UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Privacy-Preserving Process Mining

Datenschutzfreundliche Veröffentlichung von Business Process Log-Daten

Bachelorarbeit

verfasst am
Institut für IT-Sicherheit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Max Schulze

ausgegeben und betreut von
Prof. Dr. Esfandiar Mohammadi
M.Sc. Yorck Zisgen

mit Unterstützung von
Prof. Dr. Agnes Koschmider

Lübeck, den 16. November 2022

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

Max Schulze

Abstract

Process-Mining is a discipline in which event logs are mined to generate process models. With the use of these models, it is possible to improve processes and workflows in various ways. Since these models are a pure, unfiltered representation of the underlying event log, privacy concerns are of special interest.

To face this problem, we implement two approximations of existing algorithms combined with differential privacy.

The first approach uses the filtering function of the Heuristic-Miner. This function filters the transition of the process model based on their weighting. We manipulate this function to measure the transition score over the whole event log and filter by the AboveThreshold mechanism, using a threshold τ. Since the resulting graph is filtered using a differential privacy mechanism, it is a privacy-preserving representation of a process.

In the second approach, the Inductive Miner is modified to generate a privacy-preserving Process-Structure-Tree. For this, the selection process of a cut is randomised. This randomisation is done by counting the number of traces that voted for a specific cut. Using these scores in the Exponential-Mechanism, a cut is randomly selected, with a probability scaling to the scores.

In contrast to related works, we are not modifying the underlying data sets or creating a new algorithm for process mining. Instead, we implement approximations of existing process mining algorithms, which provide a differential private representation of event logs and ensure that privacy concerning data is not leaked. Since the underlying data is not changed, it can be reused for further analysis.

Zusammenfassung

Process-Mining ist eine Disziplin, bei der Event-Logs ausgewertet werden, um Prozessmodelle zu erstellen. Mit Hilfe dieser Modelle ist es möglich, Prozesse und Arbeitsabläufe auf verschiedene Weise zu verbessern. Da diese Modelle eine reine, ungefilterte Darstellung des zugrundeliegenden Event-Logs sind, ist der Datenschutz von besonderem Interesse.

Um diesem Problem zu begegnen, implementieren wir zwei Approximationen von existierenden Algorithmen in Kombination mit Differential Privacy.

Der erste Ansatz nutzt die Filterfunktion des Heuristic-Miners. Diese Funktion filtert die Transitionen des Prozessmodells auf der Grundlage ihrer Gewichtung. Wir manipulieren diese Funktion, um die Gewichtungen über das gesamte Event-Log zu messen und filtern diese mit dem AboveThreshold-Mechanismus unter Verwendung eines Thresholds τ. Da der resultierende Graph mit Hilfe von Differential Privacy gefiltert wird, handelt es sich um eine datenschutzkonforme Darstellung eines Prozesses.

Im zweiten Ansatz wird der Inductive Miner modifiziert, um einen datenschutzfreundlichen Process-Structure-Tree zu erzeugen. Dazu wird der Auswahlprozess eines Schnitts randomisiert. Diese Randomisierung erfolgt durch Zählen der Anzahl der Traces, die für einen bestimmten Schnitt gestimmt haben. Unter Verwendung dieser Werte im Exponential-Mechanismus wird ein Schnitt zufällig ausgewählt, mit einer Wahrscheinlichkeit, die mit den Werten skaliert.

Im Gegensatz zu verwandten Arbeiten modifizieren wir nicht die zugrundeliegenden Datensätze oder entwickeln einen neuen Algorithmus für Process Mining. Stattdessen implementieren wir Approximationen von bestehenden Process Mining Algorithmen, die eine Datenschutzkonforme Darstellung von Event-Logs bieten und sicherstellen, dass die Privatsphäre der Daten nicht beeinträchtigt wird. Da die zugrunde liegenden Daten nicht verändert werden, können diese für weitere Analysen wiederverwendet werden.

Contents

List of Figures

1

Introduction

Process-Mining is a discipline that creates models that are a graphical visualization of an event log, which can be used to improve the efficiency and quality of processes. These models display the event log and its structure without any editing. Because of this, sharing such models is not always possible since an adversary could gain information about any individual represented in such an event log or even about sensitive workflows that are described.

There already exist approaches that face this problem [10, 11, 7]. However, while these works are creating entirely new systems to create a privacy-preserving event log or model, we are using existing and used process mining algorithms. We are doing this since we want to create an algorithm which produces accurate and privacy-preserving process models.

We use the ideas behind the Heuristic-Miner [15] and Inductive-Miner [13] to create approximations of these algorithms, to mine an event log correctly and privacy-preserving. To make these models privacy-preserving, we are using differential privacy [6], giving us privacy guarantees while the resulting models are as accurate as possible.

For the modification of the Heuristic-Miner [15], we look at the creation of a Directly-Follows-Graph. This graph is then so created by us that we filter transitions, with differential privacy, that is not fulfilling a specific property. The Heuristic-Miner also filters transitions. However, this filtering aims to filter out transitions potentially generated through the noise in the event log. So with our algorithm, we create a filtering system that preserves differential privacy and uses the technique of the Heuristic-Miner to generate a process model. This makes an accurate and privacy-preserving process model.

In addition, we are doing something similar in our approximation of the Inductive-Miner. The model generated through the Inductive-Miner is a so-called Process-Structure-Tree. These trees can be highly adaptive. So it is possible to use this model and generate any wanted visualization with it, such as Petri-nets or Directly-Follows-Graphs. Because of this property, the Inductive-Miner has no filter function like the Heuristic-Miner and shows the structure of an event log in its complete form. To minimize the leakage, we implement a mechanism to modify the cut selection of the Inductive-Miner. This modification results in a Process-Structure-Tree that is privacy-preserving and still shows the overall structure of an event log.

In summary, we create two algorithms that generate two different process models. However, both are privacy-preserving and still hold the property of showing the structure and workflow of an event log.

1.1 Contribution

Our contribution to this thesis is as follows

1. We implement two algorithms that generate process models that are both privacy-preserving and as accurate as possible
2. We test our algorithms with the use of three synthetically generated event logs [17] and the Sepsis data set [8]
3. The proposed algorithms are approximations of existing mining algorithms. Therefore further improvements can be adapted easily

1.2 Structure

In chapter 2, we are talking about work related to ours. We define terms in chapter 3 and explain existing algorithms used throughout the thesis. Chapter 4 gives a detailed overview of our motivation for the approaches and describes possible thread models. In chapter 5, we are looking at the idea behind our approaches and how we implemented them. In chapter 6, we evaluate our systems and discuss the usage based on the experimental results. In chapter 7, we are concluding the whole work and looking at future work.

2

Related Work

Process mining uses several algorithms to mine event logs, like presented in [1, 13, 15]. These algorithms fulfil the purpose of mining so that an event log is reconstructed based on the underlying data [3]. Due to the focus on constructing a model that shows the behaviour of a process, privacy concerning data is not secured in these algorithms.

The Heuristic Miner [15] is the closest standard mining algorithm to achieve some privacy-preserving model. It uses a threshold between -1 and 1 to filter out transitions that most likely occur due to potential noise in the event log and are, therefore, rare. The filtering is possible since the Heuristic Miner calculates the dependency relations between two events. Should the computed value be under a defined threshold τ, the corresponding transition is filtered out. Since the filtering is not randomised, the generated Directly-Follows-Graphs are not privacy-preserving. Therefore the Heuristic Miner is neither.

[14] states that an event log contains sensitive information and has a closer look at the re-identification risk of individuals in an event log. The proposed approach can measure how unique data is. Therefore, the risk of identifying a specific individual can be determined.

Several approaches exist for privacy-preserving process mining to face the risk of identifying individuals in an event log [7, 10, 11].

In [7], the authors focus on creating a privacy-preserving Directly-Follows-Graph. This graph uses the weighting of transitions to visualise different scenarios, for example, the average time from one event to another or the number of times one event follows after the other. Due to the closeness of these weightings to the actual data, [7] aims to do privacy-preserving weighting using differential privacy. Therefore their generated weighting system generates weights that do not leak sensitive information about the event log. Nevertheless, this weighting is insufficient as soon as the underlying data set is too small.

[10] and [11] focus on generating event logs that are privacy-preserving rather than developing a privacy-preserving visualisation. [11] focuses on modifying the XES standard to generate a more private one. This modification of the XES-Standard faces the privacy problem concerning metadata in event logs. However, this focuses only on one standard for an event log system and is modifying the event log, which might not be wanted. [10] focuses on a general approach of generating a privacy protection model for event logs. This approach is the closest one to our approach. While this approach uses different privacy stages in which data leakage is possible, we are implementing approximations of already existing process mining algorithms, which ensures a privacy-preserving and accurate visualisation.

3

Preliminaries

In this chapter, we focus on the two main topics of this work. We start with an overview of process mining and look closely at some algorithms from this field. After that, we will define differential privacy, and we have a look into relevant theorems.

3.1 Process Mining

Process Mining is a discipline that aims to analyse and reconstruct processes based on event logs containing process execution data, or simple workflows. The output of such mining algorithms is a graphical visualisation, such as a Petri net, a Directly-Follows-Graph or a workflow net. Therefore Process Mining is used to gain an overview of working processes [1, 2, 3, 16].

3.1.1 Event Log

An event log E is a data set with different columns which describe a process. While each data set has a different length and number of columns, three necessary ones are needed for every event log to make mining possible. These columns are **case id**, **activity name** and **timestamp**, these names may differ, but the underlying meaning is always the same [3, 10].

1. Each event $e \in E$ refers to a *case id*, which is a process instance. Every *case id* is a defined trace $T \in E$. If we have the *case id's* [1, 2, 3, 4], we have four traces in E. *case id's* can be every form of enumeration
2. The *activity name* is the defined name of each event $e \in E$
3. The *timestamp* refers to the time of execution of an event $e \in E$, this also defines the total order of events

3.1.2 Heuristic Miner

The Heuristic Miner mines an event log E by viewing the order of activities within a trace T. In each trace, T, the activities can have different dependency relations. To define these relations let $a, b \in T$:

1. $a >_E b$ iff a is followed by b
2. $a \rightarrow_E b$ iff a is always followed by b, and b is never followed by a
3. $a \#_E b$ iff a and b never follow each other directly
4. $a||_E b$ iff $a >_E b$ and $b >_E a$

These are the fundamental relations we use throughout the thesis. For more details, see [15]. To create a Directly-Follows-Graph, the Heuristic Miner calculates a frequency-based value to check how specific the dependency relation between two activities a and b is. Let $a, b \in T$, then $|a >_E b|$ defines the number of times $a >_E b$ occurs in the event log E. With this, we can calculate a value that determines how specific a dependency relation is. For this, the two equations below are needed, for additional information see [15].

$$a \Rightarrow_E b = \left(\frac{|a >_E b| - |b >_E a|}{|a >_E b| + |b >_E a| + 1} \right) \tag{3.1}$$

$$a \Rightarrow_E a = \left(\frac{|a >_E a|}{|a >_E a| + 1} \right) \tag{3.2}$$

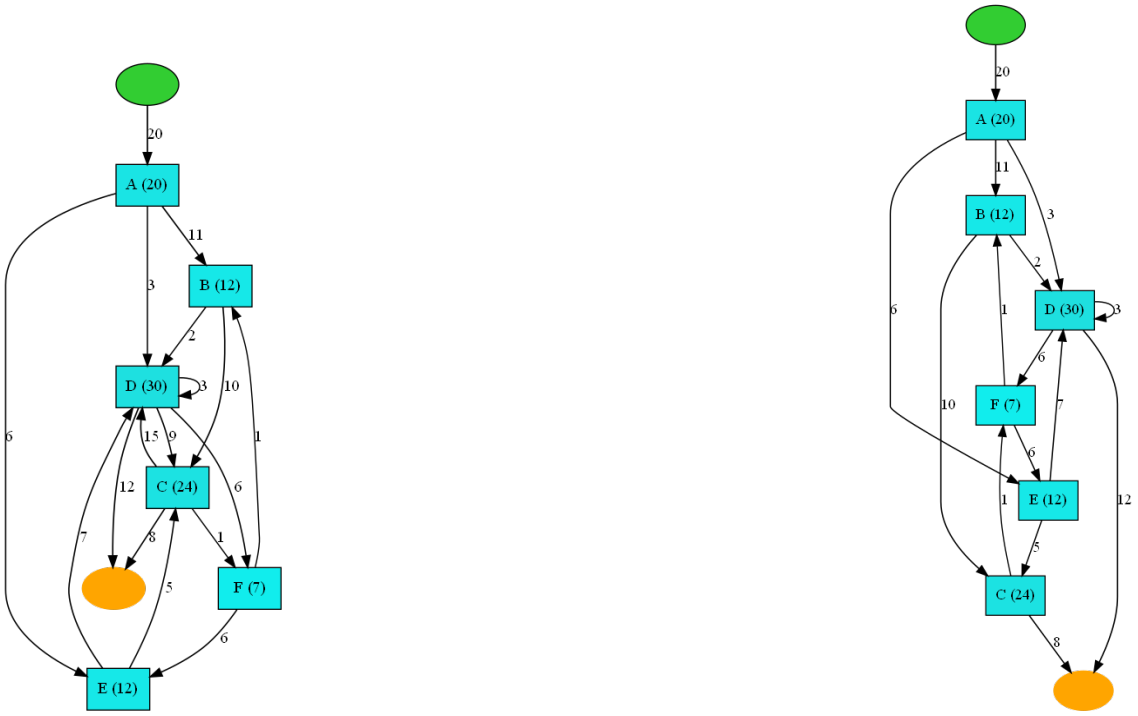
Equation 3.1 determines the dependency of two different activities a and b, with $a \neq b$ and b being the successor of a. On the other hand, equation 3.2 determines the dependency of a self-referencing event a, a so-called self-loop. The calculated value of both equations is always between -1 and 1. With a high positive value, we can presume that a dependency relation between event a and b exists.

To create a Directly-Follows-Graph, all traces $T_i \in E$ are viewed, and similar traces are summarised. Let E contain a finite number of traces i, then we have traces $T_1, ..., T_i \in E$. For each trace $T_n \mid n \in L$, we look at the successor of an event $a_m \mid m \in$ num of activities in T_n. With these, we generate a Trace-Log $L = [T_1, ..., T_i]$ that does not contain duplicates due to the summarising of traces that occur more than once in E. A summarised trace is defined as T^j, with $1 < j <= i \mid i =$ number of traces. With this information, tuples are created that display a_m and its successor a_{m+1}. These tuples represent the transition from one event to another. Every tuple is stored once, alongside a value representing the number of occurrences, of this tuple, over all traces $\in L$.

For example, if we have twenty traces with six activities, a Trace-Log could be $L = [ABDC^2, ABCD^3, AECD^5, AEDCFBCD, ADD^3, ABCDFEDC^6]$. The list of tuples t would then be $t = [(A,B) : 11; (A,E) : 6; (A,D) : 1; (B,C) : 10; (B,D) : 2; (C,D) : 15; (C,F) : 1; (D,C) : 9; (D,D) : 3; (D,F) : 6; (E,C) : 5; (E,D) : 7]$. With this, the dependency relations are calculated using the equations 3.1 and 3.2. The results are shown in the matrix below.

$\Rightarrow_E$	A	B	C	D	E	F
A	0.0	0.917	0.0	0.75	0.857	0.0
B	0.0	0.0	0.909	0.666	0.0	0.0
C	0.0	0.0	0.0	0.05	0.0	0.3
D	0.0	0.0	-0.24	0.75	0.0	0.875
E	0.0	0.0	0.83	0.875	0.0	0.0
F	0.0	0.5	0.0	0.0	0.875	0.0

With this dependency matrix and the list of tuples t, the Directly-Follows-Graph is created, using a threshold τ to filter for unlikely transitions.



(a) The Directly-Follows-Graph without a τ being applied

(b) The Directly-Follows-Graph with $\tau = 0.5$ applied

Figure 3.1: The resulting Directly-Follows-Graph of the example Trace-Log L, without a threshold τ, Figure 3.1a, and with a threshold $\tau = 0.5$, Figure 3.1b.

3.1.3 Inductive Miner

The Inductive Miner mines an Event-Log E by viewing all predecessors and successors of an event a over all traces $T \in E$, where a occurs. That is done since the Inductive Miner creates a Process-Structure-Tree and not a Directly-Follows-Graph like the Heuristic Miner. A Process-Structure-Tree shows the overall structure of an Event-Log and not the relations of the individual events.

To create a Process-Structure-Tree, we first define the relations below, with $a, b \in T_i \mid i$ being any trace $T \in E$.

1. $a >_E b$ iff a is followed by b
2. $a \rightarrow_E b$ iff a is always followed by b, and b is never followed by a
3. $a \#_E b$ iff a and b never follow each other directly
4. $a ||_E b$ iff $a >_E b$ and $b >_E a$

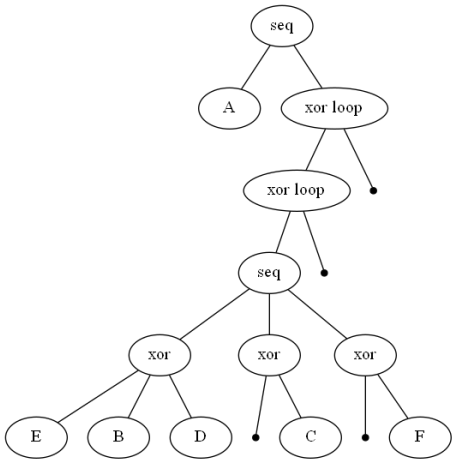
With these relations, we can create cuts. These cuts are used to display the structure

of the E. We define four cuts as follows, with $a, b, c \in T_i$ and $a \neq b \neq c$, for more detail see [13].

