(p,q) = 2^{-\sum_{x\in A} p(x)\cdot log_2 q(x)} \tag{3.3}$$

For this experiment, we will only use the original method for comparison and do not consider other state machine learning methods. The motivation behind this choice is that we have a very specific use case of learning state machines in real-time on a small embedded device. We are interested in how well we can learn state machines using LSH and compare it to the original method. The state machines that are learned using LSH should be just as good the ones that are learned using the original method. By also doing a comparison with other methods, it means that we would need to modify them such that they are compatible with the current implementation of the system and this is out of the scope of this research. Additionally, learning a small and consistent state machine from the given training data is an NP-Hard Problem [29]. This means that each state-machine learning algorithm would learn a state machine with a different number of states and it is then difficult to tell which state machine is the best.

3.3.3. Run-time Analysis of the State Merging Process

In this experiment, we want to evaluate whether we gain an improvement in the run-time of the state-merging process by using LSH. After all, the goal of using LSH to cluster the states is to improve the run-time of the state-merging process. We divide this experiment into two parts:

1. In the first part, we will use artificially generated vectors to represent the states that will be used for merging. The state-merging process is run in isolation for both methods and we measure the time that it takes to find a most similar state using each method. We then compare the run-time between the two methods. This part of the experiment is to show on an abstract level the performance improvements that we can gain using LSH.
2. The second part of this experiment is quite similar to the first part but instead of running the state-merging process in isolation with artificially generated data, we will use the whole system that was designed by Schouten together with artificially generated NetFlow data. We again measure and compare the run-time between the two methods. This part of the experiment is to show whether we gain an improvement on the run-time of the state-merging process by using our approach in the actual system itself.

3.4. Using LSH to State Machines

One interesting extension to the usage of LSH is to investigate whether we can use LSH to cluster state machines. The intuition behind the hashing of states can also be applied to the hashing of state machines; similar state machines should be hashed into the same buckets. Each bucket then describes state machines that have similar behaviour.

In the original implementation, clustering of state machines was not implemented. To find a similar state machine, we would need to do a pair-wise comparison between all state machines. This process is the same as when we are searching for a most similar state for a merge. The run-time of this process is quadratic and its performance will be impacted as the number of state machines grow larger. By clustering the similar state machines together using LSH, we can find a most similar state machine in a quicker manner. There is no need for us to all the pair-wise comparisons between all the state machines to find a most similar state machine. Figure 3.2 provides a high-level illustration on which part of the original method was modified to our approach of using LSH to cluster similar state machines together.

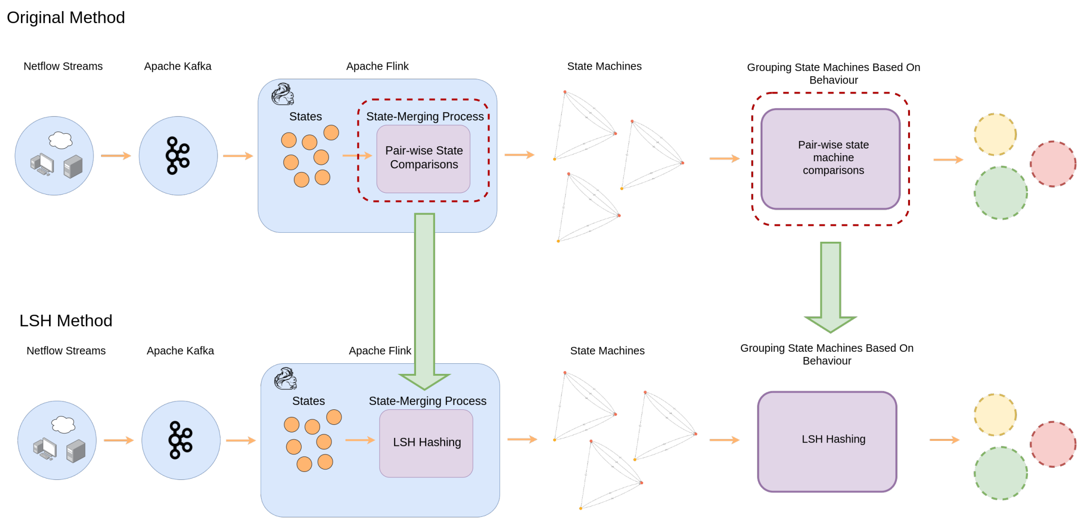


Figure 3.2: A high-Level illustration that shows which part of the original method was modified to the usage of LSH. In our approach, LSH is used to hash state machines to different buckets. The intuition is that the state machines that are hashed into the same bucket have similar behaviour.

3.4.1. State Machine Representation for Hashing

Just as with the hashing of the states, we need to transform a state machine into a vector that contains numerical values. The question this time is: how can we transform a state machine into a vector that contains numerical values? It turns out that there is no trivial answer to this question; there is no solution that does not include a large overhead for it to work. We came up with three different methods to transform a state machine into a vector that contains numerical values. We describe each method in the following paragraphs.

USING THE STATE TRANSITION TABLES OF THE STATE MACHINES

For the first method, we use the state transition tables of the state machines. A state transition table shows in which state the state machine will end up, based on the given input and the current state that the state machine is in [28]. A visual representation of the state transition table that we will be using for our approach is shown in Table 3.1. As we are dealing with state machines where each transition has a probability for it to occur and we need a vector that contains numerical values, we have added the probability column in the state transition table. This column is used as the vector representation of the state machine.

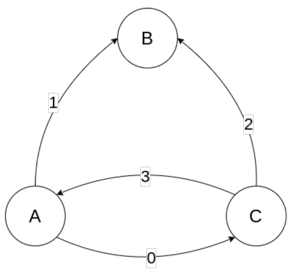
Input	Current State	Next State	Probability
I_1	S_1	S_i	P_a
$\dots$	$\dots$	$\dots$	$\dots$
I_n	S_1	S_j	P_z
I_1	S_2	S'_i	P'_a
$\dots$	$\dots$	$\dots$	$\dots$
I_n	S_2	S'_j	P'_z
$\dots$	$\dots$	$\dots$	$\dots$
I_1	S_m	S''_i	P''_a
$\dots$	$\dots$	$\dots$	$\dots$
I_n	S_m	S''_j	P''_z

Table 3.1: The state transition table that is used in our approach.

Notice that rows that are shown in Table 3.1 are sorted. The rows are first sorted by the state symbols and then by the input symbols. The purpose of sorting the rows is to make sure that we have a generalised table for all state machines so that the size of the table is the same for any arbitrary state machine. The sorting also makes sure that the probability values are filled into the correct cells. This is important for capturing the structure of a state machine. The probability values are filled into the table following the procedure that is shown in Algorithm 5. Figure 3.3 provides an example of how the state transition table is constructed for a state machine.

Algorithm 5: Constructing the vector representation of a state machine using the State Transition Table method

```
Input: A state machine $m$, a sorted list $S$ containing all possible state symbols and a sorted list $I$
       containing the all possible input symbols
vectorRepresentation = new List()
for each symbol $s$ in $S$ do
    if $m$ contains $s$ then
        for each symbol $i$ in $I$ do
            state = m.getState($s$)
            transitions = state.getOutTransitions()
            if transitions contains $i$ then
                vectorRepresentation.add(state.getTransitionProbability($i$))
            else
                vectorRepresentation.add(MIN_PROBABILITY) // add very small probability
            end
        end
    else
        for each symbol $i$ in $I$ do
            vectorRepresentation.add(MIN_PROBABILITY)
        end
    end
end
return vectorRepresentation
```



Input	Probability
0	0.8
1	0.2
2	0.24
3	0.76

Input	Current State	Next State	Probability
0	A	C	0.8
1	A	B	0.2
2	A	-	0.000001
3	A	-	0.000001
0	B	-	0.000001
1	B	-	0.000001
2	B	-	0.000001
3	B	-	0.000001
0	C	-	0.000001
1	C	-	0.000001
2	C	B	0.24
3	C	A	0.76

Figure 3.3: Example showing how the state transition table is constructed for a state machine. In this example, the possible inputs are only the numbers from 0 till 3.

The motivation for using this representation is based on the assumption that if two state machines have the same transition on the same state for a given input, then a value that is higher than the minimum probability is filled into the corresponding cell for both state machines. If this is not the case, then the minimum probability is filled in for the state machine that does not have the transition and a value that is higher than the minimum probability is filled in for the other state machine.

If two state machines have a large difference in the values that are assigned to the same cells of the column, then this would produce two different dot products and thus also produce two different hashes for the state machines. This is a property that we want to capture as our goal of using LSH is to cluster similar state machines together.

ADVANTAGE & DISADVANTAGE OF USING STATE TRANSITION TABLE METHOD

The advantage of using the state transition table method is that we are using the structure of a state machine to create its vector representation. Thus we have a method that can map the structure of the state machine to a single vector of numerical values. This method provides a way for us to cluster state machines based on their structure.

The disadvantage of this method is that we need to have a generalised state transition table that contains all possible state symbols and all possible input symbols. As it is not always true that all state machines have the same states and transitions, this method can create very sparse vectors. These sparse vectors cause us to use more space than needed to store information on the structure of a state machine. Space consumption is important for us as our goal is to run our method on an embedded device that has less resource than a desktop computer or laptop.

USING THE LANGUAGE OF THE STATE MACHINES

In the second method, we use the language of the state machine to create the vector representation of the state machine. This is done by first generating a fixed set of strings from the language of a fixed set of state machines. We opted for the generation of a set of string as it would represent the behaviour of a state machine better than a single string. Each set will be run/simulated on each state machine and we record the average probability of each set. Algorithm 6 shows the procedure for generating the vector representation of a state machine from the fixed set of strings. The average probabilities are then used to construct the vector that would represent the state machine. Figure 3.4 provides an example of how the vector representations are created for multiple state machines using their language.

Algorithm 6: Constructing the vector representation of a state machine using its language

```
Input: A list $L$ contains different sets of strings and a state machine $m$
vectorRepresentation = new List()
for each set $s$ in $L$ do
    probabilities = new List()
    for each string $str$ in $s$ do
        probability = similateString(str, m)
        probabilities.add(probability)
    end
    vectorRepresentation.add(mean(probabilities))
end
return vectorRepresentation
```

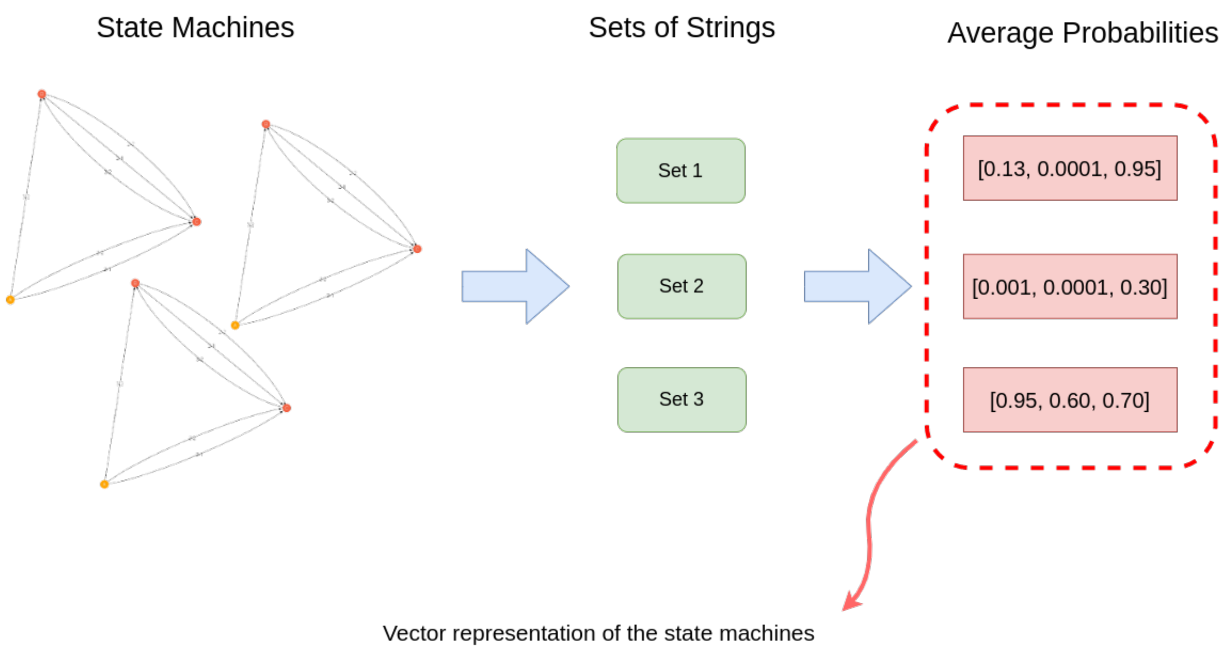


Figure 3.4: A high-level visualisation that shows the process of creating the vectors that are used to represent the state machines.

Unlike the first method, this method tries to capture the behaviour of a state machine by using its language and not the structure of the state machine. The intuition behind this method is that if two state machines have similar behaviour, then the probability for a string to occur in both state machines would be close to each other. On the other hand, if two state machines have different behaviour, then the probability for a string to occur in both state machine would be different; for one state machine, there is a high probability for the string to occur and for the other, the probability would be close to zero. This property helps put state machines in clusters.

ADVANTAGE & DISADVANTAGE OF USING THE LANGUAGE OF THE STATE MACHINE

The advantage of using the language of a state machine to create its vector representation is that we do not have to generate large sparse vector. In this method, we do not have to generalise anything to make sure that the method works for any arbitrary state machine. this solution uses less space to store the vector representation of a state machine.

The disadvantage of using this method is that we need state machines for which we can generate the fixed sets of strings. These strings are needed to create the vector representation of the state machines. The strings can be seen as fingerprints that will be used to match state machines. One thing that should be noticed from this method is that if we would like to introduce fingerprints for matching, we need to generate more sets of strings from state machines. This means that the sets of strings we have, the more space we will be using to store the sets of strings.

USING THE DISTRIBUTION OF THE STATES OVER THE BUCKETS

For the third method, we use the distribution of the states over the buckets to create the vector representation of a state machine. This method arose from the hashing of the states for the state-merging process. As the states of a state machine are hashed into different buckets, we can use the distributions of the states over the buckets to represent a state machine. Under the assumption that similar state machines would have a similar distribution of the states over the buckets, we can use the distribution to cluster state machines. Algorithm 7 shows the procedure on how the vector representation of a state machine is created using the distribution of its states over the buckets. Figure 3.5 provides an example illustration on how the vector representation of a state machine is created using the distribution of its states over the buckets.

Algorithm 7: Constructing the vector representation of a state machine using the the distribution of its states over the buckets

```
Input: A list B containing the buckets sorted by their hash ID
vectorRepresentation = new List()
for each bucket b in B do
    vectorRepresentation.add(bucket.getSize())
end
return vectorRepresentation
```

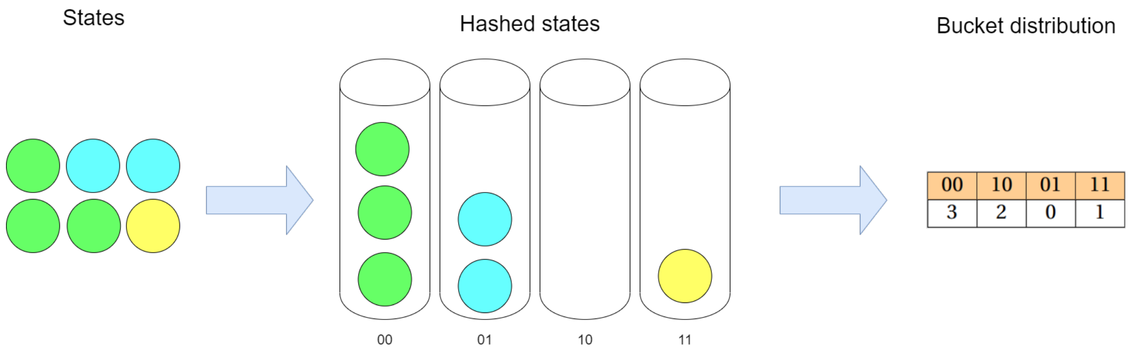


Figure 3.5: Example illustration that shows how the distribution of the states over the buckets is computed for a state machine.

ADVANTAGE & DISADVANTAGE OF USING THE DISTRIBUTION OF THE STATES OVER THE BUCKETS

The advantage of using this method is we do not have to construct any sparse vectors nor do we have to generate and simulate strings on state machines to create their vector representation. As we have already hashed the states of the state machines, we can use the buckets to create its vector representation. With this method, we consume less space as we do not have to store sets of strings and we store vectors that are smaller than the large sparse vectors from the first method.

The disadvantage of using this method is that the way how the states are put into their buckets are arbitrary. Our LSH method only makes sure that similar states get hashed into the same bucket with a high probability. LSH does not make sure that a state will always get the same hash because random vectors are used for the computation of the dot product. This means that the distribution of the states over the buckets is arbitrary for every state machine and may not show any relation regarding the similarity of state machines. The locality-sensitive properties of the states may therefore not carry over to the state machines and this can cause LSH to cluster the state machines incorrectly.

3.5. EVALUATING THE HASHING OF STATES MACHINES

To assess whether LSH works well for the clustering of state machines, we have set up four different experiments. Just as with the hashing of states, we will compare the clustering results of LSH to the clustering results of KMeans Clustering and Random Clustering. A description of each experiment is given in the following subsections.

3.5.1. ASSESSING THE USAGE OF STATE TRANSITION TABLES

In this experiment, we want to evaluate how well LSH can cluster state machines using the state transition method. The state machines that we will be using for this experiment are the ones that were generated for the PAutomaC Competition. The competition provides a comparison between the state-of-the-art techniques for learning state machines, in which one can see which technique performs the best based on a particular setting [48].

We selected five state machines that have the same alphabet and we manually made modifications on the state machines to create ten variants for each state machine. These variants are used to evaluate how well state machines are clustered using our method. The intuition is that the variants should be hashed into the same bucket as their base state machine.

In our experiment, the vector representations of the state machines are created following the procedure that is shown in Algorithm 5 and we will use LSH to hash all variants together with their base state machines to buckets. The same state machines will also be clustered using KMeans Clustering and Random Clustering. We will compare the clustering results between the methods using the same procedure and metrics that were mentioned in Subsection 3.3.1; we will visually compare the clustering results using heatmaps and we will assess the quality of the clusters using the Silhouette Coefficient score and the accuracy.

To generate the heatmaps in this experiment, we need to make some modifications to Algorithm 3; we will be using state machines instead of vectors and we will be using the KL-Divergence metric to compute the similarities between the state machines. We will be using the KL-Divergence metric to compute the similarities as we are working with state machines and this metric tells us how different (or how similar) two pair of state machines are. The cosine similarity is a better fit for computing the similarities between vectors but not for computing similarities between state machines. The procedure for creating the heatmaps in this experiment is shown in Algorithm 8. The whole clustering procedure of this experiment is shown in ALgorithm 9.

Algorithm 8: Computing the similarity matrix and creation of the heatmap

```
Input: A list $M$ containing all state machines
similarity_matrix = create a matrix of size $\|M\| \times \|M\|$
// Compute the similarity between the state machines and store it in a matrix
for each model $m_1$ in $M$ do
    for each model $m_2$ in $M$ do
        if $m_1 = m_2$ then
            similarity_matrix[ $m_1$][$m_1$] = 0.0
        else
            similarity_matrix[ $m_1$][$m_2$] = KL-Divergence($m_1$, $m_2$)
        end
    end
end
// Sort the matrix for the creation of the heatmap
sortRows(similarity_matrix)
sortColumns(similarity_matrix)
// Create the heatmap using the sorted matrix
createHeatMap(similarity_matrix)
```

Algorithm 9: Clustering procedure of state machines using the state transition table.

```
Input: A list $M$ containing all state machines
createVectorRepresentations($M$) // ffollowing the procedure of Algorithm 5
for each model $m$ in $M$ do
    clusterModel($m$) // using LSH
end
// Create the similarity matrix and heatmap following the procedure of
   Algorithm 8
createMatrixHeatmap($M$)
```

Furthermore, we also have to modify our accuracy metric for us to use the metric in this experiment. Instead of searching for a most similar vector, we are now searching for a most similar state machine. The modification on the algorithm can be seen in Algorithm 10.

Algorithm 10: Computing the accuracy for a clustering method (for clustering state machines)

```
Input: A list $M$ containing all state machines
correct = 0
wrong = 0
for each model $m$ in $M$ do
    closestModel = getClosestModel($m$)
    cluster = getModelCluster($v$)
    if cluster contains closestModel then
        correct++
    else
        wrong++
    end
end
return correct ÷ (correct + wrong)
```

3.5.2. ASSESSING THE USAGE OF THE LANGUAGE

This experiment is almost exactly as the experiment that is described in Subsection 3.5.1. The same state machines will be used for clustering and the similarity matrix is constructed for each clustering method to visually inspect how well the state machines are clustered together. The difference in this experiment is how we construct the vector representation of a state machine. In this experiment, the vector representation of a state machine is constructed following the procedure that is shown in Algorithm 6.

Algorithm 11: Clustering procedure of state machines using the language of the state machines.

```
Input: A list $V$ containing all variants and a list $B$ containing all base state machines
Let $M = V \cup B$
stringsSets = generateSetsOfStrings($B$)
createVectorRepresentations(stringsSets, $M$) // following the procedure of Algorithm 6
for each model $m$ in $M$ do
    clusterModel($m$) // using LSH
end
// Create the similarity matrix and heatmap following the procedure of
   Algorithm 8
createMatrixHeatmap($M$)
```

The clustering procedure of this experiment is shown in Algorithm 11. As we want to cluster the variants together with their base state machine, we will generate the sets of string using the base state machines. The intuition behind this is that a variant is generated using its base state machine, which means that its language is very similar the language of its base state machine and it would have a higher probability of getting hashed into the same bucket as its base state machine.

The difference in this experiment is that we use the language of a state machine to create its vector representation, which will be used for clustering. The same state machines and comparison methods that are described in Subsection 3.5.1 will also be used for this experiment.

3.5.3. ASSESSING THE USAGE OF STATE DISTRIBUTION OVER THE BUCKETS

This experiment is similar to the experiments that are described in Subsection 3.5.1 and 3.5.2. It is similar in the sense of how we construct the similarity for each clustering method and how we will compare the clustering results between the clustering methods. The difference in this experiment lies in how we will construct the vector representation of the state machines and which kind of state machines will be used for clustering.

