Malware Detection - Final Project - Part A

Noam Sitton 203654199, Itay Uralevich 203537717 and Shoham Danino 204287635

January 2021

1 Research Question

What is the best Deep Learning algorithm (from already known frameworks) for intrusion detection?

2 Article

http://isyou.info/jisis/vol9/no4/jisis-2019-vol9-no4-01.pdf

3 Data-set

CSE-CIC-IDS2018 : A. L. I. Sharafaldin and A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In Proc. of the 4th International Conference on Information Systems Security and Privacy (ICISSP'18), Funchal, Madeira, Portugal, pages 108–116. ICISSP, January 2018.

4 Research Question Importance

In an ever-changing world of computer technology and rapidly evolving cyber threats, attackers and defenders are in a constant loop of cat-and-mouse game. Naturally, a growing body of work is concerned with the protection and security of computer systems. One of the crucial tools—besides network firewall(s)—defenders must add to their security layer a defense-in-depth strategy to effectively deter advanced persistent cyber threats, a Network Intrusion Detection System (NIDS). A NIDS differs from a firewall in that it monitors the flow of incoming and outgoing network traffic, raising alarms if a threat is detected. A firewall on the other hand acts as a "guarded gate" by allowing traffic to and from trusted devices or network defined in the ruleset. If a trusted network is already compromised, a firewall can be useles.

5 The Article Technique

The challenge with the existing literature is its reliance on data sets with shortcomings and issues to train ANIDS models, including, but not limited to: poor representation (outdated attack traffic and no representation of modern attack types), redundancy, anonymity (due to privacy or ethical issues), simulated traffic, lack of traffic diversity, and the lack of an all-encompassing data set. The article technique builds on existing research to highlight the utility of various deep learning frameworks (e.g., Keras, TensorFlow, Theano, fast.ai, and PyTorch) in detecting network intrusion traffic and also classifying common network attack types. To do so, they exploit a recent CSE-CIC-IDS2018 data set, which overcomes some of the common limitations of previous data sets.
Anomaly detection solutions of the dataset were proposed utilizing Convolutional Networks (CNN). CNNs are specially formulated for multidimensional data (images, video, sound) requiring transformation of input into a 2D array mimicking an image.
The CNN consists of an input layer, a convolutional layer, a pooling layer, a fully connected layer, and an output layer. The CNNs of different structures have varying numbers of convolution layers and pooling layers.
Assume that the input characteristic of the CNN is feature map of the layer i is Mi(M0 = X). Then, the convolution process can be expressed as: Mi = f (Mi1 Wi + bi) where Wi is the convolution kernel weight vector of the i layer; the operation symbol “ “ represents the convolution operation; bi is the offset vector of the i layer; and f (x) is the activation function.

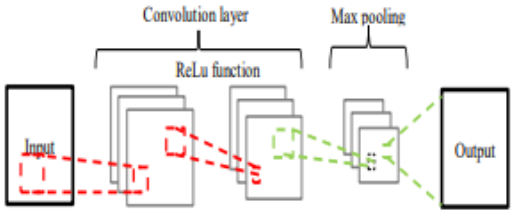

Figure 1: CNN general structure

In the article they experimented with several state-of-the-art deep learning frameworks for each data set. There are two common approaches to evaluate machine learning models: n-fold crossvalidation (normally 10-fold) and train-test split (normally 70-30 or 80-20). n-fold cross-validation is normally used when the number of samples in some categories are small or disproportionate whereas train-test split is used when the data set contains a large number of samples in every category. In the article they used both the approaches where appropriate. they combined all the individual files and generated two other data sets to experiment with binary-class and multi-class classification. For binary-class classification they relabeled all the attack traffic as 1 and benign traffic as

0. For multi-class classification they either left the labels as they were (textual) or converted them into one-hot (a.k.a. one-of-K scheme) encoding depending on the framework requirements.

6 Data sets

In the article they downloaded and used a recent network intrusion data sets generated and published in 2018 (https://www.unb.ca/cic/datasets/ids-2017.html). For more information about the data sets, including the experiments and testbeds used to generate the data sets, we refer to read CSE-CIC-IDS2018 that we mention in section 3. The data sets consist of benign (normal) network traffic and malicious traffic generated from several different network attacks: Brute force attack, Denial of Service (DoS) attack, Bot attack, Web attack, Infiltration attack, Benign traffic.

