Data Science Toolbox Project 4

(Mis) Identifying Users by Command Usage

Sydney Vertigan, Daniel Jones, Sam Wise

March 22, 2019

The Jupyter notebook accompanying this report can be found on GitHub.

1 Introduction

Whether driving cars or assisting with medical diagnoses, neural networks are increasingly utilised to make important, safety-critical decisions. In this project, we investigate the security weaknesses of neural networks themselves. We perform our investigation in the cybersecurity context of network intrusion detection systems (IDS). We start by implementing an IDS as described by Ryan et al [3], which aims to identify users by the frequency they execute different shell commands. We then develop a novel attack, allowing intruders to masquerade as their victims without detection.

As a group, we collectively decided to challenge ourselves and take this project further. First, we develop our intrusion detection system by creating a neural network that classifies the user given a list of commands. In production, this system would monitor command usage for each account, and input the command usage statistics into the neural network. If the neural network fails to classify the usage statistics as the owner of the account, an alert is raised.

Our attack model gives us black-box access to the intrusion detection system; allowing the attacker to send it inputs, receive it's classifications, and nothing else. In this context, we call the intrusion detection system our oracle. We use this access to the oracle to build a replica, the surrogate model.

Finally, with complete access to a model that approximates our oracle, we generate adversarial examples that trick the original model into mis-classifying it's inputs. As an example from the literature, constructed noise can be added to inputs so that it would still appear the same to the human observer, but cause the neural network to make a classification error [12]. We modify this approach to fit our attack model and requirements.

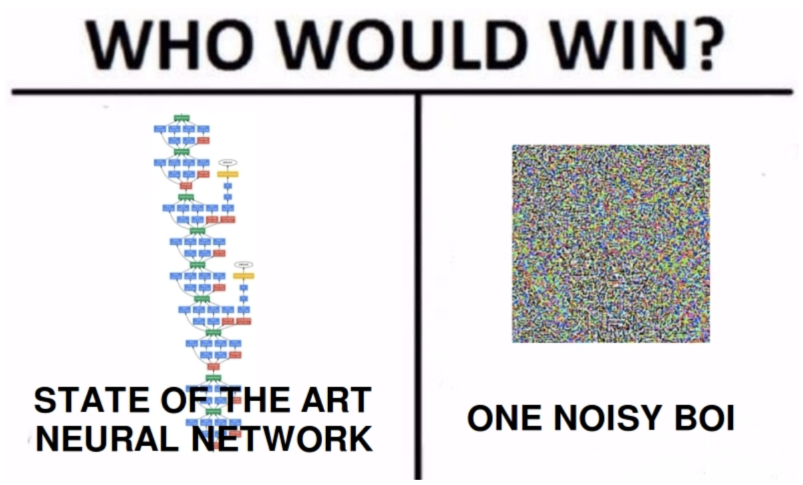


Figure 1: The effectiveness of adversarial examples via constructed random noise [12]. Generating misclassified examples without any restrictions can be done quickly without great success.

We read many articles on different examples in which neural networks have been used for cybersecurity purposes: for incident detection [1] and malware classification [2] etc. The topic we found the most interesting was to 'break' a neural network that has been created by building a surrogate model.

The main steps we had to perform in order to complete the task we had set ourselves of misleading a neural network were:

1. Create an Oracle Model that can identify users
2. Create a Surrogate Model that we can use to help break the original oracle
3. Create adversarial examples that trick the oracle

2 Intuition and Research

Different users will display different styles of behaviour depending on their computing needs. For example, some may use the computer for the sole purpose of sending and receiving emails and hence have no need for programming and compilation services.

Others may use the system for several different activities such as web browsing, editing, programming, etc. Thus different users will gravitate towards different sets of commands. In addition, the command frequency can alter between two users who use the system for the same purpose. Hence, it appears feasible to determine a user based on the commands they used and frequency of command execution. In the situation where a user's behaviour deviates from the norm, a system administrator can be alerted of a potential security breach.

Consider this example, the IDS will receive an input like "user 1 ran: sudo 10 times, grep 3 times, and source 1 in the time period 10am-11am", which is fed into the neural network. The neural network will attempt to classify the input. If it classifies the input as user 1, the system will consider the usage harmless, if not, the system will consider it malicious.

Our attack will then take in a malicious script, calculate the input of said script and give it to the neural network learn what input would pass as harmless.

Say the neural network classifies "user 1 ran: sudo 10 times, grep 3 times, and source 1 in the time period 10am-11am" as malicious, our new system would learn that running a smaller percentage of "sudo" calls would result in being classified as "harmless", e.g. "user 1 ran: 'sudo' 10 times, 'grep' 30 times, and 'source' 10 in the time period 10am-11am"

Our system would then train our adversarial network to take in the script that would cause an intrusion detection with a number of no-operation calls to 'grep' and 'source' such that it produces that input.

Our attack model and approach follow the work of Papernot et al in their influential paper *'Practical Black-Box Attacks against Machine Learning'* [4]. This paper explains how to use black box attacks in order to 'fool' a neural network. In particular, they are able to develop effective attacks on commercial image classification services using onbly black-box access. We have followed their intuition and ideas of how to approach the problem throughout our project, adapting these ideas as needed to our problem domain.

3 Motivating Example

Instead of completing this project ourselves, we aim to sabotage and steal the work of rival group J. Dones, W. Size and V. Sertigan. Luckily, J. Dones is notorious for his re-use of passwords and, following a successful phishing attack, we have logged in to his Blue Crystal account. Our next challenge is to run our malicious commands and exfiltrate their work without Blue Crystal's state-of-the-art IDS noticing.

