Journal of Technology Informatics and Engineering
Vol.1, No.1 April 2022
e-ISSN: 2961-8215; p-ISSN: 2961-9068, Hal 80-107

ADVANCED MALICIOUS SOFTWARE DETECTION USING DNN

Sulartopo[1], Dani Sasmoko[2], Zaenal Mustofa[3], Arsito Ari Kuncoro[4]
Universita Sains dan Teknologi Komputer

Abstract

The special component of malicious software analysis is advanced malicious software analysis which implicates interested the main framework of malicious software that can be executed after executing it and aggressive malicious software investigation depend on inquisitive of the practice of malicious software after running it in a composed habitat. Advanced malicious software analysis is usually performed by contemporary anti-malicious software operating systems using signature-based analysis.

The purpose of this research is to propose also decide a DNN for the progressive identification of portable files to study the features of portable executable malicious software to minimize the occurrence of distorted likeness when aware of advanced malicious software. The model proposed in this study is a NN with a Dropout model contrary to a resolution tree model to examine how well it performs in detecting real malicious PE files. Setup-skeptic methods are used to extract features from files. The dataset is used to train the proposed approach and measure outcomes by alternative common malicious software datasets.

The results from this study illustrate that the use of simple DNNs to study PE vector elements is not only efficient but more fewer system comprehensive than the traditional interested disclosure approach. The model proposed in this study achieves an A-UC of ninety-nine point eight with ninety accurate specifics at one percent inaccurate specific on the R-OC curve. For shows that this model has the potential to complement or replace conventional anti-malicious software operating systems so for future research, it is proposed to implement this model practically.

Keywords: *Malicious software detection, DNN, Portable Executable*

Received Maret 05, 2022; Revised Maret 20, 2022; April 13, 2022

INTRODUCTION

Since the emergence of the anti-malicious software operating system, an increase in sophisticated malicious software categorically calculated to outsmart this software has in turn led to an inquiry into a higher progressive disclosure approach. The concept of malicious software detection relates to the investigation assassinate files to determine whether malicious is resolved. Malicious software analysis or malicious software disclosure can be done in two ways: progressively and dynamic. Progressive malicious software disclosure is the action of correlating binary files without running them. This can involve deploying a complete file and being able to inspect each element, adopting a dismantler to turn builder it, or turning it into accumulation code to check the flow Sikorski & Honig, (2012). "The engineering can still be extended to the source code of the software if it is available" Manuel et al, (2012). "DMS is the 1st channel of defense contrary to malicious software used by all anti-malicious software". Effective malicious software revelation uses nature analysis while malicious software is fuctioning to resolve malevolent alert, this scheme of analysis is usually resource-intensive and can be circumvented in several ways. These analysis schemes are usually resource intensive and can be carried out in a several of ways. "The debugger can also be used to analyze system calls or other behavior patterns that cannot be detected using black-box testing" Dilshan (2016).

The outlook of this research will only focus on advanced malicious software detection. Machine Learning (ML) has long been used to classify data with complex characteristics that cannot be determined easily using mathematical functions. DNNs are currently used in a variety of different applications including data allocation, data forecasting, image perception, NLP, and so on. The versatility of NNs are accessible, but computationally the process of obtaining certain results is very costly.

Until now, the loss of opportunity of labeled datasets termed for managed study has different evolution in using ML or DL for malicious software revelation. Igor et al, (2013) proposed "OPEM as an advanced-aggressive model to use ML to detect anonymous malicious software". They propose analyzing operational code obtained from disassembling executables and identifying their implementation tracks to decide malicious intent. Similarly, "a dynamic malicious software detection framework for Android called DroidDolphin managed to achieve 86.1% accuracy" Wu & Hung, (2014) accepting energetic malicious software investigation. Both approaches are commonly calculational accelerated and have defined opportunities for title data.

Destination

The essential purpose of this research is to construct and classify a DNN to advanced ally analyze compact assassinatable files to analyze them as malevolent. So then, for the purpose, this research uses a data set that consists of data extracted from harmless and harmless portable executable files. The trick used for extraction of this portable file is quarrel by Smola, (2009) "to make a numerical arbitrary of the appearance and standardize the input vector". Furthermore, this research model is compared with a similar model proposed previously to handle malicious software advanced analysis. This research demonstrates a model that simulates a productive application of a similar approach for further analysis. The entire authority for breeding the expected approach and its derivatives is provided in this study.

Literature Review

Here many different objections are associated with progressive malicious software investigation. On this, mainly issues can be resolved using effective malicious software investigation equally file extortion all along processing timr, obfuscation Code, or executable encrypted binary files. In this literature review, some of the problems and drawbacks of semantic

analyzers are explored. Commonly, antivirus software uses signature-based methods to detect malicious software. "The procedure in the malicious software executable is deciphered to derive a unique signature that identifies the malicious software which is then compared to a large database of known malicious software signatures" Ammar et al, (2012) and Philip et al, (2011). Bonfante, (2007) planned "a control flow chart method to overcome this problem use a graph with nodes for all commonly used assembly instructions, then use a reduced version of this graph as a signature to classify malicious software".

Advanced malicious software investigation has primarily been planned from the aspect of linguistic investigation and origin code inquiry for distribution. Andreas, (2007) "explain an approach to obfuscate code from a linguistic investigation by simply adopting hazy specification constants to obfuscate program control flow". It serves to highlight a important weakness in currently existing advanced malicious software analysis techniques with semanticware analysis where linguistic investigation could defeated by proposing a random method to computing restriction in actual time. One of the methods mentioned is to use a random seed to generate the location where the variable is stored or to chain processes and store the variable at a location that exists in another address.

Christodorescu and Jha, (2006) discussed "Obfuscation Code using encrypting double annals different times and combining tools for decryption". This mode of bafflement is simple to snap during processing time by identification crack files, but to resolve the level of file encryption without decrypting and analyzing it dynamically first is difficult. "A semantics-based approach that proposes a metric to measure the similarity between original malicious software code and obfuscated malicious software code was" Preda et al, (2007). There are many dynamic malicious software disclosure approaches including call graph analysis Ammar et al, (2012) and identifying behavior based on triggers Brumley et al, (2008). However, This method is calculational comprehensive and requires a lavatory framework that can carefully run and analyze malware.

Philip et al, (2011) described that "File packing is a common technique used when bundling large software in a small and compact package". Alike bundle procedure commonly affects some models of encryption that can probably avoid accessible descriptions of malicious software. Oberheide et al, (2009) "serve that one such tool called 'PolyPack' was specifically designed to prove that packers are an effective method of evading anti-virus and anti-malicious software operating system". This bring ten bundles that given data independently, formerly check the bundle data packers with the best results selected using 10 well-known antivirus scanners. Their study found that this increased his evasion rate by a factor of 2.58, as opposed to most antivirus programs.

Banko and Brill, (2001) explain that the evidence that ML works finer with bigger sets of data is build well. A lot of research has been released that uses ML for the classification of malicious software. Heller et al, (2003) presents "various methods such as effective investigation of system calls" (Kolosnjaji et al, 2016) "registry access monitoring hidden Markov model-based analysis" Attaluri et al, (2009) that has "suggested for effective malicious software study". Kolter & Maloof, (2004) "using n-grams by combining 4-byte sequences to produce about 255 million distinct n-grams". This research focuses on the problem of malicious software classification rather

than the problem of malicious software detection. This research proposes using the problematic model to determine which aspects are applicable and using the top 500 n-grams for analysis. This research introduce how to applying 'Support Vector Machine' (SVM), 'Naive Bayes', and 'J48 decision trees' to analyze data. In this study, a small sample set was used and it don't have access to the exact dataset used made it difficult to determine the validity of all outcomes when using datasets of a larger size. Bagga, (2017) "using approach with Microsoft Malicious software Classification which is a relatively large data set".

Raman, (2012) serve "Product Incident Response Team at Adobe Systems Inc" to analyze malicious software by separating the 7 last practiced appearances from compact executables. The extracted features are "`DebugSize, ImageVersion, IatRVA, ExportSize, ResourceSize, Virtu-alSize, NumberOfSections`". In each dataset is overall 100K malicious executables and 16K malware was used for the experiment. Quinlan, (1993) proposed "Various models were tested using this data, J48 decision tree obtained the best results: a true positive rate of 0.986 with a false positive rate of 0.057". The output of tester model liberated as a free tool for malicious software classification, but the data set is not published to perform any form of comparative research. Anderson and Roth next tested this trained model contrary to the dataset and found that it exhibited a false positive rate of 0.53 and a false negative rate of 0.08.

A dynamic malicious software classification model using a DNN called MtNet was proposed by Huang and Stokes in 2016 (Huang and Stokes, 2016). The dataset used for this study was provided by Microsoft Corporation and consisted of 6,6M sample files. 2,8M malicious files and 3,6M good files were extracted from this dataset. The tester features extracted during runtime file execution consist mainly of his two types of data: System function calls and null-terminated objects. Feature selection is performed using mutual information reported by Manning et al. (2010) taking input fitures amount of 50K. The end goal is to first classify malicious software as benign or harmful, then analyze mischievous software into one of 100 known malicious software families. The ReLU activation function is used in conjunction with an added dropout layer for better model performance. Although this model showed impressive results with an inaccurate specific estimate below 0.07 percent, the loss of opportunity of the analysis data set and the approach code used for the test made the reproduction of the particular outcome absurd.

Pascanu et al, (2015) "Echo state networks and recurrent NN-based classifier malicious software also tested for effective investigation of malicious software". That research establishes the usable of echo state network-based iterative model with a sigmoid activation function (logistic regression) for effective investigation of malicious software. The exact input vector is not disclosed but comes from an API call made by the file over the processing time for execution. The model achieves a true positive rate of 0.983 with a false positive rate of 0.001. The data set used in this study is internally sourced and is not openly accessible. The goal of this research is to authorize that a recurrent NN can be used for effective malicious software study. However, because the sets of data is unavailable, and the level appropriate to duplicate the prospective model are unavailable, based on this, the results would be very difficult to verify.

