Machine Learning Approaches for Malware Classification based on Hybrid Artefacts

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

Rajchada Chanajitt

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2023

Abstract

Malware could be developed and transformed into various forms to deceive users and evade antivirus and security endpoint detection. Furthermore, if one machine in the network is compromised, it could be used for lateral movement–when malware spreads stealthily without sending an alarm to monitoring systems. Malware attacks pose security threats to modern enterprises and can cause massive financial, reputation, and data loss to major enterprises. Therefore, it is important to detect these attacks effectively to reduce the loss to the minimum level. The current research uses different approaches, including static and dynamic analysis, to detect and analyze malware categories using distinct feature sets, such as imported modules, opcodes, and API calls, which can improve performance in binary and multi-class classification problems.

This thesis proposes a method for identifying and analyzing malware samples via static and dynamic approaches, including memory analysis and consecutive application operation sequences performed on the Windows 10 virtual environment. Standard classifiers and frequently used sequence models are utilized to expose the malware characteristics and benefit predictive capabilities. The features used in these algorithms are extracted from the static and dynamic analysis of malware samples, such as the rich header feature, debug information, temporary files, prefetch files, and event logs. The measurement of the classifiers and the degree of correctness are calculated using the accuracy, f1-score, Mean Absolute Error (MAE), confusion matrix, and Area under the ROC Curve (AUC). Combining two feature sets can provide the best classification performance on static file properties and dynamic analysis results, regardless of whether applying feature selection or not, achieving the accuracy and f1_score at 97% for integrating two datasets. For consecutive sequences, concatenating the Gated Recurrent Unit (GRU) and Transformers model can yield the highest accuracy at 97% for Noriben operations, while GRU can achieve the maximum accuracy for Opcode sequences at 89%.

Acknowledgements

Firstly, I would like to thank my supervisors: Bernhard Pfahringer and Heitor Gomes. Throughout my time as a PhD student, they had encouraged me to explore the topics that interested me most while providing useful feedback and guidance. I appreciated Bernhard and Heitor for taking me on as a student, even though my initial topic was not aligned with their usual research interests. I was grateful to have been around many interesting people from the Machine Learning group in the Department of Computer Science. A special thank you went to Vithya Yogarajan, for the many thought-provoking conversations and for guidance. The Department of Computer Science had been a wonderful environment to work in as a student, and I would like to extend my gratitude to all those who have made my time there an enjoyable period of my life. Lastly, I would like to thank my family and other friends for the encouragement and support throughout my studies, for asking me how my research was going when I started my Ph.D., and for not asking as I drew closer to finishing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Malicious software, commonly known as malware, is any program designed to disrupt normal functions on computers, servers, and networks to gain access to sensitive information and unauthorized resource usages, such as mining and interrupting services. With global internet access and a dramatic increase in internet usage, malware becomes a significant security threat to society. In addition, attackers are becoming sophisticated in their methods to conceal their malicious payloads and evade security mechanisms. For example, cybercriminals send spam emails with infected file attachments or fake websites with fraudulent content to deliver their malicious payloads. Malware attacks can cause a disastrous financial impact on businesses through both direct and indirect costs, such as over $49,207,908 in losses caused by ransomware in 2021 [1]. Such security detection is mainly divided into signature-based, behavior-based, statistical-based, heuristic techniques, and anomaly-based [2]. Each detection technique has its advantages and disadvantages. For signature-based, it is effective against known attacks, but it requires an updated database to recognize unknown malware. For anomaly-based, it can identify changes in behavior from the baseline and reduce false positives resulting from signature-based; however, high false alarm rates can be generated if a reliable baseline is not built [2].

Malware signatures are one of the essential resources provided by security vendors. On the one hand, the signatures could be obtained by inspecting executables statically. On the other hand, installing a monitoring tool to record malware behavior is an alternative way, then generating features describing its dynamic behavior. Provided that debugging is enabled, the code syntax could reveal anti-detection techniques implemented to thwart reverse engineering and cause damages after the system is infected. When unseen malware appears, sophisticated skill is required to analyze and dissect the malware parts, to determine the position of the actual code execution and the embedded payload of the main application. The effectiveness of malware detection results could be improved using machine learning techniques with extensive data analysis [3–5].

Malicious binaries in the Windows Portable Executable (PE) files seriously threaten organizations. Windows executables, object code, and DLLs use the Portable Executable (PE) file format [6]. This thesis analyzes EXE files due to being standalone programs loaded as a new independent process. In contrast, DLL files need to be loaded into a process and cannot be run independently. Therefore, the execution of a DLL file depends on the OS loader having to have created a process that requires using some functions contained in the DLL file. Regarding malware analysis techniques, they consist of static, dynamic, and hybrid. The static approach mainly focuses on static properties using standard data extraction, such as running a strings command to examine printable characters [7, 8], whereas disassembling the assembly instructions using static analysis tools, such as IDA pro disassembler and Python-developed modules to collect static opcodes and API calls, is an advanced static analysis [9, 10]. However, for a better understanding of applications' inner workings, dynamic analysis plays an essential role by executing suspicious programs within a virtual sandbox environment and collecting their behavior during the run-time. Additionally, analysis of malicious code in-depth with interactive mode can be achieved during advanced dynamic analysis by using a debugger with setting breakpoints to analyze functions for encoding and encryption throughout a malware sample and manually unpacking a packed malware sample. To obtain complete information on malware analysis, integrating two approaches (hybrid) can result in a higher detection rate of malware [11, 12]. This thesis uses three approaches to obtain malware characteristics and behavior.

With the sheer volume of malware instances, machine learning algorithms plays a vital role in extracting significant features and tackling problems. Machine learning is the automatic process of discovering hidden insights in data fabric by using algorithms that could find those insights without being specifically programmed to create models that solve a particular (or multiple) problem(s) [13]. For example, deep convolutional neural networks (CNNs) proved their effectiveness in binary malware detection through image classification [14]. In addition, machine learning competitions were opened for data scientists, allowing access to the public dataset to build their learning models, submit their predictive results, and finally announce the winner who could achieve the highest classification performance. The public dataset released for security researchers was available for download via GitHub [15] or Amazon S3 [16]. This campaign could attract many researchers to focus on malware classification and propose innovative approaches. Among malware categories, the learning model could expose malware's activities—for example, injected code, downloaded files, unpacked contents of packed programs, and network connections. In this thesis, we download binaries from the public repository and use our custom Python script to extract the features.

In this thesis, we aim to meet two main purposes: (1) to better standard classifiers for predicting binary classes from static and dynamic artifacts and (2) to boost the accuracy of predictions of multiclass malware categories for long Noriben and Opcode sequences. The rest of this chapter discusses the previously mentioned concepts in more detail and also summarises the contributions and structure of the remainder of the thesis.

1.1 Overview and Challenges

Statistics from NetMarketShare [17] indicated that Windows 10 was the most popular platform in academic and business aspects, at around 90% of computer users. Additionally, Windows 10 was updated with new security features, compared to prior versions. Due to a popular attack platform, only Windows executables are taken in the experiments. Targeting samples on other operation systems requires tools and techniques to perform data extraction and preparation for different data types. In this thesis, we analyze malicious files on Windows 10 (version 1903) to comprehensively understand integrated static and dynamic attributes and a long text sequence for improving malware prediction. The following common malware types and definitions are investigated.

- **Miner**: takes over a computer's resources to mine digital currency without authorization [18].
- **Ransomware**: prevents users from accessing their files and demands a ransom to decrypt them. The trend of this type of attack increases to threaten data owners to steal (exfiltrate) sensitive documents if the ransom is unpaid [19].
- **Rootkit**: enables unauthorized system access to control a computer remotely and remains hidden without notice [20].
- **Trojan**: can disguise as legitimate applications to gain access and take control of the system (e.g. delivered as a Microsoft Word file with embedded malicious VBA macros) or to use existing communication channels to transfer files which is undetectable by security mechanisms [21].
- **Virus**: uses malicious code to infect existing system/program files and requires user interaction to spread on an infected system [22].
- **Backdoor**: can be a deliberately hidden vulnerability in the software code, allowing privileged users to bypass standard security mechanisms [23].

A review of research [3, 4, 24–27] performed before the release of Windows 10 showed that mostly only static or a partial dynamic analysis, and a few integrations of both datasets, were utilized for malware detection. When the new version of Windows 10 came out, more sophisticated security measures changed how digital artifacts are used for analysis. For instance, the introduction of Windows Real-Time Protection (RTP) and Ransomware Protection provided some features to defend against viruses and other threats. From Windows 10, the following few types of application files are analyzed: MSI files (an installer package file format), EXE files (an executable file format), AppxBundle files (a Windows 8.1 App Bundle Package format), and DOC files (focusing only on the behavior-based analysis). Due to the archive file formats of MSI and AppxBundle, embedded EXE files inside the package are extracted with "msiexec" and "Add-AppxPackage" and executed via an automated PowerShell command. There are options for automatic silent installation of MSI and AppxBundle files; however, some applications required user interaction to launch the service after installation. To prevent damage from unknown files, static analysis came as the first choice when inspecting the characteristics of binaries.

However, we encounter challenges in classification. For example, advanced encryption and obfuscation techniques are more developed, to hinder static inspection. Although various security solutions are available such as antivirus, SSL certificate encryption, and firewall protection, these countermeasures could be bypassed. Due to their defensive mode, they required the system to update signatures consistently to protect the information against malicious activities or software. In addition, unprecedented malware applications are increasing, with anti-mechanisms (e.g., anti-debugging, anti-virtualization) obstructing analysts from observing and detecting malware during runtime. Intrusion detection techniques based on virtual machine introspection (VMI) for externally monitoring the runtime state of a system-level virtual machine (VM) could have high temper resistance compared to the traditional host-based antivirus. Nevertheless, we do not place this monitoring on our experiment to inspect and analyze the code running. Instead, we enable a process monitor (Procmon) to capture the application operations. According to a study of malware detection techniques [2], an automated approach by considering feature selection for data extraction to improve accuracy is needed. Unfortunately, some approaches, like using the anomaly-based approach, could not extract the best features to feed into a classifier and a certain amount of malware is a limitation for evaluations. With inadequate computing and storage resources, massive malware samples could cause a scalability issue, resulting in many false positives and negatives.

1.2 Goals

This thesis sets out to improve binary classification performance using static and dynamic analysis data. In addition, research focuses on improving the accuracy of predictions of multi-class malware classification from consecutive Noriben and opcode sequences. In order to achieve these goals, the following three research directors are formulated. The main research directions are:

- Extensively extract static and dynamic analysis data from the PE header, sandbox reports, and memory analysis using Python modules. Samples are downloaded from trusted public sources, and the raw data is stored as a CSV output file.

- Explore preprocessing data to handle long sequences of both opcode and Noriben.

- Use a concatenation of models to improve malware classification

- Use hyperparameter tuning for learning approaches to discover the best classifier for each type of feature set.

Using different feature sets to classify malware categories based on machine learning models helps improve accuracy. However, the limitation of previous research is an integrated set of data from various sources by automation and class imbalance. In this thesis, we use an automated custom Python script to collect and output data as a CSV file for data preprocessing before feeding it into learning approaches. Nevertheless, an imbalanced dataset in the number of instances per class is used in our experiments.

1.3 Contributions and Thesis Organization

The main contribution of this thesis is malware classification on Windows Portable Executable files by applying machine learning approaches and different feature extraction techniques. The hypothesis is that combining static and dynamic features extracted from memory analysis, sandbox reports, the sequences of dynamic Noriben operations, and the static opcode sequences could improve classification performance. The two ways of malware investigation are as follows:

- Binary classification utilizing data from the dynamic features (Falcon sandbox reports and memory artifacts) and complement static features.
- Multiclass classification utilizing Noriben time-series data and the static opcode sequences compared to the public dynamic API calls dataset.

The structure of the remainder of this thesis is as follows:

Chapter 2 describes the source of data collection and the significant features used to feed into the algorithms.

Chapter 3 provides background information about static and dynamic analysis on binaries and setting up the environment to acquire the raw data, and also reviews the representation learning literature related to utilizing machine learning approaches on malware classification.

Chapter 4 presents the improvement of classification by implementing SHAP feature selection on extracted features. The method is evaluated empirically using a combination of static and dynamic features.

Chapter 5 shows how the Noriben time-series data captured during the runtime helped to predict malware categories by applying state-of-the-art sequence models and a hybrid approach to extracted sequences.

Chapter 6 introduces a comparison of static opcodes and dynamic API calls for multiclass malware classification using traditional sequence models.

Chapter 7 provides a summary of the contributions in this thesis and speculated about future research directions resulting from the undertaken work.

The content of Chapter 4 is based on an article presented at the 8th IEEE International Conference on Data Science and Advanced Analytics in research and application tracks, Porto, Portugal. It collected raw data from static and dynamic, including Noriben operations that are put into a list with the first occurrence of each operation. The text sequence method described in Chapter 5 is presented at the 31st International Conference on Artificial Neural Networks, Bristol (UK). These consecutive Noriben sequences are used in a separate dataset to observe different predictive results. Finally, chapter 6 compares static opcodes and dynamic API calls presented at the 35th Australasian Joint Conference on Artificial Intelligence, Perth, Western Australia. This extended the work using the same learning approaches as in Chapter 5 and evaluates the classification performance.

1.4 List of Publications

During this research, the following papers are published in peer-reviewed conference proceedings:

- R. Chanajitt, B. Pfahringer and H. M. Gomes, "Combining Static and Dynamic Analysis to Improve Machine Learning-based Malware Classification," 2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA), 2021, pp. 1-10, doi: 10.1109/DSAA53316.2021.9564144.

- R. Chanajitt, B. Pfahringer and H. M. Gomes, "A comparison of Neural

Network Architectures for Malware Classification based on Noriben Operation Sequences," Artificial Neural Networks and Machine Learning-ICANN 2022, Springer, eBook ISBN 978-3-031-15919-0.

- Chanajitt, R., Pfahringer, B., Gomes, H.M., Yogarajan, V. (2022). Multiclass Malware Classification Using Either Static Opcodes or Dynamic API Calls. In: Aziz, H., Corrêa, D., French, T. (eds) AI 2022: Advances in Artificial Intelligence. AI 2022. Lecture Notes in Computer Science(), vol 13728. Springer, Cham. https://doi.org/10.1007/978-3-031-22695-3_30

- Data Extraction Code Repository. `https://github.com/rajchadach/malware`.

Chapter 2

Background and Related Work

This chapter provides information about static and dynamic file analysis applied to benign and malicious files, accompanied by tools used in the research. The analysis does not include the payload, including malicious code or content delivered after a system is successfully compromised. The classification algorithms utilized for training our datasets are also detailed.

2.1 Static File Analysis

In general, Windows Portable Executable (PE) Format consists of executable (.exe), driver (.sys), and library (.dll) files. Static malware detection does not require executing the malware and uses the structural information as file format [28, 29] available in applications. Windows binary consists of PE header [28], code, data, and resource part. Figure 2.1 shows the file format layout. The PE header has a COFF Header, OPTIONAL Header, and section tables. We can identify the most relevant information in OPTIONAL_HEADER data directory entries by using the "`pefile`" [30] Python module to explore the actual file contents, which might contain executable code, data, imports, exports, and resources (e.g., icons). There might be more than pre-defined section names in Windows: .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata and .debug [31].

Additionally, we could locate if there was a Program Database (PDB) associated with an executable when it was built. The PDB file contains information about the executables' creation and the symbols in the latest CodeView format. A path and filename for the PDB file in the DEBUG directory could reveal the linkers/compilers and how many times an executable and the corresponding PDB file are remade by the linker. The DEBUG directory in the executable is generally in the "rdata" section. The data appended to the PE file and not mapped into memory is called an overlay, usually at the end of the file. Similarly, the discovery of the minimum security requirement requested by the application could be seen in the application manifest "`requestedExecutionLevel`" tag:

- **asInvoker**: The application run with the same permission as the process that started it (user's default security settings).
- **highestAvailable**: The application run with the highest permission level.
- **requireAdministrator**: The application run with administrator permission.

After the structure of PE files was thoroughly analyzed, some common features conducted in previous research [33–36] were extracted. In the thesis, we use only static imported DLLs and API calls obtained from memory analysis.

- **Byte code sequence**: it is one of the static analysis techniques that analyzes raw byte contents of binaries. This method is vulnerable to code obfuscation technologies.
- **APIs/System calls**: it is based on N-grams and a bag of words[1] (counted the words independently without considering their sequences inside the documents [37]).

[1]The bag-of-words approach (BOW) is one of the feature extractions for sentences and documents. It only concerns the occurrence of the words within a document regardless of the order or structure of words.

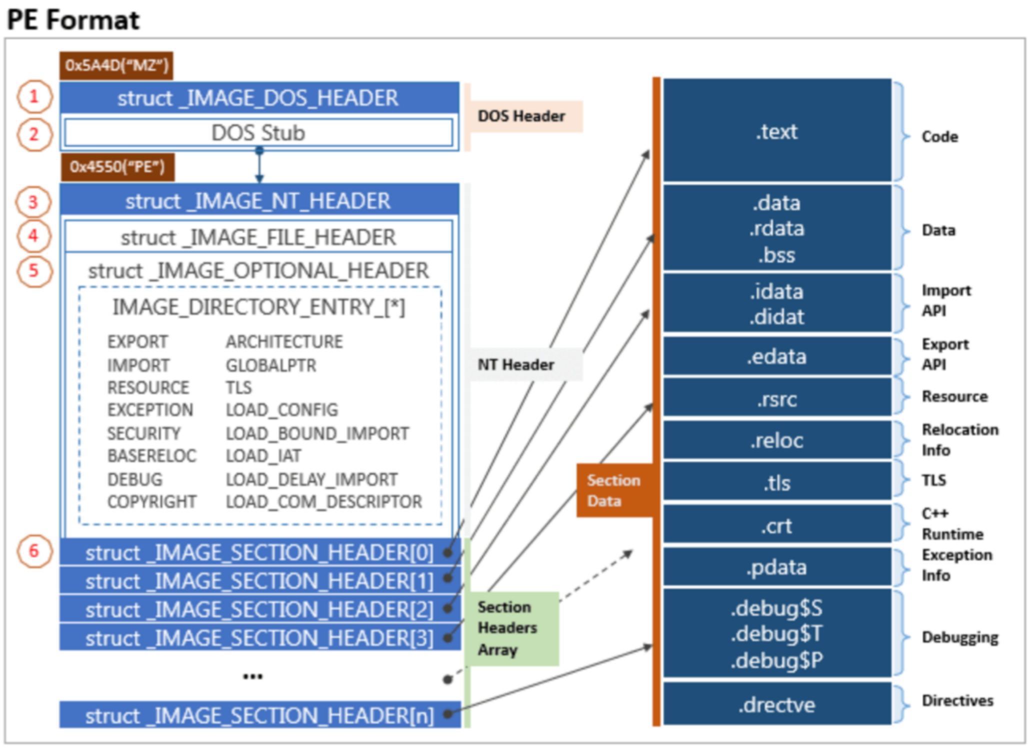
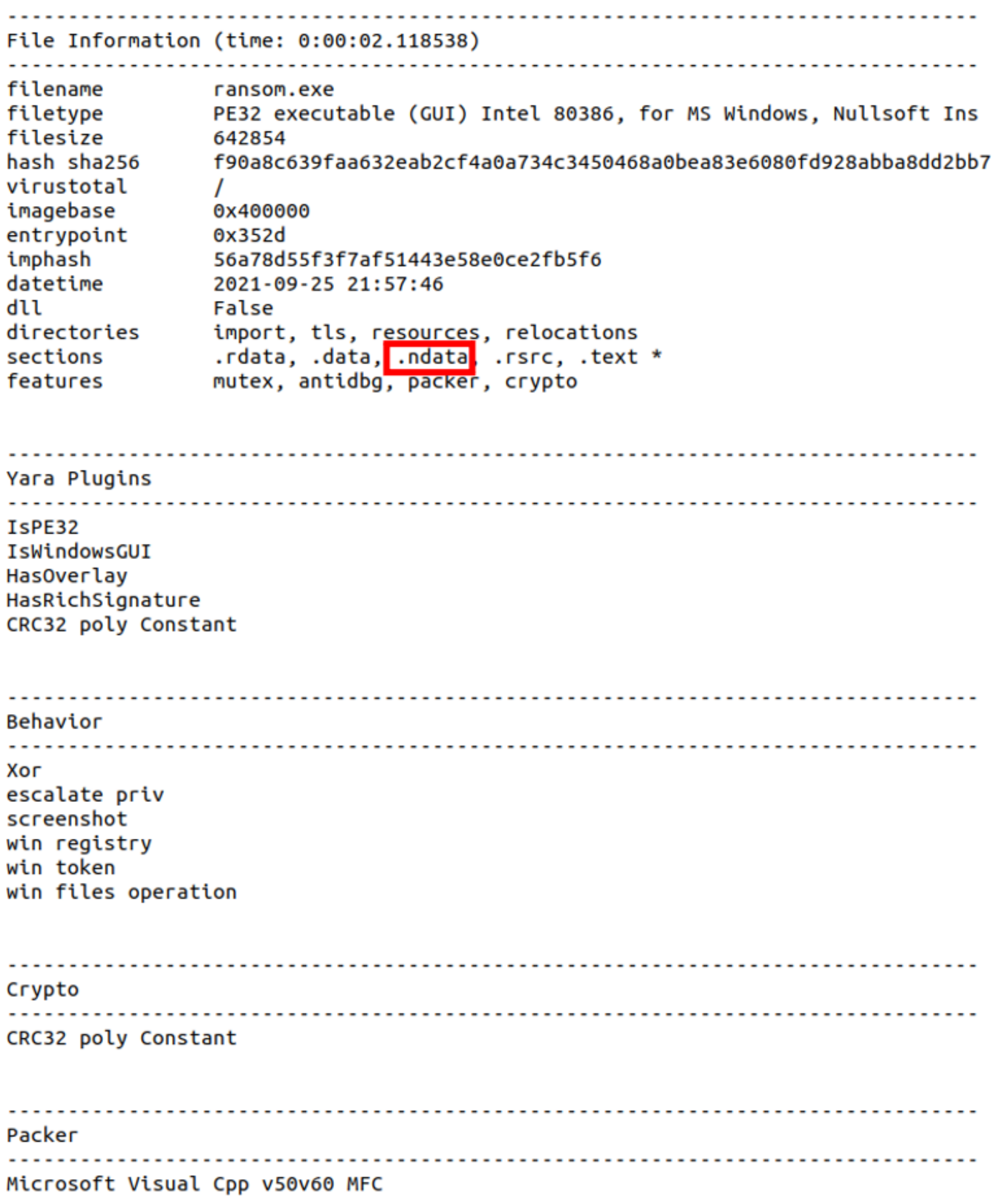


Figure 2.1. Anatomy of PE Files. Source: Kevin's Attic for Security Research [32].

- **Call gram**: it is an extraction of a call graph from a binary program's disassembled instructions and then converts the graph to a call gram.
- **Control flow graph**: a block of assembly instructions represented in the graph.
- **Strings**: a search for characters or words to provide information about the functionality and indicators associated with a PE file [38].
- **Byte/Entropy Histogram**: an entropy is calculated for each PE section independently. It creates a histogram of the binary contents with a maximum of 256 different bytes and approximates the probability of each byte multiplied by its count, adding them all up and dividing by filesize.
- **Imported DLLs**: scans the Import Address Table for functions to be imported from DLLs for .exe files.

A sample result for static analysis on LockBit ransomware is depicted in Figure 2.2, achieved by running the "`peframe`" command-line tool [39]. This sample file analysis presents a suspicious section name (.ndata) which is not in the default sections for standard PE files and have been implemented with packer and encryption.



```
--------------------------------------------------------------------------------
File Information (time: 0:00:02.118538)
--------------------------------------------------------------------------------
filename         ransom.exe
filetype         PE32 executable (GUI) Intel 80386, for MS Windows, Nullsoft Ins
filesize         642854
hash sha256      f90a8c639faa632eab2cf4a0a734c3450468a0bea83e6080fd928abba8dd2bb7
virustotal       /
imagebase        0x400000
entrypoint       0x352d
imphash          56a78d55f3f7af51443e58e0ce2fb5f6
datetime         2021-09-25 21:57:46
dll              False
directories      import, tls, resources, relocations
sections         .rdata, .data, .ndata, .rsrc, .text *
features         mutex, antidbg, packer, crypto


--------------------------------------------------------------------------------
Yara Plugins
--------------------------------------------------------------------------------
IsPE32
IsWindowsGUI
HasOverlay
HasRichSignature
CRC32 poly Constant


--------------------------------------------------------------------------------
Behavior
--------------------------------------------------------------------------------
Xor
escalate priv
screenshot
win registry
win token
win files operation


--------------------------------------------------------------------------------
Crypto
--------------------------------------------------------------------------------
CRC32 poly Constant


--------------------------------------------------------------------------------
Packer
--------------------------------------------------------------------------------
Microsoft Visual Cpp v50v60 MFC
```

Figure 2.2. Sample static analysis with attributes such as XOR, Anti-debugger, Suspicious section data (.ndata), and File metadata.

2.2 Static Analysis Tools

Many tools to examine portable executable files are open-source, accessible publicly, and used by most malware researchers to analyze suspicious files. In this research, a few tools are used as follows:

PEfile: a multi-platform Python module is used to parse Portable Executable (PE) files [28]. Most of the information in the PE file headers is accessible as attributes in the PE instance and all the sections' details and data. "`pefile`" [30] could be installed through pip and requires some fundamental understanding of a PE file format layout.

Disassembling Tools: We use distinct disassemblers consistent with different file extensions. These tools are freeware to extract assembly instructions statically in sequential order. In our research, '`UPX`" [40], '`Zlib`" [41], and '`monodis`" [42] are responsible for disassemble executables. If opcodes could not be extracted via the above-mentioned tools,"`objdump`" [43] is used instead.

VirusTotal: an online scanning tool for suspicious files and websites. It allows a maximum file size upload of 650MB and combines the output of over 70 antivirus scanners and URL/domain blacklisting services. In addition, it offers file submission methods, including the primary public web interface, desktop uploaders, browser extensions, and a programmatic API. For public VTAPI, uploading and scanning files or hash validation and accessing finished scan reports are available with throttling to 4 requests per minute.

2.3 Dynamic analysis

Compared to static analysis, dynamic focuses on malware execution during the runtime. A controlled environment is configured to prevent suspicious applications from escaping outside. For instance, by attaching the debugging to the running application and observing the execution flow, the analyst would notice the system language, available keyboard layouts, and the decryption process. After the malware files have completed their activities, the RAM could be captured. Some common features extracted during malware execution in a virtual environment are:

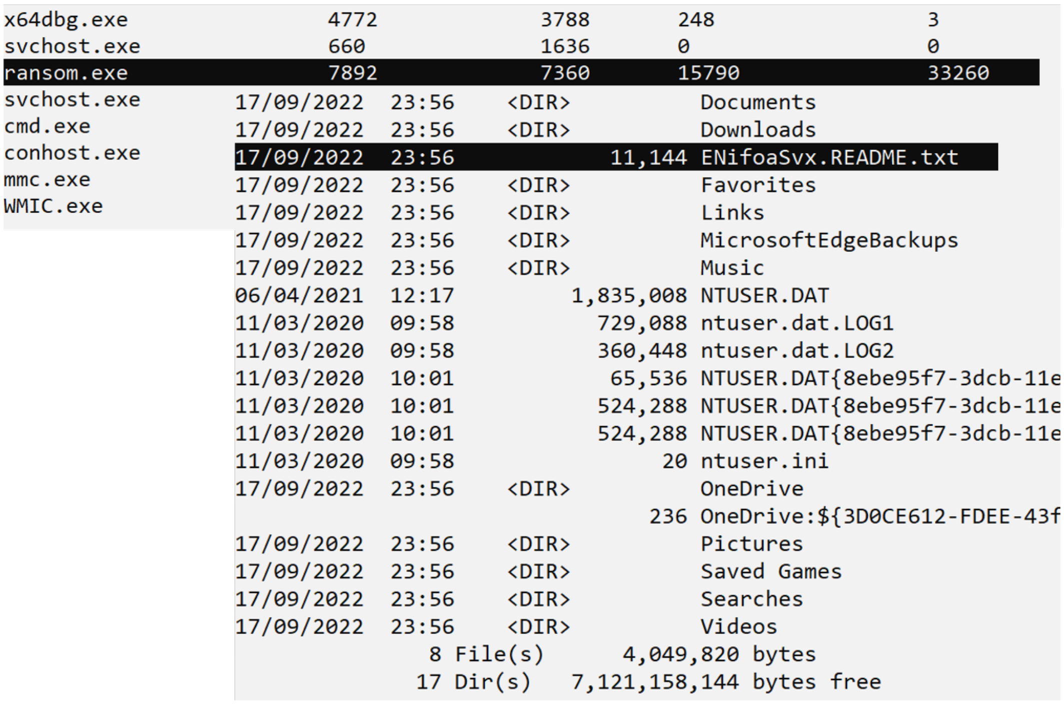
Processes, devices and drivers	Loaded modules
Network socket information	Passwords
Order of execution	Runtime State
Configuration Information	Logged in users
Opened files	Unsaved documents
Live registry/CPU registers	BIOS memory
Native Opcode at runtime (e.g. ja, adc, sub)	Windows' Prefetch files
Event Logs	Encryption/Decryption
Powershell commands	Payload

Table 2.1. Some common dynamic features extracted from previous research [44–46].

We use the same LockBit ransomware analyzed via the static method to execute in the virtualized environment. The analysis results from manual running can be seen in Figure 2.3. We analyze the sample using the Windows Management Instrumentation Command-line tool (wmic.exe) and dir command-line. On the one hand, "`wmic`" lists the process to get ParentProcessId, ProcessId, ReadOperationCount, and WriteOperationCount. On the other hand, "`dir`" command displays all files and subdirectories contained in a specific directory.

2.4 Dynamic Analysis Tools

The tools used in this research to examine dynamic artifacts are: (1)Noriben for activity carving, (2) HybridAnalysis site [47] for retrieving sandbox reports, and (3) Volatility Framework [48] for memory forensics analysis. For volatility,



```
x64dbg.exe          4772        3788    248         3
svchost.exe         660         1636    0           0
ransom.exe          7892        7360    15790       33260
svchost.exe
cmd.exe
conhost.exe
mmc.exe
WMIC.exe
```

```
17/09/2022  23:56    <DIR>          Documents
17/09/2022  23:56    <DIR>          Downloads
17/09/2022  23:56            11,144 ENifoaSvx.README.txt
17/09/2022  23:56    <DIR>          Favorites
17/09/2022  23:56    <DIR>          Links
17/09/2022  23:56    <DIR>          MicrosoftEdgeBackups
17/09/2022  23:56    <DIR>          Music
06/04/2021  12:17         1,835,008 NTUSER.DAT
11/03/2020  09:58           729,088 ntuser.dat.LOG1
11/03/2020  09:58           360,448 ntuser.dat.LOG2
11/03/2020  10:01            65,536 NTUSER.DAT{8ebe95f7-3dcb-11e
11/03/2020  10:01           524,288 NTUSER.DAT{8ebe95f7-3dcb-11e
11/03/2020  10:01           524,288 NTUSER.DAT{8ebe95f7-3dcb-11e
11/03/2020  09:58                20 ntuser.ini
17/09/2022  23:56    <DIR>          OneDrive
                                236 OneDrive:${3D0CE612-FDEE-43f
17/09/2022  23:56    <DIR>          Pictures
17/09/2022  23:56    <DIR>          Saved Games
17/09/2022  23:56    <DIR>          Searches
17/09/2022  23:56    <DIR>          Videos
               8 File(s)      4,049,820 bytes
              17 Dir(s)   7,121,158,144 bytes free
```

Figure 2.3. The ProcessId of ransomware is 7360 with 7892 as ParentProcessId, ReadOperationCount as 15790, and WriteOperationCount as 33260. There is also a text file in the user folder with a suspicious filename (ENifoaSvx.README.txt), which is likely to be a ransomware note.

it could be installed via "`pip`", while Noriben is a Python script that can be placed anywhere on the machine to monitor application activities.