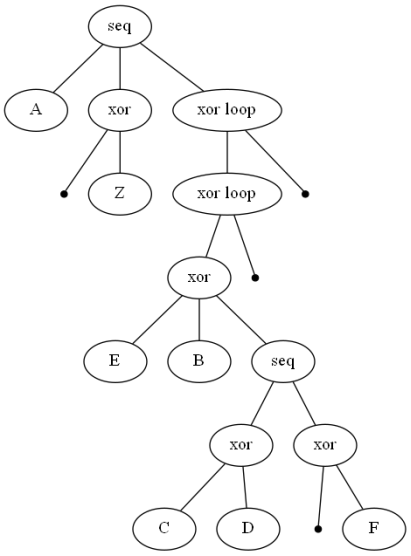
- **SEQUENCE-Cut:** iff $a \rightarrow_E b$, with $a \neq b$, then a SEQUENCE-Cut $(a \rightarrow b))$ exists
- **XOR-Cut:** iff $a \rightarrow_E b$ and $a \rightarrow_E c$, with $a \neq b \neq c$, then a XOR-Cut $(a \oplus (b, c))$ exists
- **AND-Cut:** iff $a >_E b$ and $b >_E a$, with $a \neq b$, then a AND-Cut $(a \bigwedge b)$ exists
- **LOOP-Cut:** iff $a \rightarrow_E b$, $b \rightarrow_E c$ and $c >_E a$, then a LOOP-Cut $(a \circlearrowleft (b, c))$ exists

Due to the property of the Inductive Miner to check the predecessor and successors of an event a over all traces $T \in E$, a single trace can change the cut that is made.

For example, if we have twenty traces and in these, we always have the relation AB, we know that a SEQUENCE-Cut is made. By now adding the twenty-first trace with the relation BA, we do an AND-Cut instead of a SEQUENCE-Cut. Because of this property, the Inductive Miner is prone to noise, as can be seen in Figure 3.2.



(a) The Process-Structure-Tree of an event log without noise



(b) The Process-Structure-Tree of the same event log, expect that one activity got changed in the event log changed

Figure 3.2: The Process-Structure-Trees of the event log used for the Directly-Follows-Graphs in Figure 3.1. Figure 3.2a being the resulting tree of the event log without noise and 3.2b being the resulting tree of the same event log, except one activity got changed.

3.2 Differential Privacy

Differential privacy aims to hide the influence of a single data point on a resulting data set when analysing this set. Therefore it should not be possible for an adversary or any other individual to gain enough information from the resulting data set to be sure about the absence or attendance of a single data point.

More formally, it is assumed that an adversary has access to two neighbouring data sets D and D', with one of them being the output and the other one being the input for a randomised differential private mechanism M. Neighbouring means that the data sets D and D' differ in at most one element.

In this scenario, the adversary has access to the output data set of M as well as to the two neighbouring data sets D and D'. The goal of the adversary is now to identify whether D or D' was the input data set.

Two needed parameters to define the privacy guarantees M can give are ϵ and δ. While ϵ defines the maximum difference between outputs of M, δ describes the probability that elements are not covered by ϵ. It is important to note that the lower the ϵ, the higher the privacy, but also the lower the accuracy of the output data set D of the mechanism M [6].

Definition 3.1 (Differential Privacy). A randomized mechanism $\mathcal{M} : \mathcal{D} \to \mathcal{R}$ is (ϵ, δ)-differentially private, with $\epsilon > 0$ and $\delta \geq 0$, if for all $\mathcal{S} \subseteq \mathcal{R}$ and for all neighboring data sets D, $D' \in \mathcal{D}$:

$$\Pr[\mathcal{M}(D) \in \mathcal{S}] \leq exp(\epsilon) \Pr[\mathcal{M}(D') \in \mathcal{S}] + \delta$$

Important mechanisms of differential privacy that we use throughout the thesis are described below, for further information see [6].

Sensitivity An important aspect of differential privacy is sensitivity. Sensitivity describes the maximum difference between two data sets due to the addition of a single data point. The worst-case difference a single data point can cause is considered to determine the sensitivity.

Counting Queries Another important aspect are counting queries. These are used to get the number of elements which satisfy a predefined property in a data set. For example, if we consider a data set D with n elements, a valid request on this data set could be: "How many elements in D are greater than x?". The output of this request is a counting query, with the condition may changing depending on the task.

Post-Processing Theorem Any output of a differential private mechanism M is immune to post-processing. More formally, any individual without additional knowledge about the differential private data set D cannot generate a function that uses the differential private data set D as input that has a less differential private data set D' as output.

AboveThreshold This mechanism is used to check if an element from a given set is above a defined threshold τ. This threshold defines a value which has tho reached to be fulfilled so that the query can stay in the set of consideration. The first element that fulfils this condition is returned for further computations.

BelowThreshold The BelowThreshold-mechanism is an approximation of the AboveThreshold-mechanism. Instead of checking if an element is above a defined threshold τ it checks if the element is below this threshold.

Exponential-Mechanism The exponential mechanism takes a data set D, some objects O and the results of a utility scoring function u as input. With u, it is counted how many objects from O satisfy property P on D. These computed scores are tied to the corresponding objects in O. The exponential mechanism outputs an object $x \in O$ with the maximum possible utility score. The probability that this x is selected is proportional to $exp(\frac{\epsilon u(D,x)}{2\Delta u})$, with Δu being the sensitivity of the utility function u.

Definition 3.2. The exponential mechanism preserves ϵ-differential privacy if the sensitivity of the utility function is limited.

4

Problem Statement

In this chapter, we look at our motivation for our approaches in more detail, some challenges we have to address and an overview of possible thread models.

4.1 Motivation

Process Mining is a discipline that uses event logs to produce a graphical visualisation called process models. These event logs consist of different entries, with the main needed ones being the **trace id**, the **timestamp** and the **activity name**. With these three values, it is possible to differentiate between all the events in the event log. With these three being the needed part, adding extra specifiers, such as *names* or *gender*, is likely. These extra specifiers are not used for process mining, but when publishing such an event log, an adversary can gain additional information. Due to this, event logs usually are not published. Only their generated models are. [14] also states that the re-identification risk of event logs must not be ignored. A measuring method to calculate the uniqueness of data is proposed to show the relevance of this risk.

Nonetheless, publishing the event log used for specific tasks might be wanted. This could be the case for scientific research to prove the correctness of an approach. Since it is impossible to publish these event logs, as discussed before, there exist techniques to create privacy-preserving event logs [10, 11]. The downside of such event logs is that information is lost, even for the owners of the original event log. [7] creates a Directly-Follows-Graphs that uses a differential private mechanism to manipulate the weighting of the transitions. While this approach does not change the underlying data, it has the problem that with a small amount of underlying data, it is possible to analyse which of the transitions has a small weighting and is, therefore, rarely used.

Due to these problems, we aim to create approximations of already existing process mining algorithms. This helps us to generate process models that are, on the one hand, privacy-preserving and, on the other hand, as accurate as possible. Since we focus on creating process models, the underlying event logs are not manipulated, ensuring that the owner of an event log still has access to the original data and can use it for further work. Additionally, one can approximate the actual event log using the generated process model. Because of the privacy-preserving property of our generated models, the generated event

log can be published without privacy concerns.

The algorithms we want to approximate are the Heuristic-Miner [15] and the [13]. We use techniques from [6] for both of these approximations. The Heuristic-Miner uses a filtering system to filter for transitions with a weighting under a defined limit of $[-1, 1]$. This filter functionality is changed by us using the AboveThreshold-Mechanism, see description 3.2 and [6] so that the filtering is privacy-preserving. For the approximation of the Inductive-Miner, we are manipulating the choice of cuts. This manipulation is done by using a scoring function that counts the number of traces that voted for cut and then using the Exponential-Mechanism, see description 3.2 and [6], to select a cut randomly. The property of the Exponential-Mechanism ensures us that the resulting Process-Structure-Tree is privacy-preserving.

4.2 Challenges

This section describes the challenges we must consider throughout the thesis. These are privacy guarantees and the persistent accuracy of the generated process models.

Privacy We have to create privacy-preserving models. The used ϵ value should stay the same when using our algorithms on different event logs. Additionally, we have to ensure that no additional data is published. Only the data initially used by the two algorithms are processed.

Accuracy While we want to ensure privacy-preserving models, we also have to ensure that the resulting models are accurate. This is important since the output of our algorithms should be used to show how the underlying event log might have looked like. Therefore the used ϵ must be reasonable since this directly influences the accuracy of the output, see section 3.2 and [6].

4.3 Thread Models

This section discusses two possible thread models for our generated process models and the implementations in general. Additionally, we discuss the protection we can assume from different work against these threads.

For both models, we assume that the underlying event log of the process model is not being leaked. Therefore the receiver only has access to the process models generated and to two neighbouring event logs.

Honest In this scenario, the receiver of the process model is honest. For him, it might be possible to gain information about the processes by analysing the model. This analysation is normal and must always be considered when publishing data. Since we are using differential privacy, this scenario is of no concern and is our consideration throughout the whole thesis.

Malicious In this scenario, the process model receiver wants to gain information about the underlying event log actively. For that, a Membership Inference attack can be assumed. This attack is used to decide whether a particular data point is part of the data set. For

that, we assume that the adversary has access to the implementations and can generate new outputs using the neighbouring event logs. Nevertheless, the adversary may have access to the event logs and the implementations and can analyse various data sets from [5, 12]. We know that differential privacy is a good protection against such attacks. Therefore our privacy assumptions are not violated in this thread model.

5

Approach

This chapter looks at two implementations that create differential private process models. In the first section, we discuss a modification of the Heuristic-Miner (See chapter 3.1.2 on page 4). In the second section, we look at the computation of a differential private Process-Structure-Tree using the approximation of the Inductive-Miner (See chapter 3.1.3 on page 6).

5.1 The modified Heuristic-Miner

This section focuses on the first approach of generating a privacy-preserving process model. For that, the concept of the Heuristic-Miner is used. For the corresponding complete pseudocode see chapter A.1 on page iv.

5.1.1 The Idea Behind The Implementation

While the original Heuristic-Miner generates a Directly-Follows-Graph and filters based on the dependency relations, see section 5.1, of a transition. Our approach views the hole Directly-Follows-Graph. Due to that, we filter based on the occurrence, in percent, of a transition in all traces, of the given Event-Log. Therefore we count the occurrence of all transitions and are saving this count alongside the corresponding transition. After that, we use the AboveThreshold- and BelowThreshold-Mechanism (See description 3.2 on page 8) to filter out all transitions that occur in less than threshold τ-percent traces of the Event-Log. Every count is capped at *cap* to minimize the possible leakage of the specific transition.

5.1.2 Implementation

For the implementation, we use Python 3.10 along with the PM4PY library [4].
At first, we are determining the type of the input event log file. It is possible to use the **csv** or **xes** standard. Additionally, it is possible to use a trace log as input. These trace logs have to be stored in an *.txt* file. For clarification, a trace log is a simple documentation of all of the traces from an event log, no other information, besides the traces, is given, see Listing 5.1.

Listing 5.1 main function

```
class main(InputFile i) {
    #determine file type
    if filetype of i == '.csv':
        log = read (event lgo)
        call csvFile(log)
    else if filetype of i == '.xes':
        log = read (event log)
        call traces(log)
    else if filetype of i == '.txt':
        log = read (trace log)
        call traces(log)
    else:
        print('Please choose a valid file')
}
```

Depending on the input a different class is executed. Since **xes** files and trace log files are providing full traces, for which we can traverse each trace in tuples of size two. By doing this we store each tuple in a dictionary along with a score of one, if the tuple is not already stored. If a tuple is already stored, we increment the score by one. By doing this we generate a dictionary that has knowledge about every transition and the amount this transition exists in the event log, see Listing 5.2.

Listing 5.2 Getting the edges from a XES file or a trace log file

```
class traces(EventLog D, Threshold T, Epsilon ϵ1, cap, n) {
    queries = dict()

    for i in range(len(D)) do
        for j in range(i+2, len(D)+2)
            if tuple(i, j) not in queries:
                queries.update({tuple(i, j): 1})
            else:
                queries[tuple(i, j)] += 1
    end

}
```

Should the input file be of type **csv** we first check the case id and the timestamp for each activity and sort the log based on these. After that, we check if the case ids for two consecutive activities are the same. Should this be the case then we store these as a tuple that looks like (first activity, second activity) as a key in a dictionary, if the key does not already exist. The corresponding value is set to one if the tuple is new, otherwise, we increment the corresponding value by one. This is done for all entries in the event log (See Listing 5.3).

Listing 5.3 Getting the edges from a CSV event log file

```
1:  class csvFile(EventLog D, Threshold τ, Epsilon ϵ1, cap, n) {
2:      #define nedded dictionary
3:      edges = dict()
4:
5:      #EventLog is sorted based on the TraceIDs and Timestamps
6:      sort D
7:
8:      #get transitions
9:      for i in range(len(D.TraceID)) do:
10:         if D.TraceID(i) == D.TraceID(i+1) do:
11:             for j in range(i+2, len(D.TraceID)+2) do:
12:                 if tuple(D.ActivityName[i:j]) not in edgesList do:
13:                     edges.update({tuple(D.ActivityName[i:j]): 1})
14:                     break
15:                 else:
16:                  edges[tuple(D.ActivityName[i:j]] += 1
17:                  break
18:             end
19:     end
20: }
```

After generating the dictionaries with the transitions and their values we use the AboveThreshold-Mechanism, see description 3.2 on page 8, to check if a tuple occurs in less than τ-percent of the traces. With τ being the value to determine the overall percentage of a transition to occur in the event log. With the BelowThreshold-Mechanism, we verify the results of the AboveThreshold-Mechanism. The BelowThreshold-Mechanism does the same as the AboveThreshold-Mechanism. However, it returns the index of values below the defined τ.

So the resulting lists of the two mechanisms should be different. A tuple may be in both lists. For such a tuple, we can assume that it is near τ, and therefore the change by ϵ_1 can cause the value to fit the condition, although it does not fit. The AboveThreshold- and BelowThreshold-Mechanism are executed n-times. After the nth iteration, we have two lists of tuples. By comparing these lists, we exclude all tuples to ensure the edge cases are not shown in the Directly-Follows-Graph, see Listing 5.4 for both mechansims. Therefore the remaining tuples of the resulting list from the AboveThreshold-Mechanism, are stored as keys in a dictionary, along with their corresponding count values, which are computed in Listing 5.2 or Listing 5.3.

Listing 5.4 Exectuion of the AboveThreshold- and BelowTHreshold mechanism, to filter the transitions and generate a privacy-preserving Directly-Follows-Graphs

```
1:  class Thresholds(Data dpDict, EventLog D, Threshold τ, Epsilon ϵ1, cap, n) {
2:
3:      edgesList = dpDict.keys()
4:      cQueries = dpDict.values()
5:
6:      #filter the transitions and output the filtered list of transitions -> which
```

```
        queries are above the threshold
7:      a := 1
8:      def AboveThreshold(Database edgesList, Queries cQueries, τ, ε1){
9:          while a < n do:
10:             for Each query i do:
11:                 Let vi = τ,+ Lap(4/ε1)
12:                 if ((numID/numTraces)*100 + vi) ≥ 70% do:
13:                     Output ai = ⊤
14:                     Halt.
15:                 else do:
16:                     Output ai = ⊥
17:                 end
18:             end
19:             a += 1
20:         end
21:     }
22:
23:     #filter the transitions, so that it is known which queries are below the
        threshold
24:     b := 1
25:      def BelowThreshold(Database edgesList, Queries cQuries, τ, ε1){
26:          while b < n do:
27:             for Each query i do:
28:                 Let vi = τ + Lap(4/ε1)
29:                 if ((numID/numTraces)*100 + vi) < 70% do:
30:                     Output ai = ⊤
31:                     Halt.
32:                 else do:
33:                     Output ai = ⊥
34:                 end
35:             end
36:             b += 1
37:         end
38:     }
39: }
```

For each value, we then check if it is below or above the defined *cap*. Should a value be above *cap* then we are capping it at *cap* and update the corresponding entry in the dictionary This resulting dictionary is used to create the Differential Private Directly-Follows-Graph. This algorithm is $n\epsilon_1 \cdot \frac{cap}{\tau \# traces}$ Differential Private, which is proven below, in Theorem 5.1.2.

Lemma 5.1 (The modified Heuristic Miner is $n\epsilon_1 \cdot \frac{cap}{\tau \# traces}$ Differential Private).

Proof. We apply n times the AboveThreshold-Mechanism with noise parameter ϵ_1 to all elements in the Event-Log so that we can exclude edges that were visited in less than τ number of traces in the Event-log D.

For the ith invocation of AboveThreshold, we have a sequence $[(\bot)_{i=1,\dots,m_i-1}, \top]$ (for some $m_i \in \mathbb{N}$). We then compute BelowThreshold for the same queries and noise parameter ϵ_1, until we get a $\bot$, so a sequence looks like $[(\top)_{i=1,\dots,m_{i+1}-1}, \bot]$. So if we get

$\top$ from the AboveThreshold-Mechanism, we have to get $\bot$ for that same query with the BelowThreshold-Mechanism. The BelowThreshold-Mechanism provides the same privacy as the AboveThreshold-Mechanism, which can be proven like the AboveThreshold-Mechanism in [6]. These results can be used without additional leakage due to the post-processing theorem, see 3.2.

