Mining Attack Strategy

Using Process Mining to extract attacker strategy from IDS alerts

Geert Habben Jansen

TU Delft

Mining Attack Strategy

Using Process Mining to extract attacker strategy from IDS alerts

by

Geert Habben Jansen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 1, 2021 at 10:00.

Thesis committee:	Dr. ir. S. E. Verwer,	TU Delft, supervisor
	Prof. dr. ir. M. J. T. Reinders,	TU Delft
	Ir. A. Nadeem,	TU Delft

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

TU Delft

Abstract

Ever since the invention of the Internet, more and more computers are connected throughout the world. Though this has brought numerous new inventions used every day, like social media, e-commerce, and video conferencing, it also opens up new opportunities for cyber criminals. As the intrusion detection systems used to identify malicious behavior in a computer network can generate large amounts of alerts, methods have been developed to aid security analysts in gaining insights into what is happening on the network.

Of course, there is always room to improve these methods, which is the topic of this thesis. Currently, one of the state-of-the-art methods uses state machines to model the alert sequences. State machines are a good fit as they can extract the context of different alerts, but they cannot extract information like parallelism between different alerts. That is where field process mining comes in, with process mining algorithms being able to extract parallelism from sequential data. In this thesis, state-of-the-art algorithms from process mining are evaluated for modeling alert datasets from intrusion detection systems with the aim of improving the current methods. As a comparison, different methods for learning state machines also tested for the same data.

The results of the evaluation and comparison show that the state machines perform better in modeling the alert datasets with respect to explaining the data. On the other hand, thee process mining algorithms were not able to construct sound models for the datasets, and a fourth mining algorithm gave false implications about the data.

Furthermore, the possibility of combining state machines with process mining was also tested, with the idea that the combination can use the state machines to extract context and the process miner to extract parallelism. This method did not yield any improvements for the alert datasets tested, but that does not mean it is not viable in other cases.

Preface

Long story short, writing a thesis is hard. Ever since I started working on my thesis back in September, I was already dreading the days I would undoubtedly be spending writing this thesis. Well, I was wrong: apparently, it takes weeks, if not months, to write a complete thesis. However, I am willing to bet that once I got some time to rest, I will be looking back at the entire process with good memories.

First of all, I want to thank my supervisor, Sicco Verwer, for the ideas and feedback he gave me during my thesis. Second, I also want to thank Azqa Nadeem for introducing me to the topic of this thesis and for all the feedback you gave on my work throughout the process. I wouldn't have been able to achieve this without the help and guidance of you two.

Finally, I also want to thank my friends and family. I am sure that at some point, some of you got tired of me complaining every time an experiment crashed. Thank you for your support at those times and helping me get through this all. It wouldn't have been the same without you.

Geert Habben Jansen
Delft, June 2021

Contents

1 Introduction

Ever since the invention of the Internet, more and more computers are connected throughout the world. This connectivity has brought numerous new inventions used every day, like social media, e-commerce, and video conferencing. However, as the Internet opens up new opportunities for people, businesses, and governments, it also provides new possibilities for criminals. One does not have to search the news for long to find a story about denial-of-service attacks, ransomware, and data theft.

To combat cybercrime and detect malicious behavior, new cyber defense methods have been developed. One of these methods for detecting malicious behavior is the intrusion detection system, or IDS for short. When operating on a computer network, the IDS scans the traffic for anomalous or otherwise suspicious traffic and raises alerts when something suspicious is identified.

However, there is one big issue related to these alerts generated by an IDS: there can easily be too many of them. One example is the IDS placed between the Internet and the network of the University of Maryland, a network consisting of around 40,000 computers. On November 6^{th} 2012, this IDS produced a total of 26.873.302 alerts [10]. Even after applying filtering on the dataset, the researchers were still left with a set of 329.264 events for that one day.

As such volumes of alerts are too large for security analysts, several methods have been developed to extract attacker strategy from the data in the form of attack graphs. These attack graphs help security analysts by visualizing the paths taken by the attackers in order to reach certain objectives like performing a denial-of-service attack on the webserver. One of the most recent works towards constructing attack graphs comes from Nadeem et al.[31]. This new method differs from the existing works by only requiring the IDS alerts to construct the attack graphs. In contrast, previous methods require external information like network topology or the known vulnerabilities of the software running on the hosts in the network. Working with such additional information provides new challenges like the rapid changes in network topology or software, not to mention the issue of undiscovered or unpublished vulnerabilities. Therefore, a method relying solely on the IDS alerts provides a more flexible solution that requires less maintenance.

The new method uses a three-step process: alert aggregation, model construction, and attack graph generation. This thesis builds upon this method, more specifically on the model construction step. Currently, a suffix-based automaton is used to model the alert sequences. In this thesis, the option to use methods from process mining to replace or enhance the current models is explored. The reasoning behind this is that the methods from process mining can model concurrency in the underlying dataset more comprehensively compared to the state machines currently used.

1.1. Research questions

The objective of this thesis is to evaluate the performance of methods from process mining for modeling IDS alerts. Following this goal, the main question to be answered is as follows:

> To which extend can process mining be used to improve the models for IDS alert datasets?

As this question is too large to be answered directly, it is broken down into three sub-questions.

> RQ1: How well can state-of-the-art process mining techniques model the IDS alert datasets?

The first step is to determine how well the current process mining algorithms can model the datasets used. To quantify this, an evaluation setup is needed to measure the quality of the produced models for the datasets. With this evaluation setup, the most effective known process mining algorithms can be evaluated.

> RQ2: How well can state-of-the-art state machine learning techniques model the IDS alert datasets?

Following the main question, we want to know whether the process mining techniques offer an improvement over the techniques currently used. In order to determine this, we have to know how well the current techniques perform.

> RQ3: How much improvement can be gained by combining process mining with state machine learning?

Replacing the current modeling techniques is not the only option for improving the current models as the process mining techniques can also be applied to enhance the state machines currently used. The strength of process mining is that it is able to extract parallelism from a dataset, but one of the drawbacks is that the process mining algorithms are not effective in determining the context of different events. This is where state machines come in as these two factors are flipped with state machine inference: the models aren't effective in modeling parallelism but are able to extract the context from the dataset. By combining the two methods, it could be possible to end up with a modeling approach that takes both context and parallelism into account.

1.2. Contributions

1. First and foremost, this thesis gives an extensive evaluation of state-of-the-art process mining algorithms regarding their performance in modeling IDS alert sequences. These results can help with further evaluation of process mining algorithms for constructing attack graphs.

2. Second, this work outlines the differences between state machines and process models from the perspective of process mining. Such a comparison can help with future research, which has to choose between using either of the two methods.

3. Finally, this work provides a setup for using state machines to identify duplicate events in an event log, an issue with which most process mining algorithms cannot deal. Even though this thesis only evaluates this method on one dataset, the method can be used as a setup for more extensive research towards feasibility and effectiveness.

4. Finally, this work provides a setup to combine state machines with process mining. By combining the two methods, the resulting method might be able to combine the context modeling from state machines with the parallelism extraction from process mining. Even though this thesis only evaluates this method on one dataset, the method can be used as a setup for more extensive research towards feasibility and effectiveness.

1.3. Outline

This thesis starts with an overview of the background knowledge and concepts used throughout this thesis in Chapter 2. After that, Chapter 3 gives an outline of the evaluation experiments, which form the basis of this thesis. Building on the experiment setup, Chapters 4 and 5 contain the experimental results of the methods from process mining and state machine inference, respectively. Then, Chapter 6 briefly goes into the effectiveness of combining state machines with process mining. Finally, Chapter 7 finishes with the limitations, conclusions and future work.

2

Background

2.1. State Machines

This section starts with a formal definition of state machines. Following this is a short description of the most important state machine inference algorithms. Finally, the flexfringe tool for constructing state machines is introduced.

2.1.1. Definition

The most basic version of a state machine is a deterministic finite automata, or DFA. Following the notation of [40], a DFA M is represented by the five-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states
- Σ is the set of symbols, or alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
- q_0 is the start state
- $F \subseteq Q$ is the set of accepting states, also referred to as final states

The transition function δ is often represented as a subset of $Q \times \Sigma \times Q$, where the triple (q, s, q') is equivalent to the function-based notation $\delta(q, s) \rightarrow s'$.

For a string $S = s_1, s_2, \ldots, s_n, s_i \in \Sigma$ we can compute the corresponding state sequence $q_0, q_1, \ldots, q_n$ as $q_i = \delta(q_{i-1}, s_i), 1 \leq i \leq n$ and $q_n \in F$. The sequence starts with q_0, the initial state of the DFA, and records all states encountered through applying the transition function for each symbol. If $q_n \in F$, the DFA accepts the string S, else the DFA rejects S. The language of M is defined as the set of all strings accepted by M and is denoted as $L(M) = \{w \in \Sigma^* | M \text{ accepts } w\}$. One special case occurs when $n = 0$ and the string is empty, denoted as ϵ. For the empty string, the state sequence is q_0.

Figure 2.1 shows the graphical representation of a DFA accepting all strings which contain the substring aab. The set of states is represented by the nodes, and the alphabet is shown as labels on the edges. For this DFA, the initial state is q_0, as denoted by the incoming arrow, and the only accepting state is q_3, as denoted by the double edge. An entry for the transition function $\delta(q, s) : q'$ is denoted with a directed edge from state q to state q', labelled with symbol s.

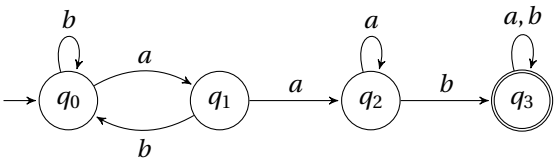


Figure 2.1: Example of a deterministic finite automaton, accepting all strings with the substring aab

It is not uncommon that a state q has no outgoing transition for some symbol $s \in \Sigma$. In these cases, the DFA will always reject the input when such a transition would be taken. This can be modeled through some rejecting state with self-loops for all symbols in Σ. For the sake of readability, this rejecting state and all its incoming transitions are usually not shown.

PDFA

An evolution of the DFA is the probabilistic deterministic finite automata, or PDFA for short. Compared to the regular DFA, the PDFA also defines some probability function $P : \delta \rightarrow [0,1]$. This probability function assigns a probability to each transition to indicate which paths in the automaton are more likely compared to others. In addition, the probability function also gives a probability that the execution will end in a given state. Furthermore, this probability function is defined such that for each state, the sum of the probabilities for all outgoing edges and the probability of the trace terminating in that state equals one.

With this probability function, we can assign some probability to all strings accepted by the PDFA. The probability of a string is the product of the probabilities of all traversed transitions and the probability of the PDFA terminating in the final reached state.

Suffix-based models

Regular state machines are used to define behavior given some prefix, but we wish to reverse this logic in some cases. For this, suffix-based automata can be used. These automata operate just like their regular prefix-based counterparts, but instead of a state describing some prefix, the states represent some suffix. A suffix-based model can be obtained by using the regular DFA learning algorithms but reversing the strings used for training.

The suffix-based variants of a DFA or PDFA are referred to as S-DFA or S-PDFA, respectively.

2.1.2. State Machine Inference

The goal of state machine inference is to construct a minimal state machine which is consistent with a set of training samples. A training set S consists of a set of strings accepted by the underlying state machine. In some cases, a set of strings which are rejected by the underlying state machine is also available, in which case the training data is referred to as S^+ and S^- for the strings accepted and rejected respectively.

Blue-fringe and Evidence-driven State Merging

One of the basic methods for constructing a DFA is through Evidence-Driven State Merging (EDSM) [17]. This method also introduced the blue-fringe algorithm, which is used as the basis for multiple other methods of constructing state machines.

As the first step, an augmented prefix tree acceptor (APTA) is constructed from the input strings. The APTA is a tree-shaped state machine that exactly fits the training data. Furthermore, a state q is an accepting state if some positive string from the training set ends in q. In the case negative samples are also available, the states in which a negative string ends are marked as rejecting. The APTA perfectly fits the logs but does not provide any generalization and hence won't accept any string not in the training set. In order to transform the APTA to a more general DFA, states are merged through an iterative process.

The blue-fringe algorithm limits the number of possible merges evaluated at each step, which in turn reduces the amount of time needed for the merging process to finish. With this method, states are colored red, blue, or are not colored at all. Red states indicate states in the DFA which have successfully been identified. Blue states are the states not colored red but reachable from a red state through a single transition. These blue states are the states which will be evaluated for merging. States not colored red or blue remain uncolored. Using this construction, the red states can be seen as a partially defined DFA, separated from the rest of the training data by a fringe of blue states. These blue states are the roots of trees of uncolored states.

Initially, the root state is colored red, and the states directly reachable from the root are colored blue. Then, the merging process tires all possible merges between a red and a blue state, computing a merge score for each possible merge. If there are blue states which cannot be merged, the state closest to the root is colored red. Else, the merge with the highest score is performed, and the resulting merged state is colored red. Then, the coloring of the states is updated, and the merging process starts over again until all states are red.

Merging a blue state q_b into a red state q_r is a straight-forward procedure: all incoming transitions for q_b are re-directed to q_r and the outgoing transitions from q_b are added to q_r. If an accepting state is merged with a rejecting state, the merge is invalid and cannot be performed. Otherwise, the resulting state is accepting or rejecting if either q_r or q_b was accepting or rejecting, respectively. If both q_r and q_b were neutral, the resulting state is also neutral. However, the merge can also introduce a non-deterministic automaton if in the DFA before the merge, there exists some symbol $a \in \Sigma$ such that $\delta(q_r, a) = q'_r$, $\delta(q_b, a) = q'_b$ and $q'_r \neq q'_b$. For all these cases, the states q'_r and q'_b are also merged until the automaton is deterministic again. This process is called determinization. As each blue node is the root of an uncolored sub-tree in the APTA, the merging and determinization process can be implemented relatively efficiently.

Determining the best merge is done through a scoring metric. The most basic metric is the evidence-based metric, which just counts the number of state merges between two accepting or two rejecting states during the merge[17], but alternatives can easily be defined. This scoring metric also makes the blue-fringe algorithm highly flexible as custom metrics can easily be defined.

Alergia

The Alergia algorithm[9] provides an alternative merging function for the blue-fringe algorithm based on statistics. Whereas the evidence-driven state merging considers a merge between two states valid when there are no conflicts in the outgoing transitions, the merging function used by Alergia also requires that the relative frequencies of the outgoing transitions are similar.

In order to deal with the fluctuations in the frequencies, the equivalence of frequencies is defined by the Hoeffding bound[14]. Let n_i be the frequency of state q_i and f_i^a be the frequency of the transition labelled a leaving state q_i. The value α denotes the confidence interval. The frequency of the outgoing transition labelled a for states q_i and q_j is considered equivalent based on the following condition:

$$\left| \frac{f_i^a}{n_i} - \frac{f_j^a}{n_j} \right| < \sqrt{\frac{1}{2} \log \frac{2}{\alpha}} \left(\frac{1}{\sqrt{n_i}} + \frac{1}{\sqrt{n_j}} \right)$$

When the condition holds, we know that the frequencies are equivalent with some probability larger than $(1-\alpha)$. Alergia considers two states equivalent if the condition holds for all possible symbols $a \in \Sigma$, as well as for the probability of a string terminating in the states (denoted with f_i instead of f_i^a). When two equivalent states are merged, the determinization process uses the same conditions to perform the consecutive merges.

2.1.3. Flexfringe

The flexfringe[43] tool provides an open-source implementation[1] for the blue-fringe algorithm and many different equivalence checking methods. In addition, the tool provides additional configuration options to fine-tune the state merging process. In the context of this thesis, two features of flexfringe are important: the `Markovian` setting and `sink` states.

First is the `Markovian` setting. When enabled, this setting ensures that for the resulting model, each incoming transition for a state has the same label. Through this constraint, the resulting model will always be a Markovian graph. One thing to note is that the incoming label for a state is not unique to the state: given some label a, multiple states may exist which only feature incoming transitions labeled with a. Furthermore, the Markovian property also enables models to be constructed based on n-grams over the data. For example, given some input event sequence $\langle$`a b c d`$\rangle$, a 2-gram (also called bigram) uses the sequences of two events over the original data: $\langle$`(a b) (b c) (c d)`$\rangle$.

The second feature of flexfringe is the `sink` states. Sinks are states which occur infrequently in the dataset. In order for a state to be marked as a sink, its frequency from the training data must be below some set threshold (set by the `sinkcount` setting). The idea of marking a state as a sink is that due to the infrequent nature of the state (and by extension also all its outgoing transitions), the statistical tests used for merging might not yield reliable results. Therefore, by marking states as sinks, a different merging strategy can be applied.

[1] https://bitbucket.org/chrshmmmr/dfasat

2.2. Process Models

The state machines introduced in the previous section aren't able to concisely model concurrency. For this, two other model types are used: Petri nets and Business Process Modeling Notation.

2.2.1. Petri nets

First introduced in [34], Petri nets are the most commonly used models for representing processes. The reason for this is that a Petri net (or PN for short) is able to model the concept of concurrency, which is a key factor in constructing process models.

Following the notations used by [1], a Petri is denoted by the triple (P, T, F) where:

- P is a set of places
- T is a set of transitions, such that $P \cap T = \emptyset$
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, or directed edges from a place to a transition or a transition to a place

The notation $\bullet t$ is used for the set of places with an arc to t, and $t\bullet$ is used for the set of places with an arc from t. Similarly, $\bullet p$ and $p\bullet$ represent the set of transitions with an arc to or from p respectively.

Each place in the PN can hold any number of tokens, which are used to determine the state of the net. Such a state is referred to as a marking. Given a marking M, the notation $M(p)$ denotes the number of tokens in place p. When all places in $\bullet t$ hold a token, transition t becomes enabled. An enabled transition can fire, consuming one token from all places in $\bullet t$, and producing one token for all places in $t\bullet$. This firing of t given a marking M is denoted as $M \xrightarrow{t} M'$. Similarly, $M \xrightarrow{*} M'$ indicates there is some sequence of transitions $t_1, \ldots t_n$ such that $M \xrightarrow{t_1} M_1, \ldots, M_{n-1} \xrightarrow{t_n} M'$, i.e. starting at marking M, there is some firing sequence which eventually yields marking M'.

Figure 2.2 shows an example of a Petri net. Here, places are shown as circles, transitions as squares, and tokens as black dots. Given the marking shown, only transition a is enabled, and firing a will give the marking $(c1 + c2)$. At this point, transitions b and c are enabled as $c1$ holds a token. As b and c cannot both fire, the construct shows a block of exclusive choice. At the same time, transition d is also enabled as there is a token in $c2$. Executing d is independent of transitions b and c, so the two branches can be executed in parallel. Merging the two parallel branches is done by transition e, which requires both d and either b or c to have fired. A transition marked with τ (alternatively shown as a black square) is a silent transition, i.e. it can be fired without showing up in the resulting trace. With this functionality, transition f can be skipped when $c5$ holds a token. By keeping track of the sequence in which transitions fire, a trace can be constructed. For the net shown, there are eight possible traces: `abde, abdef, acde, acdef, adbe, adbef, adce, adcef`. This set of traces is said to fit the Petri net.

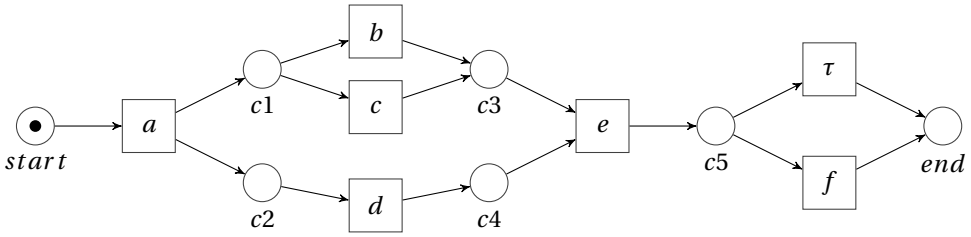


Figure 2.2: Example of a Petri net. Places are shown as circles, transitions as squares, and the token as a black dot.

A special type of Petri net is the flower model, as shown in Figure 2.3. This model accepts all traces containing the events a, b, c and d.

Workflow nets

A workflow net, often abbreviated as WF-net, is a special variant of Petri nets. For a Petri net PN to be considered a WF-net, PN must contain a special source place p_i and sink place p_o such that both $\bullet p_i$ and $p_o\bullet$ are empty. Using these special places, the initial marking (denoted as i) only has one token in p_i, and the final marking (denoted as o) has a token in p_o. Besides, by adding a transition $t*$ to PN where $\bullet t = p_o$ and $t\bullet = p_i$, PN becomes fully connected: there exists a path from any place or transition in the net to any other place or

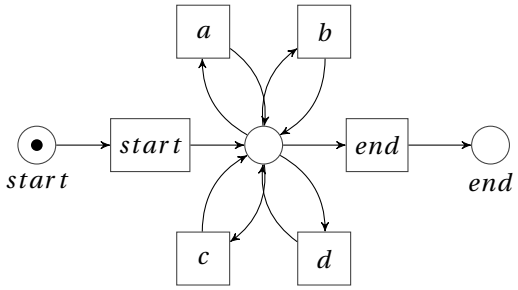


Figure 2.3: Example of a flower Petri net for transitions a, b, c and d

transition. The Petri nets in Figure 2.2 and Figure 2.3 are also WF-nets where p_i is the place labelled $start$, and p_o is the place labelled end.

Workflow nets are often used in the area of modeling processes as both the initial marking and final marking are clearly identifiable. Besides, the requirement that each place lies on a path between p_i and p_o makes it easier to reason about the correctness of the model.

2.2.2. Business Process Modeling Notation

A more high-level representation of a process model is achieved using the Business Process Modeling Notation[32], or BPMN for short. Compared to Petri nets, BPMN gives a more graphical representation for a process model while still retaining the same descriptive power.

The core functionality of BPMN is represented by the gateways. A parallel split gateway (represented with $+$) continues execution in all branches, and similarly, a parallel join gateway requires all incoming branches to have finished execution before continuing. For the exclusive-choice split gateways (represented with $\times$), only one branch is executed, and only one branch needs to finish before continuing at an exclusive-choice join gateway. Finally, the OR split gateway (represented with $\bigcirc$), one or more branches are executed, and the OR join gateway requires all incoming branches which have started execution to finish before continuing. More gateways are defined in the BPMN specification[2], but these are not relevant in the context of this thesis.

Figure 2.4 shows the BPMN model for the same process as modelled in Figure 2.2. With BPMN, squares represent events, and diamonds represent split and join gateways. The process starts at the start node, represented by the thin circle, and event a is executed. Then, the process reaches a parallel split gateway, splitting the process into two parallel branches. The first branch goes through an exclusive-choice split to choose between either b or c (but not both) and joins again at the second exclusive-choice gateway. The second branch of the parallel split executes only event d. After joining at the second parallel gateway, event e is executed, and a second exclusive-choice split/join block facilitates the optional execution of f. Finally, the process reaches the terminal node, represented by the thick circle.

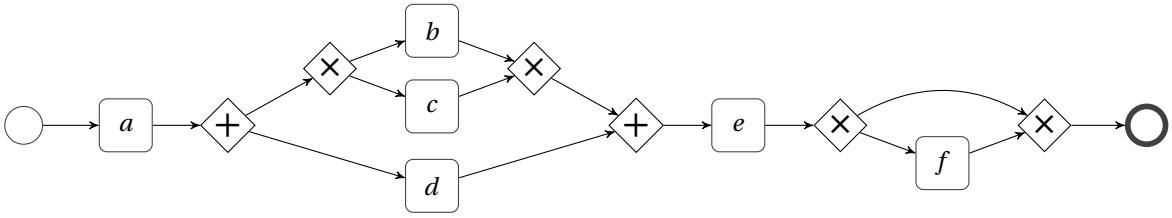


Figure 2.4: Example of a BPMN model

[2] https://www.omg.org/spec/BPMN/2.0/

2.3. Measuring process model quality

In the field of Process Mining, model quality is measured in four dimensions: fitness, precision, generalization and complexity [39]. Furthermore, a fifth constraint is also often required: model soundness. This section discusses the different measures proposed to evaluate these dimensions of process models and process discovery algorithms.

2.3.1. Soundness

The most important requirement for a discovered process model is for the model to be valid, or in formal terms: the model should be sound. In [1], Van der Aalst shows that a WF-net can be proven to be sound based on three conditions, and that these conditions can be checked in polynomial time.

Let $M \geq M'$ be true if each place in marking M holds at least as many tokens as in marking M', or formally $\forall p \in P, M(p) \geq M'(p)$. Using this notation, a WF-net is sound if and only if:

1. $\forall_M (i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o)$: for each marking reachable from the initial marking, the final marking can be reached.

2. $\forall_M (i \xrightarrow{*} M \wedge M \geq o) \Rightarrow (M = o)$: starting at the initial marking, the only time the output place holds a token is when it is the only place holding a token

3. $\forall_{t \in T} \exists_{M,M'}$ s.t. $i \xrightarrow{*} M \xrightarrow{t} M'$: starting at the initial marking, some firing sequence exists such that any transition t can be fired.

The first rule indicates that the WF-net cannot get stuck at any point during the execution. For the corresponding process, no valid execution can result in the process getting deadlocked or reaching a livelock. The second rule indicates that when the sink place is marked, and therefore the process terminated, the only token left in the net is in the sink. In terms of a real-world process, the tokens represent different concurrent branches of the process execution. Following this, the second rule states that the process can only end when all concurrent branches are completed. The third rule indicates that each transition in the WF-net is part of at least one valid execution trace. When comparing the transitions in the WF-net to events in a process, this rule indicates that all events must actually be part of the process.

As these properties are required for a usable process, soundness is a hard requirement for a model.

2.3.2. Fitness

Fitness describes the ability of a process model to explain an event log, similar to recall in other areas of machine learning. For all metrics proposed, the fitness score lies between zero and one, where a higher score means the model is better at explaining the data.

Completeness

The most basic computation for fitness is completeness [12]. With this measure, all traces in the evaluation set are replayed over the discovered model to evaluate if the trace fits the model. Then, completeness is the fraction of traces in the evaluation set perfectly fitting in the model. The main drawback of this method is that it uses a binary classification for each trace, penalizing both minor and significant deviations equally. Hence, this measure can not be relied upon when factors like noise in the training data are expected.

Fitness

A more refined measure for fitness is introduces in [38], also referred to as just fitness. This method computes a score based on the number of tokens missing and remaining compared to the total number of tokens produced and consumed when replaying a trace in the Petri net.

For trace i, let p_i and c_i denote the number of tokens produced and consumed during replay, m_i and r_i denote the number of tokens missing and remaining, and n_i denote the number of occurrences of trace i in the event log. When replaying a trace on a WF-net, the firing of transition t consumes one token for each place in $\bullet t$ and produces one token for each place in $t\bullet$. If the trace requires transition t to be fired, m_i is incremented by one for each place $p \in \bullet t$ which does not hold any tokens. In addition, one token is produced in the source place before the replay starts, and one token is consumed from the sink place after the replay finishes. Finally, r_i is the number of tokens in the net. Note that each token counted as missing or remaining is also counted as produced or consumed respectively.

Using these values, the fitness over an event log with k traces is computed as:

$$f = \frac{1}{2}(1 - \frac{\sum_{i=1}^{k} n_i m_i}{\sum_{i=1}^{k} n_i c_i}) + \frac{1}{2}(1 - \frac{\sum_{i=1}^{k} n_i r_i}{\sum_{i=1}^{k} n_i p_i})$$

As $m_i \leq c_i$ (each missing token was also created), $r_i \leq p_i$ (each remaining token has been produced), $p_i > 0$ (the initial token was produced) and $c_i > 0$ (the final token is consumed), the fitness score f is both defined and $0 \leq f \leq 1$.

A drawback of this method is that all types of errors are penalized the same amount. Violating an arguably less critical synchronization transition is penalized the same amount as skipping a transition corresponding to an event occurring in the trace.

Alignment-based Fitness

Building further, the method used for measuring fitness is the alignment-based fitness proposed in [4]. This method allows for assigning a unique cost for skipping a transition that should happen according to the model but not the trace or inserting (firing) a transition that should happen according to the trace but cannot happen in the model.

By using the A* algorithm, the optimal transition sequence for a Petri net given a trace is computed. At each point in replaying the trace, three options are considered: moving only in the model (inserting a transition), moving only in the log (skipping a transition), or moving both the log and the model. Skipping or inserting a transition in the net is always an option, whereas moving both in the model and the net is only viable when the transition corresponding to the next event in the trace is enabled. When a transition is skipped or inserted, the cost associated with the execution path is increased. This cost for inserting or skipping a transition a is denoted by $k^i(a)$ and $k^s(a)$ respectively, where different transitions can be assigned any non-negative cost. By using the cost assigned to each execution path, the algorithm can find the alignment with the lowest cost between the log and the model.

Let A_s be the multiset of skipped activities for the optimal alignment and $A_s(a)$ be the number of times a occurs in A_s. Furthermore, T is the set of events in a trace and $E_i \subseteq T$ is the set of events inserted in the optimal path. $\alpha(e)$ denotes the transition corresponding to event e. Using these values, the alignment-based fitness for trace T is computed as:

$$f = 1 - \frac{\sum_{a \in A_s} A_s(a) \cdot k^s(a) + \sum_{e \in E_i} k^i(\alpha(e))}{\sum_{e \in T} k^i(\alpha(e))}$$

With this computation, the fitness is 1 if the trace aligns perfectly with the net and decreases as the trace and the net differ more. The bound in the denominator is based solely on the maximum number of insertions which could occur as the theoretical maximum number of skipped activities is infinite if the model contains a loop.

Through the possibility to define custom costs for skipping or inserting transitions, the alignment-based fitness is the most flexible fitness metric discussed. Similar to the fitness measure, alignment-based fitness can also deal with non-fitting traces during fitness evaluation. Furthermore, it enables the usage of alignment-based precision, as discussed later in this section. This metric is used to evaluate fitness in [7, 8, 19].

2.3.3. Precision

The complement to fitness is precision. Where fitness indicates how well the model can explain the traces, precision measures how close the fit between the two is. The precision score lies between zero and one, where a higher score indicates that the model allows less behavior unrelated to the evaluation set.

Behavioral Appropriateness

The simplest measure for precision is behavioral appropriateness as described in [38]. This measure uses the mean number of enabled transitions to measure how often a choice can be made in the model, and therefore the amount of variance allowed by the model. Let $|T_V|$ denote the number of visible transitions in the Petri net, x_i be the mean number of enabled transitions when replaying trace i (not counting invisible tasks), and

n_i denote the number of occurrences of trace i. Then, the behavioral appropriateness a_B is computed as:

$$a_B = \frac{\sum_{i=1}^{k} n_i(|T_V| - x_i)}{(|T_V| - 1) \cdot \sum_{i=1}^{k} n_i}$$

As long as there are multiple visible transitions (i.e., $|T_V| > 1$), the metric value ranges from 0 (all transitions are always enabled) to 1 (only one transition is enabled at a time). An issue does arise when a trace does not fit the model. In such cases, the precision is either not defined or is incorrectly skewed towards 1 as the lack of enabled transitions reduces the value x_i.

Advanced Behavioral Appropriateness

In the same paper, the authors also define advanced behavioral appropriateness[38], which uses a 'precedes' and 'follows' relation to compare the behavior of the model and the log. This relation is defined for all combinations of two events (including an implicit start and for the end of a trace) and can have one of three values: always, never, or sometimes. Given the events a and b, the 'follows' relation is 'always' if all traces follow the pattern $\langle \dots a \dots b \dots \rangle$, 'never' if this pattern is not present in any of the traces, or 'sometimes' if it is present in some (but not all) of the traces. The 'precedes' relation describes similar behavior but looks in the opposite direction.

When a relation is defined as 'always' or 'never,' there is a strong indication that there is no choice to be made between different execution paths, and there is no room for variation. Hence, the advanced behavioral appropriateness only looks at the relations with value 'sometimes.'

