SOFTWARE PIRACY DETECTION

A Term Project

Presented to

Dr. Mark Stamp

Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements for the Class

CS 271

By

Shreya Kundu and Gayathri Hanuma Ravali Kuppachi

Apr 30, 2019

I. INTRODUCTION

Software piracy is unauthorized use , copying or distribution of licensed software. When we purchase a software, we agree to an end-user license agreement (EULA) which prevents us from copyright infringement. Typically the EULA states that we can install only one copy of the software bought and can make a backup copy in case the original is lost or damaged.

Any kind of following activities fall under the definition of software piracy:

- Softlifting : Borrowing and installing a copy of software application from another person that has not been paid for.
- Client-server overuse: Installing more copies of the software than you have licenses for.
- Hard-disk loading: Installing and selling unauthorized copies of software on refurbished or new computers.
- Counterfeiting: Duplicating and selling copyrighted programs.
- Online piracy: Typically involves downloading illegal software from peer-to-peer network, Internet auction or blog.

The objective of this project is to develop techniques that help to detect modified pirated software. Detection of unmodified software is comparatively a trivial problem. An attacker might modify software as a way to obtain the functionality, while trying to maintain an air of legitimacy and also avoid copyright infringement issues [5]. Ideally, we would like to force the attacker to make major changes to the software before we cannot reliably detect it.

II. BACKGROUND

The goal of this project is to distinguish pirated copies of software from original software. This section focuses on the background required to implement this project.

The first step is collection of data or creating a dataset. For this project, we have created our own dataset by extracting opcodes of Elmedia Video Player (a video/audio executable file) using a disassembler tool called Hopper Debugger. To gather enough data, 50000 opcodes were extracted from this software. These 50000 opcodes sequence is the base model or benign software used throughout the project.To avoid overfitting, we selected only 29 most frequent opcodes and labeled everything else as “Other”. These 30 most frequent opcodes are sufficient to analyze the opcode frequency and patterns statistically.

The next step is to create multiple benign and malware copies from the base file used for testing purposes.A morphing generator was used to create malware samples and benign samples of the base version to be used for testing purposes which is discussed in the implementation section.

Different machine learning techniques have been applied on this data which are Hidden Markov Models, Opcode Graph Similarity and Support Vector Machines. The implementation and results are discussed in following sections and the results were decently organized to be able to classify pirated software from the actual software.

III. IMPLEMENTATION

This section focuses on the implementation details of this project. From the previous section, we have a dataset of benign file with 50000 sequence of opcodes. Now, we have implemented a morphing generator which creates test files of benign as well as pirated softwares.

3.1 Morphing Generator:

There are two approaches used to create morphed files of the original sequence.

1) Code Permutation:

In this method code is divided into smaller frames or modules and these are randomly rearranged keeping the logic of the program as it is by inserting various jump instructions in between.

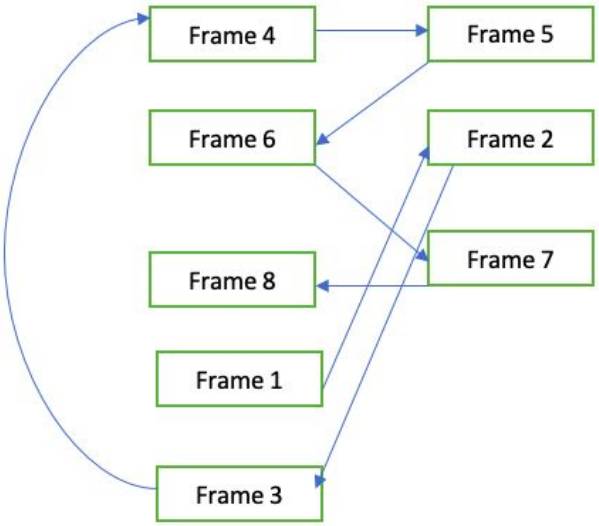


Fig : Code Permutation

2) Dead Code Insertion:

In this method sequence of unrelated opcodes is inserted into random position of our original opcode sequence. This is similar to null operation. This does not alter the logic and functionality of the program but skews the statistical property of the sequence and makes statistical analysis difficult.We selected a completely unrelated file and extracted the opcodes of total length 5000. In our case the original opcode sequence length is 50,000. 10% morphing means we inserted 5000 opcodes as dead code. 20% morphing means we inserted 10000 opcodes as dead code and so on. The method followed was multiple chunks of 5000 opcodes extracted from an unrelated file was inserted at random positions of the original sequence.If 10% morphing was used 2 random chunks were inserted.If 20% morphing is considered 4 random chunks were inserted.Both of this methods were implemented in Python.

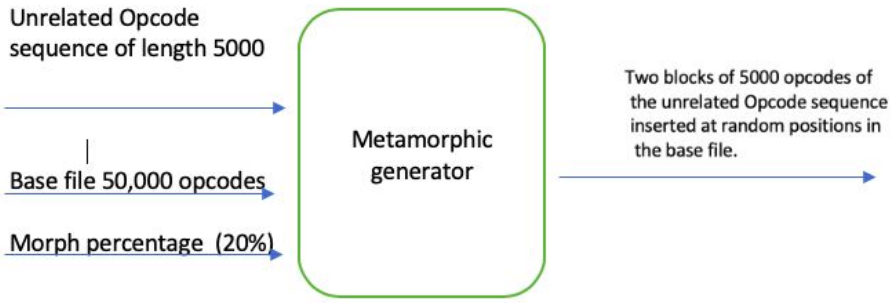


Fig : Metamorphic Generator

3.2 Implementation of Hidden Markov Model:

A Hidden Markov model (HMM) is a machine learning technique [6]. As the name suggests, an HMM includes a Markov process and this process is the "hidden" part of the HMM. That is, the Markov process is not directly observable. But we do have indirect information about the Markov process via a series of observations that are probabilistically related to the underlying Markov process. The utility of HMMs derives largely from the fact that there are efficient algorithms for training and scoring.

For training the Hidden Markov Model, we extracted 50000 opcode sequences from the base software (Elmedia Video Player). Then, we trained our model on these opcode sequences. We started with random initialization of A, B and pi matrices where A refers to state transition probabilities , B refers to observation probability matrix and pi refers to initial state distribution. Then, we experimented the model with various values of N where N refers to number of states in the model and M refers to number of observation symbols which is 30 in this case as the number of opcodes used were 29 which belong to most frequent opcodes and the 30th opcode is "Other" which refers to all the other opcodes in 50000 opcode sequences.

After various experiments on N values, we found that an N value with 3 was converging and we were able to extract the A, B and pi matrices along with the log likelihood probability.
The next phase was the scoring phase, where we scored each of the benign files which was extracted from 2 to 11 jumps in morphing generator and also morphed copies of 5 to 90 percent were extracted. Once, the scores were determined we normalized the values as they were on a

larger scale by dividing them with the length of opcode sequences of test files. The scores and the graphs obtained before and after normalization are shown in the next section in results.

3.3 Implementation of Opcode Graph Similarity:

Opcode similarity Graph is graph based technique analogues to digraph matrix constructed from Brown Corpus for English Language based key-detection for substitution ciphers. In case of digraph matrix we calculated the probability of particular alphabet followed by another based on the brown corpus data .

In OGS we are constructing a matrix that gives us the probability of Opcode "i" followed by Opcode "j" in our original opcode sequence. We consider the top 29 most frequent distinct opcodes as mentioned above and everything else as "Other". Hence our OGS matrix is 30X30. We traverse through the entire sequence and whenever we find PUSH followed by POP we increment value of row_index equivalent of PUSH and col_index equivalent of POP by 1. After the matrix is formed it is normalized by row sum.

