NIDS - ML: Project Report

Authors: Jeremy Choo, Samuel Ng, Yang Chenglong, Phan Duy Nhat Tan, Nguyen Dang Huu (NUS - School of Computing)

Abstract

Attacks from hackers can be a serious security threat especially with the prevalence of the internet. Our group has attempted to apply machine learning in building a Network Intrusion Detection System (NIDS). Our project aims to apply neural network to identify and differentiate the features between normal traffic and intrusive traffic and, as a result, classify a packet as normal or intrusive. We also explore the possibility of zero-day prediction on network attack to review if the model is able to detect unlearned attacks. Last but not least, we evaluate the real-time performance of the trained network to identify trade-off between speed and accuracy of the network in practical scenarios of real-time NIDS. The project achieves good accuracy in differentiating normal traffic and attack traffic and it shows possibility of identifying unacknowledged attacks that has not observed. The project also gives a structured methodology for feature extraction of time-series network packets.

Introduction

The evolution of the Internet has become inevitable in networking and communication. Up until July 2019, more than half of the world were active internet users. However, in 2017 alone, cybercrimes cost about 11.7 million dollars, with up to approximately 50-day resolving time. [1] These attacks include breach of critical data such as users' data or company's data. These statistics has caused cyber-security to start attracting a tremendous amount of intellectual and financial investments to develop several protection systems against those malicious attempts.

Network Intrusion Detection System (NIDS) is one of the commonly developed systems that identifies network attacks. One example of such a system is SNORT. NIDS passively inspects traffic traversing the devices on which they sit. The traffic is analyzed using certain predefined rules which requires a considerable amount of research and attention in manual work and may lead to missing unobserved types of attacks due to unobserved behaviors [3]. The overview of a NIDS in a network can be seen in Fig 1, which illustrates the two parts of NIDS one is used for listening to network conversations in promiscuous mode and the other is used for control and reporting.

With the observation on the importance of network attack detection and strength and weaknesses of a typical NIDS, our group aims to develop an enhanced NIDS system that can itself identify differences between observed attacks and normal traffic and classify them correctly for reporting. Moreover, we proceed to investigate the possibility of zero-day detection, which can allow the NIDS to detect new attacks in the wild that it wasn't trained on. This could potentially support administration on identifying and annotating new types of attacks. This enhanced system can enable more automatic rule set and support on the broadening attention on unseen attack behaviors.

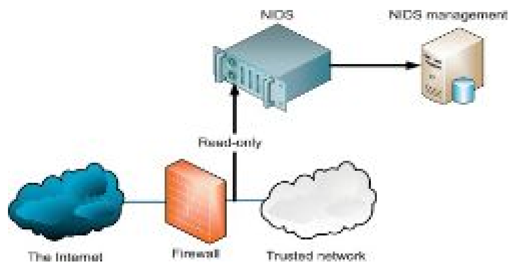


Fig 1: Overview of Network Intrusion Detection System (NIDS) [3]

Our deep learning model investigates the features in one packet and examines packets in a network transaction in a time series manner. These are enabled using a Long Short-term memory Auto-encoder, and the results will be passed to the classifier to classify if the traffic is normal or malicious.

Related Work

Building a NIDS in order to detect malicious traffic is not a new field. Many other studies have been done that uses machine learning in order to detect malicious traffic. One of the common ways in order to do so involves using an autoencoder to learn to encode normal traffic, with the idea being that malicious traffic are sufficiently different from normal traffic such that it is able to detect them. However, in practice the use of such methods has shown many flaws. F. Farahnakian and J. Heikkonen showed that not only is a large and diverse set of traffic required to train the model on all of the possible types of network traffic, it can result in a large number of false positives or false negatives depending on which features autoencoder considers important [4].

Many other methods have been developed in order to address this issue. A. Goyal and C. Kumar showed GA-NIDS, a Genetic Algorithm based Network Intrusion Detection System that had encouraging results with an accuracy of 92.3% [2]. Y. Mirsky, T. Doitshman, Y. Elovici and A. Shabtai showcases Kitsune, an ensemble of Autoencoders for Online Network Intrusion Detection [8]. Unfortunately, their accuracy for detecting attacks were not as good, achieving AUC of 0.80458 to 0.99857 depending on the type of attack. P. Amoli and T. Hamalainen proposes a Real Time Unsupervised NIDS for detecting Unknown and Encrypted Network Attacks in High Speed Network [7]. M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas and J. Lloret demonstrates that a Conditional Variational Autoencoder can be used to classify network intrusions with an accuracy of 0.8010 [6]. The ID-CVAE employed can also perform feature reconstruction, allowing packets that have been corrupted to be potentially salvaged. M. A. Salama, H. F. Eid, R. A. Ramadan, A. Darwish and A. E. Hassanien demonstrated the use of a Deep Belief Network as a feature selector along with a Support Vector Machine which performs the classification [5]. When applied on training data, this resulted in an accuracy of 0.93.

Here we will expand on the use of LSTM autoencoders in order to perform feature selection, along with a 0 - 1 classifier that attempts to learn which features are indicative of malicious traffic and which are indicative of normal traffic. Our main contributions are as follows:

1) We employ a LSTM autoencoder in order to detect the important features that differ between malicious and normal traffic, as well as simplify the time-based data into

a single feature vector. Instead of training the autoencoder solely on normal data and rely on attacks being anomalies, we train the autoencoder on both sets of data.

