Robust Attack Graphs

Dennis Mouwen

Delft University of Technology

Robust Attack Graphs

by

Dennis Mouwen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday April 20, 2022 at 9:00.

Student number:	4452070
Project Duration:	February 2021 - April 2022
Thesis Committee:	Dr. Ir. S. E. Verwer, TU Delft, Supervisor
	Dr. M.M. de Weerdt, TU Delft

TU Delft

Preface

This master's thesis is the result of more than a year of hard work, and the concluding part of my master's degree in Computer Science. Graduating during the COVID-19 pandemic was an extra challenge. It is always a project you commence on your own, but without the regular talks and coffee breaks with other students it is easy to become an island, especially when going to campus is not allowed.

I would like to express my gratitude for Sicco's enthusiasm and constructive criticism. Our meetings always seemed to steer me in the right direction.

I would also like to thank my other supervisors, Azqa and Chris, for their encouraging words and critical minds, my roommates, Tom and Stephan, for the fun times we had despite the pandemic, and my parents, for their unconditional support.

Most of all, I would to thank my girlfriend Manon for helping my find a way through this web, and always reminding me to relax and have fun outside studying hours.

Dennis Mouwen
Delft, April 2022

Abstract

Every day, Intrusion Detection Systems around the world generate huge amounts of data. This data can be used to learn attacker behaviour, such as Techniques, Tactics, and Procedures (TTPs). Attack Graphs (AGs) provide a visual way of describing these attack patterns. They can be generated without expert knowledge and vulnerability reports. The goal of AGs is to reduce alert fatigue, and to give another perspective on attacker behaviour.

SAGE, the state-of-the-art method for generating AGs uses FlexFringe's state merging algorithms to learn the underlying state machine to model these attacks. A big challenge of these state merging algorithms is learning infrequent behaviour. Next to that, the underlying state machines cannot deal with noisy input data.

In this work, a sequence-automaton alignment algorithm is used to align sequences of states to a state machine. Our method iteratively aligns infrequent sequences to the model, effectively learning both frequent and infrequent behaviour.

The method is evaluated on a security competition dataset, where experiments show that the algorithm is able to recover from noise such as added or removed events. The learned models are also reduced to half its original size, while better fitting to the training data.

Last, we show how our method can be used to learn the model of an anomaly detection dataset. The test data is predicted using three anomaly conditions, and results in all anomalous data being labelled correctly. The F1-score is competitive compared to other state of the art methods.

Contents

1 Introduction

Since the dawn of the Internet, more and more computers have been connected to the rest of the world. Nowadays, even our phones, printers, and refrigerators are part of the Internet. Criminals have figured out there is money to be made, data to be stolen, or services to be disrupted, when these devices are hacked. The cost of these attacks put a big financial stress on companies.

To protect users from these attacks, several defence mechanisms have been created. The Intrusion Detection System (IDS) is such a system, which generates *alerts* based on a set of rules. These rules are triggered by suspicious or malicious network traffic.

1.1. Intrusion Detection System Alerts

An Intrusion Detection System (IDS) is a software system or hardware device that monitors network connections in the network. Its goal is to identify suspicious activities. Typically, IDS's form the first line of defense against adversaries. IDS's can generally be divided into two types: Signature-based IDS's and Anomaly-based IDS's [16].

Signature-based IDS's generate alerts based on prior attacks. They can detect attacks of known vulnerabilities with high accuracy, but fail in detecting zero-days. Anomaly-based IDS's learn patterns from regular network traffic, and will generate alerts from irregular or anomalous traffic. They have a high accuracy for zero-days, but generate more false positives in the process. This research will focus on Signature-based IDS's, as they will give the most consistent alerts based on network traffic.

The problem with Intrusion Detection Systems however, is that they often generate more alerts than security administrators can handle. Alert Correlation techniques are used to reduce and group alerts belonging to the same attack stage [1, 36, 35]. They can show *what* an attacker did, but not *how* they exploited the infrastructure. To combat this, Attack Graph (AG) generation can be used.

1.1.1. IDS Alert Datasets

As real IDS data is hard to come by due to privacy concerns, a security testing competition dataset is used as a surrogate. While not real data, they can still provide complex enough alerts for multi-stage attacks. In this research, the CPTC-2018 dataset [32] is used. It will be explained in detail in Chapter 4.

From the IDS alerts, alert categories are extracted. An alert category is an identifier which tells in which attack phase the attacker is. The Attack-Intent framework of Moskal [23] is used, but any other framework can be used. The alert category can also be used to describe the severity of the alert.

Notice that this work does not rely on specific vulnerabilities or CVEs, but rather on higher level attack stages. This way, the system can be automated more easily, and does not need to be updated every time a new vulnerability is discovered.

1.2. Attack Graph Generation

Attack Graphs (AG) are models that visually describe attacker strategies[2, 20]. While a lot of research has been done on AG generation, most methods that generate AGs rely heavily on expert knowledge [28, 1, 29], which is both expensive and time-consuming, or published vulnerability reports [31, 17, 33, 15], which are always one step behind attackers. Alert-based AG generation methods only use (IDS) alerts. To the best of our knowledge, SAGE [26] is the first and only method for generating Alert-based AGs without prior knowledge. SAGE uses a suffix-based probabilistic deterministic finite automaton (S-PDFA) to learn and model these attacks. This model accentuates infrequent severe alerts, without discarding low-severity alerts. After the S-PDFA is learned, it is used to recognize known attacks in new incoming alerts.

After the model is learned, it is used to recognize known attacks in new incoming alerts. A pitfall of SAGE however, is that these alerts must exactly follow a path in the state machine. When this is not the case - for example, an alert is missing due to a new technique or simply a defective IDS - SAGE will not generate an AG. This can be seen in CPTC-2018 model learned by SAGE, where from the 171 training episodes containing high-severity alerts, 21 are not present precisely in the model. This means 12% of the attacks are not fully represented in the model.

Another shortcoming of SAGE is in its learning phase: The underlying automaton learning package FlexFringe [42] tries to highlight infrequent severe alerts, without discarding low-severity alerts [26]. Still, infrequent alerts can be missed due to the nature of the statistical tests in these automaton learning algorithms, which leads to missing alerts in the model. FlexFringe uses so-called *sinks states* to deal with infrequent states, which effectively means the algorithm did not learn from those states at all. In the model learned by SAGE of the CPTC-2018 dataset, 259 of the 536 learned sequences (48%) prematurely end in a sink state.

1.3. Contributions and Objectives

In this work, we propose Sequence-Automaton Alignment Tool (SAAT). SAAT computes an *alignment* between a sequence and an automaton, and computes a normalised score for each alignment. This normalised score describes how accurate a sequence fits the model. The average normalised score over all training data can then be used to quantify how accurate the model has learned the data. Our method iteratively uses these alignments to learn smaller and better models. We also show how iterative SAAT can be used for anomaly detection, to show further applications are possible.

The goal of this work is to design and evaluate a method to make alert-based attack graph generation based more robust to noisy data, as well as improve the ability to learn infrequent behaviour. Thus, the main research question we will answer is:

How can we improve the robustness of SAGE using sequence-automaton alignment, and how can we use this alignment to improve the ability to learn infrequent behaviour?

This research question is divided into four sub-questions:

RQ1 How does SAGE perform with noisy input data?

RQ2 How can we improve the robustness of SAGE using sequence-automaton alignment against noise, such as added or missing alerts?

RQ3 How can we use the computed alignments to better learn infrequent behaviour?

RQ4 How can we use sequence-automaton alignment for anomaly detection?

The main contributions of this research are:

1. We propose Sequence-Automaton Alignment Tool (SAAT), which computes an *alignment* between a sequence and an automaton.
2. We provide a definition of the normalised alignment score, a metric that quantifies how well a model fits the data.
3. We show in experiments that iterative SAAT reduces the model size by almost a half, while improving the normalised score.
4. We learn the model of an anomaly detection dataset, and how such a model can be used to correctly predict all anomalies.

1.4. Outline

In Chapter 2 the necessary background information is provided. Chapter 3 describes the proposed methodology of this research. In Chapters 4 and 5 this method is evaluated. In Chapter 6 we describe how SAAT can be used for anomaly detection, and evaluate this method. In Chapter 7, we discuss the limitations of our work. We conclude in Chapter 8 and list future work in Chapter 9.

2

Background

In this chapter, the background information which will be needed for further chapters will be provided.

2.1. Metrics

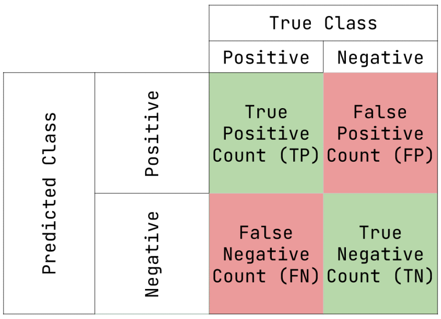
When measuring the performance of machine learning models, usually a coincidence matrix is used [30]. Figure 2.1 shows the coincidence matrix for a two-class classification problem. Since there can be an imbalance in positive and negative testing data, several other metrics are used to quantify the performance:

- Accuracy is the simplest metric, and quantifies the ratio of correctly labelled test cases to the total amount of test cases.
- Precision quantifies the percentage of correctly labelled positive test cases out of all positive labelled test cases.
- Recall quantifies percentage of positive test cases that were correctly labelled.
- F1-score is the harmonic mean between Recall and Precision, and expresses both of these with a single score.

They are defined as:

$$\begin{aligned}
\text{Accuracy} &= \frac{TP+TN}{TP+TN+FP+FN} \\
\text{Precision} &= \frac{TP}{TP+FP} \\
\text{Recall} &= \frac{TP}{TP+FN} \\
\text{F1-score} &= \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}
\end{aligned}$$

For all of these metrics, the range is $[0, 1]$, with 1 being the best possible score.



		True Class	
		Positive	Negative
Predicted Class	Positive	True Positive Count (TP)	False Positive Count (FP)
	Negative	False Negative Count (FN)	True Negative Count (TN)

Figure 2.1: A coincidence matrix for a two-class classification problem

2.2. Breadth First Search

Breadth First Search (BFS) is a graph search algorithm, used for finding specific nodes. It starts at a node s, and explores all neighbours of s, adding them to a queue. If the target node is not found, the first node from the queue is then popped, and that node's neighbours are searched. It is called breadth-first because nodes near the starting node are searched before exploring a node's neighbours in depth, as opposed to Depth First Search. In other words, nodes at distance k from s are searched before nodes at distance $k+1$.[10] A few variations of BFS are used in this work to traverse graphs, for example to determine a node's level.

2.3. State Machines

A State Machine, or a Finite State Automaton (FSA), is a mathematical model that is used to describe a system based on states and transitions. Formally, a FSA is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ [37], where:

- Q is a finite set of states.
- Σ is a finite set of symbols that are used as input, also called the alphabet.
- δ is a transition function that maps a state and an input symbol to the next state : $\delta : Q \times \Sigma \to Q$.
- q_0 is the start state, where $q_0 \in Q$.
- F is the set of *accepting* or *final* states, where $F \subseteq Q$.

An FSA reads an input string one character at a time, and moves to the next state according to the transition function δ. After the whole input string has been parsed, the FSA will *accept* the string if it is currently in an accepting state. It will reject the string if it is not in an accepting state.

Example An example of an FSA can be found in Figure 2.2, with $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $F = \{q_2\}$. For the input string *aba*, the FSA will first move from q_0 to q_1, from q_1 to q_1, and lastly from q_1 to q_2, where it is in an accepting state. More example strings that this FSA will accept are *aa* or *abba*. The set of strings that are accepted by an FSA is called the language of the FSA. For this FSA, the language is ab^*a, where b^* means there can be zero or more b characters.

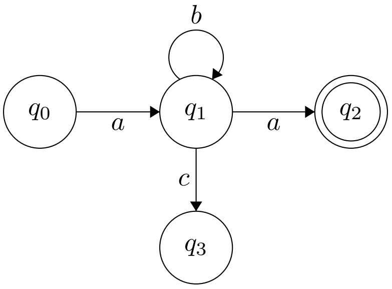


Figure 2.2: Example of a Finite State Automaton. The accepting final state is marked with a double circle.

2.3.1. Probabilistic Finite Automata

An FSA is deterministic when there is only one unique accepting or rejecting path for each possible input. It is then called a Deterministic Finite Automaton (DFA). A Finite Automata is a non-deterministic Finite Automaton (NFA) when multiple paths with the same symbol can be taken from the same node. An example can be seen in Figure 2.3.

NFAs can also be used to model distributions of strings, using a variant called the Probabilistic Finite Automaton (PFA). This is an NFA where the edges have a probability between 0 and 1. Formally, a PFA is a NFA where the transition function is defined as:

$$\delta : Q \times \Sigma \times Q \to [0, 1]$$

with the constraint that the sum of the probabilities of all outgoing edges should be 1, or:

$$\forall q \in Q : \sum_{a \in \Sigma, q' \in Q} \delta(q, a, q') = 1$$

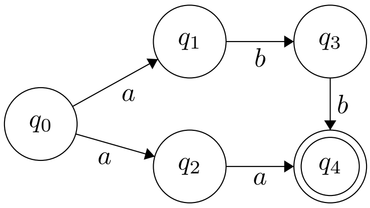


Figure 2.3: Example of a non-deterministic Finite State Machine (NFA)

2.3.2. Suffix-based Probabilistic Deterministic Finite Automata

To make parsing and generating strings using a PFA deterministic, another constraint is needed: For each state, for each symbol, the next state is unique, i.e. a state can only have one outgoing edge for each symbol. If a PFA adheres to these constraints, it is called a Probabilistic Deterministic Finite Automaton (PDFA). A variation of this used in this research is the Suffix-based Probabilistic Deterministic Finite Automata (S-PDFA). This variation is nothing more than an automaton that parses tokens from right to left, instead of left to right.

We transform the traces for use with an S-PDFA by reversing them. As such, the start of a trace will be the event that happens latest in time, and any event next in the trace, will be earlier than the one before.

2.3.3. Automata Learning

Automata can be learned using multiple machine learning (ML) techniques. However, the input data mostly contains low-severity alerts, while the high-severity alerts are what is interesting to security analysts. This means most ML methods are not going to work, as they focus on frequently occurring high-severity alerts. These infrequent high-severity alerts can be learned using the Alergia [8] state-merging algorithm. Combined with the S-PDFA representation, this method accentuates these infrequent severe alerts, without discarding low-severity alerts. [26].

To learn the models in this research, the FlexFringe automaton learning framework [42] is used. FlexFringe does not learn from infrequent states, but uses *sink states* to keep infrequent states in the model. FlexFringe contains multiple algorithms and heuristics to learn these models. In [26, 40] more theoretical and practical information can be found. As these works contain similar datasets as this work, their parameters will be used to train the models in this work.

