Malware Classification

Feature Engineering and Classifier Building

DataMiners Co., CSCI 565, Prof. Dalkilic
Das, Srijita **Das, Mayukh**
Rivero, Andres

Email: {sridas,maydas,arivero}@indiana.edu

May 8, 2015

Contents

Mission Statement

Security is an important part of the digital era. Users store large amounts of private information in personal devices, on websites, or the cloud. But, how sure can we be that this information is being protected using all the means available? We live in a time where flaws in the system can be exploited to infiltrate even the a biggest corporations in the World. Companies such as Anthem, Target, and even Apple have been targets to well-crafted attacks. This mistakes bring down the reputation and confidence of their users to trust them with sensitive information.

We take the malware problem serious, because it affects us all. Consequently, we have taken a vast data set of well-known viruses and their labels in hope that this will contribute to classification of future incoming data. Our approach takes a training set used to take common features in a family of malware. As a result, we have a large set with features extracted that are used to classify and validate out techniques. This is possible with a second set that is used as testing set that has no label information. Malware categorization never seemed easier than it is today. Good models can a be built to predict the intentions of a program thanks to the large data collected on malware over the year and the new data mining techniques.

DataMiners Co.® provides customized software to your workplace. We evaluate the needs of your company to provide the best analysis software in the market place. Our staff knows that vulnerabilities in your system are the door way to your money vault. Therefore, our software becomes the security that checks the ID at the door. We have access to all kinds of malware, which we use to predict whether incoming transmission is benign or malware. Our goal is to offer the best service to your corporation. The happier your users are with your security measures, the happier you will be with our assistance.

DataMineWare is the system that has been used to predict whether there is a thread in the payload. Where your antivirus fails to recognize zero-day attacks, DataMineWare can predict accurately that the payload is malware. The goal is to protect vulnerable systems from being exploited. We know how important the security of our customers is, therefore, we know how important the security of your customers is for you.

Members

We are very proud to count with very capable, experienced, and dedicated graduate students from Indiana University. Each one of them with experience in different areas of the problem at hand to develop only the best on the market. They came together to cater to the security area with the best product yet. Their names and skills are as follow:

- **Andres Rivero** - Computer Science student at Indiana University from Brazil. His main interest is Security and Networks. He will graduate in December with Masters of Science in Computer Science. His experience and interest in Security and Networks helps with preprocessing and analysis phase for any project in malware detection. Moreover, with the skills developed in Data Mining promotes his expertise to an excellent analysis of provided malware. Some level of experience managing and querying MySQL and PostGre databases. His first experience with Data Mining techniques was at the beginning of Spring semester. Eager and fast learner, he likes to encounter new problems that can challenge his intellect.

- **Srijita Das** - 1^{st} year Computer Science Ph.D. student at Indiana University, Bloomington. She is a Research Assistant at the Privacy lab. Her research interest includes privacy concerns of queriers in Online Social Networks, security in networked systems and data mining techniques to explore behavior of people in Online Social Networks. She has three years of industry experience with Mainframes and databases like DB2 and Oracle. Her experience with Databases and the various data mining techniques learned from this course will help her contribute towards some of the non trivial data mining tasks like feature extraction and data management part of this project.

- **Mayukh Das** - 2^{nd} year Computer Science Ph.D. student at Indiana University, Bloomington. He is presently a Research Assistant at the StarAI Lab. His main interests lie in Scalability of Inference algorithms in Statistical Relational AI, and Relational Inverse Reinforcement learning.

He has industry experience in working with large scale Data warehouses and Analytics systems (across varied Database platforms such Oracle,MySQL & PostGreS). His skills are nicely aligned for the job at hand and the Datamining concepts learned from this course will enhance his abilities to contribute to this project.

0.1 Executive Summary

In this challenge, we started with a business proposal for a company dedicated to malware classification using state-of-art datamining techniques. We focus on the team member that made this idea come true with their dedication and expertise in the area. This makes our work environment feel like home. None of this would be possible if there was not a need to improve security in systems. Additionally, we strive to provide improvements on what was offered in the past. That is why we discussed in detail previous solutions in malware classification.

The problem itself has various approaches, but (with the evolution of technology) the new approach deals with real-time analysis to avoid infiltration. Therefore, we provide a solution to the malware classification challenge that extracts best features from the training set. This is later validated with another test set. Feature extraction takes from one of the types of files provided for training and testing. It has quality to offer, and that is why we take into consideration features from *byte code* file of the malware.

As in any other project, there is always room for improvements. But, there is always a lack of resources, in this case, *time*. There are many features that constitute the signature of a virus, but without the specific knowledge on malware is not possible to translate this into code that always selects this features as top. Another important advance, there are many datamining algorithms that might have a good chance of producing better outcomes. This step might need more time to generate results using different approaches for classification and comparing. Therefore, selecting the best out of all the models.

Finally, we can still inform that the project was a success. We were able to retrieve important features that categorize malware as belonging to a specific family. This was achieved thanks to the usage of $n - grams$ from byte code files. Furthermore, we made use of many libraries available that provide efficient implementations of the classification techniques.

0.2 Problem Statement

Nowadays, the common user performs all sort of financial transactions online. This is an issue when we look at the news and discover that big corporations have data breaches leaking thousands of private financial information. In addition, users also worry about their personal information leaking to just anybody. Most common registration forms ask for name, address, and phone number. If we combine financial and private information, we have a potential for impersonating.

The Microsoft Malware classification challenge [1] is an attempt to gather the best of the best to mitigate this issue to minimum. However, they are not the answer. The challenge consist in providing the all the information required to build a system able to filter malware based on known families of malware. In other words, we are expected to introduce techniques from Data Mining, Machine Learning, and Algorithms to predict whether the payload contained in the incoming packet is a threat.

The problem of the problem is to extract a large amount of malware data to train a software to recognize malicious content. Issues emerge based on storage, and what needs to be stored. How many files should we analyze without hurting efficiency? Is byte code or assembly instructions a better predictor of malicious code? What are the best algorithms to tackle the massive data? There and many more questions are posed by the Microsoft Malware Classification challenge [1]. But at the end, we are trying to protect malicious code from exploiting vulnerabilities in the existing systems. We expect that avoiding malicious code to penetrate the system will lower the chances from data breaches that leak private and personal information about users.

