Detection of Anomalous In-Memory Process based on DLL Sequence

Binayak Panda[1], Dr. Satya Narayan Tripathy[2]
PG. Dept. of Computer Science
Berhampur University, Berhampur
Odisha, India

Abstract*—The use of Computer systems to keep track of day to day activities for single-user systems as well as the implementation of business logic in enterprises is the demand of the hour. As it plays a vital role in making available information on one click as well as impacts improvement in business and influences the profit or loss. There is always a possible threat from unauthorized users as well as untrusted or unknown applications. Trivially a host is intended to run with a list of known or trusted applications based on user's preference. Any application beyond the trusted list can be called as untrusted or unknown application, which is not expected to run on that host. Untrusted applications becomes available to a host from sources like websites, emails, external storage devices etc. Such untrusted programs may be malicious or non-malicious in nature but the presence must be detected, as it is not a trusted program from user's view point. All such programs may target the system either to steal valuable information or to decrease the system performance without the knowledge of the user of the system. Antimalware vendors provide support to defend the system from malicious programs. They do not include users trusted program list in to consideration. It is also true that new instances of attacks are found very frequently. Hence there is a need for a system which can be self-defending from anomalous activities on the system with reference to a trusted program list. In this paper design of an "Anomalous In-Memory Process detector based on the use of the DLL (Dynamic Link Library) sequence" is proposed, which does accountability of trusted programs intended to run on a particular host and create a knowledgebase of classes of processes with TF-IDF (Term Frequency-Inverse Document Frequency) multinomial logistic regression based learning approach. This knowledgebase becomes useful to map a suspected In-memory process to a class of processes using loaded DLL's of it. With a cross-validation approach, the suspected process and processes of its predicted class are used to conclude whether it is a trusted, variant of the trusted or untrusted process for that host. Not necessarily the untrusted program is a malware but it may be a program not listed in the trusted program list for the specific host. Hence this work aims to detect anomaly in concern with list of trusted applications based on user's preference by doing a dynamic analysis on In-memory processes.

***Keywords*—Anomalous In-memory Process; dynamic analysis; DLL hijacking; DLL injection; TF-IDF multinomial logistic regression**

I. INTRODUCTION

In the 21st century use of computers is becoming quite obvious in all fields, starting with the banking sector, education sector, health sector, e-commerce, etc. The use of computers is not only limited to such big domains; but also are extended to be used by individuals in their home's, small offices, and various goods' retail counters to keep track of their day to day activities. Whether large commercial sectors or small retail counters or individual use of computers increases day by day with the availability of Internet facilities.

On the contrary, the risk of the exploitation of data and information kept on computers also increases day by day because of the exposure of computers to the outside world due to internet connectivity. There are intelligent programmers, who somehow put a piece of code (a small program which is unknown or untrusted) on a computer of interest with an intention of either stealing or misusing the data kept on computers or making computers non operable. Such programs are referred to as malware or potentially unwanted application (PUA). There exist many categories of such malware like viruses, worm, spyware, adware, ransomware, etc. The adverse effect of the presence of malware on a computer system scales from a very small impact to an extremely large impact. PUA do not have any specific types as they seem to be normal programs but there may a possible threat due to the presence of them.

Quick heal annual threat report 2019 says that prediction of becoming vicious about ransomware happened to be true in 2018. Only in one month, the ransomware detection reached 2Million in 2018. Also, the prediction about small and medium-sized businesses to be in the red zone became true. Cryptojacking is a new buzzword suppressing ransomware, which is a process of using someone's computer to earn money. The only sign of a computer used for cryptojacking is a little slower computer performance while executing programs. The CPU is targeted up to 100% by cryptojacking which leads to hardware faults slowly. The owner of the compromised computer becomes unaware of being a victim of cryptojacking. The report says about detection of more than 800k cases of cryptojacking in 2018. It also has given information about an Infector named W32.Pioneer.CZ1, which injects the files on to disk and then decrypts the malicious DLL present in the file and drops it to do malicious activities. "Fig. 1" shows the frequency of attack of various malware types per day, per hour, and per minute referring to Quick heal threat report 2019 [1]. Internet security threat report, Symantec 2019 says 69 million events detected in 2018 which is 4 times to cases in 2017. The report also speaks about PUA which is not necessarily harmful but may lead to security risks. The existence of such PUA may also result in Host-

Based exploits. It also says about cases of Cryptojacking and its consequences. The report mentioned about supply chain attacks which target third party software by injecting code into its libraries. These libraries are integrated into larger software projects. Injection of code in to libraries can be understood as DLL injection, which is a possible approach of exploiting a host [2].

The case of infector W32.Pioneer.CZ1, supply chain attack and possibility of threat due to the presence of PUA points out a need of a system for real time detection of host based exploitations. Antimalware vendors do provide support in detecting malicious programs with signature based static analysis but they don't take users preference in to consideration. Hence some unwanted or unknown programs referring to users preferred list left undetected. Such unknown programs may be a malware or PUA, which is a possible threat to that host. These observations motivated to apply multi-class classification approach on known or trusted processes using their respective list of loaded DLL's on a host considering its users preferred list of programs. This knowledge helps in detecting a deviation from known In-memory processes which is either a malfunctioned known process or unknown process or untrusted process using some potentially unwanted DLL or malfunctioned DLL.

The organization of the remaining sections in this paper is as follows: Section 2 speaks about the related works on malware analysis considering In-memory processes and injection of unwanted DLL's. Section 3 speaks about the design of the System in detecting anomalies or deviations with respect to In-memory processes and their respective loaded DLL's. It is about designing an Anomalous In-Memory Process detector based on the use of DLL's, which learns the trusted programs intended to run on a particular host and creates multiple class of them referring their usage of DLL's. With a cross validation approach a suspected process gets validated with processes of a class it is mapped to and gets detected as either trusted or variant of trusted or untrusted for the specific host. Section 4 speaks about the experimental setup for the empirical evaluation of the system. Section 5 describes the concluding remark of the work.

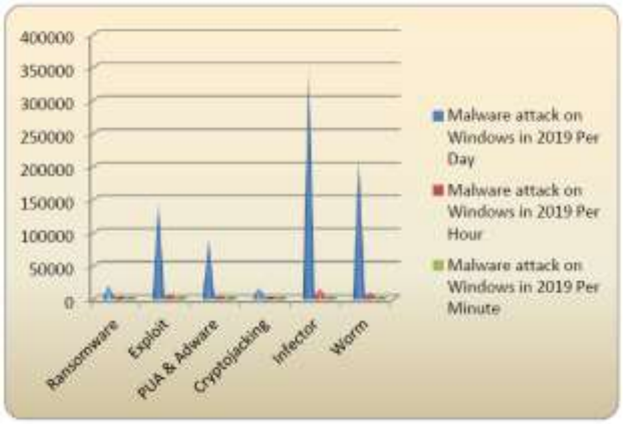


Fig. 1. Malware Attack on Windows in 2019.

