Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Speculative Reordering based on Neural Networks for a Latency-optimized Privacy Protection in Complex Event Processing

Robin Schweiker

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	M.Sc. Saravana Murthy Palanisamy
Commenced:	April 16, 2018
Completed:	October 16, 2018

Abstract

Nowadays, many devices are connected to the Internet. They are part of IoT applications, that offer us new services to improve our daily lives. A state-of-the-art paradigm to transform the raw data collected by these devices into meaningful information is Complex Event Processing (CEP). However, CEP systems often have access to privacy-sensitive information about the users, that they don't want to expose. If an application is not able to conceal this information, it significantly reduces its acceptance. Access control is one technique for such privacy protection. Most access control mechanisms protect privacy only at the level of single attributes of data or events. However, sensitive information is often revealed via patterns of events. Palanisamy et al. [PDTR18] proposed such a pattern-level access control component, as part of a CEP application, that conceals private patterns by reordering events. This component achieves high Quality of Service (QoS) but has the downside that it incurs high latency when the window size is significantly large. The reason is that it processes the event stream window by window and can therefore only start reordering when all events of the current window are known. In this thesis, we extend the reordering approach to a speculative reordering strategy, that can already reorder before all events of the window are available. The evaluation results show that this can drastically reduce latency and also has other advantages.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The number of IoT devices has increased enormously in recent years and will continue to rise [Meu17]. These devices collect raw data through sensors, that is required to operate various IoT applications. A state-of-the-art paradigm to process the raw data into meaningful information is Complex Event Processing (CEP). On the one hand, this enables us new applications, that improve our everyday life. On the other hand, the data that is necessary to operate them often contain privacy-sensitive information about the users, which they do not want to expose about themselves. If an application is not able to conceal this information, it significantly reduces its acceptance. A recent study by the Pew Research Center concluded that only 37% of Americans would immediately agree to share their location and driving speed with their car insurance company to get the possibility of discounts [Lee18]. The vast majority is uncertain or would reject it.

Access control is one prominent technique for such privacy protection. Most access control mechanisms protect privacy only at the level of single attributes of data or events. However, sensitive information is often revealed via patterns of events. A pattern is a sequence of events, that must occur in a specified order within a time-window to match. There are two types of patterns: private patterns and public patterns. Private patterns are defined by the user and contain privacy-sensitive information, while public patterns are required by the service provider to provide their services. In order to maximize the Quality of Service (QoS), private patterns should be concealed while preserving a maximum number of public ones.

Palanisamy et al. [PDTR18] proposed such a pattern-level access control component, as part of a CEP application, that conceals private patterns by reordering events. This component performs reordering window by window. It has to wait to reorder until all the events in the current window are available since it is necessary to know all public and private pattern that match in the complete window. Although reordering is effective, latency increases significantly with an increase in window size. There are further factors that have a negative impact on latency. It includes the number of patterns and the number of event instances that are part of patterns.

In this thesis, we propose a speculative reordering strategy, that extends the reordering approach, so that it can perform reordering before all events of the current window are available. The primary goal is to minimize or completely eliminate the latency after the window is complete.

To achieve this goal, a prediction model based on Deep Neural Networks predicts after every incoming event of a window, the probability for every pattern that it will occur in the full window. If these probabilities attain a certain threshold, then the window is already reordered even before all the events are available. For this purpose, the Neural Network is trained with historical data. The tradeoff, in this case, is that the speculative strategy requires little more computation power in exchange for saving latency. The evaluation results show that our speculative approach reduces latency in many situations, without much overhead.

1.1 Structure of the Thesis

The thesis is structured as follows: In chapter 2, we address related work regarding privacy protection in CEP and prediction models for sequential input. Then in chapter 3, we explain the basics of CEP, the non-speculative reordering algorithm and Recurrent Neural Networks - a type of Deep Neural Network that we use for the predictions. The above mentioned descriptions are essential for the proposed speculative reordering strategy. After that in chapter 4, we discuss the architecture and the training algorithm of the prediction model. We begin chapter 5 by describing the requirements for the speculative reordering strategy. In the remainder of the chapter, we focus on the design of our implementation. We also discuss a modified strategy that requires less computation time at the cost of QoS. In chapter 6, we evaluate the implementation and compare it with the non-speculative strategy using a real-world dataset. The last chapter consists of the conclusion and future work.

2 Related Work

This chapter is divided into two sections. First, we describe related work on pattern-level privacy protection in CEP. The second section addresses different approaches for prediction models.

2.1 Pattern-Level Privacy Protection in CEP

Privacy is essential in CEP, but most privacy-preserving mechanisms are preserving privacy only at the level of attributes. For example, Li et al. [LOW08] proposed such a privacy-preserving mechanism for data stream engines that process transactions of multiple users. It modifies attributes in transactions so that it can no longer be assigned to a specific user. Access mechanisms for single attributes are important, but this is not in the scope of this thesis.

Private information can also be revealed via patterns of events. The first one who analyzed the problem of private information revealed via patterns is He et al. [HBWN11]. They presented a theoretical approach, that is based on suppression of events. Wang et al. [WHRN13] took a similar approach. The authors introduced an algorithm that finds the event instances that have to get suppressed to maximize the Utility. They claim that their system works in near real time. It was shown by Palanisamy et al. [PDTR18], that a suppressed event, that is part of both a private and public pattern also impacts the public pattern and unnecessarily diminishes the QoS. They proposed a reordering strategy that instead of suppression changes the order of events, to conceal private patterns, but also preserve as many public patterns as possible. However, this approach delays the forwarding of the event stream. First, it has to wait until all events of the defined window are available, then it can start the reordering, which can take a long time with large windows. In this thesis, we extend their approach, to a speculative reordering strategy with the primary goal to reduce the latency to forward the privacy-preserved window.

2.2 Prediction Model

Since for a successful reordering, the patterns that should be concealed or kept must be known, a prediction model is needed that consumes a stream of events and outputs the probability for each pattern that it will match when all events of the window will be available.

Deep Learning is nowadays used in wide domains, and one promising subcategory called the RNNs if often the choice for supervised learning problems with sequential input. For instance, Choi et al. [CBS+16] proposed *Doctor AI*, a system that helps physicians to make diagnoses for their patients. It uses medical records (e.g., medication or previous diagnoses) to make predictions with up to 79% recall. Not only in the field of medical diagnosis but also in forecasting models such as stock forecasts [Jia16], language modeling [Kar15] etc. RNNs achieved state-of-the-art results.

Inspired by the results in other domains, the idea is to create a new simple architecture around an RNN, that predicts the probability of pattern matches. Traditional RNNs have trouble with learning long sequences, but there are enhanced versions like LSTM [GSC99] or GRU [CGCB14], that stabilize the learning process. These are mostly used in practice instead of normal RNNs. We will use LSTM for this thesis.

Besides Deep Neural Network architectures, another approach is the use of probabilistic models like Hidden Markov Models (HMM). For example, Alevizos et al. [AAP17] proposed an efficient probabilistic model based on HMM, that can predict the completion of patterns, defined as regular expressions. However, one limitation of HMMs and similar models is that their state space can explode for sophisticated models [Edd98]. For example, a stream of 100 events with 10 different event types can theoretically have up to 10^{100} different states. In reality, similar states are often treated as equal, which might lead to loss of information and in turn leading to a loss in quality of results.

A third approach would be a reinforcement learning algorithm, like Deterministic Policy Gradient [SLH+14], where after every new event the system decides if it is useful to reorder or not. The downside of this approach is that it would not return the probabilities of the patterns, rather only the decision to reorder or not. This makes it difficult to debug and optimize. Furthermore, the matching probabilities of the patterns are also needed to adjust the patterns, as we will show in Chapter 5.

3 Basics

In this chapter we describe the basics of Complex Event Processing (CEP). Furthermore, the non-speculative reordering algorithm by Palanisamy et al. [PDTR18] is briefly explained as this is used as a basis for our speculative reordering strategy. The third section of the chapter addresses the basics of Recurrent Neural Networks, that is required for predicting patterns.

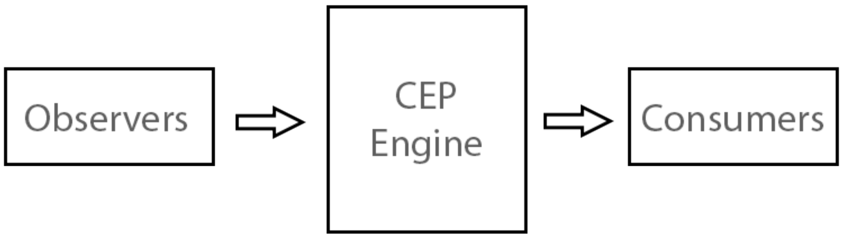


Figure 3.1: High-level view of a CEP system

3.1 Complex Event Processing

At a high-level traditional CEP systems consist of three types of components: Event observers (or sources), the CEP engine and consumers (or sinks) [CM12]. The relationship between the different components is illustrated by Figure 3.1. CEP systems can be seen as an extension to simple publish-subscribe systems, in that consumers express their interest in composite events [CM12]. An example of a composite event is the complex event *person enters the room* if a sensor notices that the door has opened followed by a movement in the room, detected by a second sensor. Simple events here are *door opened* and *movement detection in the room*. They are both tracked by the observers. Mostly the user of an IoT application can only control the observers. The CEP engine and the consumers are expected to be operated by a different stakeholder. Imagine a car insurance company, that offers discounts to customers who share driving information, collected by sensors. The source is the car, owned by the car owner and the CEP engine, and the sink are both operated by the insurance company, which uses the information to decide whether a user should get a discount.

3.1.1 Event Observers

Event observers (or sources) generate simple events that are sent to the CEP engine. These Events are often sensor data, but can also be other types, e.g., user input [CM12]. Every event is associated with a timestamp. A CEP system can have multiple observers.

3.1.2 Complex Event Processing Engine

This is the heart of the CEP system. The CEP engine consists of operators, that transform the incoming stream of basic events into complex composite events [CM12]. These new composite events are then forwarded to the next level of operators or the consumers. CEP operators can detect several types of patterns, e.g., sequence, negation, etc [WDR06]. For example, the complex event $E = SEQ(A, B)\&\bar{C}$ would be triggered if instances of events A and B have been observed, but not an example of event C. Furthermore, at least one instance of B must occur later than the first instance of A. For this work, only the sequence operator is considered since it is one of the most common.

3.1.3 Consumers

Consumers (sinks) are usually service providers who receive complex events from the CEP engine and provide services depending on these events. Mostly they are also the ones who define the complex rules that trigger the complex events [CM12]. The consumer offers the desired service, but can also gain access to sensitive information about the user, who is mostly a different stakeholder.

3.1.4 Pattern-Level Access Control

In addition to the described CEP system, Palanisamy et al. [PDTR18] introduced a Pattern-Level Access Control (PAC) component. This component is responsible for the pattern-level privacy protection. It conceals private patterns by reordering events, and forwards the privacy-preserved event stream window by window. It is located between the observers and the CEP Engine, as illustrated by Figure 3.2, because it is assumed, that the CEP engine is operated by a different stakeholder than the initiator of the events. As a consequence, the CEP engine only processes the event streams, where the private patterns have already been concealed. It is assumed, that all events of different sources are merged into a single stream of events before they enter the PAC component. For this thesis, we use this component to run the speculative reordering strategy.

3.2 Non-Speculativ Event Reordering

The purpose of reordering is to conceal the private patterns, while the QoS is maximized by maintaining as many public patterns as possible. This section explains the non-speculative reordering proposed by Palanisamy et al. [PDTR18]. Note that reordering is only necessary if the window has at least one private pattern. Otherwise, the window can be forwarded without further processing.