In this challenge [1], kaggle in collaboration with Microsoft, offers the hacker community the opportunity to compete for a monetary prize. Moreover, the winner will also contribute to the improvement of

contemporary tools that analyze and classify by grouping new files into families. Given the amounts of information being exchanged online, we need an efficient, real-time detection tool that can predict with high accuracy the purpose of incoming files.

0.3 Investigation/Research

There have been many papers published that try to answer the questions asked in the problem statement. The reason being that security and analysis of malicious content is not a new hot area to study. In the past, experts would analyze programs by hand. Followed by the apparition of automated tools that look for text string, filenames, or byte code signature. Nowadays, with the advance of technology, we have new areas increasing in popularity. Data Science and Big Data are two fields related that attempt to solve issues related with huge amounts of data. Thus, we have techniques from Big Data, Data Mining, Machine Learning and Statistic fusing to solve the holes in security. This is possible due to large collection of archives containing malware information we are able to analyze and classify new files.

Current approaches attempt to match new files to already known existing malicious programs. However, the question remains as, what do we need to match known malware to a possible threat? Current research is focused at extracting byte code, known as signature, to classify programs into the different categories. However, this is what antivirus software does. It is not enough to compare to a known signature, we now need to use the information given to predict whether we are facing malicious or benign programs. Therefore, state-of-art security relies on data mining techniques to select the best features from a given collection of malware in a family. There are three possibilities for selecting features that experts tend use: byte code, assembly instructions, and a hybrid approach.

Schultz *et al.* [6] offered a solution based on antivirus methods and techniques to analyze files, signature-based analysis. In addition they offered two more tests based on extracting features by what libraries were being called, and the text string being used as static data. These three approaches generated features and rules of their own. The experiment consisted of a small data set with malicious and benign code. Therefore, they run into possibility of refusing execution of a legitimate program. They evaluated their solution by categorizing the test data into four classes: true positive, true negative, false positive, and false negative.

In Siddiqui's thesis [8] we start seeing the first appearance of a hybrid approach using both byte sequence and assembly instructions to effectively classify each program. In addition to having the best approach, Siddiqui invested time in studying the importance of the areas that are static and dynamic to produce a better result. Various models and algorithms were borrowed from data mining to get an optimal result. In this thesis, the author played more with the different families of malware, However, this experiment was done with a limited data set of malware. The same as *Schultz*, the author was mainly concerned with the Windows OS.

Masud *et al.* has various papers and instructional manuals in the area. [MasudTechReport] used another hybrid approach to extract features from byte code, signature-based, and analyzing sequence of instructions from assembly code. They obtained better results from by extracting features from the byte code, but that was not a discouraging find. Instead, they mixed those two approaches to improve the statistical chances of classifying new programs.

[kolter] is another experiment where the authors decide to use techniques and algorithms from data mining and machine learning. Not only that, but they become smart on their approach. The authors dedicated most of their efforts in the extraction of features, using already existing libraries to classify the new programs. Even though they focused on features, their outcome was pretty much similar to other experts that came before them.

It is hard to encounter pure techniques that use either byte code or assembly. One of the major issues is to find the source code for given malicious program. All the previous solutions use same approach to extract the signature from the byte code. N-grams are used to analyze similar sequence of byte code in the given families, or just malicious code. Authors did not assume that the signature given from

antivirus detection was the only signature for the malware, instead they preprocessed the code knowing it was malicious.

The most common approach to use the assembly was using n-grams as well. However, Masud et al encountered issues. They do not specify how they instruction were broken down, but the some issues with facing memory addresses, registers, or constant data seem like a possibility. Other expert suggest parsing the source code to leave out the operands. Also, the breakdown would stop at any jump having all these sequence of instructions as a single feature and then performing all possible permutations to compare against other malicious programs in the same family. This approach increased the effectiveness of using instructions as a way to extract features for [Author].

0.4 Existing Solutions

Malware Classification is a data mining task which has been studied in past and researchers have come up with various possible solutions to this task. Some of the existing solution to this task is jotted down as below.

One of the solutions proposed by Mohammad M. Masud, Latifur Khan and Bhavani Thuraisingham was a hybrid model for feature extraction and feature selection from the byte code and Assembly code malware files. Below jotted down are the steps they followed in their approach.

- They extracted N-Gram features from the *malware byte codes*. In order to do efficient memory management, they used AVL trees to organize the N-grams and copy the them to hard disk once the number of N-grams contained in the AVL tree exceeded a threshold limit.After extracting all the N-grams from the byte code, they selected the best N-grams from them by using the *Information Gain criteria*.
- They extracted features from the *assembly code* file of the malwares. Each Instruction was represented as a tuple and they extracted all the instructions corresponding to a N-gram byte code feature. They applied heuristics like *"Most frequent Substring"* and *"Most distinguishing string"* to select the best instruction sequence corresponding to a byte code as feature.
- The *DLL function calls* in the assembly codes were also used in this approach as features.
- They combined the features extracted from byte code and assembly code and used the final feature set to predict whether a file is malicious or benign.[6]

Weakness:

- The weakness of this approach was that extracting features from the assembly codes of the malwares required some *domain knowledge* of malwares. Without some knowledge about how malware code looks like, which are the portions of the assembly code that are more likely to be infected by malwares, it is difficult to efficiently extract useful features for malware detection from the assembly code malware file.

Another solution proposed in regard with this task was the construction of N-gram features of the opcodes of assembly level instructions present in the assembly code malware file. An Opcode is a portion of the machine level instruction that specifies the operation to be performed. Opcodes provide more meaningful insight into the features than the byte code N-grams. Since, opcodes are the basic entities of machine level instructions, so consecutive opcode sequence were used to construct N-gram patterns as features.The steps taken in this approach are jotted below.

- *Opcode N-gram patterns* were created which represented the instruction flow in the assembly file.
- Opcode N-grams were created for various values of n and later normalized *term frequency(TF)* and *TF inverse document frequency(TFIDF)* were calculated for each of these Opcode N-gram representations.

- Feature extraction was done on the basis of highest Document Frequency values.
- Feature Selection algorithms were applied to select features and then various classification algorithms like Logistic Regression, Random Forest, Artificial Neural networks were employed to predict the classes of malicious and benign files.[9]

Weakness:

- This approach also has the same bottlenecks as the first approach. Extracting features from the assembly code require*domain knowledge* about malwares.
- It also requires a good knowledge of *Assembly level instructions* and various opcodes that are present in Assembly level codes.