7 Accuracy

Dataset	Framework	Accuracy (%)
02-14-2018.csv	fast.ai	99.85
	Keras-TensorFlow	98.80
	Keras-Theano	98.58
02-15-2018.csv	fast.ai	99.98
	Keras-TensorFlow	99.32
	Keras-Theano	99.17
02-16-2018.csv	fast.ai	100.00
	Keras-TensorFlow	99.84
	Keras-Theano	99.41
02-22-2018.csv	fast.ai	99.87
	Keras-TensorFlow	99.97
	Keras-Theano	99.97
02-23-2018.csv	fast.ai	99.92
	Keras-TensorFlow	99.94
	Keras-Theano	99.95
03-01-2018.csv	fast.ai	87.00
	Keras-TensorFlow	72.20
	Keras-Theano	72.04
03-02-2018.csv	fast.ai	99.97
	Keras-TensorFlow	98.12
	Keras-Theano	93.95
Multi-Class	Keras-TensorFlow	94.73
	fast.ai	98.31
Binary-Class	Keras-TensorFlow	94.40
	fast.ai	98.68

8 Our ML Algorithm : RNN

RNN is a classification of artificial neural networks connected between nodes to form a directed graph alongside a temporal sequence. If you want to predict the next word in a sentence you better know which words came before it. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being dependent on the previous computations and you already know that they have a "memory" that captures information about what has been calculated so far. We chose to implement RNN algorithm, RNN is a classification of artificial neural networks connected between nodes to form a directed graph alongside a temporal sequence.
Recurrent Neural Network (RNN) is a practical technique in classifying sequences. There are a number of tasks that include the RNN in their operation such as image captioning, speech recognition, sentiment analysis, and scene labeling.
RNN is an extension of regular ANN with the purpose to enhance performance. The ANN possesses a few drawbacks such as inability to deal with temporal data which requires fix input and output size. This is because ANN is independent and omits everything from previous feed-forward input.It concentrates only on a particular input and then maps them directly to the output vector. The output from sequential input is only significant when all inputs are dependent on each other because the whole input is useful.
In contrast, the RNN offers more flexibility in processing various sizes of input and output using its memory that allows it to produce output that functions dependently based on the entire history of input. The layer in RNN consists of

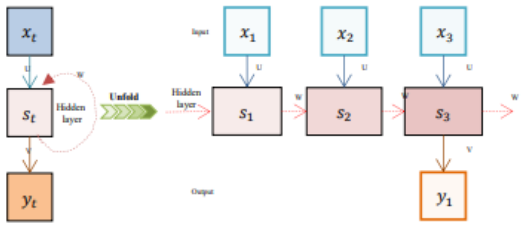

Figure 2: RNN example

an input layer, hidden layer, and output layer. Fig. 3 gives a simple example of RNN that consists of many input units, one output unit and a hidden layer. The x_t represents input, while the y_t is an output and s_t is the hidden state value. All these variables are measured at a time step. The hidden state is known as the memory; computed at a particular input and is carried forward by the network. By tracing the arrow coming towards hidden state value, there are two variables to compute a new hidden state which are input and previous hidden state value. U, V, and W refer to the parameters for the different layers. In ANN, these parameters varied at each layer but in RNN, the same parameter

values are used for each layer throughout to the end.

The authors of the paper suggested in their future work to test this method - and we want to see if it will bring more value as it stat-of-the-art deep learning method have been widely employed in researches which have been published

9 Our Features

The features with the most impact are as followings:

Label	Feature	Weight
Benign	B.Packet Len Min	0.0479
	Subflow F.Bytes	0.0007
	Total Len F.Packets	0.0004
	F.Packet Len Mean	0.0002
DoS GoldenEye	B.Packet Len Std	0.1585
	Flow IAT Min	0.0317
	Fwd IAT Min	0.0257
	Flow IAT Mean	0.0214
Heartbleed	B.Packet Len Std	0.2028
	Subflow F.Bytes	0.1367
	Flow Duration	0.0991
	Total Len F.Packets	0.0903
DoS Hulk	B.Packet Len Std	0.2028
	B.Packet Len Std	0.1277
	Flow Duration	0.0437
	Flow IAT Std	0.0227
DoS Slowhttp	Flow Duration	0.0443
	Active Min	0.0228
	Active Mean	0.0219
	Flow IAT Std	0.0200
DoS slowloris	Flow Duration	0.0431
	F.IAT Min	0.0378
	B.IAT Mean	0.0300
	F.IAT Mean	0.0265
SSH-Patator	Init Win F.Bytes	0.0079
	Subflow F.Bytes	0.0052
	Total Len F.Packets	0.0034
	ACK Flag Count	0.0007
FTP-Patator	Init Win F.Bytes	0.0077
	F.PSH Flags	0.0062
	SYN Flag Count	0.0061
	F.Packets/s	0.0014
Web Attack	Init Win F.Bytes	0.0200
	Subflow F.Bytes	0.0145
	Init Win B.Bytes	0.0129
	Total Len F.Packets	0.0096
Infiltration	Subflow F.Bytes	4.3012
	Total Len F.Packets	2.8427
	Flow Duration	0.0657
	Active Mean	0.0227
Bot	Subflow F.Bytes	0.0239
	Total Len F.Packets	0.0158
	F.Packet Len Mean	0.0025
	B.Packets/s	0.0021
PortScan	Init Win F.Bytes	0.0083
	B.Packets/s	0.0032
	PSH Flag Count	0.0009
DDoS	B.Packet Len Std	0.1728
	Avg Packet Size	0.0162
	Flow Duration	0.0137
	Flow IAT Std	0.0086

