

Project Report

Intelligent Cybersecurity Solutions
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

Gruppe 2

Viktor Schlüter
Binayak Ghosh
Christoph Gerneth

Supervisor: Leily Behnam Sani

WS 2018/2019

1 Motivation

with industry 4.0: more critical systems connected to internet important: ICS, because often important for essential services for population energy water gas traffic controlling otherwise: downtimes very expensive in the past often very vulnerable -> defence against cyber attacks is necessary and important

defense complicated: systems in past not developed to withstand attacks: only in closed, protected networks not feasible anymore because connection to outside from everywhere with IoT

2 Related Work

Our work is based on the work of previous research. The dataset we are basing our work on is a result of the research from Uttam Adhikari, Shengyi Pan and Thomas Morris[1] and officially published in 2014. In 2011, Morris built an Industrial control testbed for SCADA control Systems at the Mississippi State University SCADA Security Laboratory and Power and Energy Research laboratory. He described it in detail in his paper[2].

The researchers published in total 6 different datasets from Industrial Control Systems (ICS) including different known attack. It has since been used by many other researchers for their work.

In 2013, Morris and Gao introduced a classification for attacks on ICS systems [3]. In their work, they took a set of 17 attacks on an SCADA system and classified them into 4 groups: Reconnaissance, response and measurement injection, command injection and denial of service attacks.

There has been lot of research utilizing this dataset in the most recent years.

[4] described how attacks on Industrial Control System can be detect by an Intrusion Detection System. Shengyi Pan, Thomas Morris and Uttam Adhikari used a technique called common path mining and were able to classify different disturbances, normal control operations, and cyber-attacks in Industrial Control systems.

3 Problem Specification and Objectives

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Problem: how well can we build a IDS for ICS traffic with our given set of technologies?

our tech stack pyspark pandas jupyter
no scikit! (because that is what's usually used)

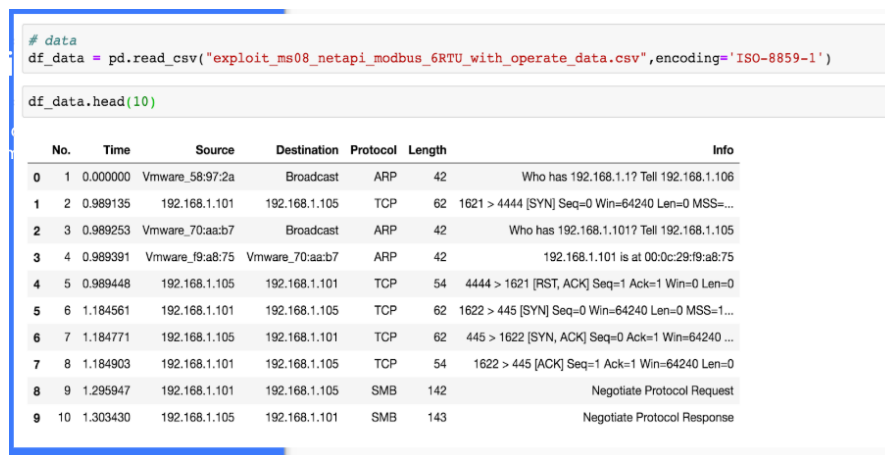
4 Approach

Our approach to this project included several steps and methods which we tried till we got better results. We tried around four different approaches and finally compared the results and selected the best method.

In the beginning, we had already decided to use Spark Machine Learning Library (MLlib) for our machine learning classification. In addition for pre-processing of the dataset, we decided to use Python as our programming language of choice. We used Python module Pandas for processing the dataset and for cleaning it.

4.0.1 First Approach: Basic Features that Wireshark can extract

After initially loading the Modbus dataset and using Pandas to process it, the raw data looked like Figure 4.1.



```
# data
df_data = pd.read_csv("exploit_ms08_netapi_modbus_6RTU_with_operate_data.csv", encoding='ISO-8859-1')

df_data.head(10)
```