Another approach proposed for malware detection was a way to identify the signature of various malwares as features to detect malicious files. The steps of this approach are as jotted below:

- In this approach, the viruses were allowed to infect some clean programs and then these programs were analyzed to detect the characteristics of malware and hence extract the features. So, basically *signature of malwares* were detected from these infected programs and the best signature was the one with the *lowest false positive probability.*
- The best features are selected and classification algorithms are employed to detect malicious and non-malicious files.[10]

Weakness:

- The problem with this technique was the *code obfuscation techniques* used by malware attackers. Obfuscation is a technique of hiding the malware program's typical signature by simple actions like bit manipulation and hiding the contents of file with a special program. Malwares which employ code obfuscation in their codes cannot be detected with this technique of signature detection which are further used as feature.

0.5 Optimum Solution

0.5.1 Optimal N in N-Gram

One of the major challenges in taking the N-Gram approach to extract features from the byte code malware file was the correct choice of N. Below are some of the reasons why 4-Grams works the best and we used it in our approach.

- As per Shabtai[9], 4-Grams works the best and gives the best accuracy because 4-Grams best *represents the instruction* in the assembly code malware files. The authors experimented with every possible combination of N-Grams ranging from 1 to 6 and found out that 4 is the value of N for which the classification algorithm gives the *maximum accuracy.*
- We also carried out the same experiment on the byte code malware file and saw that classification accuracy for 4-Gram was the best.
- Another reason for using 4 as N in the N-Gram feature extraction was the fact that smaller value of N like 1 and 2 resulted in large number of N-Grams and hence, the frequency of occurence of these N-Grams across the Byte code files were extremely high. Such *high frequency count* of N-Grams *did not give us a meaningful insight* into the features and hence feature extraction was not accurate. Similarly a high value of N ($N > 5$) would resulted in *lesser number of N-Grams* and hence the occurrence of these N-Grams accross all the byte codes *were too low to extract meaningful features from them.*

Hence, its seems that $N = 4$ is an optimal choice of N-gram for the N-Gram feature extraction process.

0.5.2 ByteCode or ASM files

The next decision that was to be made in course of this project was which file to use for feature extraction- the Byte code or the Assembly code file. Jotted below are few of the reasons why we decided to extract features from the Byte code files instead of the Assembly code files.

- The Byte code consists of the hex dumps corresponding to the assembly level code. The Assembly file code consists of the assembly level instructions for some of the Byte code. We thought that extracting N-Gram features out of Byte code file will be more pragmatic and feasible approach because our problem statement is to classify the malware files into the 9 available classes. The application of such classification of malwares is over the Internet and in huge organizations where malwares corrupt important datasets serving a specific purpose. The *Malware infected files* over the internet is essentially in the form of *Byte codes (i.e: Compiled version) and not Assembly code (uncompiled vesion)* and hence, extracting features out of Byte code file should be more appropriate than from Assembly level code.
- Another reason that compelled us to extract features from the byte code file is that extracting features from the *Assembly code file* requires a *good domain knowledge* of malwares which in real time is quite difficult to achieve. The assembly code File contains assembly level instructions and extracting something meaningful from it requires a good insight into what are the portions of the code which are more likely to be attacked by malwares, how does the code looks different from another code when malware attackers employ code obfuscation techniques etc.

Hence, the above intuitions made us extract features exclusively from the byte code file.

0.5.3 Flat features vs Function call graph

Another decision which we had to take while doing this project was whether to consider the flat N-Gram features from the Byte code malware files or to decide features by creating function call graph of the Assembly code malware files. We choose the first approach due to the following reasons.

- According to us, creating graph of all function calls from the assembly level code *may not give us anything meaningful*. What it would signify is essentially the *entire flow of the program* and the order in which modules are called from one another. It is highly unlikely that *different classes of malwares will have different module flow structure*. If different categories of malwares have different flow structures, then it would be too easy a task to classify them to their respective classes. It is *more likely* that the *module flow would be the same*, but some small chunk of code within these modules will have similarity with respect to a particular class of malwares.
- These days a large number of *code obfuscation techniques* have been employed by malware attackers so that signature of a particular malware remains undetected. Function call graph will not be able to deploy anything meaningful with respect to code obfudcation techniques because these techniques include minute changes inside the modules. So, these changes are not applied on the overall structure of the code but intelligently applied to small portions of code inside modules which performs the same functionality as the desired malware.

Hence, our assumption is that mining malware features from function call graph is going to be less effective in extracting meaningful features for classification of malwares to their respective classes. On the other hand extracting flat features from the Byte code file is going to be more effective because real time malware files on Internet are in the form of byte codes, so it is very likely that these byte codes will be similar in some respect to each other for a particular class of malware.

0.5.4 Dimensionality

The last but one of the most important decisions we had to take during this project was the number of features to be considered for the classification problem. Below are the points that we considered for choosing a small subset of the total number of features for classification of Malwares.

- Large number of features implies *high dimensionality* and that brings into play the *curse of dimensionality*. High dimensional data leads to *poor classification accuracy*.
- In high dimensional space, *data* becomes *extremely sparse* and most of the *mass* gets concentrated across the *corners of the hypercube*. As a result, with high number of dimensions, methods become less local.
- The number of *neighborhoods corresponding to each data point decreases* which leads to classification algorithm working poorly in high dimensional space. This is what is called curse of dimensionality.

Our 4 Gram approach generated a huge number of features. The number of features extracted from approximtely 11000 Byte code files were close to 10 million. With such a huge feature set, the classification algorithm would have worked poorly. Hence, we employed some methods to reduce the feature set. We got *rid of the less meaningful feature* and *kept the most prominent features*. For that, we filtered the features based on how prominent one feature was across one class and less prominent across other classes of malware. Our assumption is that the N-Grams which have *high frequency count* in almost all classes are the ones which are generally present in all Malwares and hence, these features are *not going to give us any meaningful insight* about the class of malware. On the other hand, the features which have a *very high frequency count in one class of malware and not present or is low in other classes* of Malwares are the one which will characterize that particular class of Malware and hence we considered such features extracted from Byte code to be *relevant*.