The attack we develop through this report allows intruders to disguise their scripts using the following process:

1. Our system takes in a malicious script/set of commands:

```
ls
scp -r top-secret-research dans-server:/my-stash-of-stolen-files
sudo rm -r top-secret-research
```

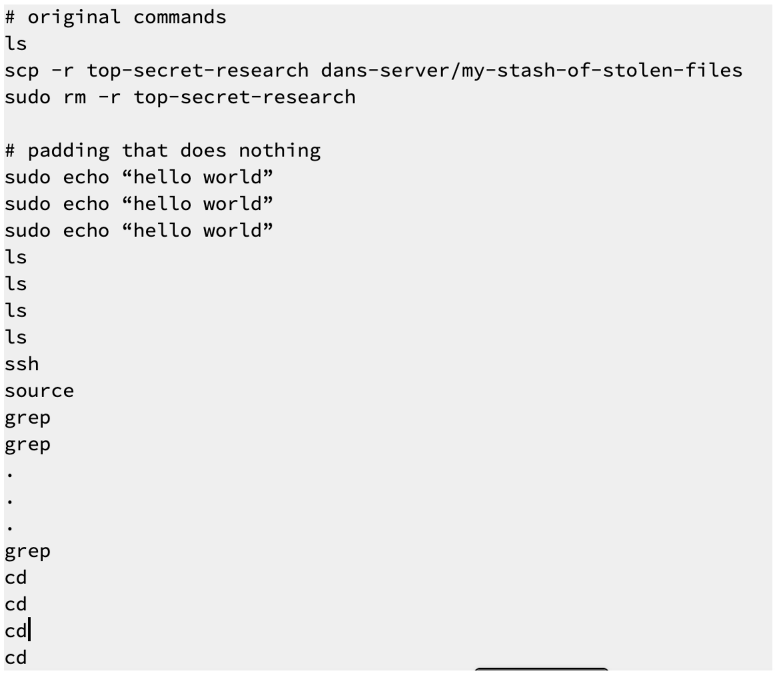
2. Calculates the input the malicious script would give to the Neural Network:

```
User, sudo, ls, scp, ssh, source, grep, cd
??, 1, 1, 1, 0, 0, 0, 0
```

3. Our system searches for a command distribution classified by our surrogate model as "JDones", with minimal distance to the original datapoint. It may find an adversarial example like the following:

```
User, sudo, ls, scp, ssh, source, grep, cd
??, 8, 5, 1, 1, 1, 1, 10, 5
```

4. Following the command counts in the adversarial example, the input script is then padded with no-op commands to create a sequence of commands classified as our target user. For example:



```
# original commands
ls
scp -r top-secret-research dans-server/my-stash-of-stolen-files
sudo rm -r top-secret-research

# padding that does nothing
sudo echo “hello world”
sudo echo “hello world”
sudo echo “hello world”
ls
ls
ls
ls
ssh
source
grep
grep
.
.
.
grep
cd
cd
cd
cd
```

This should give an input that the intrusion detection system identifies as being J. Dones. We run the code, evade the intrusion detection system, and steal the project!

4 Data Preprocessing

We were able to find two data sets which provided the labelled command usage per-user. our data source [5], *Masquerading User Data* was the larger of the two. It contains 50 files, each corresponding to a user. Within each file, there are 15,000 commands. The first 5,000 commands do not contain any masqueraders, i.e. they are the genuine user, with the intention of being training data. It is then suggested that the next 10,000 commands are divided into 100 blocks of 100 commands each. These lines contain masquerading users, in other words: data of another user which in not among the 50 users.

At any given block i after the initial 5000 commands, $\mathbb{P}$(block i contains a masquerader) $= 0.01$, $\mathbb{P}$(block i contains a masquerader $|$ block $i-1$ contains a masquerader) $= 0.8$. About 5% of the test data contain masqueraders.

We realised that our resulting dataset would be very small if we were to naively group the commands into blocks of 100, as this would result in 250 data points in total for training. As an alternative approach, we decided that we will take a rolling window sample, since we know that the commands are ordered. This provides us with a training data set with 4,901 samples per user and 245,050 data points in total.

This idea stemmed from [3], although they do not implement a sliding window themselves, they discuss other papers which use it: "Several IDSs that employ neural networks for on-line intrusion detection have been proposed (Debar et al. 1992; Fox et al. 1990). These systems learn to predict the next command based on a sequence of previous commands by a specific user. Through a shifting window, the network receives the most recent commands as its input. The network is recurrent, that is, part of the output is fed back as the input for the next."

Our second issue with the data source was that the test data was not in our desired format. The test data is formatted so that we are given the expected user and whether that is true or not. If it is true we know the exact user, but if it is false it could therefore be any other user. So, as a solution we agreed to create a test set from our original training set. This would still provide a sufficient amount of data points for training and evaluation.

5 Intrusion Detection

The idea of using neural networks for intrusion detection was first introduced in the NIPS conference in 1998 [3]. We have followed the approach that was implemented in the paper for intrusion detection.

5.1 Oracle Model

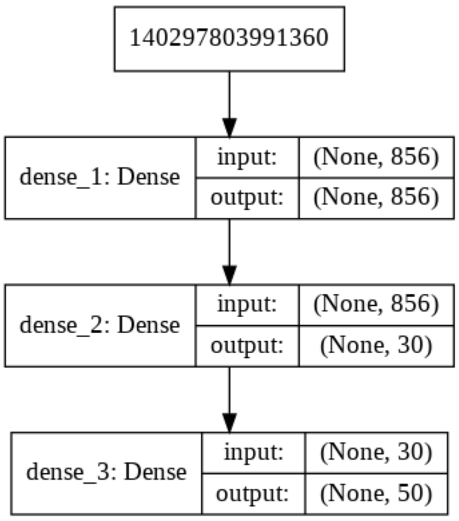


Figure 2: Visualisation of Oracle Architecture

We develop our first model, the network intrusion detection system, directly following the design of NNIDS [3]. In the context of an attack, this model is referred to as the oracle, mirroring the idea of an oracle attack in cryptography and security engineering.

The aim of this neural network is to take in vectors representing absolute counts of command usage over a given time period (as in the motivating example 3). It will then return a weighted distribution of class labels, indicating how well the activity matches each user. When used inside the IDS, when activity from a particular account is not identified by the neural network, an alarm is raised.

