Detecting Malicious PowerShell Scripts Using Deep Neural Networks

Michael Lester
Georgia Tech
North Ave NW, Atlanta, GA 30332
mlester30@gatech.edu

Willie Zeng
Georgia Tech
North Ave NW, Atlanta, GA 30332
wzeng40@gatech.edu

Xiwei Chen
Georgia Tech
North Ave NW, Atlanta, GA 30332
xchen742@gatech.edu

Kumar Lama
Georgia Institute of Technology
North Ave NW, Atlanta, GA 30332
klama3@gatech.edu

Abstract

In this paper, we evaluate and compare various Neural Network designs for detecting malicious PowerShell scripts and present a new application of the Delta Debug algorithm for identifying key components of the original text that the model relies upon for classification. We then evaluate how well the best model was able to generalize to obfuscation techniques not present in the training or validation set. Overall, the fully convolutional neural network model had the best accuracy at 98.8% classification accuracy and a 1.5% false positive rate (FPR); however, the ensemble method was the best model for this domain with 95.1% accuracy and a 0.0% FPR. The ensemble model also showed significant improvements in robustness over hand crafted signatures and performed faster than Windows Defender at classification tasks.

1. Introduction/Background/Motivation

Advanced Persistent Threat actors (APTs) rely heavily on script-based execution methods for initial access, persistence, command and control, lateral movement, and exfiltration. Because these methods can run completely in memory, they are immune to some of the defenses of modern Antivirus engines. Microsoft released a new API called the Anti-Malware Scan Interface (AMSI) which programs like PowerShell use to pass commands to the installed AV before executing the script. While this API at least provides an opportunity to thwart script-based attacks, Antivirus engines have proven woefully inadequate at preventing obfuscated scripts from being executed on the system. Frameworks such as Invoke-Obfuscation can be used to mitigate static signatures in malicious PowerShell scripts without altering the functionality of the script. Bypassing modern anti-virus engines is trivial with the use of open source obfuscation techniques. Research by FireEye in 2017 yielded a proof-of-concept classifier called Revoke-Obfuscation that achieved a 96% classification accuracy on a custom-built corpus of PowerShell scripts; however, due to the high level of false positives, that script is not production ready [1]. The purpose of this project is to develop a deep learning model that provides highly effective defenses against plain or obfuscated PowerShell scripts. Significant improvements in this area of research could lead to better protections against initial access and post-exploitation activities by cyber criminals and nation-state threat actors.

The goal of this project was to construct and train a deep learning model to distinguish between legitimate and malicious files with a greater than 99.0% accuracy rate, greater than 99.9% TP rate on plaintext and obfuscated AMSI suppression techniques, and less than 0.1% FP rate.

2. Approach

Our approach was to experiment with various different architectures in order to identify the best performing model and evaluate that model against a set of malware generated with obfuscation techniques the model has never seen in order to determine how well the model was able to generalize. Best practices for machine learning were maintained throughout such as the use of separate unique train, validation, and test sets. The Delta Debug algorithm was used to gain insight into how the model was classifying samples and to evaluate how well the model was able to generalize. The architectures we looked at include a fully connected classifier, convolution, complex convolution, long short-term memory, and attention network.

3. Anticipated Issues

From a classification perspective, there are some malicious scripts that will look and feel similar to clean scripts. In addition, there are some scripts that can be used for both legitimate and nefarious purposes, so the only distinguishing factor is the context of execution of the script which is information our model will not have since it can only look at the file itself. Our model has to be able to detect small amounts of malicious code in large legitimate scripts. A 30-character cradle embedded in a 5 MiB PowerShell script is still a malicious PowerShell script. Data size will be a significant issue since our model needs to be able to handle text files anywhere from a few characters in length to hundreds of mebibytes in length. Due to memory constraints, files larger than 64 MiB were not passed through training; however, the test set was not limited by length. Our model also needs to be able to detect non-obfuscated malicious code, which will involve some amount of rote memorization. We need to be able to visualize what the model is alerting on so that we can evaluate whether or not the model is learning generic attributes of malicious scripts or just memorizing the dataset. Malicious code is not necessarily spatially related which may cause issues for CNNs that rely upon spacial relativity. This means that components of a malicious script may be rearranged within a given script without changing the classification of the script as malicious. For example, changing the order functions appear in a malicious script does not alter the behavior of the script. Let's say you had a picture of a horse in a large open plain and you sliced the horse up into 80 vertical slices and randomly moved those slices around to other parts of the image. Our model would still be required to identify the horse in the scene even though none of the slices are spatially related anymore.