From [6], we know that the AboveThreshold-Mechanism is ϵ-DP, with ϵ being the noise parameter. Using the basic composition theorem, we also know that by executing a ϵ-DP algorithm n_1 times, we get that it is $n_1\epsilon$-DP. Therefore executing the AboveThreshold-Mechanism n times, with noise parameter ϵ_1, we get that $n\epsilon_1$-DP holds.

Next, the algorithm computes the frequency for every edge evaluated with $\top$. If an edge has a frequency greater than *cap*, we will cap it at *cap*. Hence, we minimize leakage about which processes are carried out the most. Additionally, we filter out edges that occur in less than a $\tau \in [0, 1]$ fraction of traces in the Event-Log. We consider the queries of the following form: $q_e(D) :=$ "In which frequency does edge e to occur in the Event-log D?" Each query has sensitivity $cap/(\tau \# traces)$.

Therefore $n\epsilon_1 \cdot \frac{cap}{\tau \# traces}$-DP holds, due to the basic composition theorem. Everything that is done with this graph has no additional leakage due to the post-processing theorem, see 3.2. □

5.2 The DP-Inductive-Miner

This section looks at the second approach of creating a privacy-preserving graph. Though this time, we create an approximation of the Inductive-Miner [13] since the generated Process-Structure-Trees are highly adaptable. See chapter A.2 on page v for the whole pseudocode of the implementation. In chapter B.1 on page x are various generated example Process-Structure-Trees with a used ϵ of 2

5.2.1 The basic concept

To create Process-Structure-Trees, the Inductive-Miner (See 3.1.3 on page 6) is a well-known algorithm in the Process-Mining community. Our approximation aims to create Process-Structure-Trees that are differential private. Since the overall problem of the Inductive-Miner is that Process-Structure-Trees show the whole process without filtering. Therefore an adversary can gain information about processes without any effort. To create a differential private approximation, we use the Exponential-Mechanism, to alter the cut selection, see 3.2, [6].

5.2.2 Implementation

For the implementation, we use Python 3.10 along with the PM4PY library [4].
PM4PY already has an implementation of the Inductive-Miner. To create differential private Process-Structure-Trees, we use parts of this implementation and modify them.

In the class */discovery/inductive/algorithm.py*, we modify the call in the function *apply_tree*(), the same kind of modification we also do in the function *aplly_tree*(), in the class */discovery/inductive/variants/im_clean/algorithm.py*. These two modifications ensure that we use our implementation and not the PM4PY one since with every update of PM4PY, any changes, in the corresponding library, are reverted. In the function *apply_tree*() of the */discovery/inductive/variants/im_clean/algorithm.py* class we change the *tree* = *___inductive_miner*(...) call, to our function *___dp_tree*() in the new */expo_mech.py* class. The changed classes of the PM4PY library are stored in the Expo_Package, alongside our classes to avoid changes due to updates.

Listing A.6 on page v is the main class for the computation of a differential private Process-Structure-Tree. To compute such a tree, we first need to determine a value for each cut made by the original Inductive Miner. The value we specify for each cut determines the number of traces which voted for this cut. To get this value, we call the *cut_counting.___get_cutCount*() function (See listing A.7 on page vi). In this function, the occurrence of a trace in the given event log E is counted. We store this trace in a dictionary as the key and the occurrence as the value. This is done for every trace in E if the trace is not already stored in the dictionary. The resulting dictionary is needed to determine how many traces voted for a cut $\oplus$, with $\oplus$ being any possible cut the Inductive Miner can make, see description 3.1.3.

After creating this dictionary, the *___cut_recursive*() function is called. This function is needed to handle the recursion of the *___recursion*() function. The *___recursion*() function checks the activities viewed in the next cut and stores this as *base*. After this checkup the *___check_cut*() function is called. This function checks for all cuts after the

other, which cut is possible. When a cut is possible, this cut is made, and the cuts that were not checked yet are ignored. Due to our approximation, the cut is not made, but we check how many traces voted for that cut by calling the *___count_traces*() function. This function counts the traces in which the viewed activities occur, which is sufficient since the Inductive Miner always checks every trace in the event log E. Therefore every trace, in which one of the activities from the set of all activities occurs, is counted.

If the Inductive Miner detects a *sequence cut*, we call the *___count_traces_seq*(), which is the only exception among all cuts. This function counts the traces of all activities on the leftmost branch. That is enough due to the property of a sequence (See 3.1.3 on page 6).

After counting the traces that voted for the cut, we increment two global values by one. The first value is *numCuts*, we need to determine the ϵ, which we work later with when using the Exponential Mechanism. The second value depends on the cut that is made. Therefore one out of four values is incremented. These four values are the **sequenceCount**, the **xorCount**, the **parallelCount** and the **loopCount**. These values differentiate between all made cuts because every cut differs from the others, even if the same cut is made again. With these values, we create a key for the *cutDict*-dictionary. This key contains the name of the cut with the corresponding value appended. The value we store along with the key is the previously calculated number of traces that voted for this cut. An example entry that could be stored in the *cutDict*-dictionary looks like **xor3: 34**. As soon as the creation of the *cutDict* is finished, we return to the *___get_cutCount*() function. This function then returns the *cutDict*-dictionary and the *numCuts* value.

The returned *cutDict*-dictionary is a global variable in the *expo_mech.py* file. The *cutDict*-dictionary is used in the *choose_cut*()-function to select a cut with the exponential mechanism. Since we call this function from another file and change the dictionary with every call, the dictionary must be global. With the returned *numCut* and the given *goalEpsilon*, we calculate the ϵ that has to be used in the Exponential Mechanism later on. To calculate ϵ we do $\frac{goalEpsilon}{numCut}$, this ϵ is stored global for a later use in the *choose_cut*() function.

In the next step, we call the $log\_im\_modified.\_\_\_inductive\_miner()$, see listing A.8 on page viii, to compute the differential private Process-Structure-Tree. In this function, we create the desired tree step by step. The modified part of this implementation is that if a cut is detected, we call the *expo_mech.choose_cut*() function. This function implements the Exponential Mechanism; see 3.2 on page 8. Our used score function is the *cut_counting.py* file, listing A.7, which we described before. For the implementation, we used the code from [9] and modified some parts to match our approach. One change is that we do not pass the scores and the set of cuts to choose from to the function. The needed scores alongside the cuts are in the *cutDict* dictionary, which we set global to have access to it. We create two lists from this dictionary, the first contains the cuts we choose from, and the second includes the scores. We make the *probabilities* list the second list, which we also normalize. With these probabilities, we then choose a cut. This choice is ϵ-differential private since the Exponential Mechanism is ϵ-differential private 3.2. After this, we delete the chosen cut from the dictionary, so with every call of this function, we have a smaller set of cuts. As soon as the $log\_im\_modified.\_\_\_inductive\_miner()$ function is finished, a Process-Structure-Tree is returned and displayed, which is differential private.

This approximation of the Inductive-Miner is ϵ-differential private, which is proven below in Theorem 5.2.2

Lemma 5.2 (The DP-Inductive-Miner is ϵ-Differential Private).

Proof. From [6] and description 3.2, we know that the Exponential Mechanism is ϵ-differential private if the sensitivity of the utility function is limited. It can be assumed that every process terminates at some time.

Because we are working on event logs, which represent the individual execution of a specific process, we can assume that these processes also have a start and endpoint of some sort. Even though an event log represents multiple individual executions of a process, the number of represented executions is finite since an event log has to be a finished one to be mine-able. Therefore the event log file is also finite. So because the number of process executions is finite and, therefore, the event log itself, the sensitivity of the utility function is limited. □

6

Experiments

In this chapter, we look at the evaluation of the previously described approaches chapter 5. We first explain how we evaluated the implementations. After that, we have a look at the results of the modified Heuristic-Miner, see section 5.1 and after that, we look at the results of DP-Inductive-Miner, see section 5.2.

6.1 Experimental Setup

To evaluate our implementations, see chapter 5 for these, we used the Sepsis data set [8] and generated three synthetic event logs with the tool from [17]. The corresponding workflow nets for these event logs are shown below in Figure 6.1 and Figure 6.3. For a more detailed view, see also B.1.1 on page x in the appendix.

For the evaluation of the two implementations, we followed two different ways.

modified Heuristic-Miner

To evaluate the modified Heuristic-Miner implementation, see section 5.1, we create a Utility-Privacy-Tradeoff analysis, in which we evaluate the number of nodes present in the resulting graph for different thresholds τ and a fixed ϵ_1 of 2. This fixation of ϵ_1 is done because we want to analyse the results of a privacy-preserving Directly-Follows-Graph. Furthermore, a value of 2 is not too big to significantly impact the filtering and not too low to make the results inaccurate.

In addition, an analysis in which we checked the influence of ϵ_1 with a fixed threshold τ was done. The fixed value τ is for each event log different and is fixed at a value which creates a Directly-Follows-Graphs similar to the resulting one of the Heuristic-Miner.

DP-Inductive-Miner

To evaluate the implementation of the DP-Inductive-Miner, see section 5.2, we mined each event log one hundred times, with $\epsilon = [2,\ 1,\ 0.5,\ 0.1,\ 0.05,\ 0.01]$. Therefore each event log has been mined a total of six hundred times.

We created a placement system for the cuts we can choose from, which is refreshed in every execution. In this, the cut with the highest score is in the first place, and the cut with the lowest is in the *nth* place. If some cuts have the same score, they have the same placement. The place after this is the number of cuts plus the placement of those. For example, if we have three cuts in third place, then the next cut, or cuts, are

in sixth place ($num\ of\ cuts = 3$, $current\ placement = 3 \Rightarrow 3 + 3 = 6$). Due to the deletion of the chosen cut, after every call of the corresponding *choose_cut*() function (See section 5.2), the placements of the cuts changes. Therefore we recalculate the placements with each call of the *choose_cut*() function. Also, we store each placement of the chosen cut to have an overview of the number of times a cut with placement $m \mid m \in \#Placements$ was selected.

Additionally, we analysed the fitness of our created trees. The fitness gives us a value that says how well this trace can be reproduced with the generated Process-Structure-Tree. To do so, we used PM4PY [4] to align the traces of our event logs. This is done one hundred times for each $\epsilon = [2,\ 1,\ 0.5,\ 0.1,\ 0.05,\ 0.01]$. Each trace is stored, along with the corresponding fitness score and then these values are plotted to have an overview of the overall fitness of the Process-Structure-Tree.

It must be noted that the returned fitness score is between 0 and 1, with 1 being a perfect fit. We define good fitness at a score of 0.7 or more.

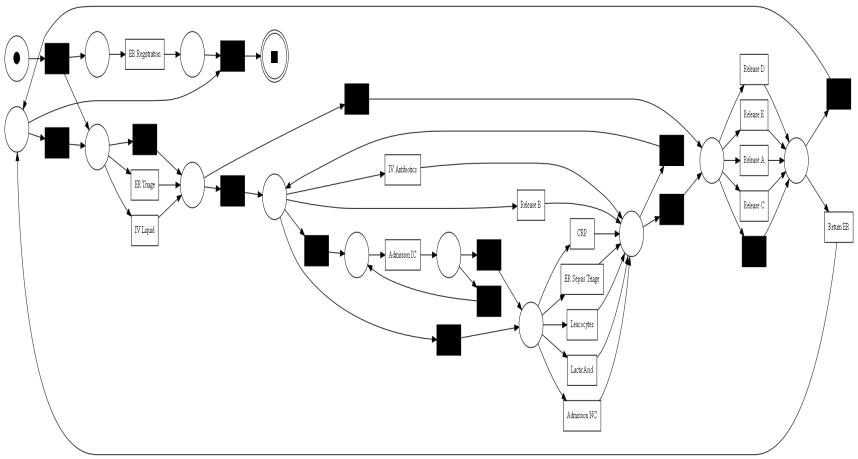


Figure 6.1: The workflow net of the Sepsis data set [8], with the black boxes being the silent transitions τ.

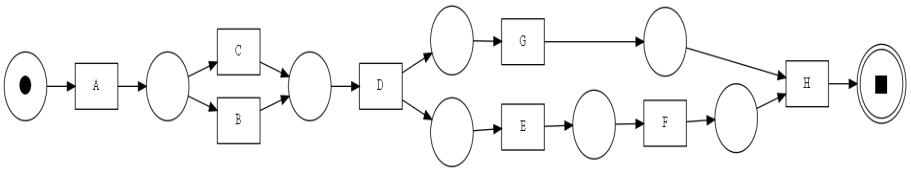
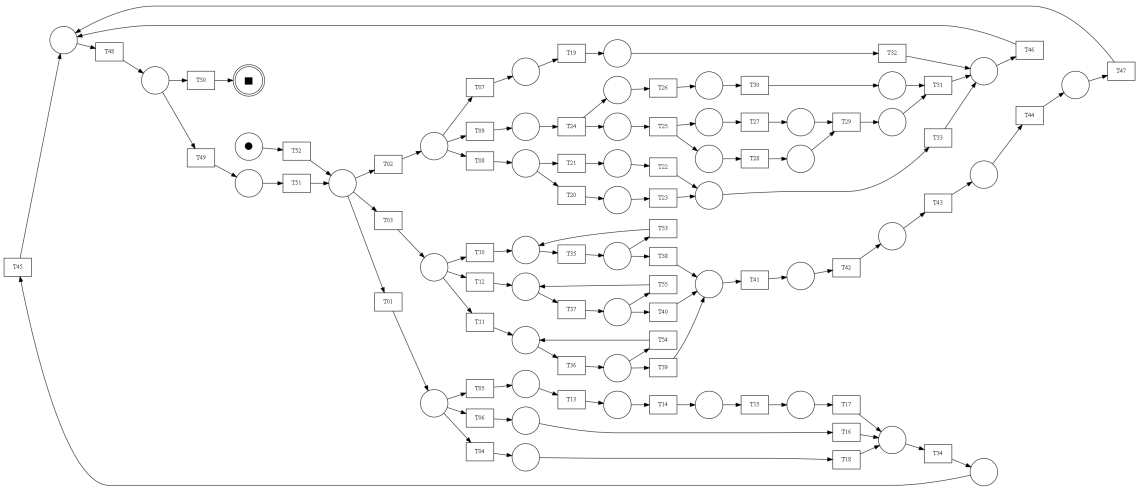
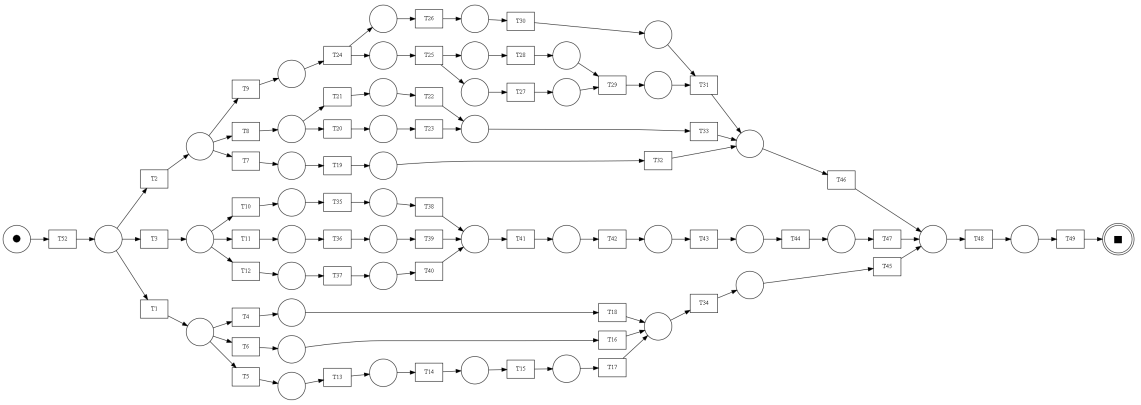


Figure 6.2: The workflow net of a non complex event log.



(a) The workflow net of a complex event log, with loops.



(b) The workflow net of a complex event log, without loops.

Figure 6.3: The workflow nets of the synthetic event logs that were generated with the tool from [17].

6.2 Evaluation Of The modified Heuristic-Miner

In this section, we look at the evaluation results of the modified Heuristic-Miner implementation, see section 5.1.

For this, we look at the Utility-Privacy-Tradeoffs, as well as at the analysis of the influence of the ϵ_1 value, see description 6.1.

6.2.1 Experimental Results

Figure 6.4 shows us the results of our Utility-Tradeoff analysis, see 6.1 for a short explanation, of the four defined workflows. For clarification, the y-axis represents the number of nodes present in the resulting Directly-Follows-Graph, and the x-axis represents the threshold τ from 0 to 100 percent.