Element Option

ML is susceptible to the aspect set that is used for training. Different research has built positive aspects that are constructive for the efficient training of ML-based malicious software classifiers. Divandari et al, (2015) "Extract opcode data from files and use a Markov Blanket approach to summarize feature sets". Bilar, (2007) "Since the opcodes themselves constitute a significant part of the executable, they have been considered a reliable feature for malicious software detection". The prospective classical uses the "Hidden Markov Model" for malicious software classification. Saxe and Berlin, (2015) "introduces a setup-skeptic method to extract

features from files and the byte histogram approach". This method is an innovative approach to extracting byte-as-feature information from a file without requiring information about the actual function of the bytes and suggests extracting a histogram of all unit values present in a binary file along with a 2-dimensional byte-entropy histogram to build an understanding of the encryption potential. or the compression used in the file.

In this research model, a setup-skeptic method to extract an element from a file is used to accompaniment the header eradication approach in such a way that it can accomplish large efficiency without the large overhead appropriate to vectorize all bytes in a portable executable. "The feature quarrel trick has often been cited and used to model machine learning" Weinberger et al. (2009). Vector input for ML-based models are advanced cant grow in capacity. Therefore, a method is needed to completely encapsulate large input features into a more manageable advanced capacity for exercise. The component quarrel deception recommended an approach for definitely reducing the dimensions of the data is sti;; adequately represents the originally intended data, but offers linearly separable features to train the model effectively.

Driven Decision Trees and NNs

With recent advances, boosting methods for resolution tree models have been shown to have equal or better performance than artificial NNs. Roe et al, (2005) explains the method is relatively easy to set up and works together with a bigger total of variables. Hastie et al, (2009) and Schapire, (2003) proposed "with the advent of AdaBoost, improved resolution tree models have successfully moved from binary allocation to multi-category classification". The approach planned in this study was correlated to an actual driven resolution tree model for the same data set used for the model proposed in this study. Caruana and Niculescu-Mizil (2006) "establish that expressed aim engine, decorated decision trees, and neural nets have proportionate achievement in the most scheme with variance mainly limited to hyper-parameter tuning".

Format Portable Executable (PE)

Since its introduction, several improvements have been made for inclusion in advanced version of Windows, unix uses an format similar to the Windows PE format (ELF). Due to the limited data available on malicious software running on Unix-based operating systems, the scope of this work is limited to Windows executable files. However, COFF headers are included in PE files common to Unix and Windows environments (Kath, 1993). The model proposed in this study analyzes and completes the appearance copied from the PE file, even if it is malicious.

Root MSDOS

This root is run every time is run in MSDOS situation. Its main purpose is to print a message indicating that the file cannot be run in an MSDOS situation. The signature added after the MS-DOS root indicates that the file is in PE format.

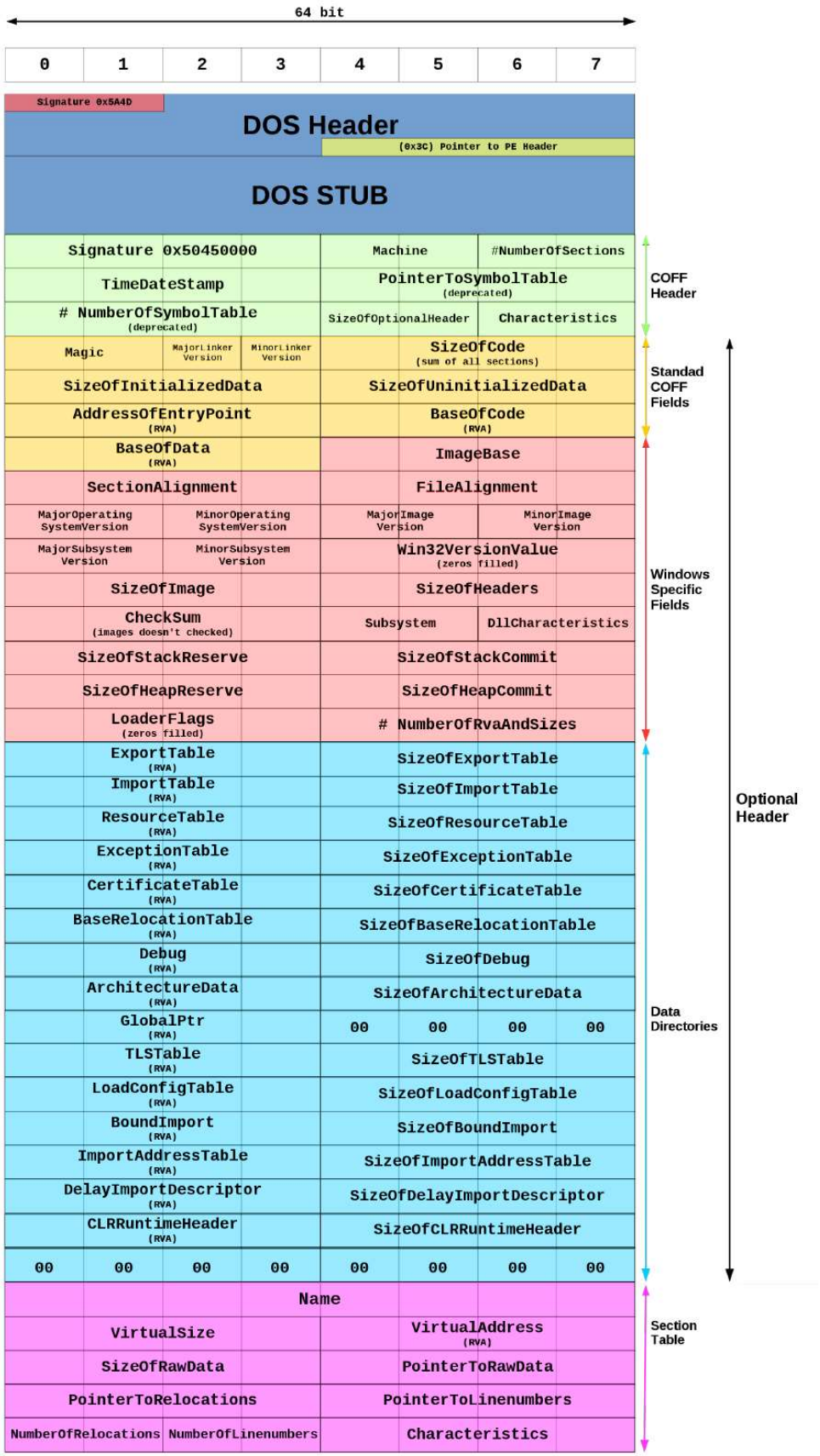

Figure 11 File Format for PE (Wikipedia)

Offset	Size	Field	Description
0	2	Machine	Identifies the target machine that the executable can run on. Refer to Table 3.2
2	2	NumberOfSections	Size of the section table. (follows the header table)
4	4	TimeDateStamp	Date of Creation. Represented as seconds after January 1, 1970.
8	4	PointerToSymbolTable	File offset of COFF symbol table. 0 for no table.
12	4	NumberOfSymbols	Number of entries in the symbol table.
16	2	SizeOfOptionalHeader	Size of the optional header (required for executables)
18	2	Characteristics	Indicates the attributes of the file. Refer to Table 3.3.

Table 1 COFF Structure

Constant	Value	Description
IMAGE_FILE_MACHINE_UNKNOWN	0x0	Applicaple to any machine
IMAGE_FILE_MACHINE_AM33	0x1d3	Matsushita AM33
IMAGE_FILE_MACHINE_AMD64	0x8664	x64
IMAGE_FILE_MACHINE_ARM	0x1c0	ARM little endian
IMAGE_FILE_MACHINE_ARM64	0xaa64	ARM64 little endian
IMAGE_FILE_MACHINE_ARMNT	0x1c4	ARM Thumb-2 little endian
IMAGE_FILE_MACHINE_EBC	0xebc	EFI byte code
IMAGE_FILE_MACHINE_I386	0x14c	Intel 386 or equivalent
IMAGE_FILE_MACHINE_IA64	0x200	Intel Itanium processor family
IMAGE_FILE_MACHINE_M32R	0x9041	Mitsubishi M32R little endian
IMAGE_FILE_MACHINE_MIPS16	0x266	MIPS16
IMAGE_FILE_MACHINE_MIPSFPU	0x366	MIPS with FPU
IMAGE_FILE_MACHINE_MIPSFPU16	0x466	MIPS16 with FPU
IMAGE_FILE_MACHINE_POWERPC	0x1f0	Power PC little endian
IMAGE_FILE_MACHINE_POWERPC	0x1f0	Power PC little endian
IMAGE_FILE_MACHINE_POWERPCFP	0x1f1	Power PC with floating point support
IMAGE_FILE_MACHINE_R4000	0x166	MIPS little endian
IMAGE_FILE_MACHINE_RISCV32	0x5032	RISC-V 32-bit address space
IMAGE_FILE_MACHINE_RISCV64	0x5064	RISC-V 64-bit address space
IMAGE_FILE_MACHINE_RISCV128	0x5128	RISC-V 128-bit address space
IMAGE_FILE_MACHINE_SH3	0x1a2	Hitachi SH3
IMAGE_FILE_MACHINE_SH3DSP	0x1a3	Hitachi SH3 DSP
IMAGE_FILE_MACHINE_SH4	0x1a6	Hitachi SH4
IMAGE_FILE_MACHINE_SH5	0x1a8	Hitachi SH5
IMAGE_FILE_MACHINE_THUMB	0x1c2	Thumb
IMAGE_FILE_MACHINE_WCEMIPSV2	0x169	MIPS little-endian WCE v2

Table 2 Machine Type of COFF

If the machine area is matching with the mark of the computer on which the file is run then the file can be run on a computer only.