9.0.1 Redirection and cloaking

Typically, before a victim is served the exploit code, several activities take place. First, the victim is often sent through a long chain of redirection operations. These redirections make it more difficult to track down an attack, notify all the involved parties (e.g., registrars and providers), and, ultimately, take down the offending sites. In addition, during some of these intermediate steps, the user's browser is fingerprinted. Depending on the obtained values, e.g., brand, version, and installed plugins, extremely different scripts may be served to the visitor. These scripts may be targeting different vulnerabilities, or may redirect the user to a benign page, in case no vulnerability is found. Finally, it is common for exploit toolkits to store the IP addresses of victims for a certain interval of time, during which successive visits do not result in an attack, but, for example, in a redirection to a legitimate web site. We monitor two features that characterize this kind of activity:
Feature 1: Number and target of redirections. We record the number of times the browser is redirected to a different URI, for example, by responses with HTTP Status 302 or by the setting of specific JavaScript properties, e.g., document.location. We also keep track of the targets of each redirection, to identify redirect chains that involve an unusually-large number of domains.
Feature 2: Browser personality and history-based differences. We visit each resource twice, each time configuring our browser with a different personality, i.e., type and version. For example, on the first visit, we announce the use of Internet Explorer, while, on the second, we claim to be using Firefox. The visits are originated from the same IP address. We then measure if the returned pages differ in terms of their network and exploitation behavior. For this, we define the distance between the returned pages as the number of different redirections triggered during the visits and the number of different ActiveX controls and plugins instantiated by the pages. An attacker could evade these features by directly exposing the exploit code in the target page and by always returning the same page and same exploits irrespective of the targeted browser and victim. However, these countermeasures would make the attack less effective (exploits may target plugins that are not installed on the victim's machine) and significantly easier to track down.

9.0.2 Deobfuscation

Most of the malicious JavaScript content is heavily obfuscated. In fact, it is not rare for these scripts to be hidden under several layers of obfuscation. We found that malicious scripts use a large variety of specific obfuscation techniques, from simple encodings in standard formats (e.g., base64) to full-blown encryption. However, all techniques typically rely on the same primitive JavaScript operations, i.e., the transformations that are applied to an encoded string to recover the clear-text version of the code. In addition, deobfuscation techniques commonly resort to dynamic code generation and execution to hide their real

purpose. During execution, we extract three features that are indicative of the basic operations performed during the deobfuscation step:
Feature 3: Ratio of string definitions and string uses. We measure the number of invocations of JavaScript functions that can be used to define new strings (such as substring, and fromChar- Code), and the number of string uses (such as write operations and eval calls). We found that a high def-to-use ratio of string variables is often a manifestation of techniques commonly used in deobfuscation routines.
Feature 4: Number of dynamic code executions. We measure the number of function calls that are used to dynamically interpret JavaScript code (e.g., eval and setTimeout), and the number of DOM changes that may lead to executions (e.g., document. write, document.createElement).
Feature 5: Length of dynamically evaluated code. We measure the length of strings passed as arguments to the eval function. It is common for malicious scripts to dynamically evaluate complex code using the eval function. In fact, the dynamically evaluated code is often several kilobytes long. An attacker could evade these features by not using obfuscation or by devising obfuscation techniques that "blend" with the behavior of normal pages, in a form of mimicry attack [38]. This would leave the malicious code in the clear, or would significantly constrain the techniques usable for obfuscation. In both cases, the malicious code would be exposed to simple, signature-based detectors and easy analysis.

9.0.3 Environment preparation

Most of the exploits target memory corruption vulnerabilities. In these cases, the attack consists of two steps. First, the attacker injects into the memory of the browser process the code she wants to execute (i.e., the shellcode). This is done though legitimate operations, e.g., by initializing a string variable in a JavaScript program with the shellcode bytes. Second, the attacker attempts to hijack the browser's execution and direct it to the shellcode. This step is done by exploiting a vulnerability in the browser or one of its components, for example, by overwriting a function pointer through a heap overflow.
One problem for the attacker is to guess a proper address to jump to. If the browser process is forced to access a memory address that does not contain shellcode, it is likely to crash, causing the attack to fail. In other words, reliable exploitation requires that the attacker have precise control over the browser's memory layout. A number of techniques to control the memory layout of browsers have been recently proposed [4, 33, 35]. Most of these techniques are based on the idea of carefully creating a number of JavaScript strings. This will result in a series of memory allocations and deallocations in the heap, which, in turn, will make it possible to predict where some of the data, especially the shellcode, will be mapped. To model the preparatory steps for a successful exploit, we extract the following two features:
Feature 6: Number of bytes allocated through string operations. String functions, such as assignments, concat, and substring are monitored at run-time