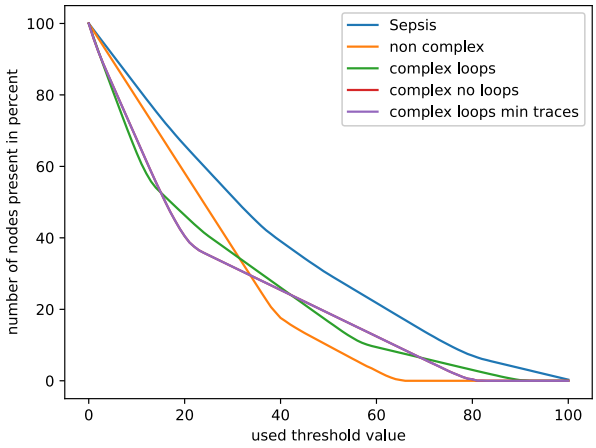


Figure 6.4: The Utility-Privacy-Tradeoffs, see 6.1, of the four previously defined workflow nets, see Figure 6.1 and below. Each analysis was done with a threshold $\tau \in [0, 100]$ and a fixed ϵ_1 of 2. At the top right is a legend in which the colour with the corresponding workflow is defined.

The first thing that can be noticed is that except for the Sepsis data set [8], all of the workflows have a strong decrease in terms of the number of nodes present in the resulting Directly-Follows-Graph. Additionally, it can be observed that the event log for the complex workflow with loops, Figure 6.3a, has two different lines. The line representing *min traces* is an event log based on the same workflows (See Figure 6.3a), but only with 5.000 traces instead of 10.000. What can be observed by comparing these two is that the line representing the larger event log has not a decrease in the number of nodes as high as the corresponding line, representing the smaller event log.

Secondly, we look at the influence the ϵ value can have on the result. For this, we look at figure 6.5, which shows us for each of the workflows the influence of ϵ_1 in terms of the number of existing nodes in the resulting Directly-Follows-Graph. As we can see each of the workflows has a different high since the total number of nodes is for all of the workflows

different. Since we are using the same workflow, only with fewer traces with the *min traces* line in Figure 6.4, we are not representing this in Figure 6.5.

The first thing that can be noticed here is that the scale goes from 0.0 to 1.0 $*$ $1e^8$. This is the index that tells us to multiply each of the x-axis values with $1e^8$to get the correct value.

As we can see all of the event logs are having all of their nodes, with a seemingly low ϵ_1 value. This on the other hand means that the resulting Directly-Follows-Graphs are showing the event log without any filtering when using a high ϵ_1 value.

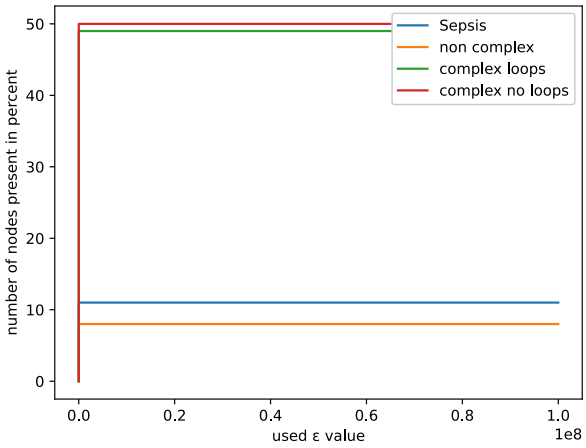


Figure 6.5: The influence of ϵ_1 on each of the event logs, see description 6.1 for a short explanation. With a static threshold τ, which differs for each event log. However, the τ is set a value so that the optimal Directly-Follows-Graph would result as an output if the ϵ_1 value is chosen correctly.

6.2.2 Interpretation Of The Results

To interpret the previously described results, we need to define the goal of this approach. The overall goal is to create a Directly-Follows-Graph that fulfils two main objectives. These are as follows.

Ob1 The graphs generated are privacy-preserving
Ob2 The graphs generated are as accurate as possible

While we have proven the first objective to be true (See 5.1.2 on page 16), because every graph generated has to fulfil this property, the second objective has to be answered with our experimental results.

Based on the results described above, we know we have some tipping point for every event log, in which the total number of nodes in the graph suddenly drops. This point can cause a massive drop in the accuracy of the generated Directly-Follows-Graphs.

Additionally, we can observe a high decrease in the number of nodes present in the resulting graph the higher the used τ is. Since we are using the differential private mechanism

AboveThreshold, we add random noise to the calculated percentage value of each transition. This random noise means that scores near the threshold could be filtered out. On top of that, we are also checking the BelowThreshold with added random noise. This random noise has the same effect as the AboveThreshold mechanism. These two mechanisms can increase the number of transitions filtered out since the random noise can cause edge cases to be added or deleted. Due to this, it is essential to use the correct threshold τ that is neither too high nor too low to get a Directly-Follows-Graph that does not leak too much information but is also not too abstract.

With this in mind, adjusting the Below-Threshold mechanism could solve some issues. However, such an adjustment would be needed to solve the tipping, since we are highly sensitive to τ. On top of that, any adjustment could cause a mechanism that no longer fulfils the property of differential privacy (See 3.2 on page 8).

Because of this property, we can have graphs that fulfil **Ob1** but are not capable of fulfilling **Ob2**, independent of ϵ_1 since for the Utility-Privacy-Tradeoff analysis, we use a ϵ_1 of 2, which is small, but the tipping points are nevertheless extreme.

By observing the Figure 6.5, it can be noticed that with a high ϵ_1 the resulting Directly-Follows-Graph shows the corresponding event log without any filtering. From this, we can conclude that we need to use a low ϵ_1 value to ensure that the resulting graph does fulfil **Ob1**. Therefore it is possible with the correct ϵ_1 to generate a graph that violates **Ob1**.

Furthermore, we can observe that the number of traces is an important factor in the filtering. When looking at Figure 6.4, we can see that we are using two different event logs of the same workflow, see Figure 6.3a. The difference between these two event logs is the number of traces, which influences the result. The line corresponding to the event log with fewer traces has, at an earlier point, fewer nodes in its generated Directly-Follows-Graph, than the line corresponding to the event log with more traces.

Due to this, we can conclude that the number of traces and, therefore, the length of the event log is a crucial factor in our filtering function.

6.3 Evaluation Of The DP-Inductive-Miner

In this section, we look closely at the experimental results and their interpretation for the DP-Inductive-Miner implementation, see section 5.2.

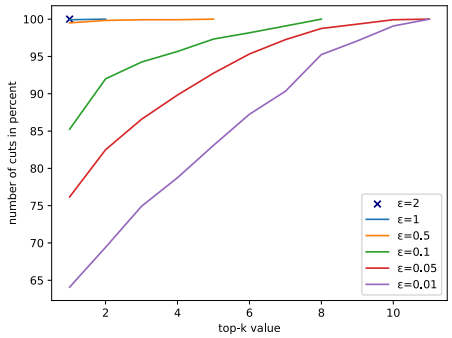
For each of the four defined workflows, see Figure 6.1 and below, we look at the placement evaluation results and the corresponding fitness evaluation. See description 6.1 for further details.

6.3.1 Experimental Results

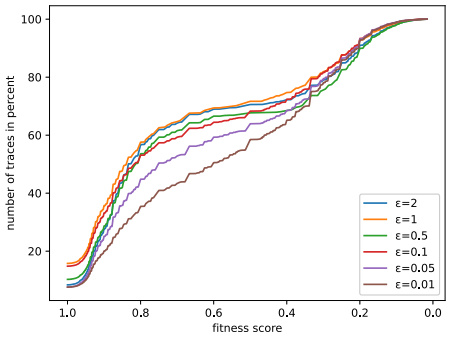
First, we have a look at the results of the Sepsis data set [8] to have knowledge about the performance, of our DP-Inductive-Miner implementation 5.2, on a widely known and accepted data set.

As we can see in figure 6.6a with $\epsilon = 2$ all of our selected cuts are in the top-1 of all cuts. Additionally, it can be observed that with $\epsilon = [1, 0.5]$, approximately 98 percent of all cuts are in the top-1. Furthermore, as soon as $\epsilon = 0.1$, or lower, the overall percentage of

cuts in the top-1 goes from 64 up to 85 percent. With $\epsilon = 0.01$, we have the steepest slope, which results in 80 percent of the cuts being in the top-5.



(a) Placement evaluation for the Sepsis data set [8]



(b) Fitness evaluation for the Sepsis data set [8]

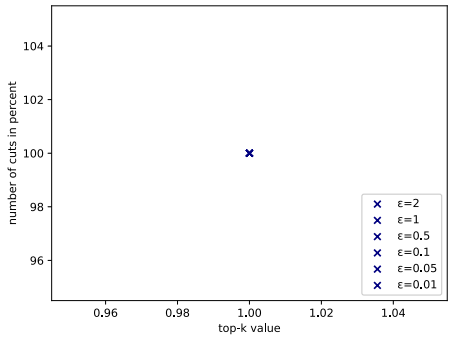
Figure 6.6: The Evaluation of the top-k cuts on the left side 6.6a and the Evaluation of the fitness of the generated trees on the right side 6.6b, for further information on these evaluations, see description 6.1 on page 21. For both evaluations, the DP-Inductive-Miner section 5.2 was executed one hundred times with a different ϵ being $[2, 1, 0.5, 0.1, 0.05, 0.05]$. At the bottom right of both diagrams is a legend which shows the colour to the corresponding ϵ that was used.

In figure 6.6b, we have the results of the fitness evaluation 6.1 for the Sepsis data set [8]. This analysis shows us how well the traces of the underlying event log can be reproduced using the resulting tree. As we can see with all ϵ values, we have a perfect fitting for some traces. The interesting part is that with $\epsilon = 2$, we have fewer traces with a perfect fitting score than with $\epsilon = 0.1$. However, the overall fitting score of $\epsilon = 2$ is the best. The increase of the fitting score is the highest, and the increase flatters as soon as we have a score of 0.4.

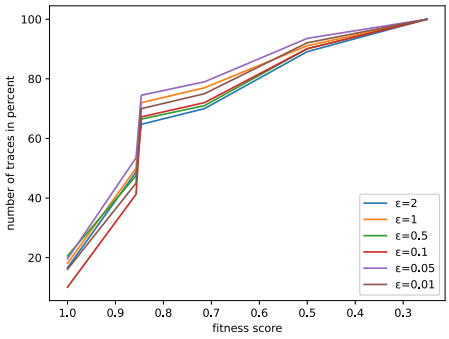
Additionally, it is observable that with $\epsilon = 1$, we have a similar curve to $\epsilon = 2$. The difference is that we have more traces with a perfect fit, but on the other hand, we have more traces with a worse fitting score. Furthermore, with $\epsilon = 0.01$, we also have perfectly fitting traces, but also the overall score of good fitting traces is fewer than for the other ϵ values.

By now, we are looking at the results in figure 6.7a, which represent the results for the workflow in figure 6.2. For each of the ϵ, one hundred percent of all traces are in the top-1. This behaviour is unique among all of the workflows.

However, if we compare this with the results of the corresponding fitness evaluation, figure 6.7b, we can observe a different behaviour. Here we can see that none of the generated trees reproduces all the traces of the underlying event log perfectly. Nonetheless, the number of perfect-fitting traces is for all of the ϵ values higher than what we observed in figure 6.6b, which corresponds with the results of the placement evaluation. Even the overall number of good fitting traces is higher than what we observed for the Sepsis data set in figure 6.6b. Additionally, it can be seen that the lowest fitting score is approximately 0.3, while we are near 0 for the Sepsis data set.



(a) Placement evaluation for the non-complex workflow 6.2



(b) Fitness evaluation for the non-complex workflow 6.2

Figure 6.7: The Evaluation of the top-k cuts on the left side 6.7a and the evaluation of the fitness of the generated trees on the right side 6.7b, for further information on these evaluations, see description 6.1 on page 21. For both evaluations, the DP-Inductive-Miner section 5.2 was executed one hundred times with a different ϵ being $[2, 1, 0.5, 0.1, 0.05, 0.05]$. At the bottom right of both diagrams is a legend which shows the colour to the corresponding ϵ that was used.

In figure 6.8, we can see the results of the two analyses done on the complex workflow, Figure 6.3a.

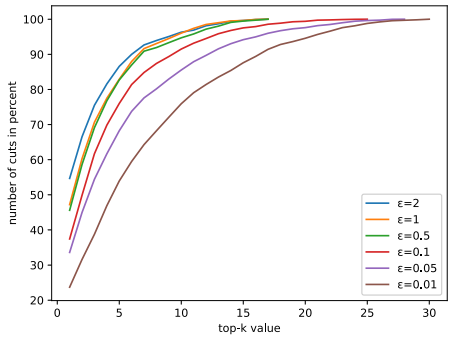
Figure 6.8a, shows us a logarithmic growth over the cut-to-placement relation. With $\epsilon = 2$, we have with 55 percent the most traces in the top-1. Also, the overall number of traces with a good placement is the highest. As soon as we reach the top-10 value, the number of cuts for $\epsilon = 1$ are on par with $\epsilon = 2$. On the other hand, we can see that with $\epsilon = 0.01$, we have the most incredible range of placements out of all of the ϵ values.

By now having a look at figure 6.8b, we can see that for none of the ϵ values, we have a perfect fitting trace. We have the first traces we can partly reproduce as soon as we reach a fitness score of approximately 0.8 with the generated Process-Structure-Tree. At a score of 0.6 for all of the ϵ values, we can reproduce some part of the trace from the event log of the underlying workflow, Figure 6.3a. Even though it is possible with $\epsilon = 2$ to reproduce parts of the most traces, we also can see that we have traces for that we can not reproduce even a part of it.

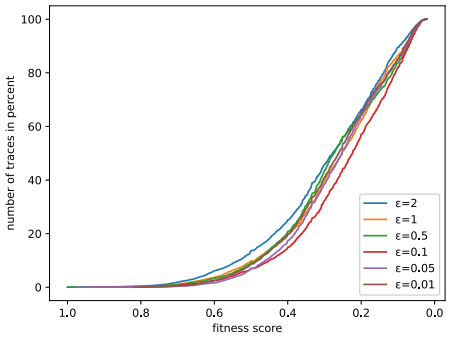
When we look at Figure 6.9, we can observe the event log results from the workflow in Figure 6.3b. This workflow is similar to the one in Figure 6.3a but differs because it has no loops.

By looking at Figure 6.9a we can see that it is similar to Figure 6.8a. However, they differ because we have a lower number of traces in the top-1, Figure 6.9a. Nevertheless, the overall growth of the cut-to-placement relation is bigger than in Figure 6.8a.

This growth directly influences the fitness score, which we can observe in Figure 6.9b. Here we have up to 5 percent of all fully reproducible traces with the generated Process-Structure-Trees. This is in direct contrast to Figure 6.8b, where we have no trace being reproducible until a fitting score of 0.8. Additionally, the overall fitness of the workflow



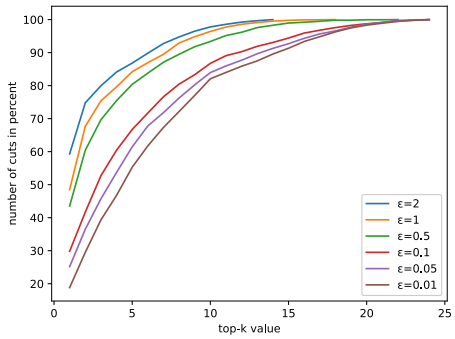
(a) Placement evaluation for the complex workflow 6.3a



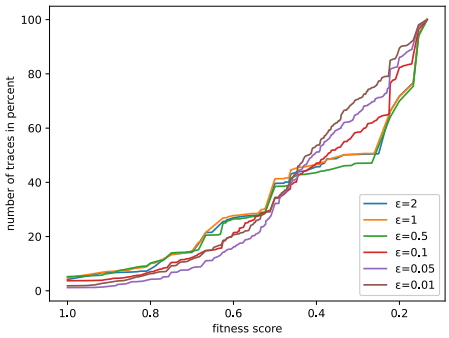
(b) Fitness evaluation for the complex workflow 6.3a

Figure 6.8: The Evaluation of the top-k cuts on the left side 6.8a and the evaluation of the fitness of the generated trees on the right side 6.8b, for further information on these evaluations, see description 6.1 on page 21. For both evaluations, the DP-Inductive-Miner section 5.2 was executed one hundred times with a different ϵ being $[2, 1, 0.5, 0.1, 0.05, 0.05]$. At the bottom right of both diagrams is a legend which shows the colour to the corresponding ϵ that was used.

6.3b is better than the workflow in Figure 6.3a since we have a minimal fitness score of 0.1, Figure 6.9b. In Figure 6.8b, we have a minimal fitness score of 0.



(a) Placement evaluation for the complex workflow without loops 6.3b



(b) Fitness evaluation for the complex workflow without loops 6.3b

Figure 6.9: The Evaluation of the top-k cuts on the left side 6.9a and the evaluation of the fitness of the generated trees on the right side 6.9b, for further information on these evaluations, see description 3.2 on page 8. For both evaluations, the DP-Inductive-Miner section 5.2 was executed one hundred times with a different ϵ being $[2, 1, 0.5, 0.1, 0.05, 0.05]$. At the bottom right of both diagrams is a legend which shows the colour to the corresponding ϵ that was used.

6.3.2 Interpretation Of The Results

When having a closer look at figure 6.7a we can get some information about the behaviour of cuts. Since the Process-Structure-Tree of the underlying workflow 6.2 has a depth of two and a $SEQUENCE - Cut$ as root, see Figure 6.10, it can be assumed that every cut has to have the same score due to the property of the $SEQUENCE - Cut$ (See 3.1.3).

By having a closer look at Figure 6.10 we can see that all events and cuts are executed sequentially or in parallel. This causes the events to be in every cut, which also causes the cuts, we want to manipulate to all have the same score. Therefore we can assume that the order of cuts is relevant to the scoring function.