	No.	Time	Source	Destination	Protocol	Length	Info
0	1	0.000000	Vmware_58:97:2a	Broadcast	ARP	42	Who has 192.168.1.1? Tell 192.168.1.106
1	2	0.989135	192.168.1.101	192.168.1.105	TCP	62	1621 > 4444 [SYN] Seq=0 Win=64240 Len=0 MSS=...
2	3	0.989253	Vmware_70:aab7	Broadcast	ARP	42	Who has 192.168.1.101? Tell 192.168.1.105
3	4	0.989391	Vmware_f9:a8:75	Vmware_70:aab7	ARP	42	192.168.1.101 is at 00:0c:29:f9:a8:75
4	5	0.989448	192.168.1.105	192.168.1.101	TCP	54	4444 > 1621 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
5	6	1.184561	192.168.1.101	192.168.1.105	TCP	62	1622 > 445 [SYN] Seq=0 Win=64240 Len=0 MSS=1...
6	7	1.184771	192.168.1.105	192.168.1.101	TCP	62	445 > 1622 [SYN, ACK] Seq=0 Ack=1 Win=64240 ...
7	8	1.184903	192.168.1.101	192.168.1.105	TCP	54	1622 > 445 [ACK] Seq=1 Ack=1 Win=64240 Len=0
8	9	1.295947	192.168.1.101	192.168.1.105	SMB	142	Negotiate Protocol Request
9	10	1.303430	192.168.1.105	192.168.1.101	SMB	143	Negotiate Protocol Response

Figure 4.1: Snippet of the Modbus dataset

We tried to train a couple of classifiers like Random Forest Classifier and KPrototypes Clustering based on the features in this dataset, except the 'INFO' column but our results were not at all promising.

4.0.2 Second Approach: Info Column

For this unconventional approach, the human readable Info column from wireshark was used to predict whether a packet is part of an attack.

Packet No.	Info
1355	> 502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
502	> 1355 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1
1355	> 502 [ACK] Seq=1 Ack=1 Win=64240 Len=0
	Query: Trans: 2260; Unit: 1, Func: 3: Read Holding Registers
1356	> 502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
502	> 1356 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1
	Response: Trans: 2260; Unit: 1, Func: 3: Read Holding Registers
1356	> 502 [ACK] Seq=1 Ack=1 Win=64240 Len=0
	Query: Trans: 2288; Unit: 1, Func: 3: Read Holding Registers
1357	> 502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
502	> 1357 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1

Figure 4.2: A snippet of the Info column

In order to process the INFO column, the 'Bag of Words' technique was used where each set of strings in the column was treated as a separate word to generate a feature vector from it. Our aim was to see if there were any words that had high correlation with the attack labels. The classifier used for the 'INFO' column was Logistic Regression.

4.0.3 Third Approach: Generated Features

To get more information from the raw network traffic, we directly extracted features from the pcap files and generated more features based on this information. The extracted features consisted of:

- eth.dst
- eth.src
- sport
- dport
- tcp
- udp
- ip.proto (tcp/udp/icmp)
- tcp.flag

4 Approach

The generated features consisted of:

- max packets
- mean packets
- min packets
- packets per ip dst
- packets per ip src
- packets per ip proto
- packets per proto
- packet rate (packets in the last 0.1 seconds)
- ip.src
- ip.dst

-> table with description of each feature

4.0.4 Fourth Approach: Combination of Data

Our best results were archived with a combination of the three approaches. We combined the whole dataset and included the Protocol type, the INFO column and some of the extracted and generated features. The now used list of features was the following:

- Source IP address
- Destination IP address
- Protocol type
- Packet Length
- INFO column
- Ethernet Destination Address
- Ethernet Source Address
- Source Port

- Destination Port
- TCP
- UDP
- Attack

When we used these features with a logistic regression, we got promising results.

4.0.5 Our metrics (written by Viktor)

Because this is a defensive system that should detect network traffic detects, we optimized our classification process for maximum recall. Recall is defined as

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \quad (4.1)$$

For an intuitive understanding the recall is the percentage of attacks that gets detected by the system and classified as an attack. High recall means that the probability of detecting attacks is very high and the probability of missing attacks is low. Because we our first goal is to avoid any attacks, optimized for this metric.

However we need a second, also important metric, because a high recall could also be archived by classifying everything as an attack: The precision. Precision is defined as

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \quad (4.2)$$

The precision is high, if many classified attacks are real attacks and not just false alarms. We new from the beginning that on a dataset, where the share of attacks in only 0.1% we would get some false alarms, but still we wanted to keep this number as high as possible. However this has less priority than recall, because a false alarm can allways be double checked, a missed actual attack can be catastrophic.

