University "Alexandru Ioan Cuza" of Iasi, Romania
Faculty of Computer Science

Meta-heuristics for anti-malware systems

PhD Candidate

Gavrilut Dragos Teodor

Supervisor

Prof. PhD Henri Luchian

- Abstract –

Introduction

This thesis is concerned with malicious software detection using machine learning techniques. With the exponential grow of malicious software during the last five years that reached 80 millions of different malware this year and the more sophisticated methods this kind of software now have to protect themselves from different forms of detection, newer methods of detection were a necessary thing in anti-malware industry.

My thesis discusses the necessary steps for creating such a malware detection system. The design of such a system is discussed from two separate perspectives (an academic one and a practical one). The practical aspects of a malware detection system came with a serious of constraints that this system should meet:

- A very low false positive rate (0 if possible). This is maybe the most important one. While from a strictly academic point of view a percentage of 0.01% false alarms is something acceptable, if this percentage

means more than 10 or 20 files it becomes unacceptable for practical usage.

- Creating a model or the signature database should be done in an acceptable time. This is necessary because this system should be able to adjust to the rapid changes in malware evolution. This involves using distributed system or different code optimizations.
- The detection part of this system should be feasible for different architectures. This part will run on a client computer so it has to be fast and it should require very few memory resources.
- Ensures that the model created will provide a sufficient detection in time (e.g. a model created now will still detect new malware in a couple of months).

The following steps should be made to create such a detection system:

- Creating a large database of both malware and clean files. As the databases grow, the number of "noises" (incorrectly classified files in the databases) will grow as well. New method for detecting these noises should be developed.

- Create a set of features that will be used with different machine learning algorithms.
- Analyzing different machine learning algorithms and determine the best one that can be used in such a system.
- Adapt and modify the selected algorithm to comply with the previously described practical constraints. This will mean to adjust or change the algorithm for new type of malware.

Some of these steps should be consider as a continuous process (e.g. creating a set of features or collecting malware and clean files). The only step in this process that was limited to a specific period of time was the analysis of the different machine learning algorithms.

Collections

In case of a clean file, the process is usually simple. The basic idea is to create a large set of files that belong to the most popular applications. So, all one needs to do is to crawl most of the biggest download sites and download every application that they have. It is

also important to have all of the OS system files and OS related (drivers, service packs) for every language. There are 2 methods usually used to collect these files:

- Having a sort of crawling system. This will search for new applications with a download site and download them. This form of collecting clean information can be extended to different FTP locations, P2P sharing, torrents and other forms of sharing.
- Using a virtual environment. This is necessary in 2 cases:
  - In case of some operating system or other application, there is no direct method to upload a new version.
  - In case of net-installers

This kind of system managed to download almost 39.000 new kits every day. This means almost 15 millions of kits every year and almost 70 millions of kits for the last 5 years. For malware collection the following methods were used:

- Honey-pots to collect different malware files.

- Use malware-URL to download malware.
- Use different external sources (web-sites that have collections of malware).
- Analyzes different feeds of spam-emails
- Exchange with different organizations that have malware collections.
- In case of file-infectors, more samples can be obtained by executing an infected file in a virtual environment, and wait to infect other files.

Feature creation

Feature creation means extracting a set of characteristics for every file. However due to the large variety of file types I've focused on executable files (as this is the most common format used by malware) and script related files. These features can be a static attribute (something obtained by static analysis of the file), a dynamic attribute or characteristic (something obtained through emulation or execution in a virtual environment and observing the behavior of the executed sample file)

or virtual attributes (that is a combination of previous types).

Attributes lists can be viewed as a pair (key, value) where the key is the name / identifier of the attribute and the value is its value. Attributes value can be one of the following (Boolean, Numeric, Enumeration values, Bit sets or strings).

In case of portable executable files, the following attributes can be extracted: header information, file hashes, imported functions, imported libraries count, exported functions, compiler used, Packers/Protectors used, Installer/Interpreters/Self extract archives/etc, resource information, disassemble information (number of call instructions used, API calls, …), different dynamically obtained information (usually counters for different actions that can be obtained from the virtual machine – number of threads created, files and registry modifications, system objects …) and many more.

Depending on the type of the script, other interesting information can be extracted. For example some languages have the ability to dynamically execute create and execute code (ECMA based languages

(JavaScript) is one of them). This ability offers them a way to create multiple layers of protection (just like a protector or packer for executable files).

Information that can be extracted from scripts base files can be classified as follows: string information (different strings that may indicate a specific behavior), API/Library function calls and different forms of obfuscation such as string addition, usage of random name variables or functions, garbage addition and so on.

During a period of almost 4 years, almost 20000 features were created. From this set more than 90% of these features are Boolean features.

Machine learning algorithms

The first step was to find a serious of algorithms that would be practical (will comply with the constraints previously described).

- Algorithms like ANN (or derivate) or SVM have a large memory and disk footprint and even if these algorithm have a good detection rate they cannot be use on a practical application for malware detection.

- I’ve selected the perceptron algorithm as it has a very small memory footprint and can be easily integrated with the features created.

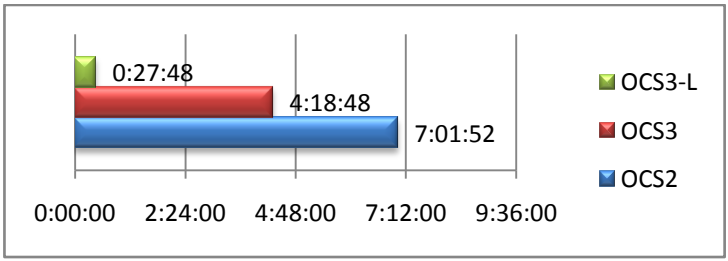
The next step was to adjust the perceptron algorithm for a 0-false positive detection. The basic idea was to split the training phase of the algorithm in two parts:

- First part that is identical to the one of a normal perceptron. The database records are analyzed and the model is adjusted for every record that is not correctly classified.
- The second part is to select a subset of records from the initial database that are labeled as clean. The model is then adjusted for every record of this subset until every record in this subset is correctly classified.

This method created the first one-side class perceptron algorithm. The main problem was that the second step was taking way to long. From the original algorithm, different versions of this algorithm have been developed that further improve the speed needed to make

this algorithm feasible (the third version of the algorithm ware named OCS-1, OCS-2 and OCS-3). Furthermore a new version based on lists of features instead of a vector of features was used to improve the final algorithm (OSC-3).

The following table shows the time needed for OCS-2, OCS-3 and OCS-3L (OCS-3 improved to used list) shows the time needed to perform 100 iterations on a database of 22 millions of records (malware and clean).



As it can be seen the speed improvement is quite substantial. If it is taken under consideration that the initial algorithm was almost 350 times slower than OCS-3 algorithm, than the final algorithm (OCS3-L) is almost 3400 times faster than the original one.

Having a 0-False positive algorithm also means a lower detection rate. The next step was to find different ways to improve the detection rate while maintaining the

0 – False positive constraints. There are few steps that can be performed in order to do just this:

- Find a method to improve data separability. This can be done by using:
  - Create new features that are more likely to separate malware files from the clean files
  - Obtain new features from the old ones in such a manner that the resulting set of features will better separate the malware files from the clean files
- Combine the same algorithms or multiple algorithms to obtain a better result
  - Use a voted system (a set of models obtain either with the same algorithm or from different algorithms).
  - Use a sort of ensemble system. This means that the algorithms are executed in a specific order. At each step, the files that are correctly classified in a specific class (or in both) are removed from the training set (so that the rest of the algorithms will only operate on a small set of data).

- Use a clustering hybrid method that combines a decision tree with a OSC based algorithm

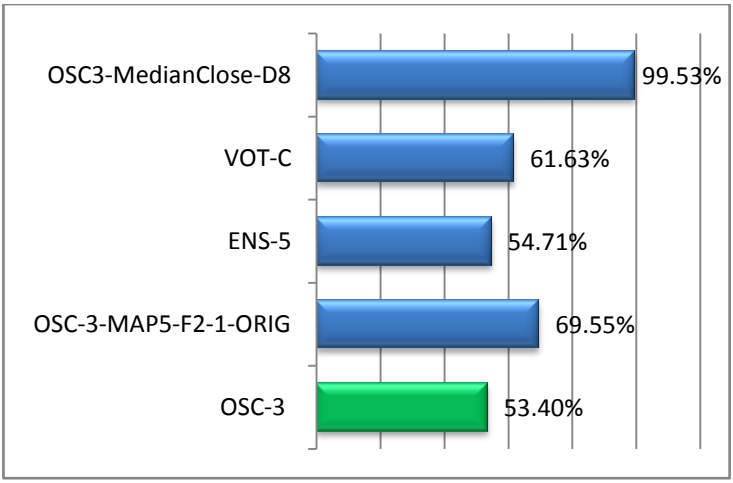
Creating new features means analyzing newly malware and find out characteristics that are specific to them. Another solution for creating new features is to start from a set of existing features and generate new features that will improve the detection rate while preserving a low false positive rate. The last idea can be done in the following way:

- Analyze value-based features and create new Boolean features from them (for example create a new feature is the number of sections from a specific file is bigger than 10).
- Apply different operations over two or more Boolean features to create a new one.

In the second case, the idea is to use different Boolean operators like AND, OR, XOR and apply them over two or more features to create a new one. One simple way of doing this is to map every two features into a higher space:

The next solution used to improve the detection rate was an ensemble-like algorithm. This works especially if the algorithms used can separate a subset of records that belong to the same class (for example KNN based algorithm). The same thing can be applied to an OSC based algorithm. Let's assume that an OSC based algorithm creates a hyper-plane that correctly classifies all of the clean files (most likely all of the clean files and some of the malware files will be on the side of the hyper-plane) and on the other side will only be the malware files correctly classified. This means that this algorithm can identify a subset of files of the sample label. For the OSC-based algorithms it is better to use different labeled subsets on every iteration.

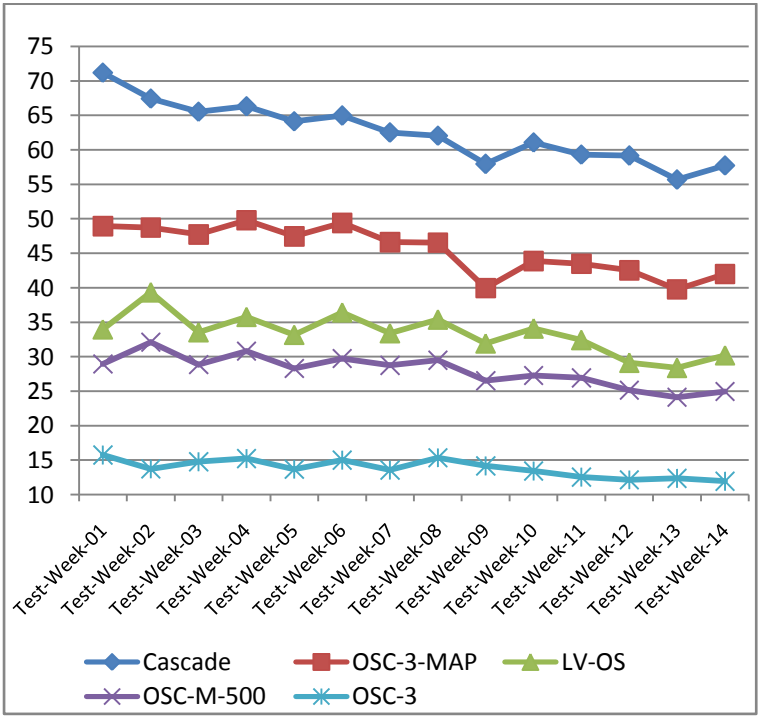
Another way to improve detection is to use a vote-like system – that uses multiple models with different weights associated to them for classification. The last solution is to create a form of decision tree that acts as a pre-filter for a set of models obtained with an OCS-3 algorithm.



The previous table describes the results obtained using some of the presented mechanisms. OSC-3 algorithm is the standard algorithm and was added as a reference. OSC3-MedianClose-D8 algorithm had the best result (it a decision tree based OSC-3 algorithm that uses MedianClose score and has a depth of 8).

The following graphic describes an experiment that was made to test proactivity. Over a period of 55 weeks both malware and clean samples were collected. The data collected after the first 40 weeks was used to create different models. Those models were then tested over the data collected from the next 15 weeks. For each record the detection rate and the false positive rate was

recorded and analyze. The following graphic shows the detection rate evolution over 14 weeks for 5 different models.



DataBase noises

In anti-malware industry, a common problem is that large data bases are not pure. The purity in most cases means that a malware sample is labeled as clean or

a clean sample is labeled as malware. There are 3 major factors that influence the purity of a sample set: human error, difference of opinions between different anti-malware vendors and samples that cannot be correctly separated through a linear classifier (this includes grayware files, file infectors, patchers, installers, interpreted languages, damaged files,).

The best way when analyzing noise is to have a manual analysis of the data. Besides the correct label of the data analyzed, this type of analysis can provide you with information regarding the feature extraction methods.

Finding possible noise can be done in the following way: searching similar files, based on the possibility of discovering with a high degree of certainty malware or clean files, if the source of a specific file is a trusted one, using the distance to a hyper-place, or a vote/ensemble system that uses the previously methods. The main problem with the similarity method is that the Similarity Score has to be computed for every pair formed from one malware and one clean file. If we're using a large database (over 20 millions of record) than

the time needed to compute the similarity distance between all of these pairs will be far too much for this method to be consider a practical solution. There are two things that can be done in this case: use a distributed system to perform all of the operations or use a clustering/classification method to split the large data base into smaller ones that are more manageable.

In case of the second idea, different methods for creating that tree were used – and for each method the total number of clusters and the estimated time needed to compute all of the similarity scores from that cluster were computed.

Method	**Clusters**	**Max cluster size**	**Estimated time**
AbsDiff	958	6,314,834	177 days 00:49:01
F1	6,420	2,522,738	16 days 07:23:35
F2	6,380	4,253,007	11 days 07:54:03
Information Gain	12	9,784,482	1 year 132 days 17:39:51
Asymetric Uncertainty	102	6,314,834	177 days 20:20:16
Median Close	**42,541**	**61,705**	**3:30:08**
ProcDiff	**31,056**	**119,270**	**1:07:32**

The selected methods were MedianClose (further denoted as MC) and ProcDiff (further denoted as PD). On the clusters/subsets created with these methods I've computed the following distances (that worked as a Similarity function): Manhattan (MH), Weighted Manhattam (WMH), CommonSet (CS) → (a score that measure the percentage of common features from the total number of features that are present in two records) and also some combination of those (votes). For each method the following data were computed: Noise percentage (NP) – represents the percentage of real noises from the possible noises, Total noise percentage (TNP) – represents the percentage of real noises found by a specific method from the total real noises that exists in a database and Global noise score (GNS) defined as follows: $GlobalNoiseScore = (NP \times k) + \big(TNP \times (1-k)\big)$

For this experiment the value of "k" was set to 0.5.

Method	**NP**	**TNP**	**GNS (k=0.5)**
MC_CS	**5.62%**	**97.64%**	**51.63**
PD_MH	14.02%	35.34%	24.68
PD_CS	7.73%	61.49%	34.61
V1_[MC_MH]_[PD_CS]	6.99%	65.34%	36.17
V1_[MC_CS]_[PD_WMH]	**5.56%**	**97.99%**	**51.78**
V1_[MC_CS]_[PD_MH]	**5.58%**	**98.03%**	**51.80**
V1_[MC_CS]_[PD_CS]	**5.38%**	**99.73%**	**52.56**
VALL_[MC_CSt]_[PD_CS]	8.74%	59.40%	34.07

The results showed that the CommonSet measure either used alone or in combination with another measures can provide very good results for noise identification in large data bases.

Script malware research

This section includes the analysis of malware that are not executed as a native code, but rather interpreted. This category includes: various script base malware (JavaScript, VBScript, AutoIT, and so on), virtual machine base malware (Java or MSIL based malware), document base malware (PDF, Microsoft Word, etc).

Since in all of these cases the original source can be obtained, the mechanisms that these malware can use are usually based on obfuscation techniques so that reading and understanding that malware is quite difficult. The most common one used are:

- Renaming every function, variable, class or namespace to a randomly generated name
- Add different comment with random content
- Split strings into multiple values

- Use different arithmetic operations to obtain the same result.
- Change the order of the code in such a way so that the execution flow will remain unchanged
- Remove indentation if the language grammar allows it
- Use specialized instructions like “eval” if the language supports it, to create different layers of obfuscation.

Even if the techniques required to analyze this sort of malware are the same for similar languages, I will focus on analyzing JavaScript malware and JavaScript based documents (especially PDF documents) since this is the most common type of malware. The special focus in this case was to create clusters that will include different malware families. The following approaches were used:

- First one is a Hierarchical bottom up clustering and it is based on a metric function that measures the distance between two files. The main problem in this

case was $O(n^2)$ time complexity needed to compute the distance between every possible pairs of files.

- The second method is based on a hash-table algorithm and it was designed so that it will have a linear time complexity and still have a very good clustering accuracy.

Both of these approaches used a file fingerprint (in this case a list of tokens from the files and their count from witch some well known tokens used for obfuscation (comments …) are removed). Two experiments were made. First one was to see the speed difference for clustering 10000 malicious PDF files using the two previously described methods.

Clustering Algorithm	**Clustering time**
Hash Table clustering	1 second
Hierarchical bottom up clustering (using custom metric distance – PDF Metric)	1 day, 21 minutes, 33 seconds

It is obvious that the hash table clustering was much faster. However, since the hierarchical bottom up clustering needed one day to complete the clustering for

around 10000 files, it was not included in the clustering process of the large data based (due to time constrains). Then only the hash-table algorithm was used to cluster a large data base of malicious PDF files (aprox.1 million malicious PDF files). 419 clusters were created. To verify the integrity of those clusters the 5 biggest one were selected for manual analysis and for each of them the actual number of files that should have been cluster in the same cluster was computed.

Cluster	**Files**	**Manually analyzed files**	**Similar files in the cluster**
#1	90502	4600	100%
#2	63792	3200	99.9%
#3	43816	2200	100%
#4	33389	1700	99.8%
#5	27080	1500	100%

In two cases, in cluster #2 and #4 were some scripts with slightly different tokens; however those scripts looked more like a derivate from the rest of the scripts than like a totally different script. This test proved than having a variable normalization factor is much appropriate for these databases than a fixed one.

GML Framework

GML stands for Generic Machine Learning Framework. It was develop as a tool to help test and research different machine learning algorithms. GML is written in C/C++ with different inline optimizations and assemble code.

It was design as a modular system with four major components:

- Algorithm component. This is the main component that receives data from other components and performs the algorithm. It has a build-in multithreading functions that allows one to easily use parallelism for a specific algorithm
- A database component. This is the component that holds the data in different formats (text, SQL, binary …).
- A connector component. This component links the algorithm component with the database component - and translates the data from the database in a format that is easily used by the algorithm components.

- A notifier component. This is a common global component that is used to show information about the state of the algorithm, database or the connectors.

All of these components have a library as a core unit that provides different functionalities:

- Interfaces for database usage and types
- Support for different string operations
- Support for a JSON-like format
- Support for parallelization (multi-threading)
- Support for different list-based function
- Different mathematical operations that are usually used in a machine learning algorithm (such as different kernel function, distance function, hyper-plane operation, and many more).

Over this architecture there is a Python wrapper that allows one to easily use and adapt different algorithms.

Conclusion and Feature Work

This thesis presented different practical aspects needed to improve malware detection mechanisms. All of the methods presented should not be consider as sufficient – in practice all of these methods are complementary and work together with static and dynamic detection and/or exclusion mechanisms as well.

However, the use of these methods will greatly improve the proactivity and the detection rate of any anti-malware product. As a particular case, applied on BitDefender engines these methods increase the detection rate to the point where most important independent tests consider BitDefender to the product with the best detection.

The work in this domain is far from being over. With the new cloud technologies that become more popular in the last two years new type of detection would emerge. The idea is to combine the data that is receive from different clients and create a semi-unsupervised system that would adapt itself to new malware attacks. This will not be an easy task, as the quantity of data that

can be received from every client can be huge. All of the algorithms will have to be design for both local and cloud decisions creating a sort of large neural network. The use of cloud system for malware detection will allow other algorithms (like support vector machine or artificial neural networks) to be used. These algorithms are normally consider not to be practical as they need space and computing power that is not available on a client local machine. However, moving this sort of detection into cloud and combining with different parallelization methods may create newer and better detection methods.

The evolution of graphical video cards is also a new direction that can be considered as important. Newer graphic video cards have multiple processors that can highly improve the time needed for data processing. Similar operations can be performed using cloud processing units.

Last but not least, similar researcher as this thesis can be perform on other platforms – as Android malware tends to grow newer and different methods should be created for this type of threat.