to keep track of the allocated memory space. Most techniques employed to engineer reliable heap exploits allocate a large amount of memory. For example, exploits using the heap spraying technique commonly allocate in excess of 100MB of data.
Feature 7: Number of likely shellcode strings. Exploits that target memory violation vulnerabilities attempt to execute shellcode. Shellcode can be statically embedded in the text of the script, or it can be dynamically created. To identify static shellcode, we parse the script and extract strings longer than a certain threshold (currently, 256 bytes) that, when interpreted as Unicode-encoded strings, contain non-printable characters. Similar tests on the length, encoding, and content type are also performed on strings created at run-time. Fine-grained control of the memory content is a necessary requirement for attacks that target memory corruption errors (e.g., heap overflows or function pointer overwrites). While improvements have been proposed in this area [34], the actions performed to correctly set up the memory layout appear distinctively in these features. The presence of shellcode in memory is also required for successful memory exploits.

9.0.4 Exploitation

The last step of the attack is the actual exploit. Since the vast majority of exploits target vulnerabilities in ActiveX or other browser plugins, we extract the following three features related to these components:
Feature 8: Number of instantiated components. We track the number and type of browser components (i.e., plugins and ActiveX controls) that are instantiated in a page. To maximize their success rate, exploit scripts often target a number of vulnerabilities in different components. This results in pages that load a variety of unrelated plugins or that load the same plugin multiple times (to attempt an exploit multiple times).
Feature 9: Values of attributes and parameters in method calls. For each instantiated component, we keep track of the values passed as parameters to its methods and the values assigned to its properties. The values used in exploits are often very long strings, which are used to overflow a buffer or other memory structures, or large integers, which represent the expected address of the shellcode.
Feature 10: Sequences of method calls. We also monitor the sequences of method invocations on instantiated plugins and ActiveX controls. Certain exploits, in fact, perform method calls that are perfectly normal when considered in isolation, but are anomalous (and malicious) when combined. For example, certain plugins allow to download a file on the local machine and to run an executable from the local file system. An attack would combine the two calls to download malware and execute it.
The exploitation step is required to perform the attack and compromise vulnerable components. Of course, different types of attacks might affect certain features more than others. We found that, in practice, these three features are

effective at characterizing a wide range of exploits.

10 Our Cleanup

The dataset includes normal (benign) and attack traffic comprised of various common attack types—stated in the previous section—distributed among seven CSV files. Because of plenty of available samples and also to keep experiments simple, we dropped samples with Infinity, NaN, or missing values. We converted timestamps to Unix epoch numeric values. We parsed and removed column headers that were repeated in some datafiles. About 20,000 samples were dropped as a result of the data cleanup process. Table 1 shows the summary of datasets we use for our experiments after the data cleanup step. Number of Samples Remaining column is the total samples left after dropping number of samples from each category represented in the last column. The number of samples after cleanup are as follows:

Table 1: Number of samples and network traffic types in each dataset

Dataset	Traffic Type	Number of Samples Remaining	Number of Samples Dropped
02-14-2018.csv	Benign	663,808	3,818
	FTP-BruteForce	193,354	6
	SSH-Bruteforce	187,589	0
02-15-2018.csv	Benign	988,050	8,027
	DoS-GoldenEye	41,508	0
	DoS-Slowloris	10,99	0
02-16-2018.csv	Benign	446,772	0
	DoS-SlowHTTPTest	139,890	0
	DoS-Hulk	461,912	0
02-22-2018.csv	Benign	1,042,603	5,610
	BruteForce-Web	249	0
	BruteForce-XSS	79	0
02-23-2018.csv	Benign	1,042,301	5,708
	BruteForce-Web	362	0
	BruteForce-XSS	151	0
	SQL-Injection	53	0
03-01-2018.csv	Benign	235,778	2,259
	Infiltration	92,403	660
03-02-2018.csv	Benign	758,334	4,050
	BotAttack	286,191	0
Binary-class	Benign	5,177,646	
	Attack	1,414,765	

Table 2: Total number of traffic data samples for each type among all the datasets

Traffic Type	Number of Samples
Benign	5,177,646
FTP-BruteForce	193,354
SSH-Bruteforce	187,589
DOS-GoldenEye	41,508
Dos-Slowloris	10,990
Dos-SlowHTTPTest	139,890
Dos-Hulk	461,912
BruteForce-Web	611
BruteForce-XSS	230
SQL-Injection	87
Infiltration	92,403
BotAttack	286,191
Total Attack Samples	1,414,765

In the following, we introduce the features used in our system by following the steps that are often followed in carrying out an attack, namely redirection and cloaking, deobfuscation, environment preparation, and exploitation.