5 Implementation

5.1 Getting the Raw Data

In this section, we describe how we were able access information for the raw PCAP recordings.

The PCAP file format[5] is used by the PCAP Packet Capture Library to store raw network packet stream data and additional metadata like timestamps. Morris[1] provided the dataset in this format. We needed a way to access specific information as described in section 5.2.

5.2 Features

The CSV files were concatenated to get all the data in a single document. The columns generated were used as features for classification with the attack labels. The generated features were:

- Source IP address
- Destination IP address
- Protocol type
- Packet Length
- INFO column (as shown by wireshark to users to get an overview over the packet payload)
- Attack

The total number of data points available was 912054, with the Attack label containing either '0' or '1' for non-attacks and attacks respectively. The concatenated document was loaded into a Pandas Dataframe in Python for preprocessing. Initially the INFO column was not considered for the preprocessing.

5.2.1 Statistical Analysis

Some statistics regarding the dataset were collected prior to implementing the classification task. Some of the major statistics are displayed below:

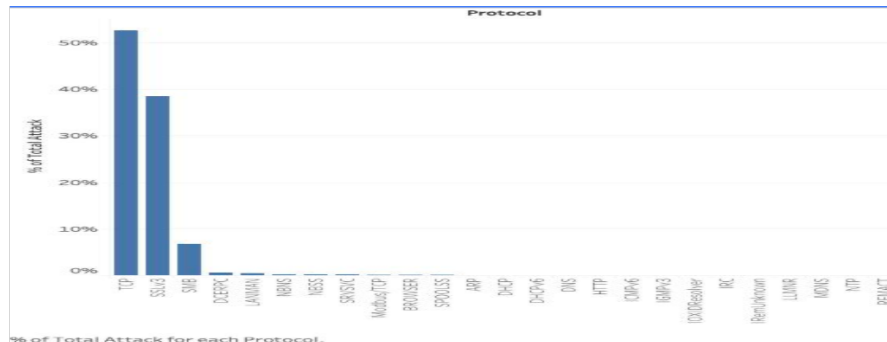


Figure 5.1: Share of Attack traffic for each Protocol

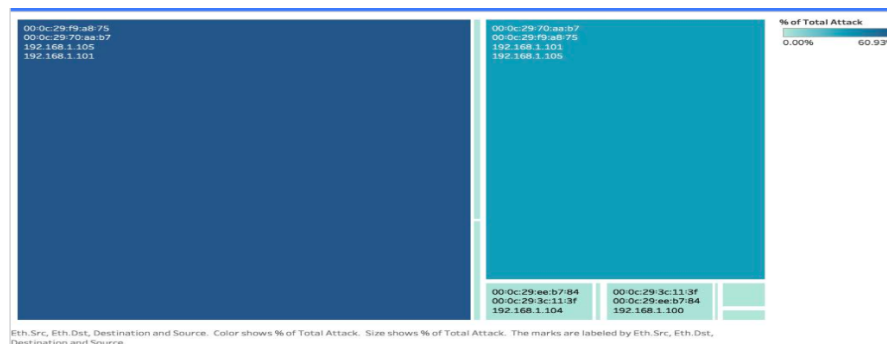


Figure 5.2: Correlation of Source and Destination addresses as well as Ethernet addresses with Attacks

5.2.2 Categorical variables and Feature Importance Mapping

Most of the attribute columns in the dataset featured Categorical data. Categorical had to be transformed in order to make it usable for classification purposes. For categorical data, Pyspark's vectorizer algorithms were used. The string categorical variables were converted using One Hot Encoding, StringIndexer and VectorAssembler in Pyspark.