Let S_P^l and S_F^l denote the 'precedes' and 'follows' relation for the log, and S_P^m and S_F^m denote the relations for the model. Then, the advanced behavioral appropriateness a'_B is defined as:

$$a'_B = \frac{|S_F^l \cap S_F^m|}{2 \cdot |S_F^m|} + \frac{|S_P^l \cap S_P^m|}{2 \cdot |S_P^m|}$$

As long as the sets S_P^m and S_F^m are not empty, the value for a'_B is defined and between 0 and 1. In the case either of those sets is not empty, the model does not have any construct for choice and is therefore completely deterministic. As a result, computing the precision is not logical in these cases.

As computing the relations does not require replaying the log on the model, the metric can be computed for non-fitting traces. However, this method requires an exhaustive search of the state space of the model to construct the corresponding S_F and S_P relations. As this search is time-consuming for large models, the metric does not scale well.

ETC Precision

An alternative metric which does not require an exhaustive search of the state space of a Petri net is ETC precision, first defined in [30] and later refined in [29]. The main idea of this method is to simulate the log traces on a model and use the number of escaping edges as a metric for precision.

Computation starts with constructing the prefix automaton for the log evaluated, which is a tree-shaped state machine similar to the APTA described in Section 2.1.2. In this automaton, each state s represents a prefix of some trace in the log, and $s_\#$ denotes how often the prefix corresponding to s occurs in the log. This automaton is defined as the four-tuple (S, T, δ, s_{in}).

Then, for each state in the automaton, the corresponding marking for the Petri net is computed by replaying the traces over the net. For traces not fitting the model, the largest prefix still conforming to the net is used.

With the markings for each state computed, the automaton is enhanced with extended states and extended transitions. Given some state s with corresponding marking M, if there is some transition t such that t is enabled given marking M and $M \xrightarrow{t} M'$, an extended state s' and extended transition (s, t, s') are added. Here, state s' has the corresponding marking M' and an occurrence count of zero. The set of extended states is denoted as S' and the set of extended transitions as δ'. The enhanced automaton is therefore represented as $(S \cup S', T, \delta \cup \delta', s_{in})$.

The basic form of ETC precision is based on the number of times an extended transition could be performed instead of the transition according to the log. Let the event log be some set $\{\sum_1, \dots, \sum_n\}$, and s_j^i denote the state in the automaton corresponding to the prefix of the first j symbols in trace $\sum_i$. Furthermore, $A_T(s_j^i)$

is the set of all transitions leaving s_j^i and $E_E(s_j^i)$ is the set of enabled transitions from s_j^i (all non-extended transitions leaving the state). Then, the basic ETC precision etc_p is computed as:

$$etc_p = 1 - \frac{\sum_{i=1}^{n} \sum_{j=1}^{|\Sigma_i|+1} |E_E(s_j^i)|}{\sum_{i=1}^{n} \sum_{j=1}^{|\Sigma_i|+1} |A_T(s_j^i)|}$$

As there are never fewer transitions leaving some state as there are enabled transitions leaving the state, this metric returns a value between 0 and 1.

The basic ETC precision is, however, sensitive to longer traces. This problem is tackled with the updated metric. In order to do this, sets of escaping states E_S^γ and outer states E_S^γ are defined based on some noise threshold $\gamma \in [0,1]$. Using the enhanced automaton $(S^*, T, \delta^*, s_{in})$, state s' is an escaping state if it occurs less than γ times as often as its predecessor s, i.e. there is some transition (s,t,s') and $\gamma \cdot s_\# \geq s'_\#$. When state s' is marked as an escaping state, all states in the subtree rooted as s' are automatically marked as an outer state (and therefore, these states cannot also be escaping states).

Let $A_T(s)$ once again the be the set of all transition leaving state s and I be the set of all included states: $I = s \in S^* | s \notin E_S^\gamma \wedge s \notin O_S^\gamma$. Then, the updated ETC precision etc'_P is defined as:

$$etc'_P = 1 - \frac{\sum_{s \in I} |E_S^\gamma| \cdot s_\#}{\sum_{s \in I} |A_T(s)| \cdot s_\#}$$

As there cannot be more escaping states reachable from any state s as there are transitions leaving s, the computed value lies between 0 and 1.

Due to the usage of the noise threshold γ, the enhanced version does not consider infrequent subtrees for computing precision. As a result, the precision is not lowered for missed branches encountered with infrequent behavior.

The main benefit of ETC precision is that it is relatively cheap to compute time-wise. However, it has shortcomings when the traces used for computing precision do not fit the model.

Alignment-based Precision

Building on ETC precision, the Alignment-based precision metric[5] tries to tackle the issue of non-fitting traces not working with the computation.

Computing the alignment-based precision begins with computing the optimal alignments using the same method used for alignment-based fitness (see Section 2.3.2). Using the model part of the optimal alignments, the alignment automaton is constructed in a way similar to the alignment automaton for ETC precision. By using the optimal alignments for the traces instead of the traces themselves, the non-fitting traces are still fully included in the automaton.

With each state in the automaton once again mapped to the corresponding marking in the Petri net, the automaton is extended with extended states and extended transitions, based on transitions that are at some point enabled in the Petri net but are never fired according to the logs. Once again, this yields an automaton similar to the one constructed for ETC precision.

Let S be the set of states originally in the alignment automaton (similar to the set I for etc'_p), and $\omega(s)$ be the count of how often the prefix corresponding to state s occurs in the log. Furthermore, $a_v(s)$ is the set of all transitions leaving state s in the automaton and $e_x(s)$ is the set of transitions to some state $s' \in S$ leaving state s. Then, the alignment-based precision a_p is computed as:

$$a_p^1 = \frac{\sum_{s \in S} \omega(s) \cdot |e_x(s)|}{\sum_{s \in S} \omega(s) \cdot |a_v(s)|}$$

As $e_x(s)$ is a subset of $a_v(s)$, the computed metric yields a value between 0 and 1.

For an even further enhancement, the alignment-based precision metric a_p is also defined. This metric differs from a_p^1 in the construction of the alignment automaton by incorporating all optimal alignments for the traces instead of just an optimal alignment. By using all optimal alignments, any bias introduced by selecting some optimal alignment is eliminated.

Computing the precision score is done in a similar way, with the only difference being the computation of $\omega(s)$ taking into account the fact that one trace can have multiple optimal alignments. If some trace s has

n optimal alignments, the value of $\omega(s)$ is incremented with $\frac{1}{n}$ for any state s corresponding to any prefix of any alignment for s. Hence, if all traces have only one optimal alignment, $n = 1$ and a_p and a_p^1 are exactly the same. Experimentation from the original paper showed that a_p^1 gives a good approximation of a_p, and that the computation of a_p is significantly slower than computing a_p^1.

Compared to the other measures for precision, the alignment-based variant is by far the most expensive to compute time-wise. However, when alignment-based fitness is also used to evaluate the model, the alignments can be re-used to mitigate the poor computational complexity. Due to the similarities with alignment-based fitness, alignment-based precision can also deal with non-fitting traces making the method more robust. Alignment-based precision is used to evaluate precision in [7, 8, 19].

2.3.4. F-score

One of the key challenges for creating a good process model is finding the right balance between fitness and precision. In order to quantify such a balance, we can look at other fields of machine learning. Here, the F-score is widely used to balance the precision and recall of a classification algorithm.

Process Mining can be seen as a form of binary classification, where the goal is to classify data points a being part of some 'positive' class or the 'negative' class. The positive and negative classes are defined by a training set containing data points for which the class is known. If an unknown data point is correctly classified as part of the positive or negative class, it is said to be a true positive (TP) or true negative (TN). When a negative data point is classified as positive, or a positive data point is classified as negative, it is said to be a false positive (FP) or false negative (FN).

Precision P is the fraction of data points classified as positive actually being positive. Recall R is the fraction of positive data points actually being classified as positive.

$$P = \frac{TP}{TP + FP} \qquad R = \frac{TP}{TP + FN}$$

Both precision and recall produce a result between 0 and 1. Using these values, the f-score is computed as the harmonic mean of precision and recall:

$$2 \cdot \frac{precision \cdot recall}{precision + recall}$$

For process mining, the training set contains only positive samples in the form of the event log, and the positive class are all traces produced by the underlying process. Classification in done in the form of constructing a process model where a trace is classified as positive if the model explains the trace. Fitness is similar to recall as it measures how many known positive data points (traces in the log) are actually valid in the model. Precision is similar in both cases as it measures how many data points classified as positive are actually (known) positive.

By using the F-measure, we can quantify how well a model balances fitness and precision. Hence, the F-measure is used for evaluating process models. Furthermore, several other evaluations for process models [7, 11] also use the F-score for balancing fitness and precision.

2.3.5. Generalization

Whereas precision measures how closely a model fits a log, a perfect fit is not always desired. In many applications, the true behavior of a process is far more extensive than the behavior captured in a log. Generalization captures how well a model can explain behavior similar to that found in the log.

Generalization

One of the methods to compute generalization is proposed in [3] and is referred to as generalization. This method works similar to the ETC precision. The difference is that it compares the number of observed transitions in different states of the model.

The main idea behind this method is that you can reason about the expected behavior of a model based on the number of times some state is reached and how many different paths have been observed. Let s be some state (marking) in the model, n is the number of times state s occurs when replaying the log, and w is the number of different paths observed leaving s. If n and w are about equal, it can be reasoned that a new

path is taken every time state s is reached, and the model can be expected to allow for more unobserved paths leaving s. On the other hand, if n is significantly higher than w, the log shows the same behavior every time state s is reached, and no unobserved paths leaving s are expected.

Let E the set of all events in the event log, with the corresponding state s of the model just before e occurs. The function $sim(e)$ gives the number of times state s is reached during replay and the function $diff(e)$ gives the number of unique activities observed when in state s. Hence, $n = sim(e)$ and $w = |diff(e)|$. The probability $pnew(w,n)$ of a new state occurring is based on Bayesian analysis and is defined as:

$$pnew(w,n) = \begin{cases} 1 & n \leq w+1 \\ \frac{w(w+1)}{n(n-1)} & n \geq w+2 \end{cases}$$

Using these definitions, the metric $generalization$ is defined as:

$$generalization = 1 - \frac{1}{|E|} \sum_{e \in E} pnew(|diff(e)|, |sim(e)|)$$

As a state is reached more often than there are unique paths leaving the state, $w \leq n$ and $pnew(w,n)$ gives a value between 0 and 1. This means that the average value for $pnew$ over all symbols is also between 0 and 1.

This method for computing generalization does have some drawbacks. First, it is assumed that all traces fit the log in order to reliably compute the state reached before some event is executed. As perfect fitness is hard to achieve, the number of traces suitable to compute generalization is likely limited. Second, if the dataset used features both a high amount of unique traces and unique events, it is to be expected that $n \leq w+1$ holds most of the time, which in turn means the generalization score will almost certainly be close to 1 for most events. This will over-approximate the actual score for generalization.

Cross-validation

An alternative approach is the usage of cross-validation[3, 39], as is often done in other fields of machine learning. By constructing the process model on a subset of the event log, we can test whether the model also accepts the traces not used for its construction. This way, we can measure how well the discovery algorithm can generalize with the model it produces.

The most common type of cross-validation is k-fold cross-validation. Here, the full event log E is split into k disjoint sub-logs of equal size, $E_1, \ldots, E_k$, such that the sub-logs together form the full event log. Using the k sub-logs, there are k evaluation iterations. Iteration i begins with constructing a model using all sub-logs except E_i. Then, the constructed model is evaluated using E_i.

As [39] indicates, there are two methods for partitioning the event log in order to deal with repeated traces. First is simple random partitioning, as is usually done in machine learning. However, if the log contains many duplicate traces, the different partitions will contain all unique traces from the log with a similar distribution. As a result, the constructed models will likely be similar, if not equal, to the model learned over the full log, and the benefit of cross-validation is lost.

The alternative is constructing the partitions based on unique traces. By splitting on unique traces, the partition used for testing is guaranteed to have traces not seen before. Here, the drawback is that the relative frequencies of the different traces are likely distorted, which can significantly alter the constructed models compared to the model constructed using the complete log.

2.3.6. Complexity

The final quality metric for a process model is complexity. With complexity, the aim is to quantify how difficult a model is to interpret. Even though the perception of difficulty is subjective, several metrics have been constructed.

In [26], Mendling gives an extensive overview of complexity metrics for process models. The process model used in this work is an Event-driven Process Chain or EPC for short. This is a modeling language for processes containing events and connectors for and-, or-, and xor-splits. However, for the sake of simplicity, we will use BPMN as the modeling language as it is both highly similar to EPC and it has good support in the area of process mining.

To easily reason about the network structure of a model, some notation is used. First, the set of events is denoted as T for transitions. The sets of split and join nodes are denoted as S and J respectively, and $C = S \cup J$ is the set of all connectors. Together, the set of nodes is the combination of these three: $N = T \cup S \cup J$. The set $A \subseteq N \times N$ contains all arcs in the model. Furthermore, the model can be considered a directed graph with vertices N and edges A.

Size

The most basic metric for model size is the number of nodes, or $S = |N|$. As a model with fewer nodes is easier to understand, a model with a smaller size is preferred.

Size is influenced by the underlying process as well as the model discovery method used. Hence, comparing the size of two models is only possible if at least one of these factors is shared between the two models. In [36], multiple other metrics relating to size for BPMN models are defined, but these are not considered due to how specific they are.

An alternative metric for model size is the diameter, or the length of the longest path between the start and end node. The main idea behind this is that it takes fewer steps to traverse a simpler model. One main shortcoming of diameter is that it is not well defined in case the model has constructs like parallelism or loops.

Overall, the size metrics can give a good quick impression of the scale of the model but cannot explain complexity on their own. For example, take a long sequential model without any connectors and a small, highly interconnected model. The sequential model will score worse on both size and diameter, even though it is easier to understand.

A second issue with size metrics is that nodes used for tasks like synchronization within the process negatively contribute to the score. However, these constructs are desired as they help enforce the quality of the model, and therefore it is beneficial to incorporate them.

Density

Where the size metric mostly looks at the number of nodes in the model, density metrics also consider the number of arcs.

The first density metric is arc density Δ, or the fraction of actual arcs over all possible arcs:

$$\Delta = \frac{|A|}{|N| \cdot (|N| - 1)}$$

A high density indicates there are many dependencies within the model, so a lower density is preferred. The main drawback of using density is that the number of possible arcs grows with the square of the number of nodes. As a result, the metric quickly becomes unreliable for large models, which is an issue as models with 100+ nodes are not uncommon[7].

A better scaling alternative to density is the coefficient of connectivity, or CNC:

$$CNC = \frac{|A|}{|N|}$$

This metric computes the ratio between nodes and arcs in the model with the idea that a higher CNC indicates relatively more arcs hence a more complex model. An alternative is using the inverse of CNC, which was used by De Alvarenga et al. in the context of measuring the complexity of attack graphs [10]. The benefit of using CNC over density is that it scales better for models with more nodes. Furthermore, as each edge is used to connect two different nodes, doubling the CNC gives an indication of the number of edges per node.

Finally, the distribution of node degree can be used as an indicator of how many connections are present in the model. The distribution of degree is computed over the set of connectors C as events always have one incoming arc and one outgoing arc and therefore do not add any complexity. Following this metric, connectors with a high degree are considered more complex, so a lower degree is desired.

The average degree on itself does not tell a lot about the model, given it is heavily influenced by the number of connectors in the model. However, by considering the minimal, maximal, and average degree, we can

get some insights into how common complex connectors are in the model. Besides, it can give insights into whether the model tends towards simple or complex connectors.

Partitionability
Visually, it is easier to understand a model if it can be broken up into smaller parts. The degree to which a model can be split is measured as partition ability.

The most basic metric for partition ability is separability, which measures which fraction of the nodes can be removed to disconnect the model. The reasoning behind separability is that if a node can be removed to split the model into two parts, an analyst can evaluate the two parts separate from each other. Separability Π is computed as:

$$\Pi = \frac{|\{n \in N | n \text{ is a cut-vertex}\}|}{|N| - 2}$$

Here, a cut-vertex (also referred to as an articulation point is a node in a graph where if the node and its corresponding edges are removed, the graph is split into multiple disconnected components. The factor -2 in the denominator represents the start and end node, as removing these nodes can never disconnect the model.

A drawback of separability is that it does not produce a reliable score when there is a lot of parallelism in the model. If a model contains long parallel paths which are sequential themselves, the model is not hard to understand. However, as no node in the path is a cut-vertex, the separability metric considers the nodes as complex to understand.

A related metric is sequentiality, which measures which fraction of arcs is between two transitions. As sequential arcs are easy to understand, a high fraction of sequential arcs indicates a simple model. The sequentiality Ξ is computed as:

$$\Xi = \frac{|A \cap (T \times T)|}{|A|}$$

Opposed to separability, the sequentiality metric does not penalize for parallel sequential sub-models. On the other hand, it also does not consider the complexity of the connectors, which might make the model hard to understand.

Structuredness is a partition ability metric taking another approach. Instead of reasoning about the complexity of singular nodes or arcs, the structuredness metric looks at how well the model can be split up into single-input single-output building blocks. In order to measure this, the original graph G is reduced to the graph G' following the rules described in Section 3.3 in [26]. These rules look for common constructs like sequentiality, connector merges, or sequential paths between similar connectors and replaces the known constructs with a single arc, reducing the number of nodes in G' compared to G. With the node-based size metric S, structuredness ϕ is computed as:

$$\phi = 1 - \frac{S(G')}{S(G)}$$

By incorporating the relations between nodes and arcs, structuredness is able to capture larger structures that are still easy to understand. This feature is lacking in both separability and sequentiality. The drawbacks of structuredness are that it does not incorporate model size (large models are still complex) and that it is relatively expensive to compute.

One thing to note is that process trees have a structuredness of 1, as all building blocks for the tree are reducible following the reduction rules.

Connector Interplay
A final aspect of complexity is the relations between the different connectors. Whereas the structuredness metric described above does capture this to some extent, it is not a good metric to describe how hard the connectors are to understand.

The most basic method for checking how well the connectors relate to each other is the mismatch. For a perfect model, each split node has a corresponding join node such that the part between the split and join can be seen as an isolated sub-model. If this is the case, the degree of all split-nodes is equal to the degree of all join-nodes. Mismatch is the difference between these degrees. Let $d(n)$ be the degree of some node, then mismatch MM is computed as:

$$MM = |sum_{c \in S_{and}} d(c) - \sum_{c \in J_{and}} d(c)| + |sum_{c \in S_{xor}} d(c) - \sum_{c \in J_{xor}} d(c)| + |sum_{c \in S_{or}} d(c) - \sum_{c \in J_{or}} d(c)|$$

The main drawback of mismatch is that it cannot capture asymmetric splits and joins. Take an example with a split-node partitioning the process into three separate paths, join-node a joining two of the split paths, and join-node b joining the third path and the (eventual) output of node a. This case is more complex than a case where the split-node is matched with a single join-node joining all three paths, but the mismatch score in both cases is zero.

Borrowing from the cyclomatic complexity in software analysis[25], the control flow complexity (CFC) measures how many execution paths are possible. For this, the number of different paths which can occur at a split is measured. At a parallel split s_{and}, only one combination is possible regardless of the out-degree of the connector as all outgoing paths are executed. With an exclusive-choice split s_{xor}, there are $|s_{xor}\bullet|$ possible combinations as exactly one of each outgoing path is executed. For an OR-split s_{or}, a total of $2^{s_{or}\bullet} - 1$ combinations are possible as any non-empty combination of the outgoing paths is possible. Using these values, the control flow complexity is computed as:

$$CFC = \sum_{c \in S_{and}} 1 + \sum_{c \in S_{xor}} |c_{xor}\bullet| + \sum_{c \in S_{or}} 2^{|c_{or}\bullet|} - 1$$

The drawback with CFC is that the type of connector heavily influences the final score. If two models have the same structure, but one model uses only AND-connectors and the other model uses OR-connectors, the CFC is significantly higher in the second case even though the models are virtually similar visually.

2.4. Process Mining

The field of process mining knows three main tasks: discovering, monitoring and improving processes by extracting knowledge from event logs[41]. In the context of this thesis, the main focus lies on the discovery aspect. This section discusses the state-of-the-art algorithms for process discovery.

2.4.1. Alpha

One of the first algorithms developed to model concurrency from event logs is the Alpha algorithm, referred to as α[2]. The algorithm uses a set of rules to construct a sound WF-net given an event log following specific requirements. As the set of rules is easily extensible, a series of evolutions over the base α algorithm have been developed.

Base algorithm: α

For the original α algorithm, mining begins with constructing the footprint matrix for the event log W. This matrix contains the directly-follows-relation, or ordering relations, for all pairs of events in W, defined as follows:

- $a >_W b$ if and only if there is some trace $\sigma = t_1 t_2 \dots t_{n-1}$ and $i \in \{1, \dots, n-2\}$ such that $\sigma \in W$ and $t_i = 1$ and $t_{i+1} = b$
- $a \rightarrow_W b$ if and only if $a >_W b$ and $b \not> _W a$
- $a||_W b$ if and only if $a >_W b$ and $b >_W a$
- $a\#_W b$ if and only if $a \not> _W b$ and $b \not> _W a$

The subscript $_W$ is often omitted if the event log is clear given the context.

Constructing the WF-net begins with creating a transition for each activity in A. Then, subsets $X, Y \subset A$ are created such that $\forall x_1, x_2 \in X, x_1 \# x_2$, $\forall y_1, y_2 \in Y, y_1 \# y_2$ and $\forall x \in X, y \in Y, x \rightarrow y$. Besides, both X and Y should be maximal: no sets X' and Y' exist such that the three conditions hold and either $X \subsetneq X' \vee Y \subsetneq Y'$.

For all these pairs of maximal sets X and Y, a place $P_{X,Y}$ is added, together with an arc form each transition $x \in X$ to P_X, Y and an arc from P_X, Y to each transition $y \in Y$. Finally, a source and sink place are added. The source place has an arc to all transitions corresponding to any activity at the beginning of any trace in W. Similarly, the sink place has an incoming arc from all transitions for which the corresponding activity is the final activity for any trace in the log.

Using these rules, α is capable of discovering certain process models of sound WF-nets containing concurrency. For the algorithm to work, some factors do have to hold. To begin, the underlying model has to be a sound WF-net with no duplicate transition labels and no invisible tasks. In addition, the footprint matrix of the event log must be complete and correct with respect to this underlying model: the footprint matrix of both the model and the event log has to be the same.

Not all process models for sound WF-nets can be discovered using the α miner. More precisely, loops of one or two transitions cannot be identified by the miner, and the miner will produce an invalid model if these loops are part of the event log. This issue is addressed in the first evolution: the $\alpha+$ miner.

Short loops: $\alpha+$
Building on the base α miner, the $\alpha+$ algorithm[44] is able to discover loops of one or two transitions. To achieve this, a new definition for log completeness is given, and a pre-processing step is added.

Whereas the base α miner requires the training log to be complete with respect to the ordering relation of underlying WF-net, the $\alpha+$ miner takes this a step further and requires the log to be loop complete. This means that the training log should be complete, and when the model can produce the sub-trace $\ldots aba \ldots, a \neq b$, some trace in the training log must also include that sub-trace. In addition, the ordering relation is also changed with respect to the base α miner:

- $a \triangle_W b$ if and only if there is some trace $\sigma = t_1 t_2 \ldots t_n$ and $i \in \{1, \ldots, n-2\}$ such that $\sigma \in W$ and $t_i = t_{i+2} = a$ and $t_{i+1} = b$
- $a \diamond_W b$ if and only if $a \triangle_W b$ and $b \triangle_W a$
- $a >_W b$ if and only if there is some trace $\sigma = t_1 t_2 \ldots t_{n-1}$ and $i \in \{1, \ldots, n-2\}$ such that $\sigma \in W$ and $t_i = a$ and $t_{i+1} = b$
- $a \rightarrow_W b$ if $a >_W b$ and ($b \not>_W a$ or $a \diamond_W b$)
- $a ||_W b$ if $a >_W b$ and $b >_W a$ and $a \not\diamond_W b$
- $a \#_W b$ if $a \not>_W b$ and $b \not>_W a$

With a loop-complete event log, the mining starts with a pre-processing step identifying and filtering loops of length one. A transition t is identified as a length-one-loop (L1L) if there is some trace $\sigma = t_1 t_2 \ldots t_n$ such that for some $i \in \{1, \ldots, n\} : t = t_{i-1} \wedge t = t_i$. Then, the training traces are filtered of all transitions in a L1L, yielding a new set of training traces which is used as input for the base α algorithm. Finally, the resulting Petri net is enhanced with all transitions representing a L1L, which yields the final Petri net.

Following the proofs in[44], the base α algorithm is able to correctly identify the loops of length two if the training log is loop-complete and free of loops of length one. The second part is enforced by the pre-processing step removing all traces in a L1L.

The main shortcoming of this miner, similar to the base α miner, is the requirement for the event log to be complete (loop-complete in the case of $\alpha+$). As long as this cannot be guaranteed, the output of the miner can not be proven to be sound.

Non-free choice: $\alpha++$
The basic α algorithm can construct process models which are able to explain the behavior in the log, but it cannot identify implicit dependencies between different events. Implicit dependencies are used to match a choice in a Petri net to an earlier choice in the net. In order to deal with implicit dependencies, the $\alpha++$ miner[47] was developed as an extension over the $\alpha+$ miner.

An implicit dependency between tasks a and b is present if three conditions hold: $a\bullet \cap \bullet b \neq \emptyset$, no marking s is reachable from the initial marking such that marking $s' = s - \bullet a + a\bullet$ enables transition b, and some marking s'' is reachable from marking s' which does enable transition b. In other words, when a and b are connected through a place, firing a can never directly enable b, but b becomes enabled somewhere after a was fired. Figure 2.5 shows an example of a WF-net with both implicit and explicit dependencies. In this net, P_1 is the initial place, and P_6 is the output place. Place P_3 is an implicit dependency between T_1 and T_4 as T_4 will always fire after T_1, but no firing sequence exists such that T_4 fires directly after T_1. Using the base α miner, the places P_3 and P_4 and their corresponding arcs will not be discovered.

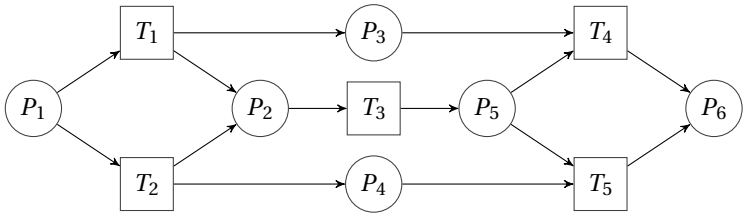


Figure 2.5: Example WF-net with implicit and explicit dependencies

In order to identify implicit dependencies, two new ordering relations are defined to enhance the representation of ordering in the log.

- $a \gg_W b$ if and only if there is some trace $\sigma = t_1 t_2 \dots t_n$ and $1 \leq i < j \leq n$ such that $t_i = a$ and $t_j = b$ and $\exists k \in [i+1, \dots j-1] : t_k \neq a \wedge t_k \neq b$
- $a \succ_W b$ if and only if $a \rightarrow_W b$ or $a \gg_W b$

Mining implicit dependencies is done through a pre-processing and post-processing step, similar to how the $\alpha+$ miner adds the loops of length one. The pre-processing for $\alpha++$ identifies all events which might have an implicit relation between them. This is done over the event log with the enhanced traces with the loops of length one removed. Using the identified relations, the post-processing is done directly after mining using the base α. This step first adds the implicit places for the implicit relations identified in pre-processing and then cleans up any redundant implicit places added. Finally, the post-processing steps from the $\alpha+$ miner are executed to yield the final Petri net.

As all actions for mining implicit tasks are executed on the event log with all length-one-loops removed, the $\alpha++$ algorithm cannot correctly detect any implicit relations for any transition in a L1L. Furthermore, the miner still requires a complete event log in order to produce a valid result.

Invisible tasks: $\alpha\#$
One of the other lIMftations of the α miner is its inability to deal with constructs like optional tasks or repeating sequences. Building on the $\alpha+$ miner, the $\alpha\#$ miner [48] provides an extension to extract invisible tasks from the event log to solve these issues.

An invisible task is a transition in a Petri net that has no label assigned to it and therefore does not show up in the event log. By using invisible tasks, concepts like skipping transitions, looping back to a previous part in the process, or switching execution paths are possible. An invisible task between events a and b is identified based on the event log W using the advanced ordering relation $a \rightsquigarrow_W b$. Given the set of all observed transitions T_V and the causal relations defined by the $\alpha+$ miner, the advanced ordering relation is defined as:

$$a \rightarrow_W b \wedge \exists x, y \in T_V : a \rightarrow_W x \wedge y \rightarrow_W b \wedge y \not\succ_W x \wedge x \nparallel_W b \wedge a \nparallel_W y$$

Figure 2.6 visualizes this relation for the invisible transition t. If a visible elementary path (a path crossing only places and visible transitions) exists between a and b, t allows for skipping a part of the model. When there is a visible elementary path from b to a, t enables a redo-construct, and when no visible elementary path exists between a and b, t allows for switching execution paths.

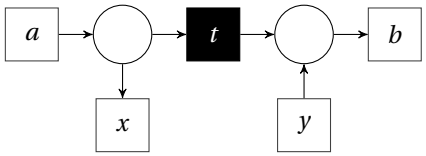


Figure 2.6: Example of the $\rightsquigarrow_W$ relation

In order for an invisible task to be included in the mined model, it has to be prime. For this, the properties for surround, succession, and necessity must hold. Surround requires that an invisible task has a direct preceding and successive visible tasks. Succession indicates that if tasks a and b are connected via an invisible elementary path (a path crossing only places and invisible tasks), b can directly follow a in some traces. Finally, necessity indicates that removing an invisible task by merging its input and output places must add new causal dependencies in the model.

Mining implicit tasks is done by adding additional steps in the $\alpha+$ miner to identify and add the invisible tasks. For this, the event log is still required to be loop-complete. The steps for identifying and incorporating the invisible tasks in the mined model are not pre- or post-processing steps, as was the case for the $\alpha++$ miner.

The drawbacks of the $\alpha\#$ miner are that it cannot detect constructs like non-free choice, even though these constructs are to be expected with skipping events. Besides, there are still some cases where an invisible task has a functional part in a sound WF-net which the $\alpha\#$ miner is not able to identify, like when an invisible task is involved in one whole branch of a parallel execution.

Combining all: $\alpha\$$
By combining the functionality of mining non-free choice constructs in the $\alpha++$ miner and mining invisible tasks with $\alpha\#$, the $\alpha\$$ miner is able to mine invisible tasks in non-free choice constructs. In addition, $\alpha\$$ improves on the method of identifying invisible tasks such that an invisible task involved with a whole branch in a parallel construct can be identified.

The mining process with $\alpha\$$ can be split into five general steps. First is the detection of invisible tasks based on improved mendacious dependency. This dependency updates the $a >_W b$ and $a \rightarrow_W b$ ordering relations to account for all events observed between occurrences of a and b in the event log W. Second, the reachable dependencies are complemented as the ordering relations from $\alpha++$ cannot account for invisible tasks. In order to achieve this, the $a >_W b$-relation from $\alpha++$ is updated to also incorporate the relation between the event before a and the event after b. Third, non-free choice constructs are detected. By slightly changing the detection method as used in $\alpha++$, non-free choice constructs with transitions in a loop of length one are also identified. Fourth, the invisible tasks are adjusted as some of the invisible tasks identified in the first step may produce an unsound model. Adjusting an invisible task can be done by merging it with another invisible task if both tasks are part of a different parallel construct. Alternatively, an invisible task can be split if it is part of both an implicit dependency and mendacious dependency. Finally, the WF-net is constructed by combining the outputs of the previous steps.

As the final evolution of the α family of algorithms, the $\alpha\$$ miner still suffers from the core issue of log completeness. As long as the training log is not fully representative of the event log produced by the underlying model, the miner cannot give a guarantee to produce a sound WF-net, severely lIMfting the miner's viability on sparse datasets. Furthermore, the extensive ruleset introduced over the different iterations of the miner also increased the time required for mining. As identified by [7], the $\alpha\$$ miner often required over one and a half hours to find a model if it was able to terminate within the four-hour time lIMft set. On the other hand, miners like the Split Miner or Inductive Miner rarely required more than ten seconds to construct a sound result.