The clustering procedure of this experiment is shown in Algorithm 12.

Algorithm 12: Clustering procedure of state machines using the distribution of its states over the buckets.

```
Input: A list $M$ containing all state machines
createVectorRepresentations($M$) // following the procedure of Algorithm 7
for each model $m$ in $M$ do
    clusterModel($m$) // using LSH
end
// Create the similarity matrix and heatmap following the procedure of
   Algorithm 8
createMatrixHeatmap($M$)
```

The vector representation of the state machines is generated following the procedure that is shown in ALgorithm 7. The state machines that we will be using in this experiment will be different than the ones that are used in the other two experiments. This is because we cannot manually create the variants in the same way as we did for the other two experiments. For the other two experiments, we can make the modifications and then read the variants from their files. In this case, the buckets distributions are computed when the training data of the models are streamed through the whole system. It is difficult for us to create training data from which we can guarantee that we can learn a specific variant. Therefore, we have decided to use a different set of PAutomaC models for this experiment.

3.5.4. RUN-TIME ANALYSIS OF FINDING A MOST SIMILAR STATE MACHINE

In this experiment, we want to evaluate the run-time performance for finding a similar state machine (given a particular state machine) using LSH. The reason for doing this experiment is because the clustering of state machines is not implemented in the original implementation of the system. To find a similar state machine, pair-wise comparisons would need to be done between all state machines. The goal of using LSH is to provide a quicker manner to find, given a state machine, its most similar state machine. We want to evaluate in this experiment whether using LSH would provide any speedup in finding a most similar state machine for a particular state machine. The same PAutomaC state machines that are used in Subsection 3.5.1 will also be used in this experiment. The state machines are clustered using LSH and then we will record the time it took for us to find a similar state machine using LSH. We will also record the time that it takes us to find a similar state machine using the pair-wise comparison approach. We then compare the time between both approaches.

3.5.5. RUN-TIME ANALYSIS OF FINDING A MOST SIMILAR FINGERPRINT

Schouten had not designed the system to only learn state machines in real-time but also to detect anomalies in the stream of network traffic data. The system can detect anomalies by computing the KL-Divergence between each state machine that was learned from the stream of NetFlow data and each fingerprint that was generated from a previous learning process. As we are trying to find the match between a fingerprint and a state machine, we are essentially looking for a fingerprint that is most similar to the behaviour of a state machine. When a match has been found, a flag can be raised that an anomaly has been detected. The procedure for finding a most similar fingerprint is shown in Algorithm 13.

Algorithm 13: Procedure for finding a most similar fingerprint.

```
Input: A list $F$ containing all fingerprints and a state machine $m$
mostSim = $\infty$
mostSimilarFingerprint = null
for each fingerprint $f$ in $F$ do
    simlarity = computeSimilarity($f$, $m$)
    if similarity < mostSim then
        mostSim = similarity
        mostSimilarFingerprint = $f$
    end
end
// Check if the fingerprint exceeds the threshold for a match.
if mostSim > MATCH_THRESHOLD then
    return mostSimilarFingerprint
end
return null
```

From the procedure that is shown in Algorithm 13, we would have to go through all fingerprints to find a most similar fingerprint for a match. The run-time performance of this process will be impacted as the number of fingerprints grow larger. The thought that the number of fingerprints would grow is not unrealistic; we want to detect any kind of anomaly that might occur in the network traffic and to do so we would need to have as many fingerprints as possible. This list of fingerprints also needs to be updated whenever a new kind of malware has been found. Thus the list of fingerprints would keep growing over time. To resolve the issue that the performance of this process will be impacted by the growing list of fingerprints, we can use LSH to cluster the fingerprints. We can use the clusters of fingerprints to find a most similar fingerprint in a quicker manner as we eliminate many of the comparisons that need to be done. The procedure for finding a most similar fingerprint using LSH is shown in Algorithm 14.

Algorithm 14: Procedure for finding a most similar fingerprint using LSH.

```
Input: A state machine $m$ and the buckets $B$ containing the hashed fingerprints
mostSim = $\infty$
mostSimilarFingerprint = null
hash = computeHash($m$) // hash is computed using LSH
bucket = getBucket(hash, $B$) // bucket containing fingerprints with same hash.
for each fingerprint $f$ in bucket do
    similarity = computeSimilarity($f$, $m$)
    if similarity < mostSim then
        mostSim = similarity
        mostSimilarFingerprint = $f$
    end
end
// Check if the fingerprint exceeds the threshold for a match.
if mostSim > MATCH_THRESHOLD then
    return mostSimilarFingerprint
end
return null
```

To assess whether we gain an improvement in the run-time for finding a most similar fingerprint using LSH, we will record the time it takes to find a most similar fingerprint for a match. We will also record the time it takes to find the most similar fingerprint using the original method. We will then compare the run-time between both versions and see whether we gain an improvement using LSH.

For this experiment, we will learn 1000 fingerprints using NetFlow data that is published by the Stratosphere IPS project. To learn as many different fingerprints as we can, we have used NetFlow data of 22 different malware. We then stream the NetFlow data of the selected malware and we record the time each method took to find a most similar fingerprint.

3.5.6. ASSESSING THE CLUSTERING OF STATE MACHINES OF MALWARE

As we are using LSH to cluster similar state machines together, it is interesting for us to know how well LSH can cluster state machines that are learned from benign and malicious network traffic data. Being able to create the clusters of benign or malicious state machines (i.e. state machines that were learned using either benign or malicious network traffic data), we provide a manner to separate the space between benign a malicious network traffic data using LSH. The clusters can then be handy for detecting anomalies on a network.

For this experiment, we will use NetFlow data that is published by Stratosphere IPS project [49] to learn state machines. We will use both benign and malicious data, and we will cluster the state machines using LSH, KMeans Clustering and Random Clustering. In the real world, there are usually more benign network traffic data than malicious network traffic data. Thus for this experiment, we will have many benign state machines and only a few malicious state machines.

For each clustering method, we will compute its Silhouette Coefficient scores based on its clustering results. We will only compute the Silhouette Coefficient scores in this experiment as we want to know how well the state machines (i.e the state machines learned from benign and malicious NetFlow data) between the clusters are separated from each other.

3.6. RUNNING THE METHOD ON A RASPBERRY PI

The ultimate goal of this thesis is to investigate whether it is possible to learn state machines in real-time on an embedded device. In this work, we will use a Raspberry Pi model 4 as the device on which we will evaluate whether we can learn state machines in real-time from a stream of network traffic data. The Raspberry Pi comes with 4 GB RAM and 32 GB storage, and the Raspberry Pi OS is installed on the device. To evaluate whether we can learn state machines in real-time on the Raspberry Pi, we have set up three different experiments. In these experiments, we will run both the original method that was designed by Schouten and our method that utilises LSH. The following subsections describe the experiments.

3.6.1. LEARNING STATE MACHINES FROM MALICIOUS NETFLOW DATA

As our goal is to investigate whether it is possible to learn state machines in real-time on an embedded device, we will export both the original method and our method that utilises LSH onto the Raspberry Pi. Each version will be run on the Raspberry Pi and we will stream malicious NetFlow data through Apache Kafka. We will be using captures of four different malware that are published on the Stratosphere IPS project, namely the captures of Emotet, Dridex, Necurs and Trickbot.

For each version, we will manually check whether state machines are learned from the stream of malicious NetFlow data. We will then compare the state machines that are learned using both versions by doing the same comparison as in Subsection 3.3.2. This is to see whether the differences are larger when we learning state machines on an embedded device.

3.6.2. ASSESSING THE SCALABILITY OF THE METHOD ON RASPBERRY PI

As we are running the method on a smaller device that has less resource than the usual desktop computers or laptops, it is interesting to see how scalable the method is on a smaller device. We will therefore assess the scalability of the method by computing the throughput of the method when it is running on the Raspberry Pi. Knowing the throughput of the method, it provides us knowledge on how fast the method can process a stream of NetFlow data on a Raspberry Pi. We will compute the throughput using the formula that is shown in equation 3.4.

For this experiment, we will use three different datasets from the Stratosphere IPS project. Each dataset has a different number of flows recorded in the file. We will stream each dataset 50 times through Apache Kafka and record the time that the method took to process all flows of the dataset.

$$Throughput = \frac{Total\ number\ of\ flows}{Time\ to\ process\ all\ flows} \tag{3.4}$$

3.6.3. RUN-TIME ANALYSIS OF FINDING A MOST SIMILAR FINGERPRINT ON RASPBERRY PI

As we are running the newly modified version that utilises LSH on the Raspberry Pi and it has much less resource than a larger device such as a desktop computer, it would interesting to see whether we also gain an improvement on the run-time for finding a most similar fingerprint on the Raspberry Pi. After all, the goal for using LSH is to provide an improvement on the run-time so that it can run more efficiently on a smaller embedded device. Thus we will compare the run-time for finding a most similar fingerprint between the original version and the version that utilises LSH. We are essentially running the same experiment that is described in Subsection 3.5.5 on the Raspberry Pi.

3.6.4. RUN-TIME ANALYSIS OF STATE-MERGING PROCESS ON RASPBERRY PI

The goal of using LSH is to improve the run-time of the state-merging process so that the modified method can run more efficiently on a smaller embedded device. Thus in this experiment, we want to evaluate whether LSH provides any speedups in the state-merging process when it is run on the Raspberry Pi. We are essentially running the same experiment that is described in Subsection 3.3.3 on the Raspberry Pi.

4

Clustering States

In this chapter, we present the results of the experiments that were mentioned in Section 3.3. We have divided the results into two different categories; experiments that used artificially generated data and experiments that used real-world data. We first present the results of the experiments that used artificially generated data in Section 4.1. Then we present the results of the experiments that used real-world data in Section 4.2. Finally in Section 4.3, we present the conclusion on the hashing of states using LSH.

4.1. Experiments that Used Artificially Generated Data

4.1.1. Clustering Artificially Generated Vectors Using LSH

To evaluate how well LSH can cluster states together in the bucket, we generated 100 vectors and hashed them using LSH into different buckets. Each vector is used to represent the Count-Min sketch vector of a state. For the generation of the vectors, we have chosen four points that we used to sample values to put into the vectors. For each point, we sampled values close to the point to generate 25 vectors. We have selected to use four points to create clear clusters of the vectors. Doing so, allows us to easily see how well similar vectors are clustered together by a particular clustering method.

Experiment Setup

After we have generated the vectors to represent the states, we clustered the vectors using LSH, KMeans Clustering and Random Clustering. For each method, we cluster the vectors ten times and the clustering results are stored for further evaluation. As we have generated vectors using four different points, it seems natural to cluster the vectors using four clusters. Thus for each clustering method, we have clustered the vectors using four clusters. Additionally, we were interested in how well each clustering method would perform if we use a higher number of clusters to cluster the vectors. By using a higher number of clusters to cluster the vectors, it becomes more difficult for each clustering method to decide to which cluster a vector should be assigned to. For this experiment, we used the implementation of KMeans Clustering that is provided by the Scikit-learn libary [50]. After the vectors are clustered, we evaluated the clustering results of each method by doing the following steps:

1. For each clustering method, cluster the vectors ten times and store the clustering results.

2. For each method, compute the Coefficient Score and the accuracy of each run.

Analysis of the Results

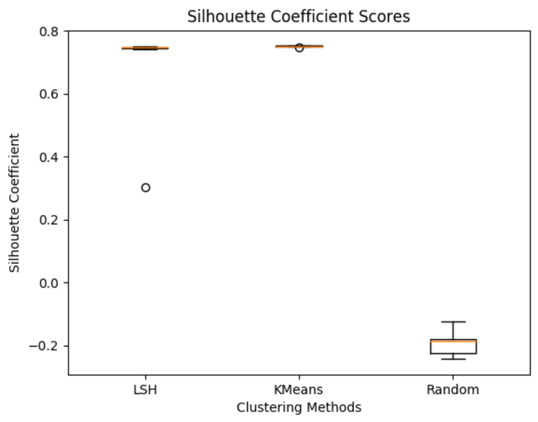
Figure 4.1 presents the silhouette coefficient scores and the accuracy of the clustering methods for clustering the vectors using four clusters. The results for using a higher number of clusters can be found in Appendix A. Figure 4.2a presents the heatmap that is generated from a similarity matrix, and the rows and columns are not sorted. It should be noted that from this heatmap it is hard to see that there are four different clusters. This is caused by the cosine similarity metric that we have used to construct the similarity matrix; the values of cosine similarity are between zero and one, and this makes it difficult to see the difference in the clusters. If we have generated the heatmap using the Euclidean distances between the vectors, then it would have been easier to see the clusters e.g. Figure 4.2b. Figure 4.3 presents the heatmaps that are generated from the sorted similarity matrices.

From the heatmaps that are generated for each clustering method, we see that LSH and KMeans Clustering produced similar heatmaps and Random Clustering a completely different heatmap. From the heatmap that is generated by Random clustering, we see that there are multiple black blocks dispersed over the whole heatmap. This means that many similar vectors were put into different clusters, thus us also meaning that there are vectors that are incorrectly clustered. This is reflected in the silhouette score and the accuracy that Random Clustering has achieved in our experiment; its performance is significantly lower than the performance of LSH and KMeans Clustering.

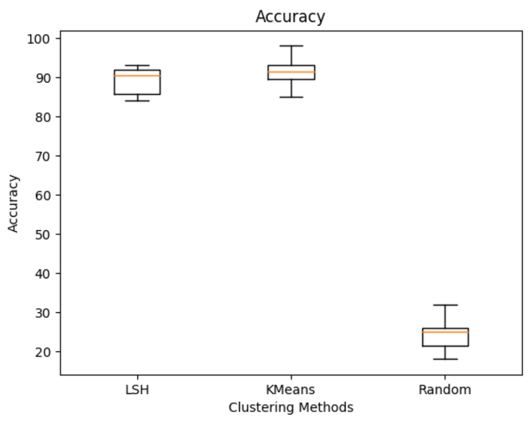
When we look at the heatmaps that were generated for LSH and KMeans Clustering, we can see that the heatmap that was generated for LSH looks like a flipped image of the heatmap that is generated using the unsorted similarity matrix. When we look at the one that was generated for KMeans Clustering, it is slightly different than the one of LSH and is difficult to say which one is better. This shows that only comparing the clustering results is not enough.

When we compare the Silhouette Coefficient scores between LSH and KMeans Clustering, we see that both methods have achieved almost the same score. However, when we compare the accuracy between the two methods, we see that KMeans Clustering has achieved slightly higher accuracy than LSH. This can be explained by the fact that KMeans directly uses the distances between the vectors to assign each vector to their corresponding clusters. LSH does directly use the distances between the vectors to assign each vector to their corresponding clusters. LSH loses information that can be used to cluster the vectors better. This is a trade-off that has to be made for using LSH; in KMeans Clustering, the centroids constantly need to be recomputed to create a better centroid for a given cluster. With LSH, there is no need to do any recomputation but it does mean that LSH would make more mistakes. Though LSH has a lower performance than KMeans Clustering, we also see that LSH can achieve a quite high Silhouette Coefficient score and accuracy.

When using a higher number of cluster to cluster the vectors (see extra results in Appendix A), we see that the performance of LSH drops significantly. This again can be explained by the fact that LSH does not directly use the distances between the vectors to cluster the vectors. This shows that LSH is not good in clustering items using a higher number of clusters than the actual number of clusters that exist in the underlying distribution of the items.



(a) Silhouette Coefficient scores for clustering vectors using four clusters.

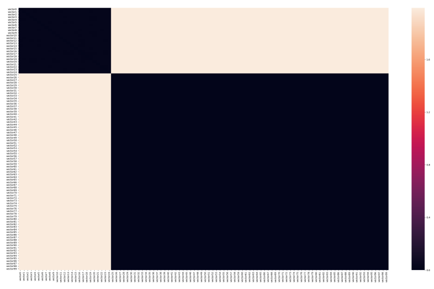


(b) Accuracy for clustering vectors using four clusters.

Figure 4.1: Silhouette Coefficient scores and accuracy of each clustering method for clustering vectors using four clusters.

Conclusion of the Experiment

Based on the results of our experiment, we the performance of LSH lower than the KMeans Clustering but it is much better than the performance of Random Clustering. The results match our expectation. Though the performance of LSH is lower than KMeans Clustering, it still managed to achieve a quite high Silhouette Coefficient score and accuracy. The lower performance of LSH can be explained by the fact that LSH does not directly use the distances between the items to assign each item to their corresponding clusters. This means that LSH loses information that can be used to cluster items better. This is a trade-off that has to be made for


(a) Heatmap generated from the unsorted similarity matrix using the cosine similarity between the vectors

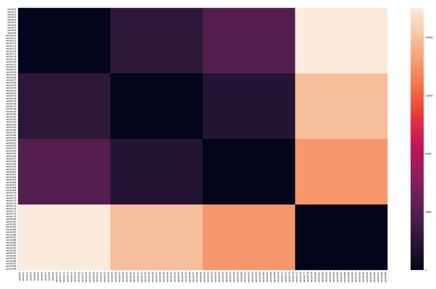
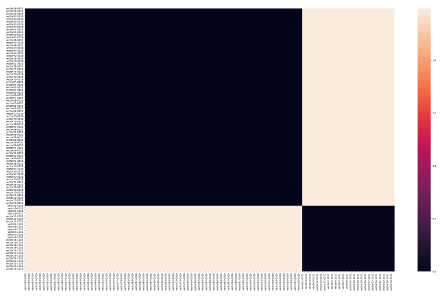
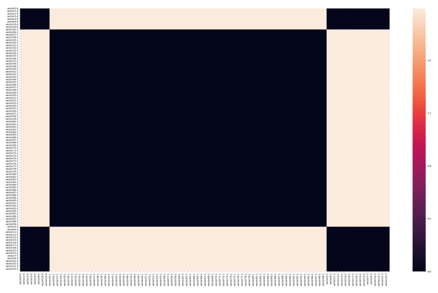

(b) Heatmap generated from the unsorted similarity matrix using the Euclidean distances between the vectors

Figure 4.2: Heatmaps generated from the unsorted similarity matrix.


(a) Clustering results of LSH


(b) Clustering results of KMeans Clustering

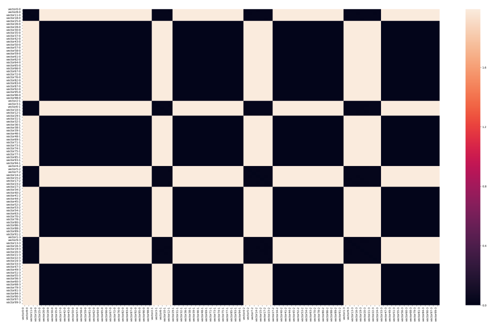

(c) Clustering results of Random Clustering

Figure 4.3: Clustering results of LSH, KMeans Clustering and Random Clustering. Four clusters were used for each method.

using LSH as it does not need to do any re-computations just like KMeans Clustering during the clustering process.

4.1.2. Run-time Analysis Of State-Merging Process - Part 1

To evaluate whether LSH provides any speedup to the state-merging process, we compared the run-time of the state-merging process between the version that uses LSH and the original version (pair-wise comparisons). In this part of the experiment, we ran the state-merging process in isolation i.e. the process is not within the system that was implemented by Schouten. This experiment is to show on an abstract level whether LSH provides any speedup in the state-merging process. We expect that LSH to provide a speedup as we do not have to do all pair-wise comparisons to find a most similar state for a merge; we only need to search within a particular bucket. The number of comparisons that we need to do is much lower.

EXPERIMENT SETUP

For this part of the experiment, we generated multiple sets of vectors that are used to represent states. Each set contains a different number of vectors and we recorded the run-time of the state-merging process by doing the following steps:

1. For each set, hash each vector into buckets using LSH.
2. For each vector v in the set, find a most similar vector v_s using the pair comparison method and record the time that it took to find an answer.
3. For each vector v in the set, find a most similar vector v_s using our LSH approach and record the time it took to find an answer.
4. Repeat Step 2 and 3, 50 times each.
5. Average the times that were recorded in Step 2 and 3 over the number of runs, which in this case is 50.

ANALYSIS OF THE RESULTS

Figure 4.4 presents the average run-time of the state-merging process (ran in isolation) between the original version that uses pair-wise comparisons and the newly modified version that uses LSH. We see from the results that approach of using LSH has a lower average run-time than the original version that uses pair-wise comparisons and the gap is even larger when we are dealing with a large number of states. This shows that LSH provides a speedup in the run-time when the number of states grows larger. The results were expected as for a large number of states, the pair-wise comparison approach would have to go through all the states to find a most similar state but there is no need to do so when we use LSH. There is much less number of states that we have to go through to find a most similar state.

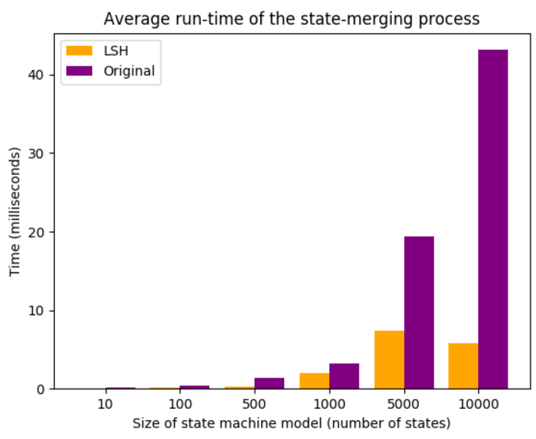


Figure 4.4: Average run-time of the state-merging process that was run in isolation.

CONCLUSION OF EXPERIMENT

In this experiment, we have created multiple sets of vectors that are used to represent states and each set contains a different number of vectors. We ran the state-merging process in isolation (i.e. outside the whole system that was implemented by Schouten) together with the sets of vectors as input data. We recorded and compared the run-time of the state-merging process between the original version that uses pair-wise comparisons and the new version that uses LSH. The results show that LSH does provide a speedup to the state-merging process as the number of states grows larger. This was expected as using LSH would mean that we eliminate many pair-wise comparisons between states and thus also meaning that we have to go through fewer states to find an answer.