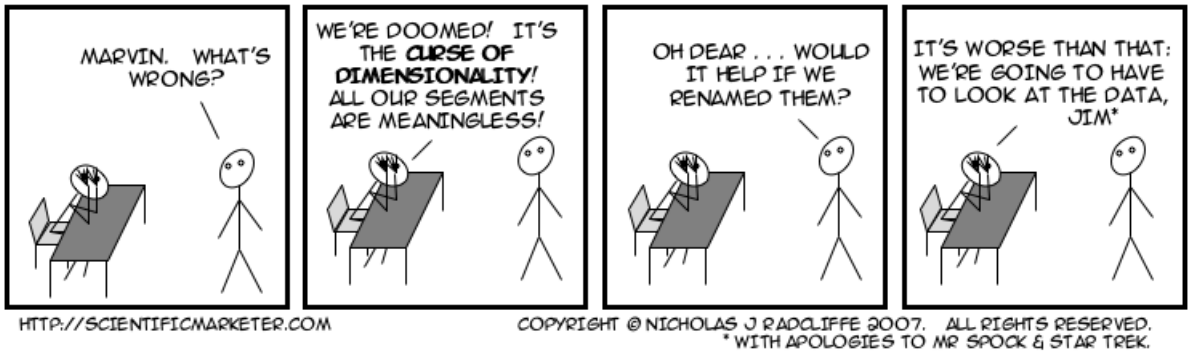

Figure 1: Curse of Dimensionality

0.6 Construction/Implementation

There are approximately 3 primary pivotal steps to this task. Though one can imagine finer divisions but there is no point in that. The steps have been laid out in the sub-sections that follow.

0.6.1 Data Engineering

The data is in the form of Bytecode and equivalent Assembly files generated by running the Bytecodes through some standard De-assembler. The main aim at this stage is to extract/generate usable and non-trivial features from this huge corpus of test [the bytecode and assembly files]. However, that is not an easy task. Feature extraction is equivalent in complexity to *structure learning* in propositional or relational models[2].

Again this task is not similar to entity of relation extractions in Natural Language Processing tasks, since, there are no entities or relations here. These are assembly level programs and their equivalent bytecodes. Here, Operators, Opcodes and operands make more sense. Thus, all we can do is look for patterns of code fragments. For pattern extraction we employ the N-Gram extraction. However, before we can proceed with extracting N-gram patterns from the bytecodes we need to clean and/or transform the data as required for optimal performance.

Data Cleaning

There were a few importing facts that we noticed on close examination of the ByteCode files as well as the Asm files. These factors guided us on what and how to clean the data.

- The pre-dominance of the the character sequence "*CC CC ....*" in most of the byte code files irrespective of the class membership. On further investigation we determined that, this character sequence is actually the Hexcode for "*alignments*" in the assembly program. Thus if we choose to exclude such character sequences, then there would not be any danger of loss of valuable features since this character sequence is present across all files.
- The character sequence "?? ??....." also seemed quite frequent across all the byte code files, across all classes. Though this particular sequence is not equivalent to any particular operator,operand or any other coding construct, instead it may represent multiple kinds of declaration statements, yet it can also be excluded from the N-Gram computation owing to the fact that it occurs massively across all files in all classes and thus has low chances of it being a useful feature.

Thus we reconstructed the files cleaning out the above mentioned character sequences. Such cleaning not only reduced the sizes of files by manifolds making them more manageable, but also improved the efficiency of N-Gram extraction/construction. The figure 3 below reports evidence[1].

Figure 2 shows tthe frequency of each class.

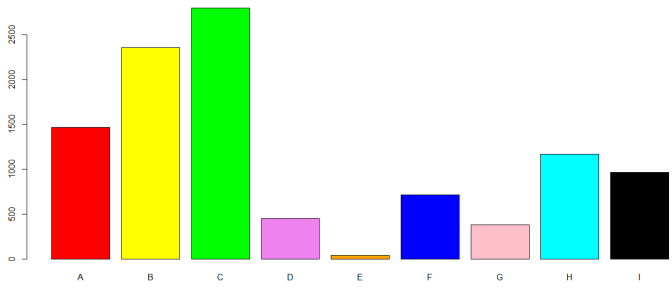

Figure 2: Frequency of each Class

Pattern/N-Gram extraction

After the files were reconstructed following the cleaning procedure, we started extracting different N-Gram patterns. There were a few interesting observations.

1. When we used $N = 2$ i.e. 2-Gram pattern extraction, the patterns extracted was too general and seemed to be having an overwhelming cardinality and thereby did not seem too useful as features.
2. When we used higher Ngrams ($N > 5$) the patterns were too specific and the cardinality of each pattern was too low to be significant.
3. $N = 4$ however seemed to be a suitable number. [6] also discusses how 4-Grams work best since 4-gram patterns of hex-codes from byte code files potentially represent identifiable instruction sets.

   We have used an *Apache Lucene Index* based extractor library in Java, for N-Gram extraction.

[1] We have only performed cleaning on and subsequently used for classification, only the byte code files. The reason for this has been explained in the previous sections. An informal statistical justification for this is also provided in the analysis section.

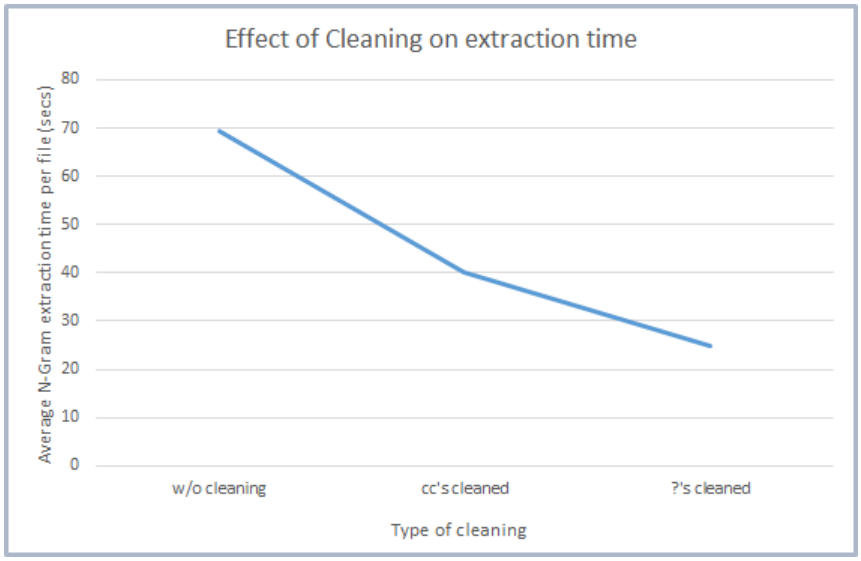

Figure 3: Improvement via cleaning

Lucene Indexing: *It is a text database indexing system. In simple terms lucene only does two things really. 1) It creates search indexes. 2) It searches for content in those indexes. An index is a efficiently navigable representation of what ever data we need to make searchable. Data might be as simple as a set of Word documents in a content management system, or it might be records from a database, HTML pages, or any kind of data object in our system.* [1]

