Master Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Intrusion Detection Using Machine Learning in Databases

Sebastian Schindler
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr. Alexandra Dmitrienko
First Reviewer

Prof. Dr.-Ing. Samuel Kounev
Second Reviewer

Christoph Sendner, M.Sc.
First Advisor

Lukas Iffländer, M.Sc.
Second Advisor

Submission
30. April 2021

www.uni-wuerzburg.de

Abstract

Ransomware is a known threat that had a severe impact on computer security in the past five years. This type of malware has caused financial losses of about $13 Billion in 2017 and 2018 combined. Ransomware makes a user's data unavailable to them, only granting access again when they pay a ransom. Traditionally, ransomware targeted the computer's filesystem. Database ransomware is a new variant of the same principle. Instead of targeting individual files, it logs into DBMSs remotely and destroys the data, leaving only a ransom message behind. In most cases, attackers do not create a backup copy of the data. In this case, the data cannot be restored by the attackers, even if the ransom is paid. In 2018, Jobst et al. presented DIMAQS, a MySQL plugin to mitigate these attacks by detecting malicious activity through a Petri net classifier.

Our work recognizes the main drawback of this approach: The Petri net cannot be easily adapted to new attack scenarios and has to be re-engineered manually. To solve this problem, we design a machine learning classifier to replace the original one. This approach yields a model that detects all attacks in our tests. Unfortunately, the model also produces a high number of false positives when trying to detect attacks before any harmful queries are issued. Overall, our approach achieves a 85.23% f1-score. The performance impact of the revised plugin is nonexistent for OLAP workloads and stays under 15% for OLTP tasks.

Zusammenfassung

Die Bedrohung durch Ransomware hat sich in den letzten fünf Jahren deutlich auf die globale IT-Sicherheit ausgewirkt. Zusammengenommen zerstörten solche Angriffe in den Jahren 2017/18 über 13 Milliarden Dollar. Das Prinzip hinter den Angriffen ist simpel: Eine Schadsoftware verhindert den Zugriff auf die Daten der Opfer, und gibt ihn erst gegen ein Lösegeld wieder frei. Die meisten dieser Angriffe nutzten das Dateisystem um Zugriff zu verhindern, jedoch existiert seit einiger Zeit eine Variante dieses Angriffes, der Datenbanksoftware zum Ziel hat. Dabei erlangen Angreifer administrative Zugriffsrechte auf ein Datenbanksystem, zerstören dort sämtliche Daten und hinterlassen eine Lösegeldforderung. In den meisten Fällen erstellen sie keine Sicherung der gelöschten Daten, sodass die Daten, selbst wenn das Lösegeld bezahlt wird, unwiederbringlich verloren sind. Jobst et al. präsentierten im Jahr 2018 DIMAQS, ein MySQL Plugin, das durch einen Petri Netz schädliche SQL-Anfragen erkennt und betroffene Daten vorher sichert.

Diese Arbeit hat zum Ziel, ein grundlegendes Problem des DIMAQS Plugins zu lösen: Das Petri Netz lässt sich nicht einfach an neue Angriffe anpassen. Um neuen Anforderungen gerecht zu werden, muss die Struktur des Netzes manuell neu entworfen werden. Wir lösen dieses Problem, indem wie das Petri Netz durch einen Klassifizierer ersetzen, der sich mittels maschinellem Lernen an neue Angriffe anpassen lässt. Das trainierte Modell verhindert in unseren Tests sämtliche Angriffe, erkennt jedoch einige Anfragen fälschlicherweise als Angriff. Insgesamt erzielt das Modell ein F1-Maß von 85.23%. Dabei sind keine Auswirkungen auf die Geschwindigkeit für OLAP Anwendungen zu erkennen. Bei OLTP arbeitet die Datenbank bis zu 15% langsamer.

Contents

1. Introduction **1**

2. Background **5**
2.1. Terminology & General Process 5
2.2. Types of Learning 6
2.3. Recurrent Neural Networks 7
2.4. The Adam Optimizer 9
2.5. Evaluating Machine Learning Models 10

3. Related Work **11**
3.1. Dynamic Identification of Malicious Query Sequences (DIMAQS) 11
3.2. Database Intrusion Detection 12
3.2.1. Data Mining Approaches 12
3.2.2. Machine Learning Approaches 13
3.3. Ransomware Detection 13
3.3.1. Analytic Approaches 14
3.3.2. Machine Learning Approaches 14
3.4. Summary 15

4. Attack Analysis **17**
4.1. Affected Parties 17
4.2. Impact 17
4.3. Threat Analysis 18
4.4. Threat Model 20
4.4.1. Areas of Reliance 20
4.4.2. Attack Sequence 20

5. Requirement Analysis **25**
5.1. Classifier 25
5.2. Data Preparation 26
5.3. Training Executable 26
5.3.1. Data Ingest 27
5.3.2. Sequence Generation 27
5.3.3. Training 27
5.3.4. Evaluation 28
5.4. MySQL Plugin 28

6. Design **29**
6.1. Plugin Architecture 30
6.1.1. Database Query Interface 31
6.1.2. Classifier Interface 31
6.1.3. Action Policy 31
6.1.4. Query Rewriter 32

6.1.5. Admin Handler . . . 32
6.1.6. Actions . . . 32
6.1.6.1. Admin Mode Action . . . 33
6.1.6.2. Rewrite Action . . . 33
6.1.6.3. Alert Action . . . 33
6.1.6.4. Backup Action . . . 33
6.2. RNN Classifier . . . 33
6.2.1. Feature Extraction . . . 34
6.2.2. Neural Network . . . 34
6.2.3. Orchestration . . . 34
6.3. Training Executable . . . 35
6.3.1. Data Ingest . . . 36
6.3.2. Feature Extraction . . . 37
6.3.3. Query Sequences . . . 37
6.3.3.1. Sequence Generation . . . 37
6.3.3.2. Streaming Query Sequence . . . 38
6.3.4. Training Loop . . . 38
6.3.5. Model Evaluation . . . 39

7. Implementation **41**
7.1. Plugin Architecture . . . 41
7.1.1. Controller . . . 43
7.1.1.1. Initialization . . . 43
7.1.1.2. Event Handling . . . 43
7.1.2. Database Query . . . 43
7.1.3. Query Rewriter . . . 43
7.1.4. Admin Handler . . . 44
7.1.5. Actions . . . 44
7.1.5.1. Admin Mode Action . . . 45
7.1.5.2. Rewrite Action . . . 45
7.1.5.3. Notification Action . . . 46
7.1.5.4. Backup Action . . . 46
7.1.6. Action Policy . . . 46
7.2. LSTM Classifier . . . 47
7.2.1. Feature Extraction . . . 47
7.2.2. Neural Network . . . 48
7.2.3. Neural Network Queue . . . 49
7.2.4. Orchestration . . . 49
7.3. Data Preparation . . . 52
7.3.1. Benign Query Traces . . . 52
7.3.2. Malicious Query Sequences . . . 55
7.4. Training Executable . . . 55
7.4.1. Data Ingest . . . 56
7.4.2. Feature Extraction . . . 56
7.4.3. Query Sequences . . . 56
7.4.3.1. Sequence Generation . . . 56
7.4.3.2. Streaming Query Sequence . . . 59
7.4.4. Training Loop . . . 60
7.4.5. Model Evaluation . . . 61
7.4.6. Tuning . . . 61

8. Evaluation 63
8.1. Testbed 63
8.2. Detection Accuracy 63
8.3. Performance 67
8.3.1. TPC-H 67
8.3.2. Sysbench 68
8.4. Summary 70

9. Conclusion and Future Work 71
9.1. Conclusion 71
9.2. Future Work 72

Acronyms 73

Bibliography 75

Appendix 81
A. Regular Expressions 81
B. Raw Results 82

1. Introduction

The last two decades have seen a dramatic rise in the amount of data we collect [1]. Every interaction with every online service generates a datapoint that is transmitted, stored and analyzed. With the increasing availability of data, we are now able to use it for business intelligence and prediction, providing smart recommendations, tailored advertisements, or predicting human behavior. All of this information serves to monetize the vast amount of data businesses can collect or generate. Today, there are whole industries focused on data analytics, but also data protection. In short: Data is more important and more valuable than ever.

While the aggregate of collected data becomes more and more valuable, single pieces of data become less significant. The value lies in the information that can be extracted from large datasets, rather than the data itself. Further degrading the value of individual datapoints is anonymization mandated by recent privacy laws. These trends have made attacks on data confidentiality and authenticity less economical and therefore less common. The most successful attacks today target data availability. Attackers create and spread ransomware, a type of malware that encrypts the victim's data (crypto ransomware) or locks them out of the system (locker ransomware). Access to the data is only granted again after the victim pays a ransom, usually in the form of cryptocurrency. These *ransomware attack*s have been in use for over 30 years [2], however, they have gained in popularity in recent years.

Perhaps the most prominent example of a crypto ransomware attack was the *WannaCry* malware, that affected large corporations like the *Deutsche Bahn* or *FedEx* [3]. The malware targeted the user's file-system, encrypted their data and supplied the decryption key in exchange for some amount of Bitcoin. By May 2017, *WannaCry* had affected more than 200.000 systems all over the world [4]. There are no accurate numbers of the malware's impact since then, but it is reported to be still spreading to unpatched systems [5].

The financial impact of these types of attacks is significant and still rising. The estimated losses grew from \$5 Billion in 2017 to about \$8 Billion in 2018 [6], representing a 60% year over year growth.

More recently though, attackers have shifted their attention towards Database Management Systems (DBMSs). In the enterprise software context, database systems are massively popular. They provide fast, managed and reliable storage with a well-defined interface, reducing complexity from the application itself. Most data today is stored in databases, making it a prime target for ransomware attacks. In 2016, an attack called

MongoDB Apocalypse affected over 28.000 databases [7]. The attack resurfaced in 2019, over two years after the initial attacks, and infected about 3.000 new databases [8]. Similar attack methods were later used on MySQL, ElasticSearch, Hadoop, CouchDB, and Cassandra systems [9].

Although these attacks fall into the ransomware category, they are very different from the traditional file-system-based attack schemes. The first difference is the lack of a malware executable. As database software is designed to be accessed remotely, there is no need to deploy an executable to a victim's device. The attack can consist of only few Structured Query Language (SQL) queries and can be distributed across multiple remote systems and user sessions. Attackers gain access by obtaining passwords either by brute-force or other methods. This also means that the data can't be encrypted and kept on the victim's system easily. The data has to be downloaded to the attacker's system instead. This leads to the other difference: Database ransomware's destructive nature [8]. The first attacks copied the victim's database to their computer, before destroying the original data and leaving a ransom message in its place. After realizing that the amount of data was too large to transfer and save, the attackers began to outright delete the data, without keeping a copy, but not without still leaving a ransom message. This has lead many companies to pay the ransom without getting their data restored.

The impact of database ransomware is potentially even higher than traditional ransomware. The market analytics firm IDC predicts that by 2025, almost half of the world's data will be stored in public cloud environments [1]. Most cloud providers use DBMSs as backend storage. Consequently, the number of new targets could be substantial. Past incidents have also shown that businesses are more likely to pay a higher ransom, as data loss directly translates to a loss of revenue [10]. Finally, because information is destroyed rather than encrypted, it can not be recovered by paying the ransom. Instead, victims have to restore from backups, losing any information that has not been saved.

Existing database security solutions fall short for detecting database ransomware, as they are mainly focused on analyzing single queries or detecting usage anomalies in specific user sessions. Not considering the global security state of a DBMS leaves these solutions unable to detect attacks that are distributed across multiple user sessions, physical origins or queries. Ransomware attacks represent such a scenario, as they are comprised of several actions:

- Query the Database for Structural Information
  First, the attacker needs to get the identifiers of tables or databases to delete.
- Delete Tables or Databases
  Deleting information is the critical part of the attack. It can happen before or after the attacker creates the ransom message.
- Create a Table for the Ransom Message
  To insert the ransom message, the attacker needs to create a database and/or table that can hold the message.
- Insert the Ransom Message
  The ransom message is likely to contain information about the ransom payment.

The Dynamic Identification of Malicious Query Sequences (DIMAQS) system [11] provides a MySQL plugin to mitigate ransomware attacks on database systems. To counter attacks across multiple user sessions or physical locations, DIMAQS analyzes every query that gets issued to a MySQL instance. Currently, it identifies attacks by classifying queries using a Petri net that models an attack pattern specific to ransomware. If the plugin detects an intrusion, queries are rewritten to create a backup instead of deleting tables and the system administrator is notified. With the DIMAQS plugin in place, attacks still incur an outage, but no data is lost.

One issue with this approach is its specificity to one attack pattern. Changing the Petri net to accommodate a new attack requires manually analyzing the attack and devising a new structure for the net. This approach is not scalable to universal use. Furthermore, there is no way to provide updates to users of the plugin, as the net is currently hard-coded into the system. Utilizing a Petri net that can detect a wider variety of attack patterns does not solve this problem, as it would likely introduce a high false positive detection rate.

To overcome these shortcomings, we propose replacing the Petri net with a machine learning approach. A Long Short-Term Memory (LSTM) based neural network can be used detect an attack in the sequence of recent queries. This approach eliminates the hard-coded attack signature and enables us to adapt to new attacks by adding them to our dataset, re-training the network and deploying a new model.

In particular, we make the following contributions in this thesis:

- Develop a model for detection of server-side ransomware attacks and evaluate it using datasets provided in [11]. Our model detected all attacks during testing, but produced some false positives. Overall, it achived an f1-score of 85.23%. The original plugin was perfectly tailored to the attack samples and produced a 100% f1-score.
- Integrate the developed model with MySQL DB. To simplify the effort, the existing plugin implementation from [11] will be re-used whenever possible, since it already monitors incoming sequences and analyzes them for attack detection.
- Evaluate the performance overhead caused by the plugin. The revised plugin does not affect the performance of Online Analytical Processing (OLAP) workloads, however, Online Transactional Processing (OLTP) workloads suffer an impact of about 15% in our testing. Compared to the original DIMAQS plugin, our version is slightly faster in OLAP workloads, however, in OLTP tasks, it is significantly slower.

Outline. The remainder of this thesis is organized as follows: In Chapter 2, we give an introduction to machine learning and LSTM networks. In Chapter 3 we put our work into context by summarizing the existing DIMAQS system and exploring other works related to machine learning and malware detection. Next, we re-iterate the attack scenario from [11], which serves as our basis for evaluation, in Chapter 4. The requirements for the proposed changes to the DIMAQS plugin are analyzed in Chapter 5. The design of our solution is presented in Chapter 6, before a detailed description of the implementation in Chapter 7. Chapter 8 evaluates the finished plugin, focusing on security and performance aspects and comparing it to the original DIMAQS implementation. We conclude the thesis in Chapter 9 by discussing the implications of our work, as well as possibilities for future research.

2. Background

This chapter will give a short introduction to machine learning and the specific type of neural network we plan to use. Machine learning is a paradigm-shift in data processing. Traditionally, programmers provided problems as input data, along with the algorithms to solve that problem. Machine learning approaches are different: Their input data is a large number of problems and solutions. The goal is to derive patterns from the data to learn a model that can solve similar problems. Machine learning is a term that describes a multitude of technologies. We will first go over the common terminology and learning process in Section 2.1, before we look at each of the general categories: supervised, unsupervised, semi-supervised and reinforcement learning in Section 2.2. Section 2.3 will explain the specific type of machine learning algorithm we plan to use in this project. In Section 2.4, we discuss the optimizer we use in our work, before we go over the metrics used to evaluate a model in Section 2.5.

2.1. Terminology & General Process

Edwards [12] identifies four most commonly used terms when talking about machine learning:

- **Dataset**
  This is the set of example problems and solutions from which the model is derived.
- **Feature**
  Features are important characteristics that describe the data in the dataset.
- **Model**
  The model is the result of the learning process. Using the model, problems that are similar to the ones in the dataset can be solved.
- **Neural Networks**
  Neural Networks are comprised of one or more connected layers of neurons. Each neuron has inputs and outputs. Inputs can be amplified or diminished using adjustable weights and biases. These parameters are optimized during the training phase. Each output is passed through an activation function, usually a sigmoid or *tanh* function, that normalizes the output.

Learning a model to solve future problems involves five high-level steps:

1. **Data Collection**
   First, we need to compile a dataset to train our model.

2. **Data Preparation**
   To prepare the data, we need to extract all the important features from it and format them appropriately.

3. **Training**
   During the training phase, the prepared data is entered into the machine learning algorithm to train the model. Comparing the known true label to the output from the machine learning algorithm yields an error. This error is used by an optimizer to refine the parameters of the machine learning algorithm to improve its accuracy. For neural networks, this process is called backpropagation, as the error is propagated backwards through the network, adjusting weights and biases.

4. **Evaluation**
   With the trained model, we can evaluate its performance. Important statistics are false and true positive and negative detection rates. From these, we can calculate common metrics like precision, recall and the f1-score.

5. **Tuning**
   The last step is to fine-tune the model to perform optimally.

This process should yield a functional model that can be used to solve problems like those in the dataset. If the model does not perform well, the machine learning method or the extracted features can be adjusted and the process repeated.

2.2. Types of Learning

According to Edwards [12], training a machine learning model can be done through several methods:

- **Supervised Learning**
  Supervised learning algorithms train using labeled data. The goal is to train a model that can predict the label for a new piece of data. Labels can be chosen from a finite set of categories (classification) or they can be a scalar (regression).

  As we can generate a lot of labeled data by combining benign query sets and attack sequences, this is our learning method of choice.

- **Unsupervised Learning**
  Unsupervised learning feeds a lot of unlabeled data into the algorithm. The method is used to find instances that are similar to each other (clustering). Using clustering, we can find related instances or anomalies. Unsupervised learning is more complex than supervised learning, but has the benefit of not requiring the effort of labeling data. Additionally, removing biases that stem from the labels of the input data can be beneficial.

- **Semi-Supervised Learning**
  One problem with supervised learning is collecting and labeling enough data. Semi-supervised learning reduces the need for labeled data, as the dataset can be a mix of labeled and unlabeled instances. This hybrid approach for supervision enables machine learning to be used on more problems, even if there is less labeled input data for them.

- **Deep Learning**
  Deep learning refers to neural network algorithms with a high number of layers. The

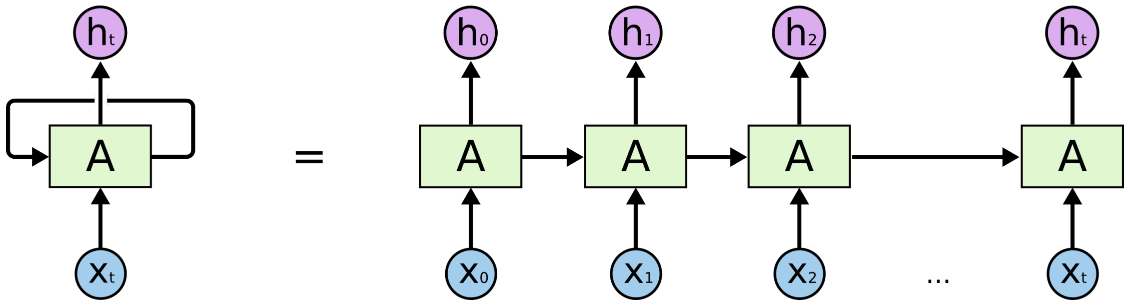


Figure 2.1.: The basic loop of an RNN [13]

key difference to more shallow networks is that deep learning networks do not need manual feature extraction. They train on and classify input data directly, reducing the amount of manual engineering. However, as these algorithms are more complex, they require more computational resources.

- **Reinforcement Learning**
  Reinforcement learning is modeled after the way humans learn. The input data for the algorithm is the state of the world around it. Every time the algorithm performs an action that influences that state, it gets positive or negative feedback. The algorithm strives to maximize the positive feedback. An example for this would be games. Every move will influence the score, the feedback can be derived from the score change. By repeatedly playing the game, a machine learning algorithm will learn the best strategies to maximize its score.

2.3. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are neural networks specifically designed to process sequences. Fundamentally, RNNs process a sequence piece by piece inside a loop. For every piece, the network generates an output. This output is the result of the network processing the sequence up to the current entry, but it also serves as a second input for the next iteration of the loop. By incorporating the last result, the network can retain information from previous iterations of the loop. The loop and its unrolled form is depicted in Figure 2.1.

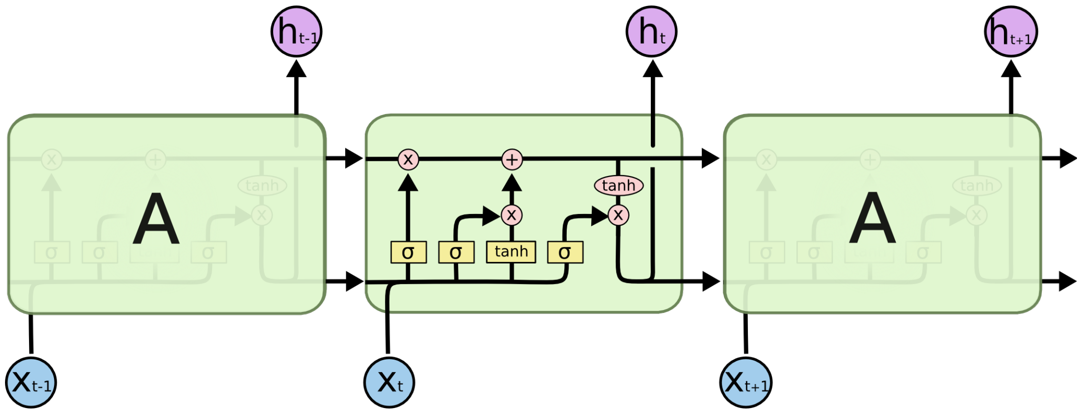


Figure 2.2.: The structure of an LSTM network [13]

The issue with RNNs is, that the information from loops further in the past get overridden by more recent iterations. LSTMs solve this by maintaining an additional cell state and deciding which data to keep during each iteration. Olah details the structure of LSTM networks in detail in [13]. Figure 2.2 shows a general overview. The flow of information from iteration to iteration through the cell state is visible in the top horizontal line. Additionally, the last output is also forwarded to the next iteration in the bottom line. We will now look at each component of the network and explain its purpose:

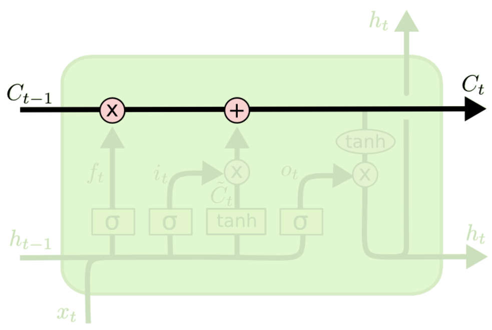


Figure 2.3.: Flow of information [13]

Flow of Information

This is the basic principle of LSTMs. The cell state C is not created new every time, but instead, the previous state C_{t-1} is modified during each iteration. In each iteration, there are two interactions with the cell state. The first decides which information to discard, and the second interaction updates the state with the information from the current iteration, yielding the new state C_t.

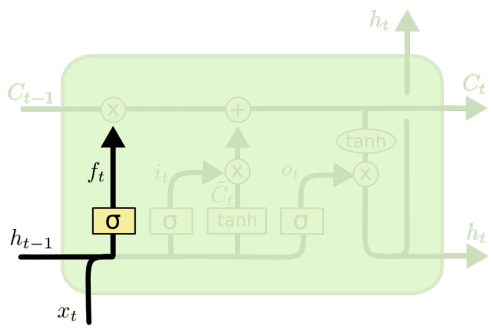


Figure 2.4.: Forget Gate [13]

Forget Gate

To decide which information to forget, the output of the previous iteration h_{t-1} and the current input x_t are concatenated and run through a sigmoid layer. The output f_t is calculated using the weights W_f and biases b_f:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

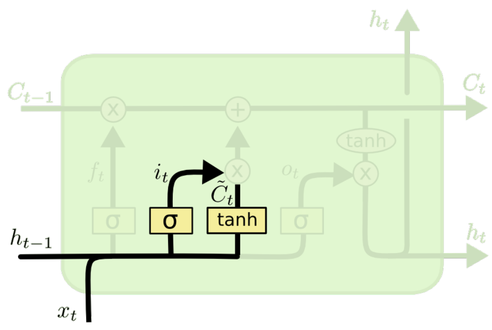


Figure 2.5.: Candidates and Filtering [13]

Creating Information Candidates and Filtering

Next, new information is selected to be stored in the state. A tanh layer creates information candidates $\tilde{C}_t$. The sigmoid layer is called the ”input gate layer”. It creates a vector i_t that filters the information candidates generated by the tanh layer.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-i}, x_t] + b_C)$$

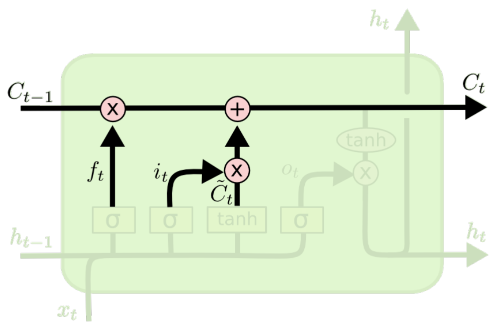


Figure 2.6.: Updating the State [13]

Updating the State

After figuring out what to forget and what to input into the state, it is time to update the values. The new state values are calculated as follows:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

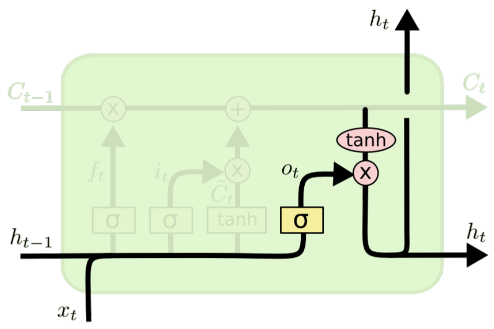


Figure 2.7.: Output [13]

Output
The output of this iteration is based on the updated cell state, but filtered. The sigmoid layer filters the state values, after they are normalized by passing them through tanh:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

This configuration allows information to be carried forward across many iterations. As the output of the LSTM is not a simple scalar that denotes the classification of the current query, the LSTM's output can be passed to a linear layer. The linear layer takes an input vector A and applies a linear function to it:

$$y = w \cdot A^T + b$$

The weight w and bias b are learned during the training phase. The result y will be a single scalar that can directly be interpreted as the label.

Some data can be classified into disjoint categories. Each of these categories can be represented by one component of the feature vector, meaning every component has a zero value, except the component corresponding to the instance's class. This encoding style is called one-hot-vector encoding. We plan to use this style to encode the type of queries issued to the DBMS. The structure of LSTMs is especially suited to to recount occurrences of one-hot-vector-encoded instances, making it easy for us to recognize queries that delete data, or queries for the database structure, that can signify the start of an attack.

2.4. The Adam Optimizer

The optimizer is responsible for refining the model during the training phase. It compares the known true label to the output of the machine learning algorithm and changes the algorithm's parameters to reduce the prediction error. Most optimizers can be influenced using a learning rate. This parameter describes the severity of the changes the optimizer makes to the machine learning algorithm.

The *Adam* optimizer was first described by Kingma et al. in [14]. Pseudocode for the optimizer is shown in Algorithm 1. The *Adam* optimizer has proven to reduce the necessary training time significantly, by starting with a high learning rate and reducing it over time. The default parameters for β_1 and β_2 are 0.9 and 0.999.