4. Poshmal Dataset

There are no publicly available datasets to support the creation of machine learning algorithms to detect malicious PowerShell scripts. The intial dataset used in this paper was built by scraping the following websites for PowerShell scripts: PowerShell Scripting Gallery, Hybrid Analysis, and Github. Samples from Hybrid Analysis were downloaded using the public API with a filter for only *.ps1 files that were categorized as a 5-threat level, meaning they were confirmed malicious. We wrote a custom scraper in python to search Github and the Microsoft PowerShell Scripting Gallery for PowerShell scripts, download, and label using Windows Defender.

5. Dataset Augmentation

Scraping various parts of the internet for PowerShell yielded approximately 283,735 clean samples and 2,042 malicious samples. Random sampling of the malicious set revealed that the set was predominantly non-obfuscated PowerShell scripts. In order to increase the number of malicious samples for training and to enable the model to be able to identify a variety of obfuscation techniques, we used the Invoke-Obfuscation library to randomly take in and generate new obfuscated samples. Each iteration would select between 1 – 5 layers of obfuscation, then randomly select one obfuscation technique from the 26 available techniques listed in Appendix A for each layer. Those obfuscation techniques were then applied to the original sample to create a new sample. 72,434 samples were generated this way. Another 44,762 samples were generated by inserting modified and unmodified malicious scripts into clean scripts.

6. Revoke Obfuscation Dataset

This dataset was generated by manually extracting features from the Poshmal dataset using the Revoke-Obfuscation library. The dataset file was deterministic, meaning that the ratio of clean to malicious data was 2:1 and that the last 166,667 rows were all malicious. Models trained on this dataset would be heavily skewed, thus randomization was performed. To avoid Out of Memory problems, the data was read in chunks of 20,000 lines and split into training and testing portions. The files needed to be trained and tested individually to avoid Out of Memory problems. For future work, the ratio of clean to malicious data in each dataset should be forced to be 1:1 during preprocessing. Doing so would avoid skewing our models toward clean data because of the data imbalance. Therefore, the following experiments done on the Obfuscated Dataset are slightly skewed toward clean data.

7. Experiments

7.1. Logistic Regression

The Logistic Model is the base model for our investigation. The purpose of our Logistic Regression model is to compare the results to the Logistic Regression Model used in the paper, Revoke-Obfuscation: Powershell Obfuscation Detection Using Science by Daniel Bohannon and Lee Holmes[1]. Their model has an accuracy of 96% on their obfuscated data while our model has a similar 93% accuracy on our obfuscated data. The similar results prove that our dataset is reasonable and similar enough to be used on our more complex models such as FCNN and LSTM. Furthermore, the results prove that we can build on top of the paper and compare the results of more complex models.

A justification for the performance done by these simple models is that there are only 2 classes in our dataset: malicious and non-malicious. Limited binary classifications such as the one used in this case is specifically what simple linear models are designed for. Lowering the batch size to

32 was the key to a higher accuracy as it controls the accuracy of the estimate of the error gradient when training. Smaller batches are noisier, which helps avoid generalization. (Logistic model)

7.2. Fully Connected Neural Network

The fully connected neural network contains 2 Linear layers with a relu activation unit. In terms of complexity, this is the next model after the logistic regression model to compare our results to. It is a cheap way to see the effects of a multi-layer approach and will be an indicator if our future models trained on the poshmal dataset would provide high test accuracies. The model is able to provide slightly better results compared to the logistic regression model. The NN benefits off our large dataset for its optimization in order to get its benefit of generalization and nonlinear mapping. Improvement is expected as the added activation function allows for the identification of non-linear relationships. For this example, we used ReLU for our activation function because the rectified linear activation function overcomes the vanishing gradient problem from sigmoid and hyperbolic activation functions, allowing models to learn faster and perform better.