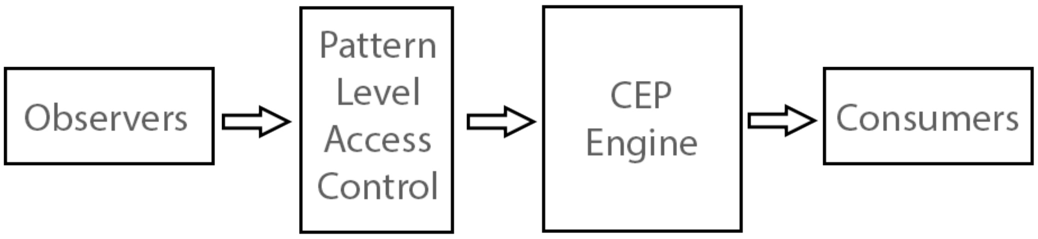


Figure 3.2: High-level view of a CEP system with PAC component

3.2.1 System Model

Let us describe the system with a simple example. Assume that the possible event types observed by the sources as $\sum = \{A, B, C, ...\}$. Every event e_i is assosiatet with a timestamp ts, so that $e_i.ts < e_j.ts$ for all $i < j$. The stream of events of the current window that has to be reordered is $S = (e_1, e_2, ...e_n)$ where e_i is an instance of any event type from $\sum$. A public pattern is represented as $Q_i = SEQ(E0, ...En)$ where $E_i \in \sum$. All event types E_i in the pattern have to occur in the specified order within a time frame. The QoS increases with an increase in the number of matched public patterns. For this work, we assume that S is a window of events that are part of the input stream within a specified time window. A public pattern Q_i is said to match if it occurs in S. The same definition applies to private patterns that are represented as $P_i \in P$, where P is the set of all private patterns. The aim is to conceal private pattern matches in S. The concealing process might impact the public patterns as well which in turn degrades QoS. The goal is, therefore, to maximize QoS while hiding private patterns by reordering events.

Each pattern $\in Q \cup P$ has a weight assigned, that indicated how important it is to conceal or preserve it. For this work, all public patterns are given a weight of 1 and all private patterns are assigned a weight of $w_p = \sum w_i + 1$. This means that revealing a private pattern always result in a negative utility. The following utility function [PDTR18] is used to measures the QoS after reordering:

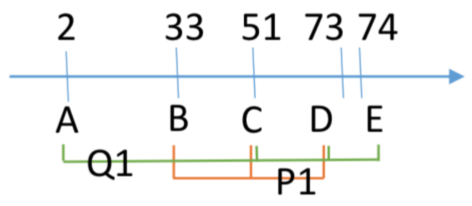
$$Utility(U) = (\sum_{i=1}^{\#public} w_i) - (2 \cdot \sum_{j=1}^{\#fp} w_j) - (\sum_{k=1}^{\#private} w_k)$$

It subtracts the summed weight of all matched private patterns from the summed weight of all matched public patterns. It also considers false positives, that is a match of a pattern after reordering, that was not a match in the original event stream. Since the sum of the matched private patterns contains both, true and false positive, the false positives (second term) are subtracted twice to reduce the overall result.

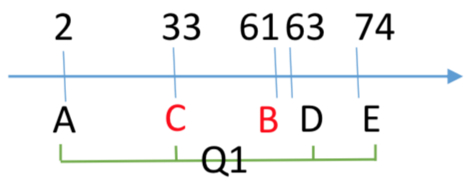
3.2.2 Algorithm

The reordering algorithm consists of a two-phase execution model [PDTR18]. The first phase is the *utility maximization phase* that finds an optimal set of pairs of event instances, that have to be reordered for maximum utility. It does that by testing all combinations of pairs that are part of private patterns. This phase returns two sets of ordered event pairs: set N contains pairs, one for each matched private pattern, those order has to be changed. Set R contains pairs of public patterns, that should be preserved. Phase 2 is called the *reorder obfuscation phase*. It performs the actual reordering with the sets N and R from phase 1. Therefore it adjusts the timestamps of the events in such a way that they look realistic, what makes it is hard for an adversary to detect afterward, that a private pattern has been concealed. The authors presented two approaches: one based on Integer Linear Programming (ILP) and the other a graph-based one. If it turns out, that the reordering with the sets N and R is unworkable, the next best solution of phase 1 is used for phase 2.

Figure 3.3a showes an example window with one private Pattern $P1 = SEQ(B,C,D)$ und one public pattern $Q1 = SEQ(A,C,D,E)$. The utility maximization phase returns the sets $R = ((A,C),(C,D),(D,E))$, that consists of event pairs of the public patterns and $N = ((B,C))$ which is part of the private pattern . Phase 2 then reorders the pair in N while preserving the pairs in R. The result, shown in Figure 3.3b indicates that the private pattern does not match anymore, but the public one is remaining. Note that also the timestamps of the events have changed.



(a) Before reordering



(b) After reordering

Figure 3.3: Reordering example with one private and one public pattern

3.3 Recurrent Neural Networks

Artificial Neural Networks are Machine Learning models, inspired by the biological information processing of the brain [Sch15]. They are mainly used for classification and regression problems. The flexibility of Deep Neural Networks has made them very popular in recent years and are nowadays used in many areas, like image processing [KSH12], speech recognition [AAA+16], natural language processing [LXLZ15] and many more. A Recurrent Neural Network (RNN) is a special type, that processes sequential input $X = (X_1, X_2, ..., X_T)$ step by step. For that, it concatenates the new input at every timestep t with the output of the last timestep $t - 1$ and forwards it through the same fully connected layer, as illustrated by Figure 3.4.

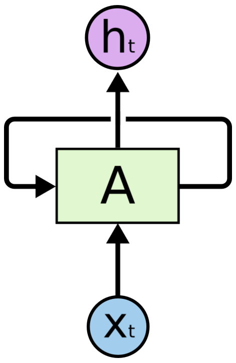


Figure 3.4: Traditional Recurrent Neural Network [Ola15]

The forward pass can be descriped by iterating over the equation $h_t = \sigma(WX_t + Uh_{t-1} + B)$ from $t = 1$ to T. X_t is the input at time t and h_t the output. The output of the layer is also part of the input for the next forward pass through the same layer. Often the Neural Network discards the outputs $h_0...h_{t-1}$ and only uses the last output h_T as result for further computations. W, U and B are learnable parameters and σ is a placeholder for the activation function (e.g. the sigmoid function) that makes it able to learn non-linear dependencies between input and output.

However, it was shown, that traditional RNNs have the problem, that they are difficult to train for long sequences [BSF94]. Thus for this thesis, we use a Long Short-Term Memory (LSTM) layer [GSC99]. It can be seen as an extension of traditional RNNs. The difference is that each node of the layer also has an additional internal cell state, that is updated after every forward pass, with the help of gates, that are controlled by learnable parameters. In that way, the model is able to learn when to remember or forget something. This makes it possible to learn with long sequences efficiently.

There are several slightly different implementations for LSTMs. The implementation with the following functions [Wik18] is widespread and therefore assumed for this thesis.

Variables

- d: Number of features of each input vector $\in X$
- n: Number of nodes in the LSTM Layer.
- $f_t, i_t, o_t \in R^n$: Parameters for forget-, input- and output gates

- $c_t \in R^n$: Cell state vector
- $W \in R^{n \times d}$, $U \in R^{n \times n}$, $b \in R^n$: Learnable weights and bias.

Functions

- Sigmoid: $\sigma_g(x) = \frac{1}{1+e^{-x}}$
- Tangens hyperbolicus: $\sigma_c(x) = \sigma_g(2x) - 1$

$\circ$ denotes the element-wise product

Forget Gate

$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$

This gate defines how much information of each cell should be forgotten. The sigmoid function computes values between 0 and 1. A 1 represents “keep everything” while 0 represents “forget everything” [Ola15].

Input Gate

$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$

This gate decides how much of the new input should be stored in the cell, by computing numbers between 0 and 1.

Cell State Update

$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$

This equation updates the last cell states C_{t-1} into the new cell states C_t by using the output of the forget and input parameters, previously defined.

Output

$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$

$h_t = o_t \circ \sigma_c(c_t)$

The output gate o_t computes weights between 0 and 1, that are multiplied with the cell states, where σ_c has been applied before. This is the final output of the layer at the current iteration.

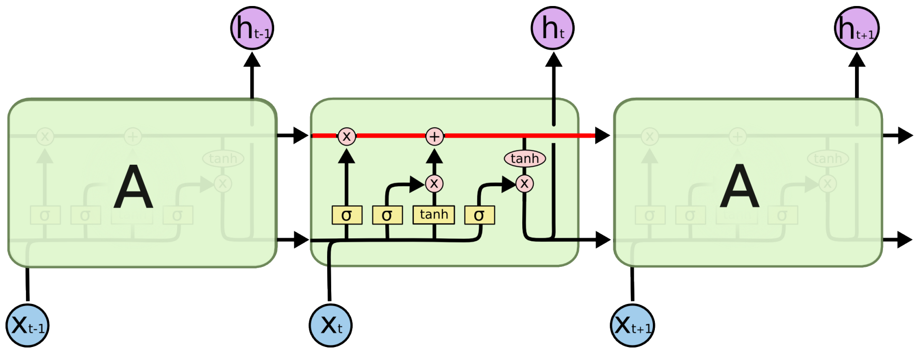


Figure 3.5: LSTM node [Ola15]

Figure 3.5 illustrates the internals of a single node in an LSTM layer. The red line contains the internal cell value. Just as with traditional RNNs, the final output is computed by iterating over the equations from $t = 1$ to T. Matrix multiplication is denoted with $\otimes$ and addition as $\oplus$.

Since all equations can be derived, the LSTM layer is trained by backpropagation as is usual with Neural Networks [Nie16]. It is also possible to stack several LSTM layers, which is useful for example with natural language processing [Kar15]. Furthermore, LSTM layers are often used as a part within a larger architecture consisting of many different building blocks. This is also the case with the prediction model that we describe later in this thesis.

4 Prediction Model

In this chapter, we describe the prediction model, which is an essential part of the PAC component. It composes of a Neural Network with an LSTM layer [GSC99], whose architecture is shown by Figure 4.1. The model consumes each arriving event separately and computes the probability for every private and public pattern that it will match later in the current window. The model has a hidden internal state that is updated with every new arriving event. More details about the architecture is explained in Section 4.1. The speculative reordering strategy uses these probabilities to estimate whether it is useful to start a reordering or not. The information is also used to adjust the patterns and weights, details are explained in the next chapter.

The model is trained via backpropagation and supervise learning [Nie16]. The training input for the model is a tuple $T = (x \in X, y \in Y)$. x is a vector and represents a window of events and $x_t \in x$ represents the event at timestep t. Furthermore, y is a vector, that contains the information for every pattern if it matches in window x or not. Through the training, the internal parameters of the model are adjusted to learning a mapping from x (and any subset of x) to y. Note that y is a vector $\{0, 1\}^n$ that only consists of binary numbers, where n defines the number of patterns. In the prediction phase, $\hat{y}_t$ which is here the output of the model at timestep t is a vector $(\hat{y}_i)^n$, where $\hat{y}_i \in [0, 1]$. For example, suppose there are 2 patterns $P_1 = SEQ(A, C, D)$ and $P_2 = SEQ(C, A, B)$ and a event stream $s = (A, B, C, D)$, then $x = s$ and $y = (1, 0)$. At timestep 3 after A,B and C of the window s is passed as input through the model, the output $\hat{y}_3$ could look like $(0.5, 0)$. After D is also passed through the model, the window is complete and $\hat{y}_4$ should be $(1, 0)$. For the prediction model, it makes no difference whether a pattern is private or public since this information doesn't affect how it computes the matching probabilities. However, to make the output more understandable, the patterns are arranged so that the first elements of the output vector $\hat{y}_t$ are private patterns, followed by the public ones. The model was implemented and trained with the Python library Pytorch [PGC+17], which is a flexible and fast deep learning research platform.