We calculated OGS matrix for each of our morphed samples and permuted samples and calculated the OGS score with the OGS matrix calculated from the original opcode sequence.The formula for calculating OGS score is as follows:

$$\texttt{score}(A,B) = \frac{1}{N^2}\left(\sum_{i,j=0}^{N-1} |a_{ij} - b_{ij}|\right)^2 . \quad [2]$$

A and B are two matrices of dimension NxN.$a_{i,j}$ and $b_{i,j}$ are the probabilities of opcode i followed by opcode j in sequences A and B respectively.

3.4 Implementation of Support Vector Machines:

Support Vector Machine (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems. In this algorithm, we have obtained the scores from OGS and HMM from the previous section and would like to see how SVM classifier performs on these scores.

Since, there are 2 to 11 conditional jumps which indicate that we have 10 benign samples with a classifier 0 and 10 malware samples from 5% to 90% morphing with a classifier 1. The data is evenly distributed and we split the data into 80-20 percentage where 80% is used for training the data and 20% is used for testing the data.

Since, SVM utilizes the concept of Kernels, we experimented our data with three different kernels which are:

1) **Linear**: A function which can be represented as $(x, \bar{x})$
2) **RBF**: An exponential function which can be represented as $\exp(-\gamma \|x - \bar{x}\|^2)$. γ where gamma, must be greater than 0.
3) **Sigmoid**: A function which is represented as $\tanh(\gamma (x,x') + \gamma)$, where γ is specified by coef0.

The results and graphs of SVM can be observed in the results section.

IV. EXPERIMENTAL RESULTS

4.1 HMM Scores:

Below are the scores obtained when we train HMM for N=3.

The log likelihood estimate is considered as a score and we use that as a base to score rest of the test samples. The scores obtained before normalization are as follows:

HMM Score	Morphed Percentage
43257.19	Benign (0)
40709.56	5
38862.73	10
33972.172	20
29546.88	30
25253.69	40
20815.815	50
16359.98	60
11809.45	70
7319.46	80
2556.76	90

Table : HMM versus Morphing Percentage before normalization

We can clearly observe that as the morphing percentage increases , the score decreases which is kind of logical which indicates that more the morphing, more the deviation from original file.

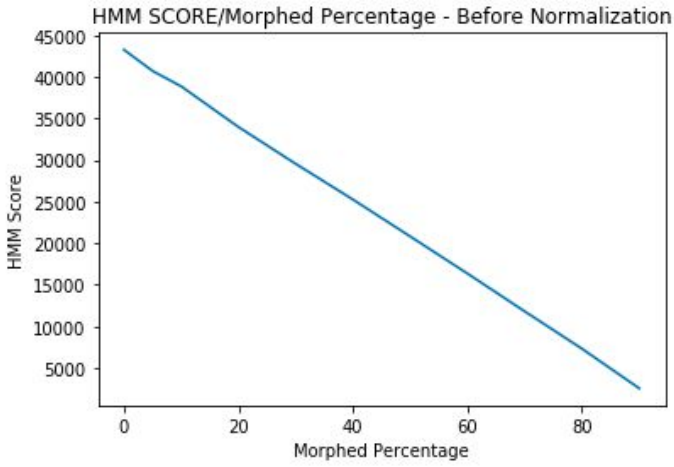


Fig : HMM versus Morphing Percentage before normalization

Since the scales are difficult to visualize, we normalize the scores by dividing each of the score with the length of opcode sequences and below are the results achieved after normalization.

HMM Score	Morphed Percentage
1.0	Benign (0)
0.94	5
0.89	10
0.78	20
0.68	30
0.58	40
0.48	50
0.37	60
0.27	70
0.16	80
0.05	90

