CSCI-B 565 DATA MINING

PROJECT REPORT

Computer Science
Spring, 2015
Indiana University,
Bloomington, IN

Sadana, Ankit
asadana@indiana.edu

Gamathige, Lahiru G.
lginnali@indiana.edu

May 08, 2015

Company Profile

Data Xpose

Malicious programs often aim at extracting data from the machine they are on and then leaking it to the attacker. This makes them dangerous not just for the machine they are on but also to the entire network of machines they are connected to in a company.
That's why we, at Data Xpose, pride ourselves in creating software that takes any program code and process it at the assembly level, to ensure we get all the information we need. This information lets us protect you against any malware that might be infecting your machines, employ the needed countermeasures and keep your data and information safe.

Sadana, Ankit

Ankit is a good programmer with knowledge of a variety of programming languages like C, C++ and Octave with experience in Windows Application Design using Visual Studio and C#. He has special interests in Machine Learning, Data Mining and Robotics and is expected to graduate Indiana University Bloomington in Spring 2016.

Gamathige, Lahiru G.

Lahiru is an excellent coder and is fluent in many of the coding languages needed to create Chirp, such as Java, PHP, and Apache Thrift (a language that helps power Facebook). He also has nearly ten years of experience in coding and is currently scheduled to complete a Masters of Computer Science from Indiana University in Fall 2015.

Executive Summary

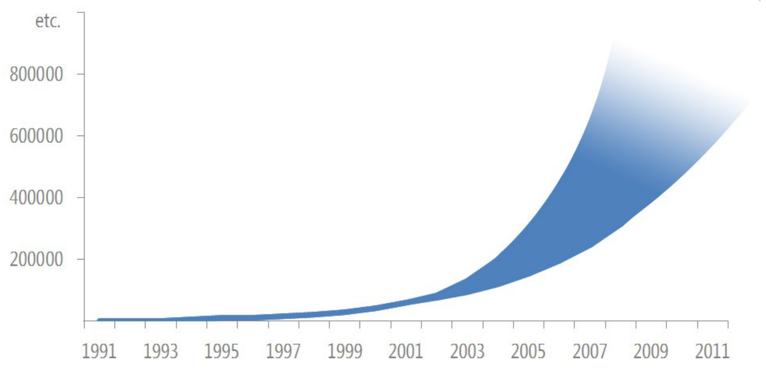
The assembly code was taken from over twenty thousand files and processed to extract the commands that were used with their frequency.
The first ten thousand files had a known type of malware, which let us learn how each malware uses commands in a pattern of frequency, also known as *n*-grams. By using this method we were able to process the remaining ten thousand files and compare them to the command patterns of the known nine malwares, allowing us to classify each one.
We now have a functional program that can process and then classify any program code as one of the nine malwares, enabling us to recommend an appropriate countermeasure on your machines. This will not only be cost effective like an antivirus software but it will provide you with a much more thorough security from malwares for your company.

Why is Malware a problem?

Malwares and Viruses have been around since the early 1970's, when the idea was to battle user written program to survive other programs in the memory area. According to Security Labs[1], This eventually led to first virus which exploited an error in Apple II computers and cause program crashes. The idea soon caught the interests of programmers, leading to experiments and finding new ways to exploit system weaknesses. What started as pranks between social groups, soon became a way to infect and alter computers over the internet.

Today, three decades later, it's no longer just the personal computers that are connected to the internet. There is gaming consoles, smart-phones, tablets and televisions besides laptops and desktop computers. With the growth and advancement in personalized computer technology, the malwares have grown as well.

This graph was generated in the Microsoft's Security Intelligence Report of Malware in the last 10 years [2], projecting the number of malware threats known to exist vs the growth since the 1990's.



As you can see, the potential malwares have grown exponentially since the year 2003.

In the current scenario, the malware do not just crash programs or pop-up message boxes saying "Surprise", they are meant to extract sensitive data or crash and delete entire systems. If this was limited to one system, companies would be able to manage it on their own but over time, malicious attackers have devised ways to spread their malware over the internet, over portable drives and even over bluetooth devices. Usually each attack on a company leads a very expensive cleanup, not to mention the loss of valuable information.

This is why our company, Data Xpose provides you with the tools you need to analyze the incoming programs and check if they are a potential malware that you need to protect yourself against. Like Benjamin Franklin said, "An ounce of prevention is worth a pound of cure".