4.1 Architecture

This section describes all components that the prediction model consists of. The whole architecture is showed in Figure 4.1. It can be seen as a pipeline that transforms the input into output over multiple, successive steps. The different layers of the architecture are described in the following.

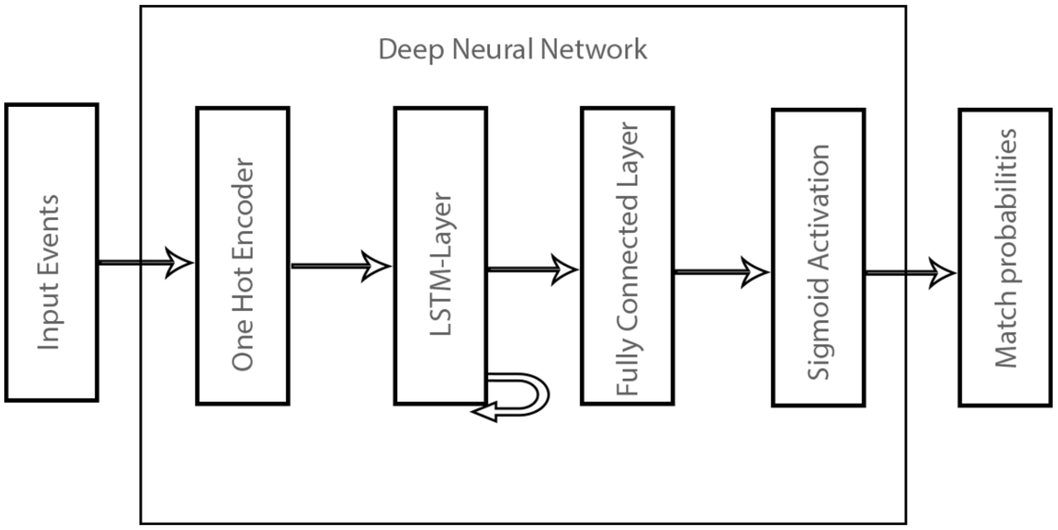


Figure 4.1: Prediction model architecture

4.1.1 Input and Output

Every new event that arrives gets transformed into an integer representation. For example, the integer representation for C is 3 . Each event $x_t \in \mathbb{N}$ gets passed separately through the network. The output after the t-th event is a vector $\hat{y}_t \in R^n$ where n defines the total number of patterns. During training it is also possible to pass the whole sequence x at once, then the model returns the complete history of output for all timesteps, this makes it faster to train, especially on GPUs.

4.1.2 One Hot Encoder

Since the integer values of the input are in a nominal scale, where the order has no meaning, a natural approach is to transform them into one hot encoding, that means that every integer becomes a vector of zeros with a maximum of one 1. For that a $c \times c$ identity matrix, where c is the number of different event types, is used to convert each value $v \in \{1, 2, ..., c\}$ into $c[v]$, which is the v-th row of the identity matrix. For example the one hot encoding of the number 3 with $c = 4$ is $(0, 0, 1, 0)$.

4.1.3 LSTM Layer

This is the most crucial part of the model. It forwards each arriving encoded event through the LSTM layer, that uses the event to compute the matching probabilities and update its internal state. This layer can learn complex and nonlinear dependencies between input events and desired output. For example, it is theoretically able to learn how to count, how many events of a particular pattern have already been passed through the network. For every new event, this layer executes multiple functions, as described in Section 3.3. They take as arguments the one hot encoded new event and the internal state that has been calculated with all the previous events.

4.1.4 Fully Connected Layer

This layer forwards the output of the previous LSTM layer through a standard fully connected layer and applies a sigmoid function that squeezes the value of each output number to a value between 0 and 1. Every output number represents a probability for a pattern that it will match in the current window. If the pattern has already matches, it is expected to be 1.

$$Sigmoid(x_t) = \frac{1}{1+e^{-x_t}}$$
$$\hat{y}_t = Sigmoid(x_t \cdot W + B)$$

x_t is the input for the layer at timestep t and is also the output of the previous LSTM layer. W and B are learnable parameters. $\hat{y}_t$ is the output of the layer and the whole model at timestep t.

4.2 Training of the Model

In this section we explain how the model above mentioned is trained. As it is usually the case with Neural Networks, training is performed iteratively with a forward pass followed by a backward pass. In the forward pass, all the events of the window are passed through the network at once. The output $\hat{y}$ is a Txn Matrix, where $\hat{y}_{t,i}$ is the matching probability of pattern i after step t. A loss function calculates the difference between the desired and the actual output. The purpose is to adjust the parameters of the model so that the loss is minimized. For this model a binary Cross-entropy loss is used, that forces the model to learn to approximate the real likelihoods that a pattern matches. The complete loss is the average binary Cross-entropy loss across all patterns and timestamps.

$$loss(y, \hat{y}) = \frac{1}{T \cdot n}(\sum_{t=1}^{T} \sum_{i=1}^{n} -(y_i \cdot log(\hat{y}_{ti})) + (1 - y_i) \cdot log(1 - \hat{y}_{ti}))$$

Furthermore, the implementation is optimized with batch processing, that means, that multiple windows are routed through the network at once, this makes it more stable and faster to train on GPUs. For simplification, this is not represented in the loss function above.

In the backward pass, the derivative of every parameter in the model corresponding to the loss is calculated. To update the parameters, we used the Adam optimizer [KB14] with an initial learning rate of 0.01 that is multiplied by 0.98 after every epoch. One epoch is when every window in the dataset has been passed through the network. The training stops when the accuracy loss stops declining for a few epochs. To evaluate the progress of the training, after each epoch the loss of a validation dataset is computed. It is important that this dataset has no duplicates with the training dataset because this would make it difficult to detect overfitting. It is advantageous to use a GPU, as it makes the training much faster. If overfitting affects the accuracy of the validation data, then an additional Dropout layer [SHK+14] between the LSTM layer and the fully connected layer can improve the accuracy. A better solution to avoid overfitting is to have a large amount of training data. Therefore we did not use dropout in the proposed model.

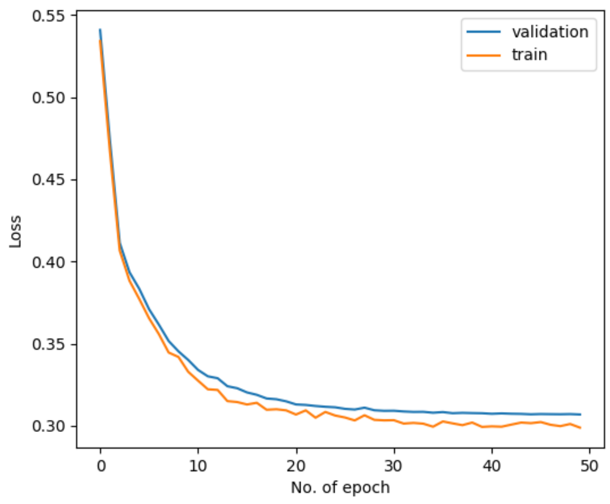


Figure 4.2: Train and validation loss

Algorithm 4.1 Training Algorithm for the Neural Network

```
procedure TrainModel
    for epoch in 1..50 do
        ShuffleDataset()
        for each batch in Dataset do
            $x, y \leftarrow batch$
            Model.InitLSTM()
            Model.ResetGradients()
            $predictions$ ←Model.Predict($x$)
            Loss ←GetLoss( $predictions$,$y$ )
            Backpropagate( $loss$ )
            AdamOptimizer.Step( )
        end for
        AdjustLearningRate( )
        EvaluateEpoch( )
        if validation accuracy stops improving then
            break
        end if
    end for
end procedure
```

Figure 4.2 shows how the training and validation loss decreases with time. Note that for this Neural Network it is almost always impossible to reach a loss of 0 because of the randomness in the data. A loss of 0 would mean, that for every pattern and timestep the model knows with confidence of 100% whether it will match or not at the end of the window.

Algorithm 4.1 shows the training algorithm. It runs for a maximum of 50 epochs, but has the option to quit, when the loss stops improving. In every epoch, the training dataset, split into batches of size 128 is passed through the model. Every forward pass is followed by a backward pass to calculate the loss and update the parameters with the Adam optimizer. The learning rate starts high at a value of 0.01 and is reduced after every epoch. If the learning rate is too small, the training needs more time and the likelihood to stuck in a local minima increases. Another important aspect is that the gradients and the LSTM state need to be reset after every new batch.

4.3 Factors that Influence the Prediction

There are some factors that influence how reliable the predictions are. We briefly describe them in this section.

Randomness in the Dataset

A significant factor that has a negative impact on predictions is the amount of randomness or noise the event stream contains. This makes it impossible for any prediction model to make 100% correct statements. Therefore, the output is not just a binary value that indicates whether a pattern matches, but a probability.

Amount of Training Data

Any predicting algorithm especially prediction with Neural Networks are known to need much training data. Otherwise, they tend to overfit. Therefore, for the experiments, it was ensured that the selected dataset contains a large number of events. The number of training data can be artificially increased by generating overlapping windows from the generated windows, e.g., by connecting the second half of a window to the first half of the following window.

Structure and Distribution of Patterns

The average matching rate of a pattern influences the prediction. This means that if a pattern is present in almost every window, the prediction model also takes this by default at the beginning and adjusts this prediction with every further event.

The number of different event types does not have a significant influence, but more event types lead to the fact that the prediction model needs more training parameters, which slows down prediction and training. Another factor is the length of the patterns. Longer patterns make it obviously more difficult to train.

5 Speculative Reordering Strategy

In this chapter, we explain the complete speculative reordering strategy inside the PAC component. It is built around the non-speculative graph-based reordering algorithm and the prediction model, described in the previous chapters. The strategy consists of two callbacks. The first callback *NewEvent*, shown in Algorithm 5.1 is externally triggered after every event that arrives from a source. The new event is passed as a parameter. After the last event of a window arrives, it forwards the privacy-preserved window to the CEP engine. This routine also performs the predictions and decides when to start the reordering. The second callback *ReorderingResponse*, shown in Algorithm 5.2 is executed when the reordering of a subset of the window has finished. Both algorithms are described in detail later this chapter. They both have access to shared global variables that are used to exchange information.

5.1 Requirements

Before describing the two algorithms, it is important to discuss the requirements that are to be met by our speculative reordering strategy. The speculative reordering strategy should obtain the positive aspects of the non-speculative one, while also improve some of its downsides.

5.1.1 Utility

The most significant advantage of the non-speculative approach is that it mostly results in a higher QoS when compared to suppression of events [PDTR18]. This advantage should, of course, be maintained with the speculative approach.

5.1.2 Latency

A disadvantage of the non-speculative reordering to consider is the latency, the additional time the PAC component needs after a window is complete until it can forward the privacy-preserved window. The main goal of this thesis is to eliminate or minimize this time taken for reordering after the window is complete. This is made possible in the speculative case since the event stream is already reordered before even the last event of the window is arrived.

5.1.3 Computation Time

For the non-speculative reordering, it is just the time for the reordering, while for the speculative approach it also includes the time for all the predictions and the preparation for the reordering. In theory, an efficient speculative reordering algorithm would consume less computation time, since it only reorders a subset of the window, that consists of the already known events. The time taken for speculation should be as small as possible. Another factor that is to be considered is the confidence with which the partial event stream is reordered since a wrong reordering because of a false prediction would result in depreciation of the utility. Not only utility but also computation time and latency since there might be a need for another reordering. Even though it is not possible to have 100% correct predictions due to randomness in the event stream, we try to reduce the number of false predictions.

Low computation time is particularly important, because many IoT devices, where the speculative reordering strategy could later be implemented have limited computation power, that they also have to share for other tasks. Furthermore, many devices own a battery, that drains faster if the processor is active.

5.1.4 Failure Rate

In the worst case, the PAC component is unable to reorder the event stream, and private patterns get revealed. This could happen in case of a timeout, what often occurs with the non-speculative reordering when the window is large or many event instances are part of patterns. Another error that can happen is an out-of-memory-exception.

