I. Project Overview

Cyber security is a topic of major concern for both individuals and corporations. As smart devices become more widespread and pack increasingly advanced connectivity, the usage of technology in our lives has increased at an exponential rate. While this has led to increased convenience and productivity, this phenomenon has also exposed our privacy and sensitive information to attackers in new ways.

As compared to the average consumer, companies fare no better against cyber-attacks. Research has shown that while 62% of businesses experienced phishing and social engineering attacks in 2018, only an average of 5% of companies' folders are properly protected (Sobers, 2020). These attacks can have devastating impacts on companies' financials as well, with the infamous Equifax breach in 2017 affecting nearly 150 million consumers worldwide and costing the company more than US$4 billion in total (Sobers, 2020). There is therefore a pressing need to efficiently identify malicious attacks before they deal irreparable damage to companies and individuals.

II. Problem Statement

The objective of this project is to leverage the power of machine learning to produce a classifier that can quickly and accurately decide if a given program is malicious in nature or not. Machine learning is ideal for this task because of its ability to recognize patterns accurately. This is also supported by the multiple papers that have successfully used machine learning to enhance existing cybersecurity protections (Berman, Buczak, Chavis, & Corbett, 2019). Deploying a machine learning classifier would also be an automated process that requires less manpower and is thus more sustainable. Since Windows is the leading desktop operating system in use worldwide with approximately 77% of the market share (Statcounter, 2020), concentrating on detecting malicious software, or malware, in Windows provides the greatest impact and reach. Windows executables use calls to application programming interfaces (APIs) to communicate with the system and accomplish tasks such as getting user input, creating and reading files, and connecting to internet sites (Microsoft, 2018). This hence provides a potential way for malware to be detected; by observing the API calls made by an application, it is possible to generalise the application's behaviour, and determine if the application is acting maliciously. These API calls can be recorded by running the program in a controlled environment such as Cuckoo Sandbox so that any malicious behaviour does not adversely affect the actual host machine.

The problem was broken down into two separate stages; given a sequence of API calls exhibited by the program being tested, a binary classifier predicts if the program is malicious or not. This requires the classifier to be trained on an API call sequence dataset of length n that has binary labels, henceforth referred to as Dataset 1. If the program is predicted to be malicious, the same sequence of length n would then be passed to the second stage, where a multi-class classifier tries to predict the type of malware that the given program is. This would require a separate API call sequence dataset that has labels corresponding to various types of malware., henceforth referred to as Dataset 2. This workflow is visualised in Figure 1 on the next page, while Datasets 1 and 2 are covered in greater detail in the following sections. By separating the prediction process into two stages, programs that are predicted to be non-malicious need not be re-processed and sent for a second prediction by the multi-class classifier. This hence allows for a quicker response to the user for non-malicious programs, improving the user experience and making users overall more receptive to using the malware prediction service by minimising inconvenience and waiting time.

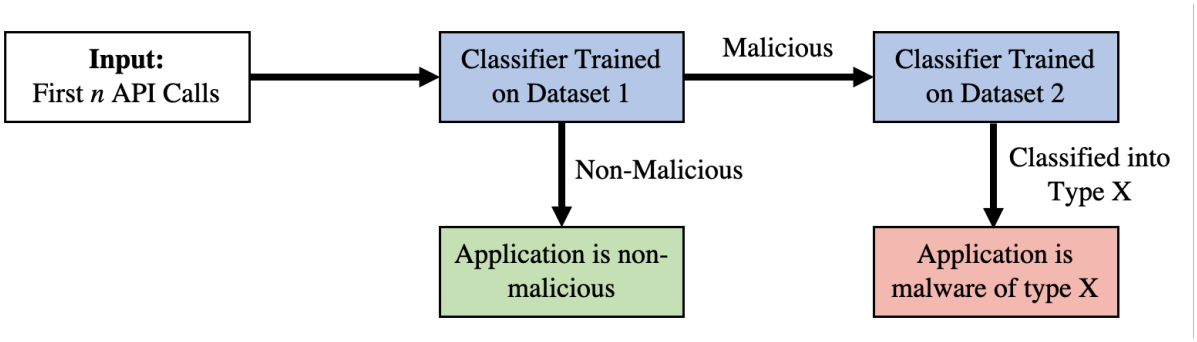

Figure 1. Proposed Pipeline for Malware Classification

III. Algorithms and Techniques

The data used for this project is ordered by time, since it consists of the first *n* API calls made by a program when run. Given that these API calls are sequential in nature, it is crucial to select a machine learning algorithm that can exploit this to produce optimal prediction performance. Consequently, a recurrent neural network (RNN) was decided on as the model of choice.

The RNN structure is specially designed to process sequential data. RNNs comprise multiple identical cells linked to each other in a chain. Each cell receives a memory state of the data that was previously seen in addition to the input token at that time-step, as seen in Figure 2 below. This internal memory maintained within the RNN allows the network to remember the inputs at previous time-steps (Jayawardhana, 2020), hence helping it determine the patterns within the data to help make suitable predictions.