Algorithm 1 The Adam optimization algorithm from [14]

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
 1: $m_0 \leftarrow 0$ (Initialize 1st-moment vector)
 2: $v_0 \leftarrow 0$ (Initialize 2nd-moment vector)
 3: $t \leftarrow 0$ (Initialize timestep)
 4: while $\theta_0$ not converged do
 5:     $t \leftarrow t + 1$
 6:     $g_t \leftarrow \bigtriangledown_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
 7:     $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased 1st-moment estimate)
 8:     $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
 9:     $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
10:     $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
11:     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
    return $\theta_t$ (Resulting parameters)
```

2.5. Evaluating Machine Learning Models

When a model is trained, one needs to evaluate its accuracy. The f1-score is a simple measure of a binary classifier's accuracy and can be calculated as follows:

$$\begin{aligned} tp &= \text{number of samples correctly classified as positive} \\ p &= \text{total number of positive samples} \\ cp &= \text{total number of samples classified as positive} \end{aligned}$$

$$\begin{aligned} precision &= \frac{tp}{p} \\ recall &= \frac{tp}{cp} \\ f1 &= 2 \cdot \frac{precision \cdot recall}{precision + recall} \end{aligned}$$

When evaluating a multi-class classifier, these metrics can be calculated for every class. To get a score for the overall model, the individual class's f1-scores can then be averaged. If the classes do not occur with the same probabilities, one can also calculate the weighted average:

$$\begin{aligned} f1_{macro} &= \frac{f1_1 + f1_2 + f1_3 + ... + f1_n}{n} \\ f1_{weighted} &= \frac{cp_1 \cdot f1_1 + cp_2 \cdot f1_2 + cp_3 \cdot f1_3 + ... + cp_n \cdot f1_n}{\sum_{i=1}^{n} cp_i} \end{aligned}$$

3. Related Work

This section will provide an overview of the previous work done by our group, and examine other works surrounding ours. Section 3.1 will first detail the existing DIMAQS system, to illustrate our point of origin. As DIMAQS serves as the basis for our project, it is of some importance to understand how it works. Afterwards, we investigate other database intrusion detection approaches, both with and without using machine learning, in Section 3.2 and do the same for ransomware detection in the subsequent Section 3.3.

3.1. Dynamic Identification of Malicious Query Sequences (DIMAQS)

The intrusion detection system this work is based on is called DIMAQS and is introduced in [11]. It is important to this work for two reasons: It serves as a base MySQL plugin that we can adapt to our malware detection scheme, and it also provides the training data and attack signature for our work. The system uses the MySQL plugin architecture to monitor incoming queries and detect malicious actions. Upon detection, the affected data is backed up and the system administrator is notified. Currently, the detection mechanism employs a Petri net that models a common attack scenario:

1. **Query Structural Information**
   The attacker executes queries that return information about the databases or database tables present in the system.

2. **Delete Information**
   The attacker drops databases or tables that were returned in step 1.

3. **Leave Ransom Message**
   The attacker creates a new database or table that contains a message asking for payment to restore the data.

The order of steps 2 and 3. is not fixed, as an attacker could leave the ransom message before deleting anything or even in between deleting multiple databases. The structure of the Petri net can be seen in Figure 3.1. The net perfectly identifies the given attack scenario without any false positives or negatives.

The attacker's actions can happen across several sessions and from several sources. It is therefore not sufficient to monitor queries on a per-session, per-user or per-endpoint basis. Instead the Petri net represents the security state of the overall system.

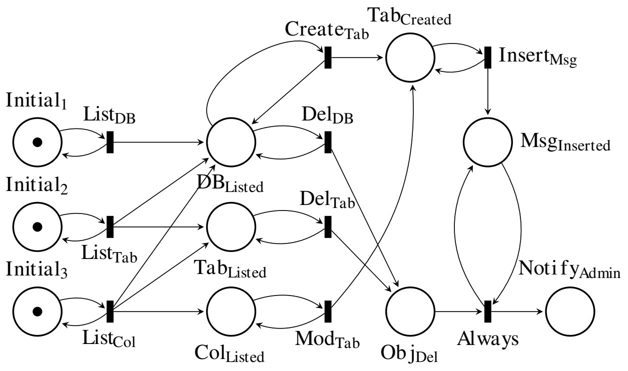


Figure 3.1.: Structure of the Petri net used by DIMAQS [11]

If an attack is detected, queries that drop databases or tables are rewritten, unbeknownst to the attacker, to rename tables rather than delete them. The databases or tables are prefixed with "dimaqs_". All queries for databases or tables that are present in the system are rewritten to exclude the renamed ones.

The issue with this implementation is that hard-coding the attack, even in this generalized way, makes the system hard to adapt to new attack strategies. We plan to use machine learning techniques to alleviate this issue. Our approach makes it easier to adapt to new attacks, as they can simply be included in the training dataset. To that end, we have designed an easily extensible way to store the training data. With a newly observed attack added to the dataset, a new model can be trained and deployed.

3.2. Database Intrusion Detection

This section will discuss some of the work regarding intrusion detection in database systems. The first Intrusion Detection System (IDS) was built in 1998 by Hofmeyr et al. [15]. Their system used UNIX system calls of benign processes to create signatures. These were compared to running processes to identify malicious applications.

The idea of finding unusual usage patterns to identify intruders is also used by modern IDSs. The practice is now called *anomaly detection.* Patcha and Park summarized and categorized existing database IDSs in 2007 [16]. According to their work, there are two types of approaches for analyzing these usage patterns: data mining and machine learning. We will look at examples for both categories in the following subsections.

3.2.1. Data Mining Approaches

The following works are based on data mining techniques to find signs of intrusions in usage patterns. Data mining is the algorithmic extraction of information from large amounts of data. Accordingly, these systems all need a trace of the database's normal operation to analyze off-line before operation.

Chung et al. propose DEMIDS [17], an IDS that identifies working itemsets for each user and application using a DBMS. It also defines a distance function between itemsets. Abnormal behavior is identified by an itemset with a high distance to its nearest neighbor. The DEMIDS system is based on user profiles and could be circumvented by distributing an attack across several users. Unfortunately, Chung et al. do not provide an evaluation of their system.

In [18, 19], Hu et al. propose and implement an IDS based on a Petri net classifier. They were able to identify a surrounding itemset for each write operation on a database. Their

Petri net models read, pre-write and post-write itemsets for write operations. The classifier could discover anomalous transactions if they did not conform to this model. Compared to our work, this system only considers the immediate vicinity of an SQL statement and might not be able to identify attacks that play out over longer time periods. While they did not provide any precision numbers in their evaluation, we can infer a maximum f1-score of 94.8% from their recall.

The DIWeDa system by Roichman and Gudes in [20] creates fingerprints of user sessions. These fingerprints are compared to a set of previously collected ones. If a session's fingerprint is too distant from known ones, it is likely compromised. The focus on sessions could enable an attacker to avoid detection by using multiple sessions. In their evaluation, they achieved an f1-score of 88.7%.

DIDAFIT [21] by Lup et al. builds a fingerprint of a specific application. This fingerprint consists of a set of previously seen SQL queries which were turned into regular expressions that match queries of the same structure. Any query that does not match the fingerprint is flagged as malicious. This approach is specific to the fingerprinted application and can't be directly deployed in a different environment. While they provided an evaluation of the systems performance, no detection accuracy numbers were present in the publication.

3.2.2. Machine Learning Approaches

Machine learning works by discovering and refining an algorithm with experience gained from training data. For more detail on this, refer to Chapter 2.

To identify intrusion in database systems with managed roles, Bertino et al. trained a Naïve Bayes classifier with the usage patterns of users from each role [22]. If a user's behavior does not fit their role, the account is likely compromised. Only considering individual users could leave attacks that involve several accounts undetected. In their evaluation, the system achieved an f1-score of 90.9%.

The focus of the work done by Valeur et al. in [23] was intrusions through SQL injection attacks from websites. They constructed several classifiers for individual queries, each one specific to a data type. The publication gave no detail on the algorithm used by the classifiers, but stated that they would likely need to be adapted to the environment where they are deployed. Classifying only individual queries makes the system exploitable by attacks comprised of several queries. The system achieved an f1-score of 90%. They later refined their work in [24] to also include the HTTP requests for classification. The improved classifier achieved a 99.4% f1-score.

3.3. Ransomware Detection

Ransomware that is focused on database systems is still a new development. Accordingly, we did not find any research focused specifically on DBMSs. However, the work put into detecting file-system based ransomware is extensive in comparison. To detect this type of malware, two approaches are wide-spread: static analysis of malware binaries and behavioral analysis of processes. The former does not apply to this work, as we have no binaries to analyze. Inspecting process behavior can also be done in several ways: Analytic approaches are created manually by analyzing the features by which ransomware can be identified and creating software to exploit these features, while machine learning approaches classify processes based on similarity to their training samples. We will look into both categories below.

3.3.1. Analytic Approaches

Analytic approaches to ransomware detection are manually crafted solutions that exploit behavioral characteristics unique to ransomware.

In [25], Scaife et al. created a system that monitors file system changes. Specifically, it looked for file type changes, changes to the amount of entropy in a file (encrypted data has high entropy) and the similarity of changed files to the previous version. Their evaluation produced a very high f1-score of 99.9%. However, the benign set of applications they used to test for false positives was flawed. There were not applications present that encrypt data. As the only false positive they recorded was produced by a compression software, which also produce files with high entropy, it is likely that including more software like this would lower their precision significantly.

UNVEIL is a system that creates artificial user environments and monitors them for changes [26]. Activity in those environments is likely not caused by the user, as they are not aware of them. This activity may be malicious. The UNVEIL software as evaluated to produce an f1-score of 97.7%. This principle would be applicable to DBMSs, by creating canary databases. Activity in them would indicate an intrusion. However, as a whole database can be dropped with a single SQL command, it is likely that a significant amount of data is already deleted before the malicious activity is detected.

Almashhadani et al. found that most crypto ransomware connects to remote command and control servers upon execution [27]. Their approach is to analyze network traffic to detect ransomware before it is able to execute. To prove the feasibility of this approach, they built an application that is able to detect the Locky ransomware before it was able to encrypt any files by analyzing the network traffic of the system.

3.3.2. Machine Learning Approaches

Machine learning solutions to ransomware detection look at some features of benign software and ransomware to learn the characteristics of malware behavior. The machine learning approaches we found either looked at a process' network activity or its Operating System (OS) Application Programming Interface (API) calls.

A similar traffic analysis system to Almashhadani et al.'s is used in [28]. They train a decision tree based classifier to find indications of ransomware activity. The classifier achieved an f1-score of 99.2%.

ShieldFS a file-system that can detect and mitigate ransomware attacks by monitoring file-system API calls [29, 30]. ShieldFS implements a copy-on-write (COW) mechanism on top of the windows-native filesystem. File changes are made reversible through that COW mechanism. Process's activities are classified by several random forest classifiers, if malicious activity is detected, affected files can be restored. The ability to restore affected databases is also a goal of this work, but a COW mechanism is hard to implement on databases, hence we create backups of affected tables. The ShieldFS system achieved a 98.8% f1-score.

Similarly to ShieldFS, Ransomwall is able to restore files affected by a ransomware attack [31]. Their system also uses a machine learning approach for classification, but they consider information from static analysis of the malware executable in addition to OS API calls. To further increase accuracy, they set monitored honey files to find malicious activity. RansomWall combines several strategies to achieve robust ransomware identification. Their Gradient Tree Boosting classifier achieved a 98.25% detection rate.

Maniath et al. built a system that can detect ransomware by classifying OS API call sequences [32]. This approach is conceptually very close to our work, only applied to a different system. To train an LSTM network, they logged the API calls of 157 malware samples and some benign software. As features, they used the API call type. The classifier produced an accuracy of 96.7% in their evaluation.

To further improve the accuracy, Agrawal et al. propose using an attention mechanism in [33]. Attention mechanisms can amplify the importance of single datapoints in a long sequence. This might prove especially useful for us, as we handle sequences that contain many benign queries and few important malicious ones. For them, the modified LSTM boosted the accuracy from 87% up to 93%.

3.4. Summary

The previous sections summarized a multitude of intrusion and ransomware detection methods. When looking at database intrusion detection, we see that all current methods either consider only single database queries or user sessions. DIMAQS tries to remedy that by monitoring queries across all sessions.

As this work is mainly concerned with ransomware classification, surveyed several ransomware detection projects and identified machine learning algorithms as a promising method for classifying sequences of API calls. Particularly the work done by Maniath et al. in [32] seems similar to our work. LSTMs seem especially suited to classify sequences of queries that are dominated by noise with only few important elements.

4. Attack Analysis

In this chapter, we analyze typical ransomware attacks this work aims to counter. First, we look at parties affected by ransomware attacks in Section 4.1. We then describe the impact of these attacks in Section 4.2 before analyzing typical attack scenarios in Section 4.3. Finally, we condense that knowledge into the threat model used by this work in Section 4.4.

4.1. Affected Parties

In this section, we investigate which parties could potentially be affected by ransomware and which ones are especially vulnerable. We also try to estimate how many targets for ransomware attacks there are today.

As discussed in Chapter 1, while ransomware attacks affect many private persons, there is a bigger incentive for attackers to target businesses. The loss of data translates directly to financial loss, making businesses more motivated to pay the comparatively small ransom to get their data back. There were some high-profile cases of large companies, like *Deutsche Bahn* [34] or *FedEx* [3] being hit by ransomware, as well as hospitals [35] and city councils [36]. Smaller entities, like public institutions or small businesses present an enticing target to ransomware attackers, as they rarely have the means to defend themselves or recover from a catastrophic data loss [37].

The list of potential targets for ransomware attacks is vast. In 2019, BinaryEdge [38], a data provider for threat intelligence and security reporting, conducted a search for database servers in Iran [39]. They were interested in servers, that were accessible without protection from the internet. 547 databases were vulnerable, with about 1.8TB of information exposed. The accessible databases account for roughly 4.9% of all Iranian database systems with remote access. If we extrapolate that number to the rest of the world, we get more than 145.000 vulnerable DBMSs.

4.2. Impact

This section looks into the impact of ransomware. We mainly focus on the financial cost of these attacks, but also mention the first human casualty linked to a ransomware attack.

The cases of ransomware attacks have seen a dramatic increase in the last five years. The estimated cost of these incidents range in the $20 Billions for the year 2020 [40], an

count	name
18613	WARNING
8153	PLEASE_READ
7267	admin
5699	local
3082	README_MISSING_DATABASES
2961	READ1
2412	PLEASE_READ_ME
1404	README
1259	README_YOU_DB_IS_INSECURE
1128	CONTACTME
1057	ENCRYPTED
1053	PWNED_SECURE_YOUR_STUFF_SILLY
905	test
345	immutables
345	piinpoint
259	mr9as_main
258	mr9as_article
257	cache
220	config
198	logs

(a) Scan from 2018-01-16

count	name
19051	PLEASE_READ
7984	admin
6987	local
6561	READ1
1891	README_MISSING_DATABASES
1096	test
312	WARNING
304	README
289	CONTACTME
274	README_YOU_DB_IS_INSECURE
266	cache
258	mr9as_main
257	mr9as_article
249	config
209	logs
192	ceilometer
175	ENCRYPTED
163	bbks
153	gpsreal
145	alarmreal

(b) Scan from 2018-01-17

Table 4.1.: Scans for most used database names in unprotected MongoDB instances

individual attack is quoted at roughly $4 Million. In total, 36% of victims paid to have their data restored, of which 83% regained access to their data.

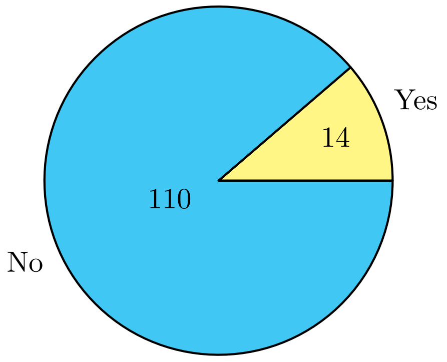
The situation for database ransomware looks worse. Security specialist Victor Gevers surveyed database ransomware attacks in the first half of January 2017. For these 15 days, he lists 124 cases from companies all over the world, as seen in Figure 4.1c. Figure 4.1b shows that only about 8% of affected companies paid the ransom, none of which had their data returned to them. The two cases seen in Figure 4.1d, where data could be restored without recent backups were through a replicated DBMS instance and forensic file recovery. An investigation [41] into MongoDB attacks also shows that ransom information is overwritten quickly, presumably by other ransomware attackers. Table 4.1 shows that table names for ransom messages change daily, if the database is not taken offline. As only the original attacker can provide a backup of the data, this makes paying the ransom to them to restore the data more difficult.

In summary, this means that the only viable strategy to recover from a ransomware attack is to perform regular backups. As shown in Figure 4.1a, only 11.3% of affected businesses followed that practice, leading to high financial impact in a majority of cases.

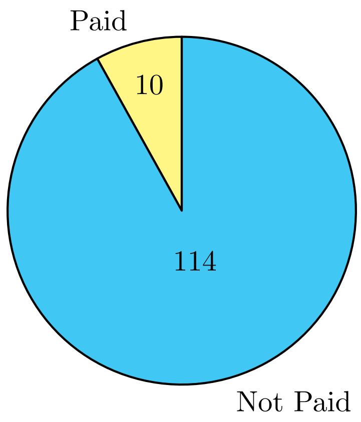
Looking at human cost, rather than financial, shows a much lower impact. The first ransomware-related death only just occurred in September 2020 in Germany [42]. A hospital was hit with a ransomware attack and could not care for a critical patient. Perpetrators of ransomware attacks have since publicly pledged to avoid attacking healthcare facilities. Although this is certainly an improvement, there are other computer systems, whose failure could cause human casualties, like traffic or airport control systems.

4.3. Threat Analysis

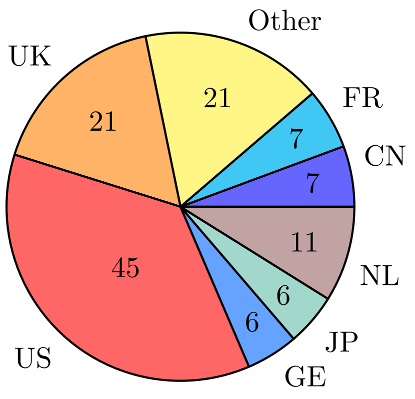
To create a realistic model of a typical ransomware attack, we investigate some examples of past attacks. This work is not concerned with how attackers gained access to the DBMS, consequently, we only look at the sequence of SQL queries issued by the attackers.



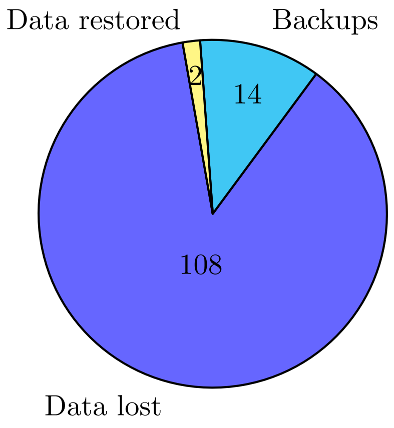
(a) Only 11.3% of victims had recent backups to restore from



(b) Over 8% of victims paid the ransom



(c) Affected companies by country



(d) Only 16 companies regained access to their data

Figure 4.1.: Analysis of ransomware attacks

From an analysis [43] of a ransomware attack from 2017, we can identify the steps taken by the attackers. To get a list of accessible database servers, services like Shodan [44] or BinaryEdge [38] can be queried for free. Exploiting misconfigured systems with default or weak passwords, the attackers then tried to brute-force the root password.

The administrative access was then used to first query information about the databases and table present in the system. In MySQL, this can be accomplished through several methods, like the `SHOW DATABASES` command or by querying the information from the built-in *information_schema* tables. In the analyzed attack, the perpetrators then created a table named `WARNING` either in an existing database or in a new one named `PLEASE_READ`. In the newly created table, the attacker inserts a message informing the victim that they need to pay a certain amount of Bitcoin to a specified Bitcoin address and provide proof via email to get their data restored. With the message in place, the attacker can optionally download a dump of the stored data, before destroying it on the server and disconnecting.

While this scenario is only a description of one specific attack, the steps taken by the attackers were the same in other attacks [45, 46]. They involve querying the DBMS for information on stored databases and tables, creating a table for a ransom message and inserting it, and deleting the stored information. Each of these steps can be accomplished by several SQL commands. We will look into them in more detail in Section 4.4.2.

4.4. Threat Model

This chapter will form a threat model for the intrusion detection system. We first identify the actors involved in the attack in Section 4.4.1, before we look at the attack sequence and its variations in detail in Section 4.4.2.

4.4.1. Areas of Reliance

This section aims to identify all parts, human actors, hardware and software, that contribute to a ransomware attack. In this section, we evaluate each of these components in a top-down approach, from actors and software down to the hardware. The evaluation of each component serves to determine their influence on the security of the overall system.

There are three potential actors: Database administrators, users of the database, and attackers. The database administrators have a stand-out role, they are responsible for deploying a securely configured DBMS. This is vital, as attackers rely on weak or default root passwords to gain access. As system administrators usually have direct access to the physical machine, we deem them trustworthy. The attacker is one of the database's users with remote access to the DBMS and enough privileges to carry out the attack. As the revised DIMAQS plugin should be adaptable to a wide range of attacks, we do not specify these privileges further. Consequently, we cannot distinguish between legitimate users and attackers and have to treat both as untrusted.

The main software component is naturally the DBMS, but the operating system also needs to be considered. On Windows systems, the DBMS can run as a service or be started directly by a user. When run as a service, the database software can only be accessed directly with administrative privileges. Using strong passwords and Windows' User Account Control (UAC) mechanism can minimize the risk in this scenario. Starting a DBMS directly from a user account is usually only practiced by developers, a scenario we do not accommodate, as there is no valuable data to protect. On Unix systems, the same two scenarios are possible, however, the DBMS usually runs under its own user account, making it accessible only by a privileged user.

With these precautions, direct local access to the DBMS is not a concern. Communication with database systems usually happens via TCP/IP networks and local sockets. As these mechanisms are commonly used, we classify them as trustworthy.

We also classify the OS and the hardware itself as trusted. An attacker with the ability to compromise those would have complete access to the system, including the configuration of the DBMS and the DBMS's storage files. It would enable them to destroy data directly or change the configuration to disable our plugin.

4.4.2. Attack Sequence

After defining the components involved in a ransomware attack, we now look at the sequence of SQL queries required to carry out the attack. As discussed before, the attack consists of three steps:

1. Gather Structural Information
2. Leave Ransom Message
3. Delete Information

It is of note, that the attacker might also delete information before leaving the ransom message. The following paragraphs will discuss how an attacker might achieve each of these steps in detail.

Gather Information

As discussed in the previous section, the previously observed ransomware attacks start by querying the database system for structural information about schemas, tables or columns. This information is necessary, as the deletion commands require the names of the targets to be deleted. Jobst created a complete overview [47] of SQL commands that query return this information, it can be seen in Table 4.2.

Delete Information

With the names of existing databases or tables known to the attacker, they can delete them. This can be accomplished in several ways:

- **Deleting a database**
  `DROP {DATABASE | SCHEMA} <db_name>`
- **Deleting one or multiple tables**
  `DROP TABLE [IF EXISTS] <tbl_name> [, <tbl_name>]`
- **Deleting all entries in a table**
  `DELETE * FROM <tbl_name>`

Create a Database for the Ransom Message

To create a database, the attacker has to issue the following SQL command:

```
CREATE DATABASE | SCHEMA [IF NOT EXISTS] <db_name>
```

Table 4.3 is a list of ransomware database names from multiple sources. These names were all used in previous attacks.

However, there were also cases, where attackers left the ransom message in a pre-existing database. Therefore, this step is not crucial for our attack model.

Create a Table for the Ransom Message

In contrast to the previous step, creating a table for the ransom message is always necessary, as the message requires a specific table structure. The following commands create a new table inside a database.

- `CREATE TABLE [IF NOT EXISTS] <tbl_name> <tbl_definition>`
- `CREATE TABLE [IF NOT EXISTS] <tbl_name> [AS] <query_expression>`
- `CREATE TABLE [IF NOT EXISTS] <tbl_name>`
  `{ LIKE <other_tbl_name> | (LIKE <other_tbl_name>)}`

The only table name that was observed in past attacks was ”`WARNING`”, however, this is not a reliable indicator for an attack, as it could be replaced easily. Instead, we consider the database names from Table 4.3 as valid table names.

Insert Ransom Message

The last step in the attack is to insert a message, informing the victim that their databases can be restored by paying some amount of Bitcoin to a specific address and contacting the attackers.

There are two commands that insert values into a table:

Reference		Exposed Information		
Command		Schema	Table	Column
SHOW {DATABASES | SCHEMAS}		🟣		
SHOW TABLES			🟣	
SHOW {COLUMNS | FIELDS}				🟣
SHOW TRIGGERS			🟠	🟠
SHOW OPEN TABLES		🟣	🟣	
SHOW TABLE STATUS			🟣	
Schema	Table	Schema	Table	Column
information_schema	COLUMNS	🟣	🟣	🟣
	COLUMM_PRIVILEGES	🟠	🟠	🟠
	EVENT	🟠		
	FILES	🟣	🟣	
	KEY_COLUMN_USAGE	🟣	🟣	🟣
	PARAMETERS	🟠		
	PARTITIONS	🟣	🟣	
	REFERENTIAL_CONSTRAINTS	🟠	🟠	
	ROUTINES	🟠	🟠	🟠
	SCHEMATA	🟣		
	SCHEMA_PRIVILEGES	🟠		
	TABLES	🟣	🟣	
	TABLE_CONSTRAINTS	🟠	🟠	
	TABLE_PRIVILEGES	🟠	🟠	
	TRIGGERS	🟠	🟠	🟠
	VIEWS	🟠	🟠	🟠
	INNODB_BUFFER_PAGE	🟠	🟠	
	INNODB_BUFFER_PAGE_LRU	🟠	🟠	
	INNODB_SYS_COLUMNS			🟠
	INNODB_SYS_DATAFILES	🟠	🟠	
	INNODB_SYS_FIELDS			🟠
	INNODB_SYS_FOREIGN	🟠	🟠	
	INNODB_SYS_FOREIGN_COLS			🟠
	INNODB_SYS_TABLES	🟠	🟠	
	INNODB_SYS_TABLESPACES	🟠	🟠	
	INNODB_SYS_TABLESTATS	🟠	🟠	
mysql	db	🟣		
	innodb_index_stats	🟠	🟠	
	innodb_table_stats	🟠	🟠	
	proc	🟠		
	procs_priv	🟠		
	tables_priv	🟠	🟠	
performance_schema	file_instances	🟣	🟣	
	file_summary_by_instance	🟣	🟣	
	objects_summary_global_by_type	🟣	🟣	
	table_handles	🟣	🟣	
	table_io_waits_summary_by_index_usage	🟠	🟠	
	table_io_waits_summary_by_table	🟣	🟣	
	table_lock_waits_summary_by_table	🟣	🟣	

Table 4.2.: Summary of SQL commands exposing structural information in MySQL.
Legend: 🟣 Full Exposure
🟠 Partial Exposure

WARNING
README_MISSING_DATABASES
PLEASE_READ
PWNED
PWNED_SECURE_YOUR_STUFF_SILLY
ReadmePlease
CONTACTME
WARNING_ALERT
to_get_DB_back_send_1BTC_to_1DGztzLNz1euFswtqMDWPMWSgwthdpxRtC
PLEASE_READ_56b41cc944bd390932e79827
README
LEIA_ME
AVISO_LEIA_ME
IHAVEYOURDATA
READ_ME
READMEPLS
ENCRYPTED
READ1
README_YOU_DB_IS_INSECURE
AVISO
DB_H4CK3D
PLEASEREAD
DB_DROPPED
REQUEST_YOUR_DATA
BACKUP_DB
Attention
PLEASE_READ_ME
PLEASEREADTHIS

Table 4.3.: Used ransom message table names in MongoDB [41, 48]

- **Insert list of values**
  `INSERT [INTO] <tbl_name> {VALUES | VALUE} <value_list> [, <value_list>]`
- **Insert select**
  `INSERT [INTO] <tbl_name> SELECT ...`

To be able to pay the ransom, a victim would require three pieces of information:

- Amount of Bitcoin to pay
- Bitcoin address to transfer the Bitcoin to
- A way to contact the attackers to restore the data

The type of digital currency is not necessarily Bitcoin, but it is the currency used in almost all recorded attacks. The address to send the money to is a string of 36 characters, containing digits, as well as upper- and lowercase characters. This is easily detectable in the ransom message.

To contact the attackers, we found two variants [43]: Email or a tor hidden service. Both are detectable in the ransom message, by looking for email addresses or .onion domains.

Downloading the Database

We previously discussed that attackers do not always perform a backup of the data they destroy. Therefore, this step is not taken into account when detecting an attack.

5. Requirement Analysis

The aim of this work is to modify the original DIMAQS plugin to be more adaptable to new attack scenarios. To accomplish this goal, we propose modifying the original plugin to replace the Petri-net-based classifier with a machine learning algorithm.

This requires creating the new classifier, along with a separate executable that trains a model for it. We must also modify the existing data samples to make them easier to read, process and extend. Lastly, we must integrate the new classifier into the existing DIMAQS MySQL plugin.

The following sections will gather the requirements for each of these steps and new components.

5.1. Classifier

The classifier is the heart of the DIMAQS plugin. It processes each query as it is issued to the system and produces on of three labels:

- **Benign:**
  These queries are harmless and not part of a ransomware attack
- **Alert:**
  These queries are likely part of an attack, but do not destroy any data. The system administrator should be alerted
- **Backup:**
  These queries are likely part of an attack and do destroy data. Perform a backup of the affected databases and alert the administrator

The reaction to the classifier output is visualized in Figure 5.1.

The main aim of this project is to replace the Petri net classifier with a machine learning algorithm that emits the same labels. The type of machine learning algorithm we chose is an RNN, as they is designed to process sequential data of variable length. As RNNs require numerical input data, we must create an embedding layer, that creates a feature vector from a database query. The feature vector must contain the type of query, as well as indicators for the presence of keywords that signify a ransom message, table or database name.

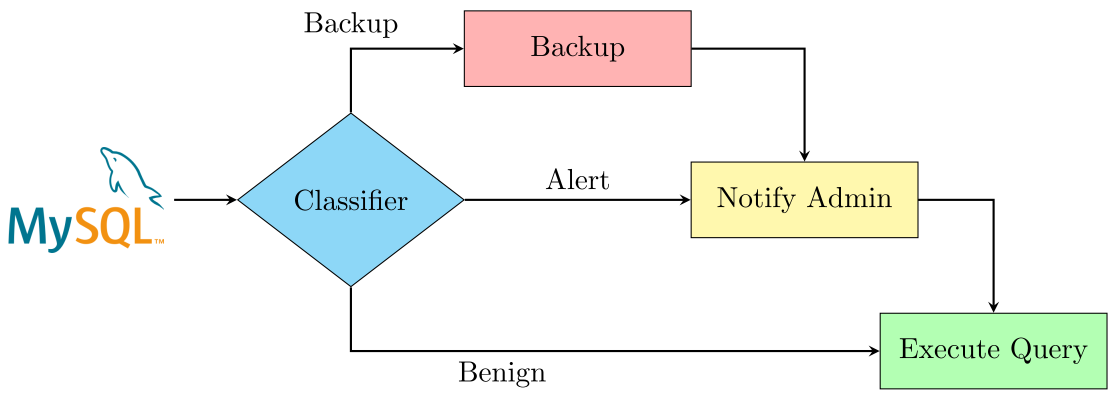


Figure 5.1.: Classifier Reaction Flow

Additionally, the Petri net in the original plugin was designed to only detect attacks that take place over a timespan of less than five minutes. As the LSTM network stores information about past queries in an internal state, we must develop a mechanism to reset that state every five minutes.

The requirements we discussed above will recreate the behavior of the original classifier. However, the machine learning approach is likely to cause an increased impact on performance. To mitigate this issue, we should add the capability of handling incoming queries asynchronously. For that, we need to implement a thread-safe queue of queries to be processed asynchronously, and a mechanism to consume all queries in the queue before processing a critical query synchronously.

5.2. Data Preparation

To create a system that is adaptable to different attack scenarios, we require a well-defined data format to store both benign and malicious SQL query sequences. Defining this format creates the framework to expand the dataset with new attacks in the future. The query sequences will later be used to train the machine learning classifier.

Currently, the benign sequences are stored as raw text in several files, while the malicious samples are generated by a bash script as they are needed. Both of these formats are not easily machine-readable or expandable. As the data needs to be parsed by the training algorithm, as well as humans to add new samples, we require an extensible, human and machine-readable format.

The benign samples must be stored in a structured way, to represent dependencies between different queries of same attack. This structure must be generalized enough to allow different attacks to be expressed by it. For our purposes, an attack can consist of multiple prerequisite queries, queries to alert the database administrator on, and queries that trigger a backup of the affected databases.

As there is a significant amount data to process, we cannot perform the transformations above manually and need a program that facilitates this task.

5.3. Training Executable

To train the machine learning model, we propose creating a separate executable. The training process consists of four steps:

1. Read Benign and Malicious Samples from the Dataset

2. Create Random Sequences of SQL Queries
3. Train the Model
4. Evaluate the Model

We will give details on each of these steps in the following.

5.3.1. Data Ingest

In Section 5.2, we discussed creating a well-defined file format to store the training samples. The training executable must read this data and parse it into its own internal representation that can be used in the subsequent steps.

5.3.2. Sequence Generation

To train the model, we must be able to generate sequences of SQL queries of varying length and type. As machine learning usually benefits from realistic training data, the sequences will be based a sequence of benign queries. Malicious sequence types have the attack inserted into the benign sequence. There are six possible types of query sequence:

- **Benign Clean**
  This is a sequence of only benign queries
- **Benign Dirty**
  In addition to benign queries, this sequence contains either parts of an attack, or a complete attack in the wrong order. These sequences serve to reduce false positive intrusion detections.
- **Alert Clean**
  This is a *Benign Clean* sequence, with the parts of an attack that should prompt an *Alert* label inserted.
- **Alert Dirty**
  This is a *Benign Dirty* sequence, with the parts of an attack that should prompt an *Alert* label inserted after the defanged attack sequence.
- **Backup Clean**
  This is a *Benign Clean* sequence, with the parts of an attack that should prompt a *Backup* label inserted.
- **Backup Dirty**
  This is a *Benign Dirty* sequence, with the parts of an attack that should prompt a *Backup* label inserted after the defanged attack sequence.

These sequences will be generated randomly, however, it must be possible to set the ratio of generated sequence type to produce a model with high detection accuracy.

Another requirement for the training sequences concerns its memory footprint. For evaluating the classifier, we will generate sequences of up to 10,000 queries. Storing those in memory would cause significant overhead. To enable the generation of long sequences, we must create a way to store them with constant memory consumption, regardless of length.

5.3.3. Training

Training a model requires four basic steps:

1. Generate training sequence
2. Extract the features for each query

3. Classify each instance in the sequence
4. Calculate the error from the calculated and known true label
5. Optimize the model

These steps need to be executed in a loop, until the model is sufficiently accurate. To accelerate the training, sequence generation and feature extraction may be run out of band. The resulting feature-extracted sequences can then be entered into a work queue to be processed in the main classify-optimize-loop.

5.3.4. Evaluation

To evaluate a trained model, we can generate and classify query sequences. The classification accuracy will be calculated by comparing the label from the classifier to the known true label. With the true and false positive/negative counts, the classifiers performance can be summarized in a simple f1-score.

5.4. MySQL Plugin

In order to change the classifier in the original DIMAQS plugin, we must change its structure to be able to accommodate the LSTM classifier. This will be done by defining a clear interface to the classifier and rewriting the plugin to use it. This will make future development of the classifier much easier. We must also create an exchangeable policy for reacting to the classifier's output.

One other requirement for the plugin redesign is preparation for porting it to a different DBMS. This port is not in the scope of this work, however, it is part of future work on the plugin. This future goal will also inform some of our design decisions throughout the next chapter.

6. Design

The original DIMAQS system was the first ransomware detection system of its kind for database systems. In this work, we aim to enhance the existing MySQL plugin with a new machine learning classifier. The new system will be more adaptable to new attack scenarios and could also be able to detect previously unseen attacks.

In the past, machine learning has been used to counter ransomware attacks. Researchers used LSTM neural networks to detect suspicious activity in OS API call sequences in [32] and [33]. As we discussed in Chapter 3.3.2, these approaches produced a flexible system with high detection accuracies. However, their research focused on file-system based ransomware instead of DBMSs. File-system ransomware is far more recognizable, as it usually encrypts the files instead of simply deleting information the way database ransomware does. The key difference is the number of API calls needed to carry out the attack. The encryption process happens on a per-file basis, meaning even if the intrusion detection system only detects the complete procedure, at most a few files are lost, as can be seen in [25]. This is not an option for database ransomware, as entire databases can be destroyed with only a single SQL query.

Bringing the flexibility of machine learning ransomware detection systems to database systems is the main goal of this work. To accomplish it, we will first re-design the existing DIMAQS plugin to be more modular. This is necessary, as the original design was highly integrated without well-defined interfaces to its individual components. Section 6.1 describes the individual components of the plugin and how they can interact with each other.

In the subsequent Section 6.2, we detail the new classifier that represents our main contribution to the plugin. We will go over the feature extraction in Section 6.2.1, the neural network in Section 6.2.2 and how the individual parts work together in Section 6.2.3.

Finally, we must create a separate program to use the training samples to train the classifier. The design for this program is detailed in Section 6.3. The application must be able to read the training data, crate realistic query sequences and use them to train and evaluate the machine learning model. Additionally, as the neural network and the algorithms have some variable parameters that can be tuned, the training application must have the functionality to evaluate different combinations of these parameters.

6.1. Plugin Architecture

This section serves to describe the high-level structure of the improved DIMAQS plugin. We will mainly focus on the changes that we are implementing, instead of a complete overview of the plugin's architecture, which is already described in [47].

The existing code is tightly integrated and interconnected, making it hard to work with. We will refactor it to make it more modular, enabling us to change its components more easily. We identify seven core components in the existing plugin:

- **MySQL Interface**
  This is the part of the code that interfaces directly with the MySQL plugin API. It translates calls and received data into the internal representation. This component is kept as-is, as it fits our needs perfectly.
- **MySQL Query**
  The internal representation of a single query. It contains information on query type, affected databases and tables, as well as inserted values. The query representation fits our needs well, however, we define a clear interface to it, so that the plugin can be adapted to different DBMSs more easily.
- **Controller**
  The *Controller* is the central point of the plugin. It initializes the plugin and controls the flow of execution and data flow during its execution. This component has to be adapted to our classifier, as it currently contains a lot of code that is specific to the Petri net classifier.
- **Petri Net Classifier**
  This is the original classifier of the DIMAQS plugin. It is deeply integrated into the system, making it hard to change. As we completely replace this classifier, we define a general classifier interface, making it easier to change the classifier implementation in future efforts.
- **Actions**
  The actions are a set of pre-defined reactions to incoming queries. They include notifying the administrator, performing backups and rewriting data queries to exclude existing backups. These are be kept as-is, as they already have a largely generic interface and can be reused.
- **Query Rewriter**
  The query rewriter is a helper class to facilitate easier modifications of incoming queries. It will be kept as-is.
- **Admin Handler**
  The admin handler is a helper class that keeps track of all database connections and is able to elevate a connection's privileges. Privileged connections can bypass the plugin's security mechanisms and access previously backed-up data. Connections can attain these privileges by presenting a pre-set secret to the admin helper. This component is kept as-is.

We aim to refactor the program to put the *Controller* at the center of the flow of execution. Figure 6.1 shows our revised flow of data and execution. All incoming queries are passed to the *Controller* **(1)**, which first directs it to the classifier **(2)** to generate a classification label for it **(3)**. The original query and the label are then passed to the action policy **(4)**, which emits the correct reactions to them **(5)**. If there are actions to perform, they are subsequently executed **(6)**. This might entail calling the admin notifier **(7)** or rewriting the query. The latter involves calling MySQL's built-in parser with the new query **(8)** to get a

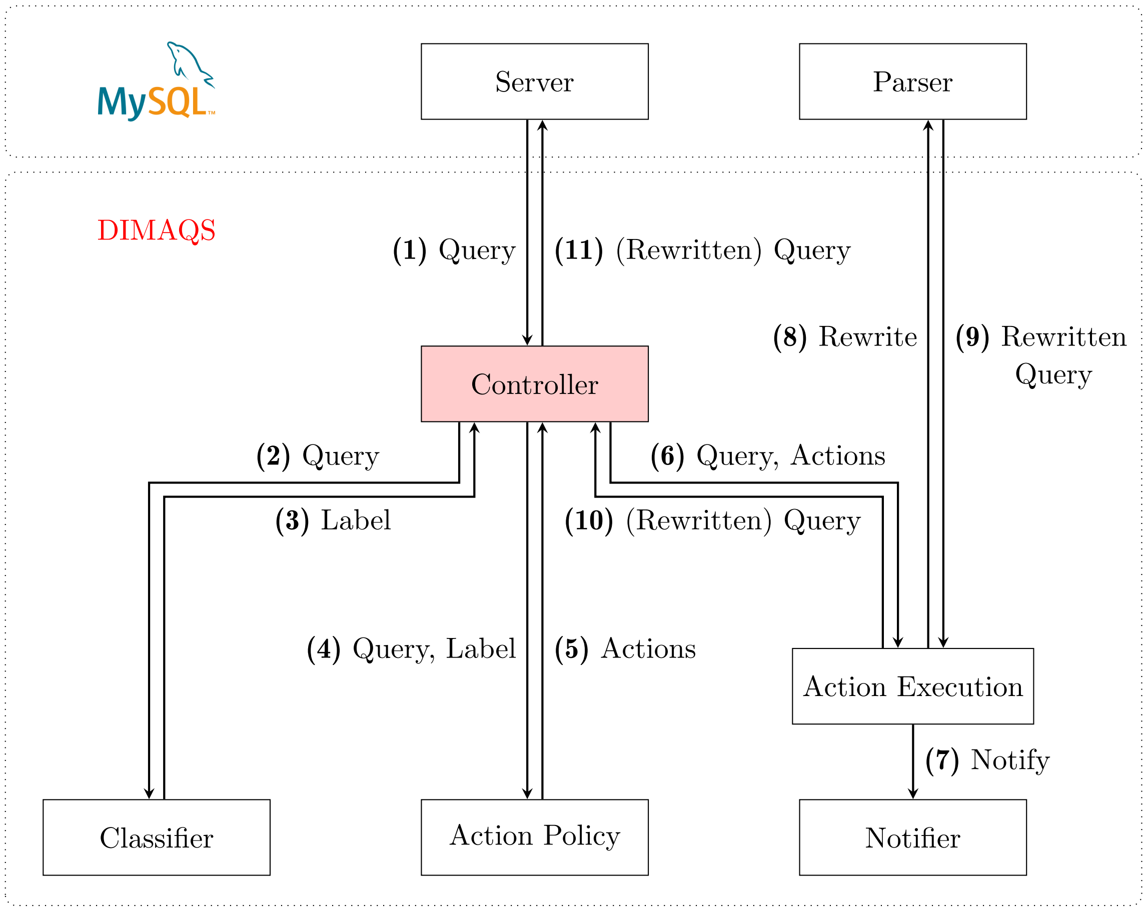


Figure 6.1.: Simplified DIMAQS data flow

parsed query object back **(9)**. The possibly rewritten query is returned to the *Controller* **(10)**, which passes it back to the MySQL server **(11)** to be executed.

To make the revised plugin more modular, we will define interfaces to each major component. The following subsections will describe each interface in detail.

6.1.1. Database Query Interface

The interface to a single database query is kept as simple as possible. The *Classifier* only requires access to the query type, and raw query string. Executing some actions additionally requires access to affected databases, tables and inserted values.

The query type returned by the database query are used by the classifier and the action policy. The possible query types are listed in Table 6.1.

6.1.2. Classifier Interface

The *Classifier* itself only has two points of interaction: a constructor and a method to classify incoming *DatabaseQueries* and return a *Label* for them. Figure 6.1 shows that this is the only interaction required. The possible *Labels* are defined by an enum with three instances: Benign, Alert and Backup. We will cover the internal design of the classifier in further detail in Section 6.2.

6.1.3. Action Policy

The *ActionPolicy* has a similar interface. The constructor requires a *QueryRewriter*, *AdminHandler* and the email address of the administrator to deliver notifications. The pri-

Access Type	Structural	Data	Administrative
Query	StructuralQuery ListColumn ListTable	DataQuery	ShowVariable
Creation	StructuralCreation CreateTable	DataCreation	
Deletion	StructuralDeletion DeleteDatabase DropTable	DataDeletion DataMassDeletion	
Update	StructuralUpdate	DataUpdate	SetVariable
Other			Administrative Other

Table 6.1.: Database query types

mary point of interaction is called *get_actions* and emits a list of *Action*s from a *Database-Query* and a *Label*.

Creating the *Action*s to emit follows the prototype pattern. When the *ActionPolicy* is created, it constructs a prototype for every possible *Action* it can emit. The concrete *Action*s are then created by copying the prototype and customizing it with information from the current query.

6.1.4. Query Rewriter

The *QueryRewriter* is a helper class that interfaces with MySQL's query parser to facilitate modifying queries that were issued to the system. It can manipulate queries to hide the backed-up tables or create backups of tables and databases. To accomplish the former, there is a single method that takes the MySQL connection identifier, the original query and the type of rewrite to perform. To back up tables, the *backup* method takes the same connection information, the original query, a database name and optionally, a table name. If the table name is omitted, all tables of the database are copied to the backup location. If the table name is set, only that table is copied.

6.1.5. Admin Handler

The *AdminHandler* is a helper class to store information about privileged connections. A database administrator can elevate their session by supplying a secret known to DIMAQS. Elevated sessions circumvent the plugin's security mechanism and view or restore backed-up tables. The *AdminHandler* has methods add and remove MySQL connection identifiers, and methods to get and set a connections elevation state. It is constructed with the secret to verify database administrators.

6.1.6. Actions

*Action*s are objects that define the reaction to query and its classification. *Action*s follow the command design pattern. Each *Action* is an object that encapsulates all necessary data to execute a function at a later time. The creation of individual *Action*s follows the prototype pattern. The *ActionPolicy* holds a base instance of every action type. Whenever the *ActionPolicy* has to emit an *Action*, a concrete instance is created from its prototype via a copy constructor. This concrete instance is then completed with information specific to the current query. Usually, these *Action*s are executed before the database query they react to. Each of the *Action*s below implements a common interface: There is a default and a copy constructor, a getter and setter for the *executeLater* property and an execute method to call the action. If the *executeLater* property is set, the reaction will be executed after the database query instead of before. Some *Action*s have additional methods or properties. Because these properties are different between *Action*s, we cannot define a universal method to set them in the interface. However, as a convention, every *Action* has a *create* method that takes the additional parameters.

6.1.6.1. Admin Mode Action

The *AdminModeAction* can toggle DIMAQS's admin mode on and off for specific sessions. Admin mode allows the administrator to view and restore backed-up databases and tables that are normally excluded from queries through the *RewriteAction*. Access to the admin mode is only granted if the user supplies a secret value which can be verified by DIMAQS. To store and manage privileged session identifiers, the prototype *AdminModeAction* has the a companion *AdminHandler* object which is inherited by the concrete instance. That way, every *AdminModeAction* has the same *AdminHandler* to manage privileged connections globally. The *AdminHandler* is passed to the prototype in the constructor. Upon creation of the concrete *Action*, the correct secret to identify administrators, as well as the database connection identifier are set.

6.1.6.2. Rewrite Action

The *RewriteAction* controls how queries are changed to perform certain actions. Queries can be adapted to hide backed-up data and to perform database backups or table backups. The implementation for rewriting queries is encapsulated inside a *QueryRewriter* object, which is passed to the *RewriteAction* in the constructor. The database connection information, original query and the type of query rewrite are set when the concrete instance is created.

6.1.6.3. Alert Action

The *AlertAction* sends notifications to the database administrator. Its base instance is constructed with the email address of the administrator, there are no further customizations necessary for the concrete instance.

6.1.6.4. Backup Action

The *BackupAction* serves to rewrite queries that delete database tables or databases into queries that move them to a secure location. The base version is created with only a *QueryRewriter*. To create a backup of an entire database, a concrete instance requires a database name and the MySQL session identifier. If the instance is also supplied with a table name, only the table is backed up.

6.2. RNN Classifier

The new classifier represents the bulk of our efforts. It will be included in both the DIMAQS MySQL plugin and a separate executable that is used to generate a model for the neural network. The classifier has to process a database query and classify it as *Benign*, *Alert* or *Backup*. Because attacks play out over several queries, the classifier must store knowledge about prior queries.

To keep the plugin as modular as possible, the classifier can only called through the interface described in Section 6.1.2. In this Section, we will first describe how a database query will be transformed into a feature vector in 6.2.1, before we go over the structure of the neural network in 6.2.2 and how both components are used in 6.2.3.

Access Type	Structural	Data	Other
Query	StructuralQuery	DataQuery	
Creation	StructuralCreation	DataCreation	
Deletion	StructuralDeletion	DataDeletion DataMassDeletion	
Update	StructuralUpdate	DataUpdate	
Other			Administrative

Table 6.2.: Internal query types used by the classifier

6.2.1. Feature Extraction

Feature extraction is the first processing step to classify a query. Its purpose is to transform a raw database query into a numerical feature vector that can be processed by the neural network. This component can be called via its *featurize* method, passing the database query as argument. The feature vector is made up of 27 binary dimensions that represent two characteristics of the original query: Its type and the presence of certain keywords in the query. The query type is encoded as a one-hot vector that makes up the first 10 dimensions of the complete feature vector. The remaining 17 dimensions each indicate the presence of a specific keyword. To generate the first part, we translate the type given by the DBMS's query representation into a more generalized query type. These basic operations are shown in Table 6.2 and should be supported by any DBMS, making the shape of the feature vector universal across most database systems.

Keywords that indicate a ransomware attack are detected by trying to match regular expressions against the raw query. This ensures complete flexibility to define even complex keyword sequences to be detected. The keywords include phrases from the ransom message table names in Tables 4.3 and 4.1, as well as the ransom messages compiled by the original DIMAQS project in [47]. All of these expressions are defined as case-insensitive to allow matching against differently capitalized text.

6.2.2. Neural Network

The neural network itself is accessed through a single method, which takes a feature vector and produces one of the three labels we defined in 5.1. As we cannot know the structure of the neural network beforehand, it is kept as flexible as possible. The network consists of a variable number of LSTM layers. We can also define a dropout between the LSTMs. As these networks do not produce a single label but rather a vector of the same dimensionality as the input data, they are followed by a linear layer that reduces those 27 dimensions to one scalar. This structure is shown in 6.2 We then interpret that scalar as follows: Values lower than 0.5 result in a *Benign* label. Everything that rounds to a value of 1 will produce the *Alert* label, and anything equal of higher than 1.5 results in a *Backup* classification.

6.2.3. Orchestration

The *QueryQueue* is the encapsulating object around the neural network, that allows for asynchronous processing of non-critical database queries. We classify all queries that do not modify data as non-critical. The resulting label for these queries will always be *Benign*, however, they still have to be processed by the neural network to capture the overall security state of the DBMS. To accomplish this, queries can be entered into a producer-consumer queue. The neural network consumes these queries off the critical path of execution on a separate thread. The labels produced by elements from the queue are discarded.

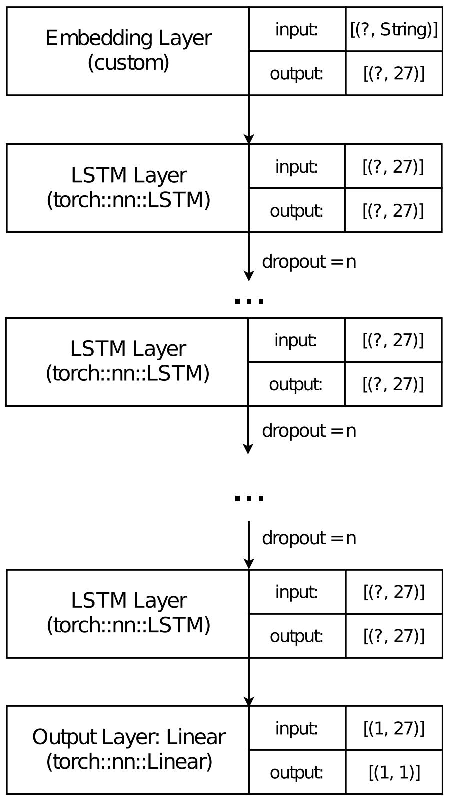


Figure 6.2.: Visual representation of the machine learning model

If a critical query arrives at the system, the queue is first locked, to prevent further queries from being entered. We then wait for the neural network to run through every query in the queue, before the critical query can be processed. Its label is returned by the classifier after unlocking the queue again.

As discussed in Section 5.1, the classifier is required to discard information about queries that were processed more than 5 minutes ago. LSTMs do not have a mechanism to forget information gradually, their internal state can only be reset completely. Our solution to this problem is to use multiple *QueryQueue*s and accordingly, multiple neural networks internal to the classifier. We can have multiple inactive *QueryQueue*s and one active queue. Inactive queues still receive and process all incoming queries, but even critical queries are handled off the critical path. All labels returned by inactive queues are discarded. Active queues are the ones used to create the labels which are returned by the classifier. This is visualized in Figure 6.3.

After an initial warm-up time of 5 minutes, the algorithm shown in Algorithm 2 manages how we activate and deactivate queues. It essentially rotates through the queues in fixed intervals that depend on the amount of *QueryQueue*s used.

This results in the time-to-forget varying between $5min$ and $\frac{n \cdot 5min}{n-1}$. This system is designed to minimize the overhead from several neural networks processing each query. We propose to use 3 instances of the neural network, swapping the active one every 2.5 minutes to have a time-to-forget of 5 to 7.5 minutes.

6.3. Training Executable

In addition to the plugin, we also need to design a separate component to train and evaluate the machine learning model. As we cannot foresee how parameters like training

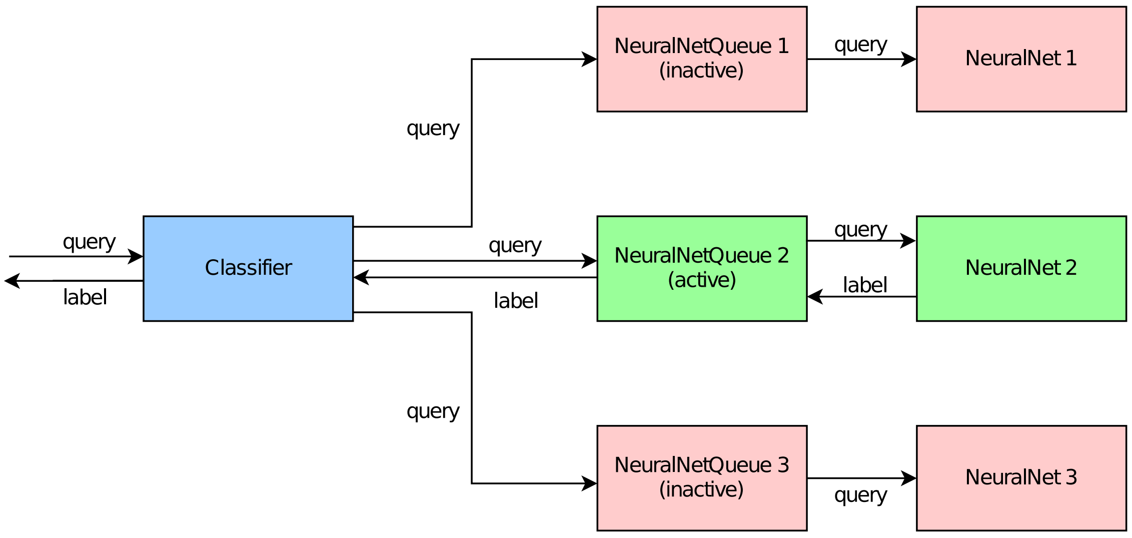


Figure 6.3.: Discarding information by using multiple query queues and neural networks

Algorithm 2 Algorithm for the classifier rotation strategy

```
Require: QueryQueues[ ]: Array of QueryQueues
Require: $n$: Length of QueryQueues[ ]
Require: $i$: Index of the currently active QueryQueue
Require: switch_lock: Global lock for the classifier
1: loop
2:     wait($5min/(n-1)$)
3:     switch_lock.lock()
4:     QueryQueues[$i$].emptyQueue()
5:     QueryQueues[$i$].resetNeuralNet()
6:     $i \leftarrow (i+1) \mod n$
7:     switch_lock.unlock()
```

time, number of LSTM layers, or learning rate influence the performance of the model, this program also serves as a testbed to evaluate various configurations.

During the tuning process, we train and evaluate many different models. This means that the components below must be thread save, to allow us to parallelize the tuning process and reduce the tuning time.

In this section, we will first look into the steps required to train the model. To that end, we will describe how the training data is read and represented in the program in Section 6.3.1, how we extract features from queries without MySQL's query parser in Section 6.3.2, how we generate query sequences in Section 6.3.3 and how the machine learning algorithm is used to train a model in Section 6.3.4. Building on these components, we discuss how models are evaluated in Section 6.3.5.

6.3.1. Data Ingest

This is the initial step of the program. During the data ingest phase, we read all training data into the main memory. This is done to eliminate performance bottlenecks from disk accesses during the training phase. First, each of the two samples of benign data are read into a simple list of strings. The malicious samples are parsed into *MaliciousSequence* structures that are mirroring the structure of the samples stored on disk.

$^2/_3$ of the malicious samples are used for training, the other $^1/_3$ for evaluating the trained model. To simplify the flow of data, we create two *TrainingData* structures. Both of these hold all the benign training samples, but one of them only references the malicious samples used for training, while the other holds the samples for the evaluation phase. As these structures will be referenced from multiple threads while multiple models are being trained simultaneously, they will be heap-allocated and will not be written to after the initial creation.

6.3.2. Feature Extraction

The feature extraction mechanism described in Section 6.2.1 is not applicable to the training executable. The executable is designed to function independently from a DBMS, therefore it can not use MySQL's built-in query parsing functionality. Without a query type supplied by the DBMS, we must emulate that functionality. The training executable determines the query type by matching regular expressions against it.

6.3.3. Query Sequences

Machine learning algorithms benefit from training data that is as close to the real world as possible. To that end, we must combine benign and malicious queries into a realistic query sequence, where the attack is intermixed with queries from regular database operations. The algorithm to accomplish this is detailed in Section 6.3.3.1.

To evaluate the trained models, we have to generate query sequences of significant length. If we want to test for the full attack window of 5 minutes, the sequence will have to be at least 600.000 queries in length. Therefore, can not store the full sequence in memory. We instead opt to create an iterator-like interface to the query sequence that calculates the next query from list indices and references to the training data. The logic behind this streaming sequence is described in Section 6.3.3.2.

6.3.3.1. Sequence Generation

This section details how we generate realistic query sequences. We differentiate three principal types of query sequence:

- **Benign Sequence**
  This sequence does not contain any queries from the attack samples. Every query from this sequence should be classified as *Benign*.
- **Alert Sequence**
  This sequence contains the general prerequisites, as well as the prerequisites for an alert. The last query from the sequence is the query that should raise an *Alert*.
- **Backup Sequence**
  This sequence contains the general prerequisites, as well as the prerequisites for a backup classification. The last query from the sequence is the query that should result in a *Backup* classification.

The type the algorithm generates is chosen at random, however, the ratio between types is influenced by weights that can be attached to them. The algorithm receives the length of the sequence to generate, as well as the weights as input parameters. Every query type is based on a benign sequence with the target length. It is created by randomly indexing into the benign dataset.

To build *Alert* or *Backup* sequences, we choose the positions of the attack queries in the sequence randomly, but in order. The attack queries are inserted into the sequence, moving the benign queries down the list. The queries that exceed the target length are cut off.

Additionally, we can modify each of the three sequence types above to be *dirty*. This is what we call inserting queries from the attack samples into the sequence in a way that should not trigger a classification other than *Benign*. This can be done by intentionally breaking the dependencies between queries in an attack, for example by reversing their order or omitting the prerequisites. To build a dirty *Alert* or *Attack* sequence, we chose a pivot element in the sequence at random. We insert the queries to dirty the sequence before the pivot and the attack queries after the pivot.

6.3.3.2. Streaming Query Sequence

The *StreamingQuerySequence* is the object that stores query sequences. The design goal of this component is a near-constant memory footprint independent of sequence length. To that end, we do not store a copy of the individual queries in the sequence. Because the vast majority of a long query sequence consists of benign queries, we can achieve this low memory footprint by storing a pointer to the complete list of benign queries, as well as a current index into that list. To insert attacks or dirty queries, we store the queries to insert as lists. We also store lists of insertion indices to place the queries at the correct position in the sequence.

The interface to the *StreamingQuerySequence* only has three methods:

- ***hasNext()***
  This method returns *true* if there is another element in the sequence and *false* otherwise.
- ***next()***
  This method returns the feature vector of the next query, as well as the correct label for it. It uses the feature extractor described in Section 6.3.2.
- ***compact()***
  This method returns a matrix of all feature vectors left in the sequence, as well as a vector of all correct labels. While there is another query left, it calls *next()* and appends the vector to the output matrix. This will consume the entire *StreamingQuerySequence.*

6.3.4. Training Loop

The training loop is where the program spends the most time. The general principle is described in Section 2. In general, a training loop for a neural network classifier repeats the following steps:

1. Reset network state
2. Fetch training data and target labels
3. Classify training data yielding the classifier labels
4. Calculate the error between classifier and target labels using an *error function*
5. Optimize the model using an *optimizer*

Usually, one iteration of this loop is called an *epoch.* Epochs are used to measure the training time. During each epoch, the machine learning algorithm runs through the entire set of training data. However, as we generate query sequences randomly, the number of possible instances is near-infinite. Therefore, we process only one query sequence during each iteration of the loop. The training time is measured by the number of processed query sequences.

As error function, we use the mean squared error. The error between a vector of target labels t and input labels i is calculated as follows:

$$\vec{e} = \begin{pmatrix} (t_1 - i_1)^2 \\ (t_2 - i_2)^2 \\ (t_3 - i_3)^2 \\ ... \\ (t_n - i_n)^2 \end{pmatrix}$$

Mean squared error is a standard error function and has proven to work well for our purposes.

To optimize the model, we use the *Adam* optimizer, which we described in Section 2.4. The *Adam* optimizer has proven to reduce the necessary training time significantly.

To speed up the training process, query sequences are generated in parallel to the training loop by a thread pool. The compacted sequence is placed in a bounded queue to be used for training. The feature extraction of many queries is quite expensive, as each query needs to be matched to 27 regular expressions. Therefore, parallelizing this step is massively beneficial.

6.3.5. Model Evaluation

When a model is trained, we need to evaluate its accuracy. For easy comparison, we break the results down into f1-scores for each class. From these, we can calculate an overall macro and weighted f1-score for the trained model. These metrics described in Section 2.5. We calculate the f1-scores for each of our three classes, as well as the arithmetic and weighted averages.

As almost all queries in a long sequence will be benign, a lower weighted average will be indicative of a higher false-positive intrusion detection rate, while the macro f1-score is more sensitive towards a high false-negative rate.

With these metrics, we can effectively evaluate and compare different models. The concrete steps for evaluating a trained model are shown in Algorithm 3. We loop for n iterations. Each iteration, we reset the neural network and generate a new query sequence. For each query in the sequence, we generate a label using the neural network. The generated label and the known true label are compared to update the statistical information.

Algorithm 3 The algorithm used for evaluating a trained model