5.1.5 Generic

It is important for the reordering strategy to be generic so that it can be used for any CEP application. For example, it should be able to conceal private patterns independent of the window-size or amount of patterns. In particular, it should also handle some noise or randomness in the event stream. This also includes, that the prediction model is able to learn the real matching probabilities independent of the dataset.

5.2 Implementation

In this section, we describe how the two callback algorithms of the speculative reordering strategy are implemented considering all the requirements mentioned above.

5.2.1 Global Variables

Before going into how the algorithms are implemented we present a list of global variables that can be accessed by all routines.

EventStream: This is a list that contains all arrived events in the original order. Each event is associated with a timestamp. Every new event that arrives from a source is appended to this list.

ReorderedStream: This list stores the reordered event stream after reordering of a partial window. Each new arriving event is also appended to this list. When the window is completed, this list is forwarded to the PAC component.

Utility:: This variable represents the current utility, that the speculative reordering achieved with the available events. It is updated after a reordering has been successfully performed or after a new event arrives. When all events of the window are available, this variable shows the final utility.

5.2.2 Callback Algorithms

The algorithm for the *NewEvent* callback is described by Algorithm 5.1. Input events are passed into this routine one by one. If the event that is passed as an argument to this routine is the last event of the current window and a reordering with the previous events is still running in the background, the algorithm has to wait until that reordering is finished. The reason is that the reordering might result in a negative utility. In this situation, a new reordering attempt with the complete window would be necessary. After that, the new event is appended to the list *eventStream*. If a reordering has already been completed, which can be recognized by the fact that *ReorderedStream* is not empty, the new event is also appended to this list, and the utility score is updated because the new event might have changed it. After that, the prediction model is executed, that returns the matching probabilities for all private and public patterns. Note, that even if it is foreseeable that a reordering cannot be started at the current time, since for example, a reordering is already running in the background, it is still important to call the prediction subroutine, as this will update the internal state of the prediction model.

Algorithm 5.1 New Event Callback

```
procedure NewEvent($event$)
    if $EventStream.length == WindowSize - 1$ then
        JoinReorderingThread( ) // wait until reordering thread terminates if it is running
    end if
    EventStream.push($Event$)
    if $ReorderedStream$ != $null$ then
        ReorderedStream.push($event$)
        Utility ← $UpdateUtility$(event)
    end if
    predictions ← $PredictMatches$(event)
    if $reordering\ thread\ not\ running$ and $utility < 0$ then
        if $(\prod_{i=0}^{\|privatePatterns\|} max(predictions[i], 1 - predictions[i])) \geq Threshold$ then
            ReducedPrivPatterns, ReducedPubPatterns ← $GetReducedPatterns$(predictions)
            weights ← $AdjustWeights$( ReducedPrivPatterns, ReducedPubPatterns, predictions)
            if $\forall p \in ReducedPrivPatterns : len(p) \geq 2$ then
                Reorder($EventStream, ReducedPrivPatterns, ReducedPubPatterns, weights$)
            end if
        end if
    end if
    if $EventStream.length == WindowSize$ then
        JoinReorderingThread( ) // wait until reordering thread terminates if it is running
        if $Utility > 0$ then
            if $ReorderedStream$ != $null$ then
                Forward($ReorderedStream$) // forward to CEP engine
            else
                Forward($EventStream$) // forward to CEP engine
            end if
        end if
        Reset()
    end if
end procedure
```

If no successful reordering has been performed in the past and no reordering is currently running in the background, the algorithm continues and computes a confidence score, this is the probability, that the binary prediction of all private patterns is right. If the matching probability of a private pattern is smaller than 0.5, it is assumed, that the pattern won't complete. Otherwise, it is assumed, that it will or is already completed. Public patterns are not considered for this, because they would dramatically decrease the confidence score and have no significant impact on the utility after the reordering. The reasons are that a public pattern, that has been predicted to not complete not necessary get destroyed after reordering and the weight of a public pattern is much smaller than the weight of private one. On the contrary, a public pattern, that was predicted to match, but won't, increases slightly the probability of false positives, which leads to a worse utility. If the confidence score is greater than a custom threshold, the algorithm continues by adjusting the patterns and weights by calling the subroutines *ReducedPatterns* and *AdjustWeights*.

Subroutine GetReducedPatterns

This subroutine returns reduced versions of the patterns. Since the reordering is performed only on a subset of the window, all private and public patterns, that are expected to complete, have to be adjusted, to that subset. For example if an individual patterns $P1 = SEQ(A, B, C)$ is predicted to complete and the previous event stream $S' = (A, A, B, D)$, the pattern must be reduced to SEQ(A,B), because otherwise, the reordering subroutine would not consider it as a match. If a private pattern has a length of 0 or 1 after it is simplified, it does not make sense to reorder, because it is impossible the conceal this pattern. All patterns that are expected to not complete are not passed to the reordering subroutine.

Subroutine AdjustWeights

The weight indicates how important a pattern is. This subroutine multiplies the original weights of the remaining patterns with the matching probabilities. This has mainly an effect of patterns that have previously the same weight. Then the pattern with the higher matching probability is treated as more important.

Reordering

After the patterns and weights have been adjusted, the last step is to check if the private patterns are long enough to allow a successful reordering, i.e., if at least the first two events of each pattern are known. After that, if all requirements are met, the *Reorder* subroutine is called. It starts a new thread, which requests the reordered stream from a server application, which contains the graph-based version of the reordering algorithm by Palanisamy et al. [PDTR18]. Details about this are explained in Section 5.2.3.

When the result is returned, the *ReorderingResponse* callback, shown in Algorithm 5.2 is executed. The result is passed as a parameter. First, this algorithm extends the new reordered stream with the events, that arrived from the sources in the time that the reordering was running. The lists *ReorderedStream* and *EventStream* now have the same length. Thereafter it calculates the utility of the reordered stream, which is done by the *ComputeUtility* subroutine.

Algorithm 5.2 Reordering Response Callback

```
procedure ReorderingResponse($NewReorderedStream$)
    ReorderedStream $\leftarrow$ $Merge$(NewReorderedStream, EventStream)
    Utility $\leftarrow$ $ComputeUtility$(ReorderedStream)
end procedure
```

Algorithm 5.3 Reset Variables

```
procedure Reset()
    Utility ← $-1$
    ReorderedStream.removeAll()
    EventStream.removeAll()
    $ResetPredictionModel(\,)$
end procedure
```

After the complete event stream has been processed, the *NewEvent* callback sends the privacy-preserved event stream to the CEP engine. Finally, the subroutine *Reset*, described by Algorithm 5.3 is called, that sets all variables back to their default values. This also includes resetting the state of the prediction model.

Utility

The calculation of the utility involves the two functions $ComputeUtility$ that are executed by the *ReorderingResponse* callback and $UpdateUtility$ that is executed by *NewEvent*. The function $ComputeUtility$ calculates the utility for a stream with the equation described in Section 3.2. It considers the original private and public patterns (not the reduced versions of it). For each possible pattern, it is counted up to which event it matches in the event stream. These values are stored in an array. If a number in that array is the same as the pattern length, it is a match. The $UpdateUtility$ function updates the array with every new event and also updates the utility as necessary. If the new event completes a private pattern, the utility becomes negative, and the event stream must be reordered again.

If the utility at the end of the window is still negative, it is assumed that nothing is passed on to the CEP engine, this way one has at least a utility of 0. This could be optimized by trying a more performant suppression strategy when the reordering strategy fails.

5.2.3 Reordering Server

In this section we describe the technical implementation of the graph-based reordering, that is implemented as a simple REST server as part of the PAC component. The reason is that it has to be executed in a Python 2 environment, while the prediction model and the rest of the algorithm is implemented with Python 3. Since all the code is executed on the same device, the latency, that comes with the REST-call has nearly no impact on the results (<0.01s per reorder).

The reordering algorithm is the same as proposed by Palanisamy et al. [PDTR18]. However, the patterns that are passed to the server are reduced, compared to those that would be used in the non-speculative approach. Furthermore, the reordering is only performed on a subset of the event stream, which makes it faster.

Request

The reordered event stream with the corresponding timestamps is requested by sending an HTTP request with a JSON, that contains the original event stream, the timestamps, and patterns with weights. The *reorder* subroutine that is called from Algorithm 5.1 starts a new thread that executes this request.

The following listing shows a simple example with 16 events and six patterns, of which two are private.

```
{
"eventStream" : [1, 2, 3, 5, 4, 2, 4, 1, 5, 7, 5, 4, 9, 4, 2, 8],
"timestamps:" : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] ,
"patternsPrivate": [[1,3,5],[4,9,2]],
"patternsPublic":  [[1,3,5,4],[2,8],[1,7,9,8],[4,1,5]],
"patternsPrivateWeights": [5,5],
"patternsPublicWeights": [1,1,1,1],
}
```

Response

The server responses with a JSON that contains the reordered event stream, the new timestamps, and the information whether the reordering failed. When the thread that called the REST Server receives the JSON response, the *reorderingResponse* callback, described in Algorithm 5.2 is executed with the reordered stream as parameter.

The listing shows the response for the exemplary request.

```
{
"eventStream" : [2, 3, 1, 5, 4, 2, 4, 1, 5, 7, 5, 4, 4, 2, 9, 8],
"timestamps:" : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] ,
"error": false
}
```

In this example, both private patterns don't match anymore in the reordered stream. However, also one public pattern has been destroyed. This could not be prevented because the private pattern SEQ(1,3,5) is a subset of the public pattern SEQ(1,3,5,4). The utility of this example would be 3, compared to -12 without reordering.

5.3 Reasons for Multiple Reorderings

In this section, we briefly discuss the reasons why sometimes several reorderings are necessary. The speculative reordering strategy tries to minimize the number of reorderings for a window, as several reorderings dramatically increase the computation time. Ideally, a subset of the window is reordered only once, and this as early as possible, that when the window is complete, no further calculations are necessary and the processed privacy-preserved window can be immediately forwarded as a whole to the PAC component. However, there are three cases, that force a further reordering.

5.3.1 Errors

If the reordering server crashes or doesn't respond, it is obviously necessary to start the reordering again. This usually happens when the task is too complex. Important factors for this are the window size and the number of event instances that are part of private patterns. Since in most cases all other reorderings also fail, one could also cancel the processing of the current window at this point. Then computation time is saved, but at the expense of the utility, since in some cases, a further reordering works. Note, that if the speculative strategy fails, it is also expected that the non-speculative strategy also fails. On the contrary, if the non-speculative strategy does not throw an error, the speculative strategy could still work, since it only reorders a substream with reduced patterns.

5.3.2 Wrong Predictions

A private pattern may be revealed due to a prediction that turns out to be wrong or inaccurate afterward. Then another reordering is necessary. It could happen similarly for public patterns, but this would massively increase the number of reorderings and has little influence on the utility since private patterns have significantly lower weights and are often not destroyed, even if they are falsely expected not to match. Therefore a new reordering will not be started if a public pattern was incorrectly predicted.

5.3.3 Multiple Instances of Events

Imagine the stream $S = (A, B, D, A, B, C)$ with the private pattern $P_1 = (A, B, C)$. The speculative reordering strategy might reorder the Substream $S' = (A, B, D)$ after theses 3 events are available, but the following events A,B,C still reveal the pattern. In this case, the event stream has to be reordered again, immediately after the problem becomes visible.