2) We demonstrate that the use of such a model is able to successfully detect other types of attacks that was not present in the training data, which was one of the main advantages of using an autoencoder for classification.
3) We attempt to use the model to perform real time traffic detection in order to detect and block attacks before they are successful.

Method

Data Cleaning

In order to successfully process the data, we had to perform data cleaning, converting the network traffic into a common set of columns.

Firstly, we converted all of the dataset to have the same number of packets in order to feed the data into the autoencoder. To do so, we calculated the average number of packets among all the files in the dataset and performed padding or truncation, depending on if the number of packets in the network traffic was larger than or less than the average number of packets. If there was less packets, then we would perform padding, padding the rows of the data frames with -1. The padding rows would then be ignored later in the model with a masking layer. We also find the average occurrence of each column, dropping the columns that are occur in less than 50% of the files we read in. We would then assign each of the newly read in dataframes to either the training dataframe, testing or validation dataframe. We also examine features in the data frame and drop features related to IP address, and time marking such as time_epoch to avoid the model learning based on the time in which the scripts are ran and use it for prediction.

To ensure that all the data frame (train, test and validation) have the same breadth i.e. columns, we drop the columns which are not shared between them. After this, we sort the columns in the dataframes to ensure that the columns match up. We then convert all the columns to either numerical or categorical data, dropping those which cannot be converted.

We then proceed with filling in missing values, which appear when those columns were not recorded in some of the network traffic. In order to do so, we populated missing numerical data with the mean of the available data, while categorical data is populated with the mode of the available data. One hot encoding is also carried out on the categorical data. In this way, we also rename identically named columns by appending a counter to them. We also scale all non-binary columns to be between 1 and 0 so that all the data are treated with the same importance by the autoencoder.

After this process, some of the features that were chosen by the autoencoder are:
• source_layers_tcp_tcp.window_size_scalefactor
• source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.type
• source_layers_tcp_tcp.flags_tree_tcp.flags.ns
• source_layers_tcp_Timestamps_tcp.time_delta
• source_layers_tcp_tcp.checksum.status
• source_layers_frame_frame.len
The full list is available in Appendix C.

To make the data consumable by the LSTM architecture, we would thus convert the 2D data frame into a 3D matrix. Thus, we stacked the data frames such that the first dimension is the number of samples, the second represents each packet and the last are the columns (features) of the data.

For GPU usage, we would need to split the data down into smaller batches to fit into memory. To accomplish that, we would need to make use of a generator function which will yield us a particular sized subset of data we specified and we will call the fit function on that subset. We manually looped for a number of epochs where one epoch is considered completed when the generator has yielded all subsets of the data set. However, within each fit function call, the weights are tweaked per sample of data in the subset of data yielded.

Autoencoder Model

We make use of an autoencoder to reduce the dimensionality of the data and compress the time series data into something that is more palatable to a regular neural network. We make use of a 3 layer LSTM architecture to perform this task.

LSTMs are a special kind of recurrent neural network. The internal machinery of a LSTM node is more complex than other "vanilla" types of neural nets. This allows it to better remember context from much further in the past. For example in a typical neural net, for each node there is an activation function which we will need to tweak as part of the training process. However in a LSTM node, there are 4 additional layers which decides the information we are going to keep, how we will update the information as well as what kind of information we will actually output from the node, as opposed to the hidden state which we will pass on to the next time step unadulterated.

Using this LSTM autoencoder, we would be able to capture the time based information of network traffic and so be able to better boil down the high dimensionality of the data set into its latent features.

The LSTM begins with a masking layer. The masking layer is required in order to adapt the model to fit network traffic of different number of packets. We perform padding on each network traffic to create packets with a value of '-1' in it's dimensions. The masking layer is then used to inform the model that all the -1's it sees are in fact not actual data.

We made use of a LeakyReLU activation function. In the beginning we tried using a regular ReLU but the training loss was generally increasing till NaN loss was reached after just a few epochs. After some checking, we felt that it may be because the data is quite sparse which causes the backpropagation process while using a ReLU to stagnate at 0, effectively shutting down parts of the network, causing underfitting to be rampant. After swapping to a LeakyReLU which allows for some negativity to pass on, the loss rate returned to normalcy.

We have also made the last LSTM layer not return its "hidden" representation of the data as we will be making use of the last layer's output as the encoded, dimensionality reduced sketch of the data. We've also made use of a DropOut layer to avoid any possible overfitting issues.

As the autoencoder reduces the set of time based features into a single set of features, we utilized a repeat vector to repeat the compressed feature set multiple times. In theory, this teaches the autoencoder to encode the relevant time based information as features, as it only receives the single set of features to reconstruct the original data.

The final time distributed layer also ensures that for each time step the LSTM node outputs, it would be be transformed into a certain dimensionality. This is accomplished by making use of dense layers applied to each of the time step to force the current output of the LSTM layer into the same dimensionality of the input date that was fed into the initial encoder. The illustration of the the LSTM auto-encoder is shown in Fig 2.