It is a standard three-layer backpropagation neural network. Backpropagation is defined by [6] as "the practice of fine-tuning the weights of a neural network based on the error rate (i.e. loss) obtained in the previous epoch (i.e. iteration). Proper tuning of the weights ensures lower error rates, making the model reliable by increasing its generalization." The input layer consisted of 856 units, representing the user vector; the hidden layer had 30 units and the output layer 50 units, one for each user.

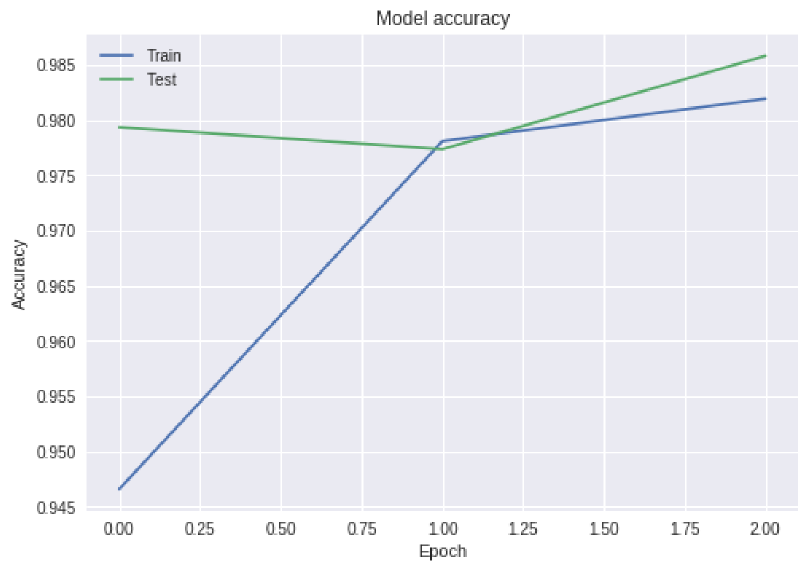


Figure 3: A plot of the oracle models accuracy against a left out test set over each training epochs. Experimentally, increasing the number of epochs did not increase it's accuracy above the three shown.

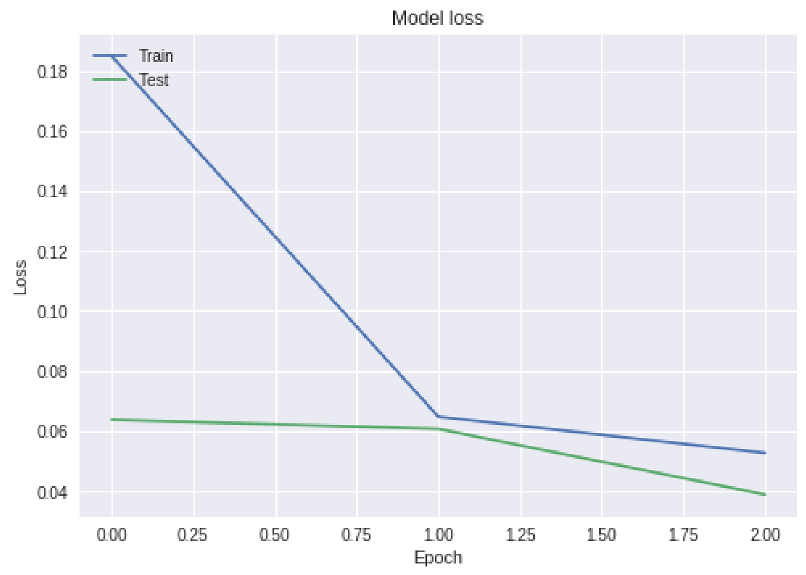


Figure 4: A plot of the oracle models loss against a left out training set over each training epoch. Experiments with a lower batch size did not decrease loss to a notable degree.

The term *backpropagation* used in neural computing literature could mean a variety of different things. For example, a backpropagation network is referring to the multilayer perceptron architecture. The term is also used to explain the method of training a multilayer perceptron via the application of gradient descent to a sum-of-squares error function.

We show the derivation of the backpropagation algorithm for a general network having arbitrary feed-forward topology, arbitrary differentiable nonlinear activation functions and a wide class of error function. The formulas which follow will then be demonstrated using a simple layered network structure; having a single layer of sigmoidal hidden units, together with a sum-of-squares error [7].

Suppose we have an error function defined by maximum likelihood for a set of i.i.d variables, comprising of a sum of terms - one for each data point in the training set:

$$E(\vec{w}) = \sum_{n=1}^{N} E_n(\vec{w}) \tag{1}$$

We consider the problem of evaluating $\nabla E_n(\vec{w})$ for one such term in the error function. We can use this directly for sequential optimization, or we can accumulate the results over the training set if we wish to implement batch methods.

Consider first a simple linear model in which the outputs y_k are linear combinations of the input variables x_i so that:

$$y_k = \sum_i w_{ki} x_i \tag{2}$$

together with an error function that, for a particular input pattern n, takes the form

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2 \tag{3}$$

where $y_{nk} = y_k(\vec{x}_n, \vec{w})$. The gradient of this error function with respect to a weight w_{ji} is given by

$$\frac{\partial E_n}{w_{ji}} = (y_{nj} - t_{nj})x_{ni} \tag{4}$$

this can be thought of as a 'local' computation involving the product of an 'error signal' $y_{nj} - t_{nj}$ related to the output end of the link w_{ji} and the variable x_{ni} related to the input end of the link.

In a general feed-forward network, each unit computes a weighted sum of its inputs of the form

$$a_j = \sum_i w_{ji} z_i \tag{5}$$

where z_i is the activation of a unit, or input, that sends a connection to unit j, and w_{ji} is the weight associated with that connection. The sum is transformed by a nonlinear activation function $h(\cdot)$ to give the activation z_j of unit j in the form

$$z_j = h(a_j) \tag{6}$$

We point out here that at least one of the variables z_i in 5 in could be an input, and similarly, the unit j in 6 could be an output.