5.4 Only One Reordering

Since computation time is an important factor we propose additionally a modified version of the *NewEvent* callback, described by Algorithm 5.4. It is more computation efficient but achieves a lower utility. Inversely, if there is no focus on conserving the processor's resources, this approach is less suitable. The main difference is that the algorithm only performs a maximum of one reordering. Another factor why this algorithm is more resource efficient is that after the first reordering is started, it can be stopped with predicting the matching probabilities for all further events. Even if the current window is complete, it is necessary to wait if a reordering is executed in parallel. The only reason is that the final window is forwarded from this callback to the CEP engine. If the utility at the end of the window is still negative, a utility of 0 is assumed as explained in Section 5.2.2. This approach is only an alternative to the standard algorithm if it achieves a positive result in most cases. This can be investigated by evaluating the standard algorithm that allows multiple reorderings to know how often multiple reorderings are necessary and choose either one of the callback algorithms accordingly.

Algorithm 5.4 New Event Callback Alternative

```
procedure NewEvent($event$)
    EventStream.push($event$)
    if $ReorderedStream$ != $null$ then
        ReorderedStream.push($event$)
        Utility ← $UpdateUtility$(event)
    else if $Reordering\ thread\ not\ running$ then
        predictions ← $PredictMatches$(event)
        if $(\prod_{i=0}^{\|privatePatterns\|} max(predictions[i], 1 - predictions[i])) \geq Threshold$ then
            ReducedPrivPatterns, ReducedPubPatterns ← $GetReducedPatterns$(predictions)
            weights ← $AdjustWeights$(ReducedPrivPatterns, ReducedPubPatterns, predictions)
            if $\forall p \in ReducedPrivPatterns : len(p) \geq 2$ then
                Reorder($EventStream, ReducedPrivPatterns, ReducedPubPatterns, weights$)
            end if
        end if
    end if
    if $EventStream.length == WindowSize$ then
        JoinReorderingThread( ) // wait until reordering thread terminates if it is running
        if $Utility > 0$ then
            if $ReorderedStream$ != $null$ then
                Forward($ReorderedStream$) // forward to CEP engine
            else
                Forward($EventStream$) // forward to CEP engine
            end if
        end if
        Reset()
    end if
end procedure
```

6 Evaluation

In this chapter, we evaluate the accuracy of the predictions of the Neural Network, followed by a comparison between the speculative and the non-speculative reordering strategies.

Table 6.1: Dataset

ID	Type	Timestamp	Value
1	step_detector	2017-06-29 07:45:16.506	[1.0]
2	battery	2017-06-29 07:45:16.514	['100']
3	IN_VEHICLE	2017-06-29 07:45:17.407	['STILL: 98']
4	accelerometer	2017-06-29 07:45:21.530	['-0.852627', '4.7876444', '9.029224']
5	wifi	2017-06-29 07:45:22.007	[7207, 3544, 5500]

6.1 Dataset and Setup

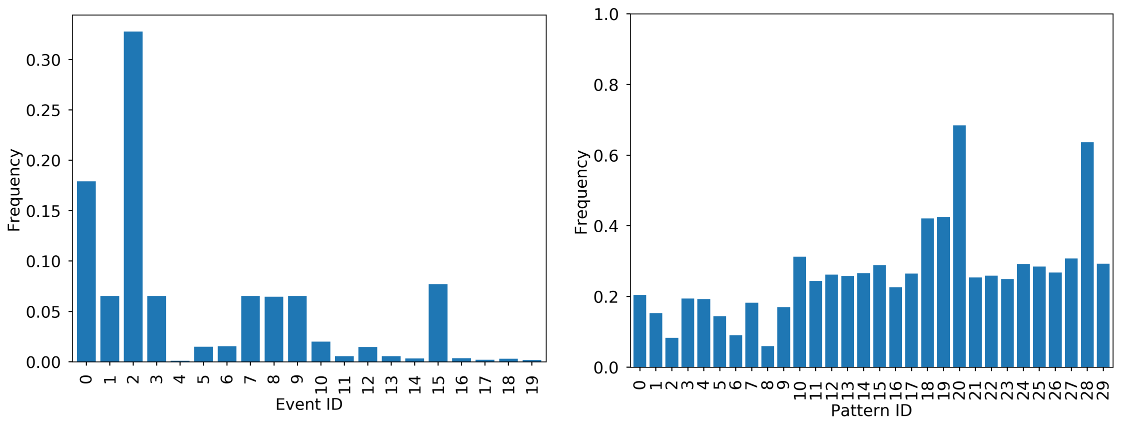
We ran all experiments on a MacBook Pro with 16 GB of RAM and a 2.2 GHz Intel Core i7 processor, that has 4 cores. For the training of the Neural Network, we used a server with a Nvidia GTX 1080 Ti GPU.

In order to evaluate the suitability under challenging conditions, all experiments were performed with a real-world dataset [FLJ+17] that contains a lot of randomnesses. The dataset consists of 1.5 million events captured on a smartphone, that a person was using for multiple days in his everyday life. The first 5 rows of the dataset are shown in Table 6.1. We removed the column *Value* and modified the column *Timestamp* so that the time between 2 successive events (arrival rate) is 1 second. The patterns were randomly generated with the available event types, but the majority was filtered out by hand because they were not suitable. It was ensured, that all patterns occur in over 1% of the windows and the private patterns do not contradict themselves, what would have made it impossible to reorder. They all have a length between 2 and 4 events. Additionally, we removed some event duplicates. The event stream was divided into 15000 windows of size 100 with 90% used to train the model and 10% for evaluation. Note that for training overlapped windows were also included to increase the training data. The dataset consists of 20 different event types that occur with probabilities from 0.08% to 32.77%. Figure 6.1a shows the actual distribution. The probabilities of the individual patterns to occur in a window are shown in Figure 6.1b. They are in the range of 2% to 39% with a mean of 24%. Both, the speculative and the non-speculative reordering strategies were tested in a real-time simulation by sending the events of the described dataset at specified intervals. The reordered windows were later forwarded and checked for the utility. For each window, 3 private and 20 public patterns were selected. Note that not all selected patterns have to match in a given window, it was only ensured, that at least one private pattern occurs in each window.

The three private patterns were chosen randomly of a pool of 10 for every window, while the 20 public patterns were always the same. It would have been complicated to test the experiment with a larger variety of patterns because the prediction model has first to learn the selected patterns. To evaluate the impact of the different parameters, always the parameter to be measured was changed, while the others remained at the default values. Table 6.2 shows the parameters that can be varied and also their default values.

Table 6.2: Default parameters

Parameter	Value
Window-size	100 events
Arrival rate	1 second
Number private patterns	3
Number public patterns	20
Number event types	20
Threshold	0.85



(a) Event distribution

(b) Pattern distribution

Figure 6.1: Distribution of patterns and events

6.2 Prediction Model Evaluation

First, we evaluate the accuracy of the outcome of the prediction model. Figure 6.2 shows the average precision, recall and accuracy values for one pattern after each timestep for the exemplary window-sizes 100 and 200. Different patterns were used for different window-sizes to meet the conditions described in Section 6.1. The following formulas were used to calculate the values.

$$Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP+FN}$$

$$Accuracy = \frac{TP+TN}{TP+FN+FP+TN}$$

TP: The pattern was correctly predicted to match.

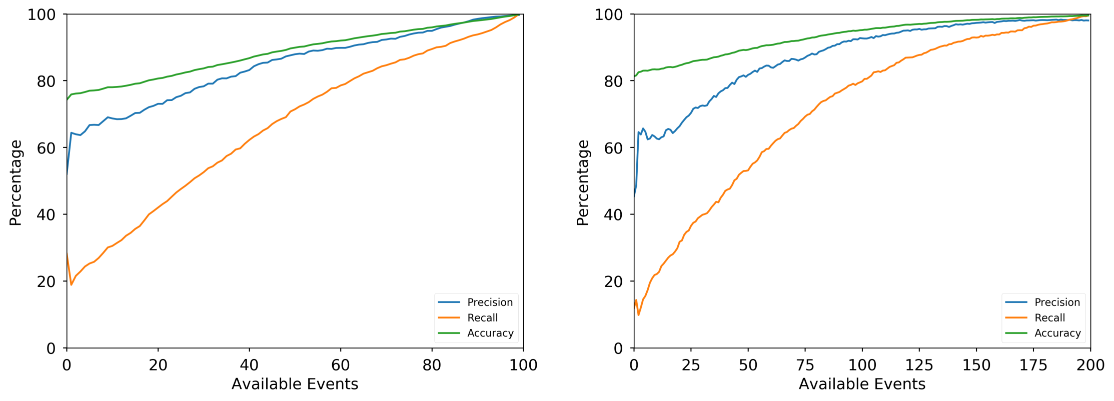
TN: The pattern was correctly predicted not to match.

FP: The pattern was falsely predicted to match.

FN: The pattern was falsely predicted not to match.

The recall is very low when only a few events are known, the reason being that a pattern is matched only in an average of 24% of windows for both window-sizes. With the increasing number of events, the recall also increases. If most events are known, the recall is reliable. With accuracy and precision, the results are better. If the model has predicted that a pattern matches, the prediction is mostly correct. With very few events available the model is not able to predict all patterns. As a consequence, there are many false negatives which in turn affects the recall value. But the model is always better in terms of false positives as can be seen by the precision values in the figure.

It doesn't make much sense to reorder very early, because the predictions (especially recall) are inaccurate, and there are often not enough events of the private patterns, since at least two are necessary for each one to be able to conceal them. In addition, later events that unintentionally complete a pattern can lead to further failures of the speculative reordering strategy. However, with medium and large windows it should be sufficient to start the reordering late when the predictions are stable. With small window-sizes, the reordering is very fast, so that no high latency should occur. When all events are known, Recall, Precision, and Accuracy tend towards 1.



(a) windowsize 100 **(b)** windowsize 200

Figure 6.2: Precision, Recall, and Accuracy for one pattern.

Figure 6.3 confirms that the predictions are reliable. The dotted orange line shows the accuracy of the binary prediction for a single pattern, while the blue line shows the average confidence after each step. The confidence is the matching probability if the output of the prediction model is greater than 0.5 and $1 - \mathit{matching\ probability}$ if it is smaller. It is the probability that the binary prediction is right. Assuming the confidence scores reflect the correct probabilities, these points must be consistent with the actual accuracy, which is the case here. The filled area represents the area that is at most one standard deviation away from the confidence line. This means that for 68% of the predictions the confidence is within this range.

What cannot be read from the diagram, but was successfully verified is that the confidence is always 1 if a pattern has already completed in the previous events.

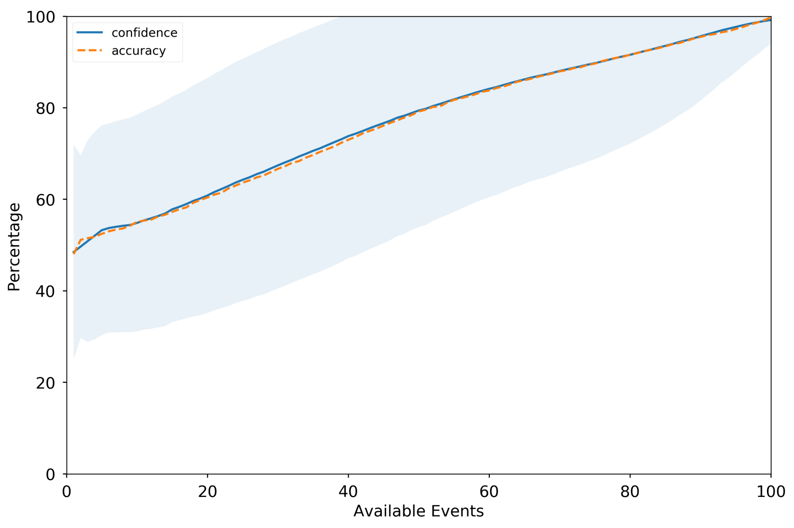


Figure 6.3: Accuracy and confidence for one pattern

6.3 Evaluation of the Speculative Reordering Strategy

This section is about the results of the simulations. The focus lays on the comparison between the speculative and the non-speculative approaches. For the most tests, we used the version of the speculative strategy that allows multiple reorderings. Only for the evaluation of the threshold, additionally the impact of the modified version, that only allows one reordering is shown. For the evaluations, we assume that windows in which the reordering subroutine fails are not considered. For this purpose, it is always specified separately how high the failure rate is. For both approaches, a negative utility is set to 0, which is realized by not forwarding the window.

