

MALWARE CLASSIFICATION:-

Using Malware Opcodes as Observations to train a Hidden Markov Model

Akshat Bansal, Jyoti Suri, Tirth Patel

San José State University

CS 185C: Advanced Practical Computing Topics

Professor Fabio D. Troia

18 May 2020

Abstract and Introduction

The aim of this project is to achieve multiclass classification of malware opcode samples according to their family. This is to be achieved by training one Hidden Markov Model per family and then combining the scores using Support Vector Machines. The training of the HMM will be done by analyzing the most frequently used assembly instructions in each sample file and then converting them into symbols. We effectively conducted a lot of tests in order to tweak our hyperparameters and showed results in the form of tables.

Experimental Setup

To set up the experiment, we began with the Hidden Markov Model code since an existing library had been used in the program and the next step was *pre-processing* the malicia dataset in the following way.

1. *Hidden Markov Model*: In the case of malware classification we after preprocessing the data, we have available families of malware samples each containing files with opcodes converted into character symbols. Using HMM is useful for this case because HMM is used when we have access to a series of observations that are probabilistically related to the underlying Markov model. We have a previously set number of malware families and data pertaining to the most probable opcodes of files that could be characteristic to that family. Here N is the number of states that is the number of families and M is the number of observation symbols or the different opcode characters that we have in the Map. This HMM model was tested previously using the Brown Corpus and classifying into vowels or consonants.

2. *Count Samples*: We start by counting the number of samples for each malware family.

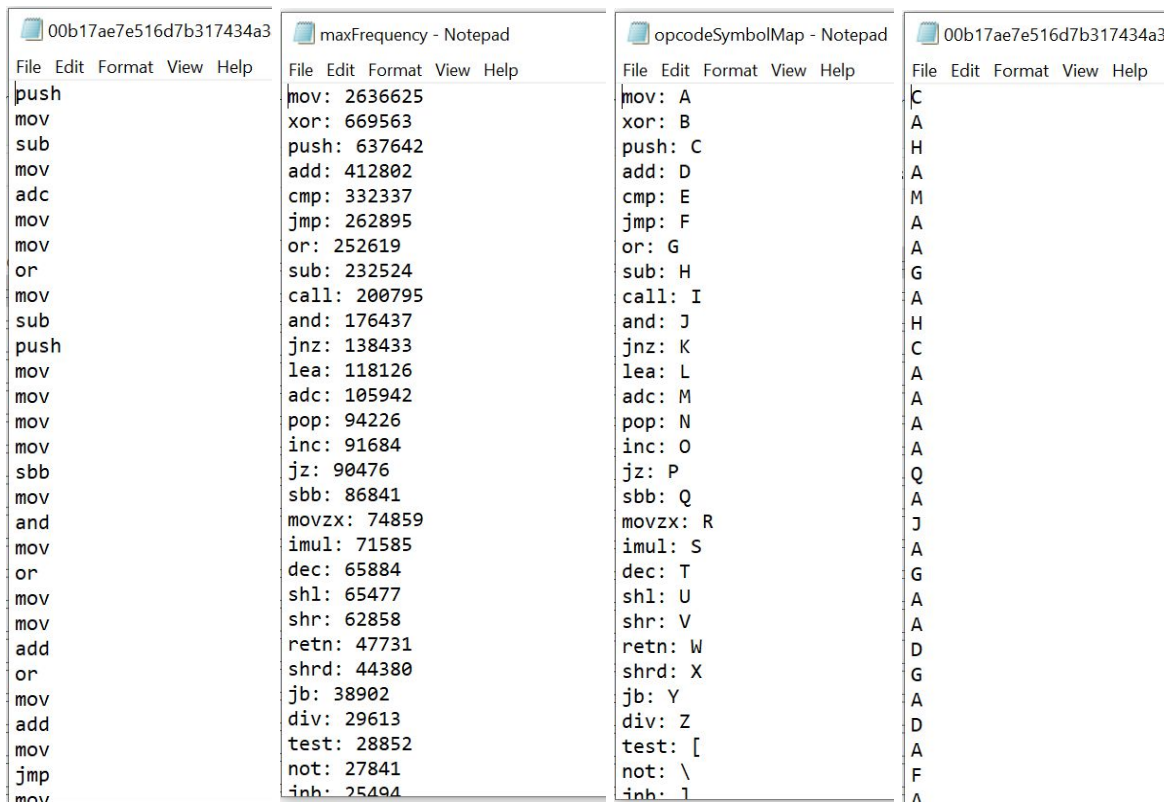
This makes it easy to determine which malware family has the most amount of files containing opcodes. For this we used a simple File method which counts the number of .txt files in a malware family and writes the number for each family in a file named counts.txt. The figure below shows the final result in counts.txt.

3. *Count Opcode per Family*: This method creates a dictionary of opcodes and their frequency for a family. For instance if we have samples from the 'zeroaccess' family it will create a new file with counts for each and every opcode mentioned in the file and its count. After this method runs we have a file called maxFrequency in each and every family that shows the opcode count.
4. *Count N-grams*: We began with two separate methods: one for grouping single opcodes according to frequency and the other for grouping the opcodes based on the N-grams. This method creates a group of a certain number of opcodes and counts their frequency. The rest of the method works in the same way as the previous method, by listing out the most used N-grams and assigning each of them a symbol, with a default symbol for the rest of the N-grams.