For each pattern in the training set, suppose we have provided the corresponding input vector to the network and calculated the activations of all of the hidden and output units in the network, by successive application of 5 and 6. This process is often called *forward propagation* because it can be regarded as a forward flow of information through the network.

Now consider the evaluation of the derivative of E_n with respect to a weight w_{ji}. The outputs of the various units will depend on the particular input pattern n. We avoid the subscript n from the network variables for ease of notation. First we note that E_n depends on the weight w_{ji}, only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \tag{7}$$

We let

$$\delta_j = \frac{\partial E_n}{\partial a_j} \tag{8}$$

where the δs are referred to as *errors*. Using 5, we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \tag{9}$$

Substituting 8 and 9 into 7, we then obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \tag{10}$$

Equation 10 tells us that the derivative we require can be obtained easily by multiplying the value of δ (for the unit at the output end of the weight) by the value of z (for the unit at the input end of the weight, where $z = 1$ in the case of a bias). Thus, in order to evaluate the derivatives, we need only to calculate the value of δ_j, for each hidden and output unit in the network, and then apply 10.

As we have seen already, for the output units, we have $\delta_k = y_k - t_k$. To evaluate the δs for hidden units, we again make use of the chain rule for partial derivatives,

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{11}$$

where the sum runs over all units k, to which unit j sends connections. The arrangement of units and weights is shown in Figure 5. Note that the units labelled k could include other hidden units and/or output units. In writing down (10, we are utilising the fact that variations in a_j lead to variations in the error function only through variations in the variables a_k. If we now substitute the definition of δ given by 8 into 11, and make use of 5 and 6, we obtain the following *backpropagation* formula:

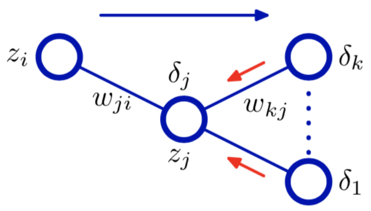


Figure 5: Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δ's from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{12}$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network.

6 Fooling the System

Our attack model, as described informally in our motivating example 3, allows the following:

1. The attacker has black-box access to the oracle model, i.e. they can freely send inputs and receive class labels, but nothing more.
2. The attacker has shell access to their victims account.
3. The IDS will signal an alarm if it observes command usage that it does not classify as the victim.

For a particular instance of this game, the attacker wins if they are able to execute their commands without the alarm being raised. In contrast, the attacker loses if they are detected.

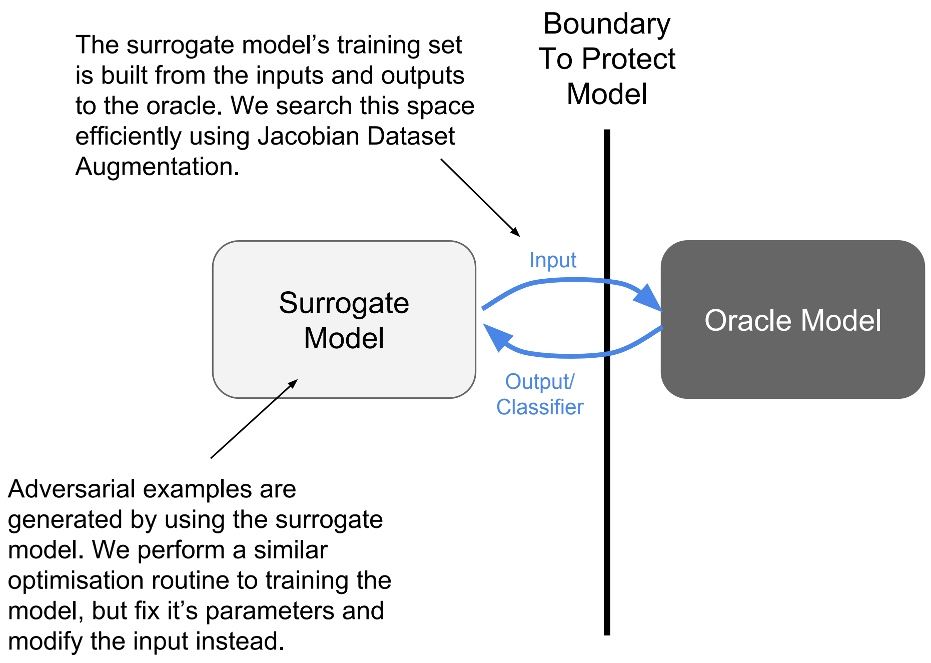

The surrogate model's training set is built from the inputs and outputs to the oracle. We search this space efficiently using Jacobian Dataset Augmentation.
Boundary To Protect Model
Input
Surrogate Model
Oracle Model
Output/ Classifier
Adversarial examples are generated by using the surrogate model. We perform a similar optimisation routine to training the model, but fix it's parameters and modify the input instead.

Figure 6: A summary of the attack strategy.

Our approach, mirroring that of Papernot et al [4], is summarised in Figure 6. Once the surrogate model has been trained, the attackers are able to generate adversarial examples offline.

In general, our neural networks and attacks work with command usage vectors, representing how often a particular used each command in a set time period. At the end of our notebook, we wrap our attack in the `masq` function, which performs the work of:

1. Calculating the command vector for a given input.
2. Using our attack to generate a similar command vector that passes as the target user.

3. Inputs this command vector into the oracle, and checks whether the attack was successful.

4. Modifies the input script to match the command counts in the adversarial example generated in step 2. It is important that the script performs an equivalent task.

5. The modified script is then returned to the caller.

This section describes how we built the surrogate model, and our survey of search strategies for finding adversarial examples.