2.4.2. Inductive Miner

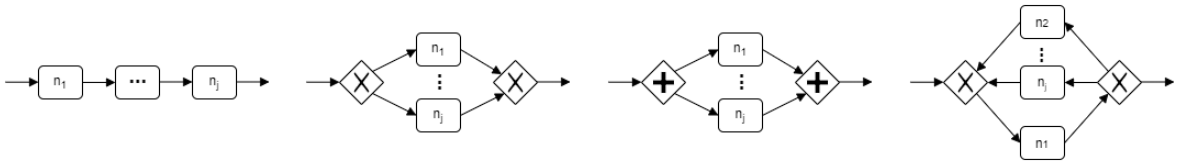
Taking another approach to process mining is the Inductive Miner[23], or IM for short. SIMflar to the α-family of miners, the Inductive Miners use a rule-based approach to construct a process model from an event log.

The difference with the α miners is that IM uses a divide-and-conquer approach to build a model using the event log as the complete truth. In contrast, the α miners use a constructive approach with strong requirements for the completeness of the event log.

The output of the Inductive Miner is a process tree, which is a special kind of process model used to represent block-structured processes. A process tree is defined by activity nodes (which serve as leaves for the tree) and a set of four recursive operational nodes.

- Sequential nodes: $\rightarrow(n_1,\ldots,n_j)$. Execution of a sequential node is done by first executing the child nodes in order.
- Exclusive-choice nodes: $\times(n_1,\ldots,n_j)$. Execution of an exclusive-choice node is done through executing exactly one child node.
- Parallel-cut nodes: $\wedge(n_1,\ldots,n_j)$. Execution of a parallel-cut node is done by executing all child nodes in any order and with the option to interleave the execution of different children.
- Structured-loop nodes: $\circlearrowleft(n_1,n_2,\ldots,n_j)$. Execution of a structured-loop starts with executing n_1. After this, block sequence n_i,n_1 can be executed any number of times, for any value : $2\leq i\leq j$. Structured-loop nodes are the only nodes with the requirement of $j\geq 2$.

A process tree model can be translated into BPMN as shown in Figure 2.7, or an equivalent Petri net. Due to the way the blocks are defined, the model corresponding to each block is sound, and therefore the entire model representing the process tree is sound. Furthermore, one can argue that the tree construction allows for easy interpretation of the model as it can be split into sub-models. This factor is discussed in Section 2.3.6.



(a) $\rightarrow(n_1,\ldots,n_j)$ (b) $\times(n_1,\ldots,n_j)$ (c) $\wedge(n_1,\ldots,n_j)$ (d) $\circlearrowleft(n_1,n_2,\ldots,n_j)$

Figure 2.7: The four different process tree nodes translated to BPMN

Base Miner

The base IM starts with constructing the directly-follows graph G for the event log L, where each activity is represented by a node. An edge between nodes a and b is added when the subsequence $\ldots ab\ldots$ occurs in any trace in the event log. Start nodes are any nodes for which the corresponding activity occurs at the beginning of any trace. Similarly, end nodes have their corresponding activity occur at the end of any trace in the log. These kinds of nodes are denoted with $Start(G)$ and $End(G)$, respectively.

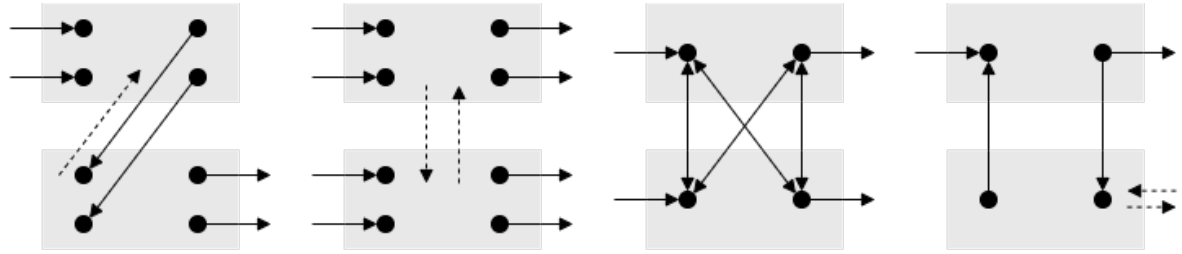
At this point, the choice of modeling a process tree becomes apparent: by defining cutting rules for the directly-follows graph, different nodes for the process tree can be detected. After making a cut in the directly-follows graph, the logs are split in a similar way, and the algorithm is applied recursively on the newly formed sublogs.

Let $a\rightsquigarrow b\in G$ indicate that there exists a path in the directly-follows graph G from node a to node b. Using this notation, the rules for the four different types of node cuts defined by [23] are defined as follows:

- Sequential cut: find $\Sigma_1,\ldots,\Sigma_n$ such that a clear sequential ordering exists between the different pars (Figure 2.8a:
  1. $\forall 1\leq i<j\leq n\wedge a_i\in\Sigma_i\wedge a_j\in\Sigma_j, a_j\rightsquigarrow a_i\notin G$
  2. $\forall 1\leq i<j\leq n\wedge a_i\in\Sigma_i\wedge a_j\in\Sigma_j, a_i\rightsquigarrow a_j\in G$
- Exclusive-choice cut: find $\Sigma_1,\ldots,\Sigma_n$ such that no dependencies exist between the different parts (Figure 2.8b:
  1. $\forall i\neq j\wedge a_i\in\Sigma_i\wedge a_j\in\Sigma_j : (a_i,a_j)\notin G$

- Parallel cut: find $\Sigma_1, \ldots, \Sigma_n$ such that the different parts are fully connected (Figure 2.8c:
  1. $\forall i : \Sigma_i \cap Start(G) \neq \emptyset \wedge \Sigma_i \cap End(G) \neq \emptyset$
  2. $\forall i \neq j \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : (a_i, a_j) \in G$
- Structured-loop cut: find $\Sigma_1, \ldots, \Sigma_n$ such that some initial part is connected to the start and the end, and all other parts are only connected to the initial part (Figure 2.8d):
  1. $Start(G) \cup End(G) \subseteq \Sigma_1$
  2. $\forall i \neq 1 \wedge a_i \in \Sigma_i \wedge a_1 \in \Sigma_1 : (a_1, a_i) \in G \Rightarrow a_1 \in End(G)$
  3. $\forall i \neq 1 \wedge a_i \in \Sigma_i \wedge a_1 \in \Sigma_1 : (a_i, a_1) \in G \Rightarrow a_1 \in Start(G)$
  4. $\forall 1 \neq i \neq j \neq 1 \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : (a_i, a_j) \notin G$
  5. $\forall i \neq 1 \wedge a_i \in \Sigma_i \wedge a_1 \in Start(G) : (\exists a'_i \in \Sigma_1 : (a_i, a'_1) \in G) \Leftrightarrow (a_i, a_1) \in G$
  6. $\forall i \neq 1 \wedge a_i \in \Sigma_i \wedge a_1 \in End(G) : (\exists a'_i \in \Sigma_1 : (a'_1, a_i) \in G) \Leftrightarrow (a_1, a_i) \in G$

For a cut to be valid, it must be non-trivial and maximal. A cut is non-trivial if $j > 1$, and a cut using operator op is maximal if $\forall 1 \leq i \leq j$, op cannot be used to split n_i further. Figure 2.8 shows a visual representation of the different cuts.



(a) Sequential cut (b) Exclusive-choice cut (c) Parallel cut (d) Structured-loop cut

Figure 2.8: Visual representation for the four different cuts. Grey areas show the different parts of the cut. Dashed arrows indicate edges which cannot exist in the directly-follows graph.

The Inductive Miner evaluates the cuts in the order of exclusive-choice, sequence, parallel and finally structured-loop. If no cut can be made, the sublog is modelled as a loop with an empty block $\circlearrowleft(\tau, a_1, \ldots, a_j)$, where $a_1, \ldots, a_j$ are all activities in the sublog, and τ is the silent activity. This construct is equivalent to a flower model.

Looking at the performance of the Inductive Miner, the base algorithm is guaranteed to produce a model capable of explaining all traces in the training log. This guarantee, however, becomes a drawback when the event log contains noise. In such cases, the produced model will either be too tight if the splits can still be made, or too general if no split could be made and the generic flower model is produced for the sublog. In more general terms: the Inductive Miner will always produce a structured model, but it might sacrifice precision or generalization in the process. Furthermore, the miner cannot extract context from duplicate activity labels as all splits are made from the directly-follows graph. As a result, a log with duplicate activity labels will likely result in a model which is too generic.

Inductive Miner Infrequent

Building on the base Inductive Miner, the Inductive Miner Infrequent[19] (IMf for short) updates the mining process such that infrequently observed events have less of an impact on the overall model.

Mining with IMf is the same as with the base IM up until a split cannot be made. At this point, the base IM will add a generic flower model to the result, which can cause the model to allow a lot of unobserved behavior. Instead, the infrequent filter of IMf is applied.

First, a filter is applied to the operator and cut selection by removing infrequent edges from the directly-follows graph. An edge (a, b) is filtered if it occurs less than k times ($0 \leq k \leq 1$) than the most frequent edge leaving a. If this still does not yield a valid cut, the directly-follows graph is replaced by the eventually-follows graph (where the edge (a, b) indicates that event b occurred somewhere after event a in some trace), which is also filtered on infrequent edges.

As this filtering on the operator and cut selection may introduce empty traces (traces of zero events) to the sublogs, the sublogs are also filtered on base cases. Whereas the only base case for IM is a log containing a single event, IMf can also have a base case where the log can contain the empty trace. In order to deal with this and the possible extra noise introduced, the empty trace is only included in the sublogs if it is frequent enough compared to the number of traces in the sublog, based on k.

The third and final filter applied is the filter on log splitting. As the previous filtering rules might have selected a cut that does not fit the infrequent behavior, the cutting rules have to be changed slightly. In order to enable splitting the log, traces are filtered based on observed event instances not fitting with the identified cuts.

Overall, the IMf variant trades a bit of fitness to achieve a higher precision compared to the base IM. Therefore, IMf provides a better alternative if the goal of mining a log is to gain a high-level understanding of the underlying process. The mining process depends on identifying the different splits, which can be done efficiently using several graph techniques, resulting in low mining times.

Other Inductive Miners

In addition to the infrequent variant of the Inductive Miners, other variants have been developed. The IM-incomplete[20] deals with incomplete logs by using a probabilistic approach to identify the cuts for the directly-follows graph. The IM-directly-follows based[21] gives an approach to use the IM by using split directly-follows in recursive calls instead of splits in the logs, improving performance on large datasets. In this work, an adaptation is also given to use the directly-follows based technique together with the infrequent or the incomplete variant of the IM. For dealing with event logs where timing information is present for the different events in a trace, the IM-life-cycle is proposed in [22].

2.4.3. Heuristics Miner

The Heuristics Miner[46] (HM) and Flexible Heuristics Miner[45] (FHM) are developed as an alternative to the base α miner. This type of miners leverages the frequency of traces and patterns in the traces to create a model more robust against noise in the event log compared to the base α miner.

Mining in the family of heuristics miners is a four-step process. In the first step, the frequency of each event in the event log is computed together with the frequency matrix. This matrix is similar to the footprint matrix from the α-family of miners, but instead of only recording if the directly-follows relation was observed, the frequency matrix keeps track of how often this relation was observed for each pair of events. In addition to direct succession frequencies ($|a > b|$), the repetition count ($|a \gg b$, the count of the subsequence $\ldots aba \ldots$) and indirect succession count ($|a \ggg b|$, or the total number of times a is eventually followed by b, without any a or b in between) are also computed.

Then, the dependency matrix is constructed using the dependency measure, where a directed edge is added if the measure exceeds a user-defined threshold. Following the original Heuristics Miner, the dependency measure is computed as follows:

$$dependency(a >_L b) = \begin{cases} \frac{|a>_L b| - |b>_L a|}{|a>_L b| + |b>_L a| + 1} & a \neq b \\ \frac{|a>_L a|}{|a>_L a| + 1} & a = b \end{cases}$$

In the third step, the parallel and exclusive-choice relations are mined based on the dependency matrix. For this step, the Flexible HM and the base HM use slightly different measures.

The fourth final step adds long-distance dependencies to the model. For this, a new dependency matrix is constructed based on the $|a \ggg b|$ frequencies. As this relation produces a lot of false positives in the model (for example, the relation holds between the start event and each event, and between each event and the final event), a filtering check is done. This check verifies if it is possible to reach the final event in the trace from a without executing b in the process before adding a long-distance dependency from a to b. This step is optional in the heuristics miners.

As discussed in [42], both HM and FHM have some shortcomings. First, the algorithms add connections based solely on frequency, meaning the miners may produce disconnected models. Second, the miners cannot deal with duplicate task labels, which can yield an overly complex model if duplicate task labels are present. Furthermore, the published implementation for the $|a \ggg B|$ relation only considers one occurrence per trace at most, whereas this relation can occur multiple times within the same trace.

Fodina

The Fodina miner[42] is developed as an evolution over the HM and FHM to mitigate the shortcomings in these miners. The miner changes some aspects of the base heuristics miners to better deal with noise, infrequent events, and duplicate activities.

Mining begins with constructing a mapping between the events in the event log and tasks in the final model. For the base miner, this is a simple one-to-one mapping, but when mining duplicates, the process is a bit more involved. By keeping track of the direct context of an event a (the event directly before and after a), a user-defined threshold is used to distinguish between previously observed occurrences of a, or if the current context is different enough to create a new task in the log. Using the event log, the frequencies for all relations are gathered in the same way as the other heuristics miners, and the basic dependency graph is constructed. For constructing the dependency graph, Fodina uses a different threshold for dependency, length-one-loops, and length-two-loops, whereas the HM uses one shared threshold for all three occasions. Then, the start and end tasks are added to the dependency graph. Following this, Fodina verifies whether all tasks in the dependency graph are reachable and enhance the dependency graph with long-distance dependencies. These two steps are, however, optional, and their inclusion depends on the settings defined by the user. Finally, the dependency graph is converted to the desired output model, which is a Causal net. This Causal net can also be converted to a Petri net if that is the desired output format.

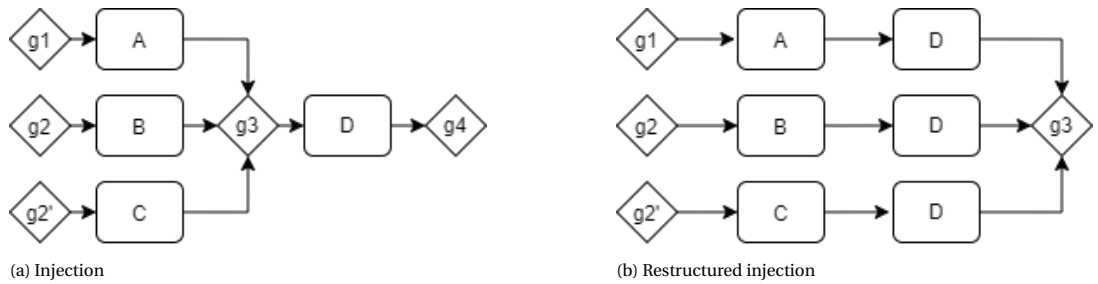
With the proposed extensions, Fodina can construct more precise models compared to the base HM and FHM. However, to achieve this, a range of settings is available, making Fodina harder to use as choosing appropriate values for these settings requires expert knowledge. On the other hand, given this knowledge, Fodina is more powerful than HM and HMF. Furthermore, the steps taken to improve the soundness of the resulting model are not enough as Fodina can still produce unsound models.

2.4.4. Structured Heuristic Miner

Also building on the Heuristics Miner is the Structured miner[6], which adds post-processing to the output of the HM to construct more structured models following the structuredness metric as defined in Section 2.3.6. The goal of the structuring process is to reduce the complexity of the discovered model. In order to achieve this, the structured miner uses a four-step process: model discovery, structuring, soundness repair, and clones removal.

In the model discovery step, a pre-existing miner is used to construct a model used as the basis for the other steps. For this, any miner producing a model with parallel and exclusive-choice gateways and a single start and end node can be used. In the original paper, the authors used the implementation of the Heuristics Miner in ProM 6 as it is able to provide accurate models. This combination is referred to as sHM.

Using the base model, the structuring process removes injections and ejections in order to make the model less complex. An injection is a structure where multiple paths are merged through a join gateway, followed by one unique path. This structure is changed by pushing down the join gateway into each of the paths leading into it. Figure 2.9 shows the general structure before and after applying the push-down operation on an injection. In order for an injection to be valid, $g3$ must have at least two unique input paths, so the path $g2', C$ is optional. Similarly, an ejection is a structure where one path reaches a split gateway, followed

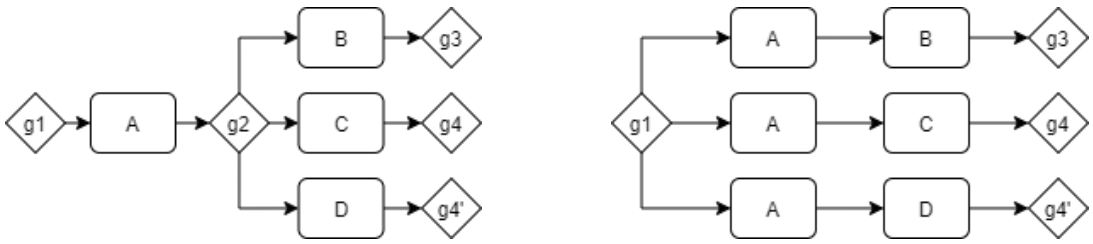


(a) Injection

(b) Restructured injection

Figure 2.9: Example of push-down to restructure injections

by multiple unique paths. This structure is changed by pulling up the split gateway into the paths leaving it. Figure 2.10 shows the general structure before and after applying the pull-up operation on an ejection. For a valid ejection, $g2$ must have at least two output paths, so the path $D, g4'$ is optional.



(a) Ejection (b) Restructured ejection

Figure 2.10: Example of pull-up to restructure injections

After restructuring, the soundness repair step makes an attempt to fix blocks where gateways are mismatched. These blocks are identified by parsing the model as a Refined Process Structure Tree as discussed in [35]. Finally, the clone refactoring step tries to merge duplicate parts introduced by the restructuring step such that the structuredness metric is not negatively impacted.

Using these four steps, SHM is able to improve a model found by another miner, even being able to make some unsound models sound. Even though SHM includes as a step to repair parts violating soundness, the fixes are not complete, and the resulting model can still be unsound. Furthermore, the theoretical runtime of the algorithm is bad with a worst-case runtime of $O\left(\binom{n}{4}^{\binom{n}{4}}\right)$, with n being the number of nodes in the model. This poor performance is reached if all gateways in the model are part of an injector or ejection.

2.4.5. Split Miner

The Split Miner[8] is one of the latest high-profile process discovery algorithms. By filtering and transforming the directly-follows relations observed in the event log, the Split Miner constructs a BPMN model focusing on a low control-flow complexity. Furthermore, it is the only miner besides the process tree miners with a soundness guarantee for acyclic process models. Opposed to a process tree, the Split Miner constructs a BPMN model and is therefore not limited to block-structured models.

Mining begins with constructing the directly-follows graph from the traces in the event log. Using this graph, $|a \rightarrow b|$ denotes the frequency of the edge from event a to event b based on the event log. Furthermore, $|a \leftrightarrow b|$ denotes the frequency of the pattern $\ldots aba \ldots$ in the event log. With these relations, edges in a self-loop or short-loop are filtered from the DFG, as these edges break the distinction between parallelism and causality. An edge for self-loop starts and ends at the same node, so detection is trivial. For a short-loop, the condition $|a \rightarrow a| = 0 \wedge |b \rightarrow b| = 0 \wedge |a \leftrightarrow b| + |b \leftrightarrow a| \neq 0$ has to hold.

With these loops removed, the edges between concurrent events are removed. For events a and b to be concurrent, three conditions must hold:

$$|a \rightarrow b| > 0 \quad \wedge \quad |b \rightarrow a| > 0 \tag{2.1}$$

$$|a \leftrightarrow b| + |b \leftrightarrow a| = 0 \tag{2.2}$$

$$\frac{||a \rightarrow b| - |b \rightarrow a||}{|a \rightarrow b| + |b \rightarrow a|} < \epsilon, \quad 0 \leq \epsilon \leq 1 \tag{2.3}$$

Condition 2.1 requires that b is observed after a and vice versa. Condition 2.2 enforces that no loop-relation exists between a and b. For condition 2.3, assuming a and b are independent, the frequency $|a \rightarrow b|$ should be about equal to the frequency of $|b \rightarrow a|$. Here, a smaller value of ϵ requires a smaller difference between the frequencies.

After removing the concurrency-related edges, the DFG is filtered based on path frequency based on some user-defined percentile η. This is done to reduce the complexity of the final model by removing low-frequency edges. By using a cost computation based on Dijkstra's algorithm, it is ensured that each node lies on some path between the start and end node.

Constructing the final BPMN model begins with adding an event for each node in the DFG. Events can be connected if the corresponding edge (a, b) in the DFG is the only outgoing edge of a and the only incoming edge of b. If a node a has multiple outgoing edges in the DFG, an AND-split or XOR-split is added, depending

on the relation between the events observed after a. Identifying the join gateways is done based on evaluating single-entry single-exit nodes in the Refined Process Structure Tree [35]. Finally, the self-loops are added to the BPMN model, and a post-processing step reduces the CFC by replacing as OR-joins with an XOR-, or AND-join where possible.

As the Split Miner operates mostly over the filtered directly-follows graph, the time required to construct a model is low. Like the process tree miners, the Split Miner has a soundness guarantee for acyclic processes. Compared to the Inductive Miner, the Split Miner is able to achieve a balance between fitness and precision through filtering, whereas process trees usually achieve a high fitness at the cost of lower precision. Based on experimental results, the Split Miner can consistently outperform the IM-family of miners on real datasets while also requiring less time to construct a model.

2.4.6. Overview

Table 2.1 gives a short overview of the capabilities of the different miners. Even though the different miners construct different types of models, all model types can be converted to a Petri net, making comparison easier.

Miner	Model Type	Detects duplicate events	Strengths	Weaknesses
α\$	Petri net	No	Has strong guarantees iff the event log is complete	Slow, cannot guarantee soundness for non-complete logs
Fodina	Causal net	Configurable	Good runtime, highly configurable	Configurations require expert knowledge, cannot guarantee soundness
IM	Process tree	No	Fast, always produces sound models, flexible	Limited to block-structured processes
sHM6	BPMN	No	Soundness repair step improves performance over other heuristics miners	Poor theoretical upper bound on runtime, cannot guarantee soundness
SM	BPMN	Configurable	Fast, guarantees soundness in most cases	Can result in complex OR-join gateways

Table 2.1: Comparison of different process discovery algorithms

2.5. Learning Attack Graphs

Through the usage of an intrusion detection system, or IDS for short, security experts are alerted of possible attacks in a computer network. One drawback of these systems with the modern internet is the volume of alerts generated by these systems. For decently sized networks, it is not uncommon when hundreds of thousands or millions of alerts are generated in a single day by the IDS active in the network[10]. In addition, some attacks consist of multiple steps or stages, meaning they are not captured by a single alert.

By filtering, aggregating, and combining the alerts generated, attack graphs mapping out attacker strategies can be constructed. Using these attack graphs, security experts can get better insights into the vulnerabilities of the network and take measures to prevent future attacks.

Most methods used for constructing attack graphs use some form of prior knowledge of the network architecture to predict vulnerabilities. In addition, there exists a method that is capable of constructing attack graphs based solely on the alerts generated by the IDS. This work is used as the basis of this thesis and is the only method of its kind to the best of our knowledge.

2.5.1. Combining Alerts with Prior Knowledge

Almost all work into constructing attack graphs for multistage attacks uses some prior knowledge like known vulnerabilities and network topology. As a basis, the MulVAL framework takes this prior knowledge to find paths an attacker could take to perform certain attacks in the network. Using this framework, several other methods have been developed to correlate the results from MulVAL with alerts generated by an IDS.

MulVAL

The Multihost, multistage Vulnerability Analysis framework, or MulVAL[33] is used as a basis for most methods in attack graph generation. This framework uses six different types of information as input: security advisories on known vulnerabilities, host configuration, network configuration, principals (information on the users of the network), interaction between different components of the network, and the access policy. All types of information are represented as predicates, which in turn are used by DATALOG. Using these predicates, clauses can be defined as shown in Figure 2.11.

```
1 execCode(Attacker, Host, Priv) :-
2     vulExists(Host, VulID, Program),
3     vulProperty(VulID, remoteExploit, privEscalation),
4     networkService(Host, Program, Protocol, Port, Priv),
5     netAccess(Attacker, Host, Protocol, Port),
6     malicious(Attacker)
```

Figure 2.11: Definition for remote code execution using MulVAL. Capitalized terms are variables (Attacker, Host) whereas non-capitalized terms are constants (remoteExploit).

Line 1 defines the clause `execCode(Attacker, Host, Priv)`, based on the variables `Attacker`, `Host` and `Priv` (all capitalized identifiers are variable). For `execCode` to evaluate to true, five other clauses have to hold. First, a vulnerability referred to as `VulID` must exist on `Host` in program `Program`. Second, the vulnerability `VulID` must have certain properties: it must be a `remoteExploit` with the consequence for the option of `privEscalation` (both are constant as indicated by the identifier starting with a lowercase letter). Third, looking at the services on the network, the `Host` must be running the vulnerable `Program`, accept connections using the `Protocol` on the given `Port`, and running with the given `Privilege`. Fourth, the network user `Attacker` must have access to `Host` using the given `Protocol` and `Port`. Finally, the network user `Attacker` must be malicious. When some configuration exists for the variables such that all clauses on lines 2 through 6 hold, then the original `execCode` clause on line 1 also evaluates to true, indicating some network user `Attacker` can execute code on the given `Host` with the privileges `Priv`.

Using similar reasoning, other types of attacks or vulnerabilities can be modeled for the network, like file access by unauthorized users or an attacker moving laterally through the network. Furthermore, by adding hypothetical vulnerabilities, a network administrator can gain insights into the network's resiliency in the case some exploit becomes available.

Roschke et al.

In the work of Roschke et al.[37], an extension of the MulVAL framework is proposed. Through the proposed five-step pipeline, the vulnerability analysis of MulVAL is combined with a set of alerts in order to create attack paths to use during forensic analysis after an incident.

The steps of the pipeline are preparation, alert mapping, aggregation of alerts, building an alert dependency graph, and searching for related subsets of alerts. In this pipeline, an alert a is defined at the tuple (t, s, d, c) where t denotes the timestamp, s and d respectively denote the source IP and destination IP of the packet triggering the alert, and c is the classification of the alert.

First is the preparation step, where a basic attack graph $AG = (V, E)$ is constructed. The vertices of the graph are the set of impacts as defined by MulVAL, where a vertex v is added for each triple of impact, host, and vulnerability reference. At this point, no edges are yet present in the graph.

The second step maps the set of alerts to the vertices in the graph. This mapping is done based on source IP (attacks by the same attacker), destination IP (attacks targeted on the same host), or the classification of the alerts (attacks of the same type). Combinations of the three features are also possible to create attack graphs for more general or specific kinds of attacks.

The third step is the aggregation of alerts with the goal of reducing the number of alerts needed for the further steps. This is achieved by grouping similar alerts. Two alerts are considered similar when they are assigned to the same vertex, the source IP, destination IP, and classification are equal, and the difference in the time difference between the two alerts is below a given threshold.

The fourth step is finding the dependencies between alerts and representing this in a dependency graph DG. The vertices for the graph are the set of aggregated alerts from step three. For the vertices of aggregated

alerts x and y, the dependency graph contains an edge (x, y) if x occurred before y and the chosen matching rule holds. Three matching rules are defined as follows:

- There is an edge from x to y in the attack graph
- There is a path from x to y in the attack graph of a maximum length n
- There is a path from x to y in the attack graph of any length

The fifth step is searching for related subsets of alerts. The DG created by the previous steps contains subsets of alerts that could be part of the attack scenario. Through finding paths through DG and ordering all alerts mapped to the vertices in the path by timestamp, an overview of paths an attacker could have taken is created.

Hu et al.
As attack strategies can continuously evolve and unknown vulnerabilities can be exploited, an IDS could miss certain attacks. With some alerts possibly missing, the attack graphs constructed might miss some key steps from the attackers. The method by Hu et al.[15] tries to tackle this problem by mapping alerts generated by an IDS to the vulnerabilities identified by MulVAL. In addition, the method aims to also detect attack paths from unsuccessful attacks, which still provide insights into the strategies used by attackers.

First, the attack graph is created using MulVAL. The alerts from the IDS are mapped to the nodes of this attack graph based on the source IP, destination IP, and the type of the alert.

Using the mapped alerts, the second step generates attack sequences by correlating attributes from the alerts. A sequence consists of alerts $a_1, a_2, \ldots, a_n$ where the mapped node for a_i is connected to the mapped node for alert a_{i+1} in the attack graph, and alert a_i was logged before a_{i+1}. New sequences are added when the previous sequences cannot be extended further.

The third step merges similar attack chains to remove duplicates, resulting in initial (possibly broken) scenarios. Merging is done by clustering the alert sequences based on the edit distance between different sequences. Each of these clusters is referred to as an attack scenario.

In the fourth and final step, the causality of multi-step and multi-stage attacks is used to fill gaps in the scenarios. Here, two attack scenarios AS_i and AS_j are merged based on the minimal possible distance between some $a_i \in AS_i$ and $a_j \in AS_j$, based on the same methods as used in step two. After merging, AS_i and AS_j are removed from the scenarios, and the new scenario is added. Merging continues until no two scenarios can be merged, meaning it is possible for one of the final scenarios to consists of three or more sub-scenarios as returned by the third step.

De Alvarenga et al.
In [10], De Alvarenga et al. introduce a method of constructing attack graphs using process mining. This is done through a three-step process: alert aggregation, model constructing, and model clustering.

To begin, the alerts from the IDS are aggregated based on a one-to-many or many-to-one scheme. With the one-to-many scheme, aggregation is done based on the source IP to group alerts based on the same attacker. For the many-to-one scheme, the alerts are aggregated based on destination IP to extract all attacks on the same target. After filtering duplicate alerts from the aggregated sets, an event log is constructed where the event labels are the alert signature, and different cases are distinguished based on the time frame of the alerts.

Then, a process model is constructed over this event log. In the paper, the process discovery algorithm is called the 'Model Discovery Algorithm', but following the description of this algorithm, it just constructs a Markov chain over the event log. These models can quickly become visually complex due to the high number of nodes and edges. When a model is labeled as complex, it is split up by clustering the traces underlying the model.

This clustering technique starts with encoding all traces based on which events are in the trace. If the complete log contains n unique events, each trace is represented by an n-bit vector. Then, the distance between all bit vectors of the traces is computed using the Jaccard distance. With this distance matrix, a hierarchical clustering technique is applied to group similar traces together. Based on a simplicity score (the inverse of the CNC as discussed in Section 2.3), the cluster hierarchy is traversed to separate clusters for which the model is not complex. If some cluster cannot be split into non-complex submodels, the traces are separated based on the timestamp of the events, and the process is repeated.

2.5.2. Solely Alert-based

To the bets of our knowledge, the method by Nadeem et al.[31] currently is the only method capable of constructing attack graphs which does not require prior knowledge and works solely on the generated alerts. This method takes a set of IDS alerts and converts these to attack graphs through a three-step process.

Aggregating alerts

The first step is aggregating the raw IDS alerts into attack sequences. For this, the information from the raw alerts is first mapped following the attack-intent framework from Moskal et al.[27]. Then, the alerts are grouped based on source and destination IP and aggregated into sequences based on the timestamps. After splitting the alert sequences based on temporal density, they are encoded based on the related attack stage and targeted service.

Learning the model

Using the encoded alert sequences, a suffix-based PDFA (S-PDFA) is constructed. A suffix-based model is chosen as different attacks can take different paths to lead to the same end goal. Furthermore, it gives insights into the past of a severe event, i.e., which low-severity alerts occur before a high-severity alert.