Due to this, every cut has the same probability of being chosen, resulting in trees with a top-1 cut score. However, the underlying event log is not perfectly reproducible even with $\epsilon = 2$, as seen in the fitness evaluation, Figure 6.7b.

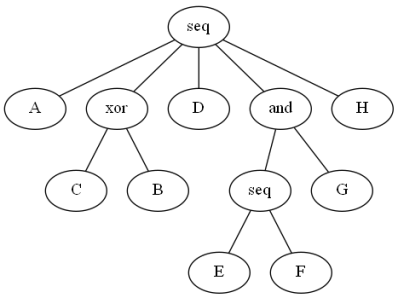


Figure 6.10: The original Process-Structure-Tree of the non-complex event log.

To check the relevance of the cut order, we modify the workflow. This modification is done by replacing the parallelism of the $AND - Cut$ with a $XOR - Cut$. The following $SEQUENCE{-}Cut$ should have a different score than the other cuts in the tree, Figure 6.11 shows the resulting workflow net. The corresponding Process-Structure-Tree of such a modified workflow can be seen below, in Figure 6.12b, alongside the original one to have a direct comparison.

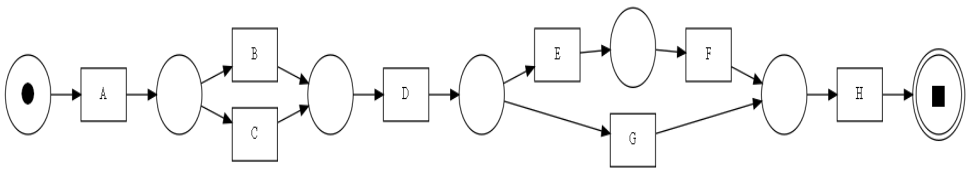
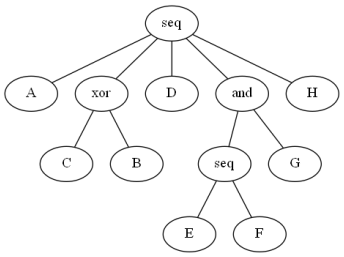


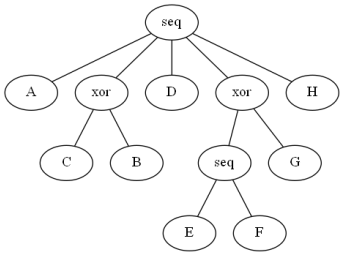
Figure 6.11: The modified workflow net, used to evaluate the importance of the order of cuts.

Due to this new workflow, another placement is made. The resulting new placement changes the cut selection and fitness score of the resulting Process-Structure-Trees, what we can see in Figure 6.13. For $\epsilon = [2, 1, 0.5]$, the results of the cut placement evaluation are the same as in Figure 6.7a. We can see a difference with $\epsilon = 0.1$ and lower. For these values, we have a cut selection that is not limited to the top-1.

With such a new placement, the fitness score also changed. While we can observe a



(a) The Process-Structure-Tree of the original workflow 6.2

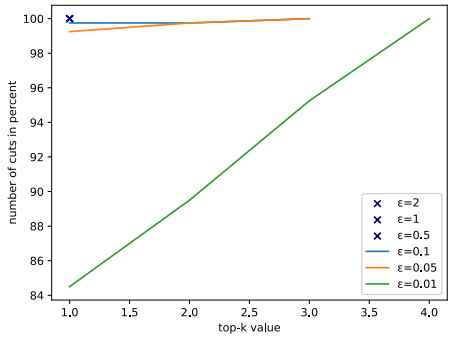


(b) The Process-Structure-Tree of the modified workflow

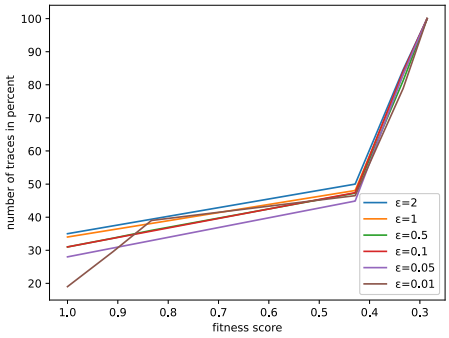
Figure 6.12: The original 6.12a and modified 6.12b Process-Structure-Tree of the original and modified non complex event log. In the modified version we switched the $AND - Cut$ from the original, with a $XOR - Cut$.

steep slope in Figure 6.7b, at a score of 0.9 to 0.8, we have a relatively weak slope in the fitness evaluation of Figure 6.13b. Not until a score of 0.4 does the degree of slope increase.

Due to this observation, we can conclude that the type and order cuts in the original Process-Structure-Tree, are essential factors for our scoring function.



(a) Placement evaluation for the modified non-complex workflow 6.11



(b) Fitness evaluation for the modified non-complex workflow 6.11

Figure 6.13: The Evaluation of the top-k cuts on the left side 6.13a and the evaluation of the fitness of the generated trees on the right side 6.13b, for further information on these evaluations, see description 6.1 on page 21. For both evaluations, the DP-Inductive-Miner section 5.2 was executed one hundred times with a different ϵ being $[2, 1, 0.5, 0.1, 0.05, 0.05]$. At the bottom right of both diagrams is a legend showing the colour to the corresponding ϵ used.

Besides the relevance of order, we correlate with the complexity of workflows and the accuracy of the cut selection. In figure 6.8a and 6.3b, we can see that we have worse placements for the selected cuts. These placements result in Process-Structure-Trees that can not reproduce any trace of the underlying original event log, see Figure 6.8b.

Additionally, we can observe that the lower the ϵ, the lower the overall fitness score of the Process-Structure-Trees, since with $\epsilon = 2$, the fitness score is the first that increases. This behaviour corresponds with the property of ϵ for differential private mechanism, which says that the lower the ϵ, the less accurate the resulting data. Conversely, the higher the ϵ, the less privacy-preserving the data, see description 3.2.

While the complexity of the workflows is essential, it can be observed in Figure 6.8 and Figure 6.9 that loops are also an essential factor. As we can see in Figure 6.9, with a workflow without loops Figure 6.3b, a wide range of possible Process-Structure-Trees is possible. This can be concluded since we have fitness scores of 1, which means we can reproduce complete traces. On the other hand, we can only reproduce small parts of the traces. This can not be observed in Figure 6.8b. Here, we are not getting any Process-Structure-Tree that can reproduce a complete trace of the original event log. It is only possible for the resulting tree to reproduce parts of the original traces. With these observed results, we can conclude that loops significantly influence the resulting trees. The main loop that may have the most significant influence is the one that restarts the workflow. This can be assumed since the self-loops from the workflow in Figure 6.3a are not repeating any of the cuts.

While the synthetic event logs show us the things which influence the results the most, the Sepsis data set [8] shows that the DP-Inductive-Miner implementation, see section 5.2, is capable of generating trees that have a high fitness but are also privacy-preserving. When looking at Figure 6.6, it can be seen that the overall placement and the fitness score generate Process-Structure-trees that are similar to the original by using the privacy parameter $\epsilon = 2$.

Looking at B.1 on page x, we can see various Process-Structure-Trees that are possible outputs of the DP-Inductive-Miner implementation. As can be seen, the trees corresponding to the Sepsis data [8] are different from the original but are similar and fulfil our privacy assumption while being capable of reproducing complete traces of the actual event log.

Based on these results, we know that the order and type of cuts are the most important for our scoring function. Furthermore, it is possible to generate usable Process-Structure-Trees based on complex workflows, as long as the degree of complexity is manageable.

7

Conclusion

In this thesis, we designed two algorithms for privacy-preserving process mining. We apply techniques from differential privacy and process mining to create approximations of algorithms that produce a privacy-preserving model of an event log. To make such a visualization, we have a deeper look into two frequently used algorithms: the Heuristic-Miner [15] and the Inductive-Miner [13]. For the approximation of the modified Heuristic-Miner, we are using the general idea of filtering transitions on a Directly-Follows-Graph. In comparison, we are modifying the Inductive-Miner, to generate a privacy-preserving model for our second algorithm. For both algorithms, we prove that these fulfil differential privacy. In the evaluation of chapter 6, we tested both algorithms on four workflows, three of them being synthetic [17] and one being the Sepsis data set [8].

Firstly we evaluate the modified Heuristic-Miner algorithm. Here it can be seen that the algorithm can fulfil our defined objectives. Nonetheless, the precision needed to select the correct threshold τ and ϵ_1 is too high. It might be possible to optimize this using automated testing of each generated Directly-Follows-Graphs before giving one as an output. However, this optimization would not fulfil the intention of getting a privacy-preserving process model if it is possible to select the output model. Another way could be a change in the scoring function. When using the intentional way of scoring and combing this with the AboveThreshold mechanism, it would be possible to use the built-in scoring function from the Heuristic-Miner and simultaneously filter transitions with a differential private mechanism.

Secondly, we evaluate the DP-Inductive-Miner algorithm. Here it is easy to see, based on the fitness scores, that our algorithm generates trees that are privacy-preserving and not too inaccurate. While it is not possible to use the resulting trees to present the workflow structure, it is possible to publish these without any privacy concerns. Nonetheless, if an event log has a specific order of cuts and/or is overly complex, the resulting Process-Structure-Tree has the potential to be highly inaccurate. This could be solved using a weighting function that supports the scoring function so that the probability is higher based on the correct order of cuts. With this, the accuracy might increase, but this would reduce the privacy of the resulting trees.

In summary, we show that both algorithms fulfil the intention of generating privacy-preserving process models. While the modified Heuristic-Miner algorithm is too inaccurate and therefore is not fulfilling our objective of generating an accurate and privacy-preserving

process model. However, the DP-Inductive-Miner algorithm can be used on most event logs and still produces process models that are, on the one hand, privacy-preserving and, on the other hand, not inaccurate for usage.

7.1 Future Work

This section describes possible future work for each of the two algorithms.

modified Heuristic-Miner As mentioned before, it is possible to use the build-in scoring function of the original Heuristic-Miner, to filter the transitions. This scoring function views the dependency of each trace without observing other transitions. By combining this with the AboveThreshold-mechanism, it is possible to filter randomly. This would create a filtering that fulfils differential privacy. By combing this with the approach from [7], where the authors make the weighting of the transitions differential private, it is possible to have a fully differential private Directly-Follows-Graph.

DP-Inductive-Miner To overcome the dependency of the scoring function on the order of cuts, an additional weighting function helps. This function checks which cut we are currently at. For example, if it is the root cut, then we would have the maximum weight added to the score of the cut. However, this weight should be a random value. This is so that it might be possible that a ”higher” cut can be lower, so it is ensured that the resulting tree is not more predictable than before. With this weighting, the resulting trees are not perfectly accurate, but when computing a complex workflow, the accuracy is higher than before.

Bibliography

[1] Aalst, W. van der, Weijters, T., and Maruster, L. Workflow mining: discovering process models from event logs. In: *IEEE Transactions on Knowledge and Data Engineering* 16(9):1128–1142, 2004. DOI: 10.1109/TKDE.2004.47.

[2] Aalst, W. van der *Process Mining*. Springer Berlin, Heidelberg, 2016. DOI: https://doi.org/10.1007/978-3-662-49851-4.

[3] Aalst, W. van der Process Mining: Overview and Opportunities. In: *ACM Trans. Manage. Inf. Syst.* 3(2), 2012. DOI: 10.1145/2229156.2229157. URL: https://doi.org/10.1145/2229156.2229157.

[4] Berti, A., Zelst, S. J. van, and Aalst, W. van der *Process Mining for Python (PM4Py): Bridging the Gap Between Process- and Data Science*. 2019. DOI: 10.48550/ARXIV.1905.06169.

[5] Chen, J., Wang, W. H., and Shi, X. Differential Privacy Protection Against Membership Inference Attack on Machine Learning for Genomic Data. In: 2021. DOI: 10.1142/9789811232701_0003.

[6] Cynthia Dwork, A. R. *The Algorithmic Foundations of Differential Privacy*. 2014. URL: https://www.cis.upenn.edu/~aaroth/Papers/privacybook.pdf.

[7] Elkoumy, G., Pankova, A., and Dumas, M. *Privacy-Preserving Directly-Follows Graphs: Balancing Risk and Utility in Process Mining*. 2020. arXiv: 2012.01119 [cs.CR].

[8] Mannhardt, F. *Sepsis Cases - Event Log*. 4TU.ResearchData, 2016. URL: https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.

[9] Near, J. P. and Abuah, C. *Programming Differential Privacy*. Vol. 1. 2021. URL: https://uvm-plaid.github.io/programming-dp/.

[10] *Privacy-Preserving Process Mining*. Springer, 2019. DOI: 10.1007/s12599-019-00613-3. URL: https://doi.org/10.1007/s12599-019-00613-3.

[11] Rafiei, M. and Aalst, W. van der Privacy-Preserving Data Publishing in Process Mining. In: *Business Process Management Forum*. Ed. by D. Fahland, C. Ghidini, J. Becker, and M. Dumas. Springer International Publishing, 2020, pp. 122–138. ISBN: 978-3-030-58638-6. URL: https://link.springer.com/chapter/10.1007/978-3-030-58638-6_8.

[12] Rahimian, S., Orekondy, T., and Fritz, M. Differential Privacy Defenses and Sampling Attacks for Membership Inference. In: 2021. DOI: 10.1145/3474369.

[13] S.J.J. Leemans D. Fahland, W. v. d. A. *Discovering Block-Structured Process Models from Event Logs - A Constructive Approach*. Springer Berlin Heidelberg, 2013. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.396.197&rep=rep1&type=pdf.

[14] Voigt, S. Nuñez von, Fahrenkrog-Petersen, S. A., Janssen, D., Koschmider, A., Tschorsch, F., Mannhardt, F., Landsiedel, O., and Weidlich, M. Quantifying the Re-identification Risk of Event Logs for Process Mining. In: *Advanced Information Systems Engineering*. Springer International Publishing, 2020, pp. 252–267. ISBN: 978-3-030-49435-3.

[15] Weijters, A., Der Aalst, W. M. van, and De Medeiros, A. A. Process mining with the heuristics miner-algorithm. In: *Technische Universiteit Eindhoven, Tech. Rep. WP* 166(July 2017):1–34, 2006.

[16] Weijters, W. v. B. v. J. H. L. M. G. S. A. Workflow mining: A survey of issues and approaches. In: 47(2):237–267, 2003. DOI: https://doi.org/10.1016/S0169-023X(03)00066-1. URL: https://www.sciencedirect.com/science/article/pii/S0169023X03000661.

[17] Zisgen, Y., Janssen, D., and Koschmider, A. Generating Synthetic Sensor Event Logs for Process Mining. In: *Intelligent Information Systems*. Ed. by J. De Weerdt and A. Polyvyanyy. Cham: Springer International Publishing, 2022, pp. 130–137. ISBN: 978-3-031-07481-3.