2.4. Attack Graphs

Attack Graphs (AGs) are models describing paths adversaries take to intrude the target network. Typically, they consist of nodes representing exploited endpoints, and edges representing the exploited vulnerabilities [22]. In SAGE, they are generated on a per-objective, per-victim basis.

An example of an attack graph is seen in Figure 2.4. AGs can help both experts and non-experts understand the structure of cyber attacks, as well as gain new insights.

2.5. Alert-based Attack Graph Generation

As explained in Chapter 1, in [26] the state of the art for alert-based attack graph generation is presented. In [26], attack graphs are generated using an S-PDFA model in a system called SAGE. A general outline of SAGE can be seen in Figure 2.5.

First, alerts are processed into attack episode sub-sequences (AESS), which are sequences of alert categories of the Moskal Attack-Intent framework [24]. The S-PDFA model is trained using these AESS. At test time, the test alerts are also processed into AESS. However, to match the incoming AESS to the model, the nodes in the model and the incoming trace need to be a one-on-one match. In practice, this is often not the case, due to incorrect or missing alerts, or attackers using slightly different techniques. As such, we can use sequence alignment techniques to match new traces with a trained model.

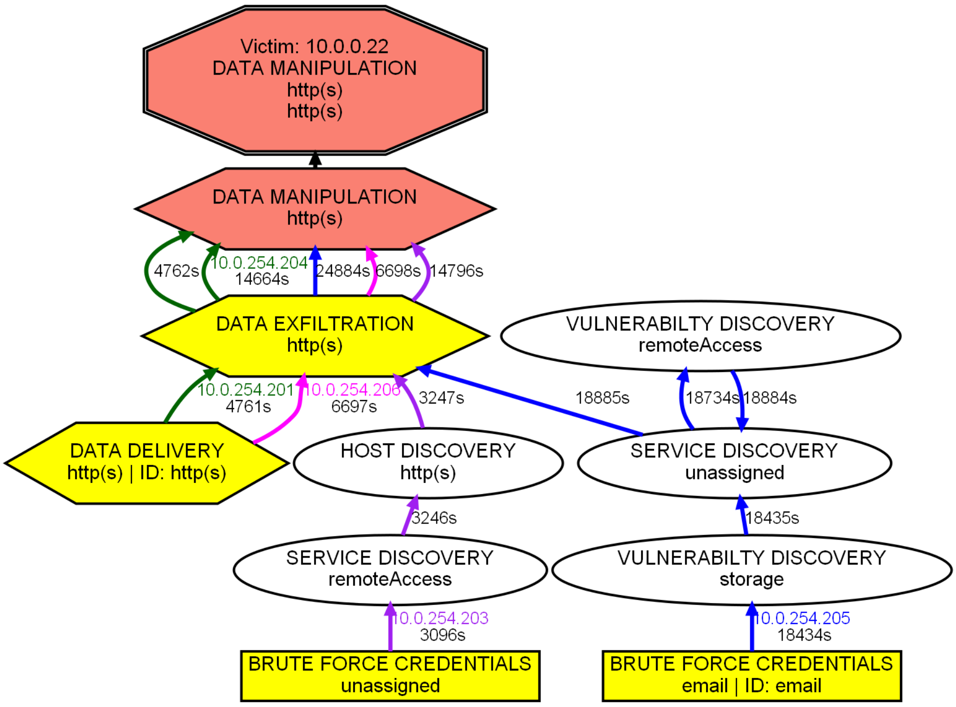


Figure 2.4: Example of an Attack Graph. Nodes: Labels show (attack stage, targeted service, state identifier). Low-severity episodes are oval, medium-severity are boxes, high-severity are hexagons. The first episode in a path is yellow, the objective is red. Sinks are dotted. Edges: Labels show seconds since the first alert. Colours show team affiliation.

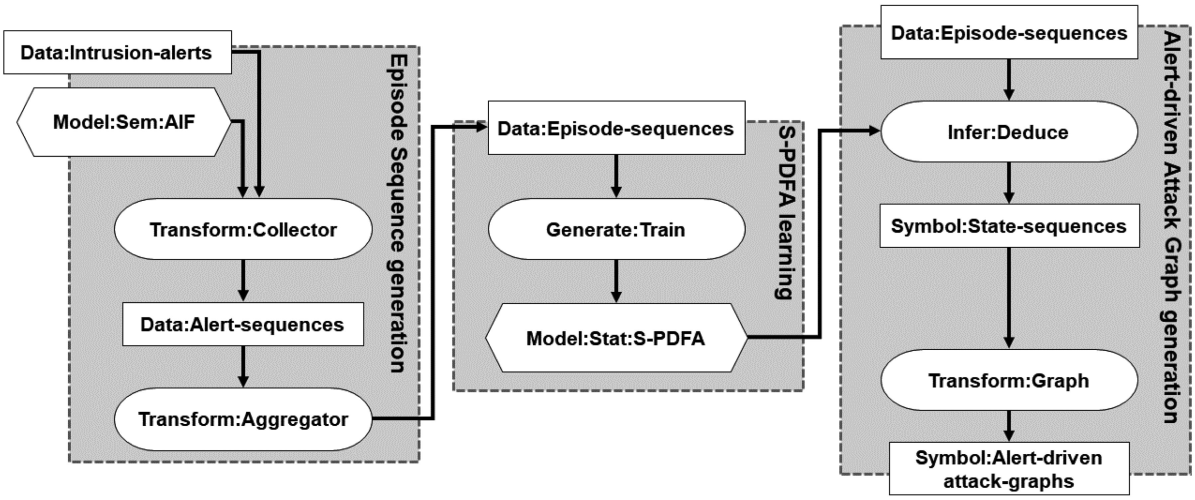


Figure 2.5: Attack Graph generation in SAGE [26]. SAGE takes intrusion alerts as input and generates attack graphs. Intrusion alerts are transformed into episode sequences. An interpretable S-PDFA model is learned from those sequences. The sequences are replayed through the S-PDFA and transformed into targeted attack graphs.

2.6. Sequence Alignment

Sequence Alignment is a technique used in biology for DNA and RNA comparison to detect homologous subsequences among sets of long sequences [38]. It usually consists of finding an alignment between two sequences and assigning a similarity score to the found alignment. There are two main alignment methods:

- Global alignment, where two sequences have to be aligned from start to end. This is most useful when the two sequences are of roughly equal length.
- Local alignment, where only parts of the sequences have to be aligned. This is more useful when the sequences are expected to only partially align.

When comparing the characters of two sequences, they can differ in three different ways:

1. *MATCH*: The two symbols at the current index are equal.
2. *MISMATCH*: The two symbols at the current index are not equal.
3. *GAP*: The symbol of one sequence aligns to a gap in the other sequence.

2.6.1. Sequence Alignment Using Dynamic Programming

Dynamic programming [5] is an approach for solving complex problems using simpler sub-problems. The solutions to the sub-problems are stored and used in the computation of the more complex problem.

Needleman-Wunsch Sequence Alignment

The Needleman-Wunsch algorithm is a dynamic programming algorithm for global sequence alignment. For two sequences x and y of size m and n, it uses a matrix M of size $(m + 1) \times (n + 1)$ to store the sub-problems. Sub-problem solutions are given a score which is stored in the matrix. Scores are based on the three options given in Section 2.6: Match, Mismatch and Gap. The time complexity is $O(mn)$, and the space complexity is also $O(mn)$[4].

	j-1	j
i-1	diagonal	up
i	left	current

Table 2.1: The cell with *current* is currently being computed. The other cells represent which values are needed to compute this value

The algorithm consists of the following steps:

1. Initialize the alignment matrix: Values at row 0 or column 0 have a score of index times the Gap penalty
2. Fill in the matrix: For every row i and column j, pick the best score out of the left, top, or top-left cell, ie. $\max(M[i-1][j], M[i][j-1], M[i-1][j-1])$, and compute the new score. This is visualized in Table 2.1
3. From the filled matrix, find the best alignment.

Pseudocode can be seen in Algorithm 1

Algorithm 1: Needleman-Wunsch algorithm for sequence alignment. The `MatchScore` function returns *MATCH* if the two arguments are equal, or *MISMATCH* if not.

```
Input: size of sequences: $m, n$
Output: matrix: $M$
for $i = 0$ to $m$ do
    $M[i][0] = GAP * i$
for $j = 0$ to $n$ do
    $M[0][j] = GAP * j$
for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
        $match = M[i-1][j-1] + \texttt{MatchScore}(x_i, y_j)$
        $delete = M[i-1][j] + GAP$
        $insert = M[i][j-1] + GAP$
        $M[i][j] = \texttt{max}(match, delete, insert)$
```

Example In Figure 2.6, the optimal alignment is given between the strings *MARIO* and *WALUIGI*. The highlighted cells represent the max value per column, ie. the optimal alignment. In Table 2.6b, the subchoiches are given for computing the bordered cell in Table 2.6a. Since "*A*" equals "*A*", the diagonal cell is equal to $-1 + 1 = 0$, which is the highest of the three. Thus, 0 gets filled in the matrix.

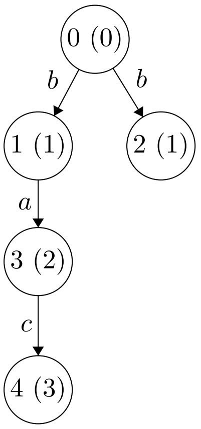
		W	**A**	**L**	**U**	**I**	**G**	**I**
	0	-2	-4	-6	-8	-10	-12	-14
M	-2	-1	-3	-5	-7	-9	-11	-13
A	-4	-3	0	-2	-4	-6	-8	-10
R	-6	-5	-2	-1	-3	-5	-7	-9
I	-8	-7	-4	-3	-2	-2	-4	-6
O	-10	-9	-6	-5	-4	-3	-3	-5

(a) The filled matrix. The options for computing the bordered cell are given in Figure 2.6b

	W	**A**
M	-1 + 1 = 0	-3 - 2 = -5
A	-3 - 2 = -5	?

(b) Subchoices for the bordered cell comparing the second characters *A* and *A*.

Figure 2.6: Optimal alignment using Needleman-Wunsch algorithm, with scores *MATCH*=1, *MISMATCH*=-1, *GAP*=-2. A green cell represents a *MATCH*, a red cells represents a *MISMATCH*, a blue cell represent a *GAP*



(a) The tree. In parentheses, the level of the node is given.

Level	Transition	-	b	a	c
-	-	0	-2	-4	-6
1	0 → 1	-2	4	2	0
1	0 → 2	-2	-4	2	0
2	1 → 3	-4	2	8	6
3	3 → 4	-6	0	6	12

(b) The alignment matrix of the tree and the sequence 'b, a, c'. The highlighted cells are the max value per column.

Figure 2.7: An example tree and alignment matrix

2.7. Tree-Sequence Alignment

In [40], a method for comparing state machines with input strings is presented. The used algorithm, which will be referred to as TS, is a dynamic programming algorithm based on the Needleman-Wunsch (NW) algorithm [27] given in Section 2.6. It will be explained using the example tree given in Figure 2.7a. The corresponding alignment matrix of this tree and the sequence b, a, c is given in Figure 2.7b. A more detailed explanation with examples will be given later.

Instead of working with two sequences, it uses a tree and a sequence as input. The general outline of the algorithm is the same as NW, with some changes due to the difference in data structure:

- In NW, the rows and columns of the alignment matrix represent symbols of the two sequences that are being compared. In TS, columns still represent symbols of the sequence, while rows represent edges in the tree. The edges of the tree are sorted based on the level of the edge[1].
- When initializing the alignment matrix, the GAP score isn't multiplied by the index of the row, but the level of the edge.
- When filling the matrix and computing the score of a cell, the left value is always directly left to the cell. However, the diagonal and up cells are not always directly above. The matrix has to be searched for the row with the edge with level equal to the current level minus one, and the destination equal to the current source. In other words, $\mathit{edge\_level} == \mathit{current\_level} - 1$ and $\mathit{edge\_destination} == \mathit{current\_source}$. For example, in Figure 2.7b, let's look at the cell of transition 1 → 3 and symbol a. The row taken for the diagonal and up values is not the row directly above, but that of transition 0 → 1, since that connects to the transition 1 → 3.

Pseudocode for the matrix filling step can be seen in Algorithm 2.

[1] The level of an edge is equal to the level of the source node, being the distance from the root of the tree to the node. The level of the root is zero

Algorithm 2: Initialize and fill alignment matrix

```
Input: edges: $E$, sequence: $S$
Output: alignment matrix: $M$
def GetThreeNeighbouringCells (i, j, edge, transition) :
    $upIndex = \texttt{FindUpIndex}(edge, i, j)$
    $left = M[i, j-1] + GAP$
    $up = M[upIndex, j] + GAP$
    $diagonal = M[upIndex, j-1] + \texttt{MatchScore}(edge.label, transition)$
    return [left, up, diagonal]
$m = \texttt{len}(E)$
$n = \texttt{len}(S)$
for $edge$ in $E$ do
    $M[i][0] = GAP * edge.level$
for $j = 0$ to $n$ do
    $M[0][j] = GAP * j$
for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
        $edge = E[i]$
        $transitionLabel = S[j]$
        $neighbouringCells = \texttt{GetThreeNeighbouringCells}(i, j, edge, transitionLabel)$
        $M[i][j] = \texttt{max}(neighbouringCells)$
```

Linear Scoring A Linear scoring system is used in the alignment, meaning that the scores for *MATCH* and *MISMATCH* are constant, but the score of *GAP* scales linearly with the length of the gap.

Backtracing After filling the matrix, the optimal alignment is determined using backtracing. From right to left, the maximum value per column is the edge that is best aligned to the symbol of the trace in that column. For example, in Figure 2.7b, symbol c is aligned to edge 3 → 4. Then, move on to the next (left) column.

Starting from when the second edge is aligned, a check is done to make sure the target and source are equal. When this is not the case, one or more edges have been skipped. A breadth-first search from the source to the root finds these skipped edges. See Figure 2.8 for an example. If there are multiple maximum values in a single row, all trace symbols except for the right-most (which is aligned first) are added edges.

Pseudocode for the backtracing step can be found in Algorithm 3.

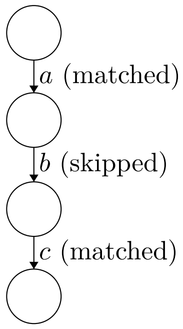


Figure 2.8: Alignment with a skipped edge. When first c is matched and then a, b must have been skipped.

2.8. Related Work

In this Section we discuss the state of the art for generating Alert-based Attack Graphs, and learning automata.

Learning automata (or state machines) from sequences is a grammatical inference [11] problem, where the sequences are modelled as the words of a language, and the objective is to construct a model for this language. This model is usually a (probabilistic) deterministic finite state automaton. Learning such models has been used for lots of different software systems, such as communication protocols [9, 3, 14] or web services [6, 18]. SAGE [26] is the first method that uses automata learning to generate attack graphs from intrusion alerts.