4.1.3. RUN-TIME ANALYSIS OF STATE-MERGING PROCESS - PART 2

As we have analysed the run-time of the state-merging process when it is run in isolation, it seems natural for us to also analyse its run-time when it is run within the system; we have seen that LSH provides a speedup to the state-merging process when it is run in isolation but will LSH also provide a speedup when the whole process is run within the system? Thus in this part of the experiment, we wanted to evaluate whether LSH provides a speedup in the state-merging process by recording its run-time when it is run within the system. The run-time was recorded for both the original version and the version that uses LSH and the results were compared to each other.

EXPERIMENT SETUP

In this part of the experiment, we wanted to analyse the run-time of the state-merging process within the system that was implemented by Schouten. Due to this reason, we cannot use artificially generated vectors to represent states. Instead, we need to stream NetFlow data through the system and record the run-time of the state-merging process. For this experiment, we have used the NetFlow data of four different PAutomaC models and each model has a different number of states. The run-time of the state-merging process is recorded as follow:

1. When a call is made to find a most similar state v_s for a given state v, record the time that it took to find an answer.
2. Repeat Step 1 ten times with the same NetFlow data.
3. Average the times that were recorded over the number of runs, which in this case is ten.

ANALYSIS OF THE RESULTS

Figure 4.5 presents the average run-time of the state-merging process of both versions. From the figure, we see the difference in run-time between the two versions is quite small. LSH managed to have a slightly lower run-time on two of the state machines that we have learned using the whole system.

An explanation for the small difference in run-time between the two versions is the fact that the sizes of the state machines that we have used in the experiment are not large. The largest state machine that we could learn was one that had 25 states. This maximum was already set within the current implementation of the system. Changing this maximum would mean that we have to make several changes to the system for us to learn larger state machines. We thought that it would cost quite some time to make these changes and it is beyond the scope of this thesis. Thus we decided to not make any changes.

Compared to the results from Subsection 4.1.2, it seems like the version that uses LSH does not provide a speedup in the run-time of the state-merging process when it is run within the whole system. The result is expected as LSH would only provide a speedup when we are dealing with a large number of states. In this case, the number of states is quite small and therefore it seems that LSH does not provide a speedup in the state-merging process.

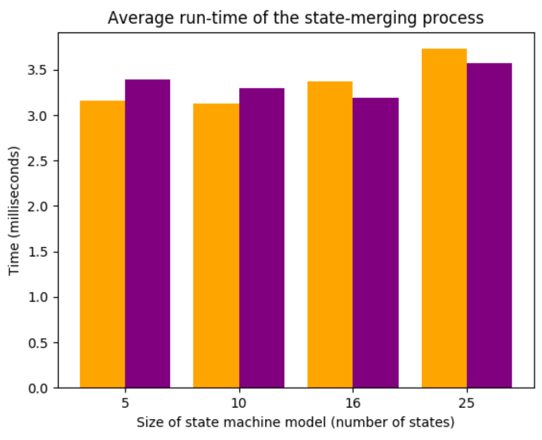


Figure 4.5: Average run-time of the state-merging process that was within the system that was designed by Schouten.

CONCLUSION OF EXPERIMENT

In this experiment, we analysed the run-time of the state-merging process when it is run within the whole system itself. We recorded and compared the run-time of the state-merging process between the original versions and the one that uses LSH. The little difference in the run-time between the two versions can be explained by the sizes of state machines that we have used in the second part of our experiment. LSH provides an improvement in the run-time when we are processing state machines with a large number of states. In our experiment, the largest state machine that we can learn is one that has 25 states and therefore we do not see an improvement in the run-time by using LSH.

4.2. EXPERIMENTS THAT USED REAL-WORLD DATA

4.2.1. COMPARISON OF THE GENERATED STATE MACHINES

From the conclusion of Section 4.1.1, we know that LSH does not cluster states as well as KMeans Clustering. This would mean that there would be an approximation error when states are merged. As the original method also learns state machines using approximations, using LSH would increase the approximation error and the state machine might be different than the one that is learned using the original method. We do not want the state machines (learned using the LSH) to be a lot different than the ones that are learned using the original method as this would mean that wrong state machines will be learned from the NetFlow data. Consequently, this would mean that a state machine that is learned from benign NetFlow data might be similar to a state machine that is learned from malicious NetFlow data or vice versa. This would increase the number of false positives/negatives. Thus the state machines that are learned using the original method will serve as ground truth for this experiment. The state machines were learned using the NetFlow captures of Dridex, Emotet, Necurs and TrickBot. The captures were downloaded from Stratosphere IPS Project [49].

EXPERIMENT SETUP

For each metric, we ran the experiment 50 times. The following steps describe how we compared the state machines using the KL-Divergence metric:

1. For each method, learn the state machines from the same set of NetFlow data.
2. For each state machine that is learned using the original method, generate 1000 random strings and compute their corresponding probabilities. Each string has a maximum length of 20.
3. Simulate the strings that were generated in Step 2 on the same state machines that are learned using LSH and compute the probabilities.
4. For each pair of state machines (i.e the state machine that is learned using the LSH and the same state machine that is learned using the original method), compute the KL-Divergence between the pair of state machines using the probabilities that were computed in Steps 2 and 3.
5. For each state machine, average the results that were computed in Step 4 over the number of runs, which in this case is 50.

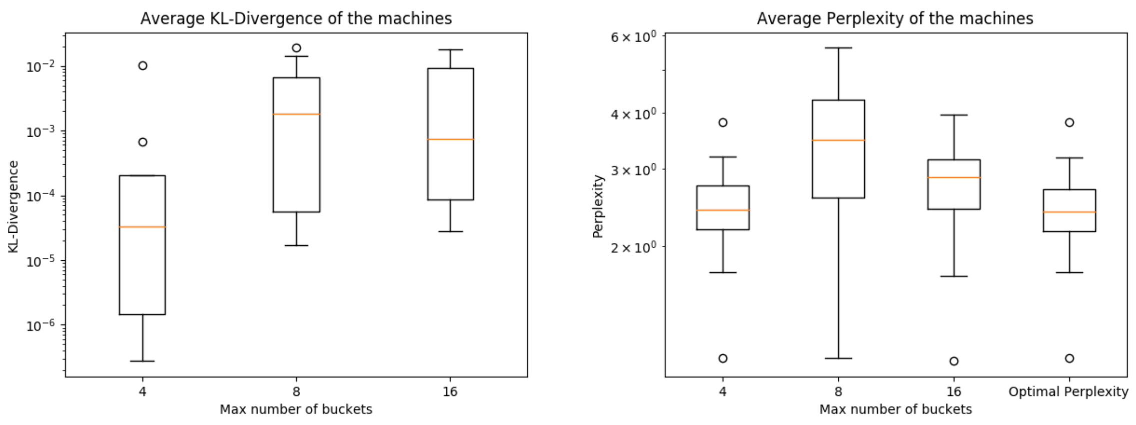
The steps for the comparison using the Perplexity metric is almost the same as the ones of KL-Divergence metric. There is one additional that is done in the comparison using the Perplexity metric: the average optimal perplexity is computed for each baseline state machine. The optimal perplexity is computed by comparing the true state machine with itself and it describes the lowest possible perplexity value that can be achieved for the given state machine.

ANALYSIS OF THE RESULTS

Figure 4.6 presents the comparison results for the state machines that were learned from the Dridex dataset. The remaining comparison results can be found in Appendix A.

Looking at the comparison results, we see that average KL-Divergence values of the state machine values are close to zero. This means that the state machines that were learned using LSH are just ever so slightly different than the ones from the original method. We also see that the difference is higher when we used a larger number of buckets, however, the difference is still close to zero.

Additionally from the comparison results, we also see that the average perplexity values of the state machines are close to the optimal perplexity value. Thus this also shows that the state machines that were



(a) Average KL-Divergence for the state machines of Dridex.

(b) Average Perplexity for the state machines of Dridex.

Figure 4.6: Boxplots showing the average KL-Divergence and average Perplexity for the state machines that were learned for Dridex. The values are computed by comparing the state machines that were learned using LSH to the ones that were learned using the original method.

learned using LSH are quite similar to the ones that were learned using the original method. We also see that for Dridex, the difference between the average perplexity values and the optimal perplexity is larger when compared to the other malware. This can be explained by the state machines that were learned; the size of the state machines for Dridex are small (largest model has only three states) and this could cause it to be difficult for LSH to cluster the states correctly. Thus we get a higher approximation error and therefore also a higher perplexity value.

Conclusion of Experiment

Based on the results, we say that the state machines that were learned using LSH are quite similar to the ones that were learned using the original method. Our approach of using LSH increases the approximation error for learning a state machine but the effect on the final state machine is low.

4.3. Conclusion on the Hashing of States Using LSH

One of the goals of using LSH is to improve the run-time of the state merging process. The idea was to use LSH to cluster states together so that we can find a similar state quicker for a merge. We have set up multiple experiments where we evaluated how well cluster states together using LSH. Based on the results of our experiments, LSH has a lower performance than KMeans clustering and this matches our expectations. Though LSH has lower performance, it still achieved a quite high Silhouette Coefficient Score and accuracy.

An explanation of why our LSH does not perform as well as KMeans Clustering could be due to the way how the clustering is done between these two methods. KMeans Clustering uses centroids to check to which cluster a particular data point should be assigned to. The distance between each data point and each centroid is computed in order to see which centroid is the closest to the data point. The data point will be assigned to the same cluster in which the closest centroid belongs to.

LSH does not use centroids to assign data points to their corresponding clusters. In our particular implementation of LSH, we use random hyperplanes to divide the space in which the data points are located into multiple regions. Each data point is put into a particular region that corresponds to the hash of the data point and each region can be seen as a cluster of data points. As we do not compute the distance between the data points in order to assign them to their corresponding clusters, we lose information between the data points and this increase the chance for an error to be made. Furthermore, the hash functions are chosen in such a way that they maximise collisions between the data points. This increases the probability that data points that are far away from each other are put within the same cluster. This also increases the chance for an error to occur.

Although LSH increases the chance for an approximation to occur, we have seen from the results of our experiment that its effect on the final state machine is low. The state machines that we have learned using LSH were very similar to the ones that were learned using the original method.

From the results of our experiment, we also saw that we did not gain an improvement in the run-time of the state-merging process using LSH. This can be explained by the fact that we cannot learn a state machine that has more than 25 states. We would only see an improvement in the state-merging process when we are dealing with state machines that have a large number of states. Thus based on the results, our approach of using LSH to cluster states does not provide a run-time improvement on the state-merging process.

5
Clustering State Machines

In this chapter, we present the results of the experiments that are mentioned in Section 3.5. Just like in Chapter 4, we have divided the experiments into two categories; experiments that used artificially generated data and experiments that used real-world data. We first present the results of the experiments that used artificially generated data in Section 5.1. Then we present the results of the experiments that used real-world data in Section 5.2. Finally in Section 5.3, we present the conclusion on the hashing of state machines using LSH.

5.1. Experiments That Used Aritficially Generated Data

5.1.1. Using State Transition Tables to Cluster State Machines

One of the methods that we have used to represent a state machine as a vector of numerical values, was to use the state transition table of the state machine. The last column of the state transition table will be used as the vector representation of the state machine. Thus it is only needed to store the last column of the state transition table in memory instead of the whole table. As mentioned in the description of this method, we need to construct the generalised state transition table so that this method would work on any arbitrarily sized state machine.

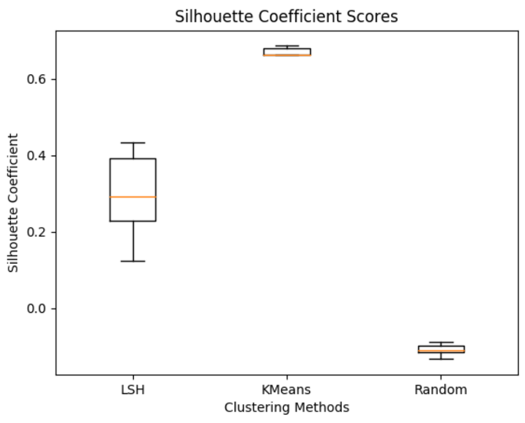
Experiment Setup

Once we have constructed the generalised state transition table, we proceeded to hash the state machines. As mentioned in Subsection 3.5.1, we selected five state machines from the PAutomaC competition and constructed ten variants for each of these state machines. For this experiment, the five state machines and their variants are used for hashing and clustering. The state machines are clustered and evaluated as follow:

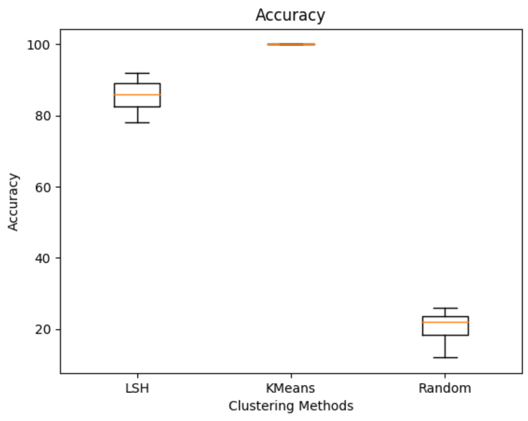
1. For each state machine, construct its vector representation following the procedure of Algorithm 5.
2. Cluster the state machines ten times following the procedure of Algorithm 9.
3. Redo step 2, but this time cluster the state machines using KMeans Clustering.
4. Redo Step 2, but this time cluster the state machines using Random Clustering.
5. For each method, compute the Silhouette Coefficient scores and accuracy of each run.

Analysis Of Results

Following the steps that we have mentioned in Subsection 3.5.1, we have constructed heatmap to visually compare the results of LSH to the results of KMeans Clustering and Random clustering. The heatmaps can be found in Appendix B. Furthermore, we have also computed the Silhouette Coefficient scores and accuracy of the clustering methods for comparison between the methods. Figure 5.1 presents the Silhouette Coefficient score and accuracy of the clustering for clustering the state machines using five clusters. More results on the scores and accuracy of the clustering methods can be found in Appendix B.
When we visually compare the clustering results between the clustering methods, we see that can capture some of the clusters well but it is still not as good as KMeans Clustering. When we look at the clusters that are formed by Random Clustering, we see that the clusters are not captured at all.



(a) Silhouette Coefficient scores for clustering the state machines using five clusters.



(b) Accuracy for clustering the state machines using five clusters.

Figure 5.1: Silhouette Coefficient scores and accuracy of each clustering method for clustering state machines using five clusters.

When we compare the scores and accuracy between the clustering methods, we see the performance of LSH is lower than KMeans Clustering and this matches the results that we have seen from the heatmaps. The same explanation that we have used to explain why the performance of LSH is lower than KMeans Clustering also applies in this case; we do not directly use the distance between the items to assign the items to their corresponding clusters. This means that we lose information when are clustering state machine using LSH. Losing this information can cause LSH to make mistakes.

We also see that the performance of LSH also follows the same trend as with the clustering of states; if we cluster the state machine s using a higher number of clusters than the actual number of existing clusters, then the performance of LSH drops. Though there is a drop in the performance of LSH, the drop is not as big as when we were clustering vectors using a higher number of clusters. This could mean that the way how we have constructed the vector representations of the state machines work well for the clustering state machines; the vectors representations of the state machines capture the similarity between state machines than the vectors representation that we have used to capture the similarity between the states.

Conclusion of the Experiment

In this experiment, we clustered state machines using the state transition table representation. From the results, we see that LSH has a lower performance compared to KMeans Clustering. This was expected based on the results that we have seen in Subsection 4.1.1. The same explanation that we have used to explain why the performance of LSH is lower than KMeans Clustering in Subsection 4.1.1 can also be used in this case. Additionally, we have seen from the results that the performance of LSH drops when a higher number of clusters were used to cluster the state machines but the drop is as big as when we clustering vectors. This could mean that the way how we created the vector representations of the state machines works well for the clustering of state machines; the vector representations of the state machines capture the similarity well between the state machines.

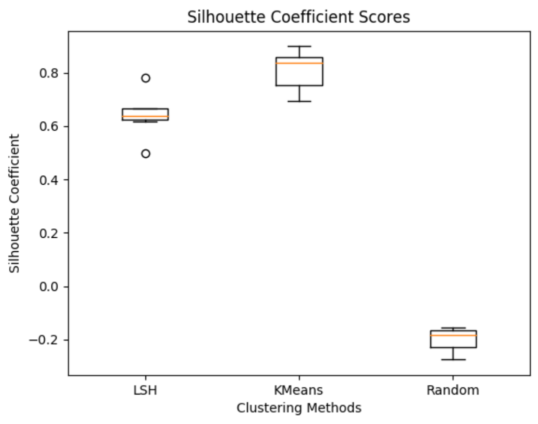
5.1.2. Using Language of State Machine to Cluster State Machines

Another method that we have used to represent a state machine as a vector of numerical values, was to use the language of the state machine. The intuition behind this method is that each state machine accepts a particular set of strings. These strings are part of the language of the state machine and they can be used to help describe the behaviour of a state machine. When two state machines accept a completely different set of strings, then the behaviour of these two state machines are completely different. This also means that the language of these two state machines is also different. On the other hand, when two state machines have similar behaviour, there is an overlap in the set of strings that both state machines accept. This means that the language of both state machines is similar. Thus we can use the language of the state machines to create its vector representation and use it for hashing.

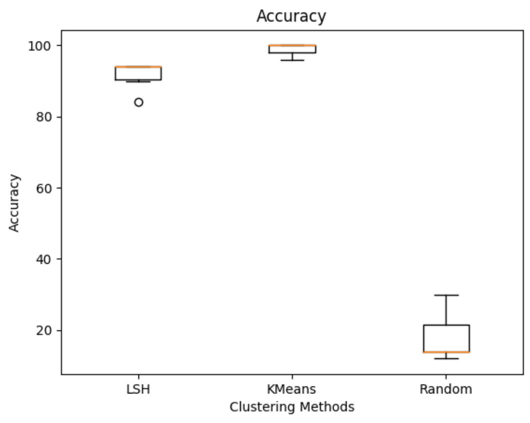
EXPERIMENT SETUP

To evaluate whether this method works well for the clustering of state machines, we have set up an experiment that is almost the same as the one for state transition table. The difference in this experiment lies in the way how we constructed the vector representations for the state machines. The state machines are clustered and evaluated as follow:

1. For each state machine, construct its vector representation following the procedure of Algorithm 6.
2. Cluster the state machines ten times following the procedure of Algorithm 11.
3. Redo step 2, but this time cluster the state machines using KMeans Clustering.
4. Redo Step 2, but this time cluster the state machines using Random Clustering.
5. For each method, compute the Silhouette Coefficient scores and accuracy of each run.



(a) Silhouette Coefficient scores for clustering the state machines using five clusters.



(b) Accuracy for clustering the state machines using five clusters.

Figure 5.2: Silhouette Coefficient scores and accuracy of each clustering method for clustering state machines using five clusters.

ANALYSIS OF RESULTS

Figure 5.2 presents the Silhouette scores and accuracy of the clustering methods for clustering the state machines using five clusters. The scores and accuracy of the clustering methods for using a higher number of clusters to cluster the state machines can be found in Appendix B. The heatmaps that were generated for this experiment can also be found in the same appendix.

From the scores and accuracy of the methods, we again see that LSH has a lower performance than KMeans Clustering. One thing that is interesting to see in this experiment, is that the scores and accuracy of LSH are higher than the scores and accuracy that were achieved in the experiment where we used the state transition table method to create the vector representations for the state machines. This shows that this method (using the language of the state machines) captures the similarity between the state machines better than the other method where we used the state transition table to capture the similarity. An explanation of why this method works better than the state transition table method could be due to the fact that we are not looking at the structure of the state machine but rather its behaviour. Two different state machines might have the exact same structure but not the same behaviour. This means that LSH could put these two state machines into the same cluster even though they have completely different behaviour. Using the language of the state machines, we disregard the structure of the state machines and we are directly looking at the behaviour of the state machines. LSH can then create better clusters of state machines with similar behaviour.

Furthermore from the results, we also see that the performance of LSH drops when a higher number of clusters are used to cluster the state machines. This follows the same trend that is also shown in the results of Subsection 4.1.1 and 5.1.1.

Conclusion of Experiment

In this experiment, we generated the vector representations of the state machines using their language and we used these vector representations for clustering. The results show that this method has achieved higher Silhouette Coefficient scores and accuracy than the state transition table method. This show that this method captures the similarity between the state machines better than the state transition table method. An explanation for this result could be due to the fact that the state transition table method could put state machines, that have exactly the same structure, into the same bucket but these state machines could have different behaviour. This method (using the language of the state machines) disregards the structure of the state machine and it looks directly at the behaviour of the state machines. LSH can then create better clusters of the state machines with similar behaviour.

5.1.3. Using the Distribution of the States Over the Buckets to Cluster State Machines

The third method that we have used to represent a state machine as a vector of numerical values, was to use the distribution of its states over the buckets. The assumption for this method is that similar state machines would have similar distributions. Thus state machines with distribution of its states over the buckets would be put into the same clusters.

Experiment Setup

As mentioned in Subsection 3.5.3, a different set of PAutomaC state machines were used in this experiment than the ones that were used into other two experiments. The reason why used a different set of state machines than the other two experiments was because we cannot manually create the variants just like in the other two experiments. We need to stream the data through the whole system to learn the state machines and it is difficult for us to create the right training data such that a specific variant is learnt from the data. Besides the set of state machines and how the vector representation is created for the state machines, the rest of the experiment is the same as for the other two methods. The state machines are clustered and evaluated as follow:

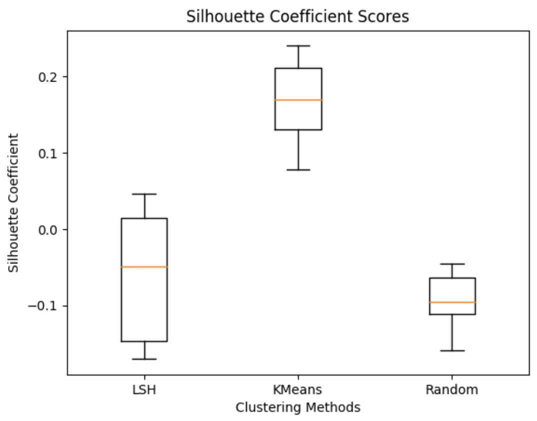
1. Stream the data of the state machines as NetFlow data using Apache Kafka.
2. For each state machine that is learned from the data, construct its vector representation following the procedure of Algorithm 7.
3. Cluster the state machines ten times following the procedure of Algorithm 12.
4. Redo step 2, but this time cluster the state machines using KMeans Clustering.
5. Redo Step 2, but this time cluster the state machines using Random Clustering.
6. For each method, compute the Silhouette Coefficient scores and accuracy of each run.