Volatility: an open-source memory forensics framework that can monitor runtime processes and the system's state using the data extracted from volatile memory (RAM). It can perform reconnaissance on process lists, ports, network connections, registry files, DLLs, crash dumps, and cached sectors [49]. We use VMWare Workstation Player [50], a virtualization tool, to create memory dump files by suspending the virtual machine and taking a snapshot. Volatile data might be found in files with extensions .vmem, .vmss, or .vmsn. A doubly-linked list tracks the currently active processes for most standard Windows calls. However, some malware attempts to hide by de-linking its process from this list. By searching through all the memory in a RAM dump for the known structure of a process object's header and other attributes, Volatility could detect processes unlinked in the standard doubly-linked process list. In addition, some plugins can look for suspicious memory areas within a process and display them along with their associated assembly code so that an analyst can determine whether it is suspicious. To ensure which kernel had been loaded, walking through this list of loaded kernel drivers in the LDR_DATA_TABLE_ENTRY could provide the name and path of DLLs loaded by the kernel and list registry hives, including their path on disk and also recently used programs.

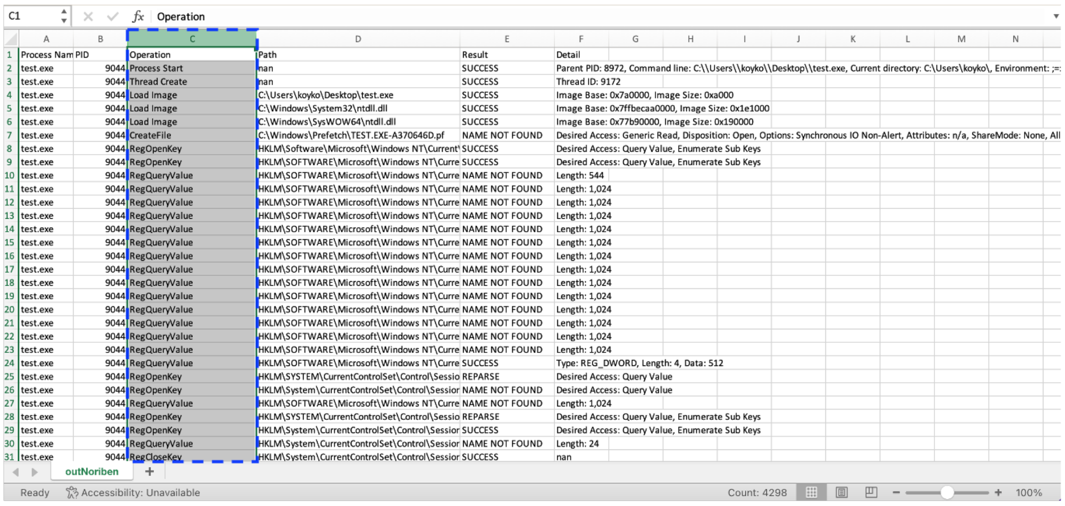
HybridAnalysis: this is a website that provides free malware analysis service with the Falcon Sandbox. It allows users to submit URLs/files for scanning and searching reports via IP, domain, and hash verification. It also matches strings, hex patterns, and YARA rules[2] at the byte level. However, there is 100 MB limit for the maximum upload size.

Noriben: a Python-based script [51] that allows automation of the Noriben execution within a guest VM and retrieves the reports by recording the output into a separate folder. Noriben is a malware analysis sandbox and only required Sysinternals procmon.exe (or procmon64.exe) to operate. Due to a whitelist for unwanted noise reduction from system activity, no pre-filtering is required. Likewise, it could accept varying command line options or user interactions, such as defining the number of seconds for collecting activity. The script logs all running processes in CSV format ordered by time, and we only focus on the tested application in the experiment as shown in Figure 2.4. In the thesis, we focus on the third column, "`Operation`", preprocessed as learning model inputs. One input is the first occurrence for each operation stored in a list used as features in Chapter 4. Another is a consecutive 10000 sequence used as features in Chapter 5.

2.5 Section Manipulation for Evasion

To obstruct pattern identification by malware analysts, authors could deliberately develop malicious applications by modifying section headers to evade

[2]YARA rule was one of the approaches to identifying malware by creating rules based on textual or binary patterns, consisting of two parts: strings definition and condition.



	A	B	C	D	E	F
1	Process Nam	PID	Operation	Path	Result	Detail
2	test.exe	9044	Process Start	nan	SUCCESS	Parent PID: 8972, Command line: C:\\Users\\koyko\\Desktop\\test.exe, Current directory: C:\Users\koyko\, Environment: ;=:
3	test.exe	9044	Thread Create	nan	SUCCESS	Thread ID: 9172
4	test.exe	9044	Load Image	C:\Users\koyko\Desktop\test.exe	SUCCESS	Image Base: 0x7a0000, Image Size: 0xa000
5	test.exe	9044	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x7ffbecaa0000, Image Size: 0x1e1000
6	test.exe	9044	Load Image	C:\Windows\SysWOW64\ntdll.dll	SUCCESS	Image Base: 0x77b90000, Image Size: 0x190000
7	test.exe	9044	CreateFile	C:\Windows\Prefetch\TEST.EXE-A370646D.pf	NAME NOT FOUND	Desired Access: Generic Read, Disposition: Open, Options: Synchronous IO Non-Alert, Attributes: n/a, ShareMode: None, All
8	test.exe	9044	RegOpenKey	HKLM\Software\Microsoft\Windows NT\Current\	SUCCESS	Desired Access: Query Value, Enumerate Sub Keys
9	test.exe	9044	RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	SUCCESS	Desired Access: Query Value, Enumerate Sub Keys
10	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 544
11	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
12	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
13	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
14	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
15	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
16	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
17	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
18	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
19	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
20	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
21	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
22	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
23	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
24	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	SUCCESS	Type: REG_DWORD, Length: 4, Data: 512
25	test.exe	9044	RegOpenKey	HKLM\SYSTEM\CurrentControlSet\Control\Sessio	REPARSE	Desired Access: Query Value
26	test.exe	9044	RegOpenKey	HKLM\System\CurrentControlSet\Control\Sessior	NAME NOT FOUND	Desired Access: Query Value
27	test.exe	9044	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\Curre	NAME NOT FOUND	Length: 1,024
28	test.exe	9044	RegOpenKey	HKLM\SYSTEM\CurrentControlSet\Control\Sessio	REPARSE	Desired Access: Query Value, Enumerate Sub Keys
29	test.exe	9044	RegOpenKey	HKLM\System\CurrentControlSet\Control\Sessior	SUCCESS	Desired Access: Query Value, Enumerate Sub Keys
30	test.exe	9044	RegQueryValue	HKLM\System\CurrentControlSet\Control\Sessior	NAME NOT FOUND	Length: 24
31	test.exe	9044	RegCloseKey	HKLM\System\CurrentControlSet\Control\Sessior	SUCCESS	nan

Figure 2.4. A sample of raw Noriben CSV output file carving for an application process by only considering the Operation column.

security countermeasures or machine learning classifiers, while maintaining their original functionality and behavior with the following methods.

- Appending random bytes (ascii/zero) to the end of a section or the file
- Modifying the optional header checksum, export, remove the signature or debug info
- Inserting/re-ordering code
- Implementing packers such as UPX (Ultimate Packer for eXececutables) [40]
- Encoding and splitting obfuscation
- Renaming/Creating new sections
- Randomizing section names
- Adding an unused function to the import table
- Creating a new entry point (which jumps to the old entry)

These techniques caused trouble for our static analysis and could provide false positives for some feature extraction. To obtain more accurate data, we need information from dynamic analysis and augmented them, using extracted features from static and dynamic results, to create a feature set for a classification task.

2.6 Environmental Setup

To perform dynamic analysis, two guest virtual machines are set to be victim and server for the application running in a secure environment. We also take snapshots while they are in a good state. For bi-directional file uploading and downloading, we use a Ubuntu machine to connect to the guest via the "`vmrun`" command–the test environment is shown in Figure 2.5. Its key parts are as follows:

- **Windows10 64-bit guest** is set to be the victim. The networking interface is set to host-only mode with a static IP address to restrict communication with Remnux Ubuntu (server). In addition, an administrator account is enabled to run commands and access the system folder to prevent permission from being denied. Moreover, the Noriben Python script is placed on the desktop to collect, analyze, and generate a report during runtime with Sysinternals Procmon put into the python folder and a ZIP program in the download folder for compression files. Furthermore, we fully disable the User Access Control (UAC) in the virtual Windows 10 guest machine group policy.

- **Remnux ubuntu server** is a simulated inetsim server to provide simulation for internet connection by configuring an IPtable to redirect internet traffic (port 80 and 443) to a pre-configured port 8443 on the server. We also modify the DNS configuration and resolve it to the pre-configured domain name.

- **Ubuntu host** is an analysis machine to run static and dynamic python scripts for collecting raw data with output in CSV format to be preprocessed and fed into learning models. Additionally, this machine analyzes other artifacts such as prefetch files, Windows event logs, temporary folders, and performs a memory dump analysis.

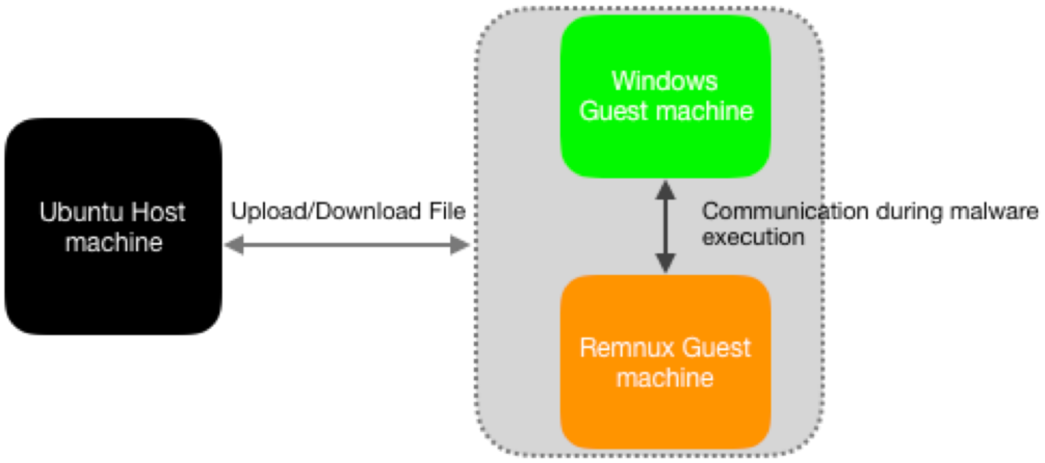


Figure 2.5. Architecture of the dynamic analysis environment.

2.7 SHAP Feature Selection

SHAP (SHapley Additive exPlanations) [52] values represent each feature's average impact on each sample prediction, based on all possible features. A (samples, features) matrix lists the contribution of each feature to each sample as a prediction. Positive SHAP values indicate positive impacts on a prediction, while negative SHAP values present an opposite direction. The features are ordered by the influence on the model's prediction. The x-axis represents the average of the absolute SHAP value of each feature, and the y-axis has all the features. Each point on the chart is one SHAP value for a prediction and feature. The red color means a higher feature value, while blue means a lower value. Based on the distribution of the red and blue dots, we could get the global direction of each feature on the final prediction. We could use the summary plot to provide a rich overview of how each feature impacts the model for each class. Such feature selection is applied to a combination of static and dynamic analysis for binary classification and static opcode sequences for multiclass classification. Applying this feature selection method to our datasets in the thesis provides high classification performance. However, other options for feature selection should have been considered.

2.8 Experimental Methodology

This thesis makes use of several open-source frameworks. The neural network and text sequence models presented are mostly implemented using Python with Tensorflow [53]. All evaluations are done using sklearn metrics [54]. We use the original open-source transformer [55] for transformer implementations.

All models are fine-tuned by using Keras-tuner [56] on all layers without freezing. TextVectorization from TensorFlow [53] is used for preprocessing text sequences and setting the sequence length for the output. RMSprop [57], an adaptive learning rate optimization algorithm, is used as the optimizer for default learning approaches presented in this thesis. This optimizer is an extension of the gradient descent algorithm with momentum, and the foundation of the Adam algorithm [58]. A non-linear sigmoid function is used as the activation function with a binary-cross-entropy loss for binary classification. Conversely, a softmax activation function with categorical-cross-entropy loss is utilized for multi-class problems.

All experiments are validated through stratified 10-fold cross-validation (CV). Due to resource restrictions, all neural network results are used to train test hold-out set validation where the results are averaged over thirty run times. 10-fold CV reduces bias when conducting hyperparameter optimization and obtaining the best model, making it more reliable. However, the variations of these thirty independent runs are measured by the standard deviation within a range of ± 0.04.

2.9 Classifiers

This section presents details of the classifiers used in this research. We can divide the algorithms into three groups: (1) traditional classifiers, including random forest, XGBoost, and neural network (2) word embeddings-based neural networks, and (3) transformers. Only text sequence models are applied to text-based data.

2.9.1 Traditional Classifiers

We use supervised machine learning algorithms to build a model to find patterns in a dataset with labels and features. Then, the trained model is employed to predict a category of a new dataset's class label. Scikit-learn [54] is utilized to implement classification models. Most importantly, the model parameters could be fine-tuned for optimization. In this research, three learning approaches are applied.

Random forest [59] builds an ensemble of multiple random decision trees in parallel from random bootstrap samples of the dataset. It benefits from less vulnerability to over-fitting than decision trees as the number of trees increases. The ensemble produces the final prediction by an average of all decision tree predictions.

XGBoost [60], which stands for Extreme Gradient Boosting, is another decision-tree-based ensemble algorithm that uses a gradient boosting framework. The trees are built in parallel, and a sum of gradient values is used to evaluate the quality of splits at every possible split in the training set.

The neural network [61] is an algorithm that reflects the behavior of the human brain to recognize data patterns. These so-called Artificial Neural Networks (ANNs) comprises an input layer, one or more hidden layers, and an output layer. Each interconnected node has an associated weight and threshold. Once an input layer is determined, weights are assigned and passed through an activation function to generate the output. If that output exceeds a given threshold, the node is activated, passing data to the next layer in the network. The above node's output becomes the input of the subsequent node.

2.9.2 Word Embeddings-based Neural Networks

This research uses sequential models popular for processing text sequences, including LSTM and GRU.

Long Short Term Memory (LSTM) [62] is an advanced Recurrent Neural Network (RNN). However, RNN could not remember long-term dependencies, due to the vanishing gradient problem. Thus, LSTM is designed to handle long-term dependency problems by introducing memory units called cell states, which maintain their state over time. The LSTM network consists of three parts: an input gate, a forget gate, and an output gate. Information could be removed or added from memory and controlled by gates. Over time, the memory cells learn the essential information based on the weights.

Gated Recurrent Unit (GRU) [62] is a type of Recurrent Neural Network (RNN) that does not include separate memory cells, which makes it faster than LSTM. GRUs addresses the vanishing gradient problem by using update and reset gates. These gates decide what information is allowed through to the output and could be trained to preserve information from farther back. In addition, GRU features an additive component, allowing each unit to remember a feature in the input for a more extended series of steps.

2.9.3 Transformers

Transformer [55] is a neural network that adopts the self-attention mechanism with no recurrence and convolutions, distinctively weighting the significance of each part of the input data. It is one of the recent models and is used primarily in natural language processing (NLP). In addition, it can achieve state-of-the-art (SOTA) results in many language tasks [63–65]. Self-attention is a method for the computation of the representation of a sequence, by capturing the relationships between the different elements of the same sequence. Furthermore, the entire sequence is trained simultaneously in Transformer networks, enabling the capture of long-range dependencies.

$$Attention(Q, K, V) = softmax(\frac{QK^{\mathrm{T}}}{\sqrt{d_{\mathrm{k}}}})V \tag{2.1}$$

The equation 2.1 presents the definition of self-attention, which is a mapping between a query and a set of key-value pairs to an output. The transformer implements scaled dot-product attention. First, a dot product for each query (q), with all the keys (k), is computed and packed together as Q, K, and V matrices. Subsequently, it is divided by the square root of the dimensionality of each key, d, and proceeded to apply a softmax function and obtained the weighted sum of the values. In addition, multi-head attention [66] is used. Multiple weight matrices produce different query, key, and value vectors for the same word, allowing the model to have multiple representations.

$$multihead(Q, K, V) = concat(head_1, ..., head_{\mathrm{n}})W^{\mathrm{O}} \tag{2.2}$$

The multi-head attention function concatenates the outputs of the number of linear layers (heads) obtained from each attention function and then multiplies with the weight matrix to generate the final result, as shown in equation 2.2. An example of three linear layers giving three attention outputs concatenated together is depicted in Figure 2.6.

2.10 Evaluation Measures

Classification is about predicting the class labels given input data. For example, in binary classification, there are only two possible output classes; in multiclass classification, more than two possible classes could be present. For performance evaluation, the evaluation metrics play a vital role. To improve the model's overall predictive result, we use different metrics, such as F1-score, AUC, and confusion matrix, to measure the performance of the classification task before we put it in place for production on unseen data. However, it could result in poor predictions when the respective model is deployed on unseen data without proper evaluation of the model and only depending on

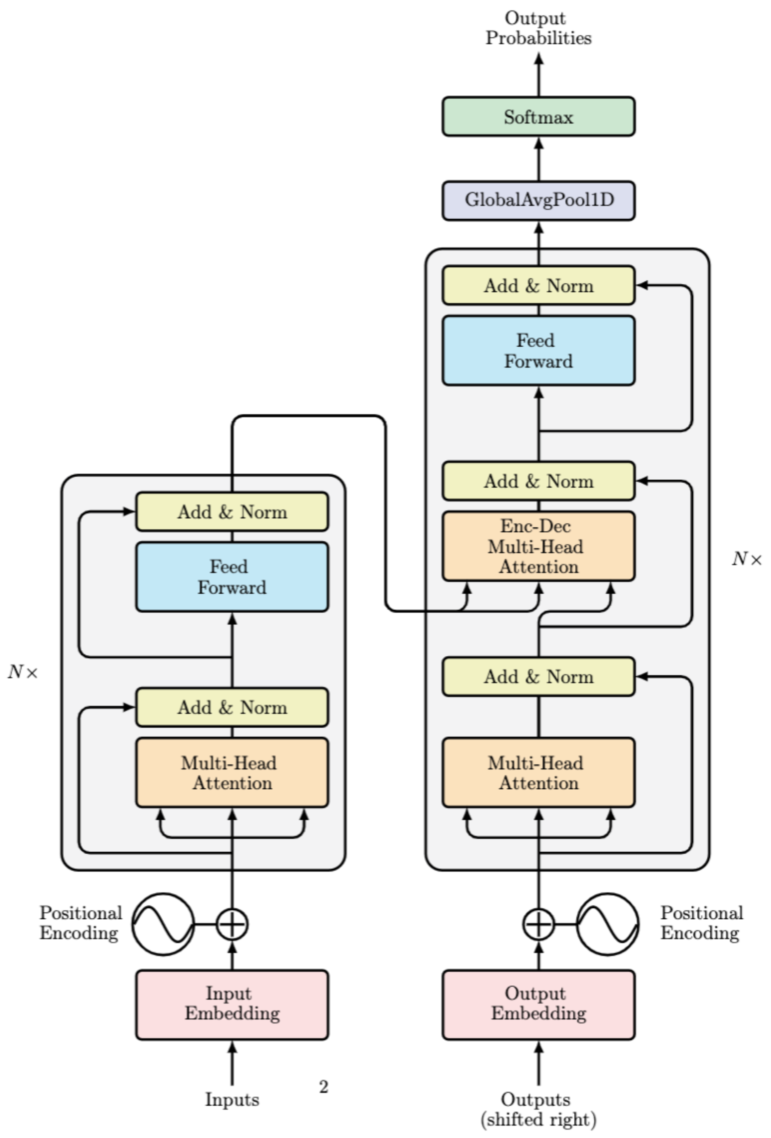


Figure 2.6. Multi-head attention algorithm with GlobalAvgPool1D layer before the softmax output layer.

accuracy, especially for imbalanced data. We also use stratified 10-fold Cross-Validation to maintain the same class ratio throughout the K folds as the ratio in the original dataset. This section presents an overview of such evaluation measures used in this thesis that could help generalize the ML classification model.

2.10.1 Accuracy

Generally, this metric is a ratio of correctly predicted observations to the total observations. For binary classification, accuracy could be calculated in terms of positives and negatives as follows:

$$Accuracy = \frac{TruePositive + TrueNegative}{TruePositive + TrueNegative + FalsePositive + FalseNegative} \tag{2.3}$$

Similarly, accuracy for multi-class classification is directly computed from the confusion matrix. The formula considers the sum of True Positive and True Negative elements at the numerator and the sum of all the confusion matrix entries at the denominator.

2.10.2 F-measure

F-measure or F1-score is used as the primary evaluation measure for binary or multi-class classification. F1-score is calculated by,

$$\begin{gathered}
TPR = \frac{TruePositive}{TruePositive + FalseNegative} \\
FPR = \frac{FalsePositive}{FalsePositive + TrueNegative} \\
Precision = \frac{TruePositive}{TruePositive + FalsePositive} \\
Recall = \frac{TruePositive}{TruePositive + FalseNegative} \\
F1\ Score = 2 * \frac{(Precision * Recall)}{(Precision + Recall)} \\
Micro\ F1\ Score = \frac{sum(F1\ Score)}{TruePositive + \frac{1}{2} * (FalsePositive + FalseNegative)} \\
Macro\ F1\ Score = \frac{sum(F1\ Score)}{number\ of\ classes}
\end{gathered} \tag{2.4}$$

For multi-classification problems, label-wise macro and micro F1 scores are used. Macro-F1 is the mean of the label-wise F1 score. Micro-F1 is the harmonic mean between the micro-recall and micro-precision.

2.10.3 MAE

The Mean Absolute Error(MAE) is the average magnitude of the errors in a set of predictions without considering their direction. It is the average of all absolute differences between the measured and actual values where all individual differences have equal weight to measure how confident a prediction is. The formula for the MAE is:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|x_{\mathrm{i}} - x| \tag{2.5}$$

2.10.4 mlogloss

The 'mlogloss' (multiclass categorical cross-entropy) [67] is a probability-based metric for scoring multiclass problems. It is a default metric set by eval_metric parameter in XGBoost's configuration. This parameter is configured with num_class and multi:softprob objective.

2.10.5 AUC-ROC

AUC stands for "`Area under the ROC Curve`" and ranges from 0 to 1. A ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold values. The higher the AUC, the better the model's performance is. When AUC equals one, the classifier could correctly differentiate all the Positive and the Negative classes. If, however, the AUC has been 0, then the classifier would predict in the opposite direction. Any class can be positive or negative. It depends on which class is selected as positive and the rest is negative. We use roc_auc_score with setting "multi_class" to "ovr" to compute the AUC of each class against the rest.

2.10.6 Confusion Matrix

A confusion matrix is a performance measurement where the output could be two or more classes. It is a table that summarizes the number of correct and incorrect predictions made by a classifier with count values broken down by each class. It overcomes the limitation of using classification accuracy alone by providing information about the number of misclassified instances.

2.11 Related Works

Malware detection using machine learning techniques was first addressed by Schultz [68]. They used static analysis techniques on features, including extracted strings, imported functions, and contiguous sequences of n bytes (N-grams). This section presents a literature review related to the research presented in this thesis. These research efforts are divided into the types of classification: (i) Binary Classification and (ii) Multi-class classification. A summary of all relevant works can be seen in Table 2.2. There are no public model parameter settings, so replicating these models and applying them to our datasets would be difficult. However, we utilize other datasets by using our model configurations, they do not work so well. On top of that, we could not find any detailed results compared to existing works, even if they are using different datasets.

2.11.1 Binary Classification

2.11.1.1 XGBoost

Kumar and Geetha [69] proposed an XGboost GBDT using limited computing resources and used the EMBER dataset [70], which was a balanced dataset for malware and benign applications. As completed by XGBoost, feature importance could be used to reduce the number of features from 2351 to 276. The authors compared the performance of nine algorithms: Gaussian NB, KNN, linear support vector classification (SVC), Decision Tree (DT), AdaBoost, Random Forest (RF), extra trees, gradient boosting (GBDT), and XGBoost. In their experiments, XGBoost outperformed the other algorithms and became the final classifier. With 276 selected features and hyperparameter tuning, the model achieved an accuracy of 98.5% and an AUC value of 0.9989.

XGBoost and Extremely Randomized Trees algorithms introduced by Jakub Palsa et al. [34] were applied to two datasets obtained using static and dynamic analysis. The two classifiers were implemented for each analysis and then used a voting mechanism for the final result. Four algorithms were compared with the best classification result obtained from XGBoost, which achieved 91.9% detection accuracy on the static analysis and 96.48% on the dynamic analysis dataset.

2.11.1.2 Deep learning

In addition to decision tree-based methods, deep learning has been brought to malware classification, such as the Static Malware-as-Image Network Analysis (STAMINA) approach [71] proposed by Intel Labs and the Microsoft Threat Intelligence Team. 197,604 benign and 584,606 malicious samples were converted to grey-scale images and split into three parts, with a ratio of $60 : 20 : 20$. The method underwent four stages: image conversion, transfer learning, evaluation, and interpretation. First, binaries were converted into a one-dimensional pixel stream and reshaped into two dimensions. Next, they used pixel file size to determine the image width, and calculated the height by dividing the number of pixels by the width. Finally, before applying transfer learning, images were resized to $224 * 224$ or $299 * 299$. For the STAMINA model, two approaches were implemented based on image segmentation. However, the authors ran into several issues, including benign files being much more extensive than malign ones and their conversion software from binary to images needing to cope better with massive binaries. Nonetheless, they reported good performance, achieving an accuracy rate of 95.97% overall and a false positive rate of 4.49%.

Moreover, Neurlux [44], a deep learning-based combination of convolution neural networks, BiLSTM, and attention without feature engineering, was presented for a document classification task. 27,540 Windows x86 executables from VendorDataset and 42,000 binaries from EmberDataset were used as the dataset. First, the JSON-formatted dynamic analysis reports retrieved from VendorSandbox and CuckooSandbox were restructured from documents to sequences of words by removing special characters (e.g., the brackets). Then, the document was tokenized to extract words, converting the top 10,000 most common words into numerical sequences. Lastly, they were formatted using one-hot encoding. Five features extracted from the reports were analyzed: API Sequence Calls, Mutexes, File System Changes, Registry Changes, and Loaded DLLs. The output was a score between 0 and 1 for classification decisions. Regarding the results, the most attention belonged to file operations and API calls, and file operations were the best-performing individual feature. Deep learning methods without feature engineering could identify malware from dynamic analysis reports as effectively as methods with feature engineering, with high accuracy at 96.8%.

The work of [72] tried to discover explicit sub-sequences, or episodes, by investigating N-grams of the API call sequences. High accuracies could be achieved by using some specialized online algorithms. However, they might need to be more accurate due to the highly imbalanced dataset, with more than 90% of the samples detected as malware.

Joshua Saxe et al. [35] proposed the eXpose deep neural network to work on short character strings, such as malicious URLs, file paths, and registry keys. The work was evaluated against an N-gram model and manual feature extraction on three problems. The neural network consisted of three components: (1) character-level embeddings to embed the alphabet of printable English- language characters into a multi-dimensional feature space, (2) feature detection to detect local sequence patterns and aggregates into a fixed-length feature vector, and (3) a classifier using a dense neural network. These components were optimized using stochastic gradient descent and evaluated against two baseline approaches on three different problems. First, identifying malicious URLs directly from the URL string by downloading 19067879 unique URLs, randomly sampled over a roughly two-month period, from VirusTotal.

Second, 5590614 unique file paths and 1661716 registry key paths from the Cuckoo sandbox were analyzed for the second and third problems. Finally, for labeling URL artifacts, a voting approach was used. If five or more antivirus engines assign a "malicious" label to a URL, it was flagged as malicious. Otherwise, it was labeled as benign. Besides the voting mechanism, the behavioral traces were inspected to label file and registry key paths, counting how often each unique file path or registry key was created or written. Apparently, eXpose outperformed the N-gram model and manual feature extraction-based machine learning baselines, yielding a 5%- 10% higher detection rate at a 0.1% false positive rate compared to these baselines.

A sliding local attention mechanism model (SLAM) was proposed in [73]. They used API calls (a public dataset for Alibaba's 3rd Security Algorithm Challenge [74]) and classified them into benign and malicious samples. Most samples comprised the benign program and five malicious types (infected virus, trojan, miner, DDoS Trojan, and extortion virus). Due to an imbalanced dataset, a random sampling method was adopted to obtain 9192 samples. After the APIs' collection in the Cuckoo sandbox, the word2vec [75] embedding approach was utilized to convert an API call sequence into a number sequence. The most commonly used 310 API attributes were selected, and classified into 17 categories, with a maximum sequence length of 2000. The authors could extract two-dimensional input vectors (API_Seq, Category_Seq) and then process them by a standard convolutional backbone network. According to the characteristics of the API execution sequence, several adjacent API calls had a particular local significance. To verify the validity of their feature extraction method, they used the one-dimensional API index sequence with no structural information as a comparison and measured by accuracy rate. On average, nearly three percentage points of the two-dimensional feature extraction method were higher than the one-dimensional feature extraction method. RF (Random Forest), ACLM (Attention_CNN_LSTM model) [76], and TCAM (a two-stream CNN-Attention model) [77] as comparison models were used to measure model effectiveness. The result showed that SLAM could achieve better performance, with a higher accuracy of 97%.

Azadech et al. [36] proposed a static signature-based malware detection method based on N-grams of opcodes with different degrees and file signatures. The data from VXheaven [78] and Windows system files were used (203 as malware and 216 as benign). There were three phases: extracting opcode and binary sequences from benign and malicious files, generating N-grams, and classifying files into benign and malicious groups. Regarding the results, combining 1, 2, and 3 grams represented a feature set with an accuracy of 78%. In conjunction with the proposed Top-K approach, by not examining the similarity between two different families to select the topmost similar k files in an unknown file classification, the highest accuracy was Top-10 on 1, 2, and 3-gram of opcodes, at 86.63%. Likewise, in the combination of opcodes and binary sequences, the K was chosen to be 3, resulting in an accuracy of 86.39%.

Furthermore, Yu Wang et al. [45] introduced binary malware classification based on deep recurrent reinforcement learning (DRL). A sequence of API calls was generated by running the emulation and then choosing one action out of two: the Continue and the Halt after the execution at the best time. The production Microsoft antimalware engine evaluated 75000 files to generate behavioral events by discarding event sequences on files shared between two classes and containing less than 50 events. Their particular version of the antimalware engine mapped multiple low-level API calls into a single high-level event, to deal with polymorphism. The engine recorded 114 event types, including file IO, registry APIs, networking APIs, thread or process creation and control, inter-process communication, timing, and debugging APIs. The model's state contained three parts: the position (i.e., index) of the current event in the file, the current event ID, and the histogram of all the previous events. Two criteria were used to design a reward for each state: (1) a higher reward for a shorter sequence and a smaller reward for a longer sequence. (2) a larger reward to the event prediction closer to the actual file label. From 20 (K[3]) consecutive events on 2000 (N[4]) files, the result showed that the DRL model halted the emulation of an unknown file earlier than the engine's heuristics and improved malware classification at 91.3%. Compared to the GRU-ESN-based baseline file classifier [79], the model's accuracy exhibited a relative increase of 61.5% at a false positive rate of 1%.

To handle adversarial learning-based attacks, Yu Wang proposed a new Actor-Critic (AC) deep reinforcement learning [46] instead of the DQN model [45]. Compared to the DQN approach, the AC model performed better for all K[3] and N[4] values, halting execution of unknown files by up to 2.5% and improving the true positive rate by 6.9%, from 69.5% compared to the DQN model.