A key challenge in automata learning is dealing with infrequent states during learning. There is usually a trade-off between keeping them intact to obtain a more interpretable model, or discard/merge them to improve the performance of the model. SAAT aligns the infrequent states to the frequent ones, and learns from them instead of keeping them out of scope.

Although alignment is not common in automaton learning, in process mining [41] it is frequently used in process conformance testing [34, 7], which uses sequence alignment to determine if log data adheres to the learned model. Using a sequence-to-model algorithm based on [40], we will perform a

Algorithm 3: Backtracing through the matrix to find the best alignment.

```
Input: alignment matrix: $M$, trace: $T$
Output: aligned sequence: $S$
for $i = \texttt{len}(T) - 1$ to $0$ do
    $edge$ = getMaxEdgeOfColumn $(M, i)$
    if multipleMaxInRow $(M, edge)$:
        $S$.append(findAddedEdges$(M, edge)$)
    else:
        $S$.append$(edge)$
    if len$(S) \geq 2$:
        $source, target$ = getSourceTarget$(S)$
        if $source \neq target$:
            $skipped\_edges$ = findPath$(source, target)$
            if $skipped\_edges \neq []$:
                $S$.append$(skipped\_edges)$
            else:
                $j$ = JumpEdge$(source, target)$
                $S$.append$(j)$
```

similar process. We will refer to this algorithm as the TS algorithm. It is a dynamic programming algorithm based on the Needleman-Wunsch sequence alignment algorithm [27].

There exist many other algorithms for sequence-to-graph alignments [19, 21], most focused on genome graph. As this work will not focus on the alignment algorithm itself, these will not be further explored.

3 Methodology

In this chapter, we introduce and explain a method to improve the robustness of SAGE, and therefore answer **RQ2**. First, some design choices are explained. Then, SAAT (Sequence-Automaton Alignment Tool) is introduced. SAAT can be used to perform alignment between a sequence and an automaton or state machine. Then, by iteratively aligning the sequences to the model and re-learning it, the models are improved, and **RQ3** is answered.

3.1. Robust State Machines

To make state machines more robust to noisy data, two options are possible: They can be made more robust at training time (when learning the machine), or at test time (when traversing the state machine).

Making the model more robust at training time can be done by adding perturbations of the input data (or random noise) to the training data. However, this will generate more complex models with lots of *filler alerts*, e.g. alerts that don't actually tell anything about the attack, but are there so the model can handle more attack paths. This makes the model less descriptive and harder to interpret, making this an unviable choice.

To improve the robustness at test time, we can "relax" some properties of a state machine: If we allow transitions to be skipped or added, we can create paths that only partially exist in the state machine, i.e. the language of the state machine is extended with words where not every symbol in the word actually corresponds with a transition in the state machine, or transitions are in the state machine are skipped.

Example In Figure 3.1, a simple state machine is shown. Normally, the sequence (or word) $a \to b \to d$ would not be accepted, as there is no edge from node 2 to node 4 with symbol d. However, to improve the robustness of the state machine - the symbol c could have been accidentally skipped - we could choose to accept this sequence by skipping the transition from node 2 to 3.
To perform this application programmatically, we can use sequence-automaton alignment.

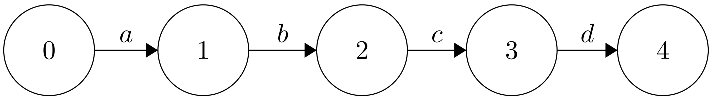


Figure 3.1: Simple example of a state machine

Another interpretation of robustness is the ability to accentuate infrequent traces. Infrequent traces that were not learned into the model at first, will be aligned to the frequent traces, resulting in

By iteratively using the alignments given by SAAT , sequences that were first filtered out can be learned back into the model. This method will be explained in detail in Section 3.3

3.2. Sequence-Automaton Alignment

SAAT is developed around the concept of an alignment between a sequence and an automaton. Recall that a sequence is a list of symbols, e.g. $[a, b, c]$, or [apple, banana, cherry]. An automaton (or state machine) is a directed graph with labels on the edges. Next to an automaton, other types of graph-like data structures can be used as well, such as directed graphs or trees.

SAAT performs *global* alignment, meaning that the full sequence will be aligned on the automaton. We have chosen for this approach because the main application of SAAT will be to generate Alert-based AGs, where the full attack needs to match to the states of the automaton. In this work, an alignment is a list of *matched*, *skipped*, or *added* edges (Section2.7). A matched edge describes both a symbol in the sequence and an edge in the model. Added and skipped edges are used to describe the differences between the sequence and the model.

A high-level overview of SAAT can be seen in Figure 3.2. SAAT takes as input a list of sequences and a model (which can be a directed graph or state machine). For each sequence, the alignment matrix is computed. Then, some backtracing steps are needed to find the alignment, and which edges are added or skipped.

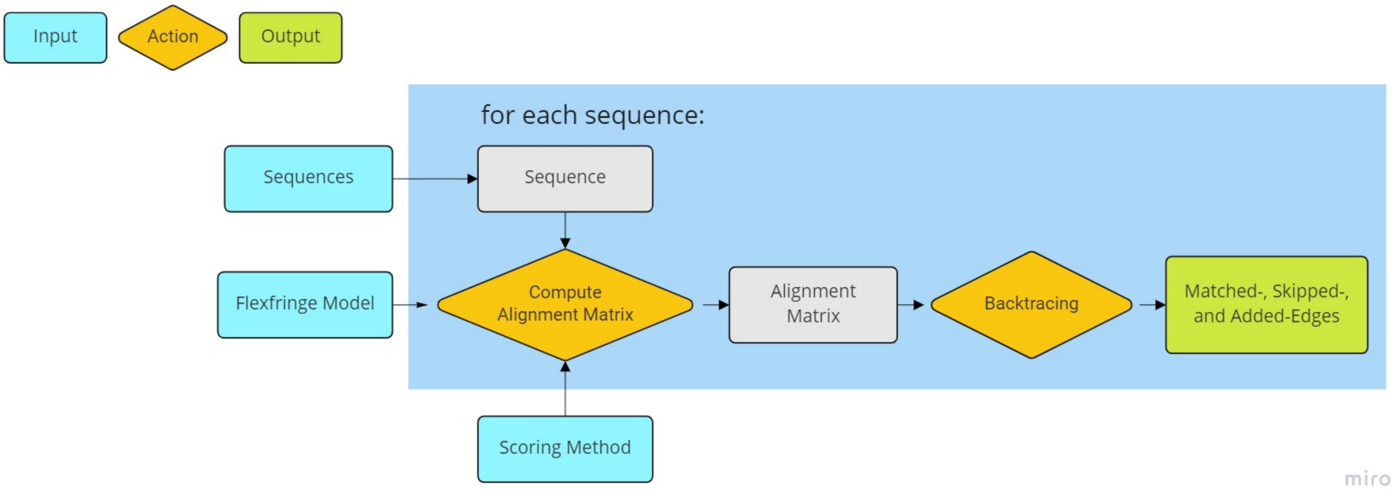


Figure 3.2: Overview of SAAT . For every sequence, an alignment matrix is computed. Using backtracing, the best alignment is computed from the alignment matrix.

3.2.1. Input Sequences & Automaton

The input sequences are aggregated from IDS alerts similarly as in SAGE [26]. The input automatons are generated using FlexFringe. FlexFringe is the state-of-the-art tool for learning state machines. As we only have positive data, the Alergia state merging algorithm [8] is used. Cycles and self-edges are removed from the automaton, as the alignment matrix cannot contain duplicate rows. These cycles can be found in multiple ways. A simple breadth-first search is used in this work, but other options, such as [39] will work too.

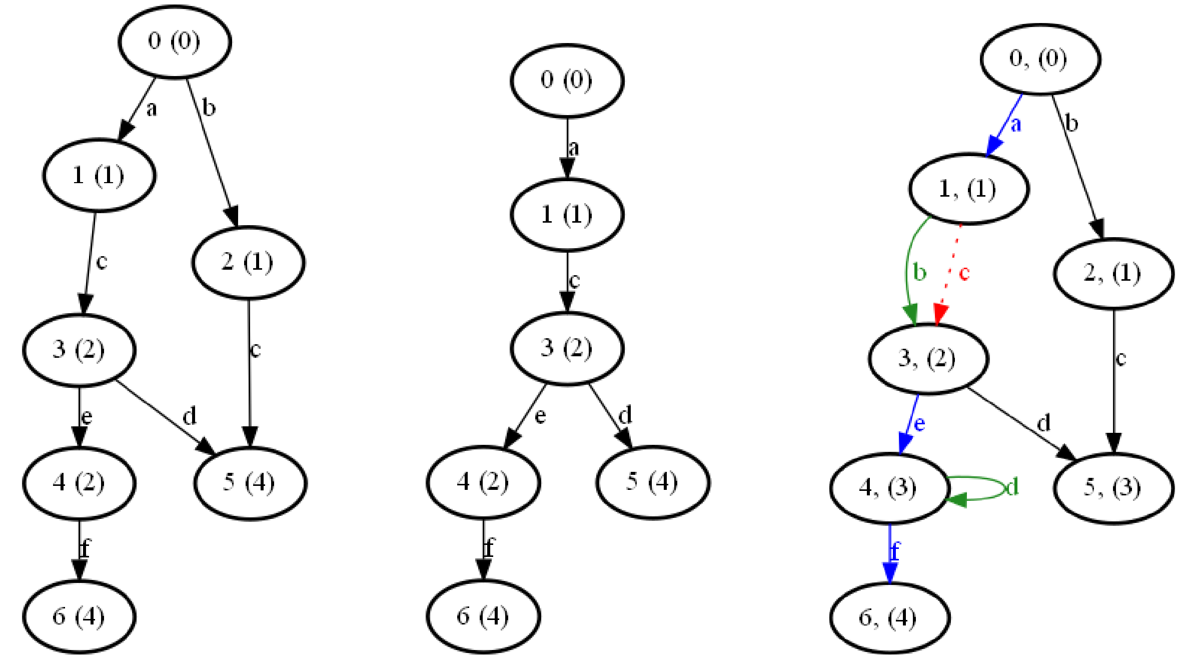
3.2.2. Computing the subtree

Just as in SAGE [26], the attack graphs are generated on a per-victim, per-objective basis. Due to this, the assumption is made that the first symbol of the sequence, the objective, is always correct. This means that first edge, the edge from root to a node N with the respective symbol s, is always *matched*. With this in mind, other nodes that are not reachable from N are not considered for matching. We call all reachable nodes from N, together with the root and the first edge with s the s-subtree of the automaton:

Definition 3.2.1 (s-subtree of A)**.** The set of all nodes reachable from subtree root node N, where N is the node reached from traversing the edge with symbol s from the root node of the automaton A.

Figure 3.3b shows an example of the a-subtree. Note that since we run FlexFringe with the `markovian` option, the root node can only have one outgoing edge with symbol a.

The s-subtree is computed using a breadth-first search from the subtree root node. The more different objectives (outgoing edges of the root node) the learned automaton has, the smaller the subtree will be compared to the original tree.



(a) Example automaton A. The number in parentheses is the level of the node.

(b) The $a-$subtree of A.

(c) The matched trace $a \rightarrow b \rightarrow e \rightarrow d \rightarrow f$

Index	Level	Transition	Symbol		0: a	1: b	2: e	3: d	4: f
0				0	-2	-4	-6	-8	-10
1	0	0 → 1	a	-2	4	2	0	-2	-4
2	1	1 → 3	c	-4	2	0	-2	-4	-6
3	2	3 → 4	e	-6	0	-2	4	2	0
4	2	3 → 5	d	-6	0	-2	-4	2	0
5	3	4 → 6	f	-8	-2	-4	2	0	6

(d) The alignment matrix

Figure 3.3: An example tree and alignment matrix, with Linear Scoring (*MATCH*=4, *MISMATCH*=-4, *GAP*=-2)

3.2.3. Computing the Alignment Matrix

The alignment matrix is computed in similar manner as the TS algorithm in Section 2.7. It is computed for every sequence, and can afterwards be used to find to which edge a symbol of the trace is best matched.

Recall: The matrix has size $(m+1) \times (n+1)$, with m = the amount of edges of the automaton, and n = the length of the sequence. The rows thus represent edges, while the columns represent the symbols. The initialization is done similarly as the TS algorithm, with the rows multiplied by the edge level, and the columns by the index of the symbol. An example of a filled in alignment matrix is shown in Figure 3.3.

Edge Levels In the TS algorithm, edge levels are used in the `GetThreeNeighbours` method to retrieve the nodes that reach to the current node. As demonstrated in Figure 3.3a edges can have different levels based on the different paths taken to that node. Having cycles in the model makes the edge level even more ambiguous, as then it can have an infinite amount of different levels for every time the loop is taken. This edge level check is thus dropped, as it does not make a difference to the algorithm. The most important check is that the nodes are a parent to the current node. This means it is possible more than three cells are considered.

Linear Scoring In [40], a Linear scoring system is used, where the scores for match, mismatch are constant, and the penalty for a gap scales linearly with the length of the gap. This method is best used when all symbols are equally important to match. Unless stated otherwise, linear scoring is used.

Dynamic Scoring A Dynamic scoring system is introduced in in [40], but it is not well-defined. Thus, it will be redefined here. Dynamic scoring makes the *MISMATCH* and *GAP* penalties scale with the level of the edge. The *MISMATCH* and *GAP* scores are first computed the same way as Linear scoring, but then multiplied by a scaling factor. The *MATCH* score remains the same, because changing all three scores results in the same (relative) scores.

This scaling factor s is determined by the level of the edge l, and a scaling parameter p: The scaling factor is computed as $s = p^l$. In Chapter 4, the optimal scaling factor will be determined.

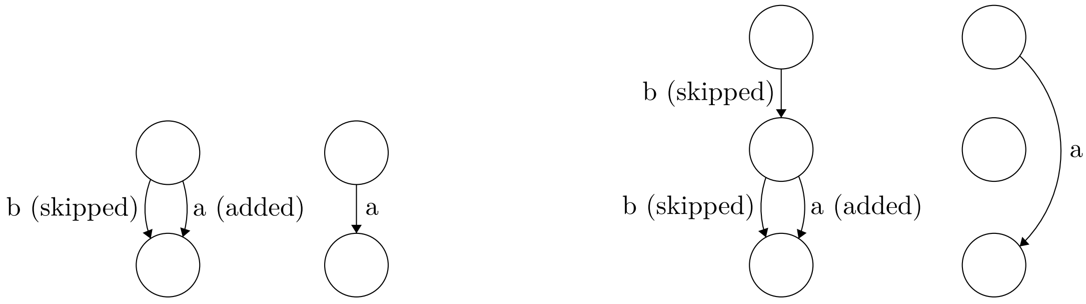
3.2.4. Backtracing to the Best Alignment

The TS algorithm will return a list of added, or correct edges. However, it is important to know which of those edges (and corresponding nodes) come from the trained model, and which do not. This distinction makes it easier to interpret the resulting attack graph, as it shows if attackers added or removed steps from their attack.