Table: HMM versus Morphing Percentage after normalization

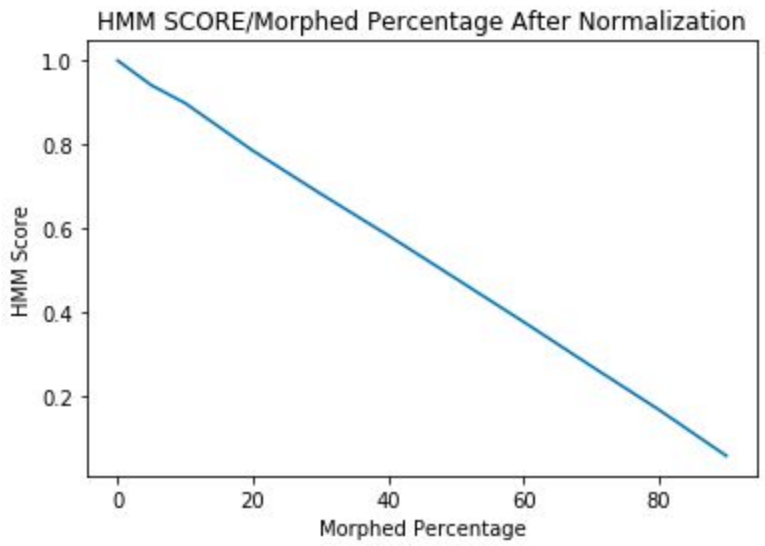


Fig : HMM versus Morphing Percentage before normalization

Similarly, benign samples were scored against base bening model of 50000 opcodes where benign samples were created using conditional jumps of 2 to 11.

Below are the scores obtained before normalization :

HMM Score	No. of Jumps
43231.15	2
43224.41	3
43224.46	4
43243.99	5
43203.11	6
43207.22	7
43220.35	8
43211.52	9
43218.25	10
43209.91	11

Table: HMM Score versus No. of Jumps before normalization

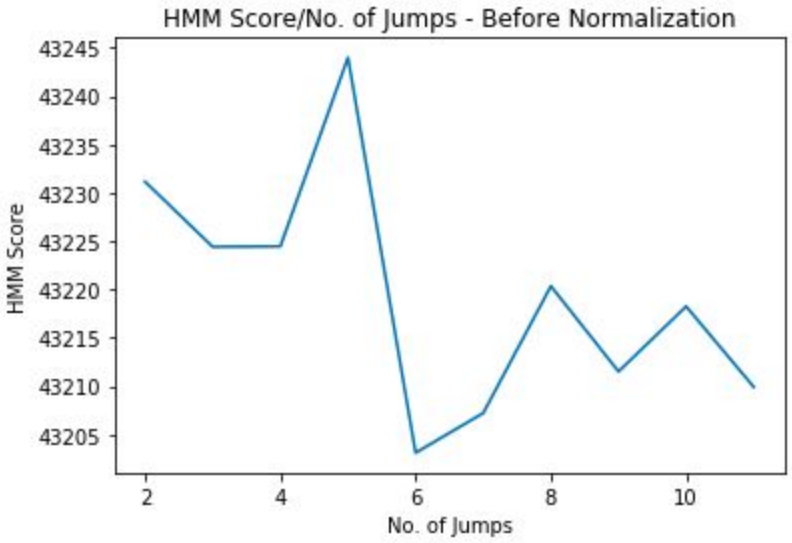


Fig: HMM versus No. of Jumps before normalization

As observed, the scores are pretty close by which is as expected because we are adding very few changes to the existing file. The graph is shown based on the scale plotted. Since, the scores are big, we normalize the scores like previously described .

HMM Score	No. of Jumps
0.9997	2
0.9995	3
0.9995	4
1.0	5
0.9990	6
0.9991	7
0.9994	8
0.9992	9
0.9994	10
0.9992	11