11 Our Data set

we will start with train-test split (70-30) and maybe add n-fold crossvalidation (10-fold), depends on how much time we will have left.

12 Our Results Measurement

We will calculate the accuracy as the percentage of correctly classified samples over the total number of samples evaluated. Accuracy by itself may not be a good performance measure when the number of samples being evaluated is disproportionate. Therefore, we also generated and reported confusion matrices that overcome the drawbacks of accuracy measure. In addition, because intrusion detection is very sensitive to false negative and positive, we will add it to our article.

Malware Detection - Final Project - Part B

Noam Sitton 203654199, Itay Uralevich 203537717 and Shoham Danino 204287635

March 2021

1 Article

http://isyou.info/jisis/vol9/no4/jisis-2019-vol9-no4-01.pdf

2 Data-set

CSE-CIC-IDS2018 : A. L. I. Sharafaldin and A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In Proc. of the 4th International Conference on Information Systems Security and Privacy (ICISSP'18), Funchal, Madeira, Portugal, pages 108–116. ICISSP, January 2018.
The dataset includes normal (benign) and attack traffic comprised of various common attack types—stated in the previous section—distributed among seven CSV files. Because of plenty of available samples and also to keep experiments simple, we dropped samples with Infinity, NaN, or missing values. We converted timestamps to Unix epoch numeric values. We parsed and removed column headers that were repeated in some datafiles. About 20,000 samples were dropped as a result of the data cleanup process.

3 Article technique

Anomaly detection solutions of the dataset were proposed utilizing Convolutional Networks (CNN). CNNs are specially formulated for multidimensional data (images, video, sound) requiring transformation of input into a 2D array mimicking an image.
The CNN consists of an input layer, a convolutional layer, a pooling layer, a fully connected layer, and an output layer. The CNNs of different structures have varying numbers of convolution layers and pooling layers.
Assume that the input characteristic of the CNN is feature map of the layer i is Mi(M0 = X). Then, the convolution process can be expressed as: Mi = f (Mi1 Wi + bi) where Wi is the convolution kernel weight vector of the i layer; the operation symbol “ “ represents the convolution operation; bi is the offset vector of the i layer; and f (x) is the activation function.

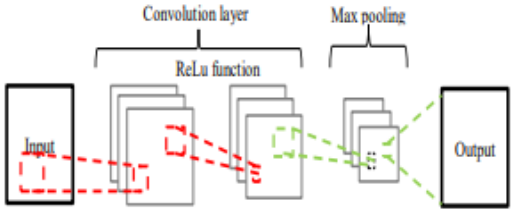

Figure 1: CNN general structure

4 CNN Implementation

We implemented our CNN network as written in the article, our input is the same, a first layer in size 79, a second layer in size 128 and the last layer in the size of our different clusters (3/4).
We Implemented the CNN algorithm on the IDS2018 dataset (in lines 176-181).
The code is :

- Cleaning each data file and saving it
- Loading each cleaned data file into the memory
- Splitting the data to train set and test set (80 percent train and 20 percent test)
- Training the CNN model with ADAM optimizer on the training set
- Evaluating the accuracy on the test set