To achieve this, some backtracing steps need to be done. This backtracing will "match" the output sequence to a list of edges, the *matched sequence*, that contains only nodes and edges that are in the model. This matched sequence is created using the following properties:

- correct edges remain in the matched sequence
- A combination of a skipped edge next to an added edge is transformed into an edge in the matched sequence. See Figure 3.4a.
- Multiple skipped edges followed by an added edge are transformed into a single edge, starting at the source of the first skipped edge, and ending at the target of the added edge. See Figure 3.4b

Notice that a *GAP* in the alignment matrix (Table 3.3d) does not always lead to a skipped edge. In Figure 3.3c, the first *GAP* leads to a combination of a *skipped* and *added* edge, while the second *GAP* results in just an *added* edge.



(a) A combination of a single skipped and added edge get transformed into an edge with the added edge's label (a).

(b) Multiple skipped edges.

Figure 3.4: Example situations to make a matched sequence. In both subfigures, the left part is transformed into the right part.

Algorithm 4: Create the *matched* sequence from the alignment output

```
Input: edges: $E$
Output: matched sequence: $S$
$S = []$
$i = 0$
while $i <$ len $(E)$ do
    $edge = E[i]$
    $i = i + 1$
    if type($edge$) == SKIPPED:
        $last\_edge = S[-1]$
        while type($E[i]$) == SKIPPED do
            $i = i + 1$
        $new\_edge = (\textbf{source} = last\_edge.\textbf{target}, \textbf{target} = E[i].\textbf{target}, \textbf{name} = E[i].\textbf{name})$
        $S$.append($new\_edge$)
    else:
        $S$.append($edge$)
```

3.2.5. Score of the Alignment

The alignment system gives a score to the computed alignment. This score gives an indication of how well the sequence is aligned to the automaton. However, using the score as-is always gives longer traces a higher score, as more points can be scored. Thus, a normalised score is used, which we define as:

Definition 3.2.2 (Normalised Score)**.** The normalised score is the score of the trace, mapped to the range of possible scores for that scoring system. The range of possible values is [0,1].

Given s the score of the alignment and n the length of the trace, the normalised score S_{norm} is:

$$S_{max} = n * MATCH \tag{3.1}$$

$$S_{min} = n * MISMATCH \tag{3.2}$$

$$S_{norm} = \frac{s - S_{min}}{S_{max} - S_{min}} \tag{3.3}$$

Since **MATCH** is the maximum score possible for a single symbol, it is clear that a score of 1 can be reached by having all symbols in the sequence aligned to a transition in the automaton.

The normalised score can be used as a metric to quantify how well a sequence fits the model. Consequently, it can also be used to quantify how well a model fits the data it is trained on.

3.2.6. Efficiency Analysis

When the predecessors for every node are determined in advance, the runtime complexity of the alignment of a single sequence requires $O(m * n)$ time, with m the amount of edges in the automaton and n the length of the sequence. Note that since the s-subtree is first computed, the runtime complexity can be reduced to $O(k * n)$ (with k being the amount of edges in the s-subtree) if all subtrees are saved or predetermined for each outgoing edge of the root. Finding the s-subtree is done using a breadth first search, which requires $O(l * m)$ time, with l the amount of nodes.

Without subtrees, the space complexity is in order $O(m * n)$, which is the space required for the alignment matrix and predecessor edges. With the saved subtrees, the space complexity is theoretically at worst $O(m(m * n))$. In practice however, it will require less space as most subtrees will not be connected and will not overlap.

3.3. Iterative Models

In this Section **RQ3** is answered by using SAAT to improve the quality of a learned model. SAAT computes the differences (or alignments) between the training data and the model. From these differences, it can be deduced where the model falls short and how it can be improved. The differences are used to generate a new FlexFringe training file, resulting in a new (and hopefully improved) model. By iteratively doing this process, the model can converge to a (possibly local) optimum.

The new training data is generated according to a set of heuristics, which will all be evaluated, to see which combination performs best. The input is a list of alert sequences. The general outline of this method is as follows:

1. Aggregate the alerts sequences into episode sub-sequences (ESS), and generate a FlexFringe training file from these ESS.
2. Use FlexFringe to learn a model from this training file.
3. Compute the alignment between the learned model and the **original** ESS (from step 1).
4. Generate a new FlexFringe training file from these matched sequences, according to a set of heuristics.
5. Repeat steps 2 to 4 with the new training file, until a maximum number of iterations or until converged.

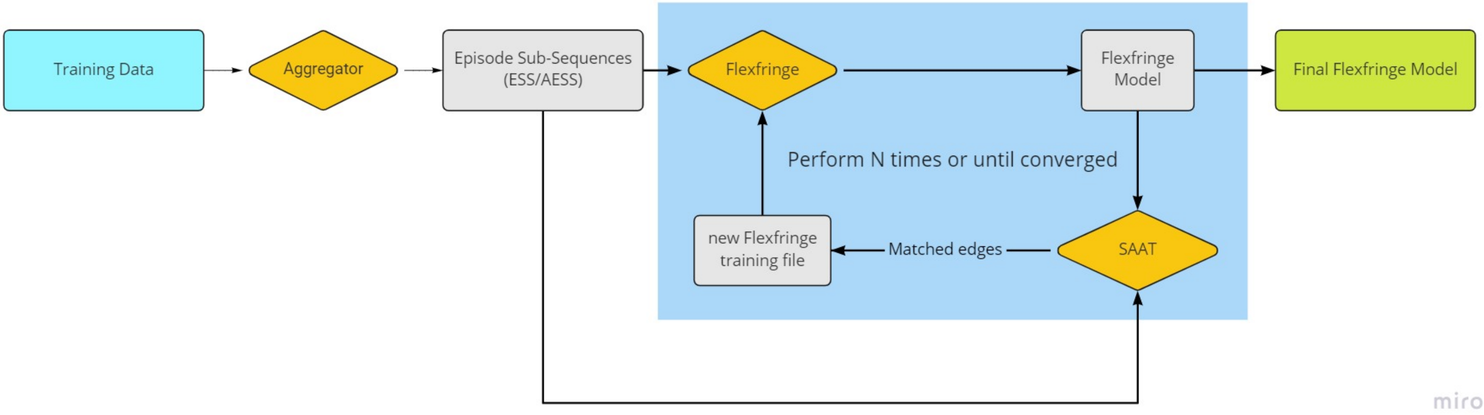

Training Data
Aggregator
Episode Sub-Sequences (ESS/AESS)
Flexfringe
Flexfringe Model
Final Flexfringe Model
Perform N times or until converged
new Flexfringe training file
Matched edges
SAAT
miro

Figure 3.5: Overview of iterative models

FlexFringe is run with the `printblue` option. This will output the sink states in the model, which will help the alignment for traces which end in a sinks state.

Initially, only matched edges will be used for the new training data. For example, if there is only one ESS, with $a \to b \to c \to d$, of which the edges a, c, and d are matched (and b is skipped), the new training data will only contain $a \to c \to d$.

The maximum number of iterations will be set to 16. This number should be large enough to converge, while keeping the training time low enough.

3.3.1. Model Evaluation

To determine the quality of the learned iterative models, the normalised score (3.2.2) is computed for each of the matched sequences. If there is an error, the sequence will get the score 0. The average normalised score is computed over all matched sequences, and quantifies how accurate the model represents the data. We opt for this method instead of other metrics like perplexity, since the sequences can be only partially present in the model. The probability of such a trace would be 0, but the normalised score will give a score between 0 and 1. The average normalised score is saved for each iteration. When the average normalised score is the same for at least 3 iterations, we say that the score has converged. From all iterations, the model with the best average normalised score is picked as the final model, and is used to generate the attack graphs.

3.3.2. Model Improvement Heuristics

In this section, several heuristics are introduced which hypothetically improve the iterated model. Every possible combination of these heuristics is tested, and the results are shown and discussed. From this evaluation, the best combination of parameters is used to train the final model.

A: Use Added Edges Normally, only the matched edges are used for the new training data. Added edges of the matched sequence can also be used, allowing symbols to be readded into the model. This can help with sequences where FlexFringe has removed such an edge.

B: Remove Sequential Duplicates To make the model more concise, sequential duplicates in the training data can be removed. As SAAT can recover these symbols in the form of added edges, this change shouldn't affect the resulting attack graphs too much, but does reduce the amount of nodes. However, this change will result in a lower average normalised score3.2.2, as there will be more added edges instead of matched edges.

C: `symbol_count` and `state_count` FlexFringe parameters Initially, `symbol_count` and `state_count` are set to 2 and 4 respectively. By reducing these values, the condition for preventing a merge is relaxed, resulting in more merges. This will result in more concise models. When using this heuristic, the `symbol_count` and `state_count` parameters are both set to 0.

D: Add missing traces If the objective of a sequence (the first symbol) is not present in the model, the trace can never be matched. When this happens, this sequence can be added back to the training data. This makes sure that objectives always stay in the training data, and all objectives lead to an attack graph.

E: Keep High Severity States Since the attack graphs are (mostly) about the high-severity states, we can choose to always leave them in the sequences, even if they are skipped. This will lead to the most "complete" model in a sense, as all possible attack paths will remain present. Note that in the final alignment before generating the attack graphs, these states can still be skipped.

3.4. Conclusions

In this Chapter, we described how sequence-automaton alignment can be used to iteratively improve a learned model. The infrequent sequences can be re-learned back in the model as they are aligned to the frequent ones. The alignments also make the model more robust to small changes.

The normalised score is introduced to quantify how well a sequence fits to the model. Averaging this value over all sequences describes how accurate the model represents the training data.

Several heuristics are introduced to improve the training data used for the iterative process.

4

SAAT Evaluation

In this chapter, SAAT 's sequence alignment algorithm is evaluated, and **RQ2** is validated. First, the used dataset is discussed. Then, seven different test cases are described and evaluated. Last, the results are shown and discussed.

4.1. CPTC-2018 Dataset

As mentioned in Chapter 1, real intrusion alerts are incredibly hard to come by due to privacy and security concerns. Instead, a security penetration testing competition is used as a surrogate. They provide an ideal controlled environment for multi-stage attacks. We evaluate SAAT on the CPTC-2018 dataset [32], a public IDS alert dataset that contains Suricata alerts generated by six different teams. Their objective is to compromise a network infrastructure. Each team uses a fixed IP address for their network traffic. The infrastructure also hosts an IDS. All IDS alerts are attributed to one of the teams. Just like in real-world SOCs, there is no ground truth knowledge on attacker strategies and progression. In [25] a more detailed explanation on this infrastructure and the dataset is given.

The raw IDS alerts are first parsed and aggregated into Episode Sub-Sequences (ESS), similarly as in SAGE.

4.2. Accuracy Testing

To verify that SAAT works as intended, seven accuracy testing experiments are evaluated. In these experiments, the focus lies on whether SAAT can recover the modifications that were made to the data. A test case will be considered as successful when SAAT is able to completely recover the original trace. For example, if c is removed from the trace $[a \to b \to c \to d]$, SAAT should be able to recover $[a \to b \to d]$ back to $[a \to b \to c \to d]$. If any of the node ids or symbols is wrong, the test is not correct.

4.2.1. Test Cases

There are many situations in which new input sequences can differ from the model. Some examples:

- An attacker can be more experienced or have newer techniques, resulting in less alerts being generated by the IDS.
- An attacker can still be learning, and trying all sorts of techniques, which generates lots of "filler" alerts between actual important alerts.
- An IDS might be misconfigured or outdated, resulting in missing alerts.
- An adversary might slightly change his Techniques, Tactics & Procedures, which leads to missing or different alerts.

These situations can be tested using a simple set of modifications. All these modifications take into account that the first transition cannot be changed, since it is the objective.

- **REMOVE** Removes n symbols. Possible variations are the removal of n nodes in a row, or n nodes anywhere in the sequence.
- **ADD** Adds n symbols. A random symbol is used, instead of sampling from the test data. This will make sure that an added symbol will not suddenly align with another edge in the model.

Using these simple modifications, more complex modification can be made, such as the modification of an alert (a REMOVE and an ADD), or swapping two alerts (two REMOVE's and two ADD's). The following test cases will be evaluated:

- REMOVE-1: Removes 1 alert.
- REMOVE-2: Removes 2 alerts.
- ADD-1: Adds 1 alert.
- ADD-2: Adds 2 alerts.
- SWAP-1: Swaps 2 sequential alerts.
- MODIFY-1: Modifies 1 alert.
- MODIFY-2: Modifies 2 alerts.

These test cases capture most of the real-world scenarios in which an attack (partially) changes.

To generate the modified sequences, all unique attack paths in the dataset are retrieved, using a Breadth First Search. Then, based on the test case, the attack paths are generated. All possible permutations are considered, except for the case where the first or last node is removed or changed.

Example For REMOVE-1, if the attack path is n nodes long, then there are $n - 2$ possible attack paths: One for every possible removed node. For the sequence $[a \rightarrow b \rightarrow c \rightarrow d \rightarrow e]$, 3 test case sequences are generated:

- $[a \rightarrow c \rightarrow d \rightarrow e]$
- $[a \rightarrow b \rightarrow d \rightarrow e]$
- $[a \rightarrow b \rightarrow c \rightarrow e]$

4.2.2. Results

In Table 4.1, the results for the test cases mentioned in Section 4.2.1 are presented. SAAT is almost able to perfectly recover from these simple modifications, and works as intended. MODIFY-2 has 6 incorrect cases. Most of these are cases where two sequential symbols are modified. The next symbol after these modified symbols is incorrectly labelled as *added* instead of *matched*.

test case	number of tests	correct	accuracy
REMOVE-1	464	464	**1.0**
REMOVE-2	1855	1855	**1.0**
ADD-1	464	464	**1.0**
ADD-2	1855	1855	**1.0**
MODIFY-1	464	462	**1.0**
MODIFY-2	1855	1849	**0.996**
SWAP-1	380	359	**0.944**

Table 4.1: Results for the verification experiments of the CPTC-2018 dataset.

4.3. Parameter Tuning

To determine how much impact the different parameters have on the alignment results, we will perform parameter tuning experiments on two of the above test cases. For these experiments, the test cases **ADD-2** and **MODIFY-2** have been chosen. The **ADD-2** was picked to see the range of the parameters that still works for a relatively easy test case. The **MODIFY-2** test case was picked to see whether small changes in these parameters actually matter for the final outcome.