II. Related Work

Analysis of the behavior of unknown programs like PUA, malware etc. is becoming truly diversified. Various forms of analysis are done on a system to identify a threat to the information stored on the computer. The analysis can be in the form of identification of untrusted programs available on secondary storage or anomalous In-memory processes. The approaches of analysis can be said as either static or dynamic or hybrid or memory-based [3]. The static analysis considers opcode's, N-gram opcode sequences, control flow graph as features to analyze further without executing the programs. The dynamic analysis considers function calls, API calls, function parameters, instruction traces, and instruction flows as features to analyze further after executing the programs [4]. The hybrid analysis is a combination of static analysis and dynamic analysis [5]. The memory-based analysis is also a kind of dynamic analysis that considers network connection information, changes in registry keys and In-memory processes and there DLL sequences for further analysis during the execution of programs [6, 7, 8]. With run time attributes of benign process using string analysis for anomaly detection in Android operating system is found effective [9]. Studying the behavior of malware is becoming popular with memory forensic techniques for malware injection and hidden processes [10]. DLL injection is a process where the malicious DLL gets injected on to an In-memory process and the control of execution gets transferred to that code block [11]. Reflective DLL injection has also gained popularity where they do malicious activities in memory only without leaving any footprint [12, 13].

A Windows application uses DLL files during runtime to load libraries. It tries to locate the DLL with a hierarchy of searches. First, it tries to find with the given path. But when it fails to locate, it searches at some predefined set of directories. Malware programs breaches this search order to load malicious DLL during run time. In this context, DLL-Side loading is becoming a very popular method for attacking Windows systems [14]. In such cases, the malware payload places the spoofed malicious DLL into a specific location so that the spoofed DLL gets loaded instead of legitimate DLL. Such DLL-Side loading bypasses the signature-based static analysis process. This DLL load order hijacking process to load a malicious DLL in run time can also be referred to as DLL hijacking. A variant of such an approach where a malware launcher loads the malicious DLL compromising a victim processes memory whereby loads the malicious DLL by creating a thread. Such an approach of entry of malicious DLL onto to system is referred to as DLL Injection. With this approach, the program loads unintended DLL's due to the presence of side-loading vulnerability of Windows side-by-side manifests [15].

Typically when malware attacks, it makes available its payload physically on the system storage and gets loaded on to memory to do the malicious activity. In such cases either the traditional static analysis using signature-based detection becomes helpful or the dynamic analysis considering the various run-time behaviors of processes becomes helpful. But Fileless malware has become a new possible attack type, where the malware is not saving the payload on system storage rather it malfunctions the trusted and legitimate processes of the Operating System. It injects the malicious program directly on to the compromised processes memory without dropping any file to the file system. As no physical file presents the Sandbox detection approach fails. Again as there is no possibility of having a signature, hence the signature detection also fails. Hence the detection complexity becomes too high for Fileless malware. The possibility of investigating Fileless malware is only limited to analysis of the behavior of the system using the snapshots of In-memory processes, which is considered here as Memory based analysis [16]. Information retrieval theory is applied with a dynamic analysis to extract API calls and system calls to classify malicious programs. They are stored in documents on which the TF-IDF weighting approach is applied to get a good accuracy of malware classification [17].

In this paper, a novel approach is proposed considering memory based dynamic analysis of In-memory processes in identifying any deviation from the trusted process list of a particular host. The DLL lists of In-memory processes are taken in to consideration for deciding a suspected process as either trusted or variant of trusted or untrusted for a specific host. The list of trusted in-memory processes are classified in to multiple classes considering the DLL sequences they use at various instances. A suspected process gets mapped in to one of the trusted class of processes based on its DLL sequence. For this multi-class classification DLL lists are formed as attribute vectors with Vector Space Model (VSM), on which TF-IDF multinomial Logistic regression is used to train the system. Objective of training process is to prepare a knowledgebase of classes of processes, which are considered as known or trusted and legitimate processes from the viewpoint of a particular user. This system can take an In-memory process at any random instance of time and do a prediction of its class using the learned knowledge base. The cosine similarity metric is used to cross-validate a suspected process with all the processes of the predicted class before concluding it as either a trusted or variant of a trusted or untrusted process for that specific host. In this work a list of processes are declared as trusted processes from the user's regular use viewpoint. A variant of a trusted process is understood as a process of an updated version application from trusted list. Any other process other than a trusted or a variant of trusted is understood as untrusted.

III. System Design

A. System Overview

The anomalous In-memory process detection system can be divided into three parts: data preprocessing, the process class prediction model of the system, and cross-validation of the predictors result. Data preprocessing is about collecting the DLL sequences loaded for all In-memory processes with reference to a given list of trusted applications of the specific host, using Windows Sysinternals Process utilities like Pslist.exe and Listdlls.exe [18]. Pslist.exe shows information about In-memory processes. Listdlls.exe shows the list of DLL's loaded for a specific process at that time instance. A TF-IDF weight matrix gets generated defining weight of each DLL in the collected DLL sequences for the list of In-memory processes. The said system applies multinomial classification on the data set of In-memory processes to classify them in to multiple classes of processes based on DLL sequences as feature vector. The process class prediction model is trained and tested using the generated data set. For the training and testing phase of the system multinomial Logistic Regression, multinomial Naive Bayes, and Support Vector Classifier (SVM-SVC for multiclass problem) mechanisms are used. The training phase of the model uses the approach of learning the usual activity of a host from In-memory processes and their respective DLL sequences to create the knowledge base of processes as multiple classes. The testing phase of the model uses the knowledge gained in the training phase, to decide accuracy of the system in classifying In-memory processes to their class. Cosine similarity measure is used for Cross Validation of the predictors result. With cosine similarity the DLL sequence of a suspected In-memory process is compared with DLL sequences of processes of the predicted class to verify the similarity of the suspected process and subsequently to say whether the process is a trusted, variant of trusted or untrusted.

B. Data Preprocessing

There are various run time attributes of an In-memory process, which speaks about the behavior of it. Some attributes are process path, process name, process priority, number of threads, number of handles, private virtual memory, path of all the DLL's loaded, etc. In this system, the focus is given on two run time attributes namely path of the process and path of all the DLL's loaded. Pslist.exe is used to collect all the process names and their respective process ids. "Fig. 2" represents a sample output which is a list of elements a.k.a. *In_Memory_Process_List* where each element is a 2-tuple say *Process_tuple* (*pname,pid*) containing process name and process id for all the In-Memory processes at a particular time instance.

Listdlls.exe is used to collect all the DLL's loaded on to the memory for each element of the In_Memory_Process_List at that time instance. "Fig. 3" represents the absolute path of the program corresponding to one of the In-memory process and the absolute path of all its loaded DLL's. There will be a list of such records based on the number of In-memory processes at that time instance. Let that be referred as a Database DLL List a.k.a. DBDLLList[].The algorithmic steps for generating a collection of DBDLLList's at various time instances is explained in Algorithm 1 which is a.k.a. IMPDLLList.