Flag	Value	Description
IMAGE_FILE_RELOCS_STRIPPED	0x0001	The file must be loaded at its preferred base address because it does not allow base relocation.
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Set for valid files. Linker error if this is not set.
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	Deprecated. Set to zero.
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	Deprecated. Set to zero.
IMAGE_FILE_AGGRESSIVE_WS_TRIM	0x0010	Obsolete for Windows 2000 and later. Set to zero.
IMAGE_FILE_LARGE_ADDRESS_ AWARE	0x0020	Capable of handling addresses more than 2GB.
	0x0040	Reserved.
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	Little Endian. Deprecated. Set to zero.
IMAGE_FILE_32BIT_MACHINE	0x0100	Machine uses 32-bit architecture.
IMAGE_FILE_DEBUG_STRIPPED	0x0200	File does not have debug information.
IMAGE_FILE_REMOVABLE_RUN_ FROM_SWAP	0x0400	Copy the image to memory if it is on removable media.
IMAGE_FILE_NET_RUN_FROM_SWAP	0x0800	Copy the image to memory if it is on network media.
IMAGE_FILE_SYSTEM	0x1000	System File
IMAGE_FILE_DLL	0x2000	DLL File. Cannot be executed.
IMAGE_FILE_UP_SYSTEM_ONLY	0x4000	Only support uniprocessor machine.
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	Big Endian. Deprecated. Set to zero.

Table 3 COFF Available Attribute Flags

Optional Heading

Files that are considered executable have a further alternative fall. This header serves instructions to reference the current in the OS. Loaders handle the gassing of shootable files. This is a must-have for shootable files, but may also current in body files. Alternative heading in body files are of no use other than increasing the file size. The capacity of the alternative heading is specified by the "SizeOfOptionalHeader" area in the COFF heading. The illusion in the alternative heading determines whether the executable is PE32 or PE32+, as shown in Table 4. PE32+ executables allow 64-bit disk space, but cannot exceed 2 gigabytes in size. Standard fields for alternative headings detailed for each COFF implementation (Windows and Unix). The summary of the information contained in this section.

- The illusion number indicates whether the file is a normal executable (0x10B), a ROM image (0x107), or a PE32+ executable (0x20B).
- Linker version to use for this PE file.
- Code region indicates of a "PE file" that consists of the actual software that runs when the file is run. There may be several such code sections in the file. In that case,

the header field shows the total capacity of all code sections. Code regions are also known as .text regions in PE files.

- The capacity of initialized and uninitialized data contained in the file. It is also referred to as the .data Area of the PE file.
- File entry point address.

Magic number	PE format
0x10b	PE32
0x20b	PE32+

Table 4: Alternative heading Illusion Numbers

Offset (PE32/PE32+)	Size (PE32/PE32+)	Header part	Description
0	28/24	Standard fields	Common for Windows and Unix COFF implementations.
28/24	68/88	Windows-specific fields	Defines windows specific features.
96/112	Variable	Data directories	Address and size of special tables used by OS.

Table 5: Alternative heading Sections

Windows-specific areas consist of specific information about the Windows environment. It consists of the OS version, image version, header size, image size, DLL properties, loader flags, and length of data list. Kath, (1993) explain "Image capacity determines how much memory the operating system must reserve to run the image". The data directory provides the location and capacity of the directory appropriate by Windows. These include but are unlimited to export or import the tables, source tables, exclusion tables, and othe element.

Area Table

Each area in a PE file consists of an area header. Characterize of the region name, , a total of rows, virtual content and different pointers (rows, raw data, displacement, and other that.). In addition to the sections above, PE files consist of executable software code. A file may contain different sections, but that is beyond the scope of this work.

Offset	Size	Field	Description
0	8	Name	Name of the section represented as an 8-byte UTF-8 string.
8	4	VirtualSize	Size of the section in memory.
12	4	VirtualAddress	Refers to the address of the first byte when loaded to memory.
16	4	SizeOfRawData	Size of the uninitialized data on disk.
20	4	PointerToRawData	File pointer to the first page of the section.
24	4	PointerToRelocations	Set to zero for executable files.
28	4	PointerToLinenumbers	Deprecated. Set to zero.
32	2	NumberOfRelocations	Set to zero for executable files.
34	2	NumberOfLinenumbers	Deprecated. Set to zero.
36	4	Characteristics	Characterstic Flags

Basic data set

Since the approach proposed here is to evaluate the PE files, then to find records containing PE files marked as malicious is the first challenge. Records that classify malicious software as malicious and benign are crucial to the treasure trove. The data set is 9 GB and contains 900K tester samples, including 300K with high risk, 300K benign, and 300K cases that are unable. It also consists of 200,000 test patterns. This has also published the source code used to create this dataset.

```
{"sha256": "0abb4fda7d5b13801d63bee53e5e256be43e141faa077a6d149874242c3f02c2",
"appeared": "2006-12",
"label": 0,
"histogram": [45521,  13095,  12167 ..... 12170,  12596,  22356],
"byteentropy": [0, 0, 0, .......... 372116, 373375, 373929, 375883],
"strings": {
    "numstrings": 14573,
    "avlength": 5.972071639333013,
    "printabledist": [1046, 817, 877 ....... 845, 804, 900],
    "printables": 87031,
    "entropy": 6.569897560341239,
    "paths": 3,
    "urls": 0,
    "registry": 0,
    "MZ": 51},
"general": {
    "size": 3101705,
    "vsize": 380928,
    "has_debug": 0,
    "exports": 0,
    "imports": 156,
    "has_relocations": 0,
    "has_resources": 1,
    "has_signature": 0,
    "has_tls": 0,
    "symbols": 0},
"header": {
    "coff": {
        "timestamp": 1124149349,
        "machine": "I386",
        "characteristics": ["CHARA_32BIT_MACHINE", "RELOCS_STRIPPED" .....
        "LOCAL_SYMS_STRIPPED"]   },
    "optional": {
        "subsystem": "WINDOWS_GUI",
        "dll characteristics": [],
        "magic": "PE32",
        "major_image_version": 0,
        "minor_image_version": 0,
        "major_linker_version": 7,
        "minor_linker_version": 10,
        "major_operating_system_version": 4,
        "minor operating system version": 0,
        "major_subsystem_version": 4,
        "minor_subsystem_version": 0,
        "sizeof_code": 26624,
        "sizeof_headers": 1024,
        "sizeof_heap_commit": 4096},},
"section": {
     "entry": ".text",
     "sections": [{
            "name": ".text",
            "size": 26624,
            "entropy": 6.532239617101003,
            "vsize": 26134,
            "props": ["CNT_CODE", "MEM_EXECUTE", "MEM_READ"]}
        .....
        {
            "name": ".rsrc",
            "size": 27648,
            "entropy": 5.020929764194735,
            "vsize": 28672,
            "props": ["CNT_INITIALIZED_DATA", "MEM_READ"]}]},
"imports": {
    "KERNEL32.dll": ["SetFileTime", "CompareFileTime" ..... "GetTempPathA", "MulDiv"]
    .....
    "snmpapi.dll": ["SnmpUtilOidCpy", "SnmpUtilOidNCmp", "SnmpUtilVarBindFree"]},
 "exports": []}
```

Figure 2: Example JSON

PROPOSED MODELS

Three main components of this approach are "Feature Extraction and Quarrel" (FEQ), "Scaling and Normalization" (SN), and "Neural Network Classifier" (NNC). To implement this research model using Python because of its integrity and resilience of adoption in machine learning applications Oliphant, (2007). Collet et al, (2015) describe the Nvidia CUDA architecture by Nickolls, (2008) as utilized parallel computing with fast speed using the "Keras library" to implement the proposed NN model.

FEQ

The main components of the PE file extracted to track the model is "Resolve Information" and "Basic Byte Information". The sets of data used in this research approach give a beneficial caliber to cutting the vital data from PE file. In total, the model used is 2351 input vectors for classification.

Resolve Information

"Each PE file consists of header information, these features are extracted in python using the LIEF library to parse PE files" (Thomas, 2017). Not all numeric is not always of the same size in this material. This research model uses a fixed-capacity input vector for exercise. It means, all of aspects cutting must be of default size before being used to train the model, to achieve it, "the module from the sci-kit-learn library **`FeatureHasher`**" Pedregosa, (2011) is also "used to implement the element quarrel trick" Weinberger et al, (2009) along a positive total number of bins per element header and 5 sets of features were extracted from the PE file

Common features obtained from PE files include:

- Virtual size
- Imported function
- Exported function
- The presence of a debug section
    - Resource
    - Relocation
- Number of Symbols
- This is the basic information obtained from the PE header.

Header Information

- Specific information from the headers in the PE file is obtained, and from the COFF header the following information is obtained:
- Timestamp
- Target Machine (rope)
    - Image Characteristics List (string list)
- From the alternative heading obtained:
    - Subsystem target
    - Characteristics of a DLL
    - Illusion Number
    - Main Image Vers.
    - Thumbnail Vers.
    - Linker vers.
    - System Vers.

- Subsystem Vers.

Features containing strings are first converted to a byte representation and then parsed through a feature hash to produce 10 fields of summary data. This will vectorize your data to an established size. Location table imported from alternative heading along with imported functions sorted by library cut from "PE file". This data is summarized using "`FeatureHasher`". Total of 256 bins used to summarize all unique libraries and 1024 bins are used to represent pairs of libraries: "**`FunctionName`**". All exported functions are extracted as strings and then encapsulated by 128 boxes using FeatureHasher.

- Area Information
- The Area Table is copied following information:
  - Area Name
  - Virtual size
  - Size
  - Entropy
  - Virtual size
  - Area Characteristics (string list)
  - Entry Point

A quarrel trick is used for the label. This combination are aggregated applying a "`FeatureHasher`", with 50 bins assigned to each set of values. The part attribute is collected personally using the same quarrel trick.

Basic Byte Information

Consists of basic byte information for analysis of platform-independent PE files. This can also be used to classify malware that is not intended for Windows environments. This study did not aanalyses the efficiency of this model with non-Windows malicious software, so it does not address the persuasiveness of this approach in similar cases. The byte information extracted from the file is independent of the file type. This is a simple representation of all the bytes that make up the file. Because these bytes vary in size, this study summarizes this data using the approach perspective by Saxe and Berlin (2015). This method is also implemented in the Extract Features from the Dataset module. A byte histogram is the number of times each unit value occurs in the file. Bytes can have values from 0 to 255. That is, the histogram for that byte consists of 256 possible occurrences of the integer value. Forward a capacity of windows in 2048 bytes and a step of 1024 bytes, we can slide this window over the file to compute the entropy histogram, compute the "Shannon H entropy" of the 2048-byte window, and plot the joint distribution of this window and each byte To do. it contains.