Analysis of the Results

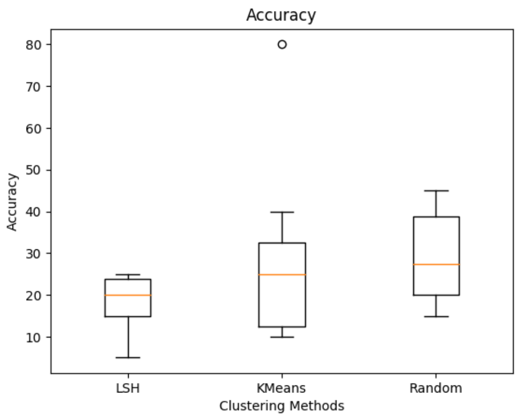
Figure 5.3 presents Silhouette scores and accuracy of the clustering methods for clustering state machines using four clusters. The scores and accuracy of the clustering methods for using a higher number of clusters to cluster the state machines can be found in Appendix B. The heatmaps that were generated for this experiment can also be found in the same appendix.

Just like with the other two methods, we see that the performance of LSH is lower than KMeans Clustering. Furthermore, we also see that the Silhouette Coefficient scores of LSH is almost similar to the scores of Random Clustering and the accuracy of LSH is lower than the accuracy of Random Clustering. This shows that this method does not work well for clustering state machines as the score is the same as random guessing and it is less accurate than random guessing.

An explanation of why this method performs the worst is related to the disadvantage of using this method for the creation of the vector representations; how LSH puts the states into the buckets is arbitrary. It is arbitrary in the sense that LSH puts similar states with a high probability into the same bucket but which bucket is used for putting the similar items together, that could be random. Therefore the distribution of the states over the buckets for each state machine is random. The locality-sensitive properties hold for the states within the buckets, but it is not carried over to the state machine. Thus the distribution of the states over the buckets shows no relation on the similarity between the state machines and this causes LSH to cluster state machines incorrectly using this method.



(a) Silhouette Coefficient scores for clustering the state machines using five clusters.



(b) Accuracy for clustering the state machines using five clusters.

Figure 5.3: Silhouette Coefficient scores and accuracy of each clustering method for clustering state machines using five clusters.

Conclusion of the Experiment

In this experiment, we evaluated whether we can use the distribution of the states over the buckets to cluster state machines. Based on the results, it seems that this approach is not a suitable approach for clustering state machines as this method performed significantly worse than the other two methods. The bad performance could be explained by the fact that the distribution of the states over the buckets is random for each state machine and therefore it shows no relation on the similarity between the state machines. The locality-sensitive properties hold for the states but they do not carry over to the state machines.

5.1.4. Run-time Analysis of Finding Similar State Machine

The clustering of state machines is not implemented in the original version of the system and thus to find a most similar state machine, given a particular state machine, pair-wise comparisons would need to be done between all the state machines. The similarity between a pair of state machines is computed by first generating one thousand strings from one model and then the probabilities of the strings are computed for both models. Afterwards, the Kullback-Leibler Divergence between the two state machines is computed with the help of the probabilities. This process needs to be repeated for each pair of state machines.

The current process of finding a most similar state machine is a quadratic operation and the run-time of this process will be impacted as the number of state machines grows larger. Our goal of using LSH is to be able to cluster state machines and find a most similar state machine faster than the pair-wise comparison approach.

Experiment Setup

To evaluate whether our LSH approach is faster than the pair-wise comparison approach, we compared the run-time of both methods. For this experiment, we chose to create the vector representations of the state machines using the language of the state machines and we will use them for hashing. We chose to use this method as this method also uses strings and it is the best performing method out of the three methods that we have tried to use for the hashing of state machines. The state machines that were used in this experiment are the same as the ones that were used in Subsection 5.1.1. The process for computing the run-time for both methods is as follows:

1. From the list of state machines, select one state machine at random.
2. Find a most similar state machine for the state machine that is selected in Step 1, using the pair-wise comparison approach and record the time it took to find an answer.
3. Find a most similar state machine using LSH and record the time it took to find an answer.
4. Repeat steps 1 till 3 fifty times and compute the average time it took to find an answer.

ANALYSIS OF THE RESULTS

The average run-time of both approaches is shown in Figure 5.4. From the results, it shows that LSH is faster than the current pair-wise comparison approach by three orders of magnitude. Though we also generate strings in our method, we generate much fewer strings compared to the pair-wise comparison approach; one thousand strings need to be generated for each comparison that is done with the pair-wise comparison approach, whereas for LSH, we use the strings that were generated from the base state machines and use them as fingerprints for the other state machines. Additionally, similar state machines are put into the same cluster by LSH. This eliminates many of the comparisons that need to be done between state machines. This explains why LSH is faster than the pair-wise comparison approach.

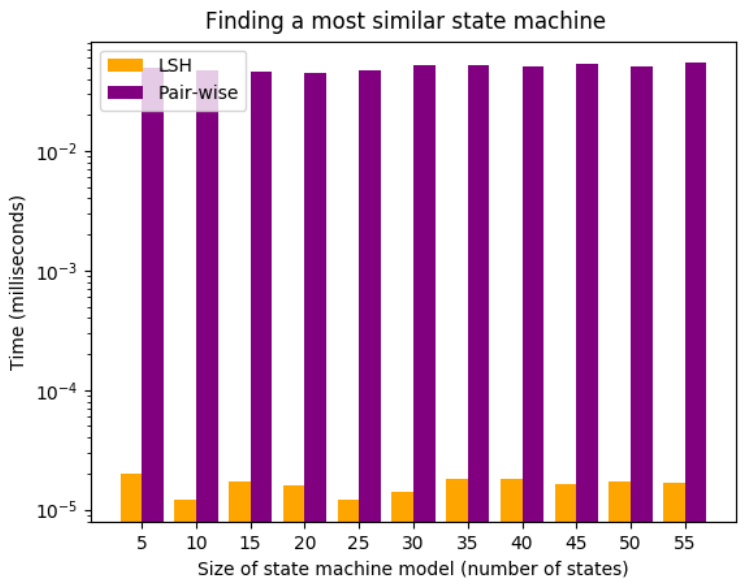


Figure 5.4: Average run-time for finding a most similar state machine.

CONCLUSION OF THE EXPERIMENT

In this experiment, we recorded the time it took to find a most similar state machine using the pair-wise comparison approach and using LSH. Based on the results, LSH is three orders of magnitude faster than the pair-wise comparison approach. Though we also generate strings, we generate much fewer strings than the pair-wise comparison approach. Using LSH, the strings are generated once and are used as fingerprints for the other state machines. For the pair-wise comparison approach, one thousand strings are generated for each comparison that needs to be done. Additionally, LSH eliminates many of the pair-wise comparisons that need to be done between state machines. This is an explanation of why LSH is faster than the pair-wise comparisons approach.

5.2. EXPERIMENTS THAT USED REAL-WORLD DATA

5.2.1. RUN-TIME ANALYSIS OF FINDING A MOST SIMILAR FINGERPRINT

To find a most similar fingerprint to match a state machine for anomaly detection, the current procedure requires us to go through each fingerprint and compute the similarity between the state machine and the fingerprint. As the list of fingerprint grows larger, the run-time of this process will be impacted. We can resolve this issue by using LSH to cluster similar fingerprints together and use these cluster to find a most similar fingerprint for a given state machine. We can find a most similar fingerprint for a state machine by first hashing the state machine and then use this hash to search for the fingerprint in the cluster that corresponds to the same hash. This way we can eliminate many of the comparisons between the fingerprints and the state machine. To assess whether we gain an improvement in the run-time of this process, we compared the run-time of the original version of the process to the one that uses LSH.

Experiment Setup

For this experiment, we learned 1000 fingerprints from 22 different malware. The datasets that we have used to learn the fingerprints come from the Stratosphere IPS project. For each version of the process, we recorded its run-time as follow:

1. Stream the NetFlow data of one malware using Apache Kafka.
2. Record the time it took to find a most similar fingerprint.
3. Repeat Step 1 and 2 fifty times.

Analysis of the Results

Figure 5.5 presents the run-time of each version for finding a most similar fingerprint. From the figure, we can see that the version that uses LSH is faster than the original version; it is faster by two orders of magnitude. The speedup can be explained by the fact that by using LSH, we do not have to go through each fingerprint to find a most similar fingerprint; by using LSH, we have eliminated many of the comparisons that need to be done between fingerprints and a state machine.

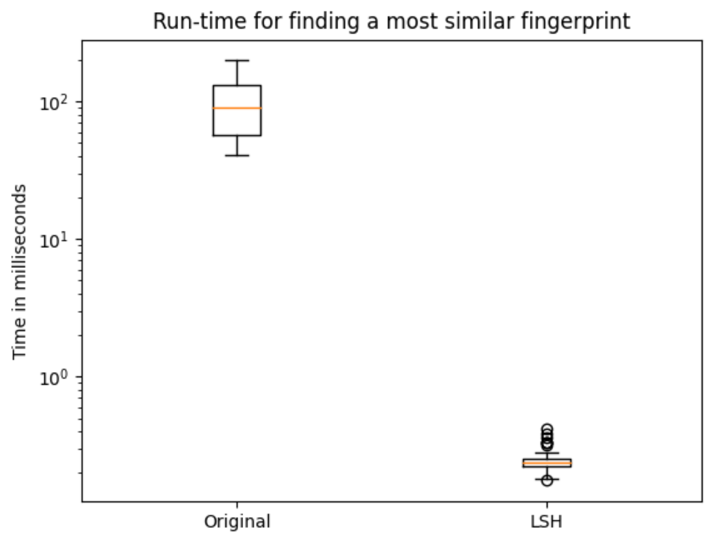


Figure 5.5: Run-time for finding a most similar fingerprint.

Conclusion of the Experiment

In this experiment, we wanted to evaluate whether we can use LSH to gain an improvement in the run-time of the process for finding a most similar fingerprint. Based on the results, LSH does provide an improvement in the run-time. The run-time of the version that used LSH was faster than the original version; it was faster by two orders of magnitude. Using LSH we managed to eliminate many of the comparisons that need to be done between fingerprints and a state machine and thus improving the run-time.

5.2.2. Clustering State Machines from Benign and Malicious NetFlow Data

One interesting use case for the clustering of state machines is to be able to cluster state machines that were learned from benign and malicious NetFlow data i.e. state machines that are learned from benign NetFlow data are clustered together and state machines that are learned from malicious NetFlow data are clustered together. To evaluate how well we can cluster benign and malicious state machines using LSH, we learned state machines from benign and malicious NetFlow data and cluster them using LSH.

5.2.3. Experiment Setup

For this experiment, we have used four different datasets from the Stratosphere IPS project to learn the state machines. Three of these datasets contains malicious NetFlow data and the last dataset contains benign

NetFlow data. In the real world, there are usually more benign traffic data than malicious data. Therefore for this experiment, we wanted to simulate network traffic that is as close as the real world network traffic data as possible. Thus for this experiment, we have more state machines that were learned from benign NetFlow data than machine machines that were learned from malicious NetFlow data. We have learned twenty state machines from the benign NetFlow data and five state machines from malicious NetFlow data. The state machines are clustered and evaluated as follow:

1. Cluster the state machines ten times using LSH.
2. Redo step 1, but this time cluster the state machines using Kmeans Clustering.
3. Redo step 1, but this time cluster the state machines using Random Clustering.
4. For each method, compute the Silhouette Coefficient Score of each run.

5.2.4. ANALYSIS OF THE RESULTS

Figure 5.6 presents the Silhouette Coefficient scores of the clustering methods. The scores of the clustering methods for using a higher number of clusters to cluster the state machines can be found in Appendix B. From the scores, we see that the performance follows the same trends as the other experiments where we hashed states and state machines; the performance of LSH is lower than KMeans Clustering and the performance of LSH drops as we use a higher number of clusters to cluster the state machines. Looking at the scores for using only four clusters to cluster the models, we see that the performance comes close to the performance of KMeans Clustering. Though LSH does not perform as well as KMeans Clustering, it can cluster state machines that were learned from real-life data quite well.

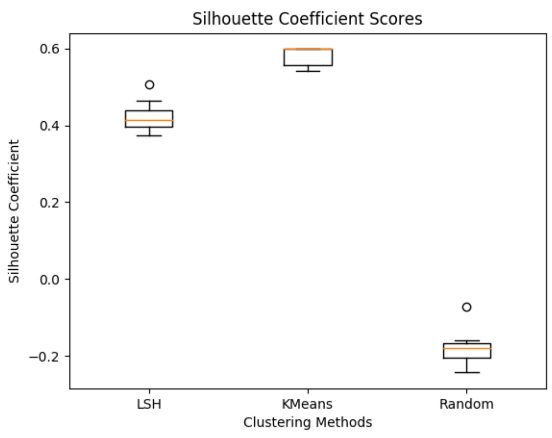


Figure 5.6: Silhouette Coefficient Scores for clustering state machines from benign and malicious NetFlow data using four clusters.

CONCLUSION OF EXPERIMENT

In this experiment, we wanted to evaluate how well LSH can cluster state machines that were learned from benign and malicious NetFlow data. Based on the results, we see that LSH follows the same trend as the other experiments where we hashed states and state machines; the performance of LSH is lower than the performance of KMeans Clustering and it drops when we use a higher number of clusters to cluster the state machines. Though LSH has a lower performance compared to KMeans Clustering, it can cluster state machines that were learned from real-life data quite well.

5.3. CONCLUSION ON THE HASHING OF STATE MACHINES USING LSH

Another goal of using LSH in this work is to cluster similar state machines together. Being able to cluster state machines together, we can find similar state machines in a much quicker manner. In the current implementation, the clustering of state machines is not implemented. To find a similar state machine, pair-wise comparisons would need to be done between the state machines. The run-time of this process will be impacted as the number of state machines grows larger.

Furthermore, the system was designed to detect anomalies by matching fingerprints to state machines. This process is essentially looking for a fingerprint that is most similar to the behaviour of a state machine. To find a most similar fingerprint to match a state machine, the current process requires us to go through all the fingerprints to find a most similar fingerprint. The run-time of this process will be impacted as the list of fingerprint grows larger. We can resolve this issue by using LSH to cluster the fingerprints. We can then hash a state machine and use its hash to look within a particular cluster for a most similar fingerprint. LSH eliminates many of the comparisons that need to be done between the fingerprints and a state machine, thus improving the run-time.

To be able to cluster the state machines using LSH, we needed to represent each state machine as a vector of numerical values. We attempted to use three different methods to transform a state machine into a vector that contains numerical values. From the three methods that we have attempted to use, the one with the best performance was the one that uses the language of a state machine to create its vector representation.

From the results of the experiment, we saw that the LSH approach does not perform as well as KMeans Clustering. The same explanation that is given in Section 4.3 can be used to explain why LSH does not perform just as well as KMeans Clustering; with LSH, we do not use the distances between the state machines directly to assign state machines to their corresponding clusters. LSH would then lose information while clustering the state machines and can make more mistakes. The results that we got from the experiments in which we try to cluster state machines are therefore expected.

When we have used a higher number of clusters to cluster the state machines, we saw that the performance of LSH does not drop as much it did when we were clustering states. This shows that our method for creating the vector representation for a given state machine works well for clustering state machines; the vector representation captures the similarity well between the state machines

Though our LSH approach does not perform as well as KMeans Clustering, we saw that our LSH approach has achieved quite a high accuracy for clustering state machines. Thus based on the results of our experiments, it seems that LSH can cluster state machine quite well. Furthermore, LSH provides a quicker manner to find similar state machines and to find a most similar fingerprint to a match a state machine for anomaly detection.

6
Learning State Machines On Raspberry Pi

In this chapter, we present the results of the experiments that were described in Section 3.6. We will again divide the experiments into two categories; experiments that used artificially generated data and experiments that used real-world data. We first present the results of the experiments that used real-world data in Section 6.1. Then we present the results of the experiments that used artificially generated data in Section 6.2. Finally in Section 6.3, we present the conclusion on learning state machines in real-time on a Raspberry Pi.

6.1. Experiments that Used Real-world Data

6.1.1. Running New Method On An Embedded Device

The goal of this thesis is to investigate whether we can learn state machines in real-time on an embedded device. To evaluate whether this is possible, we exported both the original and the modified version of the method onto a Raspberry Pi. We then streamed NetFlow data and checked whether state machines are learned by the methods.

Experiment Setup

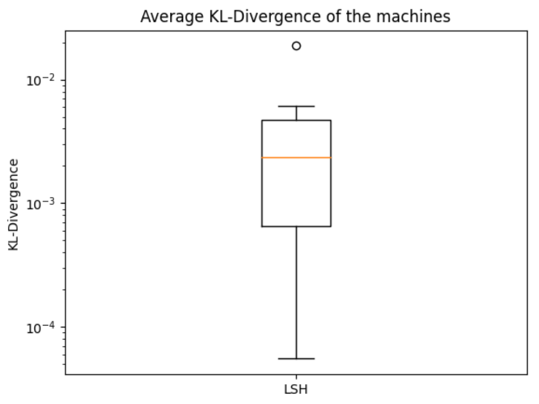
For this experiment, we used a Raspberry Pi Model 4 with 4 GB of RAM and 32 GB of storage, and we have installed Raspberry Pi OS on the device. The process of assessing whether state machines can be learned on an embedded device is as follows:

1. Stream malicious NetFlow data using Apache Kafka. The NetFlow data are downloaded from the Stratosphere IPS project [49].
2. Manually check whether state machines were learned.
3. Check how much do the state machines, that were learned using LSH, differ from the ones that were learned using the original method. The comparison of the state machines is the same as the one that was done in Subsection 4.2.1

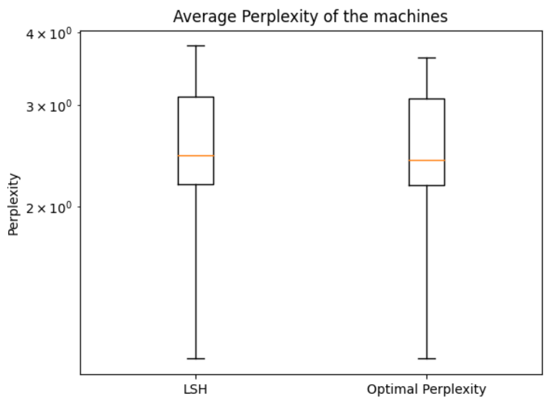
Analysis of the Results

After running both methods on the Raspberry Pi, we managed to learn state machines on the Raspberry Pi using both methods. We managed to learn state machines from all four datasets that were used for this experiment. Figure 6.1 shows the difference between the state machines that were learned using LSH and the ones that were learned using the original method for the dataset of Dridex. The results of the other datasets can be found in Appendix C.

Looking at the results and comparing them to the results of Subsection 4.2.1, we see that the KL-Divergence between the state machines are slightly higher when they are learned on the Raspberry Pi. An explanation for the higher difference could be due to randomness that is used in LSH which could have caused more mistakes to be made when a state is being merged. The randomness of LSH makes it harder to guarantee that the results are always the same.



(a) KL-Divergence between the state machines that were learned using LSH and the ones that were learned using the original method.



(b) Perplexity between the state machines that were learned using LSH and the ones that were learned using the original method.

Figure 6.1: KL-Divergence and perplexity between the state machines that were learned using LSH and the ones that were learned using the original method. The state machines were learned from the dataset of Dridex.

Though the KL-Divergence between the state machines are higher when they are learned on the Raspberry Pi, we see that the perplexity values of the state machines come close to the optimal perplexity values and the values do not differ much from the values that we have seen in Subsection 4.2.1. The differences can again be explained by the usage of randomness in LSH.

Conclusion of the Experiment

In this experiment, we wanted to evaluate whether it is possible to learn state machines in real-time on an embedded device. We have chosen to use the Raspberry Pi Model 4 as the device for learning state machines in real-time. To assess whether we can learn state machines, we have exported both versions of the method and learned state machines using each method. We have compared the state machines between the two methods in order to assess how much the state machines that were learned using LSH differ from the ones that were learned using the original method. Based on the results, the KL-Divergence values between the state machines were slightly when the state machines are learned on the Raspberry Pi. The slightly higher KL-Divergence values can be explained by the randomness that is used in LSH; the randomness cannot guarantee that the results are always the same. Though the KL-Divergence values are slightly higher, the perplexity values come very close to the optimal perplexity values. Thus the results show that we can still learn state machines that are close to the ones that were learned using the original method. The results also show that we can learn state machines in real-time from a stream of network traffic data on a small embedded device.

6.1.2. Assessing the Scalability of the Method on the Raspberry Pi

Being able to run the method on a Raspberry Pi raises a question on how scalable is the method when it is run on a device that has less resource than a desktop computer or a laptop. Schouten initially designed the method to be used on desktop computer or laptop, so it is questionable whether this method is scalable on a Raspberry Pi. In this experiment, we assessed the scalability of the method by computing the throughput of the method. This is done by streaming multiple datasets through Apache Kafka and recording how long the method took to process all flows. As our goal for using LSH is to introduce optimisations on the original method that was developed by Schouten, we only used the version that utilises LSH for this experiment.

Experiment Setup

For this experiment, we have selected and used three different datasets from the Stratosphere IPS project to compute the throughput of the method. Each dataset contains a different number of flows; the smallest dataset contains 62194 flows, the second-largest dataset contains 149911 flows and the largest dataset contains 324569 flows. The throughput is computed as follows:

1. Stream dataset fifty times using Apache Kafka.
2. For each run record the time the method took to process all the flows.