<pre> maxFrequency - Notepad File Edit Format View Help movmovmovmov: 531930 pushpushpushpush: 224329 pushpushpushcall: 167022 pushcallpushcall: 99014 pushpushcalladd: 88757 pushpushcallmov: 78798 callpushcallpush: 67524 popretnpushmov: 65039 pushcallpopmov: 59743 movpushpushpush: 59433 cmpjzpushcall: 53851 movmovcmpjz: 48827 poppopleaveretn: 48149 pushpushcalltest: 47602 pushcallpoppop: 47487 cmpjzcmpjz: 46310 </pre>	<pre> symbolOpcodeMap - Notepad File Edit Format View Help A: movmovmovmov B: pushpushpushpush C: pushpushpushcall D: pushcallpushcall E: pushpushcalladd F: pushpushcallmov G: callpushcallpush H: popretnpushmov I: pushcallpopmov J: movpushpushpush K: cmpjzpushcall L: movmovcmpjz M: poppopleaveretn N: pushpushcalltest O: pushcallpoppop P: cmpjzcmpjz </pre>	<pre> opcodeSymbolMap - Notepad File Edit Format View Help movmovmovmov: A pushpushpushpush: B pushpushpushcall: C pushcallpushcall: D pushpushcalladd: E pushpushcallmov: F callpushcallpush: G popretnpushmov: H pushcallpopmov: I movpushpushpush: J cmpjzpushcall: K movmovcmpjz: L poppopleaveretn: M pushpushcalltest: N pushcallpoppop: O cmpjzcmpjz: P </pre>
---	---	---

5. The figure below shows

- The original file in the zeroaccess family.
- The maxFrequency file in the zeroaccess family directory.
- The opcodeSymbolMap file in the zeroaccess family directory.
- The file in the a. Converted using the opcodeSymbolMap.
























After we have the most frequent opcodes, we then sort them in decreasing order with their frequency. Then we create an opcode to symbol mapping and also a file for our convenience.

This map connects a variable number of most frequently used opcodes in order to characters and the 30th code is default and all the other opcodes are given this value. The method then created a new folder in each family called 'Processed'. This file now contains all the sample files of that particular family with the opcode symbols in place of the assembly instructions.

Methodology and Dataset: preprocess, ngram, code method stubs

1. *Dataset:* We began by downloading the Malicia Dataset and looking at the type and size of data. The data were *unbalanced* and had varying numbers of samples per family and a different number of opcodes per file. We used only limited families according to the number of sample files available.

 dprn	18-05-2020 20:01	File folder
 fakeavrena	18-05-2020 20:01	File folder
 fakeav-rena	18-05-2020 20:01	File folder
 fakeavwebprotection	18-05-2020 20:01	File folder
 fakeav-webprotection	18-05-2020 20:01	File folder
 harebot	18-05-2020 20:01	File folder
 ramnit	18-05-2020 20:01	File folder
 ransomNoaouy	18-05-2020 20:01	File folder
 ruskill	18-05-2020 20:01	File folder
 securityshield	18-05-2020 20:01	File folder
 smarthdd	18-05-2020 20:01	File folder
 spyeyeep	18-05-2020 20:01	File folder
 spyeye-ep	18-05-2020 20:01	File folder
 Type	18-05-2020 20:01	File folder
 ufasoftbitcoin	18-05-2020 20:01	File folder
 ufasoft-bitcoin	18-05-2020 20:01	File folder
 unknown	18-05-2020 20:01	File folder
 WinRescue	18-05-2020 20:01	File folder
 winwebsec	18-05-2020 20:01	File folder
 zbot	18-05-2020 20:02	File folder
 zeroaccess	18-05-2020 20:02	File folder

2. *Hidden Markov Model:* The Markov Model Code effectively uses a multitude of functions line backward as well as forward pass in order to return its score. The first step was creating our own code for the Hidden Markov Model instead of using a standard library.

```

6 public class Model {
7*   public static void main(String[] args) {}
22
23*   public Model(int N, int M) {}
84*   public double score(int[] test) {}
93*   public void alphaPass(int[] observation) {}
128*   public void betaPass() {}
147*   public void gammaDigammaPass() {}
163*   public void reestimate() {}
198*   public void logProbCalc() {}
205*   public void decision() {}
213*   public void recompute() {}
223*   public int[] getObservationSequence() {}
274*   public void runTests(String familyPath) {}
370*   private void printEverything() {}
389*   public int getMaxIterations() {}