Result: List of size 256 containing Byte Histogram
Initialize *file*
Initialize list *count*[256]
while *file* ≠ *EOF* **do**
    bytes ← *file.read*(*bufsize*)
    foreach *value in bytes* **do** Increment *count*[*value*]

end

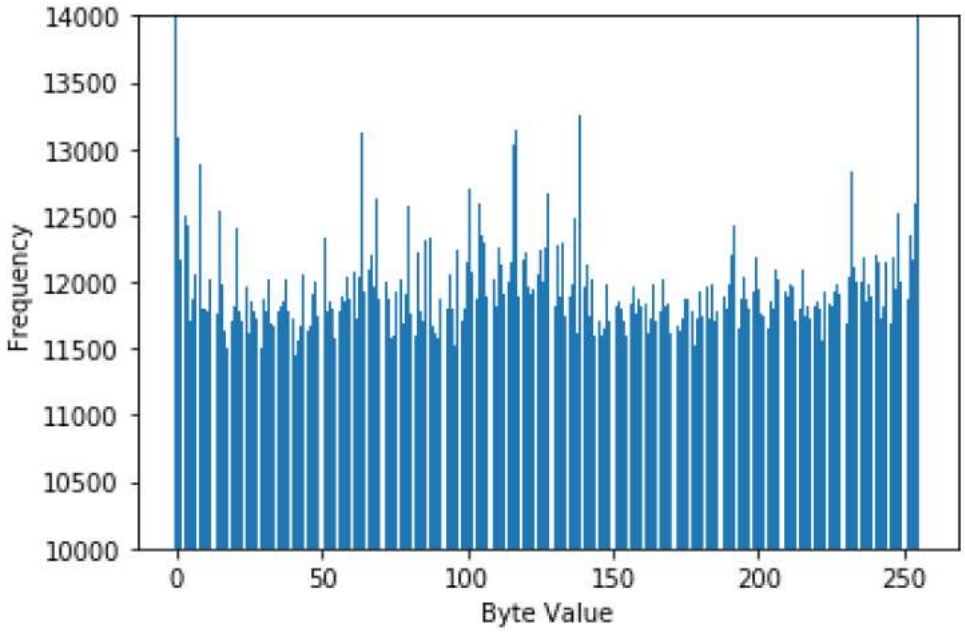

Figure 3 "Byte Histogram"

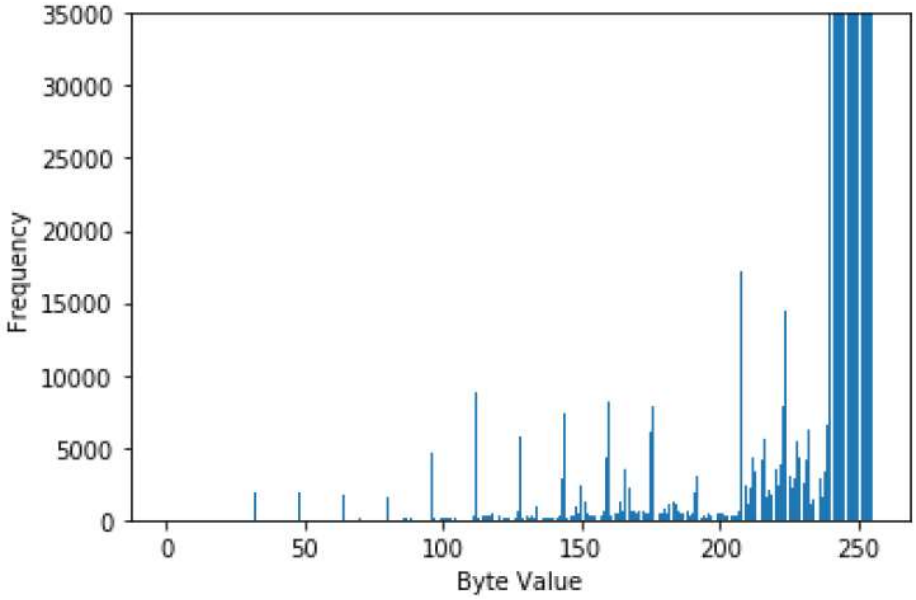

Figure 4 "Byte-Entropy Histogram"

Polite String

All of PE files consist of ability that are beneficial data sources. All abilities consisting of more than five polite elements are excerpted from the file. The excerpted ability are checked the

content to build the rooted of functions that bring summary statistics about the string content of the file. Only several components composed in the class used as classical appearance. The categories are as follows:

- C:\ (case sensitive) represents the Windows path
- HTTP:// or HTTPS:// denotes a URL
- HKEY_ denotes the Windows registry key

This serves two purposes: it can reveal certain characteristics of the file that are not revealed from the header information, and it protects the privacy of benign files because the study only summarizes string data.

After deriving the appearance from the PE file, 2351 features are obtained per sample. However, this feature value is widespread along a total whose price is close to zero, over (109), and down (-104). These aspects are illustrated as a line graph in Fig. 5. A scatterplot of these aspects is difficult to see the extreme values, so a line graph should be drawn.

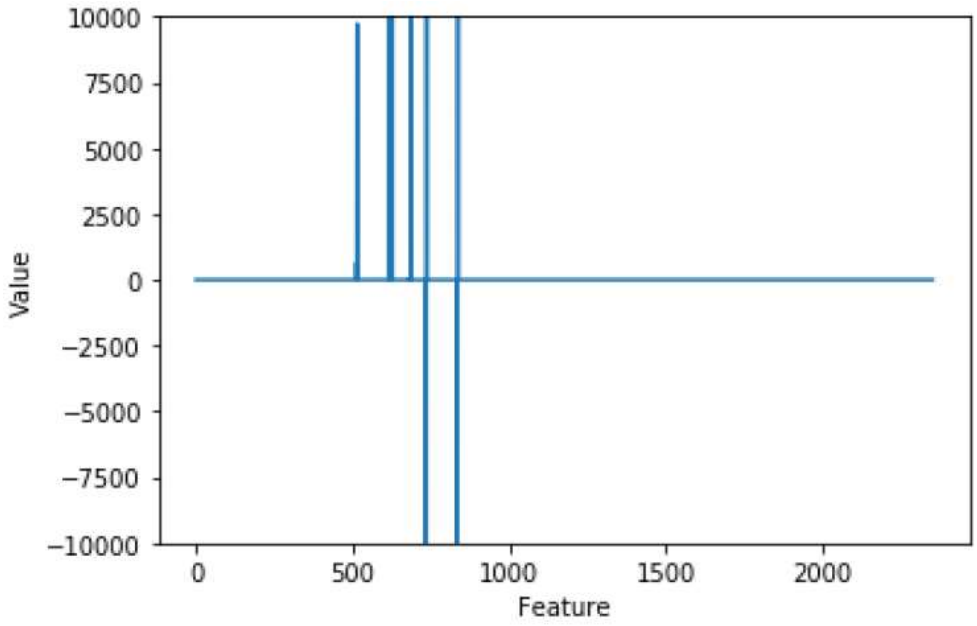

Figure 5 A raw sample line graph

When trying to practice an example with this data type, here the model does not assemble and returns an A-UC of zero: five. This is obvious because the scale of the features being extracted varies so much that it requires any model of regularization back used in exemplary. This study uses analytical normalization by Jayalakshmi and Santhakumaran, (2011).

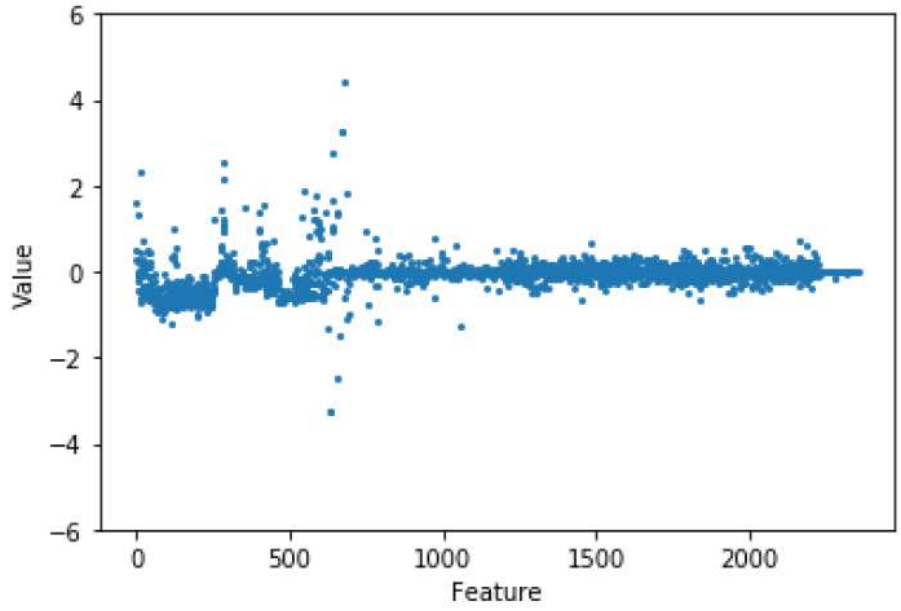

Figure 6 Scatter plot of the normalized sample

$$\sigma(X) = \sqrt{\frac{1}{2351}\sum_{l=1}^{2351}(X_I - \mathcal{X})^2}$$

Here, Pedregosa et al, (2011) "using the '**`StandardScalar function`**' provided '**`scikit-learn library`**' to normalize the sample is used". This operation achieves the same computations as the supposed raised.