Table: HMM Score versus No. of Jumps after normalization

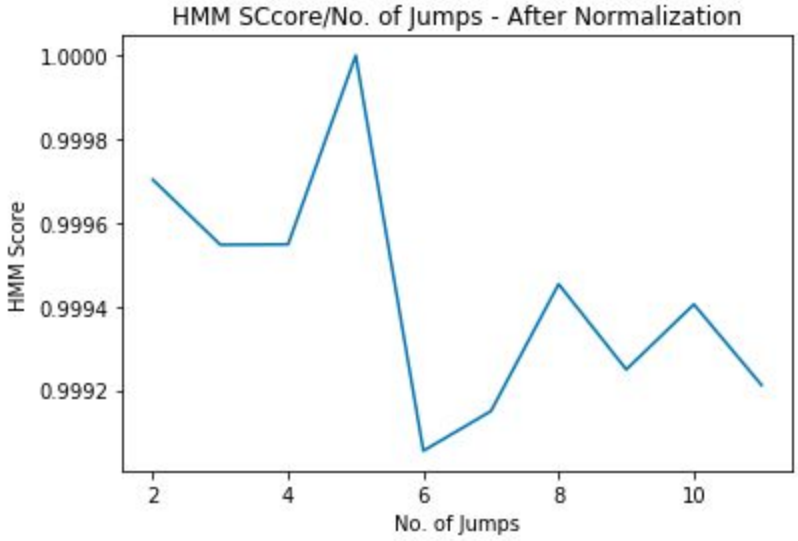


Fig : HMM Score versus No. of Jumps after normalization

4.2 OGS Scores:

Below are the scores obtained when we train and score benign and morphed samples using OGS Graph Scores.

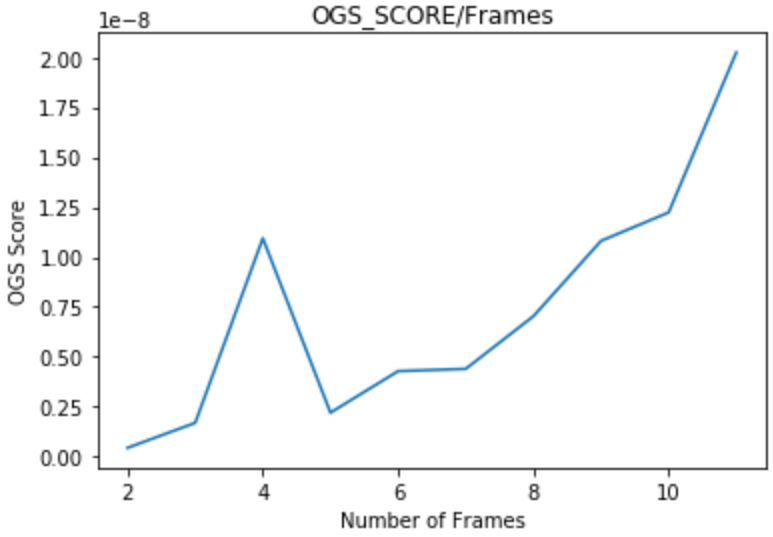
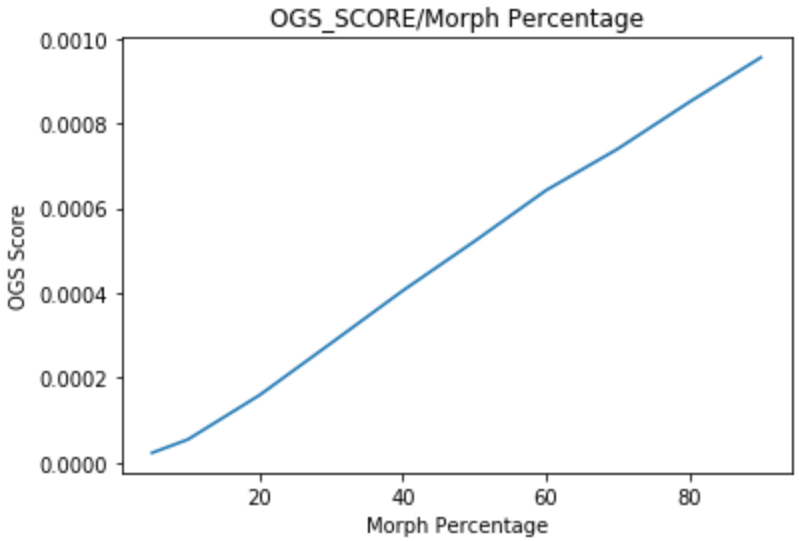