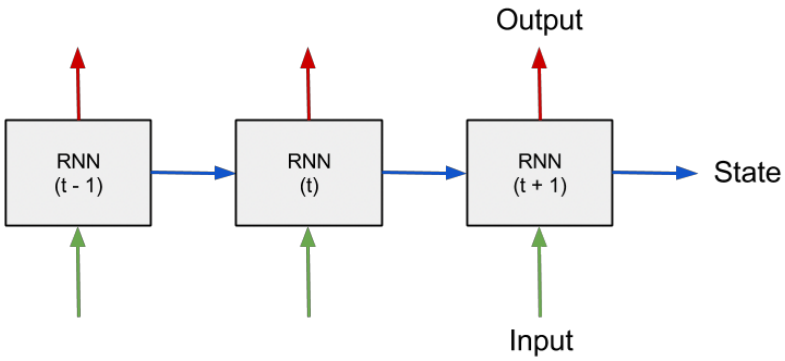

Figure 2. Structure of RNN, Reproduced from (Hallstrom, 2020)

While RNNs are able to keep track of context from previous inputs that are close-by in proximity to a token *t*, information from a token that occurs far away from the predicted token at time-step *t* is largely forgotten. This is illustrated in an example provided at (Olah, 2015): The prediction of the next word w_t in a long sentence such as "I grew up in France… I speak fluent <w_t>". A human reading the full sentence would be able to guess that the appropriate word is "French" based on the character growing up in France as seen earlier in the sentence. However, this poses a problem for RNNs, since it would have forgotten the context from the token "France" that occurred significantly earlier on in the sentence. This is also referred to as the vanishing gradient, since the gradient containing context from the token "France" gradually diminishes as we propagate through the cells across the sentence.

Long-short term memory (LSTM) networks are an evolution of the traditional RNN structure that specifically deal with this problem. LSTM networks replace the single memory state featured in RNNs with a more complicated set of forget, input, and output gates. These three gates are used to control the long-term memory represented as the cell state c_t as it propagates through the cells. The short-term memory represented as the hidden state h_t is also passed to the neighbouring cells (Ranjan, 2019). An LSTM cell structure is shown in Figure 3 below. Given the generally superior performance of RNN and LSTM networks when dealing with sequential data, both algorithms were tested with the datasets as expanded on in the following sections of this paper.

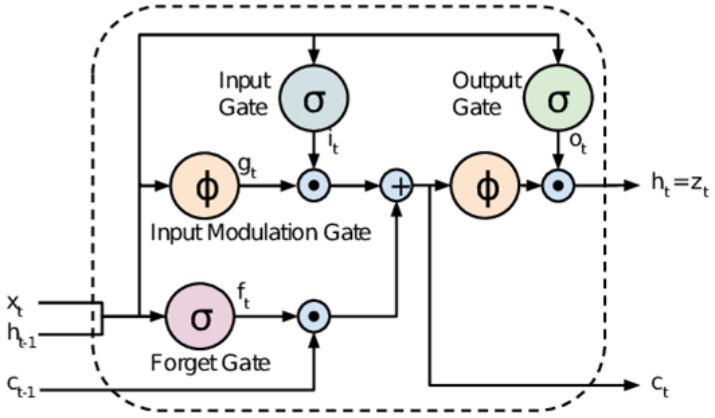

Figure 3. LSTM Cell Structure, Reproduced from (Jayawardhana, 2020)

IV. Choice of Metrics for Performance Comparison

When measuring the performance of classifiers and machine learning algorithms, accuracy is the most commonly used metric. Accuracy measures the proportion of examples that the classifier correctly predicted. While this intuitively makes sense, accuracy alone is not enough to showcase the true performance of a given classifier. The classification matrix in Figure 4 below, reproduced from (Koo, 2018), demonstrates this problem.

		Predicted/Classified	
		Negative	**Positive**
Actual	**Negative**	998	0
	Positive	1	1

Figure 4. Classification Matrix Illustrating Accuracy Problem, Reproduced from (Koo, 2018)

In the classification matrix above, the classifier would be given an accuracy of 0.999, which is an excellent result. However, when looking at the class breakdown, it is observed that there is a severe class imbalance, with 99.8% of the data belonging to one class and only 2 examples belonging to the other class. When considering only the positive examples, the accuracy drops significantly to 50%. This can also be applied to the problem explored in this project; of 1000 API call sequences, 998 of them are non-malicious, while 2 of the programs are malicious. Given this classification matrix, we would have missed 1 malicious program, which could possibly have significant detrimental impact on our users. It is therefore important to consider all classes (positive and negative in this case) fairly, hence other metrics in addition to accuracy can be employed to give the full picture of the classifier performance.