3. Compute the throughput of each run using the formula that is shown in equation 3.4.

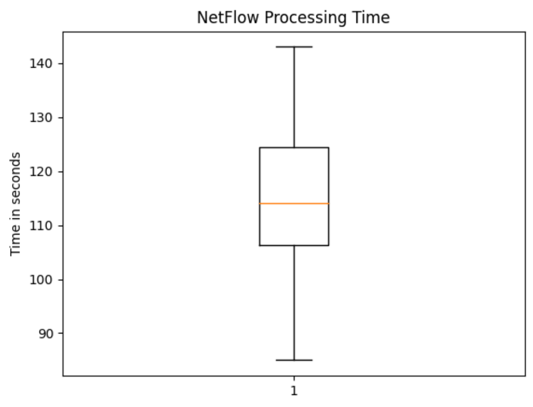
ANALYSIS OF THE RESULTS

Figure 6.2 presents the processing time and throughput of the method on the smallest dataset. The results of the other two datasets can be found in Appendix C. Looking at the figures, we see that there is something strange; the method has achieved a higher throughput on larger datasets than on the smaller one. The method has achieved an average of 553 flows/sec on the smallest dataset, 1241 flows/sec on the second-largest dataset and 2445 flows/sec on the largest dataset.

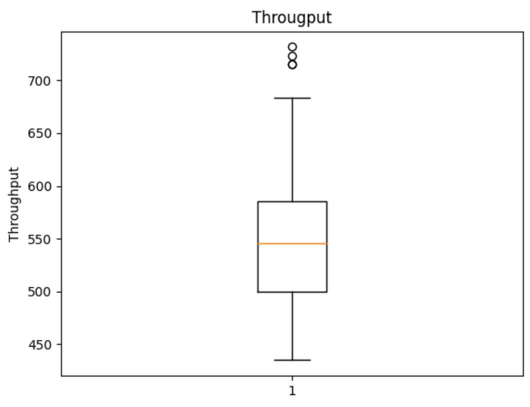
One possible explanation that could explain why the smallest dataset has achieved a much lower throughput than the second-largest dataset might be due to the delay that is caused for streaming the NetFlow data to the Raspberry Pi or the smallest dataset contains many more different connections between different pair of hosts. A state machine is learned for each pair of hosts and having many different combinations of these pair could mean that many more state machines need to be learned. This can affect the throughput of the method. Furthermore, the smallest dataset contains NetFlow data that describes the Necurs malware, which is a botnet. This matches our explanation as a device that is part of a botnet would need to send data to multiple different hosts.

The reason why the method has achieved the highest throughput on the largest dataset is due to the fact that the method initially has run out of heap space while processing the dataset. Our attempt for using Misra-Gries Summary algorithm to filter out the less common flows was not sufficient to reduce the memory consumption of the method on the Raspberry Pi. Our solution to this problem was to lower the size of the sketch vectors that are stored in each state. The sketch vectors are very large and it consumes a lot of memory as the number of states grows. By using smaller sketch vectors, the comparisons between the states are much faster as there are fewer entries to compare. This makes it faster to process the flows and therefore we got a higher throughput on the largest dataset. Though using a smaller sketch vector resolves the problem, it also increases the approximation error that can be made on the final state machine. Schouten has shown in his work that using a smaller sketch vector size results in a higher average KL-Divergence value when validating the state machines that were learned [25]. This means that the quality of a state machine drops when a smaller sketch vector is used. The result here shows that our attempt for using Misra-Gries Summary did not improve the memory consumption of the method and that a better memory optimisation is needed.

From the results, we also see that the highest throughput that we have achieved on the Raspberry Pi is much lower than the highest throughput that Schouten has achieved in his work on a desktop computer [25]. This was expected as the Raspberry Pi has much less resource compared to the setup that was used by Schouten.



(a) Processing time of the method



(b) Throughput of the method

Figure 6.2: The processing time and throughput of the method on the smallest dataset (62194 flows)

CONCLUSION OF EXPERIMENT

In this experiment, we wanted to assess the scalability of the method when it is run on a Raspberry Pi. From the results it shows that we were able to achieve an average throughput of 2445 flows/sec but to achieve this performance a trade-off has to be made; to be able to process more flows per second we would need to lower the size of the sketch vector of the states. This reduces memory consumption but it also drops the quality of the state machines that are learned using the method. It also shows that a better memory optimisation method is needed. Furthermore, we see that the highest throughput that we have achieved on the Raspberry Pi is much lower than the average highest throughput that Schouten has achieved on a desktop. This was expected as a Raspberry Pi has much less resource than the desktop that Schouten has used in his experiment.

6.1.3. Run-time Analysis of Finding a Most Similar Fingerprint on Raspberry Pi

The goal of using LSH is to improve the run-time of the process for finding a most similar fingerprint. As the Raspberry Pi has much less resource than a larger device such as a desktop computer, it can benefit from the run-time improvement that is provided by LSH. In this experiment, we want to evaluate whether LSH also provides the improvement that we have seen in Subsection 5.2.1 when the method is run on a Raspberry Pi.

Experiment Setup

The setup of this experiment is the same as the one that is described in Subsection 5.2.1. The difference in this experiment is that we ran both versions of the method on the Raspberry Pi to record the time for finding a most similar fingerprint.

Analysis of the Results

Figure 6.3 presents the run-time for finding a most similar fingerprint on the Raspberry Pi. We can see from the figure, that the version that uses LSH has a lower run-time compared to the original version. The version that uses LSH is still two orders of magnitude faster than the original version. This shows that LSH still provides an improvement on the run-time of the process even when it is run on the Raspberry Pi.

Furthermore, we also see that the run-time on the Raspberry Pi is higher than the run-time on a larger device. This was expected as the Raspberry Pi has less resource compared to a larger device.

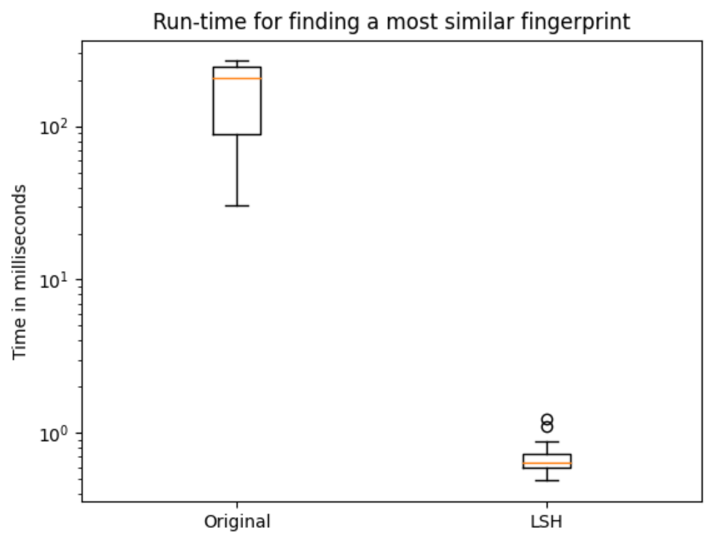


Figure 6.3: Run-time for finding a most similar fingerprint on the Raspberry Pi.

Conclusion of the Experiment

In this experiment, we wanted to evaluate whether LSH still provides an improvement on the run-time of the process for finding a most similar fingerprint when the method is run on a Raspberry Pi. Based on the results, we see that LSH still provides an improvement on the run-time when the method is run on the Raspberry Pi. Furthermore, we also saw that the run-time on the Raspberry Pi is higher than the run-time on a larger device. This was expected as the Raspberry Pi has less resource compared to a larger device.

6.2. EXPERIMENTS THAT USED ARTIFICIALLY GENERATED DATA

6.2.1. RUN-TIME ANALYSIS OF STATE-MERGING PROCESS ON THE RASPBERRY PI

As the goal of using LSH is to introduce an optimisation on the original method so that we run learn state machines more efficiently on an embedded device, we wanted to evaluate whether LSH provides any improvements in the run-time of the state-merging process when it is run on the Raspberry Pi.

EXPERIMENT SETUP

The setup of this experiment is the same as the second part of the experiment that is described in Section 4.1.3. The difference in this experiment is that we ran both versions of the method on the Raspberry Pi to record the run-time of the state-merging process.

ANALYSIS OF THE RESULTS

The average run-time of the state-merging process of both versions is shown in Figure 6.4. From the figure, we can see that both versions of the method ran slower on the Raspberry Pi than when it was run on a laptop (see Figure 4.5 for comparison). This was expected as the Raspberry Pi have less resource compared to the laptop we have used for the other experiment. This shows why the optimisation is needed for us to learn state machines efficiently on an embedded device.

From the figure, we also see that the run-time of both versions are quite close to each other, though our LSH approach has a slightly lower average run-time for all runs. It was expected that the average run-time between the two versions to come quite close to each other as this was also the case when both versions were run on a laptop. As mentioned in Section 4.1.3, this can be explained by the sizes of the state machines that we have used for the experiment.

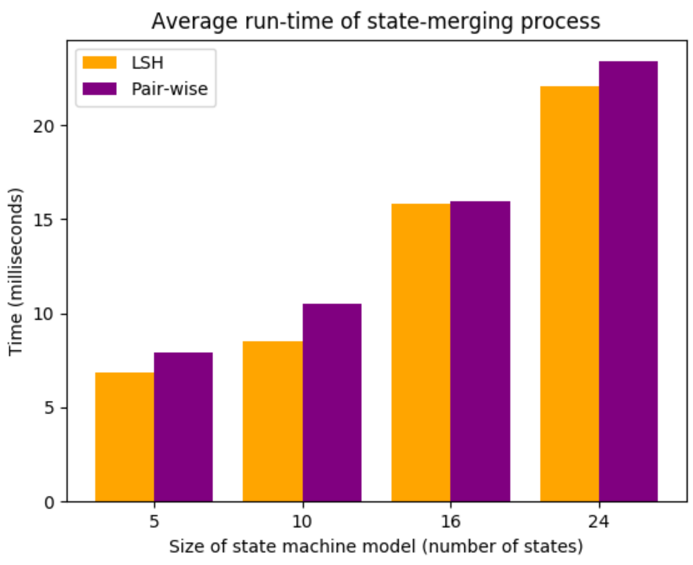


Figure 6.4: Average run-time of the state-merging process run on a Raspberry Pi.

CONCLUSION OF THE EXPERIMENT

In this experiment, we evaluated the run-time of the state-merging process when the method is run on the Raspberry Pi. From the results, we saw that the overall run-time of both versions is slower when it was run on a Raspberry Pi than when it was run on a laptop. This shows that optimisations are indeed needed for us to learn state machines efficiently on an embedded device. From the results, we also saw that the average run-time of both versions are quite close to each other, though our LSH approach had a slightly lower average run-time for all runs. LSH provides an improvement in the run-time when we are dealing with state machines that have a large number of states. As we could not learn a state machine that has more than 25 states, we cannot see an improvement in the run-time.

6.3. Conclusion on Learning State Machines on the Raspberry Pi

The ultimate goal of this thesis is to investigate whether it is possible to learn state machines in real-time on an embedded device from a stream of network traffic data. We have used a Raspberry Pi 4 as the device on which we ran our experiments. From the results of our experiments, it shows that it is possible to learn state machines in real-time from a stream of network traffic data. The state machines that were learned using the version that utilises LSH are still very similar to the ones that are learned using the original version.

Though we could learn state machines in real-time on the Raspberry Pi, the highest throughput that we have achieved was much lower than the average highest throughput that Schouten has achieved on a desktop computer. This was expected as the Raspberry Pi has much less resource than the desktop computer that Schouten has used. Additionally, the throughput that have achieved on the Raspberry Pi was possible after we have used a much smaller sketch vector for the states. This reduces the memory consumption but it also reduces the quality of the state machines (as shown in the Schouten's work). It is also shows that our attempt for using the Misra-Gries Summary algorithm was not sufficient to reduce the memory consumption and that a better memory optimisation method is needed.

Based on the results, we also saw our version that uses LSH had a lower run-time for the state-merging process but it does not differ a lot from the run-time of the original version. This was expected as we have seen the same results in Section 4.1.3. LSH provides an improvement in the run-time when we are processing state machines that have a large number of states. In our experiment we were not able to learn state machines that have more 25 states and therefore the run-time our version is the same as the original version. This shows that our LSH method does not provide an improvement in the run-time of the state-merging process when we are learning state machines on the Raspberry Pi.

Furthermore, we have also seen that LSH has a lower run-time for finding a most similar fingerprint than the original version. This shows that LSH still provides an improvement on the run-time of the process when it is run on the Raspberry Pi.

7 Discussion

In this chapter, we provide a discussion on the shortcomings of our method and our evaluation. By discussing the shortcomings, we can identify the improvements that can be done on our method.

7.1. Limitations of our LSH approach to learn State Machines

One of the limitations of our LSH method is the way how we construct the hashes for the states and state machines. For each state or state machine, we take its vector representation and compute the dot products between its vector representation and the random vectors that are used to create the hyperplanes. If a dot product is a positive value then it will get a 1, else it will get a 0. This method does not take the distances between the dot products into account; all positive dot product values will get 1 and all negative dot product values will get a 0. If the distance between two dot products is very large, our method of hashing will assign a 1 to them both. Thus this could mean that two states or state machines that are possibly dissimilar to each other will get the same value if their dot products are both positive or negative. This increases the chance that two dissimilar items will be put into the same bucket.

Being able to take the distances of the dot products into account we might be able to decrease the chance that two dissimilar items will be put into the same bucket. This might then improve the clustering performance of LSH for states or state machines.

For us to take distances of dot products into account when hashing, we would need to define several splits within the space of the dot product values. This is so that we can give dot products, that are very distant from each other, different hash values. Dot products that are close to each other will get the same hash value. For us to define these splits, we would first need to run experiments to evaluate which are the best thresholds to use for the splits. Due to the time constraints of this project, we did not investigate whether this method would improve the clustering performance of our LSH method. Furthermore, we use random vectors to define the hyperplanes and compute the dot products. This means that the dot products change every time and thus this makes it harder to define the splits within the space of the dot product values.

Another limitation of our LSH method is that our approach does not produce the same clustering results every time. This is because we are using random vectors to define the hyperplanes and to compute the dot product. This means that the dot product values are not the same every time and therefore the hashes are also not the same every time. Thus the random vectors bring inconsistency in the clustering results of our LSH method. Though the usage of random vectors brings inconsistency in our clustering results, the concepts of LSH is to hash similar items with a high probability into the same bucket. In our case, we look at the cosine distance between the vector representations of the states/state machines when we are computing the dot products. If two states/state machines are similar, then the values of the random vectors should not affect the results; if two vectors are similar, then they would point the same direction when we are computing the cosine distance. If they are not similar, then these two vectors would point in a different direction when we are computing the cosine distance.

One solution to resolve the inconsistency of the results is to use a seed for the generation of the random vectors. This solution requires us to run experiments and evaluate which seed is the best one for us to use. Due to the time constraints of this project, we did not investigate whether this solution would resolve the inconsistency of the results.

7.2. Limitation of our Evaluation

One of the limitations of our evaluation is that we did not investigate what is the optimal value to use for the number of random vectors. In our implementation of LSH, we can adapt the number of random vectors to use for clustering. The number of random vectors also influences how many possible buckets could be used for clustering. As we are using bitstrings as hashes for the states/state machines, the maximum number of buckets that can be used is calculated as follow: $maxiumum\ number\ of\ buckets = 2^{number\ of\ random\ vectors}$. Due to the time constraints of this project, we did not investigate which is the optimal number of random vectors to use for clustering. Furthermore, the focus of our work was to investigate whether it is possible for to cluster state/state machines using LSH and not to figure out which is the optimal number of random vectors to use for LSH.

Another limitation of our evaluation is that we did not use the same PAutomaC state machines, that were used in the other experiments, to evaluate whether we can cluster state machines using their buckets distributions. The state machines that were used for clustering state machines based on their buckets distributions did not have obvious clusters in the state machines itself. Unlike the other experiments, we did not use five base state machines to create the variants. This made it difficult to conclude whether the approach was indeed not suitable for clustering state machines. The reason why we opted to use other PAutomaC state machines, is because for this experiment we cannot read the state machines directly from the file. We needed to stream the training data of the state machines using Apache Kafka and then learn the state machines. For us to stream the variants that we created for the other experiments, we would need to manually generate the training data for the variants. We did not know how we can create the training data which would guarantee that a particular variant state machine would be learned using the method. Furthermore, the hashing was done on the states. The locality-sensitive properties hold for the states in the buckets but these do not carry to the state machine itself.

In our experiments, we only compared our LSH method to the original method that was designed by Schouten and to KMeans Clustering. We did not compare our solution to any other state-of-the-art clustering solutions. The reason why we did not do any comparison with another state-of-the-art solution was due to the implementation of the whole pipeline. The whole pipeline was written to support the learning of state machines in real-time from a stream of NetFlow data. To be able to do a comparison with state-of-the-art solutions, we would need to investigate whether the solution is compatible with the implementation of the pipeline. Due to the time constraints of this project, we opted not to compare our solution to a state-of-the-art solution.

The ultimate goal of this work was to investigate whether it is possible to learn state machines from in real-time on an embedded device. In our evaluation, we have shown that it is possible but we did not run many extensive experiments on the Raspberry Pi. This was due to the outbreak of Covid-19, which has caused a delay in getting the device to run the experiments. Due to the delay that was caused by the pandemic, we were not able to run more extensive experiments on the Raspberry Pi together with our LSH method.

8 Conclusions & Future Work

This chapter concludes the research that was done in this thesis by answering the research questions that are listed in Section 1.3, using the results from Chapters 4, 5 and 6. We finalise this chapter by listing some future work.

The goal of this work to investigate whether it is possible to learn state machines in real-time on a small embedded device, using the method that was proposed in the work by Schouten. As the method is designed to work on larger devices that have much more resource than a smaller embedded device, the method could run into performance issues when it is run on a smaller embedded device. Thus we would need to make some modifications to the original method to resolve the possible performance issues that the method might encounter when it is run on an embedded device.

The main research question that we have formulated for this work was the following:

RQ: How can we learn state machines in real-time from a stream of network traffic data on an embedded device?

We will first answer the subquestions that were formed from this main research question and then use the answer of the subquestions to answer the main research question.

SQ1: What kind of modifications do we need to make to the original method that is proposed by Schouten?

As a smaller embedded device has limited resources compared to a desktop computer or a laptop, some modifications would need to be made on the original method to resolve some of the possible performance issues it might encounter when it is run on a smaller embedded device. In this work, we have made two modifications to the original method.

The first modification was to use the concepts of LSH to improve the run-time of the state-merging process. When states are being merged to learn the smallest state machine, pair-wise comparisons are done between the states to find which pair of states can be merged together. This process has a quadratic run-time and the run-time performance will be impacted as the number of states grows larger. In our approach, we have used LSH to cluster similar states together. By clustering the states together, there is no need to do all the pair-wise comparisons between the states, instead we now only need to search within a particular bucket to find a most similar state for a merge. This approach allows us to improve the run-time of the state-merging process as we have eliminated many of the comparisons that need to be done for a merge.

The second modification was to use the Misra-Gries Summary algorithm to reduce the memory footprint of the method. In the implementation of the system, the NetFlow data from the stream is grouped based on the IP-pair in the connection and the protocol that was used in the connection. A state machine is learned for each group and it is stored in memory. The issue with this implementation is that many state machines are learned and stored in memory if there is a large number of groups. This is problematic for a small embedded device as it does not have a lot of resources. We have used the Misra-Gries Summary algorithm to keep track of the top ten most frequent group of NetFlow data. The infrequent groups are dropped and would not be processed. This way we reduce the number of state machines that will be learned and thus also reduce the memory footprint of the method.

SQ2: How can we cluster states using LSH?

To cluster states using LSH, we need to represent a state as a vector of numerical values. This vector is used to create the hash for a state which will be used to put a state in the corresponding cluster. In the implementation of the system, each state stores a Count-Min sketch vector. This vector contains an approximate count of the sequences that have occurred in that state and it is used in the state-merging process to check whether it can be merged with another state. As this vector is used for merging and it contains numerical values, it seemed natural for us to also use this vector for hashing. Thus we use the Count-Min sketch vector of a state to create its hash and put it in the corresponding cluster.

SQ3: How much do the state machines, that are learned using the LSH approach, differ from the ones that are learned using the original method?

From the results of the experiment that is described in Section 4.2.1, the state machines that were learned using our LSH approach are just ever so slightly different than the ones that are learned using the original method. When we have increased the number of buckets, the difference also increased but the difference is still close to zero. Our LSH approach increases the approximation error for learning state machines but the results show that the effect on the final state machine is low.

SQ4: How can we cluster state machines using LSH?

Clustering machines using LSH was not a trivial task. The most difficult part of the process was to find a vector representation of a state machine. In our approach, we have tried three different methods to create the vector representation of a state machine:

1. Using the state-transition table representation of the state machine.
2. Using the language of the state machine.
3. Using the distribution of the states of a state machine over the buckets.

Just as with the clustering of the states, the vector representation of a state machine is used to create its hash and to put it in the corresponding cluster. Based on the results that we have seen in Chapter 5, the best method for creating the vector representation for a state machine was the second method as this method has achieved the highest accuracy and the highest Silhouette Coefficient score.

SQ5: How does LSH perform in comparison to KMeans Clustering?

In our evaluation, we compared the clustering performance of LSH to the clustering performance of KMeans Clustering. We used KMeans Clustering as the baseline for the clustering results as we expect KMeans Clustering to have a better clustering performance than LSH. From the results of Chapters 4 and 5, we saw that KMeans Clustering indeed had a better clustering performance than LSH. Though LSH had a lower performance compared to KMeans Clustering, it had almost all cases much better performance than random guessing. Only in one case, LSH was performing almost just as bad as random guessing. This was when we tried to use the distribution of the state over the buckets to cluster state machines.

The lower performance of LSH can be explained by how the clustering is done for both methods. In KMeans Clustering, the distance is computed between the data point to assign the data points to their corresponding clusters. LSH does not directly take the distances between the data points into account when assigning the data points to their corresponding clusters. This means that some information is lost between the data points and this increase the chance for an error to occur. Additionally, the hash functions of LSH are chosen in such a way that maximises the collision between data points. By increasing the chance for a collision, it also increases the chance that two data points that are far away from each other to be put into the same cluster. Thus increasing the chance of collisions also increases the chance for an error to occur.

Though LSH had a lower performance in comparison to KMeans Clustering, it managed to achieve high accuracy for the clustering of states/state machines. This shows that although LSH does not perform as well as KMeans Clustering, LSH can still cluster states/state machines well.