Fig : OGS Score Vs Number of Jumps before normalization[Scale ->10^-8]

OGS Scores (Benign files)	**No. of Jumps**
4.3898435706905466e-10,	2
1.6890125462892787e-09,	3
1.0938223458975751e-08,	4
2.2005442611029853e-09,	5
4.281433750605438e-09,	6
4.394465800568494e-09,	7
7.03048964354161e-09,	8
1.0806301977215938e-08,	9
1.223138572189637e-08,	10
2.02471229231244E-08,	11



OGS Scores (Morphed Files)	**Morph Percentage**
2.2505578991129887e-05,	5%
5.400334747519937e-05,	10%

0.00015866225533431248,	20%
0.0002814667915545462,	30%
0.0004052208028375009,	40%
0.0005217241009574086,	50%
0.0006420971635389988,	60%
0.0007398751527365572,	70%
0.0008504270649230536,	80%
0.000955858537587644,	90%

We can see that OGS score kept on steadily increasing as more and more dead code is added. This is apparent as dead code insertion statiscally alters the opcode distribution of the file. For the permutation of code blocks technique, ogs score is much smaller [on a scale of approximately 10^-10] compared to dead code insertion as this does not alters the opcode distribution of the file statistically as same opcodes are randomly shuffled.

These values are then normalized from a scale of 0 to 1 before passing them on as a feature to SVM so that both HMM and OGS scores remain on a same scale before it is fed to SVM classifier.

4.3 SVM Scores:

For SVM Classifier, we provide HMM and OGS Scores as features to SVM and we already have classifiers of benign and malware represented as numbers 1 and 0. Below are few plots of HMM score vs classifier, OGS score vs classifier and SVM vs Various Kernels.

Note: Red dots indicate malware and blue dots indicate benign.