```

3. *Preprocessing*: The end goal of the preprocessor class was to order the data according to the hyperparameters provided to it like n-grams and number of instructions to be considered for observation (i.e. T).

```

4 public class Preprocessor {
5   private LinkedHashMap<String, Character> opcodeSymbolMap;
6   private LinkedHashMap<Character, String> symbolOpcodeMap;
7
8*   public Preprocessor() {
9     opcodeSymbolMap = new LinkedHashMap<>();
10    symbolOpcodeMap = new LinkedHashMap<>();
11  }
12
13*   public LinkedHashMap<String, Character> getOpcodeSymbolMap() {
14     return opcodeSymbolMap;
15  }
16
17*   public void countSamplePerFamily() {}
38
39*   public void countNGramsPerFamily(int n) {}
194
195*   public void countOpcodePerFamily() {}
330
331*   public void removeFrequencyFiles() {}
359
360*   public void removeProcessedFiles() {}
382

```

4. *Processed File Usage*: After Preprocessing, each and every family has the following
 - a. Processed Folder
 - i. Processed Filed according to n-gram and opcode symbol conversion
 - ii. 80% samples for training and 20% for testing.

- b. Original Sample Files
 - c. Frequency.txt
 - d. maxFrequency.txt
 - e. opcodeSymbolMap
5. *Result Interpretation:* Since, we faced the common problem of the alphas in the T-1 row summing to 1. We stepped through our code to understand how they behaved. We then realized that these alpha values (2 in our case because the number of states were 2) varied with the values in the PI matrix. Due to the random starting weights for the A,B, and PI matrices, the values in the PI matrix sometimes interchanged places. Sometimes, column 1 converged to 1 and sometimes column 2 converged to 1. But, we observed that the results we get from scoring the files differed greatly in the values as if the file given was from the original family over which the model was trained, the score was either very high or either very low. And if the file belonged to a new family, the score was opposite to that of the original file. Example - if the score for a winwebsec file came out to be very low for the winwebsec model then the score for files belonging to other families was very high. And if the score for the winwebsec was very high then the score for files from other families was very low. So we decided that we can extract the maximum value for the files that belonged to the original family and the minimum for the files that belonged to other families.

Experimental Results

We trained the model on seven malware families which contained more than 50 files of opcodes. These malware families and their opcode file counts are as follows. cridex(77), harebot(56), securityshield(61), smarthdd(71), winwebsec(4363), zbot(2139), zeroaccess(1310).

When the length of the observation sequence was increased it took longer to process. We experimented with 50 iterations, which yielded too low of an accuracy, 300 was too high, hence we ended up with 150 iterations for the experiments. Some of the accuracies are low due to the limited number of samples in each family.

In the following table, each malware family is trained and tested against each other's models. Most of the result cells in the table contain accuracy and some of them also contain the pi matrix and the score.

	cridex (77)	harebot (56)	securitysh ield (61)	smarthd d (71)	winwebse c (4363)	zbot (2139)	zeroacces s (1310)
cridex (N=2, M=30, I=150, 4-gram)	0.5146 [5.6E-97, 1.00] Score: -94006.6	0.4621	0.4752 [4.9E-103, 1.00] Score: -94012.42	5.50E-33 [0.9,0.0] Score: -79646.6	8.36E-46	2.6E-17	4.10E-44
harebot (N=2, M=30, I=150, 4-gram)	0.4949 [1.7E-32, 1.00] Score: -80933.8	0.5102 [4.8E-86, 0.99] Score: -80806.2	7.97E-165 [0.00,1.00] Score: -63287.62	0.4842 [0.9849,0. 0151] Score: -81413.8	6.9E-186	5.2E-13	3.4E-56
securityshield (N=2, M=30, I=150, 4-gram)	0.0026 [1.00,1.0 44E-207] Score: -70208.9	0.4567 [0.99,6.0 31E-14] Score: -76881.8	0.9979 [9.8E-210, 0.99] Score: -67265.74	0.4842 [0.99,3.24 8E-8] Score: -77125.7	0.0036	0.0018	0.0021

smarthdd (N=2, M=30, I=150, 4-gram)	0.4582 [0.0,1.0] Score: -95588.6	0.4317 [1.1E-30, 1.00] Score: -99609.8	0.4920 [0.99,1.11 E-34] Score: -99712.22	0.9827 [1.0,7.43 E-89] Score: -87877.5	0.4448	1.355 E-212	0.0155
winwebsec (N=2, M=30, I=150, 1-gram)	0.3525	0.4981	0.4364	0.3869	0.9972	0.4126	0.0954
zbot (N=2, M=30, I=150, 1-gram)	0.4897	4.32E-44	0.0420	0.7654	2.7E-36	0.5271	9.8E-322
zeroaccess (N=2, M=30, I=150, 4-gram)	6.15E-9	2.39E-14	5.78E-7	5.33E-4	1.07E-4	0.0011	0.9998

Conclusions and Future Work

The experiments conducted led to a series of results some of which showed success in detecting the malware family the testing sample belonged to. Tuning of hyper parameters helped us gain better results and faster outputs in many cases. We mainly worked on tuning the following: N, M, pi, Iteration Count(I), and n-grams.

In the future this project has a lot of work to do. Using the scores we have for Stacking using Support Vector Machines is the next major step in order to compile the data gained from all individual HMM models. To improve accuracy we also plan to use techniques such as Bagging, Boosting, and n-Fold Cross Validation.