2.11.1.3 Clustering

Research reported in [33] proposed a prototype system for malware classification by only focusing on collecting a trace consisting of a sequence of instructions written to memory or registers. The authors also implemented malware clustering and code reuse identification between different malware families. A prototype was developed using PyDBG [80] to collect only a subset of arithmetic and logic instructions, such as add, adc, sub, and xor, without logging standard API calls or the instructions executed inside these APIs. More than 16000 malware samples were run and processed in the prototype system. When the malware process terminated or reached a timeout, a snapshot of the content of the code segments was captured. After that, a transformation was applied to combine a sequence of values into pairs or bigrams for obtaining the final malware traces.

A two-step algorithm [33] that determined the similarity of two malware activities was built, which outputted a similarity measure (S) that ranged from 0 (completely different) to 1 (identical). The first step created a small "identi-

[3]a number of consecutive events

[4]a number of training files

fier" to be compared to others using the Jaccard index for fast initial filtering. Then, a second step was triggered if the Jaccard index of two malware activities was more significant than the empirically established threshold of 0.34. Otherwise, the low similarity value was directly used. The conditional second step was a full-fledged similarity comparison that computed the longest common subsequence of the entire two traces. When the new sample exhibited a trace with more significance than a threshold of 70% similarity, one or more cluster leaders (the selected trace to represent the entire cluster) merged this sample with the existing cluster, which had the highest similarity. Otherwise, the sample (and its trace) was put into a new cluster and became the cluster leader. After a set of initial clusters whose traces shared at least 70% similarity were obtained, a pairwise comparison between all cluster leaders was performed using their diff-based longest common subsequence (LCS) algorithm. Finally, the clusters were merged if their similarity exceeded the same 70% threshold. The approach produced 7,793 clusters compared to three reference clustering sets: static (7,900 clusters), behavioral (3,410 clusters), and AV labeling, generating an average precision value of 0.843 for the first two sets and a value of 0.871 for the last one.

In addition to computing a general similarity or overlap score, identical parts of traces were also identified to detect code reuse between traces. At least four consecutive shared instructions were investigated with regular expressions and matched against the dumped code segment. Based on the overlap between the two traces, only pairs of clusters with a similarity score between 10% and 30% were checked, resulting in 974 clusters being identified and 15 pairs of ten different clusters that share code between them. Specifically, seven blocks of code seem to be reused among samples.

2.11.1.4 Transformers

Miles et al. [5] presented the I-MAD (ST+) network that applied the Galaxy Transformer with Star-Plus Transformer[5] on sequences of assembly instructions for binary malware classification. The Galaxy Transformer consisted of two pre-training tasks, the Satellite-Planet Transformer and Planet-Star Transformer, that were utilized to understand the semantic meaning of the assembly functions–consisting of one opcode and two operands at basic blocks[6]. The IDA Pro [82], a commercial disassembler, disassembled their executables and obtained the assembly functions. The balanced dataset was also used; each consisted of 115,000 to create a block of lengths between 5 to 250 instructions, resulting in 38,427,440 basic blocks. Regarding the results, the classification accuracy with 10-fold cross-validation was 97%. This work differed from ours because we only considered opcodes with no arguments to extract features and used open-source tools only to disassemble opcodes.

2.11.2 Multiclass Classification

2.11.2.1 Random Forest and XGBoost

Canzanese et al. [83] presented self-protecting servers that could quickly detect a new malware variant's execution based on behavioral similarity and use it as inputs to Random Forests. More than 800 malware samples from blacklisted sites were collected using the Dionaea honeypot and a custom harvester. The samples were executed on a custom testbed to mimic the challenges of online malware classification in the real world.

Three different features were collected: (i) performance monitors from resource usage, (ii) the number of calls made to specific kernel functions per second, and (iii) the number of calls made to each unique sequence of two kernel function calls per second. The resource usage included CPU, disk, network,

[5]It is an improved version of the Star Transformer [81]

[6]Basic Block is a sequence of assembly instructions that has no in and out branches and always executes one after the other.

and memory usage statistics, individual processes and threads, VM instances, and application-specific information, such as web server and database server statistics. Meanwhile, kernel function calls were captured using an NT Kernel Tracer custom-built application. After malware execution on the testing platform, 882 distinct malware samples were selected for evaluation. Only four or more different samples with the same classes were considered to evaluate the classifier's performance. The results showed that the correct classification of malware with labels that defined a specific set of behavioral features was more accurate than labels that represented a broad range of malicious behavior. However, additional features and testing on different system configurations were needed.

In [84], Sumaya introduced the CNN-XGBoost model to convert malware binaries to greyscale images and classify them into their families. The Malimg dataset [85] was used as the dataset, containing 9339 malware examples categorized into 25 different classes. Next, the author used CNN as a feature extractor consisting of three layers: a convolution layer, a pooling layer, and a fully-connected layer to extract features from images. However, instead of CNN's most used softmax classifier, the XGBoost layer was added to classify families. As a result, integrating CNNs with EXtreme Gradient Boosting provided outstanding performance for malware classification, achieving 98.7% accuracy.

2.11.2.2 Deep Learning

In [86], Zhihua presented a deep learning method based on greyscale images. The authors proposed a convolutional neural network (CNN) and the bat algorithm [87] to resolve data imbalance between different malware families. The 25 families were collected from the Vision Research Lab [88], generating 9342 greyscale malware images. The binaries were transformed into images and reshaped to a fixed-size square image before identification and classification by CNN. Data augmentation was employed for the small number of raw data samples to increase the number of samples and improve image quality. Four malware detection systems using standard image feature extraction methods were used for comparison, and the proposed approach could achieve 94.5% accuracy.

Kolosnjaji et al. [89] proposed a deep hybrid architecture of convolutional and recurrent neural networks to process API call sequences and classify them into malware families. On the one hand, the convolutional network counted the presence and relation of N-grams in system calls, instead of consideration on the actual position in the input data. On the other hand, the recurrent part utilized the full sequence information. The dataset was acquired from three primary sources: Virus Share [90], Maltrieve [91], and some private collections. Regarding experimental results, a hybrid neural network containing two convolutional layers and one recurrent layer significantly improved HMMs or SVMs for malware classification. They achieved over 90% accuracy, precision, and recall for most malware families.

Besides, a malware classification using sequences of API calls by dynamic execution was proposed in Qian et al. [92]. A set of color mapping rules was used to turn applications into artificial images, and then used a convolutional neural network to classify malware into nine different families. 1000 samples in each family were used as a dataset. The pixels in images represented different subsets of API calls, with the pixels' brightness encoding call frequencies. A standard convolutional neural network then processed the resulting images. As a result, they achieved very high performance on their specific dataset, with an F1-score of 99% and only 0.085% for a false positive rate. Other previous research [93] explored the use of file event streams using LSTM and GRU architectures for malware detection. The study pointed out that an LSTM network trained with temporal max-pooling using a logistic regression classifier (LR-LSTM-MAX) was the best-performing model, achieving a better than 80% TPR at a 2% FPR.

Regarding the static analysis, ransomware families [94] fingerprinted the environment to evade detection by dynamic analysis. They collected 1787 ransomware samples from eight families: cryptolocker, cryptowall, cryrar, locky, petya, reveton, teslacrypt, and wannacry by VirusTotal, and 100 trusted software samples and obtained opcodes by using the IDA Pro disassembler. The opcode sequences were transformed to N-gram sequences and arranged by Term frequency-Inverse document frequency (TF-IDF) [95]. The N-grams in each family by their TF-IDF values were sorted in descending order, selecting each family's top t N-grams (feature N-grams). Then, TF values of the feature N-grams were fed into five machine learning methods: DecisionTree, Random Forest, K-Nearest Neighbor, Naive Bayes, and Gradient Boosting Decision Tree. From each family, N-grams of lengths 2, 3, and 4 were extracted, with different feature dimensions ranging from 29 to 228. Overall, the proposed method achieved high accuracy by 3-grams Random Forest, both multi-classification and binary classification. For multiclass, the best accuracy was 91.43% with a feature dimension of 123. The highest F1 measure of nearly 99% was attained on wannacry. On the other hand, with a feature dimension of 180 on binary classification, the accuracy between ransomware and trusted software was up to 99.3%.

In the work of [96], 2600 samples were run in a virtual machine on Windows XP, and API call sequences were extracted. A whole set of 534 APIs has been hooked and mapped to 26 categories. Every sequence of Windows API calls was mapped to a sequence of A to Z letters. They developed a repository of 2000 fuzzy hash signatures, 400 for each category. 520 samples for each class of malware (Worm, Backdoor, Trojan-Downloader, Trojan-Dropper, and Trojan-Spy) were selected. With an N-gram analysis of the categorized sequences, class-specific patterns for all five classes of malware were retrieved. The ssdeep algorithm calculated a fuzzy hash matching score between different categorized sequences to generate a fuzzy hash-based signature ranging from 0 to 100. A higher fuzzy hash means that the two sequences belong to the same class. It could achieve an accuracy of approximately 96%.

Additionally, a Control Flow Graph (CFG) [97] with N-grams databases of API sequences was created, using similarity score methods to discover similar characteristics among malware families. 15000 malware samples from five classes (Virus, Trojan, Backdoor, Adware, and Worm) and 4000 benign PE files were generated after disassembly done by BeaEngine. The execution flow was illustrated in basic blocks (a set of instructions without branches or control transfer instructions) and edges. Dice coefficient, Cosine coefficients, and Tversky Index were used to calculate the similarity between the 2,3 and 4 grams databases. The approach comprised four phases with cumulative learning by adding malicious and benign samples from the previous phase training dataset. The false positive rate could be reduced using more benign binaries to build a training set. A detection rate of 95.06% and the false positive rate of 26.06% were achieved using 3-grams and dice coefficient.

[7] T-distributed Stochastic Neighbor Embedding (T-SNE) was a nonlinear dimension reduction for embedding high-dimension data in a lower dimensional space.

Table 2.2. A comparison of extant classification literature

Authors	**Techniques**	**Data Acquisition**	**Classification**
Rajesh Kumar et al. [69]	Rawbyte histogram, Byte entropy, Strings, Optional Header, Sections, Imports and Exports API	Static EMBER [70]	Binary
Jakub Palša et al. [34]	Imported modules and functions and Dynamic API call counts	Static and Dynamic	Binary
Li Chen, Jugal Parikh [71]	Grayscale images	Static	Binary
Chani Jindal et al. [44]	Word Embeddings, T-SNE[7]	Static EMBER [70] and Dynamic	Binary
Abdurrahman Pektaş et al. [72]	N-grams API calls	Dynamic	Binary
Joshua Saxe et al. [35]	Short character strings, N-grams	Dynamic	Binary
Jun Chen et al. [73]	API calls, word2vec [75]	Static	Binary
Azadeh Jalilian et al. [36]	N-grams Opcodes	Static	Binary
Yu Wang et al. [45, 46]	API calls	Dynamic	Binary
André Ricardo Abed Grégio et al. [33]	System calls, Clustering	Dynamic	Binary
Miles Q.Li et al. [5]	Basic Blocks of Assembly Code (Opcode and Operand)	Static	Binary
Raymond Canzanese et al. [83]	Sensors behavioral features	Dynamic	Multiclass
Sumaya Saadat et al. [84]	Grayscale images	Static	Multiclass
Zhihua Cui et al. [86]	Grayscale images	Static	Multiclass
Bojan Kolosnjaji et al. [89]	API calls	Dynamic	Multiclass
Mingdong Tang et al. [92]	Images from API calls	Dynamic	Multiclass
Hanqi Zhang et al. [94]	N-grams Opcodes	Static	Binary and Multiclass
Sanchit Gupta et al. [96]	N-grams API calls	Dynamic	Multiclass
Anishka Singh et al. [97]	CFG, N-grams API calls	Static	Multiclass

Chapter 3

Data and Labels

This chapter presents details of the experimental data, both static and dynamic extracted features used to train machine learning models for binary and multi-class classification. We also describe the labeling process that is used to establish ground truth information for all data samples. These digital artifacts could be of utmost importance in forensic investigation, particularly in malware analysis. Malicious applications often maintains persistence by modifying an entry in Windows Registry or the Startup folder to continue execution even after the system reboots. However, they still leave some traces for tracking system activities. During the data collection and preprocessing, we encounter encryption on some sample files and larger file size upload for the sandbox site, resulting in missing collection data, which are imputed with a constant value to replace all occurrences of missing values. The summary of all datasets and extracted features are depicted in Table 3.1.

3.1 Dataset Sources

We focus on host-based applications running on the Windows 10 operating system. The diagram for downloading samples and obtaining labels from public sites is shown in Figure 3.1. Both malware and benign applications are gleaned from publicly available resources, as explained in the following. When getting samples from these resources, we select well-known and frequently used

Dataset	Static		Dynamic			
	PE	Opcode	Download	Noriben	Hybrid	MemDump
No.features	429	5000	10	10000	58	64
No.classes	2	8	2	8	2	2
No.samples	1600	4000	1600	2000	1600	1600

Table 3.1. Summary of Datasets.

sources popular among security researchers. Table 3.2 depicts the summary of all samples by category. For benign samples, we use VirusTotal and HybridAnalysis for cross-verification on the files downloaded from these sources. In any machine learning problem, some labeled data will be noisy (incorrectly labeled).

3.1.1 Malware Datasets

We download malware samples from a number of publicly available collections. These are:

- Virusshare (`https://virusshare.com/`)
- theZoo - A Live Malware Repository (`https://thezoo.morirt.com/`)
- DAS MALWERK (`http://das-malwerk.herokuapp.com/`)
- Contagio Malware Dump (`http://contagiodump.blogspot.com/`)
- Syrian Malware (`https://syrianmalware.com/`)
- Malshare (`https://malshare.com`)
- URLhaus Database (`https://urlhaus.abuse.ch/browse/`)
- Virusign (`http://www.virusign.com/`)
- ANY RUN (`https://app.any.run/submissions`)

- Endermanch Repository (`https://github.com/Endermanch/MalwareDatabase`)

3.1.2 Benign Datasets

The samples for benign applications are freeware downloaded from

- Microsoft store (`https://apps.microsoft.com/store/apps`)
- Filehorse (`https://fileHorse.com`)
- Filehippo (`https://www.filehippofile.com/`)
- Softonic (`https://en.softonic.com/windows`)

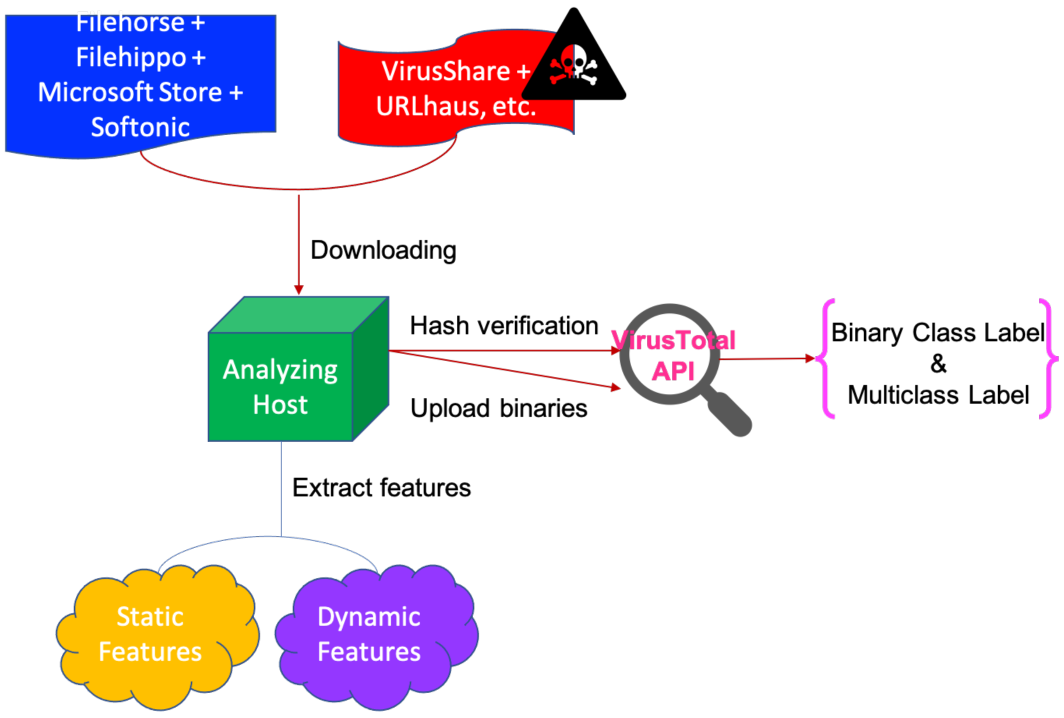


Figure 3.1. High-level Diagram for collecting samples and label retrieval.

3.2 Static Raw Data

An automated Python script is used to crawl files from an analyzing host that stores malware and benign applications to extract features for static analysis. The output from the running script is written in CSV format before

Type of Datasets	Number of instances	Category
Static and Dynamic	613	Benign
	987	Malware
Noriben Sequence	756	Benign
	1244	Malware
Opcode Sequence	744	Benign
	3256	Malware

Table 3.2. Summary of Data Category for each downloaded source.

preprocessing. We could divide static data into two parts, one being Portable Executable (PE) attributes and the other one being sequence of assembly instructions. The following subsections describe each dataset.

3.2.1 Portable Executable (PE) File Properties

A custom Python script generates all static features. We use the Python library modules "`pefile`" [30] and '`PyExifTool`" [98] to parse a binary file and carve out meaningful data. In the PE file header, we mainly analyze OPTIONAL_HEADER to extract static data such as parse_debug_directory and get_overlay_data_start_offset. We call each Python function for collecting an attribute (e.g., imported static DLLs, Imphash, manifest, pdbinfo in debug directory, rich header, overlay, and metadata) in the main function and combining all static results by writing an output file in CSV format. The raw attributes considered for analysis are as follows:

- **ImageBase**: an address in the virtual memory when the executable file is loaded, classified into default imagebase and suspicious imagebase. (Type: Categorical)
- **SI vs. SC**: a comparison between the size of the initialized data section and the code (text) section. (Type: Boolean)

- **SC vs. FS**: a comparison between the size of the code (text) section and file length specified in bytes. (Type: Boolean)

- **Sum of Raw Data**: the size of the section or the size of the initialized data on disk. In case of less than VirtualSize, zero is filled for the remainder of the section. (Type: Categorical)

- **Number of Sections**: a total number of sections in the section table. All PE files contain at least two sections: the code and the data section. The most commonly presented sections in an executable are: .text, .data, .rdata, .rsrc, .edata, .idata, and .debug. The research defines this attribute as normal and suspicious (deviating from normally presented in an executable). (Type: Categorical)

- **Imphash**: a unique value of Import Address Table (IAT) generated by the compiler's linker, based on the specific order of functions within the source file. In case two files has similar imphash values, they have the same IAT, which implies that the files are from the same malware families or performing similar activities. (Type: Categorical)

- **PDB Path**: Microsoft linker stores a directory path to a program database (PDB) file made at link-time in the PE Debug Directory. In this research, we set TRUE/FALSE for a debug file stored in the User/Document folder. (Type: Boolean)

- **Correlated Imported DLLs**: all imported libraries needed for applications to run. We split library functions into sub-features in terms of boolean. (Type: Boolean)

- **Anti-Debugger**: inspects whether anti-debugging is implemented to hinder debugging tools from observing the flow of malware. We split the anti-debugging module into sub-features in terms of boolean. (Type: Boolean)

- **App Manifest**: inspects the privilege of the application request. (Type: Categorical)
- **Rich Header**: a checksum match in rich header [99] that is used to identify similar malware and characterize the built environment. If the packer is implemented, the rich header might be omitted and/or corrupted. (Type: Categorical)
- **Overlay**: data appended at the end of PE file and not mapped into memory. (Type: Categorical)
- **File Signature**: inspects whether the file's signature is modified. (Type: Categorical)
- **File modification time vs. Created time**: a comparison between the date the application is compiled and the last modification date/time of a file. (Type: Boolean)
- **XOR**: inspects whether XOR is implemented, which is one attribute used to draw conclusion for encryption. (Type: Boolean)
- **Packer**: inspects which section implements compression. (Type: Categorical)
- **Entropy**: a maximum value of all sections table existing in a file. (Type: Numeric)
- **Obfuscation**: statically disassembled binaries by starting from an entry-point plus 2000 bytes to observe the instructions, without consideration of "rdtsc/cpuid" or consecutive "mov" instructions. (Type: Categorical)
- **File Description vs. Company Name**: a comparison between the file's description and the company name. (Type: Boolean)
- **File Description vs. Product Name**: a comparison between the description of the file and the name of the product that produces the application. (Type: Boolean)

- **Internal Name vs. Original Filename**: a comparison between the internal name embedded in the file and the original filename. (Type: Boolean)

3.2.2 Opcode Sequences

To fully automate the extraction process, a number of Python scripts are implemented. The main steps taken by this automated system for the extraction of opcode sequences are:

- Checking what kind of compression algorithms is implemented: (1) UPX, (2) Mono/.Net, (3) Zlib.
- Disassembling the executables to obtain disassembled files.
- Searching and removing flag/segment registers, interrupt instruction, illegal/(bad), and dead code.
- Extracting opcodes in a length of first 5000.

All samples are inspected by the "`file`" command to verify whether they have a "`PE`" signature and what type of compression algorithms have been implemented. Three packer types are investigated: (1) UPX [40], (2) Zlib [41], and (3) Mono/.Net. For UPX and Zlib, they are parsed to UPX unpacker and Zlib, respectively. Meanwhile, "`monodis`" [42] is used to disassemble .NET Common Intermediate Language (CIL) code and metadata. If it successfully decompresses files, it write an output file. Otherwise, it is parsed by the Capstone Python library [100] and then"`objdump`" [43] with the -d option to disassemble only sections expected to contain code. Figure 3.2 provides an example of the first ten opcodes of a sample of each of the seven malware categories.

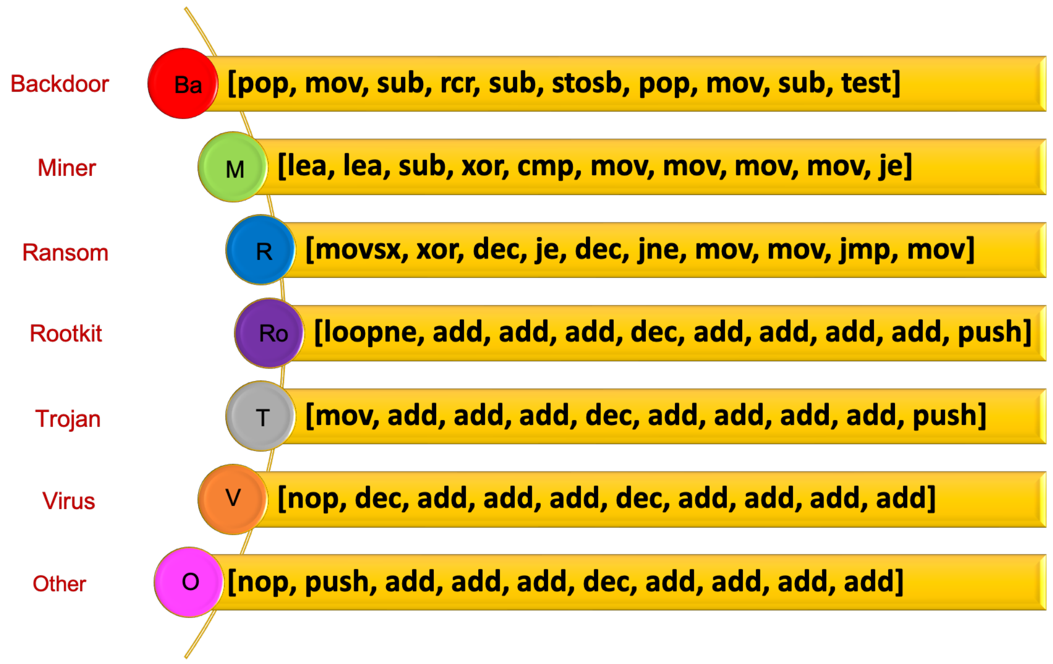

Backdoor
Ba
[pop, mov, sub, rcr, sub, stosb, pop, mov, sub, test]
Miner
M
[lea, lea, sub, xor, cmp, mov, mov, mov, mov, je]
Ransom
R
[movsx, xor, dec, je, dec, jne, mov, mov, jmp, mov]
Rootkit
Ro
[loopne, add, add, add, dec, add, add, add, add, push]
Trojan
T
[mov, add, add, add, dec, add, add, add, add, push]
Virus
V
[nop, dec, add, add, add, dec, add, add, add, add]
Other
O
[nop, push, add, add, add, dec, add, add, add, add]

Figure 3.2. The first 10 opcodes for each sample in each malware category with Ba(ckdoor), M(iner), R(ansom), Ro(otkit), T(rojan), V(irus), and O(ther).

3.3 Dynamic Raw Data

To acquire dynamic features, we create a secure virtualized environment for application execution. Before uploading applications to the guest operating system, Windows Real-Time Protection (RTP) is disabled automatically. However, Windows Defender Antivirus remains active and could detect suspicious files and place them in quarantine. To prevent termination of the active process, we pre-install a Microsoft tool named "gflags.exe" [101] and then specify the malware name for monitoring silently. Besides, we set up a Custom Dump to Type 2 to create an entire dump file in case of the process termination/unresponsiveness. The default location for dump files is %TEMP%/Silent Process Exit. So, we could obtain a terminated process identifier (PID) for memory analysis.

During the runtime, we download event logs, prefetch files, temporary file location, Noriben operations [51] to the analyzing host. We also submit binary files to HybridAnalysis sandbox [47], which offers public malware analysis service. Then, we retrieve the reports from the sandbox site and combine them with our local results for dynamic analysis. In addition, we extract data from memory analysis which helps malware analysts examine more malware behavior, such as child processes, loading modules, and enabled privileges. From a forensic's perspective, memory is fragmented, and not all data might be in memory at the time of capture. The dumped memory file is parsed by the Volatility framework [48] for analysis. The forensic artifacts retrieved from Windows 10 are presented with the ultimate goal of observing abnormal application activities.

3.3.1 Downloading Folders

We use our Python script to automate uploading binaries and download eight directories from the guest machine to the analyzing host after a sample completed execution via *vmrun* "`CopyFileFromGuestToHost`" command for data extraction. The Python library module to parse files and folders for prefetch and event logs are "`pyscca`" [102] and "`python-evtx`" [103], respectively. All findings copied from the guest virtual machine are listed below:

- **Prefetch**: This folder provides storage for recently cached applications run on the machine. It helps speed up loading applications and records the last time the application is executed, even if it is uninstalled. All libraries are loaded and displayed while the application is running. The location for this file is in the Prefetch folder under the Windows directory. In addition, we could see a child executable file left in the Program folder after the application installation from loaded resources.

- **User and System Folder**: all temporary files that occur from application execution are stored in the Temp folder, such as a spawned application from the main application. Often programs create working copies of opened documents in this area, changing their extensions. Some programs delete them when done, and others leave them behind as garbage. The artifacts are stored in two locations for recently viewed documents in the system: C:\Windows\Temp (Window temporary files) and C:\Users\%profile% \AppData\Local\Temp (This folder contains temporary information while a process is creating other files). In the experiment, we change the location of the default temp folder for the user and system to a newly created drive to facilitate the collection of temporary files. By setting Environment Variables in Advanced System Settings, we type a new path for user variables (TEMP and TMP) to store files in the temporary folder.

- **Event Logs**: Windows event logs consist of system and application records of when applications performed activities on the system. Three main event categories (system, application, and security) are considered in combination with Windows PowerShell and Anti-Malware Scan Interface. Such logs are saved in the directory %System32%\winevt\Logs in binary XML format. For example, they include information like the date and time, source, fault type, and a Unique ID corresponding to the event type that incorporates Windows events. We focus on only three types of event logs, which are:

  - Error: indicates an occurrence of a significant problem–for instance, when a service fails to load at startup.
  - Warning: not an important event, but might lead to serious problems in the future.
  - Information: indicates the successful operation of a service, application, or driver.

  Event log entries comprise the following attributes:

  - User: username of the account logged in when the event occurred.
  - Event ID: a unique identifier generated by Windows to identify the event type.

  - Source: the object which caused the event to occur.
  - Computer: computer name where the event occurred.
  - Date and time: the date and time when the event occurred.
  - Description: details of what happened to launch the event.

  In addition, PowerShell commands/scripts could be acquired at the event log file name "`Microsoft-Windows-PowerShell%4Operational.evtx`" by enabling the following options in Group Policy:

  - Module log: setting module name such as Microsoft.PowerShell.*
  - script block logging: setting to record PowerShell script inputs called interactively or automatically.
  - transcription: setting to capture the input and output of Windows PowerShell commands as text-based transcripts.

- **Registry**: It records program execution, application settings, malware persistence, and other artifacts. UserAssist data is parsed from the NTUSER.dat registry file and attributes the user's execution of GUI applications. Every program executed from explorer.exe generates such a key. In addition to the user registry file, SYSTEM and SOFTWARE keys must be considered for the evidence. In Windows 10, the Background Activity Monitor (BAM) entries within the SYSTEM key detailed the path of executable files and the last execution date and time, which is in the folder of the security identifier (SID) associated with a user.

- **JumpList**: This file records recently viewed or accessed files generated by the system. It is located in the AutomaticDestinations folder under the C:\Users\%profile%\AppData\Roaming\Microsoft\Windows\Recent whose filename is a suffix with AutomaticDestination-ms file.

- **Activity Cache**: This database file is in SQLite format and shows applications that are executed within the last 30 days by the user. It contains entries that include files opening/access, program execution with content and associated Unix timestamp, the path for executables, executable names, and the expiration time. The database storing the user's activity is located in C:\Users\%profile%\AppData\Local\ConnectedDevices\Platform\L.%profile%\ActivitiesCache.db and each user account has its own database. This file could be correlated with prefetch artifacts to track malware activities, thus creating a timeline. However, some applications might not be displayed in "Timeline" after being opened. Unfortunately, any information extracted from the Activity Cache might be incomplete.

- **Shortcut files**: the recently opened files in the Recent folder that stores the timestamps, the name and location of the original file, and the volume name.
- **Shimcache**: It identifies application compatibility issues for older versions of Windows and only contained information about executables, the last modified timestamp of the file, and file size.

After we investigate all downloaded folders, the last four artifacts do not provide substantial results. That means no useful recording from experimental data to state the misbehavior of the application. So, we do not consider this when preparing data as inputs to learning models.