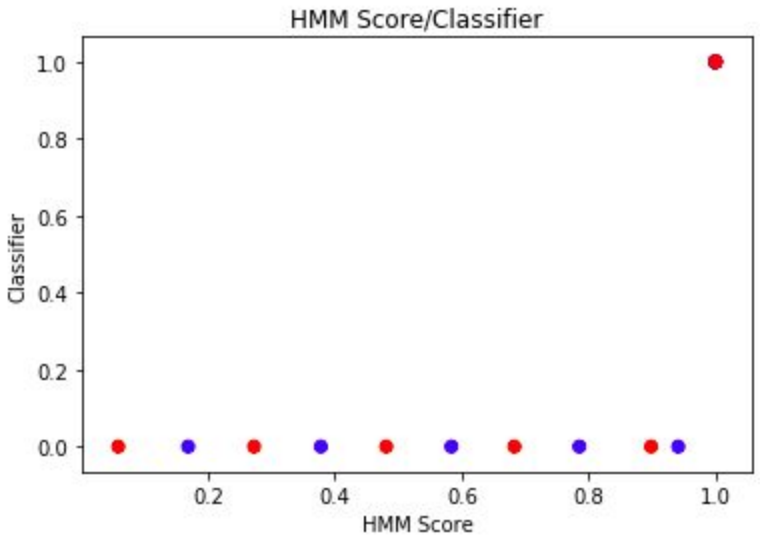


Fig : HMM Score versus Classifier

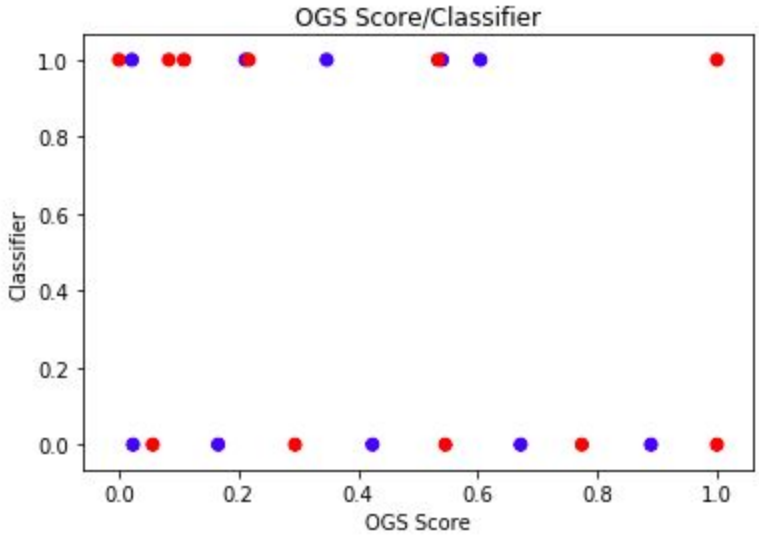


Fig : OGS Score versus Classifier

HMM and OGS data points are plotted using scatter plot and we can observe that they are jumbled up.

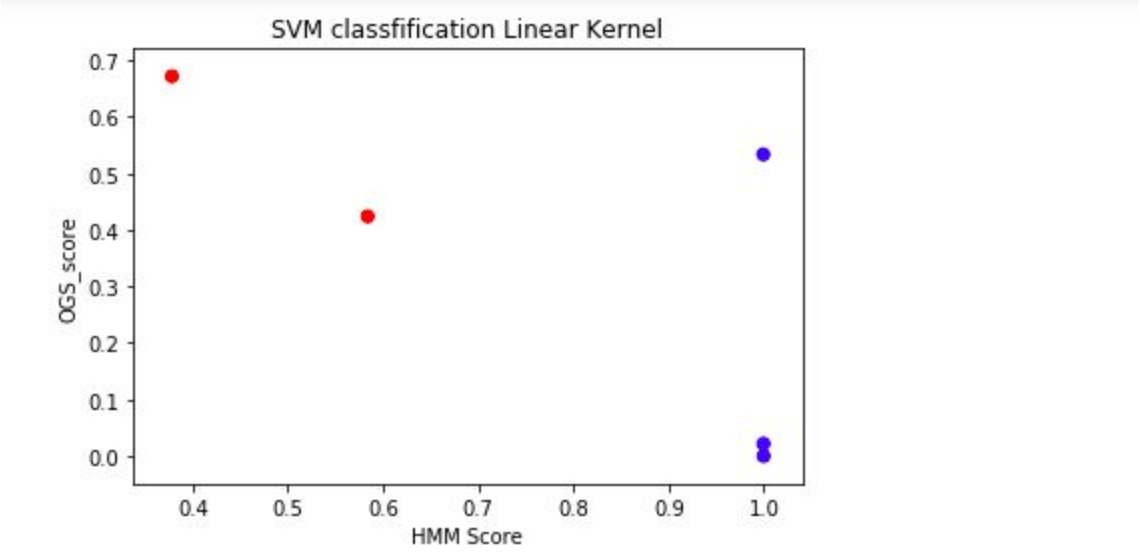


Fig : OGS Score versus HMM Score (Linear SVM)

When we give these two SVM Classifier , we see that a clear hyperplane distinction is observed when we use Linear Kernel whereas RBF and Sigmoid are still able to distinguish but not as clearly as a Linear Kernel.