```
Require: $n$: Number of iterations for the evaluation loop
Require: NeuralNet: Neural net with the trained model applied
1: Stats.init() : Object to gather the statistics from the evaluation run
2: for $n$ times do
3:     QuerySequence $\leftarrow$ generateSequence()
4:     NeuralNet.reset()
5:     while QuerySequence.hasNext() do
6:         (query, targetLabel) $\leftarrow$ QuerySequence.next()
7:         classifierLabel $\leftarrow$ NeuralNet.classify(query)
8:         Stats.update(classifierLabel, targetLabel)
   return Stats
```

7. Implementation

In this Chapter, we will detail how the components described in Chapter 6 are implemented. To achieve our goal of creating a system that is easy to adapt to new attack techniques and to prepare porting the plugin to different DBMSs, we will continue to focus on modularity.

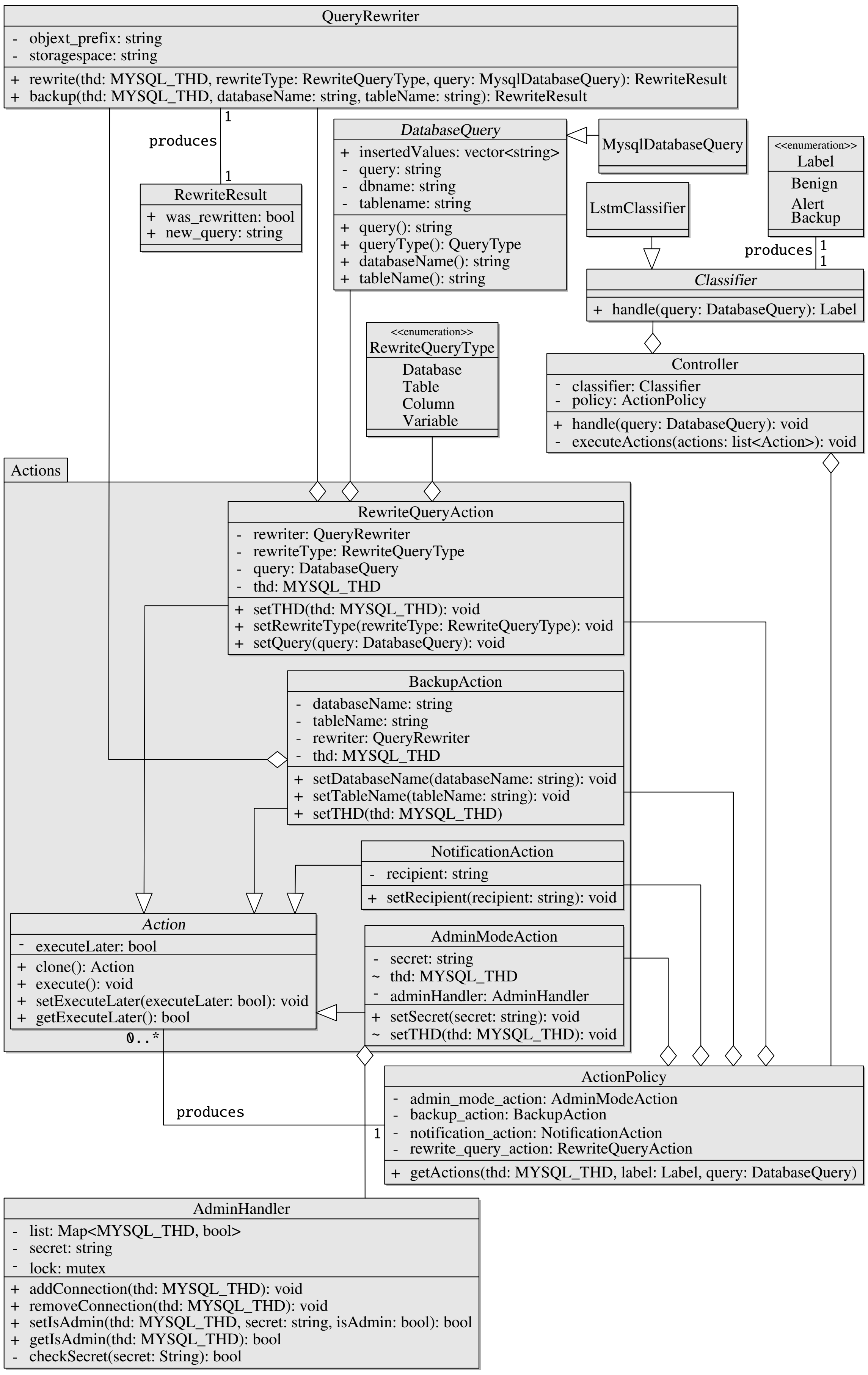
The outline of this chapter is similar to the previous one: As they are the most important components, we will start with the general architecture of the plugin in Section 7.1, before drilling down into the classifier itself in Section 7.2. The subsequent Sections 7.3 and 7.4 will cover the data preparation procedure and the training executable.

7.1. Plugin Architecture

In this chapter, we will describe the implementation details of the DIMAQS plugin. The programming language of the original DIMAQS plugin was C++, as it enables modern, object-oriented programming and can interface with the MySQL plugin's C API. As we are merely modifying the plugin, this will not change. One of the design goals for the plugin was to accomplish better modularity by creating well-defined interfaces to each component. This way, future research can adapt the plugin more easily to new databases, or change out individual components.

As the plugin's interface to the DBMS has been described in detail by Jobst in [47], we will refrain from duplicating the description. We will however explain two key objects generated by MySQL, as they are important for the comprehension of this chapter. MySQL's *LEX* object is its internal representation of a query. It specifies, among other things, the query type and -parameters. We use the *LEX* object to gather information about the query and optionally change it. The *MYSQL_THD* object is an identifier for the database connection that issued a query. This context is required to promote a certain database user to have administrative privileges to manage the backups created by DIMAQS. It is also necessary to execute rewritten database queries in the correct context.

Each of the following subsections will cover one of the major components of the DIMAQS plugin we identified in Section 6.1. The components are visually represented in a Unified Modeling Language (UML) class diagram in Figure 7.1. To simplify the diagram, we omitted the interface to the DBMS. There are also no details on the new classifier, a detailed description of it will follow in Section 7.2.



QueryRewriter
- objext_prefix: string
- storagespace: string
+ rewrite(thd: MYSQL_THD, rewriteType: RewriteQueryType, query: MysqlDatabaseQuery): RewriteResult
+ backup(thd: MYSQL_THD, databaseName: string, tableName: string): RewriteResult
produces
1
1
RewriteResult
+ was_rewritten: bool
+ new_query: string
DatabaseQuery
+ insertedValues: vector<string>
- query: string
- dbname: string
- tablename: string
+ query(): string
+ queryType(): QueryType
+ databaseName(): string
+ tableName(): string
MysqlDatabaseQuery
<<enumeration>>
Label
Benign
Alert
Backup
LstmClassifier
produces
1
1
Classifier
+ handle(query: DatabaseQuery): Label
<<enumeration>>
RewriteQueryType
Database
Table
Column
Variable
Controller
- classifier: Classifier
- policy: ActionPolicy
+ handle(query: DatabaseQuery): void
- executeActions(actions: list<Action>): void
Actions
RewriteQueryAction
- rewriter: QueryRewriter
- rewriteType: RewriteQueryType
- query: DatabaseQuery
- thd: MYSQL_THD
+ setTHD(thd: MYSQL_THD): void
+ setRewriteType(rewriteType: RewriteQueryType): void
+ setQuery(query: DatabaseQuery): void
BackupAction
- databaseName: string
- tableName: string
- rewriter: QueryRewriter
- thd: MYSQL_THD
+ setDatabaseName(databaseName: string): void
+ setTableName(tableName: string): void
+ setTHD(thd: MYSQL_THD)
NotificationAction
- recipient: string
+ setRecipient(recipient: string): void
Action
- executeLater: bool
+ clone(): Action
+ execute(): void
+ setExecuteLater(executeLater: bool): void
+ getExecuteLater(): bool
AdminModeAction
- secret: string
~ thd: MYSQL_THD
- adminHandler: AdminHandler
+ setSecret(secret: string): void
~ setTHD(thd: MYSQL_THD): void
0..*
produces
1
ActionPolicy
- admin_mode_action: AdminModeAction
- backup_action: BackupAction
- notification_action: NotificationAction
- rewrite_query_action: RewriteQueryAction
+ getActions(thd: MYSQL_THD, label: Label, query: DatabaseQuery)
AdminHandler
- list: Map<MYSQL_THD, bool>
- secret: string
- lock: mutex
+ addConnection(thd: MYSQL_THD): void
+ removeConnection(thd: MYSQL_THD): void
+ setIsAdmin(thd: MYSQL_THD, secret: string, isAdmin: bool): bool
+ getIsAdmin(thd: MYSQL_THD): bool
- checkSecret(secret: String): bool

Figure 7.1.: Simplified UML class diagram of the DIMAQS plugin

7.1.1. Controller

The controller is the central component of the plugin. It manages the calls to all other components and is also responsible for initializing the plugin.

7.1.1.1. Initialization

The initialization routine creates the plugin's *Classifier* and *ActionPolicy* objects that are used at run-time. During the initialization phase, we also interface with MySQL and set up triggers to get notified when queries are issued to the system.

7.1.1.2. Event Handling

As soon as an event, we created triggers for happens, the controller's *audit_notify* method is called. The procedure will check the event and do one of three things:

- If the event was the creation of a new connection, it is added to the *AdminHandler*.
- If the event was the completion of a database query, any post-query *Action*s are executed.
- If the event was the parsing of a new query, we classify and react to the query as described below.

When a new query is finished being parsed by the DBMS, we create our own *DatabaseQuery* object for it. This is then passed to the classifier, producing a label to the query. From the query type and the label, the *ActionPolicy* emits the appropriate reactions to the query. These are either executed before the original query or after it is completed.

7.1.2. Database Query

The interface to the database query component we described in Section 6.1.1 is implemented as the abstract class shown in Listing 7.1.

The *QueryType* returned by the database query is implemented as an enumeration with possible instances as described in Section 6.1.1.

The only implementor of this interface is the *MysqlDatabaseQuery* class. It represents a database query specific to the MySQL DBMS. To implement the *queryType* method, the implementation needs to translate the query type returned by MySQL's *LEX* object into a more generalized *QueryType* instance. To accomplish this, we introduce the *isCreateTable*, *isListColumn*, *isListDatabase*, *isListTable*, and *isListVariable* helper methods, as these commands cannot be directly inferred from the given query types and can have a more complex syntax. All other query types can be directly translated.

The *MysqlDatabaseQuery* also holds references to MySQL's internal query representation and the MySQL connection identifier, as well as getters and setters for both.

7.1.3. Query Rewriter

The *QueryRewriter* is a helper class that facilitates rewrite operations for database queries. It has two public methods to access its functionality.

The *rewrite* method takes a query, *MYSQL_THD*, and *RewriteQueryType* and changes the original query to exclude the previously backed-up tables and databases. This is accomplished by appending " `WHERE ‘Database‘ not like ’dmaqs’`" to the end of the SQL statement.

The *backup* method takes the *MYSQL_THD*, a database name, and optionally a table name. If the table name is supplied, only that table is backed up, otherwise, we perform a backup of the whole database. As there is no easy way to rename or move tables, the procedure copies the affected tables into a new database that is hidden from end-users.

Listing 7.1: Interface to the database query defined in an abstract class

```
class AbstractDatabaseQuery
{
    public:
        AbstractDatabaseQuery();
        virtual ~AbstractDatabaseQuery();

        virtual string query() const = 0;
        virtual QueryType queryType() const = 0;
        const string& databaseName() const { return dbname_; }
        const string& tableName() const { return tablename_; }

        vector<string> inserted_values_;

    private:
        string dbname_;
        string tablename_;
        string query_;
};
```

7.1.4. Admin Handler

The *AdminHandler* is another helper component. It stores information about every database connection in one location. Whenever a client connects to the DBMS, the connection identifier gets added to a Map in the *AdminHandler*. If the client presents a secret value to the DIMAQS plugin, the *AdminHandler* verifies that secret and marks the connection as elevated. Connections with elevated privileges can access backed-up databases by circumventing the query rewrite mechanism.

Implementing this functionality requires a private *checkSecret* method to verify the secret supplied by the client, and four public methods: *addConnection*, *removeConnection*, *setAdmin*, and *checkAdmin*.

The *checkSecret* method simply compares the known correct secret value to the supplied one, by way of string comparison. This is implemented as a separate function to allow for more complex verification methods in the future, like hashing the secret value first, so that it is not stored in clear-text form.

AddConnection and *removeConnection* manage the list of active database connections. They both take a connection identifier (*MYSQL_THD*) and add it to, or remove it from a Map. Each entry in the Map is identified by the *MYSQL_THD* and saves the privileged state as a Boolean.

SetAdmin takes a *MYSQL_THD* and a Boolean and sets the entry corresponding to the connection identifier to the value of the Boolean. This is the only mechanism that can elevate a connection's privileges. The *checkAdmin* method takes a connection's identifier and retrieves that connection's privileged state from the Map.

7.1.5. Actions

Actions follow the command design pattern, each *Action* is an object that holds all necessary data to execute a function at a later time. These functions are reactions to incoming

Listing 7.2: Interface to an Action

```
class AbstractAction {
    public:
        AbstractAction();
        AbstractAction(const AbstractAction& other);
        virtual ~AbstractAction();

        virtual void execute() = 0;

        virtual bool equals(const AbstractAction& other) const
            = 0;

        void setExecuteLater(bool executeLater) {
           executeLater_ = executeLater; }
        bool getExecuteLater() const { return executeLater_; }
    private:
        bool    executeLater_;
}
```

database queries. As shown in Listing 7.2, the interface to an *Action* is defined in an abstract class that serves as parent class for every specific *Action* and has to be implemented by them.

As the *ActionPolicy* follows the prototype patterns when creating *Actions*, each *Action* has to provide a default and a copy-constructor. The *setExecuteLater* and *getExecuteLater* methods manage the *executeLater_* property that specifies that the *Action* has to be executed after the query that caused its creation.

In addition to these methods, every *Action* can define its own fields and methods to set all the parameters required for its execution. As a convention, the *Action* implements a *create* method that takes all additional parameters. The *execute* method executes the action.

7.1.5.1. Admin Mode Action

The *AdminModeAction* is responsible for changing a connection's privileges. In addition to the default *executeLater_* property, it holds an *AdminHandler*, a *MYSQL_THD*, and the supplied secret. The *AdminHandler* is passed to the *Action* in the constructor. The other parameters are set when the concrete instance is created from the prototype: A *create* method takes the values for the connection identifier and secret, copies the prototype, sets the values in the concrete instance, and returns it.

The *execute* method of the *AdminModeAction* calls the *AdminHandler* to verify the secret and set the privileged state of the connection.

7.1.5.2. Rewrite Action

The *RewriteQueryAction* is responsible for hiding the backed-up tables from the end-user or attacker. It is designed with flexibility in mind and can be extended to perform more complex rewrite operations. The prototype of the *RewriteQueryAction* holds a *QueryRewriter* that is passed to it in the constructor. A concrete instance holds the *MysqlDatabaseQuery* to rewrite, a *MYSQL_THD*, and a *RewriteQueryType*. The last is an enumeration of all possible types of rewrite operations. Currently, these are *Database*, *Table*, *Column*, and *Variable*. Each type instructs the *QueryRewriter* to exclude instances of the corresponding object type that were created by the plugin from the query results. The current plugin design only uses the *Database* and *Variable* rewrite types. The *MysqlDatabaseQuery*, *MYSQL_THD*, and *RewriteQueryType* are passed to the *RewriteQueryAction* in the *create* method, that copies the prototype, applies the given parameters to the instance, and returns it.

The *execute* method passes the database query, connection identifier, and rewrite type to the *QueryRewriter* to perform the rewrite operation.

7.1.5.3. Notification Action

The *NotificationAction* is responsible for alerting the database administrator when an attack is suspected. The prototype *Action* holds the email address of the recipient as a string, it requires no additional parameters at run-time, consequently, the *create* method only calls the copy constructor to return the concrete instance.

The *execute* method calls *sendmail* to send a message to the administrator. The message is currently hard-coded and reads: "DIMAQS: Incident appeared. Please check your MySQL server." This behavior can be easily customized in the future.

7.1.5.4. Backup Action

The *BackupAction* performs the backup of databases or tables before they can be deleted by an attacker. To perform the backup task, the prototype holds a *QueryRewriter* which is passed to it in the constructor. The concrete instance requires a *MYSQL_THD*, a database name, and optionally, a table name to back up. These are passed in the *create* function, the table name is an optional parameter with an empty string as the default value.

When executed, the *Action* passes database connection identifier, database name, and table name are passed to the *QueryRewriter*'s *backup* method which either copies the entire database or, if a table name is provided, only one table to a hidden location.

7.1.6. Action Policy

The *ActionPolicy* is the object that controls which reaction is appropriate for a labeled query.

The *ActionPolicy*'s constructor requires a *QueryRewriter*, *AdminHandler*, and the email address of the database administrator as a string. These parameters are necessary to create the prototype *Action*s that serve as templates for the concrete *Action*s the policy emits at run-time.

The *get_actions* method is the only public method of the *ActionPolicy*. It takes the database connection identifier, the *DatabaseQuery*, and its label. The procedure first creates an empty list of actions. It then considers the type of *DatabaseQuery*:

- ***ShowVariable***
  If the command will leak internal DIMAQS variables to the end-user, they are excluded from the output by adding a *QueryRewriteAction* of type *Variable* to the output list.
- ***SetVariable***
  The *SetVariable* SQL command is used to provide the administrator's secret to DIMAQS. Consequently, we add an *AdminModeAction* to the output list.

- ***ListDatabases***
  The *ListDatabases* command could leak information about backed-up tables to the end-user. To mitigate this, we add a *QueryRewriteAction* of type *Database* to the output list.

Next, the policy considers the classifier label for the query:

- ***Benign***
  No further actions are required
- ***Alert***
  The database administrator needs to be informed about a suspected attack. We add a *NotificationAction* to the output list.
- ***Backup***
  This label means we suspect a malicious deletion of data. The policy consequently adds a *NotificationAction* and a *BackupAction* to the output list to perform a backup and inform the database administrator about the attack.

The *Action*s in the output list get created by calling their prototype's *create* method with all necessary information about the current query. After all *Action*s are added to the output list, it is returned by the function.

7.2. LSTM Classifier

The new classifier is the core of our efforts to bring machine learning to the DIMAQS database intrusion detection system. We already described the interface and design goals of the classifier in Sections 6.1.2 and 6.2, the following sections will cover every component of the classifier in detail.

To implement the neural network component, we decided to use *PyTorch*'s C++ library *LibTorch* [49]. It has the same functionality as *PyTorch*'s *Python* library, but can be directly integrated into the existing DIMAQS C++ codebase. *Pytorch*'s advantage over other machine learning frameworks is its simple configuration and pre-existing LSTM implementation. The decision to use *PyTorch* also influences the feature extraction component, as it dictates the file formats to use for feature vectors.

7.2.1. Feature Extraction

The feature extraction component is responsible for transforming the *DatabaseQuery* into a feature vector. The format required by *PyTorch* is a *Tensor*, a multi-dimensional matrix. In the case of LSTMs, we require a 3-dimensional *Tensor*. The first dimension specifies a batch number, the second a mini-batch number, and the last a component in the feature vector. As we are only processing single queries, there is no need for batches or mini-batches, but the shape of the matrix is still required by the built-in LSTM implementation. Hence, we only construct *Tensor*s with the shape $\{1, 1, 27\}$.

As we discussed in Section 6.2.1, features are comprised of 27 binary dimensions, 10 to encode the type of database query, 17 to indicate the presence of certain keywords. The regular expressions used to detect the ransomware keywords are listed in Appendix A.2.

The *featurize* method is the only method that gets called to transform a query into a feature *Tensor*. Its first action is to translate the *QueryType* returned by the database query into an *LstmQueryType* that can be used internally by the classifier. Because the classifier should be functional on different DBMSs, the query type used by the classifier is not the same as the one that is returned by the *MysqlDatabaseQuery*. The *translateQueryType* method simplifies the original query type to be one of the more generalized operations possible on most DBMSs with the following mapping:

- StructuralQuery, ListColumn, ListDatabase, ListTable → StructuralQuery
- StructuralCreation, CreateTable → StructuralCreation
- StructuralDeletion, DropDatabase, DropTable → StructuralDeletion
- StructuralUpdate → StructuralUpdate
- DataQuery → DataQuery
- DataCreation → DataCreation
- DataDeletion → DataDeletion
- DataMassDeletion → DataMassDeletion
- DataUpdate → DataUpdate
- SetVariable, ShowVariable, Administrative, Other → Administrative

These types are stored as an enumeration. As every instance in a C++ enumeration gets assigned an integer according to its position in the enumeration, we assign the index in the feature tensor that corresponds to that integer to 1:

```
tensor.index_put_({0, 0, queryType}, 1);
```

The presence of keywords that signify a ransomware attack is detected by matching multiple regular expressions against the query. These regular expressions are listed in Section 6.2.1. In a loop, we go over the regular expressions and update the correct index in the *Tensor*:

```
int i = LstmQueryType::Administrative + 1;
for (const std::regex &regex: ransom_regexs) {
    out.index_put_({0, 0, i}, std::regex_match(query, regex));
    i++;
}
```

The resulting *Tensor* is returned by the *featurize* method.

7.2.2. Neural Network

The neural network is the heart of the classifier. It takes feature vectors as input and produces labels for them. The feature vectors have to be presented as *PyTorch Tensors* with the following shape: $\{x, y, z\}$, where x denotes the number of batches, y the number of mini-batches, and z the number of features. The network returns a Tensor that contains only a single integer, which corresponds to the label of the query: 0 for *Benign*, 1 for *Alert*, and 2 for *Backup*.

Neural networks are initialized by loading the trained model from a file. The *load* method performs this function, taking only a file path as a parameter and returning an initialized neural network:

```
Net* load(const std::string &path) {
    torch::serialize::InputArchive input_archive;
    input_archive.load_from(path);
    Net* net = new Net();
    net->load(input_archive);
    return net;
}
```

A slightly simplified version of the code enabling the operation of the network is shown in Listing 7.3. For better readability, we left lines specific to GPU acceleration out of the code snippet.

The neural network has an LSTM layer and a linear layer. The LSTM layer takes the 27-dimensional input and produces an output of the same dimensionality, as well as an internal state. The internal state is stored in the network and the output is transformed into a single number by the linear layer. This is all done in the *step* method. A mutex ensures that the network can only be used by one thread simultaneously. The *forward* method is slightly more complex, as it takes a batch of *featurize*d queries and processes them as a batch operation. This enables some parallelization in *PyTorch* and can improve performance during the training phase.

7.2.3. Neural Network Queue

The *NeuralNetQueue* is the buffer in front of the neural network. It is designed to decouple the neural network from the rest of the system, so that its performance impact is minimal for most queries. To accomplish this, we use Cameron Desrochers *Concurrentqueue* library [50]. It is a very fast, lock-free Multi-Producer, Multi-Consumer (MPMC) queue. Its design is described in [51].

The *NeuralNetQueue* is initialized by passing the path to the machine learning model to the constructor. The constructor then creates a new neural network and MPMC queue. From those elements, a new consumer thread is created that continuously takes queries from the queue and passes them to the neural network. Access to the queue is governed by a shared mutex. This allows us to access the queue from multiple threads simultaneously, but still have the ability to lock it, if necessary.

The queue has a method to change its state from active to inactive. The *set_inactive* method locks the queue for all other threads, empties the queue, and resets the neural network. Afterward, the queue is unlocked again. This procedure is shown in Listing 7.5.

To enqueue new queries that can be handled asynchronously, the *handle_async* method acquires a shared lock on the mutex, enqueues the query, and frees the lock. If a query classification is required immediately, the *handle_now* method locks the queue and processes every element still enqueued. This is necessary to bring the neural network's state up to date with all queries the system encountered. After the queue is emptied, the new query is presented to the neural network and the queue is unlocked. The network's output is transformed into a *Label* which is returned by the method. Code for this procedure is presented in Listing 7.4.

The final piece of the *NeuralNetQueue* is the consumer thread. It is created in the constructor and processes every enqueued query in an infinite loop. The consumer first acquires a shared lock on the queue's mutex and fetches an element from the queue. Then it runs the feature extraction and presents the *Tensor* to the neural network. The result is discarded, queries in the queue are merely necessary to update the neural network's internal state. The last step in the loop is to free the shared lock.

7.2.4. Orchestration

In this section, we aim to describe how all the components of the new classifier work together. The *LstmClassifier* object is the entry point to the classifier. It holds several *NeuralNetQueues* and provides the public *handle* method that takes a database query and produces a *Label*.

The *LstmClassifier*'s constructor initializes the set of *NeuralNetQueue*s and a mutex that is locked during query handling and switching of active queues. The algorithm discussed

Listing 7.3: Simplified code for the neural network

```
struct Net : torch::nn::Module {
    torch::nn::LSTM lstm;
    torch::nn::Linear linear;
    tuple<Tensor, Tensor> hidden_state;
    bool initialized;
    std::mutex* netlock;

    //Produce labels for a single query
    torch::Tensor step(const torch::Tensor &input) {
        tuple<Tensor, tuple<Tensor, Tensor>> lstm_out;
        netlock->lock();
        if (initialized) {
            lstm_out = lstm(input, hidden_state);
        } else {
            lstm_out = lstm(input);
            initialized = true;
        }
        hidden_state = get<1>(lstm_out);

        torch::Tensor out = linear(get<0>(lstm_out));
        netlock->unlock();
        return out;
    }

    //Produce labels for a sequence of queries
    torch::Tensor forward(const torch::Tensor &input) {
        tuple<Tensor, tuple<Tensor, Tensor>> lstm_out;
        netlock->lock();
        if (initialized) {
            lstm_out = lstm(input, hidden_state);
        } else {
            lstm_out = lstm(input);
            initialized = true;
        }
        hidden_state = make_tuple(
            get<0>(get<1>(lstm_out)).index({get<0>(get<1>(
               lstm_out)).size(0) - 1, 0}).view({1, 1, 27}),
            get<1>(get<1>(lstm_out)).index({get<0>(get<1>(
               lstm_out)).size(0) - 1, 0}).view({1, 1, 27}));

        Tensor out = linear(get<0>(lstm_out));
        netlock->unlock();
        return out;
    }
};
```

Listing 7.4: Processing a critical query synchronously

```
Label handle_now(const DatabaseQuery& query) {
    add_lock->lock();
    MysqlDatabaseQuery queries[100];
    int count;
    while (queue->size_approx() > 0) {
        //process all elements in queue in bulk
        count = queue->try_dequeue_bulk(queries, 100);
        net->forward(featurize_bulk(queries, count));
    }
    float result = net->step(featurize(query));
    add_lock->unlock();
    if (result < 0.5) {
        return Label::Benign;
    } else if (result >= 1.5) {
        return Label::Backup;
    } else {
        return Label::Alert;
    }
}
```

Listing 7.5: Setting a *NeuralNetQueue* inactive by emptying it and resetting the neural network