3.3.2 HybridAnalysis Report

HybridAnalysis is a free automated malware analysis service that allows users to drag/drop and browse a file with a maximum uploaded size of 100MB. In addition to file scanning, the site provides a report search via IP address, domain, and hash. We use our Python script to perform a file submission, download a report, and carve out significant data for learning models. To download the sandbox report, we access the HybridAnalysis site by sending an HTTP request via public API [47] with the environment identifier to retrieve the data and write the output as an HTML file. This sandbox provides analysis environment options with Windows 7 64-bit as the highest operating system. A diagram for uploading an executable and retrieving a report is presented in Figure 3.3. After we open the output file, we only consider malicious, suspicious, and informative indicators. For each indicator, we mainly focus on the anchor element (<a> HTML element) under the group elements (<span class="list-group-item-heading"> HTML element) and save it in CSV format to be preprocessed later. A sample of the file analysis from the site is depicted in Figure 3.4. The difference in operating systems has no hindrance to analysis. We use the Pandas merge Python module to combine sandbox reports and other features on a filename column. All dynamic features acquired from the analysis site and preprocessed as boolean attributes for machine learning models are listed below:

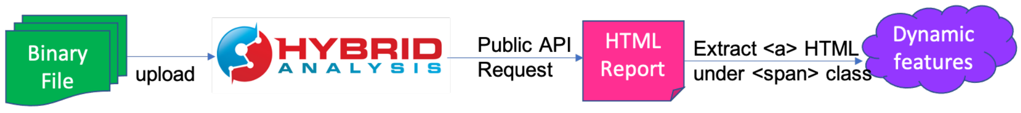
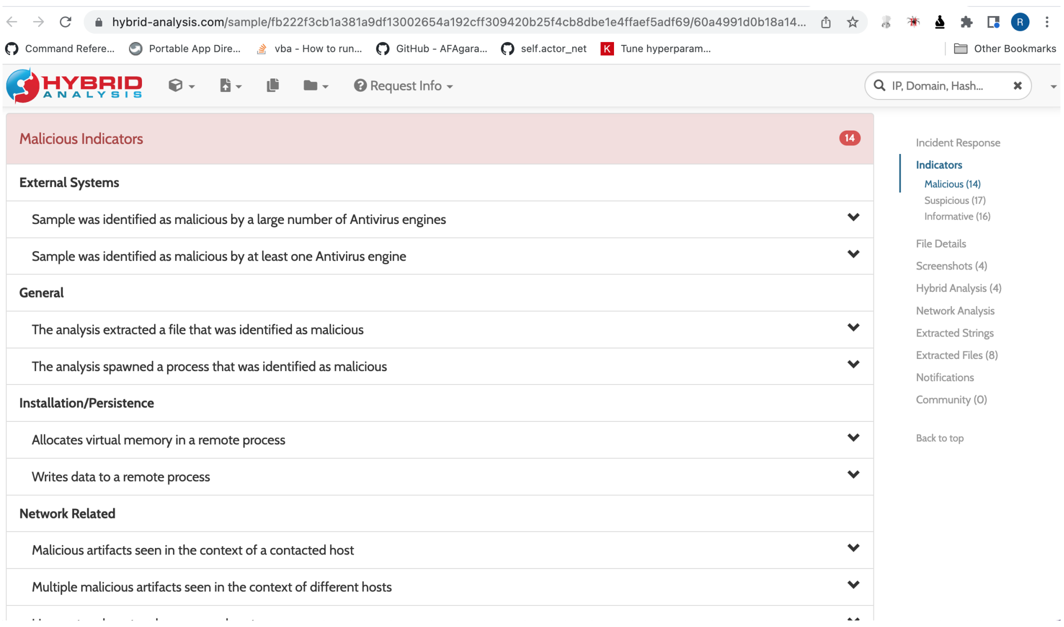


Figure 3.3. An illustration of uploading a file and obtaining a report from the sandbox site with feature extraction.

- **Opens the Kernel Security Device Driver (KsecDD) of Windows**: checks whether the Kernel Security Device Driver is opened.
- **Modifies Software Policy Settings/proxy settings/system Certificates**: checks whether configured settings are modified, such as proxy settings, system certificates, and auto-execute functionality.
- **Creates or modifies windows services**: checks whether a new windows service on the system is launched.
- **Queries sensitive IE security settings**: queried sensitive information from the Internet Explorer security settings.
- **Contains ability to elevate/lookup privileges**: an ability to query and escalate privileges.


Malicious Indicators 14

External Systems

Sample was identified as malicious by a large number of Antivirus engines

Sample was identified as malicious by at least one Antivirus engine

General

The analysis extracted a file that was identified as malicious

The analysis spawned a process that was identified as malicious

Installation/Persistence

Allocates virtual memory in a remote process

Writes data to a remote process

Network Related

Malicious artifacts seen in the context of a contacted host

Multiple malicious artifacts seen in the context of different hosts

Figure 3.4. A sample of miner analysis via HybridAnalysis whose artifacts are obtained by only considering $< a >$ HTML tag. For example, the HyperText “Allocates virtual memory in a remote process” and “Write data to a remote process” under **Installation/Persistence** are the attributes.

- **Enumerate process/module/thread**: an ability to perform process, module, and/or thread enumeration.
- **Retrieve keyboard strokes**: an ability to capture keyboard usage while the user is typing messages on the system.
- **POSTs/GETs files to a webserver**: checks whether it sends a request to some web-servers.
- **Opens file with deletion access rights**: checks whether files are opened with deleted privilege access.
- **Cryptographic Related**: checks if an encryption/decryption function is called during runtime.
- **Drop files/executable files**: checks whether an executable file is dropped onto the compromised machine.
- **Disables Command Prompt (cmd)/Disables Windows Registry**

Editor (regedit): checks if a function to disable running the command prompt is called.

- **Contains ability to query machine time/timezone/version**: an ability to query the machine timezone, machine version, and volume information.
- **Pattern Matching_YARAmatch**: checks whether there is a match with a YARA rule [47].
- **Pattern Matching_DownloadFileFromInternet**: checks whether there is a match for a file downloaded from the Internet.
- **URL_IPaddr_in_Binary**: checks whether there is a URL and/or IP address embedded in the binary file.
- **Contact_Random_DNS/Contacts diff hosts**: checks whether there is an attempt to contact different hosts with random domain names.
- **Unusual ports for process_and_protocol**: checks whether there is unusual connection establishment.
- **Touches files in the Windows directory**: an attempt to interact with files in prohibited Windows directories for persistence in order to maintain access in the future.
- **Connects to LPC ports**: an attempt to connect via an LPC port for persistence.
- **Monitors specific registry key for changes**: monitors any registry key change attempts for persistence.
- **Contains ability to lookup the windows account name**: an ability to search for Windows's account for persistence.
- **Writes data to a remote process**: an attempt to write data to a remote process for persistence.

- **Allocates virtual memory in a remote process**: checks whether there is allocated virtual memory for persistence.
- **Marks file for deletion**: an attempt to mark some files for persistence.
- **Environment Awareness_Sleep**: checks whether the sleep function is called.
- **Anti-Virtualization/Anti-VM trick**: checks whether virtualization detection is triggered during the application's installation.
- **Executes WMI queries known to be used for VM detection**: checks whether there is a reference to a WMI query string known for VM detection, remote access, or request to get CPU temperature, BIOS version, and serial number.
- **References security-related windows services**: checks whether there is a reference to windows security services.
- **Remote Access_Reads terminal service related keys (often RDP related)**: checked whether it attempts to perform an RDP connection.
- **Remote Access_Contains indicators of bot communication commands**: checks if it contains bot commands for remote execution.
- **Spreading_Opens the MountPointManager**: checks whether it uses MountPointManager to spread an infection.
- **AR_Creates guarded memory regions (anti-debugging trick to avoid memory dumping)**: checks whether it creates a memory section to avoid memory dumping.
- **AR_PE file contains zero-size sections/PE file has unusual entropy sections/Unusual_PE file contains unusual section name**: checkes whether the PE file contains zero-size sections or sections of an unusual entropy.

- **AR_Possibly checks for known debuggers/analysis tools**: checks whether there is a debugging and/or monitoring tool installed on the system.

- **AR_Contains ability to register a top-level exception handler (often used as an anti-debugging trick)**: an ability to register exception handler which is often used as an anti-debugging trick.

- **Unusual_Imports suspicious APIs/References suspicious system modules**: checks for imports of inappropriate modules to install applications.

- **Unusual_Installs hooks/patches the running process**: hooks or patches the process in case of an error occurring at runtime.

- **Read keyboard layout/Unusual_Reads information about supported languages**: an ability to read keyboard layout/language supported on the infected machine.

- **Unusual_Matched Compiler/Packer signature**: checks whether there is a match for the compiler/packer signature.

- **Unusual_Contains ability to reboot/shutdown the operating system**: an ability to run a command to reboot or shutdown the system after activity completion.

- **Unusual_Contains native function calls**: checks whether there are any unusual native function calls.

- **Unusual_Tries to access unusual system drive letters**: an attempt to access or query a system drive.

- **Contains ability to open/control a service**: an ability to open/control a new service.

- **Fingerprint_Reads the active computer name, the cryptographic machine GUID**: fingerprints the computer name and cryptographic machine GUID.

- **Queries kernel debugger information/Queries process information**: queries the kernel debugger and process information.

- **Creates a resource fork (ADS) file (often used to hide data)/Checks for a resource fork (ADS) file**: checks for the creation of an alternate data stream file, which is often used to hide data.

- **Check Presence of an Antivirus engine/adware detecting tool**: checks for any antivirus tools running on the infected machine.

- **Terminates other processes using tskill/taskkill**: terminates processes by using the tskill or taskkill command.

- **Process deletes itself**: checks whether a process deleted itself after activity completion on the infected machine.

- **Hide tracks of having downloaded a file from the internet/hide a process launching it with different user credentials/Modifies file/console tracing settings**: checks whether there is an attempt to download files from the Internet by the current user.

- **Opened the service control manager**: launches the service control manager to install or open services.

- **Runs shell commands**: executes shell scripts or commands on the infected machine.

- **Loads the .NET runtime environment/Loads the visual basic runtime environment**: loads the .NET framework or Visual Basic language to complete the application installation.

- **Contains ability to start/interact with device drivers/Contains ability to find and load resources of a specific module/Contains**

ability to register hotkeys: an ability to access device drivers and load resources onto the infected machine.

- **Unique CLSIDs touched in registry**: an access to the unique number of a software application.
- **Extracted Files**: checks whether there are hidden files in binaries such as ransom note files.
- **Connection via port 80/443**: checks whether there is standard HTTP and HTTPS connection establishment.

3.3.3 Memory Artifacts

In addition to sandbox data, we analyze the virtual memory file captured after completing execution. After the virtual machine is suspended for taking a snapshot, the machine is reverted to a clean initial snapshot. Volatility [48] parsed the 4GB memory dump (.vmem) file via our custom Python script to capture memory artifacts. While conducting the memory dump analysis, the following raw attributes are extracted:

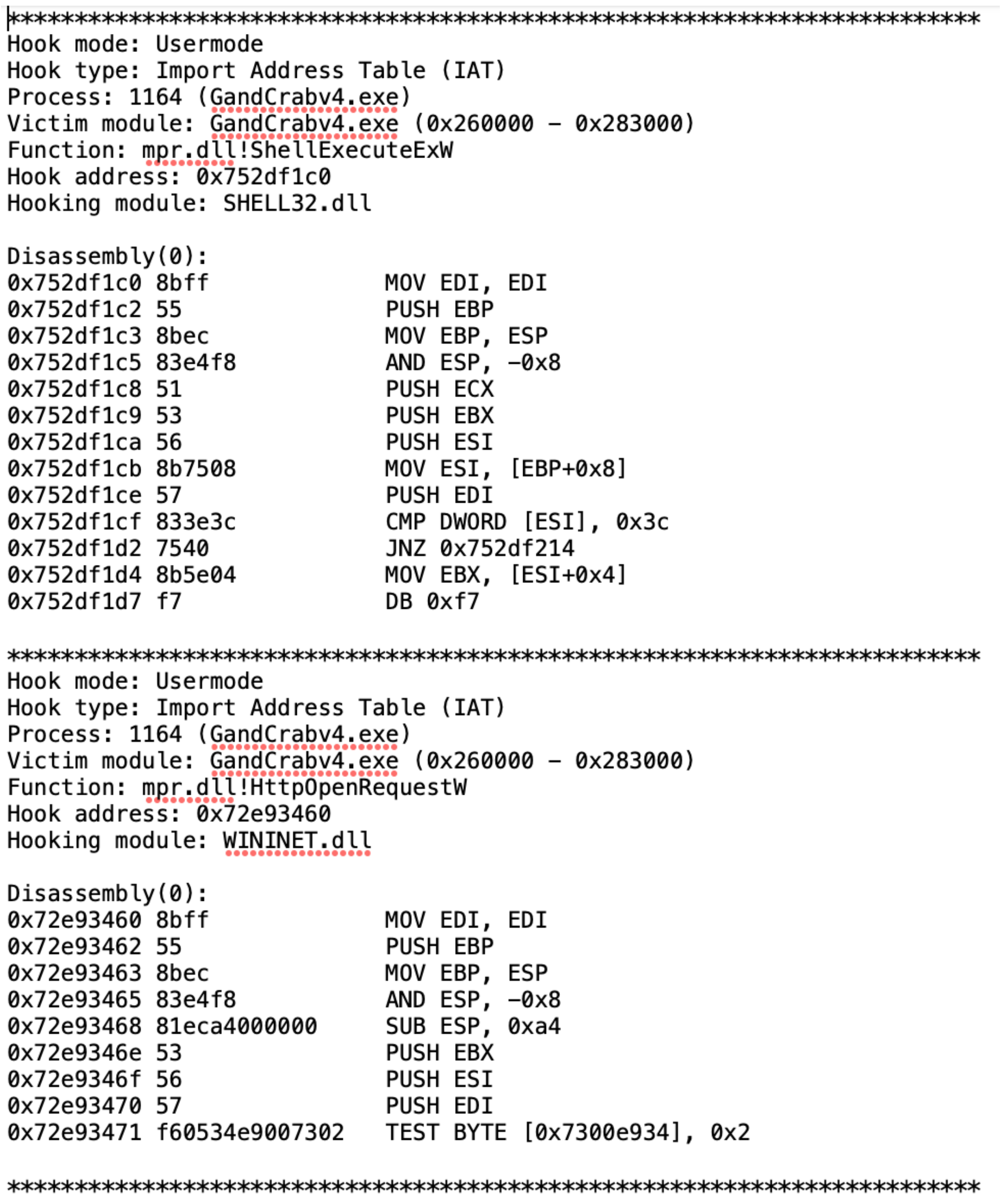
- **Wow64**: checks whether the process uses a 32-bit address space on a 64-bit kernel. (Type: Numeric)
- **Number of Pstree**: lists the processes in tree shape by using indention and periods. This measures the number of parent and child processes. (Type: Numeric)
- **LDR_InLoad_InInit_InMem**: do memory-mapped files with the three PEB lists, which shows the TRUE/FALSE status if the PE exists in the PEB lists. (Type: Boolean)
- **Dll_Command**:displays a process's loaded DLLs with the executed commands by walking through the doubly-linked list pointed by the PEB's InLoadOrderModuleList. (Type: Boolean)

- **Spawn process**: checks whether there are child processes with the same parent processID. (Type: Boolean)
- **psxview_hidden_process**: compares active process indicated within PsActiveProcessHead with other sources of process listings. A "False" in any source indicated that the respective process was missing. (Type: Boolean)
- **Virtual Address Descriptor (VAD) and page permissions**: checks whether PAGE_ EXECUTE_READWRITE found in running process. If it exists, it means no memory-mapped file is occupying the space. (Type: Boolean)
- **Creates mutants**: scans physical memory for KMUTANT objects with pool tag scanning to display named mutex of the running process if it exists. This feature is correlated with the result from the HybridAnalysis report. (Type: Boolean)
- **Privilege**: shows process privileges that are not enabled by default but explicitly enabled. We split these privileges into separate columns for preprocessing. (Type: Boolean)
- **API hooks**: shows hook mode, hook type, victim module, and function in memory when checking for hooks. We split these API hooks into separate columns for preprocessing. (Type: Boolean)

A sample of memory analysis with Volatility [48] written to an output text file is shown in Figure 3.5. The result is obtained using the apihook Volatility plugin to enumerate IAT, EAT, and Inline hooks, and list all hooked functions and modules relevant to the application process.

3.3.4 Noriben Operations

We use Noriben [51], a Python-based script that works jointly with Sysinternals Procmon, to automatically collect, analyze, and report on the application's



```
************************************************************************
Hook mode: Usermode
Hook type: Import Address Table (IAT)
Process: 1164 (GandCrabv4.exe)
Victim module: GandCrabv4.exe (0x260000 - 0x283000)
Function: mpr.dll!ShellExecuteExW
Hook address: 0x752df1c0
Hooking module: SHELL32.dll

Disassembly(0):
0x752df1c0 8bff             MOV EDI, EDI
0x752df1c2 55               PUSH EBP
0x752df1c3 8bec             MOV EBP, ESP
0x752df1c5 83e4f8           AND ESP, -0x8
0x752df1c8 51               PUSH ECX
0x752df1c9 53               PUSH EBX
0x752df1ca 56               PUSH ESI
0x752df1cb 8b7508           MOV ESI, [EBP+0x8]
0x752df1ce 57               PUSH EDI
0x752df1cf 833e3c           CMP DWORD [ESI], 0x3c
0x752df1d2 7540             JNZ 0x752df214
0x752df1d4 8b5e04           MOV EBX, [ESI+0x4]
0x752df1d7 f7               DB 0xf7

************************************************************************
Hook mode: Usermode
Hook type: Import Address Table (IAT)
Process: 1164 (GandCrabv4.exe)
Victim module: GandCrabv4.exe (0x260000 - 0x283000)
Function: mpr.dll!HttpOpenRequestW
Hook address: 0x72e93460
Hooking module: WININET.dll

Disassembly(0):
0x72e93460 8bff             MOV EDI, EDI
0x72e93462 55               PUSH EBP
0x72e93463 8bec             MOV EBP, ESP
0x72e93465 83e4f8           AND ESP, -0x8
0x72e93468 81eca4000000     SUB ESP, 0xa4
0x72e9346e 53               PUSH EBX
0x72e9346f 56               PUSH ESI
0x72e93470 57               PUSH EDI
0x72e93471 f60534e9007302   TEST BYTE [0x7300e934], 0x2

************************************************************************
```

Figure 3.5. Using Volatility's apihook plugin shows detection of IAT hook in User mode for a ransomware sample with the hooked functions as ShellExecuteExW and HttpOpenRequestW.

activities during the runtime. We put the Noriben Python script [51] onto the Windows desktop with the Sysinternals Procmon in the Python folder. First, the script is called via PowerShell command line during the runtime by setting the number of seconds to collect activity for 300 seconds (5 mins), as shown in Figure 3.6. We create an output folder named "Noriben_Logs" to store logging files and then use "`zip`" [104] to compress the output and download it to the analyzing host for analysis. Finally, to extract a sequence of operations, we decompress it by using the Python module "`zipfile`" [105] to extract all CSV files, focus only on the filename and test application, and generate a CSV output file to combine with other findings. An example of the first ten Noriben operations with no repeated occurrences is given in Figure 3.7.

```
["powershell.exe -Command Start-Process cmd \"/c C:\\Python3.8\\python.exe
C:\\Users\\koyko\\Desktop\\Noriben.py -d -t 300 --cmd
C:\\Users\\koyko\\Desktop\\test.exe --output G:\\Noriben_Logs\""]
```

Figure 3.6. Script execution to capture applications' activities.

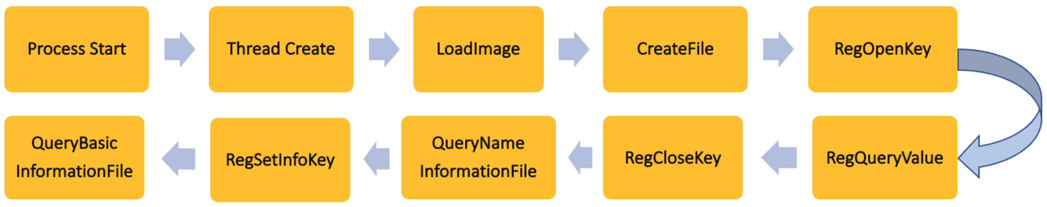
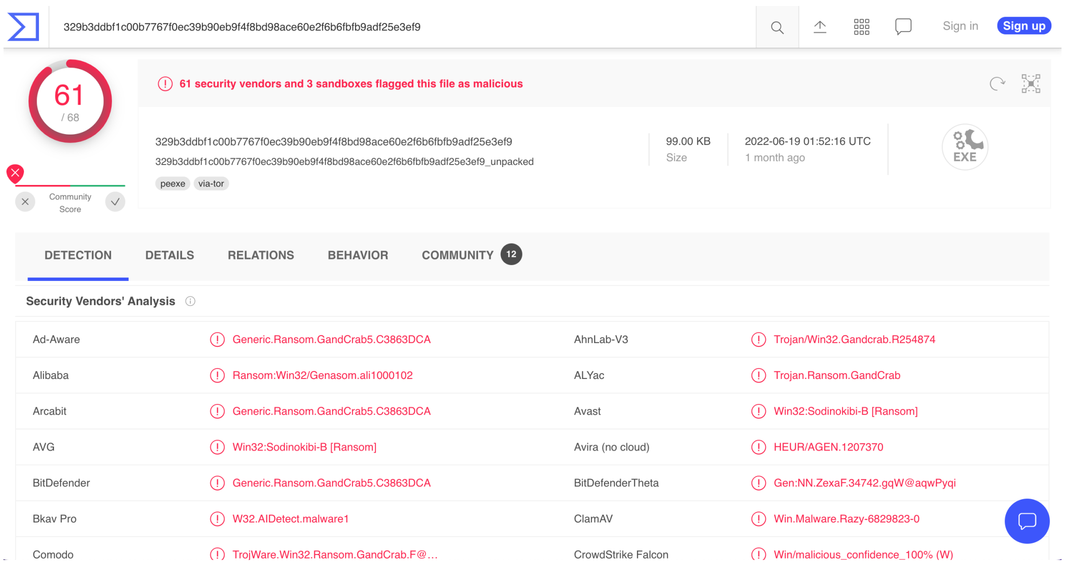


Figure 3.7. The first 10 unique Noriben operations in sequence order, for some sample applications.

3.4 Label Retrieval

To derive labeling, we use API public service from VirusTotal [106]. At first, we verify these files by searching reports via SHA256 hash to fetch the predictive label. Otherwise, binaries are uploaded. To flag as malware on a given file, at least 40 of 70 antivirus engines provided by VirusTotal [106] are considered for binary classification while using five antivirus engines at minimum to verify multiclass malware problems. Moreover, we use the Microsoft Virus/Threat detection feature to verify the types of malware. The scanning results for some sample ransomware and benign uploaded to VirusTotal are shown in Figure 3.8 and 3.9, respectively.



329b3ddbf1c00b7767f0ec39b90eb9f4f8bd98ace60e2f6b6fbfb9adf25e3ef9

61 security vendors and 3 sandboxes flagged this file as malicious

329b3ddbf1c00b7767f0ec39b90eb9f4f8bd98ace60e2f6b6fbfb9adf25e3ef9
329b3ddbf1c00b7767f0ec39b90eb9f4f8bd98ace60e2f6b6fbfb9adf25e3ef9_unpacked
peexe via-tor

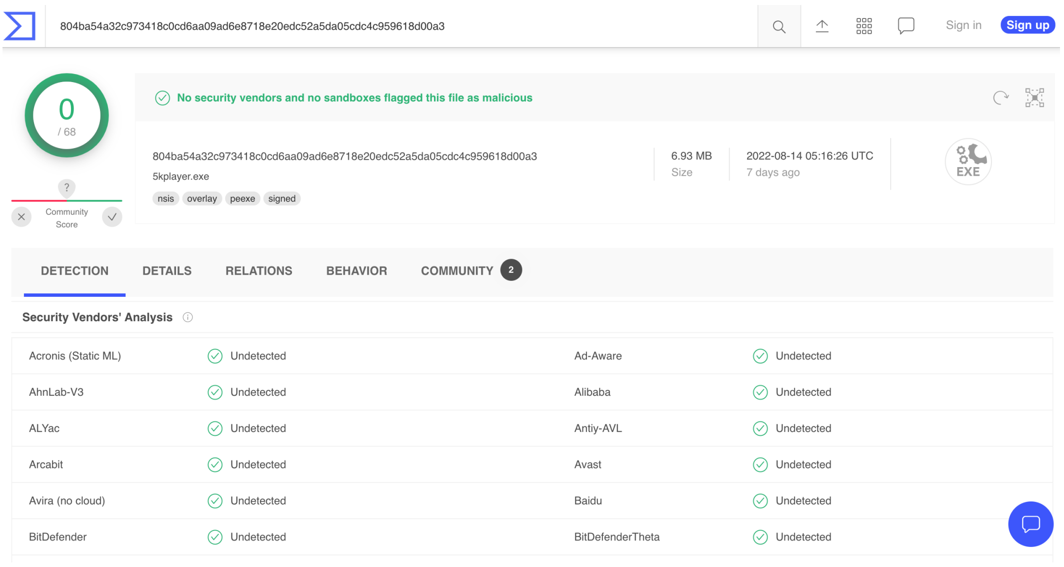
99.00 KB
Size

2022-06-19 01:52:16 UTC
1 month ago

Security Vendors' Analysis

Ad-Aware	Generic.Ransom.GandCrab5.C3863DCA	AhnLab-V3	Trojan/Win32.Gandcrab.R254874
Alibaba	Ransom:Win32/Genasom.ali1000102	ALYac	Trojan.Ransom.GandCrab
Arcabit	Generic.Ransom.GandCrab5.C3863DCA	Avast	Win32:Sodinokibi-B [Ransom]
AVG	Win32:Sodinokibi-B [Ransom]	Avira (no cloud)	HEUR/AGEN.1207370
BitDefender	Generic.Ransom.GandCrab5.C3863DCA	BitDefenderTheta	Gen:NN.ZexaF.34742.gqW@aqwPyqi
Bkav Pro	W32.AIDetect.malware1	ClamAV	Win.Malware.Razy-6829823-0
Comodo	TrojWare.Win32.Ransom.GandCrab.F@...	CrowdStrike Falcon	Win/malicious_confidence_100% (W)

Figure 3.8. A ransomware sample analyzed via VirusTotal is flagged as malicious with more than 60 antivirus engines in 70 engines.



804ba54a32c973418c0cd6aa09ad6e8718e20edc52a5da05cdc4c959618d00a3

No security vendors and no sandboxes flagged this file as malicious

804ba54a32c973418c0cd6aa09ad6e8718e20edc52a5da05cdc4c959618d00a3
5kplayer.exe
nsis overlay peexe signed

6.93 MB
Size

2022-08-14 05:16:26 UTC
7 days ago

Security Vendors' Analysis

Acronis (Static ML)	Undetected	Ad-Aware	Undetected
AhnLab-V3	Undetected	Alibaba	Undetected
ALYac	Undetected	Antiy-AVL	Undetected
Arcabit	Undetected	Avast	Undetected
Avira (no cloud)	Undetected	Baidu	Undetected
BitDefender	Undetected	BitDefenderTheta	Undetected

Figure 3.9. A benign sample analyzed via VirusTotal is flagged as benign with zero detection from all antivirus engines.

3.5 Code Repository

The Python codes for data extraction on raw data from Windows Portable Executable files are put in the Github repository, which can be accessed via: `https://github.com/rajchadach/malware`. On the one hand, static and dynamic extracted features are extracted and produced in the output files as “static.csv” and “hybrid.csv”, combining these two files into a single file before performing data preprocessing. We read static and hybrid CSV files and then merge them on the filename column using the Pandas Python module. The final number of features per instance after performing this combination is 561 features. On the other hand, each operation/instruction is extracted from applications and inserted into each column to complete the sequence length for Noriben and opcodes. The output files from these two sequences are “noriben.csv” and “opcode.csv”, respectively. Later, these sequences are used by Tensorflow [53] for text preprocessing.

Chapter 4

Combining Static and Dynamic Analysis to Improve Machine Learning-based Malware Classification

Portable Executable (PE) is a file format for executables that could be produced maliciously in Windows operating systems. The file structure and execution flow could be changed to deceive standard security detection and protection measures. For instance, creating a writable file in a user's temporary folder to bypass Windows Virus and Threat Protection, whose filename seemed like a legitimate process (e.g., svchost.exe, chrome32.exe, and dllhost32.exe), and required no user intervention. In this chapter, we leverage static properties and dynamic behavior analysis for binary malware classification. For dynamic analysis, we retrieve information from the HybridAnalysis sandbox website [47] in conjunction with running malware in an isolated Windows 10 environment to extract memory dumps and then combined it with the sandbox report. We use three classifiers in our empirical experiments: random forest [59], gradient boosting [60], and neural networks [61].

4.1 Data Collection and Preprocessing

The dataset includes a total of 1600 executable files—987[1] malicious and 613[2] benign. While the malicious samples come from the VirusShare repository [90], the benign samples are taken from the FileHorse [108]. The labels are obtained by searching via SHA256 hash, submitting the files to VirusTotal, and using at least 40 AV engines to verify them.

A total of 561 features, including static and dynamic features, are gathered using a Python script. Data shuffling and splitting are performed for all malware and benign samples, with 80% for training. On average, missing values are around 4%, with the highest being 13% for one attribute. The percentages of missing values per attribute on static and dynamic features can be seen in Figure 4.1 and Figure 4.2, respectively. They are replaced by using Scikit-Learn's *SimpleImputer* [54], using the constant strategy and setting a default for the *fill_value*. We impute numerical data with zero and object data types with "missing_value". OnehotEncoding [54] is used to convert categorical variables into numerical values, with the "ignore" setting for the handling of missing values. This setting will output a zero value for an unknown categorical feature value during transformation. To combine all data preprocessing steps into one, a *ColumnTransformer* [54] is used to apply it to all 8 numerical and 553 categorical features. The diagram for data preprocessing and feature extraction before passing through learning approaches is depicted in figure 4.3.