Two other metrics that are commonly used to describe classification performance are precision and recall. These are defined in the formulae as follows using the classification matrix:

	Predicted Positive	**Predicted Negative**
Actually Positive	True Positive (TP)	False Negative (FN)
Actually Negative	False Positive (FP)	True Negative (TN)

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

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

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

Summarised in words, precision measures the proportion of data points predicted as positive by the classifier that are actually positive. On the other hand, recall measures the proportion of actually positive data points that are correctly predicted by the classifier. Given the classification matrix in Figure 4, the recall metric would have a value of 0.5, which would have signalled a problem that needs further investigation, which would in turn have revealed the severe class imbalance. In general, for an ideal testing scenario, good classifier performance should give a high value of precision, recall, and accuracy simultaneously. With this in mind, the F1 score metric is also introduced. The F1 score provides a single score that combines the precision and recall metrics. Since precision and recall should both be high for a well-performing classifier, a single high F1 score can be taken as representative of this.

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

In the context of this project, imbalanced datasets are common because non-malicious programs are more easily accessible and collectible as compared to malware samples. This therefore makes accuracy a poor metric to use if the class imbalance is not first taken care of. This can be remedied by computing the accuracy for each individual class before taking the average over all classes to report as the final accuracy. Nonetheless, the F1 score is taken to be the most important metric for determination of classifier performance because it is more robust than accuracy, and also accounts for both precision and recall in its computation.

V. Construction of Binary Malware Classifier with Dataset 1

The first dataset to be used was sourced from Kaggle (Oliveira, 2019). The dataset consists of the first 100 non-repeated API calls made by both malware and non-malicious programs as captured with Cuckoo Sandbox. In total, there were 42,797 malware samples and 1,079 non-malicious program samples. The data is provided in a CSV file with 102 columns, comprising of the original program's hash value, 100 integers representing the API calls, and the class label for each example. A binary integer is provided as the class label, with "0" denoting that the example is non-malicious, and "1" denoting that the example is malicious.

The Pandas library in Python was used to parse the CSV file and retain only the sequence of API calls to make up the machine learning model's data input, while the labels were kept separately as the model's data output. The integers representing the API calls were renumbered to start from 0 so that the model would be compatible with the Embedding layer in Keras, which was used in the construction of the model. It was hence discovered that there was a total of 264 unique API calls found in the dataset. The data was then split into training and testing sets, with the training set comprising 80% of the full dataset.

V.I. Baseline Model

The baseline model was sourced from the original paper published by the dataset author. The paper reported that a long-short term memory (LSTM) network provided the best performance with an F1 score of 0.9953 (Oliveira & José Sassi, 2019). The best-performing LSTM network was thus taken as the baseline and recreated using Keras, and a batch size of 128 was used as per the paper. The model was then trained for 30 epochs to emulate the actions taken in the paper. The baseline model structure is shown in Figure 5 below.

Data Input Sequence of 100 API Calls
Embedding Layer Input Dimension: 264, Output Dimension: 70
LSTM Layer Hidden Dimension: 70
Dropout Layer Dropout Rate: 0.5
Sigmoid Layer
Data Output Float between 0 and 1

Figure 5. Baseline Model Structure for Dataset 1

Given the large class imbalance in the dataset (97.5% of the data was malicious), the class weight was altered to account for this while fitting the model. Early stopping was also implemented to avoid overtraining and overfitting the model. Although the paper reported that the model was able to produce an F1 score of 0.9953, the replication in Keras was only able to produce an F1 score of 0.96523. This disparity in results can be attributed to a different splitting of the train and test datasets; however, since the split datasets were not provided by the authors, there is no way to fully replicate the results. On the other hand, an F1 score of 0.96523 still represents good accuracy from the model.

V.II. Hyperparameter Tuning

To further improve on the performance of the classifier, some ideas to alter the model structure were tested. While the model used an LSTM structure, a vanilla RNN structure was also tested to verify the impact of using a more advanced LSTM network on this dataset. While LSTM networks are generally an improvement over the basic RNN structure to allow them to retain more information across long sequences (Olah, 2015), they might be too complex if the dataset is relatively easy to process. Another test that was done was to use a bidirectional RNN / LSTM network. Bidirectional networks comprise of two different networks that improve the overall predictive power and increase the context available to the model by considering the sequences in both directions (Brownlee, 2017).

Another potential improvement was to add an additional dense layer between the LSTM and sigmoid layers. This would make the model "deeper", and potentially allow it to better recognize the patterns in the data that would allow it to separate the two classes better. The hidden dimension affecting the number of hidden units in the LSTM and the embedding layer was also modified, and values of 30 and 100 were tested in addition to the original value of 70. Similar to the number of hidden layers, having more hidden units increases the ability of the model to process the data. However, if the model is already overfitting on the training data, using a smaller number of hidden units would help to reduce the overfitting (Goodfellow, Bengio, & Courville, 2016).

To test the combination of these different hyperparameters, the "GridSearchCV" function found in SciKit-Learn was used. The function runs a brute-force, exhaustive search over all specified hyperparameter values, and returns the best-performing model based on the training loss value. After the hyperparameter tuning, the best parameters that were found are shown in Table 1, while the performance comparison with the baseline model is shown in Table 2.