```
void set_inactive() {
    add_lock->lock();
    MysqlDatabaseQuery queries[100];
    while (queue->size_approx() > 0) {
        //dequeue everything
        queue->try_dequeue_bulk(queries, 100);
    }
    net->initialized = false;
    add_lock->unlock();
}
```

in Section 6.2.3 and shown in Algorithm 2 is implemented in a separate thread to minimize its impact on the classifier's performance. The constructor also starts that thread, which updates the *active_net* pointer that identifies the active *NeuralNetQueue*.

Listing 7.6 shows the main *handle* method of the *LstmClassifier*. The default output is the *Benign* label. A loop runs through every *NeuralNetQueue*. If the query is critical and the current *NeuralNetQueue* is the active one, the query is processed immediately and the output is changed to the returned *Label*. Inactive *NeuralNetQueues* handle the query asynchronously, non-critical queries are added asynchronously to all *NeuralNetQueues*. In the end, the resulting *Label* is returned.

The *query_is_critical* method is the policy that decides which query is potentially destructive, making it necessary to label that query immediately. The decision is based on the simplified *LstmQueryType* introduced in Section 7.2.1. Queries of type *StructuralDeletion*, *DataCreation*, *DataDeletion*, *DataMassDeletion*, *DataUpdate*, and *Administrative* are considered critical.

Listing 7.6: The classifier's main procedure to handle queries

```cpp
Label handle(const DatabaseQuery& query)
{
    Label result = Label::Benign;
    switch_lock->lock();
    for (auto & net : nets) {
        if (&net == active_net && query_is_critical(query)) {
            result = net.handle_now(query);
        } else {
            net.handle_async(query);
        }
    }
    switch_lock->unlock();
    return result;
}
```

The last piece of the *LstmClassifier* is the thread that manages the active states of the *NeuralNetQueue*s. Its code differs slightly from the algorithm we showed in Algorithm 2. Instead of maintaining the index of the active queue into the array that stores all queues, the implementation maintains a pointer to the active *NeuralNetQueue*. This slight change makes identifying the active queue independent from the data structure that stores the queues. Switching queues is also protected by the `switch_lock` *mutex* already shown in Listing 7.6. The lock is necessary to eliminate the possibility of the active queue changing while a query is being processed.

7.3. Data Preparation

In this section, we detail the transformation of the test data from the original DIMAQS project into formats that suit our purpose of use as training data. The data formats have to be human- and machine-readable to enable manual intervention if necessary. We also aim to design formats that are easy to expand and as generalized as possible. This allows future works to include new attack sequences.

There are two principal data types, namely traces of benign query sequences and definitions of malicious query sequences. For the benign samples, we only need to store a sequence of raw SQL queries. There is no additional structure to the data. Therefore, a simple storage format will suffice. In contrast, the malicious sequences consist of interdependent queries that can have a variable execution order. We need to consider these dependencies in our storage format to accurately model the attack sequences.

The following subsections will describe the source and destination data format for each datatype, as well as the steps required for transforming the data. To implement them, we chose the *Rust* programming language [52]. It allows for easy handling of mixed text and binary data with a string-like interface through the *bstr* library [53]. This is necessary, as the benign traces contain some binary data inserts.

7.3.1. Benign Query Traces

Benign query sequences were originally gathered by exporting traces of database accesses from two applications [47]: *MediaWiki* and *Bibspace*. Both traces are stored in multiple files with the following layout: Each query is preceded by a semicolon followed by a space.

Queries can span across multiple lines and contain comments or binary data. A query always ends with a newline character. The last line of each file only contains a single semicolon.

The *Bibspace* trace contains 58,112 queries across across 40 files, one for each day the software was observed. The *MediaWiki* trace only consists of 3 files, however, they are substantially larger and total 2,742,570 queries.

The target format is a single file for each trace. Every query should start with a new line and be concluded with a semicolon. Unnecessary line breaks are eliminated, only line breaks present in inserted data remain. This is accomplished by ignoring newline characters in between quotation marks. Comments are also eliminated from the dataset, as they may contain ransomware keywords that would falsely match against the regular expressions to detect them. Multi-line comments are enclosed in ”/* ... */”, making them easy to find and remove. We also checked for single-line comments manually, but found none.

The challenge when processing the benign query traces is separating them into individual queries and removing any comments, as they may influence the keyword-matching regular expressions. For each file, we keep six variables:

- ***open_cite_double***
  This variable is true when we encountered an uneven number of double quotes (*”*). This means we are potentially between two quotes.
- ***open_cite_single***
  This variable is true when we encountered an uneven number of single quotes (*’*). This means we are potentially between two quotes.
- ***opening_comment***
  The *opening_comment* variable is true if the last byte we read was a forward slash (*/*). Comments are initiated with */**, this variable indicates that the next byte may open a comment.
- ***closing_comment***
  The *closing_comment* variable is true if the last byte we read was an asterisk (*). Comments are closed with */, this variable indicates that the next byte may close a comment.
- ***comment_open***
  This variable indicates that the current byte is part of a comment.
- ***escaping***
  The *escaping* variable indicates that the last byte we encountered was a backslash (\). This means the next character cannot be part of a control sequence that starts or ends a quote.

Every file is processed byte-by-byte. The algorithm shown in Algorithm 4 builds a list of SQL commands without comments from each original file, while ensuring that everything between quotation marks is kept as-is. Persisting the variables listed above across lines enables us to filter out multi-line comments and detect quotes that span multiple lines. If we encounter a semicolon outside of any quotation marks or comments, we assume it marks the end of the SQL command. We then trim any whitespace from the start and end of the current query and add it to the list of commands. In the end, we write the list of queries to a file, starting each query with a new line.

Algorithm 4 Algorithm for eliminating comments from query traces

```
Require: bytes: the query trace as byte array
sql_query ← empty string
all_queries ← empty list of strings
comment_open ← false
opening_comment ← false
closing_comment ← false
escaping ← false
cite_open_single ← false
cite_open_double ← false
while bytes.hasNext() do
    byte ← bytes.next()
    if comment_open then
        if closing_comment AND byte == 0x2F then
            comment_open ← false
            opening_comment ← false
            closing_comment ← false
            continue
        else
            closing_comment ← byte == 0x2A
            continue
    else
        if !escaping AND byte == 0x22 then
            cite_open_double ← !cite_open_double
        if !escaping AND byte == 0x27 then
            cite_open_single ← !cite_open_single
        escaping ← byte == 0x5C
        if cite_open_double OR cite_open_single then
            sql_query.append(byte)
            continue
        else
            if byte == 0x3B then
                sql_query.append(byte)
                sql_query.trim()
                all_queries.append(sql_query)
                sql_query ← empty string
                continue
            if byte == 0xA then
                continue
            if opening_comment AND byte == 0x2A then
                comment_open ← true
                opening_comment ← false
                closing_comment ← false
                continue
            else
                if opening_comment AND byte != 0x2A then
                    sql_query.append(/)
                opening_comment ← byte == 0x2F
                if opening_comment then
                    continue
                else
                    sql_query.append(byte)
                    continue
return all_queries
```

7.3.2. Malicious Query Sequences

The malicious training samples are generated by parsing a list of malicious table names and ransom messages created by Jobst in [47]. Combined with the information on valid attack sequences we compiled in Section 4.4.2, we create an object that can be serialized into the JavaScript Object Notation (JSON) structure we introduced in Section 7.3.2 for every possible query sequence. The serialization is done by the serde library [54] and committed to a separate file for every object. Listing 7.7 shows the structure of these JSON files.

Listing 7.7: JSON structure for malicious training samples

```
{
    "prerequisites": [ String ],
    "alert_on": {
        "prerequisites": [ String ],
        "act_on": [ String ]
    },
    "backup_on": {
        "prerequisites": [ String ],
        "act_on": [ String ]
    }
}
```

The first first set of *prerequisites* is an array of SQL queries. These are general prerequisites for an attack, the entirety of which have to be issued to the system in-order. The *alert_on* and *backup_on* objects model actionable sequences. Their *prerequisites* have to be executed in addition to the general prerequisites, likewise entirely and in order. Each query from their *act_on* array must trigger an appropriate reaction when it is encountered by the DIMAQS plugin. For *alert_on*, this reaction is notifying the system administrator, for *backup_on*, it has to trigger a backup of the affected data and notify the system administrator.

This structure models our attack scenario perfectly while being generalized enough to be used on other attacks. Storing each attack sequence in a separate file gives us the ability to easily add new attack sequences to our dataset. The JSON file format can also be read and written by most programming languages, making the attack definitions highly portable. This enables future work to export attack definitions from almost any source. The human-readable nature of the JSON format allows us to easily inspect and verify the generated sequences, removing a possible source of errors.

For our attack scenario, as described in Section 4.4, the general *prerequisites* array contains a single query for information about the database structure. The *alert_on prerequisites* contain all the necessary queries to create a database and table to insert the ransom message in. *act_on* holds an array of data insertion queries, one for each observed ransom message. The *backup_on prerequisites* are empty, the *act_on* array contains commands to delete either a database or database table.

7.4. Training Executable

Training the model for the neural network is completely separate from the plugin's operation. Therefore, the code for training is not included in the plugin, but rather in a separate executable that is only used to create the model. As the training executable and

Listing 7.8: Structure for all training samples

```
        struct TrainingData {
            list<MaliciousSequence> Malicious;
            list<list<string>> Benign;
        };
```

plugin share the code for the neural network detailed in Section 7.2.2, this component is also implemented in the C++ programming language.

The components necessary to train, evaluate and tune a neural network are data ingest, feature extraction, sequence generation, the training loop, model evaluation, and a special operation mode for tuning. The implementation details for each of these components are detailed in the following sections.

7.4.1. Data Ingest

The benign and malicious data samples are stored in different formats. Accordingly, we built two different file parsers for them.

Each benign data sample is first read into a string. We then split that string at each occurrence of the substring "*;\n*". That way, we separate the individual SQL query in the file. We re-add the semicolon to each of the individual strings resulting from the split and enter the strings into a list. This procedure is repeated for each benign data sample, resulting in a list of lists of SQL queries.

Malicious samples are stored as JSON files. The structure of these files is described in Section 7.3.2. Each file is opened and parsed into C++ `structs` using Jørgen Lind's *json_struct* library [55]. It was chosen because of its simple interface and because it is a single-header library, making it easy to integrate into the existing project. Reading and parsing each malicious sample results in another list.

The benign and malicious data samples are then stored as a single object that is shown in Listing 7.8.

7.4.2. Feature Extraction

In Section 6.3.2, we already discussed the design of the feature extraction component. The query's type and presence of keywords are both detected by matching regular expressions against the query. All of them are listed in Appendix A.1 and A.2. We first iterate over the regular expressions that identify the query type, before we construct the feature Tensor by iterating over the expressions that identify ransom attack keywords and combining both as shown in Listing 7.9

7.4.3. Query Sequences

As we discussed in Section 6.3.3, there are two components involved in query sequences: Sequence generation and the *StreamingQuerySequence*. In this chapter, we will talk about the implementation of both.

7.4.3.1. Sequence Generation

The *random_sequence* method of the sequence generation component creates a new *StreamingQuerySequence*. It is called with a pointer to the dataset, the desired length of the

Listing 7.9: Constructing a feature *Tensor* from query type and keyword presence

```
Tensor featurize(std::string query) {
    Tensor out = zeros({1, 1, 27});
    LstmQueryType type = get_query_type(query);

    out.index_put_({0, 0, type}, 1);

    int i = LstmQueryType::Administrative + 1;
    for (const regex &regex: ransom_regexs) {
        bool match = regex_match(query, regex);
        out.index_put_({0, 0, i}, match);
        i++;
    }
    return out;
}
```

sequence and the weights of the different sequence types. The last parameter is a boolean that, if set, leads to the length of the generated sequences being randomized. In this case, the provided length parameter is treated as the maximum length of the sequence. The minimum length is 20 queries, to make sure an attack fits in the sequence. Under the following headings, we will discuss four aspects of generating these random query sequences: Sequence type creation, adding malicious query sequences, dirtying sequences, and random number generation.

Sequence Type Creation:
In Section 6.3.3.1, we defined three sequence types, *benign*, *alert*, and *attack*. Each type comes with the possibility of being *dirtied*, our term for adding parts of the attack samples to a sequence that should still result in a *Benign* classification.

To decide which kind of sequence to generate, we fetch a random number between zero and the sum of all weights. The weights for each sequence dictate ranges for the random number to be in, to result in that sequence. The table below shows an example for the accepted ranges of a random number r for each query type:

Sequence Type	Weight	Range
Benign	1	$r = 0$
Benign Dirty	5	$0 < r \leqslant 5$
Alert	1	$5 < r \leqslant 6$
Alert Dirty	3	$6 < r \leqslant 9$
Backup	1	$9 < r \leqslant 10$
Backup Dirty	3	$10 < r \leqslant 13$

Each sequence type is based on a benign sequence. This base is generated by drawing a random slice out of a randomly chosen benign data sample. The length of the slice can either be exactly the length provided by the function parameter or random with a minimum length of 20 and a maximum length equal to the length provided by the parameter.

To generate a dirtied benign sequence, we create dirty inserts that may be positioned at any point in the sequence. We proceed similarly for alert and backup sequences, inserting the attack at any point in the sequence. To generate dirtied alert and backup sequences however, we need to ensure that the dirty inserts are placed before the attack sequence.

Listing 7.10: Object holding all information about an inserted attack

```
struct ActionableInserts {
    vector<string> Prerequisites;
    vector<string> ActionablePrerequisites;
    string Actionable;
    vector<uint64_t> PrerequisitePositions;
    vector<uint64_t> ActionablePrereqPositions;
    vector<uint64_t> ActionablePositions;
};
```

If this is not ensured, actionable queries from the dirty inserts can cause a false positive intrusion detection, as the prerequisites are also present from the inserted attack. Therefore, we chose a random pivot, before which we insert the dirtying queries and after which we insert the attack.

Adding Malicious Query Sequences:
The method *make_action_inserts* is responsible for generating query sequences that will result in an *Alert* or *Backup* classification. This method requires a pointer to the training samples, the range in which to insert the sequence, and a boolean that switches between the *Alert* and *Backup* sequence types.

We can insert three kinds of queries into the sequences: General prerequisites, actionable prerequisites and the actionable queries that will raise an alert or cause a backup. At the start of the procedure, we will define random position ranges for each kind. We then generate positions for each query of each kind within its respective range and chose a random actionable. The queries and positions are then compiled into an *ActionableInserts* object and returned. The definition of the *ActionableInserts* object is shown in Listing 7.10.

Dirtying Sequences:
The *make_dirty_inserts* method takes a pointer to the dataset and an index before which the dirtying queries must be inserted. It then produces a list of actionables and prerequisite queries, along with lists of indices where each query should be inserted into the sequence. The dirty inserts come in 28 different types:

- Type 1: Include general prerequisites
- Type 2: Include alert prerequisites
- Type 3: Include backup prerequisites
- Type 4: Include general and alert prerequisites
- Type 5: Include general and backup prerequisites
- Type 6: Include alert and backup prerequisites
- Type 7: Include general, alert, and backup prerequisites
- Types 8-14: Include alert actionables before one of Type 1-7
- Types 15-21: Include backup actionables before one of Type 1-7
- Types 22-28: Include alert and backup actionables before one of Type 1-7

We first randomly chose an attack sample to draw queries from. To decide between the types, we draw a random number and match it to the scenario. According to the scenario, Listing 7.11: Object holding all information about dirty inserts

```
struct DirtyInserts {
    vector<string> Actionables;
    vector<string> Prerequisites;
    vector<uint64_t> ActionablePositions;
    vector<uint64_t> PrerequisitePositions;
};
```

we then build lists of prerequisite and actionable queries. We now have the choice of flipping the order of actionables and prerequisites, if certain criteria are met. For scenarios 8, 9, 12, and 13, the list of alert prerequisites in the attack cannot be empty, for scenarios 15, 16, 17, 18, and 20, the backup prerequisites cannot be empty. We can also flip in scenario 22 if both lists are not empty. Each of these criteria identifies a scenario where general or actionable prerequisites are required to raise an alert, but are missing from the dirty insertables. If any of these criteria are met, we flip with a 50% probability.

With the order of queries decided, we decide randomly how many queries of each category (prerequisite or actionable) to insert and generate positions for them. These are then compiled into the data object shown in Listing 7.11 and returned.

Random Number Generation:
For the purpose of training the network with randomized sequences, the random numbers we generate do not have to be of high quality. We chose the Mersenne Twister 19937 generator because of its speed. The *random_number* function takes an inclusive minimum value and an exclusive maximum value. If it is not already initialized, it creates the random number generator and seeds it with a random number obtained from a hardware random number generator.

To get the output number, we create a uniform integer distribution with the desired range and draw a random number from it. This number is then returned.

7.4.3.2. Streaming Query Sequence

The *StreamingQuerySequence* implements a query sequence with a memory footprint that is independent of the sequence length. To accomplish this, the *StreamingQuerySequence* only holds a reference to the benign query sequence, along with an index where the sequence starts and the size. If an attack or a dirtying query sequence is to be inserted, the *StreamingQuerySequence* holds those queries and their positions in the sequence.

The interface to the sequence only has four methods: *hasNext*, *next*, *compact*, and *size*.

hasNext simply checks if the sequence still has queries to return, *size* returns the total length of the sequence. The method called *next* returns one query as a *SingleQuery* object shown in Listing 7.12. To create it, the *featurize* method detailed in Section 7.4.2 is called. The target label is also encoded as a *Tensor*, so that classifier output and desired output can be directly compared.

The *compact* method enables batch processing of query sequences. It creates two output tensors, one containing the feature vectors of all queries left in the sequence, one containing all target classifier outputs. These tensors can be used directly by the neural network component to speed up training. To create these tensors, the procedure loops while the sequence can still produce queries. In every iteration, feature and target tensors are created by the *next* method. These are then accumulated in the output tensors. After calling the *compact* method, the sequence is consumed and will no longer return any queries.

Listing 7.12: Representation of a single query after feature extraction with the targeted classifier output

```
struct SingleQuery {
    Tensor features;
    Tensor target;
};
```

7.4.4. Training Loop

The training loop is the part of the program that trains the model for the classifier. It is where the training executable spends the most time. During testing, we found that compacting larger query sequences takes a considerable amount of time. Therefore, we decided to implement the sequence generation and feature extraction for the training loop out-of-band. Sequences are generated by four worker threads that also perform the feature extraction. The resulting feature and target tensors are then entered into a queue that is processed by the main training loop. To keep the memory footprint low, we only buffer four prepared sequences in the queue.

Filling the queue with prepared sequences involves seven steps:

1. Call *random_sequence* to get a new *StreamingQuerySequence*
2. Call the new sequence's *compact* method to get features and target classifications for all queries in the sequence
3. Acquire a write lock for the queue
4. Check the total number of created sequences, release the lock and return if no more sequences are necessary
5. Wait until the queue holds less than 4 items
6. Enqueue the new sequence
7. Release the lock

The *train_model* method is the entry point to training the network. It requires a pointer to a neural network, the number of sequences to train on, the weights for each sequence type, and the initial learning rate. The method first creates four threads that run the steps above in a loop. On the main thread, we then enter another loop that performs the following steps:

1. Reset the neural network
2. Get a new item from the queue
3. Run the feature tensor through the classifier, yielding a *Tensor* that contains an output for each query
4. Calculate the loss from the target and classifier's label
5. Optimize the neural network

This loop repeats for every sequence we generate. In the end, we join the worker threads that created the sequences and return.

7.4.5. Model Evaluation

Evaluating a trained model works by generating multiple query sequences and classifying each query. By comparing the label generated by the classifier to the known true label, we can generate some statistical information. The Algorithm for this was shown in Algorithm 3. The statistical information we gather for every query class includes total occurrences p, true-positive classifications tp, and false-positive classifications fp. We also record the total number of queries and the time the classifier took to classify them. This was done to get a first impression of the classifier's performance.

With the gathered statistical information, we can compute the metrics we described in Section 6.3.5. The configuration used when the model was trained, the statistical informatio,n and computed metrics are then printed to the screen.

7.4.6. Tuning

The tuning process is not involved in the regular operations of the DIMAQS plugin or the training process. We use it to test different configurations of the various variable parameters involved in training the model. We do this to find the configuration that performs best for our attack scenario. The parameters we can adjust during this phase are:

- **Neural network: Number of LSTM layers**
  As intrusion detection is not a very complex problem, we expect to get good results for 1 to 3 consecutive LSTM layers in the neural network that processes the queries. This parameter applies to the neural network directly and needs to be mirrored in the training executable and DIMAQS plugin.
- **Neural network: Dropout between LSTM layers [0,1]**
  The dropout defines a probability with which the output of a neuron is ignored by the next layer. For LSTMs, this always means discarding information about the current query or past queries. Therefore, we expect to get the best results for dropout values near or at zero. This parameter also has to be mirrored in the training executable and plugin.
- **Training loop: Optimizer stepsize [0,1]**
  The stepsize defines the initial learning rate for the *Adam* optimizer. As the learning rate decays over time, we can start with a higher value. We estimate a stepsize of 0.3 to result in fast training times and accurate classification.
- **Training loop: Training time in number of iterations**
  The training time directly influences the accuracy of the model. The longer the training, the more accurate the model gets. We expect 600.000 iterations to be sufficient for our purposes.
- **Sequence generation: Weights of sequence types**
  There are 6 sequence types in total: benign, benign dirty, alert, alert dirty, backup, and backup dirty. Each of these types has a variable weight attached to it, that influences the ratio of types we generate. The weights in the table below are our starting point, as they balance benign and attack samples while oversampling the dirtied sequences. This is done to counter a high false-positive rate.

Benign	Benign Dirty	Alert	Alert Dirty	Backup	Backup Dirty
1	3	1	2	1	2
1	5	1	3	1	3

- **Sequence generation: Training sequence length**
  The training sequence length also influences the balance between benign and malicious queries. The number of malicious queries is not dependent on the sequence length. Therefore, longer sequences contain more benign queries. For training, sequence lengths of less than 1000 queries should be sufficient for our goals.

- **Sequence generation: Evaluation sequence length**
  The sequence length during evaluation speaks mainly to the applicability of the model in the real world. As we consider an attack window of 5 minutes, we estimate the maximum realistic sequence length to be around 600.000. This is based on the time the classifier takes to classify a single query, which is about 0.5ms. We want to evaluate our model with differing sequence lengths to ensure it is also applicable to less utilized DBMSs.

To find the best possible combination of these parameters, we enter a number of values for each variable into the training program. The program then trains and evaluates a model for every possible combination of these values, printing the configuration and evaluation results described in Section 6.3.5 to a *CSV* file. This information is then evaluated manually by us to find the best configuration.

The general steps to test a single configuration are:

1. Instantiate a neural network
2. Train the network with this configuration
3. Run the evaluation
4. Print the configuration and evaluation results to a file

To test many different combinations of parameters, we first create a work-queue containing every configuration we want to evaluate. This work queue is implemented as a simple list, along with an integer that provides the index of the next configuration to test. As we want to run many tests in parallel, the index is protected by a mutex. After the list is filled with configurations, we create multiple threads that grab configurations from the queue and evaluate them through the steps above.

The output is also protected with a mutex, to prevent two threads from writing to the file simultaneously. As output format, we chose a *CSV* file, as it is easy to create and can be manually evaluated.

8. Evaluation

In this chapter, we describe the metrics and methodology we used to evaluate the revised DIMAQS plugin in detail. When evaluating an IDS, two characteristics are of importance: Detection accuracy and performance impact. Accordingly, after describing the testbed in Section 8.1, Section 8.2 describes how we trained the classifier model, what metrics we used to determine its accuracy, and lists our results. The subsequent Section 8.3 shows the impact the plugin has on the DBMS's performance.

8.1. Testbed

All of the tests below were run on a machine equipped with an Intel Core i7-6700HQ processor [56] and 32GB of main memory.

The programs ran inside a Docker container and were compiled using *cmake* 3.18.4 [57] and *gcc* 10.2.0 [58]. The version of the *boost* library used was 1.74 [59], the MySQL server version was 5.7.28 [60]. The libTorch library used was version 1.7.0+cpu [49].

The dependencies and Dockerfiles to build and run the software are included with the provided source.

8.2. Detection Accuracy

In this section, we detail our methodology and metrics for training and evaluating the classifier model. In the end, we arrive at the final model used in our plugin and show its detection accuracy.

The metrics we use to judge the accuracy of a model are the weighted and macro f1-scores, as detailed in Section 6.3.5. The parameters we need to optimize to generate the best-possible model are already described in Section 7.4.6.

Testing a single combination of these parameters involves

1. **Read the dataset**
   Read the attack and benign samples from the disk
2. **Split the dataset**
   Split the malicious queries into separate sets for training and evaluating. This is a standard step in training machine-learning algorithms, as we do not want to evaluate

the classifier with data it has already seen during the training phase. As the attack samples are somewhat sorted, we first shuffle them. We use the first 2/3 of the samples for training and the remaining 1/3 for the evaluation. Splitting the benign samples is not necessary, as they only represent noise in the sequence and are not relevant for detecting attacks.

3. **Train the classifier**
   The classifier is trained with a fixed configuration and the training dataset.
4. **Evaluate the model**
   The trained model is evaluated using the evaluation dataset.
5. **Compute metrics and save**
   The metrics we use to judge the accuracy are computed and saved to a file, along with the configuration.

We ran these steps for 232 different configurations. Training and evaluating 8 configurations in parallel takes about 28h. For all 232 configurations, the procedure ran for almost 35 days. The complete results of all tuning runs can be found in Appendix B.1.

The graphs in Figure 8.1 show how different tuning parameters influence the model's accuracy. They show the mean macro f1-score and standard deviation for each parameter value. As we honed in on our final model, the number of tested configurations for each parameter value decreased, leading to smaller deviations from the mean accuracy. This reduces the significance of the standard deviation. We concentrated the analysis on the macro f1-score, as the weighted variant was very similar across all configurations. This was due to the dominance of benign queries in the long testing sequences.

Figure 8.1a shows a steady improvement of the model with longer training times. This is expected with machine learning algorithms. Figure 8.1b shows that the algorithm benefited from shorter training sequences. As shorter sequences contain fewer benign queries but the same amount of malicious ones, the shorter length balances the data more towards the malicious samples, improving detection rates.

Figure 8.1c illustrates the model accuracy when trained with different sequence type weights. Reducing the amount of benign sequences improves detection rate. As the detection rate for the *Alert* class was low, oversampling samples that produce this label improved the model significantly.

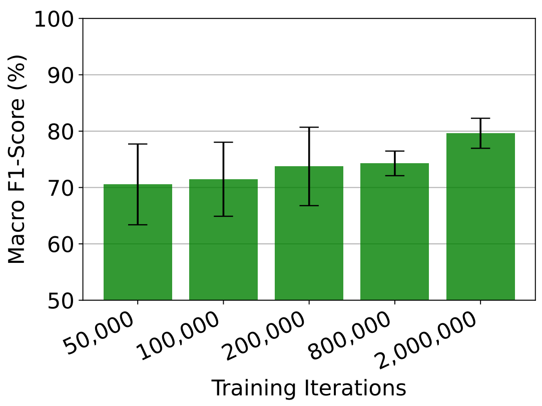
As Figure 8.1d illustrates, the step size had a negligible effect on the model's accuracy. Higher numbers of LSTM layers improved the detection rates slightly, as Figure 8.1e shows.

The dropout between LSTM layers also had little effect. The significant improvement with a dropout of 0.15 shown in Figure 8.1f is due to the fact that this parameter value was only tested with a sequence type balance of ”0 0 3 5 11 3”.

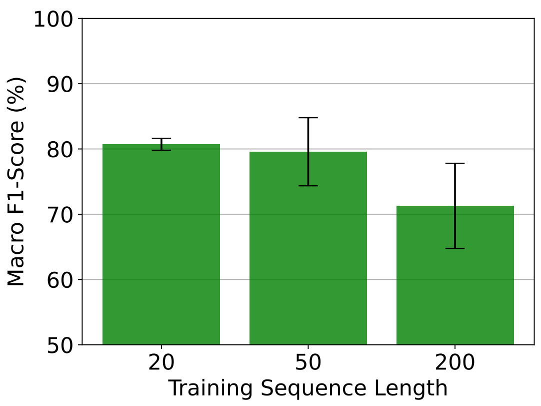
After comparing all tested configuration, we arrived at the simple configuration detailed in Table 8.1.

The resulting model is visualized in Figure 8.2. It produced the statistics shown in the confusion matrix in Table 8.2 and achieved the following scores:

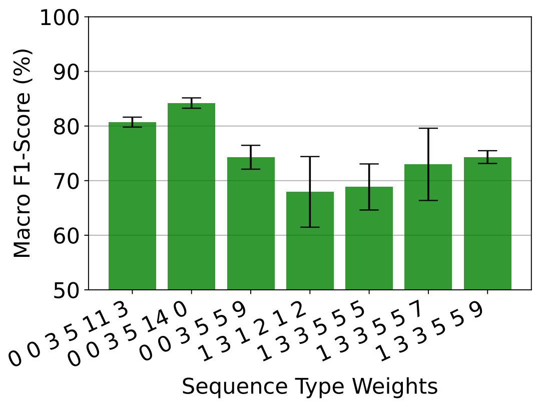
• **Benign f1-score:**	99.9663%
• **Alert f1-score:**	60.8626%
• **Backup f1-score:**	94.8704%
• **Macro f1-score:**	85.2331%



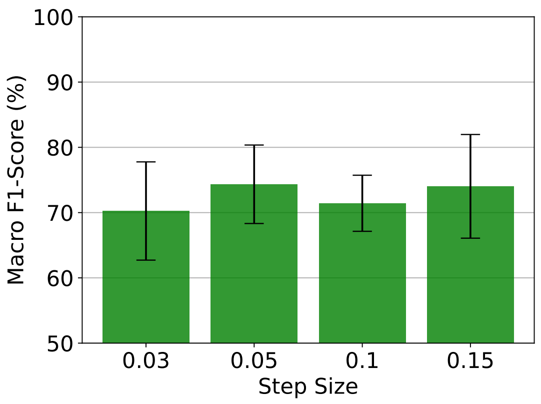
(a) Model accuracy at different training iterations



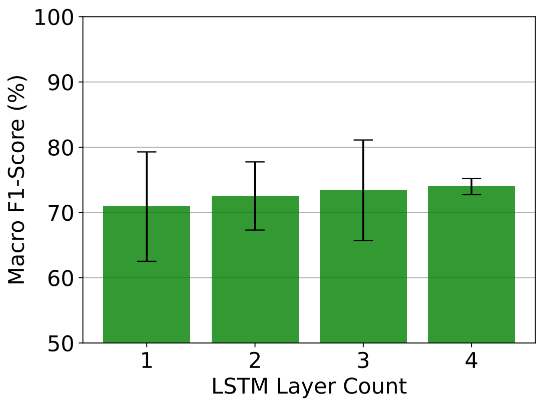
(b) Model Accuracy at different training sequence lengths



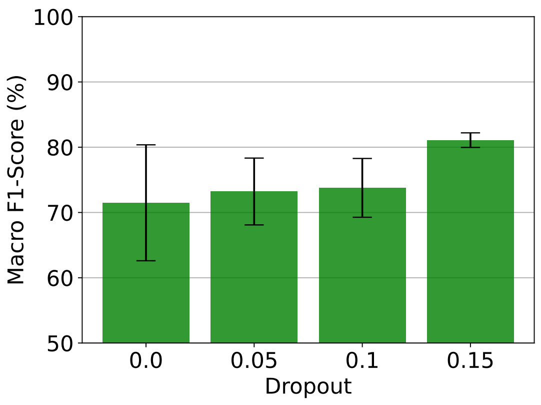
(c) Model accuracy at different sequence type weights



(d) Model Accuracy at Different Step Sizes



(e) Model accuracy at different LSTM layer counts



(f) Model accuracy at different dropouts

Figure 8.1.: Mean f1-score for different tuning parameters

Parameter	**Value**					
Training Iterations	200,000					
Training Sequence Length	50					
Number of LSTM Layers	1					
Dropout between LSTM Layers	not applicable					
Optimizer Step Size	0.15					
Sequence Type Weights	Benign 0	Benign Dirty 0	Alert 14	Alert Dirty 0	Backup 3	Backup Dirty 5
Evaluation Iterations	1000					
Evaluation Sequence Length	6000					

Table 8.1.: Configuration of the final model

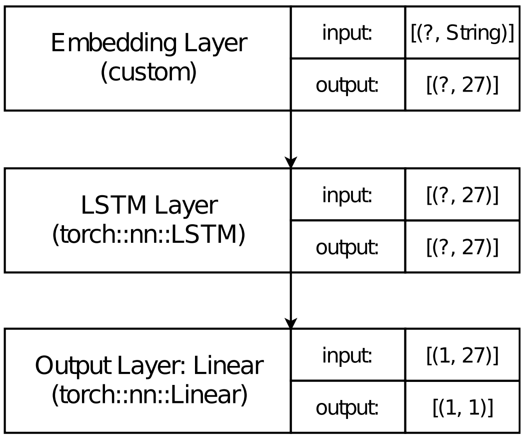


Figure 8.2.: Visual representation of the machine learning model

Predicted \ True	Benign	Alert	Backup
Benign	3050489	0	0
Alert	1960	1524	0
Backup	97	0	897

Table 8.2.: Confusion matrix of the final model

- **Weighted f1-score:** 99.9453%

These results show good detection rates when the attacker starts deleting data. We detected malicious deletions with close to 95% accuracy. However, the accuracy for detecting when the attacker leaves a ransom message is not sufficient. While the model detects every attack during the evaluation, it produces a very high false-positive rate for the alert class. This problem may be corrected by re-examining the keywords used to detect these actions or by investing more time into exploring different configurations. Unfortunately, time constraints did not allow us to train and evaluate more models.

8.3. Performance

In this section, we evaluate the revised plugin's impact on the performance of the MySQL server. To get an estimate of the real-world performance of our plugin, we used two benchmarks to cover two different scenarios: Sysbench [61] and TCP-H [62]. The former is an OLTP-focused benchmark, while TCP-H focuses on OLAP. These different scenarios give us insight into the performance of the DBMS in different situations. Together, they cover the bulk of database use cases.

We ran each of the benchmarks in three configurations: Vanilla MySQL, MySQL with the original DIMAQS plugin, and MySQL with the revised DIMAQS plugin. We ran each benchmark for each configuration 50 times to get reliable results. The complete results for all runs can be found in the Appendix Tables B.2 and B.3.

8.3.1. TPC-H

TPC-H is one of the standardized workloads provided by the Transaction Processing Performance Council (TPC). TPC-H consists of a binary to fill a database with sample data and an executable that creates SQL queries that emulate an OLAP workload. Online Analytical Processing processes large amounts of data to derive information from it. This type of workload generates a high load from very few queries. Therefore, we do not expect a significant performance impact from our plugin.

To set up the benchmark, we first generated data by running the `dbgen` utility with a scaling factor of 10. This resulted in approximately 10GB of data that is then imported into a MySQL database. The `qgen` executable is run to generate all 22 query types. These queries are saved to a file that can be piped into `mysql` to execute them.

Executing the benchmark is done by running the following bash script:

Listing 8.1: Code to run the TPC-H benchmark