To construct the model, the implementation of Alergia in the flexfringe tool is used. In order to create the best model for the data, some of the default settings were changed. First are the `state_count` and `symbol_count` parameters, which were both set to five, lowering the bar of 'sufficient evidence' for the merging process. Second is the `Markovian` setting, ensuring that all incoming transitions for a state have the same label. This setting makes the model easier to interpret, especially for states with high-severity incoming transitions. Third and finally, `sink states` were enabled with the threshold of five occurrences. Figure 2.12 shows the S-PDFA resulting from these settings for the CPTC'18 dataset. Following the paper, states and transitions occurring less than five times are not rendered to make the model more interpretable.

Constructing the attack graph

By replaying the traces over the constructed S-PDFA, the state sequences corresponding to each alert sequence can be retrieved. Then, using these state sequences, the final attack graphs are constructed. For these graphs, the objectives are first defined based on the attack stage, targeted service and the resulting state from the replay over the S-PDFA. Then, the objectives are combined with the victim hosts and all traces relating to these combinations are replayed over the S-PDFA. Using the replayed traces for each combination of objective and victim, the attack graph is constructed based on the state sequences from the replay and the timing information from the underlying alert sequences.

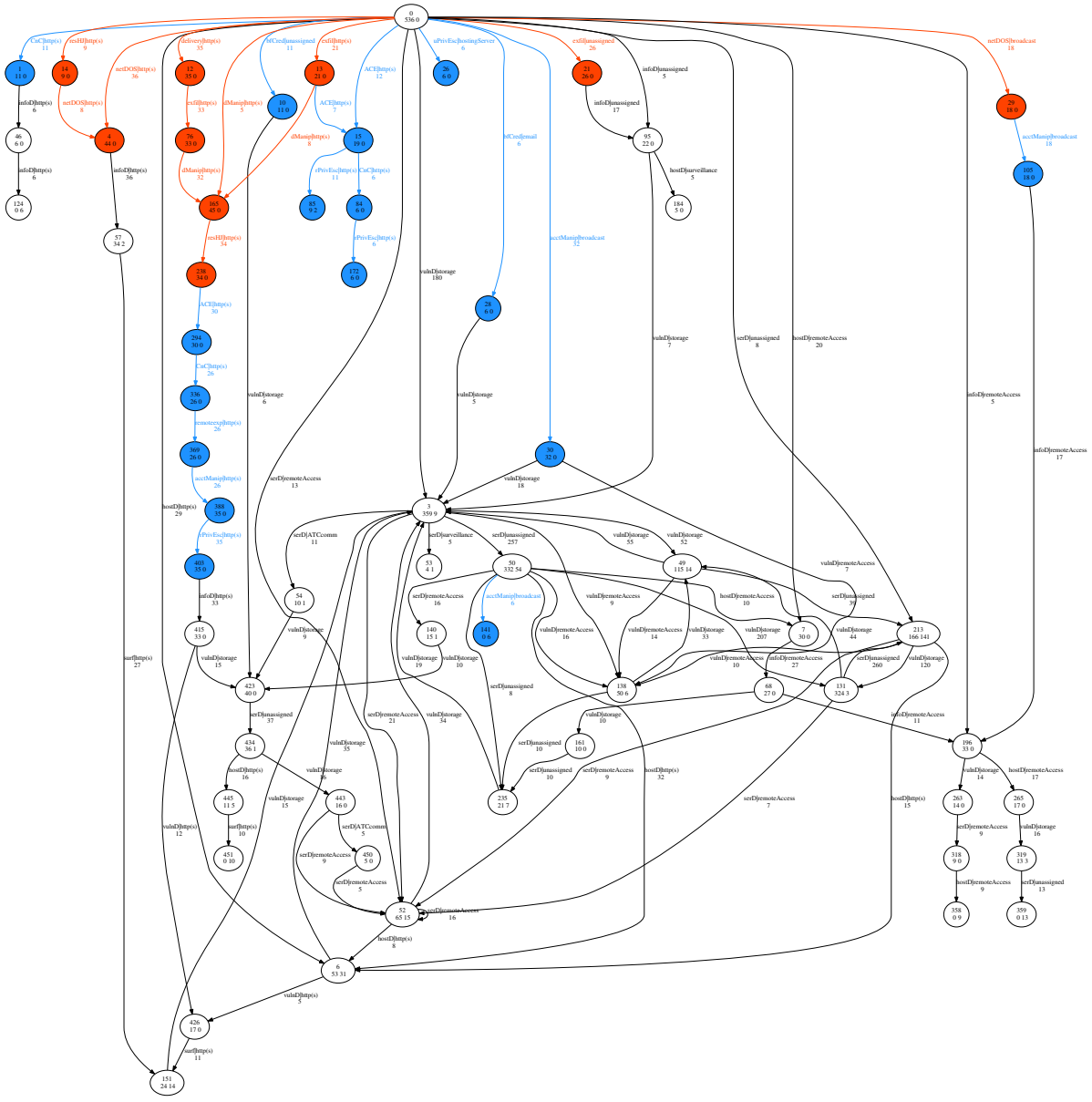


Figure 2.12: S-PDFA constructed for the CPTC'18 dataset.

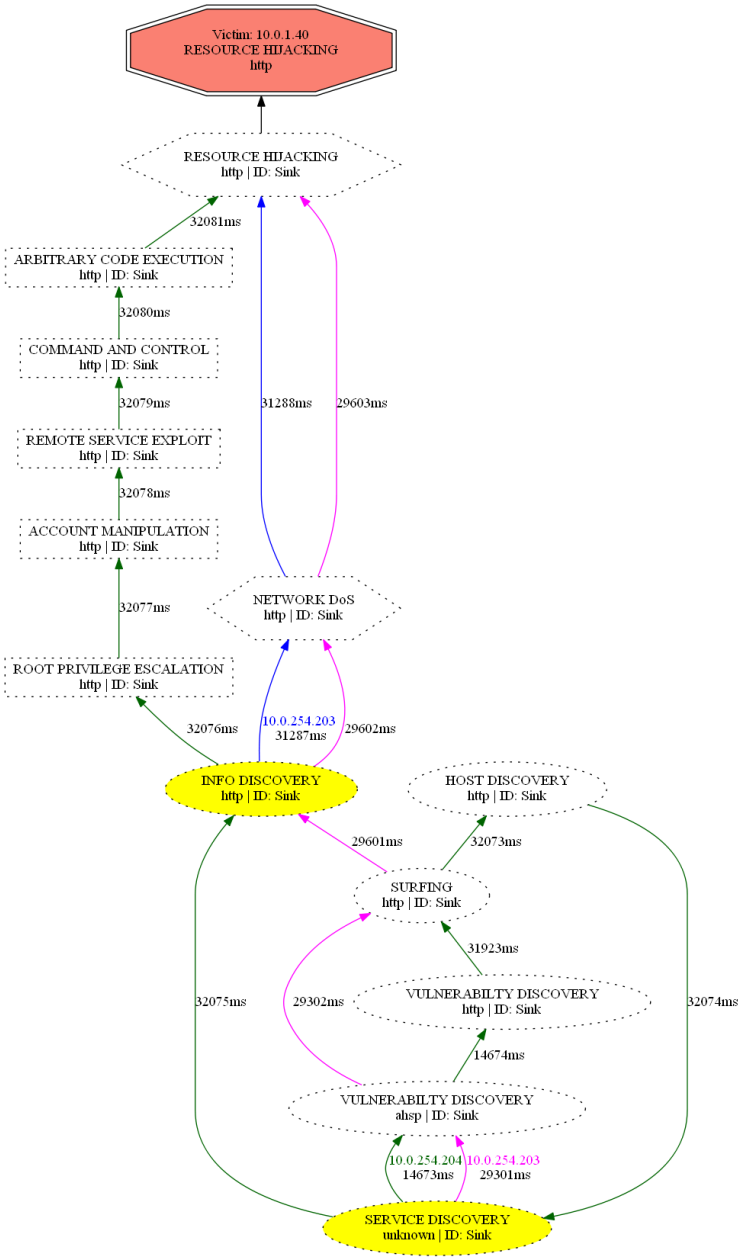


Figure 2.13: Example attack graph for Resource hijacking over http for the victim host 10.0.1.40.

3

Evaluation Setup

This chapter gives an outline of the evaluation setup used in this thesis. First, the IDS alert datasets are discussed. This is followed by the methods for constructing the process models and state machines. Finally, the metrics for measuring the quality of the models are discussed.

3.1. Datasets

The datasets used for the evaluation stem from the 2017 and 2018 editions of the Collegiate Penetration Testing Competition[1] [28], or CPTC for short. In this competition, student teams compete against each other to test their penetration testing skills in a simulated enterprise network. The competition scenario changes each year, with the 2017 version featuring an online election service company and the 2018 version revolving around a ride-sharing organization. In both years, the simulated networks were monitored by the Suricata IDS[2] which provided the alert datasets.

For the experiments in this thesis, the Suricata alerts are aggregated using the method from Nadeem et al.[31]. The event sequences resulting from the aggregation process are then used as the input for the process miners and the state machine learning. Table 3.1 shows the size of datasets before and after the alert aggregation. Even though the CPTC'17 dataset featured fewer raw alerts than CPTC'18 (43.602 vs. 101.571), the composition resulted in more aggregated alert sequences.

Dataset	Traces		Events		Trace length		
	Total	Unique	Total	Unique	Min	Mean	Max
CPTC'17	965	777 (80.5%)	5.012	92	3	5	19
CPTC'18	536	327 (61.0%)	3.943	112	3	7	29
BPIC12	13,087	4,371 (33.4%)	262.200	36	3	20	175
$BPIC13_{cp}$	1,487	183 (12.3%)	6.660	7	1	4	35
$BPIC13_{inc}$	7,554	1,512 (20.0%)	65.533	13	1	9	123
$BPIC14_{f}$	41,353	14,928 (36.1%)	369.485	9	3	9	167
$BPIC15_{1f}$	902	295 (32.7%)	21.656	70	5	24	50
$BPIC15_{2f}$	681	420 (61.7%)	24.678	82	4	36	63
$BPIC15_{3f}$	1,369	826 (60.3%)	43.786	62	4	32	54
$BPIC15_{4f}$	860	451 (52.4%)	29.403	65	5	34	54
$BPIC15_{5f}$	975	446 (45.7%)	30.030	74	4	31	61
$BPIC17_{f}$	21,861	8,766 (40.1%)	714.198	41	11	33	113
RTFMP	150,370	301 (0.2%)	561.470	11	2	4	20
SEPSIS	1,050	846 (80.6%)	15.214	16	3	14	185

Table 3.1: Size of the CPTC datasets, compared to the BPIC datasets as shown by [7].

[1] https://globalcptc.org/
[2] https://suricata.io/

As a comparison, the table also shows the size of the datasets used in the 2018 survey on process mining algorithms[7]. These are the Business Process Intelligence Challenge (BPIC) datasets, Road Traffic Fines Management Process (RTFMP) dataset, and the SEPSIS cases dataset. We can see here is that the CPTC datasets feature a relatively high amount of unique events compared to the number of unique traces. Furthermore, the CPTC datasets have both the lowest number of total events and the highest number of unique events. Only the BPIC15 datasets are comparable to the CPTC data regarding unique events and traces. However, the CPTC traces are a lot shorter than the BPIC15 traces in minimum, mean, and maximum length. In addition, we see that the mean trace length and the number of unique events differ with a factor of at most 2 compared to the other datasets, whereas this factor lies above 15 for the CPTC datasets. This all indicates that the distribution of the events throughout the traces is a lot sparser for the CPTC datasets compared to reference datasets used in process mining. In turn, this sparseness can be a good indication that the CPTC datasets do not contain enough information to construct process models.

3.2. Constructing Models

In order to evaluate models, we first need to construct the models. This section goes into detail which methods from process mining and state machine learning are used to create the models.

3.2.1. Process Miners

The state-of-the-art algorithms in process mining are selected based on the 2018 benchmark by Augusto et al.[7]. In this research, seven miners were evaluated: Alpha$ ($\alpha$$), Inductive Miner-infrequent (IMf), the Evolutionary Tree Miner (ETM), Fodina (FO), the Structured miner using Heuristics Miner in ProM 6 (sHM6), the Hybrid ILP miner (HILP) and the Split Miner (SM). Complementing these seven miners, the base Inductive Miner (IM) is also considered in this thesis as this miner guarantees perfect fitness.

From these eight miners, the Evolutionary Tree Miner and Hybrid ILP miner are not included for evaluation. Compared to IMf, the other mining algorithm constructing process trees, ETM trades fitness to achieve a higher precision, which in turn yields an improved balanced F-score. However, this comes at a considerable trade-off in mining time compared to IMf: where IMf constructs a model only a couple of seconds, ETM used the full four-hour time limit set for mining to construct the models. This significantly increased mining time combined with the lower F-score makes the ETM an overall worse fit in our case compared to IMf, hence ETM is not considered for evaluation.

Evaluating the Hybrid ILP miner had its own issues with the implementation of the miner. Furthermore, HILP was often unable to construct a sound model during the benchmark evaluation. Given that the CPTC datasets are likely harder compared to the other reference datasets, as discussed in Section 3.1, it can be expected that the models generated over the IDS alert datasets are also unsound and therefore not usable anyways. As a result of these factors, it was decided to drop the HILP miner from the evaluation.

The $\alpha$$, FO, sHM6, and SM miners are evaluated using the default settings. For the Inductive Miner infrequent, a range of noise thresholds is tested. This range starts at 0, making the miner equivalent to the base Inductive Miner, and ends with 0.20 (the default value for IMf), using increments of 0.05. This way, we can also get an insight into the effect of filtering regarding the F-score, and how the fitness/precision trade-off changes with different settings. All miners are allowed four hours to construct a model for each of the datasets. For evaluation, the resulting models are converted to a Petri net for performance measures and a BPMN diagram to measure complexity.

As a baseline for the evaluations, the flower model (FLOWER) and prefix tree model (PTM) are also included. The flower model accepts all traces over the tasks in the training data, hence gives perfect fitness and good generalization at the cost of precision. The prefix tree model is the opposite, only accepting the trace prefixes observed in the training data, giving perfect fitness and precision at the cost of generalization. By considering these trivial models in the evaluation, we can get insights into how well the other miners balance precision and generalization compared to the baseline.
Section 2.4 contains more background information on how the different process mining algorithms work.

3.2.2. State Machines

The flexfringe tool[43] is used for constructing state machines. This tool offers a large variety of methods for constructing state machines. In total, six different configurations are tested, all based on the Alergia state merging method: bigram, convert sinks, Markov, no sink merging, original and search. These different con-

figurations mainly differ in two settings implemented in flexfringe: the `Markovian` property and `sinks`. The `Markovian` setting is enabled in all configurations. It ensures that all incoming transitions for a state have the same label, generally reducing the complexity of the models. The `sinks` are states which occur less frequent than a set threshold (set to five in all configurations using sinks). Due to the low frequency of these states, the statistical tests used for merging become unreliable. In order to deal with this issue, different merging strategies can be applied to sink states.
Appendix A contains the exact configuration files used for evaluation, and a description of flexfringe and the Alergia algorithm can be found in Section 2.1.

Common settings
The custom configurations share a set of core settings. To begin, the `state_count` and `symbol_count` settings require a minimal number of occurrences of a state or transition respectively to be considered in the statistical checks. Both values are set to five for the evaluation. Next, the `extrapar = 0.01` setting initializes the α value for Alergia, meaning that a confidence level of $1 - 0.01 = 0.99$ is needed for the equivalence checking. The `largestblue = 1` setting only allows for the largest blue state to be considered for merging. Finally, the `extend = 0` setting ensures that a blue state which cannot be merged is only converted to a red state if there are no other possible merges.

Bigram and Markovian
The bigram model and the Markovian model don't use sinks during the construction of the model, relying solely on the Markovian property. The difference between the two is that the bigram model uses bigrams (sequences of two events) over the training data, whereas the Markovian model does not combine any sequential events. For example, given the trace ⟨`a b c d`⟩, the bigram configuration uses the event sequence ⟨`(a b) (b c) (c d)`⟩ whereas the Markov configuration uses the event sequence ⟨`(a) (b) (c) (d)`⟩.

Convert Sinks and No Sink Merging
Dealing with the sink states can be done in multiple ways. Generally, once a state is marked as a sink in the prefix tree, all states reachable from the sink are removed as these states cannot be dealt with due to a lack of sufficient statistical certainty. As a result, the traces reaching such a sink state do not perfectly fit the final model. The convert sinks configuration solves this issue by adding self-loops for all events observed in the subtree rooted at the sink. This way, all training traces will fit perfectly in the resulting model at the cost of added complexity in the shape of infrequent self-loops.

The 'No sink merge' configuration disallows the merging of sinks when constructing the model. When sink merges are allowed, the frequency of the resulting state can cross the infrequency threshold and can therefore be considered a normal state. As a result, the merged state is regarded as a 'normal' state for the remainder of the state merging process, even though the original sinks were considered too infrequent.

Original
The original configuration is the configuration used in the current method from Nadeem et al.[31]. This configuration both uses the Markovian property and has sinks enabled, but it differs from the other configurations for other properties. The first difference is that the threshold for the statistical tests is lowered from 0.99 to 0.95 by increasing the `extrapar` setting. Second, the lower bound for the heuristics function is increased from 0 to 3 through the `lowerbound` setting. By increasing this bound, the merging process only performs better merges at the cost of not being able to merge as many states.

Search
Finally, the search configuration uses a searching strategy to find the optimal merging of states instead of the Alergia state merging used in the other configurations. In this searching process, sinks are also considered for merging. Furthermore, the search allows for sink states to be merged into already identified sink states.

3.3. Evaluating Models
With the model construction covered, we still need some method to quantify how good the models are. This section discusses the metrics used to measure the performance and complexity of the models for both the process models as well as the state machines.

3.3.1. Metrics for Process Models

The metrics used for evaluating the performance of the miners and the models are chosen based on their identified strengths, weaknesses, and the suspected limitations of datasets. Section 2.3 contains more background information on the metrics chosen, as well as their alternatives. The code used for evaluation is based on the implementation from the 2018 process mining benchmark [10], which is available on Github[3].

Performance

The base requirement for the constructed process models is that they must be sound. An unsound model could contain disconnected parts or lead to deadlocks in proper execution and are therefore not desired for evaluation by an expert. Furthermore, the metrics used for fitness and precision require model soundness to operate properly, thus allowing unsound models would require falling back to less advanced metrics for those aspects.

Fitness and precision are computed using the alignment-based variants due to their ability to deal with non-fitting traces. Costs for skipping or inserting transitions are set to one for visible transitions and zero for invisible transitions. All visible transitions have an equal cost of misalignment. Due to the high number of possible events resulting from the combinations of attack stage and targeted service, constructing a custom cost function would be too complicated. Invisible transitions have a misalignment cost of zero to prevent penalizing models with stricter synchronization or dependencies. The alignment automaton to compute precision is constructed using only one alignment per trace instead of all optimal alignments due to the high time requirement to find all optimal alignments. Using the single-alignment-based variant still provides an accurate estimate of the complete variant[5]. With the fitness and precision, the F-score is also computed.

Completeness is also computed as this kind of fitness is desired for the use-case of modeling attacker strategy. With this use-case, the focus lies more on constructing a model which fits all data as opposed to the '80% model', which is usually desired for gaining insights into business processes. In addition, completeness also gives insight into how the process models perform related to the state machine constructed in the original method. This state machine offers perfect fitness due to the learning algorithm used, and comparison based solely on alignment-based measures would therefore provide an unfair bias towards the process models.

Generalization is computed using 5-fold cross-validation with random splits. Five folds are used as this strikes a balance between maximizing the number of traces available for constructing the models and the time needed to evaluate all folds. Random splits are used as the CPTC datasets contain a high number of unique traces, meaning that each of the five folds will include some unique traces. This ability to generalize is quantified as the mean of the fitness, precision, and F-score.

Complexity

Complexity is measured using the size metrics of the number of (different types of) nodes, the number of edges, and the coefficient of connectivity (CNC), control flow complexity (CFC), and structuredness. First, the number of nodes, number of edges, and the CNC gives a high-level overview of the overall size of the constructed model. Combining the size with the CNC value gives an insight into the 'connectedness' of the nodes and allows for simple reasoning about the general structure of the model.

CFC captures the complexity of the model in terms of branching factor, and the amount of different gateways helps explain where the branching mainly occurs. In addition, the number of parallel split and join gateways in the model also gives an insight into the degree of parallelism captured.

Finally, the structuredness metric captures if different parts of the model are interconnected or if they tend to be more self-dependent. Here, the main reasoning is that when the model has more self-dependent parts, the task of understanding the full model can be split into smaller tasks of understanding parts of the model more easily.

All complexity metrics are computed over the BPMN model equivalent to that produced by the original mining algorithm. BPMN is used as it is simpler compared to Petri nets, and most common process model types (Petri nets and Process Trees) can easily be converted to BPMN. The full set of traces is used to compute the complexity as the model constructed using all traces holds the most information of the underlying dataset, which is most valuable for an analyst.

[3] https://github.com/tudelft-cda-lab/Process-Mining-Evaluation

3.3.2. Evaluating State Machines

Since state machines aren't Petri nets, the evaluation of the process models cannot be used directly for the state machines. Fortunately, we can still use the same evaluation process by first converting the state machines to an equivalent Petri net or BPMN model. Through these conversions, it is possible to evaluate the state machines with the same metrics as the process models.

Conversion to a Petri net

Converting a State Machine to a Petri Net is a trivial process. The conversion starts with creating a unique start and end place. Then, the loop in line 3 creates a place for each state and connects all accepting states to the end place through a silent transition. Once a place exists for each state, the start place is connected to the place for q_0 through a silent transition, which is done to ensure the start place does not have any incoming arcs. Finally, the transitions in the state machine are converted in the loop at line 8 by creating a labeled transition and connecting it to the places of the corresponding states. Figure 3.1b shows a graphical representation of this conversion.

Algorithm 1: Conversion of a state machhine to a Petri net

Data: State machine $(Q, \Sigma, \delta, q_0, F)$
Result: Petri net
1 create place p_{start}
2 create place p_{end}
3 **for** each state $q \in Q$ **do**
4 | add a place p_q to P
5 | **if** $q \in F$ **then**
6 | | connect p_q to p_{end} through a silent transition
7 connect p_{start} with p_{q_0} through a silent transition
8 **for** each transition $(q, s, q') \in \delta$ **do**
9 | add transition t with label s
10 | add an arc from p_q to t
11 | add an arc from p to $p_{q'}$

Through this conversion, all valid state machines can be converted to a Petri net. In this net resulting from the conversion, all transitions have a single incoming arc and a single outgoing arc. Therefore, only one token can exist at any time during execution. As all states in the state machine are reachable from the initial state, the token can reach any state, and all transitions can become enabled. Therefore, as long as all states in the state machine that do not have an outgoing transition are accepting states (and thus, the corresponding place in the PN is connected to the final place through a silent transition), the Petri net is sound. In addition, it is a workflow net as there is a singular initial place and a single final place.

Conversion to BPMN

The conversion from a state machine to a BPMN model is similar to the conversion to a Petri net. Algorithm 2 shows the steps of this conversion process. Here, the main idea is that each state is replaced by an exclusive-choice join node merging all incoming edges and an exclusive-choice split node to handle all outgoing edges, which is achieved by the loop at line 3. Then, the loop at line 9 creates an activity for each transition in the state machine connecting it to the exclusive-choice gateways of the corresponding source and target states. Finally, the loop at line 16 simplifies the model by removing all trivial gateways.

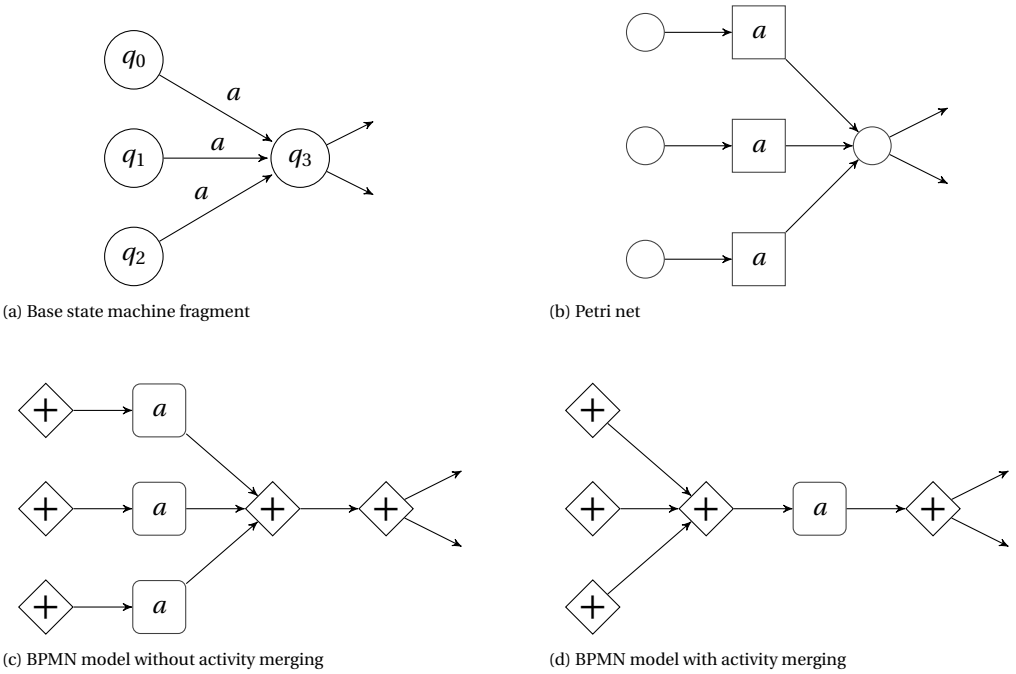
When a state has multiple incoming transitions with the same label (which always occurs due to the Markovian property in the configurations), the corresponding activities can be merged to reduce the overall number of nodes in the resulting model. The if-statement at line 10 deals with merging such activities. Figure 3.1 shows a graphical representation of the conversion with and without activity merging.

State Machine-specific complexity

The conversion from a state machine to a BPMN model introduces some overhead which negatively impacts the complexity of the state machines. To deal with this introduced negative bias, complexity metrics are also computed for the state machines themselves. Besides, the argumentation from [13] indicates that the state

Algorithm 2: Conversion of a state machine to a BPMN diagram

Data: State machine $(Q, \Sigma, \delta, q_0, F)$
Result: BPMN diagram
create nodes $start$ and end
create exclusive-choice gateway $join_{end}$ with an outgoing arc to end
for each state $q \in Q$ **do**
  add exclusive-choice gateways in_q and out_q
  add an arc from in_q to out_q
  if $q \in F$ **then**
    add an arc from out_q to $join_{end}$
add an arc from $start$ to in_{q_0}
for each transition $(q, s, q') \in \delta$ **do**
  if activity $a_{q',s}$ does not exist **then**
    add activity $a_{q',s}$ labelled with s
    add exclusive-choice gateway $xor_{q',s}$
    add an arc from $xor_{q',s}$ to $a_{q',s}$
    add an arc from $a_{q',s}$ to $in_{q'}$
  add an arc from out_q to $xor_{q',s}$
for each exclusive-choice gateway g where $|\bullet g| = |g \bullet| = 1$ **do**
  remove g together with the incoming and outgoing arc
  add an arc from $\bullet g$ to $g\bullet$;



(a) Base state machine fragment

(b) Petri net

(c) BPMN model without activity merging

(d) BPMN model with activity merging

Figure 3.1: Visual representation of the conversion from a state machine to a BPMN model.

machines are also interpretable by themselves, so it does not make sense to measure complexity solely based on the converted BPMN model.

Looking at the metrics used for process models, the number of nodes, number of edges, CFC, and CNC can also be computed for the state machines. First, the number of nodes is equal to the number of states, and the number of edges is equal to the number of transitions. Then, the CNC can be computed as the ratio between these two values. Control flow complexity is defined as the sum of the out-degree of all states with two or more outgoing edges. States with zero or one outgoing transition are not considered as those states do not introduce any choice in the model.

4

Performance of Process Mining

Using the experimental setup as described in Chapter 3, the different mining algorithms have been evaluated on the traces generated from the CPTC'17 and CPTC'18 datasets. This chapter begins with an overview and discussion on the quality of the produced models based on the performance and complexity metrics. Following this is a deeper dive into the more promising models based on the metric scores.

4.1. Model Performance

The first topic of the analysis of the models is the evaluation regarding performance. Table 4.1 shows the scores from the performance metrics for the different miners.

α\$, Fodina and Split Heuristics Miner

The most notable result is that the α\$, Fodina, and to a significant extent sHM6 cannot generate a sound model from the given data. For α\$, this could be explained by the traces not conforming to the notation of completeness. With the high number of unique events and relatively low number of traces in the datasets, combined with the possible unstructured nature of the underlying attack process, the lack of completeness was expected.

The Fodina and sHM6 miners are both based on the Heuristics Miner, which could explain why both the miners fail. Once again, the high number of events combined with the relatively low number of traces is likely to have distorted the dependency measure used as the basis in Heuristics mining. However, sHM6 can generate sound models in some cases, which can be explained by the soundness repair step in the mining process.

Inductive Miner

Due to the usage of process trees, all variants of the Inductive Miner were able to produce sound process models in all cases. However, issues arose during the evaluation of the models with implementation errors occurring during the computation of the alignments. This issue is interesting for the models produced by IM as all training traces are guaranteed to fit the modes perfectly, meaning that finding a perfect alignment in those cases does not require an extensive search for an optimal value. Even stronger, a custom replay tool for BPMN models (discussed later in this section) was able to find alignments for all perfectly fitting traces in a matter of seconds per trace for these models.

Regarding the performance of the miner with different noise thresholds, the general trends are as expected: increasing the noise threshold increases precision at the cost of lowering fitness and completeness. In all cases, this trade-off is well balances as shown by the pretty consistent F-score, especially for the reversed traces. Following the trend for precision through the different noise thresholds, it can be assumed that the precision of the model for the base IM lies around $0.42 - 0.45$ for the full dataset, resulting in an estimated F-score between 0.59 and 0.62 for the Inductive Miner without any noise filtering.

What is interesting is the drop in completeness compared to the fitness. Whereas raising the noise threshold introduces a slight drop in fitness, the completeness decreases a lot faster. This indicates that some form of infrequent behavior is present in the majority of the traces.