Hyperparameter	Baseline Model	Tuned Model
RNN / LSTM	LSTM	LSTM
Bidirectional	No	Yes
Additional Dense Layer(s)	No	1
Hidden Dimension	70	30

Table 1. Hyperparameter Comparison between Tuned Model and Baseline Model

Metric	Baseline Model	Tuned Model
Accuracy	0.93437	0.98883
Precision	0.99490	0.98978
Recall	0.93728	0.98883
F1 Score	0.96523	0.99428

Table 2. Performance Comparison between Tuned Model and Baseline Model

From the results in Table 2, it is clear that the tuned model provides significantly better performance on the same test data as compared to the baseline model. The tuned model provides a higher accuracy and F1 score, although the precision is slightly lower. By looking at the difference in hyperparameters as shown in Table 1, it can be concluded that using the bidirectional LSTM layer and making the network deeper helped the network better recognise the patterns within the API calls that made a program malicious. On the other hand, using a smaller number of hidden units helped to reduce overfitting.

V.III. Selection of Optimal Number of API Calls

While the performance of the tuned classifier was satisfactory, another improvement that was considered was to reduce the number of API calls required by the classifier to produce good classification accuracy. The current classifier uses the first 100 API calls to make its prediction; if this number can be decreased, the classifier would be able to produce a prediction result in a shorter period of time after the program is run. This would be a large benefit to users as it would limit the damage that can be done by a piece of malware before it is detected. Tests were therefore run with the classifier being given the first n API calls as input data, where n was taken to be an integer between 10 and 100 inclusive in steps of 5. The same hyperparameters as the previously tuned model were used to test the classification ability. The results of these tests are plotted and shown in Figure 6 on the next page.

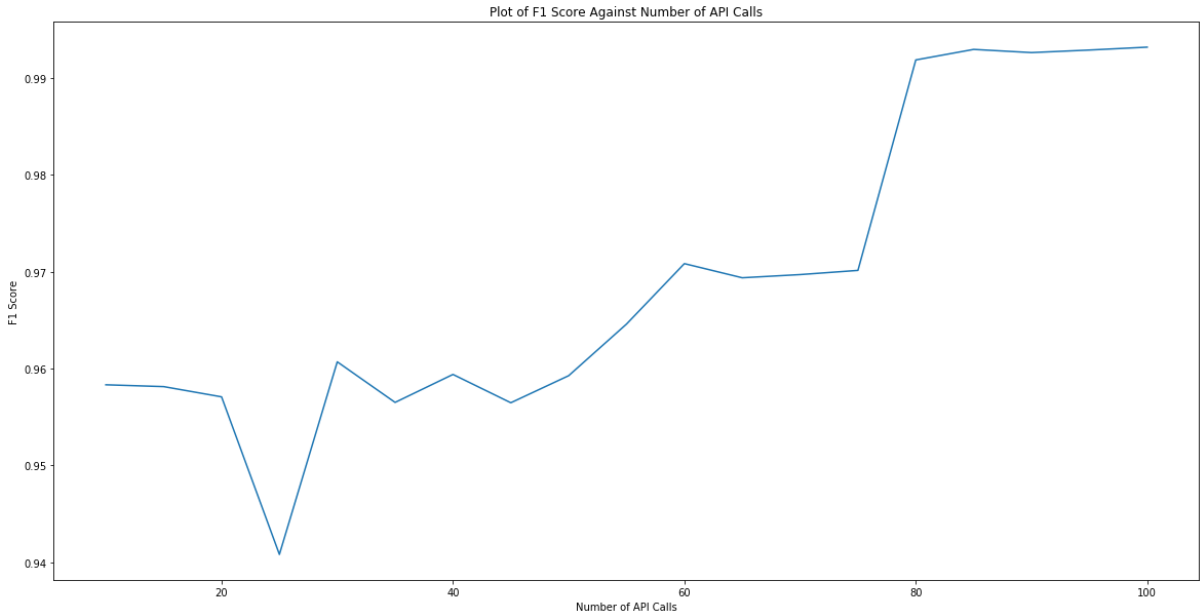

Figure 6. Plot of F1 Score Against Number of API Calls in Data Input

As expected, there is a general improvement in the F1 score with a larger number of API calls available to the classifier. This is because having more API calls provides the model with more context and improves its ability to recognise the patterns within the API calls that represent malicious behaviour. However, a surprising discovery is that the classification performance is relatively similar for the models that are given 80 to 100 API calls. This suggests that the number of API calls given to the model can be reduced to 80 with no significant performance penalty. As a result, the optimal number of API calls to be used, *n*, was determined to be 80. This represents a significant improvement, as malware can be detected using approximately 20% less time as compared to previously. This model was subsequently saved for future deployment and uploaded to GitHub as "final_model_dataset1.h5".