NNC

In this study, DNN was used to analyze the data. Two NNs are assembled, one with a dropout layer and one without a dropout layer. Logistic and rectified linear unit (ReLU) activation functions were tested for these two networks. The optimizer supplied in the Keras library by Adam and Kingma (2014) was "used for gradient-based optimization of classifiers". A schematic of the two NN is shown in Figures 7 and 8. A network without dropout layers consists of one input layer accepting capacity, DNL, and a BOL. The alternative network with two dropout layers and two high-density layers is the improvement of introducing dropouts with fewer layers. increase.

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 2400)	5644800
dense_2 (Dense)	(None, 1200)	2881200
dense_3 (Dense)	(None, 1200)	1441200
dense_4 (Dense)	(None, 1200)	1441200
dense_5 (Dense)	(None, 1)	1201

Total params: 11,409,601
Trainable params: 11,409,601
Non-trainable params: 0

Figure 7 Summary NN

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 2400)	5644800
dropout_1 (Dropout)	(None, 2400)	0
dense_2 (Dense)	(None, 1200)	2881200
dropout_2 (Dropout)	(None, 1200)	0
dense_3 (Dense)	(None, 1200)	1441200
dense_4 (Dense)	(None, 1)	1201

Total params: 9,968,401
Trainable params: 9,968,401
Non-trainable params: 0

Figure 8: Summary of a NN with dropout layers.

Models Summary

The model summary diagram is shown in Figure 9. The entire model subsists:

- Extract headers and platform-independent functions from PE files.
- Use quarrel tricks to encapsulate header features Weinberger et al, (2009).
- Reduce the appearance to a 1-spatial aim input.
- Feature normalization uses statistical normalization or Z-score.
- Pass an aim input over a densely connected DNN.

Final outcome is zero/one.

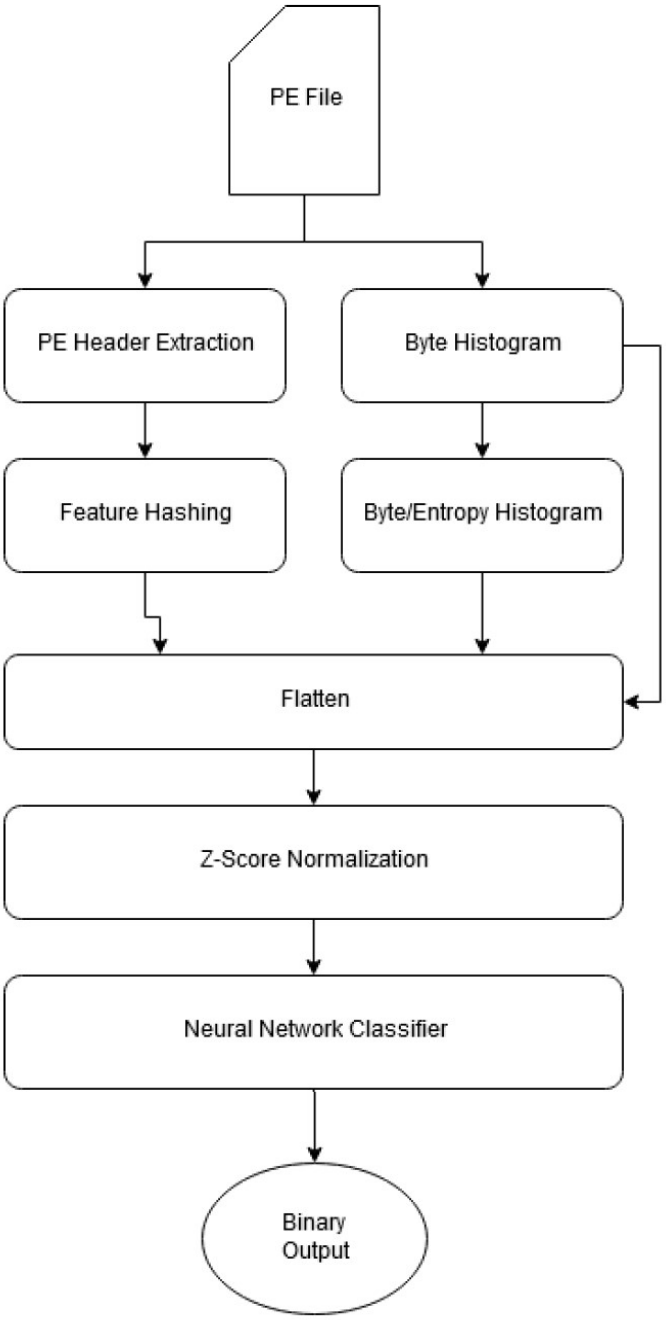

Figure 9 Model flowchart

RESULTS AND TESTING

Experiment settings

The model used in this study continues to qualify “Dell Precision Tower” and “Intel Xeon E3” CPU, “Nvidia GeForce GTX 1080 Ti”, and “64 GB of RAM”. This is then implied in “Python with the libraries installed: TensorFlow” by Abadi et al, (2016), "Loud” by Chollet et al, (2015), “NumPy” by Walt, (2011), “scikit-learn” by Pedregosa et al, (2011), “LIFE” by Thomas, (2017), “Pandas” by McKinney, (2010) and “MatPlotLib” by Hunter, (2007). Some packages and libraries bet on the raised, but usually, all of that is automatically installed as an element of the instalment

action using Anaconda Python with Python vers- 3.6.7 for all experiments (Anaconda Software 2016).

Metrics - Testing Model

Before testing the model, it is important to identify the metrics to be used for this purpose. In this case, this study tested the accuracy and diagnostic capability of the model present in the study, "receiver output characteristic" was obtained and the "area under the curve"/"AUC" was found. This method generally provides a better measure of classier diagnostic ability than commonly stating the overall accuracy of the model contrary to a given test set by Fawcett, (2006:861) and Metz, (1978:283). The distraction matrices of the NN-based model and the decision tree-based model were also found first to generate clear to find cases of misclassification that are not obvious from the R-OC curve. The R-OC curve was derived by plotting the "true positive rate" (TPR) of the classifier contrary to the "false positive rate" (FPR). TPR and FPR are calculated using the following equation:

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{P} = \frac{FP}{FP + TN}$$

Where TP is a true positive, P is the total positive samples present in the test set, and FN is a false negative. FP is a false positive, N is the total negative samples present in the test set, and TN is a true negative. `roc_curve, A-UC` is used and the module by "sci-kit `-learn.metrics"` to get the R-OC curve, and the `fusion_matrix module from` the same library to get the distraction matrix`.` All data is plotted using MatPlotLib. The 200K test sample provided in the dataset is used to test the model used and the results are compared with a decision tree-based model using LightGBM (Guolin et al, 2017).

Test results

Classifier Type	AUC	TPR @ FPR = 0.01
Neural Network (Sigmoid)	0.998	0.981
Neural Network with Dropout (Sigmoid)	0.998	0.978
Neural Network (ReLU)	0.997	0.982
Neural Network with Dropout (ReLU)	0.997	0.989
Decision Tree using LightGBM	0.999	0.982

Table 6 short of outcomes for an NN-based classifier, and a decision tree-based classifier

To test this research model using four diverse NN-based classifiers testing it using a decision tree-based classifier. A summary of the results can be seen in table 6, as can be seen from the R-OC curves in Figures 10, 12, 13, and 15, the A-UC of the NN using the ReLU activation function is slightly lower than that using the sigmoid activation function. In addition, although the A-UC of the decision tree classifier is the highest, the true positive rate of this model is the same or lower when limited to a 1% false positive rate compared to a ReLU-based NN. The distraction

matrices for the models in Figures 11, 13, 15, 17, and 19 serve to improve the decision of the analysis work for different models. As ReLU appeared to collect the best results, the R-OC curves of the ReLU-based NNs were compared, and the decision tree classifiers were in Fig. 19 and 20.

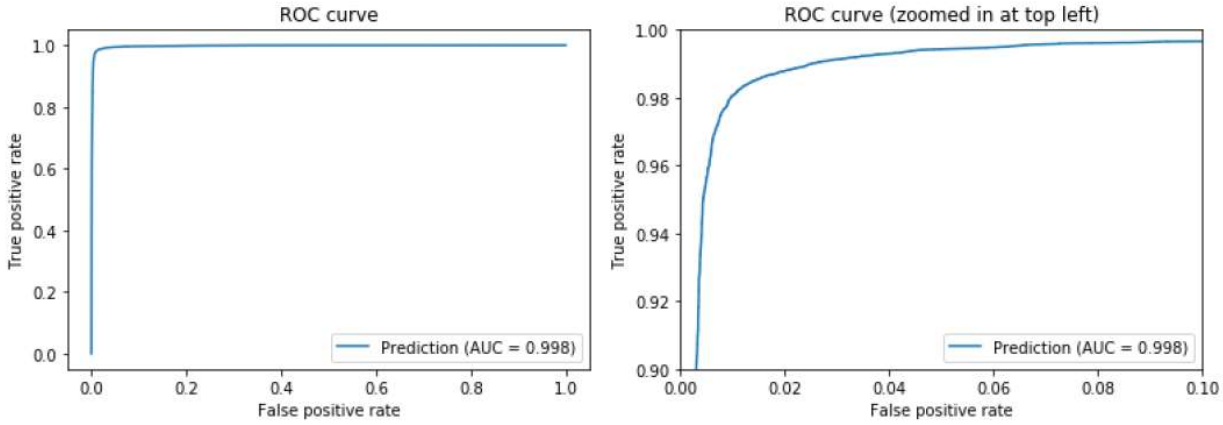

Figure 10 R-OC Curve model using a NN (Sigmoid)