4.3.1. Scoring Parameters

Recall that for both Linear (Section 18) and Dynamic scoring (Section 3.2.3), three parameters are used: the match score, mismatch score and gap score. For each of these, the following values were tested:

- Match score *MATCH*, with values from 2 to 6.

- Mismatch score *MISMATCH*, with values from -10 to -1 (included).
- Gap score *GAP*, with values from -6 to -1.

This means there are $5 * 10 * 6 = 300$ possible different options. These values were chosen with the following intuition: *MATCH* needs to be higher than *MISMATCH* and *GAP*, to reward correct matches, and *GAP* needs to be higher than *MISMATCH* to align symbols in incorrect places instead of just calling them a mismatch.

For Dynamic Scoring, an extra parameter is used: The scale factor p. To test the best value of p, the other three parameters (match score, mismatch score and gap score) were set to one of the best performing values for Linear Scoring, namely $(3, -8, -3)$. p was tested with values from 0.6 to 1.3, with a step of 0.02

4.3.2. Results

The results of the parameter tuning results will be discussed here.

Linear Scoring

For both the **ADD-2** and **MODIFY-2** test cases, the parameters did not have a big impact. Figure 4.2 shows some results for the **ADD-2** test case. Only when *GAP* is smaller than *MISMATCH* the accuracy was not 1.0. Some of these results are shown in Figure 4.3 This proves the intuition that *GAP* needs to be higher or at least equal to *MISMATCH*. This also shows that for models and sequences of this size, no parameter tweaking is needed to achieve good results.

MATCH	MISMATCH	GAP	Accuracy
2	-10	-6	1.0
2	-10	-1	1.0
6	-10	-1	1.0
6	-10	-6	1.0
3	-8	-3	1.0

Table 4.2: Some top results for Linear scoring and test case **ADD-2**. The parameter combinations listed were picked at random, as almost all had an accuracy of 1.0.

MATCH	MISMATCH	GAP	Accuracy
2	-1	-6	0.58
2	-1	-5	0.63
3	-1	-6	0.63
2	-3	-6	0.92
2	-2	-6	0.92
6	-1	-6	0.92
6	-1	-5	0.92
2	-5	-6	0.96
2	-4	-6	0.96

Table 4.3: Some bottom results for Linear scoring and test case **ADD-2**. Note the gap in the table to show more rows. Only when $GAP < MISMATCH$, the accuracy is smaller than 1.0.

Dynamic Scoring

The results in Figure 4.1 show that the scale factor p does not have any significant impact on models and sequences of this size.

4.4. Conclusions

In this Chapter, we have verified that SAAT works as intended. The results show that SAAT performs great on the CPTC-2018 dataset, and can handle most test cases with ease.

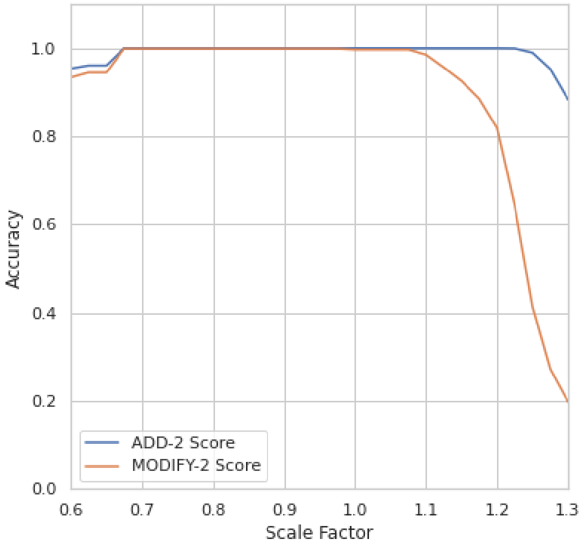


Figure 4.1: Results of Dynamic Scoring for both *ADD-2* and *MODIFY-2* with variable scaling factor p, and $MATCH = 3, MISMATCH = -8, GAP = -3$. The scaling factor has no real impact on models of this size, except when set to extreme values.

The parameter tuning experiments show that SAAT does not need extensive parameter tuning to work as intended. Dynamic scoring does not have a significant impact on the results, so it will not be used for the evaluation of Chapter 5. Further experimentation should be done on datasets with larger models, longer sequences, or more difficult experiments to test if the parameters or dynamic scoring can make a significant impact.

5
Iterative Model Generation Evaluation

In this Chapter, **RQ3** is validated by evaluating the iterated model method of SAAT (Section 3.3). We will also analyse the learned models.

5.1. Evaluation Setup

The method will be evaluated on the CPTC-2018 dataset (Section4.1). After training the model as explained in Section 3.3, the normalised score will be computed for every sequence. These will be averaged to the final score. We opt for this method instead of other metrics like perplexity, since the sequences can be only partially present in the model. The probability of such a trace would be 0, but the normalised score will give a score between 0 and 1, and will reflect partially correct sequences.

Next to the normalised score, the size of the model is taken into account. A smaller model can be less descriptive, but also easier to interpret. As such, we favour smaller models over bigger models.

The original model of the CPTC-2018 dataset, trained with the regular parameters, has an average normalised score of 0.656, and is 256 nodes large.

5.2. Heuristics Evaluation Results

Every combination of heuristics described in Section 3.3.2 is tested 6 times. In Table 5.1, the top 10 results of all combinations can be seen. They are the average of 6 runs. The standard deviation is listed next to the averages.

Without any of the heuristics set to true, the average score is 0.662, and the model has 119.7 nodes. Compared to the original score of 0.656 and 256 nodes, it has half the amount of nodes, and a slightly higher score. This means the data can be described just as well with only half the amount of nodes. The best combination of heuristics improves the average score from 0.662 to 0.706.

It is surprising that the top result and a few others in the top 10 have **B** (Remove Sequential Duplicates) true. This heuristic usually makes the average score worse, as more *added* edges are in the resulting sequences. As these models still have a high average score, they must capture the data (relatively) better than the others. It is also interesting that heuristic **C** is false in almost all top 10 results, except for the best combination. The standard deviation is pretty low for the top combination, which means it consistently scores this high. The top 10 scores are still close to each other, meaning that the heuristics do not have a huge impact on the final outcome. This proves that the core algorithm of iterative SAAT works as intended.

5.3. K-Fold Cross Validation

To further show the validity of our approach, we have also performed an evaluation using k-fold cross validation, with $k = 5$. We have chosen for this approach as there is limited input data. The training data are used to generate the iterated model, and the test data are used to compute the average normalised score used for the final evaluation. An overview can be seen in Figure 5.1.

5.3.1. K-Fold Cross Validation Results

In Figure 5.2, the results of 5-fold cross validation can be seen. Each data point is the average of all 5 training-test splits. The evaluation was run 10 times, resulting in 10 data points. The aggregated

A	B	C	D	E	Avg. Score	SD Score	Avg. Size	SD Size
FALSE	**TRUE**	**FALSE**	**TRUE**	**FALSE**	**0.706**	**0.014**	**131.3**	**8.825**
FALSE	FALSE	TRUE	FALSE	FALSE	0.703	0.016	146.2	5.757
TRUE	TRUE	TRUE	TRUE	TRUE	0.699	0.019	138.7	5.617
FALSE	FALSE	TRUE	TRUE	TRUE	0.699	0.029	143.8	4.810
FALSE	FALSE	FALSE	TRUE	FALSE	0.694	0.022	136.0	4.933
FALSE	FALSE	TRUE	FALSE	TRUE	0.693	0.038	136.8	13.359
TRUE	FALSE	TRUE	FALSE	TRUE	0.692	0.008	137.8	8.572
TRUE	FALSE	TRUE	FALSE	FALSE	0.691	0.019	135.5	15.229
TRUE	TRUE	TRUE	FALSE	TRUE	0.690	0.018	134.8	9.227
FALSE	TRUE	TRUE	TRUE	FALSE	0.687	0.011	140.0	11.299
FALSE	FALSE	FALSE	FALSE	FALSE	0.662	0.025	119.7	6.80

Table 5.1: Top 10 results of all combinations of heuristics. The best combination reduces model size by 49%, while improving the score.

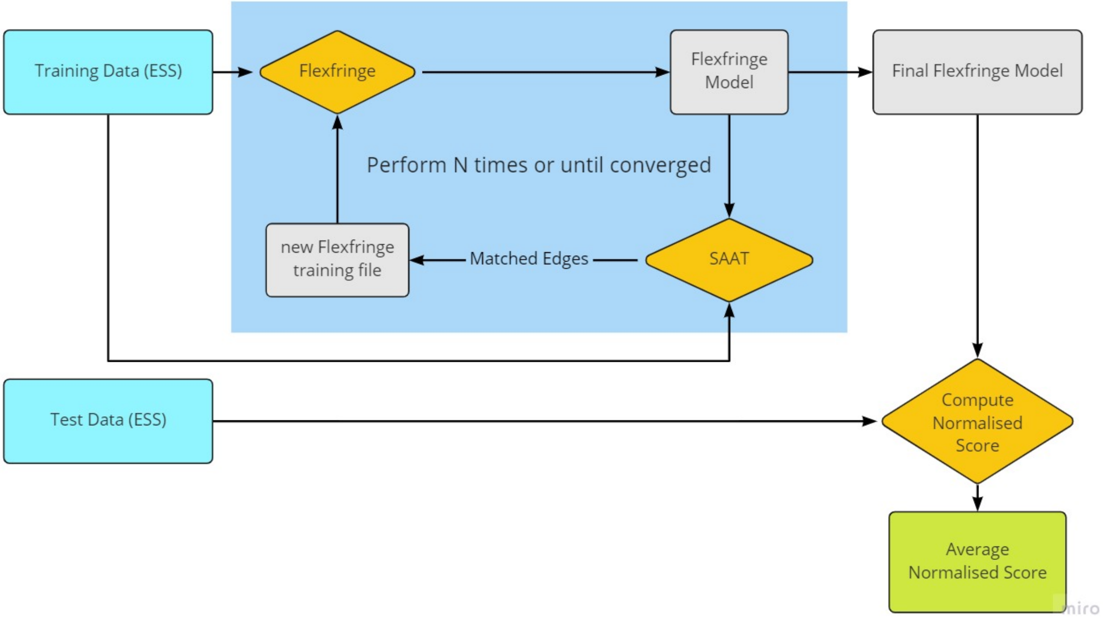


Figure 5.1: Overview of k-fold cross validation setup.

averages of all these results can be seen in Table 5.2, and the full results can be seen in Appendix C. In Figure 5.2, we can see that a higher score on the training data also leads to a higher score on the test data.

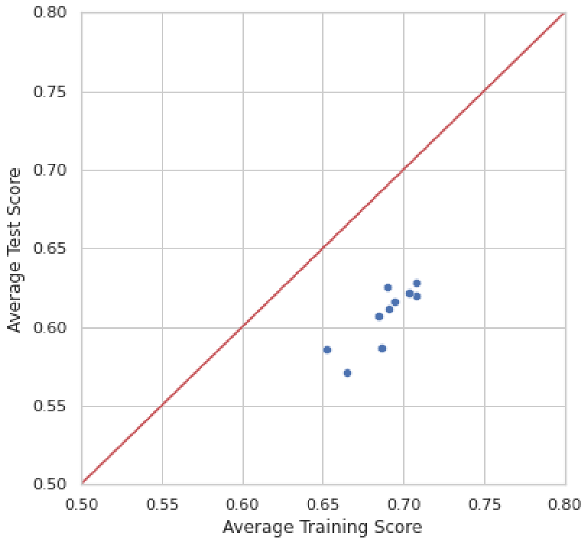


Figure 5.2: Aggregated results of k-fold cross validation with $k = 5$. The data points correspond to 10 different experiment runs. For clarity, the line $y = x$ has also been plotted. The results show a positive trend, meaning that a higher training score results in a higher test score.

Train Score	Test Score	SD Test Score	Model Size
0.688	0.607	0.057	104.9

Table 5.2: Aggregated average results and standard deviation of 10 runs of 5-fold cross validation.

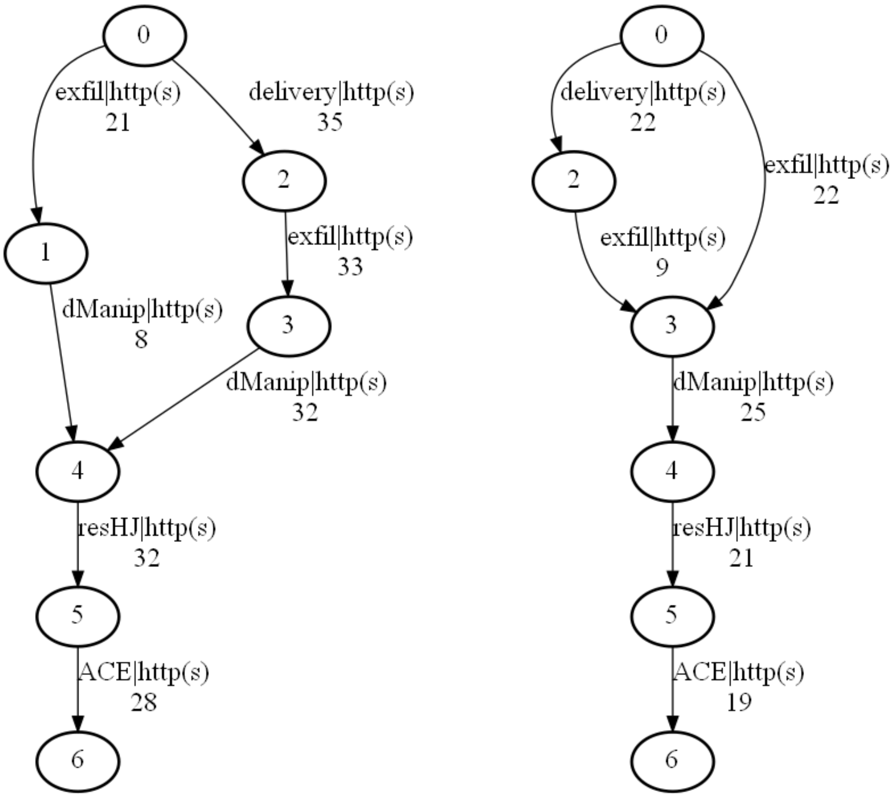
5.4. Learned Model Analysis

The best iterated model (Appendix B.2) is compared to the original model (Appendix B.1) regarding model interpretability, accuracy, and size.

The first change that is immediately visible is that the iterated model is much smaller, having only 129 nodes, instead of the original 256. Because SAAT can add edges between nodes, the model can be smaller without losing the ability to express the data. Figure B.3 shows the original and iterated model without sink states. Edges with less than 10 occurrences are highlighted red. In the original model most of the edges are thus highlighted red. This makes sense for the objectives and other high-severity alert categories, as these are exactly the paths we want to keep in the model during training. However, for low-severity alerts, these edges do not make the model more interpretable or explicit, as they can be recovered by SAAT .

