Classifying Malware into families Based on File Content

Patrick Rand and Reynier Ortiz
School of Computing and Information Sciences
Florida International University
Miami, Florida, USA

Abstract– We present a mechanism to classify a malware file into one of nine families: Rammit, Gatak, Tracur, Vundo, Simda, Kelihos_ver1, Obfuscator.ACY, Lollipop and Kelihos_ver3. The system uses a large training set of binary files, where N-grams are extracted using the kfNgram tool. These N-grams are merged to form a master list and later reduced by selecting the top 500 with the highest information gain *(IG)*. These 500 N-grams are the boolean attributes to determine if they are present or not in each of the malware executables. The training set, using the selected attributes, was transformed into an .arff file required by Weka [Witten and Frank 2005] to run several classifiers, i.e. nave bayes, decision trees and support vector machines (SVM). We then compared the different algorithms using some common measures: accuracy, error rate, true and false positive error rates.

Categories and Subject Descriptors: H.3.3 [**Information systems**]: Information retrieval—*Information Search and Retrieval*

Additional Key Words and Phrases: malware, classification, n-grams

1. INTRODUCTION

Some introduction *blahg blah* etc.

2. PROBLEM DEFINITION AND METHODS

2.1 Task Definition

Blah blah [WebGL 2014].

2.2 Algorithms and Methods

Blah blah [WebGL 2014].

2.2.1 *System Architecture.* As shown in figure Fig 1, the architecture of the system is rather simple. The main tasks were the preprocessing of the data which involved extracting the N-grams and transform the training set into a format that Weka [Witten and Frank 2005] accepts as input. And then apply several learning methods.

2.2.2 *Data Preprocessing.* The preprocessing part was a process that required a considerable effort because of the size of the training set, 10,868 files. The first step was to extract the hexadecimal N-grams from the binary files using the kfNgram tool [Fletcher 2012], we used n=4.

It was then necessary to merge the resulting sorted N-gram files to effectively calculate the *information gain (IG)* of each N-gram. Because these files were extremely long, we could not use the merge functionality built in the kfNgram tool, and we have to halt the application after running for three days. To optimize this process we merged the files maintaining a min-heap (or *PriorityQueue* in Java) in memory with the first distinct N-gram from each file, as shown in Fig 2. We were also able to keep 10,878 Java scanners in memory to speed up I/O. Then, to merging process would consist in each iteration to perform a `pop()` operation from the `PriorityQueue`, append the element to merged file, and insert the next N-gram from the same file of the element that was popped to the `PriorityQueue`. This process is represented in Fig 3.

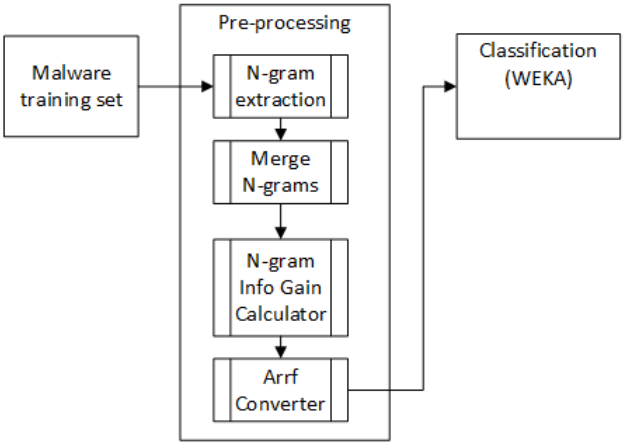


Fig. 1. System architecture.

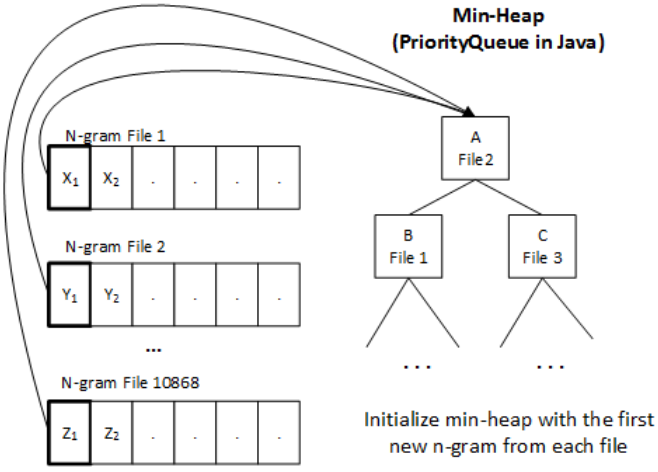


Fig. 2. Initializing PriorityQueue to merge N-grams.

Once merging was complete, we obtained a total of N-grams was 255,942,370, therefore we had to reduce them by calculating the *information gain (IG)* of each N-gram and selecting the top 500 (Fig. 4), the same approach used in [Kolter and Maloof 2006].

2.2.3 *Classification algorithms.* Blah blah

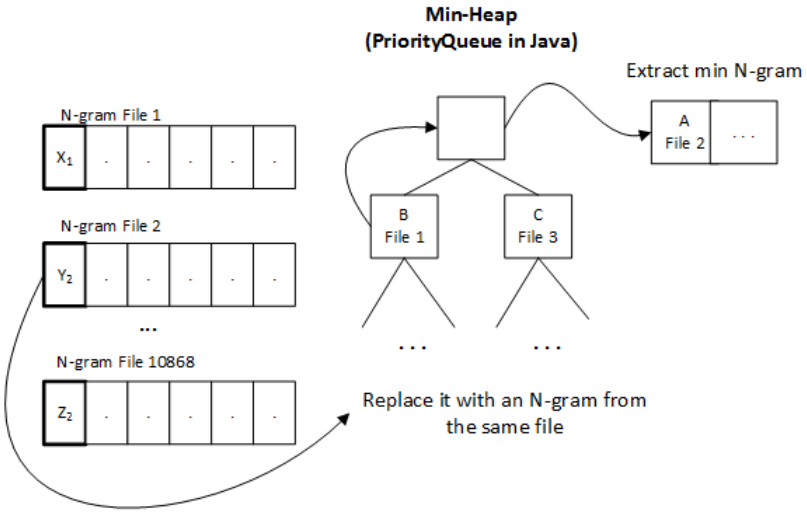


Fig. 3. Process of merging N-grams. The minimum element is extracted and replaced with another N-gram from the same file

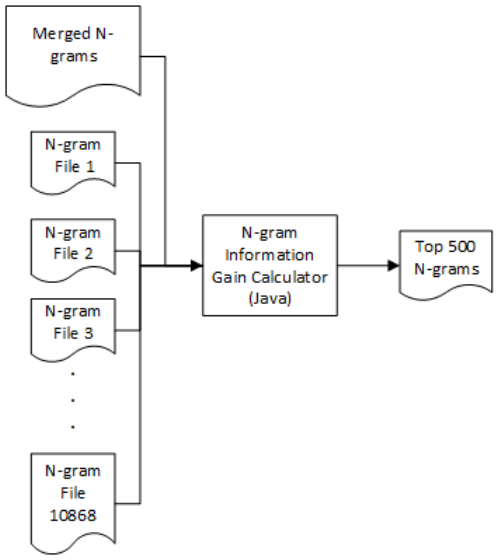


Fig. 4. Process of calculating the information gain of each N-gram.

3. EXPERIMENTAL (AND/OR THEORETICAL) EVALUATION

3.1 Methodology

Blah blah [WebGL 2014].

3.2 Results

Blah blah [WebGL 2014].

3.3 Discussion

Blah blah [WebGL 2014].

4. RELATED WORK

Malware detection and classification is a problem being addressed through several angles. In [Christodorescu et al. 2005], the goal is to detect if a program exhibits a specified malicious behavior by determining if a set of templates of sequence of instructions are present in the executable files. This approach requires to have knowledge on semantics of each of the malware families. Although the results are promising, there are several reasons we could not follow this approach, for example, as inferred by the title, it would be necessary to create a set of sequence of instructions for each of the nine families we need to classify the malwares into. This is infeasible due to the length of the course, but also it is outside of the scope of the course. As exhibited in the results, this strategy is resilient to obfuscation and showed improvements when compared to McAfee VirusScan.

Several malware classification algorithms are based on n-grams extracted from the executable files. In [Pekta et al. 2011], instead of byte sequences, the n-grams extracted are formed by machine codes. After obtaining the n-grams from the malwares, a centroid for each family is created by selecting the most frequent n-grams. Then, the strategy to classify a malware into one of the families, is to determine the centroid which the malware is more similar to by counting the number of matching n-grams. When considering this approach, one of the possible limitations analyzed was that selecting the most frequent n-gram could implicate choosing an n-gram irrelevant to the malware family.

We also explored sequential pattern extraction of n-grams. The proposed methodology in [Sornil and Liangboonprakong 2013] outlines a procedure to use the n-grams patterns to classify the malware by family. The kfNgram tool [Fletcher 2012] was used to extract the n-grams from the disassembled files with n=1, n=2, n=3 and n=4, obtaining the best results with n=4. In [Kolter and Maloof 2006] the accuracy achieved was higher with n=4, therefore we skipped this evaluation and use only n-grams with n=4. The sequential pattern extraction technique in [Zhong et al. 2012] was used to generate frequently occurred sequences of n-grams to represent the data [Sornil and Liangboonprakong 2013]. Then the patterns signif-

icance was calculated using the term frequency-inverse document frequency (TF-IDF) where the term refers to the n-gram pattern and a document to the malware file. Since the number of patterns was too large, the sequential floating forward selection (SFFS) procedure was applied to reduce the number of features, in this case, n-gram sequential patterns. With all the features extracted, three classification algorithms were used, C4.5, multilayer perceptron and support vector machine. The training set was randomly split into two partitions using 80% for training and 20% for testing achieving a 96.64% of accuracy [Sornil and Liangboonprakong 2013]. Because of the duration of the course and the complexity of sequential pattern extraction, we were not able to experiment with this approach.

Following Occam's razor, suggesting that the simplest hypothesis is the best, we applied an approach similar to the one described in [Kolter and Maloof 2006]. The n-grams extracted from the executable files represented boolean features, present (i.e., 1) or absent (i.e., 0). Since the n-grams list was too large, it was necessary to select the most relevant attributes (i.e., n-grams) by computing the *information gain (IG)* described in [Yang and Pederson 1997] for each, also called *average mutual information*. Through pilot studies, it was determined to use the top 500 n-grams, and then applied classifiers implemented in the Wakaito Environment for Knowledge Acquisition (WEKA) [Witten and Frank 2005]: IBk, Nave Bayes, SVM, and J48 (decision tree), and also *boosted* the last three of these learners [Kolter and Maloof 2006]. The results indicated 98% the highest accuracy using boosted decision trees.

```
string GetVoices();

string SpeakText(string text, string voice,
                 string audioFormat);

string SpeakSSML(string ssml, string voice,
                 string audioFormat);
```

The GetVoices() operation returns a list of the SAPI voices installed in the server, the information of each voice contains the name, gender, age and culture. Both SpeakText() and SpeakSSML() operations returns a structure with:

—The audio stream in wav or mp3 format in a base64 string
—The sequence of visemes, where each viseme contains the viseme number, the audio position in milliseconds, and the duration in milliseconds

The difference between SpeakText() and SpeakSSML() is that the first one only accepts a plain input string and synthesizes using the default options, whereas SpeakSSML() accepts a string in SSML. Voice manipulation is specified in SSML by using the `<prosody>` elements and specifying parameters such as: volume, rate and pitch [SSML 2014]. The information returned by these operations are sufficient for the lip-synchronization algorithm to generate the sequence of viseme transitions aligned with the audio stream. Fig 0**??** shows the package diagram for the HapGLService subsystem.

HapGL requires to instantiate a `HapGL()` which expects the URL of the TTS, the mesh of the 3D character and the mesh of the hair, for example:

```
// Using ThreeJS, load character and
// hair into mesh1 and mesh2 respectively

var hapgl = HapGL.init({
        ttsUrl: 'http://localhost:88/',
        character: mesh1,
        hair: mesh2
});

// Example to activate an Action Unit
hapgl.setAU('AU1', 100);
```

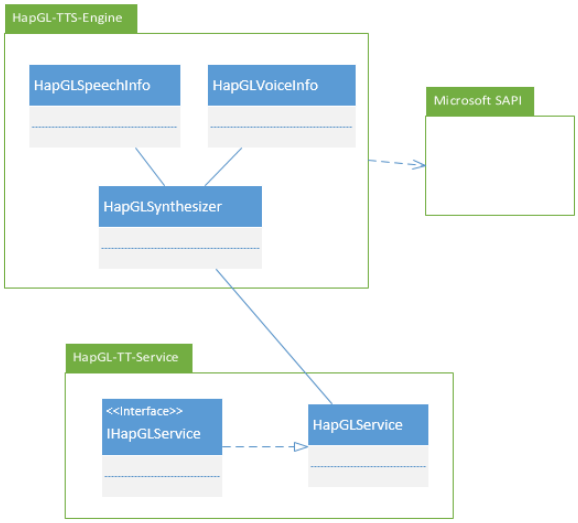


Fig. 5. HapGLService package diagram.