6.1 Surrogate Model

In reality, attackers would not have access to the oracle, which is why a surrogate model would be required. This is a defined as a replica, created of the oracle, that the attackers have access to and can inspect. Attackers can use this in order to see how the oracle behaves and therefore create their attacks from it.

There are two problems that make training the substitute model challenging. We must:

- select an architecture for our substitute without having knowledge of the targeted oracle's architecture
- limit the number of queries made to the oracle to ensure that the approach is tractable

[4] We mainly overcome these challenges by introducing *Jacobian-based Dataset Augmentation* - approximating the oracle's decisions with few label queries.

6.1.1 Surrogate Architecture

To create the surrogate model the attacker must have some knowledge of the oracle - namely the inputs and outputs. Therefore, they can adapt their model's architecture to suit this input/output relation. Attackers may also want to perform an architecture exploration to find the most suitable fit.

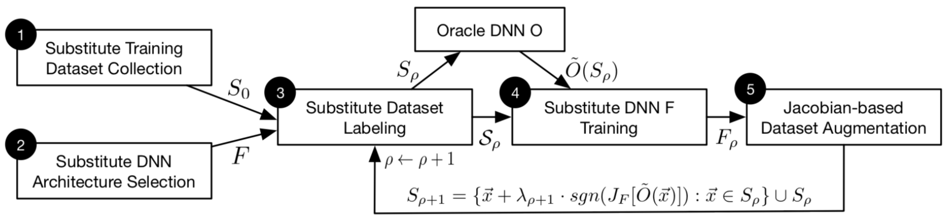

1 Substitute Training Dataset Collection
2 Substitute DNN Architecture Selection
Oracle DNN O
3 Substitute Dataset Labeling
4 Substitute DNN F Training
5 Jacobian-based Dataset Augmentation
S_0
F
S_ρ
$\tilde{O}(S_\rho)$
F_ρ
$\rho \leftarrow \rho + 1$
$S_{\rho+1} = \{\vec{x} + \lambda_{\rho+1} \cdot sgn(J_F[\tilde{O}(\vec{x})]) : \vec{x} \in S_\rho\} \cup S_\rho$

Figure 7: Stages of trainging the Surrogate Model

6.1.2 Building a Synthetic Dataset

An attacker could make an infinite number of queries to get every output for every input - giving them the oracle. However, this is not realistic and would also make it easy to detect that adversarial behaviour was happening. Therefore, we need to represent this by selecting random points to be queried and using the outputs to learn similar decision boundaries [4]. Attackers must identify directions in which the model's output is varying, around an initial set of training points. These directions are identified with the substitute Neural Network's Jacobian matrix, which is evaluated at several input points. [4]

To obtain a new synthetic training point, a term λ times the sign of Jacobian at $\overrightarrow{x}$ by the oracle at $\overrightarrow{x}$ is added to the original input point $\overrightarrow{x}$. We created our substitute training algorithm using this intuition and iteratively refining the model in directions identified using the Jacobian. This does mean that we can overfit, but that does not matter in this case.

We have added in a figure that shows the stages, using the above, of creating the surrogate model. 7

6.2 Adversarial Examples

In order to break the neural network, we had to create some adversarial examples in the form of attacks. An adversarial example is a sample of our input data which has been modified slightly, intending to cause the neural network to misclassify it [9]. There was two different types of attacks we could make:

- untargeted attacks - where we just want the oracle to output any user that is not the ‘real’, original user
- targeted attacks - where a specific user is chosen as an input (to be targeted by the attack) and we want the oracle to output this user, rather than the original user

We looked at many attacks but ended up choosing to use two different ones. One with less noise that performed well using an untargeted approach and one, more aggressive, attack that performed well targeting a specific user. Here, our main goal was to be able to write a function that took in a script and our targeted user, and have the oracle output that user.

6.2.1 Fast Gradient Sign Method

We first looked at the Fast Gradient Sign Method. This was due to it being the attack used in the black box paper [4]. The accompanying Python library to this paper [8] has a function $FastGradientSignMethod$ that performs this attack, which made it much easier to implememnnt.

Gradient descent is an optimisation algorithm. It is used to find a local minimum of a differentiable function. Tangents created by differentiating the function are used as a good approximation of the curve in a small neighbourhood. [9] We can train machine learning models using gradient descent to find the local minimum of our loss function:

$$L(x, y, a, b) = (y - a(x + b))^2 \tag{13}$$

The bigger the differences between the real and predicted values are, the larger the value of the loss function. If we want to create an adversarial example, suppose now that the model is fixed (you can’t change a and b) and you want to increase the value of the loss (so that the predicted value is further away from the real value). We can only modify the data points x and y. To make changes in a way that is not obviously detected by an observer, it makes sense to compute the derivative of the loss function according to x.

$$\frac{\partial L}{\partial x} = 2a(ax + b - y) \tag{14}$$

We can evaluate this derivative on our data points, get the slope of the tangent and update the x values by a small amount accordingly. The loss will increase and our changes will be hard to detect. This is at a high level how FGSM works. [9]

So therefore, the changes are made by the adversary crafting an adversarial sample

$$\overrightarrow{x}* = \overrightarrow{x} + \delta_{\overrightarrow{x}} \tag{15}$$

for a given sample $\overrightarrow{x}$ by computing the following change:

$$\delta_{\overrightarrow{x}} = \epsilon sgn(\nabla_{\overrightarrow{x}} c(F, \overrightarrow{x}, y)) \tag{16}$$

where perturbation $sgn(\nabla_{\overrightarrow{x}} c(F, \overrightarrow{x}, y))$ is the sign of the model's cost function gradient, given a model F with an associated cost function $c(F, \overrightarrow{x}, y))$.

As you can see from the mathematics above and heatmaps (in our Jupyter notebook) produced by this attack, there is only slight background noise added and the main commands stay the same. Therefore, to the human eye it does not look like much has changed and would be less easy to detect.