Amount of Learned Sequences Recall that in the original model 259 of the 536 learned sequences (48%) prematurely end in a sink state. In the iterated model, only 196 (37%) end in a sink state **after alignment**, meaning that we have learned a bigger portion of the data.

Merged Paths In the iterated model, some attack paths have been merged, that were separate in the original model. An example of this can be seen in Figure 5.3. Figure 5.3a shows a (simplified) view of the two different paths in the original model. In Figure 5.3b they have been merged, resulting in a model without node 1. In the training data all sequences that start with [exfil|https() → dManip|https() → resHJ|https()] are followed by “ACE|http(s)”, so this is a better representation of the data.



(a) Attack paths in original model

(b) Attack paths in iterated model

Figure 5.3: Nodes 1 and 3 have been merged in the iterative model, as the training sequences are very similar.

Low Frequency Edges In Figure B.3, the learned models are shown without sink states. All edges with less than 10 occurrences are highlighted red. In the original model most edges are low-frequency edges, which are needed to learn all low-frequency states. In the iterated model most of these have been filtered out as they can be later recovered with the alignment system.

5.5. Conclusions

In this chapter, we have shown how the alignment computed by SAAT can be used to improve a learned model. Performing this operation in an iterated process can reduce the model size from 256 nodes to 131 nodes (a 49% decrease), while actually improving how well the model captures the data. The infrequent traces have been aligned with the frequent ones, resulting in a better score *after* alignment. We have also introduced several heuristics that can improve this iterative process, and determined the best possible combination of them for the CPTC-2018 dataset. This validates that iterative SAAT works as intended.

Using K-Fold Cross Validation, we have shown that the system also works on a training-test split, and that an increase of the normalised score of the training data, also leads to a higher test data normalised score. This shows that if the model can be further improved in the future, even better test results can be obtained.

6

SAAT for Anomaly Detection

In this chapter, a method for using SAAT for anomaly detection is inroduced, and **RQ4** is answered. Anomaly detection is the task of identifying anomalous events, observations, or outliers. In the context of cyber security, it is often used to determine whether a specific alert should trigger an alert in the IDS. It helps systems such as an IDS to make better decisions, or even make it possible to detect new kinds of attacks.

The iterated model method described in Section 3.3 can be used for this purpose. From the training data, an iterated model is learned. Each sequence in the test data will be aligned to the model, and based on the alignment, it is decided whether the sequence is an anomaly or not.

6.1. HDFS Dataset

The HDFS dataset is based on logs from Hadoop File System. The HDFS is a system that handles persistence, designed for large files and batch processing. It was run on Amazon's Elastic Compute Cloud (EC2). The logs are collected from a Hadoop cluster from over 200 EC2 nodes, resulting in 24 million lines of logs over a period of two days. The logs contain messages indicating runtime performance.

Each alert is first parsed into a *log key*. In this work we use the same training and test data as in [13], which are generated by Spell [12], one of the state-of-the-art log parsers. Note that any log parser can be used, as long as the log keys are extracted. The training and test data is available on the DeepLog GitHub[1].

Dataset Analysis A sample of the normal or benign data can be seen in Appendix D. The dataset contains lots of messages that happen in parallel (or could have), which means the order of messages is not always relevant. For example, all message start with either a 5 or 22, but both are okay. After the 5's and 22's, a lot of 11's and 9's happen, often repeatedly, and in different order. State machines are usually not a good choice for parallel data, but by iteratively aligning the infrequent sequences to the frequent ones, the parallelism will be learned.

The training data consists of 4855 (normal) training sequences, 16838 abnormal test sequences, and 553366 normal test sequences. This means we need to learn a model on only a small portion of the normal data.

6.2. Iterated Model Learning

The model is learned using the method described in Section 3.3. The training process looks nearly the same, but a few differences are made to better learn this dataset: As there is a lot of parallelism in the data, heuristic B (Remove Sequential Duplicates) will not be used. This will do the opposite and filter out the parallelism. Heuristic A (Use Added Edges) will be used however, as this will give FlexFringe the opportunity to learn all parallel sequences. Also, the `sinkcount` FlexFringe parameter will be set to 50, to ensure (almost) all training data will be aligned to the model, and will not fall in a sink state.

The final learned model can be seen in Appendix E. It consists of 84 nodes, and has an average normalised score of 0.966. This means the model very accurately fits to the training data.

[1] `https://github.com/wuyifan18/DeepLog/`

6.3. Determining Anomalies

After a sequence has been aligned to the model, the sequence needs to be labeled as anomalous or benign. There are three conditions which label a sequence as an anomaly:

A. Any symbol of the sequence is not in the alphabet of the model.

B. The aligned sequence did not end in an ending state (Definition 6.3.1).

C. The normalised score (Definition 3.2.2) of the aligned sequence is less than a predefined score threshold k.

If all of these are false, the sequence is labeled benign. Condition A will makes sure sequences with log messages that have not been seen before will trigger an alert. Since the dataset contains file system events, we know that every used file (every sequence) will need to finish with a “close” event. Condition B will enforce this, and will label a sequence when it is not properly closed. Condition C works as a last resort, and can be tweaked by the score threshold k. Setting k close to 1 will make the system more “strict”, as the sequences will need to follow the model more closely.

The amount of times any of these anomaly conditions were triggered will be saved and used for analysis. We will set the score threshold $k = 0.1$, as all anomalous sequences will be filtered out using the first two conditions. However, when the alignment fails for whatever reason, the normalised score will be 0, and these sequences will be marked as an anomaly this way.

Definition 6.3.1 (Ending state)**.** The sequence is in an ending state if either the last aligned *matched* node has no outgoing edges, or the last aligned symbol is one of 2, 9, 11, 21, or 26. These specific symbols are the most occurring last symbols of the training data sequences.

6.4. Results

In Table 6.1, the false positive (FP) and false negative (FN) counts can be seen, as well as the standard metrics Precision, Recall, and F1-score (Section 2.1). They are shown compared to some standard models and DeepLog [13], the best-performing method. SAAT has a recall of 1.0, meaning that all anomalies were correctly labeled, a score no other method could achieve. The precision is not better than DeepLog however, and so is the F1-score. However, it could be argued that a recall of 1.0 is the most important metric in the network security domain.

	FP count	FN count	precision	recall	F-1 score
PCA	277	5400	0.98	0.67	0.79
IM	2122	1217	0.88	0.96	0.91
N-gram	1360	739	0.92	0.96	0.94
DeepLog	833	619	0.95	0.96	0.96
SAAT	**2130**	**0**	**0.89**	**1.0**	**0.94**

Table 6.1

One of the key strengths of using a deterministic automaton, is its explainability aspect: The decision for each prediction can be explained, evaluated, and reasoned upon. This is in stark contrast to most other machine learning methods, such as neural networks, where all decision-making is done black-box.

In Table 6.3, the amount of times the different anomaly conditions were triggered can be seen. Note that the conditions are checked in order, from top to bottom. When conditions 1 and 2 are swapped, the total amount of TP and FP remain the same, but some sequences will now fall in a different category. Those results can be seen in Appendix F.1.

In the first column, we can see that all anomalous sequences can indeed be labeled correctly using only the first two conditions. As such, the score threshold k can be a very small value bigger than 0, such as 0.1. This also means no finetuning needs to be done to compute the best value of k.

Apparently, there are 1985 benign sequences that contain symbols that were not in the alphabet of the model. These have been (incorrectly) labeled as anomaly. In Table 6.2, the amount of occurrences in the benign test data of each of those unknown symbols are shown. Note that the counts can be larger than 1985, as a sequence can have multiple unknown symbols.

There are 17 sequences in the training data containing all of 6, 16, 18 **and** 25. No other training sequences contain any of these symbols. It seems these sequences have been incorrectly left out during the training phase. The model can thus still be improved, by making sure these sequences remain in the model.

Symbol	Occurrences
6	2588
18	2514
25	2514
16	2513
20	222
28	12
27	3

Table 6.2: Occurrences of symbols in benign test data, that were not in the alphabet of the learned model.

There are no training sequences containing a 20, 27, or 28, so it is not surprising these sequences are labeled as anomalous by our system. The original messages from which these log keys were extracted, might have been infrequent but not alarming, which could explain them not being anomalous. This could be solved in future work by augmenting the alphabet of the model with additional symbols that are deemed benign.

Only 88 benign sequences did not finish in an ending state, which means the definition of ending states can be considered as correct. A closes look at these sequences reveals that they end with the symbols 3, 4, and 22. In the training data, only 1 sequence ends with a 4 or 22, and two sequences end with a 3. These sequences have been filtered out during training with the FlexFringe `state_count=5` parameter. We could lower this setting to 0 (heuristic C, Section 3.3.2), but testing has shown out that the total average normalised score is lower: 0.960 instead of 0.966. Still, this is a tradeoff that can be made, and further testing needs to be done to show if performs improves.

	TP	FP
symbol not in alphabet	10645	1985
not in ending state	6177	88
score threshold	0	0
failed	16	57

Table 6.3: Counts of the anomaly conditions for both TP and FP.

6.5. Conclusions

In this chapter, we have used SAAT for anomaly detection, and have answered **RQ4**. By iteratively training the model, the parallelistic nature of the data is captured. Using two of the three anomaly conditions, all anomalous sequences are detected, leading to a recall of 1.0. The precision is slightly lower than existing methods, as the model does not contain all needed sequences yet. Nevertheless, the F1-score is competitive compared to the state-of-the-art, and shows that sequence-automaton alignment can be used to learn models for anomaly detection.

7
Discussion

With SAAT we have created a solution for two problems in the field.

First, we have successfully used a sequence-automaton alignment algorithm to find the fitness of a sequence. Using simple modifications, such as *add* and *remove*, more complex ones can be described. The evaluation results in Section 4.2.2 show that the algorithm works very accurate for graphs of this size and sequences of this length. It is also disadvantage that the accuracy is 1.0 for most test cases, as it is hard to see the limitations of the method. We have not really seen the impact of the different scoring methods and parameters. Still, the results are very promising and allow for more concise models.

A limitation of our method is the inability to deal with cycles. Currently, edges that go back to previously seen edges are discarded, as each edge can only be present once in the alignment matrix. For our current datasets this is not a problem. In other types of processes however, this could pose an issue.

By iterative learning the models, we came up with a new way to deal with infrequent data. Instead of discarding it or not considering it at all, we align the infrequent sequences to the frequent ones. The results in Chapter 5 show that our method works and reduces the model size by half, while learning a bigger portion of the data. Of course, this is after alignment, meaning that when using such a model, the alignments always need to be computed to actually work with it.

The downside of learning the model with new training sequences, is that the original counts are lost. Some of the original information in the model, such as the distribution of the learned language, is thus gone. This also made it hard to compare the original and iterated models.

We show how SAAT can be used to learn the model of an anomaly detection dataset, resulting in a very well performing model. The F1-Score is not better than that of existing methods, but the recall of 1.0 is very promising, and shows the potential of this method. Every wrong prediction can be investigated and reasoned upon, which is a huge advantage compared to other machine learning methods.

8

Conclusion

In Chapter 1, we answer "**How does SAGE perform with noisy input data?**" by explaining the underlying algorithms of SAGE. As state machines are used, simple modifications to the input data cannot be recognized or recovered from, and will not generate an Attack Graph

How can we improve the robustness of SAGE using sequence-automaton alignment against noise, such as added or missing alerts? We answer this question in Chapter 3, where SAAT is introduced. SAAT performs global alignment between a sequence and an automaton, based on the TS algorithm. Using backtracing we recover the nodes and edges corresponding to the best alignment. The alignment is then used to recover from simple modifications.

In Chapter 3 we also answer "**How can we use the computed alignments to better learn infrequent behaviour?**", by introducing an iterative method to learn models. By re-learning the model from new sets of sequences, infrequent sequences are aligned to frequent ones. The performance of the resulting models are higher, as they better fit to the training data. On top of that, the model is also reduced in size by almost a half.

Last, we answer "**How can we use sequence-automaton alignment for anomaly detection?**" in Chapter 6, by showing an application for our method. The resulting iterated model has a very high fit to the training data. Using only two of three anomaly conditions, we correctly predict all anomalies. The F1-Score is not the best, but still competitive, compared to state-of-the-art methods.

9 Future Work

For future work, SAAT should be applied and evaluated on more datasets. Sink states could be used in the alignment process. If the alignment is done forwards instead of in reverse, then all states after a sink state can be seen as matched edges.

The order of the rows in the alignment matrix is determined by the edge level, which can be ambiguous in certain situations Section 3.2.3. However, in the backtracing step, an added edge is determined by two maximum values present in the same row, so an edge having multiple possible levels can influence the alignment. This can be fixed by using the edge level in the backtracing step too, but since edge levels can be ambiguous, this might not work as intended. A different metric (other than edge level) should be used to fix this problem.

The normalised score could take into account whether a low-, medium-, or high-severity state is being scored. For high severity states, the gains and losses should be higher.

The normalised score could also be computed with the ratio of matched, added and skipped edges. Making the score of added edges close to the score of matched edges could result in even smaller models, as there would be more emphasis on keeping only relevant edges.

For anomaly detection, the alphabet of the model could be augmented with additional symbols of which we know they are not anomalous. This will reduce the amount of false positives.

References

[1] Faeiz M Alserhani. “Alert correlation and aggregation techniques for reduction of security alerts and detection of multistage attack”. In: *International Journal of Advanced Studies in Computers, Science and Engineering* 5.2 (2016), p. 1.

[2] Marco Angelini, Nicolas Prigent, and Giuseppe Santucci. “Percival: proactive and reactive attack and response assessment for cyber incidents using visual analytics”. In: *VizSec*. IEEE. 2015.

[3] Joao Antunes, Nuno Neves, and Paulo Verissimo. “Reverse engineering of protocols from network traces”. In: *2011 18th Working Conference on Reverse Engineering*. IEEE. 2011, pp. 169–178.

[4] Shakuntala Baichoo and Christos A. Ouzounis. “Computational complexity of algorithms for sequence comparison, short-read assembly and genome alignment”. In: *Biosystems* 156-157 (2017), pp. 72–85. ISSN: 0303-2647. DOI: `https://doi.org/10.1016/j.biosystems.2017.03.003`. URL: `https://www.sciencedirect.com/science/article/pii/S0303264717300461`.

[5] Richard Ernest Bellman. *The Theory of Dynamic Programming*. Santa Monica, CA: RAND Corporation, 1954.

[6] Antonia Bertolino et al. “Automatic synthesis of behavior protocols for composable web-services”. In: *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 2009, pp. 141–150.

[7] Josep Carmona et al. “Conformance checking”. In: *Switzerland: Springer.[Google Scholar]* (2018).

[8] Rafael Carrasco and Jose Oncina. “Learning Stochastic Regular Grammars by Means of a State Merging Method”. In: Nov. 2002. ISBN: 978-3-540-58473-5. DOI: `10.1007/3-540-58473-0_144`.