The steps of N-Gram extraction using lucene based extractor is laid out below in **Algorithm 1**. Once the N-Gram extraction is done and the Ngrams are loaded into a MySQL Database (separate

Algorithm 1: N-Gram Extraction

Result: N-grams for each "Class"
Initialization:
  Clean all Byte Code files;
  Set $N = n$. Set up **MySQL** database **mDB**;
  Sample Size $K = 1$;
for *Class* $\mathcal{C}_i$ *1 TO 9* **do**
  Randomly Choose a Sample subset $\mathcal{S}_{Ci}$ of size K of files in the class $\mathcal{C}_i$;
  File ID counter $\mathcal{I} = 1$;
  for *Every file* $\mathcal{F} \in \mathcal{S}_{Ci}$ **do**
    Tokenize whitespace separated words and create lucene index;
    Extract N-Grams from indexed DB;
    for *Each N-gram* $\mathcal{N}_g$ **do**
      Insert tuple$(\mathcal{N}_g, \mathcal{I})$ into **mDB**.table_$\mathcal{C}_i$;
    end
    $\mathcal{I} = \mathcal{I} + 1$;
  end
end

tables for each class) we move to the Feature extraction Phase.

0.6.2 Feature Extraction

Feature extraction involves quite a few customizations for the following factors/problems/issues:

- *High Volume:* The volume of data itself is very huge and just running a brute force algorithm to iterate through all N-Grams and finding the best feature is not even remotely feasible.
- *Dimensionality:* Careful consideration need to given to extracting absolutely the minimal set of features from the huge set of generated 4-Grams. The reason being, using too many features is not particularly a nice idea, since it will supper from the 'curse of dimensionality'.
- *Scoring for elimination:* There needs to be a theoretically viable scoring function, in order to decide which of the features to choose and which to drop.

Now let us trace the steps of Feature extraction in an informal descriptive way. After we make the intuition clear we will go ahead and formalize the Algorithm.

1. *Phase I:* For each class $\mathcal{C}_i$ and corresponding table **mDB**.table_$\mathcal{C}_i$.
   (a) Fire a distinct query such as (example below is for class 1)

   ```
   SELECT distinct ngram, file_id FROM table_C1;
   ```

   Let the result be a Result Set $\mathcal{R}_i$
   (b) From result set $\mathcal{R}_i$, 'count' by grouping on the *ngram* attribute. This essentially gives us the *number of files* in the class $\mathcal{C}_i$ in which any particular 'ngram' string is present.

   ```
   SELECT ngram, count(1) as cnt FROM resultset_Ri GROUP BY ngram;
   ```

   We store these counts in a new table **mDB**.table_summary_$\mathcal{C}_i$. Below are some example results:

   ```
   +-------------+-----+
   | ngram       | cnt |
   +-------------+-----+
   | 00 00 01 10 |  21 |
   | 00 00 01 30 |  21 |
   | 00 00 0A 0A |  21 |
   | 00 00 0F 8F |  21 |
   | 00 00 1B 00 |  21 |
   | 00 00 39 75 |  21 |
   | 00 00 3B C2 |  21 |
   | 00 00 3B C5 |  21 |
   | 00 00 43 6F |  21 |
   | 00 00 46 69 |  21 |
   | 00 00 4C 01 |  21 |
   | 00 00 50 03 |  21 |
   | 00 00 50 55 |  21 |
   | 00 00 53 65 |  21 |
   ```

   (c) We exclude the tuples with very low count value <= 5 from **mDB**.table_summary_$\mathcal{C}_i$. The reason behind this is that if an N-gram is not found in a reasonable number of files in that class then it may not be useful at all, it could just be some random piece of bytecode very specific to the programming practices of the coder or may be generated by some specific compiler.
   (d) Repeat for all classes.
2. *Phase II:* Once the above tables are generated we move to the next phase of feature generation. Initialize final feature table **mDB.final_features** ($\mathcal{T}$)

   Now for every class $\mathcal{C}_i$ and corresponding N-Gram Table **mDB**.table_summary_$\mathcal{C}_i$
   (a) For each N-GRam $\mathcal{N}$ in **mDB**.table_summary_$\mathcal{C}_i$ with $cnt > \tau$ where τ is a tunable threshold :
   Initialize $Presence = \{C_i\}$ and $\#Presence = 1$

i. Scan through $\forall$ **mDB**.table_summary_$\mathcal{C}_j$ where $i \neq j$

ii. If $\mathcal{N} \in$ **mDB**.table_summary_$\mathcal{C}_j$ and $cnt > \tau_j$ which is tunable $\#Presence+ = 1$ and $Presence = Presence \cup \{C_j\}$

iii. If $\#Presence \leq 3$, INSERT tuple $(\mathcal{N}, Presence)$ into $\mathcal{T}$.

(b) Repeat for all N-grams strings int the table,

3. *Phase III:* Once the final feature table is generated we checked and found that there were about 5096 features . These huge number of features will literally crash any Classifier and will essentially be useless since we do not have enough data points to even reasonably cover this excessively high dimensional space. Below is are a few entries of the table **final_features**.

```
mysql> select * from final_feature limit 15;
+-------------+-----------+
| ngram       | domclass  |
+-------------+-----------+
| 00 00 70 70 | [1, 9]    |
| 00 00 E4 00 | [1, 6, 9] |
| 00 20 00 72 | [1, 6, 8] |
| 00 77 77 00 | [1, 9]    |
| 00 F0 FF FF | [1, 2]    |
| 10 2B E0 53 | [1, 2]    |
| 10 89 6C 24 | [1, 2]    |
| 10 8D 6C 24 | [1, 2]    |
| 10 FF 75 0C | [1, 2]    |
| 24 10 2B E0 | [1, 2]    |
| 24 10 89 6C | [1, 2]    |
| 24 10 8D 6C | [1, 2]    |
| 2B E0 53 56 | [1, 2]    |
| 33 F0 FF 15 | [1, 2]    |
| 44 24 08 8B | [1, 2, 9] |
+-------------+-----------+
```

We performed feature selection. The details have been explained in the next section.