Dataset	Miner	Full data					Averages from 5-fold cross-validation				
		Soundness	Fitness	Completeness	Precision	F-score	# Sound	Fitness	Completeness	Precision	F-score
CPTC'17 (chronological)	α\$	unsound	-	-	-	-	0	-	-	-	-
	FO	unsound	-	-	-	-	0	-	-	-	-
	IM[1]	sound	(1.00)	(1.00)	ERR	ERR	5	ERR	ERR	ERR	ERR
	IMf-0.05	sound	ERR	-	ERR	-	5	0.94	0.77	0.34	0.49
	IMf-0.10	sound	ERR	-	ERR	-	5	0.90	0.65	0.30	0.45
	IMf-0.15[2]	sound	t/o	-	t/o	-	5	0.80	0.43	0.30	0.43
	IMf-0.20	sound	0.80	0.46	0.25	0.38	5	0.76	0.36	0.25	0.37
	sHM6	unsound	-	-	-	-	0	-	-	-	-
	SM	unsound	-	-	-	-	1	0.91	0.64	0.44	0.59
	FLOWER	sound	1.00	1.00	0.13	0.22	5	1.00	0.99	0.08	0.14
	PTM	sound	1.00	1.00	1.00	1.00	5	0.81	0.27	0.53	0.64
CPTC'17 (reversed)	α\$	unsound	-	-	-	-	0	-	-	-	-
	FO	unsound	-	-	-	-	0	-	-	-	-
	IM[1]	sound	(1.00)	(1.00)	ERR	ERR	5	ERR	ERR	ERR	ERR
	IMf-0.05	sound	0.97	0.89	0.46	0.63	5	0.95	0.81	0.35	0.51
	IMf-0.10	sound	0.89	0.61	0.50	0.64	5	0.89	0.59	0.38	0.53
	IMf-0.15	sound	0.81	0.41	0.52	0.63	5	0.79	0.39	0.42	0.55
	IMf-0.20	sound	0.72	0.20	0.59	0.65	5	0.71	0.19	0.43	0.54
	sHM6	unsound	-	-	-	-	2	t/o	t/o	t/o	-
	SM	sound	0.93	0.68	0.73	0.82	3	0.89	0.60	0.53	0.67
	FLOWER	sound	1.00	1.00	0.14	0.25	5	1.00	0.99	0.09	0.17
	PTM[3]	sound	1.00	0.99	1.00	1.00	5	0.81	0.26	0.53	0.64
CPTC'18 (chronological)	α\$	unsound	-	-	-	-	0	-	-	-	-
	FO	unsound	-	-	-	-	0	-	-	-	-
	IM[1]	sound	(1.00)	(1.00)	ERR	ERR	5	ERR	ERR	ERR	ERR
	IMf-0.05	sound	0.96	0.82	0.30	0.46	5	0.94	0.76	0.31	0.47
	IMf-0.10	sound	0.90	0.60	0.45	0.60	5	0.89	0.57	0.34	0.49
	IMf-0.15	sound	0.87	0.46	0.25	0.39	5	0.86	0.46	0.22	0.35
	IMf-0.20	sound	0.80	0.25	0.31	0.45	5	0.80	0.27	0.25	0.39
	sHM6	-[4]	-	-	-	-	0	-	-	-	-
	SM	sound	0.94	0.74	0.52	0.67	5	0.91	0.67	0.41	0.56
	FLOWER	sound	1.00	1.00	0.07	0.12	5	0.99	0.97	0.04	0.08
	PTM	sound	1.00	1.00	1.00	1.00	5	0.84	0.42	0.55	0.66

Results for CPTC'18 reversed are on the next page

Table 4.1 – continued from previous page

Dataset	Miner	Full data					Averages from 5-fold cross-validation				
		Soundness	Fitness	Completeness	Precision	F-score	# Sound	Fitness	Completeness	Precision	F-score
CPTC'18 (reversed)	α\$	unsound	-	-	-	-	0	-	-	-	-
	FO	unsound	-	-	-	-	0	-	-	-	-
	IM[1]	sound	(1.00)	(1.00)	ERR	ERR	5	ERR	ERR	ERR	ERR
	IMf-0.05	sound	0.96	0.77	0.47	0.63	5	0.94	0.74	0.38	0.54
	IMf-0.10	sound	0.95	0.69	0.50	0.65	5	0.93	0.69	0.37	0.53
	IMf-0.15	sound	0.94	0.67	0.50	0.66	5	0.88	0.53	0.37	0.52
	IMf-0.20	sound	0.75	0.22	0.57	0.65	5	0.73	0.15	0.43	0.54
	sHM6	sound	0.70	0.03	t/o	-	3	0.68	0.02	0.24	0.35
	SM	sound	0.94	0.74	0.63	0.76	5	0.91	0.64	0.48	0.63
	FLOWER	sound	1.00	1.00	0.07	0.14	5	0.99	0.98	0.04	0.08
	PTM[3]	sound	0.99	0.95	1.00	1.00	5	0.84	0.44	0.50	0.63

Table 4.1: Performance metrics for different mining algorithms generated on the traces of CPTC'18.
[1] Computing the alignments for the models from IM produced large numbers of implementation errors from external code sources, so none of the obtained results are reliable. The only measures we know for sure are the perfect fitness guarantees.
[2] Evaluation did not produce reliable results for one of the five folds for IMf-0.15, hence values are computed over four evaluations.
[3] The prefix-tree model is ensured to have perfect fitness and completeness, so non-perfect scores indicate a minor issue in either the model construction or the evaluation method. Given that for both cases, only one trace did not fit, the issue was not investigated further.
[4] sHM6 could not produce a model within the four hours allowed for mining.

Split Miner
From all the mining algorithms evaluated, the Split Miner provided the best balanced results based on F-score across the board. The fitness of the models is generally comparable to the Inductive Miner with a noise threshold between 0.05 and 0.10 , but the Split Miner achieves a higher precision. However, the completeness of the model does tend to be slightly lower than the completeness of the Inductive Miners with comparable fitness.

One deviating result from the evaluation is the fact that the Split Miner was not able to construct a sound model for the chronological traces of CPTC'17 and two of the folds for the reversed traces of CPTC'17. The miner guarantees soundness for acyclic processes, but such cyclic behavior is likely not present in this dataset as the only loops identified by the Inductive Miner consist of only one task. Hence, this lack of soundness may indicate an issue with the implementation of the miner.

Flower Model and Prefix Tree
The flower model gives as expected high fitness, both when the model is constructed using the full dataset as well as with the cross-validation experiments. We see some slight drop in fitness and completeness with the cross-validation experiments, which occurs when a task from the evaluation dataset is not present in the training data. The precision of the model is by far the lowest, which was to be expected as any move is possible for any prefix. This low precision also causes the flower miner to score the lowest in F-score.

The prefix tree model gave the highest precision, which was expected given the complete lack of generalization in the model. However, based on the cross-validation experiments, the model's fitness, precision, and F-score are still among the best compared to all the other miners. It is only in completeness where the prefix tree model shows significantly worse results than the other models.

Validity of precision
The high F-score is mainly a result of the high precision scores, indicating that there are some issues with how this metric is computed. Following the workings of the alignment-based precision, the high precision scores can be explained by skips in the trace for the computed alignments. These skips do reduce the fitness score of the trace, but they have no impact on the precision as skips in the trace do not yield a new state in the model. Furthermore, given the nature of the prefix tree, non-fitting traces are likely to result in skips in the trace for non-fitting traces.

For example, a model only allowing the sequential trace $\langle$`a b c`$\rangle$ has a precision score of 1.00 when evaluated with the trace $\langle$`a b c d e f`$\rangle$ as all possible paths have been taken for all possible prefixes. In this case, the fitness is 0.647, resulting in an F-score of 0.80, which is pretty high given the model cannot explain half of the evaluation trace.

Generalization and the impact of cross-validation
Comparing the results of the full dataset and the cross-validation, we see that the scores for fitness and completeness hardly changed, indicating that the models actually do generalize pretty well. We see a slight drop in precision, but this can be explained by the evaluation set being only one-fifth in size of the full dataset, which causes fewer transitions to be covered during the evaluation.

Difference in Datasets and the effects of reversing traces
Looking at the differences between the scores for the two datasets, it is clear that the traces generated from CPTC'17 are generally harder to model than the traces stemming from CPTC'18. Especially the chronological traces of CPTC'17 proved to be a challenge for all mining algorithms as sound models were rarely constructed. Reversing the CPTC'17 traces did seem to make the traces easier to model as indicated by more miners constructing a sound model. The Split Miner even achieved the highest observed F-score across all miner/dataset combinations on this dataset. For CPTC'18, and to some extent CPTC'17, reversing the traces mostly improved the precision of the constructed models while having a limited impact on the fitness and completeness. Hence, for the datasets used, there is less variation in previous events given the outcome compared to more variation in possible future events given currently observed events. The reasoning behind this could be that the low-severity attack steps are shared commonly between different high-severity steps.

A note on completeness
In the context of constructing attack graphs, the models need to be able to explain the traces they are based on. Table 4.2 shows a more in-depth analysis of the completeness metric as discussed in Section 4.1. Opposed to the results shown in Table 4.1 which are computed with an external benchmark tool, the results in Table 4.2 are obtained using a custom-built replay tool that only checks completeness. This tool is avail able on Github[1].

In the table we see that the actual completeness of all models is slightly higher compared to the results found with the external tool. Also, the completeness is higher when computed over overall traces compared to just the unique traces. This was to be expected given that during the modeling process, a trace occurring twenty times should have a higher weight compared to a trace that occurs just once. Following this logic, the difference in performance on all traces compared to just the unique traces shows that the infrequent traces fit the models less.

This lack of fitness is an issue for the case of constructing attack graphs due to the focus on infrequent traces, given those traces contain unique attacker behavior. By modeling this unique behavior, an analyst might be able to identify the more unknown vulnerabilities that are exploited. Following the completeness requirement, only the models produced by Inductive Miner with low noise thresholds and the Split Miner can be considered viable.

Dataset		CPTC'17		CPTC'18	
		Chronological	Reversed	Chronological	Reversed
Size	Total	965	965	536	536
	Unique	777	777	327	327
IM	Total	1.00	1.00	1.00	1.00
	Unique	1.00	1.00	1.00	1.00
IMf-05	Total	0.83	0.89	0.82	0.56[1]
	Unique	0.79	0.88	0.76	0.53[1]
IMf-10	Total	0.72	0.61	0.60	0.69
	Unique	0.68	0.55	0.48	0.57
IMf-15	Total	0.52	0.41	0.46	0.67
	Unique	0.47	0.35	0.31	0.52
IMf-20	Total	0.44	0.20	0.21	0.23
	Unique	0.39	0.16	0.20	0.25
SM	Total	-[2]	0.68	0.74	0.74
	Unique	-[2]	0.62	0.60	0.25

Table 4.2: Amount of traces perfectly fitting each model. The 'total' score takes trace frequency into account, whereas the 'unique' score assigns the same weight to each unique trace.
[1] The recursive replay method reached a limit for 28 unique traces, automatically labeling them as non-fitting. Because the maximum recursion depth was reached, it is safe to assume the traces did indeed not fit and that the results shown are correct.
[2] The constructed model was unsound, so no data could be computed

4.2. Model Complexity

Having a model which can explain the data is of no use if it cannot be interpreted. Table 4.3 shows the complexity metrics for the BPMN models constructed for the datasets using the miners capable of constructing sound models. Obtaining BPMN models for the process trees from the Inductive Miners is a trivial process as shown in Figure 2.8. With this conversion from Process Trees to BPMN models, the results have been simplified where possible by merging directly connected split or gateways of the same type. The Split Miner constructs a BPMN model by default, so no conversion is needed.

General Observations
All models constructed feature quite a significant number of nodes and edges. Following the computed values for CNC, the relation between the number nodes and the number of edges is a function of the miner with the

[1] https://github.com/tudelft-cda-lab/Process-Mining-Evaluation

Dataset	Miner	Total Nodes	Total Edges	Tasks	Parallel		Exclusive		CFC	CNC	Struct.
					Split	Join	Split	Join			
CPTC'17 (chronological)	IM	305	450	92	9	18	91	93	218	1.48	1.00
	IMf-05	265	386	91	11	15	72	74	181	1.46	1.00
	IMf-10	248	361	92	10	14	62	68	166	1.46	1.00
	IMf-15	234	344	92	8	13	59	60	156	1.47	1.00
	IMf-20	227	335	92	7	13	56	57	150	1.48	1.00
	SM[1]	-	-	-	-	-	-	-	-	-	-
CPTC'17 (reversed)	IM	305	450	92	18	9	93	91	229	1.48	1.00
	IMf-05	204	312	91	8	8	48	47	151	1.53	1.00
	IMf-10	245	358	91	13	13	64	62	171	1.46	1.00
	IMf-15	244	354	91	15	16	60	60	163	1.45	1.00
	IMf-20	217	317	91	12	12	49	51	146	1.46	1.00
	SM	242	447	92	0	0	60	88	266	1.84	0.58
CPTC'18 (chronological)	IM	347	507	112	14	12	102	105	242	1.46	1.00
	IMf-05	268	387	106	6	9	73	72	183	1.44	1.00
	IMf-10	233	333	104	4	5	59	59	157	1.43	1.00
	IMf-15	266	385	111	5	10	69	69	175	1.45	1.00
	IMf-20	277	397	112	9	14	69	71	177	1.43	1.00
	SM	245	395	112	0	0	53	78	204	1.62	0.62
CPTC'18 (reversed)	IM	349	510	112	13	15	105	102	244	1.46	1.00
	IMf-05	255	367	107	10	11	64	61	171	1.44	1.00
	IMf-10	246	355	107	9	9	60	59	165	1.44	1.00
	IMf-15	238	344	107	8	8	57	56	161	1.45	1.00
	IMf-20	266	380	109	12	11	65	67	175	1.43	1.00
	SM	236	382	112	0	0	46	76	193	1.62	0.61

Table 4.3: Size and Complexity metrics for the models constructed by the Inductive Miner with different noise thresholds and the Split Miner. The mismatch between the split and join gateways for the inductive models is caused by merging consecutive gateways of the same type.
[1] No sound model was constructed, so results have not been computed

Inductive Miner producing models with a CNC around 1.45 and the Split Miner creating models with a CNC around 1.60 and higher.

Doubling the CNC value shows that for the models from the Inductive Miner, each node has on average about 2.9 edges connected to it whereas nodes in the model from the Split Miner are connected by around 3.2 edges on average. This indicates that the models from the Inductive Miner tend more towards longer linear models, whereas the models constructed by the Split Miner feature more dependencies between different parts of the model.

Noise threshold for the Inductive Miner

Looking at the differences introduced by the noise threshold for the Inductive Miner, we see the expected trend that increasing the threshold leads to smaller models. This is reflected in both the total number of nodes and edges, as well as the CFC.

However, the number of tasks filtered out does not follow the general trend. With thresholds of 0.05 and 0.10, more tasks were filtered out compared to the thresholds of 0.15 and 0.20. Furthermore, more tasks were filtered for the reversed traces compared to the chronological traces, indicating that the structure within the dataset changes as the order of the event sequences is reversed.

Gateway Interplay

Comparing the number of split and join gateways of similar types, there is also often quite some discrepancy. For the models constructed by the Inductive Miner, this discrepancy is quite small and can be explained by the simplification applied to the models. This simplification merges consecutive gateways of the same type. Due to the nested nature of the underlying process tree, consecutive gateways of the same type are not uncommon. Structuredness is not impacted by this simplification as it is computed before the simplification step.

The models constructed by the Split Miner are a different case. This miner constructs a BPMN model by default, and no post-processing is done on the results, so the discrepancies are caused solely by the mining process. Given the mining process of the Split Miner, such a discrepancy can be expected to a certain extent as the splits and joins are identified separately from each other. This discrepancy does negatively impact how difficult it is to interpret the model as a split gateway cannot always be matched to a corresponding join gateway. As a result, it is harder to break the model down into smaller parts, as shown by the structuredness scores of about 0.60.

Parallelism?
For none of the datasets, the Split Miner deduced a parallel relationship between different nodes. On the other hand, the Inductive Miner identified multiple parallel relationships for each dataset, but these parallel relations might not be valid as discussed in Section 4.3.1. Hence, no definite conclusion can be made whether the attacking process features actions that are executed in parallel.

However, it might also be the case that the directly-follows relations from the datasets are not complete with respect to the underlying process. If this is the case, the models don't feature any meaningful parallelism simply due to the lack of evidence.

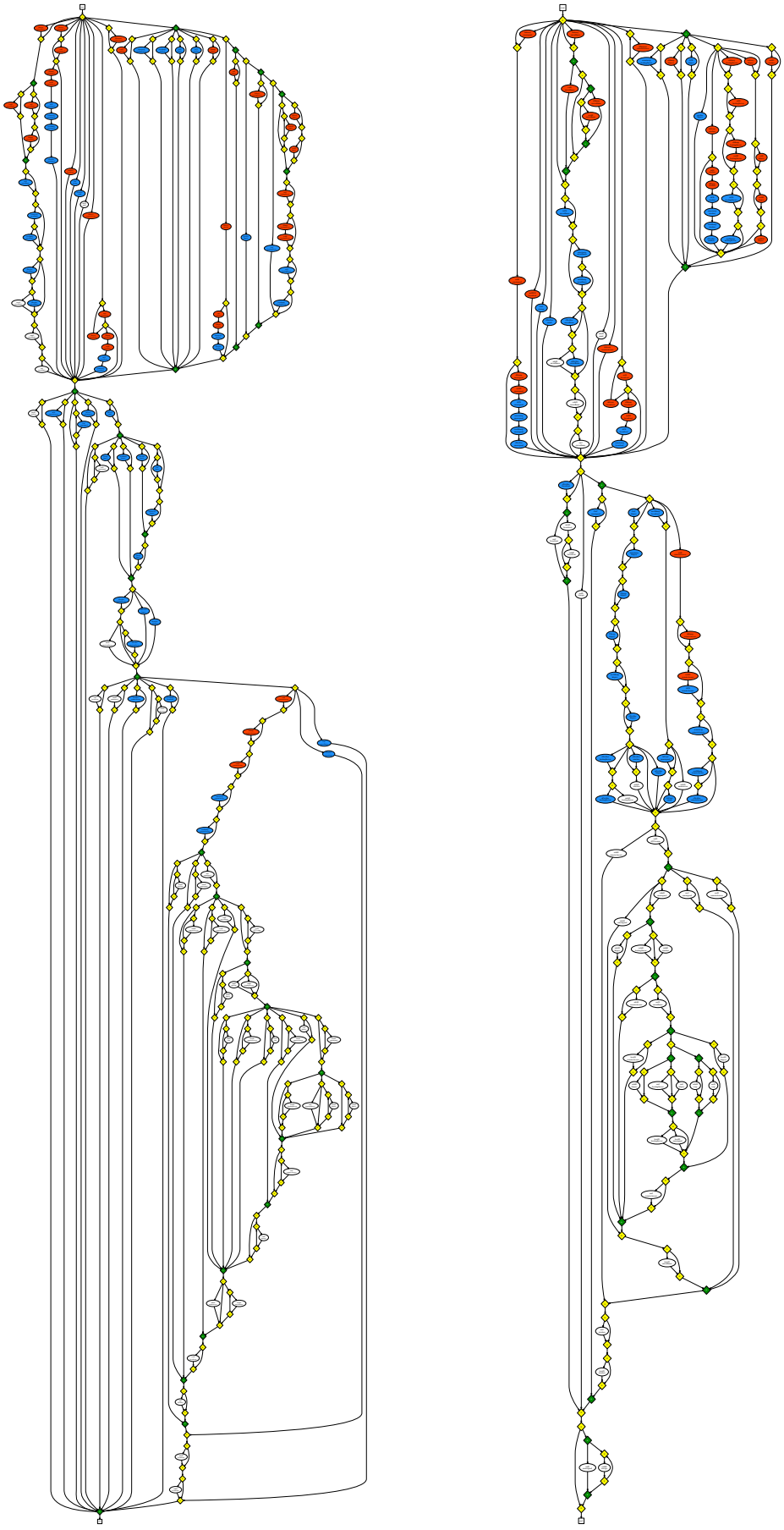
4.3. Inductive Miner Models

The full BPMN models produced by IM and IMf-20 for the reversed traces of CPTC'18 are shown in Figure 4.1. In these models, tasks are denoted as circular nodes colored red for high-severity attack stages, blue for medium-severity attack stages, and white for low-severity attack stages. Exclusive split and join gateways are shown with yellow diamonds, and parallel split and join gateways as green diamonds. As the main focus of the models is completeness, this section mainly focuses on the model constructed without any noise filtering.

General Structure
Looking at the overall structure of the models, we can see hints of the underlying tree shape of the process trees. The tasks are clearly surrounded by single-entry single-exit blocks, making up larger single-entry single-exit structures. Comparing the two models, the model constructed using the 0.20 noise threshold is more compact, containing fewer edges skipping large parts of the model than the base Inductive Miner model. This visual difference confirms the conclusions from the complexity scores for the two models.

Partitioned model
One of the notable features of both models is the convergence to a single edge at around one-third of the way down. Above this partition, almost all tasks for high-severity alerts and around half of the tasks for medium-severity alerts are placed. Given the traces are constructed such that the high-severity alerts are placed at the beginning of the trace, it was expected that most of these events to be modeled towards the top of the model. However, following the convergence in the models after this initial section, the models imply that these higher-severity events cannot explain which lower-severity events occurred earlier in the trace.

Such behavior is not unexpected from the Inductive Miner, given it does not use any rules to detect implicit relations between different tasks. In addition, the process trees constructed by the miner also model the process in isolated blocks where each block has a single entry and exit point, preventing the addition of arcs enforcing implicit dependencies. Furthermore, the additional absence of methods for detecting duplicate tasks prevents the Inductive Miner from modeling these implicit relations through different branches. Therefore, the partition indicates that any occurrence of a task before the partition is always observed before any occurrence of a task after the partition.



(a) IM (b) IMf-0.20

Figure 4.1: Full models constructed by the inductive miners on the reversed traces of CPTC'18. Tasks are shown as circular nodes colored red for high-severity attack stages, blue for medium-severity attack stages, and white for low-severity attack stages. Exclusive-choice split and join gateways are shown with yellow diamonds, and parallel split and join gateways as green diamonds. The full-scale version is available on https://github.com/tudelft-cda-lab/Process-Mining-Evaluation

Optional Tasks

Looking closer at the models, we see a specific pattern occurring somewhat frequently. Figure 4.2 shows a close-up of the parallel split gateway in the top-right of the model from the Inductive Miner without noise filtering. The first thing to notice here is that the six tasks are surrounded by an exclusive-choice construct, making them entirely optional. Of the 112 tasks in the model produced by IM, 54 tasks are surrounded by such a take-or-skip structure, and 23 tasks are placed in a loop structure where they can occur zero or more times. As a result of this abundance of optional paths in the model, the empty trace is considered valid, and the trace with a single event is valid for 90 out of the 112 events.

For the model with the highest noise threshold (IMf-20), 60 out of the 109 tasks are in a take-or-skip structure, and no tasks are in a zero-or-more loop. This model also allows for the empty trace, but 'only' 68 single-event traces are valid.

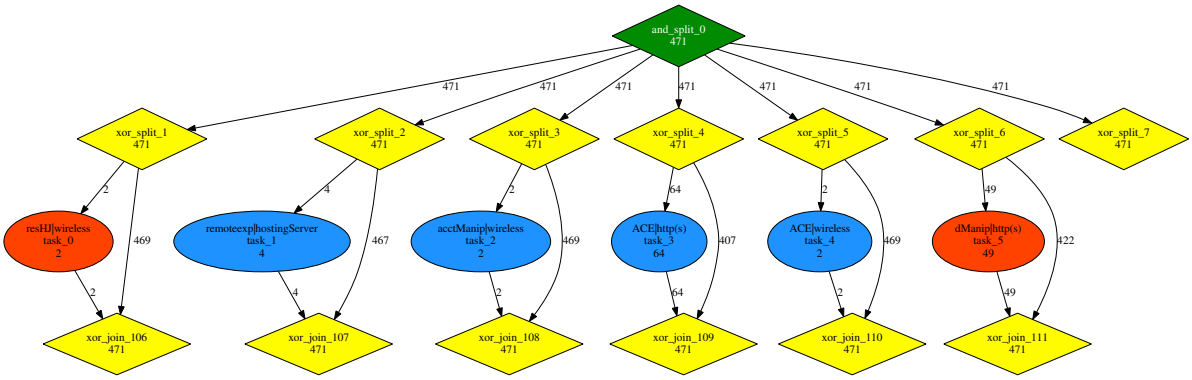


Figure 4.2: Snapshot of the model created by IM on the reversed traces for CPTC'18. Parallel gateways are shown in green, exclusive gateways in yellow, high-severity tasks in red and medium-severity tasks in blue. The numbers for each node or edge represent how often the node/edge was traversed when replaying all 536 traces.

4.3.1. Model Validity

The parallel split node from Figure 4.2 has seven child nodes: six optional tasks, which are all directly linked to the parallel join gateway, and the node `xor_split_7` which is the root of a larger sub-model containing 17 tasks. Through this parallel split gateway, the model implies that the six tasks can occur in any order, completely independent of both each other and any tasks in the branch rooted at `xor_split_7`. Furthermore, it gives no requirement for tasks occurring together in the same trace, nor does it provide any requirements regarding any ordering between the tasks. However, when looking at the traces, this independence is not valid.

Take the tasks `resHJ`, `ACE`, and `acctManip` for `wireless`, which all occurred twice in the entire dataset. These three events are all actually part of the same trace, which also occurred twice. From this, we know that the data implies a strong sequential ordering between these three tasks. Inspecting this trace even further shows that the implications made by the model are even less valid. The full trace features the sub-sequence:

```
...exfil|wireless dManip|wireless resHJ|wireless ACE|wireless
 remoteexp|wireless acctManip|wireless rPrivEsc|wireless...
```

All seven tasks in this sub-sequence occur exclusively in this trace, so it is expected that the miner would have been able to identify this strong sequential relation. Figure 4.3 shows the model for the other four tasks in the sub-sequence. This sub-model is part of the model rooted in `xor_split_7`, meaning it is executed in parallel to the other tasks from the sequence. In this sub-model, the miner did correctly find a causal ordering between the four tasks, raising the obvious question as to why the miner did not identify the sequence as a whole.

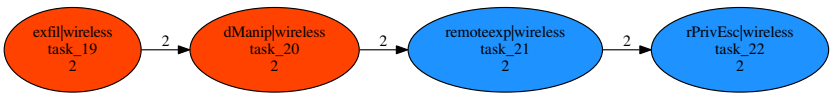


Figure 4.3: Sub-sequence observed in the model constructed by IM

More strange modeling choices are made for the other tasks in the parallel split. To begin, the task `remoteexp|hostingServer` has a strong relation to the task `acctManip|hostingServer`. All three occurrences of the `acctManip` task are directly preceded by `remoteexp|hostingServer`, again showing a clear sequential ordering. Following the traces, it is expected that the `acctManip` task would be placed in a take-or-skip structure directly after the `remoteexp` task, but once again, it is placed somewhere in the sub-model rooted at `xor_split_7`.

Furthermore, an ordered relation also exists between the task `dManip|http(s)` and `ACE|http(s)`. The two tasks occur 44 times together in the same trace, and in all of these occurrences, `dManip` occurs before `ACE`. Therefore, it would be logical to enforce such an ordering between the two tasks in the model. However, in this case, the independence between the two tasks can be somewhat accepted given that the `ACE` task occurs 20 times without `dManip`, and `dManip` occurs five times without `ACE`.

All these patterns are not unique to the model from the Inductive Miner on the reversed traces of CPTC'18. Similar patterns are also present in the models constructed using the other noise thresholds, the chronological traces of CPTC'18, and both versions of CPTC'17.

The sequence in which the Inductive Miner identifies the cuts doesn't explain these patterns in the model. First, the exclusive-choice cut is checked, requiring that the different tasks are not connected in the directly-follows graph. Such a cut cannot be made such that these seven tasks are connected into one long sequence. Second, the miner checks for a sequential cut, requiring the different partitions to have a uni-directional relation in the directly-follows graph. Such a relationship is clearly present in the first two cases, and to a slightly lesser extent also in the third case. However, this cut cannot be chosen due to additional edges in the directly-follows graph. The parallel cut (which is chosen to split the tasks and identify `and_split_0`) requires the different partitions to be fully connected. Given that the seven tasks do not occur outside the sub-sequence, this is also clearly not the case.

After contacting one of the authors of the Inductive Miner, it became clear that the miner uses a different strategy when none of the cuts is possible. Following the original paper [23], the miner adds a flower model in such cases, but an improved version introduced in [18] introduces alternative strategies for when no cuts are valid. One of these 'fallthrough' strategies is the `ActivityConcurrent` fallthrough, where a random activity is removed from the log. This activity is then placed parallel to the rest of the block, and the miner continues on the sub-log with the selected activity removed. Following Figure 4.2, the tasks shown directly under the parallel split node were removed from the sub-model rooted at the `xor_split_7` and placed in parallel to the sub-model. Table 4.4 shows how many tasks are placed as optional directly between a parallel split and join node, indicating the fallthrough method has been applied for these tasks. In most cases, the number of tasks modeled this way lies between 15% and 35%, which is too high for the model to be considered valid. Of course, it is possible that some of these occurrences are a result of actual parallelism, but these cases cannot be distinguished from the model itself.

Miner	CPTC'17 chronological		CPTC'17 reversed		CPTC'18 chronological		CPTC'18 reversed	
	Tasks	Concurrent	Tasks	Concurrent	Tasks	Concurrent	Tasks	Concurrent
IM	92	32 (0.35)	92	32 (0.35)	112	36 (0.32)	112	37 (0.33)
IMf-05	91	32 (0.35)	91	15 (0.16)	106	19 (0.18)	107	15 (0.14)
IMf-10	92	24 (0.26)	91	23 (0.25)	104	8 (0.08)	107	14 (0.13)
IMf-15	92	23 (0.25)	91	26 (0.29)	111	18 (0.16)	107	12 (0.11)
IMf-20	92	22 (0.24)	91	19 (0.21)	112	22 (0.20)	109	17 (0.16)

Table 4.4: Number of tasks likely modelled through the `ActivityConcurrent` fallthrough for the Inductive Miner with different noise thresholds.

When using the `ActivityConcurrent` fallthrough instead of inserting a flower sub-model, the resulting model scores better based on precision without sacrificing anything on fitness. However, the implication made by a model using this approach is that there exists proof that the isolated task actually occurs in parallel to the rest of the block, giving off false signals for people unaware of this feature. Besides, this method yields inconsistent results, as be discussed in the following section.

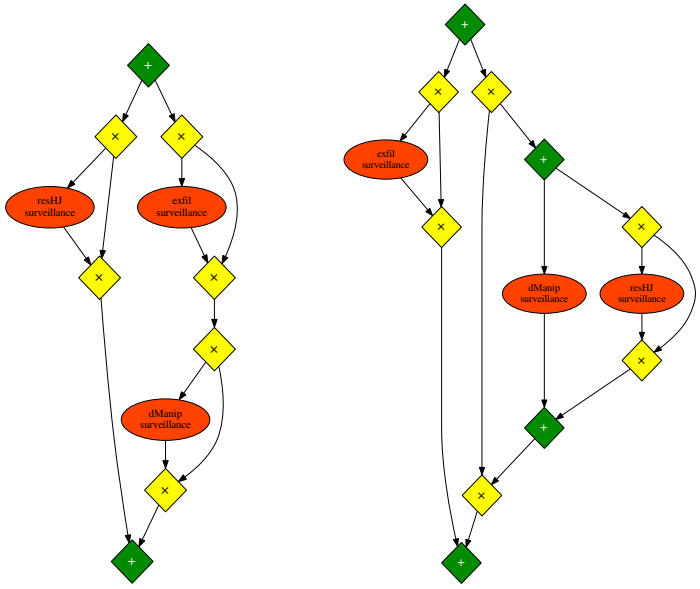
4.3.2. Model Robustness

In order to determine the effects of the `ActivityConcurrent` fallthrough, we can validate whether the choices made by the miner are at least consistent. For this, two validation methods are used: splitting the traces based on the partition in the full model and injecting dependencies based on the defined cuts for the Inductive Miner.

Splitting Traces

The first validation method uses the partition in the model. Through this partition, the model implies no dependency exists between the tasks on either side. Therefore, when we split the traces based on the events occurring before and after the partition in the model. By constructing a model on the two new datasets, we expect two new models that are both equivalent to their counterpart in the base model.

Figure 4.4 shows a snapshot from the top part of the base and its counterpart in the model generated over the split traces. Here, we see that the clear, ordered relation between `exfil` and `dManip` is lost in the new model. Furthermore, the new model enforces that every time the `resHJ` task is in a trace, `dManip` must also occur, whereas the original model does not enforce such a constraint.