[9] Paolo Milani Comparetti et al. “Prospex: Protocol specification extraction”. In: *2009 30th IEEE Symposium on Security and Privacy*. IEEE. 2009, pp. 110–125.

[10] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[11] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[12] Min Du and Feifei Li. “Spell: Streaming parsing of system event logs”. In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE. 2016, pp. 859–864.

[13] Min Du et al. “DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1285–1298. ISBN: 9781450349468. DOI: `10.1145/3133956.3134015`. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/3133956.3134015`.

[14] Paul Fiterau-Brostean et al. “Analysis of {DTLS} Implementations Using Protocol State Fuzzing”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 2523–2540.

[15] Ni Gao, Yiyue He, and Beilei Ling. “Exploring attack graphs for security risk assessment: a probabilistic approach”. In: *Wuhan University Journal of Natural Sciences* 23.2 (2018), pp. 171–177.

[16] H. Hindy et al. “A Taxonomy of Network Threats and the Effect of Current Datasets on Intrusion Detection Systems”. In: *IEEE Access* 8 (2020), pp. 104650–104675. DOI: `10.1109/ACCESS.2020.3000179`.

[17] Hao Hu et al. “Attack scenario reconstruction approach using attack graph and alert data mining”. In: *Journal of Information Security and Applications* 54 (2020), p. 102522.

[18] Kenneth L Ingham et al. “Learning DFA representations of HTTP for protecting web applications”. In: *Computer Networks* 51.5 (2007), pp. 1239–1255.

[19] Chirag Jain et al. "On the complexity of sequence-to-graph alignment". In: *Journal of Computational Biology* 27.4 (2020), pp. 640–654.

[20] Somesh Jha, Oleg Sheyner, and Jeannette Wing. "Two formal analyses of attack graphs". In: *CSFW*. IEEE. 2002.

[21] Vaddadi Naga Sai Kavya et al. "Sequence Alignment on Directed Graphs". In: *Journal of Computational Biology* 26.1 (2019), pp. 53–67. DOI: `10.1089/cmb.2017.0264`. URL: `https://doi.org/10.1089/cmb.2017.0264`.

[22] Kerem Kaynar. "A taxonomy for attack graph generation and usage in network security". In: *Journal of Information Security and Applications* 29 (2016), pp. 27–56. ISSN: 2214-2126. DOI: `https://doi.org/10.1016/j.jisa.2016.02.001`. URL: `https://www.sciencedirect.com/science/article/pii/S2214212616300011`.

[23] Stephen Moskal and Shanchieh Jay Yang. *Cyberattack Action-Intent-Framework for Mapping Intrusion Observables*. 2020. arXiv: `2002.07838 [cs.CR]`.

[24] Stephen Moskal and Shanchieh Jay Yang. "Framework to Describe Intentions of a Cyber Attack Action". In: *arXiv preprint arXiv:2002.07838* (2020).

[25] Nuthan Munaiah et al. "Characterizing Attacker Behavior in a Cybersecurity Penetration Testing Competition". In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2019, pp. 1–6. DOI: `10.1109/ESEM.2019.8870147`.

[26] Azqa Nadeem et al. "Alert-Driven Attack Graph Generation Using S-PDFA". In: *IEEE Transactions on Dependable and Secure Computing* 19.2 (2022), pp. 731–746. DOI: `10.1109/TDSC.2021.3117348`.

[27] Saul B. Needleman and Christian D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453. ISSN: 0022-2836. DOI: `https://doi.org/10.1016/0022-2836(70)90057-4`. URL: `https://www.sciencedirect.com/science/article/pii/0022283670900574`.

[28] Peng Ning, Yun Cui, and Douglas S. Reeves. "Constructing Attack Scenarios through Correlation of Intrusion Alerts". In: CCS '02. Washington, DC, USA: Association for Computing Machinery, 2002, pp. 245–254. ISBN: 1581136129. DOI: `10.1145/586110.586144`. URL: `https://doi.org/10.1145/586110.586144`.

[29] Peng Ning et al. "Building Attack Scenarios through Integration of Complementary Alert Correlation Method." In: *NDSS*. Vol. 4. 2004, pp. 97–111.

[30] David L. Olson and Dursun Delen. *Advanced Data Mining Techniques*. 1st. Springer Publishing Company, Incorporated, 2008. ISBN: 3540769161.

[31] Xinming Ou, Sudhakar Govindavajhala, and Andrew W Appel. "MulVAL: A Logic-based Network Security Analyzer." In: *USENIX security symposium*. Vol. 8. Baltimore, MD. 2005, pp. 113–128.

[32] RIT. *CPTC Dataset*. `https://mirror.rit.edu/cptc/` [Accessed Nov. 2020]. 2018.

[33] Sebastian Roschke, Feng Cheng, and Christoph Meinel. "A new alert correlation algorithm based on attack graph". In: *Computational intelligence in security for information systems*. Springer, 2011, pp. 58–67.

[34] Anne Rozinat. "Process mining: conformance and extension". In: (2010).

[35] Reza Sadoddin and Ali Ghorbani. "Alert Correlation Survey: Framework and Techniques". In: *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services*. PST '06. Markham, Ontario, Canada: Association for Computing Machinery, 2006. ISBN: 1595936041. DOI: `10.1145/1501434.1501479`. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/1501434.1501479`.

[36] Saeed Salah, Gabriel Maciá-Fernández, and Jesús E. Díaz-Verdejo. "A model-based survey of alert correlation techniques". In: *Computer Networks* 57.5 (2013), pp. 1289–1317. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.comnet.2012.10.022`. URL: `https://www.sciencedirect.com/science/article/pii/S1389128612004124`.

[37] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012. ISBN: 9781285401065.

[38] T.F. Smith and M.S. Waterman. “Identification of common molecular subsequences”. In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(81)90087-5. URL: https://www.sciencedirect.com/science/article/pii/0022283681900875.

[39] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 1971, pp. 114–121. DOI: 10.1109/SWAT.1971.10.

[40] Sofia Tsoni et al. “Log Differencing using State Machines for Anomaly Detection”. MA thesis. Delft University of Technology, Aug. 2019. URL: http://resolver.tudelft.nl/uuid:b0b39832-c921-412c-b6f8-9ac4c52b57f6.

[41] Wil Van Der Aalst. “Process mining”. In: *Communications of the ACM* 55.8 (2012), pp. 76–83.

[42] S. Verwer and C. A. Hammerschmidt. “flexfringe: A Passive Automaton Learning Package”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2017, pp. 638–642. DOI: 10.1109/ICSME.2017.58. URL: https://doi-ieeecomputersociety-org.tudelft.idm.oclc.org/10.1109/ICSME.2017.58.

A

FlexFringe Configuration

Listing A.1: Default FlexFringe Configuration

```
[default]
heuristic-name = alergia
data-name = alergia_data
symbol_count = 2
state_count = 4
satdfabound = 2000
sinkson = 1
sinkcount = 5
extrapar = 0.05
largestblue = 1
finalred = 0
extend = 0
lowerbound = 3
finalprob = 1
markovian = 1
mcollector = -1
mergelocal = -1
mergesinks = 0
correction = 1.0
printblue = 1
```

B

Learned Models

In Figure B.1, the original learned model of the CPTC-2018 dataset can be seen. In Figure B.2, the best iterated model of the CPTC-2018 dataset can be seen. In Figures B.3a and B.3b, the models are shown without sink states, and all edges less than 10 occurrences are highlighted red.

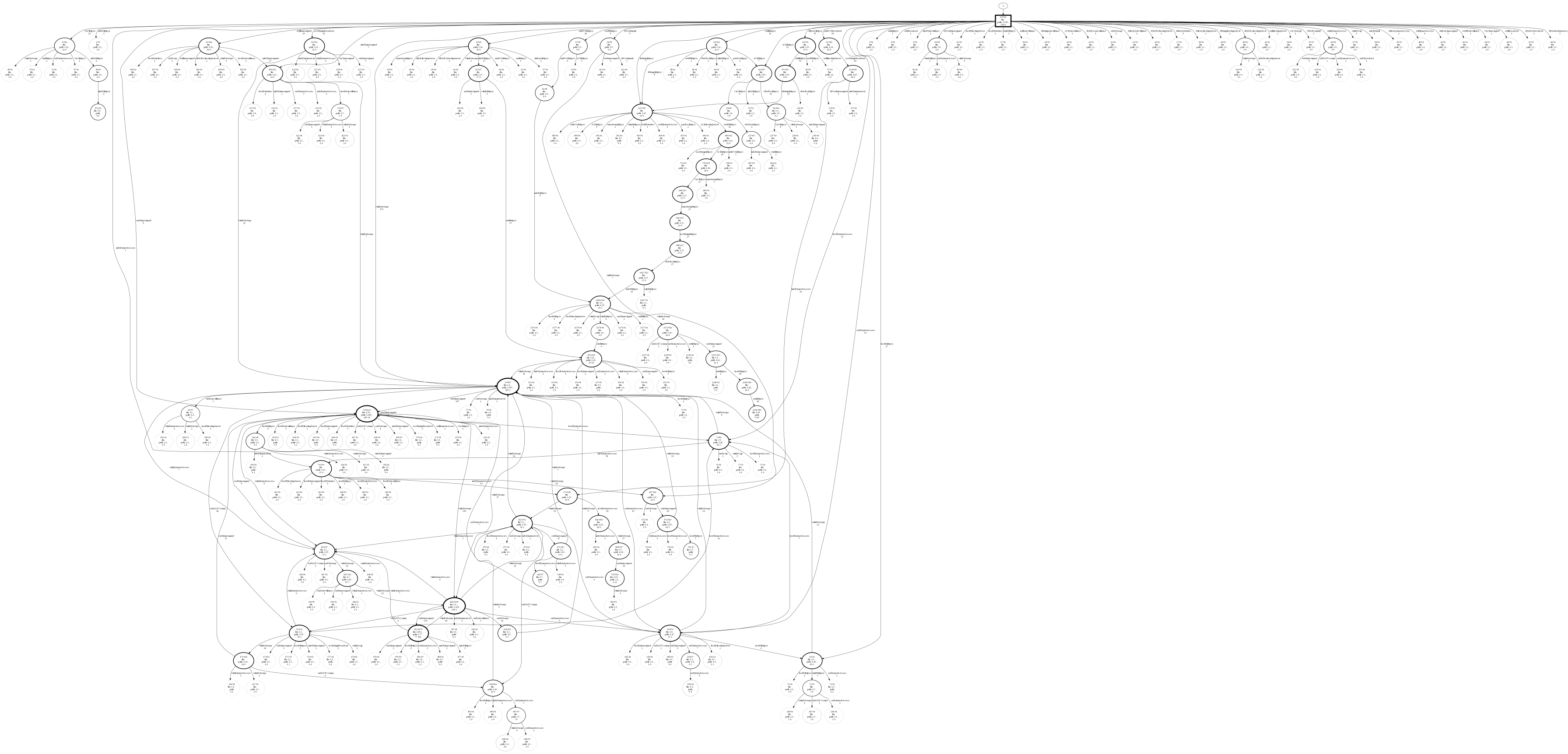


Figure B.1: The original model of the CPTC-2018 dataset, with an average normalised score of 0.66