Table 6.3: Summary of the results

	speculative	non-speculative
Error probability	5.6%	6.4%
Utility	86.1%	85.1%
FP probability of one pattern	3.6%	3.95%
Computation time	4.40s	3.21s
Complete elimination of latency	83.2%	0%
Average latency per window	1.01s	3.17s
Beginning of last reordering	78.4%	100%
Number of reorderings	1.81	1

Table 6.3 shows the results of the experiment, where all parameters were set to their default values, described in Section 6.1. The utility is measured relative to the maximum possible utility, which is the sum of all public patterns that match. The speculative version starts the final reordering, which results in a positive utility on average after 78.4% of the window is known.

On average 1.81 reorderings per window are necessary until it is privacy-preserved. In around 71.2% of the windows, only 1 reordering is needed, but there are some outliers where 5 or more reorderings are required, which has a significant impact on the average. With the non-speculative variant, the reordering is always executed one time after the window is complete.

Error Rate

The speculative strategy fails in 5.6% of the windows, which has the consequence that private patterns are not concealed. With the non-speculative strategy, this happens with 6.4% slightly more frequently. One reason, why the non-speculative strategy fails more often is that it always has to reorder the full window, while the speculative version mostly only a subset of it. Moreover, the patterns are usually longer in the non-speculative approach, since they can not be reduced. As a consequence, a single reordering with the speculative strategy is faster resulting in fewer timeouts and out-of-memory-exceptions.

Utility

The speculative strategy achieves with an average of 86.1% a slightly higher Utility. On the one hand, this variant has the advantage, that the graph-based reordering subroutine is executed with a reduced event stream and reduced patterns. Moreover, this approach has less false positives, which is also a consequence of the reduced event stream and patterns. On the other hand, the final reordering relies on speculative data, which might be partly incorrect and there are fewer possibilities for the event pairs that can be reordered since the future events are all untouched. However, the downsides have noticeably a smaller impact than the advantages in terms of utility.

Latency

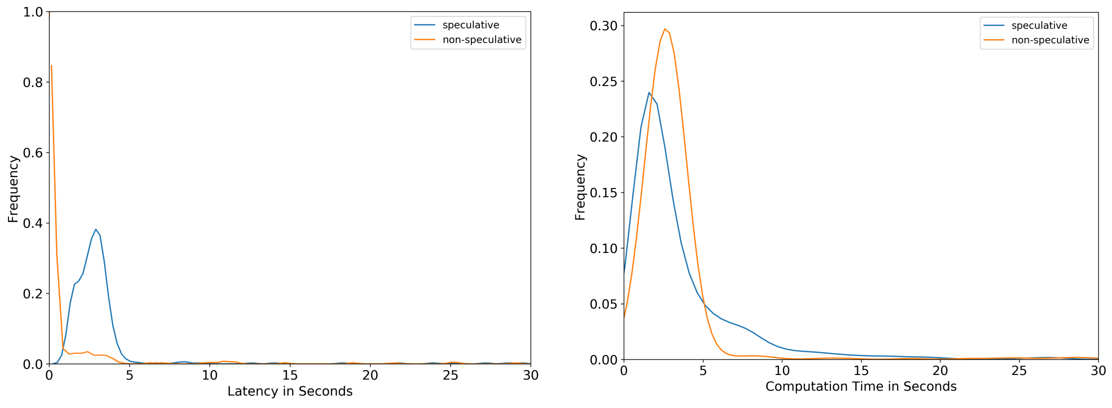
Latency is the additional waiting time required after the last event of the window to forward the privacy preserved event stream to the CEP engine. The speculative reordering strategy completely eliminates the latency in 83.2% of the windows. For the non-speculative version, it is obviously 0%, because it always has to wait with the execution of the algorithm until the window is complete.

Figure 6.4a shows the distribution of the latency. The speculative strategy has no latency at all in most cases, while the non-speculative one has to wait 3.17 seconds on average. Another difference is that there are cases where the reordering started before the last event but is not yet finished when the last event arrives. This also has a positive effect compared to the non-speculative version.

What is difficult to see in the figure is that there are few outliers in both speculative and non-speculative strategy where the latency is large, since the reordering subroutine takes a very long time. As a consequence, the average latency of the speculative approach of 1.01 seconds is only three times faster. If all windows where the latency is more than 10 seconds are ignored, 97% of the windows are remaining that have an average latency of only 0.28 seconds.

Computation Time

The distribution of the computation time is shown by Figure 6.4b. Despite the overhead due to predictions, the computation time is in most cases still lower with the speculative strategy. However, the average computation time is significantly lower with the non-speculative approach. Responsible for this contradiction are the cases where multiple reorderings are necessary. The computation time for the non-speculative approach is only the time needed for the reordering. For the speculative reordering strategy, it consists almost entirely of the time to execute the predictions, calculate the utility, and performing the reorderings. The time for utility and predictions is so low that it can be almost neglected. However, the possibility that multiple reorderings are necessary is a disadvantage in terms of computing time.



(a) Latency

(b) Computation time

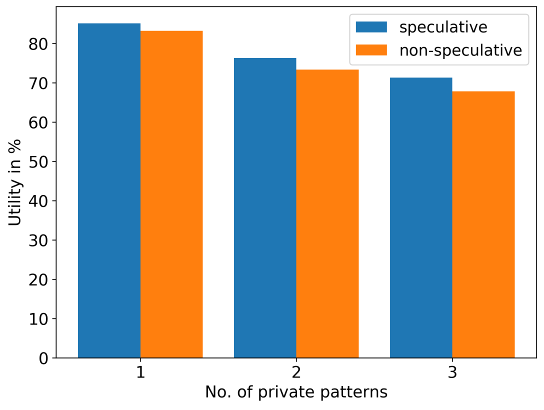
Figure 6.4: Latency and computation time

6.3.1 No. of private and public Patterns

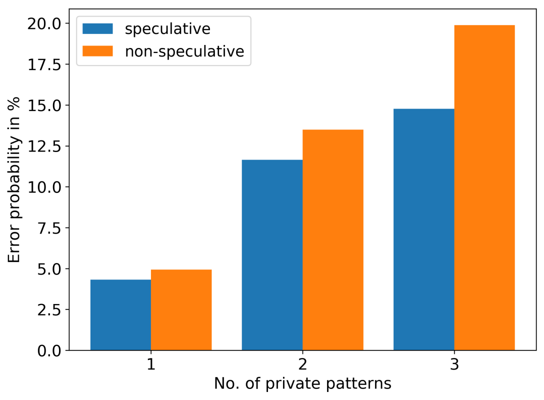
The number of private patterns significantly influences the results, while the number of public patterns does not have much impact. Only the patterns that match in the window are considered for counting the number of private and public patterns.

No. of private Patterns

Figure 6.5a represents the average utility, based on increase in the number of private patterns. Also with this evaluation, the utility is a little bit higher with the speculative approach, regardless of the number of private patterns. With the increase in private patterns, the utility decreases slightly. This can be explained by the fact that in order to conceal several private patterns, more event pairs have to be exchanged, which destroys public patterns and leads to more false positives and negatives.



(a) Utility



(b) Error probalility

Figure 6.5: Impact of number of private patterns on utility and error probability

In Figure 6.5b it can be seen how the error rate increases dramatically with the increase of private matches. While with one private pattern the speculative reordering fails in less than 4%, with 3 patterns the failure rate is nearly 15%. With the non-speculative strategy, it is even more evident - the error rate grows from 4.8% with one private pattern to almost 20% with 3. Interestingly, the majority of errors with 3 private patterns are caused by an out-of-memory-exception, while with one private pattern it is mostly because of a timeout during the shifting of the timestamps in the *reorder obfuscation phase*.

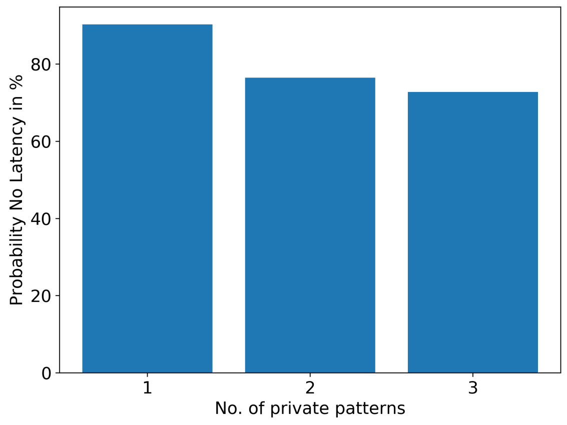
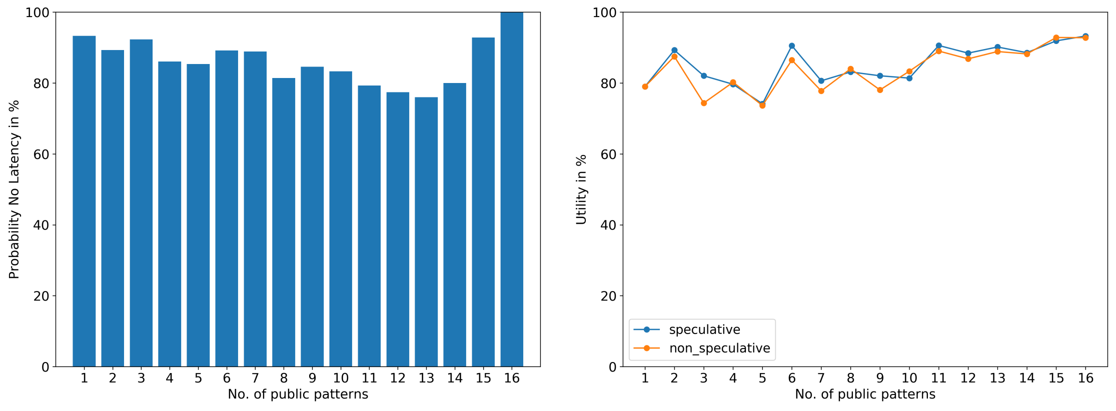


Figure 6.6: Probability for complete elimination of latency due to private patterns

The likelihood of complete elimination of latency decreases with an increase in private patterns, as visualized by Figure 6.6. The reason is that the average time for a single reordering is longer with more private patterns. This has a negative effect on latency and computation time.

No. of public Patterns

Even with a higher number of public patterns, latency is eliminated in most cases, as can be seen in Figure 6.5b. Despite the fluctuations, no clear trend can be identified with increase in number of public patterns. The fluctuations in the height of the bars are due to the characteristic of the dataset and patterns.



(a) Latency **(b)** Utility

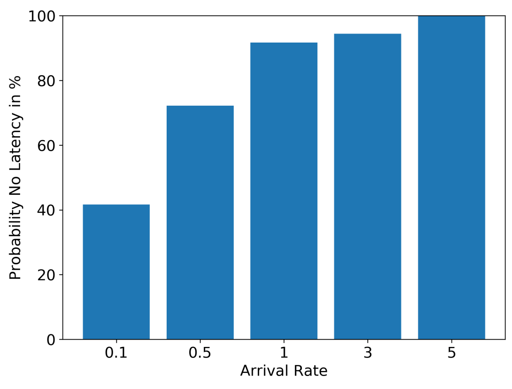
Figure 6.7: Impact on latency and utility due to number of public patterns.

The utility, illustrated in Figure 6.7b increases slightly with the increase in the number of public patterns. It is noteworthy that the utility is measured in terms of maximum possible utility. The main reason is that the number of patterns that do not match decreases with an increase in matches. Therefore, many matches of private patterns will result in less false positives. Another reason is that there are two public patterns that match in lots of windows but often cannot be preserved. This influence is reduced with increase in the number of matched public patterns.