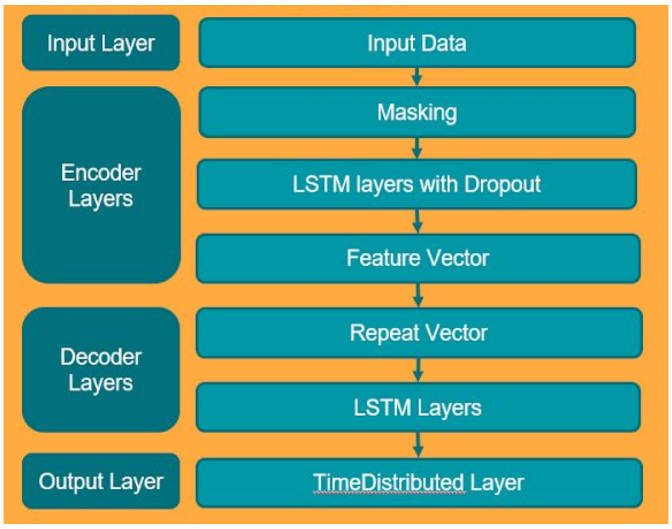


Fig 2: Overview of LSTM auto-encoder's architecture

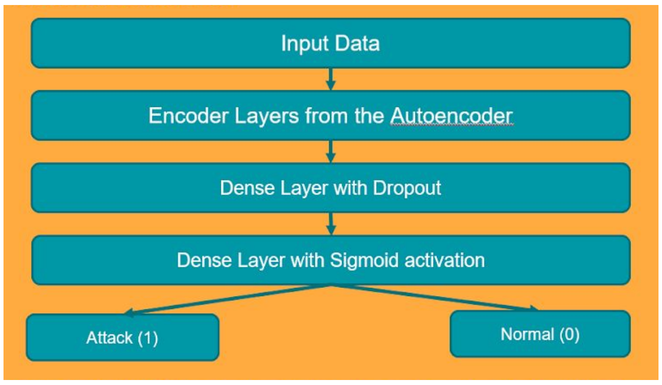
Classifier Model

The classifier is built with three main blocks: the pre-trained autoencoder, dense layers with dropout and dense layer with sigmoid function as output block.

The classifier model use the technique of transfer learning, as the best epoch results from auto-encoder is used with pre-trained weights to pass into the next layers in classifier. The first auto-encoder block, we use the last LSTM decoder layer as the input to classifier, which means disregarding the time distributed layer. The features are laid out from temporal features to spatial features in next layers of classifier.

The dense block with dropout performs as hidden layers in the network to learn the spatial features, using leaky ReLU to avoid weights vanishment in these layers during back propagation. Dropout layers with 50% dropout are added in between dense layers to reduce overfitting due to complex neural network architecture.

The output block which uses one dense layer and sigmoid activation to convert the classifier output into binary classification. We recorded the probability of prediction as known as confidence from the binary prediction to intend for threshold setting in later stage to evaluate if there are any uncertain prediction at around 50:50 chances of normal or intrusive. The overview model of classifier is illustrated as follows.



Input Data
Encoder Layers from the Autoencoder
Dense Layer with Dropout
Dense Layer with Sigmoid activation
Attack (1)
Normal (0)

Fig 3: Overview of Classifier's architecture

Metrics

For autoencoder, we use root mean square distance as the evaluation metric of the autoencoder to consider its performance. The result is expected to be near the original data representation but with a reduction of dimensions. PCA analysis is also used to understand the result of autoencoder as successful auto-encoder will be able to separate data representation between normal and attack traffic.

For classifier, we use accuracy as the main metric. However, we also use precision and recall metrics to further assess the performance of classifier. As NIDS includes two main problems: detecting attack traffic and allowing normal traffic; assessing these two scores help to understand if the performance in considered reasonable in the practices. Harmonic between these two results is assessed via F1 scores. The computation of these scores is illustrated in Appendix B (in which positive is normal and negative is intrusive, this is used throughout the report).

Evaluation

Dataset

Our dataset is kindly provided to us by Dr. Lin Yun, a researcher at NUS. We mainly made use of the traffic involving Poodle, Breach and RC4 attacks and normal traffic. In order to process these files, we utilized tshark to convert the data into JSON. Our program would then read in this JSON data and flatten it recursively i.e. each variable would have its own column in the resulting dataframe.

Autoencoder Result

The best results the autoencoder was achieved after 16 epochs, and is recorded below:

Table 1: LSTM auto-encoder result

Model	Training Loss	Validation Loss	Testing Loss
LSTM Autoencoder	0.0710	0.06919	0.0793

Another way to visualize how well the autoencoder is doing is to perform PCA analysis on the encoded dataset provided by the autoencoder. The result is shown below, with 0.896 explained variance for the first component and 0.095 explained variance for the second.

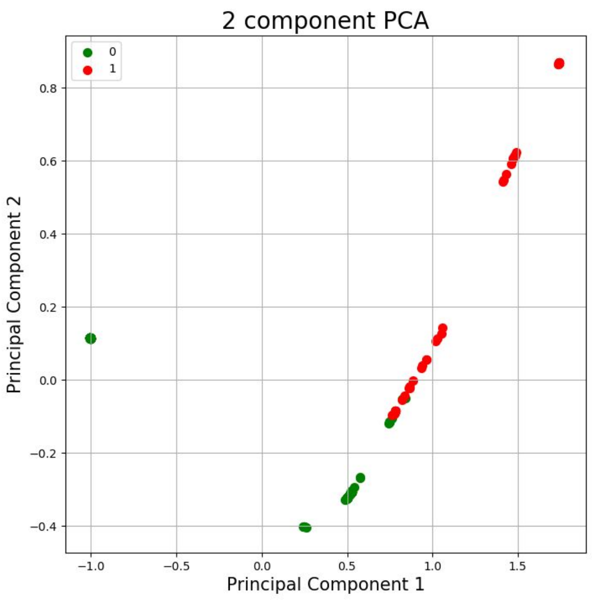


Fig 4: Two component PCA for all attacks

The PCA analysis shows promising results for the autoencoder. The dataset is clearly separable with the two principal components with very little overlap, showing the autoencoder is doing a good job with its feature selection.