0.6.3 Feature Selection

Selecting a minimal subset of features is quite a difficult task. As mentioned earlier it will affect the performance and accuracy of any algorithm we try to run. Many approaches exist for this task, and research is still active. But there is not single solution that can potentially work every where. The feature selection methods may vary based on a few important criteria, namely:

- *Nature of the data:* Like type or distribution.
- *Functional Dependencies among features*
- *Size of the data set.*

There are several approaches based on the criteria mentioned above. Some of them are:

- Correlation based Approaches[11]: Where we exclude all but one in a highly-correlated set of features. But calculating correlation between every feature-pairs in a huge dataset is many a times not *feasible.*
- Entropy/Info-gain based methods: We exclude features that do not cause considerable drop in Cross-entropy between the class and the feature. It is indeed more time efficient than correlation based methods however it may not be able to filter out as many features as Correlation based methods may.

Thus we had to find the sweet spot based on the challenges we had, namely, huge number of features to deal with.
Thus here's the approach we took.

From the final feature table **mDB.final_features**($\mathcal{T}$) we selected those features whose $n(\mathcal{T}.Presence) == 1$. This is essentially ends up being both Correlation based and Entropy Based. Here's how:

- If the N-Gram is has presence in 1 Class then it has essentially high Cross-entropy with the Class variable. Thus this is similar to Entropy based methods. Thus, now, a particular feature will be able to strongly identify a class.
- Presence of N-Gram in multiple classes hints that There could exist some correlation with the features that have similar class presence. Thus choosing features with one class membership ensures that the features are de-correlated from each other. below is a sample output.

```
mysql> select * from final_feature where domclass = '[3]'  limit 10
    -> ;
+-------------+----------+
| ngram       | domclass |
+-------------+----------+
| 01 8B 96 87 | [3]      |
| 58 50 FF B0 | [3]      |
| 83 EC 2C 8D | [3]      |
| 83 EC 38 8D | [3]      |
| F0 01 8B 96 | [3]      |
| FF 8B 97 86 | [3]      |
| 28 5D C3 00 | [3]      |
| 3C 5D C3 00 | [3]      |
| 50 5D C3 00 | [3]      |
| 83 EC 28 8D | [3]      |
+-------------+----------+
10 rows in set (0.00 sec)
```

We experimented with features with multiple class presence later and it has been discussed in the next section.

0.6.4 Dataset Generation

Once we had the features (874 after feature selection) we generated a binary valued data set. The steps are mentioned below

- For every file $\{\mathcal{F}_i\}_1^N$
  - For every N-Gram in the feature set $\{\mathcal{N}_j\}_1^K$ check If exists in File then 1 else 0
  - insert tuple $(\langle x_1, \ldots, x_K\rangle_i, \ell_i)$ into the data set where $\langle x_1, \ldots, x_K\rangle_i$ is the bit vector we just created.

Once the data set is generated we take 10% of the tuples of each class into a separate Set called the Test Set. and leave the rest as Training set.

0.7 Analysis & Testing

After the feature selection we got about 874 features and that was reasonable number to work with. We used weka to learn different Classifiers:

0.7.1 Naive Bayes

With the training Set we learnt a **Naive Bayes** Modelwith 5 fold Cross Validation and tested it on Test Set we had created. Below is the test set evaluation:

491	9:I	9:I		0	0	0	0	0	0	0	0	*1
492	9:I	9:I		0	0	0	0	0	0	0	0	*1
493	9:I	9:I		0	0	0	0	0	0	0	0	*1
494	9:I	9:I		0	0	0	0	0	0	0	0	*1
495	9:I	9:I		0	0	0	0	0	0	0	0	*1
496	9:I	9:I		0	0	0	0	0	0	0	0	*1
497	9:I	5:E	+	0	0	0	0	*0.996	0.004	0	0	0
498	9:I	9:I		0	0	0	0	0	0	0	0	*1
499	9:I	9:I		0	0	0	0	0	0	0	0	*1
500	9:I	9:I		0	0	0	0	0	0	0	0	*1
501	9:I	9:I		0	0	0	0	0	0	0	0	*1
502	9:I	9:I		0	0	0	0	0	0	0	0	*1
503	9:I	9:I		0	0	0	0	0	0	0	0	*1
504	9:I	9:I		0	0	0	0	0	0	0	0	*1
505	9:I	9:I		0	0	0	0	0	0	0	0	*1
506	9:I	9:I		0	0	0	0	0	0	0	0	*1
507	9:I	9:I		0	0	0	0	0	0	0	0	*1
508	9:I	9:I		0	0	0	0	0	0	0	0	*1
509	9:I	9:I		0	0	0	0	0	0	0	0	*1
510	9:I	9:I		0	0	0	0	0	0	0	0	*1
511	9:I	9:I		0	0	0	0	0	0	0	0	*1
512	9:I	9:I		0	0	0	0	0	0	0	0	*1

```
=== Summary ===

Correctly Classified Instances         510               94.6197 %
Incorrectly Classified Instances        29                5.3803 %
Kappa statistic                          0.935
Mean absolute error                      0.012
Root mean squared error                  0.1093
Total Number of Instances              539

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.987     0          1         0.987     0.993      1         A
                 0.927     0.002      0.991     0.927     0.958      0.967     B
                 0.986     0          1         0.986     0.993      0.999     C
                 0.87      0          1         0.87      0.93       0.998     D
                 0.5       0.015      0.111     0.5       0.182      0.987     E
                 1         0.038      0.661     1         0.796      0.996     F
                 0.842     0.002      0.941     0.842     0.889      0.993     G
                 0.934     0          1         0.934     0.966      0.996     H
                 0.88      0          1         0.88      0.936      0.999     I
Weighted Avg.    0.946     0.003      0.969     0.946     0.954      0.991

=== Confusion Matrix ===

   a   b   c   d   e   f   g   h   i   <-- classified as
```

```
  76   1   0   0   0   0   0   0   0 |   a = A
   0 114   0   0   3   6   0   0   0 |   b = B
   0   0 145   0   0   2   0   0   0 |   c = C
   0   0   0  20   0   3   0   0   0 |   d = D
   0   0   0   0   1   1   0   0   0 |   e = E
   0   0   0   0   0  37   0   0   0 |   f = F
   0   0   0   0   0   3  16   0   0 |   g = G
   0   0   0   0   1   2   1  57   0 |   h = H
   0   0   0   0   4   2   0   0  44 |   i = I
```

Note that the Alphabets (A,B,C,D,E,F,G,H,I) represent the classes (1-9). Also note that the Accuracy is exceptionally high, about **(94.6197 %)** and RMSE is **0.1093**. The results above also shows some predicted probabilities (Not everything could be displayed in this report).