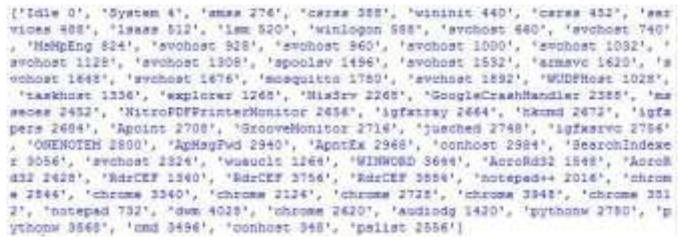
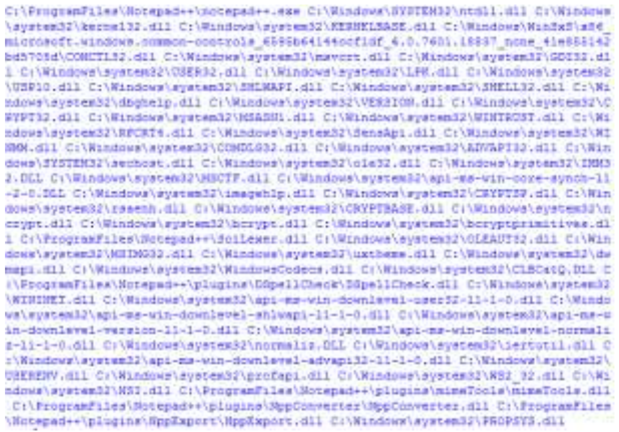

```
['Idle 0', 'System 4', 'smss 276', 'csrss 388', 'wininit 440', 'csrss 452', 'ser
vices 488', 'lsass 512', 'lsm 520', 'winlogon 588', 'svchost 640', 'svchost 740'
, 'MsMpEng 824', 'svchost 928', 'svchost 960', 'svchost 1000', 'svchost 1032', '
svchost 1128', 'svchost 1308', 'spoolsv 1496', 'svchost 1532', 'armsvc 1620', 's
vchost 1648', 'svchost 1676', 'mosquitto 1780', 'svchost 1892', 'WUDFHost 1028',
 'taskhost 1336', 'explorer 1268', 'MisSrv 2268', 'GoogleCrashHandler 2388', 'ms
seces 2432', 'NitroPDFPrinterMonitor 2456', 'igfxtray 2664', 'hkcmd 2672', 'igfx
pers 2684', 'Apoint 2708', 'GrooveMonitor 2716', 'jusched 2748', 'igfxsrvc 2756'
, 'ONENOTEM 2800', 'ApMsgFwd 2940', 'ApntEx 2968', 'conhost 2984', 'SearchIndexe
r 3056', 'svchost 2324', 'wuauclt 1264', 'WINWORD 3644', 'AcroRd32 1648', 'AcroR
d32 2628', 'RdrCEF 1340', 'RdrCEF 3756', 'RdrCEF 3884', 'notepad++ 2016', 'chrom
e 2844', 'chrome 3340', 'chrome 2124', 'chrome 2728', 'chrome 3948', 'chrome 351
2', 'notepad 732', 'dwm 4028', 'chrome 2620', 'audiodg 1420', 'pythonw 2780', 'p
ythonw 3568', 'cmd 3496', 'conhost 348', 'pslist 2556']
```

Fig. 2. In_Memory_Process_List at a Time Instance.



```
C:\ProgramFiles\Notepad++\notepad++.exe C:\Windows\SYSTEM32\ntdll.dll C:\Windows
\system32\kernel32.dll C:\Windows\system32\KERNELBASE.dll C:\Windows\WinSxS\x86_
microsoft.windows.common-controls_6595b64144ccf1df_6.0.7601.18837_none_41e855142
bd5705d\COMCTL32.dll C:\Windows\system32\msvcrt.dll C:\Windows\system32\GDI32.dl
l C:\Windows\system32\USER32.dll C:\Windows\system32\LPK.dll C:\Windows\system32
\USP10.dll C:\Windows\system32\SHLWAPI.dll C:\Windows\system32\SHELL32.dll C:\Wi
ndows\system32\dbghelp.dll C:\Windows\system32\VERSION.dll C:\Windows\system32\C
RYPT32.dll C:\Windows\system32\MSASN1.dll C:\Windows\system32\WINTRUST.dll C:\Wi
ndows\system32\RPCRT4.dll C:\Windows\system32\SensApi.dll C:\Windows\system32\WI
NMM.dll C:\Windows\system32\COMDLG32.dll C:\Windows\system32\ADVAPI32.dll C:\Win
dows\SYSTEM32\sechost.dll C:\Windows\system32\ole32.dll C:\Windows\system32\IMM3
2.DLL C:\Windows\system32\MSCTF.dll C:\Windows\system32\api-ms-win-core-synch-l1
-2-0.DLL C:\Windows\system32\imagehlp.dll C:\Windows\system32\CRYPTSP.dll C:\Win
dows\system32\rsaenh.dll C:\Windows\system32\CRYPTBASE.dll C:\Windows\system32\n
crypt.dll C:\Windows\system32\bcrypt.dll C:\Windows\system32\bcryptprimitives.dl
l C:\ProgramFiles\Notepad++\SciLexer.dll C:\Windows\system32\OLEAUT32.dll C:\Win
dows\system32\MSIMG32.dll C:\Windows\system32\uxtheme.dll C:\Windows\system32\dw
mapi.dll C:\Windows\system32\WindowsCodecs.dll C:\Windows\system32\CLBCatQ.DLL C
:\ProgramFiles\Notepad++\plugins\DSpellCheck\DSpellCheck.dll C:\Windows\system32
\WININET.dll C:\Windows\system32\api-ms-win-downlevel-user32-l1-1-0.dll C:\Windo
ws\system32\api-ms-win-downlevel-shlwapi-l1-1-0.dll C:\Windows\system32\api-ms-w
in-downlevel-version-l1-1-0.dll C:\Windows\system32\api-ms-win-downlevel-normali
z-l1-1-0.dll C:\Windows\system32\normaliz.DLL C:\Windows\system32\iertutil.dll C
:\Windows\system32\api-ms-win-downlevel-advapi32-l1-1-0.dll C:\Windows\system32\
USERENV.dll C:\Windows\system32\profapi.dll C:\Windows\system32\WS2_32.dll C:\Wi
ndows\system32\NSI.dll C:\ProgramFiles\Notepad++\plugins\mimeTools\mimeTools.dll
 C:\ProgramFiles\Notepad++\plugins\NppConverter\NppConverter.dll C:\ProgramFiles
\Notepad++\plugins\NppExport\NppExport.dll C:\Windows\system32\PROPSYS.dll
```

Fig. 3. One Record of DBDLLList[].

Algorithm 1: ***IMPDLLList***

Require: Pslist.exe and Listdlls.exe are expected to be present locally. They are to be invoked with administrative rights.

Ensure: *DBDLLList[]* A data set consisting of records of all the In-memory processes, where each record will be the absolute path of executable of the In-memory process and absolute path of all the DLL's loaded at that particular time instance. Fig. 3 represents one record of *DBDLLList[]*.

```
1. In_Memory_Process_List = Os.System(Pslist.exe)
2. Initialize DBDLLList[] = NULL
3. For each Process_tuple in In_Memory_Process_List
   a. Process_Id =Process_tuple[1]
   b. Temp_DLL_List=Os.System(Listdlls.exe, Process_Id)
   c. Temp_DLL_List=Prune.out('\s+',' ',Temp_DLL_List)
   d. DBDLLList[].append(Temp_DLL_List)
4. Return DBDLLList[]
```

From "Fig. 3" related to one of the records of DBDLLList[], it is observed that absolute path of program as well as DLL's contains symbols like forward-slash (/), hyphen (-), and dot (.) , which are considered as special characters and separators in various platforms. To fit the collected data well in the system, an encryption process is carried out on processes and DLL's. A unique class label say p_i is assigned for all instances of a process-i considering its absolute path. Each individual DLL in the DLL sequences is encoded with a unique id, named as dll_i. The process of encryption on DBDLLList[] is explained in Algorithm 2 which is a.k.a. Encrypt_DBDLLList. It helps in preparing the data set ready for TF-IDF weight matrix construction.

Algorithm 2: ***Encrypt_DBDLLList***

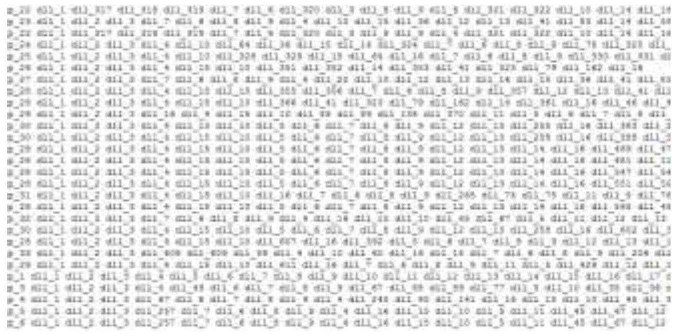
Require: *DBDLLList[]* : Collection of records as shown in "Fig. 3."

Ensure: A list *EncrDBDLLList[]* and a dictionary *DictDBDLLList{}* as explained below.

(a) *EncrDBDLLList[]*: An encrypted set consisting of records of all the In-memory processes where each record will be in the form of a process class label *(p_i)* followed by a sequence of DLL id's *(dll_i)*. A sample of the expected output is shown in "Fig. 4."

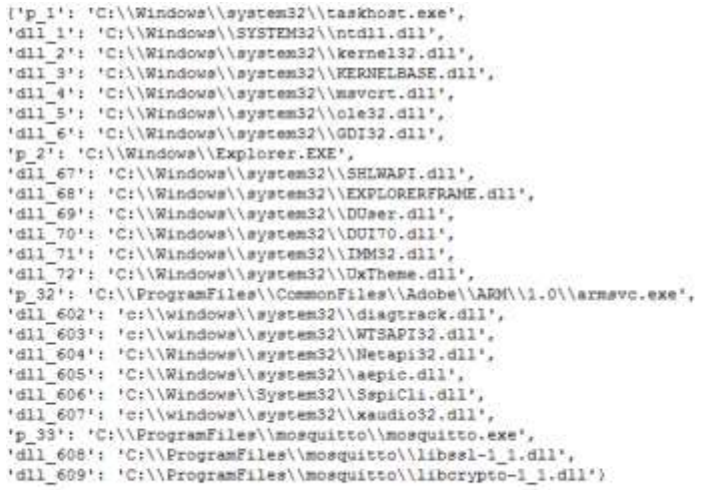
(b) *DictDBDLLList{}*: A dictionary of *{key: value}* pairs. When the key is some *p_i*, the value is the absolute path of the respective In-memory process. When the key is some *dll_i*, the value is the absolute path of respective DLL. A Sample of the expected output is shown in "Fig. 5."

```
1. Initialize EncrDBDLLList[]=NULL
2. Initialize DictDBDLLList{}=EMPETY
3. For each record in DBDLLList[]
   a. words = split ( record , " ")
   b. If words[0] is in DictDBDLLList.values()
              p_i= DictDBDLLList.values(words[0])
              EncrDBDLLList[].append(p_i)
      Else
              p_j= generate_next_process_class_id()
              DictDBDLLList[p_j]= words[0]
              EncrDBDLLList[].append(p_j)
   c. For each word in words[1..n]
      If word is in DictDBDLLList.values()
              dll_i= DictDBDLLList.values(word)
              EncrDBDLLList[].append(dll_i)
      Else
              dll_j= generate_next_dll_id()
              DictDBDLLList[dll_j]= word
              EncrDBDLLList[].append(dll_j)
   d. EncrDBDLLList[].append(LineBreak)
4. Return EncrDBDLLList[] and DictDBDLLList{}
```



[illegible]

Fig. 4. Some Records of EncrDBDLLList[].



```
{'p_1': 'C:\\Windows\\system32\\taskhost.exe',
'dll_1': 'C:\\Windows\\SYSTEM32\\ntdll.dll',
'dll_2': 'C:\\Windows\\system32\\kernel32.dll',
'dll_3': 'C:\\Windows\\system32\\KERNELBASE.dll',
'dll_4': 'C:\\Windows\\system32\\msvcrt.dll',
'dll_5': 'C:\\Windows\\system32\\ole32.dll',
'dll_6': 'C:\\Windows\\system32\\GDI32.dll',
'p_2': 'C:\\Windows\\Explorer.EXE',
'dll_67': 'C:\\Windows\\system32\\SHLWAPI.dll',
'dll_68': 'C:\\Windows\\system32\\EXPLORERFRAME.dll',
'dll_69': 'C:\\Windows\\system32\\DUser.dll',
'dll_70': 'C:\\Windows\\system32\\DUI70.dll',
'dll_71': 'C:\\Windows\\system32\\IMM32.dll',
'dll_72': 'C:\\Windows\\system32\\UxTheme.dll',
'p_32': 'C:\\ProgramFiles\\CommonFiles\\Adobe\\ARM\\1.0\\armsvc.exe',
'dll_602': 'c:\\windows\\system32\\diagtrack.dll',
'dll_603': 'c:\\windows\\system32\\WTSAPI32.dll',
'dll_604': 'C:\\Windows\\system32\\Netapi32.dll',
'dll_605': 'C:\\Windows\\system32\\aepic.dll',
'dll_606': 'C:\\Windows\\System32\\SspiCli.dll',
'dll_607': 'c:\\windows\\system32\\xaudio32.dll',
'p_33': 'C:\\ProgramFiles\\mosquitto\\mosquitto.exe',
'dll_608': 'C:\\ProgramFiles\\mosquitto\\libssl-1_1.dll',
'dll_609': 'C:\\ProgramFiles\\mosquitto\\libcrypto-1_1.dll'}
```

Fig. 5. Some Records of DictDBDLLList{}.

C. Process Class Prediction Model

The records present in EncrDBDLLList[] are being tokenized using the classical separator blank space. With this an In-memory process P is represented as a text $P_{text} = \{S_0, S_1, S_2, \ldots, S_n\}$where each S_i is considered as a string of the text. Here S_0 represents the process class p_i for an In-memory process. S_1 to S_n represents DLL sequence of that process where each S_i represents dll_i for$1 \leq i \leq n$. Let C is the set of the text representation of m In-memory processes such that$C = \{P_{text}^1, P_{text}^2, P_{text}^3, \ldots, P_{text}^m\}$, where each P_{text}^irepresents the text representation of ith In-memory process. Further C is split into two lists named as C_{tags} and C_{docs}. Where S_0 will be included in C_{tags} when $S_0 \in P_{text}^i$ and $\forall P_{text}^i \in C$. S_j will be included in C_{docs} when $S_j \in P_{text}^i$with $1 \leq j \leq n$ and $\forall P_{text}^i \in C$.

The use of VSM is very common in representing textual documents algebraically as vectors in a multidimensional space [19]. The components of such a vector represent the importance of a term in a document. TF-IDF is very popular in evaluating how important a word is in a document. TF-IDF weighting schema is the most popularly used approach in converting textual documents to a VSM [20].

In this context, C_{tags} is the document representing the list of classes of In-memory processes, where existence of more than 3 classes observed. C_{docs} is the list of DLL's for process classes in $C_{tags.}$ C_{docs} is treated as a textual document, which is the list of the text representation of DLL sequences of all In-memory processes. The TF-IDF weighting schema is applied to find out the VSM view of the system for a particular host. Considering TF-IDF over raw frequencies of occurrences of words is to scale down the impact of very frequently occurring words in a document which is empirically less informative than the words of less frequency. C_{docs} is represented by a "Feature-DLL to In-Memory-Process" weight matrix, where the element (i,j) illustrates an association of ith DLL to jth In-memory Process. Using TF-IDF weighting schema, the weight of ith DLL to jth In-Memory Process is denoted as $W_{i,j}$ and defined as given in (1).

$$W_{i,j} = TF_{i,j} \times IDF_i \tag{1}$$

$TF_{i,j}$ in (1) is the L_2 normalized term frequency for ith DLL with respect to the jth In-memory process. The Term Frequency$TF_{i,j}$ is defined as given in (2).

$$TF_{i,j} = \frac{n_{i,j}}{\sqrt{\sum_k (n_{k,j})^2}} \tag{2}$$

Here $n_{i.j}$ is the number of occurrences of ith DLL in jth In-memory Process, and $\sqrt{\sum_k (n_{k,j})^2}$ is the magnitude of the vector representation of DLL's present in the jth In-memory Process.

IDF_i in (1) is the Inverse document frequency for ith DLL in C_{docs}. The Inverse Document FrequencyIDF_i is defined as given in (3).

$$IDF_i = \log\left(\frac{|C_{docs}|}{1 + |C_{docs}|S_i \in C_{docs}|}\right) \tag{3}$$

Here $|C_{docs}|$ represents the total number of In-memory processes and $|C_{docs}|S_i \in C_{docs}|$ represents the number of In-memory processes in C_{docs} containing the ith DLL i.e. S_i. Using (1) W the weight matrix of C_{docs} is found for the 'Feature-DLL to In-Memory-Process' matrix representation of the system. W is typically a sparse matrix and tells statistically how important a DLL is to an In-memory process in the collection of all the In-memory processes.

Weight matrix *W* is then split into a training and testing data set with 3:1 ratio with random sample selection. Multinomial logistic regression, multinomial Naïve Bayes, and SVM-SVC (SVC) learning methods are applied on the proposed model. The objective is to choose the classifier which results with highest accuracy in process class prediction by using a DLL sequence as attribute vector. "Fig. 6" shows the functional representation of the model.

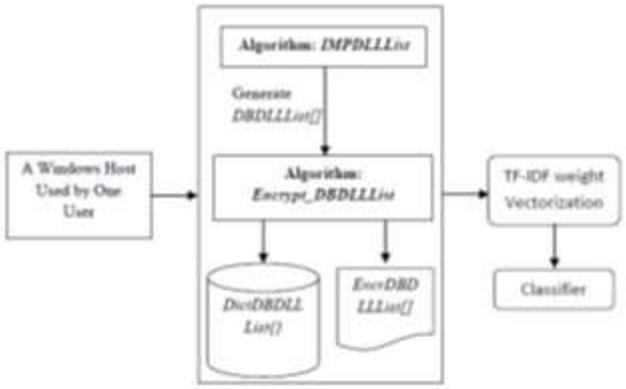

Algorithm: IMPDLLList
Generate DBDLLList[]
A Windows Host Used by One User
Algorithm: Encrypt_DBDLLList
DictDBDLL List[]
EncrDBD LLList[]
TF-IDF weight Vectorization
Classifier

Fig. 6. Process Class Prediction Model.

D. Cross Validation

The cosine similarity measure is used to cross-validate the suspected process with all the processes of the predicted process class. It is used to find the relative closeness of the suspected process with the trusted processes of the predicted class. Cosine Similarity is a similarity distance measure which finds the cosine angle between two vectors *u* and *v*, which is defined as given in (4).

$$\cos(\theta) = \frac{u \cdot v}{|u| \cdot |v|} \tag{4}$$

Here $u \cdot v$ is the dot product of two vectors *u* and *v*. $|u| \cdot |v|$represents product of magnitudes of vectors *u* and *v* , respectively. Cosine angle as 0^o (i.e. Cosine distance measured as 1) between two vectors concludes both are similar where as an angle close to 0^o (i.e. Cosine distance measured is close to 1) indicates they are closely similar. What must be the accepted value to consider case of closely similar vectors to case of similar vectors depends on field of application and experiential results? But a larger angle says they are dissimilar.

The proposed system has the objective of detecting anomalous In-memory processes on a specific host with reference to trusted applications list. With VSM and TF-IDF any In-memory process can be represented as a weighted vector considering DLL sequence as attribute vector. Hence any suspected process can be applied on the model to find its process class. All the processes of the predicted process class can be compared with the suspected process using the Cosine similarity measure. This cosine similarity distance is used to conclude whether the suspected process is to be considered as trusted, variant of trusted or untrusted. The algorithmic steps of cross validation are explained in Algorithm 3 which is a.k.a. Cross_validate_suspected_process. The algorithm needs a Suspected_Process information similar to the sample record shown in "Fig. 3", i.e. in the form of [process_path, dll1_path, dll2_path,….,dllm_path]. It also refers EncrDBDLLList[] and DictDBDLLList{} which are found during learning stage of model using Algorithm 2 **Encrypt_DBDLLList**, to encode the Suspected Process information such that it can be applied on Process Class Prediction Model. β_1 and β_2 are the threshold values for considering cosine distance measure to decide process as a trusted and a variant of some trusted program, respectively.

The Suspected_Process will be a trusted process if the number of processes of the predicted class whose Cosine distance is measured as 1 with the Suspected_Process becomes $\geq$ the threshold β_1. Here β_1 will be the minimum count for the number of processes of the predicted class whose Cosine distance is measured as 1 with the Suspected_Process. An optimize value for β_1 to be found from experiment for a specific host.

The Suspected_Process will be a variant of some trusted process if the average of Cosine distances measured between the processes of predicted class and the Suspected_Process becomes $\geq$ the threshold β_2. Here β_2 will be the minimum average cosine distance between the processes of predicted class and the Suspected_Process. An optimize value for β_2 to be found from experiment for a specific host.

The Suspected_Process which fails to qualify the threshold β_1 followed by β_2 will be an untrusted process.

Algorithm 3: *Cross_validate_suspected_process*

Require: *Suspected_Process* Information, *EncrDBDLLList[] and DictDBDLLList{}* as explained above. β_1 is the threshold to conclude *Suspected_Process* is trusted and β_2 is the threshold to conclude *Suspected_Process* is a variant of some trusted process as explained above.

Ensure: Trusted or A variant of trusted or Untrusted as explained above

```
1.  encr_process= Encrypt_DBDLLList (Suspected_process ,
    DictDBDLLList{})
2.  predict_process_class=
    ProcessClassPredictionModel(encr_process)
3.  Initialize verify_proces_list[]=NULL
4.  verify_proces_list.append(encr_process)
5.  For p_class in EncrDBDLLList[]
        If (p_class[0] == predict_process_class)
            verify_proces_list.append(p_class[1..n])
6.  TF_IDF_mtrix= tf_idf_vectorizer(verify_proces_list[])
7.  Cosine_mesure[]=cosine_similarity(TF_IDF_mtrix[0],
    TF_IDF_mtrix[1..n])
8.  If (count(Cosine_mesure[].value( '1')) >= $\beta_1$)
        Print (Trusted)
    Else if (average(Cosine_mesure[]) >= $\beta_2$)
        Print (A variant of Trusted Application hence
        assumed as trusted)
    Else
        Print (Untrusted)
9.  End
```

IV. Experimental Setup and Evaluation

For the experimental setup and evaluation of the proposed system following steps are taken.

- A questionnaire is used to collect the list of application programs with reference to specific users' interest. It is considered as the trusted application list for this host and any other application is assumed as untrusted. Table I and Table II show a sample list of system processes and trusted processes of the host respectively. Combination of such system processes and processes of the trusted application is considered as the list of trusted processes on which anomalous activity is monitored.
- The Algorithm-1 *IMPDLLList* is invoked with a fresh installation of the Windows operating system along with all the listed trusted application software's. The invocation of the algorithm is scheduled depending on use of various application programs time to time. *IMPDLLList* is invoked aperiodically for approximately 20 times a day for a continuous run of a specific time duration (e.g. 5 hours a day) to generate *DBDLLList[]*. With the above-said schedule *IMPDLLList* is invoked for 10 days to generate the final trusted *DBDLLList[]*, which contained around 10000 records. One record of *DBDLLList[]* is shown in "Fig. 3.".
- The algorithm *Encrypt_DBDLLList* is invoked on the trusted *DBDLLList[]*, to generate *EncrDBDLLList[]* and *DictDBDLLList{}*. The model is trained and tested with training and testing set of 3:1 ratio with random sample selection on these 10000 records.

The proposed model works on a multi-class problem where the In-memory processes of a host are classified into several classes and a suspected process gets predicted to belong to a specific class of the processes. The performance of three classifiers are compared in terms of accuracy, {Micro | Macro | Weighted} Precision, {Micro | Macro | Weighted} Recall and {Micro | Macro | Weighted} F1-score considering the multinomial classification approaches named multinomial Logistic Regression, multinomial Naïve Bayes and SVM-SVC (further referred as SVC in this paper). For a binomial classification case evaluation of performance metrics is done based on positive class and negative class, whereas for a multinomial classification case evaluation of performance metrics is done based on One-vs.-Rest (OvR) classes. For each class in case of multinomial classification the below mentioned basic parameters are found, which are used to evaluate overall performance metrics of the model. The basic parameters referred above are True Positive (TP) — the classifier correctly predicts the class, True Negative (TN) — the classifier correctly predicts which are not of the class, False Positive (FP) — the classifier incorrectly predicts other classes to be of the class and False Negative (FN) — the classifier incorrectly predicts the class to be of other class. Table III explains pictorially a sample case of three classes in which how TP, FP, FN, and TN for CLASS1 to be considered in a multinomial classification scenario.

In the proposed model the below-given performance metrics are calculated for three multi-classification approaches named as OvR Logistic Regression, OvR Naïve Bayes and OvR SVC. Recall for a class says a fraction of all samples of that class which is predicted correctly, which is evaluated as given in (5). Precision for a class says a fraction of all predicted samples of that class which is predicted correctly, which is evaluated as given in (6). F_1 score of a class will be the harmonic mean of precision and recall of that class, which is evaluated as given (7).

$$recall = \frac{TP}{TP+FN} \tag{5}$$

$$precision = \frac{TP}{TP+FP} \tag{6}$$

$$F_1Score = 2 \times \frac{precision \times recall}{precision+recall} \tag{7}$$

TABLE I. LIST OF SYSTEM PROCESSES TO SUPPORT TRUSTED PROCESSES

Process name	Process Path
Cmd	C:\Windows\System32\cmd.exe
Conhost	C:\Windows\system32\conhost.exe
Conhost	C:\Windows\system32\conhost.exe
Csrss	C:\Windows\system32\csrss.exe
Csrss	C:\Windows\system32\csrss.exe
Dwm	C:\Windows\system32\Dwm.exe
Hkcmd	C:\Windows\System32\hkcmd.exe
Igfxpers	C:\Windows\System32\igfxpers.exe
Igfxsrvc	C:\Windows\system32\igfxsrvc.exe
Igfxtray	C:\Windows\System32\igfxtray.exe
Lsass	C:\Windows\system32\lsass.exe
Lsm	C:\Windows\system32\lsm.exe
SearchIndexer	C:\Windows\system32\SearchIndexer.exe
Services	C:\Windows\system32\services.exe
Smss	C:\Windows\System32\smss.exe
Spoolsv	C:\Windows\System32\spoolsv.exe
Svchost	C:\Windows\system32\svchost.exe
Svchost	C:\Windows\System32\svchost.exe
Svchost	C:\Windows\system32\svchost.exe
Svchost	C:\Windows\system32\svchost.exe
Taskhost	C:\Windows\system32\taskhost.exe
Wininit	C:\Windows\system32\wininit.exe
winlogon	C:\Windows\system32\winlogon.exe
wuauclt	C:\Windows\system32\wuauclt.exe
WUDFHost	C:\Windows\System32\WUDFHost.exe
explorer	C:\Windows\Explorer.EXE

TABLE II. A SAMPLE LIST OF TRUSTED PROCESSES OF A HOST

Process namez	Process Path
notepad	C:\Windows\system32\NOTEPAD.EXE
devcpp	C:\ProgramFiles\Dev-Cpp\devcpp.exe
firefox	C:\ProgramFiles\MozillaFirefox\firefox.exe
SnippingTool	C:\Windows\system32\SnippingTool.exe
notepad++	C:\ProgramFiles\Notepad++\notepad++.exe
Vlc	C:\ProgramFiles\VideoLAN\VLC\vlc.exe
FreeCell	C:\ProgramFiles\MicrosoftGames\FreeCell\FreeCell.exe
Hearts	C:\ProgramFiles\MicrosoftGames\hearts\hearts.exe
Chess	C:\ProgramFiles\MicrosoftGames\chess\chess.exe
chrome	C:\ProgramFiles\Google\Chrome\Application\chrome.exe
EXCEL	C:\ProgramFiles\MicrosoftOffice\Office12\EXCEL.EXE
Zoom	C:\Users\binu\AppData\Roaming\Zoom\bin\Zoom.exe
AcroRd32	C:\ProgramFiles\Adobe\AcrobatReaderDC\Reader\AcroRd32.exe
pythonw	C:\Users\binu\AppData\Local\Programs\Python\Python36-32\pythonw.exe
POWERPNT	C:\ProgramFiles\MicrosoftOffice\Office12\POWERPNT.EXE
WINWORD	C:\ProgramFiles\MicrosoftOffice\Office12\WINWORD.EXE

TABLE III. TP, FP, TN, AND FN FOR CLASS1 IN MULTINOMIAL CASE

Predicted Class	True Class		
	Class 1	Class 2	Class 3
Class 1	TP	FP	
Class 2	FN	TN	
Class 3			

For a multi-class scenario: Micro precision, Micro recall and Micro F_1Score are calculated globally considering total TP, total FP, and total FN of the model instead of considering individual classes. In such case Accuracy of the model is same as Micro precision, Micro Recall, and Micro F_1Score measured for the model globally. Macro precision, Macro recall and Macro F_1Score are calculated considering the precision, recall, and F_1Score of individual classes and taking the un-weighted mean of the measures. The Weighted precision, weighted recall and Weighted F_1Score are calculated considering the precision, recall, and F_1Score of individual classes and taking the weighted mean of the measures. The weight for each class is the total number of samples of that class. Accuracy and all the calculated Micro, Macro and Weighted performance metrics for the considered classifiers on the testing data is shown in Table IV.

The comparison of accuracy for the three classifiers OvR Logistic Regression, OvR Naïve Bayes and OvR SVC is plotted in "Fig. 7". It has been found that the performance of OvR SVC is better than OvR Naïve Bayes but OvR Logistic Regression is found better than OvR SVC. The comparison of precision, recall and F_1score for all three classifiers is plotted in "Fig. 8", "Fig. 9" and "Fig. 10", respectively. Considering all these metrics, it is found that OvR Logistic Regression

Classifier is most efficient for the model in comparison to other two classifiers. OvR Logistic Regression Classifier outperformed others with accuracy rate as 97% in predicting a process to its class and highest rates in precision, recall and F_1Score.

To ensure the result of the system in identifying a suspected process as either trusted or variant of a trusted or untrusted, the processes of predicted class for a suspected process need to be cross-validated. Cross-validation of the suspected process is done with processes of predicted class considering cosine distance measure. As the OvR Logistic Regression classifier resulted with higher accuracy over the other two, it is chosen to predict the class of a suspected process at any time instance. As explained in Algorithm 3 Cross_validate_suspected_process, Cosine distance is measured between DLL sequence vector of a suspected process and DLL sequence vectors of all the processes of the predicted class. The calculated list of cosine distance measures are used to reach a conclusion about the suspected In-memory process as either a trusted or a variant of trusted or untrusted type for the specific host. To evaluate the systems performance 300 processes of mixed samples are selected from the considered host. In these samples 200 processes are from trusted process list, 40 processes are from variant of the trusted process list and 60 processes are neither from the trusted list of processes nor from variant of any trusted process. Table V shows the confusion matrix for the 300 processes considered as suspected processes, which are initially applied to process class prediction model and further Cross validated using cosine distance measure. During the Cross validation of suspected process with processes of predicted class the optimized threshold value for β_1 and β_2 are calculated by several iterations. For the considered host the optimized value for β_1 is found as 1, which is the minimum number of processes from the predicted class with cosine distance as 1 to the suspected process. The optimized value for β_2 is found as 0.90, which is the average cosine distance of processes from the predicted class to the suspected process. "Fig. 11" speaks about precision and recall of the case study on the preferred host. In all cases, precision is above 95%. Whereas for trusted and untrusted processes recall is more than 93%, but for variant of process it is 84%.

TABLE IV. PERFORMANCE METRICS OF THE 3 MULTINOMIAL CLASSIFIERS

Performance metrics	OvR Log Reg	OvR Naïve Bayes	OvR SVC
Accuracy	0.97	0.88	0.92
Micro Precision	0.97	0.88	0.92
Micro Recall	0.97	0.88	0.92
Micro F1-score	0.97	0.88	0.92
Macro Precision	0.79	0.48	0.75
Macro Recall	0.8	0.5	0.76
Macro F1-score	0.79	0.47	0.75
Weighted Precision	0.95	0.7	0.9
Weighted Recall	0.97	0.88	0.92
Weighted F1-score	0.96	0.74	0.91

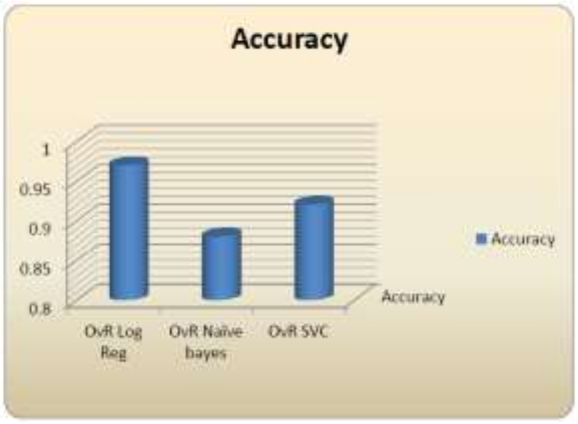


Fig. 7. Accuracy Comparison of the 3 Classifiers.

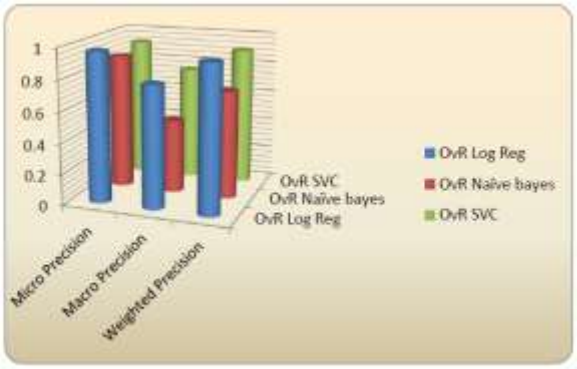


Fig. 8. Precision Comparison of the 3 Classifiers.

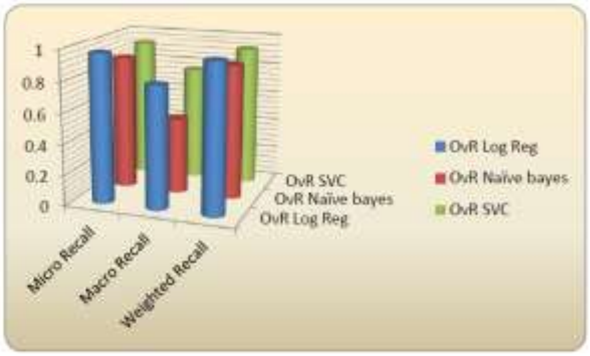


Fig. 9. Recall Comparison of the 3 Classifiers.

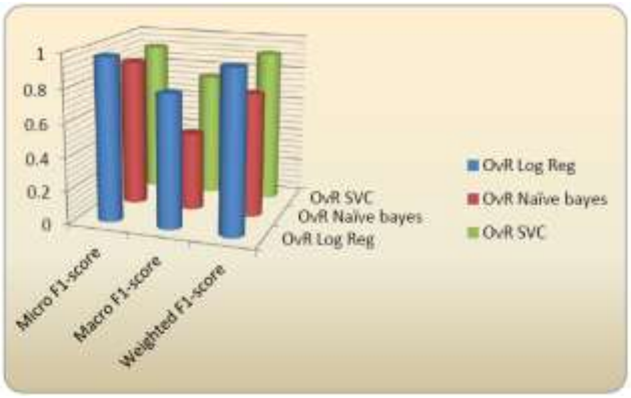


Fig. 10. F_1Score Comparison of the 3 Classifiers.

TABLE V. A CASE STUDY OF CROSS VALIDATION FOR THE SPECIFIC HOST

Prediction Cross Validated	**Actual Cases applied to Model**		
	Trusted	**Variant of Trusted**	**Untrusted**
Trusted	191	2	1
Variant of trusted	5	38	2
Untrusted	4	0	57

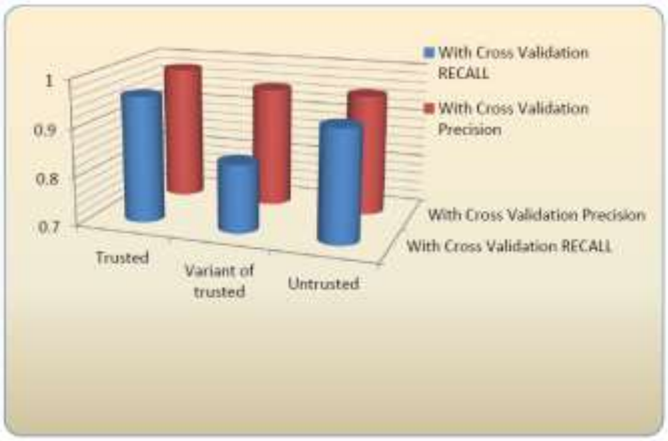


Fig. 11. Recall and Precision after Cross Validation of the Specific Host.

V. CONCLUSION

The said system considers the trusted process list of a specific host as a multi-class problem considering DLL sequences as attribute vectors for In-memory processes. The objective of the system is to detect any deviation in the In-memory processes of the specific host. The system works in two stages. First stage is the process class prediction model, which is used to predict the class of a suspected process referring its DLL sequence as attribute vector. Second stage is Cross validation of the suspected process with the processes of predicted class. Three different multinomial classification approaches considered during evaluation of the process class prediction model where OvR Logistic Regression is proven to be the best performer compared to others. With OvR Logistic Regression 97% of accuracy and more than 95% of weighted precision, recall, and F_1 score achieved for the model. To identify anomaly or deviation with some In-memory process during Cross validation of the suspected process with processes of the predicted class, use of cosine distance measure is found very effective. The case study during evaluation of system shows precision above 95% for all trusted, variant of trusted and untrusted processes. Recall of variant of trusted process is found as 84% where as 93% for trusted and untrusted processes. These results are quite impressive for finding any deviation with respect to In-memory processes of the host under consideration. An optimized value for threshold's β_1 and β_2 plays significant role for concluding the suspected process as either trusted or variant of a trusted or untrusted. In the case study using β_1 as 1 and β_2 as 0.9 shows the best performance on the host under consideration. It is also observed higher β_1 moves less occurring trusted processes to a variant of trusted process and higher β_2 moves a variant of trusted to untrusted process. Hence β_1 and β_2 has impact on false negative cases. A lower value of β_2 moves untrusted process to variant of trusted process and variant of trusted to trusted process. Hence β_2 has impact on false positive cases. So an optimized value of β_1 and β_2 has significant impact on the performance of the system. It is also to be understood that the model's performance relies on the agreed list of trusted processes by the user on a specific host. The data collection for the training of process class prediction model is to be done under a proper supervision, as a biased data may result in higher false negatives or higher false positives. This system is found effective with memory based dynamic analysis for detection of anomaly or deviation from its normal operation with reference to known or trusted In-memory processes of a specific host. This system may help to have zero-day detection with respect to the presence of anomalous In-memory processes on a specific host which can be either an unknown program or a PUA or a malware. This system can be extended to find anomalies with In-memory processes considering a group of hosts with possible communication among the hosts. Using an efficient protocol for exchanging information about processes may help in reducing false negative or false positive cases. Analysis of communication cost with expectations in decrease in false positive and false negative cases may be crucial in performance evaluation of the system.

REFERENCES

[1] Quick Heal threat report (2019): https://www.quickheal.co.in/documents/threat-report/QH-Annual-Threat-Report-2019.pdf.

[2] Internet Security threat report, Symantec (2019) : https://docs.broadcom.com/doc/istr-24-2019-en.

[3] Sihwail, Rami & Omar, Khairuddin & Zainol Ariffin, Khairul Akram. (2018). A survey on malware analysis techniques: static, dynamic, hybrid, and memory analysis. 8. 1662. 10.18517/ijaseit.8.4-2.6827.

[4] E. Gandotra, D. Bansal, and S. Sofat, "Malware Analysis and Classification: A Survey," J. Inf. Secur., vol. 05, no. 02, pp. 56–64, 2014.

[5] P. V. Shijo and A. Salim, "Integrated static and dynamic analysis for malware detection," in Procedia Computer Science, 2015, vol. 46, pp. 804–811.

[6] Watson, Michael & Shirazi, Syed Noorulhassan & Marnerides, Angelos & Mauthe, Andreas & Hutchison, David. (2015). Malware Detection in Cloud Computing Infrastructures. IEEE Transactions on Dependable and Secure Computing. 13. 1-1. 10.1109/TDSC.2015.2457918.

[7] R. Mosli, R. Li, B. Yuan, and Y. Pan, "Automated malware detection using artifacts in forensic memory images," in 2016 IEEE Symposium on Technologies for Homeland Security, HST 2016, 2016, pp. 1–6.

[8] C. Rathnayaka and A. Jamdagni, "An efficient approach for advanced malware analysis using memory forensic technique," Proc. - 16th IEEE Int. Conf. Trust. Secur. Priv. Comput. Commun. 11th IEEE Int. Conf. Big Data Sci. Eng. 14th IEEE Int. Conf. Embed. Softw. Syst., pp. 1145–1150, 2017.

[9] Sanz, Borja & Santos, Igor & Ugarte-Pedrero, Xabier & Laorden, Carlos & Nieves, Javier & Bringas, Pablo. (2014). Anomaly Detection Using String Analysis for Android Malware Detection. 10.1007/978-3-319-01854-6_48.

[10] C. W. Tien, J. W. Liao, S. C. Chang, and S. Y. Kuo,(2017) "Memory forensics using virtual machine introspection for Malware analysis," in 2017 IEEE Conference on Dependable and Secure Computing, 2017, pp. 518–519.

[11] S. Kim, J. Park, K. Lee, I. You, and K. Yim, (2012): "A Brief Survey on Rootkit Techniques in Malicious Codes," J. Internet Serv. Inf. Secur., vol. 3, no. 4, pp. 134–147, 2012.

[12] Navaki Arefi, Meisam & Alexander, Geoffrey & Rokham, Hooman & Chen, Aokun & Faloutsos, Michalis & Wei, Xuetao & Oliveira, Daniela

& Crandall, Jedidiah. (2018): FAROS: Illuminating In-memory Injection Attacks via Provenance-Based Whole-System Dynamic Information Flow Tracking. 231-242. 10.1109/DSN.2018.00034.

[13] A. Hosseini, “Ten Process Injection Techniques: A Technical Survey of Common and Trending Process Injection Techniques,” 2017. [Online]. Available: https://www.endgame.com/blog/technicalblog/ ten-process-injection-techniques-technical-survey-common-andtrending-process.

[14] Amanda Steward. (2014): FireEye DLL Side-Loading: A Thorn in the Side of the Anti-Virus Industry. Retrieved March 13, 2020.

[15] Microsoft. (2018, May 31): About Side-by-Side Assemblies. Retrieved March 13, 2020.

[16] K., Sudhakar & Kumar, Sushil. (2019): An emerging threat Fileless malware: a survey and research challenges. In Cyber Security 3. 1. Publisher: Springer (Biomed Central Ltd.) Dec- 2019 DOI:10.1186/s42400-019-0043-x.

[17] Cheng, J.Y.C., Tsai, T.S., Yang, C.S., (2013): An information retrieval approach for malware classification based on Windows API calls. Int. Conf. on Machine Learning and Cybernetics, p.1678-1683. https://doi.org/10.1109/ICMLC.2013.6890868.

[18] Windows Sysinternals Process Utilities: https://docs.microsoft.com/en-us/sysinternals/downloads/process-utilities.

[19] Baeza-Yates, R.A., Ribeiro-Neto, B (1999): Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999).

[20] Salton, G., McGill, M.: Introduction to modern information retrieval. McGraw-Hill New York (1983).