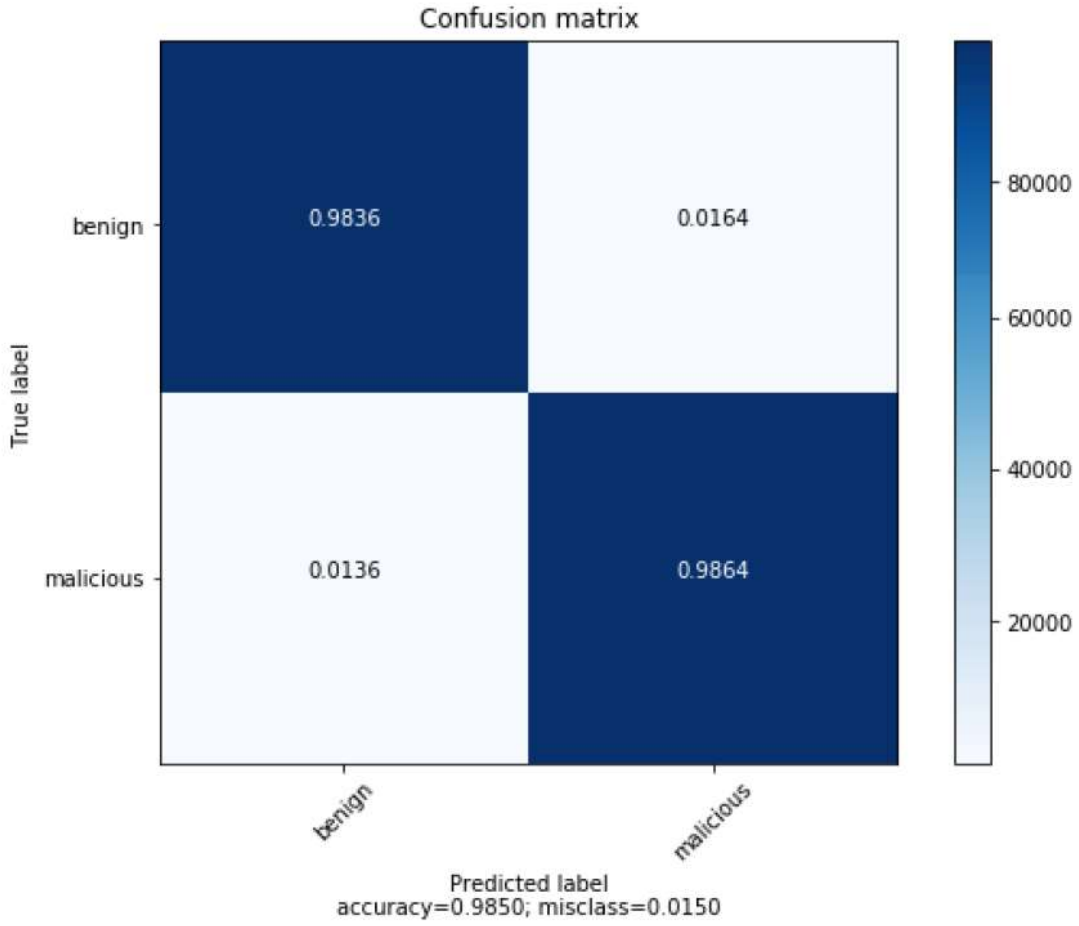

Figure 11 Aberration matrix model using NN (sigmoid)

Real World Testing

Next, this study tests the first-operating model of the recommended model (RelU)) contrary to a resolution tree model to check how well it performs in detecting real malicious PE files and

the results are summarized in Table 7. For testing, this research uses a sampling bath of nine hundred and ninety-seven samples from VirusShare.com Robert, (2011).