Classifier Result

The result obtained for the implemented classifier is illustrated as follows. A more detailed tabulation is available under Appendix A.

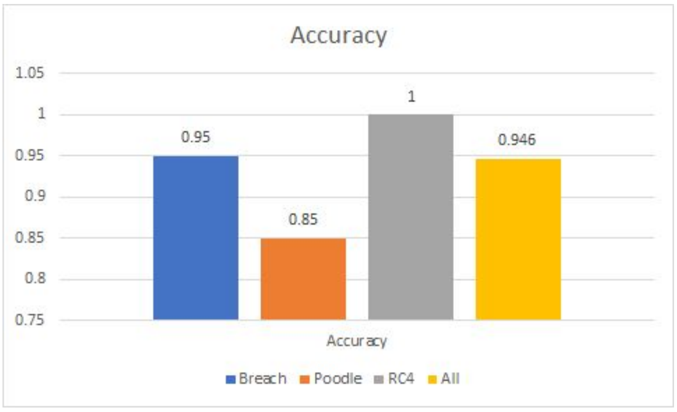


Fig 5: Classifier accuracy score on each type of attack

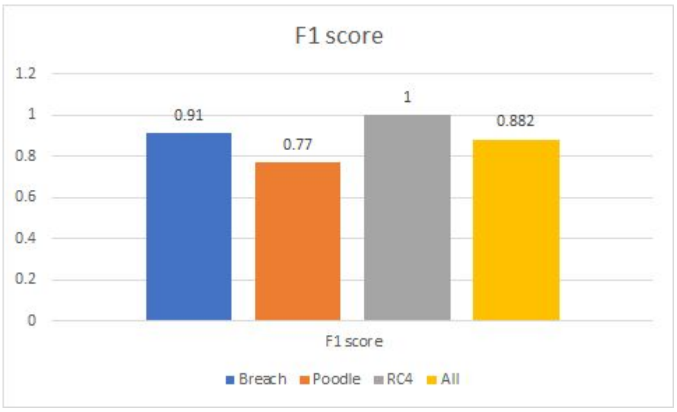


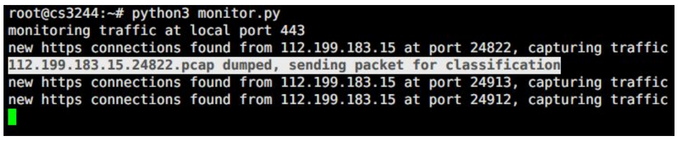
Fig 6: Classifier F1 score on each type of attack

We test our model on different types of attack with equal amount of data, and once on all types of attacks together. From the results, we can see that our model can classify normal and attack data fairly accurately, especially in the case of RC4 attacks. Furthermore, we the confidence for detecting attacks is also quite high at 70%.

Additionally, we achieved a perfect recall score on all types of attacks but a lower precision score. This means that the model can accurately detect all malicious data, but at the expense of misclassifying some normal data. We, hence, run our test on the confidence of the false positive prediction to observe if there is any prediction considered weak prediction with around 50% confidence. However, the prediction confidence level for even false positive cases are also quite high so sacrificing threshold may lead to leaking of intrusive results to be considered normal. In the context of a NIDS, the model is suitable for a server that stores sensitive information as the cost of letting malicious data in will be much worse than blocking normal traffic.

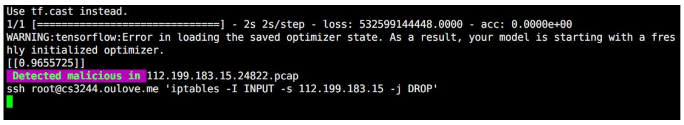
Real-time testing

Real-time detection is possible with pre-trained models. On the server side, a monitor.py python script is monitoring the network connection on port 443, and if clients are connecting to that port, tcpdump will be invoked to dump a continuous stream of 500 packets between the server and client into a pcap file and send it to the classifier to perform classification. it turned out that the real-time_IDS.py is doing its job relatively well. it could detect all the malicious traffic but sometimes misclassify normal traffic as malicious traffic. This type of model is good to use in a scenario where data privacy is of top priority so that it would rather misclassify a normal traffic as malicious than letting any malicious traffic to pass through.



```
root@cs3244:~# python3 monitor.py
monitoring traffic at local port 443
new https connections found from 112.199.183.15 at port 24822, capturing traffic
112.199.183.15.24822.pcap dumped, sending packet for classification
new https connections found from 112.199.183.15 at port 24913, capturing traffic
new https connections found from 112.199.183.15 at port 24912, capturing traffic
```

Fig 7: Connection Monitoring and Traffic Dumping on Server



```
Use tf.cast instead.
1/1 [==============================] - 2s 2s/step - loss: 532599144448.0000 - acc: 0.0000e+00
WARNING:tensorflow:Error in loading the saved optimizer state. As a result, your model is starting with a fres
hly initialized optimizer.
[[0.9655725]]
 Detected malicious in 112.199.183.15.24822.pcap
ssh root@cs3244.oulove.me 'iptables -I INPUT -s 112.199.183.15 -j DROP'
```

Fig 8: Classification Process on the aws server.