VI. Construction of Multi-Class Malware Classifier with Dataset 2

Similar to Dataset 1, the second dataset was also sourced from Kaggle (Catak, 2019). The dataset consists of sequences of API calls made by a total of 7,107 malware samples. The API calls were recorded using the same software as the first dataset (Cuckoo Sandbox). In contrast to the binary labels provided in Dataset 1, this dataset separates the different samples into a total of 8 unique categories based on their behaviour. These categories are taken as the majority label given to the program by VirusTotal, an online service that consolidates results given by multiple antivirus solutions to determine if a given program is malicious or not. This process is further elaborated on in the paper written by the dataset author (Catak & Yazi, 2019).

Similar to the pre-processing steps used with the first dataset, the integers representing the API calls were first checked to see their range. It was noted that the API call numbers ranged from 1 to 340, making them suitable for use with the Keras embedding layer. Further exploration of the data revealed that the samples in the dataset had very different lengths of API call sequences. This is visualised in Figure 7 on the next page. The minimum length of the sequences observed was 10, while the maximum length of the sequences in the dataset was 1,764,421 API calls.

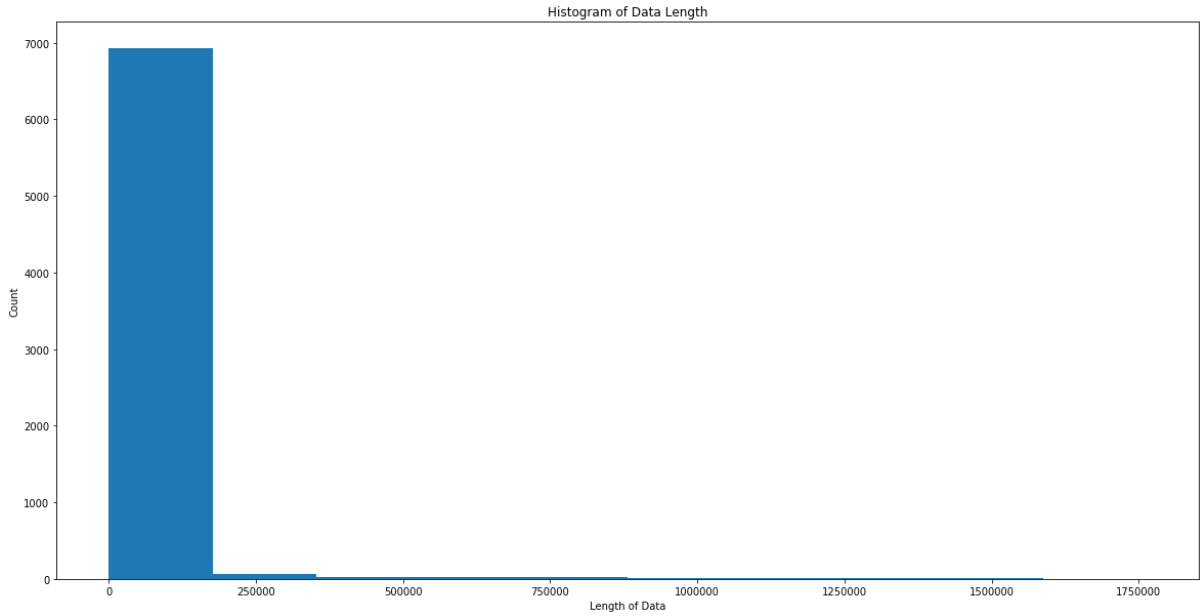

Figure 7. Distribution of API Call Sequence Lengths in Dataset 2

Given the wide range of sequence lengths for the data, it was necessary to further process the data before using it with the machine learning model. Since it was determined previously in section V.III that only 80 API calls were needed to provide an accurate prediction of whether a program is malicious or not, it was decided that a sequence of 80 API calls would also be tested with Dataset 2. This is also driven by the need for the number of API calls between dataset 1 and dataset 2 to be equal, since a user should be able to pass the same API call sequence to both classifiers as seen in the workflow previously described in Figure 1. The composition of the data after pre-processing is shown in Table 3 below.

Class Label	**Number of Data Points**
Trojan	931
Backdoor	932
Downloader	890
Worms	926
Spyware	767
Adware	334
Dropper	756
Virus	750
Total	**6,286**

Table 3. Composition of Pre-Processed Dataset 2

After obtaining a dataset of 80 API call sequences, the dataset was split into train and test sets. As the number of examples per class was not equal, a stratified split was used to produce a train set with 80% of the examples of each class, and a test set with 20% of the examples of each class. This ensures that the model has enough data of each class to train on, as compared to a random split that might cause the model to have insufficient examples of some classes to train on, thus reducing its ability to predict examples belonging to those class labels.