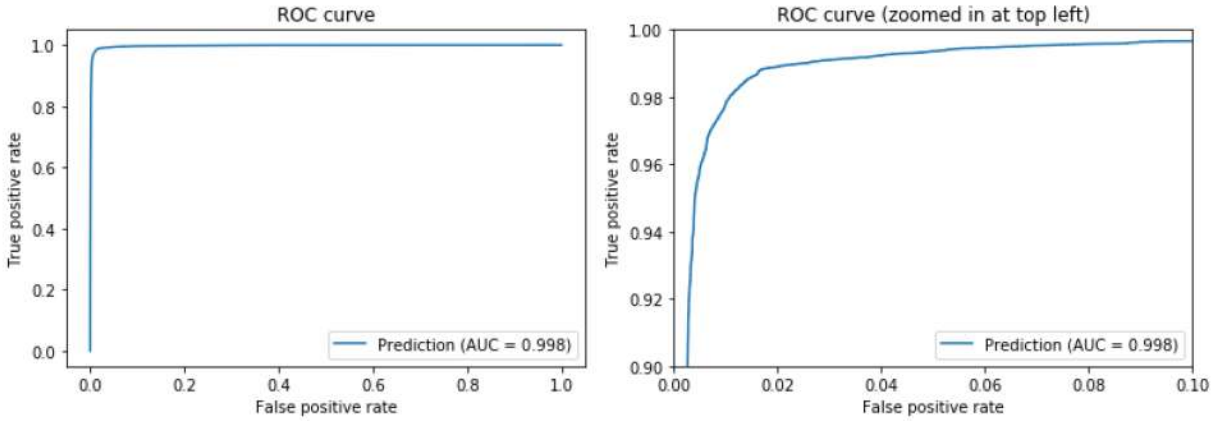

Figure 12 R-OC model curve (sigmoid) using NN with dropout

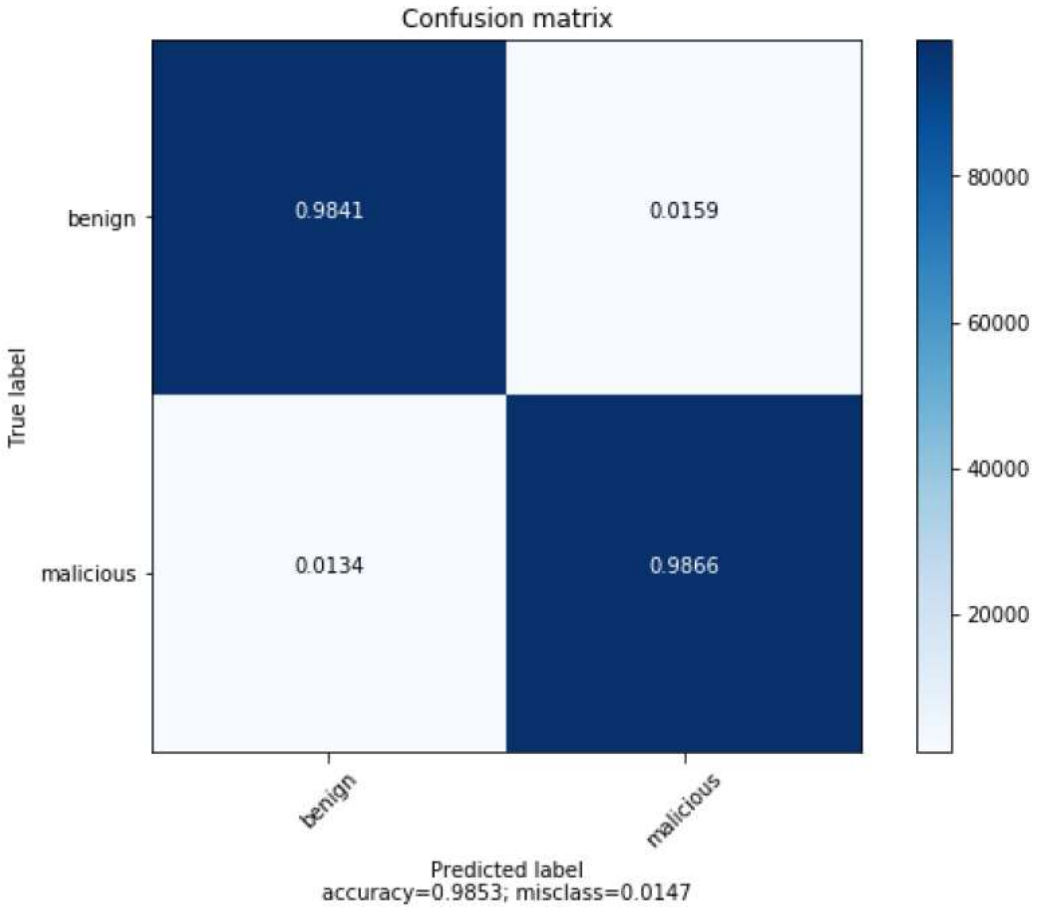

Figure 13 Sigmoid - Aberration matrix model using NN with Drop-out

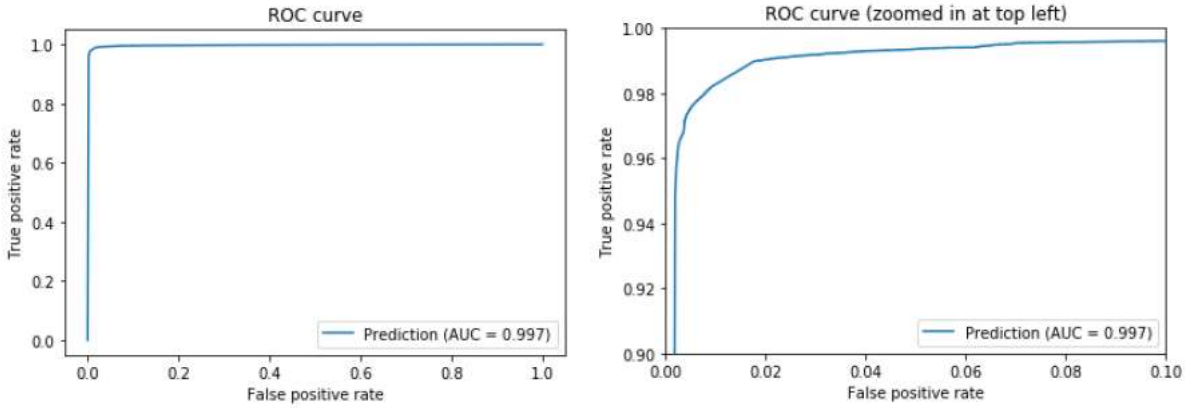

Figure 14 ReLu - R-OC Curve Model using NN

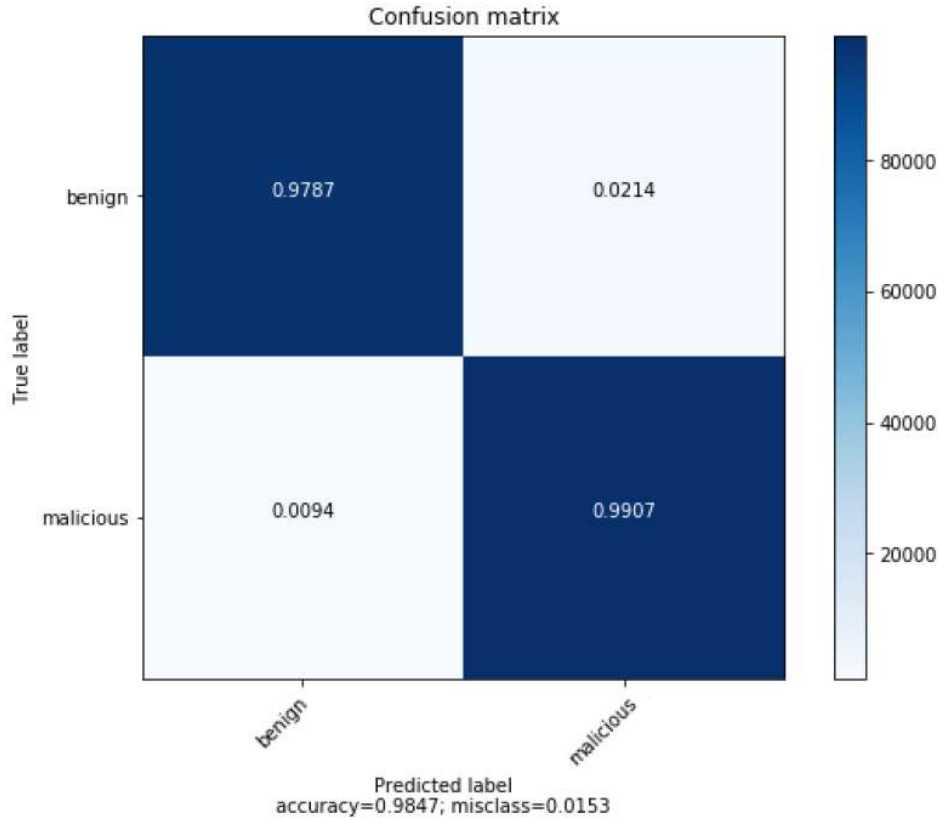

Figure 15 ReLU - Aberration matrix model using NN

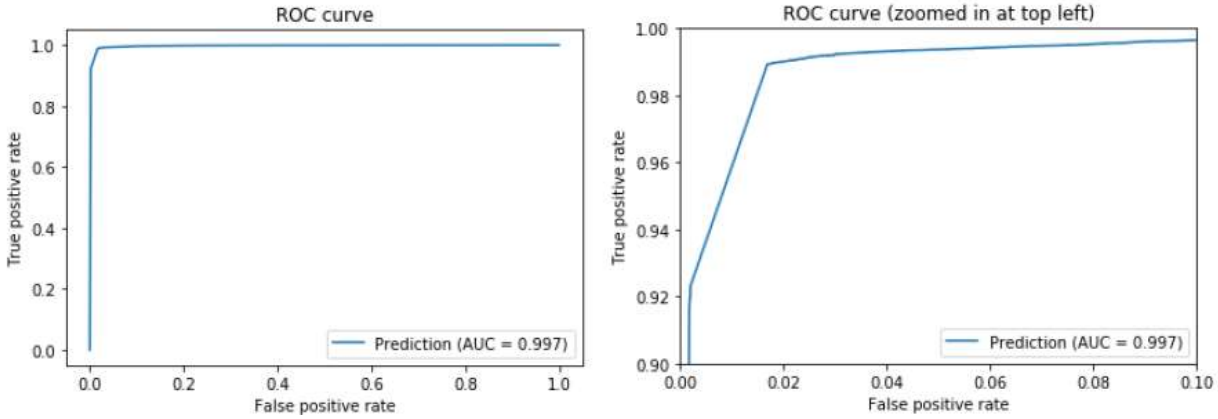

Figure 16 ReLU – model R-OC curve using NN with Dropout

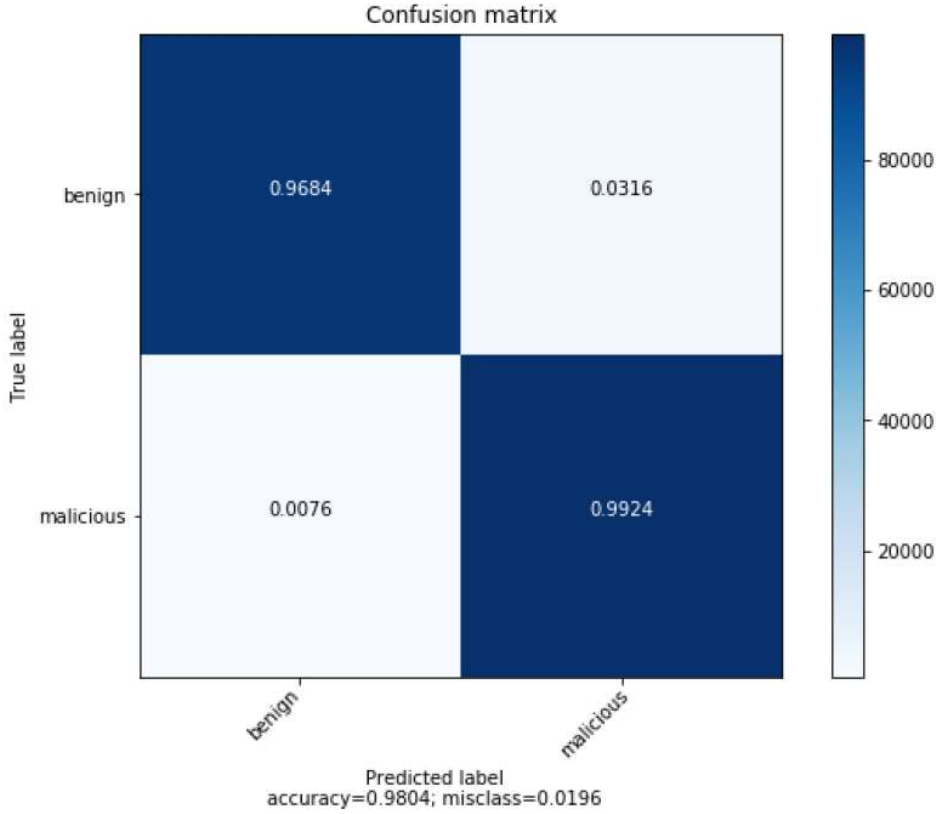

Figure 17 ReLU - Aberration matrix model using dropout NN

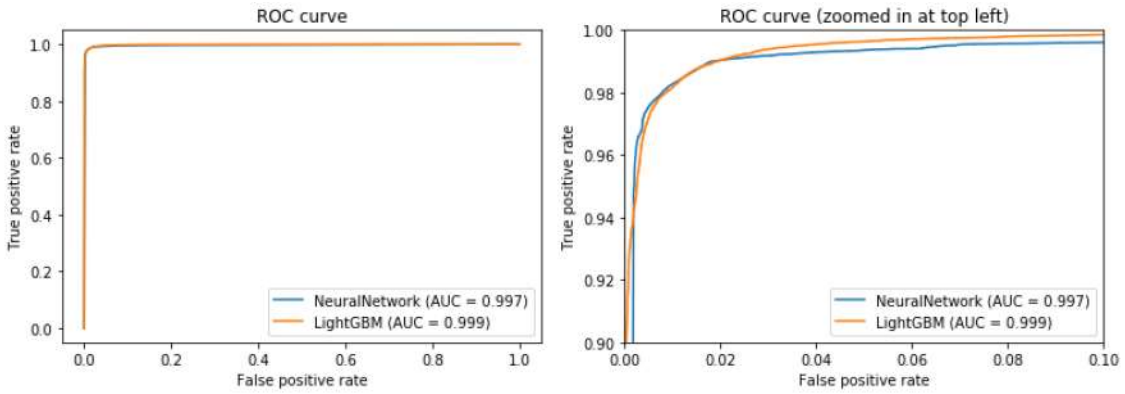

Figure 18 ReLU - R-OC curve model using NNs and decision trees.

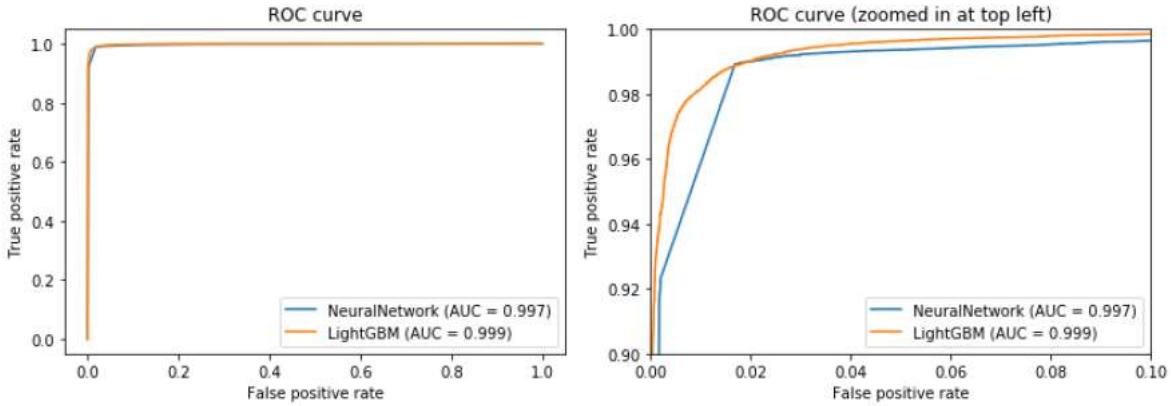

Figure 19 ReLU - The R-OC curve model uses dropout NN and uses decision trees.