The dataset is unbalanced, the ratio of clean to malicious data is 2:1, which lead to slight inaccuracies. Furthermore, an unbalanced dataset allows one class to be better trained and introduces skewing. A future improvement to increase accuracy would be to add more malicious samples into our dataset to avoid skewing. An additional change would be to manual feed even ratios of malicious and clean data into training in order to improve testing accuracy. (simple fully connected model)

7.3. FCNN-384

This convolutional model applies a stack of convolutions with an AdaptiveMaxPool1d layer that consolidates all of the input down into a fixed size that is then passed through a fully connected layer. This model has the advantage of being able to see all of the data in the input as opposed to other techniques such as only looking at the first n characters of a script or command [3]. Additionally, this model can correlate between malicious code separated by many non-malicious commands in between. This can be helpful in making the model more robust to adversary obfuscation techniques such as inserting legitimate commands in between malicious commands.

This model performed very well at 98.80% test accuracy and learned many robust features discussed in the Visualization and Results sections. This model could be useful for offline research or suggesting static signatures to forensic analysists, but the accuracy was not high enough to be viable in a production environment. (FCNN-384)

7.4. CFCNN-128

While FCNN-384 performed well, it had a very limited number of learnable parameters near the raw input. Unlike with images, text has very sharp characteristics. Small changes to characters can completely change the meaning of the script versus an image where small changes in pixel value may not result in perceivable changes. One potential theory as to why FCNN-384 failed to achieve the desired accuracy and false positives scores is because some of the input features were obfuscated to early due to the max pooling layers and limited number of parameters in the convolutions. We attempted to address that issue in CFCNN-64 by implementing a convolution comprised of fully connected layers where each "kernel" is actually a stack of linear fully connected layers and rectified linear activations. This design places a dense number of parameters near the raw input. We selected 25 as the kernel size because most of the 1-minimal strings identified in the Delta Debug algorithm were 25 characters or less.

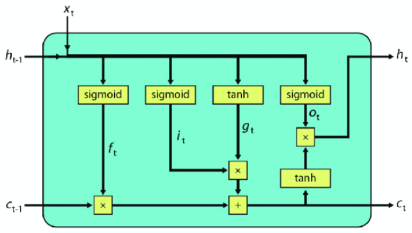
This model did not perform well, tapping out at 81.30% classification accuracy. CFCNN-64 was extremely slow to train; however, the speed could be partially due to poor implementation of the convolution. Input tensors were simply reshaped from a (1, 1, x) to a (1, x / 25, 25) before being passed through 64 fully connected "kernels". Due to variable lengths in the dataset, only one sample could be passed through the model at a time. Those layers could likely be accelerated via implementation in CUDA. Reshaping the input tensor effectively created a poor man's convolution where the stride was equal to the input dimension to the fully connected layer. In CFCNN-128 we attempted to improve the accuracy by increasing the number of kernels to 128, reduce the potential of overfitting by adding dropout as the first layer of the kernel, improve the learning rate by adding a residual layer, and improve accuracy by adding additional layers. This model also maxed out at 84.0% accuracy.

One potential contributing factor to the poor performance could have been the large stride. Taking smaller strides may help improve performance. (CFCNN-128)

7.5. LSTM

LSTM is a recurrent neural network that is capable of processing sequential/time series data and is widely used in areas like text, speech, and video where each of the small data sections is related together. The model holds an advantage in such applications because it works by combining observed examples with new inputs and trying to understand the data with the help of context. The model is built from cells 1 which each contain an input gate, an output gate, and a forget gate. With the intuition of humans only keeping important pieces of information when processing large-scale input, LSTM is designed to both keep relevant information

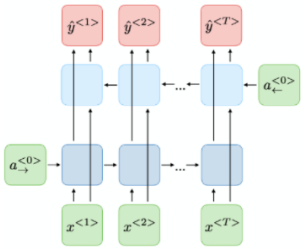
and forget irrelevant ones to make better predictions. After passing in the current input and the previous hidden state, the combined data first arrives at the forget gate, where it decides to forget or keep using a sigmoid function. Then, the data comes to the input gate that has a tanh function in addition to the sigmoid function, which together determines the importance of the tanh function output. Lastly, we have the output gate that takes both the combined data of the previous hidden state, the input state, and the processed data from the forget and input gate, to decide the final information that should get carried by the hidden state by multiplying results from sigmoid and tanh functions. These gates work together, determine how data is going to flow through the network, and enable the model to remember partial information in some random time interval.