Zero day vulnerability testing

Zero day vulnerability testing was carried out by training with the data without a specific attack, and testing with the data of that particular type of attack. The result was recorded below:

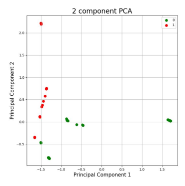
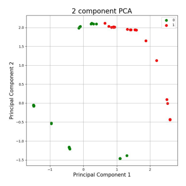
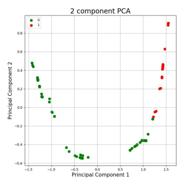
Table 2: Zero day vulnerability testing results

Models without the attack	Loss	Accuracy
Breach	0.2200	0.864
Poodle	0.2036	0.821
RC4	0.1895	0.987

The model is able to detect other types of attacks with fairly high accuracy, despite the fact that it was never trained on such attacks. This is probably because the autoencoder is able to encode the useful features which can be used to differentiate different types of traffic and classifier is able to detect if the features which are consistent in the normal traffic still exist in the encoded features.

To better visualize how the autoencoder is doing, as compared to the normal autoencoder that was trained on all 3 attacks, we can perform PCA analysis on the encoded traffic and compare it with the original PCA analysis. We can also take a look at the explained variance for each of the columns, which represents how much each columns contribute to the variance in the encoded features. A high number means that the column is useful in differentiating each of the features, while a low number means that it does not have a large impact.

Table 3: PCA analysis, and explained variance of zero-day detection. The first number is the explained variance of principal component 1, the second is the explained variance of principal component 2.

Without Breach	Without Poodle	Without RC4
		

0.7374404, 0.17805818	0.51650846, 0.315041	0.781878, 0.11523332

It is interesting to note that the PCA analysis for the model trained without RC4 is very similar to the PCA analysis trained with the full model. This might explain why the accuracy for RC4 detection is so high, indicating that there are many similarities with the RC4 attack traffic that are seen in the Breach and Poodle attack traffic.

Discussion

Shape of Input Data

The shape of the input a Keras LSTM layer expects is (sample, time-step, feature). Without a formal introduction to the methods available in the Keras framework, there was confusion as to what exactly constituted the first, second and third dimension of the input. In the end, we decided on taking each file we had, i.e. a single connection to a particular site as a sample. The packets recorded within for the connection would be taken as the time steps in the data. The actual properties of the connection would be taken as the features. However, to model all these into a 3 dimensional matrix required us to take some liberties with the data. For example, to allow us to layer the 2 dimensional matrix (consisting of the time-step and feature) on top of one another to form the third dimension forced us to either pad the files with rows of -1 or completely chop of their rows if it exceeded the average number of rows for all the files we will be looking at.

This is obviously not ideal and perhaps a more fitting method would be to find a number of rows that is at least 3 sigmas away from the mean where we would be able to discount certain files that are known to have extremely lengthy connections i.e. many rows thereby preventing padding of -1 on a massive scale but still be able to keep the information present in the vast majority of samples. However, this would require getting a distribution of the number of rows in our dataset which may be a bit computationally expensive.

Fitting test data

Unfortunately, traffic saved by wireshark differ in the amount of data that is saved. Some traffic had certain features within their dataset while others were missing those features. This posed a problem when performing testing and when doing real-time detection, as the autoencoder would expect certain features in the dataset to be passed in as input, but those same features would be missing in the test data. On the other hand, removing those features that are present in the training data but not in the test data would mean that we were contaminating the training data with information from the test set.

To resolve this issue, we decided to fill missing values with the mean if they were numerical and with the mode if they were categorical. This however meant that the classifier could potentially be performing prediction based on the data that was filled in instead of the data that was present in the test set. While we have tried to mitigate this problem by using the mean and the mode, more could be done to resolve this issue.

Due to the way we cleaned our data by removing columns that appeared in less than 50% of the training data, depending on what training data was used some features could end up being kept but not others. Therefore, in order to ensure that the same process was done for unseen test data, such as the data retrieved from performing real time detection, we had to save the data used in the cleaning stages, such as the entire training data and the columns that we chose to kept into files so that they could be re-used in order to clean the test data.

Feature extraction and Model evaluation

The PCA analysis emphasizes the effectiveness of the LSTM auto-encoder. The features are extracted and reduced to only 50 features out of originally 150 features and these features in higher concept effectively separated normal and attack traffic in the point plots. There is still a problem of false positives in the prediction as normal packets are predicted as intrusive. This problem can be difficult to resolve as these normal packets have similarities with the intrusive packets as can be observed in PCA of all attacks and PCA of attacks without RC4. As RC4 shows the most distinction, the precision in RC4 alone is much higher.

Real-time Detection

The accuracy of the real-time detection model depends on the number of packets read before server sends the result to the model for classification. If the packet size is too small, not enough data is gathered to decide whether the traffic is malicious or normal, which may result in misclassification. alternatively, if the packet size is too large, it will take a longer time to capture the data, which might allow attackers to perform the attack successfully before they get detected and blocked.