A

Pseudocodes

A.1 The modified Heuristic-Miner

Listing A.1 main function

```
1:  class main(InputFile i) {
2:      #determine file type
3:      if filetype of i == '.csv':
4:          log = read (event lgo)
5:          call csvFile(log)
6:      else if filetype of i == '.xes':
7:          log = read (event log)
8:          call traces(log)
9:      else if filetype of i == '.txt':
10:         log = read (trace log)
11:         call traces(log)
12:     else:
13:         print('Please choose a valid file')
14: }
```

Listing A.2 Getting the edges from a CSV event log file

```
1:  class csvFile(EventLog D, Threshold τ, Epsilon ϵ1, cap, n) {
2:      #define nedded lists
3:      id = list()
4:      edges = dict()
5:
6:      #EventLog is sorted based on the TraceIDs and Timestamps
7:      sort D
8:
9:      #get the number of traces
10:     for TraceID i in D do:
11:         if i = first Element in D do:
12:             add i to id
13:         else if i is in id do:
```

```
14:             skip
15:         else do:
16:             add i to id
17:     numTraces = len(id)
18:
19:     #get transitions
20:     for i in range(len(D.TraceID)) do:
21:         if D.TraceID(i) == D.TraceID(i+1) do:
22:             for j in range(i+2, len(D.TraceID)+2) do:
23:                 if tuple(D.ActivityName[i:j]) not in edgesList do:
24:                     edges.update({tuple(D.ActivityName[i:j]): 1})
25:                     break
26:                 else:
27:                  edges[tuple(D.ActivityName[i:j]] += 1
28:                  break
29:             end
30:     end
31:
32:     #computethe privacy-preserving Directly-Follows-Graph
33:     privacy(edges, D, τ, ϵ1, cap, n)
34:
35: }
```

Listing A.3 Getting the edges from a XES file or a trace log file

```
1:  class traces(EventLog D, Threshold T, Epsilon ϵ1, cap, n) {
2:      queries = dict()
3:
4:      for i in range(len(D)) do
5:          for j in range(i+2, len(D)+2)
6:              if tuple(i, j) not in queries:
7:                  queries.update({tuple(i, j): 1})
8:              else:
9:                  queries[tuple(i, j)] += 1
10:     end
11:
12:     #computethe privacy-preserving Directly-Follows-Graph
13:     privacy(queries, D, τ, ϵ1, cap, n)
14:
15: }
```

Listing A.4 Exectuion of the AboveThreshold- and BelowTHreshold mechanism, to filter the transitions and generate a privacy-preserving Directly-Follows-Graphs

```
1:  class privacyComputation(Data dpDict, EventLog D, Threshold τ, Epsilon ϵ1, cap,
        n) {
2:
3:      edgesList = dpDict.keys()
4:      cQueries = dpDict.values()
```

```
5:
6:    #filter the transitions and output the filtered list of transitions -> which
          queries are above the threshold
7:    a := 1
8:    def AboveThreshold(Database edgesList, Queries cQueries, τ, ϵ1){
9:        while a < n do:
10:           for Each query i do:
11:               Let v_i = τ,+ Lap(4/ϵ1)
12:               if ((numID/numTraces)*100 + v_i) ≥ 70% do:
13:                   Output a_i = ⊤
14:                   Halt.
15:               else do:
16:                   Output a_i = ⊥
17:               end
18:           end
19:           a += 1
20:       end
21:   }
22:
23:   #filter the transitions, so that it is known which queries are below the
          threshold
24:   b := 1
25:    def BelowThreshold(Database edgesList, Queries cQuries, τ, ϵ1){
26:        while b < n do:
27:          for Each query i do:
28:               Let v_i = τ + Lap(4/ϵ1)
29:               if ((numID/numTraces)*100 + v_i) < 70% do:
30:                   Output a_i = ⊤
31:                   Halt.
32:               else do:
33:                   Output a_i = ⊥
34:               end
35:           end
36:           b += 1
37:       end
38:   }
39:
40:   #Each query that is ⊤ from AboveThreshold and ⊥ from BelowThreshold is
          considered, every other is filtered out
41:   for each Query q from return AboveThreshold:
42:       for each Query i from return BelowThreshold:
43:           if q = i and q = ⊤ and i = ⊥
44:               add q to consideredQueries
45:
46:   #cap the frequency of the considered Traces
47:   def capQuries(cQueries) {
48:       for k, v in dpDict.items():
49:           if v > cap:
50:               dpDict.update({k: cap})
51:   }
```

```
52:
53:     #output the directly follows graph
54:     directly-follows-graph(dpDict)
55: }
```

A.2 DP-Inductive-Miner

Listing A.5 Main Class

```
1:  class main(EventLog E):
2:      read E
3:
4:      call expo_mech.__dp_tree(E)
5:
6:      show process-structure-tree
7:  end class
```

Listing A.6 Chooses the cut with the Exponential-Mechansim and retruns the computed Process-Structure-Tree

```
1:  class expo_mech(EventLog E):
2:      def __dp_tree(E):
3:          cut_dict, numCuts = cut_counting.__get_cutCount(E)
4:          tree = log_im_modified.inductive_miner(E)
5:          return tree
6:      end def
7:
8:      #uses the exponential mechanism to choose a cut
9:      def choose_cut():
10:         elements = list(cut_dict.keys())
11:         scores = list(cut_dict.values())
12:
13:         probabilities = exp((scores*ϵ)/(sensitivity_u)) / [(∑ cut')*exp((scores*ϵ)]/(2*sensitivity_u))
14:
15:         #chooses a random cut based on the calculated probabilities
16:         cut = numpy.random.choice(elements, 1, p=probabilities)[0]
17:
18:         #return the cut to use and change the score, since we used a cut
19:         if 'sequence' in cut:
20:             cut_dict.update({cut: 0})
21:             return SEQUENCE-Cut
22:         elif 'xor' in cut:
23:             cut_dict.update({cut: 0})
24:             return XOR-Cut
25:         elif 'and' in cut:
26:             cut_dict.update({cut: 0})
27:             return AND-Cut
28:         elif 'loop' in cut:
29:             cut_dict.update({cut: 0})
30:             return LOOP-Cut
31:     end def
32:
33: end class
```

Listing A.7 Counts the Traces that voted for a cut

```
1:  class cut_counting(EventLog E):
2:      traceDict = dict()
3:      cut_dict = dict()
4:
5:      numCuts = 0
6:      sequenceCount = 0
7:      xorCount = 0
8:      parallelCount = 0
9:      loopCount = 0
10:
11:     def __get_cutCount(log E):
12:         for trace in E:
13:             trActivities = list(activities in trace)
14:             traceKeys = list(traceDict.keys())
15:
16:             if tuple(trActivities) not in traceKeys:
17:                 traceDict.update({tuple(trActivities): 1})
18:             elif tuple(trActivities) in traceKeys:
19:                 traceDict[tuple(trActivities)] += 1
20:
21:         __cut_recursive(E)
22:
23:         return(cutDict, numCuts)
24:     end def
25:
26:     def __cut_recursive(log E):
27:         tree = __recursion(E)
28:         return tree
29:     end def
30:
31:     def __recursion(log E):
32:         if E = {ϵ}:
33:             base = {τ}
34:         elif ∃a ∈ ∑: E = {⟨a⟩}:
35:             base = {a}
36:         else:
37:             base = ∅
38:
39:         P = __check_cut(E)
40:
41:         if |P| == 0:
42:             if base = ∅
43:                 return {↻(τ, a_1, ..., a_m) where {a_1, ..., a_m} = ∑(L)}
44:             else:
45:                 return base
46:         retrun P ∪ base
47:     end def
48:
```

```
49:     def __check_cut(log E):
50:         for L in E:
51:             if L_i == len(1):
52:                 return ∅
53:             elif there is is a nontrivial XOR-Cut c in DFG(L):
54:                 activities = list(events in L)
55:                 count = count_traces(E, activities)
56:                 numCuts += 1
57:                 xorCount += 1
58:                 cut_dict.update({'xor' + str(xorCount): count})
59:                 return{(X, ((L_1, 0), ..., (L_i,0))}
60:             elif there is is a nontrivial SEQUENCE-Cut c in DFG(L):
61:                 activities = list(events in L)
62:                 count = count_traces_seq(E, activities)
63:                 numCuts += 1
64:                 sequenceCount += 1
65:                 cut_dict.update({'sequence' + str(sequenceCount): count})
66:                 return{(→, ((L_1, 0), ..., (L_i,0))}
67:             elif there is is a nontrivial PARALLEL-Cut c in DFG(L):
68:                 activities = list(events in L)
69:                 count = count_traces(E, activities)
70:                 numCuts += 1
71:                 parallelCount += 1
72:                 cut_dict.update({'parallel' + str(parallelCount): count})
73:                 return{(⋀, ((L_1, 0), ..., (L_i,0))}
74:             elif there is is a nontrivial LOOP-Cut c in DFG(L):
75:                 activities = list(events in L)
76:                 count = count_traces(E, activities)
77:                 numCuts += 1
78:                 loopCount += 1
79:                 cut_dict.update({'loop' + str(loopCount): count})
80:                 return{(↺, ((L_1, 0), ..., (L_i,0))}
81:             return ∅
82:     end def
83:
84:     def __count_traces(log, activitis):
85:         count = 0
86:         for act in activities:
87:             for trace in log:
88:                 if act in trace:
89:                     count += 1
90:         return count
91:     end def
92:
93:     def __count_traces_seq(log, activities):
94:         count = 0
95:         seqGroup = activities[0]
96:         for act in seqGroup:
97:             for trace in log:
98:                 if act in trace:
```

```
99:                     count += 1
100:        return count
101:    end def
102:
103: end class
```

Listing A.8 Discover Cuts and create a Process-Structure-Tree (See [13] for the original implementation)

```
1:  class log_im_modified():
2:
3:      def inductive_miner(Log E)
4:          tree = inductive_miner_internal(E)
5:          return tree
6:      end def
7:
8:      def inductive_miner_internal(E):
9:          if L == {ϵ}:
10:             base = {τ}
11:         elif ∃a ∈ ∑: L = {⟨a⟩}:
12:             base = {a}
13:         else:
14:             base = ∅
15:
16:         P = choose_cut(E)
17:         if |P| == 0:
18:             if base == ∅:
19:                 return {↻(τ, a_1, ..., a_m) where {a_1, ..., a_m} = ∑(L)}
20:             else:
21:                 retrun base
22:         return P ∪ base
23:     end def
24:
25:     def choose_cut(log E):
26:         if L_i == len(1):
27:             return ∅
28:         elif there is is a nontrivial XOR-Cut c in DFG(L):
29:             divide Activities to Sets ∑_1, ..., ∑_i
30:             ⊕ = expo_mech.choose_cut()
31:             return{(⊕, ((L_1, 0), ..., (L_i,0))}
32:         elif there is is a nontrivial SEQUENCE-Cut c in DFG(L):
33:             divide Activities to Sets ∑_1, ..., ∑_i
34:             ⊕ = expo_mech.choose_cut()
35:             return{(⊕, ((L_1, 0), ..., (L_i,0))}
36:         elif there is is a nontrivial PARALLEL-Cut c in DFG(L):
37:             divide Activities to Sets ∑_1, ..., ∑_i
38:             ⊕ = expo_mech.choose_cut()
39:             return{(⊕, ((L_1, 0), ..., (L_i,0))}
40:         elif there is is a nontrivial LOOP-Cut c in DFG(L):
```

```
41:             divide Activities to Sets ∑_1, ..., ∑_i
42:             ⊕ = expo_mech.choose_cut()
43:             return{(⊕, ((L_1, 0), ..., (L_i,0))}
44:         return ∅
45:     end def
46: end class
```

B

Figures

B.1 Workflow Nets and Resulting Trees Of The Experiments

B.1.1 Workflow Nets Of The Used Event Logs

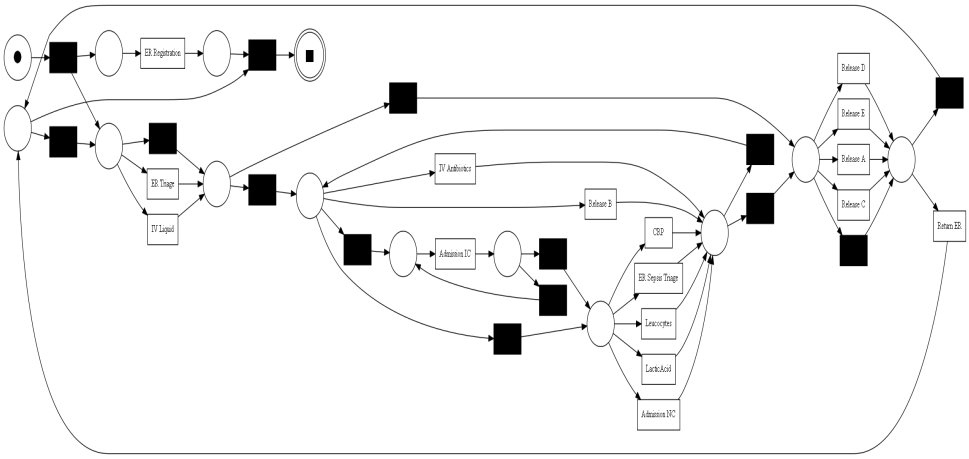


Figure B.1: The workflow net of the Sepsis data set [8], with the black boxes being the silent transitions τ

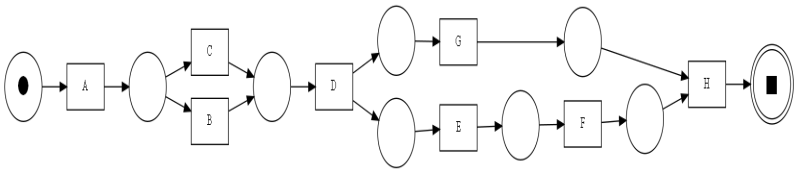


Figure B.2: The workflow net of an event log that represents a simple process structure. Generated with [17]

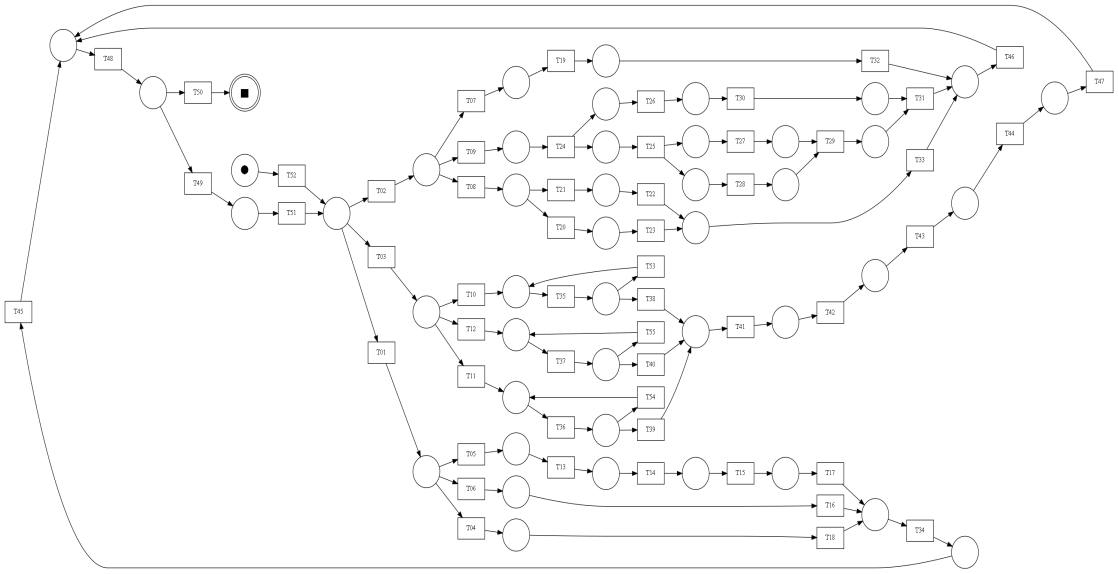


Figure B.3: The workflow net of an event log that represents a complex process structure, with repetitions in it. Generated with [17]

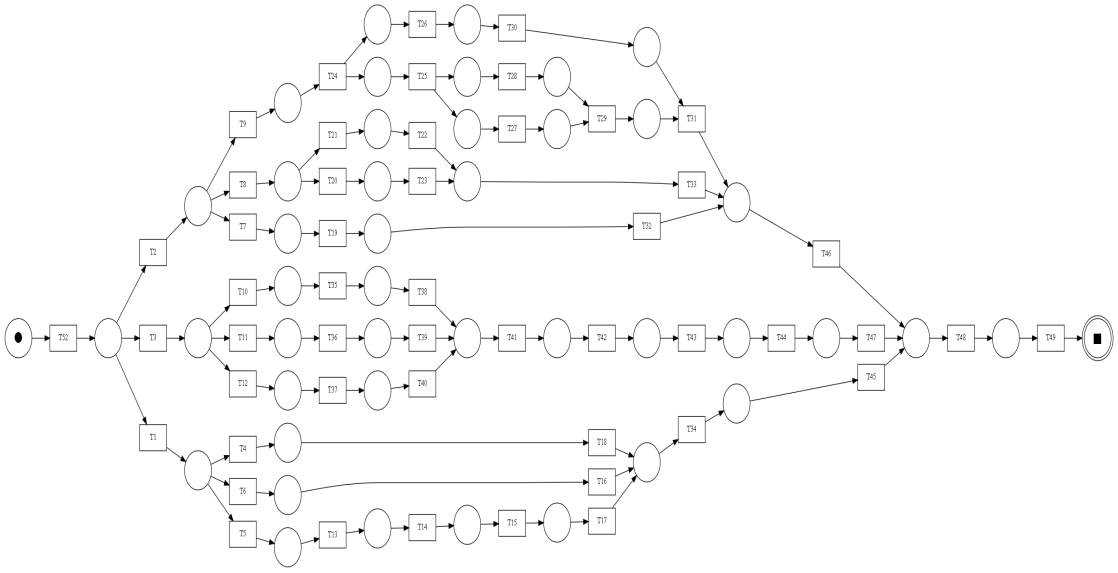
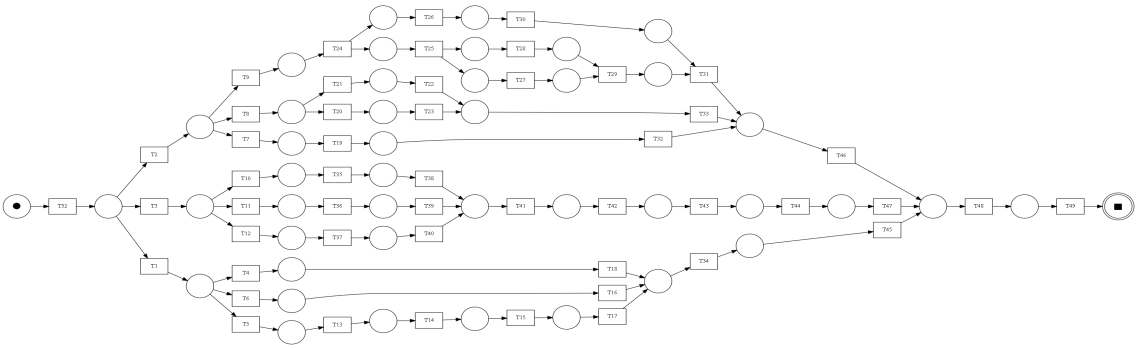