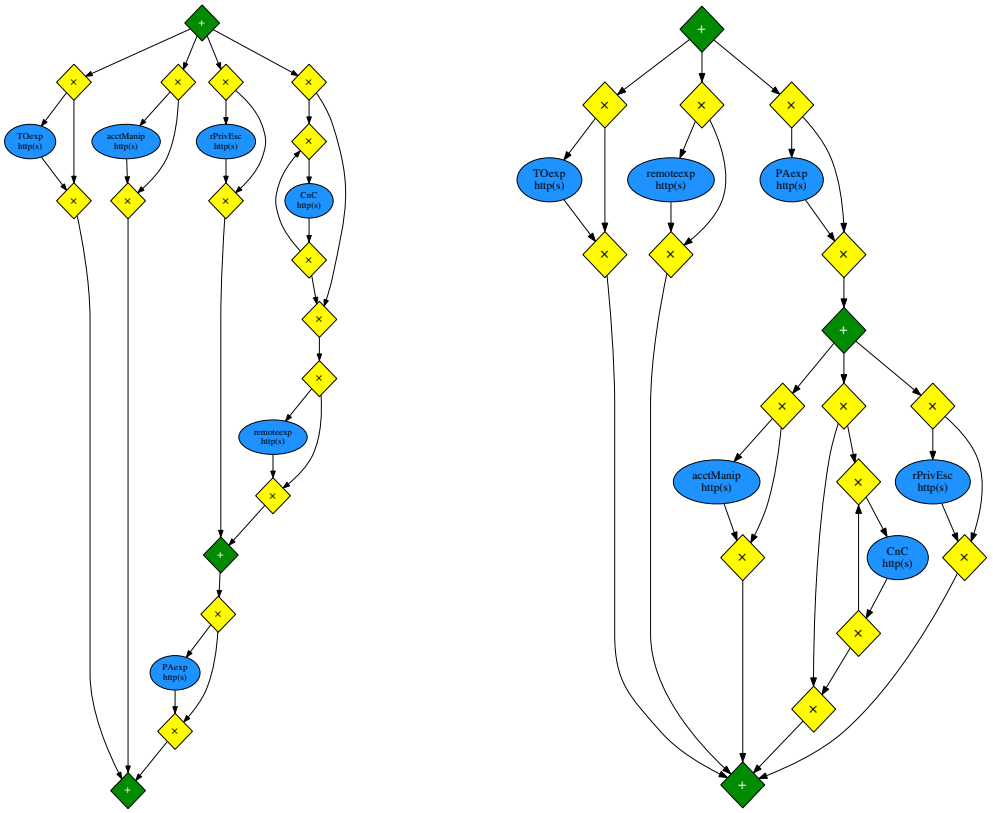


(a) Base model (b) Split Model

Figure 4.4: Difference in structure in the upper part of the model introduced by splitting the traces. The left sub-model shows a structure in the base model. The right sub-model same structure in the split model

Similarly, Figure 4.5 shows a snapshot from the bottom part of the base model and its counterpart in the split model. Here, the relations of `TOexp` and `remoteexp` relative to the other four tasks are unchanged. However, the ordering relation for the other four tasks has been altered quite significantly. Whereas `PAexp` was placed before `acctManip`, `CnC` and `rPrivEsc` in the original model, the model constructed over the split traces places that this task after those three other tasks. Furthermore, an ordering relation is suddenly defined between `CnC` and `acctManip`, whereas their order was independent in the base model.

These two examples are not the only discrepancies between the produced models, indicating that the implementation of the Inductive Miner cannot reliably reproduce results. This also questions the meaning of the defined relations: both models accept all the traces, yet both models show deviating relations between different tasks.



(a) Base Model

(b) Split Model

Figure 4.5: Difference in structure in the lower part of the model introduced by splitting the traces. The left sub-model shows a structure in the base model. The right sub-model same structure in the split model

Injecting dependencies

An alternative method of checking the validity of the Inductive Miner and the `ActivityConcurrent` fallthrough is by injecting artificial dependencies in the training data based on the four different cuts defined by the Inductive Miner. By altering the training data, we can insert structures in the traces that satisfy the different cuts and verify whether the miner correctly models the structures. When these structures are inserted for only one task, the Inductive Miner should only alter the structure around that task and leave the other parts of the model unchanged.

Given the base trace ⟨`a b c`⟩, the traces can be altered to introduce the cut-dependencies for task `b` as follows:

1. Exclusive-choice: add the additional trace ⟨`a b_1 c`⟩
2. Parallel: replace the original trace with the traces ⟨`a b_1 b_2 c`⟩ and ⟨`a b_2 b_1 c`⟩
3. Sequential: replace the original trace with the trace ⟨`a b_1 b_2 c`⟩
4. Loop: add the trace ⟨`a b b c`⟩

More information about the different cuts and their corresponding dependency relations is given in Section 2.4.2.

To strengthen the case for the different cuts, more artificial tasks can be injected which satisfy the dependency. Furthermore, the alterations can be made for multiple tasks in the dataset. This experiment uses the reversed traces from CPTC'18 with the six tasks from Figure 4.2. The four different dependencies are introduced for each of the six tasks by inserting one through five artificial tasks into the traces.

Loops

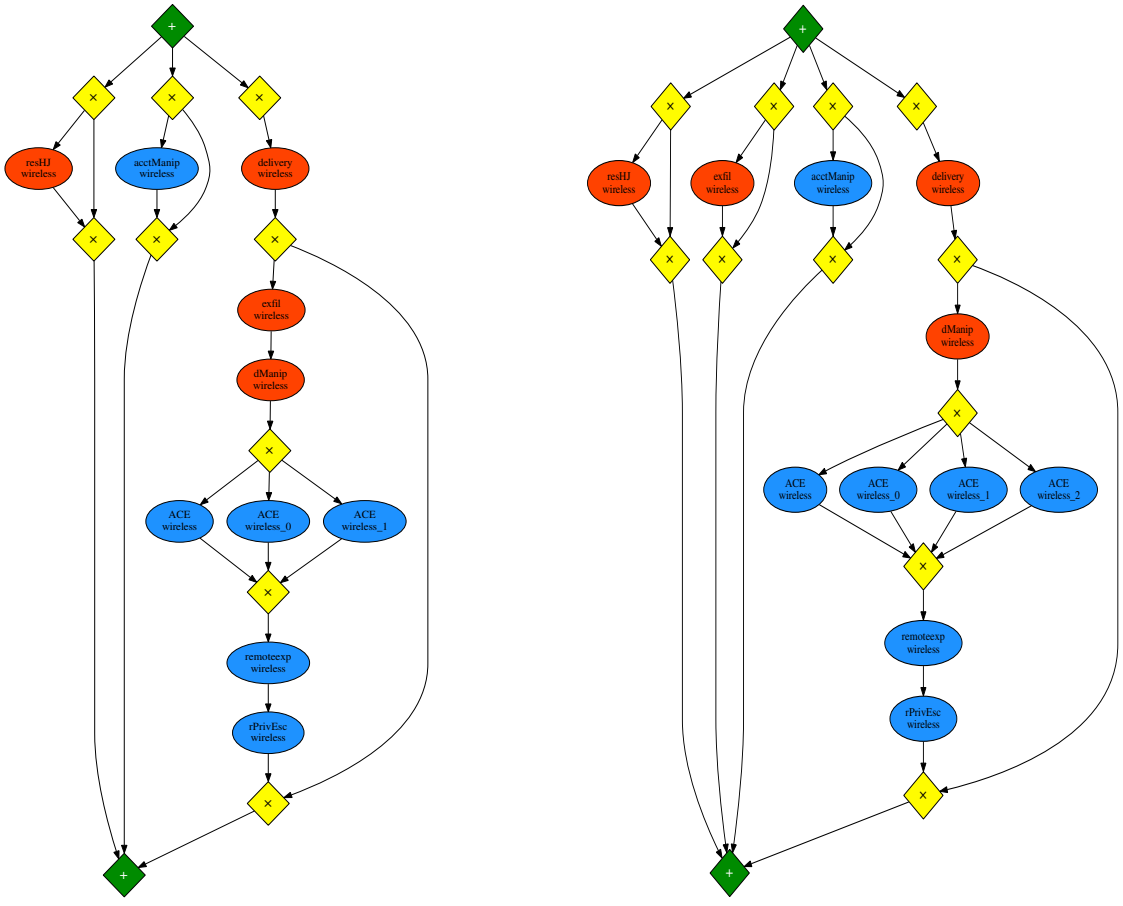
The injected loops were all identified as expected, where the original task was surrounded with an optional

loop-back edge. Some side-effects still occurred in the form of deviations in other parts of the model were present, similar to those observed when splitting the traces.

Exclusive-choice and parallelism

Exclusive-choice and parallel structures were also modeled as expected in the sense that all inserted tasks were encapsulated between a split and a join gateway of the corresponding type. However, the task's location in the model was changed in all cases, moving the new dependency to a more logical place in the model. The new structures for `wireless`-tasks were placed in the sequence from Figure 4.3. Altering the `remoteexp|hostingServer` task resulted in the new block being placed directly before the `acctManip` task on the same service. Changes for the `ACE|http(s)` and `dManip|http(s)` tasks resulted in the tasks being moved to the sub-model rooted at `xor_split_7`. Once again, other unrelated parts of the model also changed despite not being affected by the changes in the traces.

For the `wireless` tasks, the affected task was always inserted in the logical place in the full sequence if two or fewer artificial events were inserted. When four or five artificial events were inserted, `resHJ|wireless` was moved to the parallel structure. With three artificial tasks inserted, such a move did sometimes also occur, but not always. These effects are shown in Figure 4.6. The models show the relation between the different `wireless` tasks where two artificial events are inserted for the left model and three for the right model. On the left, the exclusive-choice block between the different `ACE|wireless` tasks is placed in the correct place in the sequence, but on the right, `resHJ|wireless` is moved to the higher-level parallel split.



(a) Two artificial tasks

(b) Three artificial tasks

Figure 4.6: Sub-models for wireless tasks after inserting artificial events for `ACE|wireless`. Two artificial tasks are inserted for the let model, and three artificial tasks are inserted for the right model.

The `ActivityConcurrent` fallthrough why the unaffected tasks were moved in the model as those tasks might have been randomly chosen by the fallthrough method. However, if this method picks a task at random to place parallel to the rest of the model, we would expect that some of the artificial tasks will be picked at

some point. However, with the experiments with artificial parallel or exclusive-choice tasks, this did not happen, indicating that there is yet more undocumented behavior around the method. At the moment of writing this thesis, the author of the Inductive Miner has not commented on this hypothesis.

Sequential dependencies
The Inductive Miner was not able to correctly identify the injected sequential dependencies, which is likely also caused by the `ActivityConcurrent` fallthrough. For all combinations of the base task and the number of artificial tasks, some tasks from the sequence were placed elsewhere in the model. Besides, in almost all cases, changes were made in unrelated parts of the model similar to those observed with splitting the traces.

For the three `wireless` tasks, the common pattern was that the base task remains in the same place at the high-level parallel split, but the artificial tasks are placed in the sequence as expected. With `acctManip`, this is the case for one through five artificial tasks, but for `resHJ` and `ACE`, adding three or more artificial tasks results in some of the artificial tasks also being placed under the parallel split node instead of the sequence. Figure 4.7 shows the resulting sub-model for the `wireless` tasks after inserting three sequential events for `ACE|wireless`. The original task and one of the duplicates are placed in the higher-level parallel split, and the two other artificial tasks are placed in the sequence.

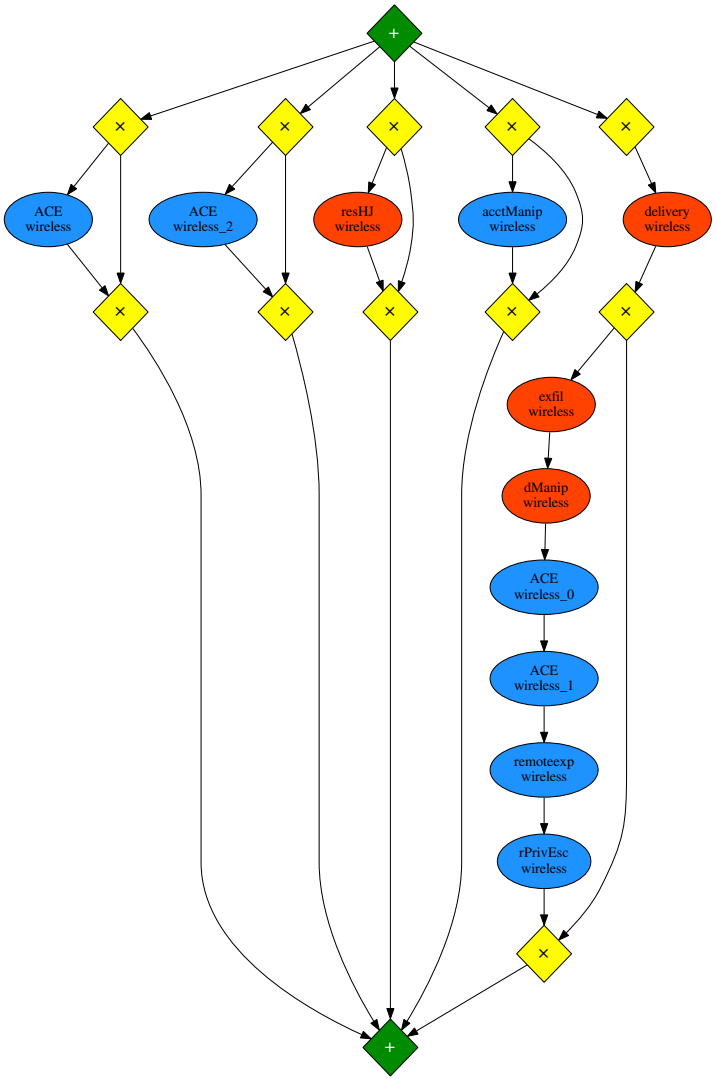
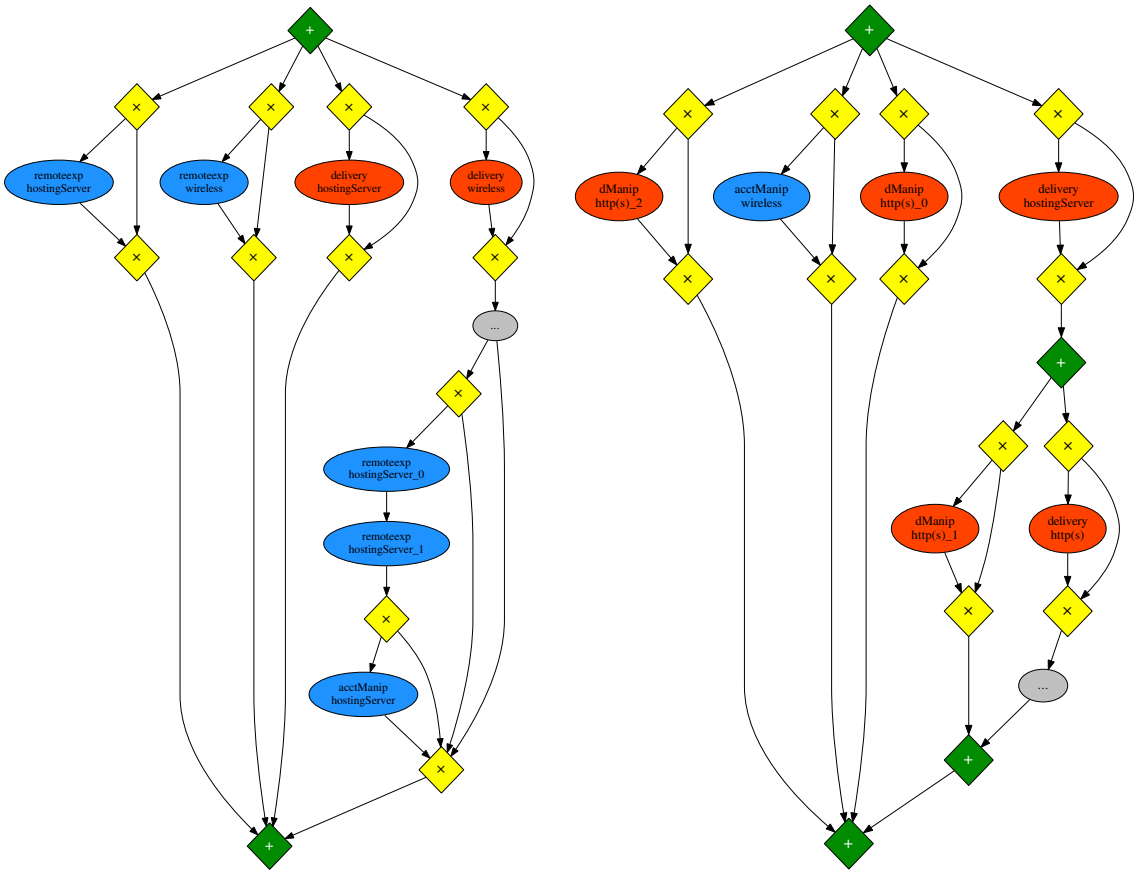


Figure 4.7: Sub-models for wireless tasks after inserting three artificial sequential tasks for `ACE|wireless`.

For the task `remoteexp|hostingServer`, inserting artificial sequential tasks resulted in the dependency with `acctManip|hostingServer` being identified, as shown in Figure 4.8a. For the sake of simplicity, a part of

the model has been removed from the visualization. The dependency is only identified for the two artificial tasks, and the base task is still placed directly under the parallel split node. Furthermore, the unrelated tasks `remoteexp|wireless` and `delivery|hostingServer` have also been moved in the model even though these events have not been affected by the artificial events.

With `dManip` and `ACE` on `http(s)`, no sequential dependency was identified. After inserting an artificial dependency for `ACE`, the artificial tasks were almost exclusively placed under the parallel split node. One exception here was that one task was placed elsewhere in the model if four or five artificial tasks were inserted. Inserting tasks for `dManip` resulted in the new tasks being placed either under the same parallel spit node as the original task or under a new parallel split node, as shown in Figure 4.8b. This model also shows that the ordered relation between three other tasks (not shown for the sake of simplicity) is only defined for the original task and not for the artificial tasks. In both cases, dependencies around unrelated tasks were also changed in the model despite no alterations being made around those tasks.



(a) `remoteexp|hostingServer` (b) `delivery|http(s)`

Figure 4.8: Sub-models after inserting artificial sequential tasks for `remoteexp|hostingServer` (left) and `dManip|http(s)` (right). Some parts have been removed for the sake of simplicity.

4.4. Split Miner Models

Figure 4.9 shows the full model produced by the Split Miner on the reversed traces of CPTC'18. This model is able to perfectly replay 399 out of the 536 traces in the dataset. Opposed to the models from the inductive miners, this model does not feature any parallelism.

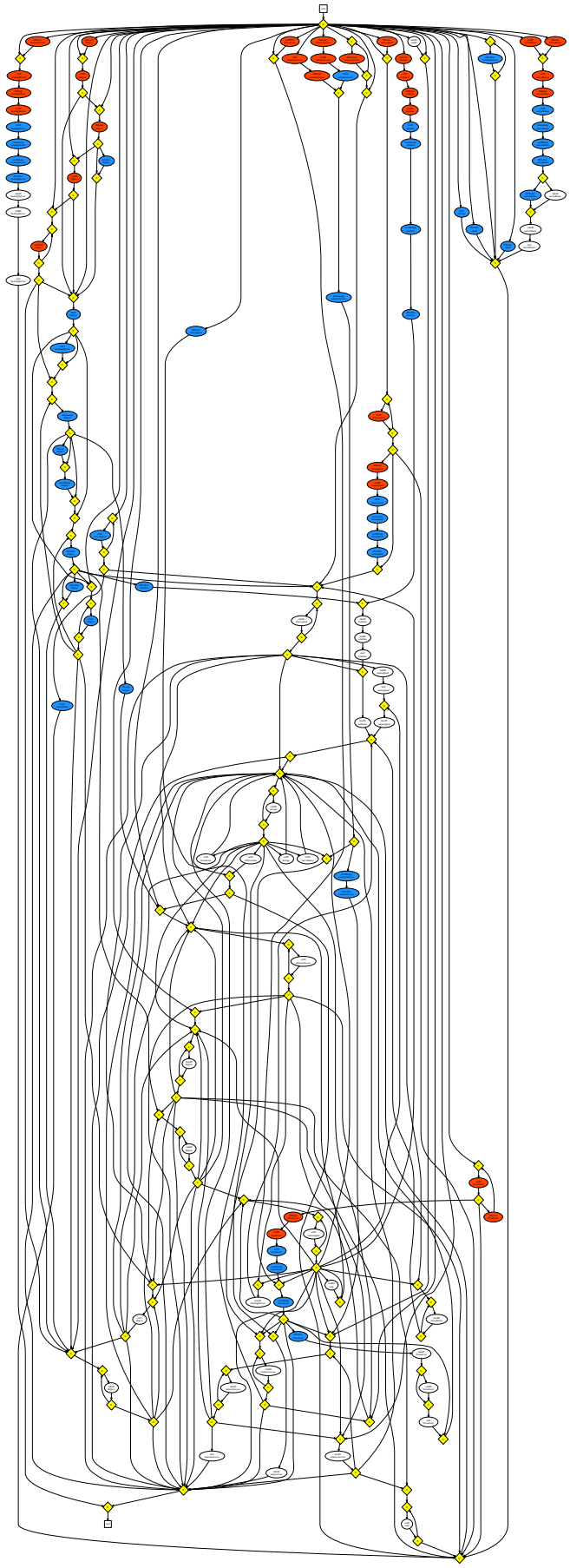


Figure 4.9: Full models constructed by the Split Miner on the reversed traces of CPTC'18. Tasks are shown as circular nodes colored red for high-severity attack stages, blue for medium-severity attack stages, and white for low-severity attack stages. Exclusive-choice split and join gateways are shown with yellow diamonds. The full-scale version is available on https://github.com/ghabbenjansen/bpmn_replay

Structure
The model from the Split Miner starts with a sizeable exclusive-choice split gateway, sending the process into one of 35 different starting paths. These paths mostly begin with isolated sequences of tasks that are mostly disconnected from each other. Through this low level of connectivity between the different paths here, the model implies that the high-severity and medium-severity steps of the attack traces are primarily independent of each other.

Lower in the model, the different tasks become more interconnected, and the clear distinction between different paths disappears. This connectivity is mostly modeled through high-degree split and join gateways. Over the entire model, the exclusive split gateways branch out the incoming path into an average of 4.20 outgoing paths, and each join gateway merges on average 2.93 paths back into one. Such high averages were expected given the model features significantly less exclusive-split and join gateways (46 and 76 respectively) compared to the models from the Inductive Miner (105 and 103 respectively). Yet, the CFC of the two models is still comparable, with the Split Miner having a CFC of 193 and the Inductive Miner resulting in a model with a CFC of 244. Combining this insight with the statistics from Table 4.3, we can conclude that the models from the Split Miner have high connectivity between tasks.

One thing to note is that despite the high connectivity lower in the model, it is not the only reason for the visual noise in that part of the model. The rendering method used placed certain nodes lower in the model even though they are closely connected to the root. As a result, the model gives an inflated representation of the actual connectivity lower in the model. One example comes from the red high-severity tasks towards the bottom of the model, which seemingly imply that these tasks usually occur later in the traces. However, these tasks are actually closely connected to the exclusive split gateway at the root of the model.

Opposed to the models from the Inductive Miner, tasks in the model from the Split Miner are rarely optional. In total, only 14 tasks are in a take-or-skip structure, and two tasks are in a loop structure which allows them to be executed zero or more times. As a result of this low frequency of optional tasks, the model does not permit the empty trace, and only eight single-event traces are possible. These values are a lot lower than seen with the Inductive Miner model, which has 56 optional tasks, 23 tasks in a zero-or-more loop and allows for 90 single-event traces.

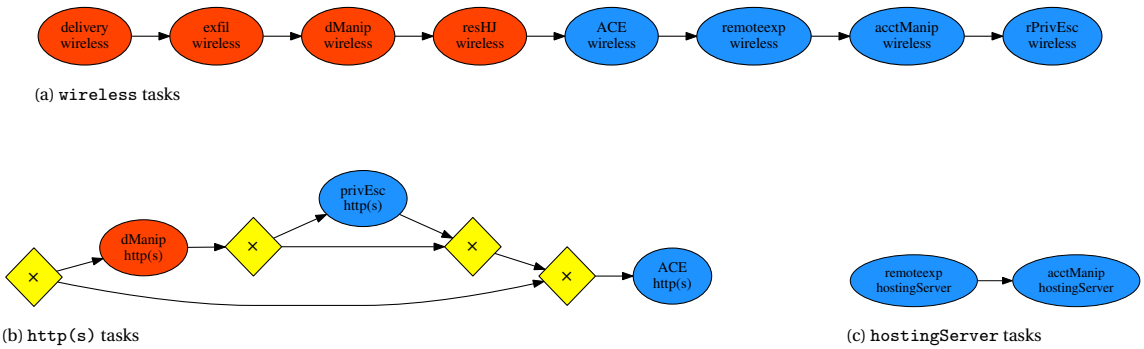
Overall, the structure of the model shows similarities with the state machine models which are currently used. Looking at the workings of the Split Miner, the mining method essentially produces a filtered Markov chain, with pre-processing and post-processing steps to detect and model parallelism and self-loops. As no parallelism was identified during the mining process, the resulting model is highly similar to a state machine at a conceptual level.

4.4.1. Model Validity
Opposed to the Inductive Miner, the model produced by the Split Miner does not offer perfect fitness, so one might argue that the model is not completely valid given the data. However, this does not automatically mean that all implications made by the model can be assumed to be false. Therefore, it is still possible to argue about the validity to some extend.

Figure 4.10 shows the dependencies between tasks as discussed in Section 4.3.1. To begin, the Split miner correctly identified the sequence of `wireless` tasks (4.10a). Looking at the relation between `dManip|http(s)` and `ACE|http(s)`, the model shows a clear sequential dependency between the two tasks where `ACE` can never occur before `dManip` (4.10b). Through a series of exclusive-split gateways, which are not all shown in the sub-model, the possibility exists to execute both the `dManip` and `ACE` tasks independently of each other, conforming with the observations in the traces.

For `remoteexp|hostingServer` and `acctManip|hostingServer`, the model also shows the sequential relation between the tasks (4.10c), but the lack of completeness is showing. No option exists to execute the `remoteexp` task without also executing the `acctManip` task, which does not conform to the data.

Following the fact that 137 out of the 536 traces are not valid according to the model, we know more structures exist which incorrectly disallow specific observed behavior. Looking at this from the perspective of the mining algorithm, this was to be expected due to the filtering step in the mining process. This step removes dependencies from the directly-follows graph to reduce noise in the resulting model. However, there is no way we can distinguish noise from infrequent behavior in this data, so any filtering has may limit valid behavior in the model.



(a) `wireless` tasks

(b) `http(s)` tasks

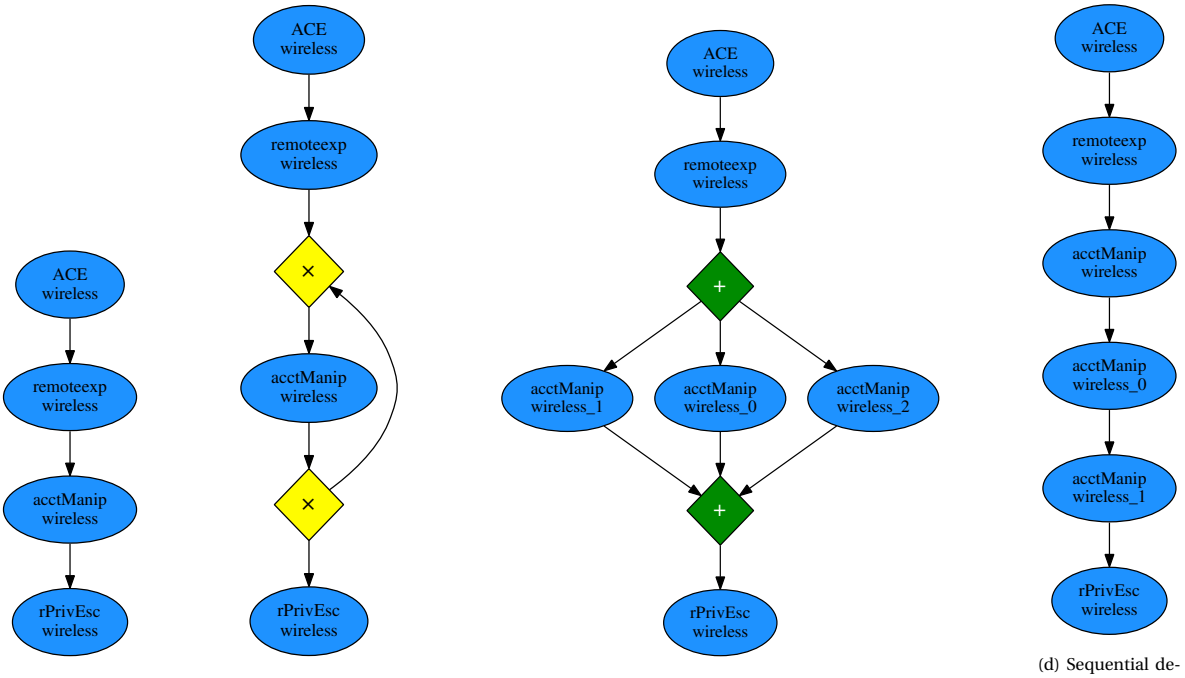
(c) `hostingServer` tasks

Figure 4.10: Sub-models from the Split Miner model for the reversed traces of CPTC'18 for all tasks identified in Section 4.3.1.

Removing the filtering step from the mining process results in a model which creates just a Markov chain. The short-loop removal from the directly-follows graph is mainly present because the split and join identification cannot handle these constructs. A closer inspection of the mining process showed that no parallelism is detected when constructing the model for CPTC'18. Therefore, the filtering step is necessary to distinguish the Split Miner from a Markov chain, and it has to be accepted that the model will not fit certain traces.

4.4.2. Model Robustness

By applying the same methodology from Section 4.3.2, we can verify whether the Split Miner is also produces consistent models. Even though the Split Miner does not use the cutting rules from the Inductive Miner, we can still use the same method of injecting artificial dependencies. Furthermore, we can verify whether the Split Miner is able to identify simple forms of parallelism when we know for sure this is present in the dataset.



(a) Base model

(b) Loop dependency

(c) Parallel dependency

(d) Sequential dependency

Figure 4.11: Example of new models after injecting different dependencies for `acctmanip|wireless`. The model for the injected exclusive-choice dependency is not shown as this is highly similar to the model for the parallel dependency.

Figure 4.11 shows the resulting sub-models after injecting artificial dependencies for the task `acctManip|wireless`. In all cases, the resulting model is as expected for the altercation made. Here, the model for the exclusive-choice dependency is not shown as it is equivalent to the model for the parallel dependency in Figure 4.11c, but with the parallel gateways replaced by exclusive-choice gateways. Furthermore, the fact that the miner

can identify the injected parallel relation shows that the lack of parallelism in the full model is a result of the data and not the mining algorithm.

One notable occurrence for these experiments is that the miner did not produce a sound model in all cases where a parallel dependency was introduced. For the tasks `dManip|http(s)` and `ACE|http(s)`, an unsound model was produced when one or two artificial tasks were inserted. Adding artificial parallelism to all six tasks resulted in an unsound model for all tested numbers of artificial tasks. This lack of soundness indicates that the rules used to detect parallelism are not completely valid or that some mistake is present in the implementation of the miner.

4.5. Conclusions

The main takeaway from this evaluation is that the effectiveness of process mining techniques for modeling the traces generated for CPTC'17 and CPTC'18 is limited at best. Three process mining algorithms, $\alpha\$$, Fodina, and the Structured Miner using Heuristics Miner, could not (reliably) produce sound process models on the datasets. The Slit Miner can produce a sound model in most cases but still has issues with both the CPTC'17 dataset and some of the artificially enhanced datasets. Only the Inductive Miner can always generate a sound model as the miner constructs process trees.

Looking at only the performance metrics, the Inductive Miner with low noise thresholds achieves the best fitness and completeness, and the Split Miner scores best based on precision and F-score. Furthermore, the models from the Split Miner are less complex than the models from the Inductive Miner scoring similar on performance.

The validity of the models produced by the Inductive Miner can be argued due to the `ActivityConcurrent` fallthrough method. Instead of using a flower model when the miner cannot identify a cut on a sublog, the fallthrough removes one random activity, places it in parallel to the model for the sublog, and continues mining on the sublog with the activity removed. As a result, the models give false implications of parallelism and are sensitive to slight changes in the training data. Besides, the parallel activities added with the `ActivityConcurrent` fallthrough cannot be distinguished from actual parallel behavior in the dataset.

The models produced by the Split Miner are visually similar to the state machines currently used. As the miner does not find sufficient evidence for parallelism in the datasets, the resulting model closely resembles a Markov model where infrequent edges are filtered out. However, as the method for filtering infrequent edges is more sophisticated than just removing edges below a certain threshold, the models can provide more insights than a Markov model.

5

Performance of State Machines

With the second research question we want to know how well state machines perform on the IDS alert datasets. This chapter answers that question by showing the results from the evaluation outlined in Chapter 3. The chapter starts with the results of the performance metrics for the different state machine configurations, followed by the results for the complexity metrics. After this, the different types of models are discussed as well as the validity of the resulting models.