The performance of the real-time detection model is also dependent on the amount of preprocessing required. Currently it is doing a lot of pre-processing work of traffic data, so it might take some time for the model to correctly identify a malicious traffic. However, an attack such as Poodle usually takes a relatively long time, and our real-time detection model is able to detect the attack before the attacker successfully gain one byte of sensitive information on the client side.

Another challenge we encounter when designing real-time detection model is that tcpdump, which is the tool we used to dump the traffic between client and server, might hang for a long time without stopping, thus leading to a waste of OS resources. We found out that it is because the command that we are using to dump the traffic data has specified the number of packets in one pcap file, so tcpdump will only stop until enough packets are dumped. This can be solved by setting a timer for every tcpdump command, however, it might result in fragmented pcap files, which would adversely affect the accuracy of classification.

In addition, we found that different ways of browsing websites might also lead to different results. The training data is generated by a simple automated script visiting the homepage of the websites and nothing else, which is different from the behaviour of human when browsing websites. Hence, a possible improvement in the future can be done by training with more human-like normal traffic data and make the model more practical in real-life scenarios.

Conclusion

Through this project, we have modelled and evaluated the use of time series deep learning model to perform intrusive network detection and achieved a decent performance. We have performed pre-processing on the dataset such as removing irrelevant features in pcap files, removing time markings to avoid using learning based on the time the training data was generated and converted 2D dataset into 3D to enable the use of a LSTM autoencoder to analyse features with time series. Using PCA analysis, we evaluated the performance of the LSTM autoencoder in our dataset, including the whole data set and subsets of data set based on different types of attacks. The performance of segmenting normal and intrusive network differences suggests that features related by time can be effectively exploited to identify the normal and intrusive data. The classifier performance is very good with more than 90 percent accuracy in intrusion detection.

Our zero-day testing results suggest the potential for zero-day detection of our model, as the model performs decently for zero day testing of Breach, RC4, and Poodle attack. However, this does not imply it will be successful for all other types of attack and more research for specific types of attack need to be done. We have tried to test the ability of real-time detection by stimulating an attack to a network and allowing receiving packets to be collected in various batch sizes to find the trade-off between the detection accuracy and detection time. We have found out that for real-time detection, bigger batch sizes give better accuracy as they allow the network to capture more packets in a transaction to detect whether the transaction is intrusive. However, that leads to longer delay time which may be detrimental as the attack may have been completed. For this area, further research needs to be carried on to break down to potential effects of these trade-off.

In the future work, fine-tuning of model can be explored by changing parameters inside the neural network such as how many blocks of LSTM encoder-decoder in LSTM autoencoder gives better results and what the limit is that we can witness over-fitting, or changing layers in the classifiers, inclusive of the dropout rate. Moreover, manual feature extraction can be done in further due, such as choosing the subsets of current feature set to pass into the auto encoder and observing the results' variation to pick out the most relevant features, which was unachievable due to the time constraint of the project (longer re-training time than expected). Hence, more comparative research can benefit the exploration of the full potential of the projects.

Acknowledgements

Dr. Lin, Yun (llmhyy@gmail.com), who provided us with the dataset that we are using and inspired us to use a RNN + Autoencoder to perform feature extraction.

References

[1] "Accenture/Ponemon Institute: The Cost of Cybercrime", Network Security, vol. 2019, no. 3, p. 4, 2019. Available: 10.1016/s1353-4858(19)30032-7.
[2] A. Goyal and C. Kumar, GA-NIDS: A Genetic Algorithm based Network Intrusion Detection System. 2008, Available: 10.1.1.568.7035.
[3] E. Conrad, S. Misenar and J. Feldman, "Domain 7", Eleventh Hour CISSP®, pp. 145-183, 2017. Available: 10.1016/b978-0-12-811248-9.00007-3.
[4] F. Farahnakian and J. Heikkonen, "A deep auto-encoder based approach for intrusion detection system", 2018 20th International Conference on Advanced Communication Technology (ICACT), 2018. Available: 10.23919/icact.2018.8323688.
[5] M. A. Salama, H. F. Eid, R. A. Ramadan,A. Darwish, and A. E. Hassanien, "Hybrid IntelligentIntrusion Detection Scheme," inSoft computing industrial applications, pp. 293–303, Springer, 2011.
[6] M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas and J. Lloret, "Conditional Variational Autoencoder for Prediction and Feature Recovery Applied to Intrusion Detection in IoT", Sensors, vol. 17, no. 9, p. 1967, 2017. Available: 10.3390/s17091967.
[7] P. Amoli and T. Hamalainen, "A real time unsupervised NIDS for detecting unknown and encrypted network attacks in high speed network", 2013 IEEE International Workshop on Measurements & Networking (M&N), 2013. Available: 10.1109/iwmn.2013.6663794.
[8] Y. Mirsky, T. Doitshman, Y. Elovici and A. Shabtai, Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection. 2018, Available: arXiv:1802.09089.