[2]

Figure 1. LSTM Cell

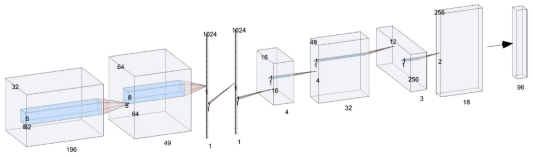
In usual cases, LSTM will have an embedding layer at the front that serves the purpose of reducing input dimensions and retaining important information. After getting processed by the embedding layer, we then have a vector representation of the input token that could be used to feed into the LSTM layers and further investigate the shared content between those vectors. However, in our project, we implemented an LSTM model that does not have an embedding layer. Because of how variable malicious scripts are, the embedding layer was replaced by a CNN network [4] to extract features and strive to achieve results that outperform the conventional method given that CNN networks are known for their robustness on classification tasks like ours. The model starts with a CNN taking in one example at a time and generating vectors of size 2048, which then transforms and get passed into the 32 layer Bidirectional-LSTM(BiLSTM) 2 with hidden layers of size 32. Here BiLSTM is used to read the input from both forward and backward sequences, which allows for better context interpretation since code could be obfuscated. The unbalanced dataset and randomness of the script contents were of the major concerns to this model. But after data augmenting, we were able to get a balanced dataset that is suitable for training on our model after padding and cleaning. The first prototype we built learned rather fast with shallow CNN and LSTM, but it was only stable at the beginning stage of training which then suffers hard from novel cases. Problems encountered include getting trapped at local optimum,



[7]

Figure 2. Bidirectional LSTM Cell

significant variance in performance between training and testing, unknown causes for not being capable of learning anything after certain epochs. And we overcame these by hyper-tuning and experimenting on different model structures. The final model 3 was measured on the testing set which takes up 10 percent of the whole dataset with metrics such as accuracy, precision, recall, and false-positive ratio. The final model yields an accuracy over 96% with an FP ratio of 0.02%, which is decent but can still be improved. (LSTM model)



[5]

Figure 3. CNN-LSTM Model

7.6. LSTM Based Attention Model

Transformer models have become state-of-the-art architecture in the machine learning world, especially in language modelling (Natural Language Processing). Key attribute of transformers enabling success is the attention mechanism utilized by encoder and decoders where matrix dot product along with other operations quantifies the relative importance of the words within the sentence. Vaswani et. al. states self-attention[8], sometimes called intra-attention as an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. For experimentation, sentence embedding model[6] from Lin et. al. was used although the penalizing part was not used which was proposed in the paper to address redundant issue of same weights of attention layers. The goal was to extract different aspect of PowerShell scripts into vector-representations effectively constituting an intra-sentence level self-attention mechanism.

First part is bidirectional LSTM, and the second part is the self-attention module. 2 different set of linear layers represent the layer learned by self-attention modules. There are two attention heads. In the attention module, the output of the LSTM output is fed into linear layer for first attention. Then, the output of this layer is fed into tanh and into next linear layer for second attention. Then, the SoftMax

of this output is the result of attention processing. The attention outputs are fed into a fully connected layer and its output is used to get the final label prediction using another linear layer. Structure of the model is basically an embedding layer with input size and hidden size followed by bidirectional LSTM, 2 linear layers for attention heads, fully connected layer, and a final linear layer to output the prediction.

Some challenges were foreseen given the nature of data. Highly varying number of words within each script, the challenge is inherent in the nature of attention network to rely on specific size input. Consistent with observation from Vaswani et. al., the model was flailing as the length of scripts varied from couple hundred words to hundreds of thousands of words. According to Vaswani et. al., The Self-attention layers[8] are faster than recurrent layers when the sequence length n is smaller than the representation dimensionality d, which is most often the case with sentence representations used by state-of-the-art models in machine translations, such as word-piece and byte-pair representations. For the purpose of continuing the research, we did not set maximum number of words per script. This is an area set aside for future work. Future work is to create a words selector that selects a fixed number of important words from the given scripts.

The model was not provided a set of specific vocabulary. Since the PowerShell script itself can have any kind of variable names, creating vocab from training set may not address any new forms of variables in actual test and result in incorrect predictions. Hence, pre-built vocabulary was not created and used. No penalization[6] was implemented and may not improve the situation when attention is not learning the weights. This may cost efficiency and accuracy of the model. Reading binaries from a file instead of utf-8 encoded format was made it difficult to tokenize the words in a customized way. Instead, the integer values read from the script file were utilized as the tokens directly.

Despite the challenges presented by the varying script sentence size and lack of standard vocabulary, attention network was able to get 90% accuracy in test set. Embedding layer may been impacted due to absence of vocabulary set. While the results resemble that the network may have memorized the 100 rows of clean and obfuscated training rows, the improvement in learning over epochs indicates that the learning is heading into right direction. Also, the network may have suffered loss of accuracy from the fact that we decided to opt out of alpha penalization which was originally implemented by Lin et. Al. Also, due to underlying LSTM network on top of attention layer calculation, the training was excessively slow compared to convolutional neural network. (attention)

8. Visualization

For visualization of the model, we implemented the Delta Debug algorithm in order to find the 1-minimal substring from the original script that the model still flags as malicious. Originally, the Delta Debug algorithm was designed to find the smallest possible input to a program that would cause it to crash; however, we re-purposed this algorithm and changed "crash" to "is malicious". This technique allows us to see which strings the model finds interesting for classification purposes. The algorithm works by recursively slicing the original script into smaller and smaller chunks until the model no longer flags the current substring as malicious. The last substring that was flagged as malicious is returned. It guarantees that if any one character is removed from the result, that script will no longer flag as malicious.

For example, we ran the Delta Debug algorithm on a post-exploitation credential harvesting script with the FCNN-384 model which yielding the following output:

```
3,0xb4,0x35,0xb5,0x14,0x04,0xee);
```

The snippet above was from a portion of the script that is setting up a constant variable representing an empty LANMAN password hash for a subsequent credential dumping routine.

```
$antpassword=[Text.Encoding]::ASCII.GetBytes
    ("NTPASSWORD`0");
$almpassword = [Text.Encoding]::ASCII.GetBy
    tes("LMPASSWORD`0");
$emptylm=[byte[]](0xaa, 0xd3,0xb4,0x35,0xb5
    ,0x14,0x04,0xee,0xaa, 0xd3,0xb4,0x35,0xb
    5,0x14,0x04,0xee);
```

To explore other "signatures" from the model, we removed the last line from the script above and ran Delta Debug on the new script, which yielded the following output.

```
$emptynt=[byte[]]@(0x31,0xd6,0xcf,0xe0,0xd1,
0x6a,0xe9,0x31,0xb7,0x3c,0
```

It appears that this particular signature is very specific to hexadecimal characters and potentially the name of the variable itself. In order to determine whether or not the model is alerting off of any hex characters or just the specific ones in this script, we ran the model on the script below. The model correctly identified the script as non-malicious.

```
$temp=[byte[]]@(0x1b,0x54,0xa7,0xff,0x02,0x9
a);
Write-Host "hello world!"
```

This shows that the model was not generically alerting off of any hex characters, but specifically malicious strings

	Validation				Test			
Model	Accuracy	Precision	Recall	FP Ratio	Accuracy	Precision	Recall	FP Ratio
RO-Logistic	93.30	93.77	90.85	5.80	93.51	93.79	90.0	5.39
RO-simple-FCNN	94.57	94.87	83.52	0.84	93.60	93.54	79.39	1.01
FCNN-384	98.30	98.30	98.50	1.90	98.80	98.80	99.10	1.50
CFCNN-128	85.12	85.54	80.52	10.21	84.00	86.11	72.54	4.28
LSTM	96.35	96.54	95.07	0.02	96.65	96.79	95.46	0.02
ATTENTION*	0.93	0.96	0.92	0	0.86	0.93	0.79	0.00
Ensemble					95.10	95.54	90.20	0.00

Table 1. Validation and Testing Results. * Based of 20 epochs 140 scripts

of hex characters.

9. Results

Table 1 lists the results for each model we developed. The three CNN models encountered overfitting but was able to resolve this issue by adding dropout layers and/or decreasing the LR by an order of a magnitude. All models had a CrossEntropy loss function and SGD optimizer. The best performing model is FCNN-384 at 98.80% classification accuracy. Upon manual inspection of the false positives identified by the model during testing, several scripts were actually found to be mislabeled. One script contained a function for extracting local credentials; however, when submitted to VirusTotal it was not flagged as malicious. The source of that function actually came from a Microsoft repository hosting utilities for Azure. This particular sample is not malicious in and of itself as it could be used for both legitimate administration or by attackers. There may not be a static way to classify samples such as this without more information about the context of the employment regardless of the model.