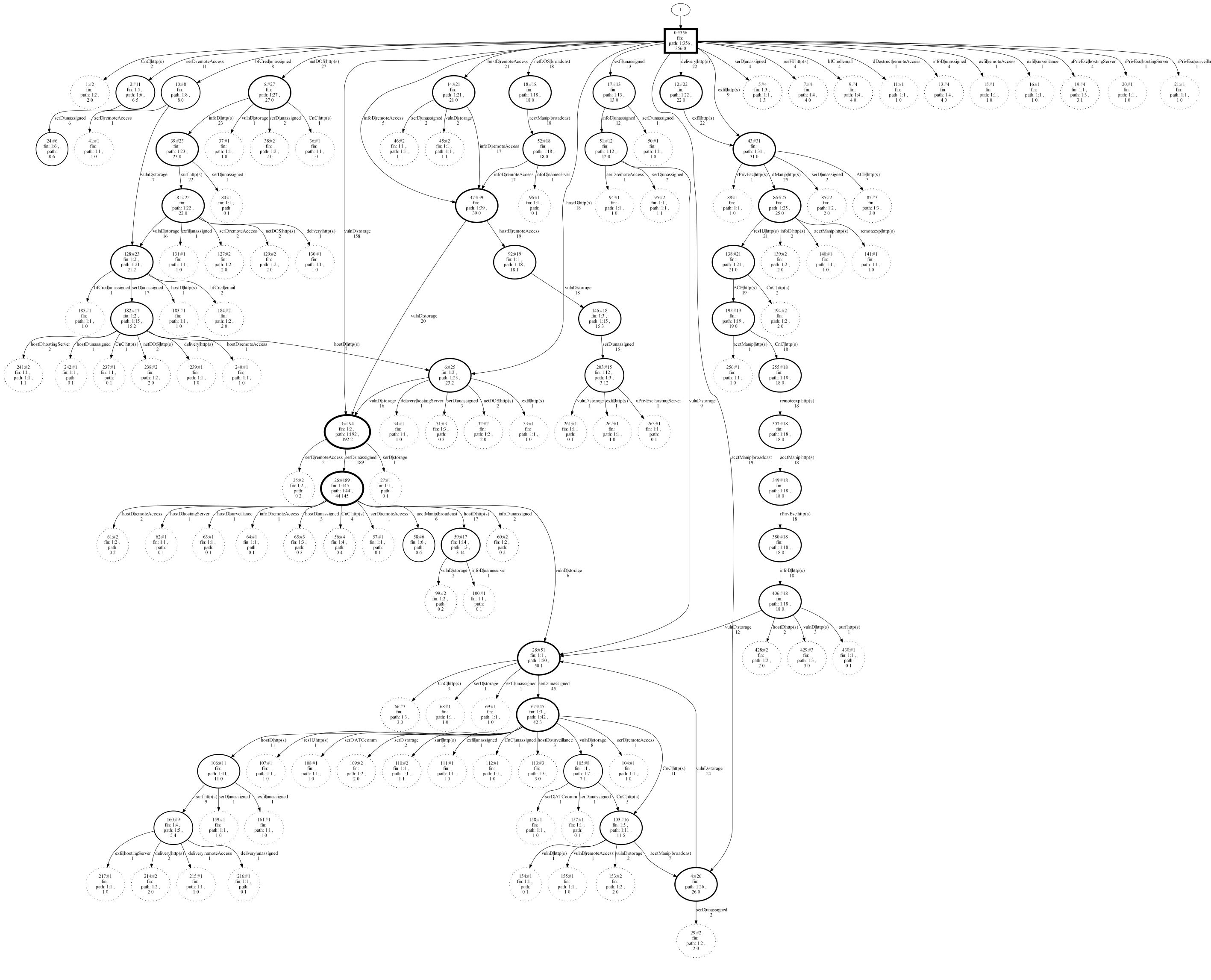
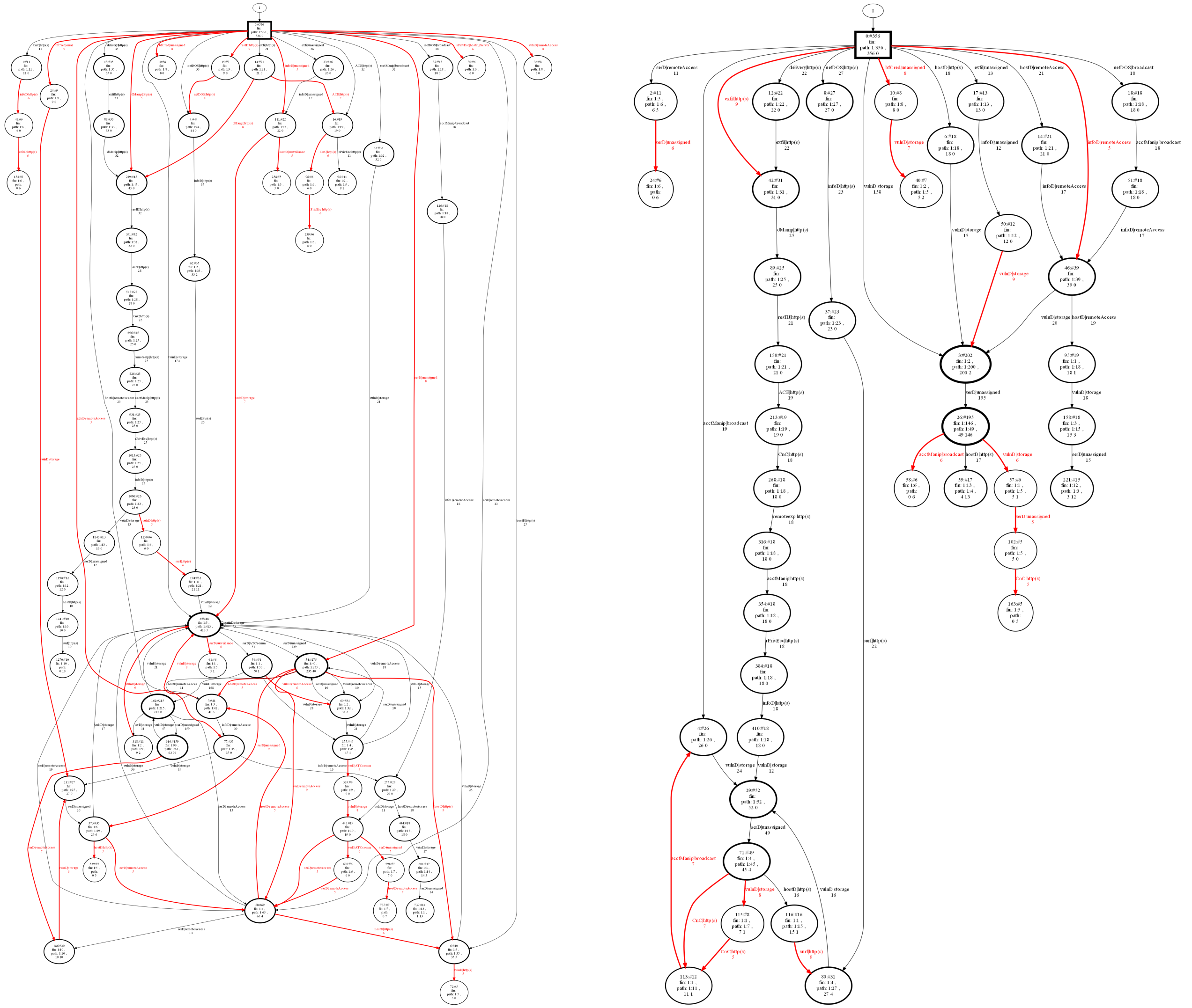


Figure B.2: The best iterated model of the CPTC-2018 dataset, with an average normalised score of 0.71. Heuristics **B** and **C** have been used.



(a) Original model.

(b) Iterated SAAT model

Figure B.3: Learned models of the CPTC-2018 dataset, without sink states. Edges with less than 10 occurrences are highlighted red. The iterated SAAT model has very few low-frequency edges, because SAAT can recover them.

C

Iterated SAAT K-Fold Cross Validation Results

Average Train Score	Average Test Score	Std. Dev. Test Score	Average Model Size
0.690	0.625	0.054	102.0
0.652	0.586	0.045	105.8
0.664	0.571	0.040	107.8
0.694	0.616	0.069	108.8
0.708	0.629	0.079	102.2
0.707	0.620	0.065	105.0
0.686	0.587	0.046	105.0
0.685	0.607	0.054	101.0
0.691	0.611	0.054	105.8
0.703	0.622	0.063	105.8

Table C.1: Aggregated results of 10 runs of iterative model learning of SAAT, using k-fold cross validation with $k = 5$.

D

HDFS Extracted Log Keys Sample

```
5 5 5 22 11 9 11 9 11 9 26 26 26
5 22 5 5 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
5 5 22 5 11 9 11 9 26 26 11 9 26 23 23 23 21 21 21
22 5 5 5 26 26 26 11 9 11 9 11 9 3 4 3 3 4 3 3 4 3 23 23 23 21 21 21
5 5 5 22 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
22 5 5 5 26 26 11 9 11 9 26 11 9 4 3 4 2 2 2 23 23 23 21 21 21
5 5 5 22 11 9 11 9 11 9 26 26 26 2 23 23 23 21 21 21
22 5 5 5 26 26 26 11 9 11 9 11 9 4 3 4 2 23 23 23 21 21 21
22 5 5 5 26 26 26 11 9 11 9 11 9 4 4 3 2 2 2 23 23 23 21 21 21
5 5 5 22 11 9 11 9 11 9 26 26 26 4 4 3 2 23 23 23 21 21 21
22 5 5 5 26 26 11 9 11 9 11 9 26 3 3 4 3 4 3 3 3 4 3 3 4 23 23 23 21 21 21
22 5 5 5 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
5 5 5 22 11 9 11 9 11 9 26 26 26
22 5 5 5 26 26 11 9 11 9 26 11 9 3 3 4 3 3 4 3 3 4 23 23 23 21 21 21
5 5 5 22 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
22 5 5 5 26 26 26 11 9 11 9 11 9 23 23 23 21 21 21
5 22 5 5 11 9 11 9 11 9 26 26 26
5 22 5 5 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
22 5 5 5 26 26 26 11 9 11 9 11 9 3 3 4 3 3 4 3 4 3 3 4 3 23 23 23 21 21 21
5 22 5 5 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
22 5 5 5 26 26 26 11 9 11 9 11 9 2 4 3 4 23 23 23 21 21 21
22 5 5 5 26 11 9 11 9 26 26 11 9 2 4 4 3 2 23 23 23 21 21 21
5 22 5 5 11 9 11 9 11 9 26 26 26 4 4 3 2 2 23 23 23 21 21 21
22 5 5 5 11 9 26 26 26 11 9 11 9 23 23 23 21 21 21
5 5 22 5 11 9 11 9 11 9 26 26 26
5 22 5 5 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
22 5 5 5 26 26 26 11 9 11 9 11 9 4 4 3 2 23 23 23 21 21 21
5 5 5 22 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
22 5 5 5 26 11 9 11 9 26 26 11 9 2 4 3 4 23 23 23 21 21 21
5 5 22 5 11 9 11 9 11 9 26 26 26 23 23 23 21 21 21
22 5 5 5 26 26 26 11 9 11 9 11 9 23 23 23 21 21 21
```

E

Learned model of the HDFS dataset

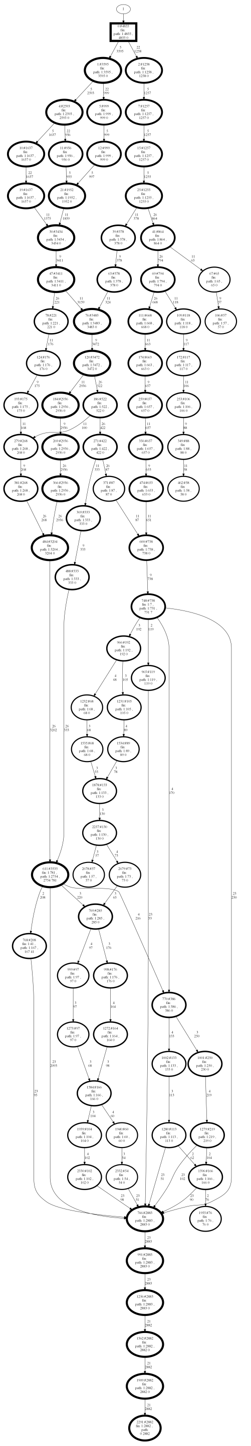

1
0#4855 fin: path: 1:4855 , 4855 0
5 3595
22 1258
1#3595 fin: path: 1:3595 , 3595 0
2#1258 fin: path: 1:1258 , 1258 0
5 2593
22 999
5 1257
4#2593 fin: path: 1:2593 , 2593 0
5#999 fin: path: 1:999 , 999 0
7#1257 fin: path: 1:1257 , 1257 0
5 1637
22 956
5 999
5 1257
10#1637 fin: path: 1:1637 , 1637 0
11#956 fin: path: 1:956 , 956 0
12#999 fin: path: 1:999 , 999 0
15#1257 fin: path: 1:1257 , 1257 0
22 1637
5 955
5 997
5 1255
19#1637 fin: path: 1:1637 , 1637 0
21#1952 fin: path: 1:1952 , 1952 0
25#1255 fin: path: 1:1255 , 1255 0
11 1575
11 1859
11 378
26 864
30#3434 fin: path: 1:3434 , 3434 0
39#378 fin: path: 1:378 , 378 0
41#864 fin: path: 1:864 , 864 0
9 3411
9 378
26 794
11 65
47#3411 fin: path: 1:3411 , 3411 0
63#378 fin: path: 1:378 , 378 0
69#794 fin: path: 1:794 , 794 0
67#65 fin: path: 1:65 , 65 0
26 221
11 3159
11 324
26 668
11 118
9 57
78#221 fin: path: 1:221 , 221 0
76#3483 fin: path: 1:3483 , 3483 0
111#668 fin: path: 1:668 , 668 0
109#118 fin: path: 1:118 , 118 0
106#57 fin: path: 1:57 , 57 0
11 176
9 3472
11 663
9 117
124#176 fin: path: 1:176 , 176 0
120#3472 fin: path: 1:3472 , 3472 0
174#663 fin: path: 1:663 , 663 0
172#117 fin: path: 1:117 , 117 0
9 175
11 2936
26 522
9 657
11 106
193#175 fin: path: 1:175 , 175 0
184#2936 fin: path: 1:2936 , 2936 0
186#522 fin: path: 1:522 , 522 0
259#657 fin: path: 1:657 , 657 0
253#106 fin: path: 1:106 , 106 0
11 168
9 2936
11 100
26 422
11 657
9 88
279#268 fin: path: 1:268 , 268 0
269#2936 fin: path: 1:2936 , 2936 0
271#422 fin: path: 1:422 , 422 0
356#657 fin: path: 1:657 , 657 0
349#88 fin: path: 1:88 , 88 0
9 268
26 2936
11 333
26 87
9 653
11 58
381#268 fin: path: 1:268 , 268 0
366#2936 fin: path: 1:2936 , 2936 0
371#87 fin: path: 1:87 , 87 0
474#653 fin: path: 1:653 , 653 0
462#58 fin: path: 1:58 , 58 0
369#333 fin: path: 1:333 , 333 0
26 268
26 2936
11 87
11 651
484#3204 fin: path: 1:3204 , 3204 0
600#738 fin: path: 1:738 , 738 0
9 333
9 738
488#333 fin: path: 1:333 , 333 0
748#738 fin: 1:7 , path: 1:731 , 731 7
3 192
2 119
966#192 fin: path: 1:192 , 192 0
965#119 fin: path: 1:119 , 119 0
4 68
3 105
1232#68 fin: path: 1:68 , 68 0
1231#105 fin: path: 1:105 , 105 0
26 3202
26 333
3 68
4 89
1535#68 fin: path: 1:68 , 68 0
1534#89 fin: path: 1:89 , 89 0
3 55
3 78
1878#133 fin: path: 1:133 , 133 0
4 170
3 130
2257#130 fin: path: 1:130 , 130 0
3 57
4 73
611#3535 fin: 1:781 , path: 1:2754 , 2754 781
2678#57 fin: path: 1:57 , 57 0
2679#73 fin: path: 1:73 , 73 0
2 208
3 220
3 65
23 250
769#285 fin: path: 1:285 , 285 0
4 216
23 53
768#208 fin: 1:41 , path: 1:167 , 167 41
4 97
3 176
999#97 fin: path: 1:97 , 97 0
998#176 fin: path: 1:176 , 176 0
3 97
4 164
770#386 fin: path: 1:386 , 386 0
1273#97 fin: path: 1:97 , 97 0
1272#164 fin: path: 1:164 , 164 0
4 133
3 250
23 2093
3 68
3 98
1002#133 fin: path: 1:133 , 133 0
1001#250 fin: path: 1:250 , 250 0
23 95
1584#166 fin: path: 1:166 , 166 0
3 113
4 219
3 104
4 60
1939#104 fin: path: 1:104 , 104 0
1940#60 fin: path: 1:60 , 60 0
1280#113 fin: path: 1:113 , 113 0
1279#219 fin: path: 1:219 , 219 0
4 102
3 54
2 62
2 104
2330#102 fin: path: 1:102 , 102 0
2332#54 fin: path: 1:54 , 54 0
23 51
23 102
1596#166 fin: path: 1:166 , 166 0
23 98
23 51
23 90
2 76
766#2883 fin: path: 1:2883 , 2883 0
1955#76 fin: path: 1:76 , 76 0
23 2883
991#2883 fin: path: 1:2883 , 2883 0
23 2883
1256#2883 fin: path: 1:2883 , 2883 0
21 2882
1562#2882 fin: path: 1:2882 , 2882 0
21 2882
1909#2882 fin: path: 1:2882 , 2882 0
21 2882
2291#2882 fin: 1:2882 , path: 0 2882

Figure E.1: Learned mdoelf of the HDFS dataset. It consists of 84 nodes, and has an average normalised score of 0.966.

F

Anomaly Detection Results

	TP	FP
not in ending state	8062	180
symbol not in alphabet	8760	1893
score threshold	0	51
failed	16	57

Table F.1: Counts of the anomaly conditions, for both TP and FP, when the order of the conditions is swapped. The conditions are checked in order from top to bottom.