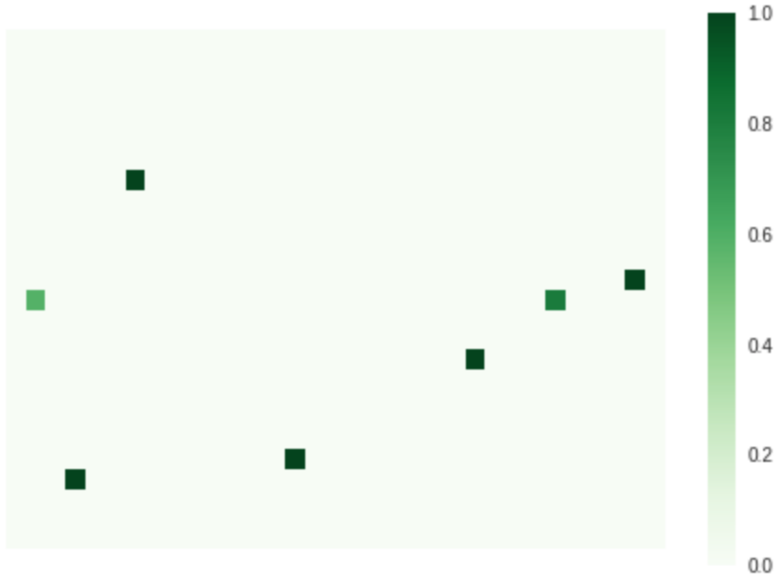


Figure 8: A heatmap showing command usage, acting as a users "fingerprint" in the NNIDS. Each square represents one particular command, with its shade signifying how often it was used. This heatmap in particular shows the original commands before modification by the attack.

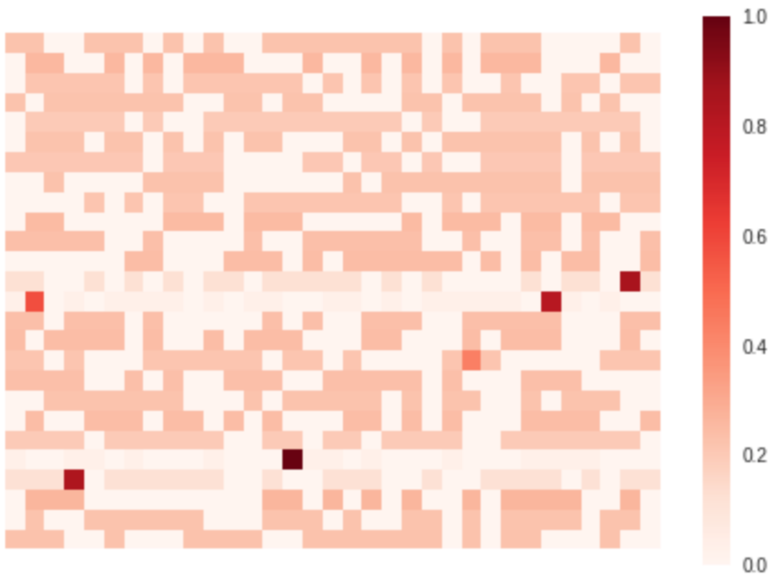


Figure 9: A heatmap showing the command usage vector generated by the FGSM attack from the input in Figure 8. Whilst noise has clearly been added, the key structure remains the same.

We found that the FGSM attack performed very well in the general case. We used Cleverhans function *model_eval* [8] to gain the accuracy for the oracle under this attack and it got 0.016%, therefore the oracle fails to predict the correct user 98% of the time.

However, we wanted to create a targeted attack. We can do this using the same $fgsm$ function by including the argument y_target. After several iterations, it was successful on average in 1/25 cases. The FGSM's attack ability is limited, because it tends to underfit the desired attack model [11]. Consequently, we decided to look at other attacks as this is quite a simple one. Therefore, something more intricate may perform better in the targeted cases.

6.2.2 Moment Iterative Method

The intrusion detection system uses integers and absolute counts as inputs, rather than percentages. A common approach to limiting adversarial attacks for image classifiers is to threshold the input image. This requires the attacker to increase the size of their perturbations to a point where they will be detected by human users. The use of integer inputs serves a similar purpose in the intrusion detection system's model.

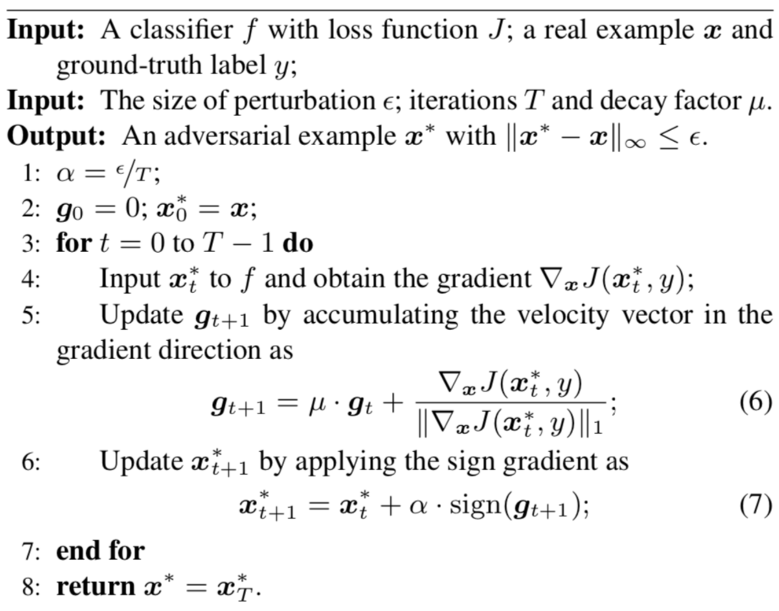
Further, unlike images, negative perturbations on the input vector are not acceptable (removing commands will change the functionality of our script). For this reason, the existing, off-the-shelf attack algorithms were not sufficient for this attack. We explored different methods of overcoming these restrictions.