One significant improvement in the use of a Deep Neural Network for classification is that the number of signatures that the model can build makes obfuscation efforts significantly more challenging for an attacker. For most static signaturized PowerShell scripts, there are only a handful of human developed signatures loaded into the Antivirus engine that need to be mitigated. This makes obfuscation relatively easy as the attacker only needs to obfuscate a few tokens in a script to get it back into production. We evaluated the robustness of the FCNN-384 model on sample 1B7DE918C7E5777F4E15DB4728A6E4EAFAE1CA59FB52571AB3EC8551D3ADD913 by repeatedly running the Delta Debug algorithm to identify and remove malicious substrings one-by-one. Within that one file, the model was able to alert on more than 36 different components of the script. This significantly increases the difficulty for an adversary to achieve code execution in PowerShell as they have to mitigate significantly more portions of the script, and those obfuscation techniques must also not be detected by the model.

To evaluate the model against novel obfuscation techniques, we generated an additional 1000 malicious samples using random combinations of six obfuscation techniques not used in the training set with one to five layers of obfuscation. The FCNN-384 model performed well against this dataset with 95.40% classification accuracy while Windows Defender tapped out at 36.47% accuracy. Our model also took less than 50% of the time compared Windows Defender.

Using the Delta Debug algorithm yielded some visibility into what the model was detecting. The code snippets in Appendix B.2. show what the model flagged on with different layers of obfuscation. The first layer randomized variable names, but the model still flagged that line as malicious, so the model was able to generalize the signature to the fact that variable names can be changed arbitrarily. The model needed a variable there (otherwise the 1-minimal string would have excluded it), but it did not matter what the variable name was. In the last layer, member tokens were converted to strings and strings were obfuscated with Base64. The signature the model was alerting on changed. Previous key string tokens were now obfuscated and not available to alert off of, but the model was able to detect the Base64 obfuscation technique in conjunction with a few other code snippets.

Lastly, we implemented an ensemble of 7 pre-trained FCNN-384 networks where any disagreement would result in a negative classification. This model was able to classify at 95.10% accuracy without any false positives. These results show promise for the technique to be used in production. With some tuning and more samples, this model could be used for cloud analysis of samples uploaded from remote endpoints.

Considerations for future work would include investigating better ensembling techniques to improve classification, publishing a dataset for researchers (as none exists), and eliminate mislabeling (there are at least 2) by running VirusTotal on the clean dataset.

References

[1] Daniel Bohannon and Lee Holmes. Revoke-obfuscation: Powershell obfuscation detection using science, 2017. 1, 2

[2] Tayfun Gokmen, Malte Rasch, and Wilfried Haensch. Training lstm networks with resistive cross-point devices. *Frontiers in Neuroscience*, 12, 10 2018. 4

[3] Danny Hendler, Shay Kels, and Amir Rubin. Detecting malicious powershell commands using deep neural networks, 2018. 3

[4] Manjunath Jogin, Mohana, M S Madhulika, G D Divya, R K Meghana, and S Apoorva. Feature extraction using convolution neural networks (cnn) and deep learning. In *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, pages 2319–2323, 2018. 4

[5] ALEXANDER LENAIL. Alexnet. 4

[6] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding, 2017. 4, 5

[7] Stanford.edu. Introduction to recurrent neural networks cs-230. 4

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. 4, 5

A. Work Division

Student Name	Contributed Aspects	Details
Michael Lester	Data Creation, Modeling, and Visualization	Scraped the dataset for this project, performed data augmentation, trained CNN-384 and CFCNN-128, and implemented Delta Debug algorithm for visualization.
Xiwei Chen	Implementation and Analysis	Trained the LSTM of the encoder and analyzed the results. Analyzed effect of number of nodes in hidden state. Implemented Convolutional LSTM.
Kumar Lama	Implementation and Analysis	Trained the attention model based on bidirectional LSTM and its drawbacks due to the nature of data.
Willie Zeng	Data Scraping, Implementation and Analysis	Scraped the Obfuscated dataset and trained the Logistic Regression and simple fully connected model. Analyed the models efficiency and drawbacks

Table 2. Contributions of team members.