- **StringIndexer:** StringIndexer encodes a string column of labels to a column of label indices. The indices are in $[0, \text{numLabels})$, ordered by label frequencies, so the most frequent label gets index 0. If the input column is numeric, it is casted it to string and index the string values. <https://spark.apache.org/docs/2.1.0/ml-features.html##stringindexer>

5 Implementation

- One Hot Encoding: One-hot encoding maps a column of label indices to a column of binary vectors, with at most a single one-value. This encoding allows algorithms which expect continuous features, such as Logistic Regression, to use categorical features. <https://spark.apache.org/docs/2.1.0/ml-features.html#onehotencoder>
- VectorAssembler: VectorAssembler is a transformer that combines a given list of columns into a single vector column. It is useful for combining raw features and features generated by different feature transformers into a single feature vector, in order to train ML models like logistic regression and decision trees. VectorAssembler accepts the following input column types: all numeric types, boolean type, and vector type. In each row, the values of the input columns will be concatenated into a vector in the specified order. <https://spark.apache.org/docs/2.1.0/ml-features.html#vectorassembler>

One Hot Encoding and StringIndexer led to high dimensionality of the data. RandomForest classifier was used to rank the relative importances of the features, but due to high dimensionality of the data, the Feature Importance Mapping was not very helpful.

5.3 Initial Result with Classifiers

With the initial set of features and the dataset, a couple of preliminary classification algorithms were implemented to check the results. Prior to this, the dataset was divided randomly into training and test sets in a ration of 3:1. A supervised learning algorithm, Random Forest Classifier, was implemented on the training dataset and the tested on the test dataset.

Random forests are ensembles of decision trees. Decision trees and their ensembles are popular methods for the machine learning tasks of classification and regression. Decision trees are widely used since they are easy to interpret, handle categorical features, extend to the multiclass classification setting, do not require feature scaling, and are able to capture non-linearities and feature interactions. Tree ensemble algorithms such as random forests and boosting are among the top performers for classification and regression tasks.

The Spark Machine Learning Library(MLLib) implementation supports decision trees for binary and multiclass classification and for regression, using both continuous and categorical features. The implementation partitions data by rows,

5 Implementation

allowing distributed training with millions or even billions of instances. Random forests combine many decision trees in order to reduce the risk of overfitting. <https://spark.apache.org/docs/2.1.0/ml-classification-regression.html##decision-trees> The results of the implementation are presented below.

	Prediction: No Attack	Prediction: Attack
Label: No Attack	182278	729000
Label: Attack	185	90

Figure 5.3: Confusion Matrix for the Results of the Classification Task by the Random Forest Classifier.

An unsupervised learning algorithm, KModes (Huang, Z.: Clustering large data sets with mixed numeric and categorical values, Proceedings of the First Pacific Asia Knowledge Discovery and Data Mining Conference, Singapore, pp. 21-34, 1997.) was implemented on the training set and tested on the test set. The results are the following:

	Prediction: No Attack	Prediction: Attack
Label: No Attack	579826	330823
Label: Attack	1405	0

Figure 5.4: Confusion Matrix for the Results of the Classification Task by the KModes Classifier.

From figures 5.3 and 5.4, it can be seen that none of the algorithms are performing quite well, with very low prediction accuracy and high false positives.

After we applied the undersampling technique to the training data (explained in the next section) and used these features that are extracted by wireshark, we got improved results as shown in table 5.1. This time we used a logistic regression which is the algorithm we have gotten the best results off of. As described in the approach chapter the logarithmic regression was optimized for recall: The recall was 98.76%, the precision 17.18%. Because of the optimization the recall was expected to be that high, the precision is not very high, but its performance stands in clear contrast to the classifications we made before.

5 Implementation

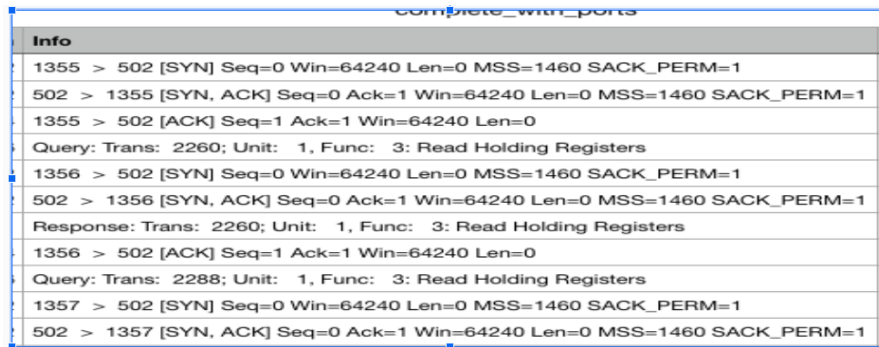
	attack	no attack	Precision
Attack detected	0.1449%	0.6984%	17.18%
no attack detected	0.001824%	99.15%	
recall	98.76%		

Table 5.1: The confusion matrix for the features extracted by wireshark

5.4 Undersampling the Training Dataset

One of the reasons behind the low performance of the classifiers is the extremely small number of Attack labels in the dataset. The length of the dataset was 912054 whereas the number of attack labels was only 1405. This resulted in an extremely imbalanced data set. The technique implemented to overcome this problem was to undersampled the training data. The dataset was initially divided into training and test set randomly, which resulted in Length of training data: 637997 data points for the training set and 274057 data points for the test set. The training set was then undersampled in proportion of the ratio of the Attack labels in the whole dataset, which resulted in a ratio of 0.51 in the training set.

5.5 The 'INFO' column



Info
1355 > 502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
502 > 1355 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1
1355 > 502 [ACK] Seq=1 Ack=1 Win=64240 Len=0
Query: Trans: 2260; Unit: 1, Func: 3: Read Holding Registers
1356 > 502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
502 > 1356 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1
Response: Trans: 2260; Unit: 1, Func: 3: Read Holding Registers
1356 > 502 [ACK] Seq=1 Ack=1 Win=64240 Len=0
Query: Trans: 2288; Unit: 1, Func: 3: Read Holding Registers
1357 > 502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
502 > 1357 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1

Figure 5.5: A snippet of the INFO column

In order to process the INFO column, the 'Bag of Words' technique was used where each set of strings in the column was treated as a separate word to generate a feature vector from it. The Regular Expression Tokenizer (RegexTokenizer) package from PySpark was used to extract each word from the column. RegexTokenizer allows more advanced tokenization based on regular expression (regex) matching. By default, the parameter

5 Implementation

“pattern” (regex, default: "s+") is used as delimiters to split the input text.

The StopWordsRemover package from PySpark was used to split the text into words. Stop words are words which should be excluded from the input, typically because the words appear frequently and don't carry as much meaning. StopWordsRemover takes as input a sequence of strings (e.g. the output of a Tokenizer) and drops all the stop words from the input sequences. The list of stopwords is specified by the stopWords parameter, which in this case was "->".

The CountVectorizer package from PySpark was used to map the generated words into a feature vector. CountVectorizer aims to help convert a collection of text documents to vectors of token counts. When an a-priori dictionary is not available, CountVectorizer can be used as an Estimator to extract the vocabulary, and generates a CountVectorizerModel. The model produces sparse representations for the documents over the vocabulary, which can then be passed to other algorithms like Linear Discriminant Analysis.

During the fitting process, CountVectorizer selects the top vocabSize words ordered by term frequency across the corpus. An optional parameter minDF also affects the fitting process by specifying the minimum number (or fraction if < 1.0) of documents a term must appear in to be included in the vocabulary. Another optional binary toggle parameter controls the output vector. If set to true, all nonzero counts are set to 1. This is especially useful for discrete probabilistic models that model binary, rather than integer, counts.

Info attack	words	filtered	features
502 > 2217 [FIN...]	0 502, 2217, fin, ...	502, 2217, fin, ...	(10000,[0,1,3,4,5...
2228 > 502 [SYN...]	0 2228, 502, syn, ...	2228, 502, syn, ...	(10000,[1,2,3,4,5...
2217 > 502 [ACK...]	0 2217, 502, ack, ...	2217, 502, ack, ...	(10000,[0,1,3,4,5...
502 > 2228 [SYN...]	0 502, 2228, syn, ...	502, 2228, syn, ...	(10000,[0,1,2,3,4...
2218 > 502 [FIN...]	0 2218, 502, fin, ...	2218, 502, fin, ...	(10000,[0,1,3,4,5...

only showing top 5 rows

Figure 5.6: INFO column processing

Figure 5.6 shows the transformation of the INFO column into a feature vector.

After we applied the undersampling strategy, we measured how well the bag of words model on the input performed in classifying network packets. We used a logistic regression, the recall was 99% and the precision 11.5%, table 5.7 shows the full confusion matrix. We were surprised how well the text alone was able to classify

5 Implementation

packets, but a precision of roughly 10% was not high enough, only every 10th detected attack would be an actual attack.

	attack	no attack	precision
attack detected	0.1489%	1.14%	0.1155%
no attack detected	0.001106%	98.71%	
recall	0.9926%		

Table 5.2: Confusion matrix of only the Bag of Words model on the INFO column

5.6 Direct Feature Extraction and Generated Features (From here by Viktor)

Because the features extracted by wireshark were limited and did not include the source and destination ports, we used both the wireshark commandline utility tshark and the python pcap parsing library dpkt to get more features. Many features, for example the ports, could be extracted with either of them, we decided to go with tshark whenever possible and to only use our own extracted features where we needed to, because tshark and wireshark are highly tested software, in our experience that is usually more reliable than new implementing functionality.

5.6.1 Extracting features with tshark

Because tshark is a command line utility, we wrote a simple bash script to export an extracted csv file for every pcap file. The features that we extracted were:

- frame number
- timestamp
- source MAC address
- destination MAC address
- source ip address
- destination ip address
- tcp source/destination port

5 Implementation

- udp source/destination port
- The features marked with were already included in the wireshark feature extraction

Because the source and destination port were either included in the tcp or the udp port fields, we aggregated them to two features: sport and dport, each describing the source and the destination port, regardless whether tcp oder udp is used. Two more features, 'tcp' and 'udp', were extracted as binary values, marking whether the packet was sent over tcp or udp. The now used list of features was the following:

- Source IP address
- Destination IP address
- Protocol type (as String, e.g. "ARP" or "SSL")
- Packet Length
- INFO column
- Ethernet Destination Address
- Ethernet Source Address
- Source Port
- Destination Port
- TCP
- UDP

When we used these features with a logistic regression, we got again improved results:

	attack	no attack	Precision
Attack detected	0.1519%	0.4324%	25.99%
no attack detected	0%	99.42%	
recall	100.0%		

Table 5.3: Confusion matrix of the classification with the extracted features

This looked more promising, however we could now combine these features with the wireshark info column. This lead to our best results.

5 Implementation

	attack	no attack	Precision
Attack detected	0.1456%	0.07882%	64.88%
no attack detected	0.001095%	99.77%	
recall	99.25%		

Table 5.4: Confusion matrix of our best results: Subset of the extracted features with Logistic Regression

As shown in table 5.4, the recall was 100% while the precision improved to 65%. This means that two out of three attack 'alarms' are an actual attack. While this might still sound like a bad performance, be reminded that only 0.01% of the packets in our dataset are attacks. Having a precision of 65% means that only 0.079% of all harmless packets are marked as an attack, which a little less than 1 in 1000. On an earlier test we used the port numbers as scalar features, this yielded slightly worse results. When we used the ports as categorical features instead, we realized that the order of port numbers does not have meaning and therefore should be categorical features.

5.6.2 Generating features from literature

Because the authors of [2] had generated features from the directly extracted ones, we wanted to try this approach as well. As the authors of [2] already mentioned that a weakness of the dataset is the periodical schedule all legit traffic is following. This is caused by the Modbus protocol itself, periodical polling is used as a way to distribute information in the network. To get some time information, we generated the following time based feature: 'packet_rate'. The features 'min_packet', 'max_packet' and 'mean_packet' were assigned the minimum, maximum and mean rate of the packet rate in the last second.

This feature measures how many packets were observed in the last 0.1 seconds. The idea behind this was to see whether the packet is part of a packet burst or not. If a packet is not part of a packet burst, it is likely outside of the regular polling schedule and more likely to be an attack.

To get more network protocol based information out of the pcap files, we also extracted the following features:

- TCP sequence Number
- TCP flag
- IP payload length

5 Implementation

Some other features already contained in the wireshark feature set and the tshark features set were also extracted but not used because they turned out to be redundant.

The TCP sequence number could have been used to implement a stateful connection analysis, however we did not have time for this, it is noted in the outlook chapter. The TCP flag and the payload length of the IP layer were just further information that we had not extracted already. In another classifier we checked the performance of only these new features:

	attack	no attack	Precision
Attack detected	0.1519%	0.4324%	25.99%
no attack detected	0%	99.42%	
recall	100.0%		

Table 5.5: Confusion matrix of only the dpkt and generated features

The solo performance of these features were better then for example the Info Column, so this looked promising. However after we used them in combination with the Info Column and the features from 5.6, we were disappointed:

	attack	no attack	Precision
Attack detected	0.1519%	0.1758%	46.34%
no attack detected	0%	99.67%	
recall	100.0%		

Table 5.6: Confusion matrix of Info column, wireshark features, dpkt features and generated time features

These new features actually *decreased* our classification precision at the same recall. We thought that maybe logistic regression is not the best algorithm to use after all and tried a RandomForest:

	attack	no attack	Precision
Attack detected	0.1475%	0.3675%	28.63%
no attack detected	0.004424%	99.48%	
recall	97.09%		

Table 5.7: Confusion matrix of RandomForest classification with all features

Now the performance decreased even more, with only 29% of precision and high recall. One problem with the machine learning on spark is the limited number of classification algorithms, DecisionTrees are not yet supported for the modern DataFrame

5 Implementation

API. The last algorithm that was suitable for highly categorical data like ours was Multi layer perceptron (MLP). This is a feedforward artificial neural network, however with the disadvantage that the user can tweak far fewer parameters than in other libraries such as Tensor Flow. The results were disappointing as well:

	attack	no attack	Precision
Attack detected	0.143%	3.671%	3.75%
no attack detected	0.008847%	96.18%	
recall	94.17%		

Table 5.8: Confusion matrix for the MLP classification with all features

The arrays of neurons we used was [10369, 100, 10, 2]. With bigger machines at hand we could have tried even higher neuron numbers, but our available hosts were already struggling with this amount. The first number of neurons can not be chosen, it is the number of features that are present in the training dataset. We have that many features because we have a lot of categorical values, we encode them using one hot encoding and one hot encoding produces data with very high dimensionality.

Our last idea was that maybe some of the new added features were random enough to overlap the positive effects of others so we did empirical tests where we would only use one of the newly added features with the successful ones from 5.6, but this always slightly decreased our performance. Our only explanation is that the new features we generated and extracted with dpkt were not as good as we thought. Even more were we surprised to see that the features from the paper [TODO: pyramid paper] did not work for us, either did not implement them correctly or they worked in better in the paper because they had smaller data chunks specific to certain attacks. We, on the other hand, made the challenge bigger by combining all the data chunks and accepting our very low percentage of attack traffic.

However we thought that our best performance case was sufficiently, because we lost a lot of time in the project on setting up the spark environment we decided to stop our search for new features here.

To test whether the Logistic Regression is really the best algorithm for our best performing feature set, we tested the feature set from our best results with a RandomForest Classifier. The results can be seen in table 5.9.

With 8.1% precision this was clearly not better suited than the logistic regression. Additionally a trial run with a Multilayer Perceptron could have been done, but it turned out that crashed everytime because it needed to much memory. Comparing to the results

5 Implementation

	attack	no attack	Precision
Attack detected	0.1511%	1.719%	8.082%
no attack detected	0.002212%	98.13%	
recall	98.56%		

Table 5.9: The confusion matrix of the classification with the best features and RandomForest

of table 5.8, it is not likely that the MLP would have produced better results with this similar feature set so these trials were not done.

6 Evaluation

In this chapter we will present and discuss our results. In the first part these will be our prediction results from the classifier we trained, in the second part we will evaluate how well our set of given technologies is suited for developing intrusion detection systems.

6.1 Training Results

The best performance our classifier reached was a recall of 100% with a precision of 55.8%. For our project this seemed like very sufficient, but when this is compared with realistic usecases, it is still likely not good enough. One big modelling error we make in projecting our results to the real world is the dataset itself. It is unknown to us, whether the share of 0.015% being attacks is realistic for industrial control systems or not. If trivial attempts such as ssh logins with common passwords are counted this is entirely possible, but this dataset modeled rather sophisticated attacks with metasploit, this is likely not going to happen every 10000 packets in the real world, as it is happening in our dataset. With the real world attack share being even lower, our precision would increase even lower. This would lead to network administrators to receiving long logs about detected attacks, but all being false alarms.

On the other side it is very complicated to automatically train a classifier with a generated dataset like the one we used. For real world applications, at least some configuration will be required by the network administrators. Our approach of machine learning makes that difficult, it is not trivial to add networking rules or whitelist traffic in an already trained classifier.

We are over all happy with the performance of our intrusion detection system, however one of its biggest shortcomings is that it is not possible to use it on live traffic. This is the case because it depends on the network packet info column from the wireshark packet analyzer, calling wireshark for every packet or every chunk of packets would introduce significant delay to a firewall. Higher performance might have been possible with more sophisticated implementations of Neural Networks, the Multilayer Perceptron from Spark is inferior to other popular implementation such as Tensor Flow or Caffe.

Probably the most interesting part of our implementation is the bag of words model used on the wireshark info column, which added 24% of precision to the wireshark features (26% solo performance) leading to our best results. However this features prevented us from building a live system and makes the implementation platform dependent.

Our most important step to good results was the idea to undersample the training data, only with an attack ration of around 50% the spark MLlib algorithms worked well, all results prior to that were not satisfactory.

6.2 The suitability of our chosen Technologies (The last part that Viktor wrote)

6.2.1 Modbus Dataset

Although not strictly a technology and rather a choice we had to make for this project, the Modbus worked well for building our classifier. As with all datasets from controlled lab environments it is questionable how well it reflects reality, but it is certainly better than dataset that were generated completely or don't have labeled attacks. Because the dataset had only one attack label and very few attacks, we did not build an attack classification process, this might have been interesting as well.

6.2.2 Spark

Using the python interface of spark and its machine learning library MLlib was probably the most interesting part of our project: Spark has powerful capabilities for automated parallelization of calculations, however this comes with a more complicated setup process. We especially missed the time needed to setup multicore processing from jupyter notebooks dearly in later stages of the project. Spark is better suited for projects with bigger datasets: Is single hosts are overwhelmed with the amount of data it is very easy with spark to run operations on clusters of slaves. This can enable the processing of much bigger dataframes but as we did not need it, spark was not the most efficient choice for our project. A simpler framework with more sophisticated algorithms as scikit-learn for python would probably have been better in terms of classification performance.

6.2.3 Pandas

Pandas is a very common python package to handle and process dataframes. It was usefull to load csv files, process the data and then load it into spark dataframes. An interesting difference to spark is that it does all computations in a single CPU core, because spark uses all four (most modern CPUs have four cores). This was noticable for some of the more expensive calculations

6.2.4 Jupyter

Jupyter is also a very commonnly used tool for data processing in python. It allows the execution of chunks of code in so called cells. Cells consist arbitrary numbers of lines of code which are executed together, afterwards the state of the variables is saved. This allows programmers to save results, such as preprocessed data frames. If some operation in the middle of the script fails, the calculation can easily be restarted from the top of the cell that threw the error. In contrast to traditional python files the script does not have to be executed again from the beginning, usually saving a lot of time.

7 Outlook

With this work, we showed that defending against Cyberattacks in Industrial systems using machine learning models can be done. We showed that the the quality of the results mostly depend on three factors: The quality of the training data, the choice of the features used for training and the machine learning algorithm.

Even if our system showed some good results, it's unknown if it would be feasible to use in in a real world scenario. The resulting model was trimmed to fit best with the training data.

As next step, we will evaluate it against another, similar dataset or a real wold scenario. Further a stateful connection analysis could improve the results, this would connect packets that come from the same connection between to hosts.

An interesting approach could be a potential benchmark against other, comparable IDS systems for this kind of Industrual Control System. The results could then be used to create an even better model.

Bibliography

- [1] T. Morris, *Industrial control system (ics) cyber attack datasets*, <https://sites.google.com/a/uah.edu/tommy-morris-uah/ics-datasets>, 2014.
- [2] T. Morris *et al.*, “A control system testbed to validate critical infrastructure protection concepts”, *International Journal of Critical Infrastructure Protection*, vol. 4, no. 2, pp. 88–103, 2011.
- [3] T. H. Morris and W. Gao, “Industrial control system cyber attacks”, in *Proceedings of the 1st International Symposium on ICS & SCADA Cyber Security Research*, 2013, pp. 22–29.
- [4] S. Pan *et al.*, “Developing a hybrid intrusion detection system using data mining for power systems”, *IEEE Transactions on Smart Grid*, vol. 6, no. 6, pp. 3104–3113, 2015.
- [5] V. Jacobson *et al.*, *Of the lawrence berkeley national laboratory*, <https://www.tcpdump.org/manpages/pcap.3pcap.html>, 2007.

Appendix A