0.7.2 Random Forest

With the hope that an Ensemble will learn a better classifier we moved towards **Random Forest**. Below is the output when the classifier was trained via **5-fold Cross Validation**

```
Random Forest.
=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances       10167               98.4316 %
Incorrectly Classified Instances       162                1.5684 %
Kappa statistic                          0.981
Mean absolute error                      0.0078
Root mean squared error                  0.054
Relative absolute error                  4.2504 %
Root relative squared error             17.8484 %
Total Number of Instances            10329

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.995     0          1         0.995     0.997      1         A
                 0.994     0.002      0.994     0.994     0.994      1         B
                 0.997     0          1         0.997     0.998      1         C
                 0.98      0.005      0.9       0.98      0.939      0.999     D
                 0.2       0          0.8       0.2       0.32       0.881     E
                 0.966     0.007      0.916     0.966     0.941      0.998     F
                 0.947     0.001      0.973     0.947     0.96       1         G
                 0.967     0.002      0.983     0.967     0.975      0.998     H
                 0.993     0.001      0.994     0.993     0.993      0.998     I
Weighted Avg.    0.984     0.001      0.984     0.984     0.984      0.999

=== Confusion Matrix ===

    a    b    c    d    e    f    g    h    i   <-- classified as
 1456    0    0    8    0    0    0    0    0 |    a = A
    0 2341    0    2    0    7    1    4    0 |    b = B
    0    0 2786    3    0    4    0    1    1 |    c = C
    0    0    0  443    1    4    1    2    1 |    d = D
    0    3    0    1    8   15    5    6    2 |    e = E
    0    4    0   16    1  690    0    3    0 |    f = F
```

```
  0    1    0    7    0   11  359    1    0 |    g = G
  0    5    0   10    0   19    3 1128    2 |    h = H
  0    0    0    2    0    3    0    2  956 |    i = I
```

Now we evaluate this Model on our Test Set (Validation set) And below are the O/Ps.
Some of the predicted probabilities are shown.

```
510        9:I        9:I          0      0      0      0      0      0      0      0.067 *0.933
  511        9:I        9:I          0      0      0      0      0      0      0      0     *1
  512        9:I        9:I          0      0      0      0      0      0      0      0     *1
  513        9:I        9:I          0      0      0      0      0      0      0      0     *1
  514        9:I        9:I          0      0      0      0      0      0      0      0     *1
  515        9:I        9:I          0      0      0      0      0      0      0      0     *1
  516        9:I        9:I          0      0      0      0      0.067  0.067  0      0.133 *0.733
  517        9:I        9:I          0      0      0      0      0      0      0      0     *1
  518        9:I        9:I          0      0      0      0      0.067  0      0      0     *0.933
  519        9:I        9:I          0      0      0      0      0      0      0      0     *1
  520        9:I        9:I          0      0      0      0      0      0      0      0     *1
  521        9:I        9:I          0      0      0      0      0      0      0      0     *1
  522        9:I        9:I          0      0      0      0      0      0      0      0     *1
  523        9:I        9:I          0      0      0      0      0      0      0      0.067 *0.933
  524        9:I        9:I          0      0      0      0      0      0      0      0.067 *0.933
  525        9:I        9:I          0      0      0      0      0.067  0.2    0      0     *0.733
  526        9:I        9:I          0      0      0      0      0      0      0      0     *1
  527        9:I        9:I          0      0      0      0      0      0.067  0      0     *0.933
  528        9:I        9:I          0      0.067  0      0      0.067  0.067  0.067  0.067 *0.667
  529        9:I        9:I          0      0      0      0      0      0      0      0     *1
  530        9:I        9:I          0      0      0      0      0      0      0      0     *1
  531        9:I        9:I          0      0      0      0      0      0      0      0.067 *0.933
  532        9:I        9:I          0      0      0      0      0.067  0.067  0.133  0.133 *0.6
  533        9:I        9:I          0      0      0.067  0      0      0      0      0     *0.933
  534        9:I        9:I          0      0      0      0      0      0.067  0      0     *0.933
  535        9:I        9:I          0      0      0      0      0      0      0      0     *1
  536        9:I        9:I          0      0      0      0      0      0.4    0      0     *0.6
  537        9:I        9:I          0      0      0      0      0      0      0      0     *1
  538        9:I        9:I          0      0      0      0      0      0      0      0     *1
  539        9:I        9:I          0      0      0      0      0      0      0      0.133 *0.867
```

And the statistics are as shown below:

```
=== Summary ===

Correctly Classified Instances         531               98.5158 %
Incorrectly Classified Instances         8                1.4842 %
Kappa statistic                          0.982
Mean absolute error                      0.0078
Root mean squared error                  0.0529
Total Number of Instances              539

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 1         0          1         1         1          1        A
                 0.976     0.002      0.992     0.976     0.984      1        B
```

```
                 0.993     0          1          0.993     0.997      1          C
                 1         0.006      0.885      1         0.939      0.999      D
                 0.5       0.002      0.5        0.5       0.5        0.998      E
                 0.973     0.004      0.947      0.973     0.96       0.999      F
                 1         0.002      0.95       1         0.974      1          G
                 0.967     0          1          0.967     0.983      0.991      H
                 1         0          1          1         1          1          I
Weighted Avg.    0.985     0.001      0.986      0.985     0.985      0.999

=== Confusion Matrix ===

   a   b   c   d   e   f   g   h   i   <-- classified as
  77   0   0   0   0   0   0   0   0 |   a = A
   0 120   0   0   1   2   0   0   0 |   b = B
   0   0 146   1   0   0   0   0   0 |   c = C
   0   0   0  23   0   0   0   0   0 |   d = D
   0   0   0   0   1   0   1   0   0 |   e = E
   0   0   0   1   0  36   0   0   0 |   f = F
   0   0   0   0   0   0  19   0   0 |   g = G
   0   1   0   1   0   0   0  59   0 |   h = H
   0   0   0   0   0   0   0   0  50 |   i = I
```

As we can see the Accuracy is exceptionally high **(98.5158 %)** and the RMSE is negligible **(0.0529)**. Thus Random Forests being Low Bias classifiers, works really well.