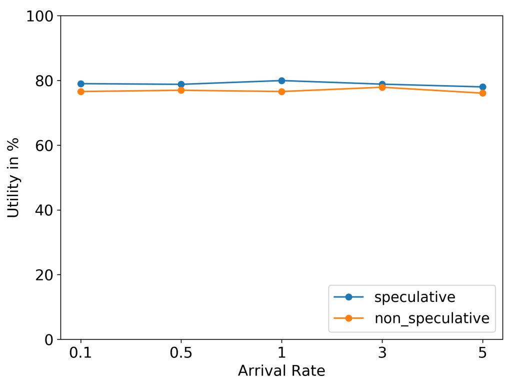
6.3.2 Arrival Rate

Arrival rate defines the rate at which two consecutive events arrive. Low arrival rate has a negative effect on the latency, as Figure 6.8a indicates. If the arrival rate is one second or higher, the latency gets eliminated for nearly all windows. With an arrival rate of 0.5 seconds, this is the case with approximately 70% of the windows.

The explanation is that the time, that the reordering subroutine needs and the time when the threshold, that defines when it is useful to start the reordering is big enough are independent of the arrival rate, but the window completes faster with fast arriving events. A low arrival rate also decreases the number of reorderings, since reorderings cannot be performed in parallel and it must always be awaited until the previous reordering has been completed to start a new one in case of multiple reorderings.



(a) Latency



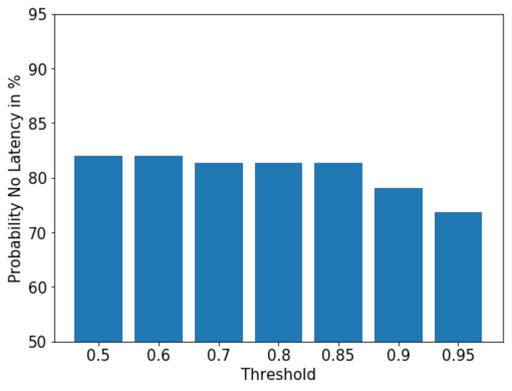
(b) Utility

Figure 6.8: Impact of arrival rate on latency and utility

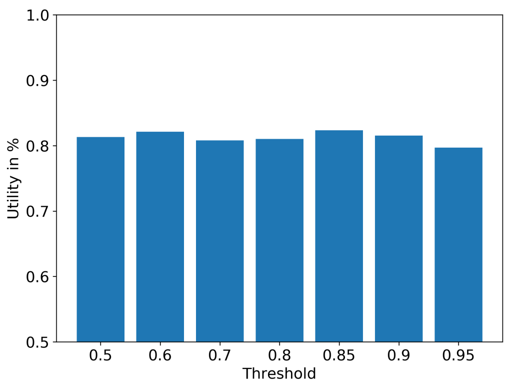
The arrival rate has no noteworthy influence on the utility that is shown in Figure 6.8b. Obvisouly the arrival rate has no impact on the non-speculative reordering strategy.

6.3.3 Threshold

The threshold that is used in Algorithm 5.1 defines when to start reordering at the earliest. A high threshold represents that the reorderings are performed later, and this causes the time taken for reordering to exceed after the end of the window and hence increase the latency. This is also illustrated by Figure 6.9a. However, under the tested conditions, the results show that up to 0.85 the threshold has no visible influence on the latency. This implies that the remaining time in the window is sufficient to complete the reordering.



(a) Latency



(b) Utility

Figure 6.9: Impact of threshold on latency and utility

Figure 6.9b shows the utility for different thresholds in terms of maximum possible utility. The threshold has no significant influence on this result. It is only to discover that the biggest threshold of 0.95 lowers the utility a bit.

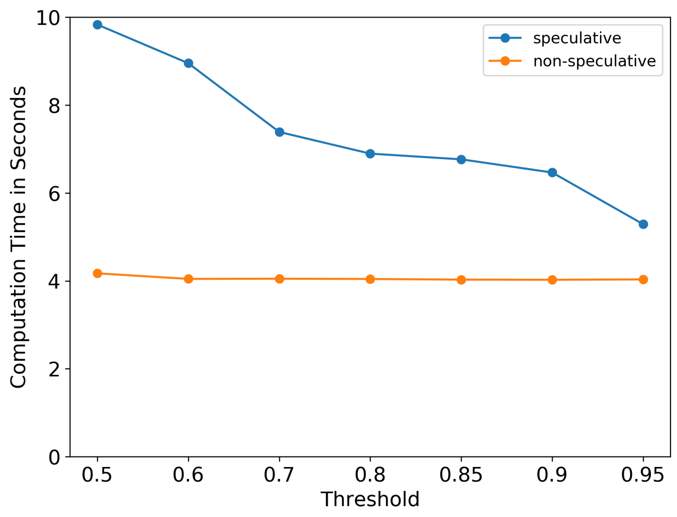


Figure 6.10: Impact of threshold on computation time

The impact of the threshold can be seen most strongly in the computation time, as illustrated by Figure 6.10. The higher the threshold, the lower the computation time. Even with a very high threshold, the computation time is still higher than with the non-speculative approach. Considering that a low threshold has no significant advantages, but massively increases the number of reorderings and thus the computation time, it is important to adjust this value accordingly to the use case to efficiently reduce latency. The threshold for the non-speculative reordering cannot be evaluated, as there is no threshold parameter.

6.3.4 Only One Reordering

The advantage with the modified speculative reordering algorithm that reorders only one time is, that it consumes much less computation time, but has the downside that it is not able to preserve QoS in all windows. The low computation time is confirmed by Figure 6.11. For this figure, only windows were considered which did not lead to a timeout or other error in all three tested methods. The explanation for the computation-friendliness of the modified speculative strategy is again that much time is saved by only reordering a subset of the window. Moreover, computation time is saved by only reordering once.

On average the last reordering with a threshold of 0.85 takes places after 73.8% of the window is available, that is earlier than with the normal speculative reordering strategy. There is also an improvement in latency, which with 0.52s is approximately two times less. Furthermore, 89.4% of the windows have no latency at all, compared to 83.2% with the standard speculative algorithm. The better latency, however, is explained by the fact that after the first reordering there is no longer any attempt to obtain the QoS. With a larger threshold, the computation time increases with this strategy. If multiple reorderings are allowed, it is reversed, so that a larger threshold leads to lower computation time.

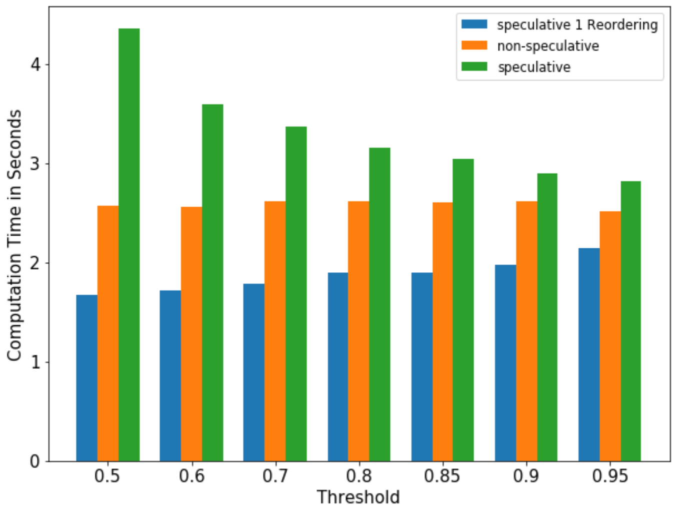
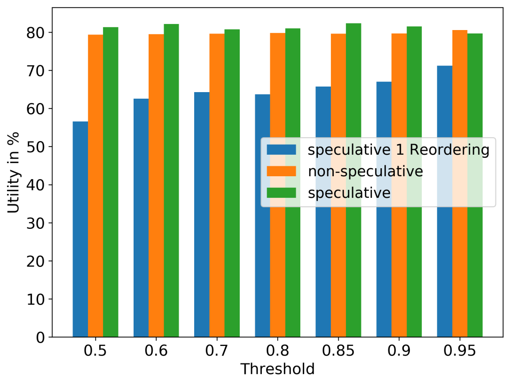
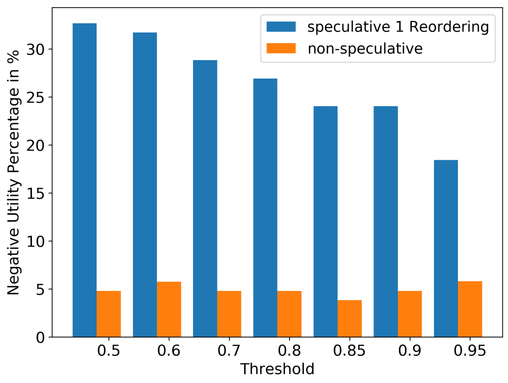


Figure 6.11: Comparison of computation time

Figure 6.12a shows the utility for different thresholds. As expected, with the increasing of threshold, the utility increases as well. Even with the highest threshold of 0.95, which represents that in less than 5% of the prediction at least one private pattern was predicted incorrectly, the utility with 70% is seemingly under the utility of the non-speculative algorithm. Another reason for that in addition to the possibility of inaccurate predictions is that events, that were unknown at the time of the reordering sometimes unintentionally complete a private pattern. The lower the threshold, the more frequently these two problems occur. The probability for negative utility is significantly higher for the modified speculative strategy than for the non-speculative strategy, as Figure 6.12b indicates. Note that this figure also considers cases, where the reordering routine failed because of an error in the reorder subroutine.



(a) Utility



(b) Negative utility

Figure 6.12: Comparison of utility

6.3.5 Window-Size

The last parameter to be evaluated is the window-size. For window-sizes of 50,100 and 200, separate patterns were created, which on average are of the same length and match in the same number of windows. It is important that the speculative strategy also works with medium and large size windows since the non-speculative reordering strategy is already fast with small ones. It is noticeable that with a window-size of 200, the reordering subroutine often fails with both strategies. With a window-size of 100, this only happens in 5.4% of the cases and with 50 in almost none. With all three sizes, it is a bit more common for the non-speculative variant. These results are illustrated by Figure 6.13.

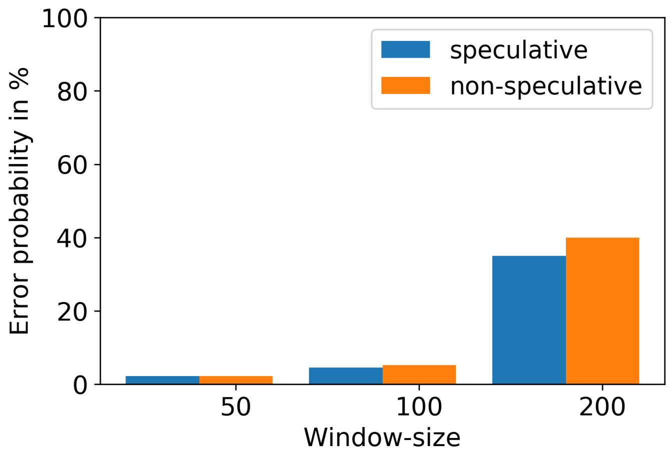
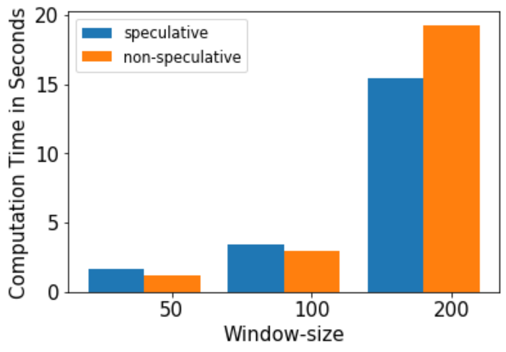


Figure 6.13: Impact of window-size on error probability

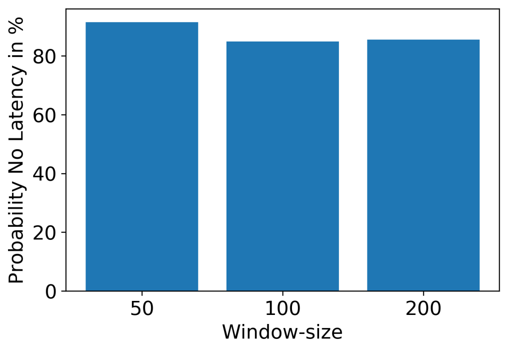
The average computation time for the window-size is shown by Figure 6.14a. Remember that for this chart, all cases with errors have been filtered out. The computation time for the size 200 is the biggest by far. While the non-speculative strategy requires less computing than the other two sizes, the opposite is true for the largest tested window-size, where the speculative approach is more efficient. A closer look at the experiment reveals that the reordering procedure needs an unusually long time with the largest window-size. This effect is a bit less with the speculative version because it reorders on average with 72% of the events.

