# 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 all labels from the DB and the values of the 11 features that the article chose to use. We implemented a generic KNN algorithm and fed it the data which was split to 50% training and 50% testing (using sklearns ‘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 analyzed our results using the real labels from the DB and sklearns metrics (confusion\_matrix, classification\_report).

### 8.b – Technique complexity

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

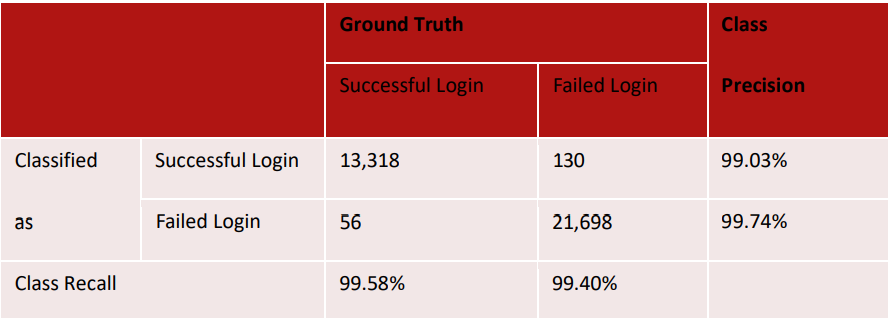
### 9.a – Results

Our results specifically on with the given features are as follows:

* The confusion matrix is <INSERT\_RESULT>
* The classification report is <INSERT\_RESULT>

### 9.b – Analyze

These results are really good. It should be noted that in the article they chose the features specifically so they show the best results for this database, and we are not sure that they would exhibit the same results for more traffic that would be generated in a different manner. It is worth noting that in the article they achieved similar results:



### 10.a – Implementation of our technique

We did a few things different than the article.

First, feature selection. In the article they simply played with features and “guessed” what would result in good results and performed “trial and error” until they got good results. Initially we tried to use Information Gain in order to discern which features would be valuable. This logic is implemented in ‘features\_extractor.py’ but it yielded much worse results (~85% precision). 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 also chose 11 features.

Second, the ML technique. In addition to testing on KNN, we tested our results on SVM. We wanted to see what would be the best results on 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. Similar to 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 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 (<VALIDATE>):

* The confusion matrix is <INSERT\_RESULT>
* The classification report is <INSERT\_RESULT>

You can observe in the following charts the results for all KNN’s and all SVM’s with different parameters:

* Precision results: <INSERT\_CHART>
* Recall results: <INSERT\_CHART>

### 11.b – Analyze

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. Indeed by looking at the samples that were labelled wrong, we can see that they have values that are different than most of those what were declared benign and more similar to those who were labeled as bruteforce.

### 12.a – Comparing precision

* Precision for 5NN with their features: <INSERT\_RESULT>
* Precision for 5NN with our features: <INSERT\_RESULT>
* Precision for gaussian SVM with our features: <INSERT\_RESULT>
* All precisions chart: <INSERT\_CHART>

### 12.a – Comparing precision and recall

* Precision and recall for 5NN with their features: <INSERT\_RESULT>
* Precision and recall for 5NN with our features: <INSERT\_RESULT>
* Precision and recall for gaussian SVM with our features: <INSERT\_RESULT>
* All precisions chart: <INSERT\_CHART>
* All recalls chart: <INSERT\_CHART>

### 12.b – Comparing errors

* Errors for 5NN with their features: <INSERT\_RESULT>
* Errors for 5NN with our features: <INSERT\_RESULT>
* Errors for gaussian SVM with our features: <INSERT\_RESULT>
* All errors chart: <INSERT\_CHART>

### 12.c – Analyze

Something about them choosing features so it will work well with KNN and us choosing features using SVM.