Solutions

Current Solution

The current solution that your zoo employs is using an anti-virus software on each system. The anti-virus software though inexpensive, only scans the known virus and malware signatures. This works for all known threats that have been known to infect computer systems. But there are new threats appearing every single day. And if the anti-virus fails to account for each one, it can easily let some pass. This makes anti-virus ineffective in a larger picture.

Our Solution

We at Data Xpose, offer a completely different approach to malware protection. We understand that signature verification is very unreliable, so instead we scan each program's assembly code and observe what commands it uses. This eliminates the need to account for different programming languages since all language codes need to be translated to the assembly code before their execution. Although this process requires slightly more time than simple signature verification, but it ensures that your computers are secure against all malwares by scanning each executable program.

What we did

The Data

The data we used was part of Kaggle's Microsoft Malware Classification Challenge [3]. The training and test set each consisted of 10,868 files of assembly code, and the same number of byte files. Using a dissembler on the byte files, we were getting almost the same information as the assembler files so we focused our approach on processing the asm files.
After doing extensive research on assembly language for domain knowledge and malware classification approaches, like Statistical Structures: Fingerprinting Malware for Classification and Analysis by Bilar D.[4], Proposal of n-gram Based Algorithm for Malware Classification by Pektas A., Eris M. [5] and Acarman T. and Classification and Detection of Metamorphic Malware using Value Set Analysis by Leder F., Steinbock B. and Martini P. [6] we came to a conclusion that processing the assembly code through a method known as n-gram approach would be the best course.

n-gram model is probabilistic model that contains individual keywords like 'the' (1-gram) or a pair of sequential keywords like 'I-am' (2-gram) or 'My-name-is' (3-gram). In English language we know these words occur frequently and often in that order, this allows us to map these keywords or pairs of keywords and use it build a predicting model. In this data set, we used the same approach on assembly words like 'mov' (1-gram), 'mov-jmp' (2-gram) and 'push-push-call' (3-gram) to map commands used by each type of malware. And from our research we saw that n-gram works best between n=1,2,3 depending on the data, so we narrowed our approach to only those values of n.

This data had nine distinct types of malware class labels, which were assigned to training set files. Using the information used from the training set, we tried to classify each file in the test set.

Different Classes of Malware

There are many different types of malware, and a vast range of programs within each type. There are malwares that display ads, that spy and track activities, destroy data, record keystores and so on. And this doesn't even scratch the surface of malwares that exist.
However, for this dataset we are focusing on a small set of 9 malwares. Instead of using name of malwares

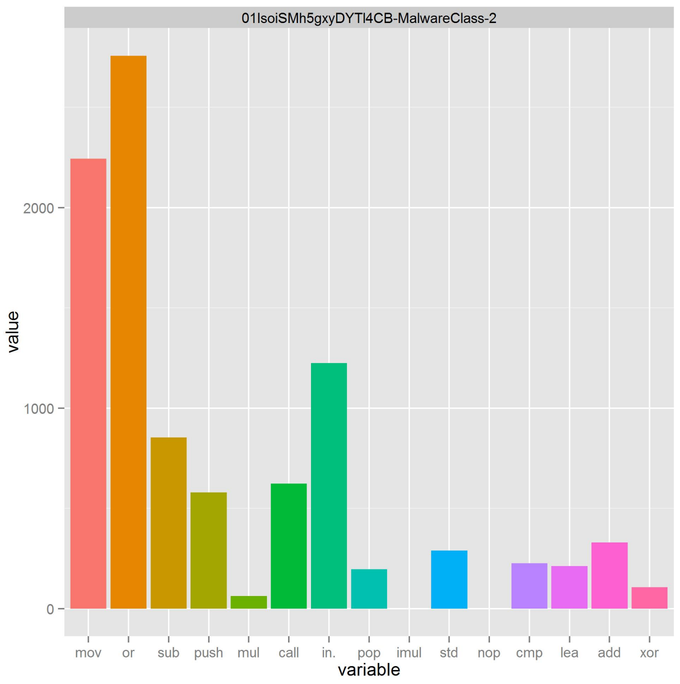
as a label, the data uses the numbers 1 to 9 for their respective malware types.