Notably, the feature importance analysis is vital in optimizing model performance. SHapley Additive exPlanations (SHAP) [52] is used to calculate how much each feature contributed to the model's prediction based on a training data split from the initial set of features. From the SHAP Python [109], the TreeSHAP and DeepSHAP functions are applied to the static and dynamic attributes described in chapter 2. The features from static-only (429 attributes) and the dynamic-only (132 attributes) full sets are chosen before

[1]Downloads from VirusShare [90] and VirusSign [107]

[2]Downloads from FileHorse [108]

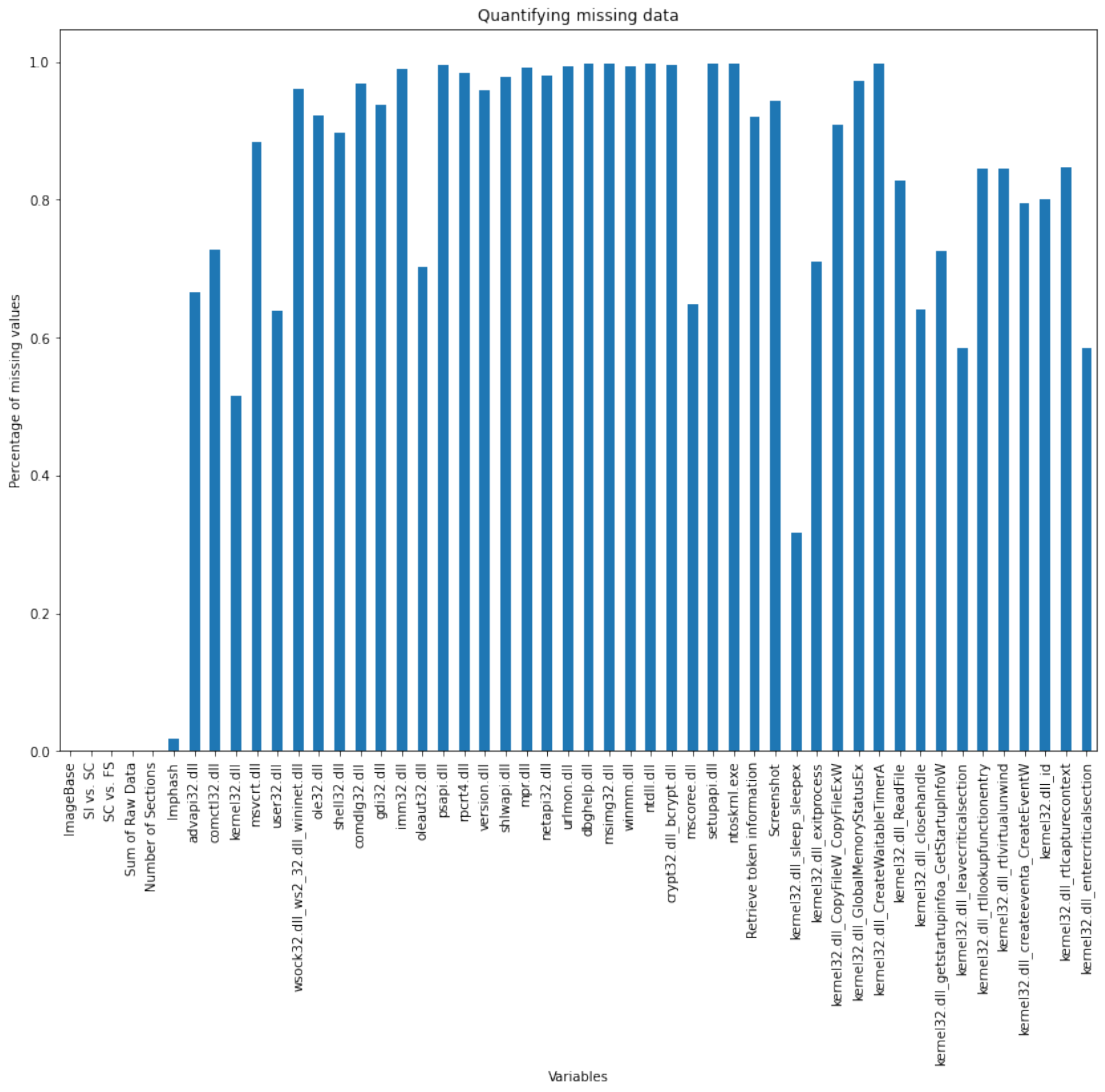


Figure 4.1. The bar plot displays the percentage of missing data for the first 50 static features.

the two types of selected features were assembled. We use this SHAP Python library to calculate SHAP values and plot charts. First, the learning model is trained on the initial training set of features, and each feature's importance is obtained by comparing model predictions with and without the feature. Then, each feature's SHAP importance is computed individually by taking the average of the absolute SHAP values across the data. Before the most significant impact of 50 static and 50 dynamic attributes were chosen, we sort them in descending order according to their importance. Using the Tree and Deep explainer, we have generated a summary plot based on SHAP values from train

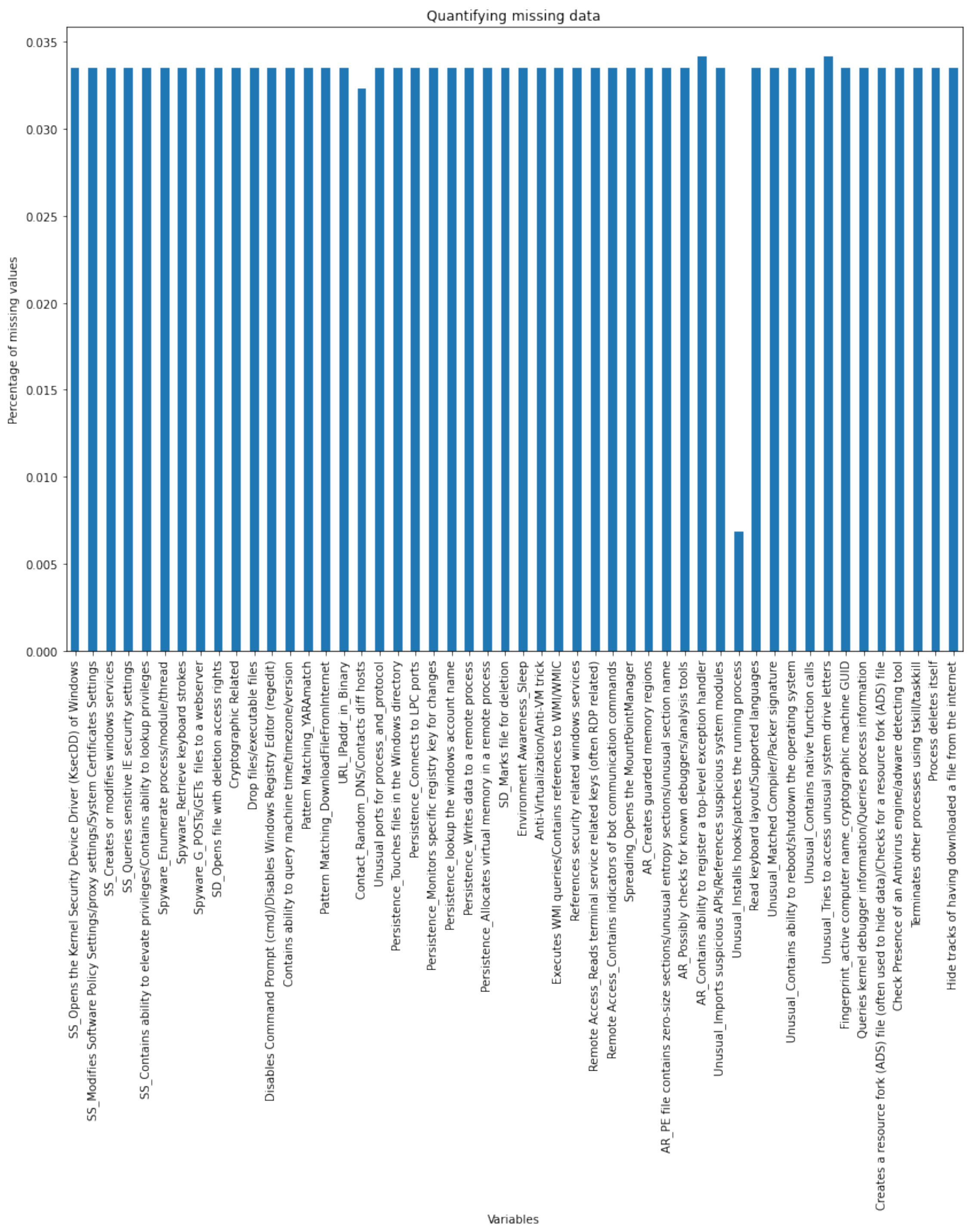


Figure 4.2. The bar plot displays the percentage of missing data for the first 50 dynamic features. Most missing values that seem equal mean no report extraction from HybridAnalysis [47] due to the limited upload file size (100MB). These values are left empty to avoid confusion with the "false" value in case of no attribute in the report.

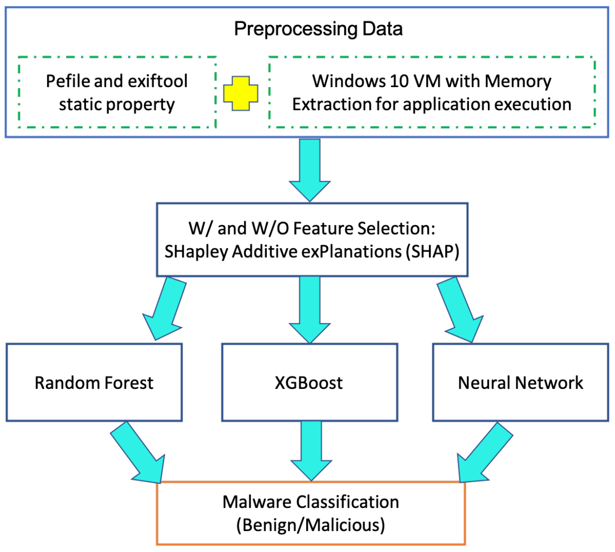

Preprocessing Data
Pefile and exiftool static property
Windows 10 VM with Memory Extraction for application execution
W/ and W/O Feature Selection: SHapley Additive exPlanations (SHAP)
Random Forest
XGBoost
Neural Network
Malware Classification (Benign/Malicious)

Figure 4.3. High-level diagram of our Malware Classification setups

data. The plot aggregates SHAP values for all the features and all samples in the train data. Then SHAP values are sorted, so the first one shown is the most important feature. In addition to that, each point on the summary plot shows the relationship between SHAP value and a feature, ordered by their importance.

As presented in Figure 4.4 and Figure 4.5, mscoree.dll and Windows Defender/Operational are the most influential features. For Random Forest, an absence of Windows Defender has negative SHAP values, which indicates a decrease in the predicted malware, changing the predicted absolute malware probability on average by six percentage points (0.06 on the x-axis). Conversely, an absence of Windows Defender in XGBoost has positive SHAP values that indicates the sample, in the absence of Windows Defender, has higher malware prediction. The reverse is seen for an imphash_48aa5c8931746a9655524f67-b25a47ef[3], whose higher import hash counts lead to a higher risk of malware.

Whereas, in Figure 4.6, the most influential feature is a comparison between the size of initialized data and the size of code, followed by Windows Defender/Operational and ldrmodiles_LIM. If the condition of the initialized data size being less than the code's size is met, it means higher values resulted

[3]A hash of the imports in a PE file.

in higher benign prediction. In contrast, a low number of undetected samples by Windows Defender lead to higher malware prediction, similar to XGBoost output.

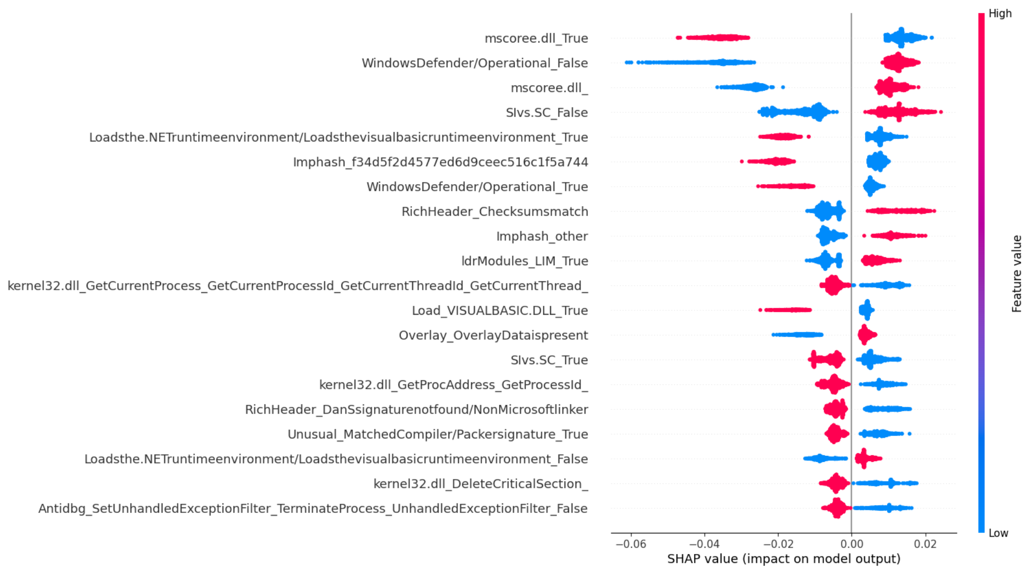


Figure 4.4. SHAP Summary Plot of the top 20 combination features for the Random Forest.

4.1.1 Static features

We conduct data preprocessing on the static features listed in chapter 2 to be input for learning approaches. For example, we inspect whether an Exclusive OR (XOR) is implemented in an application by using XORSearch [110] to detect the string "This program cannot be run in DOS mode" in the DOS portion of the PE header. We could obtain an XOR encoded key as a return value. This XOR feature is translated into a boolean feature to be an input for classifiers. The top 10 significant static features identified by SHAP feature importance for each model are shown in Table 4.1, 4.2, and 4.3. The shared attributes among the three classifiers are highlighted, with five out of ten.

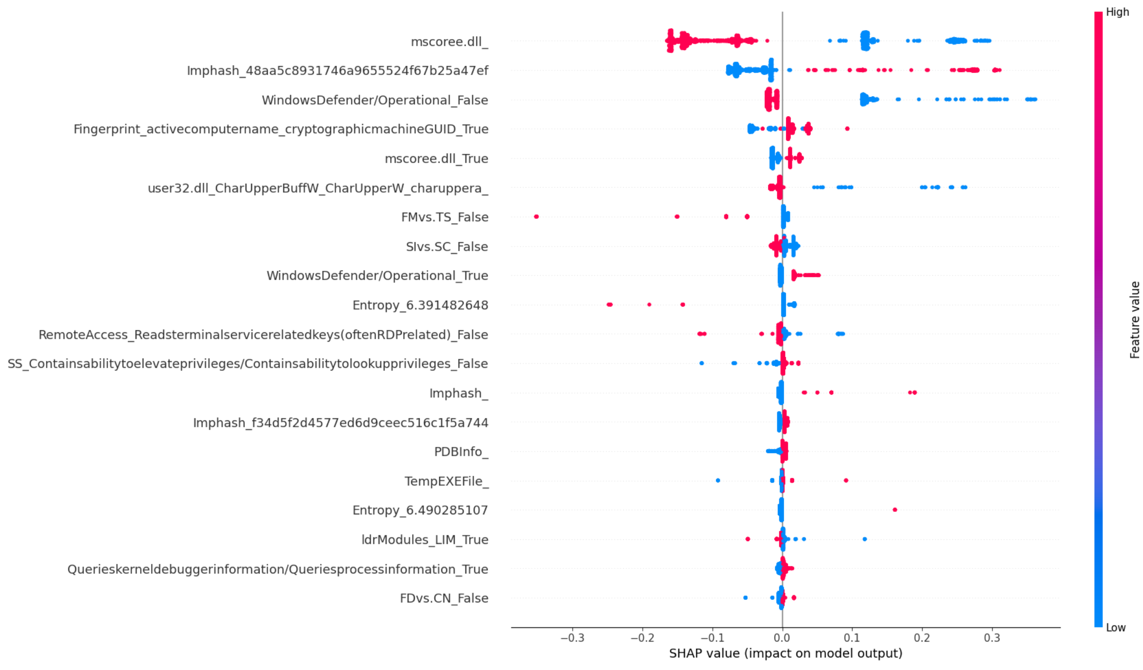


Figure 4.5. SHAP Summary Plot of the top 20 combination features for the XGBoost.

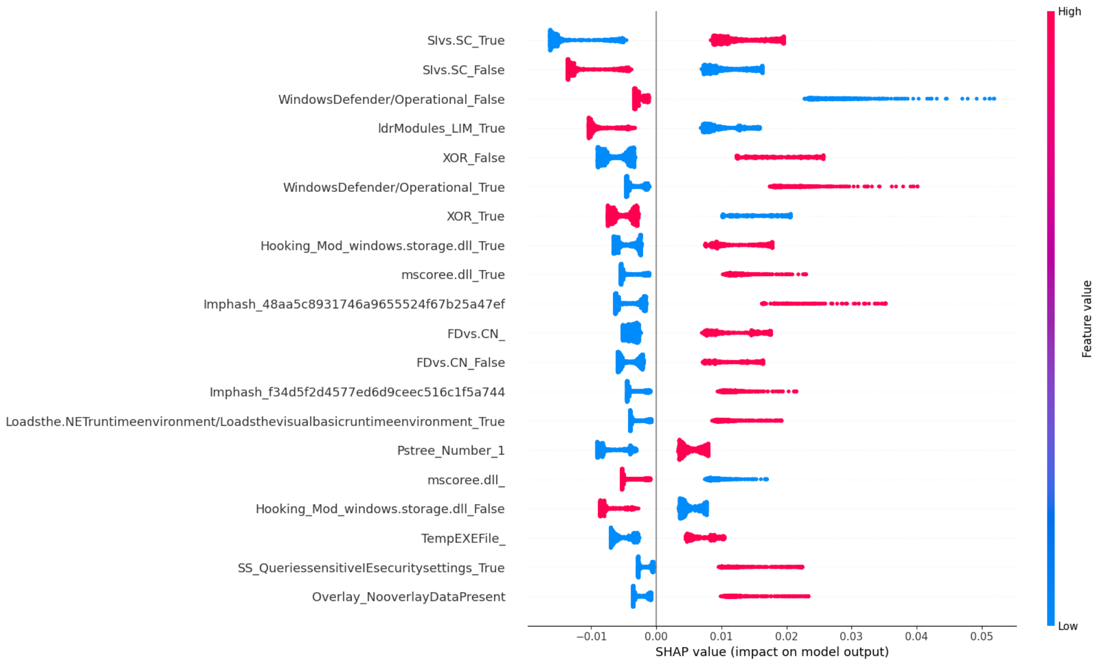


Figure 4.6. SHAP Summary Plot of the top 20 combination features for the Neural Network.

4.1.2 Dynamic features

We use the downloaded four dynamic artifacts described in chapter 3 and combine them with the Falcon Sandbox [47] report for the classifiers using

Table 4.1. Top-10 selected static features from SHAP feature selection for the Random Forest

Static Features	**Description**	**Shared Attributes**
ws2_32.dll/wininet.dll_initcommoncontrolsex	checked a library import for loading common control classes from Comctl32.dll before creating a common control.	XGB,NN
ws2_32.dll/wininet.dll_internetopena	checked a library import for initializing an application's use of the Win32 Internet functions.	XGB,NN
Number of Sections	checked the total number of sections in the data directory more than default sections.	XGB
Imphash	"import hash" was a hash over the imported functions by PE file.	XGB,NN
kernel32.dll_WriteFile_ReplaceFileW_Write-ProcessMemory	checked a library import for writing files or processes in memory, replacing files with write access.	XGB,NN
user32.dll_mapwindow-points	checked a library import for mapping a set of points from one window's coordinate space to another window's coordinate space.	XGB,NN
user32.dll_translate-acceleratorw	checked a library import for a Windows Message key on the keyboard was pressed.	XGB
kernel32.dll_CreateFileA CreateFileW_Create-DirectoryW	checked a library import for creating or opening a file and a new directory.	NN
user32.dll_isdialog-messagew	checked a library import for a message dialog box if it processes the message.	N/A
user32.dll_monitorfrom-point	checked a library import for handling the display monitor containing a specified point.	N/A

the Pandas merge Python module on the filename column. Regarding the features from downloaded files from the guest VM, all loaded resources for a prefetch file, especially in APPDATA_ROAMING, and temporary files in the Temporary File Location (User/System/Program File Folder) of the executables are examined. Using a process list obtained from "`updateProcList`", we could observe any potentially loaded resources/child processes for running applications and include them in memory extraction. As input for training the model, loaded prefetch resources, temporary files, event logs, and registry are converted into boolean attributes, such as LOAD_APPDATA_RESOURCE, Load_VISUALBASIC.DLL, Temp File Executable, Windows Defender, and NT_Persistence.

We extract anti-debugging library functions imported statically and convert

Table 4.2. Top-10 selected static features from SHAP feature selection for the XGBoost

Static Features	**Description**	**Shared Attributes**
ws2_32.dll/wininet.dll_ initcommoncontrolsex	checks a library import for loading common control classes from Comctl32.dll before creating a common control.	RF,NN
kernel32.dll_exitprocess	checks a library import for ending the calling process and all its threads which provides a clean process shutdown. If a process is terminated by calling TerminateProcess, the DLLs attached to that process are not notified of the process termination.	NN
Number of Sections	checks the total number of sections in the data directory more than default sections	RF
user32.dll_translate-acceleratorw	checks a library import for a Windows Message key on the keyboard is pressed.	RF
ws2_32.dll/wininet.dll_ internetopena	checks a library import for initializing an application's use of the Win32 Internet functions.	RF,NN
gdi32.dll	checks whether there are statically imported gdi32.dll's functions containing the Windows GDI (Graphical Device Interface) to draw two-dimensional objects to video displays and printers.	N/A
psapi.dll_enumprocesses_ enumprocessmodules	checks a library import for retrieving the process identifier.	N/A
Imphash	"import hash" is a hash over the imported functions by PE file.	RF,NN
user32.dll_mapwindow-points	checks a library import for mapping a set of points from one window's coordinate space to another window's coordinate space.	RF,NN
kernel32.dll_WriteFile_ ReplaceFileW_Write-ProcessMemory	checks a library import for writing files or processes in memory, replacing files with write access.	RF,NN

them as boolean features. However, we do not employ debugging tools in our virtualized environment so the application could continue its activities without interruption. Meanwhile, we also retrieve an anti-virtualization detection from the HybridAnalysis sandbox site, which reports that one hundred and seventy-five applications enabled anti-virtualization, which may halt the execution process.

In the analysis of Windows Application, System, PowerShell, and Defender/Operational event logs by extracting Defender/Operational events, two event ids (Event ID: 1116-malware detected, Event ID: 1117-malware action taken) are used as the sole inputs for a classifier. These two event logs provide

Table 4.3. Top-10 selected static features from SHAP feature selection for the Neural Network

Static Features	Description	Shared Attributes
ws2_32.dll/wininet.dll_internetopena	checks a library import for initializing an application's use of the Win32 Internet functions.	RF,XGB
kernel32.dll_WriteFile_ReplaceFileW_Write-ProcessMemory	checks a library import for writing files or processes in memory, replacing files with write access.	RF,XGB
kernel32.dll_exitprocess	checks a library import for ending the calling process and all its threads, providing a clean process shutdown. If a process is terminated by calling TerminateProcess, the DLLs attached to that process are not notified of the process termination.	XGB
Imphash	"import hash" is a hash over the imported functions by PE file.	RF,XGB
ws2_32.dll/wininet.dll_initcommoncontrolsex	checks a library import for loading common control classes from Comctl32.dll before creating a common control.	RF,XGB
Overlay	Data is appended to the end of the file. This data could be significant, such as setup packages. Malware applications could load more suspicious code into memory from the overlay once they have appropriate permissions.	N/A
kernel32.dll_CreateFileA CreateFileW_Create-DirectoryW	checks a library import for creating or opening a file and a new directory.	RF
user32.dll_messageboxa	checks a library import for displaying a dialog box that contained a system icon, a set of buttons, and a brief application-specific message, such as status or error information, to contact a user. The message box returns an integer value indicating which button the user clicked.	N/A
kernel32.dll_GetModule-FileNameW_GetModuleHandleW_GetModuleHandleExW_GetModuleHandleExA_getmodulehandlea	checks a library import to retrieve the fully qualified path for the file containing the specified module. The current process must have loaded the module.	N/A
user32.dll_mapwindow-points	checks a library import for mapping a set of points from one window's coordinate space to another window's coordinate space.	RF,XGB

a 70% accurate thread detector, which means that it was a strong feature by itself, but some malware could be undetected. However, using more sophisticated machine learning approaches could improve the detection rate. The Windows Defender detector's output is transformed into a boolean feature to feed into classifiers. Additionally, the Noriben output [51] is used as a feature by representing a list of the one-time occurrence for the process operations ascending by time, such as ReadFile, QueryDirectory, and TCP Connect. The top 10 significant dynamic features as selected by SHAP for each model are shown in Table 4.4, 4.5, and 4.6. Among these tables, only Cryptographic Related is shared across all three classifiers. Overall, about two-fifths of the selected dynamic features are novel, whereas the remaining parts are used before in prior works [44, 73].

4.2 Model architectures

In the following sections, we describe the implementation of three machine learning algorithms used in our binary classification: Random Forest, XGBoost, and neural networks. The list of tuning ranges across all tables is used to search values for each parameter when performing hyperparameter search space. As detailed in all tables, we start with the initial parameter values for all learning approaches.

4.2.1 Random Forest

The Random Forest model is implemented using Scikit-Learn [54]. To obtain the best model, the model's hyperparameters are tuned through grid search using 10-fold cross-validation. Micro-F1 is used as the objective to maximize the grid search. This scoring will calculate each label's metrics and find the unweighted mean. The hyperparameter search spaces are run through the entire set and SHAP selected features. The eight hyper-parameters subject to the grid search, their respective ranges of possible values, and selected values attaining the highest accuracy performance are presented in Table 4.7 and Table 4.8, respectively.

4.2.2 XGBoost

The XGBoost model is implemented using the XGBClassifier from Scikit-Learn [54]. We use Ray Tune [111] to perform a grid search by setting re-

Table 4.4. Top-10 selected dynamic features from SHAP feature selection for the Random Forest

Dynamic Features	**Location**	**Description**	**Shared Attributes**
Hook type Inline	Memory Dump	could be called inline patching. The target API function's first few bytes (instructions) are modified (patched) with a jump statement to redirect the API to the malicious code. So, the malicious code could intercept the input parameters, filter output, and turn the control back to the original function.	N/A
Contains the ability to query machine time/timezone/version	Falcon Sandbox site	checks whether an application could query for timezone information and version.	XGB
Hooking_Mod_windows. storage.dll	Memory Dump	detects API hook associated with a set of procedures and driver functions to ensure that Windows programs operate properly.	XGB
Priv_SeAuditPrivilege	Memory Dump	the privilege that causes the system to grant all read access control to any files, regardless of the access control list (ACL) specified for the file. It generates audit-log entries, allowing users to add them to the security log.	N/A
Hook type EAT	Memory Dump	hooks functions in the target DLL by modifying entries in the PE File Exports section (EAT). It works for libraries that are delay loaded. Subsequent calls to LoadLibrary on the target DLL would return the reference to the already loaded library that had its EAT entry for the target process patched.	N/A
Executes WMI queries/Contains references to WMI/WMIC	Falcon Sandbox site	the Windows Management Instrumentation (WMI), which could query the repository to retrieve class, instance, or schema data.	N/A
childPPID	Falcon Sandbox site and Memory Dump	checks whether an application spawns child processes in memory and what activities they perform.	N/A
80/443	Falcon Sandbox site	checks whether an application establishes a connection via port 80/443.	N/A
Cryptographic Related	Falcon Sandbox site	checks whether an application calls for crypto functions during runtime.	XGB,NN
Disables Command Prompt (cmd)/Windows Registry Editor (regedit)	Falcon Sandbox site	checks if a function to disable running the command prompt is presented.	XGB

source per trial to "gpu' to determine an effective set of values for the essential hyper-parameters. We also use 'logloss' and 'auc' as the evaluation metrics

Table 4.5. Top-10 selected dynamic features from SHAP feature selection for the XGBoost

Dynamic Features	**Location**	**Description**	**Shared Attributes**
psxview_all	Memory Dump	checks memory-mapped files by enumerating processes from seven data sources and cross-referenced them to observe malicious discrepancies. We also verify the process with "pslist" result, whether it is zero threads, zero handles, or a non-empty exit time to set as a boolean feature. However, three DLLs listed in process memory and the mapped path in VAD cross-referencing would also be considered to uncover hidden processes.	N/A
Disables Command Prompt (cmd)/Windows Registry Editor (regedit)	Falcon Sandbox site	checks if a function to disable running the command prompt is presented.	RF
Cryptographic Related	Falcon Sandbox site	checks whether an application calls for crypto functions during runtime.	RF,NN
Contains ability to query machine time/timezone/version	Falcon Sandbox site	checks whether the application could query for timezone information and version.	RF
Hooking_Mod_imm32	Memory Dump	detects API hooks associated with Windows Input Method Manager (IMM).	N/A
Hooking_user32.dll!get-activewindow	Memory Dump	detects API hook associated with handling the active window attached to the calling thread's message queue.	N/A
Priv_SeSystemProfile-Privilege	Memory Dump	the privilege that used Windows performance monitoring tools to monitor or profile the system performance.	N/A
Hooking_Mod_msimg32	Memory Dump	detects API hook associated with Graphical Device Interface (GDI), which is used to draw lines, boxes, and text of the user interface.	N/A
Unusual ports for process_and_protocol	Falcon Sandbox site	checks whether an application uses unusual ports or protocols to create a connection.	N/A
Hooking_Mod_windows. storage.dll	Memory Dump	detects API hook associated with a set of procedures and driver functions to ensure that Windows programs operate properly.	RF

to measure the classifier's performance. The 'logloss' metric is the default evaluation metric used with the objective 'binary:logistic'. The five hyper-parameters, their ranges of possible values, and the selected best-performing values are presented in Table 4.9 and Table 4.10, respectively.

A single tree each from the XGBoost ensembles on static features, dy-

Table 4.6. Top-10 selected dynamic features from SHAP feature selection for the Neural Network

Dynamic Features	**Location**	**Description**	**Shared Attributes**
Hooking_user32.dll! impersonateddeclient-window	Memory Dump	detects API hook associated with enabling a Dynamic Data Exchange (DDE) to impersonate a DDE client application's security context.	N/A
Drop files/executable files	Falcon Sandbox site	checks whether temporary writable/hidden files are extracted from binary files.	N/A
Cryptographic Related	Falcon Sandbox site	checks whether an application calls for crypto functions during runtime.	RF,XGB
POSTs/GETs files to a webserver	Falcon Sandbox site	checks whether an application sends a request to the server via the HTTP GET/POST method.	N/A
Hooking_Mod_shell32	Memory Dump	detects API hook associated with Windows Shell API functions, which are used when opening web pages and files or inserting code injection.	N/A
Priv_SeIncrease-WorkingSetPrivilege	Memory Dump	the privilege that allows to increase a process working set.	N/A
SS_Queries sensitive IE security settings	Falcon Sandbox site	checks whether an application queries the security level setting information for Internet access to run scripts or ActiveX controls.	N/A
SS_Opens the Kernel Security Device Driver (KsecDD) of Windows	Falcon Sandbox site	checks whether an application opens the Kernel-Mode Security Support Provider Driver Interface (ksecdd.sys).	N/A
Hooking_WS2_32_func	Memory Dump	detects API hook associated with handling network connections.	N/A
Hook_usermode	Memory Dump	detects whether the hooking occurs in user-land; it is the inline hook, which involves rewriting the target function to redirect control flow to a custom handler. Inside the handler, the parameters are preserved, and the handler could decide whether to execute or analyze the requested function.	N/A

namic features, and their combination are shown in Figure 4.7, 4.8, and 4.9, respectively. One could extract rules by tracing the path from the root of a tree to some leaf. For the 'static features' in Figure 4.7, one such path is $mscoree.dll\_ == N/A, FMvs.TS\_ == False$, which increased the malware prediction total by 0.00193, a sum of probability value of reaching leaf nodes given a specific branch of the tree. We could conclude that $WindowsDefender$ was a powerful feature. As shown in the summary plot, SHAP analysis has also been identified as a highly influential factor for improving performance. For the 'dynamic features' in Figure 4.8, only one attribute $WindowsDefender ==$

Table 4.7. Hyperparameter tuning for the Random Forest Classifier on the full feature set

Name	Initial Parameters	Tuning Ranges	Static	Dynamic	Combination
n_estimators	200	range(100,650,50)	300	250	600
criterion	gini	['gini', 'entropy']	entropy	gini	entropy
max_depth	15	[6,8,10,12,15,20,25,30]	25	15	20
max_features	sqrt	['sqrt', 'log2']	sqrt	sqrt	sqrt
bootstrap	True	[True, False]	True	True	True
min_samples_leaf	2	range(2,12,2)	2	2	2
min_samples_split	2	range(2,12,2)	4	2	6
max_leaf_nodes	30	range(15,55,5)	30	45	50

Table 4.8. Hyperparameter tuning for Random Forest Classifier on SHAP selected features

Name	Initial Parameters	Tuning Ranges	Static	Dynamic	Combination
n_estimators	200	range(100,650,50)	100	100	150
criterion	gini	['gini', 'entropy']	entropy	entropy	gini
max_depth	15	[6,8,10,12,15,20,25,30]	25	8	30
max_features	sqrt	['sqrt', 'log2']	sqrt	sqrt	sqrt
bootstrap	True	[True, False]	True	True	True
min_samples_leaf	2	range(2,12,2)	2	2	2
min_samples_split	2	range(2,12,2)	8	4	2
max_leaf_nodes	30	range(15,55,5)	50	40	40

$False$ from the root could improve the malware prediction total by 0.00199. For a combination of static and dynamic, one of the extracted rules was $imphash <> f34d5..., imphash <> 48aa5..., WindowsDefender == False$, which yielded the malware prediction total by 0.00197.

Table 4.9. Hyperparameter tuning for XGBoost Classifier on initial features

Name	Initial Parameters	Tuning Ranges	Static	Dynamic	Combination
n_estimators	200	range(100,650,50)	350	450	250
max_depth	6	range(4,20,2)	6	6	4
learning_rate/eta	0.001	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.002	0.002	0.01
gamma	1.5	[0.8,1.0,1.5,2.0, 2.5,3.0,4.0,5.0]	1.5	2.5	5.0
min_child_weight	1	range(1,6,2)	1	3	1

Table 4.10. Hyperparameter tuning for XGBoost Classifier on SHAP selected features

Name	Initial Parameters	Tuning Ranges	Static	Dynamic	Combination
n_estimators	200	range(100,650,50)	400	500	550
max_depth	6	range(4,20,2)	8	6	8
learning_rate/eta	0.001	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.002	0.01	0.002
gamma	1.5	[0.8,1.0,1.5,2.0, 2.5,3.0,4.0,5.0]	2.5	3.0	0.8
min_child_weight	1	range(1,6,2)	1	1	1

4.2.3 Neural Network

The neural network learning model is implemented by using Keras [112] and TensorFlow [53]. In the beginning, models are trained with a batch size of 10 samples on 200 episodes. The network architecture follows a standard setup using nonlinear activations and employsBatchNormalization and regularizers to avoid overfitting [113]. A dropout layer with a dropout rate of 0.2 is inserted after the first dense layer, followed by a BatchNormalization layer. Then, seven/eight hidden layers are stacked with a BatchNormalization layer before the final sigmoid layer to produce the output. The model uses RMSprop optimizer with learning_rate 0.002, calculated loss based on binary cross-entropy, and computed accuracy and f1_score for evaluation. In order to obtain a good learning model, all hyperparameters are tuned by utilizing Keras Tuner [56] to perform RandomSearch. The learning rate is reduced when accuracy has stopped improving for 5 epochs (patience) with the maximum total number of 20 test trials. Table 4.11 shows the model configuration for each layer. In addition, the summary of the hyperparameters optimization search space for a full dataset with and without SHAP feature selection from the RandomSearch is provided in Table 4.12 and Table 4.13, respectively.

