# Malware Intrusion Detection – Final Project

## Detection of SSH Brute-Force Attacks Using Machine Learning

## Gilad Samuels and Dor Aharonson

### 8.a - Our implementation of the articles ML technique

Using sqlite3, we extracted from the DB all the entries along with their labels (bruteforce and non-bruteforce) and the values of the 11 features that the authors of the article chose to use. We implemented a generic KNN algorithm implementation and fed it the data which was split to 50% training and 50% testing (using sklearn’s “train\_test\_split” method). This is also how they split the data in the article. For each point in the testing data, we checked what label it should get using the KNN algorithm and then we compared the results with the real labels from the DB, and used sklearn’s metrics (confusion\_matrix, classification\_report) to calculate the appropriate measurements.

### 8.b – Technique complexity analysis

Let’s use as the training data size, as the testing data size and as the number of features. Finding Euclidian distance is in run-time complexity, iterating over current closest neighbors is , we do this times for each testing point and there are such points. Thus, the complexity is . Notice that in our case, so the run-time complexity really is .

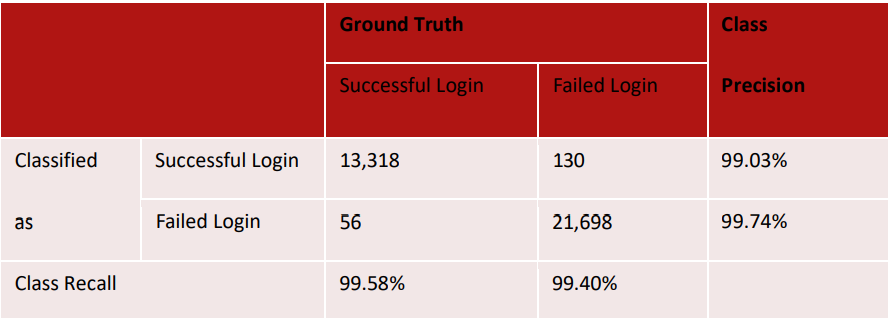
### 9.a – Results

Our results for with the given features (as described in part A of our work) are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **Ground Truth** |  | **Class** |
|  |  | Successful Login | Failed Login | **Precision** |
| Classified | Successful Login | 3264 | 2 | 99.94% |
| as | Failed Login | 11 | 4223 | 99.74% |
| Class Recall |  | 99.66% | 99.95% |  |

### 9.b – Analysis

These results are impressive. It should be noted that in the article they chose the features specifically so they show the best results for this dataset, and we are not sure that they would exhibit the same results for other traffic that would be generated in a different manner. In the article they achieved similar results:



### 10.a – Implementation of our technique

We did a few things differently than the authors of the article.

First, feature selection. In the article they removed some features that had a low information gain ratio, then they reduced the number of features based on run time performance. Initially we tried to use Information Gain (the mutual information between the variable of the labels, and the variable of the feature) in order to discern which features would be valuable. Doing so is straightforward for a feature that describes a discrete value but is harder for a feature that has a continuous value. Eventually we settled on sklearn’s implementation for this problem. This logic is implemented in “features\_extractor.py”. Using these features yielded much worse results (~85% precision) than using the previous features. Instead, we ran SVM with a linear kernel on the full DB and chose the 11 features that had the highest weight. In SVM, these features are supposed to be the most influential in determining the label (creating the margin between the labels). We chose 11 features specifically because in the article they chose 11 features, so that comparing the results would be easier.

Second, the ML technique. In addition to testing on KNN, we tested our results on SVM. We theorized SVM would do better than KNN, and we wanted to test this theory and compare the results of both algorithms, using different parameters. So, we ran KNN for and ran the SVM on linear, polynomial and gaussian kernels. In order to run the SVM we used the accepted implantation by sklearn. Like in the article, half of the data was used for learning and half for testing.