```bash
start=‘date +%s%3N‘
for i in {1..22}; do
    mysql -u root -p[root password] -h [db host] -D [tpch db]
       < query-$i.sql &> /dev/null
done
end=‘date +%s%3N‘
echo $((end-start))
```

This script is run 51 times. The first run is discarded, as the database fill load required data into the main memory and warm-up caches the first time a query is executed. The runtimes of the subsequent 50 runs are logged to a file. Figure 8.3 shows the mean runtime of the benchmark in seconds for each configuration. The error bars represent the standard deviation. Compared to the vanilla MySQL server, the Petri-net-based classifier leads

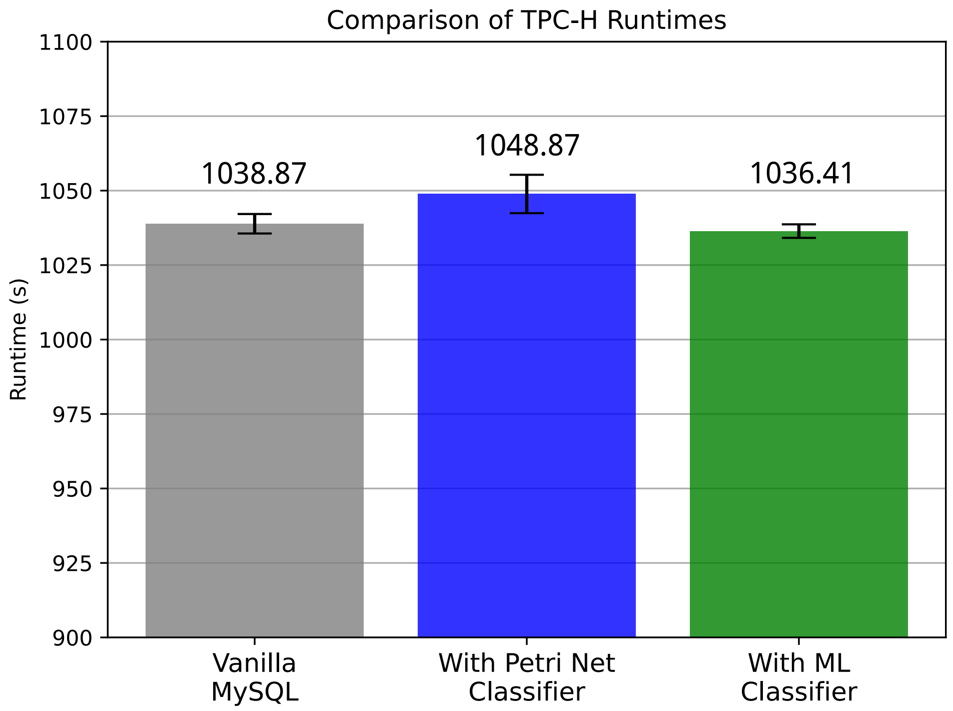


Figure 8.3.: Runtimes of the TPC-H benchmark (lower is better)

to a statistically significant performance degradation of 0.96%. The revised plugin performs better, with the mean runtime decreasing by 0.24% compared to the vanilla MySQL system. However, this change is not statistically significant.

These results are a strong indication, that the machine-learning-based classifier does not negatively impact the performance of a database system that mainly performs OLAP-related work.

8.3.2. Sysbench

Sysbench comes with a suite of benchmarks for SQL databases related to OLTP. OLTP workloads consist of many queries that perform smaller operations in the database, like adding or removing single rows of data. With the higher query frequency, the classifier is invoked more often, therefore, we expect the machine-learning-based classifier to incur a higher impact on the DBMS's performance. As the Petri-net-based classifier was only invoked for a small subset of queries, we expect it to perform better than the new classifier.

The Sysbench suite contains 8 benchmarks related to OLTP:

- **Delete**
  This benchmark will repeatedly execute only `DELETE` statements based on primary keys.
- **Insert**
  This benchmark will repeatedly execute only `INSERT` statements.
- **Point Select**
  This benchmark repeatedly executes `SELECT` statements based on a primary key.
- **Read-Only**
  This benchmark executes the following queries repeatedly in a row:

1. One `SELECT` statement based on a primary key.
2. One `SELECT` statement based on a range of primary keys.
3. One `SELECT SUM` statement that filters rows based on a range of primary keys and sums up the values in a column that is used in an index.
4. One `SELECT` statement that filters rows based on a range of primary keys and produces an ordered list of values contained in a column that is not used in an index.
5. One `SELECT DISTINCT` statement based on a range of primary keys.

- **Write-Only**
  This benchmark executes the following queries repeatedly in a row:
  1. One `UPDATE` statement that updates a column that is used in an index, the row to update is selected based on the primary key.
  2. One `UPDATE` statement that updates a column that is not used in an index, the row to update is selected based on the primary key.
  3. One `DELETE` statement that deletes a row based on the primary key.
  4. One `INSERT` statement that inserts a single row.
- **Read-Write**
  This benchmark executes the statements from the Read-Only benchmark, followed by the statements from the Write-Only benchmark repeatedly.
- **Update Index**
  This benchmark updates a column that is used in an index. It selects the row based on the primary key.
- **Update Non-Index**
  This benchmark updates a column that is not used in an index. It selects the row based on the primary key.

As with the TPC-H benchmark, we ran each of the Sysbench tests 51 times. Once to warm up caches and main memory, and 50 times to gather results. Sysbench runs each test in a loop for about a second and reports the completed transactions per second afterward. We will use this metric to estimate the DBMS's performance.

The reported transactions per second vary a lot between benchmark types, therefore, we opted to calculate the performance relative to the vanilla MySQL system for each type, as well as the standard deviation. The results can be seen in Figure 8.4.

The results show that the Petri net classifier does impact the performance slightly, but the decline is not statistically significant. Our results for the machine-learning classifier indicate that the performance impact of the new plugin is highly dependent on the workload. On average, the execution speed decreased by about 14%. However, the Delete, Point Select, Update Index and Update Non-Index benchmarks show almost no performance degradation. The Insert and Read-Only benchmarks lose almost 44% of their performance when the machine-learning classifier is enabled. The Write-Only benchmark degrades by almost 18% compared to the vanilla MySQL server. The performance losses are likely due to some false-positive alert classifications that cause additional overhead.

The Read-Write benchmark is a noteworthy case. Although it only executes the queries of the Read-Only and Write-Only benchmark, the benchmark's performance impact is lower than the impact of the individual benchmarks. This leads us to believe that MySQL can mask some of the performance impact when queries of different types are executed at the same time.

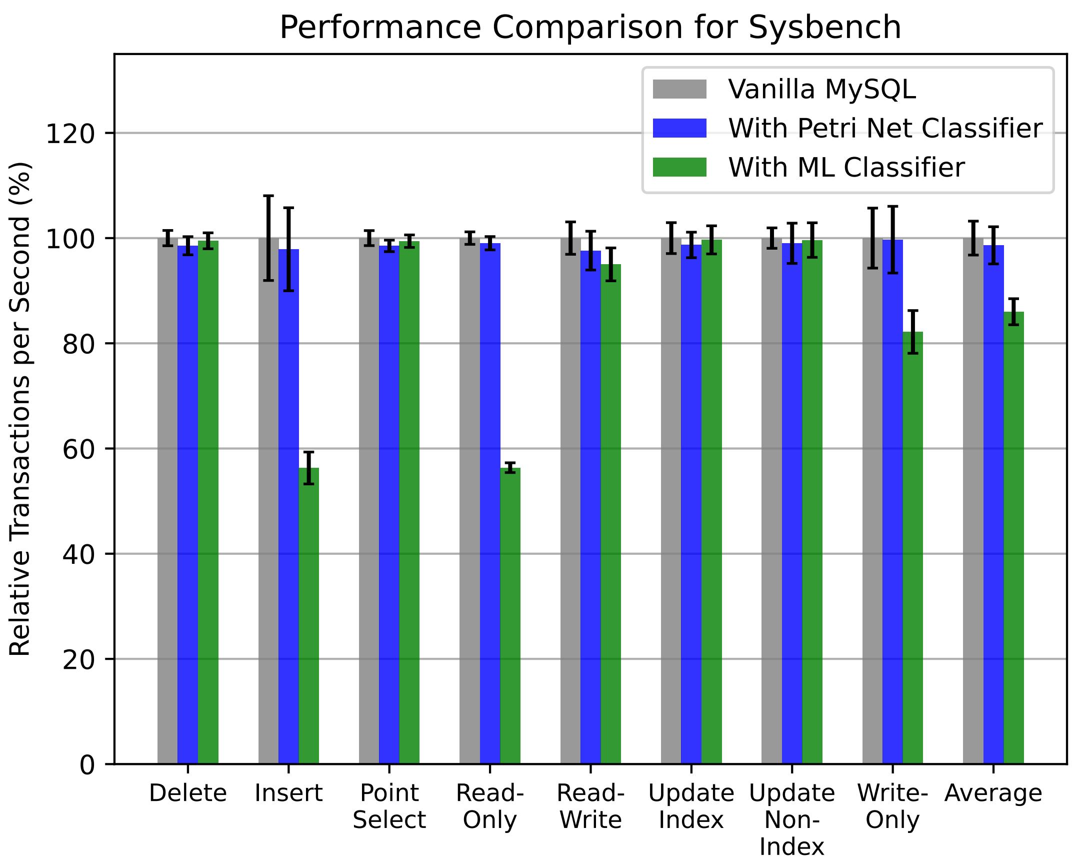


Figure 8.4.: Relative transactions per second for all Sysbench benchmarks (higher is better)

8.4. Summary

During the evaluation phase, we found the optimal configuration for training the classifier. With this configuration, we created a model that can detect malicious data deletions with almost 95% confidence and detect some attacks before any attempt to delete data occurs. These early alerts triggered many false positives, in a real-world scenario, we would likely disable them. The time between an attacker leaving a ransom message and the attacker deleting data is likely short, eliminating the possibility of manual intervention between alert and attempted deletion. For that reason, disabling the early alerts would likely not impact the security of our solution negatively.

We tested the performance of the plugin and compared it to a MySQL server without an intrusion detection system and to the performance of the original DIMAQS plugin with the Petri net classifier. We conclude that the performance impact of both plugins is minimal for OLAP workloads. In OLTP workloads, the Petri net classifier also caused almost no slow-downs. However, we saw some performance degradation with the machine-learning classifier in specific OLTP tasks. These were masked by the DBMS when multiple different query types were processed at the same time. While the average performance impact across all OLTP tests was about 14%, we estimate the real-world performance to be much better, as scenarios, where only one query type is being processed, are very unlikely.

9. Conclusion and Future Work

In this chapter, we conclude our work by summarizing our contributions and the results we archived In Section 9.1. The subsequent Section 9.2 will look into potential future works and improvements of the DIMAQS project.

9.1. Conclusion

The recently emerged server-side ransomware attacks are a growing threat to the security of database systems. These new attacks are more destructive than traditional ransomware, as attackers usually don't create a backup of the data before deleting it. This means, that even if the victim pays the ransom, the affected data can not be restored. Our research showed that only 11.3% of past victims had recent backups to mitigate the data loss caused by such an attack.

Previous work [47] addressed this issue by creating the DIMAQS (Dynamic Identification of Malicious Query Sequences) MySQL plugin to detect intrusions in the DBMS. The original plugin uses a Petri net classifier to detect query sequences that are typical for ransomware attacks. However, this approach was limited to a specific attack sequence. Adding the ability to detect other attacks would require manually re-engineering a new Petri net after analyzing the attack.

In our work, we replaced the Petri net classifier with a new LSTM-based algorithm. This eliminates the manual effort of creating a new Petri net when a new attack emerges, as the machine-learning classifier can be easily re-trained with the new attack. We then trained and evaluated a model for the classifier. Our new plugin achieved an f1-score of 85.23%. However, the detection rate for malicious data deletions were much higher, at close to 95%. The performance degradation was nonexistent for OLAP workloads and under 14% on average for OLTP workloads.

In our research, we came across multiple works related to ransomware detection and database intrusion detection. However, we found nothing specifically on database ransomware prevention. To our knowledge, our revised MySQL plugin is the only approach that incorporates all of these technologies, making it the first of its kind.

9.2. Future Work

The developed classifier model is the key piece to our work. As it is currently, the detection accuracy is not optimal. Especially the detection of an intruder leaving a ransom message is still lacking. More time to optimize the model would be required to eliminate these shortcomings and arrive at a solution that is ready for widespread adoption.

Another future improvement is performance-related. The plugin we developed is a proof of concept. For use in a real-world scenario, it should be refined into a more performant and maintainable product.

Adapting the plugin to different DBMSs is also necessary to maximize its usefulness. There are many database systems in use today, protecting only one of them is not enough to stop database ransomware attacks.

The last improvement we want to propose concerns the project's long-term operation. To be able to constantly improve the classifier, a system that reports attacks is required. These new attacks would be included in the dataset and a new model can be trained and deployed. However, there are concerns about data privacy, as attacks are mixed with regular queries that may contain privileged information. To further improve this process, a federated learning approach could be employed. Eliminating the need for a central training and model distribution system and crowd-sourcing this process would create a system that constantly adapts to new attack scenarios, yet is cheap and easy to maintain.

Acronyms

DIMAQS	Dynamic Identification of Malicious Query Sequences
LSTM	Long Short-Term Memory
RNN	Recurrent Neural Network
DBMS	Database Management System
SQL	Structured Query Language
IDS	Intrusion Detection System
HTTP	HyperText Transfer Protocol
API	Application Programming Interface
OS	Operating System
COW	copy-on-write
UAC	User Account Control
TCP	Transmission Control Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
UML	Unified Modeling Language
MPMC	Multi-Producer, Multi-Consumer
OLTP	Online Transactional Processing
OLAP	Online Analytical Processing
TPC	Transaction Processing Performance Council

Bibliography

[1] D. Reinsel, J. Gantz, and J. Rydning, "The Digitization of the World From Edge to Core," Seagate, Tech. Rep., 2018. [Online]. Available: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf [Accessed: 2020-04-27]

[2] R. Richardson and M. M. North, "Ransomware: Evolution, Mitigation and Prevention," 2017. [Online]. Available: https://digitalcommons.kennesaw.edu/facpubs/4276 [Accessed: 2020-04-27]

[3] S. Mohurle and M. Patil, "A brief study of Wannacry Threat: Ransomware Attack 2017," *International Journal of Advanced Research in Computer Science*, vol. 8, no. 5, 2017.

[4] "Cyber attack hits 200,000 in at least 150 countries: Europol | Reuters." [Online]. Available: https://www.reuters.com/article/us-cyber-attack-europol/cyber-attack-hits-200000-in-at-least-150-countries-europol-idUSKCN18A0FX [Accessed: 2020-12-15]

[5] "WannaCry – the worm that just won't die – Naked Security." [Online]. Available: https://nakedsecurity.sophos.com/2019/09/18/wannacry-the-worm-that-just-wont-die/ [Accessed: 2020-12-15]

[6] S. Radu, "Financial Losses From Cybercrimes Rose in 2018, Group Says | Best Countries | US News," 2019. [Online]. Available: https://www.usnews.com/news/best-countries/articles/2019-07-12/financial-losses-from-cybercrimes-rose-in-2018-group-says [Accessed: 2020-06-29]

[7] "MongoDB Apocalypse: Professional Ransomware Group Gets Involved, Infections Reach 28K Servers." [Online]. Available: https://www.bleepingcomputer.com/news/security/mongodb-apocalypse-professional-ransomware-group-gets-involved-infections-reach-28k-servers/ [Accessed: 2020-12-15]

[8] C. Cimpanu, "MongoDB databases still being held for ransom, two years after attacks started," 2019. [Online]. Available: https://www.zdnet.com/article/mongodb-databases-still-being-held-for-ransom-two-years-after-attacks-started/ [Accessed: 2020-12-15]

[9] "Database Ransom Attacks Have Now Hit MySQL Servers." [Online]. Available: https://www.bleepingcomputer.com/news/security/database-ransom-attacks-have-now-hit-mysql-servers/ [Accessed: 2020-12-15]

[10] "South Korean hosting co. pays $1m ransom to end eight-day outage | The Register." [Online]. Available: https://www.theregister.com/2017/06/20/south_korean_webhost_nayana_pays_ransom/ [Accessed: 2020-12-15]

[11] L. Iffländer, A. Dmitrienko, C. Hagen, M. Jobst, and S. Kounev, “Hands Off my Database: Ransomware Detection in Databases through Dynamic Analysis of Query Sequences,” 2019. [Online]. Available: http://arxiv.org/abs/1907.06775 [Accessed: 2020-04-27]

[12] G. Edwards, “Machine Learning | An Introduction | Towards Data Science.” [Online]. Available: https://towardsdatascience.com/machine-learning-an-introduction-23b84d51e6d0 [Accessed: 2020-07-21]

[13] C. Olah, “Understanding LSTM Networks – colah’s blog.” [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ [Accessed: 2020-07-22]

[14] D. P. Kingma and J. Lei Ba, “Adam: A Method for Stochastic Optimization,” in *ICLR 2015*, 2015.

[15] S. A. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion Detection using Sequences of System Calls,” *Journal of Computer Security*, 1998.

[16] A. Patcha and J.-M. Park, “An overview of anomaly detection techniques: Existing solutions and latest technological trends,” *Computer Networks*, 2007.

[17] C. Y. Chung, M. Gertz, and K. Levitt, “DEMIDS: A Misuse Detection System for Database Systems,” *Working Conference on Integrity and Internal Control in Information Systems*, 1999.

[18] Y. Hu and B. Panda, “Identification of malicious transactions in database systems,” *Proceedings of the International Database Engineering and Applications Symposium, IDEAS*, 2003.

[19] ——, “A data mining approach for database intrusion detection,” *Proceedings of the ACM Symposium on Applied Computing*, 2004.

[20] A. Roichman and E. Gudes, “DIWeDa - Detecting Intrusions in Web Databases.” *Working Conference on Data and Applications Security*, 2008.

[21] W. Lup, J. Lee, and P. Teoh, “DIDAFIT: Detecting Intrusions in Databases Through Fingerprinting Transactions,” *ICEIS*, 2002.

[22] E. Bertino, A. Kamra, E. Terzi, and A. Vakali, “Intrusion detection in RBAC-administered databases,” *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 2005.

[23] F. Valeur, D. Mutz, and G. Vigna, “A Learning-Based Approach to the Detection of SQL Attacks,” *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2005.

[24] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel, and E. Kirda, “Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and SQL queries,” *Journal of Computer Security*, vol. 17, 2009.

[25] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler, “CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data,” *International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[26] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, “UNVEIL: A large-scale, automated approach to detecting ransomware,” in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kharaz [Accessed: 2020-04-27]

[27] A. O. Almashhadani, M. Kaiiali, S. Sezer, and P. O'Kane, "A Multi-Classifier Network-Based Crypto Ransomware Detection System: A Case Study of Locky Ransomware," *IEEE Access*, vol. 7, 2019.

[28] R. Moussaileb, N. Cuppens, J. L. Lanet, and H. Le Bouder, "Ransomware Network Traffic Analysis for Pre-encryption Alert," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12056 LNCS. Springer, nov 2020.

[29] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "ShieldFS: A self-healing, ransomware-aware file system," in *ACM International Conference Proceeding Series*, vol. 5-9-Decemb. New York, NY, USA: Association for Computing Machinery, dec 2016.

[30] A. Continella, P. Di Milano, A. Guagnelli, G. Zingaro, P. Di, M. Giulio, D. E. Pasquale, D. Milano, A. Barenghi, S. Zanero, and F. Maggi, "ShieldFS: The Last Word In Ransomware Resilient Filesystems," *Black Hat USA*, 2017.

[31] S. K. Shaukat and V. J. Ribeiro, "RansomWall: A layered defense system against cryptographic ransomware attacks using machine learning," in *2018 10th International Conference on Communication Systems and Networks, COMSNETS 2018*, vol. 2018-January. Institute of Electrical and Electronics Engineers Inc., mar 2018.

[32] S. Maniath, A. Ashok, P. Poornachandran, V. G. Sujadevi, A. U. Sankar, and S. Jan, "Deep learning LSTM based ransomware detection," in *2017 Recent Developments in Control, Automation and Power Engineering, RDCAPE 2017*. Institute of Electrical and Electronics Engineers Inc., may 2018.

[33] R. Agrawal, J. W. Stokes, K. Selvaraj, and M. Marinescu, "Attention in Recurrent Neural Networks for Ransomware Detection," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 2019-May. Institute of Electrical and Electronics Engineers Inc., may 2019.

[34] "Ransomware WannaCry befällt Rechner der Deutschen Bahn | heise online." [Online]. Available: https://www.heise.de/newsticker/meldung/Ransomware-WannaCry-befaellt-Rechner-der-Deutschen-Bahn-3713426.html [Accessed: 2020-12-18]

[35] "Ransomware Hits Dozens of Hospitals in an Unprecedented Wave | WIRED." [Online]. Available: https://www.wired.com/story/ransomware-hospitals-ryuk-trickbot/ [Accessed: 2020-12-18]

[36] "Cities that have been crippled by cyberattacks and their responses - Business Insider." [Online]. Available: https://www.businessinsider.com/cyberattacks-on-american-cities-responses-2020-1?r=DE&IR=T [Accessed: 2020-12-18]

[37] "'Ransomware' a Growing Threat to Small Businesses - WSJ." [Online]. Available: https://www.wsj.com/articles/ransomware-a-growing-threat-to-small-businesses-1429127403 [Accessed: 2020-12-18]

[38] "BinaryEdge." [Online]. Available: https://www.binaryedge.io/ [Accessed: 2020-12-18]

[39] "Database exposed in Iran." [Online]. Available: https://blog.binaryedge.io/2019/04/18/database-exposed-in-iran/ [Accessed: 2020-12-18]

[40] "The impact and cost of ransomware in 2020." [Online]. Available: https://betanews.com/2020/10/09/ransomware-in-2020/ [Accessed: 2020-12-18]

[41] “The compendium of database ransomware.” [Online]. Available: https://blog.binaryedge.io/2017/01/18/the-compendium-of-database-ransomware/ [Accessed: 2020-12-18]

[42] “Cyberattack On A Hospital Leads To The First Ransomware-Linked Death.” [Online]. Available: https://www.forbes.com/sites/leemathews/2020/09/17/ransomware-attack-hospital-leads-to-death/ [Accessed: 2020-12-18]

[43] “0.2 BTC Strikes Back, Now Attacking MySQL Databases | GuardiCore.” [Online]. Available: https://www.guardicore.com/2017/02/0-2-btc-strikes-back-now-attacking-mysql-databases/ [Accessed: 2020-12-21]

[44] “0.2 BTC Strikes Back, Now Attacking MySQL Databases | GuardiCore.” [Online]. Available: https://www.guardicore.com/2017/02/0-2-btc-strikes-back-now-attacking-mysql-databases/ [Accessed: 2020-12-21]

[45] “PLEASE_READ_ME Ransomware Attacks 85K MySQL Servers | Threatpost.” [Online]. Available: https://threatpost.com/please_read_me-ransomware-mysql-servers/162136/ [Accessed: 2021-04-22]

[46] “Hacker ransoms 23k MongoDB databases and threatens to contact GDPR authorities.” [Online]. Available: https://www.zdnet.com/article/hacker-ransoms-23k-mongodb-databases-and-threatens-to-contact-gdpr-authorities/ [Accessed: 2021-04-22]

[47] M. Jobst, “DIMAQS - Dynamic Identification of Malicious Query Sequences,” 2018.

[48] “MongoDB ransacking - Google Tables.” [Online]. Available: https://docs.google.com/spreadsheets/d/1QonE9oeMOQHVh8heFIyeqrjfKEViL0poLnY8mAakKhM/edit#gid=2122582863 [Accessed: 2020-12-18]