0.7.3 Boosted Learner

We wanted to see how Boosting may work here. So we Tried AdaBoost, with the Decision Tree(J48) as the internal predictor. The test set evaluation is as given below.

```
=== Summary ===

Correctly Classified Instances         527               97.7737 %
Incorrectly Classified Instances        12                2.2263 %
Kappa statistic                          0.973
Mean absolute error                      0.007
Root mean squared error                  0.0641
Total Number of Instances              539

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 1         0          1         1         1          1         A
                 0.984     0          1         0.984     0.992      1         B
                 0.993     0          1         0.993     0.997      1         C
                 0.87      0.004      0.909     0.87      0.889      0.999     D
                 0         0.002      0         0         0          0.968     E
                 0.973     0.012      0.857     0.973     0.911      0.999     F
                 0.947     0.004      0.9       0.947     0.923      1         G
                 0.967     0.002      0.983     0.967     0.975      0.999     H
                 1         0          1         1         1          1         I
Weighted Avg.    0.978     0.001      0.977     0.978     0.977      1
```

```
=== Confusion Matrix ===

   a   b   c   d   e   f   g   h   i   <-- classified as
  77   0   0   0   0   0   0   0   0 |   a = A
   0 121   0   0   1   1   0   0   0 |   b = B
   0   0 146   0   0   1   0   0   0 |   c = C
   0   0   0  20   0   3   0   0   0 |   d = D
   0   0   0   0   0   0   2   0   0 |   e = E
   0   0   0   1   0  36   0   0   0 |   f = F
   0   0   0   0   0   0  18   1   0 |   g = G
   0   0   0   1   0   1   0  59   0 |   h = H
   0   0   0   0   0   0   0   0  50 |   i = I
```

As we can clearly though it works really well, yet it doesn't reach Random Forests. Boosting tries to reduce variance in the small number of iterations. However, it may have been the case the generated dataset may not have too much Variance.

0.8 Negative outcomes

We could finish the project successfully and build a classifier that successfully predicts a malware to its desired class but we faced quite a few technical hurdles while doing this project. This was a learning for us and this will help us a lot in our future data mining projects.Below are some of the negative results that we encountered during the course of project.

- In the Data Cleaning process, we tried to clean data by removing the string "00 00.." from the byte file. This is because in some of the files this string denoted declaration statement and we thought that it is uniform throughout. But we later discovered that this string signified different assembly level instructions in different files and removing this string while data cleaning impacted our classification and gave poor results.
- We tried loading the entire feature set into R and creating the correlation matrix but it hanged up and did not let us do anything. Later on, we followed a different approach to remove correlation as discussed in the Feature Selection part above.
- We tried working with a really large feature set of the order of 5000 for our classification because we did not want to loose important information embedded in the features. But using so many features brought all our algorithms crashing down. So we had to optimize our feature set to reduce it in size and choose the most relevant features from this subset of feature.
- We also tried to run the Classification algorithm in R but it did not gave us good prediction and accuracy. That is when we tried doing the predictions in Weka and it gave us very good predictions as well as accuracy,
- We tried creating Opcode N-Gram pattern from the Assembly code and creating features. By this approach, we could create features but it was difficult for us to extract the relevant features from this set of features extracted from Byte and Assembly code. They were not quite consistent with each other and when we tried classifying them, we got poor prediction results.

0.9 Final Evaluation

Thus Random Forests work the best. And the feature set we worked with also seems to nearly optimal. The accuracies are a proof of that.

Data and Feature Engineering are the pivotal points of this project. Better is the feature and data generation, smoother and more efficient will the classification be. Learning and testing the Classifiers is quite straight forwards once we have an optimal set of features. However, there are a few things to note:

1. We tried a few other classifier but no one beat the Random Forest Accuracy.
2. We tried with some more features,(the features with multiple class presence)

0.10 Appendix

Tools and Packages used:

- Weka for learning and running classifiers.
- Lucene Java library for easier N-Gram extraction.
- CSV-JDBC library for querying on CSV files (train set and test set).
- MySQL database for temporary data storage and N-Gram storage.

0.11 Attachments

- Zipped source codes
- Zipped Database extracts
- Model files from Weka.

Bibliography

[1] Chad Davis. Clarifying lucene index creation, a beginner's approach.

[2] Liang Du and Yi-Dong Shen. Unsupervised feature selection with adaptive structure learning. *arXiv preprint arXiv:1504.00736*, 2015.

[3] Kaggle. Microsoft malware classification challenge (big 2015), February 2015.

[4] M.M. Masud, L. Khan, and B. Thuraisingham. A hybrid model to detect malicious executables. In *Communications, 2007. ICC '07. IEEE International Conference on*, pages 1443–1448, June 2007.

[5] Mohammad M Masud, Tahseen M Al-Khateeb, Kevin W Hamlen, Jing Gao, Latifur Khan, Jiawei Han, and Bhavani Thuraisingham. Cloud-based malware detection for evolving data streams. *ACM Transactions on Management Information Systems (TMIS)*, 2(3):16, 2011.

[6] Mohammad M Masud, Latifur Khan, and Bhavani Thuraisingham. A hybrid model to detect malicious executables. In *Communications, 2007. ICC'07. IEEE International Conference on*, pages 1443–1448. IEEE, 2007.

[7] Igor Santos, Felix Brezo, Borja Sanz, Carlos Laorden, and Pablo Garcia Bringas. Using opcode sequences in single-class learning to detect unknown malware. *IET information security*, 5(4):220–227, 2011.

[8] Matthew G Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001.

[9] Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev, and Yuval Elovici. Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, 1(1):1–22, 2012.

[10] Muazzam Ahmed Siddiqui. *Data Mining Methods for Malware Detection.* PhD thesis, University of Cetral Florida, 2008.

[11] Lei Yu and Huan Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *ICML*, volume 3, pages 856–863, 2003.