In addition, in order to look at each feature in an equal manner, we normalized all the values using the following formula

### 10.b – Complexity

According to sklearns’ documentation, SVM implementation scales between and depending on the implementation of the libsvm cache ([1.4. Support Vector Machines — scikit-learn 0.24.1 documentation (scikit-learn.org)](https://scikit-learn.org/stable/modules/svm.html#complexity)). In our case so the complexity is similar to that of KNN. Interestingly, the SVM runs much faster on our computers than the KNN (which we implemented ourselves, so it makes sense that it’s not as efficient as possible).

### 11.a – Results

We took the following 11 features:

['avgp\_len', 'ldp\_len', 'duration', 'dp\_13\_bytes', 'bytes\_in', 'med\_ipt', 'aes192-cbc', 'num\_pkts\_out', 'aes256-cbc', 'dp\_16\_bytes', 'arcfour128']

They had the highest (absolute) values in the linear SVM. The full results (weights) for all features are:

[('avgp\_len', -10.07403554353644), ('ldp\_len', -9.08677730453092), ('duration', 7.227974391477782), ('dp\_13\_bytes', -2.891264861889827), ('bytes\_in', -2.8438336135850943), ('med\_ipt', 2.567802790076197), ('aes192-cbc', 2.400719707128059), ('num\_pkts\_out', -2.116824596050655), ('aes256-cbc', -2.1003584231139416), ('dp\_16\_bytes', -1.9122931029614072), ('arcfour128', 1.7206300167104804), ('arcfour', 1.7170878033346204), ('blowfish-ctr', -1.6892465838874375), ('num\_pkts', -1.363307307992732), ('aes256-gcm@openssh.com', -1.3113942412645128), ('cast128-cbc', -1.2581464344212914), ('rijndael-cbc@lysator.liu.se', -1.2581464344212914), ('3des-ctr', 1.1988124046887942), ('aes128-ctr', -0.9047289804745879), ('aes128-cbc', 0.8996415768860588), ('kex\_algos\_lenght', 0.8386358209099964), ('dp\_7\_bytes', -0.7898202701021602), ('arcfour256', 0.7367145788686501), ('aes192-ctr', 0.70829016119118), ('dp\_15\_bytes', -0.6839157590263363), ('dp\_8\_bytes', 0.6746184915191143), ('medp\_len', 0.6445620390756319), ('aes256-ctr', 0.5111175286787997), ('dp\_10\_bytes', -0.5056429507327966), ('dp\_11\_bytes', 0.404495420648992), ('dp\_12\_bytes', -0.3601622981122324), ('encryption\_algo\_lenght', -0.3576141760375551), ('dp\_9\_bytes', 0.2314864245095346), ('dp\_6\_bytes', -0.1724604441272813), ('bytes\_out', -0.14486275527174725), ('num\_pkts\_in', -0.13724527996628602), ('fin\_count', 0.13067432148217017), ('reset\_count', 0.12442328467710662), ('aes128-gcm@openssh.com', -0.11576479862515954), ('chacha20-poly1305@openssh.com', -0.11576479862515954), ('dp', 0.06403119737366779), ('3des-cbc', -0.05579181635663799), ('blowfish-cbc', -0.05579181635663799), ('fdp\_len', 0.0339759671421509), ('sp', -0.004032962878471769), ('twofish128-cbc', 0.0035422133758599935), ('twofish256-cbc', 0.0035422133758599935), ('idp\_in', 0.0), ('idp\_out', 0.0), ('ssh', 0.0), ('des-cbc-ssh1', 0.0), ('support\_other\_enc', 0.0), ('fp\_bytes', 0.0)]

The best results were for a gaussian kernel:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **Ground Truth** |  | **Class** |
|  |  | Successful Login | Failed Login | **Precision** |
| Classified | Successful Login | 3315 | 3 | 99.91% |
| as | Failed Login | 0 | 4182 | 100.00% |
| Class Recall |  | 100.00% | 99.93% |  |