B. Diagrams

Technique	Description
COMPRESS\1	Compresses the original script using GZip and adds self-extracting PowerShell commands.
TOKEN\STRING\1	Splits strings in a PowerShell script using concatenation. Example: ('co'+'ffe'+'e')
TOKEN\STRING\2	Splits strings using replacement formatting. Example: ('{1}{0}'-f'ffee','co')
TOKEN\COMMAND\1	Inserts backticks that don't change the functionality of the script. Example: Ne`w-O`Biect
TOKEN\COMMAND\2	Converts commands to strings, applies TOKEN\STRING\1, and then executes using the ampersand operator. Example: &('Ne'+'w-Ob'+'ject')
TOKEN\COMMAND\3	Converts commands to strings, applies TOKEN\STRING\2, and then executes using the ampersand operator. Example: &('{1}{0}'-f'biect','New-O')
TOKEN\ARGUMENT\1	Randomly changes the case of letters in argument tokens. Example: nEt.weBclIenT
TOKEN\ARGUMENT\2	Inserts backticks into arguments. Example: nE`T.we`Bc`lIe`NT
TOKEN\ARGUMENT\3	Converts arguments to strings and applies TOKEN\STRING\1. Example: ('Ne'+'t.We'+'bClient')
TOKEN\ARGUMENT\4	Converts arguments to strings and applies TOKEN\STRING\2. Example: ('{1}{0}'-f'bClient','Net.We')
TOKEN\MEMBER\1	Randomly changes the case of letters in a member token. Example: dOwnLoAdsTRing
TOKEN\MEMBER\2	Inserts backticks into member tokens. Example: d`Ow`NLoAd`STRin`g

Figure 4. Obfuscation Techniques

TOKEN\MEMBER\3	Converts member tokens to strings, applies TOKEN\STRING\1, and invokes using the .Invoke() method. Example: ('dOwnLo'+'AdsT'+'Ring').Invoke()
TOKEN\MEMBER\4	Converts member tokens to strings, applies TOKEN\STRING\2, and invokes using the .Invoke() method. Example: ('{1}{0}'-f'dString','Downloa').Invoke()
TOKEN\VARIABLE\1	Randomly changes the case of letters in a variable token, inserts curly brackets, and inserts backticks. Example: ${c`hEm`eX}
TOKEN\TYPE\1	Converts type tokens to strings and applies TOKEN\STRING\1 then converts to a type by type casting. Example: [Type]('Con'+'sole')
TOKEN\TYPE\2	Converts type tokens to strings and applies TOKEN\STRING\2 then converts to a type by type casting. Example: [Type]('{1}{0}'-f'sole','Con')
TOKEN\WHITESPACE\1	Inserts random whitespace characters between tokens. Exmaple: .('Ne' +'w-Ob' + 'ject')
TOKEN\COMMENT\1	Removes all comments.
STRING\1	Concatenates the entire command.
STRING\2	Reorders the entire command after concatenation.
STRING\3	Reverses the entire command after concatenating.
ENCODING\1	Encodes entire command as ASCII.
ENCODING\2	Encodes entire command as Hex.
ENCODING\3	Encodes entire command as Octal.
ENCODING\4	Encodes entire command as binary.

Figure 5. Obfuscation Techniques Cont.