Figure 6.14b shows that with all three window-sizes the speculative reordering strategy is able to completely eliminate the latency for over 80% of the windows. This is most effective with the smallest window-size of 50 events, whereby on the one hand less time for reordering is available, but on the other hand, a single reordering is performed very fast, which is an essential benefit. The other two window-sizes are about the same, although they differ in that with 100 events per window less computation time is needed, while with 200 events there is more time to complete the reordering before the last event of the window arrives.

Despite the fact that all three window-sizes are tested on slightly different conditions, no noticeable difference can be seen in the utility, that is slightly over 80% with all window-sizes.



(a) Computation time



(b) Latency

Figure 6.14: Impact of window-size on computation time and latency

6.3.6 Limitations

Although the evaluation shows that the speculative reordering strategy is effective to preserve privacy, it also has some limitations, that we discuss briefly in this section.

Failing of the Reordering Subroutine

With the window-size and the complexity of the patterns, the probability of this problem increases dramatically. Especially during the test with a window-size of 200 events, this became visible. With the speculative algorithm, this problem occurs less frequently than with the non-speculative one, but it still has an enormous impact on the overall result. A timeout exception also worsens the latency and computing time, which is increased by the fact that a new reordering attempt is started afterward.

In order to make the speculative reordering strategy work in practice for more complex tasks, this problem must be solved. A simple solution would be to check in advance whether a reordering is possible or not. Since the failure is often related to the number of event pairs that can be exchanged, this should be easy to implement but does not solve the problem that it significantly reduces the expected utility. The better and more elaborate solution would be to make the reordering subroutine more efficient so that it can handle more complex event streams. The current implementation always delivers the best possible solution. This could be softened so that the algorithm offers under challenging cases any solution instead of none at all.

Dataset

What makes using the speculative variant more complicated to use in practice than the non-speculative one is that for each use case a real dataset consisting of a long event stream is needed. The prediction model must first be trained with this dataset before it can be used. The training can take several hours if a lot of test data is available. If on the contrary, it is trained with a small event stream, it is proportionally faster but has a negative effect on the accuracy.

As a consequence, the effort involved in training the model made it complicated to perform the individual evaluations, that is why only three window-sizes and up three private patterns were tested. This can be eliminated with an online training system as a future work instead of an offline system model.

Programming Languages

Large parts of the code are programmed with Python, which is slow compared to many other programming languages. Also, the fact that the reordering subroutine was implemented as a server application, fulfills the purpose as a prototype, but the whole speculative reordering strategy could have been even faster. Note that the prediction model was also programmed with python, but the used libraries consist mostly of C and C++ code, which makes it efficient.

7 Conclusion

The number of IoT devices and their corresponding applications have increased enormously in recent years. These devices collect data, which is then processed by CEP applications, causing the problem that private information is revealed in the form of patterns.

In this thesis, we proposed a latency-optimized speculative reordering strategy, that conceals these private patterns in windows of events, before the data leaves the trusted environment. To do so, we extended the reordering approach by Palanisamy et al. [PDTR18], which has to wait until all events of the current window are known. Our approach can process an incomplete window so that all subsequent events can simply be appended to that intermediate result. The idea is that this eliminates or reduces latency. To effectively conceal the private patterns of a partial window, the information is needed for each pattern whether it will match in the complete window or not. For this purpose, we developed a Deep Neural Network, which consumes events of an event stream and outputs the matching probabilities for all patterns. We showed that with a sufficient number of available events, this prediction model can make accurate predictions, even for large window-sizes. These values are used to calculate a confidence score that decides when it is an optimal time to start the reordering. If the forecasts do not come true later, or for any other reason the privacy cannot be preserved, a further reordering is necessary.

With the event stream of a real-world dataset, we compared this approach with the non-speculative strategy. In fact, we have shown that the latency, which is time after completion of the window that is additionally required for processing is completely eliminated in 83.2% of the windows. In the remaining cases, latency is often reduced compared to the non-speculative approach. Further advantages are that the QoS is slightly higher and the number of false positives is a little lower, which has to do with the fact that only a subset of the window is used for reordering and the remaining events remain untouched. This also makes a single reordering faster, but this is not the case for the entire algorithm since sometimes several reorderings are necessary for the speculative variant. The overhead caused by the predictions and the pre-processing is so low that it can be almost neglected. A significant impact has the confidence threshold, which decides when to start the reordering. If this is too low, much processor time will be used unnecessarily, and if it is too high, it has negative effects on the latency. Thus an optimal value should be chosen for the respective application. Furthermore, our approach slightly reduces the risk of large windows leading to a timeout where private patterns could be revealed, compared to the non-speculative strategy.

We also presented a modified version of the strategy that only reordered once, resulting in visibly less computation time being required than with the non-speculative variant, but reduces QoS. This is especially suitable for IoT devices that have a small processor or battery.

7.1 Future Work

The evaluations indicated that our speculative reordering strategy has great potential to ensure privacy in IoT applications with little latency. A logical further step to demonstrate the practical suitability is to test it in a real use-case, i.e., as part of a complete CEP system that receives data in real time and operates a real service. By reimplementing the entire algorithm, including the reordering subroutine with a faster programming language such as C++, performance could be significantly improved, which would, in particular, enhance use in common IoT devices. Therefore, it would be essential to realize one of the presented solutions against timeouts and out-of-memory-exceptions.

It would make sense to extend the PAC component that includes the reordering algorithm to a universal privacy protection component that can handle further CEP operators than sequence, e.g., conjunction. Access control should be not only available at pattern-level, but also for single events and attributes. Since a single event cannot be concealed by reordering, a suppression approach could be used for that. Moreover, it must be considered, that events are often associated with a value, e.g., an event generated from a temperature sensor contains a temperature value, that could also reveal private information.

The used reordering algorithm makes the time stamps of the modified event stream look realistic, but in contrast to the paper of the non-speculative reordering, we did not consider an adversary that tries to recognize that a private pattern was removed. Especially a mechanism against *Causal order constraint attacks* [PDTR18] would be useful.

Bibliography

[AAA+16] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. “Deep speech 2: End-to-end speech recognition in english and mandarin”. In: *International Conference on Machine Learning*. 2016, pp. 173–182 (cit. on p. 19).

[AAP17] E. Alevizos, A. Artikis, G. Paliouras. “Event forecasting with pattern markov chains”. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM. 2017, pp. 146–157 (cit. on p. 14).

[BSF94] Y. Bengio, P. Simard, P. Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166 (cit. on p. 19).

[CBS+16] E. Choi, M. T. Bahadori, A. Schuetz, W. F. Stewart, J. Sun. “Doctor ai: Predicting clinical events via recurrent neural networks”. In: *Machine Learning for Healthcare Conference*. 2016, pp. 301–318 (cit. on p. 13).

[CGCB14] J. Chung, C. Gulcehre, K. Cho, Y. Bengio. “Empirical evaluation of gated recurrent neural networks on sequence modeling”. In: *arXiv preprint arXiv:1412.3555* (2014) (cit. on p. 14).

[CM12] G. Cugola, A. Margara. “Processing flows of information: From data stream to complex event processing”. In: *ACM Computing Surveys (CSUR)* 44.3 (2012), p. 15 (cit. on pp. 15, 16).

[Edd98] S. R. Eddy. “Profile hidden Markov models.” In: *Bioinformatics (Oxford, England)* 14.9 (1998), pp. 755–763 (cit. on p. 14).

[FLJ+17] S. Faye, N. Louveton, S. Jafarnejad, R. Kryvchenko, T. Engel. “An Open Dataset for Human Activity Analysis using Smart Devices”. In: (2017) (cit. on p. 39).

[GSC99] F. A. Gers, J. Schmidhuber, F. Cummins. “Learning to forget: Continual prediction with LSTM”. In: (1999) (cit. on pp. 14, 19, 23).

[HBWN11] Y. He, S. Barman, D. Wang, J. F. Naughton. “On the complexity of privacy-preserving complex event processing”. In: *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2011, pp. 165–174 (cit. on p. 13).

[Jia16] H. Jia. “Investigation into the effectiveness of long short term memory networks for stock price prediction”. In: *arXiv preprint arXiv:1603.07893* (2016) (cit. on p. 13).

[Kar15] A. Karpathy. “The unreasonable effectiveness of recurrent neural networks”. In: *Andrej Karpathy blog* (2015) (cit. on pp. 13, 21).

[KB14] D. P. Kingma, J. Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 25).

[KSH12] A. Krizhevsky, I. Sutskever, G. E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105 (cit. on p. 19).

[Lee18] M. D. Lee Rainie. *5. Scenario: Auto insurance discounts and monitoring*. 2016 (accessed August 16, 2018). URL: http://www.pewinternet.org/2016/01/14/scenario-auto-insurance-discounts-and-monitoring/ (cit. on p. 11).

[LOW08] J. Li, B. C. Ooi, W. Wang. “Anonymizing streaming data for privacy protection”. In: (2008) (cit. on p. 13).

[LXLZ15] S. Lai, L. Xu, K. Liu, J. Zhao. “Recurrent Convolutional Neural Networks for Text Classification.” In: *AAAI*. Vol. 333. 2015, pp. 2267–2273 (cit. on p. 19).

[Meu17] R. van der Meulen. *Gartner Says 8.4 Billion Connected “Things” Will Be in Use in 2017, Up 31 Percent From 2016, Gartner*. 2017 (cit. on p. 11).

[Nie16] M. A. Nielsen. “Neural networks and deep learning (2015)”. In: *Also available at: http://neuralnetworksanddeeplearning. com* (2016) (cit. on pp. 21, 23).

[Ola15] C. Olah. *Understanding-LSTMs*. 2015 (cit. on pp. 19–21).

[PDTR18] S. M. Palanisamy, F. Dürr, M. A. Tariq, K. Rothermel. “Preserving Privacy and Quality of Service in Complex Event Processing through Event Reordering”. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. ACM. 2018, pp. 40–51 (cit. on pp. 11, 13, 15–18, 29, 33, 34, 53, 54).

[PGC+17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer. “Automatic differentiation in pytorch”. In: (2017) (cit. on p. 23).

[Sch15] J. Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117 (cit. on p. 19).

[SHK+14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958 (cit. on p. 25).

[SLH+14] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, M. Riedmiller. “Deterministic policy gradient algorithms”. In: *ICML*. 2014 (cit. on p. 14).

[WDR06] E. Wu, Y. Diao, S. Rizvi. “High-performance complex event processing over streams”. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, pp. 407–418 (cit. on p. 16).

[WHRN13] D. Wang, Y. He, E. Rundensteiner, J. F. Naughton. “Utility-maximizing event stream suppression”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 589–600 (cit. on p. 13).

[Wik18] Wikipedia. *Long short-term memory — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Long%20short-term%20memory&oldid=856243812. [Online; accessed 26-August-2018]. 2018 (cit. on p. 19).

Declaration