5.1. State Machine Performance

The results of the performance metrics for the state machines are shown in Table 5.1. As a reference, the table also contains the performance of the prefix tree to show how well the different configurations generalized over the base model.

Fitness

As expected, almost all models achieve high, if not perfect, scores for fitness when evaluated over the full datasets. For the bigram and Markov configurations, the perfect fitness is a result of the lack of sinks in the model. When sinks are not used, no states are removed during the merging process, and all behavior of the original prefix tree is retained in the models. With the convert sinks configuration, the behavior in infrequent sink sub-trees is also retained by merging all child nodes into their respective sink root, retaining all transitions in the final model.

The only models that do not feature this perfect fitness over the full data are the models which feature sinks without any sink conversion method: no sink merge and search. Due to this lack of sink conversion, some transitions and states are removed during the state merging process, removing some parts of the training data. The significant difference between fitness and completeness for these two configurations reflects this: the high fitness indicates that the majority of each trace is valid according to the model and the low completeness indicates that for a lot of traces, some smalls part of the traces (likely the infrequent suffixes) are left out.

The only configuration which does not follow this pattern is the original configuration, which uses sinks without any explicit mitigation method. Here, the lower threshold for the statistical test and higher minimum requirement for the merge heuristic score ensured the merging did not destroy these transitions.

Looking at the fitness results from cross-validation, we again see a similar pattern where the no sink merge and the search configurations score lower than the others. Despite these lower scores, the configurations still generalize better with respect to fitness compared to the prefix tree model.

The convert sinks and Markov configurations consistently score the highest concerning generalized fitness and completeness, consistently scoring around 0.95 and 0.75, respectively. However, the bigram and original configurations do not trail far behind, achieving fitness scores just over 0.90. In the area of completeness, these two configurations do provide significantly worse models, as indicated by the scores around 0.54 for CPTC'17 and 0.64 for CPTC'18.

Reversing the traces does not seem to affect the fitness and conformance of the produced models.

Dataset	Configuration	Full data					Averages from 5-fold cross-validation				
		Soundness	Fitness	Completeness	Precision	F-score	# Sound	Fitness	Completeness	Precision	F-score
CPTC'17 (chronological)	Bigram	sound	1.00	0.99	0.68	0.81	5	0.90	0.54	0.40	0.55
	Convert Sinks	sound	1.00	1.00	0.44	0.61	5	0.95	0.73	0.26	0.40
	Markov	sound	1.00	1.00	0.46	0.63	5	0.95	0.79	0.28	0.43
	No sink merge	sound	0.90	0.64	0.44	0.59	5	0.86	0.52	0.27	0.42
	Original	sound	1.00	0.99	0.57	0.72	5	0.90	0.59	0.31	0.47
	Search	sound	0.90	0.64	0.44	0.59	5	0.87	0.52	0.28	0.42
	Prefix tree	sound	1.00	1.00	1.00	1.00	5	0.81	0.27	0.53	0.64
CPTC'17 (reversed)	Bigram	sound	1.00	1.00	0.76	0.86	5	0.90	0.54	0.44	0.59
	Convert Sinks	sound	1.00	1.00	0.58	0.74	5	0.95	0.76	0.35	0.51
	Markov	sound	1.00	1.00	0.56	0.72	5	0.95	0.78	0.34	0.50
	No sink merge	sound	0.90	0.59	0.45	0.60	5	0.87	0.50	0.32	0.47
	Original	sound	1.00	1.00	0.70	0.82	5	0.90	0.58	0.40	0.56
	Search	sound	0.90	0.60	0.46	0.61	5	0.87	0.51	0.33	0.48
	Prefix tree[1]	sound	1.00	0.99	1.00	1.00	5	0.81	0.26	0.53	0.64
CPTC'18 (chronological)	Bigram	sound	1.00	0.99	0.61	0.76	5	0.92	0.64	0.32	0.47
	Convert Sinks	sound	1.00	1.00	0.40	0.57	5	0.95	0.73	0.19	0.31
	Markov	sound	1.00	1.00	0.50	0.67	5	0.94	0.74	0.25	0.39
	No sink merge	sound	0.88	0.58	0.53	0.66	5	0.85	0.47	0.26	0.40
	Original	sound	1.00	1.00	0.62	0.76	5	0.90	0.60	0.35	0.50
	Search	sound	0.88	0.57	0.50	0.64	5	0.85	0.48	0.26	0.40
	Prefix tree	sound	1.00	1.00	1.00	1.00	5	0.84	0.42	0.55	0.66
CPTC'18 (reversed)	Bigram	sound	1.00	1.00	0.68	0.81	5	0.92	0.65	0.37	0.53
	Convert Sinks	sound	1.00	1.00	0.41	0.58	5	0.96	0.76	0.23	0.37
	Markov	sound	1.00	1.00	0.55	0.71	5	0.94	0.76	0.31	0.46
	No sink merge	sound	0.88	0.56	0.48	0.62	5	0.85	0.49	0.32	0.47
	Original	sound	1.00	1.00	0.65	0.78	5	0.91	0.64	0.36	0.51
	Search	sound	0.88	0.57	0.45	0.60	5	0.86	0.50	0.31	0.45
	Prefix tree[1]	sound	0.99	0.95	1.00	1.00	5	0.84	0.44	0.50	0.63

Table 5.1: Performance metrics for state machines constructed using different configurations for the traces of CPTC'17 and CPTC'18.
[1] The prefix tree is ensured to have perfect fitness and conformance, hence non-perfect scores indicate a minor issue in either the model construction or the evaluation method. Given that for both cases only one trace did not fit, the issue was not investigated further.

Precision
With the precision computed over the full dataset, we see more variations between the different configurations. The bigram and original configurations are consistently the best performing configurations, scoring between 0.10 and 0.20 higher than the others. For this, the high number of states in the models (see Table 5.2 is a strong indication. With more states in the model, the prefix to reach each state becomes less common, decreasing the number of possible suffixes of the state, which in turn is beneficial for precision.

Looking at the impact of cross-validation, we see that precision drops quite significantly. This drop is explained by combining the idea of state merging and the relatively small evaluation dataset. With state merging, all (or most when sinks are used) transitions from the training data remain in the model, including all infrequent transitions which occur only once or twice in the full dataset. The evaluation data does, in turn, not feature this infrequent behavior as it is unique to the training data, meaning a relatively high number of transitions in the model are not accounted for during evaluation.

Following this reasoning, it would make sense for the prefix tree to also score low on generalized precision, but what we see is the opposite. The reason for this is the way precision is calculated based on the model alignment prefixes. When any model prefix does not cover a state during the evaluation, the outgoing transitions of that state, and by extension, the entire sub-tree, are ignored in the final precision score. As merging states increases the number of different prefixes to reach a state, it becomes less likely that certain states are not covered during evaluation, and fewer parts of the model are ignored when computing precision. Following this reasoning, a state merging method that only performs merges of states with identical futures should yield higher precision scores.

Reversing the traces does have a slight positive effect on the precision, with an 10% increase in precision for reversed models compared to their chronological counterparts.

5.2. State Machine Complexity

Once again, having a model which can describe the data well is limited in usage if it cannot be interpreted. Table 5.2 shows the complexity metrics as computed over the state machines directly. In order to compare the metrics somewhat better to the process models, Table 5.3 shows the complexity metrics of the machines converted to BPMN.

State Machine Complexity
Looking at the complexity of the state machines directly, we see that there is a large variance in the number of states and transitions between the different configurations.

On the low end are the models from convert sinks and Markov. For the Markov model, the low number of states is expected as the resulting model is close to a Markov chain. In this chain, one state exists for each event in the dataset, so it is no surprise that the number of states in the resulting model closely represents this. The convert sinks model even features fewer states than there are events in the datasets. This is caused by the presence of sinks and the way they are dealt with: all infrequent events are identified as a sink, and all sub-trees rooted at a sink are merged into the root node. As a result, many states corresponding to infrequent events are squashed into their preceding sinks and thereby removed from the model.

With the configurations with more states, bigram and original, we see a relation between the number of transitions and CFC. For the convert sinks, Markov, no sink merge, and search configurations, these two values are basically equal, whereas a more substantial difference exists for the bigram and original configurations. Given the CFC is the sum of out-degrees for states with more than one outgoing transition, the difference between the number of transitions and CFC is the number of states with only one outgoing transition. Hence, the machines from the bigram and original configurations feature more states with a known direct future.

This is partially explained by the high amount of states for the bigram and original configurations. Given the base prefix tree has a fixed number of states where most only have one outgoing transition, it is expected that models with more resulting states have more of these single-transition states remaining.

Looking at the CNC of the different models, we see more uniformity. Here, the only real outliers are the convert sinks models, and to some extent, the Markov models. For the convert sinks models, the high CNC is a product of the sink conversion method, which basically results in a model where most of the transitions

of the original prefix tree are retained. Pairing this with the low number of states makes the high CNC score inevitable.

The Markov configuration also results in models with a high CNC. Here, similar reasoning applies: all transitions from the base prefix tree are retained in the model, but the number of states is closely related to the number of events in the training data. This again results in a high imbalance between the two which increases the CNC score.

Dataset	Configuration	States	Transitions	CFC	CNC
CPTC'17 (chronological)	Bigram	541	1213	966	2.24
	Convert Sinks	80	825	817	10.31
	Markov	100	574	551	5.74
	No sink merge	324	394	388	1.22
	Original	706	926	611	1.31
	Search	315	394	389	1.25
CPTC'17 (reversed)	Bigram	564	1254	1002	2.22
	Convert Sinks	100	766	762	7.66
	Markov	100	603	588	6.03
	No sink merge	323	396	390	1.23
	Original	724	996	669	1.38
	Search	308	393	391	1.28
CPTC'18 (chronological)	Bigram	412	752	524	1.83
	Convert Sinks	74	656	647	8.86
	Markov	125	474	431	3.79
	No sink merge	219	258	239	1.18
	Original	530	692	383	1.31
	Search	208	256	248	1.23
CPTC'18 (reversed)	Bigram	431	775	517	1.80
	Convert Sinks	75	557	552	7.43
	Markov	123	467	428	3.80
	No sink merge	207	250	243	1.21
	Original	454	657	397	1.45
	Search	199	251	247	1.26

Table 5.2: Size and complexity metrics computed for the different DFA

BPMN Complexity

Looking at the complexity metrics as computed over the state machines converted to BPMN, we see the same general trends for the different configurations. As a result of the Markovian setting used in all configurations, the activity merging caused all incoming transitions to each state to be merged into a single task in the resulting model. The converted machines have the same number of tasks as the original state machine has states, where the initial state causes the discrepancy of one as this state has no incoming transitions. The only exception to this rule is the convert sinks configuration. Because of the sink conversion, the incoming transitions for a sink state are not all equal, and therefore they cannot be merged. As a result, the converted models have a lot more tasks balancing the number of nodes and the number of edges in the model, as shown by the CNC.

For the other configurations (except Markov), the conversion process also balanced the number of nodes with the number of edges, resulting in highly similar CNC values between the configurations. With the Markov configuration, the CNC value remains higher as the model does not really change compared to the state machine.

The overall structuredness of the models is quite low, indicating a high degree of interdependence between different parts of the models. Especially the bigram and Markov configurations stand out with their exceptionally low structuredness scores. However, these low scores were expected given the highly connected nature of these kinds of models. Only the no sink merge and search configurations score high on this metric due to the tree-shaped nature of the resulting models.

Dataset	Configuration	Total Nodes	Total Edges	Tasks	Exclusive		CFC	CNC	Struct.
					Splits	Joins			
CPTC'17 (chronological)	Bigram	1021	1927	540	242	237	1149	1.89	0.17
	Convert Sinks	889	1704	630	77	180	893	1.92	0.68
	Markov	259	803	99	77	81	622	3.10	0.34
	No sink merge	390	770	323	43	22	424	1.97	0.91
	Original	1237	2136	705	434	96	1334	1.73	0.67
	Search	386	770	314	41	29	426	1.99	0.87
CPTC'17 (reversed)	Bigram	1057	1960	563	282	210	1186	1.85	0.13
	Convert Sinks	850	1588	560	96	192	835	1.87	0.63
	Markov	260	809	99	85	74	635	3.11	0.35
	No sink merge	404	764	322	57	23	418	1.89	0.86
	Original	1283	2235	723	474	84	1427	1.74	0.65
	Search	389	754	307	50	30	416	1.94	0.83
CPTC'18 (chronological)	Bigram	704	1165	411	144	147	606	1.65	0.19
	Convert Sinks	721	1357	525	64	130	701	1.88	0.71
	Markov	292	698	124	83	83	490	2.39	0.28
	No sink merge	273	499	218	36	17	263	1.83	0.85
	Original	987	1645	529	379	77	1038	1.67	0.64
	Search	263	499	207	35	19	272	1.90	0.85
CPTC'18 (reversed)	Bigram	725	1163	430	165	128	604	1.60	0.14
	Convert Sinks	632	1162	417	70	143	601	1.84	0.61
	Markov	287	665	122	85	78	464	2.32	0.26
	No sink merge	265	483	206	39	18	258	1.82	0.77
	Original	872	1489	453	349	68	967	1.71	0.46
	Search	259	483	198	37	22	262	1.86	0.75

Table 5.3: Size and complexity metrics for the BPMN models equivalent to the state machine.

5.3. State Machine Models

The produced state machines for the reversed CPTC'18 traces are shown in Appendix B. In these models, states and transitions corresponding to high-severity attack stages are shown in red, medium severity in blue and low severity in white/black, and sink states are shown in yellow. For the sake of simplicity, the convert sinks model is simplified by rendering only one self-loop for each sink state.

Models

When comparing the different models, we can distinguish three different types. First are the bigram and Markov models, which both feature a model with many highly connected states. Towards the top of both models, we see the states for the high and medium severity events, connected through a relatively small number of transitions. Moving lower in the model, we see the states for the low severity events with a higher transition density between them compared to the medium and high severity states.

Second are the models from no sink merge, search and convert sinks. Given that both the performance and complexity metrics for no sink merge and search are highly similar, it was expected that these two models look alike. The models feature what is effectively a state machine for the frequent part of the training data with the sinks at the roots of the infrequent paths. The convert sinks configuration features the same core state machine for the frequent parts of the training data but allows for sinks to have incoming transitions with different labels. As a result, sinks with the same preceding state can be merged, resulting in an overall simpler model. The drawback of removing the infrequent parts of the models is that all information from these parts is lost. Perhaps there are similarities between different infrequent sub-trees from the original prefix tree.

Third is the model from the original configuration, which at first does not show many similarities to the other five models. This configuration does not filter out infrequent parts of the prefix tree but instead continues the merging process despite the lack of statistical certainty due to the low frequency of these parts. Following the configurations used, it is expected that when the infrequent sub-parts of the model are removed, the remaining machine is also similar to the model from convert sinks.

Model Validity
Due to the approach used when constructing the state machines, we can have high confidence that the produced models are valid with respect to the training data. The process of constructing the models starts with creating the prefix tree of the training data, capturing all observed behavior. Then, the state merging process generalizes the tree by allowing multiple paths to and from the same state. It is in this merging process where new behavior is introduced as the sub-trees rooted at the merged states do not have to be equivalent, but only equivalent enough.

With the bigram and Markov configurations, certain enough is defined through the Markovian principle that conditioned on the present, the future is independent of the past. Due to this principle, the validity of the models depends entirely on whether the assumption that the Markovian principle holds is valid.

With the other four configurations, the addition of sinks plays a role in the validity of the models. At the core, the meaning of the sink states can be summarized as "there is not enough evidence for these parts, so we are not making any assumptions here". Following such a statement, omitting parts of the model is valid as no incorrect implications can be made about the data as no implications are made about the data. Yet, there needs to be a balance as not all infrequent behavior can automatically be labeled as noise.

5.4. Conclusions

Following just the performance metrics, the state machines generally achieve a high fitness at the cost of precision. In terms of fitness and precision, the convert sinks and Markov configurations produce the best models. For precision, the bigram and original configurations are the best-performing.

However, the validity of the metrics can be questioned given the performance of the prefix tree compared to the state machines. All state machines are a generalization over this model, yet based on balanced performance measured by the F-score, the prefix tree is the best model available.

With respect to the metrics, two factors are in play. First is the fact that the fitness metric cannot deal with the concept of sink states. When computing the alignment of a trace extending past a sink state, skips for the trace are inserted, leading to a worse fitness score. However, following the concept of sink states where the model basically says that any behavior is allowed, the penalty is unjustified.

The second issue is the way precision is computed and how it deals with non-fitting traces. If some trace is too 'long' for the model, the computed alignment does not advance the model, and no new states are considered for the precision, introducing a positive bias in the scores. This can be seen with the prefix tree scoring remarkably high on precision compared to the other state machines. Besides, the precision metric requires that for any model prefix to reach some state, as many outgoing transitions as possible should be taken. For the state machines, this means that perfect precision can only be achieved when states are only merged if the sets of outgoing transitions are completely equal. Finally, the precision is also negatively impacted by the cross-validation experiments. By evaluating the model on only one-fifth of all traces, it is not unexpected that a sizable portion of the paths is not covered.

Compared to the process mining algorithms, the fitness and completeness of the state machines are slightly higher, but the precision is significantly lower. In terms of complexity, the no sink merge and search configurations are about equivalent to the process models, with the other four configurations scoring significantly higher on the number of nodes, number of edges, or both.

6

Combining Process Mining and State Machines

The third topic of this thesis is evaluating whether the process mining algorithms can be used in combination with the state machines. The main difference between state machine inference and process mining is that state machine inference models the context of events, whereas process mining deals with parallelism in the data. By combining the two methods, we can possibly try to get the best of both worlds with a method that is aware of both the context of events and possible parallelism in the data.

This chapter starts with a description of how the combination works. Then, the results from the hybrid state machine-process mining algorithms are discussed.

6.1. Replaying Traces

The first step of the combination process is replaying the input traces over the constructed state machine. This replay procedure is shown in Algorithm 3. The general idea is that the event sequence is replayed over the given state machine. The replay starts at the initial state. Then, for each next event, line 4 checks if an outgoing transition exists for the current state. The new state is recorded when this is the case, and the replay continues with the next event. When no outgoing transition exists, the check at line 7 checks if the replay currently is in a sink state and stops the replay if this is the case. This termination is in accordance with the concept that any trace ending in a sink state is valid. Finally, if no transition exists for the current state and the replay is not in a sink, an error label is recorded in the trace, and the replay continues without updating the current state.

Algorithm 3: Replay algorithm to extract the state sequence of a trace for a given state machine.

Data: State machine $(Q, \Sigma, \delta, q_0, F)$
Data: Trace $e_1, e_2 \dots e_n$
Result: New event sequence
1 $new\_sequence \leftarrow []$
2 $current\_state \leftarrow q_0$
3 **for** $1 \leq i \leq n$ **do**
4 **if** transition $(current\_state, e_i) \rightarrow q_{next}$ exists in δ **then**
5 append q_{next} to $new\_sequence$
6 $current\_state \leftarrow q_{next}$
7 **else if** $current\_state$ is a sink **then**
8 stop replaying the trace
9 **else**
10 append an error label to $new\_sequence$

With this replay procedure, a new dataset is constructed by first constructing a state machine over the training data and then replaying the evaluation traces over that machine. For the full dataset, the training

and evaluation traces are both equal to the full dataset. With the cross-validation experiments, the original dataset is randomly split into five partitions. Then, for each of the five folds, the state machine is constructed using four of the partitions. The evaluation data is the product of replaying the fifth partition on this machine.

The new models are created by using the process mining algorithms with the datasets resulting from the replay procedure.

6.2. Performance

Using the replayed traces, the performance evaluation can again be performed for the different process mining algorithms. Due to time limitations and the large number of datasets, the evaluation is only performed over the reversed traces of CPTC'18.

Inductive Miner

On the base datasets, the Inductive Miner was often not able to identify cuts during the mining process, resulting in a high number of applications of the `ActivityConcurrent` fallthrough and, by extension, unreliable models. Hence, the main improvement to be gained for this miner is a reduction in the number of tasks placed in concurrent blocks.

Table 6.1 shows the number of tasks modeled as optional directly between a parallel split and a parallel join node. Overall, between 20% and 50% of all tasks has been modeled in such a manner, with the exception of the combination of no sink merge and a 20% noise threshold where only 6% of tasks is affected by the fallthrough. For all these tasks, it is possible that sufficient evidence exists to justify the model, but given the large volume of tasks modeled in this manner, it is highly unlikely that this holds for all tasks. Hence, it is safe to conclude that using state sequences does not help the Inductive Miner with identifying splits. Therefore, no benefits can be gained from this combination.

Miner	Bigram		Convert Sink		Markov	
	Tasks	Concurrent	Tasks	Concurrent	Tasks	Concurrent
IM	430	212 (0.49)	74	36 (0.49)	122	42 (0.34)
IMf-05	407	195 (0.48)	73	34 (0.47)	117	21 (0.18)
IMf-10	384	179 (0.47)	72	26 (0.36)	117	19 (0.16)
IMf-15	387	181 (0.47)	72	19 (0.26)	115	21 (0.18)
IMf-20	386	187 (0.48)	72	14 (0.19)	117	22 (0.19)

Miner	No sink merge		Original		Search	
	Tasks	Concurrent	Tasks	Concurrent	Tasks	Concurrent
IM	206	38 (0.18)	453	197 (0.43)	198	31 (0.16)
IMf-05	136	31 (0.23)	417	170 (0.41)	125	31 (0.25)
IMf-10	106	27 (0.25)	388	147 (0.38)	99	29 (0.29)
IMf-15	101	26 (0.26)	374	138 (0.37)	95	26 (0.27)
IMf-20	98	6 (0.06)	372	138 (0.37)	86	21 (0.24)

Table 6.1: Number of tasks likely modelled through the `ActivityConcurrent` fallthrough for the Inductive Miner with different noise thresholds.

Split Miner

With the base datasets, the Split Miner performed best based on the precision metric and the F-score, so it is expected that the miner also performs well on the hybrid approach. Table 6.2 shows the performance of the Split Miner using the hybrid approach. On the full dataset, we see a flat-out improvement in both fitness and precision compared to the performance on the base dataset. For the cross-validation, only the convert sink configuration provides an overall improvement over the performance for the base data. The convert sink, no sink merge, and search configurations show an improvement in fitness, which could mean that the Split Miner benefits from the traces being truncated in sink states, given that replay on these three configurations does just that. With the bigram, Markov, and original configurations, we see a slight drop in performance which could be explained by these configurations either not using sinks (bigram and Markov) or not removing nested sink states (original).

However, none of the produced models feature any parallel gateways meaning the models are again effectively filtered Markov chains. As a result, the Split Miner does not add anything new over the state machines used for replaying the traces but only removes transitions deemed infrequent. Therefore, using the Split Miner in combination with state machines does not provide any benefits for this dataset.

Configuration	Full data				
	Soundness	Fitness	Completeness	Precision	F-score
SM-Bigram	sound	0.95	0.81	0.76	0.85
SM-Convert sink	sound	0.96	0.82	0.84	0.90
SM-Markov	sound	0.93	0.70	0.73	0.81
SM-No sink merge	sound	1.00	0.99	0.78	0.88
SM-Original	sound	0.97	0.88	0.70	0.81
SM-Search	sound	1.00	0.99	0.78	0.87

Configuration	Averages from 5-fold cross-validation				
	# Sound	Fitness	Completeness	Precision	F-score
SM-Bigram	5	0.83	0.58	0.44	0.57
SM-Convert sink	5	0.91	0.70	0.72	0.81
SM-Markov[1]	4	0.88	0.65	0.51	0.64
SM-No sink merge	5	0.93	0.81	0.46	0.62
SM-Original	5	0.83	0.61	0.42	0.56
SM-Search	5	0.94	0.82	0.46	0.61

Table 6.2: Performance of the hybrid approach using the Split Miner on the reversed traces of CPTC'18.
[1] Only four folds were sound, hence the averages are computed over the four sound folds instead of all five folds.

Other Miners

One final possibility is that the state sequences transform the dataset enough to make the other miners, $\alpha\$$, Fodina, and sHM6, viable. In order to test this, the three miners have been evaluated on the full dataset using the state sequences to determine whether they are able to model the data.

The Fodina miner was able to construct a sound model for the bigram, convert sink, and original configurations. Running the cross-validation experiments using the bigram configuration also resulted in sound models for all five folds, however, the convert sinks and original configurations yielded only one and three sound models, respectively.

The Structured Heuristics Miner and $\alpha\$$ both were not able to generate a sound model for any of the six configurations. Even stronger, $\alpha\$$ did not finish constructing a model within the four-hour mining period for four of the configurations. The sHM6 miner did construct a model for all six configurations within the time limits, but it was somewhat unexpected that all the models for the replayed traces were unsound since the model for the base reversed traces for CPTC'18 was sound.

6.3. Conclusions

This chapter introduced a method of combining state machines with process mining. Initial experiments on this combination showed that combining state machines with process mining is not guaranteed to yield improved results.

Three of the five mining algorithms could not construct a sound model using this method, but these miners also couldn't produce a sound model on the base data directly. With the Inductive Miner, we saw that the clustering performed by the state machines could not transform the data such that the miner can identify more cuts. Finally, the Split Miner again could not identify parallelism in the data, meaning that the resulting model is a filtered Markov chain constructed over a state machine.

However, the evaluation has only been performed over one dataset. Therefore, we cannot rule out that this method of combining state machines with process mining might better results than the two methods can achieve on their own.

7

Conclusion

This chapter starts with discussing the limitations encountered in the evaluation process of this thesis. Following this is the conclusion which answers the research questions. Finally, an indication for future work is given.

7.1. Limitations

Throughout this thesis, several issues have been encountered. This section discusses the three main issues and how they impacted the results.

Datasets

Within this thesis, evaluation is only done over the CPTC'17 and CPTC'18 datasets, both a product of a collegiate penetration testing competition. As a result, these datasets might not fully reflect real-world behavior and issues like background noise or approaches used by professional hackers. The lack of representative intrusion datasets is a broader obstacle in this area of research[16].

Furthermore, the evaluation results for the process mining algorithms can be attributed to the possibility that the datasets are not complete as required by several process mining algorithms.

Effects of the metrics

One of the recurrent themes in this thesis is the effectiveness of the precision metric, or the lack thereof. To begin, the metric combines poorly with the cross-validation experiments as a smaller evaluation set automatically leads to a smaller fraction of the paths in the model being taken. Second, the metric cannot deal with traces that are 'too large' for the models as it only uses the states seen in the model part of the alignments. This means that non-fitting traces where the alignment skips parts of the trace don't negatively contribute to the precision score. As a result, the metric has a bias towards models which explain less of the data and indirectly penalizes models which take the effort to model outliers in the data.

Differences in models

The issue with comparing process models with state machines is the conceptual difference between the types of models used. An example of this is how parallelism is modeled in a Petri net as opposed to a state machine. For this thesis, the comparison has been made by converting automata to an equivalent Petri net in order to run the same evaluation metrics on the different types of models. A negative bias in the shape of additional complexity might be introduced within such a conversion by translating the conceptual differences between the different models. Alternatively, a model might loose some of its expressiveness in the conversion: the sink states from the state machines cannot be directly translated into a BPMN model.

7.2. Conclusions

The goal of this thesis is to evaluate whether methods from process mining can be used to improve the models for the IDS alert datasets. In order to answer this question, different process mining algorithms have been evaluated. To verify if these algorithms provide improvements over the state machines currently used, different methods of constructing state machines have also been evaluated. In addition, a method for combining state machines and process mining was also tested.

This section briefly summarizes these results from all these evaluations in order to answer the research questions set out at the beginning of this thesis.

RQ1: How well can state-of-the-art process mining techniques model the IDS alert datasets?

In Chapter 4, an evaluation is done using six of the most prominent process mining algorithms: $\alpha\$$, Fodina, the Inductive Miner, Structured Heuristics Miner - ProM 6 and the Split Miner. Of these miners, only the Inductive Miners and the Split Miner were able to generate sound models for the CPTC'17 and CPTC'18 datasets. The mining process of the Inductive Miners ended up often using a fallthrough method as the algorithm could not identify patterns in the datasets. As a result, the generated process models contain tasks placed in parallel with other parts of the model and thereby falsely implying certain relations in the data and resulting in inconsistent choices in the resulting model.

Based on the performance metrics, the Split Miner produced the best models. However, no parallelism was identified during the mining process. As a result, the mining algorithm produced a filtered Markov chain. Besides, the miner was not always able to generate sound models on the CPTC'17 dataset showing the reliability of the miner can not be taken for granted.

Overall, the process mining techniques cannot provide new insights through modelling the CPTC datasets .

RQ2: How well can state-of-the-art state machine learning techniques model the IDS alert datasets?

In Chapter 5, the state machines for six different configurations for the Flexfringe tool have been evaluated. For evaluation, the state machines were converted to Petri nets and BPMN models such that the metrics for process models could also be applied to state machines. The metric scores showed that the fitness and completeness of the state machines are on par with the models from process mining, but through the low precision scores, the metrics indicate that the state machines allow more behavior compared to the process models.

The low precision for the state machines can largely be attributed to the alignment-based precision metric used. With most of the configurations used for constructing state machines, the infrequent transitions of the training data were retained in the final model. As all transitions are retained, it is to be expected that the evaluation data cannot cover the full model.

Therefore, if we blindly follow the metrics, the state machines can only provide a fitting but imprecise model for the CPTC datasets. However, it can be argued that the precision metric is not suitable for evaluating the state machines.

RQ3: How much improvement can be gained by combining process mining with state machine learning?

Chapter 6 introduced a method of combining state machines with process mining. This method was evaluated using the reversed traces of CPTC'18 in combination with the six state machine configurations and five process mining algorithms from the previous chapters. Three of the five miners could not produce sound models, and the Inductive Miner and Split Miner suffered from the same issues also encountered on the base data. However, this does not indicate that the method is ineffective just yet as it only has been evaluated on one dataset, which is shown to be hard for the process mining algorithms.

Therefore, we can conclude that the method does not offer improvements over the base state machines and process miners for the reversed traces from CPTC'18. However, we cannot rule out that the negative results are a factor of the dataset used, meaning the method might be effective for other datasets.

Main question: To which extend can process mining be used to improve the models for IDS alert datasets?

In this thesis, multiple process mining algorithms have been evaluated for the CPTC'17 and CPTC'18 datasets. This evaluation showed that the performance is not sufficient: three miners could not produce sound models, the fourth miner gives inconsistent models with false implications, and the final miner produces a simplified Markov chain while also being unable to reliably construct sound models. Therefore, the process mining algorithms are either invalid, unreliable, or only give limited new insights over the state machines currently used. Furthermore, the evaluation also showed that combining state machines and process mining is ineffective for the reversed traces of CPTC'18.

Therefore, we can conclude that the evaluated methods from process mining cannot improve the models used for the IDS alert datasets.

7.3. Future work

Following the findings in this thesis, new directions for future research have been identified.

Combining state machines and process mining

Following the results from Chapter 6 we know that combining state machines with process mining is not effective for the reversed traces of CPTC'18. However, these experiments are not sufficient to show that combining the two is not effective at all. In future work, the method can be evaluated over other datasets, including synthetic datasets where it is known that parallelism and the context of events play a role.

Datasets

In the base work of Nadeem et al., the method for constructing attack graphs was also evaluated with the CCDC'18 dataset[1]. This dataset can also be used for the evaluations performed in this thesis. By performing the evaluation over more datasets, we can verify whether the patterns observed exist or if they are a by-product of the two datasets used. Of course, other IDS alert datasets can also be suitable for the evaluation.

Modeling algorithms

In this thesis, only a limited number of process mining algorithms have been evaluated. As new algorithms are continuously being developed and existing algorithms are being improved upon, it is possible that a better algorithm exists but has not been evaluated.

Two possible miners for further evaluation are the Induplet Miner and the Evolutionary Tree Miner. The Induplet Miner[24] uses the Inductive Miner as a base but tries a bottom-up approach before resorting to the fallthrough methods. This miner has come to the author's attention late during the evaluation process, and due to the horrible worst-case runtime ($O(2^{2^{|\Sigma|}})$, where $|\Sigma|$ is the number of unique events in the log), the miner was not included. Alternatively, the Evolutionary Tree Miner is a genetic algorithm constructing process trees. By playing around with the scoring function for the resulting process trees, it might be possible to construct process trees suitable for our applications.