SQ 6: How does the new version of the method, that utilises LSH, perform on a smaller embedded device?

From the results that are shown in Chapter 6, we saw that we were able to learn state machines in real-time from a stream of NetFlow data. The highest throughput that we have achieved was on average 2445 flows/sec and we were able to achieve this result after using a much smaller sketch vector for the states. This lowers the memory footprint of the method and the processing time of each flow but this does come with a price; lowering the size of the sketch vectors that are used in the states also lowers the quality of the state machines that will be learned. This effect is shown in the work that was done by Shouten. Thus to achieve a higher throughput on the Raspberry Pi, a trade-off has to be made between how fast we can process each flow and the quality of the state machines that will be learned.

Comparing the throughput that we have achieved on the Raspberry Pi to the throughput that Schouten has achieved on a desktop computer, we see that the Schouten has managed to achieve a much higher average throughput. This was expected as the Raspberry Pi has much less resource compared to the desktop computer that Schouten has used in his experiment.

One of the goals for LSH was to improve the run-time of the state-merging process by eliminating many of the pair-wise comparisons that need to be done between all states to find a most similar state for a merge. When compared the run-time of the state-merging process of the original version of the method to this new version that uses LSH, we see that LSH did not provide any improvements in the run-time of the state-merging process. This can be explained by the fact that we cannot learn state machines with more than 25 states. LSH would only show a speedup if we are dealing with a large number of states. As we are dealing with a low number of states, we do not see any improvement in the run-time.

Although we were not able to improve the run-time of the state-merging process using LSH, we were able to use LSH to improve the run-time of the process for finding a most similar fingerprint to match a state machine. From the results that we have seen in Chapter 6, LSH was two orders of magnitude faster than the original method for finding a most similar fingerprint to match a state machine.

RQ: HOW CAN WE LEARN STATE MACHINES IN REAL-TIME FROM A STREAM OF NETWORK TRAFFIC DATA ON AN EMBEDDED DEVICE?

To learn state machines in real-time from a stream of network traffic data on an embedded device, we used the method that was proposed by Schouten. Since the method was designed to work on a larger device such as a desktop computer or laptop, the method might run into performance issues when it is run on a smaller embedded device that has much less resource. To resolve the possible performance issues that the method might encounter while running on a smaller device, some modification would need to be made to the method. In this work, we have made two modifications to the original method; we used the concepts of LSH to improve the run-time of the state-merging process and we used the Misra-Gries Summary algorithm to reduce the memory footprint of the method.

Based on the results of our evaluations, LSH did not provide much improvement in the run-time of the state-merging process. This could be explained by the fact that we cannot learn state machines with more than 25 states. LSH would only show an improvement in the run-time when we are dealing with a large number of states.

Although we were not able to improve the run-time of the state-merging process using LSH, we were able to use LSH to improve the run-time for finding a most similar fingerprint to match a state machine. LSH was two orders of magnitude faster than the original method for finding a most similar fingerprint for a match.

When we look at the state machines that were learned using LSH, they are very similar to the state machines that were learned using the original method. This shows that although using LSH increases the approximation error, the effect on the final state machine is low.

In our attempt to run the method on the Raspberry Pi, the highest throughput that we managed to achieve was on average 2445 flows/sec. We managed to achieve this result by using a lowering the size of the sketch vectors that were used in the states and by using the Misra-Gries Summary Algorithm to process only the most frequent flows. This lowered the memory footprint of the method and the processing time of each flow but this solution comes with a price; lowering the sketch sizes of the states also lowers the quality of the state machines that will be learned. Thus there is a trade-off to be made. This also shows that a better memory optimisation method is needed.

8.1. FUTURE WORK

USING OTHER LSH METHODS

In this work, we only used the Random Hyperplanes approach to hash and cluster state/state machines. It would be interesting to use other LSH methods to cluster state/state machines and compare their clustering performance to the one that we have used in this work.

Using Other Methods To Hash State Machines

In this work, we have used three different methods to construct the vector representation of a state machine and use it to hash the state machine. There is an overhead in two of the methods that we have used in this work; for one of the method we would have to construct the largest possible state transition table and for the other method we would have to generate strings using the language of the base state machines. Space is needed to store the tables and strings. It would be interesting to see whether there is a method with much less overhead than the ones that we have used in this work. It would also be interesting to see how well it performs in comparison to the methods that were used in this work.

Comparison to More Clustering Algorithms

In this work, we have only used KMeans Clustering and Random Clustering as a comparison to our LSH approach. There exist many more clustering algorithms and it would be interesting to see how our LSH approach compares to the other clustering algorithm. We suspect our LSH approach would not have a better performance than the other clustering algorithm.

More extensive experiments on the Raspberry Pi 4

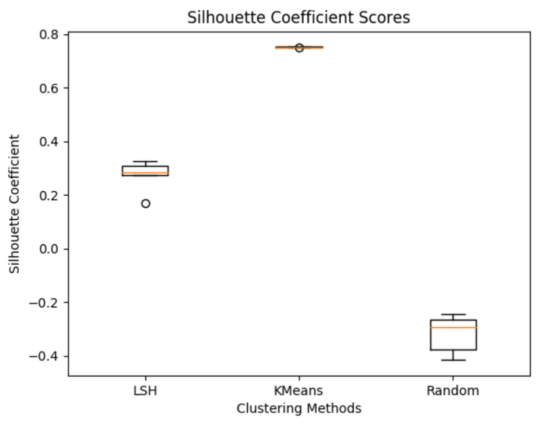
The outbreak of Covid-19 has delayed the process of running experiments on the Raspberry Pi. Due to time constraints, we were not able to run more extensive experiments on the Raspberry Pi 4. Given more time, we could have come up with more experiments to do better evaluations on the Raspberry Pi 4.

Better Memory Optimisation Method

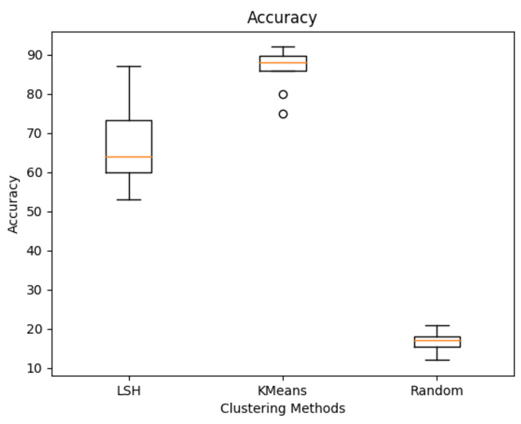
In this work, we have attempted to reduce the memory consumption while learning state machines from a stream of NetFlow data. We have used Misra-Gries Summary algorithm to filter out the infrequent flows and only processes the top ten most occurring flows. This was shown to be insufficient and we needed to also reduce the size of the sketch vectors that are used by the states. This lowers the memory footprint of the method but it also lowers the quality of the state machines that will be learned. Thus a better memory optimisation is needed.

A
More Clustering Results of States

A.1. Clustering Artificially Generated Vectors

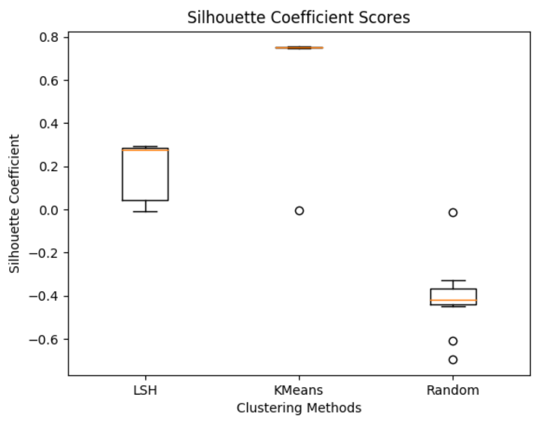


(a) Silhouette Coefficient scores for clustering the vectors using six clusters.

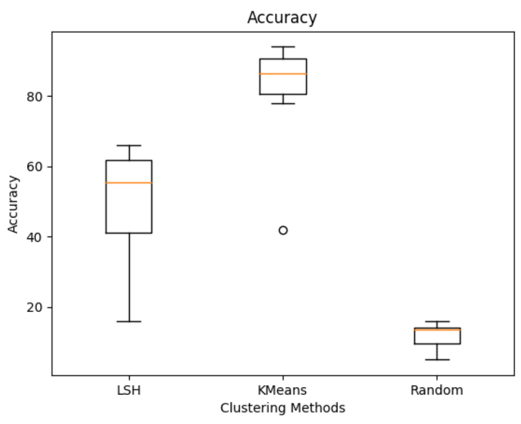


(b) Accuracy for clustering the vectors using six clusters.

Figure A.1: Silhouette Coefficient scores and accuracy of each clustering for clutering the vectors using six clusters.

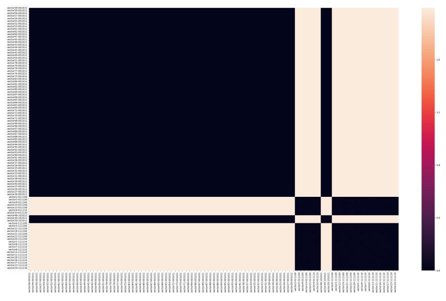


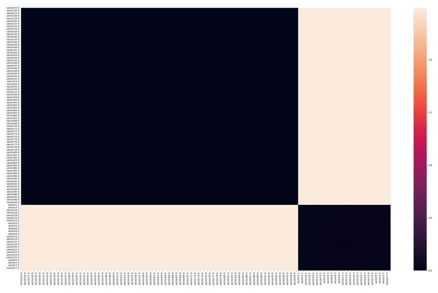
(a) Silhouette Coefficient scores for clustering the vectors using eight clusters.



(b) Accuracy for clustering the vectors using eight clusters.

Figure A.2: Silhouette Coefficient scores and accuracy of each clustering for clutering the vectors using eight clusters.


(a) Clustering results of LSH


(b) Clustering results of KMeans Clustering

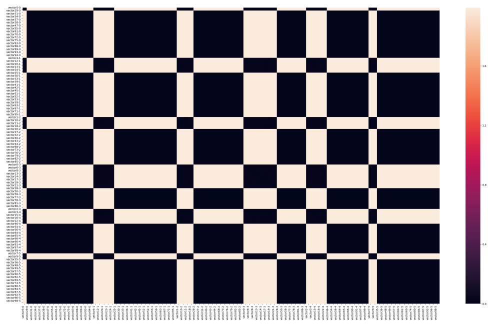
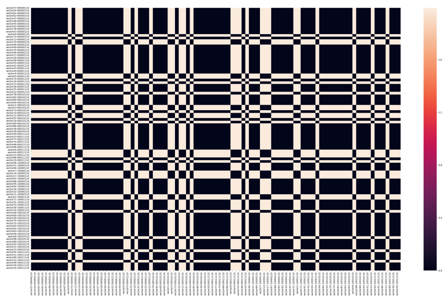
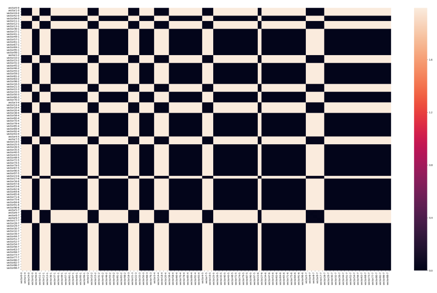

(c) Clustering results of Random Clustering

Figure A.3: Clustering results of LSH, KMeans Clustering and Random Clustering. Six clusters were used for each method.


(a) Clustering results of LSH


(b) Clustering results of KMeans Clustering

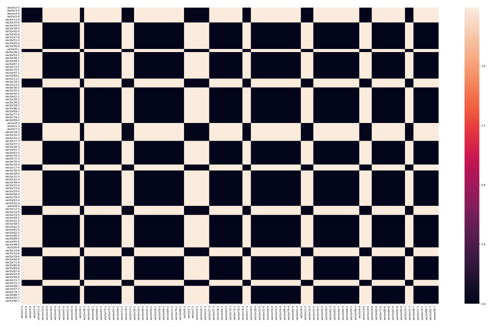
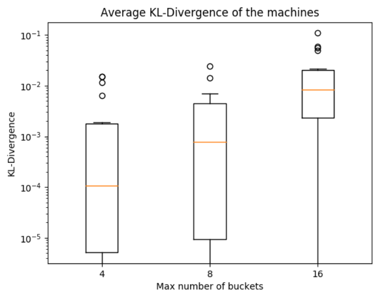
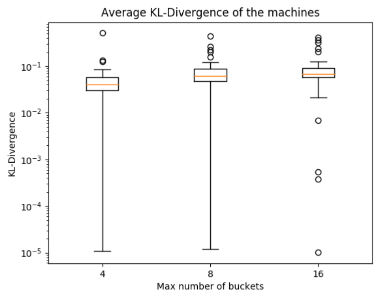

(c) Clustering results of Random Clustering

Figure A.4: Clustering results of LSH, KMeans Clustering and Random Clustering. Eight clusters were used for each method.

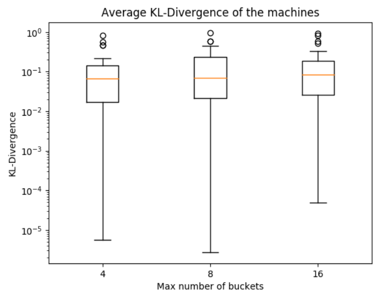
A.2. Comparison Results of State Machines



(a) Average Kullback-Leibler divergence for the state machines of Emotet.

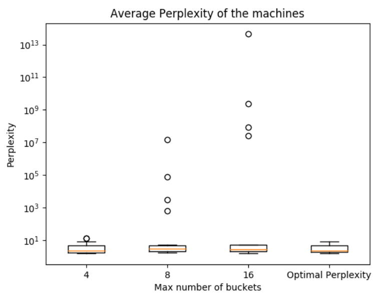


(b) Average Kullback-Leibler divergence for the state machines of Necurs.

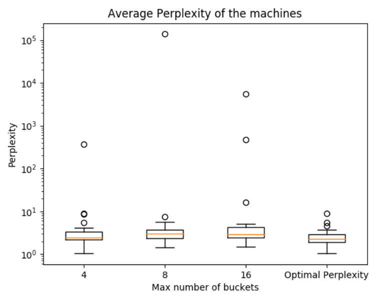


(c) Average Kullback-Leibler divergence for the state machines of TrickBot.

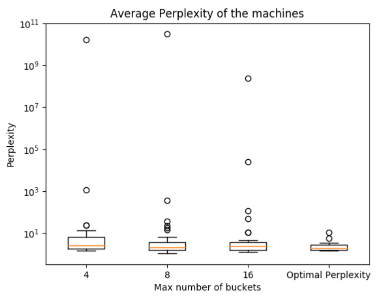
Figure A.5: Boxplots showing the average Kullback-Leibler divergence of the state machines that were learned for each malware plotted in logarithmic scale.



(a) Average Perplexity for the state machines of Emotet.



(b) Average Perplexity for the state machines of Necurs.



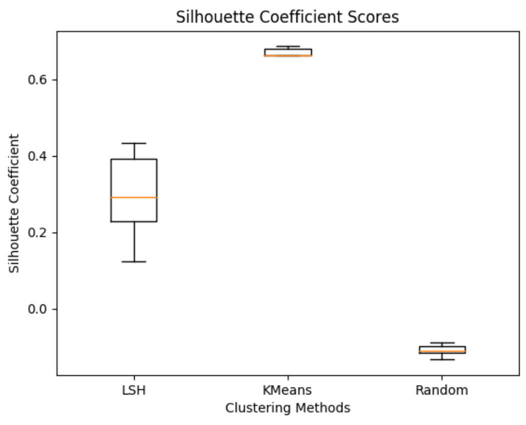
(c) Average Perplexity for the state machines of TrickBot.

Figure A.6: Boxplots showing the average Perplexity of the state machines that were learned for each malware plotted in logarithmic scale.

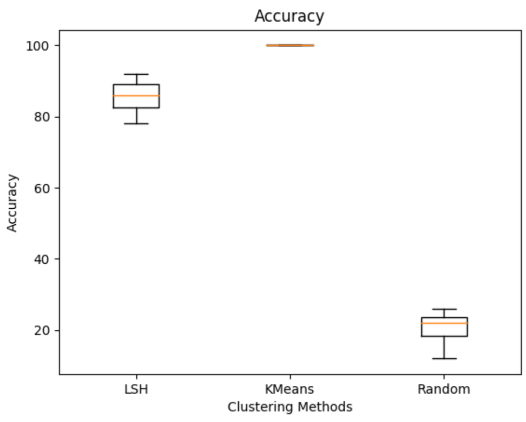
B

More Clustering Results of State Machines

B.1. Clustering Results Using the State Transition Table of the State Machines

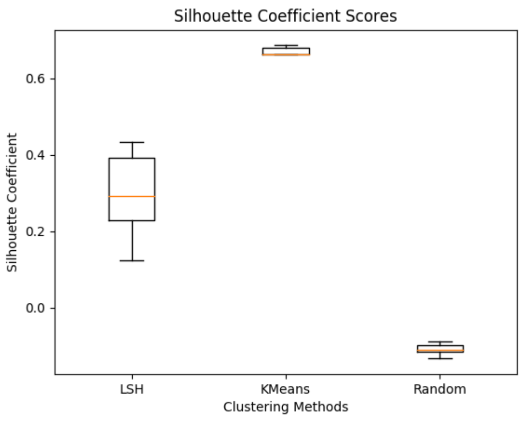


(a) Silhouette Coefficient scores for clustering the state machines using seven clusters.

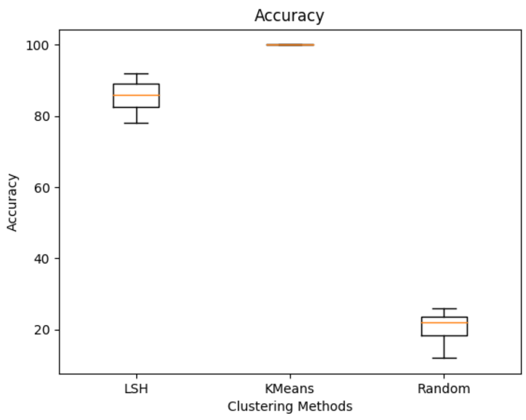


(b) Accuracy for clustering the state machines using seven clusters.

Figure B.1: Silhouette Coefficient scores and accuracy of each clustering method for clustering state machines using seven clusters.



(a) Silhouette Coefficient scores for clustering the state machines using ten clusters.



(b) Accuracy for clustering the state machines using ten clusters.

Figure B.2: Silhouette Coefficient scores and accuracy of each clustering method for clustering state machines using ten clusters.

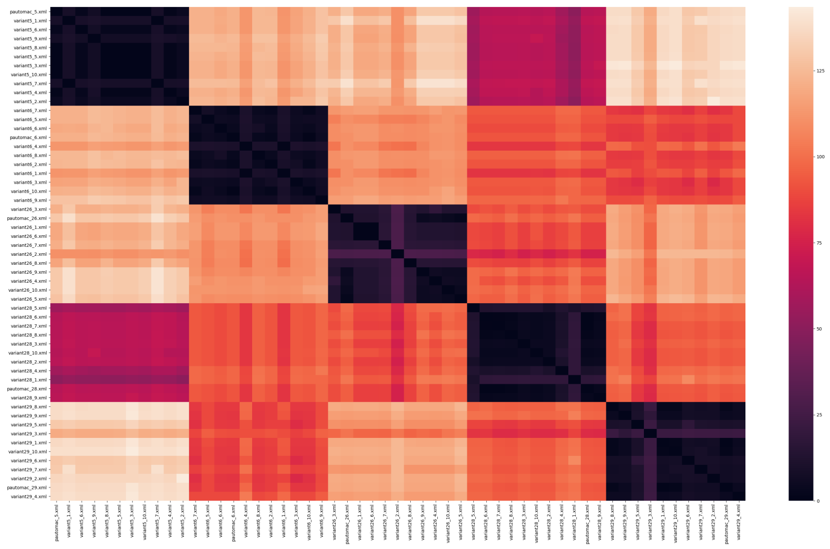
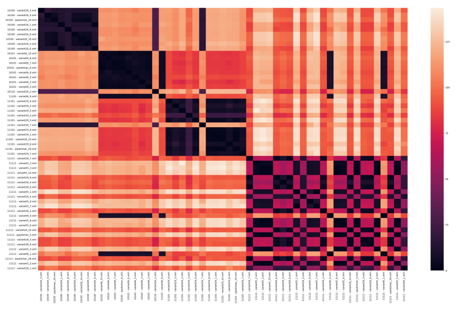
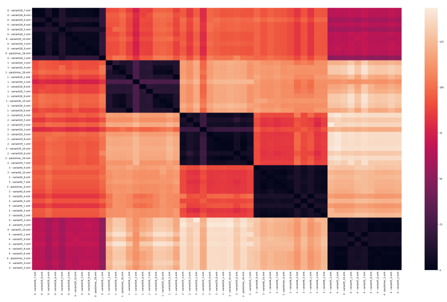


Figure B.3: Heat map generated from the unsorted distance matrix.


(a) Clustering results of LSH


(b) Clustering results of KMeans Clustering

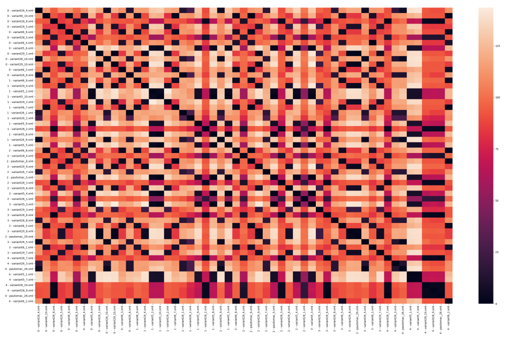
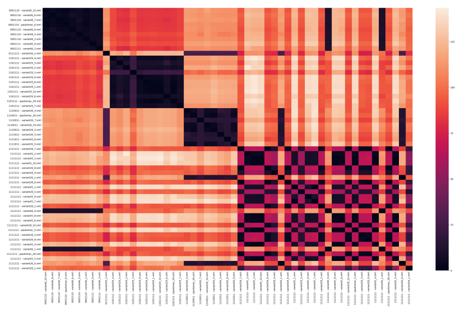
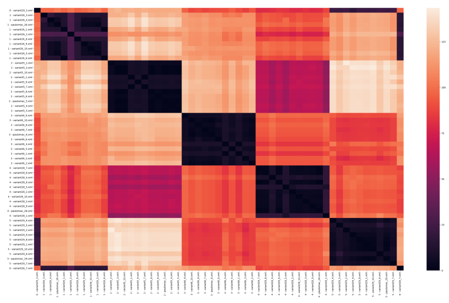

(c) Clustering results of Random Clustering

Figure B.4: Clustering results of LSH, KMeans Clustering and Random Clustering. Five clusters were used for each method.