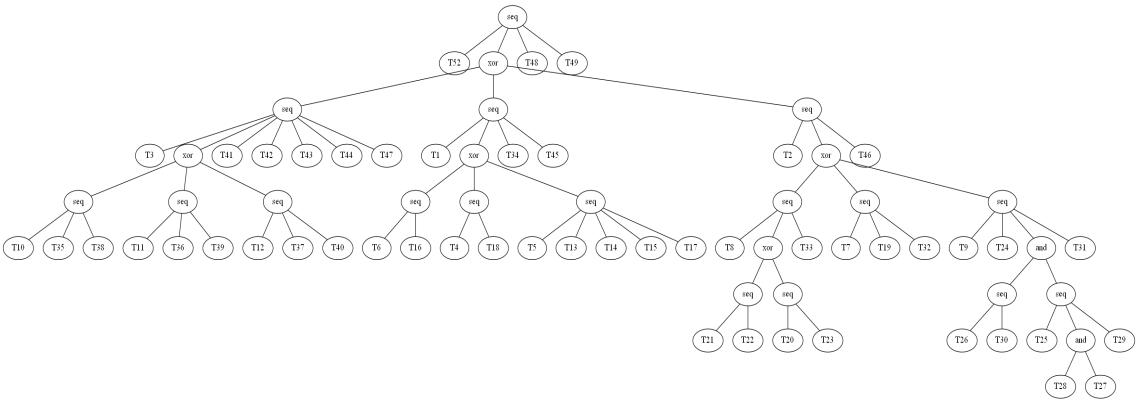
Figure B.4: The workflow net of an event log that represents a complex process structure, without any repetitions. Generated with [17]

B.2 Resulting Process-Structure-Trees of the DP-Inductive-Miner implementation

B.2.1 Results of with complex workflow without loops

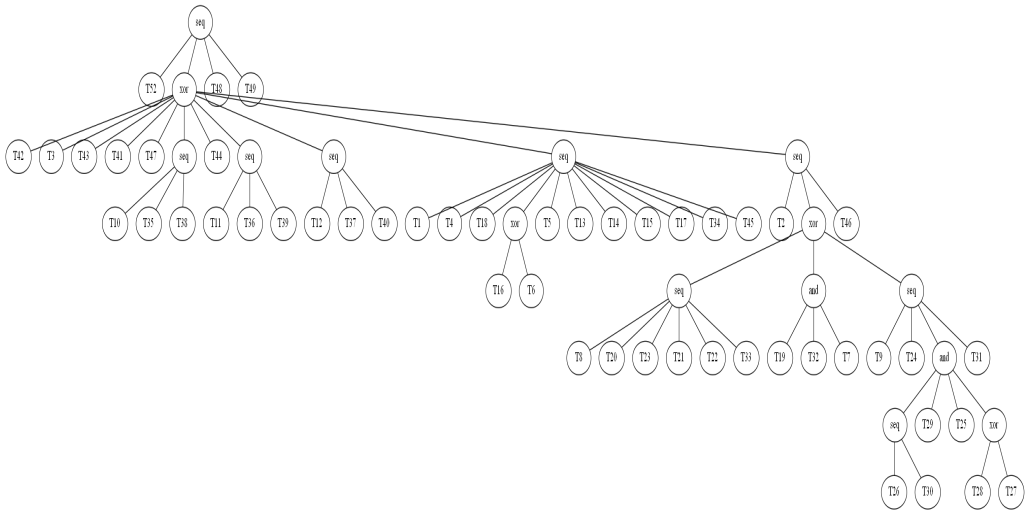


(a) complex workflow-net without loops

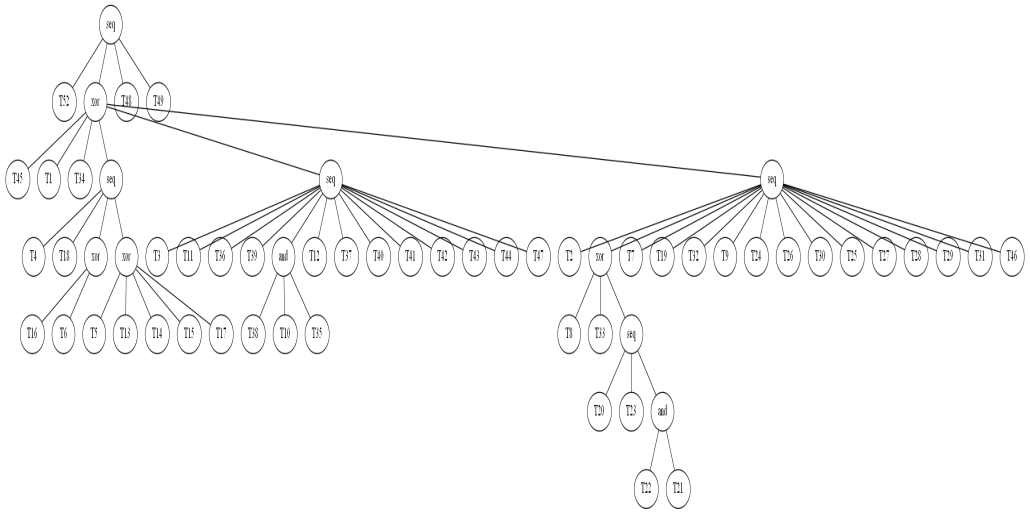


(b) Original Process-Structure-Tree of the above workflow-net

Figure B.5: A complex workflow-net without any loops, with the corresponding Process-Structure-Tree



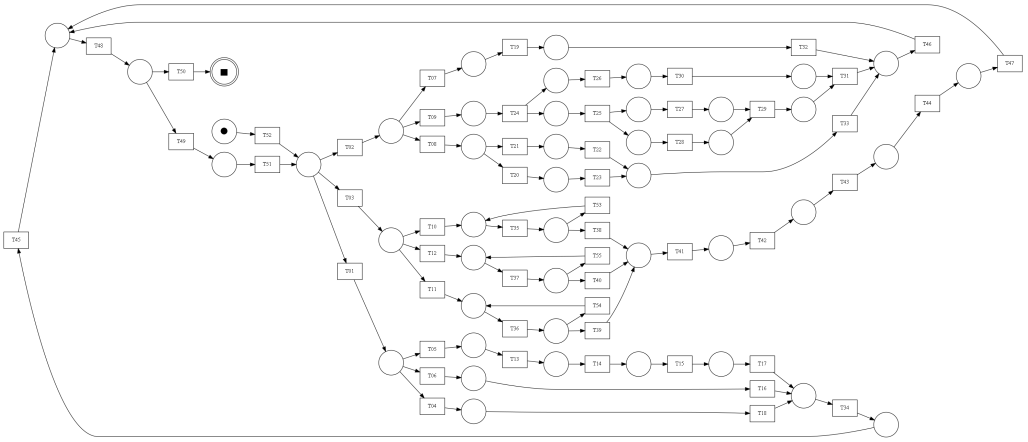
(a) One possible version of the computed Process Structure-Tree



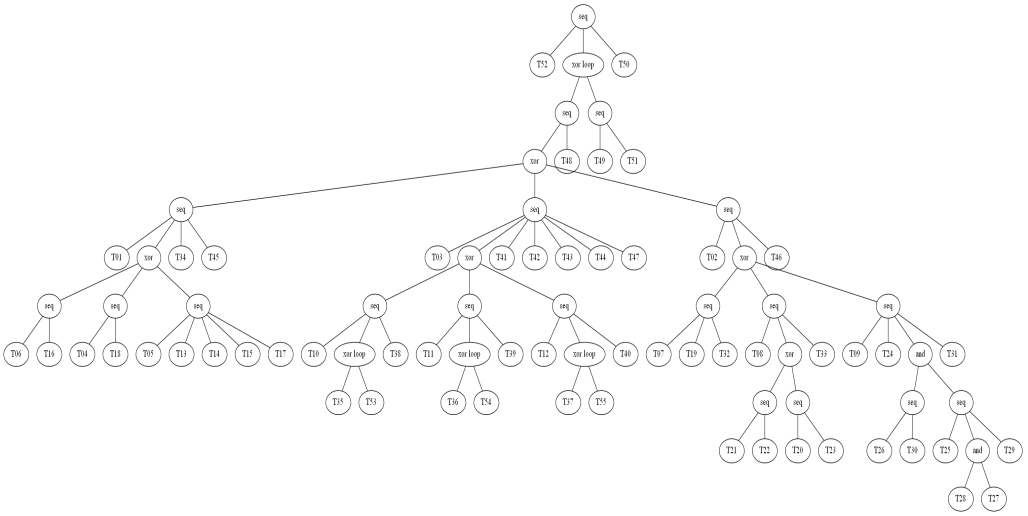
(b) Second possible version of the computed Process Structure-Tree

Figure B.6: Two possible versions of the computed Process-Structure-Tree with privacy parameter $\epsilon = 1$ and for 1.050 Traces, for the complex workflow-net without loops (See B.5 on page xii)

B.2.2 Results of with a complex workflow with loops

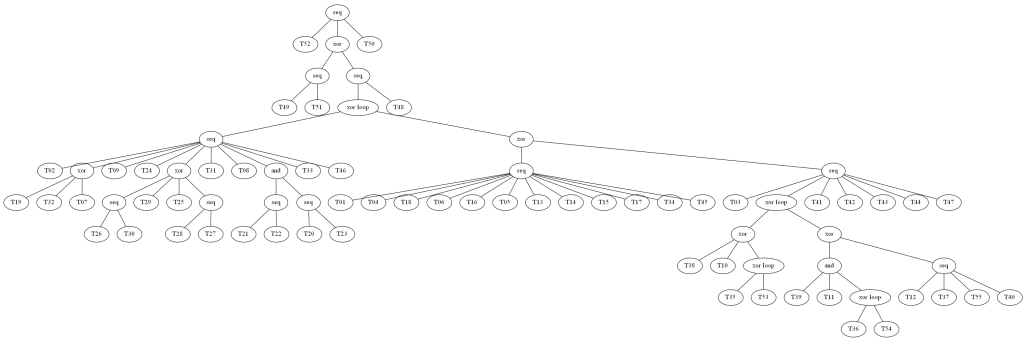


(a) complex workflow-net with loops

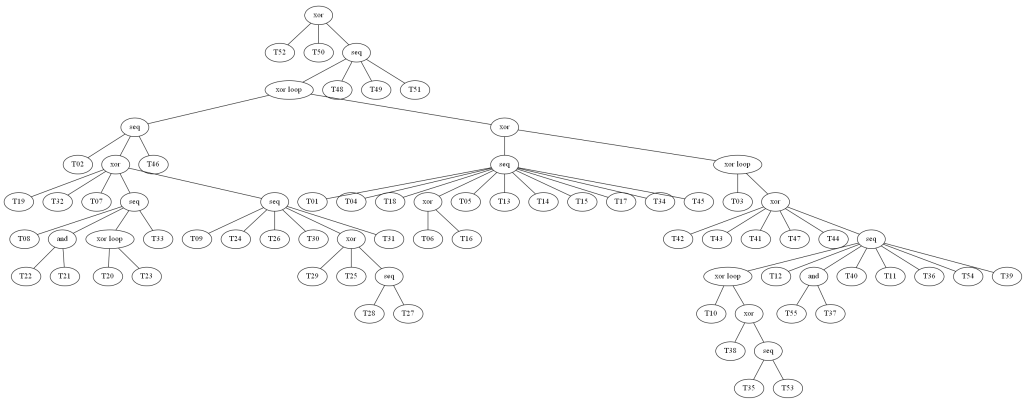


(b) Original Process-Structure-Tree of the above workflow-net

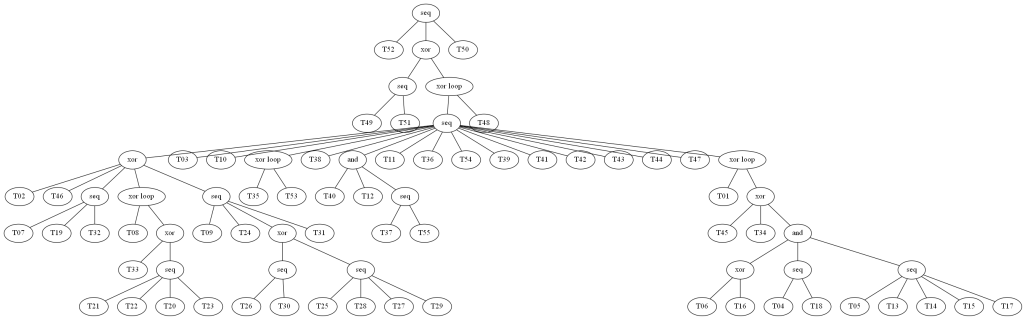
Figure B.7: A complex workflow-net with loops, with the corresponding Process-Structure-Tree



(a) The first possible version of the computed Process Structure-Tree



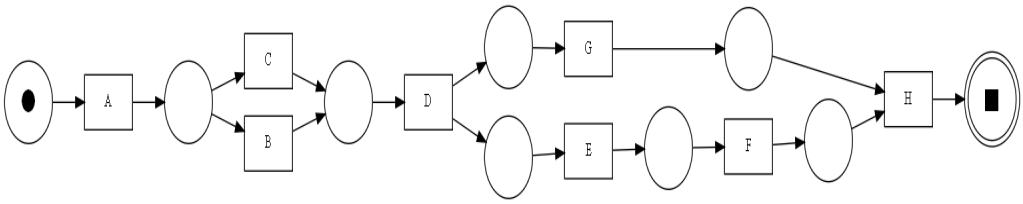
(b) The second possible version of the computed Process Structure-Tree



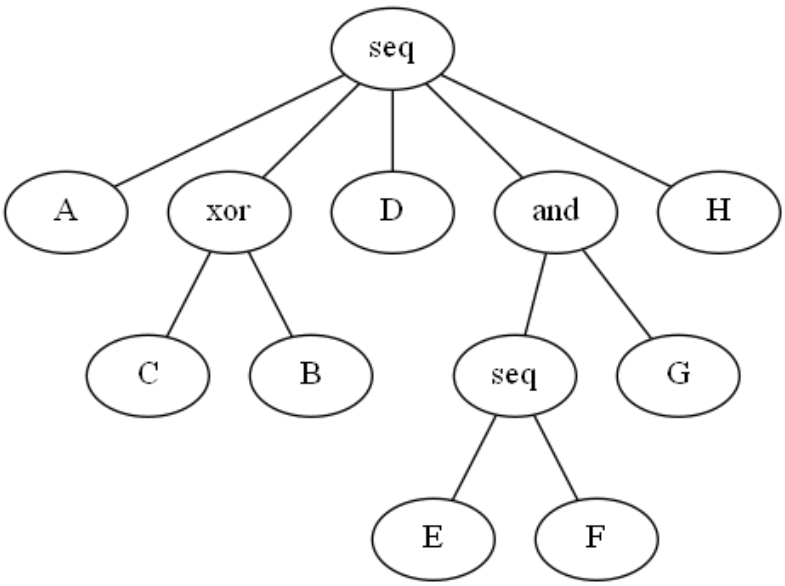
(c) The second possible version of the computed Process Structure-Tree

Figure B.8: Three possible versions of the computed Process-Structure-Tree with privacy parameter $\epsilon = 1$ and for 5.000 Traces, for the complex workflow-net with loops (See B.7 on page xiv)

B.2.3 Results with a non complex workflow

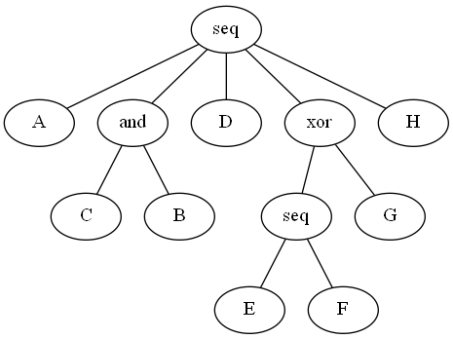


(a) a non complex workflow-net

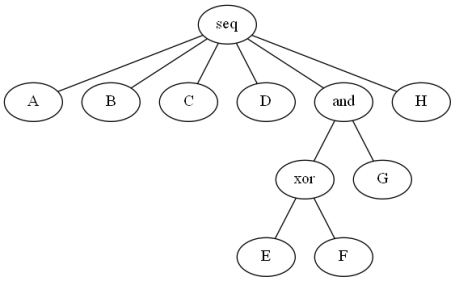


(b) Original Process-Structure-Tree of the above workflow-net

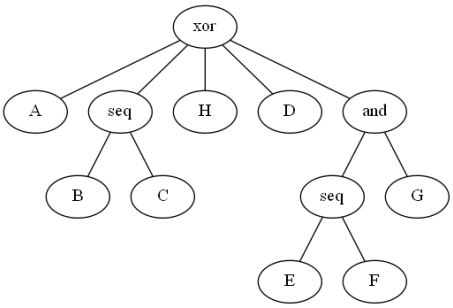
Figure B.9: A non complex workflow-net, with the corresponding Process-Structure-Tree



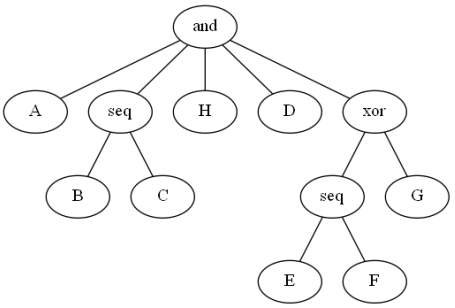
(a) The first possible version of the computed Process Structure-Tree



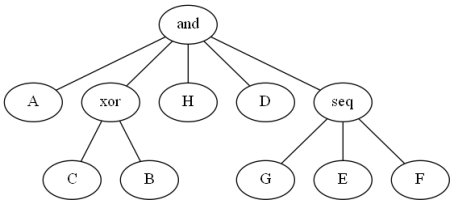
(b) The second possible version of the computed Process Structure-Tree