VI.I. Baseline Model

Although the dataset authors released a paper detailing their machine learning approach to this multi-class problem (Yazi, Çatak, & Gül, 2019), the hyperparameters used were not fully listed in the paper. Apart from knowing that an LSTM model was used, no other information was available. Hence, to construct a baseline model for comparison, similar hyperparameters were used as the model for Dataset 1. An LSTM layer with 70 hidden units was constructed, and a dropout layer with a dropout rate of 0.5 was included. To allow the model to learn for a longer period of time, the model was allowed to train for 100 epochs with a batch size of 128, although early stopping was also used to prevent overtraining. The results detailed in the paper, as well as the results obtained from this baseline model, are shown in Table 4 below.

Metric (Average over Classes)	Paper's Findings	Trained Baseline Model
Accuracy	0.890	0.229
Precision	0.601	0.473
Recall	0.524	0.285
F1 Score	0.571	0.355

Table 4. Performance Comparison between Paper Results and Baseline Model

The large disparity in performance observed in Table 4 can be largely attributed to the different hyperparameters employed by the models. As there is no information on the hyperparameters used in the paper, there is no way to fully emulate the reported results. In addition, the difference in performance could also be due to the change in sequence lengths; while the data input was limited to 80 API calls per data point, this limitation was not enforced in the paper. Additionally, as seen in Table 3, this also caused the amount of usable data for model training to decrease by more than 10% from the original number of samples in the full dataset (7,107). As a result, the models trained by the paper would have had more context and thus been able to predict with significantly greater accuracy. Nonetheless, the baseline model acts as a starting point on which improvements are made in the following sections.

VI.II. Hyperparameter Tuning

Similar to the steps taken to improve the machine learning model's performance on Dataset 1, hyperparameter tuning was employed to find the optimum set of hyperparameters for use with Dataset 2. The hyperparameters that were altered were largely similar to those used with Dataset 1; hidden dense layers were added after the LSTM layer, an RNN layer was swapped with the LSTM layer, and a bidirectional LSTM layer was also tested. In addition, the number of hidden units was also altered. Since the original hyperparameters used in the paper were not available, a larger number of different hyperparameters were tested to try and find the optimal set for the best performance. The best hyperparameters and the performance comparison with the original baseline model are shown in Tables 5 and 6 respectively.

Hyperparameter	Baseline Model	Tuned Model
RNN / LSTM	LSTM	RNN
Bidirectional	No	No
Additional Dense Layer(s)	No	2
Hidden Dimension	70	30

Table 5. Hyperparameter Comparison between Tuned Model and Baseline Model

Metric (Average over Classes)	Baseline Model	Tuned Model
Accuracy	0.229	0.341
Precision	0.473	0.384
Recall	0.285	0.343
F1 Score	0.355	0.362

Table 6. Performance Comparison between Tuned Model and Baseline Model

The change in hyperparameters from the baseline model to the tuned model suggests that the original baseline model was overfitting to the data. As a result, steps were taken to reduce the predictive power of the model, such as changing from an LSTM layer to an RNN layer and reducing the number of hidden units. Overall, these changes helped the model perform better on the unseen test data, as can be seen by the large improvement in accuracy in Table 6. The tuned model was subsequently saved for future deployment and uploaded to GitHub as "final_model_dataset2.h5".

VI.III. Testing with Different Number of API Calls

As the previous model trained on Dataset 2 only used 80 API calls per example, an idea for improvement was to include a larger number of API calls for each example. However, this also causes the number of available data points to decrease, since some data points in the dataset have relatively shorter API call sequences. Nonetheless, models were trained using 85, 90, 95, and 100 API calls to verify if there is a significant improvement in the classifier performance. The maximum number of API calls was kept as 100 since this second classifier would have to be compatible with the data input to the first model, which has a maximum of 100 API calls. The same hyperparameters from the tuned model were used for every model, and the results are shown in Table 7 below and visualised in Figure 8 as follows.

Metric (Average over Classes)	80 API Calls	85 API Calls	90 API Calls	95 API Calls	100 API Calls
Dataset Size	6,286	6,223	6,163	6,114	6,049
Accuracy	0.341	0.372	0.322	0.349	0.317
Precision	0.384	0.442	0.392	0.374	0.376
Recall	0.343	0.367	0.314	0.379	0.310
F1 Score	0.362	0.401	0.349	0.377	0.340