(a) Clustering results of LSH


(b) Clustering results of KMeans Clustering

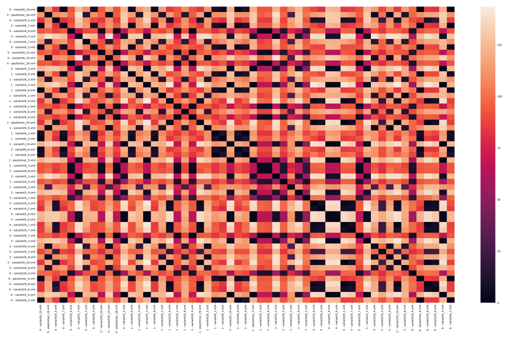
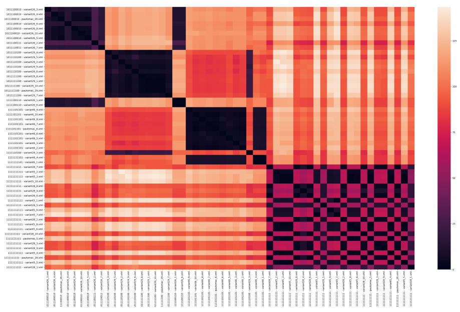
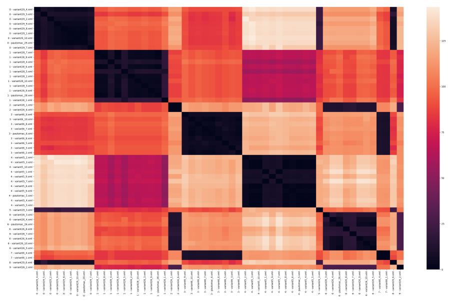

(c) Clustering results of Random Clustering

Figure B.5: Clustering results of LSH, KMeans Clustering and Random Clustering. Seven clusters were used for each method.


(a) Clustering results of LSH


(b) Clustering results of KMeans Clustering

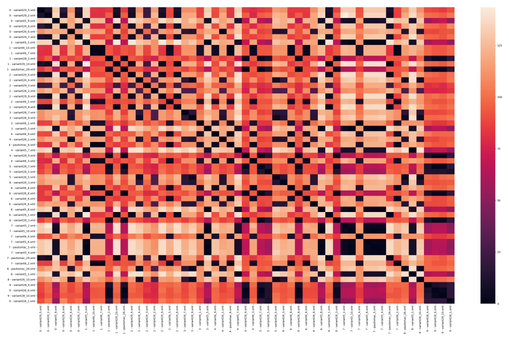
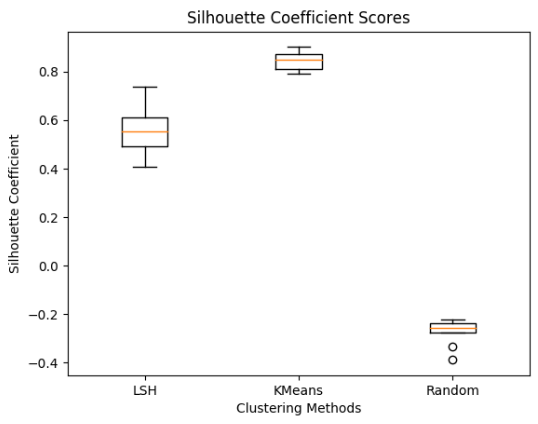
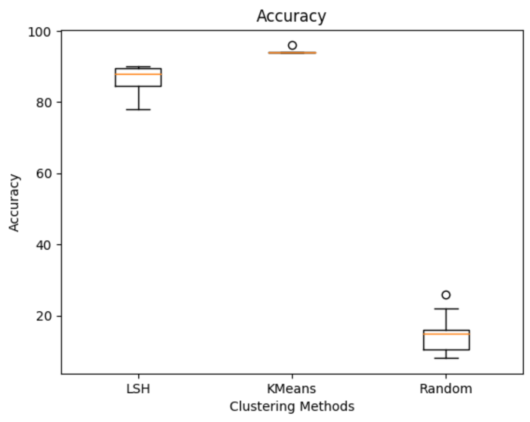

(c) Clustering results of Random Clustering

Figure B.6: Clustering results of LSH, KMeans Clustering and Random Clustering. Ten clusters were used for each method.

B.2. CLUSTERING RESULTS USING LANGUAGE OF THE STATE MACHINES

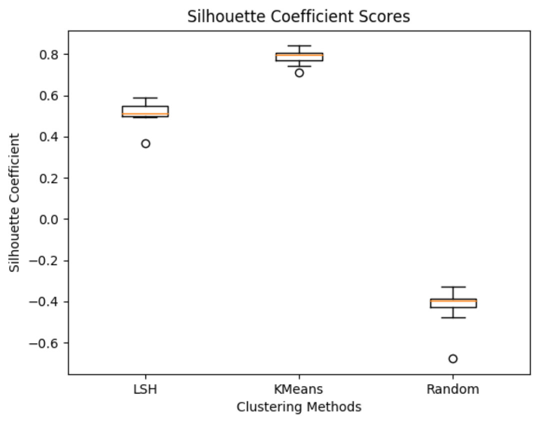


(a) Silhouette Coefficient scores for clustering the state machines using seven clusters.

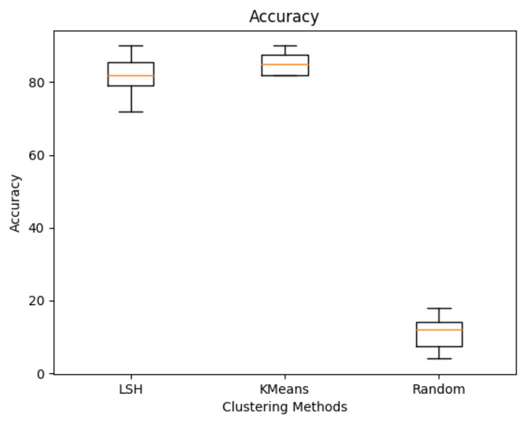


(b) Accuracy for clustering the state machines using seven clusters.

Figure B.7: Silhouette Coefficient scores and accuracy of each clustering method for clustering state machines using seven clusters.



(a) Silhouette Coefficient scores for clustering the state machines using ten clusters.



(b) Accuracy for clustering the state machines using ten clusters.

Figure B.8: Silhouette Coefficient scores and accuracy of each clustering method for clustering state machines using ten clusters.

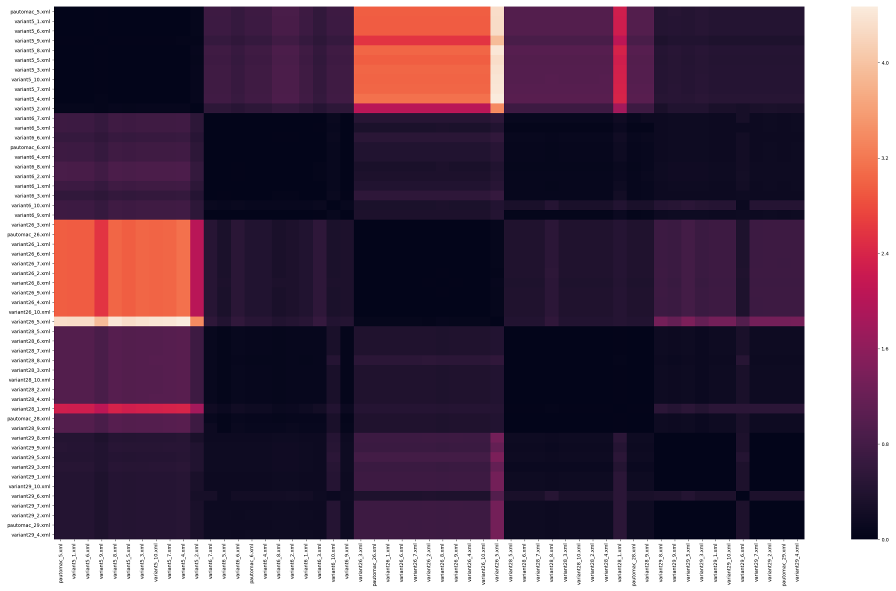
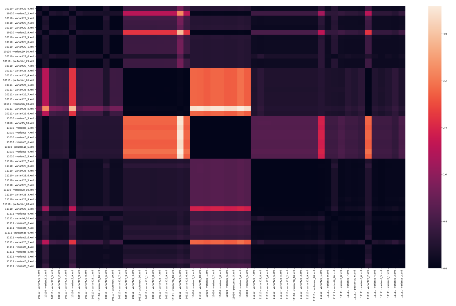
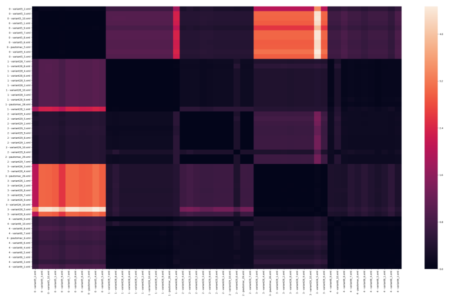


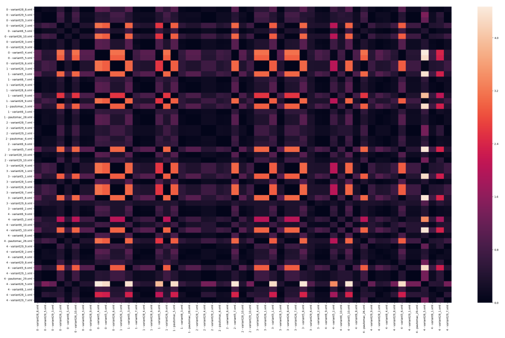
Figure B.9: Heat map generated from the unsorted distance matrix.



(a) Clustering results of LSH

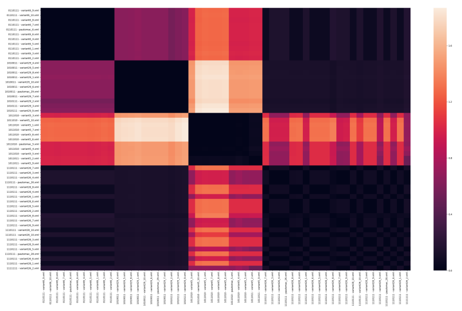


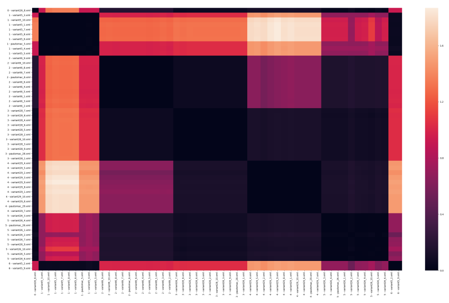
(b) Clustering results of KMeans Clustering



(c) Clustering results of Random Clustering

Figure B.10: Clustering results of LSH, KMeans Clustering and Random Clustering. Five clusters were used for each method.


(a) Clustering results of LSH


(b) Clustering results of KMeans Clustering

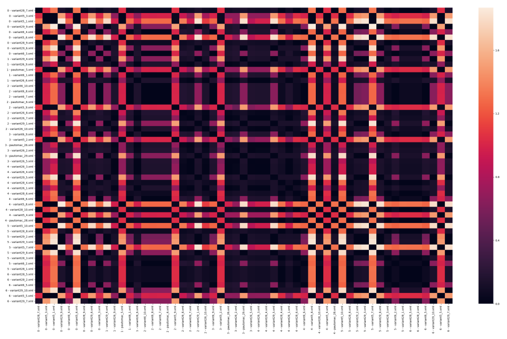
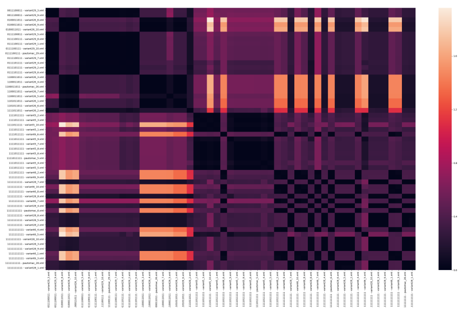
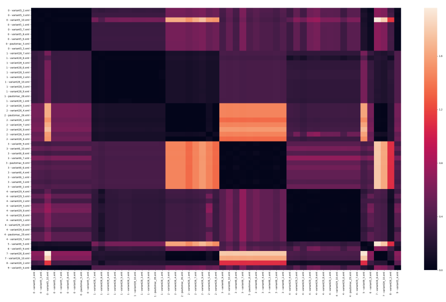

(c) Clustering results of Random Clustering

Figure B.11: Clustering results of LSH, KMeans Clustering and Random Clustering. Seven clusters were used for each method.


(a) Clustering results of LSH


(b) Clustering results of KMeans Clustering

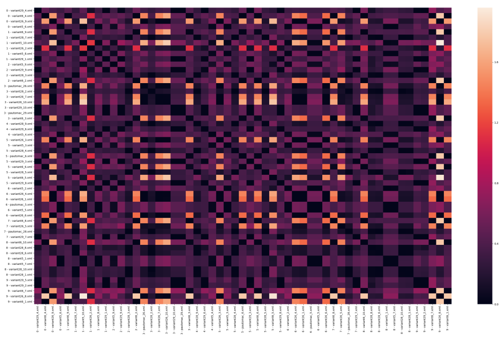
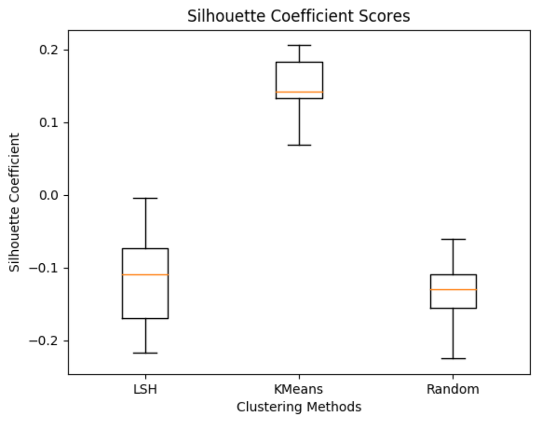
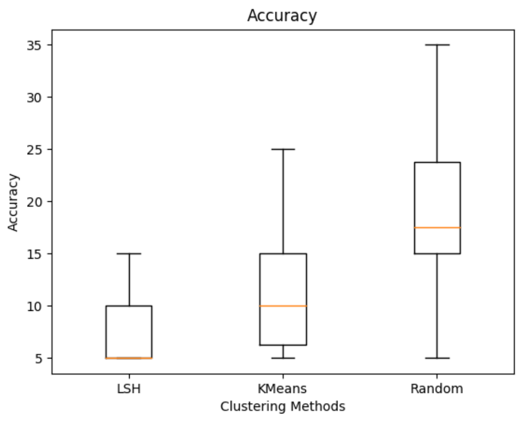

(c) Clustering results of Random Clustering

Figure B.12: Clustering results of LSH, KMeans Clustering and Random Clustering. Ten clusters were used for each method.

B.3. Clustering Results Using the Distribution of States Over the Buckets

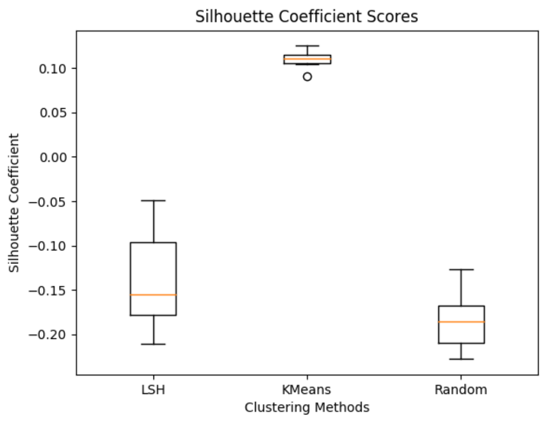


(a) Silhouette Coefficient scores for clustering the state machines using six clusters.

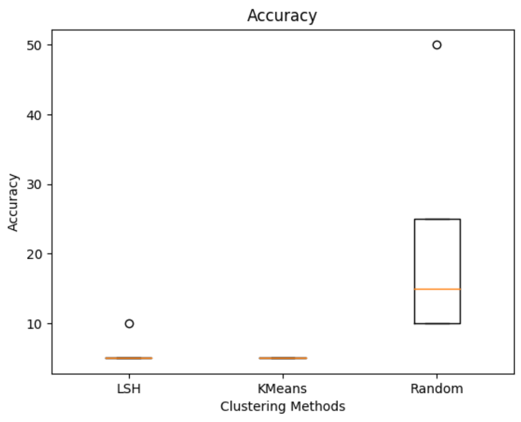


(b) Accuracy for clustering the state machines using six clusters.

Figure B.13: Silhouette Coefficient scores and accuracy of each clustering method for clustering state machines using six clusters.



(a) Silhouette Coefficient scores for clustering the state machines using eight clusters.



(b) Accuracy for clustering the state machines using eight clusters.

Figure B.14: Silhouette Coefficient scores and accuracy of each clustering method for clustering state machines using eight clusters.

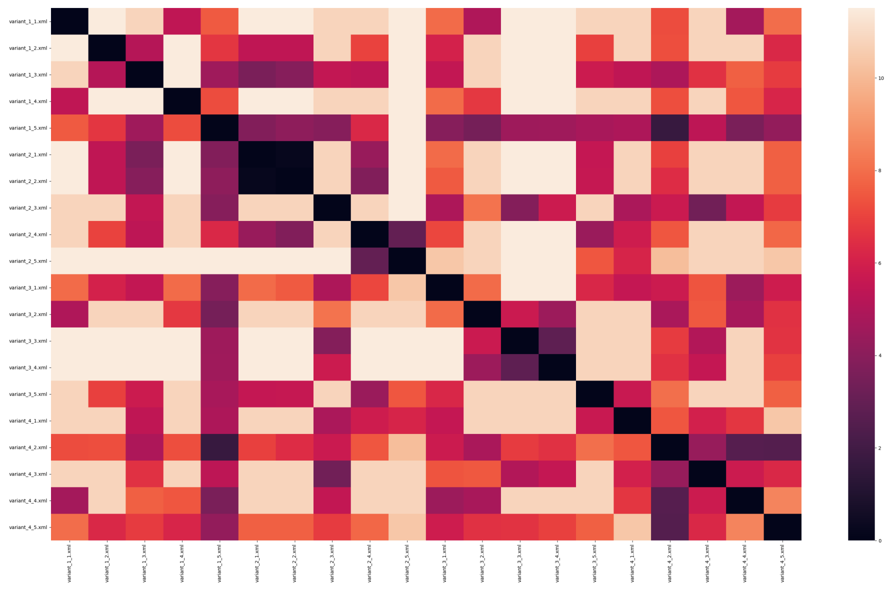
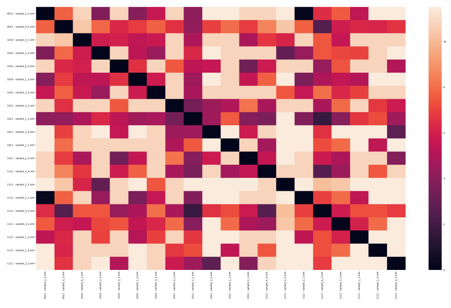
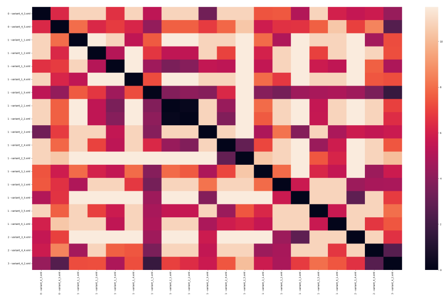


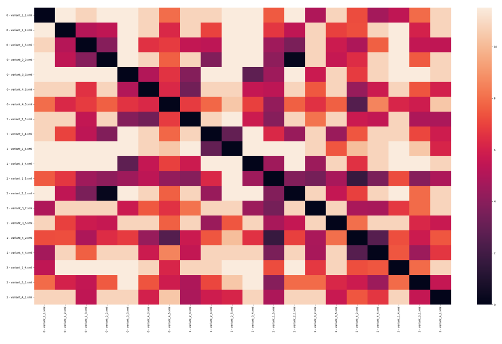
Figure B.15: Heatmap generated from the unsorted distance matrix.



(a) Clustering results of LSH

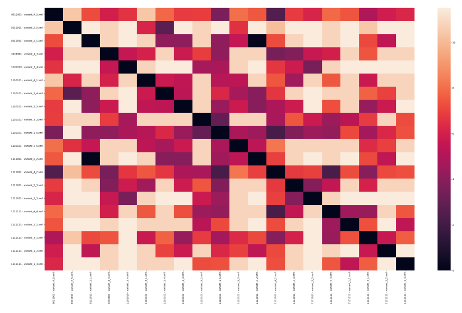


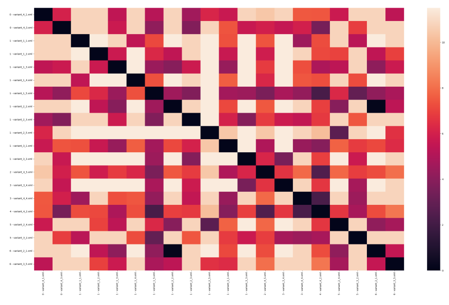
(b) Clustering results of KMeans Clustering



(c) Clustering results of Random Clustering

Figure B.16: Clustering results of LSH, KMeans Clustering and Random Clustering. Four clusters were used for each method.