SVM Accuracy Scores

Linear SVM - 100%

RBF SVM - 80%

Sigmoid SVM - 80%

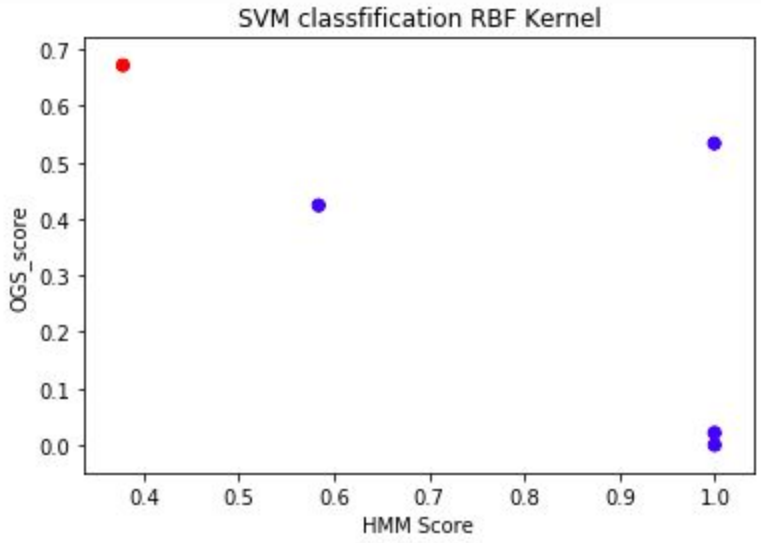


Fig : OGS Score versus HMM Score (RBF SVM)

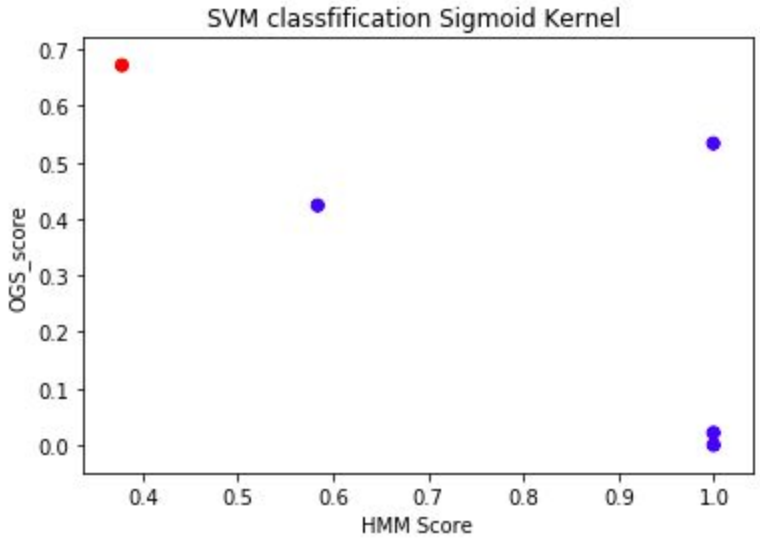


Fig : OGS Score versus HMM Score (Sigmoid SVM)

V. CONCLUSION

HMM scores remained steady with code permutation and steadily decreased as expected with more dead code insertion and increment in morph percentage.

We saw that OGS score was able to distinguish morphed copies from permuted copies and there is clear distinction between the scale and pattern of these scores.But in order to distinguish a lightly permuted file let's say (2 frames) from a heavily permuted file (20 frames) OGS score is not a good metric.

When normalized HMM and OGS scores were plotted for both benign and pirated files we saw they were not linearly separable.But when both these scores were passed as a feature to SVM , a linearly separable hyperplane was obtained.

REFERENCES

[1] Hardikkumar Rana , Mark Stamp :"Hunting for Pirated Software Using Metamorphic Analysis" (2014).Master's Projects. 345.

[2] Neha Runwal , Mark Stamp , Richard M. Low : "Opcode Graph Similarity and Metamorphic Detection"

[3] Borello, J., Me, L.: Code Obfuscation Techniques For Metamorphic Viruses. Journal in Computer Virology, 4(3) 30–40 (2008).

[4] Muhaya, F., Khan M. K., Xian, Y.: Polymorphic Malware Detection Using Hi- erarchical Hidden Markov Model. In Proceedings of the 2011 IEEE Ninth Inter- national Conference on Dependable, Autonomic and Secure Computing, 151–155 (2011).

[5] Raysman, R., Brown, P.: Copyright Infringement Of Computer Software And The Altai Test. New York Law Journal, 235(89) (2006).

[6] Stamp, M.: A Revealing Introduction To Hidden Markov Models. (2004). Re- trived from http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf