International Journal of Recent Technology and Engineering (IJRTE)
ISSN: 2277-3878, Volume-8 Issue-5, January 2020

Malware Detection by Risky Zone "Signature" Extraction from API Calls String Transformation using (AOSSR) Technique

Gamal A. N. Mohamed, Norafida Bte Ithnin

Abstract: ***Low accuracy of malware detection exists in malware detectors that are based on various malware representation architectures due to several problems as in case of API call graph construction and matching algorithms, where a major issue of building a precise call graph from the information collected about malware samples, also graph matching algorithms are having NP-Complete Problems and slow because of their computational complexity [1], [2]. Moreover, increasing the malware detection accuracy based on API call graphs by enhancing the graph matching and construction algorithms, experiences more computational time taken for matching process and in construction process as there are many graphs created which makes it so difficult to fetch or identify the malware (Elhadi et al., 2013) [3]. It has been further argued by (Li et al., 2018) that in case of malevolent activities, it is comparatively intricate to find whether the software will fall under the spell of malicious occurrences or not, since the duty to determine whether the conduct of a program is malicious is quite complex [4]. This research proposes enhancement of API String-based representation technique through the implementation of a malware detection framework that adopts String-based representation of the malware signature which is a compact representation of the malware risky zone where only the set of API calls representing the actual malware behaviour is accounted in the String using Absolute Order Signature String Representation technique (AOSSR) to represent the malware Strings resulting in a better performance of malware detection accuracy. The Methodology this research work follows mainly composed of three phases. The first phase deals with the conversion of the known malware samples from text format to string format. The second phase addresses the extractaction of the risky zone of an input file which is the file that needs to be checked, by the help of the file signatures already been presented in the database. The third phase addresses how to match two strings efficiency. The Application of the research is very clear as last experiment conducted on 515 malware samples demonstrates that the proposed malware detection architecture has 98% accuracy and 0 false positive rates. Comparing three families, using Analysis of Variance (ANOVA) test it is proven there is no significant difference (p>0.05) in the detection rate of algorithm across the three families of malwares. The algorithms performance is consistent.***

The result also shows that the Receiver Operating Characteristics (ROC) curves display a better True Positives Rate (TPR) for the proposed architecture over the previous attempts, which reflect significant improvement of TPR.

Keywords: Malware, Detection Techniques, API String, Risky Zone, ANOVA, AOSSR.

I. INTRODUCTION

A. Malware and Detection System

In April 2009 Symantec and GFI have published a security study which focuses on the increase of spam and malware. The development of antivirus systems is in increase process as viruses are still considered to be of great threats that can harm our application programs and systems as well. With the complexity on the malicious software and its ability to harm and infect computer systems researches have been conducted and still ongoing process by researchers on computer security field to deal with all the threats caused by this malicious software (Jaramillo, L. E. S. et al., 2019) [5]. Malware which is basically stands for malicious software, and it could be simply defined as such software that is designed by programmers with strong intent to harm the computer system or the information or both. As a software it has many forms that it can come or been created into, these forms are such as Viruses, Worms, Trojan horses, Backdoors, Spyware, Rootkits, botnet in addition to other types of software with unwanted behavior (Anderson et al., 2018). Hence basically a virus is a malware that does self-replication into other existing files or programs that could be executed and this action is repeated by the infected file for other uninfected files to make the virus spread within the same computer system or even can shift between various computers through usage of infected external media such as USB, CD/DVD, Floppy Disk and so on (Ranum and Gula, 2019) [6]. In polymorphic virus, to evade detection of the decryption routine in an encrypted virus, malware writers began to morph the decryption code. These viruses are said to be polymorphic. The code to decrypt malware is morphed after every infection. Morphing is done in such a way that the internal structure of co de changes completely without any change in the functionality. In Metamorphic Virus the internal structure of the malware gets changed after every execution with its overall functionality remains the same [7]. They are shown in Fig.1 & Fig2. Below:

Manuscript published on January 30, 2020.
* Correspondence Author
Gamal A. N. Mohamed*, Faculty of Computing, University Technology Malaysia, Skudai, Johor Bahru, Malaysia. E-mail: gamal.utm@gmail.com
Department of Computing, Muscat College, Muscat, Sultanate of Oman. E-mail: gamal@muscatcollege.edu.om
Dr. Norafida Bte Ithnin, Faculty of Computing, University Technology Malaysia, Skudai, Johor Bahru, Malaysia. E-mail: afida@utm.my

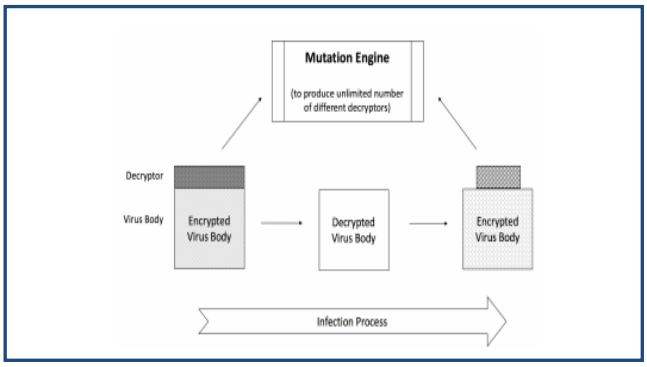

Fig.1. PolymorphicVirus Structure (Dhanasekar et al., 2018)

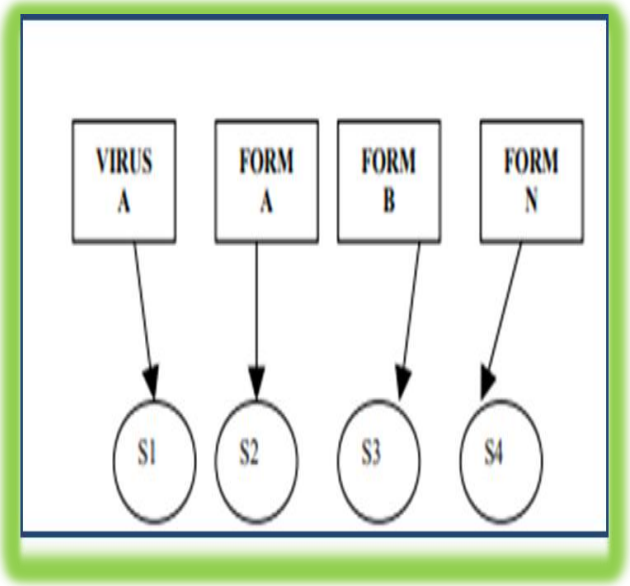

Fig.2. Metamorphic Viruses (Broeck et al., 2019)

B. Detection Definitions, Types and Stages

It's obviously well known that the malware writers or cyber offenders otherwise known as hackers, applying some certain sophisticated techniques to evade detection following the methods that will modify or morphing the malware by means of using several packing techniques and/or program obfuscation and in this regard there are certain common obfuscation techniques, here comes the concept of Malware detector as a system that attempts to identify malware using both signatures and other heuristics techniques; the Antivirus scanner for example is a good example of a malware detector however, in the antivirus scanner there are possibilities of occurrence of false negative where a virus fails to detect the infected file (Sogukpinar, 2019) [8]. The Signature-Based is a sequence of instructions unique to a malware used to generate a malware signature, (this signature is captured by researchers in a laboratory environment), a signature should be able to identify any malware exhibiting the malicious behaviour specified by the signature and most antivirus scanners are signature based. Whereas the Behaviour-Based detection techniques focus on analyzing the behaviour of known and suspected malicious code. Such behaviors include factors such as the source and destination addresses of the malware, the attachment types in which they are embedded and statistical anomalies in malware infected systems [9].The below Fig3. Explains the different detection techniques:

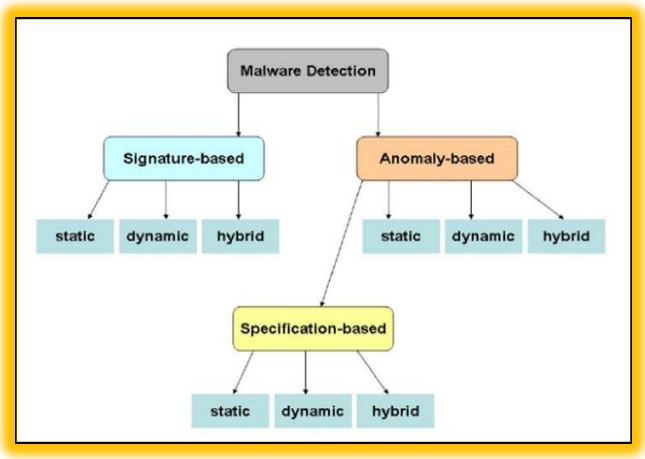

Fig.3. Instance of malware detection John and Thomas (2019)

Each of them could be implemented using any of the three methods namely static, dynamic and hybrid [8]. In this research paper we will present our proposed initial phases for detection model which is based on a String (Signature) representation technique aims at representing the malware file in text format (input space) into string format (Feature space) following Absolute Order Signature String Representation technique (AOSSR). In absolute order signature, we order the number of API calls alphabetically according to their API string and then extract the signature accordingly and then we find a value that represent the similarity of the two strings using String matching algorithm (see phases 1,2 & 3) that summarize our whole process of designing this malware detection technique using AOSSR.

II. RELATED WORK AND THE RESEARCH GAP

In order to emphasize and analyze the real problem this research is addressing, a gap related to various techniques been used to detect the malware on the previous researches that have been using the graph representation techniques (Ammar A. E. E. Elamin et al., 2014 and Sharma et al., 2016; Jazi, and Ghorbani, 2016; Narayanan et al., 2016), data mining techniques (e.g., Hou et al., 2016; Barhoom and Nasman 2016; Bhattacharya and Goswami, 2017) as well as machine learning techniques (e.g., Narudin et al., 2016; Jerlin and Marimuthu, 2017; Chen and Bourlai, 2017). To detect the malware and clear issues been observed which could be summarized into three parts; 1st related to the nature of constructing the graph matching algorithm as it experienced difficulty in building a precise API Call graph from information collected about malware samples. 2nd is more major as it relates to a clear NP-Complete problems and computational matching time complexity. 3rd is related to the nature of the graph itself, since its composed of nodes and edges then a lot of parameters and functions are needed to be represented which will certainly results into a lot of memory locations been used hence a major problem of wastage in the memory usages could be clearly observed. All of these can help in identifying the real gap which is clearly shown and explained through the below Fig. 4:

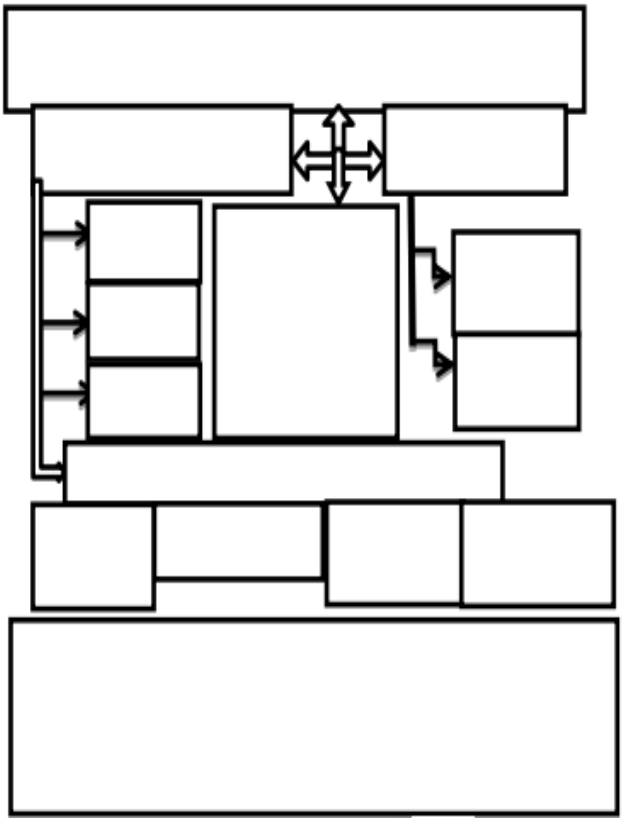

Fig.4. Scenario leading to the research problem.

III. METHODOLOGY

The methodology addresses the proposed framework that we have designed following the AOSSR method as it consist of mainly three different phases and they explain the overall logical flow of the proposed detection system. The first phase deals with the conversion of the known malware samples which was in text format to be converted to the string format. The second one addresses the extractaction of the risky zone of an input file which is the file that needs to be checked, by the help of the file signatures already been presented in the database. The third one addresses how to match two strings efficiency. This will work by providing the two strings those need to be checked for their matching and applying the dynamic programming to get a value that represent the similarity of the two strings. These three main phases are explained below with their corresponding algorithms that are been developed and used in the implementation part.

A. Phase 1: From Text to Database

The main driver algorithm is the one who has been used to convert the files in text format which is represented as a set of API calls and insert them into the database. In addition to this algorithm, there is other two other algorithms that transform the API call into a set of signatures they have been shown with logical relation to the proposed method used with the AOSSR. These signatures are used afterwards to recognize the risky zone of the file that need to be checked.

Input: filename
Output: List of API calls
readFile () *{* *open the file "fileName"* *if (the file is not open)* *{* *return* *}* *while (we did not each end of the file)* *{* *line=read next line for the file* *extract the function call string* *add the function name to the list* *}* *Close the file* *Return the list of the API calls* *}*
Read API Calls from a file

Fig.5. Main Driver Algorithm

The above Figure3 shows the Input which is a set of files representing the sample dataset, whereas the Output is a database filled with the string representation of the sample database. For each sample in the sample dataset perform:

1. Read the file line by line
2. Construct a signature of the file
3. Divide the whole signature into a smaller strings limited by the database field maximum size
4. Insert a record for this file
5. Insert a record for each signature string and link it with the file record via database keys.

For the file signature, we employ Absolute Order Signature String Representation technique (AOSSR) to evaluate its effectiveness. We assumed only the provided sample data set contains only the API calls related to risky zone. This can be extracted by experts in the provide malware files. Furthermore, only one of these signature types is required to perform a check. We list Absolute Order Signature String Representation technique (AOSSR) here to evaluate their performance. Also other methods can be used for this phase as part of our framework.

B. Absolute Order Signature String Representation (AOSSR) Technique

In absolute order signature, we order the number of API calls alphabetically according to their API string and then extract the signature according to the below shown order.

Input: a file with API calls, m: the max length of the signature
Output: a string representing the file signature according to absolute order based technique

1. For each line in the input file, Extract the API function call
2. Sort the set of API calls according to the API call string
3. Combine the API call string in the order of Step 2
4. Extract the first m characters of the combined string

The above steps are basically been implemented using the transformation algorithm shown below as the list of API calls will be taken as an input transforming them into string representing the file signature according to absolute order based technique. The logical sequence of the transformation algorithm that includes extracting the API function call then Sort the set of API calls according to the API call string then Combine the API call string in the order previously done then finally Extract the first m characters of the combined string. This is clearly explained through the transformation algorithm shown below:

Input: A list of API calls
Output: A string representing the file signature according to absolute order based technique
Sort the list of API calls alphabetically excluding repetition *let signature=""* *while(not end of the call list){* *nextCall=extract the next call on the top of the hash* *signature = signature + nextCall* *}* *return signature;*
Absolute Order Signature String Technique

Fig.6. Transformation Algorithm

C. Phase 2: Risky Zone Extraction

When techniqueing this phase the conversion from text to string is done through main driver algorithm and the sginature is extracted through the transformation algorithm. Here comes the stage where we have to implement the extraction of the risky zone of an input file (which need to be checked), by the help of the file signatures in the database. Then we perform the logical sequence for identfying the input and obtaing the output. As the input will be a file with API calls, that need to be checked. Whereas the output will be a risky zone of the input file. This is further explained as per the Risky Zone extraction algorithm shown below that implement this second phase of our sting representation technique framework. The below steps explain in a detailed manner the major steps of the implemented Risky Zone extraction Algorithm:

Input: a file with API calls, that need to be checked

Output: a risky zone of the input file

1. Create the signature of the input file according to one of the methods described in phase 1
2. For each file in the database

The real logical form of the Risky Zone extraction Algorithm is shown as per the algorithm in the below Fig.7:

Input: : a file with API calls, that need to be checked
Output: a risky zone of the input file
Create the signature of the input file according to one of the methods described in phase 1 *Get a list of file list* *while (not end of the files){* *let nextFile= next file in the file list* *for each signature related to nextFile{* *let nextSignature=the next signature of this file* *try to match nextSignature with the input file signature* *if(nextSignature is matched){* *add nextSignature to the risky zone of the input file* *}* *}* *}* *return the total risky zone*
Risky Zone extraction

Fig.7. Risky Zone extraction Algorithm

D. Phase3: File Checker Using Dynamic Programming

For this purpose, we use a dynamic programming to match two strings efficiency. Here the input will be two strings those we have identified and they need to be checked for the matching possibility, whereas the output will be a value that represents the similarity of the two strings. Here for the similarity some values ranges between 25% and 50% of threshold are been used in the implementation of this matching and detecting process. Overall, the file checker using dynamic programming will work as per the following logical sequence:

1. Input: Two strings those need to be checked for their matching.

2. Output: a value that represent the similarity of the two strings.

This is further used in the detection process as the percentage of similarity shows the various values for the TP, FP, TN, FN, Accuracy and Detection & False Alarm Rates.

The real logical form of the String matching Algorithm is shown as per the Algorithm3 in the below Fig.8:

Input: two strings: str1 and str2, those need to be checked for their matching
Output: a value that represent the similarity of the two strings
1. let lenStr1 and lenStr2 be the length of str1 and str2, respectively *2. Create a 2D array called lenCommonStr of size lenStr1+1 that will hold the common substrings of the two strings* *3. Initially, set the values of lenCommonStr to zero* *4. for i=0 to lenStr1* *5. for j=0 to lenStr2* *6. if i=0 or j=0* *7. set lenCommonStr[i][j]=0* *8. else if (str1[i - 1] == str2[j - 1])* *9. lenCommonStr[i][j] = lenCommonStr[i - 1][j - 1] + 1;* *10. result = max(result, lenCommonStr[i][j]);* *11. else* *12. lenCommonStr[i][j] = 0;* *13. Return result*

Fig.8. String Matching Algorithm

IV. RESULTS CALCULATIONS

For proposed architecture, for any given input to it there are four possible outcomes. If the input is malware and the architecture classified as malware, it is counted as a true positive (TP); if it is classified as Benign, it is counted as a false positive (FP). If the input is Benign and it is classified as Benign, it is counted as a true negative (TN); if it is classified as malware, it is counted as false negative (FN) this is clearly explained as in Table 1 of the two-by-two confusion matrix.

Table- I: A Two-By-Two Confusion Matrix

Tested Class	True Class	
	Malware	***Benign***
Malware	True Positives	False Positives
Benign	False Negatives	True Negatives

A. Equations of Calculations [10]

A calculation of Detection Rate, False Alarm Rate, and Accuracy are obtained from these TP, FP, FN and TN. Description and equations are discussed below:

Detection Rate

$$\text{Detection Rate} = \frac{TP}{TP + FN} \quad (1)$$

False Alarm Rate

$$\text{False Alarm Rate} = \frac{FP}{FP + TN} \quad (2)$$

Accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (3)$$

V. EXPERIMENTAL RESULTS AND ANALYSIS

For designing and implementing a Malware Detection Technique this proposed research work results into reaching an accurate and fast malware detection system that reduces the drawbacks observed in various malware detection techniques specially that used API Call Graph representation by enhancing Absolute Order Signature String Representation technique (AOSSR). This is done through implementation of the various algorithms shown above and the dynamic programming tools been utilized to come out with the final research outcomes which are clearly illustrated as shown in the below results and their discussion.

A. Dataset for Implementation

Computer security corporations do have an extensive malware collection, but unfortunately, they do not share their malware databases for research purposes. In this study, the malware dataset has been downloaded from http://www.nexginrc.org/ web site; these malware samples are obtained from a publicly-available database called 'VX Heavens Virus Collection'. This dataset is free to download and has been used by many researchers (Ahmed et al., 2009; Tabish et al., 2009; Mehdi et al., 2010; Ammar A. E. E. Elamin et al., 2014). Such a dataset has been utilized by a multitude of researchers as of the latest (Basu, Sinha, Bhagat, 2016). Recently it has been used by (Mahrin et al., 2018; Bist, 2018; Touli, 2019). In this dataset the malware samples, which are used as inputs, are preprocessed using the API monitor tool to extract API calls and their parameters (OSRs). Figure 7 below shows a part of an extracted API call from a bigger malware sample file; each line presents an API call with other fields separated with "*" mark.

```
explorer*0x644*IsBadReadPtr*lp:0xDEECF8,
ucb:0x10*0x0*SUCCESS*0
explorer*0x644*HeapAlloc*hHeap:0x90000,
dwFlags:0x0, dwBytes:0x280*0x1162A0*SUCCESS*0
explorer*0x644*HeapFree*hHeap:0x90000, dwFlags:0x0,
lpMem:0x1162A0*0x116501*SUCCESS*0
explorer*0x644*HeapAlloc*hHeap:0x90000,
dwFlags:0x8, dwBytes:0x434*0x128AB8*SUCCESS*0
explorer*0x644*HeapAlloc*hHeap:0x90000,
dwFlags:0x0, dwBytes:0x38*0xF5DA0*SUCCESS*0
explorer*0x644*HeapFree*hHeap:0x90000, dwFlags:0x0,
lpMem:0xF5DA0*0xF5901*SUCCESS*0
explorer*0x644*HeapFree*hHeap:0x90000, dwFlags:0x0,
lpMem:0x128AB8*0x1*SUCCESS*0
explorer*0x644*IsBadReadPtr*lp:0xDEEF50,
ucb:0x10*0x0*SUCCESS*0
explorer*0x644*IsBadReadPtr*lp:0xDEE8B4,
ucb:0x10*0x0*SUCCESS*0
```

Fig.9. Loaded Malware in Text Format (Part of extracted API call from malware)

For the Sting representation for the above sampled loaded malware in text format which shows the exact extracted part of API call of that malware, if we use Absolute Order representation on this example, then we can get the following simple string shown below:

If we use the absolute order string representation on this example, then the API orders will be:

HeapAlloc; HeapFree; IsBadReadPtr; IsBadWritePtr; LocalAlloc; LocalFree;

This is because if we sort the unrepeated API called alphabetically hence we will get them in the above shown order.

B. Risky Zone Detection of Submitted Malware

When we submit a file for checking we try to match this file with any of the files on our database. We do this step by the help of matching the risky zone of the submitted file. In our representation the risky zone will be a string. The following Figure10 illustrates the risky zone a file to our demo program:

```
HeapAlloc;HeapFree;IsBadReadPtr;IsBadWritePtr;Loca
lAlloc;LocalFree;IsBadStringPtrW;RegQueryValueExW;
RegSetValueExW;GetCurrentThread;RegOpenKeyExW;R
egCloseKey;GlobalAlloc;RegOpenKeyExA;RegEnumKey
W;GetCurrentThreadId;RegQueryValueW;GlobalLock;G
lobalReAlloc;GlobalUnlock;GlobalSize;GlobalFree;Get
DriveTypeW;GetCurrentProcessId;GetFileAttributesW;R
egEnumValueW;GetEnvironmentVariableW;GetFullPath
NameW;LoadLibraryA;GetProcAddress;LoadLibraryEx
W;CreateProcessW;GetLongPathNameW;GetThreadPrio
rity;SetThreadPriority;OpenProcess;SearchPathW;GetM
oduleFileNameW;RegCreateKeyExW;
```

Fig.10. Rep. of Text Malware in String Format-(Risky Zone A File To Our Demo Program-*Trojan.Win32.AddUser.c.txt*)

Above shown is a pinpointing of risky zone. It is a real signature showing the risky zone for the malware name **XTrojan.Win32.AddUser.c.txt** which belongs to Trojan Family where we sort the API calls according to their number of occurrence and then get a string representation applying the AOSSR technique. This is clearly shown as we implement our malware checker program to obtain the above risky zone of the file. This is shown as in the below Figure11:

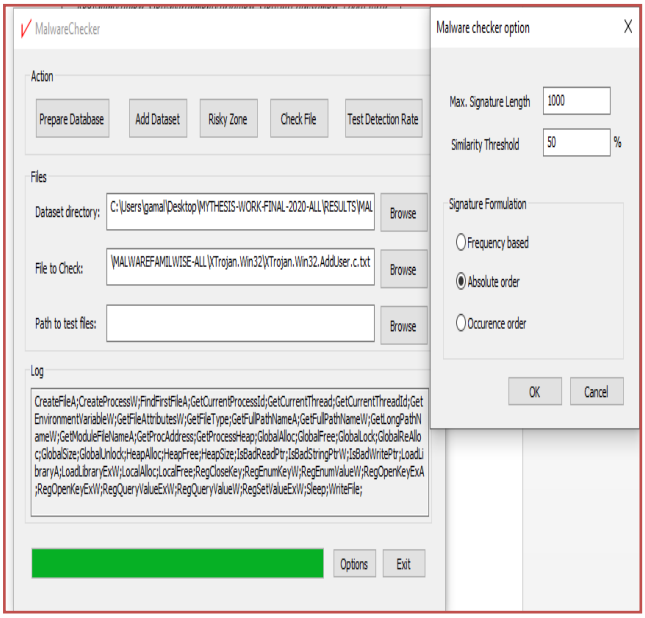

Fig.11. Malware XTrojan.Win32.AddUser.c.txt signatuere extraction (Risky Zone)

VI. RESULT AND DISCUSSION

A. Enhancing the String Matching Algorithm

To determine a threshold value that will allow the architecture to classify the sample as either malware or benign the similarities between all samples in the dataset are calculated, for example, for each malware sample its similarities with all other samples will been calculated and the higher similarity value is recorded. Then the threshold value will be determined according to that. A threshold value of 50% has been determined as a measure of similarity to classify a sample as either malware or benign and has been used during the experiment. Any sample whose similarity to a malware sample is more than or equal to the threshold, has been classified as malware. The approach taken by the presented strategy greatly contributes to finding similarity between provided samples and malware samples and overcomes other approaches. As can be seen in Table II, the proposed string-based strategy manages to detect similarity with samples taken from different malware families, namely, Trojan horse, Virus, and Worm. The Table shows that, based on the threshold, normal LCS has failed to find similarity for one of the samples, which is the Virus and Trojan, whereas the proposed string-based strategy using the matching algorithms used succeeds with all provided samples and with high rate unlike call graph.

Table- II: Similarity Value for Some Malware Samples
*** When compared to XVirus dataset**
**** When compared to XTrojan dataset**

Sample	Graph-based	LCS	Proposed String-Based
XTrojan.Win32.Sweet.apm	67.1	48.29	100*
XVirus.Win32.BingHe-ee.apm	61.45	14.61	63
XWorm.Win32.Bymer.a.apm	66.77	56.73	100**

To summarize this, a proposed string-based strategy is used to reduce the computation complexity of the matching process. Then, the proposed string-based strategy calculates the similarity using Longest Common Subsequence (LCS) algorithm. The proposed string-based strategy has been implemented and the experimental results that are conducted on a malware dataset show that the strategy has 98% detection rate and 0% false positive rates.

B. Detecting Of Obfuscated Malware

For this experiment, bellow shown results are obtained out of malware checker implementation as all dataset samples were grouped into three; Virus group, Worm group and Trojan group and the architecture runs for each group separately. For this run, the similarity threshold was kept as 50% for all the samples from the three different families and then matching checking is done which produced a matching percentage of 100% that proves the high efficiency of our matching algorithm used. Hence the selected samples are shown in Figures 12 & 13 also in Tables III to V shown below:

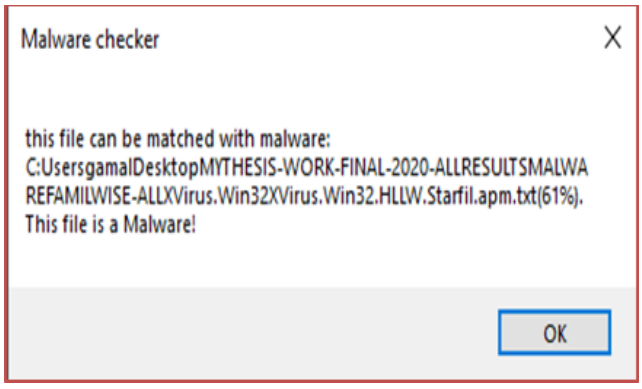

Fig.12. Malware Checker (Malware Similirty) A File To Our Demo Program- *XTrojan.Win32.AddUser.c.txt*

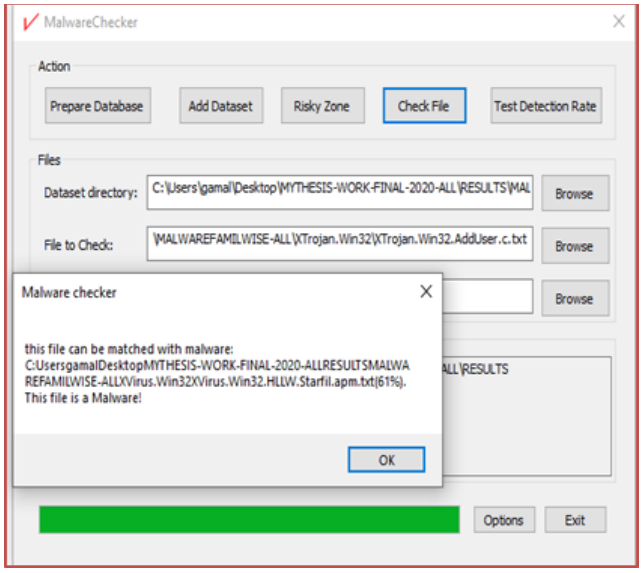

Fig.13. Malware Checker (Similarity Checker)

Table-III shows the various single files of worm family that are been checked against all virus family with a threshold = 50% and shows matching percentage ranges from 60%, 80%, 90% & 100% also the risky zone for each file has been displayed. Table-IV shows the various single files of Virus family that are been checked against all Trojan family with a threshold = 50% and shows matching percentage ranges from 65%, 70%, 80%, 90% & 100% also the risky zone for each file has been displayed. Table-V shows the various group files of Trojan family that are been checked against all Virus family with a threshold = 50% and shows matching percentage ranges from 65%, 85%, 90% & 100% also the risky zone for each file has been displayed. The analysis shown in tables III, IV & V proves that the proposed string based representation design has shown efficiency in the matching algorithm used.

C. Detecting Malware Variation

In this experiment, there were three groups of tests: Virus vs. benign group, Worm vs. benign group and Trojan vs. benign group, and the system ran for each group separately. Also the results of previous experiments were included to generate the following comparison. The results of this experiment were illustrated in Tables VI and VII shown below. We have then conducted another experiment and run the malware checker for checking malware familywise against each other and also against the Whole dataset of Malware samples with (Similarity Threshold = 50% - Using Absolute Order Signature String Representation technique (AOSSR), we have obtained values shown in the below Tables VIII, IX & X and Figures 14, 15 & 16. In the below Table VIII, It is very clear that when the detection test has been performed using the Absolute Order Signature String Representation (AOSSR) with the similarity threshold 50%, it shows 100% detection rate for all the worm samples checked against samples of Trojan & Virus and with the whole malware dataset. Various different outputs for the True Positive Rate (122, 118 & 134) and the False Positive Rate (13, 18 & 1) for all with both False Negative and True Negative set to 0.0 with the False Alarm Rate set to 1.0 in all.

The Accuracy for the first two tests i.e. against Virus and Trojan samples and against the whole malware dataset is shown as (90.370%, 86.667% & 99.259% simultaneously).

Table-III: Single Malware Samples Worm Family Matched with VRIUS Family (Similarity Threshold=50%-Using AOSSR).

Malware Name	Risky Zone displayed? Y=1 N=0	Malware name matched	Matching %
XWorm.Win32.Chainsaw.a.apm.txt	1	XViurs.Win32.HLLC.Asive.apm.txt	100%
XWorm.Win32.FlyVB-ee.apm.txt	1	XViurs.Win32.HLLP.Alcaul.g.apm.txt	87%
XWorm.Win32.Foxma.apm.txt	1	XViurs.Win32.HLLC.Asive.apm.txt	100%
XWorm.Win32.Fozer-ee.apm.txt	1	XViurs.Win32.HLLP.VB.c.apm.txt	63%
XWorm.Win32.Hai.apm.txt	1	XVirus.Win32.Crypto.apm.txt	100%
XWorm.Win32.Listas-ee.apm.txt	1	XViurs.Win32.HLLP.VB.c.apm.txt	94%
XWorm.Win32.Nautical.apm.txt	1	XViurs.Win32.HLLP.Text.a.apm.txt	66%
XWorm.Win32.Naxe-ee.apm.txt	1	XViurs.Win32.HLLP.Nilop-ee.apm.txt	85%
XWorm.Win32.Onver.apm.txt	1	XViurs.Win32.HLLC.Asive.apm.txt	100%

Table-IV: Single Malware Samples VRIUS Family Matched TROJAN Family (Similarity Threshold = 50%-Using AOSSR)

Malware Name	Risky Zone displayed? Y=1 N=0	Malware name matched	Matching %
XVirus.Win32.Akez-ee.apm.txt	1	XTrojan.Win32.Aldy-ee.apm.txt	85%
XVirus.Win32.Crypto.apm.txt	1	XTrojan-PSW.Win32.Gip.112.apm.txt	100%
XVirus.Win32.Butter.apm.txt	1	XTrojan-PSW.Win32.Ghostar.50.apm.txt	93%
XVirus.Win32.Auryn.1155-ee.apm.txt	1	XTrojan.Win32.Aldy-ee.apm.txt	85%

XVirus.Win32.Belial.2609-ee.apm.txt	1	XTrojan.Win32.Adder-ee.txt	67%
XVirus.Win32.Ditex-ee.apm.txt	1	XTrojan.Win32.Akuan-ee.apm.txt	100%
XVirus.Win32.Ditto.1488-ee.apm.txt	1	XTrojan.Win32.Aldy-ee.apm.txt	85%
XVirus.Win32.Doser.4187.apm.txt	1	XTrojan.Win32.Aname-ee.apm.txt	73%
XVirus.Win32.Flechal.apm.txt	1	XTrojan.Win32.Aname-ee.apm.txt	100%

Table-V: Single Malware Samples in TROJAN family Matched VIRUS family(SimilarityThreshold=50%-Using AOSSR)

Trojan Malware Name	Risky Zone displayed? Y=1 N=0	Malware name matched	Matching %
XTrojan.Win32.AddShare.a-ee.txt	1	XVirus.Win32.FunLove.4070.apm.txt	88%
XTrojan.Win32.AddShare.e-ee.txt	1	XVirus.Win32.FunLove.4070.apm.txt	69%
XTrojan.Win32.Alcalup.b.apm.txt	1	XVirus.Win32.HLLP.Alcaul.g.apm.txt	94%
XTrojan.Win32.AntiBTC.c.apm.txt	1	XVirus.Win32.Dream.4916.apm.txt	100%
XTrojan.Win32.Aspam.a.apm.txt	1	XVirus.Win32.FunLove.4070.apm.txt	85%
XTrojan.Win32.AVPatch.c.apm.txt	1	XViurs.Win32.HLLO.Bubica.apm.txt	100%
XTrojan.Win32.Batman.a.apm.txt	1	XViurs.Win32.HLLC.Trafix-ee.apm.txt	100%

Table-VI: Detection Rate of Methods Used in Experiments for Different Malware Families

Methods (Detection Rate)	Virus	Trojan	Worm
N-gram	0.9455	0.8103	0.8881
Lee	1	0.9397	0.9701
Call Graph	1	0.9569	0.9701
Our Proposed Method String-based representation (*)	0.998	0.989	0.984

* Based on the average rate with similarity of 25%

Table-VII: Accuracy between the methods used in experiment for different malware Families

Methods (Accuracy)	Virus	Trojan	Worm
N-gram	0.9658	0.8930	0.9353
Lee	**1**	0.9628	0.9828
Ahmed	0.9900	0.9700	**0.9840**
Call Graph	**1**	0.9721	0.9828
Our Proposed Method String-based representation	**1**	**0.9842**	**0.9812**

Findings: Our proposed model shown high level of accuracy across all the above models using same dataset that are compared with; we consider the three malware families i.e. Virus, Trojan and Worm. The highest accuracy was obtained by our proposed architecture, Call Graph and Lee method in the Virus group (i.e. 1), our proposed architecture obtained a higher accuracy in the Trojan group (i.e., 0.9842) while ours and Ahmed method shared obtaining the highest accuracy in the Worm group (i.e., 0.9812 and 0.9840 respectively).

Table-VIII: Worm Family Vs Virus & Trojan Families & Malware Dataset (Similarity Threshold = 50% Using AOSSR)

Worm Samples Checked?	Checked against?	No. of Samples Checked Against?	Similarity Threshold	DR	TP	FP	FN	TN	FAR	ACC
135	Virus	165	50%	100%	122	13	0	0	1	90.370%
135	Trojan	117	50%	100%	118	18	0	0	1	86.667%
135	Malware Dataset	417	50%	100%	134	1	0	0	1	99.259%

Table-IX: Virus Family Vs Trojan & Worms Families & Malware Dataset (Similarity Threshold = 50% - Using AOSSR)

Trojan Samples Checked?	Checked against?	No. of Samples Checked Against?	Similarity Threshold	DR	TP	FP	FN	TN	FAR	ACC
117	Virus	165	50%	100%	88	29	0	0	1	85.21%
117	Worm	135	50%	100%	85	32	0	0	1	72.65%
117	Malware Dataset	417	50%	100%	117	0	0	0	*	100%

Table-X: Trojan Family Vs Virus & Worms Families & Malware Dataset (Similarity Threshold = 50% - Using (AOSSR)

Virus Samples Checked?	Checked against?	No. of Samples Checked Against?	Similarity Threshold	DR	TP	FP	FN	TN	FAR	ACC
165	Trojan	117	50%	100%	129	36	0	0	1	78.182%
165	Worm	135	50%	100%	135	30	0	0	1	81.818%
165	Malware Dataset	417	50%	100%	165	0	0	0	*	100%

*FAR- False Alarm Rate shown as indefinite Result. Trojan have ability to hide themselves hence most dangerous types of malware which can spread and attack their targets with dificulty in detecting them by normal malware scanners or detectors. Our proposed method has proven this and considered while designing our proposed string based representation model for detecting unknown malware files.

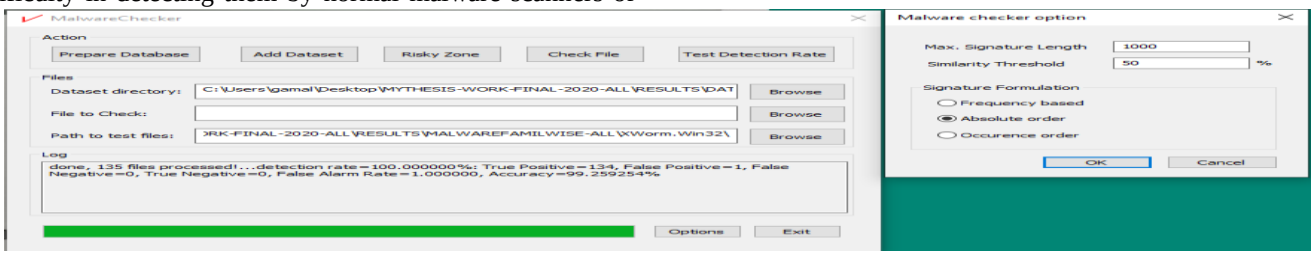

Fig.14. Worm Vs Whole Malware Dataset

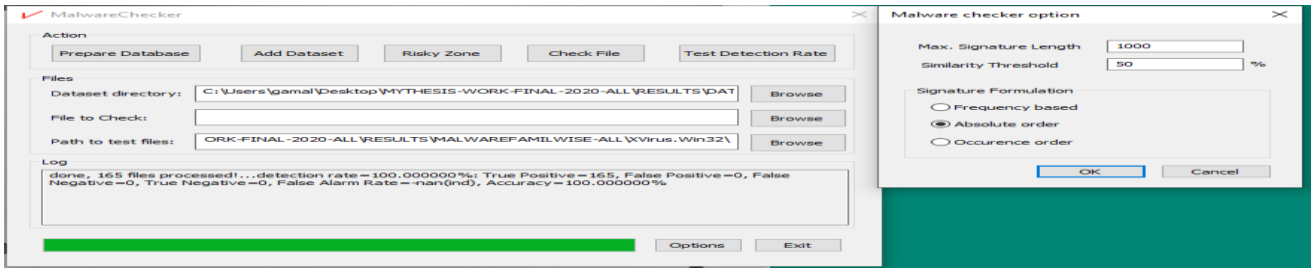

Fig.15. Virus Vs Whole Malware Dataset

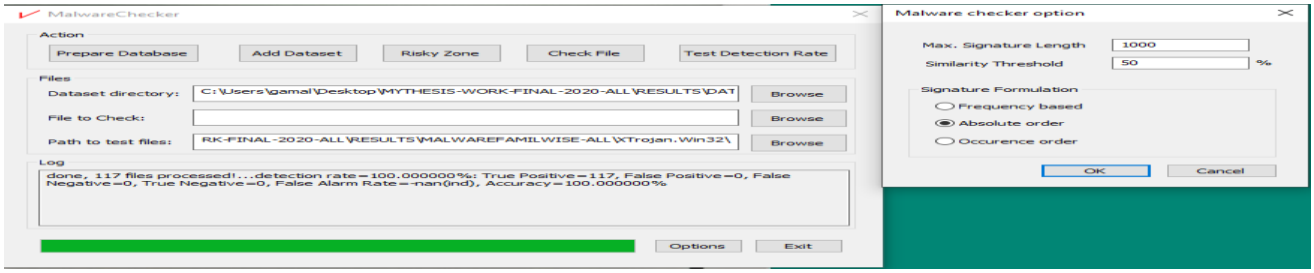

Fig.16. Trojan Vs Whole Malware Dataset

In the above Tables IX & X, It is very clear that when the detection test has been performed using the Absolute Order Signature String Representation (AOSSR) with the similarity threshold 50%, it shows 100% detection rate for all the checked samples of Virus Vs Worm & Trojan also the Trojan samples Vs Virus & Worm and with the whole malware dataset. Various different outputs for the True Positive Rates (129, 135 & 165) and the False Positive Rates (36, 30 & 0) in Table IX and TPR (88, 85 &117) FPR(29, 32 & 0) in Table X with both False Negative and True Negative set to 0.0 and the False Alarm Rate set to 1. Expect when checked against the whole malware dataset the False Positive set to 0.0 with the False Alarm Rate set to *. The Accuracy for the first tests i.e. Virus against Trojan and Worm samples in Table IX are shown as (78.182%, 81.818% & 100% simultaneously). The Accuracy for the second tests i.e. Trojan against Virus and Worm samples in Table X are shown as (85.213%, 72.649% & 100% simultaneously).

D. Detection Rate Results Analysis

The below Table XI shows the details for different samples used in this study for the sake of malware detection analysis that produces the results which is further been used in the analysis, whereas Table XII shows the detection rates obtained. In Table XIII since we are comparing three families, using Analysis of Variance (ANOVA) test the below result's analysis could be summarized as per the below tables (final findings as per Table XIII).

Table XI: Samples Details Information

Samples Used In This Research Test		
Malware Type	**No. of Families**	**No. of Samples**
XTrojan	15	34
XWorm	19	88
XVirus	21	71

Table XII: Detection Rate

Sample	Virus Family	Trojan Family	Worm Family
1	91	97	99
2	97	98	98
3	91	98	91
4	98	100	98

Table XIII: Detection Rate Analysis

ENOVA SINGLE FACTOR						
SUMMARY						
Groups	**Count**	**Sum**	**Average**	**Variance**		
Virus	4	377	94.25	14.25		
Trojan	4	393	98.25	1.583		
Worm	4	386	96.5	13.66		
ANOVA						
Source of Variation	**SS**	**df**	**MS**	**F**	**P-value**	**F crit**
Between Groups	32.1667	2	16.08	1.636	0.2478	4.256
Within Groups	88.5	9	9.833			
Total	120.667	11				

[F (2, 9) =1.636]

Findings: There is no significant difference ($p>0.05$) in the detection rate of algorithm across the three families of malwares. The algorithms performance is consistent.

VII. CONCLUSION

As a conclusion in this research paper we have shown how we found the malware signature using Absolute Order Signature String Representation technique (AOSSR). We have implemented the transformation algorithm and then we use the signature obtained to extract the risky zone of the input file to be checked as malicious code or not. In our developed system we have used the dynamic programming to detect the unknown malware in a very fast and accurate manner reducing the NP-Complete problem, less the complexity in the matching time & saved the usage of memory space.

APPENDIX

Raw data i.e. the code for the setting manager used in this study is shown as Appendix-A in the last page no. 10 of this research paper.

REFERENCES

1. Anderson, Blake & Quist, Daniel & Neil, Joshua & Storlie, Curtis & Lane, Terran. (2011). Graph-based malware detection using dynamic analysis. Journal in Computer Virology. 7. 247-258. 10.1007/s11416-011-0152-x.
2. Kostakis, Orestis & Kinable, Joris & Mahmoudi, Hamed & Mustonen, Kimmo. (2011). Improved call graph comparison using simulated annealing. Proceedings of the ACM Symposium on Applied Computing. 1516-1523. 10.1145/1982185.1982509.
3. Elhadi, Ammar & Maarof, Mohd & Barry, Bazara. (2013). Improving the Detection of Malware Behaviour Using Simplified Data Dependent API Call Graph. International Journal of Security and Its Applications. 7. 29-42. 10.14257/ijsia.2013.7.5.03.
4. Ze Li, Duoyong Sun, Bo Li, Zhanfeng Li, and Aobo Li, "Terrorist Group Behavior Prediction by Wavelet Transform-Based Pattern Recognition," Discrete Dynamics in Nature and Society, vol. 2018, Article ID 5676712, 16 pages, 2018. Available: https://www.hindawi.com/journals/ddns/2018/5676712/
5. Jaramillo, L. E. S. Malware Threats Analysis and Mitigation Techniques for Compromised Systems. Journal of Information Systems Engineering Management, 4(1), em0087. Available: https://doi.org/10.29333/jisem/5742 2019.
6. Ranum, M.J. and Gula, R., Tenable Inc and Tenable Network Security Inc, 2019. System and method for strategic anti-malware monitoring. U.S. Patent Application 16/200,812 – 2019.
7. Dhanasekar, Dhiviya & Di Troia, Fabio & Potika, Katerina & Stamp, Mark. (2018). Detecting Encrypted and Polymorphic Malware Using Hidden Markov Models: An Artificial Intelligence Approach. 10.1007/978-3-319-92624-7_12.
8. Kakisim A.G., Nar M., Carkaci N., Sogukpinar I. (2019) Analysis and Evaluation of Dynamic Feature-Based Malware Detection Methods. In: Lanet JL., Toma C. (eds) Innovative Security Solutions for Information Technology and Communications. SECITC 2018. Lecture Notes in Computer Science, vol 11359. Springer, Cham
9. Mohamed, G.A.N. and Ithnin, N.B., 2017. Survey on representation techniques for malware detection system. Am. J. Appl. Sci, 14(11), pp.1049-1069.Available: - https://thescipub.com/abstract/10.3844/ajassp.2017.1049.1069 2017
10. Mohamed, G.A. and Ithnin, N.B., 2017, April. SBRT: API signature behaviour based representation technique for improving metamorphic malware detection. In International Conference of Reliable Information and Communication Technology (pp. 767-777). Springer, Cham.-April 2017.

AUTHORS PROFILE

Gamal Abdel Nassir Awad Ali Mohamed received the degrees Bachelor & Master in Computer Science from Baharathidasan University, India in 1993 & 1997 respectively. He is a Ph.D. research student of University Technology Malaysia. Currently, he is a teaching faculty in Department of Computer Science at Muscat College, Muscat- Sultanate of Oman which is affiliated with the University of Stirling in the UK. His research interests include Information and Communication Security, Malware Prevention and Detection System. He is associated with the Information Assurance & Security Research Group (IASRG) at UTM.

Dr. Norafida Bte Ithnin received her Bachelor in Computer Science (Computer System) from (UTM), KL-Malaysia. Master of Information Technology (Computer Science) from (UKM), Bangi, Malaysia, Ph.D. (in Computation) from University of Manchester, Institute of Science and Technology (UMIST), Manchester, UK. Currently, she is an Associate Professor at Faculty of Computer Science, UTM-Malaysia. Her research interests include Security Management, Unified Threat Management, Information Hiding & Forensics, Network, Data and Computer Security. She is also a member of (IASRG) at UTM.

APPENDIX-A

The following chapter contains the setting manager for the main malware checker raw data used for the experiment.

A.1 Setting Manager C/C++ Raw Data

```
#include "stdafx.h"
#include "settingmanager.h"
int SettingManager::getSignatureLen() {
    CWinApp* pApp = AfxGetApp();
    return pApp->GetProfileInt(_T("MalwareChecker"), _T("SigLen"), 0);
}
void SettingManager::setSignatureLen(const int& len) {
    CWinApp* pApp = AfxGetApp();
    if (len > 1000 || len <= 0) {
        pApp->WriteProfileInt(_T("MalwareChecker"), _T("SigLen"),
1000);
    }
    else {
        pApp->WriteProfileInt(_T("MalwareChecker"), _T("SigLen"), len);
    }
}
void SettingManager::setSignatureLenStr(const CString& len) {
    int val= std::stoi(convert(len));
    setSignatureLen(val);
}
std::string SettingManager::convert(const CString& cstring) {
    // Convert a TCHAR string to a LPCSTR
    CT2CA pszConvertedAnsiString(cstring);
    // construct a std::string using the LPCSTR input
    std::string strStd(pszConvertedAnsiString);
    return strStd;
}
int SettingManager::getSignatureMethod() {
    CWinApp* pApp = AfxGetApp();
    return pApp->GetProfileInt(_T("MalwareChecker"), _T("SigMethod"),
0);
}
void SettingManager::setSignatureMethod(const int& method) {
    CWinApp* pApp = AfxGetApp();
    pApp->WriteProfileInt(_T("MalwareChecker"), _T("SigMethod"),
method);
}
void SettingManager::setSignaturemethodStr(const CString& method) {
    int val = std::stoi(convert(method));
    setSignatureMethod(val);
}
int SettingManager::getSimilarityThreshold()
{
    CWinApp* pApp = AfxGetApp();
    return pApp->GetProfileInt(_T("MalwareChecker"),
_T("simThreashold"), 25);
}
void SettingManager::setSimilarityThreshold(const int& sim) {
    CWinApp* pApp = AfxGetApp();
    pApp->WriteProfileInt(_T("MalwareChecker"), _T("simThreashold"),
sim);
}
void SettingManager::setSimilarityThreshold(const CString& sim) {
    int val = std::stoi(convert(sim));
    setSimilarityThreshold(val);
}
```

A.2 Setting Manager Header File

```
#pragma once

#include <string>

class SettingManager {
public:
    SettingManager() {}

    static int getSignatureLen();
    static void setSignatureLen(const int& len);
    static void setSignatureLenStr(const CString& len);

    //similarity threshold
    static int getSimilarityThreshold();
    static void setSimilarityThreshold(const CString& len);
    static void setSimilarityThreshold(const int& len);

    static int getSignatureMethod();
    static void setSignatureMethod(const int& len);
    static void setSignaturemethodStr(const CString& len);

protected:

    static std::string convert(const CString& cstring);
private:
};
```