You can observe in the following charts the results for all SVM’s:

Linear:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **Ground Truth** |  | **Class** |
|  |  | Successful Login | Failed Login | **Precision** |
| Classified | Successful Login | 3228 | 3 | 99.91% |
| as | Failed Login | 42 | 4227 | 99.02% |
| Class Recall |  | 98.72% | 99.93% |  |

Polynomal:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **Ground Truth** |  | **Class** |
|  |  | Successful Login | Failed Login | **Precision** |
| Classified | Successful Login | 3330 | 452 | 88.05% |
| as | Failed Login | 0 | 3718 | 100.00% |
| Class Recall |  | 100.00% | 89.16% |  |

### 11.b – Analysis

We can see that we got very good results. The SVM worked very well using the features we chose. We expected to get good results for SVM since we also used it in order to choose features. By observing the weight given to each feature we can see that there are 3 main features that pulled the most weight - average packet length, last data packet length and duration. .

### 12.a. – Comparing precision

* Precision for 5NN with their features:

|  |  |
| --- | --- |
|  | Percision |
| Successful Login | 99.94% |
| Failed Login | 99.74% |
| Average | 99.84% |
| Weighted Average | 99.83% |

* Precision for 5NN with our features:

|  |  |
| --- | --- |
|  | Percision |
| Successful Login | 99.97% |
| Failed Login | 99.81% |
| Average | 99.89% |
| Weighted Average | 99.88% |

* Precision for gaussian SVM with our features:

|  |  |
| --- | --- |
|  | Percision |
| Successful Login | 99.91% |
| Failed Login | 100.00% |
| Average | 99.95% |
| Weighted Average | 99.96% |

### 12.a. – Comparing recall

* Recall for 5NN with their features:

|  |  |
| --- | --- |
|  | Recall |
| Successful Login | 99.66% |
| Failed Login | 99.95% |
| Average | 99.81% |
| Weighted Average | 99.83% |

* Precision and recall for 5NN with our features:

|  |  |
| --- | --- |
|  | Recall |
| Successful Login | 99.76% |
| Failed Login | 99.98% |
| Average | 99.87% |
| Weighted Average | 99.88% |

* Recall for gaussian SVM with our features:

|  |  |
| --- | --- |
|  | Recall |
| Successful Login | 100.00% |
| Failed Login | 99.93% |
| Average | 99.96% |
| Weighted Average | 99.96% |

### 12.b – Comparing errors

* Errors for 5NN with their features:

|  |  |
| --- | --- |
|  |  |
| False positive | 11 |
| False negative | 2 |
| Error rate | 0.17% |

* Errors for 5NN with our features:

|  |  |
| --- | --- |
|  |  |
| False positive | 8 |
| False negative | 1 |
| Error rate | 0.12% |

* Errors for gaussian SVM with our features:

|  |  |
| --- | --- |
|  |  |
| False positive | 0 |
| False negative | 3 |
| Error rate | 0.04% |

### 12.c – Analysis

When we chose our features, we looked at the weights of the features when classified by SVM, thus choosing features that are suited for this kind of classification. The authors of the paper chose their features by trying to calculate the information gain. This is hard to do for some type of values (the available features have categorical, discrete, and continues data), and because of that it might not be the best way to select features for learning on this data.

Generally, it makes sense that these two methods performed well on the given data. When you look at these kinds of features of an SSH login attempt you observe parameters that are determined mostly by the protocol, and some by implementation. Because of that, the behavior is almost predictable. The reason it is not fully predictable is the complexity of the protocol and its implementations. Using SVM and KNN can help us avoid dissecting every possible flow of the protocol for each implementation in the case of a successful login attempt and in the case of a failed login attempt, and instead help us rely on some possible flows, and classify other flows by their closeness to those flows. The “closeness” that affects KNN and SVM probably translates well to the similarity of two flows with regards to what caused them.