Most adversarial attacks are mounted against image classification networks. They aim to minimise visual difference in their generated examples. Our attack has a different set of requirements:

- The set of commands should perform an equivalent task on the target computer.
- We can append the script with as many commands as we like.
- No commands can be removed from the script.
- All inputs must be integers.

The Moment Iterative Method seemed like a sensible choice, being an extension of the FGSM. We modify the Momentum Iterative Method introduced by Dong et al (2017) [11] to produce additive perturbations only. To do this we extended the $MomentumIterativeMethod$ attack implementation in the Cleverhans, [8] module; our implementation can be found in the $AdditiveMomentumIterativeMethod$ class in the report notebook. This work required writing and modifying custom TensorFlow code, designed to run on a GPU.

Cleverhans documentation cites Dong et al's paper [11], which states: "The momentum method is a technique for accelerating gradient descent algorithms by accumulating a velocity vector in the gradient direction of the loss function across iterations". The Moment Iterative FGSM is as follows in Figure 10.



```
Input: A classifier $f$ with loss function $J$; a real example $\boldsymbol{x}$ and
    ground-truth label $y$;
Input: The size of perturbation $\epsilon$; iterations $T$ and decay factor $\mu$.
Output: An adversarial example $\boldsymbol{x}^*$ with $\|\boldsymbol{x}^* - \boldsymbol{x}\|_\infty \leq \epsilon$.
1: $\alpha = \epsilon/T$;
2: $\boldsymbol{g}_0 = 0$; $\boldsymbol{x}_0^* = \boldsymbol{x}$;
3: for $t = 0$ to $T - 1$ do
4:     Input $\boldsymbol{x}_t^*$ to $f$ and obtain the gradient $\nabla_{\boldsymbol{x}} J(\boldsymbol{x}_t^*, y)$;
5:     Update $\boldsymbol{g}_{t+1}$ by accumulating the velocity vector in the
       gradient direction as
           $\boldsymbol{g}_{t+1} = \mu \cdot \boldsymbol{g}_t + \frac{\nabla_{\boldsymbol{x}} J(\boldsymbol{x}_t^*, y)}{\|\nabla_{\boldsymbol{x}} J(\boldsymbol{x}_t^*, y)\|_1}$;    (6)
6:     Update $\boldsymbol{x}_{t+1}^*$ by applying the sign gradient as
           $\boldsymbol{x}_{t+1}^* = \boldsymbol{x}_t^* + \alpha \cdot \mathrm{sign}(\boldsymbol{g}_{t+1})$;    (7)
7: end for
8: return $\boldsymbol{x}^* = \boldsymbol{x}_T^*$.
```

Figure 10: The above algorithm is taken directly from [11]. Our custom implementation ignores negative perturbations.

This method adds a lot more noise to the data than the previous method, as can be seen from the heatmaps in the Jupyter notebook. The main commands are not kept the same and the patterns are lost. For an observer this may seem easy to detect, but this method is actually fairly difficult for a computer to detect.

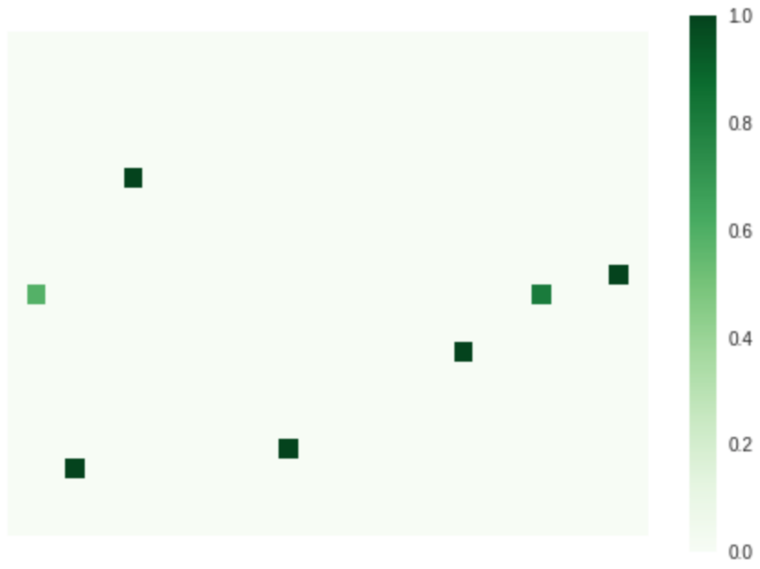


Figure 11: An example command usage fingerprint before being being processed by the MI-FGSM attack.

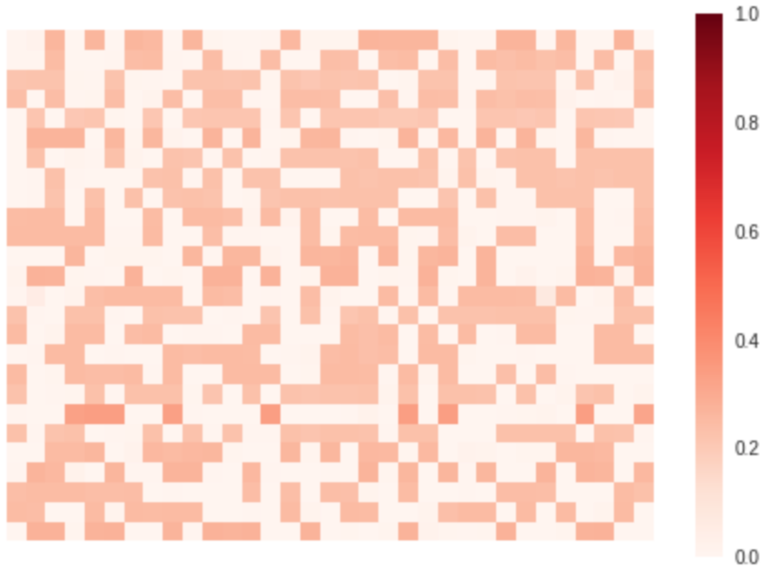


Figure 12: The resulting fingerprint from running the MI-FGSM attack on the input in Figure 11. This image bears little resemblance to the original, but we are able to generate an equivalent computer program nonetheless. This highlights the difference in requirements between our attack, and those which work on image classifiers.