(c) The third possible version of the computed Process Structure-Tree



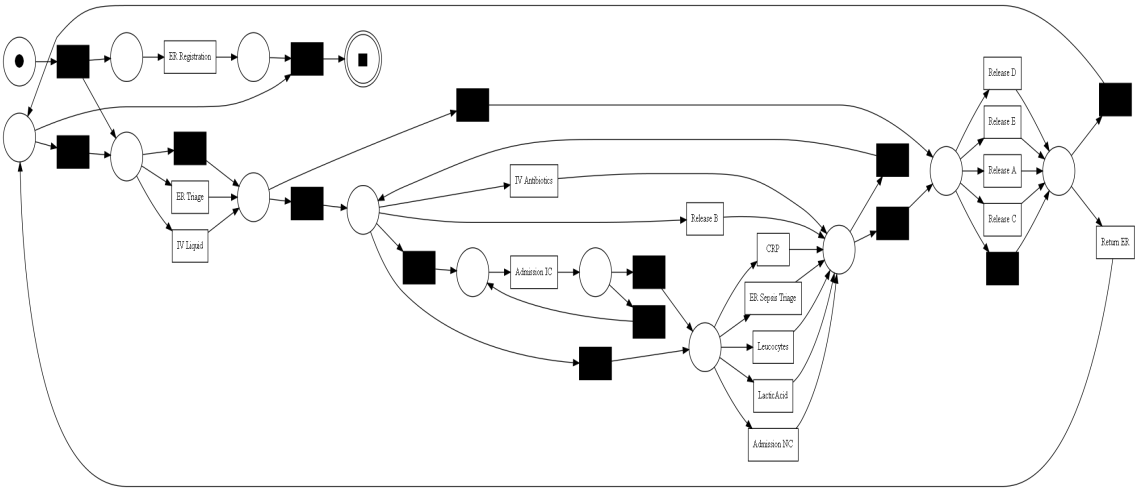
(d) The fourth possible version of the computed Process Structure-Tree



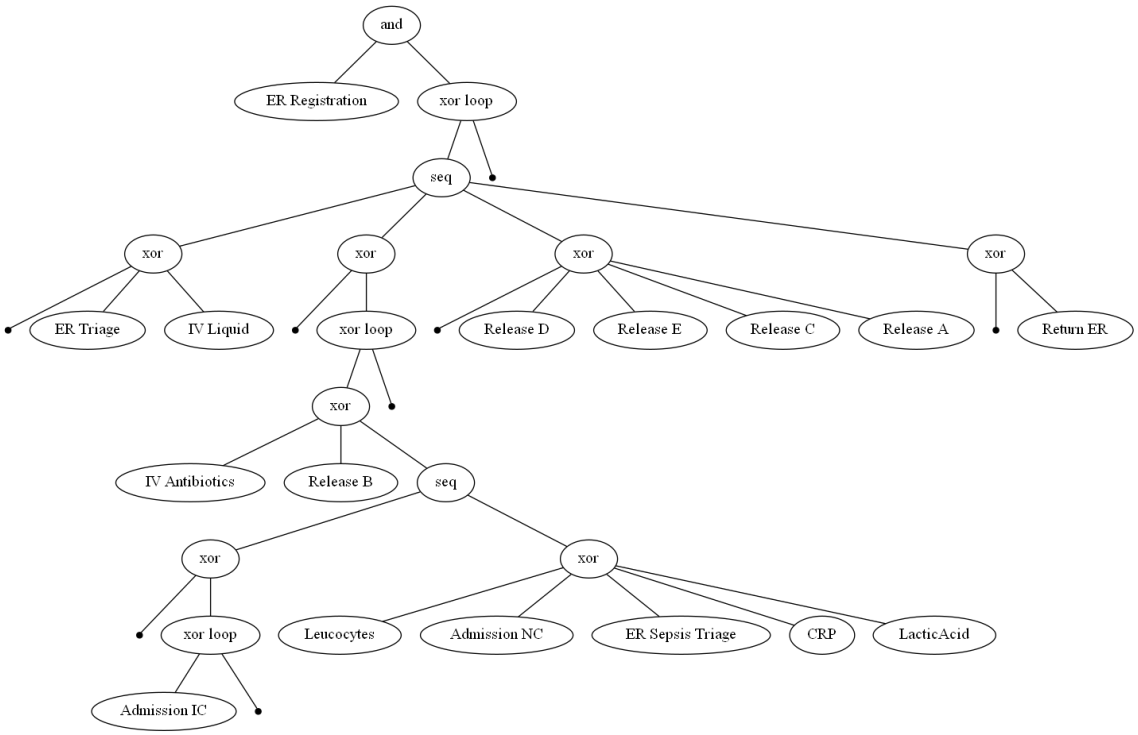
(e) The fifth possible version of the computed Process Structure-Tree

Figure B.10: Five versions possible versions of the computed Process-Structure-Tree with privacy parameter $\epsilon = 1$ and for 500.000 Traces, for the non complex workflow-net (See B.9 on page xvi)

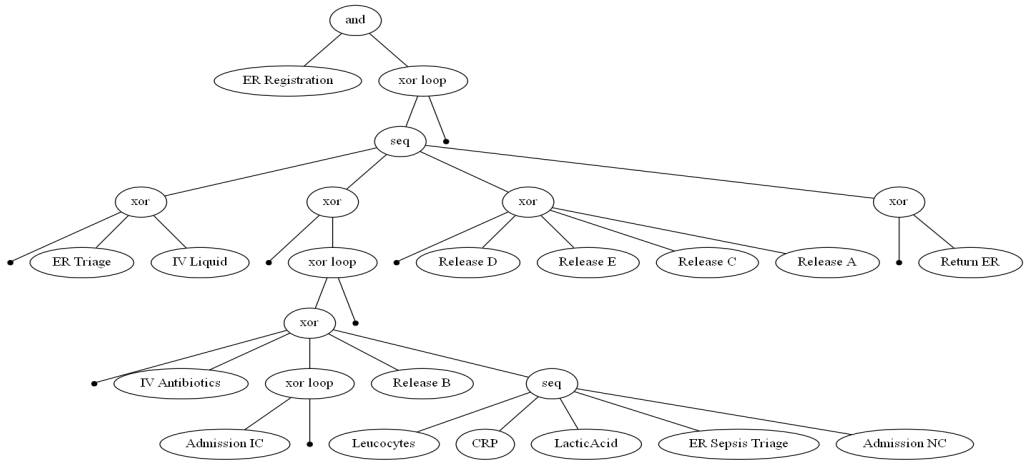
B.2.4 Results with the SEPSIS data cases



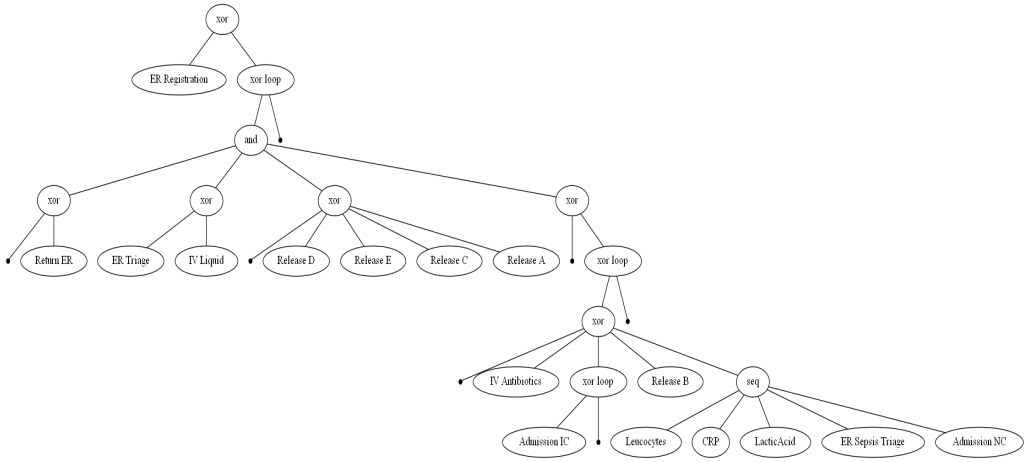
(a) Workflow-net of the SEPSIS-data



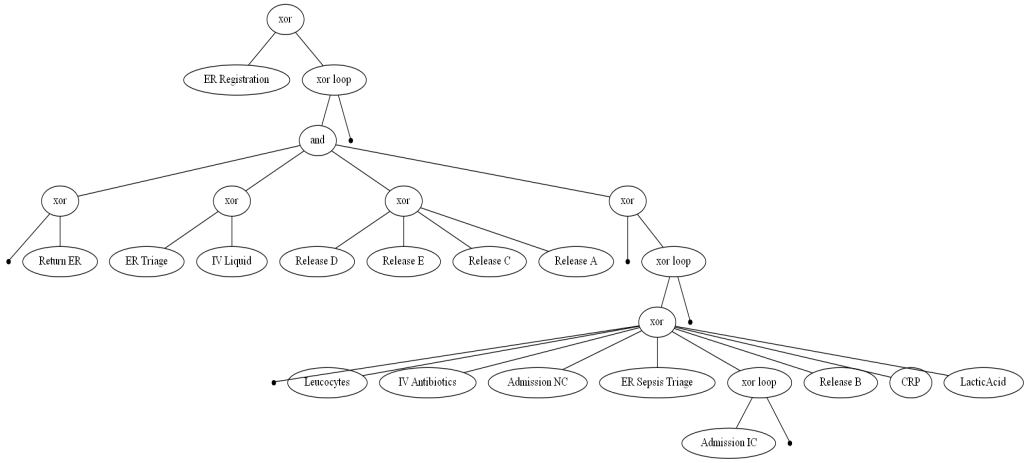
(b) Original Process-Structure-Tree of the above workflow-net



(a) The first possible version of the computed Process Structure-Tree

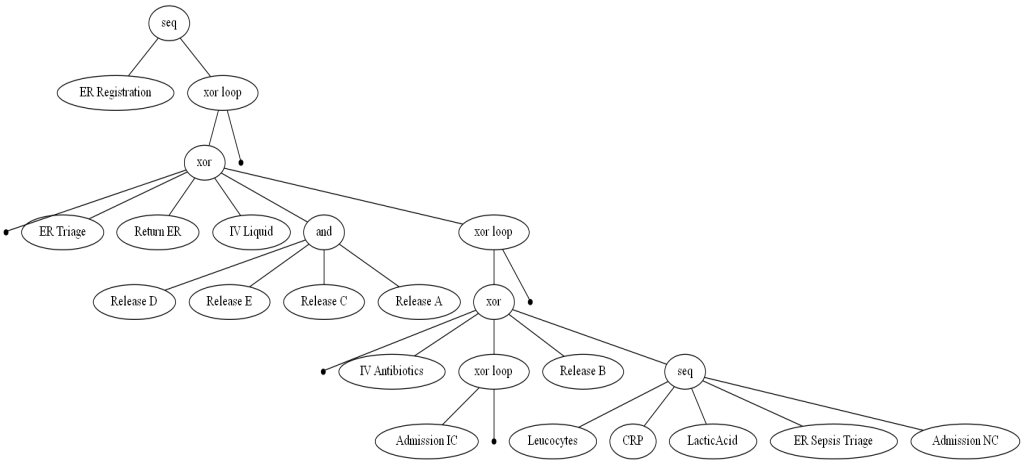


(b) The first possible version of the computed Process Structure-Tree

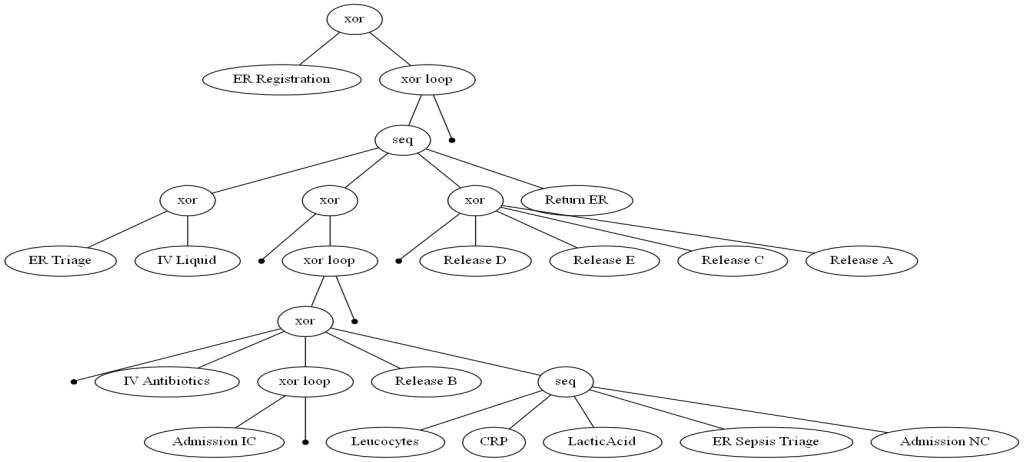


(c) The first possible version of the computed Process Structure-Tree

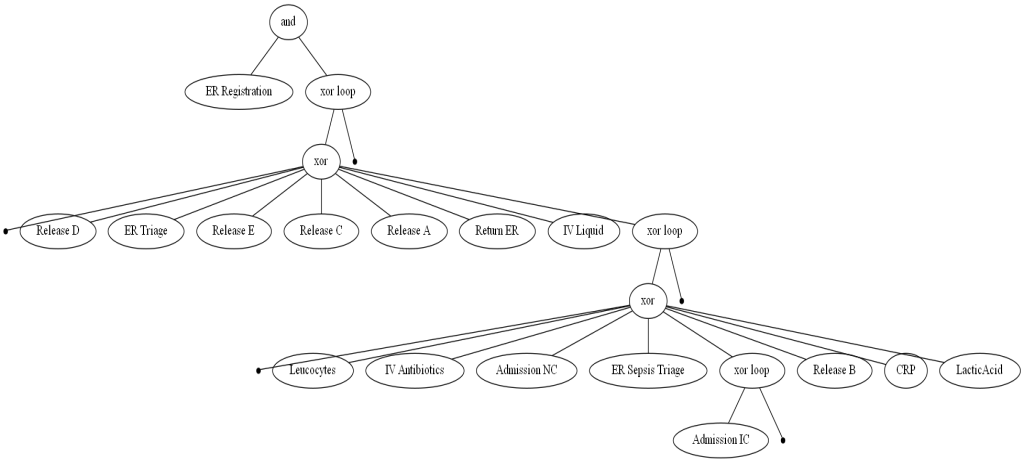
Figure B.12: The first three possible versions of the computed Process-Structure-Tree with privacy parameter $\epsilon = 1$ and for 1.050 Traces, for the SEPSIS-data cases (See B.11b on page xviii)



(a) The first possible version of the computed Process Structure-Tree



(b) The first possible version of the computed Process Structure-Tree



(c) The first possible version of the computed Process Structure-Tree

Figure B.13: The next three possible versions of the computed Process-Structure-Tree with privacy parameter $\epsilon = 1$ and for 1.050 Traces, for the SEPSIS-data cases (See B.11b on page xviii)

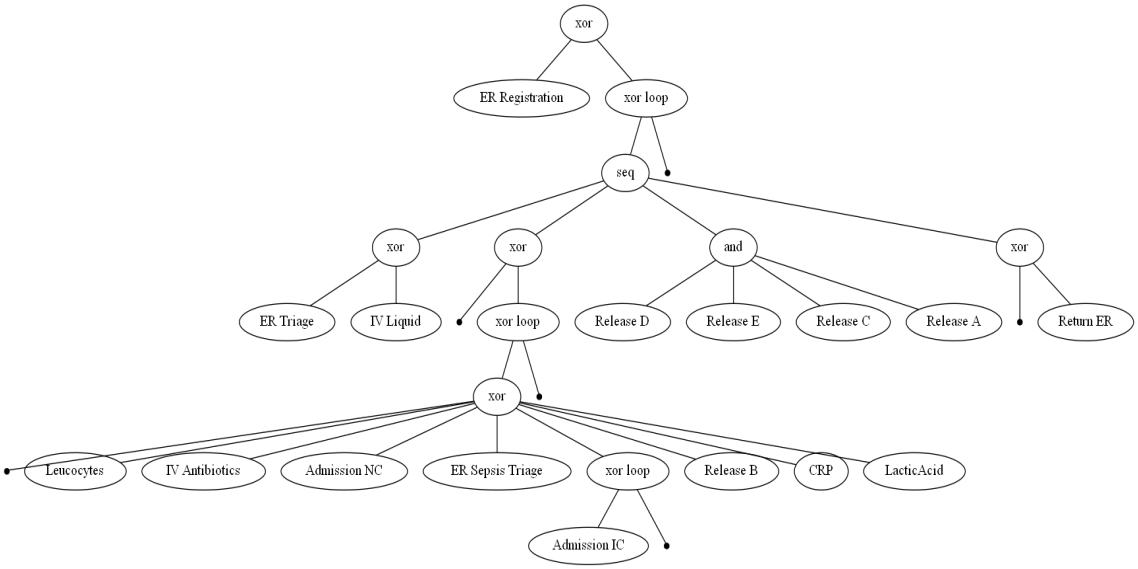


Figure B.14: The last possible version of the computed Process-Structure-Tree with privacy parameter $\epsilon = 1$ and for 1.050 Traces, for the SEPSIS-data cases (See B.11b on page xviii)