Fig. 6. Result of setting AU1 to 100 intensity.

Generating emotions in HapGL was done also in the same manner as in HapFACS. Emotion FACS (EmFACS) introduces a mapping of subsets of action units to universal emotion identified by Ekman [P. Ekman and Freisen 1983] namely fear, anger, surprise, disgust, sadness, and happines. The emotions implemented in HapGL were:

—Happines, combining AU6, AU12, and AU25
—Sadness, combining AU1, AU4, and AU15

—Surprise, combining AU1, AU2, AU26, and AU5
—Anger, combining AU4, AU5, AU7, AU23, and AU24
—Disgust, combining AU9, AU15, and AU16

The current version of HapGL provides three methods related to the speaking portion:

—`getVoices(function getVoicesCallback)`
—`speak(String text, String voice)`
—`speakssml(String ssml, String voice)`

`getVoices(function getVoicesCallback)` returns a JSON object with an array of voices and will immediately call the `getVoicesCallback` function with the result. Since the HapGLService uses Microsoft Speech API (SAPI), the voices are SAPI-compatible installed on the web-server. More sophisticated voices can be purchased and installed separately. The following is an example of the result of calling `getVoices`:

```
hapgl.getVoices(function(output) {...});

// Example of voices returned
{
   voices : [{
      id : "MS-Anna-1033-20-DSK",
      age : "Adult",
      name : "Microsoft Anna",
      gender : "Female",
      culture : "en-US"
   }]
}
```

`speak` and `speakssml` are similar, the only difference is that `speak` will only accept a plain string, whereas `speakssml` can accept a string in SSML format as input [SSML 2014]. The `voice` parameter corresponds to the *name* attribute of the voices returned by `getVoices`, if no `voice` is passed in then the HapGLService will synthesize using the default voice in the TTS Server. The *ssml* string argument for the `speakssml()` function should be a well-formed SSML Version 1.0 [SSML 2014]. SSML allows to manipulate the voices by modifying parameters such as: *volume*, *rate*, and *pitch* in the `<prosody>` elements. Several `<prosody>` elements can be combined to produce the desired pronunciation of sentences.

The output of `speak(..)` and `speakssml(..)` contains the necessary information to render the sequence of viseme transitions synchronized with the audio stream to produce a realistic talking virtual human. The following example shows the output when speaking the word "Hello".

```
hapgl.speak(``Hello");

// The output of speaking ``Hello"
{
    audioFormat: ``data:audio\/wav;base64",
    audioStream: ``...",
    visemes: [{
       number : 0,    // silence
       audioPosition : 0.0,
       duration : 3.0,
       emphasis : 0
    }
}
```

The sequence of visemes is already sorted in the correct order to be animated. All this information is sufficient for the lip-synchronization algorithm to render each viseme at the correct time. The viseme transitions are done smoothly, otherwise, just displaying the viseme in its maximum intensity would create an undesired illusion. Each viseme transition takes a pair of visemes V_0 and V_1, where V_0 is the starting viseme and V_1 is the ending viseme. To do the transformation $V_0 \rightarrow V_1$ we consider the duration d_0 of V_0. In d_0 time, V_0 "fades out" and V_1 "fades in". By "fade out" we mean interpolating from V_0 by setting the value of the corresponding morph gradually from 1 to 0 in d_0 time. Conversely, "fade in" means interpolating to V_1 by setting the value from 0 to 1. Each viseme maps to a corresponding phoneme, we use the mapping provided by the Microsoft Speech API as seen in Table 0**??**.

Table I. Viseme to phonemes mapping in Microsoft Speech API

Viseme	Phoneme(s)	Viseme	Phoneme(s)
0	silence	11	ay
1	ae, ax, ah	12	h
2	aa	13	r
3	ao	14	l
4	ey, eh, uh	15	s, z
5	er	16	sh, ch, jh, zh
6	y, iy, ih, ix	17	th, dh
7	w, uw	18	f, v
8	ow	19	d, t, n
9	aw	20	k, g, ng
10	oy	21	p, b, m