Table 7. Performance Comparison with Varying Number of API Calls

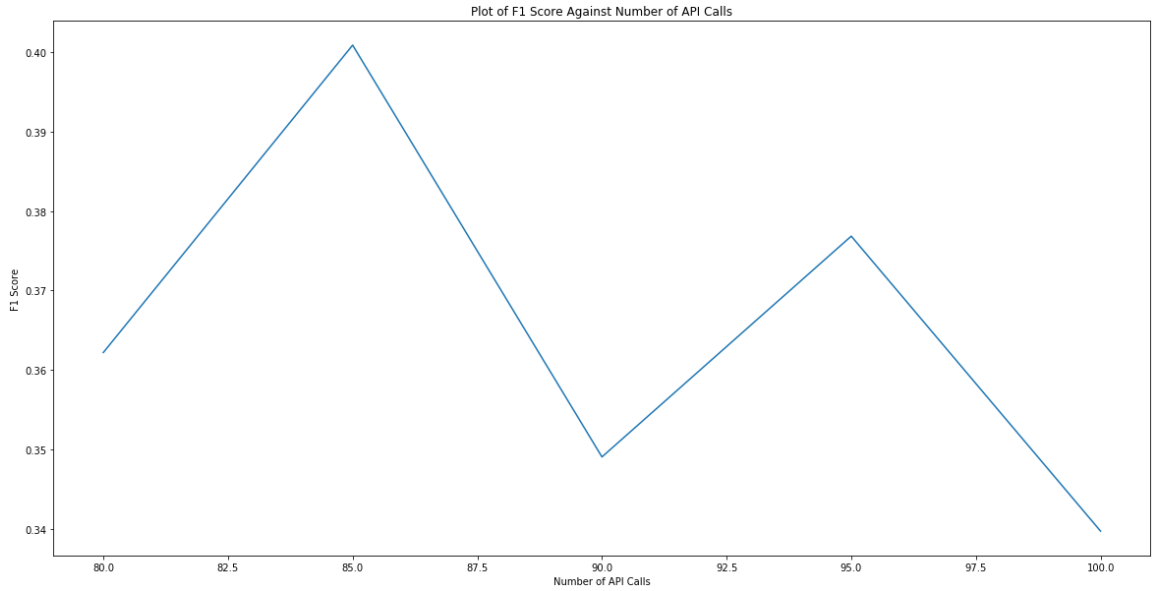

Figure 8. Plot of F1 Score Against Number of API Calls

The results shown in Table 7 and Figure 8 suggest that there is no clear improvement when the number of API calls is increased. While the models trained with 85 and 95 API calls performed better than the original model trained with 80 API calls, it was also noted that when retraining the models, the performance fluctuated greatly. As a result, the models trained with more API calls did not reliably perform better than the original tuned model. It was therefore determined that adding more API calls would not lead to a significant improvement in the model performance. 80 API calls was thus kept as the dataset size to balance model performance with quick predictions as explained previously in Section V.III.

VI.IV. Testing with Different Problem Setup

As the results from the hyperparameter tuning were still relatively poor, with a final F1 score of 0.362, and an increase in the number of API calls available in the data input did not significantly help the model performance, a change in the setup of the problem was proposed. Instead of training a single model to distinguish between the 8 different classes, another way of tackling this problem would be to train 8 independent binary classifiers.

With this setup, classifier *x* would hence be responsible for predicting whether a program with a given API call sequence belongs to class *x* or not. This makes the problem easier for each classifier, which hopefully makes each classifier better able to determine the unique patterns and behaviour exhibited by each class. All 8 predictions are then used for the final prediction; the classifier that has the most confidence in the program belonging to its class would result in the data point being classified into that label. Each classifier was tuned independently to each other to ensure that each classifier would be optimally fit to the individual class. The results of the classifiers are shown in Table 8 as follows.

Metric	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7
Accuracy	0.852	0.847	0.875	0.852	0.878	0.171	0.880	0.948
Precision	0.000	0.368	0.800	0.333	0.000	0.060	0.000	0.947
Recall	0.000	0.037	0.157	0.005	0.000	0.985	0.000	0.593
F1 Score	0.000	0.0680	0.263	0.0106	0.000	0.112	0.000	0.730

Table 8. Performance of Tuned Individual Binary Classifiers

The results in Table 8 suggest that this approach might not be well-suited for this particular problem. The results highlighted in yellow are particularly worrying, since the classifiers for classes 0, 4, and 6 predicted all examples as non-malicious, hence resulting in a zero score for precision, recall, and F1 score. This would have serious ramifications if deployed in a production environment, since the classifiers would basically miss all malware of those 3 types. This is likely to be a result of severe class imbalance, since some classes have significantly lesser examples as compared to other classes. Different weights were used while fitting the model to account for this, although it appears to be insufficient.

In addition, the results for the other classes are relatively poor with the exception of the classifier for class 7. These results suggest that viruses (class 7) are relatively unique in terms of their behaviour as compared to other types of malware, which are more difficult to separate. Overall, the poor results obtained with this approach suggests that the original approach might be more well-suited to this particular problem.

VI.V. Testing with Different ML Model Types

While going back to the multi-class model, another potential way to improve the overall model performance is to try different model types apart from neural networks. This is further supported by the findings in Table 5, where it was determined that the model favoured hyperparameters that reduced the overall predictive power. This suggested that the model was overfitting, and hence it might be useful to consider other model types that are less complex as compared to RNNs and other forms of neural networks.

While there are multiple possible machine learning algorithms to choose from, popular algorithms such as the decision tree classifier, the random forest classifier, and the support vector machine (SVM) algorithm were eventually decided upon. These algorithms were chosen because they commonly provide good performance, and also provide a good spread in complexity, with the decision tree classifier being one of the least complex machine learning algorithms around. As with the previous neural networks, multiple hyperparameters were tested for each algorithm, such as the number of splits for the decision tree classifier, the kernel used and regularisation strength for the SVM algorithm, and the number of classifiers used for the random forest algorithm. After hyperparameter tuning was done, the performance of these algorithms is shown in Table 9 as follows.