Table 4.11. Neural Network Layers

Layers	**Layer Details**
Dense Layer	units: *256*, activation Function: *swish*, use_bias: *True*, kernel_initializer: *he_uniform*, bias_initializer: *ones*, bias_regularizer: *l2(0.002)*, kernel_regularizer: *l2(0.01)*, kernel_constraint: *maxnorm(2.)*, activity_regularizer: *l2(0.0001)*
Dropout Layer	Dropout(rate=0.2, seed=5000)
Batch Layer	BatchNormalization(trainable=True)
Hidden 7 or 8 Dense Layers	units: *256*, activation function: *swish*, use_bias: *True*, kernel_initializer: *he_uniform*, bias_regularizer: *l2(0.02)*, kernel_regularizer: *l2(0.002)*, kernel_constraint: *maxnorm(1.5)*, activity_regularizer: *l2(0.001)*
Batch Layer	BatchNormalization(trainable=True)
Dense Layer	units: *1*, activation function: *sigmoid*, kernel_initializer: *he_uniform*

Table 4.12. Hyperparameter tuning for Neural Network on initial features

Name	**Initial Parameters**	**Tuning Ranges**	**Static**	**Dynamic**	**Combination**
batch_size	10	range(10,50,5)	20	30	25
epochs	200	range(100,650,50)	600	400	600
optimizer	RMSprop	[‘Adam’,‘RMSprop’, ‘Adadelta’,‘Adagrad’]	Adagrad	Adam	RMSprop
learning_rate	0.002	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.01	0.01	0.0002
initial neuron	256	range(128,1024,32)	432	272	576
hidden layers	4	range(2,10)	7	6	7
hidden neurons	256	range(16,512,32)	[176,464,176, 48,496,400, 256]	[80,112,496, 368,208,496]	[288,256,480, 192,256,448, 288]
bias_ regularizer	[0.002,0.02]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0002,0.02]	[0.01,0.002]	[0.0001,0.02]
kernel_ regularizer	[0.01,0.002]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.02,0.001]	[0.0001,0.001]	[0.0001,0.0002]
kernel_ constraint	[2.0,1.5]	range(1,3,0.5)	[1.0,3.0]	[2.0,2.0]	[2.5,1.5]
activity_ regularizer	[0.0001,0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.01,0.001]	[0.02,0.01]	[0.002,0.0002]

4.3 Experimental Results

In this section, we present the three algorithms' classification results on the three datasets: static features, dynamic features, and their combination. The predictive performance is measured using four different metrics, including accu-

Table 4.13. Hyperparameter tuning for Neural Network on SHAP selected features

Name	**Initial Parameters**	**Tuning Ranges**	**Static**	**Dynamic**	**Combination**
batch_size	10	range(10,50,5)	25	20	10
epochs	200	range(100,650,50)	350	600	100
optimizer	RMSprop	[‘Adam’,‘RMSprop’, ‘Adadelta’,‘Adagrad’]	RMSprop	Adagrad	RMSprop
learning_rate	0.002	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.002	0.01	0.0002
initial neuron	256	range(128,1024,32)	240	432	480
hidden layers	4	range(2,10)	8	7	9
hidden neurons	256	range(16,512,32)	[240,464,208, 304,176,272, 112,256]	[176,464,176, 48,496,400, 256]	[320,160,256, 256,256,256, 256,256,256]
bias_ regularizer	[0.002,0.02]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0001,0.001]	[0.0002,0.02]	[0.02,0.001]
kernel_ regularizer	[0.01,0.002]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0002,0.001]	[0.02,0.001]	[0.0002,0.002]
kernel_ constraint	[2.0,1.5]	range(1,3,0.5)	[1.0,2.5]	[1.0,3.0]	[3.0,2.0]
activity_ regularizer	[0.0001,0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.002,0.0002]	[0.01,0.001]	[0.02,0.0002]

racy, f1_score, similarly to previous works [71,83], mean absolute error (MAE), and AUC.

Table 4.14 lists results for running all algorithms with their default parameter settings and the full datasets. Overall, XGBoost performed the best by achieving an f1_score of 86% for a combination of static and dynamic attributes. However, the accuracy of running the model on only separate features with the neural network outperformed XGBoost.

Table 4.15 shows the improvements achieved by adequately tuning the potential hyperparameters. The overall performance increased for almost all of the nine method and feature set combinations, compared to the default performances from Table 4.14. Compared to the other two algorithms, the neural network enjoyed a dramatic increase in performance and achieved the best accuracy and f1_score. During parameter tuning, using cross-validation helped avoid overfitting, compared to the default setting. The best results for every dataset, using the combined dataset of static and dynamic trained by the neural network, could provide both accuracy and f1_score at 97.3%. The figure represented the correct class predicted by integrating static and dynamic features, but the absolute error was 0.04. So a low MAE implied higher average confidence.

After feature subset selection was applied to the initial set, the results obtained by the default models were illustrated in Table 4.16. Again, the neural network performed the best for the overall three sets of features, even though there was a decrease in the performance of all algorithms, compared to the "full" dataset results from Table 4.15. It implied that robust classifiers could extract small amounts of information from low-quality features when using feature selection.

In addition to the default model configuration, the results for applying both feature selection and hyperparameter tuning were shown in Table 4.17. Although this combination improved results over the simple default parameter setting, the overall best results were still achieved without feature selection. Nevertheless, there was a slight difference in predictive results with and without feature selection by conducting hyperparameter optimization. SHAP feature selection has been proven to be very effective to remove redundant, or irrelevant features, which also helped to improve the effectiveness of various machine learning algorithms.

In addition to the evaluation metrics reported, we also compared the total elapsed time on selected features, both with and without hyperparameter tuning for each estimator, as shown in 4.18. We used stratified 10-Folds cross-validation. XGBoost and the neural network were trained using the GPU. We observed that the RandomSearch of the neural network took less time than GridSearch used for the tree-based approaches. Although a more significant number of parameters needed to be optimized for the neural network, not all combinations were considered by RandomSearch. Conversely, GridSearch exhaustively enumerates all possible combinations. Therefore, Random Forest training was the slowest and the most exhaustive. XGBoost laid in between, still exhaustive but benefiting from a GPU. Regarding memory usage, on average, tuning XGBoost performed worst, utilizing memory up to 9GB of main memory for a relatively small dataset.

We also analyzed the detailed results of the classification report and confusion matrix generated by the hyperparameter tuning on initial and selected features; these results are shown in Table 4.19 and Table 4.23, respectively. Figure 4.11 and Figure 4.12 are the confusion matrices for the globally best-performing setup. The precision, recall, and F1 score were very high, with a false positive rate lower than 1.5%. When using only static features, the false positive rate was slightly higher than the other two feature sets, but less than 2%. Regarding the false negative rate, using the static features alone only achieves 9.75%, whereas dynamic or combination features yielded smaller values of 2.11% and 1.99%, respectively. Nevertheless, the SHAP selection applied to combination attributes in Figure 4.14 yielded the smallest value for a false negative rate, compared to others.

Finally, we generated ROC curves for all three classifiers before and after feature selection, applied to the dataset as in Figures 4.10 and 4.11. Apparently, a gap between the neural network and the tree-based methods could be noticed. While feature selection slightly decreased the AUC value for the neural network, XGBoost and Random Forest performance also dropped.

4.4 Discussion

It was possible to obtain highly accurate machine learning models by integrating static and dynamic analysis artifacts which consistently yielded a higher F1 score for every model than training them separately. The best models achieved

Table 4.14. Model Evaluation using default hyperparameter settings.

Features	Algorithm	Avg. MAE	Avg. ACC	Avg. F1	Avg. AUC
Static	RF	0.193±0.02	0.807±0.02	0.804±0.02	0.785±0.02
	XGB	**0.160±0.01**	0.840±0.01	0.842±0.01	0.852±0.01
	NN	0.180±0.03	**0.849±0.03**	**0.849±0.03**	**0.900±0.03**
Dynamic	RF	0.182±0.02	0.818±0.02	0.816±0.02	0.800±0.03
	XGB	0.171±0.03	0.829±0.03	0.831±0.03	0.833±0.03
	NN	**0.160±0.02**	**0.842±0.02**	**0.842±0.02**	**0.882±0.02**
Combination	RF	0.165±0.04	0.835±0.04	0.835±0.04	0.823±0.04
	XGB	**0.140±0.02**	**0.860±0.02**	**0.862±0.02**	0.874±0.02
	NN	0.158±0.02	0.851±0.03	0.851±0.03	**0.916±0.02**

Table 4.15. Model Evaluation with hyperparameter tuning.

Features	Algorithm	Avg. MAE	Avg. ACC	Avg. F1	Avg. AUC
Static	RF	0.189±0.03	0.811±0.03	0.809±0.03	0.791±0.03
	XGB	**0.158±0.01**	0.842±0.01	0.844±0.01	0.853±0.01
	NN	0.184±0.05	**0.888±0.04**	**0.888±0.04**	**0.954±0.03**
Dynamic	RF	0.173±0.03	0.827±0.03	0.826±0.03	0.814±0.03
	XGB	0.166±0.02	0.834±0.02	0.836±0.02	0.838±0.02
	NN	**0.050±0.04**	**0.968±0.04**	**0.968±0.04**	**0.988±0.03**
Combination	RF	0.150±0.03	0.850±0.03	0.851±0.03	0.846±0.03
	XGB	0.139±0.01	0.861±0.01	0.863±0.01	0.880±0.01
	NN	**0.041±0.05**	**0.973±0.04**	**0.973±0.04**	**0.988±0.02**

Table 4.16. Model Evaluation for SHapley Additive exPlanations (SHAP) on three types of features.

Features	Algorithm	Avg. MAE	Avg. ACC	Avg. F1	Avg. AUC
Static	RF	0.199±0.03	0.801±0.03	0.798±0.03	0.777±0.03
	XGB	**0.163±0.02**	0.837±0.02	**0.840±0.02**	0.856±0.02
	NN	0.222±0.02	**0.839±0.02**	0.839±0.02	**0.887±0.03**
Dynamic	RF	**0.174±0.02**	0.826±0.02	0.826±0.02	0.818±0.03
	XGB	0.179±0.02	0.821±0.02	0.822±0.02	0.821±0.02
	NN	0.198±0.02	**0.828±0.02**	**0.828±0.02**	**0.886±0.02**
Combination	RF	0.153±0.03	0.847±0.03	0.847±0.03	0.842±0.04
	XGB	**0.137±0.02**	**0.863±0.02**	**0.865±0.02**	0.877±0.02
	NN	0.174±0.02	0.836±0.03	0.836±0.03	**0.889±0.03**

an F1 score of up to 97.3%. However, one significant disadvantage was the labor-intensive nature of data acquisition on the dynamic features compared to static features. Another issue with the dynamic analysis was that some applications terminate prematurely or were unlinked in a doubly-linked process

Table 4.17. Model Evaluation for hyperparameter tuning and SHapley Additive exPlanations (SHAP) on three types of features.

Features	Algorithm	Avg. MAE	Avg. ACC	Avg. F1	Avg. AUC
Static	RF	0.187±0.03	0.813±0.03	0.813±0.03	0.803±0.04
	XGB	**0.155±0.02**	0.845±0.02	0.847±0.02	0.861±0.02
	NN	0.209±0.02	**0.873±0.03**	**0.873±0.03**	**0.924±0.03**
Dynamic	RF	0.168±0.02	0.832±0.02	0.833±0.02	0.828±0.03
	XGB	0.171±0.01	0.829±0.01	0.830±0.01	0.827±0.02
	NN	**0.052±0.04**	**0.965±0.04**	**0.965±0.04**	**0.986±0.03**
Combination	RF	0.153±0.03	0.847±0.03	0.848±0.03	0.843±0.04
	XGB	0.136±0.02	0.864±0.02	0.866±0.02	0.872±0.02
	NN	**0.036±0.05**	**0.972±0.05**	**0.972±0.05**	**0.984±0.03**

Table 4.18. Elapsed time used for training for each algorithm on selected attributes

Algorithm	Static Training	Static Tuning	Dynamic Training	Dynamic Tuning	Combination Training	Combination Tuning
RF	7s	9h:46m:33s	5s	9h:48m:08s	7s	9h:56m:31s
XGB	11s	5h:52m:48s	16s	5h:57m:20s	22s	6h:38m:03s
NN	16m:20s	3h:56m:45s	18m:54s	4h:12m:59s	39m:22s	3h:22m:51s

list before a memory acquisition was completed. This case could be caused by ransomware samples encrypting and automatically restarting the machine. Nonetheless, it still left child processes instantiated from the main application process, which could be correlated with significant features retrieved from prefetch files, event log files, and temporary (temp) files to obtain meaningful attributes. When applying the algorithm to these features obtained by only static and dynamic analysis, the detection accuracy was 88% and 96.8%, respectively. Thus, the achieved results showed that training separately on the given dynamic results increased the accuracy, similar to a combination of static and dynamic analysis. Conversely, only combination attributes outperformed other feature sets in the experimental result with SHAP feature selection.

Table 4.19. Performance for Neural Network for three types of initial features with hyperparameter tuning

	Static Features			Dynamic Features			Combination Features		
	Precision	Recall	f1_score	Precision	Recall	f1_score	Precision	Recall	f1_score
Malicious	0.98	0.84	0.90	0.98	0.97	0.97	0.99	0.97	0.98
Benign	0.79	0.97	0.87	0.95	0.97	0.96	0.95	0.98	0.96

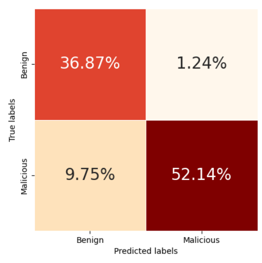


Table 4.20. Static Confusion Matrix for NN.

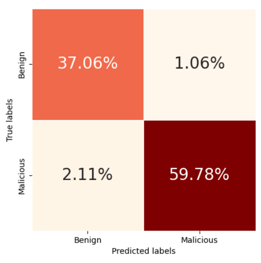


Table 4.21. Dynamic Confusion Matrix for NN.

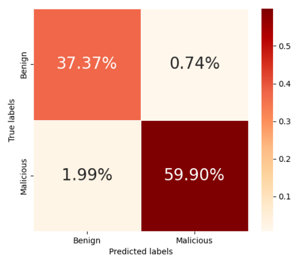


Table 4.22. Combination Confusion Matrix for NN.

Table 4.23. Performance for Neural Network for three types of features with hyperparameter tuning and feature selection

	Static Features			Dynamic Features			Combination Features		
	Precision	Recall	f1_score	Precision	Recall	f1_score	Precision	Recall	f1_score
Malicious	0.92	0.87	0.89	0.98	0.96	0.97	0.98	0.97	0.98
Benign	0.80	0.88	0.84	0.94	0.96	0.95	0.96	0.97	0.96

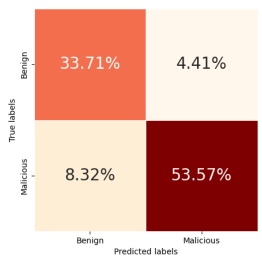


Table 4.24. Static Confusion Matrix for NN.

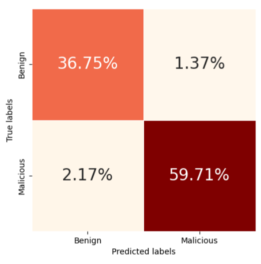


Table 4.25. Dynamic Confusion Matrix for NN.

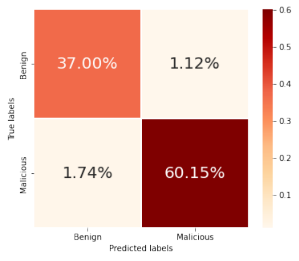


Table 4.26. Combination Confusion Matrix for NN.

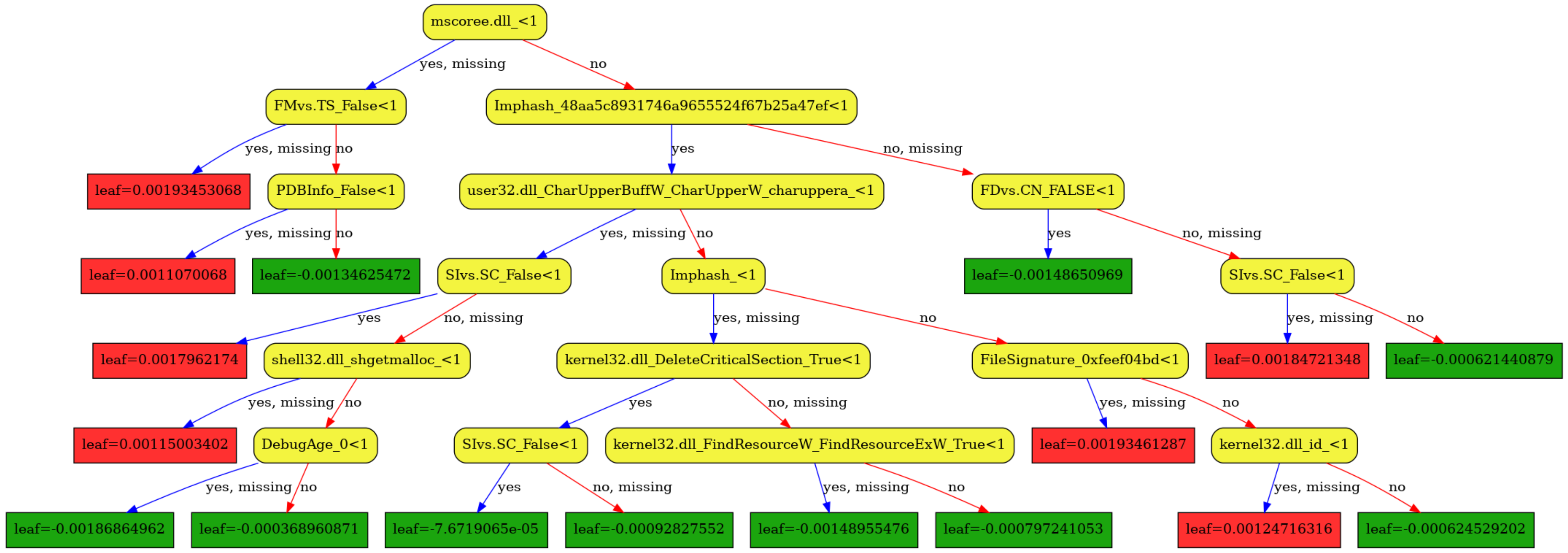


Figure 4.7. One example tree out of an 200 trees XGBoost ensemble on the initial static features. Red leaves indicate Malware.

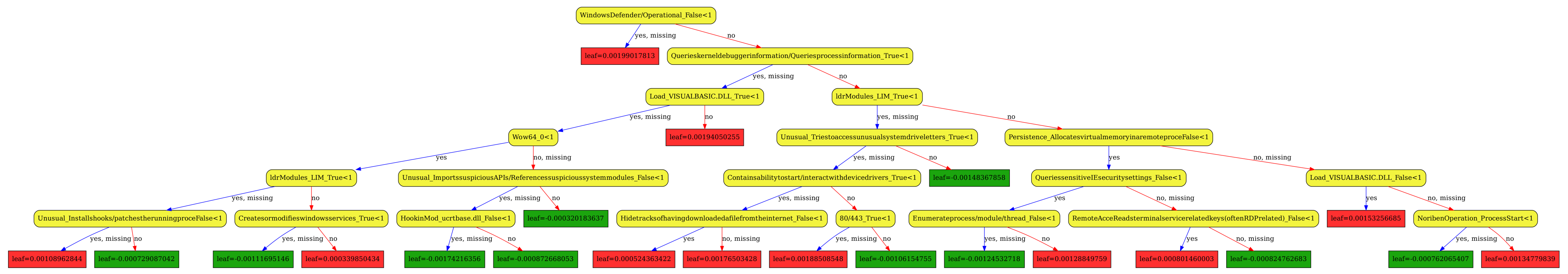

WindowsDefender/Operational_False<1
yes, missing
no
leaf=0.00199017813
Querieskerneldebuggerinformation/Queriesprocessinformation_True<1
yes, missing
no
Load_VISUALBASIC.DLL_True<1
ldrModules_LIM_True<1
yes, missing
no
Wow64_0<1
leaf=0.00194050255
Unusual_Triestoaccessunusualsystemdriveletters_True<1
Persistence_AllocatesvirtualmemoryinaremoteproceFalse<1
yes
no, missing
ldrModules_LIM_True<1
Unusual_ImportssuspiciousAPIs/Referencessuspicioussystemmodules_False<1
Containsabilitytostart/interactwithdevicedrivers_True<1
leaf=-0.00148367858
QueriessensitiveIEsecuritysettings_False<1
Load_VISUALBASIC.DLL_False<1
Unusual_Installshooks/patchestherunningproceFalse<1
Createsormodifieswindowsservices_True<1
HookinMod_ucrtbase.dll_False<1
leaf=-0.000320183637
Hidetracksofhavingdownloadedafilefromtheinternet_False<1
80/443_True<1
Enumerateprocess/module/thread_False<1
RemoteAcceReadsterminalservicerelatedkeys(oftenRDPrelated)_False<1
leaf=0.00153256685
NoribenOperation_ProcessStart<1
leaf=0.00108962844
leaf=-0.000729087042
leaf=-0.00111695146
leaf=0.000339850434
leaf=-0.00174216356
leaf=-0.000872668053
leaf=0.000524363422
leaf=0.00176503428
leaf=0.00188508548
leaf=-0.00106154755
leaf=-0.00124532718
leaf=0.00128849759
leaf=0.000801460003
leaf=-0.000824762683
leaf=-0.000762065407
leaf=0.00134779839

Figure 4.8. One example tree out of an 200 trees XGBoost ensemble on the initial dynamic features. Red leaves indicate Malware.

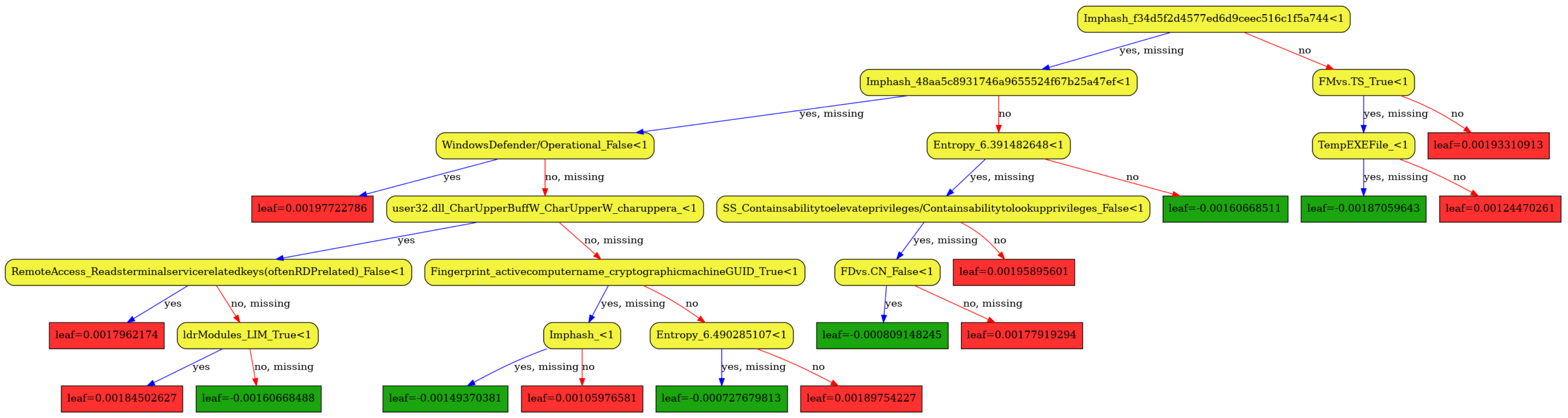


Figure 4.9. One example tree out of an 200 trees XGBoost ensemble on the initial combination features. Red leaves indicate Malware.

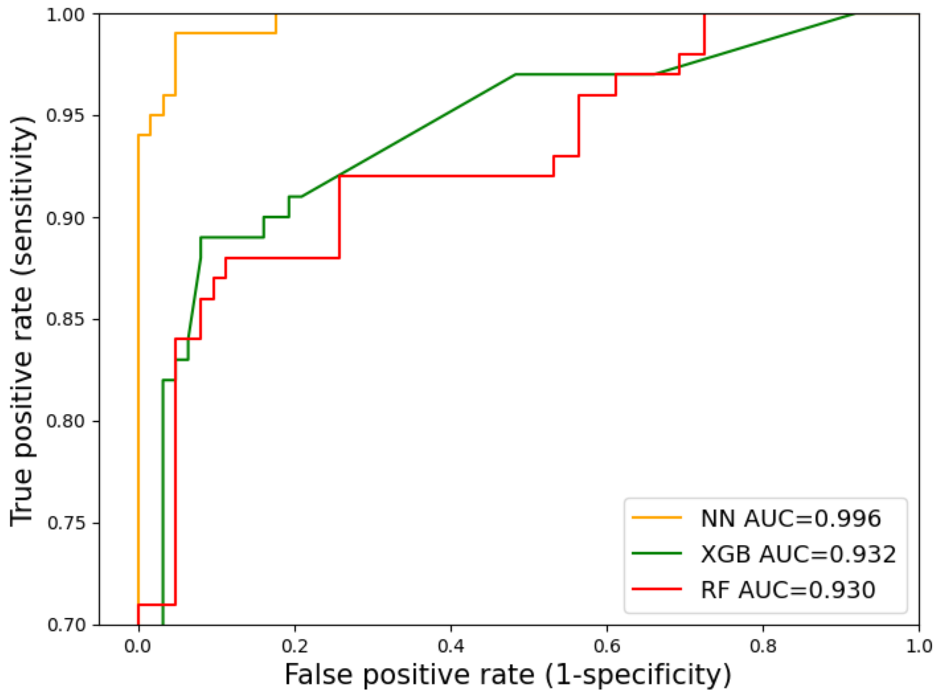


Figure 4.10. ROC Curve for Neural Network Initial Combination Features.

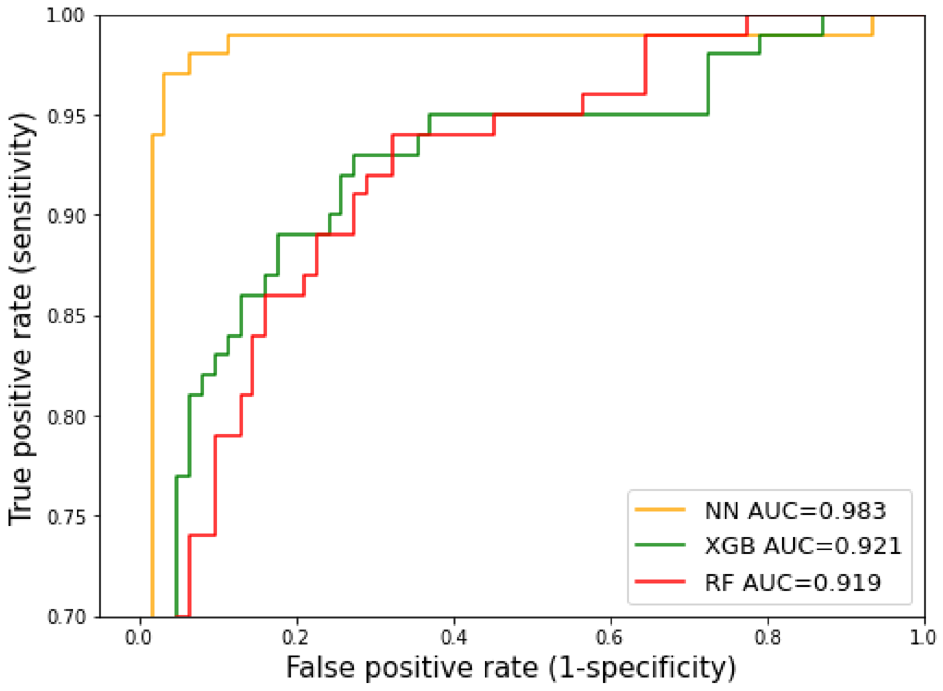


Figure 4.11. ROC Curve for Neural Network Combination Selected Features.

Chapter 5

A comparison of Neural Network Architectures for Malware Classification based on Call Sequences

This chapter explores utilizing dynamic call sequences captured by the open-source Noriben tool [51], which employs dynamic analysis in a virtualized environment. All running processes are recorded in Noriben log files. However, we focus only on our application process, which writes the output on the log file over time. The first 10000 operations are each extracts for a total of 2000 benign and malware samples. A few types of application files are analyzed: MSI files (an installer package file format), EXE files (an executable file format), AppxBundle files (a Windows 8.1 App Bundle Package format), and DOC/DOCX files. Seven malware families are classified: ransomware, trojan, backdoor, rootkit, virus, miner, and other. We use eight learning approaches: Random Forest, XGBoost, fully connected neural networks, GRU, LSTM, Transformers, and two combination approaches.

5.1 Data Collection and Preprocessing

This section describes datasets collected with our custom Python script and a public dataset [114] for predicting eight categories, which consist of Benign (756), Ransom (139), Trojan (980), Miner (43), Rootkit (10), Back-door (23), Virus (11), and Other (38). Noriben sequences and API call sequences [114] are retrieved via dynamic analysis from a virtualized environment.

5.1.1 Noriben sequences for Multi-class Classification

We collect 2000 samples for malware and benign applications from publicly available sources. Precisely, 1244 malware samples are collected from both VirusShare and VirusSign [1], and 756 goodware samples are collected from FileHorse [2]. The labels are obtained from two primary sources: uploading the files to VirusTotal [106] by using at least five or more anti-virus engines to retrieve a class label and obtaining threat names from the Windows Defender/Operational event log. In case an application is not in the seven categories mentioned before, it would be assigned to the "`other`" class. We set up a secure virtualized environment described in chapter 2 to collect dynamic behaviour. The Windows Defender Real-Time Protection (RTP) is disabled to allow the malware to run smoothly over the virtual machine. To prevent malware from actually infecting the Internet, we use host-only networking between the host and the guest.