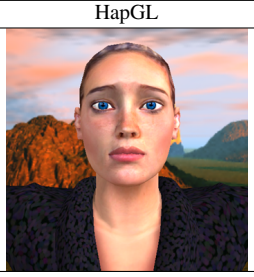
Since not all the phonemes have its corresponding Haptek morph register, we choose the most similar morph. We used the following viseme to morph mapping in HapGL:

```
var visemeMorphMapping = {
  '0': {name: 'neutral'}, '11': {name: 'aa'},
  '1': { name: 'aa' }, '12': { name: 'ih' },
  '2': { name: 'aa' }, '13': { name: 'n' },
  '3': { name: 'aa' }, '14': { name: 'n' },
  '4': { name: 'ey' }, '15': { name: 's' },
  '5': { name: 'er' }, '16': { name: 'ch' },
  '6': { name: 'ih' }, '17': { name: 'th' },
  '7': { name: 'uw' }, '18': { name: 'f' },
  '8': { name: 'ow' }, '19': { name: 'd' },
  '9': { name: 'aa' }, '20': { name: 'g' },
 '10': { name: 'ow' }, '21': { name: 'm' }
};
```

Table II. AUs with recognition rate of less than 100%. Taken from HapFACS [Amini and Lisetti 2013].

AU	Recognition Rate	AU	Recognition Rate
10	66.67%	16	33.33%
11	66.67%	20	66.67%
12	66.67%	23	33.33%
13	66.67%	25	33.33%
14	66.67%	28	33.33%

Table III. AUs comparisson between HapGL and HapFACS.

AU	HapGL	HapFACS
AU1		

5. FUTURE WORK

Malware classification is a topic that is always evolving because viruses' authors are constantly developing techniques to avoid been detected, for example by obfuscating the code or by designing polymorphic behaviours. The proposed approach is agnostic of these techniques, therefore, we assume intuitively that a set of the extracted N-grams are product of this ingenious methods. Mutation engines are capable of generating millions of variations of the same virus, therefore, to overcome this problem, it would be necessary to have knowledge of the malware behaviors to develop heuristics or detect specific sequence of instructions to find in the malicious files.

For time reasons, we could not research on the possibilities of removing the irrelevant computations in the malware files created to evade anti-viruses. This would help, if using sequential N-gram patterns extraction, to detect patterns that are truly correlated to the malware family. These patterns could, somewhat, describe the common semantics present in each class of malwares.

6. CONCLUSION

Nevertheless, HapGL is still far from being used in production systems as it lacks of other necessary functionalities which could be part of future works. To name some, and not intended to be a comprehensive list, we suggest the following:

—*Detailed Evaluation*, due to time constraints, a thorough evaluation could not be performed. We suggest to measure the *believability* of the system by surveying a random sample of users, preferably greater than 20. Although this is a subjective measure, a user study is still a good indication about the quality of the system.

REFERENCES

Reza Amini and Christine Lisetti. 2013. HapFACS: an Open Source API/-Software to Generate FACS-Based Expressions for ECAs Animation and for Corpus Generation. (2013). http://ascl.cis.fiu.edu/hapfacs.html

Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. 2005. Semantics-Aware Malware Detection. (2005), 32–46.

William H. Fletcher. 2012. Data mining: Practical machine learning tools and techniques. (2012). http://kwicfinder.com/kfNgram/kfNgramHelp.html

J. Zico Kolter and Marcus A. Maloof. 2006. Learning to Detect and Classify Malicious Executables in the Wild. 7, Article 19 (2006), 2721–2744.

R. W. Levenson P. Ekman and W. V. Freisen. 1983. Autonomic Nervous System Activity Distinguishes among Emotions. 221 (1983).

Abdurrahman Pekta, Mehmet Eri, and Tankut Acarman. 2011. Proposal of n-gram Based Algorithm for Malware Classification. (2011).

Ohm Sornil and Chatchai Liangboonprakong. 2013. Malware Classification Using N-grams Sequential Pattern Features. (2013).

SSML. 2014. Speech Synthesis Markup Language (SSML) Version 1.0. (Nov. 2014). http://http://www.w3.org/TR/speech-synthesis/

WebGL. 2014. Getting Started - WebGL Public Wiki. (2014). https://www.khronos.org/webgl/wiki/Getting_Started

I. H. Witten and E. Frank. 2005. Data mining: Practical machine learning tools and techniques. (2005). http://www.cs.waikato.ac.nz/ml/weka/index.html

Y. Yang and J. O. Pederson. 1997. A comparative study on feature selection in text categorization. (1997), 412–420.

N. Zhong, Y. Li, and S. T. Wu. 2012. Effective Pattern Discovery for Text Mining. 24, Issue 1 (2012), 30–44.