Metric (Average over Classes)	Neural Network	Decision Tree	SVM	Random Forest
Accuracy	0.341	0.370	0.336	0.417
Precision	0.384	0.385	0.450	0.417
Recall	0.343	0.398	0.363	0.445
F1 Score	0.362	0.391	0.402	0.430

Table 9. Performance Comparison between Machine Learning Algorithms

From the results observed in Table 9, it is clear that the neural network performs the worst as compared to the other algorithms judging by the F1 score. The random forest performs the best with the highest accuracy and F1 score and can be said to be the best model. The results further show that the neural network model was too complex for this problem type despite the hyperparameter tuning that reduced the overall complexity. Being the best classifier tested with Dataset 2, the random forest model was saved as “random_forest_model_dataset2” for future use and deployment.

An important takeaway from these experiments is that using the most powerful machine learning algorithm or model might not always produce the best performance. Testing multiple algorithms before selecting the best model will give the best classifier for each problem since there is no one-size-fits-all solution.

VII. Conclusion

Through the course of this project, two separate machine learning models were trained on two datasets to provide a two-stage malware detection classifier. With the use of hyperparameter tuning and other optimisation tools, the binary classifier trained on the first dataset was able to provide very good performance (F1 score of 0.994). The length of the input data sequence was also successfully reduced without significant impact to the model performance, hence improving the user experience. While the models trained on the second dataset did not produce results that were as good as the first model, there was a lot to learn through the process of optimising the models' performance on the second dataset. By testing multiple different algorithms and various problem setups, the best-performing algorithm was successfully found.

VIII. References

Sobers, R. (2020, September 24). *110 Must-Know Cybersecurity Statistics for 2020*. Retrieved from Varonis: https://www.varonis.com/blog/cybersecurity-statistics/

Statcounter. (2020, September). *Desktop Operating System Market Share Worldwide*. Retrieved from Statcounter: https://gs.statcounter.com/os-market-share/desktop/worldwide

Microsoft. (2018, May 31). *Windows API Index*. Retrieved from Microsoft: https://docs.microsoft.com/en-us/windows/win32/apiindex/windows-api-list

Oliveira, A., & José Sassi, R. (2019). Behavioral Malware Detection Using Deep Graph Convolutional Neural Networks. *TechRxiv*.

Oliveira, A. (2019). *Malware Analysis Datasets: API Call Sequences*. Retrieved from Kaggle: https://www.kaggle.com/ang3loliveira/malware-analysis-datasets-api-call-sequences

Olah, C. (2015, August 27). *Understanding LSTM Networks*. Retrieved from Colah's Blog: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Brownlee, J. (2017, June 16). *How to Develop a Bidirectional LSTM For Sequence Classification in Python with Keras*. Retrieved from Machine learning Mastery: https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras/

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning Book.* MIT Press. Retrieved from http://www.deeplearningbook.org/

Catak, F. (2019). *API Call Based Malware Dataset*. Retrieved from Kaggle: https://www.kaggle.com/focatak/malapi2019

Catak, F., & Yazi, A. (2019, May). *A Benchmark API Call Dataset For Windows PE Malware Classification.* Retrieved from ResearchGate: https://www.researchgate.net/publication/332877263_A_Benchmark_API_Call_Dataset_For_Windows_PE_Malware_Classification

Yazi, A., Çatak, F., & Gül, E. (2019). Classification of Methamorphic Malware with Deep Learning (LSTM). *2019 27th Signal Processing and Communications Applications Conference (SIU).* Sivas: IEEE.

Berman, D., Buczak, A., Chavis, J., & Corbett, C. (2019, April 2). *A Survey of Deep Learning Methods for Cyber Security.* Retrieved from Multidisciplinary Digital Publishing Institute: https://www.mdpi.com/2078-2489/10/4/122/pdf

Hallstrom, E. (2020, July 11). *How to Build a Recurrent Neural Network in TensorFlow*. Retrieved from Educaora: https://educaora.com/@erikhallstrom/How_to_build_a_Recurrent_Neural_Network_in_TensorFlow

Jayawardhana, S. (2020, July 27). *Sequence Models & Recurrent Neural Networks (RNNs)*. Retrieved from Towards Data Science: https://towardsdatascience.com/sequence-models-and-recurrent-neural-networks-rnns-62cadeb4f1e1

Ranjan, A. (2019, November 2). *Understanding the Flow of Information through LSTM Cell*. Retrieved from Medium: https://medium.com/@ashish.cse16/understanding-the-flow-of-information-through-lstm-cell-4b8eee2c4c9d

Koo, P. (2018, March 15). *Accuracy, Precision, Recall or F1?* Retrieved from Towards Data Science: https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9