Appendix A: Full tabulation of classifier's results

Table 1: Classifier results on different type of attack

Attack Type	Accuracy	F1 score	Precision	Recall
Breach	0.95	0.91	0.83	1.0
Poodle	0.85	0.769	0.625	1.0
RC4	1.0	1.0	1.0	1.0
All	0.946	0.882	0.789	1.0

Appendix B: Computation formulas of metrics used

		Actual	
		Positive	Negative
Predicted	Positive	True Positive	False Positive
	Negative	False Negative	True Negative

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

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

$$F1 = \frac{2 \times precision \times recall}{precision + recall}$$

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

$$specificity = \frac{TN}{TN + FP}$$

Fig 9: Computation formulas of metrics used

Appendix C: Columns used for the autoencoder

'_source_layers_eth_eth.dst',
'_source_layers_eth_eth.dst_tree_eth.addr',
'_source_layers_eth_eth.dst_tree_eth.addr_resolved',
'_source_layers_eth_eth.dst_tree_eth.dst_resolved',
'_source_layers_eth_eth.dst_tree_eth.ig',
'_source_layers_eth_eth.dst_tree_eth.lg',
'_source_layers_eth_eth.padding', '_source_layers_eth_eth.src',
'_source_layers_eth_eth.src_tree_eth.addr',
'_source_layers_eth_eth.src_tree_eth.addr_resolved',
'_source_layers_eth_eth.src_tree_eth.ig',
'_source_layers_eth_eth.src_tree_eth.lg',
'_source_layers_eth_eth.src_tree_eth.src_resolved',
'_source_layers_eth_eth.type',
'_source_layers_frame_frame.cap_len',
'_source_layers_frame_frame.encap_type',
'_source_layers_frame_frame.ignored',
'_source_layers_frame_frame.len',
'_source_layers_frame_frame.marked',
'_source_layers_frame_frame.number',
'_source_layers_frame_frame.offset_shift',
'_source_layers_frame_frame.protocols',
'_source_layers_frame_frame.time',
'_source_layers_frame_frame.time_delta',
'_source_layers_frame_frame.time_delta_displayed',
'_source_layers_frame_frame.time_relative',
'_source_layers_ssl_ssl.record_ssl.alert_message',
'_source_layers_ssl_ssl.record_ssl.app_data',
'_source_layers_ssl_ssl.record_ssl.handshake',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension: SessionTicket TLS (len=0)_ssl.handshake.extension.data',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension: SessionTicket TLS (len=0)_ssl.handshake.extension.len',