Any missing values after the data preprocessing are replaced with a constant default value using Scikit-Learn's *SimpleImputer* [54]. Figure 5.1 shows the percentages of missing values per column. The extracted operation sequences are treated like a string of words. Therefore, they have the same method for data preprocessing, utilizing Tensorflow's [53] TextVectorization with a dictionary size of 500 and producing an output in "`count`" mode. Table 5.2 provides some statistics for the top-10 most frequently occurring Noriben

[1]Downloads from VirusShare [90] and VirusSign [107]

[2]Downloads from FileHorse [108]

operations. The first number is the average number of times an application in a family is calling an operation. Only applications that called the operation at least once are considered for the average. The second number is the total number of applications in that family that use this operation at least once. Regarding the applications, less frequent families are more uniform, with the

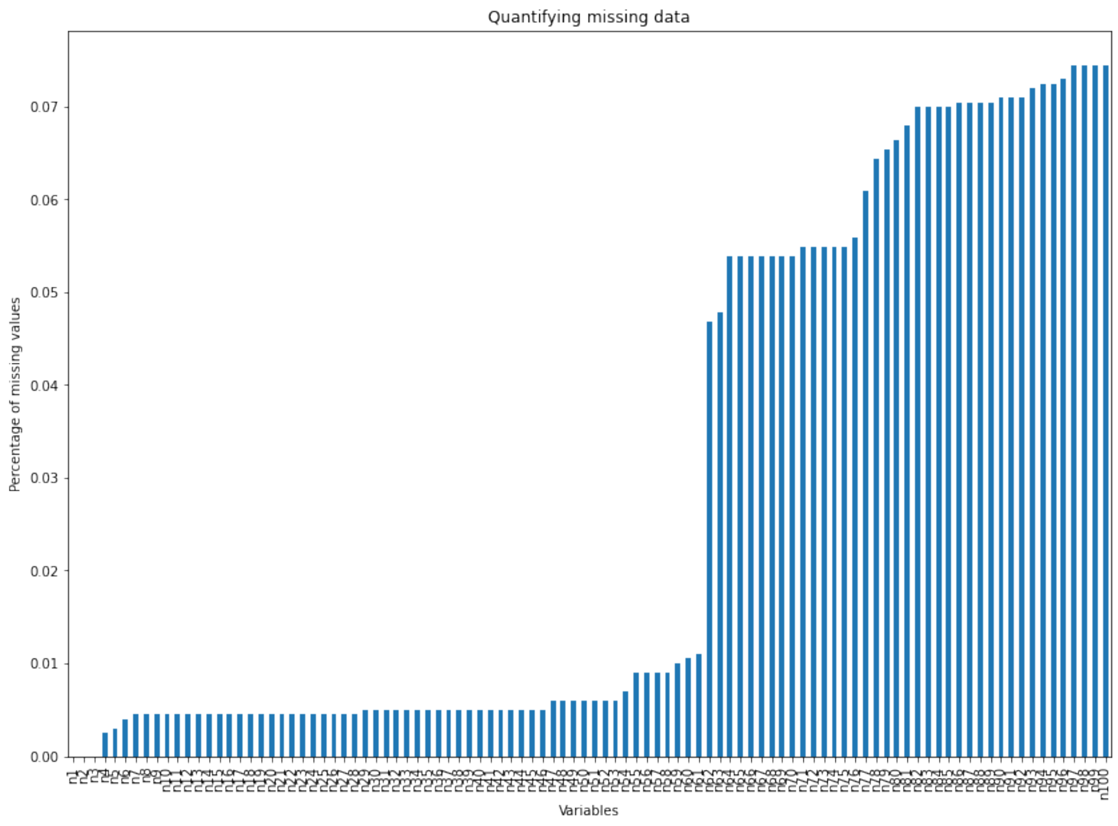


Figure 5.1. The bar plot displays the percentage of missing data continuously increasing by sequence length from 1 to 100.

most extreme value for $Virus$, where 11 samples use almost all of the top 10 operations. However, there is one operation,"`RegQuery KeySecurity`", that is never called. The $Virus$ family is also an outlier concerning the averages, with each operation being called only a few times, except for "`WriteFile`", which features a very high average call count. The highest average call count is associated with the "`ReadFile`" action in the $Miner$ family, followed by three operation counts that are all featured in the *Ransom* family: "`RegQueryValue`", "`WriteFile`", and "`ReadFile`". For the averages for the *Benign* family, there are no clear-cut patterns. All these averages are in the middle of all the values for any operations when looking across families.

Table 5.1. Top 30 Noriben Operation Description.

Name	Description	Name	Description
1. Process Start	Launching a process	16. QueryBasicInformationFile	Querying basic information from a file
2. Thread Create	Creating thread for a process	17. CreateFileMapping	Creating a mapping for files
3. Load Image	Loading an imagebase from a file into memory	18. QuerySecurityFile	Querying a security setting for a file
4. CreateFile	Creating a file	19. QueryStandardInformation File	Querying a standard information header from a file
5. RegOpenKey	Opening registry key	20. RegQueryKey	Querying a registry key
6. RegQueryValue	Querying a value in registry key	21. QuerySizeInformation Volume	Querying size information from a volume/drive
7. RegCloseKey	Closing registry key	22. QueryAttributeInformation Volume	Querying attribute information from a volume/drive
8. QueryName InformationFile	Querying file information	23. RegQueryMultipleValueKey	Querying multiple values from registry keys
9. CloseFile	Closing a file	24. RegEnumKey	Enumerating keys in registry
10. RegSetInfoKey	Setting information value in registry key	25. RegEnumValue	Enumeraring values in registry
11. ReadFile	Reading a file	26. QueryInformationVolume	Querying the creation time and the serial number of a volume
12. RegCreateKey	Creating a key in registry	27. QueryAttributeTagFile	Querying a tag attribute in the DesiredAccess parameter from a file
13. RegSetValue	Setting a value in registry key	28. SetRenameInformationFile	Setting a file to be allowed for renaming
14. WriteFile	Writing a file	29. TCP Connect	Establishing a connection with TCP
15. QueryDirectory	Querying a directory	30. TCP Send	Sending data with TCP

5.1.2 API call sequence for Binary Classification

In the work of Oliveira [114], a behavioral malware detection method based on Deep Graph Convolutional Neural Networks (DGCNNs) [115] for binary classification was proposed. They extracted the graph structure from the API

Table 5.2. Top-10 frequently occurrences of Noriben sequences classified by two numeric values: (1) average number for each category and (2) the number of applications for each and every operation-category pair.

Noriben Operations/ Total	**Benign (756)**	**Ransom (139)**	**Trojan (980)**	**Miner (43)**	**Rootkit (10)**	**Back-door (23)**	**Virus (11)**	**Other (38)**
Query Standard Information File	48.4/ 602	590.5/ 122	2920.6 /964	82.7 / 40	19.7/10	3.0/23	2.2/11	315.9/ 38
RegQuery Value	95.6 / 748	4185.4 / 124	200.9 /971	1955.9 / 40	331.9/10	164.1 / 23	7.5 /11	2546.5/ 38
RegQuery KeySecurity	62.2/86	2542.7 / 110	2196.7 /202	845.4 / 39	1118.4/8	161.5/ 21	0/0	416.2 / 9
RegQuery Key	224.7/ 613	2273.9 / 123	457.9 /969	2456.7 / 40	894.7/10	181.7 / 23	13.1/11	2299.6 / 38
Query Directory	143.4/ 373	1029.6 / 98	562.2 /754	135.5/ 30	10657.3 /8	3.2/20	1.9/11	203.2 / 25
WriteFile	507.9/ 498	3774.1/ 98	236.2 /447	200.4/ 35	819.6/7	4.3/16	585.0/4	49.4/ 31
ReadFile	453.5/ 605	2926.5/ 124	110.8 / 970	5373.1 /39	1195.6/10	60.2/ 23	34.8/11	308.6/ 38
RegOpen Key	99.4/ 748	1292.0/ 124	308.8 /971	2081.8/ 40	564.8/10	131.1 / 23	19.3 /11	2445.3/ 38
CreateFile	172.6/ 602	2239.5 /124	255.3 /966	787.6/ 40	2592.8/10	30.7 /23	16.0/11	500.5/ 38
CloseFile	114.4 / 745	1807.3 / 124	245.8 /971	785.3/ 40	2591.7/10	22.5/23	13.6/11	436.4 / 38

call sequences to generate their associated behavioral graphs. There were two experiments on an imbalanced public dataset [116]: (1) a random undersampling of the majority class (malware) to obtain a balanced dataset containing 1,079 goodware and 1,079 malware samples, and (2) an original imbalanced dataset consisting of 42,797 malware and 1,079 benign API call sequences. The API call sequences and the set of API calls were passed to a graph convolutional layer to generate behavioral graphs among the API calls. Each sequence comprised the first 100 uniquely consecutive API calls associated with the parent process, extracted from the Cuckoo Sandbox reports, and then classified into binary, representing one as malware and zero as benign. For feature columns, there were 100 columns ordered by [t_0 ... t_99], which contained an index (between 0 and 306) mapped to 307 distinct API calls by each index representing the API call name. Their experimental results showed that the proposed method achieved similar performance to LSTM networks, around 97% to distinguish between malicious and benign patterns through the graph structure when trained and tested on a balanced dataset.

5.2 Model architectures

Eight learning algorithms are evaluated for their malware detection performance: Random Forest, XGBoost, a fully connected neural network (FCNN), gated recurrent unit (GRU) and long short-term memory (LSTM), Transformers, and two combination approaches. To the best of our knowledge, the Transformers [117] are used in static analysis on the Android applications source code; however, using the full sequences of Noriben operation for malware prediction has yet to be used before.

For Random Forest and XGBoost, Scikit-Learn [54] is used to construct the models, while the other six approaches are implemented using TensorFlow [53]. Furthermore, to obtain good performance for these six models, all hyperparameters are tuned using KerasTuner's [56] RandomSearch. For proper evaluation, 10-fold cross-validation and a scheduled decrease in the learning rate are employed. We also initialize parameter values for all learning approaches before performing hyperparameter search space. The tuning ranges are represented in a list across all tables, which are used for searching for each parameter.

5.2.1 Random Forest

The Random Forest model is implemented using Scikit-Learn [54]. To obtain the best model, the model's hyperparameters are tuned using GridSearchCV 10-fold cross-validation, with micro-F1 as the objective function. The eight hyper-parameters subject to the grid search, their ranges of possible and selected values achieving the highest accuracy performance are presented in Table 5.3.

Table 5.3. Hyperparameter tuning for Random Forest Classifier.

Name	**Initial Parameters**	**Tuning Ranges**	**Best Value (Noriben)**	**Best Value (API calls)**
n_estimators	200	range(100,600,50)	250	500
criterion	gini	[‘gini’, ‘entropy’]	gini	entropy
max_depth	14	range(6,20,2)	14	6
max_features	sqrt	[‘sqrt’, ‘log2’]	sqrt	log2
bootstrap	True	[True, False]	True	True
min_samples_leaf	2	range(2,12,2)	4	2
min_samples_split	2	range(2,12,2)	2	2
max_leaf_nodes	30	range(15,55,5)	45	50

5.2.2 XGBoost

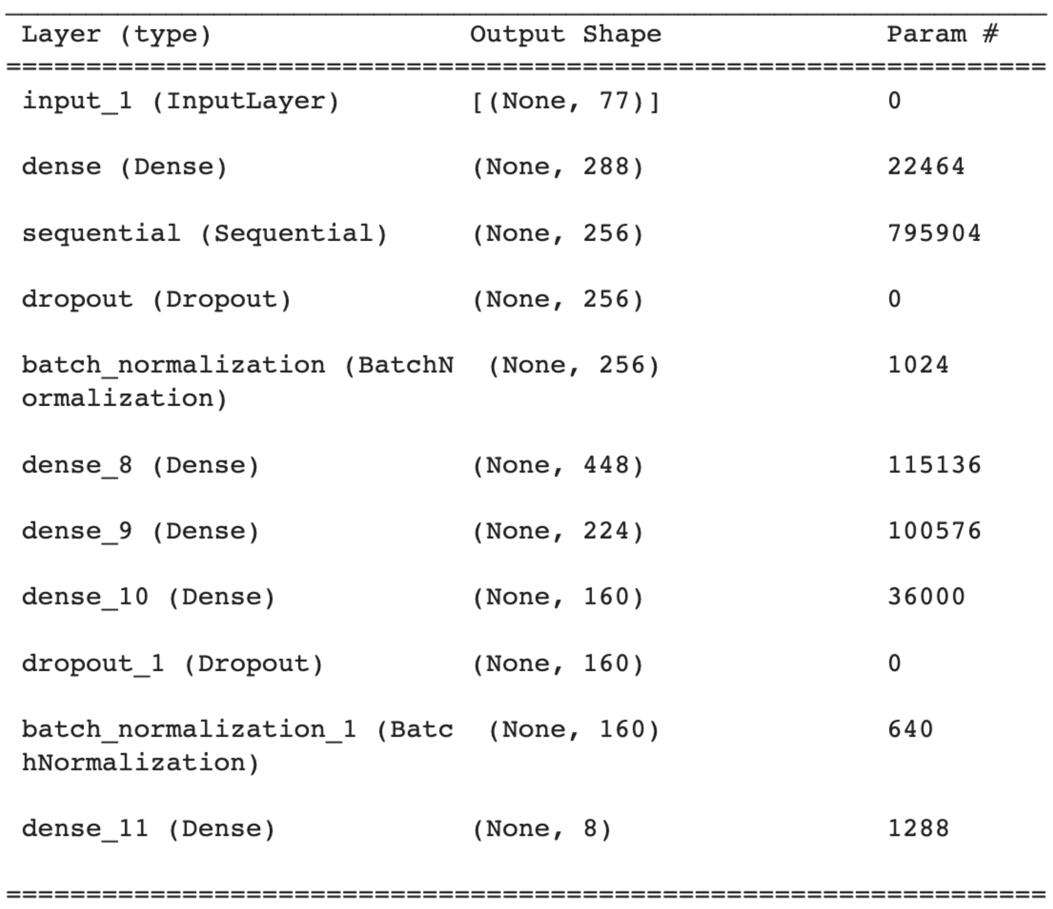
The XGBoost model is implemented using the XGBClassifier from Scikit-Learn [54]. We perform a grid search by using Ray Tune [111] with a resource per trial to “gpu’ to determine a well-performing set of values for the potential hyperparameters. ‘mlogloss’ and ‘AUC’ are used as evaluation metrics to measure the model performance. The five hyper-parameters, their ranges of possible values, and the selected best-performing values are presented in Table 5.4.

Table 5.4. Hyperparameter tuning for XGBoost Classifier.

Name	**Initial Parameters**	**Tuning Ranges**	**Best Value (Noriben)**	**Best Value (API calls)**
n_estimators	200	range(100,650,50)	200	400
max_depth	6	range(4,20,2)	10	14
learning_rate/eta	0.001	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.02	0.02
gamma	1.5	[0.8,1.0,1.5,2.0, 2.5,3.0,4.0,5.0]	1.5	2.5
min_child_weight	1	range(1,6,2)	3	1

5.2.3 FCNN

The fully connected network is configured using nonlinear activations and employing BatchNormalization, Dropout, and other regularizers to avoid overfitting. We use RandomSearch to determine the architecture and the best values for the hyperparameters. Table 5.5 presents the search parameters and the selected best values, whereas Figure 5.2 depicts the final best architecture for the FCNN, as obtained by RandomSearch.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 77)]              0

 dense (Dense)               (None, 288)               22464

 sequential (Sequential)     (None, 256)               795904

 dropout (Dropout)           (None, 256)               0

 batch_normalization (BatchN  (None, 256)              1024
 ormalization)

 dense_8 (Dense)             (None, 448)               115136

 dense_9 (Dense)             (None, 224)               100576

 dense_10 (Dense)            (None, 160)               36000

 dropout_1 (Dropout)         (None, 160)               0

 batch_normalization_1 (Batc  (None, 160)              640
 hNormalization)

 dense_11 (Dense)            (None, 8)                 1288

=================================================================
```

Figure 5.2. The best FCNN as determined by RandomSearch in Tensorflow.

5.2.4 GRU and LSTM

LSTM [118] and GRU [62] are standard approaches for sequence learning. We experiment with Bidirectional[3] CuDNNGRU [119] and CuDNNLSTM [119]

[3]Bidirectional GRU (Bi-GRU) is a type of bidirectional recurrent neural networks that is used to extract features in both directions from the sequences.

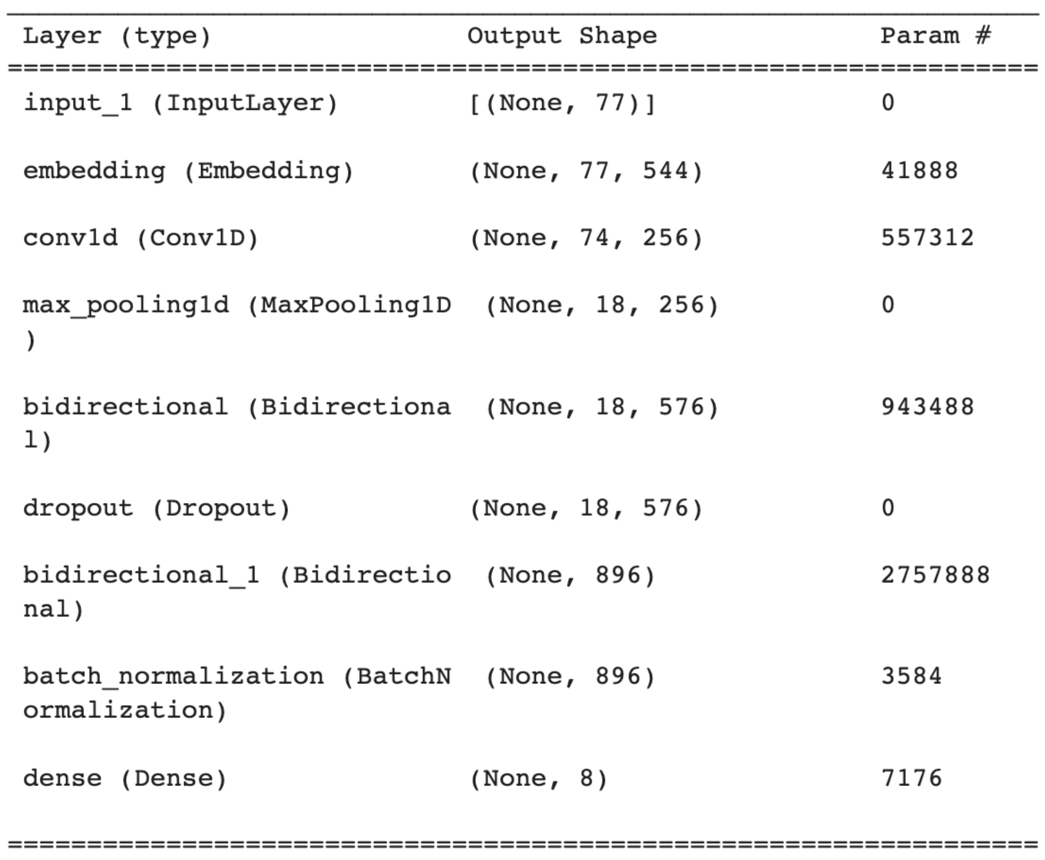
Table 5.5. FCNN hyperparameter tuning setup and outcome.

Name	**Initial Value**	**Tuning Ranges**	**Best Value (Noriben)**	**Best Value (API calls)**
batch_size	15	ranges(10,50,5)	40	50
epochs	100	ranges(100,600,50)	300	300
optimizer	RMSprop	[‘Adam’,‘RMSprop’, ‘Adadelta’,‘Adagrad’]	Adam	Adagrad
learning_rate	0.0002	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.0001	0.002
Neurons	[256,256,128,128]	range(128,1024,32)	[288,448,224, 160]	[832,320,160, 480]
Hidden Layers	5	range(2,10)	7	7
Hidden Neurons	[256,256,256, 256,256]	range(128,512,32)	[512,448,480, 160,320,128,256]	[224,128,288, 512,352,128,288]
bias_ regularizer	[0.002, 0.02, 0.002, 0.002, 0.002]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.02, 0.0002, 0.002, 0.02, 0.0001]	[0.02, 0.02, 0.0001, 0.002, 0.001]
kernel_ regularizer	[0.01,0.02,0.01, 0.01,0.01]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.001, 0.01, 0.01, 0.02, 0.002]	[0.0001, 0.02, 0.002, 0.002, 0.0001]
kernel_ constraint	[2.0,1.5,1.5, 1.5,1.0]	range(1,3,0.5)	[1.0,3.0,1.0, 2.5,1.5]	[2.5,1.5,2.0, 2.0,2.0]
activity_ regularizer	[0.0001,0.001, 0.0001,0.0001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0002, 0.001, 0.001, 0.01, 0.0002]	[0.02, 0.0001, 0.001, 0.0001, 0.01]

architectures, where a convolutional layer first processes the embedded input. To determine the best configurations, RandomSearch in Keras is used, and the setup with outcomes is provided in Table 5.6 and Figure 5.3.

5.2.5 Transformers

Transformers [55] is an attention-based encoder-decoder architecture. The attention mechanism focuses on different tokens while processing words to model opcode sequences. The first layer is an embedding layer to convert words into vectors, followed by the positional encoding layer to add the position information for each word. Next, all input sequences are encoded to attention representations in the encoder layers, consisting of two sub-layers: multi-headed attention and two fully-connected layers, with a ReLU activation in between. Multi-headed attention computed the attention weights for the input by processing each word separately and had a score corresponding to other words in



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 77)]              0

 embedding (Embedding)       (None, 77, 544)           41888

 conv1d (Conv1D)             (None, 74, 256)           557312

 max_pooling1d (MaxPooling1D  (None, 18, 256)          0
 )

 bidirectional (Bidirectiona  (None, 18, 576)          943488
 l)

 dropout (Dropout)           (None, 18, 576)           0

 bidirectional_1 (Bidirectio  (None, 896)              2757888
 nal)

 batch_normalization (BatchN  (None, 896)              3584
 ormalization)

 dense (Dense)               (None, 8)                 7176

=================================================================
```

Figure 5.3. The best GRU as determined by RandomSearch in Tensorflow.

the sequence. The higher the score, the more focus. It, therefore, represents the attention scores for each target sequence word that also captured the influence of the attention scores from the input sequence. Later, a hidden state is passed to the decoding stage after obtaining an output vector on how each word should attend to all other words. Each hidden state is used to determine where the network should pay attention. The decoder has a similar two-sub-layer setup as the encoder, but an attention layer operated slightly differently to attend to values from the encoder. The decoder receives a start token, a list of previous outputs, and the encoder outputs as inputs, then generates a token at a time. Lastly, GlobalAveragePooling1D is added, which averaged over sequence dimension and returned a fixed-length output vector before feeding into the last softmax layer to get the word probabilities. The details about the model's configuration and optimization from the RandomSearch are provided in Table 5.7 and Table 5.8, respectively.

Table 5.6. Hyperparameter tuning for GRU and LSTM for Noriben and API call sequences.

Name	**Initial Value**	**Tuning Ranges**	**GRU/LSTM Best Value (Noriben)**	**GRU/LSTM Best Value (API calls)**
batch_size	15	range(10,50,5)	25	25
epochs	200	range(50,650,50)	600	600
optimizer	RMSprop	[‘Adam’,‘RMSprop’, ‘Adadelta’,‘Adagrad’]	Adam	Adam
learning_rate	0.0001	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.0002	0.0002
embedding	128	ranges(256,1024,32)	544	448
embedding regularizer	0.0001	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.02	0.0001
gru/lstm units	[512, 256]	ranges(128,512,32)	[288, 448]	[416, 288]
recurrent regularizer	[0.0001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.01, 0.001]	[0.01, 0.01]
kernel regularizer	[0.001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.002, 0.01]	[0.0002, 0.002]
bias regularizer	[0.01, 0.02]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.01, 0.01]	[0.02, 0.01]
kernel constraint	[2.0, 1.5]	range(1,3,0.5)	[1.5, 2.0]	[1.5, 2.0]
activity regularizer	[0.0001, 0.0002]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.02, 0.002]	[0.002, 0.02]

5.2.6 Combination of GRU and transformers

This classifier is created to integrate a GRU and a Transformers network. Therefore, it is a simple concatenation of the outputs of the penultimate layers of both networks. Specifically, the output of the final BatchNormalization layer of the GRU and the output of the final GlobalAveragePooling1D layer of the Transformers are concatenated and passed to a final softmax layer. The tuning information for this combination is provided in Table 5.9.

5.2.7 Combination of FCNN and Transformers

This classifier is built as a hybrid architecture of a fully connected neural network and Transformers. Effectively, the FCNN replaces the GRU in the previous combination approach. The details of the hyperparameter tuning are

Table 5.7. Transformer Layers

Layers	**Layer Details**
(Enc) Embedding Layer	input_dim: *input_vocab_size*, output_dim: *trans_dim*, mask_zero: *True*, embeddings_initializer: *he_uniform*, embeddings_regularizer: *l2(1e-4)*
(Enc) Positional Encoding	positional_encoding(maximum_position_encoding, trans_dim)
(Enc) MultiHeadAttention	num_heads, key_dim:*2*, dropout:*0.1*, use_bias:*True*, bias_initializer:*ones*, kernel_initializer:*glorot_uniform* (query, value, key)
(Enc) Dense Layer	units: *trans_dim*, activation function: *swish*, use_bias: *True*, kernel_initializer: *he_uniform*, bias_initializer: *zeros*, bias_regularizer: *l2(1e-3)*, kernel_regularizer: *l2(1e-4)*, kernel_constraint: *MaxNorm(1.)*, activity_regularizer: *l2(2e-4)*
(Enc) Normalize Layer	LayerNormalization(epsilon=1e-8)
(Enc FFN) 2 Dense layers	units: *feed_forward_dim*, activation function: *relu* and units: *trans_dim*
(Enc) Normalize Layer	LayerNormalization(epsilon=1e-8)
(Dec) Embedding Layer	input_dim: *target_vocab_size*, output_dim: *trans_dim*, mask_zero: *True*, embeddings_initializer: *he_uniform*, embeddings_regularizer: *l2(1e-4)*
(Dec) Positional Encoding	positional_encoding(maximum_position_encoding, trans_dim)
(Dec) MultiHeadAttention	num_heads, key_dim:*2*, dropout:*0.1*, use_bias:*True*, bias_initializer:*zeros*, kernel_initializer:*glorot_uniform* (query, value, key)
(Dec) Dense Layer	units: *trans_dim*, activation function: *swish*, use_bias: *True*, kernel_initializer: *he_uniform*, bias_initializer: *zeros*, bias_regularizer: *l2(1e-3)*, kernel_regularizer: *l2(1e-4)*, kernel_constraint: *MaxNorm(1.)*, activity_regularizer: *l2(2e-4)*
(Dec) Normalize Layer	LayerNormalization(epsilon=1e-8)
(Dec) Encoder-Decoder attention	MultiHeadAttention(num_heads,key_dim:*2*, dropout:*0.1*, use_bias:*True*, bias_initializer:*zeros*,kernel_initializer:*glorot_uniform*) (query, value, key) and Dense(trans_dim)
(Dec) Dense Layer	units: *trans_dim*, activation function: *swish*, use_bias: *True*, kernel_initializer: *he_uniform*, bias_initializer: *zeros*, bias_regularizer: *l2(1e-3)*, kernel_regularizer: *l2(1e-4)*, kernel_constraint: *MaxNorm(1.)*, activity_regularizer: *l2(2e-4)*
(Dec) Normalize Layer	LayerNormalization(epsilon=1e-8)
(Dec FFN) 2 Dense layers	units: *feed_forward_dim*, activation function: *relu* and units: *trans_dim*
(Dec) Normalize Layer	LayerNormalization(epsilon=1e-8)
(Dec) GlobalAverage-Pooling1D	GlobalAveragePooling1D(decoder)
Dense Layer	units: *8*, activation function: *softmax*, kernel_initializer: *he_uniform*

shown in Table 5.10.

Table 5.8. Hyperparameter tuning for Transformers.

Name	**Initial Value**	**Tuning Ranges**	**Best Value (Noriben)**	**Best Value (API calls)**
batch_size	10	range(10,50,5)	25	45
epochs	100	range(50,650,50)	600	550
optimizer	Adam	[‘Adam’,‘RMSprop’, ‘Adadelta’,‘Adagrad’]	Adam	RMSprop
learning_rate	0.0001	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.0002	0.001
trans_dim	512	ranges(512,2048,32)	704	928
num_heads	12	ranges(6,20,2)	12	10
feed_forward_dim	1024	ranges(1024,3172,32)	2048	2944
enc_layer	5	ranges(2,10)	2	3
dec_layer	5	ranges(1,10)	2	3
embedding regularizer	[0.001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.002, 0.01]	[0.0002, 0.01]
kernel regularizer	[0.001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.02, 0.01]	[0.01, 0.01]
bias regularizer	[0.01, 0.02]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.001, 0.001]	[0.0001, 0.01]
kernel constraint	[2.0, 1.5]	range(1,3,0.5)	[2.5, 2.0]	[3.0, 2.5]
activity regularizer	[0.001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.001, 0.002]	[0.0001, 0.0001]

5.3 Experimental Results

This section presents experimental results for the eight algorithms applied to Noriben sequences. We use different evaluation metrics to measure the performance, including accuracy, F1-measure, Mean Absolute Error (MAE, and Area Under the receiver operating characteristic Curve (AUC).

Table 5.11 lists results for passing sequences through all algorithms with their default parameter settings. Overall, GRU performed best out of all eight classifiers, achieving an accuracy of 76.4%.

Table 5.12 shows significant improvements by properly tuning the potential hyperparameters. The overall performance increased for all classifications compared to the default models from Table 5.11. The integration of GRU with Transformers exhibited the best performance, reaching an accuracy of 97% and achieving the overall best F1, MAE, and AUC results. The figure represented the correct class, but the absolute error is 0.012. So a low MAE

Table 5.9. Hyperparameter tuning for combining GRU and Transformers.

Name	**Initial Value**	**Tuning Ranges**	**Best Value (Noriben)**	**Best Value (API calls)**
batch_size	15	ranges(10,50,5)	25	25
epochs	100	ranges(100,600,50)	300	150
optimizer	RMSprop	[‘Adam’,‘RMSprop’, ‘Adadelta’,‘Adagrad’]	Adam	Adagrad
learning_rate	0.002	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.0001	0.0001
embedding	256	range(256,1024,32)	448	512
embedding regularizer	[0.001,0.001,0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0001,0.002,0.01]	[0.02,0.01,0.01]
gru units	[512, 256]	ranges(128,512,32)	[224, 384]	[224, 224]
recurrent regularizer	[0.0001,0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0001,0.002]	[0.002,0.0002]
kernel regularizer	[0.001,0.001,0.001, 0.001,0.01]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.002, 0.01, 0.01, 0.002, 0.01]	[0.0002,0.01,0.01, 0.002,0.0002]
bias regularizer	[0.01,0.02,0.01,0.01, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.002,0.002,0.0002, 0.01,0.002]	[0.001,0.0001,0.01, 0.001,0.001]
kernel constraint	[2.0, 1.5, 1.5, 2.0, 1.0]	range(1,3,0.5)	[3.0, 3.0, 2.0, 1.5, 3.0]	[2.0,3.0,2.5,1.0,3.0]
activity regularizer	[0.001,0.001,0.001, 0.001,0.0001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.01,0.0001,0.01, 0.01,0.0002]	[0.01,0.0001,0.0001, 0.001,0.01]
trans_dim	128	ranges(512,1024,32)	864	896
num_heads	4	ranges(6,20,2)	20	6
feed_forward_dim	1024	ranges(1024,3172,128)	2944	1920
enc_layer	3	ranges(2,10)	4	3
dec_layer	2	ranges(1,10)	6	7

implied higher average confidence. To measure our model effectiveness, we also evaluated our models on the public dynamic API call sequences dataset [114], which contained 42,797 malware and 1,079 goodware. We use the same architecture presented in eight classifiers for the API calls dataset and optimized the model parameters. The result showed that an LSTM network could provide the best predictive result with the F1 score at 99%, as depicted in Table 5.13.

Regarding measuring performance, Table 5.15 lists each class’s precision, recall, and F1 score. Mostly, the samples are correctly classified into their

Table 5.10. Hyperparameter tuning for combining FCNN and Transformers.