[49] “PyTorch.” [Online]. Available: https://pytorch.org/ [Accessed: 2021-02-02]

[50] C. Desrochers, “cameron314/concurrentqueue: A fast multi-producer, multi-consumer lock-free concurrent queue for C++11,” 2014. [Online]. Available: https://github.com/cameron314/concurrentqueue [Accessed: 2021-02-03]

[51] ——, “Detailed Design of a Lock-Free Queue,” 2014. [Online]. Available: https://moodycamel.com/blog/2014/detailed-design-of-a-lock-free-queue.htm [Accessed: 2021-02-03]

[52] “Rust Programming Language.” [Online]. Available: https://www.rust-lang.org/ [Accessed: 2021-02-03]

[53] “bstr - Rust Library.” [Online]. Available: https://docs.rs/bstr/0.2.14/bstr/ [Accessed: 2021-02-03]

[54] “serde - Rust Library.” [Online]. Available: https://docs.serde.rs/serde/ [Accessed: 2021-02-03]

[55] “Github: joergen/json_struct.” [Online]. Available: https://github.com/jorgen/json_struct/ [Accessed: 2021-02-04]

[56] “Intel® Core™ i7-6700HQ Processor (6M Cache, up to 3.50 GHz) Product Specifications.” [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/88967/intel-core-i7-6700hq-processor-6m-cache-up-to-3-50-ghz.html [Accessed: 2021-04-23]

[57] “CMake.” [Online]. Available: https://cmake.org/ [Accessed: 2021-04-23]

[58] “GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF).” [Online]. Available: https://gcc.gnu.org/ [Accessed: 2021-04-23]

[59] “Boost C++ Libraries.” [Online]. Available: https://www.boost.org/ [Accessed: 2021-04-23]

[60] “MySQL.” [Online]. Available: https://www.mysql.com/de/ [Accessed: 2021-04-23]

[61] “akopytov/sysbench: Scriptable database and system performance benchmark.” [Online]. Available: https://github.com/akopytov/sysbench [Accessed: 2021-04-20]

[62] “TPC-H Homepage.” [Online]. Available: http://www.tpc.org/tpch/ [Accessed: 2021-04-21]

Appendix

A. Regular Expressions

- **Structural Query**
  `(^\s*(show\s*databases|show\s+.*\s*tables|show\s*schemas|select\s+.*\s+from\s*information_schema\.).*)`
- **Structural Creation**
  `(^\s*create\s*(table|database|index|view).*)`
- **Structural Deletion**
  `(^\s*(alter\s*table.*)?drop\s*(database|index|view|table).*)`
- **Structural Update**
  `(^alter\s*table.*)`
- **Data Query**
  `(^\s*select\s*.*\s*from.*)`
- **Data Creation**
  `(^\s*insert\s*into\s.*)`
- **Data Deletion**
  `(^\s*delete(\s*\*)?\s*from.*)`
- **Data Mass Deletion**
  `(^\s*(truncate\s*table|delete\s*\*\s*from).*)`
- **Data Update**
  `(^\s*update\s*.*\s*set.*)`

Figure A.1.: Regular expressions that determine the query type

- `(.*\sbitcoin[,;\.:\s_-]*.*)`
- `(.*wallet.*)`
- `(.*btc.*)`
- `(.*pwned.*)`
- `(.*\s(address|adress).*)`
- `(.*restore.*)`
- `(.*(email|e-mail).*)`
- `(.*send.*)`
- `(.*\s-?\d+(,\d+)*(\.\d+(e\d+)?)?.*(bitcoin|btc).*)`
  Note: This regular expression matches against a bitcoin amount
- `(.*warning.*)`
- `(.*please.*read.*)`
- `(.*leia.*me.*)`
- `(.*read.*me.*)`
- `(.*contact.*me.*)`
- `(.*encrypted.*)`
- `(.*aviso.*)`
- `(i.*have.*your.*data.*)`

Figure A.2.: Regular expressions that determine the presence of keywords

B. Raw Results

Table B.1.: Configuration and results of all tuning runs

Training		**Testing**		**LSTM**			**Sequence**	**Statistics**							**Scores**				
iterations	**sequence len**	**iterations**	**sequence len**	**layers**	**dropout**	**step size**	**Type Balance**	**total queries**	**benign tp**	**benign fp**	**alert tp**	**alert fp**	**backup tp**	**backup fp**	**benign f1**	**alert f1**	**backup f1**	**macro f1**	**weighted f1**
50000	200	1000	6000	1	0	0.15	1 3 1 2 1 2	2977363	2975851	248	0	502	714	48	0.999874	-nan	0.841485	-nan	-nan
50000	200	1000	6000	1	0	0.03	1 3 1 2 1 2	3106179	3104646	231	0	539	761	2	0.999876	-nan	0.867236	-nan	-nan
50000	200	1000	6000	1	0	0.05	1 3 1 2 1 2	2965772	2964289	207	0	493	778	5	0.999882	-nan	0.881087	-nan	-nan
50000	200	1000	6000	1	0	0.1	1 3 1 2 1 2	3055569	3054153	244	0	477	694	1	0.999882	-nan	0.851012	-nan	-nan
100000	200	1000	6000	1	0	0.15	1 3 1 2 1 2	2886098	2884637	240	0	474	746	1	0.999876	-nan	0.865932	-nan	-nan
100000	200	1000	6000	1	0	0.1	1 3 1 2 1 2	2954122	2952539	269	0	505	808	1	0.999869	-nan	0.862787	-nan	-nan
100000	200	1000	6000	1	0	0.05	1 3 1 2 1 2	2970933	2969518	236	0	468	182	529	0.999881	-nan	0.392665	-nan	-nan
100000	200	1000	6000	1	0	0.03	1 3 1 2 1 2	3041973	3040420	315	0	536	130	572	0.999860	-nan	0.311751	-nan	-nan
50000	200	1000	6000	2	0	0.15	1 3 1 2 1 2	2947829	2945762	802	268	307	488	202	0.999812	0.311991	0.747320	0.686374	0.999618
50000	200	1000	6000	2	0.05	0.15	1 3 1 2 1 2	2986596	2983137	2151	466	103	719	20	0.999622	0.302401	0.908976	0.737000	0.999467
200000	200	1000	6000	1	0	0.15	1 3 1 2 1 2	3070732	3069210	242	0	513	766	1	0.999877	-nan	0.874929	-nan	-nan
200000	200	1000	6000	1	0	0.1	1 3 1 2 1 2	3094928	3093361	244	0	590	724	9	0.999865	-nan	0.853774	-nan	-nan
200000	200	1000	6000	1	0	0.05	1 3 1 2 1 2	3020710	3019257	241	0	488	719	5	0.999879	-nan	0.880049	-nan	-nan
200000	200	1000	6000	1	0	0.03	1 3 1 2 1 2	3067752	3066203	239	0	468	841	1	0.999885	-nan	0.875130	-nan	-nan
50000	200	1000	6000	2	0	0.1	1 3 1 2 1 2	3039342	3037666	442	33	470	730	1	0.999850	0.088472	0.861865	0.650062	0.999666
50000	200	1000	6000	2	0.1	0.15	1 3 1 2 1 2	3008636	3007192	249	0	512	679	4	0.999873	-nan	0.842955	-nan	-nan
50000	200	1000	6000	2	0.05	0.1	1 3 1 2 1 2	2943691	2940614	1772	396	139	759	11	0.999675	0.306146	0.916667	0.740829	0.999527
50000	200	1000	6000	2	0.1	0.1	1 3 1 2 1 2	3034967	3033503	226	0	479	635	124	0.999884	-nan	0.814625	-nan	-nan
50000	200	1000	6000	2	0	0.05	1 3 1 2 1 2	3070454	3068908	245	0	514	787	0	0.999876	-nan	0.875904	-nan	-nan
50000	200	1000	6000	2	0.05	0.05	1 3 1 2 1 2	2868097	2865076	1696	482	93	750	0	0.999688	0.380877	0.871080	0.750548	0.999530
50000	200	1000	6000	2	0	0.03	1 3 1 2 1 2	3112278	3109289	1725	352	202	699	11	0.999690	0.281375	0.902518	0.727861	0.999540
50000	200	1000	6000	2	0.1	0.05	1 3 1 2 1 2	3051731	3050214	254	0	477	670	116	0.999880	-nan	0.868438	-nan	-nan
50000	200	1000	6000	2	0.05	0.03	1 3 1 2 1 2	2965508	2964020	214	2	588	682	2	0.999865	0.006483	0.876044	0.627464	0.999639
50000	200	1000	6000	2	0.1	0.03	1 3 1 2 1 2	3027765	3023626	2880	382	118	751	8	0.999504	0.213289	0.884570	0.699121	0.999346
100000	200	1000	6000	2	0	0.15	1 3 1 2 1 2	2972447	2970642	536	67	428	764	10	0.999837	0.145022	0.888889	0.677916	0.999666
100000	200	1000	6000	2	0.05	0.15	1 3 1 2 1 2	2976077	2972381	2482	413	88	600	113	0.999568	0.247676	0.806994	0.684746	0.999395
100000	200	1000	6000	2	0	0.1	1 3 1 2 1 2	2960900	2959400	245	0	498	757	0	0.999874	-nan	0.860716	-nan	-nan
100000	200	1000	6000	2	0.1	0.15	1 3 1 2 1 2	3016283	3014919	253	0	459	453	199	0.999882	-nan	0.753117	-nan	-nan
100000	200	1000	6000	2	0.1	0.1	1 3 1 2 1 2	3051899	3050406	247	0	501	738	7	0.999877	-nan	0.858140	-nan	-nan
100000	200	1000	6000	2	0.05	0.1	1 3 1 2 1 2	2998079	2996597	249	0	447	785	1	0.999884	-nan	0.874165	-nan	-nan
100000	200	1000	6000	2	0	0.05	1 3 1 2 1 2	3017932	3016427	269	0	469	767	0	0.999878	-nan	0.880092	-nan	-nan
100000	200	1000	6000	2	0.05	0.05	1 3 1 2 1 2	3078360	3076869	241	0	495	751	4	0.999880	-nan	0.877336	-nan	-nan
100000	200	1000	6000	2	0	0.03	1 3 1 2 1 2	2996427	2990553	4657	30	470	714	3	0.999220	0.028626	0.286689	0.438178	0.998887
100000	200	1000	6000	2	0.1	0.05	1 3 1 2 1 2	3036083	3034524	284	0	512	763	0	0.999869	-nan	0.843094	-nan	-nan
100000	200	1000	6000	2	0.05	0.03	1 3 1 2 1 2	2995481	2994006	235	0	505	734	1	0.999876	-nan	0.872771	-nan	-nan
100000	200	1000	6000	2	0.1	0.03	1 3 1 2 1 2	3017084	3014962	824	129	388	776	5	0.999799	0.197097	0.901278	0.699391	0.999636
200000	200	1000	6000	2	0	0.15	1 3 1 2 1 2	3126018	3124834	0	0	490	0	694	0.999811	-nan	-nan	-nan	-nan
200000	200	1000	6000	2	0.05	0.15	1 3 1 2 1 2	3061503	3060183	0	0	520	0	800	0.999784	-nan	-nan	-nan	-nan
200000	200	1000	6000	2	0.1	0.15	1 3 1 2 1 2	3105335	3103833	221	0	520	536	225	0.999881	-nan	0.820827	-nan	-nan
200000	200	1000	6000	2	0	0.1	1 3 1 2 1 2	2977256	2976021	0	0	532	0	703	0.999793	-nan	-nan	-nan	-nan

200000	200	1000	6000	2	0.1	0.1	1 3 1 2 1 2	3040653	3039144	242	0	478	779	10	0.999882	-nan	0.864595	-nan	-nan
200000	200	1000	6000	2	0.05	0.1	1 3 1 2 1 2	3012995	3011800	0	0	484	0	711	0.999802	-nan	-nan	-nan	-nan
200000	200	1000	6000	2	0	0.05	1 3 1 2 1 2	3062127	3060117	714	270	265	707	54	0.999840	0.392727	0.848739	0.747102	0.999696
200000	200	1000	6000	2	0.05	0.05	1 3 1 2 1 2	3028495	3027025	220	0	518	521	211	0.999878	-nan	0.826984	-nan	-nan
200000	200	1000	6000	2	0	0.03	1 3 1 2 1 2	2959317	2956186	1888	314	205	549	175	0.999646	0.223726	0.804985	0.676119	0.999463
200000	200	1000	6000	2	0.1	0.05	1 3 1 2 1 2	3036765	3035090	475	32	514	527	127	0.999837	0.055411	0.873964	0.643071	0.999640
200000	200	1000	6000	2	0.05	0.03	1 3 1 2 1 2	3030731	3029115	366	68	437	708	37	0.999867	0.166463	0.878412	0.681581	0.999699
200000	200	1000	6000	2	0.1	0.03	1 3 1 2 1 2	2978869	2976805	735	164	374	784	7	0.999814	0.263242	0.884377	0.715811	0.999650
50000	200	1000	6000	3	0	0.15	1 3 1 2 1 2	3023479	3019211	3034	488	42	693	11	0.999491	0.248283	0.906475	0.718083	0.999338
50000	200	1000	6000	3	0.05	0.15	1 3 1 2 1 2	3049642	3048188	230	0	475	243	506	0.999884	-nan	0.475073	-nan	-nan
50000	200	1000	6000	3	0.1	0.15	1 3 1 2 1 2	2957509	2956041	207	0	550	706	5	0.999872	-nan	0.883605	-nan	-nan
50000	200	1000	6000	3	0	0.1	1 3 1 2 1 2	3064791	3061789	1745	214	274	768	1	0.999670	0.191928	0.875214	0.688937	0.999511
50000	200	1000	6000	3	0.05	0.1	1 3 1 2 1 2	2942141	2939001	1921	428	81	710	0	0.999660	0.322532	0.874384	0.732192	0.999512
50000	200	1000	6000	3	0.1	0.1	1 3 1 2 1 2	2997040	2992642	3131	446	65	756	0	0.999466	0.231568	0.864989	0.698674	0.999301
50000	200	1000	6000	3	0	0.05	1 3 1 2 1 2	2929755	2924315	4225	82	417	714	2	0.999274	0.051218	0.416205	0.488899	0.998970
50000	200	1000	6000	3	0.05	0.05	1 3 1 2 1 2	2934330	2930533	2565	311	154	743	24	0.999536	0.195536	0.877214	0.690762	0.999377
50000	200	1000	6000	3	0	0.03	1 3 1 2 1 2	2978372	2975812	1312	311	185	703	49	0.999749	0.321115	0.833926	0.718263	0.999594
50000	200	1000	6000	3	0.1	0.05	1 3 1 2 1 2	3010425	3008917	240	0	502	627	139	0.999877	-nan	0.848444	-nan	-nan
50000	200	1000	6000	3	0.05	0.03	1 3 1 2 1 2	2983912	2982356	291	31	521	622	91	0.999864	0.076449	0.835460	0.637258	0.999654
50000	200	1000	6000	3	0.1	0.03	1 3 1 2 1 2	3055146	3053669	220	0	535	679	43	0.999876	-nan	0.910798	-nan	-nan
100000	200	1000	6000	3	0	0.15	1 3 1 2 1 2	2937638	2936294	0	0	560	0	784	0.999771	-nan	-nan	-nan	-nan
100000	200	1000	6000	3	0.05	0.15	1 3 1 2 1 2	3103189	3101732	217	0	494	746	0	0.999885	-nan	0.874048	-nan	-nan
100000	200	1000	6000	3	0	0.1	1 3 1 2 1 2	2970267	2966673	2451	321	102	524	196	0.999570	0.198700	0.746439	0.648236	0.999394
100000	200	1000	6000	3	0.1	0.15	1 3 1 2 1 2	3067170	3065960	0	0	427	0	783	0.999803	-nan	-nan	-nan	-nan
100000	200	1000	6000	3	0.05	0.1	1 3 1 2 1 2	2934748	2928919	4545	484	11	788	1	0.999223	0.181409	0.892412	0.691015	0.999056
100000	200	1000	6000	3	0.1	0.1	1 3 1 2 1 2	2982389	2980881	232	0	439	837	0	0.999887	-nan	0.878279	-nan	-nan
100000	200	1000	6000	3	0	0.05	1 3 1 2 1 2	2937266	2936040	0	0	509	0	717	0.999791	-nan	-nan	-nan	-nan
100000	200	1000	6000	3	0.05	0.05	1 3 1 2 1 2	2989029	2987574	241	0	456	753	5	0.999883	-nan	0.873550	-nan	-nan
100000	200	1000	6000	3	0	0.03	1 3 1 2 1 2	3189261	3186442	1541	102	410	766	0	0.999694	0.105426	0.874429	0.659850	0.999520
100000	200	1000	6000	3	0.1	0.05	1 3 1 2 1 2	2973141	2971632	256	0	506	540	207	0.999872	-nan	0.834621	-nan	-nan
100000	200	1000	6000	3	0.05	0.03	1 3 1 2 1 2	3091259	3089385	564	55	484	736	35	0.999830	0.105566	0.887817	0.664404	0.999647
100000	200	1000	6000	3	0.1	0.03	1 3 1 2 1 2	3020367	3018489	659	274	209	733	3	0.999856	0.433544	0.903266	0.778889	0.999742
200000	200	1000	6000	3	0	0.15	1 3 1 2 1 2	3011410	3010192	0	0	469	0	749	0.999798	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0.05	0.15	1 3 1 2 1 2	3001251	2999968	0	0	557	0	726	0.999786	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0	0.1	1 3 1 2 1 2	2953658	2952420	0	0	481	0	757	0.999790	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0.1	0.15	1 3 1 2 1 2	2920945	2919191	454	111	425	764	0	0.999849	0.262722	0.856502	0.706358	0.999677
200000	200	1000	6000	3	0.1	0.1	1 3 1 2 1 2	3115252	3113844	214	5	458	551	180	0.999892	0.012165	0.833585	0.615214	0.999706
200000	200	1000	6000	3	0.05	0.1	1 3 1 2 1 2	2997232	2996052	0	0	508	0	672	0.999803	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0	0.05	1 3 1 2 1 2	3005958	3004682	0	0	521	0	755	0.999788	-nan	-nan	-nan	-nan
50000	200	1000	6000	1	0	0.15	1 3 3 5 5 7	2958768	2953284	3604	721	341	785	33	0.999329	0.276087	0.882518	0.719311	0.999037
200000	200	1000	6000	3	0.05	0.05	1 3 1 2 1 2	3117861	3112627	3949	525	12	747	1	0.999364	0.217571	0.888757	0.701897	0.999203
200000	200	1000	6000	3	0	0.03	1 3 1 2 1 2	2981904	2980191	414	60	493	736	10	0.999848	0.144578	0.871522	0.671983	0.999657
200000	200	1000	6000	3	0.1	0.05	1 3 1 2 1 2	2924340	2919388	3685	443	45	772	7	0.999362	0.198922	0.897674	0.698653	0.999201
50000	200	1000	6000	1	0	0.1	1 3 3 5 5 7	3022800	3017280	3559	943	228	790	0	0.999373	0.346500	0.872928	0.739600	0.999087

50000	200	1000	6000	1	0	0.05	1 3 3 5 5 7	2908270	2905979	270	0	1173	848	0	0.999752	-nan	0.864424	-nan	-nan
50000	200	1000	6000	1	0	0.03	1 3 3 5 5 7	2956545	2954294	274	3	1161	813	0	0.999757	0.005080	0.862142	0.622327	0.999328
200000	200	1000	6000	3	0.05	0.03	1 3 1 2 1 2	2956923	2955429	290	4	479	504	217	0.999870	0.008197	0.812248	0.606771	0.999662
200000	200	1000	6000	3	0.1	0.03	1 3 1 2 1 2	3024131	3021294	1549	359	149	772	8	0.999719	0.319679	0.892486	0.737295	0.999577
100000	200	1000	6000	1	0	0.15	1 3 3 5 5 7	3123592	3121337	236	0	1187	832	0	0.999772	-nan	0.875789	-nan	-nan
100000	200	1000	6000	1	0	0.1	1 3 3 5 5 7	2976795	2974567	247	0	1127	854	0	0.999769	-nan	0.873657	-nan	-nan
100000	200	1000	6000	1	0	0.05	1 3 3 5 5 7	3012698	3006163	4590	1063	33	849	0	0.999232	0.326775	0.874807	0.733605	0.998952
100000	200	1000	6000	1	0	0.03	1 3 3 5 5 7	3070092	3063778	4334	1138	54	788	0	0.999284	0.354021	0.870237	0.741181	0.999001
50000	200	1000	6000	2	0	0.15	1 3 3 5 5 7	3026348	3019575	4855	1078	0	840	0	0.999197	0.316593	0.893142	0.736311	0.998924
50000	200	1000	6000	2	0.05	0.15	1 3 3 5 5 7	3054954	3048353	4604	1021	65	911	0	0.999235	0.315026	0.888347	0.734203	0.998958
200000	200	1000	6000	1	0	0.15	1 3 3 5 5 7	3092226	3090035	247	0	1173	524	247	0.999770	-nan	0.806774	-nan	-nan
200000	200	1000	6000	1	0	0.1	1 3 3 5 5 7	2962292	2960143	226	0	1118	804	1	0.999773	-nan	0.878689	-nan	-nan
200000	200	1000	6000	1	0	0.05	1 3 3 5 5 7	2979274	2973131	4209	1021	77	835	1	0.999280	0.334370	0.882664	0.738771	0.999002
200000	200	1000	6000	1	0	0.03	1 3 3 5 5 7	3020533	3018233	362	25	1092	165	656	0.999759	0.023148	0.334686	0.452531	0.999217
50000	200	1000	6000	2	0.1	0.15	1 3 3 5 5 7	2939707	2937062	634	111	1034	865	1	0.999716	0.132458	0.889460	0.673878	0.999346
50000	200	1000	6000	2	0	0.1	1 3 3 5 5 7	2946780	2940909	3869	928	286	786	2	0.999339	0.316724	0.789157	0.701740	0.999002
50000	200	1000	6000	2	0.05	0.1	1 3 3 5 5 7	3011568	3005643	3958	1055	66	841	5	0.999331	0.354920	0.895157	0.749803	0.999062
50000	200	1000	6000	2	0.1	0.1	1 3 3 5 5 7	3110801	3104726	4045	1159	21	846	4	0.999345	0.373811	0.899522	0.757559	0.999081
50000	200	1000	6000	2	0	0.05	1 3 3 5 5 7	3045984	3039929	4016	1093	28	917	1	0.999335	0.360310	0.917459	0.759035	0.999075
50000	200	1000	6000	2	0.05	0.05	1 3 3 5 5 7	3017674	3010805	4853	1240	22	754	0	0.999191	0.348217	0.866169	0.737859	0.998886
50000	200	1000	6000	2	0	0.03	1 3 3 5 5 7	2968983	2963350	3696	1033	32	858	14	0.999372	0.366377	0.902208	0.755986	0.999116
50000	200	1000	6000	2	0.1	0.05	1 3 3 5 5 7	3033315	3026516	4824	1190	3	781	1	0.999203	0.340340	0.878515	0.739353	0.998913
50000	200	1000	6000	2	0.05	0.03	1 3 3 5 5 7	3052523	3045952	4633	1047	0	889	2	0.999240	0.319548	0.908998	0.742595	0.998981
50000	200	1000	6000	2	0.1	0.03	1 3 3 5 5 7	3021280	3015414	3831	970	215	809	41	0.999330	0.328981	0.904416	0.744242	0.999040
100000	200	1000	6000	2	0	0.15	1 3 3 5 5 7	3013111	3006323	4825	1063	75	819	6	0.999186	0.313893	0.860746	0.724608	0.998889
100000	200	1000	6000	2	0.05	0.15	1 3 3 5 5 7	3036781	3034623	220	0	1125	813	0	0.999778	-nan	0.880823	-nan	-nan
100000	200	1000	6000	2	0	0.1	1 3 3 5 5 7	2999245	2992526	4699	1102	61	840	17	0.999205	0.326229	0.874089	0.733174	0.998909
100000	200	1000	6000	2	0.1	0.15	1 3 3 5 5 7	3014320	3009655	2639	1123	80	820	3	0.999548	0.467722	0.906578	0.791283	0.999311
100000	200	1000	6000	2	0.05	0.1	1 3 3 5 5 7	3003889	2999534	2380	813	318	690	154	0.999550	0.372594	0.837887	0.736677	0.999269
100000	200	1000	6000	2	0.1	0.1	1 3 3 5 5 7	2956942	2950571	4307	1163	51	850	0	0.999262	0.360676	0.878553	0.746164	0.998965
100000	200	1000	6000	2	0	0.05	1 3 3 5 5 7	2959310	2953967	3415	883	198	818	29	0.999389	0.338249	0.883369	0.740336	0.999114
100000	200	1000	6000	2	0	0.03	1 3 3 5 5 7	2978233	2972238	4002	1121	17	855	0	0.999324	0.368629	0.905241	0.757731	0.999056
100000	200	1000	6000	2	0.05	0.05	1 3 3 5 5 7	3043015	3036281	4798	1147	11	774	4	0.999209	0.331359	0.891705	0.740758	0.998927
100000	200	1000	6000	2	0.1	0.05	1 3 3 5 5 7	3040570	3034524	4048	1148	28	822	0	0.999329	0.370741	0.901810	0.757293	0.999059
100000	200	1000	6000	2	0.05	0.03	1 3 3 5 5 7	3091479	3085970	3471	1152	38	841	7	0.999432	0.410695	0.884797	0.764975	0.999174
100000	200	1000	6000	2	0.1	0.03	1 3 3 5 5 7	2952414	2946763	3720	1091	14	810	16	0.999367	0.381802	0.874258	0.751809	0.999101
200000	200	1000	6000	2	0	0.15	1 3 3 5 5 7	3058094	3056115	0	0	1116	0	863	0.999676	-nan	-nan	-nan	-nan
200000	200	1000	6000	2	0	0.1	1 3 3 5 5 7	3126642	3124396	249	0	1160	837	0	0.999775	-nan	0.871421	-nan	-nan
200000	200	1000	6000	2	0.05	0.15	1 3 3 5 5 7	3041969	3040041	0	0	1136	0	792	0.999683	-nan	-nan	-nan	-nan
200000	200	1000	6000	2	0.1	0.15	1 3 3 5 5 7	2939627	2933159	4567	1047	38	0	816	0.999215	0.278717	-nan	-nan	-nan
200000	200	1000	6000	2	0.05	0.1	1 3 3 5 5 7	3062997	3057673	3310	1048	81	880	5	0.999446	0.393837	0.909561	0.767614	0.999197
200000	200	1000	6000	2	0.1	0.1	1 3 3 5 5 7	3147109	3144892	221	0	1231	763	2	0.999769	-nan	0.890835	-nan	-nan
200000	200	1000	6000	2	0	0.05	1 3 3 5 5 7	3061983	3058300	1689	856	307	831	0	0.999674	0.492095	0.878900	0.790223	0.999448
200000	200	1000	6000	2	0	0.03	1 3 3 5 5 7	3015779	3010284	3556	1043	74	820	2	0.999397	0.377625	0.893246	0.756756	0.999138

200000	200	1000	6000	2	0.05	0.05	1 3 3 5 5 7	2998270	2991676	4646	1102	1	550	295	0.999224	0.310817	0.758621	0.689554	0.998903
200000	200	1000	6000	2	0.1	0.05	1 3 3 5 5 7	3098828	3092429	4386	1116	53	694	150	0.999283	0.331551	0.853104	0.727979	0.998991
200000	200	1000	6000	2	0.05	0.03	1 3 3 5 5 7	3007664	3002049	3669	1023	83	839	1	0.999375	0.365357	0.893504	0.752745	0.999113
200000	200	1000	6000	2	0.1	0.03	1 3 3 5 5 7	3069001	3064153	2818	979	201	844	6	0.999507	0.407917	0.900747	0.769390	0.999252
50000	200	1000	6000	3	0	0.15	1 3 3 5 5 7	3048207	3044561	1631	170	1046	6	793	0.999561	0.089262	0.014888	0.367904	0.998939
50000	200	1000	6000	3	0.05	0.15	1 3 3 5 5 7	3063270	3056379	4881	1097	52	861	0	0.999194	0.318248	0.880818	0.732753	0.998905
50000	200	1000	6000	3	0.1	0.15	1 3 3 5 5 7	3054769	3047798	5025	1095	34	817	0	0.999171	0.310903	0.888526	0.732867	0.998887
50000	200	1000	6000	3	0	0.1	1 3 3 5 5 7	3065390	3058855	4561	1143	3	828	0	0.999255	0.344692	0.883671	0.742539	0.998979
50000	200	1000	6000	3	0.05	0.1	1 3 3 5 5 7	3093469	3091314	207	0	1205	743	0	0.999772	-nan	0.877732	-nan	-nan
50000	200	1000	6000	3	0.1	0.1	1 3 3 5 5 7	3015154	3009888	3255	1052	137	811	11	0.999439	0.394377	0.892680	0.762165	0.999171
50000	200	1000	6000	3	0	0.05	1 3 3 5 5 7	3013900	3007357	4560	1080	20	883	0	0.999242	0.332154	0.874257	0.735218	0.998962
50000	200	1000	6000	3	0	0.03	1 3 3 5 5 7	2926790	2920614	4197	955	155	869	0	0.999256	0.315130	0.896338	0.736908	0.998965
50000	200	1000	6000	3	0.05	0.05	1 3 3 5 5 7	3059944	3057696	226	0	1178	843	1	0.999770	-nan	0.893482	-nan	-nan
50000	200	1000	6000	3	0.1	0.05	1 3 3 5 5 7	2986381	2979947	4464	1091	11	868	0	0.999250	0.337928	0.896694	0.744624	0.998976
50000	200	1000	6000	3	0.05	0.03	1 3 3 5 5 7	2992831	2987320	3547	1174	6	783	1	0.999406	0.410059	0.898451	0.769305	0.999147
50000	200	1000	6000	3	0.1	0.03	1 3 3 5 5 7	2979023	2973093	3958	1087	16	869	0	0.999332	0.364276	0.906152	0.756587	0.999070
100000	200	1000	6000	3	0	0.15	1 3 3 5 5 7	3025376	3023338	0	0	1161	0	877	0.999663	-nan	-nan	-nan	-nan
