Microsoft Malware Classification Challenge: 6th place solution

21.04.2015, Oleksandr Lysenko

Intro

According to papers available in the Internet, there are at least three main approaches:

- classification based on opcode n-grams
- classification based on bytecode n-grams
- classification based on malware images

Papers used for inspiration

- Metamorphic Virus Detection in Portable Executables Using Opcodes Statistical Feature, Babak Bashari Rad, Maslin Masrom.
- Detecting unknown malicious code by applying classification techniques on OpCode patterns. Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev and Yuval Elovic
- Malware Images: Visualization and Automatic Classification. L. Nataraj, S. Karthikeyan, G. Jacob, B. S. Manjunath.

Approaching the problem

1. Software fingerprints
After basic observation of sample .asm files, it became apparent that each file contains some sort of unique information about frameworks and libraries used to write the code. The first key was “std::exception” in 0A32eTdBKayjCWhZqDOQ.asm which was absent in 0ACDbR5M3ZhBJajygTuf.asm. That means that in first case developers were using C++ Standard Library (std). Some other parts of std, that were found later:

- std::basic_string
- std::allocator
- std::char_traits

This were found in classes 1, 2 and 9.

From other side, in some other classes were found traces of WinAPI level exceptions. For example, “UnhandledExceptionFilter” in class 5, 6, 8 and 9. There is some overlapping between C++ exceptions and WinAPI exceptions in case of 9, no exception handling in 3 and 4, and different types exceptions in 1 and 2 versus 5, 6 and 8. This was the idea to start with: finding fingerprint of software used to write the code.

My first feature set contained list of manually generate list of keywords describing the software: “std::”, “exception”, “error”, “dword”, “lpvoid”, “memcpy”, “allocate”, “hmodule”, “mutex”, “size_t”, “bool”, “hwnd” etc.

2. Byte ngrams
It was the easiest model to implement and it was well discussed in the Internet and on Kaggle forum in particular. As it mentioned in paper “Detecting unknown malicious code by applying classification techniques on OpCode patterns”, in average it works worse than opcodes ngrams. But by combining it with software fingerprints I achieved the first significant result: log_loss around 0.014.

3. Malware images
For me it was dead end. I created images based on paper “Malware Images: Visualization and Automatic Classification” and then extracted features using GIST image descriptor, but it didn’t work even close as good and combination of the first two feature sets.

4. PE sections
Going deeper with “software fingerprints” I noticed how different file structures from class to class, especially, how different class 5 from other - it was definitely the result of some kind of obfuscation. In most cases there is a small list of common sections: “:text”, “:data”, “:idata”, “:rdata” and some others. But in class 5 they were renamed using different techniques.

The feature set consisted of 445 features and according to a classifier feature importance metric, many of those feature are the most important, e.g. “kyqic:” (feature number 1 in the list), “.w:”, “.tdat:”, “.aspack4:”, “jfsx_:”, etc.

5. DLL imports
This feature set consisted of all dlls from section “:idata”. This didn’t worked out at all and was abandoned due to poor performance.

6. Opcode ngrams
ngrams were extracted from code section of PE files - normally “:text”. To avoid obfuscation issues, this section name were detected automatically based on section header using key phrase “segment type: pure code”. Due to enormous number of features for ngrams with n>1, it was decided to use top N features per class by frequency (number of feature occurrences divided to number of class occurrences). Starting from n=3, each feature group were improving classification performance by log_loss ~ 0.01.

7. Top N unique opcodes per class
The idea is very close to previous, but top N unique features per class were used. This didn’t improve performance at all.

8. Artificial features

There were 3 artificial feature added to each feature set: standard deviation, row sum and number of non null values per row.

Final solution

Around 28000 features, grouped into 9 feature sets:

- hex (bytes) unigrams
- top 1000 hex (bytes) bigrams per class
- asm file sections (":text", ":idata", ":rdata", etc)
- misc feature extracted from asm files ("std::", "exception", "memcpy", "function chunk", etc)
- opcodes unigrams
- top 1000 opcodes bigrams per class by frequency
- top 1000 opcodes 3-grams per class by frequency
- top 1000 opcodes 4-grams per class by frequency
- top 1000 opcodes 5-grams per class by frequency

The biggest challenges were:

- build PE format parser to extract opcode ngrams only from code sections and properly identify those sections (in most cases it called ":text" but looks like in malware class 5 all sections were obfuscated and their names changed randomly, for example ".FOAu:").
- find optimal value of N parameter.

Classification algorithm used: Gradient Boosting with max_depth=3 (XGBoost: eXtreme Gradient Boosting), 6-fold cross validation.

Results achieved

By combining all successful feature sets, I achieved minimum log_loss = 0.0053. No feature scaling, feature selection, dimensionality reduction or multi-models with ensembling were done.