The nine malwares in this data are:

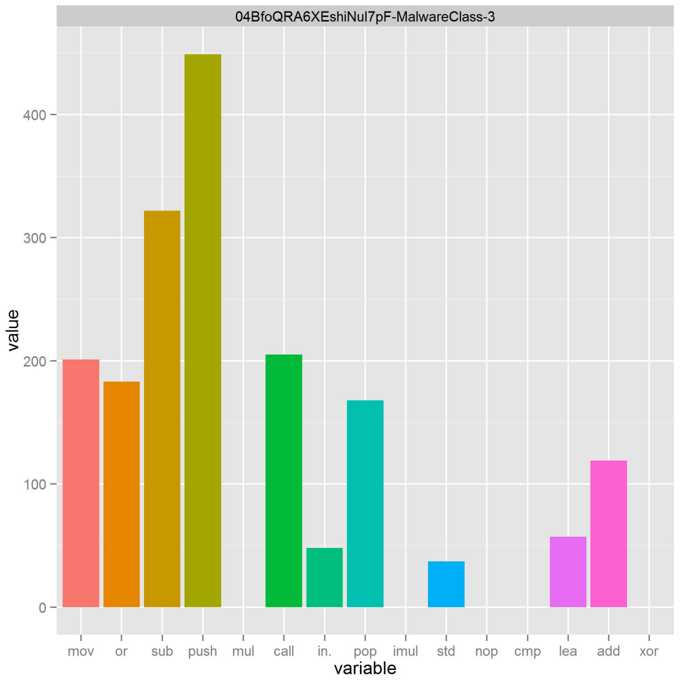
1. Ramnit
2. Lollipop
3. Kelihos_ver3
4. Vundo
5. Simda
6. Tracur
7. Kelihos_ver1
8. Obfuscator.ACY
9. Gatak

Each type of malware has a different command or opCode frequency set which lets us identify each set as one of the malware classes in the test set. The following were generated using 3 samples from our 1-gram database by using R with the help of code found on stackoverflow [7].

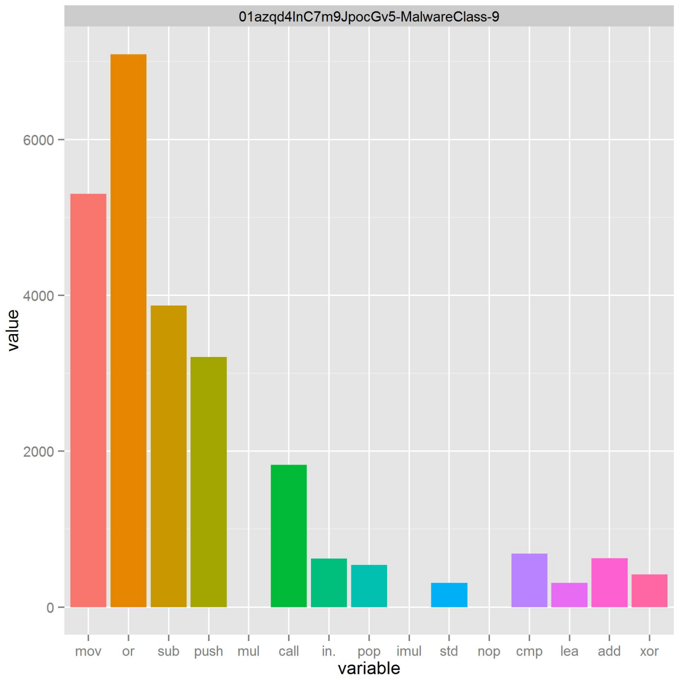
opCode frequency of training file with malware class #2.



opCode frequency of training file with malware class #3.



opCode frequency of training file with malware class #9.



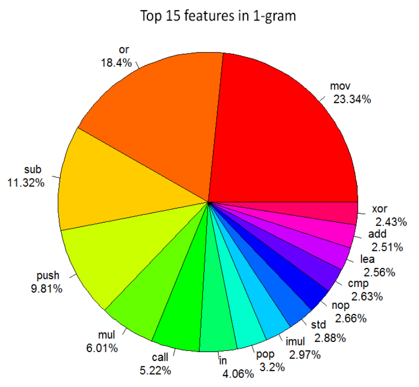
Processing and Classification

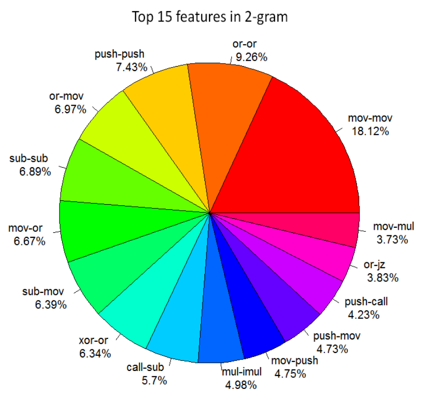
Our initial approach to the data asm files was to traverse each file and get each assembly code used in that file and the frequency with which it was used, but traversing over 10,800 files for each training and set wasn't going to be memory friendly.

Lahiru used JAVA to carefully implemented our data pre-processing module while consuming minimum memory space. He designed the program so that all asm files from the training set were read to extract assembly codes (also referred to as opCodes). Since one file at a time would take much more than a few hours, we were able to implement multiple threads running simultaneously calculating all n-grams (n=1,2,3). We kept a Map data-structure for each file's opCode count in form of a serializable structure enabling us to read the file once and saving it to three file systems for future use.
This was still an issue since a lot of the files had code worth over 20MB, so a lazy approach was implemented to do the computation cumulatively. This allowed us to process the entire data on commodity hardware in minimum amount of time.
All files were logged to observe the progress and checkpoints created to ensure we didn't have to repeat the process but fortunately we didn't encounter any memory crashes, proving that our program is fault-tolerant to a certain level.

For each file, a local opCode count was created to select the most occurring commands. These counts were then stored into csv files for each individual file in form of (opCode, frequency). With each file traversed, we also maintained a global count for all opCodes encountered. At the end of each set, this was also stored into a csv file in the same format. This was repeated for each n-gram simultaneously.
We were expecting the opCodes in the top 100 to be different in training and test sets, but to our surprise almost all the opCodes were same in both sets, and mostly in the same order or occurrence, which although unlikely, worked out to our advantage.
The code used for this pre-processing can be found in Appendix A.

Pie-charts were plotted of all 100 features to see a general trend in the frequency of the opCodes before selecting top 60. The following are projections are of the top 15 1-gram and 2-gram global counts.





Once we had generated all the n-gram csv files for both training and test data using Lahiru's JAVA program, we had to create a database for all files.
Ankit used C# to read the global opCode count file, as a reference to the commands that were observed most frequently in all files. A custom list was implemented to store and sort the opCode counts, allowing us to select the most occurring 60 features, for each n-gram.
After we had the features that we wanted, individual csv opCode files were read, and stored in another custom class with their file name, a list of all opCode counts and their label (for the training set). Each feature we had selected from the global count was then searched through the individual file and stored in the order of occurrence in the global. This ensured that the database was created in a fixed order for all files.
Once we had read and stored all the information in a class data structure, we started outputting it into a new csv file. The ourput format was kept: FileName, opCode01, opCode02 . . . opCode60, Label for the training set and the same for the test set without the Label column. Each line represented an asm file and it's respective counts for each opCode we selected delimited by a comma (,). This was repeated for each n-gram, n = 1,2,3, giving us a merged database for each n-gram that we could easily use in a classifier. The

code used for this can be found in Appendix B.

The training and test database were inputted into R to be used for the Random Forest algorithm. We used the library randomForest to build a model using Label as the class and all the opCodes as the features for all n = 1,2,3. Once the model was made, we used it to predict a probability for each class of malware. This gave us a table in the form: FileName, Class1, Class2 . . . Class9.
Therefore, for each file in the test set we were able to predict with found probability if the file would belong to a certain type of malware or not. We were able to observe that 2-gram performed slightly better than 3-gram but a lot better than 1-gram. Therefore we decided to employ 2-gram as our method of classification. The code used to use Random Forest in R with our dataset is given in Appendix C.

Strategy of Application

Now that we have an established system of identifying if a program is a certain type of malware, we can deploy this system to all computers in the zoo network. Our program will scan each executable file and analyze it's assembly code before it is executed. This will allow you to be secure for any incoming threats from malware.
As for the existing malwares, the program can scan and identify which malware is infecting your system, our program will then alert us regarding the problem and we can provide you with the appropriate countermeasures to remove the infection as fast as possible.
Our program will always be active as a background service, using minimal memory space (unless deactivated by the admin) ensuring that all executable files pass through an inspection before they have a chance to infect any of your files. And we will work with the program to make sure that you get the tools you need to counter any existing or new threats that might appear in the future, giving you peace of mind all round.

Future Works

We were able to provide you with an elegant and fault resistant system in a short amount of time. With further development, we not only intend to improve upon our initial design of our malware detection system, increasing the speed and reducing the memory usage considerably, but we also hope to deploy our preprocessing and classifier to a cloud. This cloud will get updates directly from Malware Protection Center, which keeps track of every single malware discovered around the planet. Each new malware or new program strain of an existing malware would be added to our classification system, and thereby to the proection software you have installed in your computers.
Given the opportunity, we would like to expand our program to be embedded into zoo's main network server so as to combat against malwares before they get a chance to infect any of the computers connected to the zoo. This would add an extra and much needed security layer to your company's functionality.

References

[1] Security Labs. History of malware. Retrieved on May 07, 2015 from: https://www.gdatasoftware.com/securitylabs/information/history-of-malware.

[2] Microsoft. Microsoft security intelligence report: Special edition, Feburary 2012. Retrieved on May 07, 2015 from: http://www.microsoft.com/en-us/download/details.aspx?id=29046.

[3] Kaggle.com. Microsoft malware classification challenge (big 2015). Retrieved from: https://www.kaggle.com/c/malware-classification/data.

[4] Bilar D. Statistical structures: Fingerprinting malware for classification and analysis. *Black Hat Briefings*, 2006. Retrieved from: https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Bilar.pdf.

[5] Acarman T. Pektas A., Eris M. Proposal of n-gram based algorithm for malware classification. In *SECURWARE 2011 : The Fifth International Conference on Emerging Security Information, Systems and Technologies*, 2011.

[6] Martini P. Leder F., Steinbock B. Classification and detection of metamorphic malware using value set analysis. *University of Bonn.*

[7] user20650. Single barplot for each row of dataframe, May 2014. Retrieved on May 07, 2015 from: http://stackoverflow.com/a/23973380/4756508.

A Appendix A

The following JAVA code was used to traverse each asm file in training and test set to ouput csv files with individual files with opCodes and frequencies along with one global opCode count file. This was executed for n = 1,2,3 *n*-grams.
Due to space restriction, we are only including the main file of the program. The entire code is available at: https://github.com/glahiru/Data-mining/tree/master/ms-malware-classifier

```
\\ Main.java

package com.datamining;

import com.datamining.preprocessing.ASMNGramImpl;
import com.datamining.utils.CommonUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.*;
import java.util.HashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {
    private final static Logger logger = LoggerFactory.getLogger(Main.
        class);

    public static void main(String[] args) {
        //final File file = new File("/Users/lginnali/masters/data-
            mining/Data-mining/ms-malware-classifier/data/asm");
        //final String outputPath = "/Users/lginnali/masters/data-
            mining/Data-mining/ms-malware-classifier/data/asm/output";
        final File file = new File("/media/ankit/D/data-mining-train");
        final String outputPath = "/media/ankit/D/data-mining-train/
            output";
        final int[] grams = {1,2,3};
        final int bestN = 100;

        if (file.isDirectory()) {
            File[] files = file.listFiles();
            int total = 0;
            for(File asm: files){
                if(asm.getAbsolutePath().endsWith("asm")){
                    total++;
                }
            }
            ExecutorService executorService = Executors.
                newFixedThreadPool(100);

            final int tFinal = total;
            int completedCount = 0;
            for (final File asm : files) {
                if(asm.getAbsolutePath().endsWith(".asm")) {
```

```
                final int count = completedCount++;
                for (int i = 0; i < grams.length; i++) {
                    final  int finali=i;
                    executorService.submit(new Thread(){
                        @Override
                        public void run() {
                            String absolutePath = asm.
                                getAbsolutePath();
                            String name = asm.getName();
                            ASMNGramImpl twoGram = new ASMNGramImpl
                                (absolutePath, file.getAbsolutePath
                                () + File.separator + name.split("
                                \\.")[0] + "." + grams[finali] + "
                                gram-best-"+bestN, grams[finali]);
                            twoGram.buildNGram();
                            twoGram.storeBestNCountNGramToCSV(bestN
                                , outputPath);
                            logger.info(count +"␣files␣completed␣
                                out␣of␣"+tFinal +"␣files␣!");
                        }
                    });
                }
            }
        }
        executorService.shutdown();
        try {
            executorService.awaitTermination(Long.MAX_VALUE,
                TimeUnit.NANOSECONDS);
        } catch (InterruptedException e) {
            logger.error(e.getMessage(),e);
        }
    }


    logger.info("Finished␣computing␣n␣gram␣files␣now␣computing␣best
        ␣features␣");
    File file1 = new File(outputPath);
    if (file1.isDirectory()) {
        File[] files = file1.listFiles();
        for (int i = 0; i < grams.length; i++) { // we iterate for
            each gram we want to do the computation
            HashMap<String, Integer> allCount = new HashMap<String,
                Integer>();
            for(File asm: files){
                if(asm.getAbsolutePath().endsWith(".csv") && asm.
                    getAbsolutePath().contains(grams[i]+"gram-best-
                    "+bestN)){ // we only take the best csv files
                    try {
                        InputStreamReader fileReader = new
                            InputStreamReader(new FileInputStream(
                            asm));
                        BufferedReader bufferedInputStream = new
                            BufferedReader(fileReader);
                        String line = null;


                        while ((line = bufferedInputStream.readLine
                            ()) != null) {
                            if(line.contains(",")){
                                String[] split = line.split(",");
                                if(allCount.get(split[0])!=null) {
                                    allCount.put(split[0], allCount
                                        .get(split[0]) + Integer.
                                        parseInt(split[1]));
                                }else {
                                    allCount.put(split[0], Integer.
                                        parseInt(split[1]));
                                }
                            }
                        }
                        bufferedInputStream.close();
                        fileReader.close();
                    } catch (FileNotFoundException e) {
                        logger.error(e.getMessage(), e);
                    } catch (IOException e) {
                        logger.error(e.getMessage(), e);
                    }
                }
            }
            // finished a single gram computation
            CommonUtils.storeFinalBestNCSV(grams[i], bestN,
                outputPath, allCount);
        }
    }
  }
}
```

B Appendix B

The following C# code was used to read global opCode count csv file to sort and select 60 most occurring opCodes. These were then searched in individual files to create a single database with merged vales in form of: FileName, opCode01, opCode02 ... opCode60, Label for the training set and the same for the test set without the Label column. This was repeated for each *n*-gram, *n* = 1,2,3.
The code is available at: https://github.com/asadana/CSVFileProcessor

```
// Processes CSV best-n-gram files produced for the Malware
    classification
// Program then reads labels from trainLabels.csv and creates a
    database using fileName, opCodes and Labels
// Created By: Ankit Sadana
// Created On: 04/29/2015
// Last Modified By: Ankit Sadana
// Last Modified On: 05/05/2015

using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSVFileProcessor
{
    // Class for storing data
    class DataExample
    {
        public string fileName;
        public List<uint> opCode;
        public string label;

        public DataExample(string file, List<uint> counts)
        {
            fileName = file;
            opCode = counts;
            label = "0";
        }
    }

    // Class for storing fileNames and labels from trainLabels.csv
    class fileNLabel
    {
        public string fileName;
        public string label;

        public fileNLabel(string file, string lbl)
        {
            fileName = file;
            label = lbl;
        }
    }
```

```
// Class for storing opCodes and their respective counts
class opNCount : IComparable<opNCount>
{
    public string opCode;
    public uint count;

    public opNCount(string op, uint num)
    {
        opCode = op;
        count = num;
    }

    // Overriding system function to compare and sort this class'
        list
    public int CompareTo(opNCount value)
    {
        return this.count.CompareTo(value.count);
    }
}

// Main class
class Program
{
    // Main function
    static void Main(string[] args)
    {
        string ngram = "2";

        // List to store all file names in folder
        List<string> allFileNames = new List<string>();

        // Reading all files in the directory with specific ending
        DirectoryInfo dInfo = new DirectoryInfo("..\\..\\csvFiles\\
            train");
        FileInfo[] fInfo = dInfo.GetFiles("*-" + ngram + "gram-best
            -100.csv");

        // Reading the best count n-gram csv file
        string[] bestFile = File.ReadAllLines("..\\..\\
            othercsvFiles\\best-" + ngram + "-gram-100.csv");

        // List to store opCodes and their respective counts
        List<opNCount> best = new List<opNCount>();

        foreach(string x in bestFile)
        {
            string[] temp = x.Split(',');
            best.Add(new opNCount(temp[0], Convert.ToUInt32(temp
                [1])));
        }

        // Sorting the list of opCodes in Descending order
```

```
best.Sort();
best.Reverse();

// Change the size of string array for number of features
    to be selected from the top count
// check stores all the opCodes we are using
string[] check = new string[60];
int i = 0;

foreach(opNCount bg in best.Take(check.Length))
{
    check[i] = bg.opCode;
    i++;
}


/*          // Writing top 30 into a file
           StreamWriter writer = new StreamWriter("..\\..\\
               outputCSV\\1gram-top30.csv");

            foreach(opNCount bg in best.Take(30))
            {
                writer.WriteLine(bg.opCode + "," + bg.count)
                    ;
            }
            writer.Close();
            */

// Removing the extra elements from the name
foreach (FileInfo file in fInfo)
{
    allFileNames.Add(file.Name.Replace("-" + ngram + "gram-
        best-100.csv", ""));
}

// data stores all the data
DataExample[] data = new DataExample[allFileNames.Count];
int counter = 0;

foreach(FileInfo file in fInfo)
{
    Console.WriteLine("Processing␣file#" + counter);

    string[] fileText = File.ReadAllLines(file.FullName);

    // listForFile stores all counts and opCodes
    List<opNCount> listForFile = new List<opNCount>();

    foreach (string x in fileText)
    {
        string[] temp = x.Split(',');
        listForFile.Add(new opNCount(temp[0], Convert.
            ToUInt32(temp[1])));
    }

    // count stores all the counts found in check, in the
        same order of check
    List<uint> count = new List<uint>();

    bool found;

    foreach(string a in check)
    {
        found = false;

        foreach(opNCount ops in listForFile)
        {
            // if current string in check is found in
                listForFile, it is added to the count list
            if(a.Equals(ops.opCode))
            {
                count.Add(ops.count);
                found = true;
                break;
            }
        }

        // if not found, 0 is added as a default value
        if (!found)
        {
            count.Add(0);
        }
    }

    // data is initialized with fileName and count list
    data[counter] = new DataExample(allFileNames[counter],
        count);
    counter++;
}

// Reading trainLabels.csv
string[] labelFile = File.ReadAllLines("..\\..\\
    othercsvFiles\\trainLabels.csv");

// List to store file names and their labels
List<fileNLabel> lbls = new List<fileNLabel>();

foreach (string x in labelFile)
{
    string[] temp = x.Split(',');
    lbls.Add(new fileNLabel(temp[0], temp[1]));
}

for (counter = 0; counter < data.Length; counter++)
{
    foreach (fileNLabel labels in lbls)
```

```
                {
                    // Checking file name match and adding it's
                        respective label
                    if (data[counter].fileName.Equals(labels.fileName))
                    {
                        data[counter].label = labels.label;
                    }
                }
            }

            Console.WriteLine("Writing␣to␣the␣file");

            // Writing all data to a csv file
            StreamWriter writeMe = new StreamWriter("..\\..\\outputCSV
                \\test-" + ngram + "gram.csv");

            // Writing Headers
            //string line = "";
            string line = "FileName,";

            foreach(string x in check)
            {
                line = line + x + ",";
            }

            // For test set, removing the last ','
            // line = line.TrimEnd(',');

            writeMe.WriteLine(line);

            // Reading and writing data
            foreach (DataExample de in data)
            {
                line = "";
                line = de.fileName + ",";
                foreach(uint counts in de.opCode)
                {
                    line = line + counts + ",";
                }

                // For test set, removing the last ','
                // line = line.TrimEnd(',');

                writeMe.WriteLine(line);
            }

            writeMe.Close();

            Console.ReadLine();
        }
    }
}
```

C Appendix C

We inputted the training and test database into R. Using the randomForest library we generated a Random Forest classifier with Label class and all the opCodes as the features. The program outputs a table with each FileName and probability for each malware class. This was done for all n-grams (n = 1,2,3).

```
library(randomForest);

dat <- read.csv("train-data-2gram.csv", sep=",", h=T);

test <- read.csv("test-data-2gram.csv", sep=",", h=T);

attach(dat);

rfmodel <- randomForest(Label~., dat);

predicted <- predict(rfmodel, test, type="response");

prop.table(table(test$FileName, predicted),1);
```