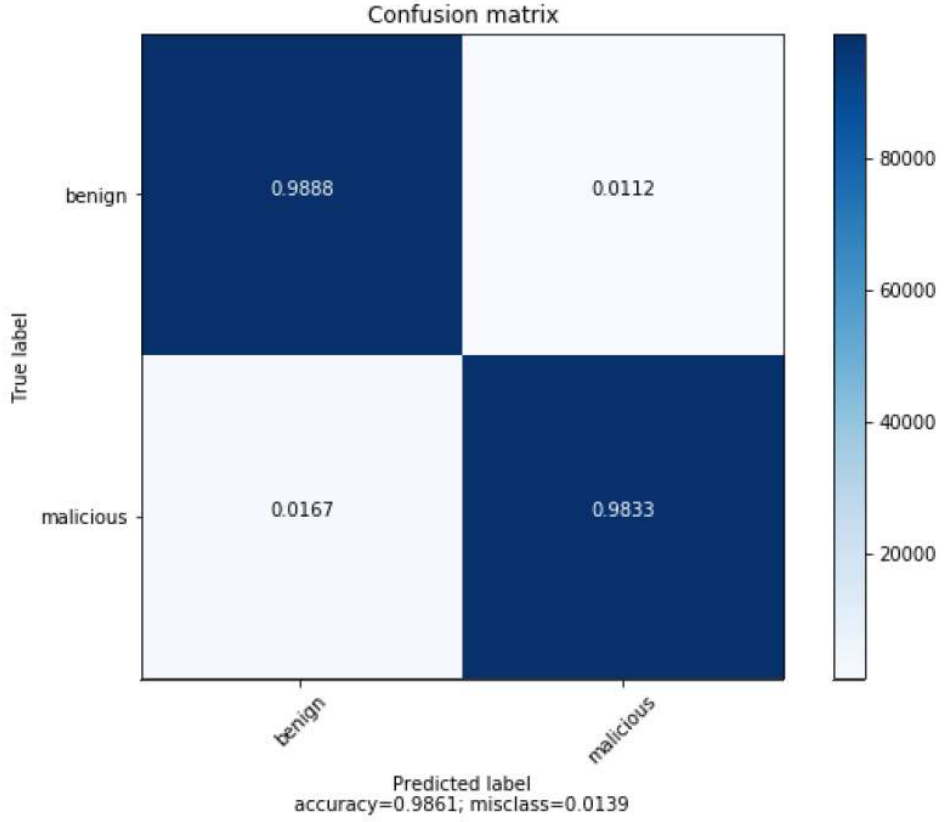

Figure 20 Aberration matrix models Using Decision Trees

Model Classifier	Execution Time (seconds)	Accuracy
Neural Network with Dropout (ReLU)	128.2	0.997
Decision Tree	133.6	0.117

Table 7. Actual test result

CONCLUSIONS

In this research, it is shown that the use of DNNs for advanced malicious software disclosure is feasible and has the possibility for more enhancements. Experiments in ionic research spectacle that same in position compelling measured data, NNs can still be used more efficiently than decision trees. 5 atomization method that can completely recap wide files for distribution is defined in this study. The interest in having large datasets available in the domain cant be forgotten. This study present that advanced malicious software investigation can be an active tool for classifying malicious software regardless of the presence and level of detection of effective malicious software study. increase.

For future suggestions, further research is needed in this area to determine how efficiently a NN can be a classifier for measured data related to a resolution tree model. Testing under here-and-now conditions has presented that there are closed unexplored gaps in this area that require further testing for practical implementations.

REFERENCE

Ammar AE Elhadi, Mohd A Maarof, and Ahmed H Osman. Malicious software detection based on hybrid signature behavior application programming interface call graph. American Journal of Applied Sciences, 9(3):283, 2012.

Anaconda. Anaconda software distribution version 2-2.4.0, November 2016.

Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malicious software detection. In Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pages 421-430. IEEE, 2007.

Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malicious software system call sequences. In Australasian Joint Conference on Artificial Intelligence, pages 137-149. Springers, 2016.

Byron P Roe, Hai-Jun Yang, Ji Zhu, Yong Liu, Ion Stancu, and Gordon McGregor. Boosted decision trees as an alternative to artificial NNs for particle identification. Nuclear Instruments and Methods in Physics Research Area A: Accelerators, Spectrometers, Detectors, and Associated Equipment, 543(2-3):577-584, 2005.

Charles E Metz. Basic principles of R-OC analysis. In Seminars in nuclear medicine, volume 8, pages 283-298. Elsevier, 1978.

Christopher Manning, Prabhakar Raghavan, and Hinrich Schutze. Introduction to information retrieval. Natural Language Engineering, 16(1):100-103, 2010.

Daniel Billar. Opcodes as predictors for malicious software. International Journal of Electronic Security and Digital Forensics, 1(2):156-168, 2007.

David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malicious software. In Botnet Detection, pages 65-88. Springer, 2008.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

Dilshan Keragala. Detecting malicious software and sandbox evasion techniques. SANS Institute InfoSec Reading Room, 16, 2016.

Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. Journal of machine learning research, 12(Oct):2825-2830, 2011.

Francois Chollet et al. Hard. https://keras.io , 2015.

Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Control flow graphs as malicious software signatures. In International workshop on the Theory of Computer Viruses, 2007.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In Advances in Neural Information Processing Systems, pages 3146-3154, 2017.

Hamid Divandari, Bassir Pechaz, and Majid Vafaie Jahan. Malicious software detection using Markov blanket based on opcode sequences. In the 2015 International Congress on

Igor Santos, Jaime Devesa, Felix Brezo, Javier Nieves, and Pablo Garcia Bringas. Open A static-dynamic approach for machine-learning-based malicious software detection. In International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions, pages 271-280. Springers, 2013.

JD Hunter. Matplotlib: A 2d graphics environment. Computing in Science & Engineering, 9(3):90 -95, 2007.

Jeremy Z Kolter and Marcus A Maloof. Learning to detect malicious executables in the wild. In Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 470-478. ACM, 2004.

J-Michael Roberts. Virus share.(2011). URL https://virusshare.com , 2011.

John Nickolls, Ian Buck, and Michael Garland. Scalable parallel programming. In 2008 IEEE Hot Chips 20 Symposium (HCS), pages 40-53. IEEE, 2008.

Jon Oberheide, Michael Bailey, and Farnam Jahanian. Polypack: an automated online packing service for optimal antivirus evasion. In Proceedings of the 3rd USENIX conference on Offensive technologies, pages 9. USENIX Association, 2009.

Joshua Saxe and Konstantin Berlin. DNN-based malicious software detection using two-dimensional binary program features. In 2015 10th International Conference on Malicious and Unwanted Software (MALICIOUS SOFTWARE), pages 11-20. IEEE, 2015.

Karthik Raman et al. Selecting features to classify malicious software. InfoSec Southwest, 2012.

Katherine Heller, Krysta Svore, Angelos D Keromytis, and Salvatore Stolfo. One class support vector machines for detecting anomalous windows registry accesses. In ICDM Workshop on Data Mining for Computer Security, 2003.

Kilian Weinberger, Anirban Dasgupta, Josh Attenberg, John Langford, and Alex Smola. Quarrel feature for large-scale multitask learning. arXiv preprint arXiv:0902.2206, 2009.

M. Sikorski and A. Honig. Practical Malicious software Analysis: The Hands-On Guide to Dissecting Malicious Software. No Starch Press, 2012.

Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malicious software-analysis techniques and tools. ACM computing surveys (CSUR), 44(2):6, 2012.

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensor flow: A system for large-scale machine learning. In 12th fUSENIXg Symposium on Operating Systems Design and Implementation (fOSDIg 16), pages 265-283, 2016.

Michele Banko and Eric Brill. Scaling to very very large corpora for natural language disambiguation. In Proceedings of the 39th annual meeting on association for computational linguistics, pages 26-33. Association for Computational Linguistics 2001.

Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. Technical report, WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES, 2006.

Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malicious software detection. ACM SIGPLAN Notices, 42(1):377- 388, 2007.

Bagga's name. Measuring the effectiveness of generic malicious software models. Master's thesis, San Jose State University, 2017.

Philip OKane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The hidden malicious software. IEEE Security & Privacy, 9(5):41-47, 2011.

Randy Kat. The portable executable file format from top to bottom. MSDN Library, Microsoft Corporation, 1993.

Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malicious software classification with recurrent networks. In 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 1916 -1920. IEEE, 2015.

Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In Proceedings of the 23rd international conference on Machine learning, pages 161-168. ACM, 2006.

Robert E Schapire. The boosting approach to machine learning: An overview. In Nonlinear estimation and classification, pages 149-171. Springer, 2003.

Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com/, April 2017.

Ross Quinlan. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi.

Srilatha Attaluri, Scott McGhee, and Mark Stamp. Profile hidden Markov models and metamorphic virus detection. Journal in computer virology, 5(2):151-169, 2009.

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2):22, 2011.

T Jayalakshmi and A Santhakumaran. Statistical normalization and backpropagation for classification. International Journal of Computer Theory and Engineering, 3(1):1793-8201, 2011.

Technology, Communication, and Knowledge (ICTCK) page 564-569. IEEE, 2015.

Tom Fawcett. An introduction to R-OC analysis. Pattern recognition letters, 27(8):861-874, 2006.

Travis E Oliphant. Python for scientific computing. Computing in Science & Engineering, 9(3):10-20, 2007.

Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. Adaboost multi-class. Statistics and its Interface, 2(3):349-360, 2009.

Wen-Chieh Wu and Shih-Hao Hung. Droiddolphin: A dynamic android malicious software detection framework using big data and machine learning. In Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems, RACS '14, pages 247-252, New York, NY, USA, 2014. ACM.

Wenyi Huang and Jack W Stokes. Mtnet: a multi-task NN for dynamic malicious software classification. In International Conference on Detection of Intrusions and Malicious software, and Vulnerability Assessment pages 399-418. Springers, 2016.

Wes McKinney et al. Data structures for statistical computing in python. In Proceedings of the 9th Python in Science Conference, volume 445, pages 51-56. Austin, TX, 2010.