'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
SessionTicket TLS (len=0)_ssl.handshake.extension.type',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
ec_point_formats (len=2)_ssl.handshake.extension.len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
ec_point_formats (len=2)_ssl.handshake.extension.type',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
ec_point_formats
(len=2)_ssl.handshake.extensions_ec_point_formats_length',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
ec_point_formats
(len=2)_ssl.handshake.extensions_ec_point_formats_ssl.handshake.
extensions_ec_point_format',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
ec_point_formats (len=4)_ssl.handshake.extension.len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
ec_point_formats (len=4)_ssl.handshake.extension.type',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
ec_point_formats
(len=4)_ssl.handshake.extensions_ec_point_formats_length',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
ec_point_formats
(len=4)_ssl.handshake.extensions_ec_point_formats_ssl.handshake.
extensions_ec_point_format',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
extended_master_secret (len=0)_ssl.handshake.extension.len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
extended_master_secret (len=0)_ssl.handshake.extension.type',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
renegotiation_info (len=1)_Renegotiation Info
extension_ssl.handshake.extensions_reneg_info_len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
renegotiation_info (len=1)_ssl.handshake.extension.len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
renegotiation_info (len=1)_ssl.handshake.extension.type',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
server_name (len=0)_ssl.handshake.extension.len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
server_name (len=0)_ssl.handshake.extension.type',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
status_request (len=5)_ssl.handshake.extension.len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
status_request (len=5)_ssl.handshake.extension.type',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
status_request
(len=5)_ssl.handshake.extensions_status_request_exts_len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
status_request
(len=5)_ssl.handshake.extensions_status_request_responder_ids_le
n',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
status_request
(len=5)_ssl.handshake.extensions_status_request_type',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.comp_
method',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.comp_
methods_length',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.comp_
methods_ssl.handshake.comp_method',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.extensi
ons_length',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.length',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.rando
m',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.rando
m_tree_ssl.handshake.random_bytes',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.rando
m_tree_ssl.handshake.random_time',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.session
_id',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.session
_id_length',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.type',
'_source_layers_ssl_ssl.record_ssl.handshake_ssl.handshake.version
',
'_source_layers_ssl_ssl.record_ssl.record.content_type',
'_source_layers_ssl_ssl.record_ssl.record.length',
'_source_layers_ssl_ssl.record_ssl.record.version',
'_source_layers_tcp.segments_tcp.reassembled.data',
'_source_layers_tcp.segments_tcp.reassembled.length',
'_source_layers_tcp.segments_tcp.segment',
'_source_layers_tcp.segments_tcp.segment.count',
'_source_layers_tcp_Timestamps_tcp.time_delta',
'_source_layers_tcp_Timestamps_tcp.time_relative',
'_source_layers_tcp_tcp.ack',
'_source_layers_tcp_tcp.analysis_tcp.analysis.ack_rtt',
'_source_layers_tcp_tcp.analysis_tcp.analysis.acks_frame',
'_source_layers_tcp_tcp.analysis_tcp.analysis.bytes_in_flight',
'_source_layers_tcp_tcp.analysis_tcp.analysis.initial_rtt',
'_source_layers_tcp_tcp.analysis_tcp.analysis.push_bytes_sent',
'_source_layers_tcp_tcp.checksum',
'_source_layers_tcp_tcp.checksum.status',
'_source_layers_tcp_tcp.dstport', '_source_layers_tcp_tcp.flags',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.ack',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.cwr',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.ecn',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.fin',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.fin_tree__ws.expert__
ws.expert.group',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.fin_tree__ws.expert__
ws.expert.message',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.fin_tree__ws.expert__
ws.expert.severity',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.fin_tree__ws.expert_tc
p.connection.fin',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.ns',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.push',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.res',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.reset',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.reset_tree__ws.expert
__ws.expert.group',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.reset_tree__ws.expert
__ws.expert.message',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.reset_tree__ws.expert
__ws.expert.severity',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.reset_tree__ws.expert
_tcp.connection.rst',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.str',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.syn',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.syn_tree__ws.expert__
ws.expert.group',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.syn_tree__ws.expert__
ws.expert.message',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.syn_tree__ws.expert__
ws.expert.severity',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.syn_tree__ws.expert_t
cp.connection.sack',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.syn_tree__ws.expert_t
cp.connection.syn',
'_source_layers_tcp_tcp.flags_tree_tcp.flags.urg',
'_source_layers_tcp_tcp.hdr_len', '_source_layers_tcp_tcp.len',
'_source_layers_tcp_tcp.nxtseq', '_source_layers_tcp_tcp.options',
'_source_layers_tcp_tcp.options_tree_tcp.options.mss',
'_source_layers_tcp_tcp.options_tree_tcp.options.mss_tree_tcp.opt
ion_kind',
'_source_layers_tcp_tcp.options_tree_tcp.options.mss_tree_tcp.opt
ion_len',

```
'_source_layers_tcp_tcp.options_tree_tcp.options.mss_tree_tcp.opt
ions.mss_val',
'_source_layers_tcp_tcp.options_tree_tcp.options.nop',
'_source_layers_tcp_tcp.options_tree_tcp.options.nop_tree_tcp.opt
ion_kind',
'_source_layers_tcp_tcp.options_tree_tcp.options.sack_perm',
'_source_layers_tcp_tcp.options_tree_tcp.options.sack_perm_tree_
tcp.option_kind',
'_source_layers_tcp_tcp.options_tree_tcp.options.sack_perm_tree_
tcp.option_len',
'_source_layers_tcp_tcp.options_tree_tcp.options.timestamp',
'_source_layers_tcp_tcp.options_tree_tcp.options.timestamp_tree_
tcp.option_kind',
'_source_layers_tcp_tcp.options_tree_tcp.options.timestamp_tree_
tcp.option_len',
'_source_layers_tcp_tcp.options_tree_tcp.options.timestamp_tree_
tcp.options.timestamp.tsecr',
'_source_layers_tcp_tcp.options_tree_tcp.options.timestamp_tree_
tcp.options.timestamp.tsval',
'_source_layers_tcp_tcp.options_tree_tcp.options.wscale',
'_source_layers_tcp_tcp.options_tree_tcp.options.wscale_tree_tcp.
option_kind',
'_source_layers_tcp_tcp.options_tree_tcp.options.wscale_tree_tcp.
option_len',
'_source_layers_tcp_tcp.options_tree_tcp.options.wscale_tree_tcp.
options.wscale.shift',
'_source_layers_tcp_tcp.payload', '_source_layers_tcp_tcp.port',
'_source_layers_tcp_tcp.segment_data',
'_source_layers_tcp_tcp.seq', '_source_layers_tcp_tcp.srcport',
'_source_layers_tcp_tcp.stream',
'_source_layers_tcp_tcp.urgent_pointer',
'_source_layers_tcp_tcp.window_size',
'_source_layers_tcp_tcp.window_size_scalefactor',
'_source_layers_tcp_tcp.window_size_value', '_type', 'class',
'_source_layers_ssl',
'_source_layers_tcp_tcp.analysis_tcp.analysis.flags__ws.expert__ws
.expert.group',
'_source_layers_tcp_tcp.analysis_tcp.analysis.flags__ws.expert__ws
.expert.message',
'_source_layers_tcp_tcp.analysis_tcp.analysis.flags__ws.expert__ws
.expert.severity',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
encrypt_then_mac (len=0)_ssl.handshake.extension.len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
encrypt_then_mac (len=0)_ssl.handshake.extension.type',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
signature_algorithms (len=32)_ssl.handshake.extension.len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
signature_algorithms (len=32)_ssl.handshake.extension.type',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
signature_algorithms (len=32)_ssl.handshake.sig_hash_alg_len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
signature_algorithms
(len=32)_ssl.handshake.sig_hash_algs_ssl.handshake.sig_hash_alg',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
signature_algorithms
(len=32)_ssl.handshake.sig_hash_algs_ssl.handshake.sig_hash_alg_t
ree_ssl.handshake.sig_hash_hash',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
signature_algorithms
(len=32)_ssl.handshake.sig_hash_algs_ssl.handshake.sig_hash_alg_t
ree_ssl.handshake.sig_hash_sig',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
supported_groups (len=10)_ssl.handshake.extension.len',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
supported_groups (len=10)_ssl.handshake.extension.type',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
supported_groups
(len=10)_ssl.handshake.extensions_supported_groups_length',
'_source_layers_ssl_ssl.record_ssl.handshake_Extension:
supported_groups
(len=10)_ssl.handshake.extensions_supported_groups_ssl.handshak
e.extensions_supported_group'
```