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 79)                6320
_________________________________________________________________
dense_4 (Dense)              (None, 128)               10240
_________________________________________________________________
dense_5 (Dense)              (None, 4)                 516
=================================================================
```

Figure 2: CNN structure example from our code

4.1 CNN complexity

In general we can say that the complexity of our algorithm is liner to the input, it is known that for this algorithm the complexity is $O(2n)$.
in order to assess the complexity of a model, it is often useful to determine the number of parameters that its architecture will have. In a given layer of a convolutional neural network, it is done as follows:

	CONV	POOL	FC
input size	$I*I*C$	$I*I*C$	N_in
output size	$O*O*K$	$O*O*C$	N_out

Receptive field - The receptive field at layer k is the area denoted $R_k * R_k$ of the input that each pixel of the k-th activation map can 'see'.
Rectified Linear Unit - The rectified linear unit layer (ReLU) is an activation function g that is used on all elements of the volume. It aims at introducing non-linearities to the network, in our case we used ReLU - Non-linearity complexities biologically interpretable.
Softmax - The softmax step can be seen as a generalized logistic function that takes as input a vector of scores and outputs a vector of output probability through a softmax function (which is linear) at the end of the architecture.
Our data after the pre-processing is 3 CSV files in size of 300 MB each, each iteration the CNN model gets as input one file, train the model and calculate the accuracy so we don't need to save the files and the only space required is for the network (which is small and negligible).
in our data we have 1,000,000 instances in each file, and 3,215,000 instances in total, for checking if our complexity $O(2n)$ is correct, we run the our algorithm all our data after the preprocessing and then on each file individually, we expect to see a linear dependence between the different runs as a function in the fraction of the file from our entire data. The results for all data:

time (sec)	cpu	gpu
global data (02-15-2018,02-22-2018,02-23-2018)	212.45	306.88

02-15-2018 is 0.344 from the global data
02-22-2018 is 0.305 from the global data
02-23-2018 is 0.351 from the global data

time (sec)	cpu	gpu
02-15-2018	70.29	98.38
02-22-2018	69.96	89.77
02-23-2018	71.55	89.46

When we compare the relative part of the file from our global data algorithm run, we expect to get is fraction from the data. and we can see:
for 02-15-2018 (we expect to get: 0.344): 70.29/212.45 = 0.33 ; 98.38/306.88 = 0.32
for 02-22-2018 (we expect to get: 0.305): 69.96/212.45 =0.329 ; 89.77/306.88 = 0.292
for 02-23-2018 (we expect to get: 0.351): 71.55/212.45 = 0.336 ; 89.46/306.88 = 0.291
It can be seen from the results that indeed the complexity of the algorithm is linear to the input size as we noted.
We expected to see a CNN run on the GPU faster than the CPU, because the GPU is designed to handle better matrix multiplication (which happens a lot on the CNN network) compared to a CPU, but we got the opposite result. This is probably due to a resource problem we had (we ran everything in the colab

notebook on Google resources), unfortunately we could not get other computational resources to examine the problem but for the sake of examining the complexity it was enough for us to get the results.

4.2 CNN Results

To evaluate our algrithm, we compiled the two datasets: a known-good dataset and a known-bad dataset. the known-good dataset consists of web pages that (with high confidence) do not contain attacks. We use this dataset to train our models, to determine our anomaly thresholds, and to compute false positives. The known-bad dataset contain pages and scripts that are known to be malicious. We use these dataset to evaluate the detection capabilities of our algorithm and compute false negatives.

4.2.1 False positives

We randomly divided the known-good dataset in three subsets and used them to train our network and compute its false positive rate. More precisely, we ran our algorithm on 80 percent of the data to train the models. We then ran it on 30 percent from our train data to establish a threshold, The remaining 70 percent were used to determine the false positive rate.

4.2.2 False negatives

For the next experiment, we used the technique of the article and take the data after they compared the detection capabilities of the algorithm with respect to three other tools: ClamAV, PhoneyC, and Capture-HPC. These tools are representative of different detection approaches: syntactic signatures, low-interaction honeyclients using application-level signatures, and high-interaction honeyclients, respectively.
ClamAV is an open-source anti-virus, which includes more than 3,200 signatures matching textual patterns commonly found in malicious web pages.
PhoneyC is a browser honeyclient that uses an emulated browser and application-level signatures. Signatures are expressed as Java-Script procedures that, at run-time, check the values provided as inputto vulnerable components for conditions that indicate an attack. Thus, PhoneyC's signatures characterize the dynamic behavior of an exploit, rather than its syntactic features.
Capture-HPC is a high-interaction honeyclient. It visits a web page with a real browser and records all the resulting modifications to the system environment (e.g., files created or deleted, processes launched).
The accuracy results (True positive True negative):

Data File	CNN
02-15-2018	98.90
02-22-2018	99.97
02-23-2018	99.95

4.3 CNN errors

The accuracy results (False positive False negative):

Data File	FN	FP
02-15-2018	1.1	0
02-22-2018	0.03	0
02-23-2018	0.05	0

We see that the algorithm does not have FP errors, which means it does not describe that the code is malicious even though it is not, but we have a FN errors which means it detects that the code is not malicious even though it is. We randomly checked for 30 examples from the error. Of these, we manually verified that 6 did actually launch drive-by downloads by checking about them online (https://sitechecker.pro/website-safety/). 8 URLs are not online and We were unable to find them mentions online. The remaining 16 URLs appeared to be benign, and, thus, are false positives. The majority of them used different ActiveX controls, some of which were not observed during training.
we can see it is hard to trust our "ground truth" datasets, and we do not know how to distinguish between the types of errors perfectly, so we will treat them comprehensively as error:

Data File	error rate
02-15-2018	1.1
02-22-2018	0.03
02-23-2018	0.05

5 RNN technique

We chose to implement RNN algorithm, RNN is a classification of artificial neural networks connected between nodes to form a directed graph alongside a temporal sequence.
Recurrent Neural Network (RNN) is a practical technique in classifying sequences. There are a number of tasks that include the RNN in their operation such as image captioning, speech recognition, sentiment analysis, and scene labeling.
RNN is an extension of regular ANN with the purpose to enhance performance. The ANN possesses a few drawbacks such as inability to deal with temporal data which requires fix input and output size. This is because ANN is independent and omits everything from previous feed-forward input.It concentrates only on a particular input and then maps them directly to the output vector. The output from sequential input is only significant when all inputs are dependent on each other because the whole input is useful.
In contrast, the RNN offers more flexibility in processing various sizes of input and output using its memory that allows it to produce output that functions dependently based on the entire history of input. The layer in RNN consists of

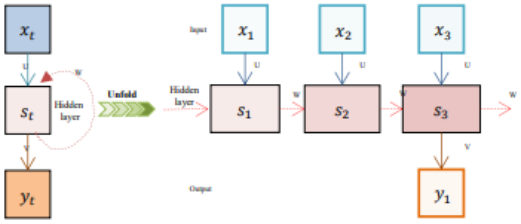

Figure 3: RNN example

an input layer, hidden layer, and output layer. Fig. 3 gives a simple example of RNN that consists of many input units, one output unit and a hidden layer. The x_t represents input, while the y_t is an output and s_t is the hidden state value. All these variables are measured at a time step. The hidden state is known as the memory; computed at a particular input and is carried forward by the network. By tracing the arrow coming towards hidden state value, there are two variables to compute a new hidden state which are input and previous hidden state value. U, V, and W refer to the parameters for the different layers. In ANN, these parameters varied at each layer but in RNN, the same parameter values are used for each layer throughout to the end.
We implement our RNN network as written in the article, our input is the same, a first embedding layer in size 64, a second simple rnn layer in size 64, the third layer as well in size 64 and the last layer in the size of our different clusters (3/4).
Similar to what we did with the CNN, we implemented the RNN algorithm on the IDS2018 dataset (in lines 279-283). The code is :

- Cleaning each data file and saving it
- Loading each cleaned data file into the memory
- Splitting the data to train set and test set (80 percent train and 20 percent test)
- Training the RNN model with ADAM optimizer on the training set
- Evaluating the accuracy on the test set

5.1 RNN complexity

In general, it can be said that the complexity of our RNN is linear depends to our input, the number of layers in our network and their sizes $O(n)$.
We have 4 small layers so in our case it is negligible and we can say that our complexity is $O(n)$. Similar to CNN, in our data we have 1,000,000 instances in each file, and 3,215,000 instances in total, for checking if our complexity $O(n)$ is correct, we run the our algorithm all our data after the preprocessing and

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 64)          5056
_________________________________________________________________
simple_rnn (SimpleRNN)       (None, 64)                8256
_________________________________________________________________
dense_9 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_10 (Dense)             (None, 3)                 195
=================================================================
```