Another topic for future research lies with the Inductive Miner. The original paper uses a flower sub-model when no cut can be identified, but the implementation evaluated uses the `ActivityConcurrent` fallthrough. By using the flower method instead, we can get insight into how the difference affects the precision of the resulting model as well as get a better idea of when the miner encountered a sub-model which it could not identify.

Finally, other state machine inference methods can also be evaluated. In particular, it would be interesting to see how a merging algorithm works, which only allows merges of states with an equal set of outgoing transition labels. Theoretically, such a merging algorithm should produce a model which scores perfectly on the current precision score. This raises questions about the impact on performance and how different the resulting model would be from the original prefix tree.

[1] http://www.nationalccdc.org/

A

Configuration files for flexfringe

The following configuration files have been used to construct state machines using Flexfringe[1]. These configurations can be supplied to the program by linking an `ini` file through the `-i` flag.

Bigram

```
[default]
heuristic-name = alergia
data-name = alergia_data
symbol_count = 5
satdfabound = 2000
state_count = 5
sinkcount = 5
extrapar = 0.01
largestblue = 1
finalred = 0
extend = 0
lowerbound = 0
finalprob = 1
mcollector = -1
mergelocal = -1
printwhite = 0
printblue = 0
markovian = 2
sinkson = 0
```

Figure A.1: Configuration for the bigram model

[1] Available on Bitbucket: https://bitbucket.org/chrshmmmr/dfasat/src/master/

Convert Sink

```
[default]
heuristic-name = alergia
data-name = alergia_data
symbol_count = 5
satdfabound = 2000
state_count = 5
sinkcount = 5
extrapar = 0.01
largestblue = 1
finalred = 0
extend = 0
lowerbound = 0
finalprob = 1
mcollector = -1
mergelocal = -1
printwhite = 0
printblue = 0
sinkson = 1
convertsinks = 1
markovian = 1
```

Figure A.2: Configuration for converting sinks

Markov

```
[default]
heuristic-name = alergia
data-name = alergia_data
symbol_count = 5
satdfabound = 2000
state_count = 5
sinkcount = 5
extrapar = 0.01
largestblue = 1
finalred = 0
extend = 0
lowerbound = 0
finalprob = 1
mcollector = -1
mergelocal = -1
printwhite = 0
printblue = 0
markovian = 1
sinkson = 0
```

Figure A.3: Configuration for the base Markovian model

No sink merging

```
[default]
heuristic-name = alergia
data-name = alergia_data
symbol_count = 5
satdfabound = 2000
state_count = 5
sinkcount = 5
extrapar = 0.01
largestblue = 1
finalred = 0
extend = 0
lowerbound = 0
finalprob = 1
mcollector = -1
mergelocal = -1
printwhite = 0
printblue = 0
mergesinks = 0
sinkson = 1
markovian = 1
```

Figure A.4: Configuration for not merging sinks

Original

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

Figure A.5: The original configuration as used by [31]

Search

```
[default]
heuristic-name = alergia
data-name = alergia_data
symbol_count = 5
satdfabound = 2000
state_count = 5
sinkcount = 5
extrapar = 0.01
largestblue = 1
finalred = 0
extend = 0
lowerbound = 0
finalprob = 1
mcollector = -1
mergelocal = -1
printwhite = 0
printblue = 0
mode = search
sinkson = 1
mergesinks = 1
mergesinkscore = 1
searchsinks = 1
markovian = 1
```

Figure A.6: Configuration for the search mode

B

State Machine Models

Figure B.1: State machine generated using the 'bigram' configuration.

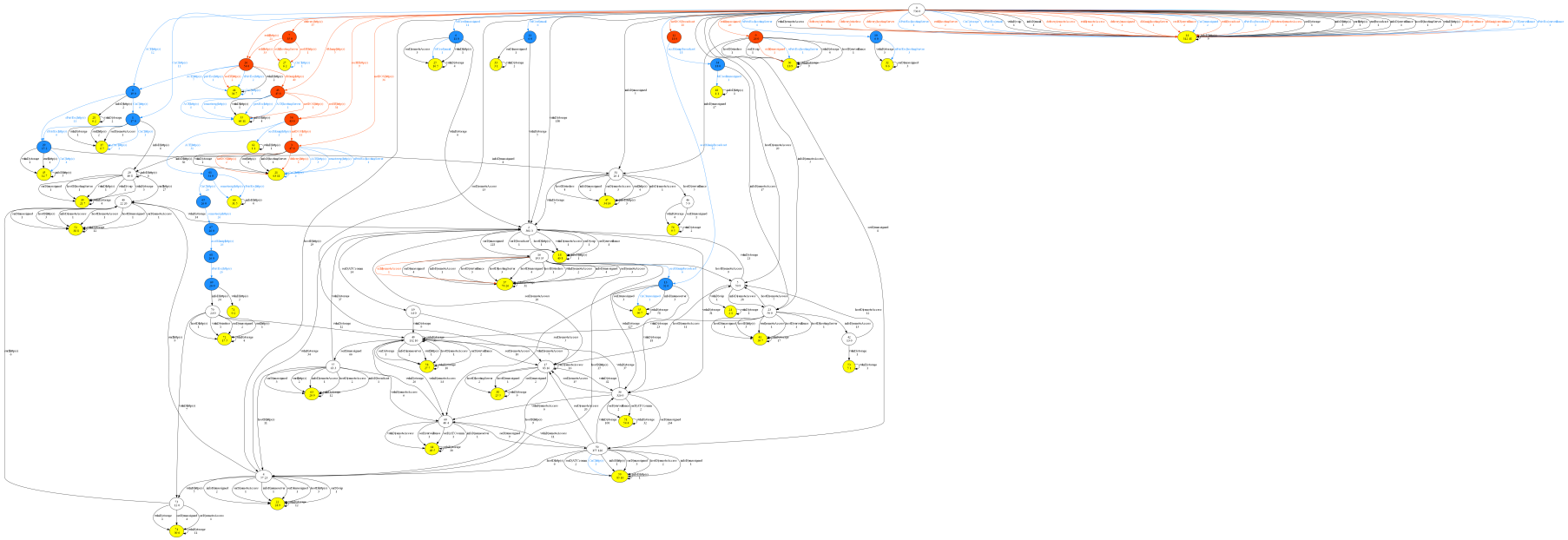


Figure B.2: State machine generated using the 'convert sink' configuration. For the sake of simplicity, one one self-loop per converted sink state is rendered. In the full model, all sinks (show in yellow) feature multiple self-loops with different labels.

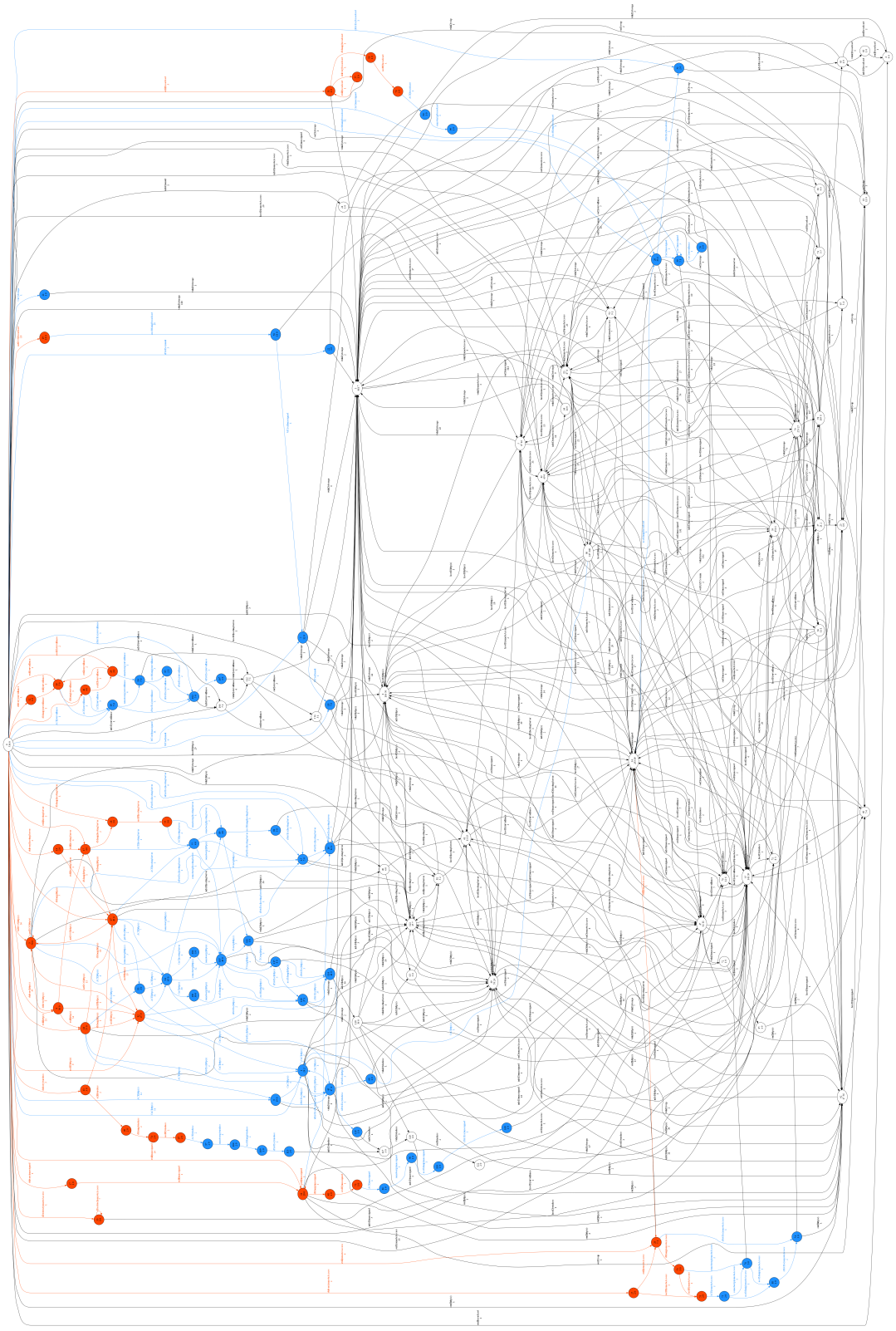


Figure B.3: State machine generated using the 'Markov' configuration.

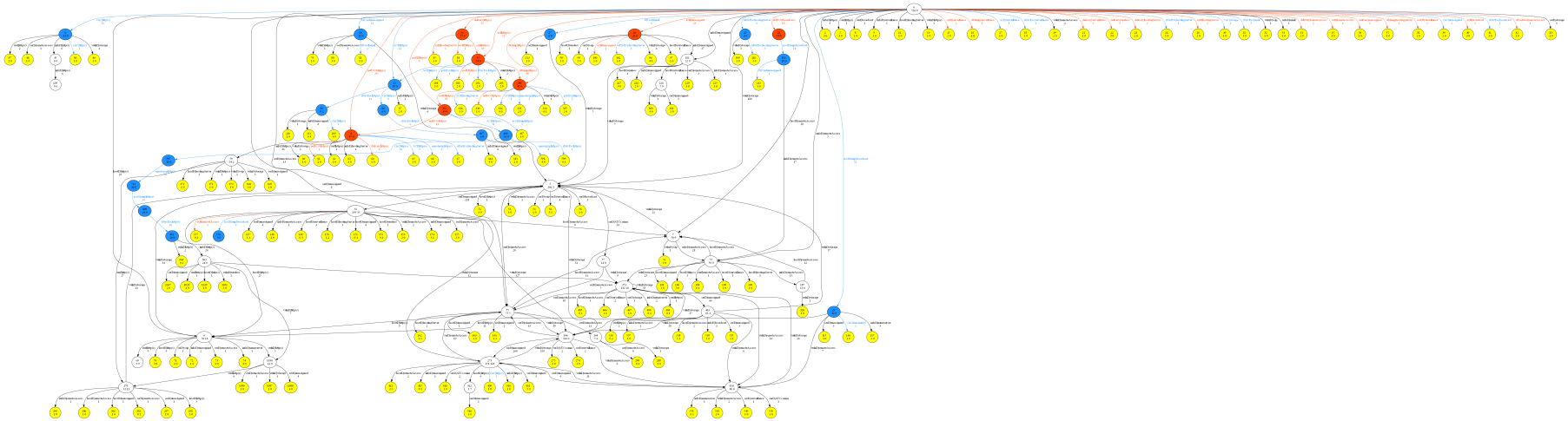


Figure B.4: State machine generated using 'the no sink merge' configuration.

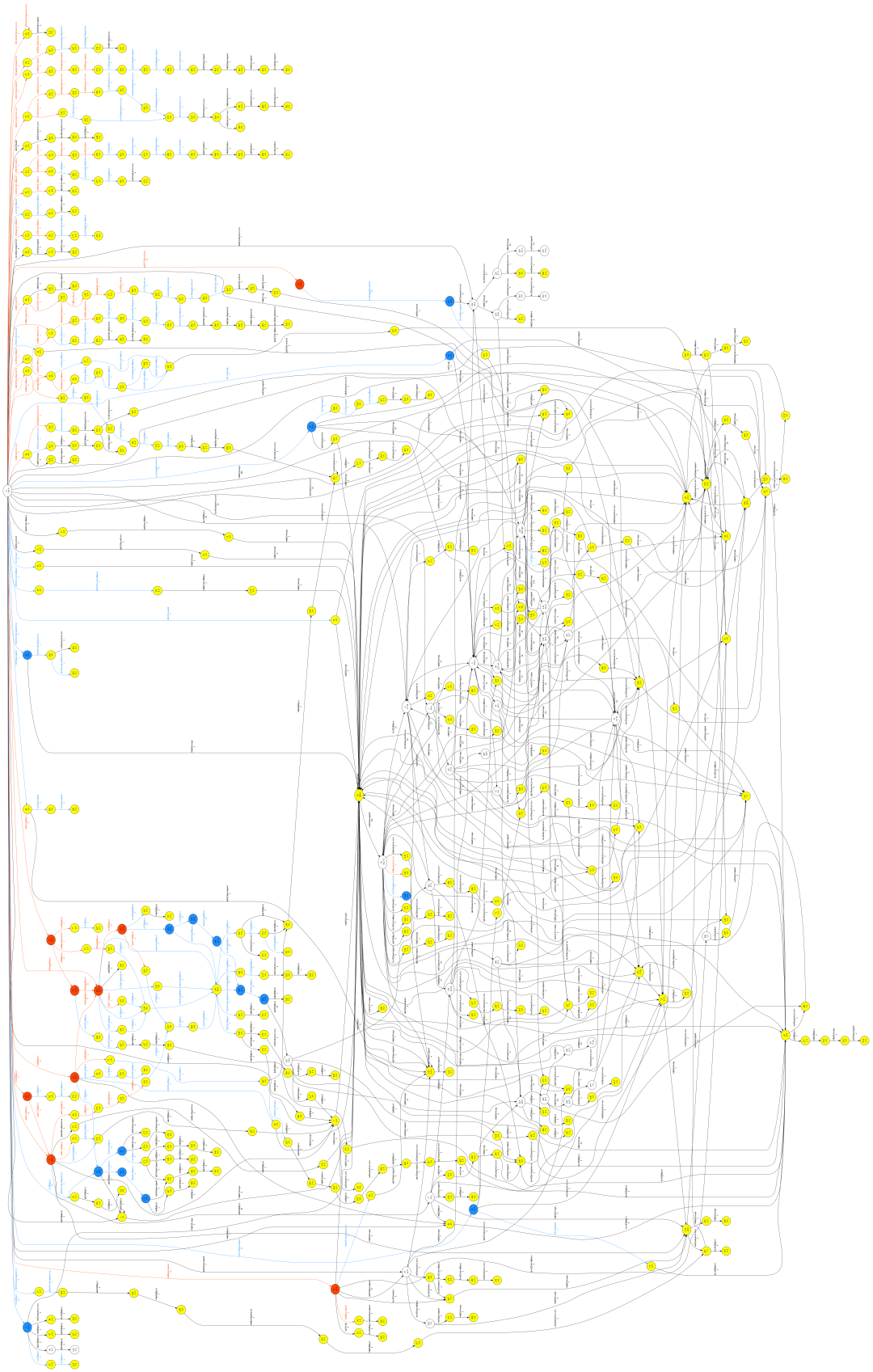


Figure B.5: State machine generated using the 'original' configuration.

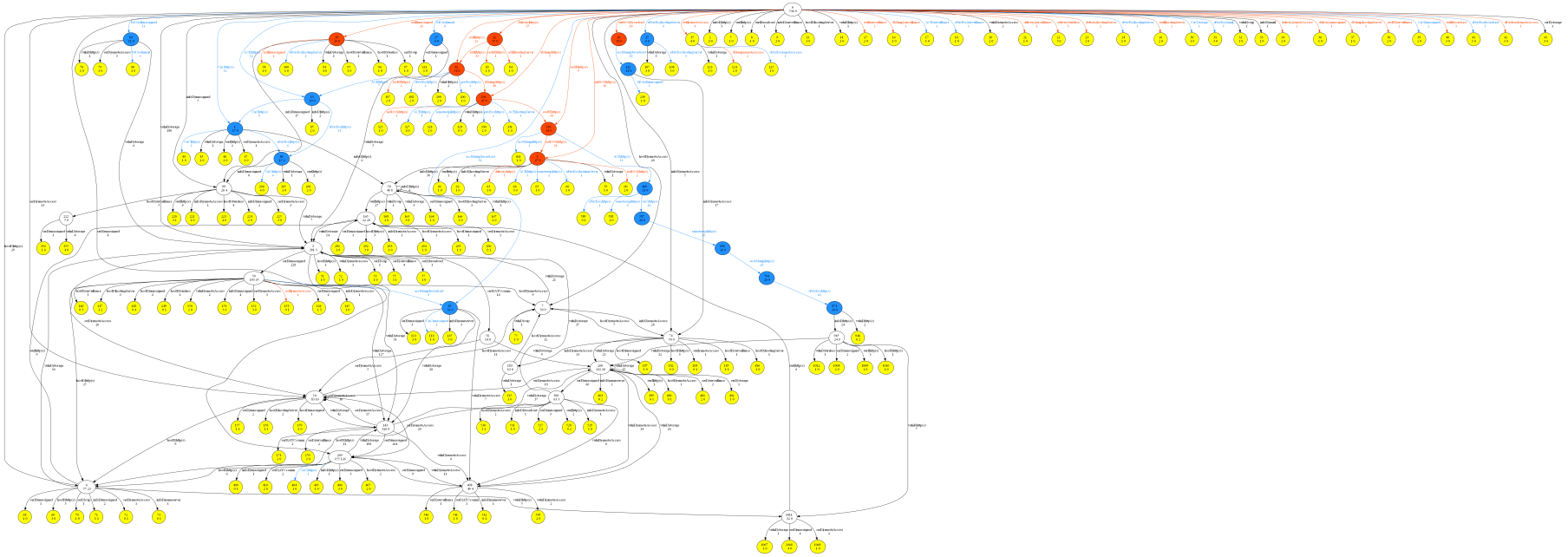


Figure B.6: State machine generated using the 'search' configuration.

Bibliography

[1] Wil Aalst. Verification of Workflow nets. pages 407–426, January 1997. ISBN 978-3-540-63139-2.

[2] Wil Aalst, A. Weijters, and Laura Măruşter. Workflow Mining: Discovering Process Models from Event Logs. Knowledge and Data Engineering, IEEE Transactions on, 16:1128–1142, October 2004. doi: 10.1109/TKDE.2004.47.

[3] Wil Aalst, Arya Adriansyah, and Boudewijn Dongen. Replaying History on Process Models for Conformance Checking and Performance Analysis. WIREs Data Mining and Knowledge Discovery, 2:182–192, March 2012. doi: 10.1002/widm.1045.

[4] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst. Conformance Checking Using Cost-Based Fitness Analysis. In 2011 IEEE 15th International Enterprise Distributed Object Computing Conference, pages 55–64, August 2011. doi: 10.1109/EDOC.2011.12. ISSN: 1541-7719.

[5] Arya Adriansyah, Jorge Munoz-Gama, Josep Carmona, Boudewijn Dongen, and Wil Aalst. Alignment Based Precision Checking. volume 132, September 2012. doi: 10.1007/978-3-642-36285-9_15.

[6] Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, and Giorgio Bruno. Automated discovery of structured process models from event logs: The discover-and-structure approach. Data & Knowledge Engineering, 117, April 2018. doi: 10.1016/j.datak.2018.04.007.

[7] Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, Andrea Marrella, Massimo Mecella, and Allar Soo. Automated Discovery of Process Models from Event Logs: Review and Benchmark. arXiv:1705.02288 [cs], January 2018. URL http://arxiv.org/abs/1705.02288. arXiv: 1705.02288.

[8] Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, and Artem Polyvyanyy. Split miner: automated discovery of accurate and simple business process models from event logs. Knowledge and Information Systems, 59, May 2019. doi: 10.1007/s10115-018-1214-x.

[9] Rafael Carrasco and Jose Oncina. Learning Stochastic Regular Grammars by Means of a State Merging Method. November 2002. ISBN 978-3-540-58473-5. doi: 10.1007/3-540-58473-0_144.

[10] Sean Carlisto de Alvarenga, Sylvio Barbon, Rodrigo Sanches Miani, Michel Cukier, and Bruno Bogaz Zarpelão. Process mining and hierarchical clustering to help intrusion alert visualization. Computers & Security, 73:474–491, March 2018. ISSN 0167-4048. doi: 10.1016/j.cose.2017.11.021. URL http://www.sciencedirect.com/science/article/pii/S0167404817302584.

[11] Jochen De Weerdt, Manu De Backer, Jan Vanthienen, and Bart Baesens. A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Information Systems, 37(7):654–676, November 2012. ISSN 0306-4379. doi: 10.1016/j.is.2012.02.004. URL https://www.sciencedirect.com/science/article/pii/S0306437912000464.

[12] Giuseppe Greco, Antonella Guzzo, Luigi Pontieri, and Domenico Saccà. Discovering Expressive Process Models by Clustering Log Traces. 18:1010–1027, September 2006. doi: 10.1109/TKDE.2006.123.

[13] Christian Albert Hammerschmidt, Sicco Verwer, Qin Lin, and Radu State. Interpreting Finite Automata for Sequential Data. arXiv:1611.07100 [cs, stat], November 2016. URL http://arxiv.org/abs/1611.07100. arXiv: 1611.07100.

[14] Wassily Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. Journal of the American Statistical Association, 58(301):13–30, March 1963. ISSN 0162-1459. doi: 10.1080/01621459.1963.10500830. URL https://www.tandfonline.com/doi/abs/10.1080/01621459.1963.10500830. Publisher: Taylor & Francis _eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1963.10500830.

[15] Hao Hu, Jing Liu, Yuchen Zhang, Yuling Liu, Xiaoyu Xu, and Jinglei Tan. Attack scenario reconstruction approach using attack graph and alert data mining. Journal of Information Security and Applications, 54, October 2020. ISSN 2214-2126. doi: 10.1016/j.jisa.2020.102522. URL http://www.sciencedirect.com/science/article/pii/S2214212619310002.

[16] A. Kenyon, L. Deka, and D. Elizondo. Are public intrusion datasets fit for purpose characterising the state of the art in intrusion event datasets. Computers & Security, 99:102022, December 2020. ISSN 0167-4048. doi: 10.1016/j.cose.2020.102022. URL https://www.sciencedirect.com/science/article/pii/S0167404820302959.

[17] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In Vasant Honavar and Giora Slutzki, editors, Grammatical Inference, Lecture Notes in Computer Science, pages 1–12, Berlin, Heidelberg, 1998. Springer. ISBN 978-3-540-68707-8. doi: 10.1007/BFb0054059.

[18] S. Leemans. Robust process mining with guarantees. In BPM, 2018.

[19] Sander Leemans, Dirk Fahland, and Wil Aalst. Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour. volume 171, pages 66–78, May 2014. doi: 10.1007/978-3-319-06257-0_6.

[20] Sander Leemans, Dirk Fahland, and Wil Aalst. Discovering Block-Structured Process Models from Incomplete Event Logs. June 2014. ISBN 978-3-319-07733-8. doi: 10.1007/978-3-319-07734-5_6.

[21] Sander Leemans, Dirk Fahland, and Wil Aalst. Scalable Process Discovery with Guarantees. volume 214, pages 85–101, June 2015. doi: 10.1007/978-3-319-19237-6_6.

[22] Sander Leemans, Dirk Fahland, and Wil Aalst. Using Life Cycle Information in Process Discovery. volume 256, pages 204–217, July 2016. ISBN 978-3-319-42886-4. doi: 10.1007/978-3-319-42887-1_17.

[23] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. In José-Manuel Colom and Jörg Desel, editors, Application and Theory of Petri Nets and Concurrency, Lecture Notes in Computer Science, pages 311–329, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-38697-8. doi: 10.1007/978-3-642-38697-8_17.

[24] Sander J. J. Leemans, Niek Tax, and Arthur H. M. ter Hofstede. Indulpet Miner: Combining Discovery Algorithms. In Hervé Panetto, Christophe Debruyne, Henderik A. Proper, Claudio Agostino Ardagna, Dumitru Roman, and Robert Meersman, editors, On the Move to Meaningful Internet Systems. OTM 2018 Conferences, Lecture Notes in Computer Science, pages 97–115, Cham, 2018. Springer International Publishing. ISBN 978-3-030-02610-3. doi: 10.1007/978-3-030-02610-3_6.

[25] T. J. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, SE-2(4):308–320, December 1976. ISSN 1939-3520. doi: 10.1109/TSE.1976.233837. Conference Name: IEEE Transactions on Software Engineering.

[26] Jan Mendling. Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness., volume 6. January 2008. ISBN 978-3-540-89223-6. doi: 10.1007/978-3-540-89224-3. Journal Abbreviation: Lecture Notes in Business Information Processing Publication Title: Lecture Notes in Business Information Processing.

[27] Stephen Moskal and Shanchieh Jay Yang. Cyberattack Action-Intent-Framework for Mapping Intrusion Observables. arXiv:2002.07838 [cs], February 2020. URL http://arxiv.org/abs/2002.07838. arXiv: 2002.07838.

[28] Nuthan Munaiah, Justin Pelletier, Shau-Hsuan Su, S Yang, and Andrew Meneely. A Cybersecurity Dataset Derived from the National Collegiate Penetration Testing Competition. January 2019.

[29] J. Munoz-Gama and J. Carmona. Enhancing precision in Process Conformance: Stability, confidence and severity. In 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), pages 184–191, April 2011. doi: 10.1109/CIDM.2011.5949451.

[30] Jorge Munoz-Gama and Josep Carmona. A Fresh Look at Precision in Process Conformance. volume 6336, pages 211–226, September 2010. ISBN 978-3-642-15617-5. doi: 10.1007/978-3-642-15618-2_16.

[31] Azqa Nadeem, Sicco Verwer, Stephen Moskal, and Shanchieh Jay Yang. Sage: Intrusion alert-driven attack graph extractor. KDD Workshop on Artificial Intelligence-enabled Cybersecurity Analytics (AI4Cyber).

[32] Object Management Group (OMG). Business process model and notation, version 2.0.2, 2013. URL https://www.omg.org/spec/BPMN/.

[33] Xinming Ou, Sudhakar Govindavajhala, and Andrew Appel. MulVAL: A logic-based network security analyzer. pages 8–8, July 2005.

[34] Carl Adam Petri. Kommunikation mit Automaten. http://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/pdf/diss_petri.pdf, 1962. URL https://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/.

[35] Artem Polyvyanyy. Structuring process models. PhD thesis, March 2012.

[36] Elvira Rolón, Francisco Ruiz, Felix Garcia, and Mario Piattini. Applying Software Metrics to evaluate Business Process Models. CLEI Electron. J., 9, June 2006. doi: 10.19153/cleiej.9.1.5.

[37] Sebastian Roschke, Feng Cheng, and Christoph Meinel. A New Alert Correlation Algorithm Based on Attack Graph. In Álvaro Herrero and Emilio Corchado, editors, Computational Intelligence in Security for Information Systems, Lecture Notes in Computer Science, pages 58–67, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-21323-6. doi: 10.1007/978-3-642-21323-6_8.

[38] Anne Rozinat and Wil Aalst. Conformance checking of processes based on monitoring real behavior. Information Systems, 33:64–95, March 2008. doi: 10.1016/j.is.2007.07.001.

[39] Anne Rozinat, C Günther, A. Weijters, and Wil Aalst. Towards an evaluation framework for process mining algorithms. Reactivity of Solids, January 2007.

[40] Michael Sipser. Introduction to the Theory of Computation. Course Technology, Boston, MA, third edition, 2013. ISBN 113318779X.

[41] W. Van Der Aalst, A. Adriansyah, A.K.A. De Medeiros, F. Arcieri, T. Baier, T. Blickle, J.C. Bose, P. Van Den Brand, R. Brandtjen, J. Buijs, A. Burattin, J. Carmona, M. Castellanos, J. Claes, J. Cook, N. Costantini, F. Curbera, E. Damiani, M. De Leoni, P. Delias, B.F. Van Dongen, M. Dumas, S. Dustdar, D. Fahland, D.R. Ferreira, W. Gaaloul, F. Van Geffen, S. Goel, C. Günther, A. Guzzo, P. Harmon, A. Ter Hofstede, J. Hoogland, J.E. Ingvaldsen, K. Kato, R. Kuhn, A. Kumar, M. La Rosa, F. Maggi, D. Malerba, R.S. Mans, A. Manuel, M. McCreesh, P. Mello, J. Mendling, M. Montali, H.R. Motahari-Nezhad, M. Zur Muehlen, J. Munoz-Gama, L. Pontieri, J. Ribeiro, A. Rozinat, H. Seguel Pérez, R. Seguel Pérez, M. Sepúlveda, J. Sinur, P. Soffer, M. Song, A. Sperduti, G. Stilo, C. Stoel, K. Swenson, M. Talamo, W. Tan, C. Turner, J. Vanthienen, G. Varvaressos, E. Verbeek, M. Verdonk, R. Vigo, J. Wang, B. Weber, M. Weidlich, T. Weijters, L. Wen, M. Westergaard, and M. Wynn. Process mining manifesto. 99 LNBIP:169–194, 2012. doi: 10.1007/978-3-642-28108-2_19.

[42] Seppe K. L. M. vanden Broucke and Jochen De Weerdt. Fodina: A robust and flexible heuristic process discovery technique. Decision Support Systems, 100:109–118, August 2017. ISSN 0167-9236. doi: 10.1016/j.dss.2017.04.005. URL https://www.sciencedirect.com/science/article/pii/S0167923617300647.

[43] Sicco Verwer and Christian A. Hammerschmidt. flexfringe: A Passive Automaton Learning Package. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 638–642, Shanghai, September 2017. IEEE. ISBN 978-1-5386-0992-7. doi: 10.1109/ICSME.2017.58. URL http://ieeexplore.ieee.org/document/8094471/.

[44] A. Weijters. Process Mining: Extending the alpha-algorithm to Mine Short Loops. June 2004.

[45] A. J. M. M. Weijters and J. T. S. Ribeiro. Flexible Heuristics Miner (FHM). In 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), pages 310–317, April 2011. doi: 10.1109/CIDM.2011.5949453.

[46] AJMM Weijters, Wil MP van Der Aalst, and AK Alves De Medeiros. Process mining with the heuristics miner-algorithm. Technische Universiteit Eindhoven, Tech. Rep. WP, 166:1–34, 2006.

[47] Lijie Wen, Wil Aalst, Jianmin Wang, and Jiaguang Sun. Mining process models with non-free-choice Constructs. Data Min. Knowl. Discov., 15:145–180, October 2007. doi: 10.1007/s10618-007-0065-y.

[48] Lijie Wen, Jianmin Wang, Wil Aalst, Biqing Huang, and Jiaguang Sun. Mining Process Models with Prime Invisible Tasks. Data & Knowledge Engineering, 69:999–1021, June 2010. doi: 10.1016/j.datak.2010.06.001.