B.1. FCNN-384

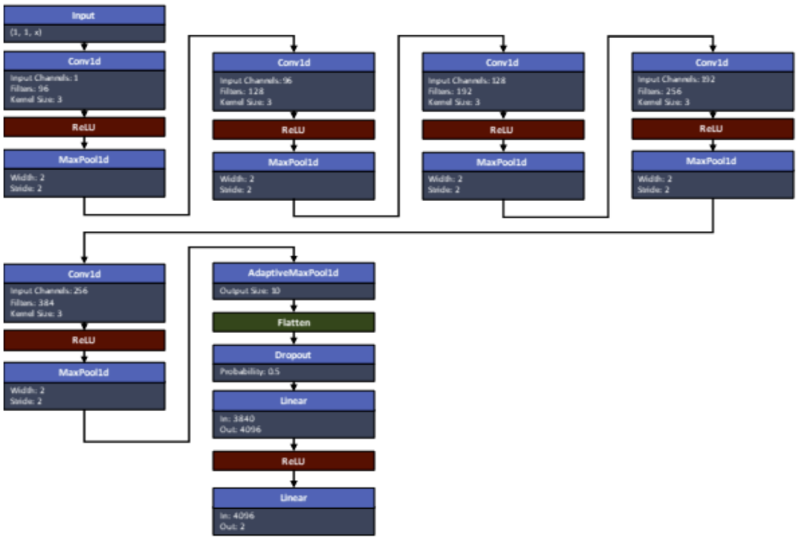


Figure 6. An overview of our FCNN-384 model

B.2. CFCNN-64

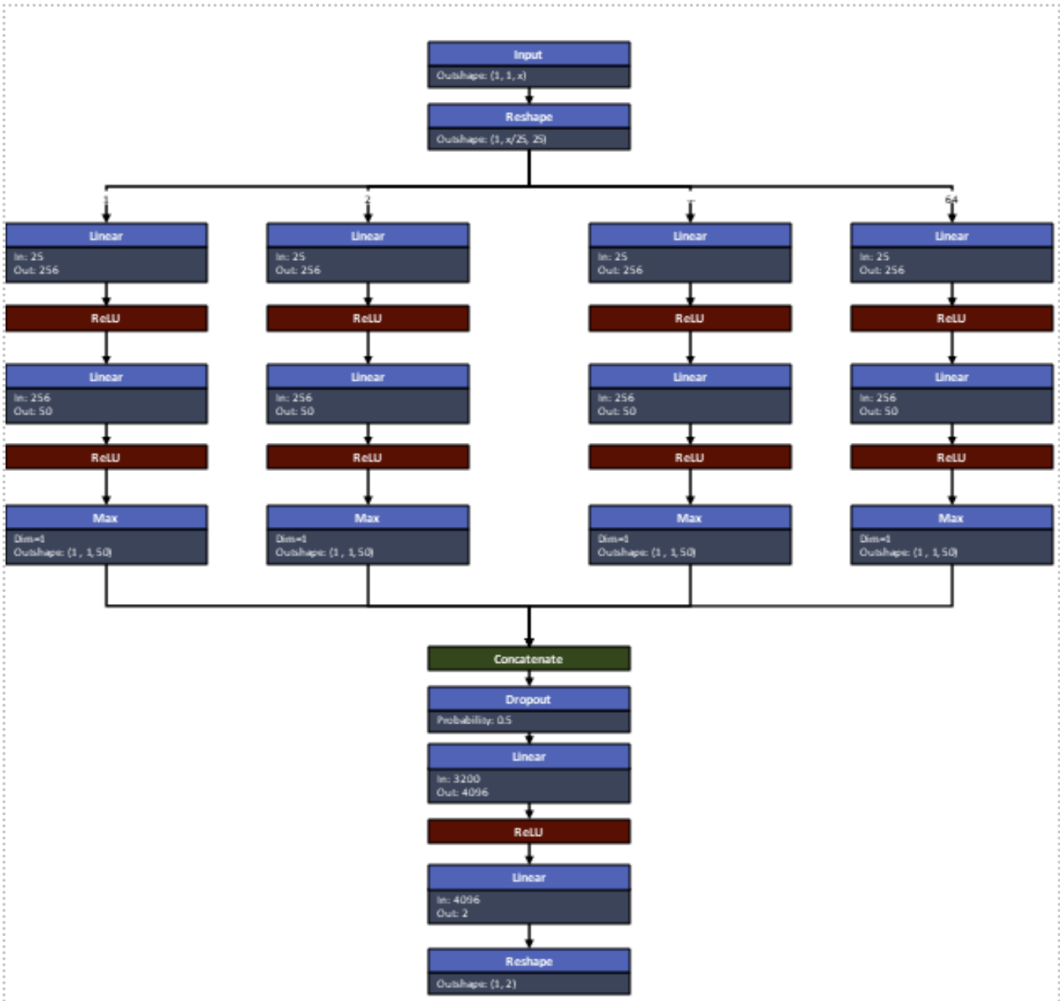


Figure 7. An overview of our CFCNN-64 model

B.3. Model Analysis using Delta Debug

```
$address = $GetProcAddress.Invoke($null, @(
[System.Runtime.InteropServices.Handle
Ref]$HandleRef,'AmsiScanBuffer'))
```

```
$ok = $right.Invoke($null, @([System.Runtim
e.InteropServices.HandleRef]$file,'AmsiScanB
uffer'))
```

```
$length = New-Object System.Reflection.Asse
mblyName(([System.Text.Encoding]::UTF8.GetSt
ring([System.Convert]::FromBase64String('V2l
uMzI='))))
$text = [AppDomain]::([System.Text.Encoding
]::UTF
```