The main disadvantage with this method is it can do the opposite to the normal FGSM and overfit the model. This attack did perform better on targeted users - usually getting just under 50% correct. However, the overfitting may be why it was not even better.

Lastly, we wrote a custom search function that could take in any user of our choosing and any list of commands and be able to fool the oracle into predicting that user. As you can see, this works for our example in the notebook. It is able to successfully produce examples around half of the time, this is in line with current work in producing adversarial examples with a specific input and target [13, Figure 12].

7 Evaluation

Overall, we have constructed a good oracle by successfully reproducing the results of [3], achieving an accuracy of 98.6% against the left-out test data. Thanks are due to the authors for describing the architecture and workings of their system in sufficient detail.

Our surrogate model was also very successful, against both true labels (96.0% accuracy) and against the oracle (96.4% accuracy). It is interesting to note that the surrogate was a better replica of the oracle than it was a true classifier.

As stated in the introduction, our goal was to break this Intrusion Detection System. We attempted two types of attacks: untargeted and targeted. We justify our attack design by fooling the Intrusion Detection System, gaining misclassification on 98% of the examples and source/targeted misclassification on just under 50%.

Source/target misclassification examples are the hardest class of attack to generate; with the input and label both set, the attacker has minimal flexibility in their search space. Our results mirror those of the literature [13, Figure 12].

Further, not only have we shown this works in theory, we have developed a working proof-of-concept for this attack.

Table 1: Comparison of targeted attack classification accuracies using FSGM vs AMI

FSGM	AMI
4.1%	32.6%

It is worth highlighting an inconsistency within our attack. In particular, we only train the oracle on input vectors which sum to 100, but we generate examples whose input vectors can sometimes sum to over 10,000. This is a practical issue that could be worked through via careful dilution of individual instructions, rather than acting on the script as a whole.

Further, if the IDS were to take constant readings every 100 commands, then the output of our "masq" script might not work. In short, we know the counts from the whole 8000 line script input at once will fool the network, but any particular 100 command block within it might not. We believe these restrictions could be overcome.

8 Future Directions

Given the chance to return to this project, instead of copying the architecture of the oracle model, we would research into the different potential models that our surrogate could follow in order to flexibly model an unknown architecture. As suggested in the surrogate architecture section, if we had time to do so it would be good to perform architecture exploration: building many models in order to see which architecture works best.

In terms of the targeted attacks, we would also like to add an extension of our attack model which allows us to remove commands from scripts. This would require a more intricate script rewriting technique, however.

It would also be interesting to develop an adversarial example optimisation function from scratch. This would be a much more involved process, and certainly outside the scope of this project, but it would allow us to further exploit the difference in requirements between the image classification and user fingerprinting.

To make our attack model more practical for attackers, it may be possible to create a surrogate that just classifies a specific user by monitoring their activity. This provides a more accessible attack model. For example, an attacker could monitor J. Dones's key strokes, and build a surrogate model that only performs binary classification on the J. Dones. Using this surrogate, we could evaluate the effectiveness of this binary classifier transferred to the multi-class oracle.

We wanted to make our model transferable and to do this we would need to see how well it worked against the oracle. We didn't have time to create a metric for this but, if we had, it would have been an even better indicator of how good our model is.

9 Conclusion

Overall, we have really enjoyed experimenting on this project and feel we have made a very successful attempt at completing our inference goals. Given more time, we would have gained great fulfilment from exploring the ideas in our *Future Directions* section.

The final code cell of our report notebook allows you to input your own malicious scripts and, if it's successful, will output an equivalent script identified by the IDS as your target user. We encourage you to try your own examples, and see how the "aggressiveness" parameter controls the trade-off between flexibility in choosing targets and resulting length of the modified script.

We hope you enjoyed our exploration into neural networks and their security implications as much as we did. Thanks for reading!

References

[1] Vinayakumar R, Barathi Ganesh HB, Prabaharan Poornachandran, Anand Kumar M, Soman KP (Dec 2018). Deep-Net: Deep Neural Network for Cyber Security Use Cases

[2] Leonid Perlovsky, Olexander Shevchenko (July 2014) Cognitive neural network for cybersecurity

[3] Ryan, J., Lin, M.J. and Miikkulainen, R., 1998. Intrusion detection with neural networks. In Advances in neural information processing systems (pp. 943-949).

[4] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, Ananthram Swami Practical Black-Box Attacks against Machine Learning

[5] Schonlau, M. Masquerading User Data http://www.schonlau.net/intrusion.html

[6] Ognjanovski, G. (2014). Everything you need to know about Neural Networks and Backpropagation - Machine Learning Easy and Fun. *Towards Data Science.* https://towardsdatascience.com/everything-you-need-to-know-about-neural-network

[7] Christopher M. (2006). Bishop. Pattern Recognition and Machine Learning (Information Science and Statistics). SpringerVerlag New York, Inc., Secaucus, NJ, USA.

[8] Cleverhans Python Library https://github.com/tensorflow/cleverhans/

[9] Gregory Chatel (Jun 2017) Adversarial examples in deep learning

[10] Yanpei Liu, Xinyun Chen, Chang Liu, Dawn Song Delving into Transferserable Adversarial Examples and Black-Box Attacks

[11] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, Jianguo Li (March 2018) Boosting Adversarial Attacks with Momentum

[12] Daniel Geng, Rishi Veerapanieni (Jan 2018) Tricking Neural Networks - Creating your own Neural Networks, https://medium.com/@ml.at.berkeley/tricking-neural-networks-create-your-own-adversarial-examples-a61eb7620fd8

[13] Papernot, Nicolas, et al. "The limitations of deep learning in adversarial settings." 2016 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2016.