Figure 4: RNN structure example from our code

then on each file individually, we expect to see a linear dependence between the different runs as a function in the fraction of the file from our entire data. The results for all data:

time (sec)	cpu	gpu
global data (02-15-2018,02-22-2018,02-23-2018)	519.63	148.29

02-15-2018 is 0.344 from the global data
02-22-2018 is 0.305 from the global data
02-23-2018 is 0.351 from the global data

time (sec)	cpu	gpu
02-15-2018	183.27	54.77
02-22-2018	180.98	54.30
02-23-2018	165.05	54.72

When we compare the relative part of the file from our global data algorithm run, we expect to get is fraction from the data. and we can see:
for 02-15-2018 (we expect to get: 0.344): 183.27/519.63 = 0.35 ; 54.77/148.29 = 0.369
for 02-22-2018 (we expect to get: 0.305): 180.98/519.63 = 0.348 ; 54.30/148.29 = 0.366
for 02-23-2018 (we expect to get: 0.351): 165.05/519.63 = 0.317 ; 54.72/148.29 = 0.369
It can be seen from the results that indeed the complexity of the algorithm is linear to the input size as we noted.

In addition, it can be seen that the ratio between GPU and CPU runs is the opposite of a CNN network, this is because a CNN network performs a lot of duplication matrices and this is a task for which a GPU processor (for graphical display) is designed. while for synchronous multiplication performed in an RNN network a standard CPU processor is preferable and gives better results in terms of computational speed.

5.2 RNN Results

Our evaluation method of the RNN network is simmilar to what we did in the CNN network, we compiled the two datasets: a known-good dataset and a

known-bad dataset. the known-good dataset consists of web pages that (with high confidence) do not contain attacks. We use this dataset to train our model.

5.2.1 False positives

We randomly divided the known-good dataset in three subsets and used them to train our network and compute its false positive rate. More precisely, we ran our algorithm on 80 percent of the data to train the models. We then ran it on 30 percent from our train data to establish a threshold, The remaining 70 percent were used to determine the false positive rate.