100000	200	1000	6000	3	0	0.1	1 3 3 5 5 7	3007294	3000951	4362	1130	47	803	1	0.999266	0.349197	0.889751	0.746071	0.998982
100000	200	1000	6000	3	0.1	0.15	1 3 3 5 5 7	2967021	2961268	3760	1184	5	803	1	0.999365	0.398787	0.890738	0.762963	0.999095
100000	200	1000	6000	3	0.05	0.15	1 3 3 5 5 7	3024097	3018419	3720	1099	34	824	1	0.999379	0.384400	0.874735	0.752838	0.999114
100000	200	1000	6000	3	0.1	0.1	1 3 3 5 5 7	2967057	2964906	216	15	1104	816	0	0.999777	0.026178	0.888889	0.638281	0.999380
100000	200	1000	6000	3	0.05	0.1	1 3 3 5 5 7	2928704	2921586	5174	1110	0	0	834	0.999115	0.269810	-nan	-nan	-nan
100000	200	1000	6000	3	0	0.05	1 3 3 5 5 7	2923694	2919471	2204	1027	178	811	3	0.999592	0.482726	0.896628	0.792982	0.999351
100000	200	1000	6000	3	0	0.03	1 3 3 5 5 7	2926106	2920221	3961	1092	9	822	1	0.999321	0.367553	0.884346	0.750407	0.999051
100000	200	1000	6000	3	0.05	0.05	1 3 3 5 5 7	3022790	3016444	4344	976	85	404	537	0.999266	0.285213	0.569415	0.617965	0.998882
100000	200	1000	6000	3	0.1	0.05	1 3 3 5 5 7	2911318	2904732	4543	1178	14	851	0	0.999216	0.352062	0.885075	0.745451	0.998918
100000	200	1000	6000	3	0.05	0.03	1 3 3 5 5 7	3041191	3035105	4123	1090	19	852	2	0.999318	0.356326	0.891213	0.748952	0.999053
100000	200	1000	6000	3	0.1	0.03	1 3 3 5 5 7	3023772	3017360	4475	1097	5	823	12	0.999258	0.336968	0.897981	0.744736	0.998989
200000	200	1000	6000	3	0	0.15	1 3 3 5 5 7	3029600	3027651	0	0	1142	0	807	0.999678	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0.1	0.15	1 3 3 5 5 7	2927139	2925129	0	0	1140	0	870	0.999657	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0	0.1	1 3 3 5 5 7	3056806	3054838	0	0	1083	0	885	0.999678	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0.05	0.15	1 3 3 5 5 7	3137357	3135366	0	0	1204	0	787	0.999683	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0.05	0.1	1 3 3 5 5 7	2984944	2982943	0	0	1135	0	866	0.999665	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0.1	0.1	1 3 3 5 5 7	2965674	2958921	4760	1140	35	764	54	0.999190	0.328957	0.858427	0.728858	0.998886
200000	200	1000	6000	3	0	0.05	1 3 3 5 5 7	3046434	3039830	4599	1216	48	738	3	0.999236	0.354003	0.873373	0.742204	0.998938
50000	200	1000	6000	1	0	0.15	1 3 3 5 5 5	3146963	3144801	381	49	889	843	0	0.999798	0.085143	0.885970	0.656970	0.999495
200000	200	1000	6000	3	0	0.03	1 3 3 5 5 7	2937252	2931376	3913	1192	10	759	2	0.999331	0.389224	0.890845	0.759800	0.999054
50000	200	1000	6000	1	0	0.1	1 3 3 5 5 5	3039349	3036852	474	78	992	953	0	0.999759	0.111270	0.896519	0.669183	0.999414
200000	200	1000	6000	3	0.05	0.05	1 3 3 5 5 7	3014914	3010349	2650	893	174	848	0	0.999531	0.405725	0.890756	0.765337	0.999290
200000	200	1000	6000	3	0.1	0.05	1 3 3 5 5 7	3090699	3084888	3853	1121	77	757	3	0.999363	0.373542	0.896388	0.756431	0.999095
50000	200	1000	6000	1	0	0.05	1 3 3 5 5 5	3019428	3017279	247	0	983	919	0	0.999796	-nan	0.881535	-nan	-nan
200000	200	1000	6000	3	0.05	0.03	1 3 3 5 5 7	2904322	2898536	3771	1157	34	824	0	0.999344	0.391407	0.888410	0.759720	0.999063
50000	200	1000	6000	1	0	0.03	1 3 3 5 5 5	2989487	2987320	205	5	1024	933	0	0.999794	0.009606	0.904070	0.637823	0.999424
200000	200	1000	6000	3	0.1	0.03	1 3 3 5 5 7	2977989	2972069	4019	928	117	856	0	0.999305	0.321943	0.882929	0.734726	0.999034

100000	200	1000	6000	1	0	0.15	1 3 3 5 5 5	3065095	3062942	219	0	1028	905	1	0.999796	-nan	0.891626	-nan	-nan
100000	200	1000	6000	1	0	0.1	1 3 3 5 5 5	3021239	3016685	2583	706	360	905	0	0.999512	0.341228	0.892945	0.744562	0.999248
100000	200	1000	6000	1	0	0.05	1 3 3 5 5 5	3015258	3013101	260	8	985	904	0	0.999793	0.015595	0.884973	0.633454	0.999435
100000	200	1000	6000	1	0	0.03	1 3 3 5 5 5	2957765	2954843	895	119	930	975	3	0.999691	0.131057	0.885157	0.671968	0.999345
50000	200	1000	6000	2	0	0.15	1 3 3 5 5 5	2996878	2990772	4113	1042	35	915	1	0.999307	0.344748	0.906389	0.750148	0.999043
50000	200	1000	6000	2	0.05	0.15	1 3 3 5 5 5	3048157	3042792	3485	906	49	925	0	0.999420	0.351435	0.906863	0.752573	0.999189
200000	200	1000	6000	1	0	0.15	1 3 3 5 5 5	2888602	2886074	580	87	981	880	0	0.999730	0.115308	0.886203	0.667080	0.999368
200000	200	1000	6000	1	0	0.1	1 3 3 5 5 5	2901133	2898935	227	0	1003	968	0	0.999788	-nan	0.895882	-nan	-nan
200000	200	1000	6000	1	0	0.05	1 3 3 5 5 5	2945212	2941442	1773	365	681	951	0	0.999583	0.246622	0.894638	0.713614	0.999282
200000	200	1000	6000	1	0	0.03	1 3 3 5 5 5	2912397	2909919	534	89	877	975	3	0.999758	0.130403	0.894495	0.674885	0.999434
200000	200	1000	6000	2	0	0.15	1 3 3 5 5 9	2973906	2968134	3767	1081	115	808	1	0.999346	0.368000	0.904309	0.757218	0.999067
200000	200	1000	6000	2	0.05	0.05	1 3 3 5 5 9	2963776	2956586	5156	1263	2	769	0	0.999128	0.338742	0.871388	0.736420	0.998813
200000	200	1000	6000	2	0	0.05	1 3 3 5 5 9	3000536	2994339	4202	1194	73	717	11	0.999287	0.368007	0.879755	0.749016	0.998991
200000	200	1000	6000	2	0.1	0.15	1 3 3 5 5 9	3028765	3022476	4240	1167	83	793	6	0.999285	0.362873	0.871908	0.744688	0.998988
200000	200	1000	6000	2	0.05	0.15	1 3 3 5 5 9	3004666	2998279	4390	1148	82	762	5	0.999254	0.349361	0.881944	0.743520	0.998958
200000	200	1000	6000	2	0.1	0.05	1 3 3 5 5 9	3069987	3063389	4574	1180	51	781	12	0.999246	0.347110	0.881490	0.742615	0.998954
200000	200	1000	6000	3	0	0.15	1 3 3 5 5 9	3033761	3031711	0	0	1271	0	779	0.999662	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0.05	0.15	1 3 3 5 5 9	3063322	3061281	0	0	1297	0	744	0.999667	-nan	-nan	-nan	-nan
200000	200	1000	6000	3	0	0.05	1 3 3 5 5 9	3002711	2996612	4071	1246	3	773	6	0.999321	0.389619	0.895194	0.761378	0.999040
200000	200	1000	6000	3	0.1	0.05	1 3 3 5 5 9	3039532	3032555	4943	1214	56	764	0	0.999176	0.339248	0.849833	0.729419	0.998863
200000	200	1000	6000	3	0.05	0.05	1 3 3 5 5 9	2938850	2932419	4427	1131	118	755	0	0.999226	0.344030	0.866820	0.736692	0.998913
200000	200	1000	6000	3	0.1	0.15	1 3 3 5 5 9	3049823	3047766	0	0	1211	0	846	0.999663	-nan	-nan	-nan	-nan
200000	200	1000	6000	4	0	0.15	1 3 3 5 5 9	3055862	3050358	3555	1125	69	751	4	0.999406	0.397527	0.871230	0.756054	0.999139
200000	200	1000	6000	4	0.05	0.15	1 3 3 5 5 9	3042508	3040523	0	0	1282	0	703	0.999674	-nan	-nan	-nan	-nan
200000	200	1000	6000	4	0	0.05	1 3 3 5 5 9	3020830	3018797	0	0	1291	0	742	0.999663	-nan	-nan	-nan	-nan
200000	200	1000	6000	4	0.1	0.15	1 3 3 5 5 9	3009588	3007538	0	0	1217	0	833	0.999659	-nan	-nan	-nan	-nan
200000	200	1000	6000	4	0.05	0.05	1 3 3 5 5 9	2991104	2984949	4209	1107	95	720	24	0.999279	0.349708	0.860729	0.736572	0.998984
200000	200	1000	6000	4	0.1	0.05	1 3 3 5 5 9	2954554	2947652	4865	1184	53	666	134	0.999166	0.323012	0.856592	0.726257	0.998845
2000000	200	1000	6000	3	0	0.05	1 3 3 5 5 9	2987573	2985533	0	0	1297	0	743	0.999658	-nan	-nan	-nan	-nan
2000000	200	1000	6000	3	0	0.15	1 3 3 5 5 9	3101160	3099115	0	0	1291	0	754	0.999670	-nan	-nan	-nan	-nan
2000000	200	1000	6000	3	0.05	0.05	1 3 3 5 5 9	3006419	3004163	242	0	1261	682	71	0.999750	-nan	0.913597	-nan	-nan
2000000	200	1000	6000	3	0.1	0.05	1 3 3 5 5 9	3050173	3043006	5217	1212	34	704	0	0.999138	0.326465	0.849216	0.724940	0.998829
2000000	200	1000	6000	3	0.1	0.15	1 3 3 5 5 9	3005863	2999657	4193	1251	13	749	0	0.999299	0.383155	0.893795	0.758750	0.999014
2000000	200	1000	6000	3	0.05	0.15	1 3 3 5 5 9	2988656	2986650	0	0	1279	0	727	0.999664	-nan	-nan	-nan	-nan
200000	50	1000	6000	1	0	0.05	0 0 3 5 14 0	2981114	2976503	2269	1496	0	846	0	0.999619	0.579620	0.944724	0.841321	0.999393
200000	50	1000	6000	1	0	0.15	0 0 3 5 14 0	3054967	3050489	2057	1524	0	897	0	0.999663	0.608626	0.948704	0.852331	0.999453
200000	50	1000	6000	3	0	0.15	0 0 3 5 14 0	2993411	2988915	2134	1404	0	958	0	0.999643	0.580165	0.949455	0.843088	0.999430
200000	50	1000	6000	3	0.1	0.15	0 0 3 5 14 0	3092444	3087458	2662	1444	0	880	0	0.999569	0.530687	0.942184	0.824147	0.999334
200000	50	1000	6000	3	0	0.05	0 0 3 5 14 0	2994326	2990083	1842	1516	8	877	0	0.999692	0.643327	0.908338	0.850452	0.999484
200000	50	1000	6000	3	0.05	0.15	0 0 3 5 14 0	3034817	3030153	2230	1512	0	922	0	0.999632	0.586729	0.948560	0.844974	0.999411
200000	50	1000	6000	3	0.05	0.05	0 0 3 5 14 0	3052259	3047372	2488	1417	44	938	0	0.999585	0.537353	0.953252	0.830063	0.999349
200000	50	1000	6000	3	0.1	0.05	0 0 3 5 14 0	3085090	3080581	2130	1549	0	830	0	0.999654	0.603193	0.947489	0.850112	0.999441
800000	50	1000	6000	1	0	0.05	0 0 3 5 5 9	3067936	3059975	5513	1506	0	942	0	0.999100	0.364428	0.878731	0.747420	0.998751
800000	50	1000	6000	1	0	0.15	0 0 3 5 5 9	3145107	3137050	5729	1442	0	886	0	0.999088	0.344235	0.882910	0.742077	0.998755

800000	50	1000	6000	3	0	0.15	0 0 3 5 5 9	3050678	3048354	0	0	1396	0	928	0.999619	-nan	-nan	-nan	-nan
800000	50	1000	6000	3	0	0.05	0 0 3 5 5 9	3086500	3078570	5509	1451	2	968	0	0.999106	0.355681	0.883615	0.746134	0.998767
800000	50	1000	6000	3	0.05	0.05	0 0 3 5 5 9	2961195	2953548	5219	1544	0	884	0	0.999117	0.383745	0.871795	0.751553	0.998758
800000	50	1000	6000	3	0.1	0.05	0 0 3 5 5 9	2975197	2967926	4919	1418	0	934	0	0.999172	0.374884	0.907677	0.760578	0.998846
800000	50	1000	6000	3	0.1	0.15	0 0 3 5 5 9	3002585	2993731	6470	1515	6	863	0	0.998920	0.341063	0.735094	0.691692	0.998511
800000	50	1000	6000	3	0.05	0.15	0 0 3 5 5 9	3122207	3114957	4858	1470	15	907	0	0.999218	0.386893	0.894477	0.760196	0.998897
2000000	20	1000	6000	1	0	0.05	0 0 3 5 11 3	3005986	2999749	3777	1438	0	1022	0	0.999371	0.442189	0.932057	0.791206	0.999081
2000000	20	1000	6000	1	0	0.15	0 0 3 5 11 3	3005612	2999913	3294	1522	0	883	0	0.999451	0.490335	0.931435	0.807074	0.999173
2000000	20	1000	6000	3	0	0.15	0 0 3 5 11 3	2966421	2960982	2997	1495	0	947	0	0.999494	0.511111	0.932546	0.814384	0.999227
2000000	20	1000	6000	3	0	0.05	0 0 3 5 11 3	2870309	2864784	3110	1509	0	906	0	0.999457	0.503420	0.931620	0.811499	0.999175
2000000	20	1000	6000	3	0.05	0.15	0 0 3 5 11 3	2988082	2982378	3267	1511	0	926	0	0.999453	0.490903	0.932997	0.807784	0.999175
2000000	20	1000	6000	3	0.15	0.15	0 0 3 5 11 3	3037340	3031758	3251	1412	0	919	0	0.999464	0.476946	0.922691	0.799700	0.999198
2000000	20	1000	6000	3	0.1	0.15	0 0 3 5 11 3	2998518	2993221	2913	1475	0	909	0	0.999514	0.514924	0.931352	0.815263	0.999255
2000000	20	1000	6000	3	0.05	0.05	0 0 3 5 11 3	3145033	3139448	3229	1532	0	824	0	0.999486	0.497160	0.926884	0.807844	0.999222
2000000	20	1000	6000	3	0.1	0.05	0 0 3 5 11 3	3002449	2996631	3470	1422	0	926	0	0.999421	0.461913	0.921852	0.794395	0.999143
2000000	20	1000	6000	3	0.15	0.05	0 0 3 5 11 3	2979734	2974403	2870	1543	0	916	2	0.999518	0.528515	0.938044	0.822026	0.999255

Table B.2.: Results of all TPC-H runs

Run	Vanilla	Original Plugin	With New Machine Learning Plugin
1	1042658	1054007	1036182
2	1039925	1050762	1034200
3	1035200	1048119	1035120
4	1042403	1044674	1036611
5	1041397	1049328	1035628
6	1041351	1051518	1037412
7	1039175	1055153	1033510
8	1045547	1049618	1037504
9	1042099	1046696	1037705
10	1040021	1048140	1036597
11	1040775	1047674	1037161
12	1043359	1046899	1035756
13	1044932	1048048	1036199
14	1038196	1049510	1031695
15	1041640	1048792	1035186
16	1044502	1047157	1036083
17	1045897	1047395	1037151
18	1040179	1047033	1037719
19	1042933	1048465	1037381
20	1042540	1046888	1035049
21	1037441	1052230	1035620
22	1035923	1047602	1033994
23	1040443	1050589	1036850
24	1040374	1047381	1035636
25	1036788	1050823	1035076
26	1037901	1046013	1041343
27	1035382	1047982	1042648
28	1034830	1047639	1034271
29	1038869	1045581	1032974
30	1035616	1050994	1035132
31	1038062	1049426	1037158
32	1036523	1049702	1032722
33	1037392	1044023	1035265
34	1037892	1046949	1038293
35	1037497	1048342	1036540
36	1037372	1047188	1036681
37	1037144	1049830	1039298
38	1035350	1045764	1037855
39	1043482	1048842	1033883
40	1039243	1045275	1034179
41	1037274	1044753	1036005
42	1032720	1045669	1034621
43	1036054	1045343	1032747
44	1035408	1044741	1036430
45	1037938	1043600	1041133
46	1033760	1045493	1039737
47	1036132	1046627	1036902
48	1036517	1046081	1037377
49	1034635	1052677	1040130
50	1035028	1090480	1039972

Table B.3.: Results of all sysbench runs

Run	Delete			Insert			Point-Select			Read-Only			Read-Write			Update-Index			Update-Non-Index			Write-Only		
	vanilla	original	new	vanilla	original	new	vanilla	original	new	vanilla	original	new	vanilla	original	new	vanilla	original	new	vanilla	original	new	vanilla	original	new
1	132.30	127.49	125.96	108.30	102.31	59.28	8508.06	8757.72	8840.78	442.02	449.48	250.66	42.59	39.74	39.68	88.49	87.10	89.88	127.48	105.37	112.96	67.96	63.84	55.75
2	125.08	128.36	129.91	99.97	103.15	63.75	9054.78	8906.29	8808.24	455.18	435.73	250.09	40.84	39.73	37.19	88.06	92.28	91.05	127.15	123.88	124.59	63.73	60.86	51.23
3	130.67	125.04	127.36	105.24	102.35	68.03	9083.01	8851.26	8876.19	443.06	438.47	246.07	39.84	38.38	38.95	90.65	91.36	90.54	131.13	122.29	130.85	64.04	64.34	52.69
4	130.33	129.08	129.52	107.62	101.08	61.53	8906.92	8690.09	8922.66	446.98	436.12	246.54	40.41	37.53	40.96	91.79	89.80	90.31	132.98	126.85	128.73	62.46	58.38	53.06
5	131.15	127.26	129.94	107.12	107.45	61.05	8973.72	8841.15	9011.88	442.40	443.78	245.06	40.27	40.46	38.06	92.31	88.99	92.52	131.86	132.48	128.07	61.78	63.95	53.04
6	131.52	124.99	132.31	110.66	106.73	63.75	8807.95	8994.07	8906.64	448.25	450.79	246.51	39.65	41.36	39.87	91.81	93.81	93.45	132.05	129.81	128.80	63.29	61.97	52.36
7	129.64	129.16	127.73	106.22	104.92	65.64	8698.24	8693.76	8951.87	444.82	438.23	251.42	41.62	39.61	38.28	92.55	90.85	94.65	132.31	132.50	134.01	63.49	62.86	53.79
8	133.14	124.13	129.11	110.13	107.25	63.75	9005.99	8637.47	8995.17	456.82	439.22	251.18	40.88	38.97	38.54	90.46	94.15	92.74	133.21	130.38	128.87	62.98	63.66	52.81
9	133.87	126.88	130.79	110.36	111.63	67.84	8817.83	9007.22	8862.38	449.70	437.86	252.78	40.84	38.77	37.16	96.44	94.87	94.98	133.45	131.56	129.53	63.72	64.33	52.89
10	131.53	131.68	128.94	110.77	108.56	65.97	8969.94	8839.79	8913.46	449.51	439.60	247.79	41.64	40.59	38.03	95.25	91.19	100.91	134.46	132.61	134.51	68.77	64.57	54.74
11	130.16	126.73	129.22	112.73	111.72	63.75	8922.44	8774.82	8662.79	457.91	453.39	248.95	43.67	39.38	38.85	91.53	92.38	93.12	130.28	133.19	132.31	64.53	65.38	56.12
12	129.77	128.44	128.76	111.57	108.00	61.78	8913.86	8790.80	9019.84	451.05	444.96	251.89	43.13	40.33	39.47	94.89	95.69	92.26	135.75	132.83	135.46	66.45	62.16	54.57
13	131.86	126.27	129.17	112.69	109.48	68.44	8889.68	8878.51	9080.52	450.43	439.56	251.34	42.08	41.01	41.86	93.88	95.93	93.69	133.93	131.92	132.64	65.29	69.76	55.63
14	131.64	127.93	130.41	117.31	111.96	64.04	8900.07	8847.68	8938.95	442.48	446.37	264.10	41.03	41.64	41.15	95.51	91.47	94.99	135.08	133.47	137.28	62.82	62.53	53.64
15	131.72	131.32	130.88	114.76	116.65	70.28	8795.49	8930.65	8846.47	451.80	435.90	254.12	44.21	39.92	40.09	91.89	94.73	90.89	131.87	132.61	130.42	67.44	60.23	53.54
16	130.27	127.03	129.58	119.24	115.88	70.17	9054.54	8954.43	8896.41	452.91	445.76	257.77	40.83	42.81	41.83	96.25	92.73	96.37	132.69	132.67	131.55	66.15	66.67	51.88
17	132.81	127.99	129.68	121.21	118.93	66.77	8943.83	8840.31	8982.41	439.65	439.50	257.18	42.48	40.89	40.59	94.64	92.28	94.38	134.89	132.70	135.25	67.65	64.59	54.19
18	129.71	130.47	129.23	119.99	119.43	68.94	9060.79	8877.02	8958.73	448.34	443.17	248.62	42.33	41.93	38.72	96.27	94.28	94.53	133.16	132.14	131.98	68.96	65.55	55.66
19	130.45	129.08	129.64	118.48	113.41	71.56	9045.63	8750.58	8860.95	443.95	444.37	256.01	42.94	43.65	41.41	96.77	95.11	97.32	133.25	132.25	137.14	67.78	66.41	51.29
20	132.03	130.00	130.64	120.31	119.13	72.29	8926.55	8741.79	8893.41	443.23	447.76	260.83	42.50	41.98	40.77	93.52	96.27	93.46	133.89	136.45	136.01	64.29	69.76	58.62
21	133.40	134.00	134.62	119.07	120.14	68.82	9073.98	8862.44	8837.69	442.76	438.98	249.54	41.50	40.34	40.99	95.59	94.63	91.50	135.79	136.06	136.09	68.47	63.96	53.15
22	136.84	132.26	131.95	120.04	115.67	68.09	8977.62	8793.25	9035.41	437.95	451.11	245.70	42.29	41.24	41.15	97.54	94.18	97.18	137.45	137.82	140.41	66.83	68.23	55.48
23	132.92	130.85	130.96	122.72	119.93	69.70	8978.59	8976.32	8952.36	440.88	449.88	251.58	42.95	38.54	39.74	95.03	96.08	96.93	139.17	137.42	139.41	65.37	72.98	57.54
24	133.94	128.97	130.04	124.58	120.98	70.38	8996.64	8660.92	8944.66	440.20	448.15	251.68	40.81	41.77	40.74	96.45	90.75	94.76	139.33	137.78	139.65	64.53	67.77	52.65
25	134.70	132.24	133.23	124.28	122.44	70.58	9199.26	8896.11	8830.84	437.41	441.21	254.47	42.17	41.73	39.49	92.48	96.78	93.95	136.04	137.33	137.45	68.07	64.36	58.33
26	132.28	127.04	133.16	127.80	119.50	69.04	8886.83	8954.55	8758.52	458.48	446.09	250.17	43.75	40.40	39.36	95.11	92.17	95.33	135.50	133.41	136.69	71.93	69.52	59.88
27	134.93	133.08	130.45	124.92	122.44	70.69	8980.90	8779.47	8964.33	444.59	434.69	249.42	44.03	42.59	40.55	99.46	96.65	93.19	138.38	137.18	134.06	66.98	68.84	55.59
28	131.87	129.11	130.45	128.02	125.87	71.38	9159.90	8921.92	8922.48	448.18	446.94	248.86	42.05	43.96	41.20	95.78	93.42	97.04	136.75	136.45	135.95	70.32	68.37	54.77
29	132.02	130.08	130.92	128.04	125.68	70.63	9075.71	8670.05	9028.41	446.88	458.30	255.10	43.95	42.65	41.02	96.47	95.37	98.81	136.27	134.43	136.22	68.46	70.68	57.28
30	131.92	130.88	131.54	127.73	125.68	68.57	9039.48	8937.80	9039.76	448.04	439.30	247.67	42.93	42.69	40.67	100.16	93.45	93.78	136.86	136.28	135.98	69.68	69.32	54.54
31	133.31	130.29	132.39	130.04	125.86	72.97	8806.66	8843.95	9017.60	444.85	445.77	253.27	41.66	42.91	42.38	93.26	93.36	98.76	136.77	135.12	135.26	70.18	65.24	53.48
32	131.42	132.17	129.72	129.82	124.55	70.66	9065.69	8551.51	8998.57	442.66	444.73	252.38	45.15	41.11	39.48	99.57	90.06	91.87	133.72	138.18	137.45	70.09	69.28	56.74
33	131.56	129.93	132.38	128.07	126.40	71.99	8818.75	8829.30	8789.38	440.74	442.88	248.19	42.07	41.79	41.56	93.96	98.25	95.98	137.68	133.64	136.11	68.45	71.47	56.04
34	129.57	130.06	129.73	130.37	124.15	67.48	9114.51	8866.82	9030.12	446.39	433.90	248.47	41.13	41.95	39.22	96.83	92.17	95.08	135.83	133.97	136.02	67.71	70.93	54.58
35	128.33	125.77	132.03	128.52	126.65	73.04	9176.00	8941.59	8835.98	444.67	444.07	254.86	44.32	41.75	42.64	93.62	93.85	95.45	137.38	135.98	136.15	71.38	72.67	58.93
36	130.24	132.12	133.17	129.34	128.14	70.56	8979.83	8735.76	8939.24	447.71	447.68	250.59	42.58	41.39	39.86	94.87	97.49	94.69	136.24	138.38	136.93	62.89	72.76	56.36
37	130.33	131.07	129.19	126.95	128.36	72.42	9027.38	8840.22	9081.30	448.35	441.45	249.27	42.95	42.77	40.97	98.34	95.29	100.12	135.27	136.27	136.57	67.84	67.34	59.84
38	131.12	128.97	128.75	128.90	126.05	69.35	8816.54	8843.19	8683.46	444.65	441.11	259.08	43.52	42.48	38.88	95.59	93.87	95.46	135.01	136.25	137.26	74.48	70.74	57.94
39	131.41	131.12	133.44	131.32	127.68	72.38	9145.00	8857.26	8831.26	451.99	436.25	256.46	42.11	41.71	41.80	95.36	95.69	95.28	133.02	133.55	135.67	71.27	71.56	61.15
40	132.05	131.44	131.43	133.31	128.08	73.34	8834.25	8744.05	8778.14	449.09	433.18	252.91	43.09	44.35	40.72	98.55	95.43	97.14	138.99	136.08	137.33	69.23	69.60	57.98

41	128.82	132.36	134.44	133.95	130.69	73.49	9116.60	8844.98	8835.79	451.11	451.36	255.15	43.15	42.04	40.98	95.13	94.67	93.32	137.18	135.39	136.36	69.61	76.12	58.93
42	129.77	132.58	130.18	134.00	129.53	71.29	8959.18	8958.38	9011.73	443.78	446.73	253.07	43.59	42.96	41.47	97.75	91.84	97.42	135.86	135.01	135.57	73.12	72.79	59.64
43	130.93	131.97	128.63	134.12	131.27	72.42	9032.67	8918.00	9016.69	459.73	438.55	253.19	43.15	41.54	42.19	99.91	96.33	95.29	135.94	136.23	136.25	68.65	74.17	58.23
44	129.66	127.73	135.91	132.66	135.18	70.43	8927.71	8820.17	8898.59	451.36	436.13	246.00	43.73	42.17	41.44	97.57	96.29	98.66	136.31	135.45	135.51	73.98	69.94	59.93
45	133.36	131.10	131.52	133.20	131.94	75.75	8947.12	8899.52	8698.47	436.07	436.00	252.56	43.99	44.68	40.58	93.91	96.23	95.64	134.09	136.54	136.56	73.57	73.88	60.53
46	132.26	130.03	130.71	136.87	128.18	70.05	9029.86	8853.28	8715.59	448.46	447.09	255.58	43.85	42.96	40.86	95.09	93.36	95.15	135.65	135.68	137.37	74.08	73.97	58.96
47	128.11	128.98	134.01	135.06	132.84	70.19	8973.92	8717.69	9124.49	445.56	443.43	256.48	41.82	41.39	42.22	98.31	96.72	90.85	134.41	135.20	134.88	71.56	75.45	59.97
48	130.76	128.54	134.04	136.06	135.09	70.84	9108.98	8921.14	8952.36	449.26	436.61	252.64	44.46	42.44	40.58	96.09	94.54	94.09	135.98	135.51	134.67	77.46	72.57	59.83
49	132.55	131.22	130.47	135.94	131.58	74.89	8857.52	8774.31	8905.09	448.35	442.31	255.99	44.44	41.63	41.65	99.76	94.71	96.89	137.78	135.93	137.68	76.64	72.42	57.76
50	131.13	130.35	129.02	132.61	132.25	72.48	8935.56	8875.99	9012.18	447.29	435.41	245.09	41.98	43.76	41.01	94.79	94.44	97.65	137.41	135.91	135.15	74.79	72.26	59.22

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Edingen-Neckarhausen, 30. April 2021

. . . .

(Sebastian Schindler)