(a) Clustering results of LSH


(b) Clustering results of KMeans Clustering

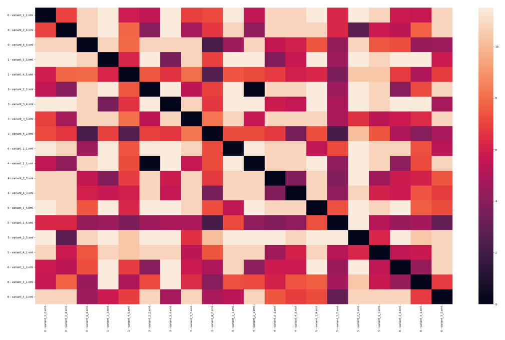
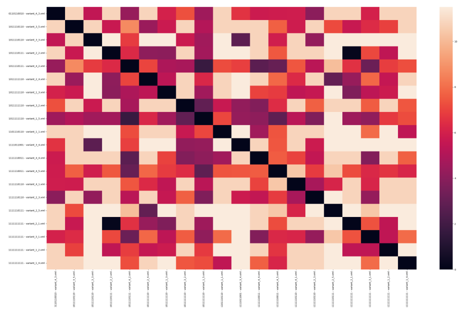
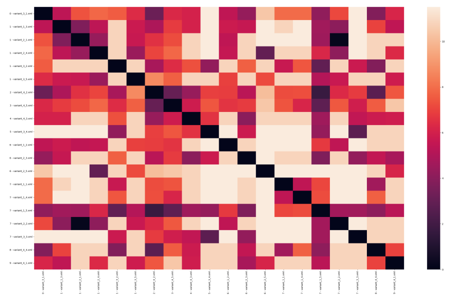

(c) Clustering results of Random Clustering

Figure B.17: Clustering results of LSH, KMeans Clustering and Random Clustering. Six clusters were used for each method.


(a) Clustering results of LSH


(b) Clustering results of KMeans Clustering

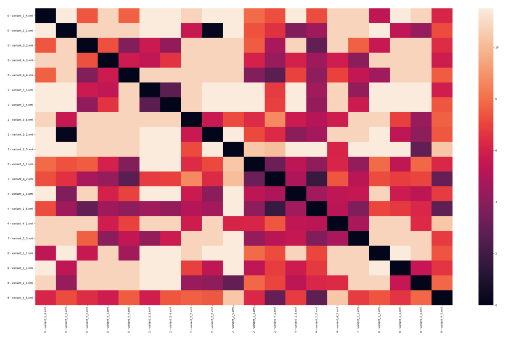
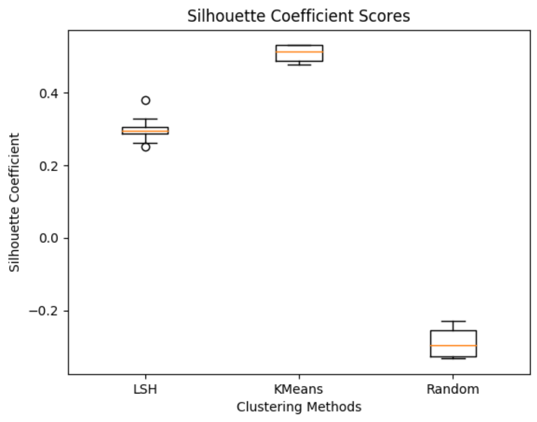
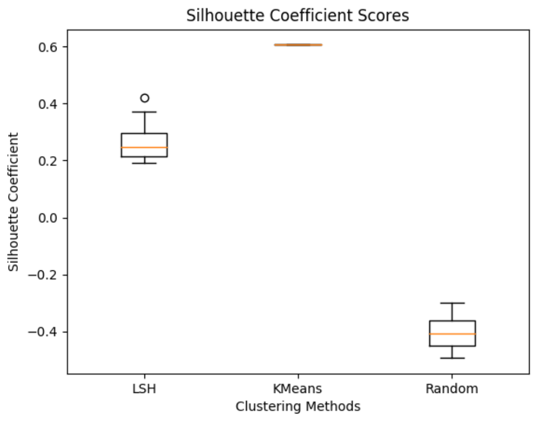

(c) Clustering results of Random Clustering

Figure B.18: Clustering results of LSH, KMeans Clustering and Random Clustering. Eight clusters were used for each method.

B.4. Clustering State Machines From Benign and Malicious NetFlow Data



(a) Silhouette Coefficient scores for clustering the state machines using six clusters.

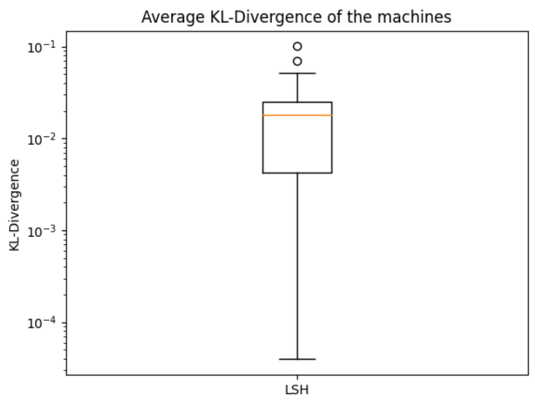


(b) Accuracy for clustering the state machines using eight clusters.

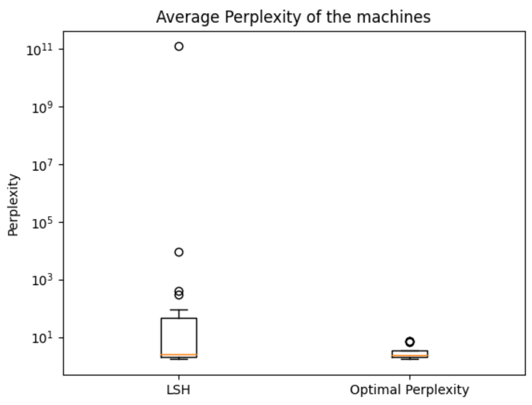
Figure B.19: Silhouette Coefficient scores and accuracy of each clustering method for clustering state machines using six and eight clusters.

C
More Results from the Raspberry Pi Experiments

C.1. Comparison Between State Machines that were Learned on the Raspberry Pi

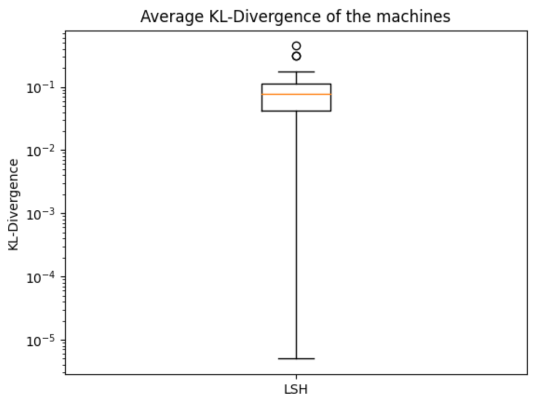


(a) KL-Divergence between the state machines that were learned using LSH and the ones that were learned using the original method.

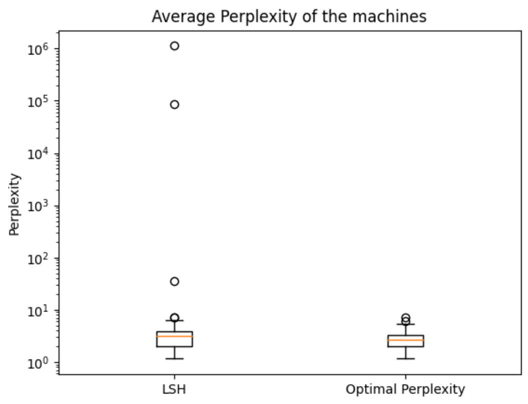


(b) Perplexity between the state machines that were learned using LSH and the ones that were learned using the original method.

Figure C.1: KL-Divergence and perplexity between the state machines that were learned using LSH and the ones that were learned using the original method. The state machines were learned from the dataset of Emotet.

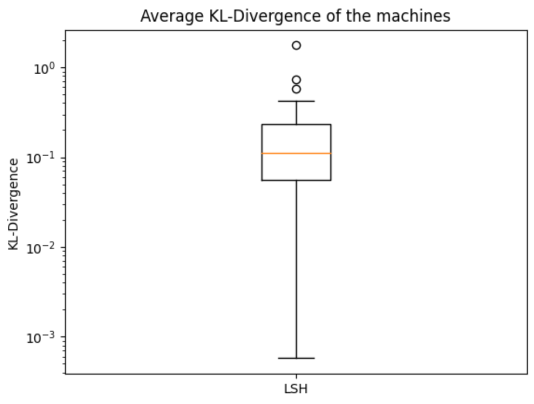


(a) KL-Divergence between the state machines that were learned using LSH and the ones that were learned using the original method.

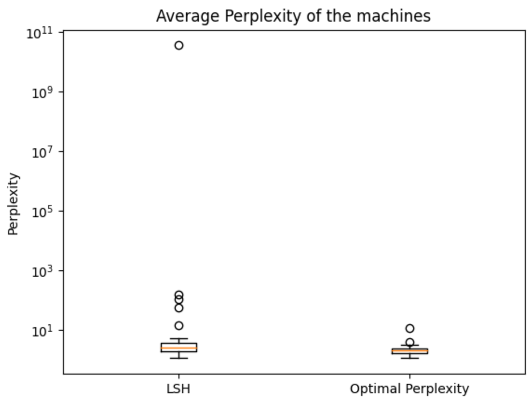


(b) Perplexity between the state machines that were learned using LSH and the ones that were learned using the original method.

Figure C.2: KL-Divergence and perplexity between the state machines that were learned using LSH and the ones that were learned using the original method. The state machines were learned from the dataset of Necurs.



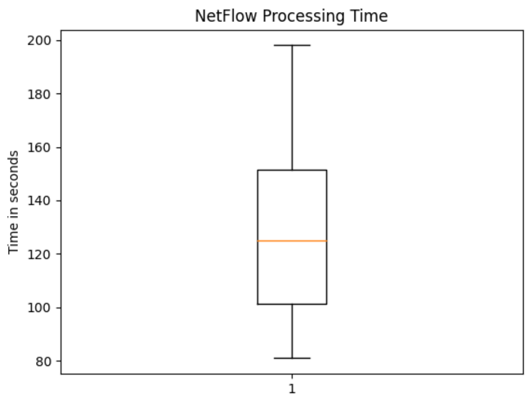
(a) KL-Divergence between the state machines that were learned using LSH and the ones that were learned using the original method.



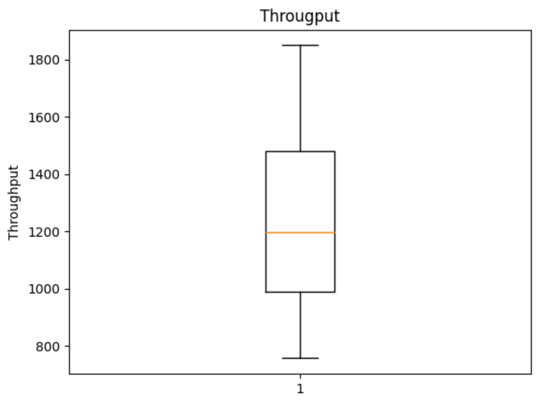
(b) Perplexity between the state machines that were learned using LSH and the ones that were learned using the original method.

Figure C.3: KL-Divergence and perplexity between the state machines that were learned using LSH and the ones that were learned using the original method. The state machines were learned from the dataset of TrickBot.

C.2. ASSESSING THE SCALABILITY OF THE METHOD ON THE RASPBERRY PI

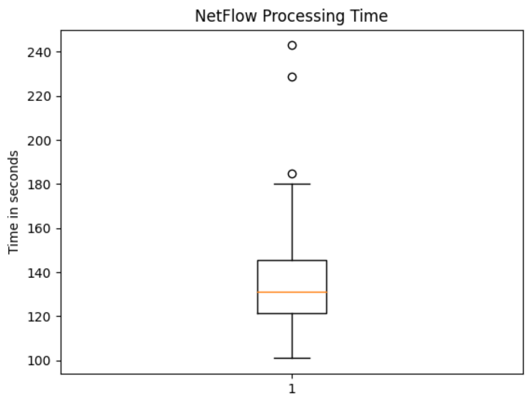


(a) Processing time of the method

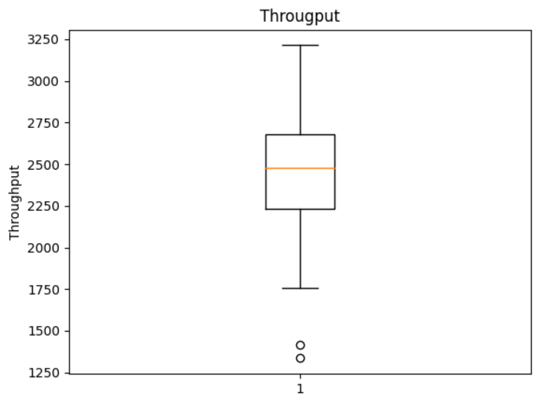


(b) Throughput of the method

Figure C.4: The processing time and throughput of the method on the second-largest dataset (149911 flows)



(a) Processing time of the method



(b) Throughput of the method

Figure C.5: The processing time and throughput of the method on the largest dataset (324569 flows)

BIBLIOGRAPHY

[1] • *Global digital population 2019 | Statista,* https://www.statista.com/statistics/617136/digital-population-worldwide/, (visited on 2019-10-28).

[2] Verizon Communications, *2019 Data Breach Investigations Report*, Tech. Rep. (2019).

[3] S. Back, J. LaPrade, L. Shehadeh, and M. Kim, *Youth Hackers and Adult Hackers in South Korea: An Application of Cybercriminal Profiling,* (Institute of Electrical and Electronics Engineers (IEEE), 2019) pp. 410–413.

[4] G. Pogrebna and M. Skilton, *A Sneak Peek into the Motivation of a Cybercriminal,* (2019), 10.1007/978-3-030-13527-0_3.

[5] A. R. Breurkes, R. Jongerius, M. M. H. A. Kerkhof, and T. D. Westerborg, *Bachelor End Project Real-time anomaly detection in critical Rabobank processes*, Tech. Rep.

[6] A. Kortebi, Z. Aouini, M. Juren, and J. Pazdera, *Home Networks Traffic Monitoring Case Study: Anomaly Detection,* in *2016 Global Information Infrastructure and Networking Symposium, GIIS 2016* (Institute of Electrical and Electronics Engineers Inc., 2017).

[7] Q. Lin, S. Verwer, S. Adepu, and A. Mathur, *TABOR: A graphical model-based approach for anomaly detection in industrial control systems,* in *ASIACCS 2018 - Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security* (Association for Computing Machinery, Inc, 2018) pp. 525–536.

[8] G. Pellegrino, Q. Lin, C. Hammerschmidt, and S. Verwer, *Learning behavioral fingerprints from Netflows using Timed Automata,* in *Proceedings of the IM 2017 - 2017 IFIP/IEEE International Symposium on Integrated Network and Service Management* (Institute of Electrical and Electronics Engineers Inc., 2017) pp. 308–316.

[9] V. Chelak, E. Chelak, and S. Semenov, *DEVELOPMENT OF ANOMALOUS COMPUTER BEHAVIOR DETECTION METHOD BASED ON PROBABILISTIC AUTOMATON OPRACOWYWANIE METODY WYKRYWANIA ZACHOWANIA KOMPUTEROWEGO W ZAKRESIE AUTOMATYKI PROBABILISTYCZNEJ,* (2019), 10.1002/9780470118474.ch5.

[10] G. Guo, Y. Xin, X. Yu, L. Liu, and H. Cao, *A Fast IP Matching Algorithm Under Large Traffic,* (2019) pp. 246–255.

[11] K. S. Umadevi, P. Balakrishnan, and G. Kousalya, *Intrusion detection system using timed automata for cyber physical systems,* Journal of Intelligent & Fuzzy Systems **36**, 4005 (2019).

[12] N. Mohamudally and M. Peermamode-Mohaboob, *Building An Anomaly Detection Engine (ADE) for IoT Smart Applications,* in *Procedia Computer Science*, Vol. 134 (Elsevier B.V., 2018) pp. 10–17.

[13] Khaled Alrawashdeh, C. Purdy, C. Ali Minai, R. K. Bhatnagar, P. A. Wilsey, and B. Gonen, *Toward a Hardware-assisted Online Intrusion Detection System Based on Deep Learning Algorithms for Resource-limited Embedded Systems*, Tech. Rep. (2018).

[14] M. O. Ezeme, Q. H. Mahmoud, and A. Azim, *Hierarchical attention-based anomaly detection model for embedded operating systems,* in *Proceedings - 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2018* (Institute of Electrical and Electronics Engineers Inc., 2019) pp. 225–231.

[15] G. Gracioli, M. Dunne, and S. Fischmeister, *A Comparison of Data Streaming Frameworks for Anomaly Detection in Embedded Systems*, Tech. Rep. (2018).

[16] A. K. Kyaw, Y. Chen, and J. Joseph, *Pi-IDS: Evaluation of open-source intrusion detection systems on Raspberry Pi 2,* in *2015 2nd International Conference on Information Security and Cyber Forensics, InfoSec 2015* (Institute of Electrical and Electronics Engineers Inc., 2016) pp. 165–170.

[17] A. Sforzin, M. Conti, F. Gómez Mármol, and J.-M. Bohli, *RPiDS: Raspberry Pi IDS A Fruitful Intrusion Detection System for IoT,* (2016), 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.114.

[18] S. Tripathi and R. Kumar, *Raspberry Pi as an Intrusion Detection System, a Honeypot and a Packet Analyzer,* (Institute of Electrical and Electronics Engineers (IEEE), 2019) pp. 80–85.

[19] A. A. Gendreau and M. Moorman, *Survey of intrusion detection systems towards an end to end secure internet of things,* in *Proceedings - 2016 IEEE 4th International Conference on Future Internet of Things and Cloud, FiCloud 2016* (Institute of Electrical and Electronics Engineers Inc., 2016) pp. 84–90.

[20] F. Hugelshofer, P. Smith, D. Hutchison, and N. J. Race, *OpenLIDS: A lightweight intrusion detection system for wireless mesh networks,* in *Proceedings of the Annual International Conference on Mobile Computing and Networking, MOBICOM* (2009) pp. 309–320.

[21] S. Misra, K. I. Abraham, M. S. Obaidat, and P. V. Krishna, *LAID: A learning automata-based scheme for intrusion detection in wireless sensor networks,* Security and Communication Networks **2**, 105 (2009).

[22] Y. Fu, Z. Yan, J. Cao, O. Koné, and X. Cao, *An Automata Based Intrusion Detection Method for Internet of Things,* Mobile Information Systems **2017** (2017), 10.1155/2017/1750637.

[23] C. Nykvist and M. Larsson, *Lightweight Portable Intrusion Detection System for Auditing Applications-Implementation and evaluation of a lightweight portable in-trusion detection system using Raspberry Pi and Wi-Fi Pineapple*, Tech. Rep. (2019).

[24] S. M. JAYAPRAKASH, *BEHAVIOUR MODELLING AND ANOMALY DETECTION IN SMART-HOME IOT DEVICES*, Tech. Rep. (2019).

[25] H. Schouten, *Learning State Machines from data streams and an applica-tion in network-based threat detection*, Tech. Rep. (2018).

[26] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets* (2010).

[27] S. Kullback and R. A. Leibler, *On information and sufficiency,* Ann. Math. Statist. **22**, 79 (1951).

[28] Michael Sipser, *Introduction to the Theory of Computation, Third Edition* (2012).

[29] A. I. L. Pitt and M. K. Warmuth, *The Minimum Consistent DFA Problem Cannot be Atxxoximated within any Polynomial*, Tech. Rep. (1989).

[30] K. J. Lang, B. A. Pearlmutter, and R. A. Price, *Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm,* in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 1433 (Springer Verlag, 1998) pp. 1–12.

[31] W. Wieczorek, *Studies in Computational Intelligence 673 Grammatical Inference Algorithms, Routines and Applications* (2017).

[32] J. S. Vitter, *Random Sampling with a Reservoir,* ACM Transactions on Mathematical Software (TOMS) **11**, 37 (1985).

[33] Claudia Hauff, *TI2736-B Big Data Processing,* `https://chauff.github.io/documents/bdp/streaming1.pdf`, (visited on 2019-11-19).

[34] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson, *Synopsis diffusion for robust aggregation in sensor networks,* in *SenSys'04 - Proceedings of the Second International Conference on Embedded Networked Sensor Systems* (2004) pp. 250–262.

[35] B.-Y. Choi and S. Bhattacharyya, *On the Accuracy and Overhead of Cisco Sampled NetFlow*, Tech. Rep. (2005).

[36] P. Flajolet and G. Nigel Martin, *Probabilistic counting algorithms for data base applications,* Journal of Computer and System Sciences **31**, 182 (1985).

[37] J. Misra and D. Gries, *Finding repeated elements,* Science of Computer Programming **2**, 143 (1982).

[38] G. Cormode, *Count-Min Sketch*, Tech. Rep. (2009).

[39] *Apache Hadoop,* `https://hadoop.apache.org/`, (visited on 2019-11-22).

[40] *Apache Flink: Stateful Computations over Data Streams,* `https://flink.apache.org/`, (visited on 2019-11-24).

[41] *Apache Kafka,* `https://kafka.apache.org`, (visited on 2019-11-24).

[42] *Snort - Network Intrusion Detection & Prevention System,* `https://www.snort.org/`, (visited on 2019-11-26).

[43] A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search*, Tech. Rep. (1975).

[44] D. E. Knuthf, J. H. Morris, and V. R. Pratt, *SIAM J. COMPUT*, Tech. Rep. 2 (1977).

[45] A. Guttman, *R-trees: A dynamic index structure for spatial searching,* in *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1984) pp. 47–57.

[46] *WiFi Pineapple - Hak5,* `https://shop.hak5.org/products/wifi-pineapple`, (visited on 2019-11-27).

[47] P. J. Rousseeuw, *Silhouettes: A graphical aid to the interpretation and validation of cluster analysis,* Journal of Computational and Applied Mathematics **20**, 53 (1987).

[48] S. Verwer, R. Eyraud, and C. De La Higuera, *PAutomaC: A probabilistic automata and hidden Markov models learning competition,* Machine Learning **96**, 129 (2014).

[49] *Datasets Overview — Stratosphere IPS,* `https://www.stratosphereips.org/datasets-overview`, (visited on 2020-7-16).

[50] *sklearn.cluster.KMeans — scikit-learn 0.23.1 documentation,* `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`, (visited on 2020-07-24).