5.2.2 False negatives

we used the technique of the article and take the data after they compared the detection capabilities of the algorithm with respect to three other tools: ClamAV, PhoneyC, and Capture-HPC. These tools are representative of different detection approaches: syntactic signatures, low-interaction honeyclients using application-level signatures, and high-interaction honeyclients, respectively.
When we encountered a problem and could not decide according to those tools that are whether the URL is malicious or not, we tested it in other ways we found on the Internet.
The accuracy results (True positive True negative):

Data File	RNN
02-15-2018	94.95
02-22-2018	99.96
02-23-2018	99.95

5.3 our RNN errors

The accuracy results (False positive False negative):

Data File	FN	FP
02-15-2018	5.05	0
02-22-2018	0.04	0
02-23-2018	0.05	0

We see at first that the algorithm does not have FP errors, which means it does not describe that the code is malicious even though it is not, but we have a FN errors which means it detects that the code is not malicious even though it is. We randomly checked for 50 examples from the error. Of these, we manually verified that 3 did actually launch drive-by downloads by checking about them online (https://sitechecker.pro/website-safety/). 39 URLS are not online now and We were unable to find them mentions online. The remaining 18 URLs appeared to be benign, and, thus, are false positives. As happened to us in the CNN network (in the 02-15-2018 dataset), the majority of them used different ActiveX controls, some of which were not observed during training.

we can see it is hard to trust our "ground truth" datasets, and we do not know how to distinguish between the types of errors perfectly, so also here we will treat them comprehensively as error:

5.4 our RNN errors

The accuracy results (False positive False negative):

Data File	RNN
02-15-2018	5.05
02-22-2018	0.04
02-23-2018	0.05

6 CNN VS RNN Results

As we can see the accuracy is very close and very high between our two methods, apart from the data 02-15-2018 that this case is higher. We tried to examine why there is such a big difference. When we looked at this dataset we see the relative share of the malicious URLs in 02-15-2018 is small compared to others (0.2 percent compare to 20). Therefor the training set have less examples to practice on because we randomly divide the tests into 80-20 and statistically 2 percent is not enough. In addition, When we look at the types of malicious code that is in 02-15-2018, we see that most of the malicious code is used the same ActiveX controls and there is little that use different ActiveX controls so that is increases our statistic error. When we look at the errors of the RNN network we see that almost all of them has failed to diagnose the malicious ActiveX controls sequence. If we look specifically at the different ActiveX controls sequences we see that there are lots of ways to do malicious things with different order of commands, because some are completely legitimate even when they appear as a synchronous sequence and some are not. We assume this is probably what makes the RNN network difficult to differentiate between the two in this case and because the CNN does not look at that as sequence but as a image it can recognize these subtleties better.

6.1 Results

Data File	CNN	RNN
02-15-2018	98.90	94.95
02-22-2018	99.97	99.96
02-23-2018	99.95	99.95

6.2 Errors

Data File	CNN	RNN
02-15-2018	1.1	5.05
02-22-2018	0.03	0.04
02-23-2018	0.05	0.05

6.3 CNN And RNN Difference

CNN	RNN
It is suitable for spatial data such as images.	RNN is suitable for temporal data, also called sequential data.
CNN is considered to be more powerful than RNN.	RNN includes less feature compatibility when compared to CNN.
This network takes fixed size inputs and generates fixed size outputs.	RNN can handle arbitrary input/output lengths.
CNN is a type of feed-forward artificial neural network with variations of multilayer perceptrons designed to use minimal amounts of preprocessing.	RNN unlike feed-forward neural networks - can use their internal memory to process arbitrary sequences of inputs.
CNNs use connectivity pattern between the neurons. This is inspired by the organization of the animal visual cortex, whose individual neurons are arranged in such a way that they respond to overlapping regions tiling the visual field.	Recurrent neural networks use time-series information - what a user spoke last will impact what he/she will speak next.
CNNs are ideal for images and video processing.	RNNs are ideal for text and speech analysis.

6.4 Hypotheses for the differences between CNN and RNN

As we can see, there is a difference between CNN and RNN networks, CNN networks are better for images while RNNs are suitable for sequential data like text and speech analysis.
When we thought of another type of network that would get high accuracy we thought of the RNN network because we guessed that maybe malware is more like sequential data with context than image (we and the authors of the original article).
Although RNN has fewer features than CNN (as we can be seen in the attached code) we thought that maybe a few features would prevent over-fitting and give us better results.
We can see that the results of the two networks are very close, when we look at the difference between the two we see that the problem is mainly due to the malicious code behaving not synchronously, in such cases the RNN is not good enough and on the other hand the CNN was able to absorb these differences.
As we saw in the previous section we assume the problem is also due to a statistical gap that RNN has a hard time dealing with, because it has less features to examine so it needs more examples to refine them and choose a

more accurate threshold, in the case of Data 02-15-2018 this is not the case and the RNN failed more.
In the case of CNN there are more features so it manages to create a good threshold even with a few examples, it can cost us in overfitting but that is not the case that happens to us in the experiments we performed.
RNN networks has been proven in a number of articles we have read and even gave very good results in some, even in our work we got very good results, but less good than the original CNN network.
We think that in future work it is worthwhile to combine the methods and try to do first a run with a CNN network and then, the result we get, insert as an input to RNN network and thus get context for the malicious cases we encounter.