Name	**Initial Value**	**Tuning Ranges**	**Best Value (Noriben)**	**Best Value (API calls)**
batch_size	15	ranges(10,50,5)	30	10
epochs	100	ranges(100,600,50)	150	100
optimizer	RMSprop	[‘Adam’,‘RMSprop’, ‘Adadelta’, ‘Adagrad’]	RMSprop	Adam
learning_rate	0.0002	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.0001	0.0002
Neurons	[256,256,128,128]	range(128,1024,32)	[704,224,288, 320]	[640,320,192,128]
Hidden Layers	5	range(2,10)	7	7
Hidden Neurons	[256,256,256, 256,256]	range(128,512,32)	[192,288,480, 352,480,320,288]	[256,256,256,256, 256,256,256]
bias_ regularizer	[0.001,0.002,0.002, 0.001,0.001,0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0001,0.02,0.01, 0.02,0.0001,0.002]	[0.0001,0.0001,0.02, 0.02,0.02,0.0002]
kernel_ regularizer	[0.01,0.02,0.02, 0.01,0.01,0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.002,0.01,0.02, 0.001,0.002,0.002]	[0.01,0.02,0.01, 0.01,0.002,0.01]
kernel_ constraint	[2.0,1.5,1.5,1.5, 1.0,1.0]	range(1,3,0.5)	[1.5,1.5,2.5,1.5, 2.0,2.0]	[2.0,2.0,1.0, 1.5,3.0,2.5]
activity_ regularizer	[0.0001,0.001,0.0001, 0.0001,0.001,0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0002,0.0002, 0.001, 0.01,0.001,0.0001]	[0.001,0.01,0.002, 0.01,0.0002,0.02]
trans_dim	512	ranges(512,1024,32)	608	896
num_heads	8	ranges(6,20,2)	10	10
feed_forward_dim	1024	ranges(1024,3172,128)	2176	2688
enc_layer	6	ranges(2,10)	6	6
dec_layer	6	ranges(1,10)	4	5
embedding regularizer	[0.001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.01, 0.01]	[0.02,0.01]

respective families.

In addition to reports by evaluation metrics, we presented the total elapsed time on the training default configuration and hyperparameter tuning with ten maximum trials for each classifier. We use stratified 10-Folds cross-validation using the GPU. Although RandomSearch considered not all possible combinations, integrating GRU and Transformers took the most prolonged hours for optimization, as represented in Table 5.14.

We also analyzed the confusion matrix, which compared actual and predictive values, as depicted in Figure 5.4. Each class is calculated. For example, all columns and rows except the values of the trojan class are classification errors. The matrix showed the domination of the diagonal values, with most off-diagonal values being very close to zero. This classifier had the best classification capability for trojans, followed by benign. Figure 5.5 presents ROC curves generated by combining a GRU with Transformers for all classes. Again, the almost perfect prediction was achieved across all categories.

Finally, the cross-validation with hyperparameter tuning results is presented. From Table 5.16 to Table 5.20, they evaluated different max tokens with distinct sequence lengths on Noriben operations from 1000 to 10000. To provide a more visual representation, Figure 5.6 to Figure 5.9 provides line graphs for the various evaluation metrics. It is noteworthy that LSTM, GRU, and a combined GRU with Transformers exhibited similar trends.

Table 5.11. Cross-validation results for default hyperparameter values on Noriben operations.

Algorithm	Avg. ACC	Avg. F1	Avg. MAE	Avg. AUC
RF	0.607±0.02	0.607±0.02	0.098±0.01	0.776±0.01
XGB	**0.771±0.01**	**0.771 ±0.01**	**0.057±0.01**	0.869±0.01
LSTM	0.597±0.11	0.601±0.12	0.107±0.03	0.880±0.03
GRU	0.764±0.03	0.765±0.03	0.063±0.03	0.952±0.02
Transformers	0.749±0.04	0.749±0.04	0.066±0.03	0.941±0.02
FCNN	0.653±0.03	0.646±0.03	0.114±0.01	0.923±0.01
GRU + Transformers	0.754±0.05	0.759±0.04	0.066±0.03	**0.954±0.02**
FCNN + Transformers	0.704±0.05	0.705±0.04	0.095±0.03	0.941±0.03

Table 5.12. Cross-validation results after hyperparameter tuning on Noriben operations.

Algorithm	Avg. ACC	Avg. F1	Avg. MAE	Avg.AUC
RF	0.637±0.02	0.637±0.02	0.091±0.01	0.793±0.01
XGB	0.798±0.01	0.798±0.01	0.050±0.01	0.885±0.01
LSTM	0.939±0.07	0.941±0.06	0.020±0.02	0.989±0.02
GRU	0.941±0.06	0.942±0.06	0.021±0.02	0.990±0.02
Transformers	0.935±0.06	0.936±0.06	0.029±0.02	0.986±0.02
FCNN	0.882±0.06	0.878±0.06	0.049±0.01	0.967±0.04
GRU + Transformers	**0.970±0.06**	**0.970±0.06**	**0.012±0.02**	**0.995±0.02**
FCNN + Transformers	0.928±0.06	0.929±0.06	0.021±0.02	0.989±0.02

Table 5.13. Cross-validation results after hyperparameter tuning for dynamic API call sequences [114] on binary classification.

Algorithm	Avg. ACC	Avg. F1	Avg. MAE	Avg.AUC
RF	0.906±0.01	0.906±0.01	0.094±0.01	0.906±0.01
XGB	0.990±0.01	0.990±0.01	0.010±0.01	0.990±0.01
LSTM	**0.996±0.002**	**0.996±0.002**	**0.008±0.002**	**0.995±0.003**
GRU	0.994±0.002	0.994±0.002	0.014±0.001	0.993±0.003
Transformers	0.992±0.001	0.992±0.001	0.010±0.001	0.984±0.002
NN	0.975±0.001	0.975±0.001	0.053±0.001	0.979±0.003
GRU + Transformers	0.987±0.001	0.987±0.001	0.051±0.001	0.968±0.001
FCNN + Transformers	0.993±0.001	0.993±0.001	0.010±0.001	0.988±0.001

Table 5.14. Training time for each algorithm on Noriben operations.

Algorithm	Default Noriben Training	Noriben Tuning	Default API Training	API Tuning
RF	43s	40h:41m:51s	2m:10s	48h:16m:02s
XGB	2m:28s	9h:43m	6m:17s	84h:07m:53s
LSTM	1h:41s	3h:45s	4h:44m:33s	9h:58m:15s
GRU	1h:39s	3h:08s	3h:57m:15s	8h:54m:55s
Transformers	10h:48m:12s	12h:30m:09s	31h:01m:18s	48h:38m:42s
NN	16m:20s	1h:49m:15s	1h:35m:13s	2h:40m:13s
GRU + Transformers	10h:58m:20s	12h:59m:40s	38h:52m:24s	44h:58m:28s
NN + Transformers	10h:35m:27s	12h:41m:50s	41h:13m:31s	46h:01m:12s

Metrics	Classifiers	Benign	Backdoor	Miner	Ransom	Rootkit	Trojan	Virus	Other
Precision	GRU	0.95±0.01	**1.00±0.01**	0.88±0.02	**0.98±0.01**	0.20±0.03	0.89±0.02	0.75±0.02	0.50±0.02
	Transformers	0.94±0.01	0.92±0.01	**1.00±0.01**	0.90±0.01	0.70±0.02	0.95±0.01	**1.00±0.01**	0.30±0.02
	GRU+Transformers	**0.99±0.04**	**1.00±0.01**	1.00±0.30	0.93±0.16	**0.90±0.05**	**0.97±0.23**	0.79±0.15	**0.97±0.30**
Recall	GRU	0.91±0.01	0.69±0.02	0.76±0.02	0.48±0.02	0.15±0.03	0.97±0.01	0.54±0.03	0.50±0.02
	Transformers	0.95±0.01	0.76±0.02	0.71±0.01	0.80±0.01	0.70±0.02	0.94±0.01	0.71±0.03	0.30±0.02
	GRU+Transformers	**0.96±0.06**	**0.96±0.15**	**0.90±0.30**	**0.93±0.20**	**0.90±0.04**	**0.98±0.01**	**1.00±0.22**	**0.92±0.30**
F1-Score	GRU	0.93±0.02	0.62±0.02	0.79±0.02	0.63±0.02	0.16±0.03	0.93±0.01	0.61±0.03	0.50±0.02
	Transformers	0.94±0.02	0.81±0.02	0.78±0.01	0.84±0.01	0.70±0.02	0.95±0.01	0.78±0.03	0.30±0.02
	GRU+Transformers	**0.97±0.04**	**0.98±0.09**	**0.95±0.30**	**0.93±0.18**	**0.90±0.04**	**0.98±0.17**	**0.88±0.20**	**0.95±0.30**

Table 5.15. Precision, recall, and F1 score per class for 10-fold cross-validation on three classifiers (GRU, Transformers, and a hybrid of GRU and Transformers) on Noriben operations.

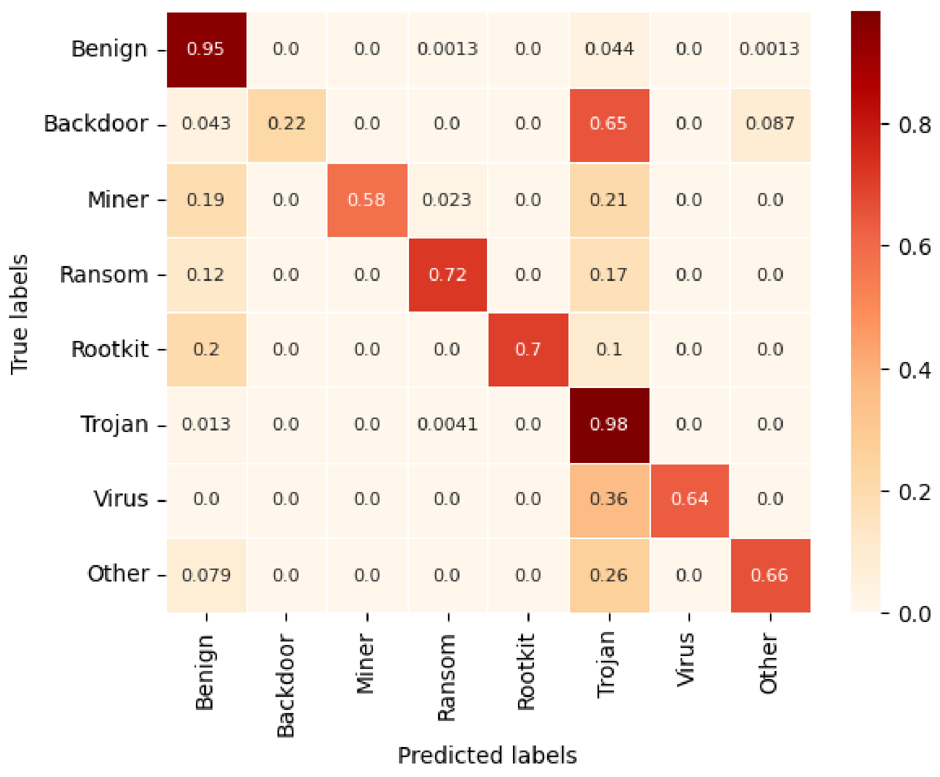


Figure 5.4. Confusion matrix among different malware families for a combination of GRU and Transformers.

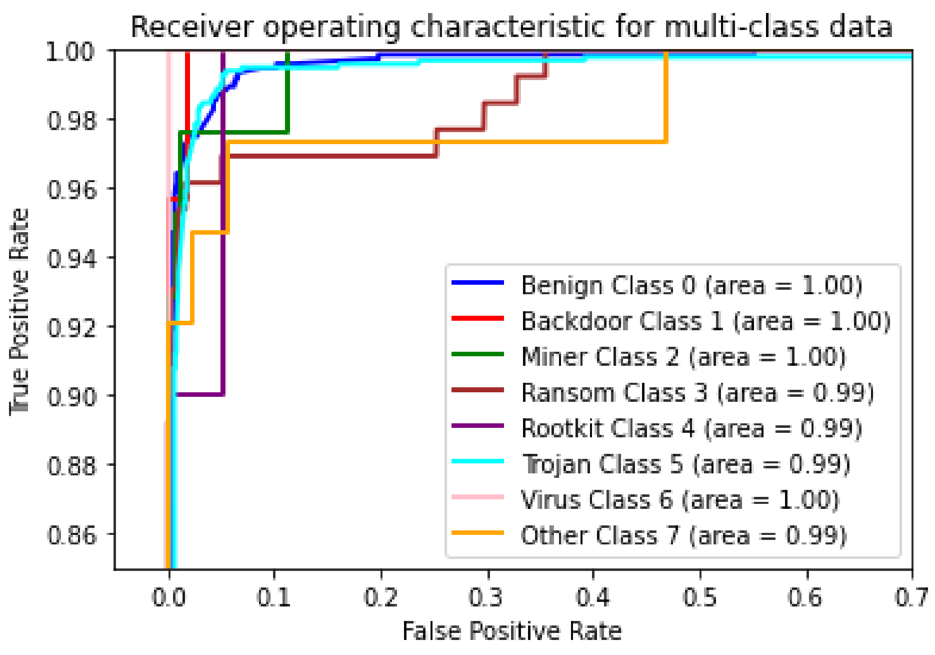


Figure 5.5. ROC Curve for a combination of GRU and Transformers.

Table 5.16. Cross-validation results after hyperparameter tuning on different max tokens with different sequence lengths for LSTM.

Metrics	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Avg. ACC	0.894±0.05	0.927±0.07	0.936±0.06	0.880±0.05	0.939±0.06	0.934±0.06	0.937±0.07	0.932±0.07	0.934±0.06	0.928±0.08
Avg. F1	0.899±0.05	0.928±0.07	0.938±0.06	0.888±0.05	0.939±0.06	0.934±0.06	0.937±0.07	0.934±0.07	0.933±0.07	0.927±0.08
Avg. MAE	0.039±0.02	0.024±0.02	0.021±0.02	0.035±0.01	0.021±0.02	0.023±0.01	0.020±0.02	0.022±0.02	0.023±0.02	0.023±0.02
Avg. AUC	0.987±0.01	0.988±0.02	0.988±0.02	0.981±0.02	0.988±0.03	0.990±0.02	0.991±0.02	0.988±0.02	0.990±0.02	0.988±0.02

Table 5.17. Cross-validation results after hyperparameter tuning on different max tokens with different sequence lengths for GRU.

Metrics	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Avg. ACC	0.909±0.06	0.925±0.06	0.932±0.06	0.928±0.05	0.939±0.06	0.934±0.06	0.933±0.07	0.936±0.06	0.915±0.06	0.933±0.07
Avg. F1	0.906±0.06	0.922±0.06	0.932±0.06	0.924±0.05	0.937±0.06	0.934±0.06	0.936±0.07	0.936±0.06	0.914±0.07	0.934±0.07
Avg. MAE	0.042±0.01	0.029±0.01	0.027±0.01	0.035±0.02	0.023±0.02	0.022±0.01	0.026±0.02	0.021±0.01	0.030±0.02	0.022±0.02
Avg. AUC	0.978±0.03	0.987±0.02	0.989±0.01	0.987±0.01	0.989±0.02	0.990±0.01	0.987±0.02	0.991±0.01	0.987±0.02	0.991±0.02

Table 5.18. Cross-validation results after hyperparameter tuning on different max tokens with different sequence lengths for Transformers.

Metrics	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Avg. ACC	0.817±0.04	0.736±0.02	0.835±0.04	0.791±0.04	0.767±0.03	0.739±0.03	0.737±0.03	0.736±0.03	0.709±0.04	0.817±0.04
Avg. F1	0.823±0.03	0.715±0.02	0.832±0.05	0.792±0.04	0.763±0.03	0.729±0.02	0.733±0.03	0.666±0.03	0.689±0.04	0.817±0.04
Avg. MAE	0.064±0.01	0.109±0.01	0.058±0.01	0.068±0.01	0.080±0.01	0.096±0.01	0.095±0.01	0.110±0.01	0.110±0.01	0.060±0.01
Avg. AUC	0.970±0.01	0.937±0.01	0.973±0.01	0.956±0.01	0.962±0.01	0.949±0.01	0.948±0.01	0.943±0.01	0.942±0.01	0.975±0.01

Table 5.19. Cross-validation results after hyperparameter tuning on different max tokens with different sequence lengths for FCNN.

Metrics	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Avg. ACC	0.828±0.04	0.814±0.03	0.803±0.02	0.789±0.02	0.789±0.02	0.822±0.02	0.810±0.03	0.786±0.02	0.778±0.02	0.810±0.03
Avg. F1	0.831±0.04	0.815±0.03	0.797±0.03	0.772±0.02	0.788±0.03	0.821±0.02	0.810±0.02	0.785±0.03	0.770±0.02	0.811±0.03
Avg. MAE	0.062±0.01	0.067±0.01	0.077±0.01	0.085±0.01	0.080±0.01	0.066±0.01	0.070±0.01	0.086±0.01	0.091±0.01	0.072±0.01
Avg. AUC	0.973±0.01	0.973±0.01	0.963±0.01	0.961±0.01	0.958±0.01	0.971±0.01	0.966±0.01	0.958±0.01	0.954±0.01	0.966±0.01

Table 5.20. Cross-validation results after hyperparameter tuning on different max tokens with different sequence lengths for GRU+Transformers.

Metrics	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Avg. ACC	0.926±0.06	0.926±0.05	0.880±0.04	0.898±0.05	0.936±0.06	0.934±0.06	0.934±0.06	0.932±0.07	0.935±0.07	0.936±0.06
Avg. F1	0.927±0.06	0.926±0.05	0.878±0.04	0.899±0.05	0.936±0.06	0.935±0.06	0.936±0.06	0.932±0.07	0.936±0.07	0.937±0.06
Avg. MAE	0.026±0.01	0.027±0.01	0.052±0.01	0.040±0.01	0.020±0.02	0.020±0.02	0.020±0.02	0.024±0.02	0.021±0.02	0.021±0.02
Avg. AUC	0.993±0.01	0.990±0.02	0.984±0.02	0.983±0.01	0.990±0.02	0.991±0.02	0.990±0.02	0.991±0.02	0.991±0.01	0.991±0.01

Table 5.21. Cross-validation results after hyperparameter tuning on different max tokens with different sequence lengths for FCNN+Transformers.

Metrics	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Avg. ACC	0.826±0.03	0.906±0.06	0.943±0.07	0.884±0.06	0.883±0.06	0.828±0.06	0.868±0.05	0.837±0.04	0.866±0.05	0.883±0.06
Avg. F1	0.820±0.03	0.905±0.06	0.943±0.07	0.887±0.06	0.885±0.06	0.829±0.06	0.870±0.05	0.834±0.04	0.863±0.06	0.877±0.06
Avg. MAE	0.060±0.01	0.035±0.01	0.019±0.02	0.046±0.01	0.041±0.01	0.068±0.01	0.046±0.01	0.067±0.01	0.050±0.01	0.054±0.01
Avg. AUC	0.975±0.01	0.985±0.02	0.992±0.02	0.985±0.02	0.985±0.01	0.976±0.02	0.983±0.02	0.971±0.01	0.979±0.02	0.985±0.01

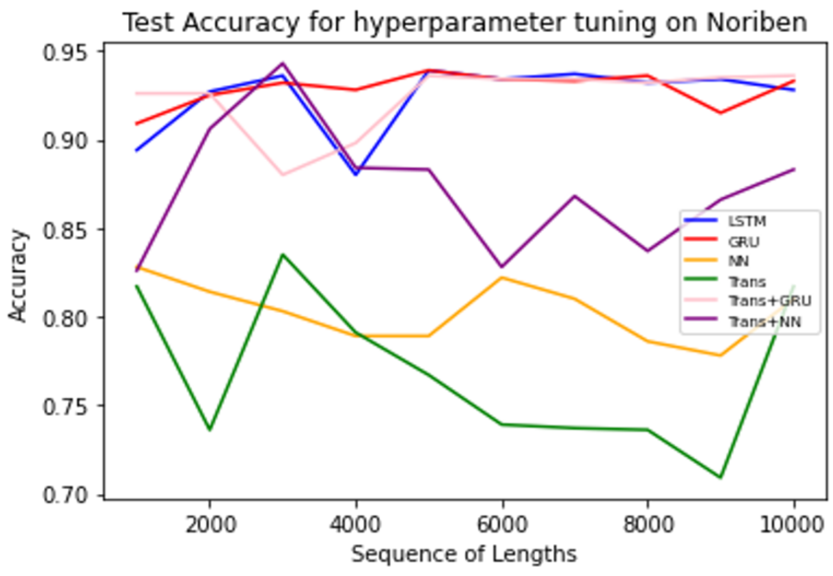


Figure 5.6. A comparison of accuracy among sequence models with LSTM and a hybrid model of Transformers and GRU exhibited very similar trends. In contrast, GRU had an upward trend, but dropped when a sequence length was 9000, and then up again.

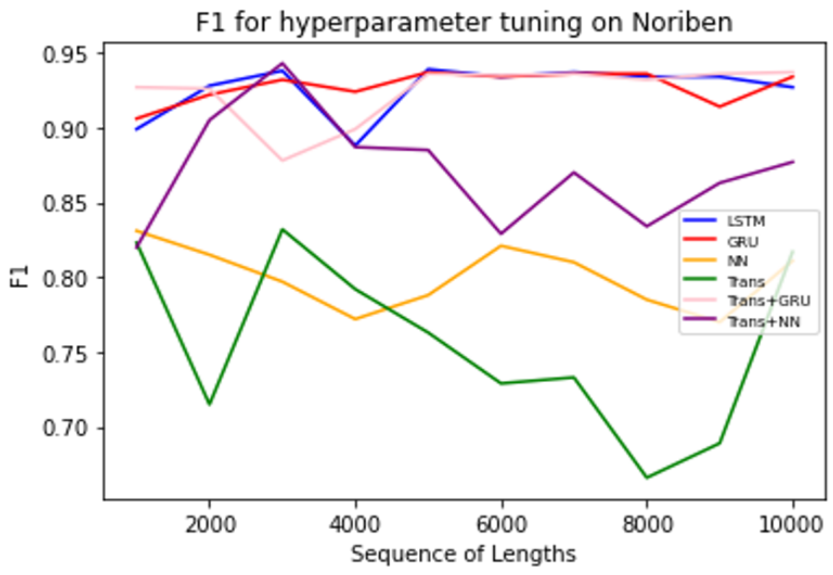


Figure 5.7. A comparison of the F1 score among sequence models by almost all classifiers showed the same movement as test accuracy except for Transformers, which had a different pattern after the sequence length was 4000.

5.4 Discussion

We used dynamic analysis to extract Noriben sequences and then preprocessed and fed them into eight classifiers to predict malware categories. Based on a

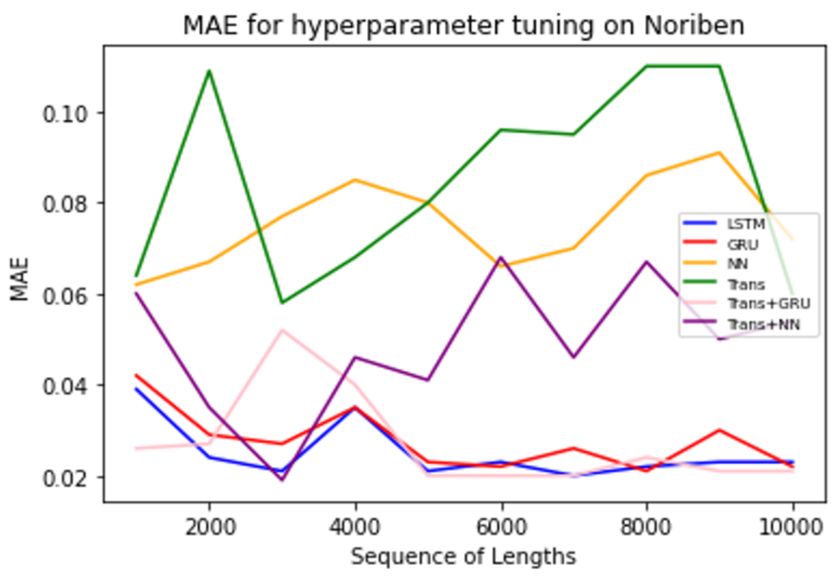


Figure 5.8. A comparison of MAE among sequence models with Transformers had the highest error value. LSTM and GRU had the least MAE values, which had a downward trend for a sequence length between 1000 and 4000. While a combination of Transformers had an opposite direction, it nearly stayed the same as LSTM after passing a 4000 in length.

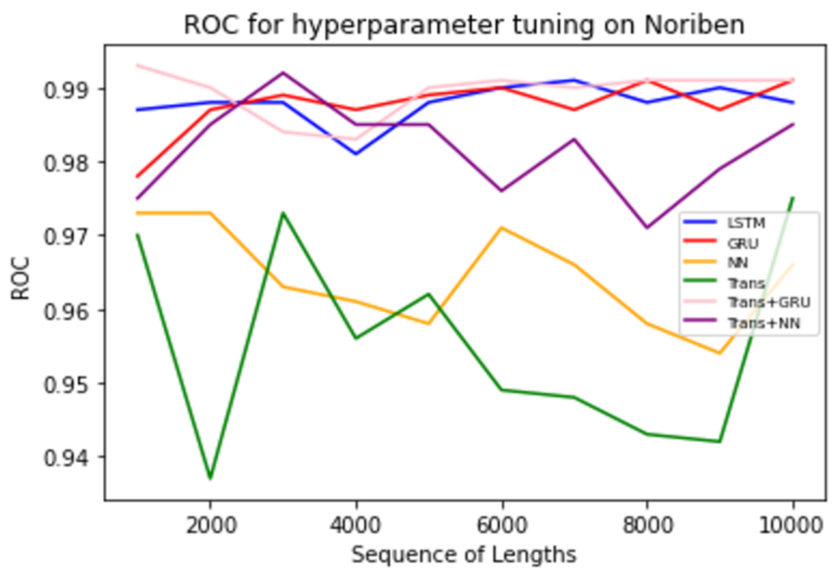


Figure 5.9. A comparison of the ROC Curve among sequence models with LSTM, GRU, and a combined Transformers and GRU shared a similar trend since the sequence length is 5000.

consecutive sequence of Noriben operations, a hybrid architecture of a GRU network and Transformers outperformed other approaches, yielding the highest F1 score of 97%. This combination helped us obtain better results than simpler architectures with only LSTM or GRU. Thus, taking dynamic sequences captured from Noriben Python script as a model's input can provide excellent predictive results. While dynamic analysis information was essential, we only looked at Noriben operations, which contained information on activities captured during malware execution. One limitation of our approach was that some applications encrypted their log files during the experiment, making extracting raw sequences impossible. In addition to our dynamic Noriben sequences, we applied our classifiers to the imbalanced public dataset of [116]. The results showed that our models could correctly classify malware producing high values for all evaluation metrics. For further improvement, we would need a large number of samples with more diversity and augment them with other dynamic artifacts to obtain more feature sets.

Chapter 6

Multiclass Malware Classification using either Static Opcodes or Dynamic API Calls

This chapter presents an automated method using static analysis for extracting opcode sequences of a length of up to 5000 for 4000 samples, and employing these sequences for classifying potential malware into eight classes, namely ransomware, trojan, backdoor, rootkit, virus, miner, benign, and other. We also compare our work with a public malware dataset that comprised more than 7000 samples in the sequence length of 342 API calls from eight different malware families. Seven different classifiers are employed to train these datasets: RF, XGB, MLP, LSTM, GRU, Transformers, and a combination of GRU and Transformers. Acquiring static data from mnemonic instructions (opcodes) becomes prevalent to prevent damage from execution. It could provide a holistic view of the application statically on what operation to perform, even though manipulating address parameters and changes in the execution flow could be obstacles. Obtaining opcode sequences could be conducted by both static and dynamic analysis [120, 121]. Nevertheless, it is less time-consuming to extract them via static analysis because it can provide an overall picture of what the program looks like without execution. For dynamic analysis, extracting API calls is the most popular for security researchers to observe functions during runtime.

6.1 Data Collection and Preprocessing

6.1.1 Opcodes

We focus on "`BaseOfCode`", a relative offset of code in code sections (.text) loaded into the memory. First, we disassemble the application. Then, the opcode sequences are carved out from the address of the .text section until the end of the file. Lastly, these sequences are analyzed to identify the behavior statically. The applications used in this experiment are implemented by: UPX (Ultimate Packer for eXececutables), .NET assemblies by Microsoft .Net CLI and Mono, and Zlib compression. An opcode is a part of a machine instruction that determines the function to be executed by a machine. Each instruction operated on operands that could be stored in registers, memory, or constants.

The experimental dataset is collected from publicly available sources, comprising 3256[1] malicious files and 744[2] benign samples. The labels are obtained by submitting the files to VirusTotal [106] and using at least five or more antivirus engines to assign a class label. The application would be assigned to the ""`other`" class if it is not in one of the seven categories. Before training sequences by learning models, TextVectorization from Tensorflow [53] is used for preprocessing the opcode sequences.

All missing values are replaced using Scikit-Learn's *SimpleImputer* [54], using the constant strategy with a default *fill_value*. Figure 6.2 shows the percentages of missing values per column. The int sequence mode is set to integer indices, one integer index per token. Also, we set a max_token with 5000 vocabulary sizes and the output sequence length to 500. Then an 'adapt'method is called on a full dataset to create an index on the vocabulary. We put

[1]Downloads from VirusShare [90] and URLhaus [122]

[2]Downloads from FileHorse [108]

each opcode per column for all samples by indexing numbers from 1 to 5000 (n1,n2,...,n5000). The opcode for each column can be any value, which means any operations in the first position. Figure 6.1 shows the system architecture of such an automated data extraction process.

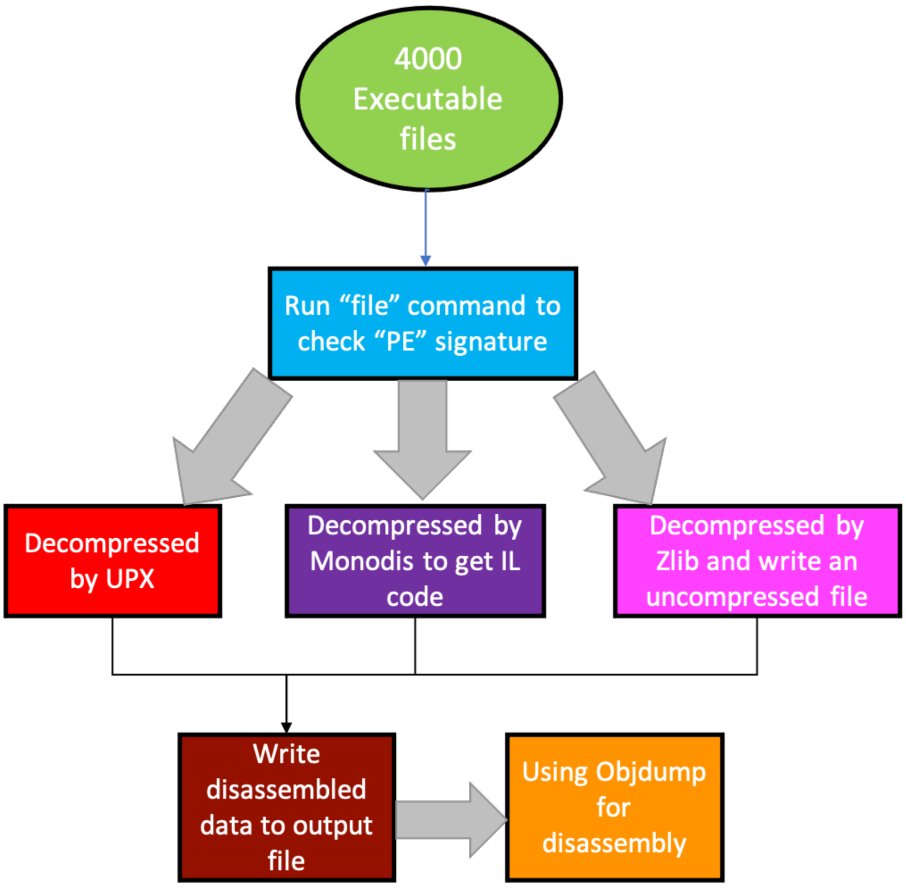

4000 Executable files
Run "file" command to check "PE" signature
Decompressed by UPX
Decompressed by Monodis to get IL code
Decompressed by Zlib and write an uncompressed file
Write disassembled data to output file
Using Objdump for disassembly

Figure 6.1. High-level diagram of our Opcodes Extraction.

Moreover, we present the statistics in terms of the minimum and the maximum number of total samples, grouped by each type. To select the top 20 MLP significant features, SHapley Additive exPlanations (SHAP) [52] is used. The DeepSHAP function from the open-source Python package [109] is applied. First, the learning model is trained on the initial set of features, and each feature's importance is obtained by comparing model predictions with and without the feature on trained data. Each feature's SHAP importance is computed individually by taking the average of the absolute SHAP values across the data and then sorting them in descending order according to their

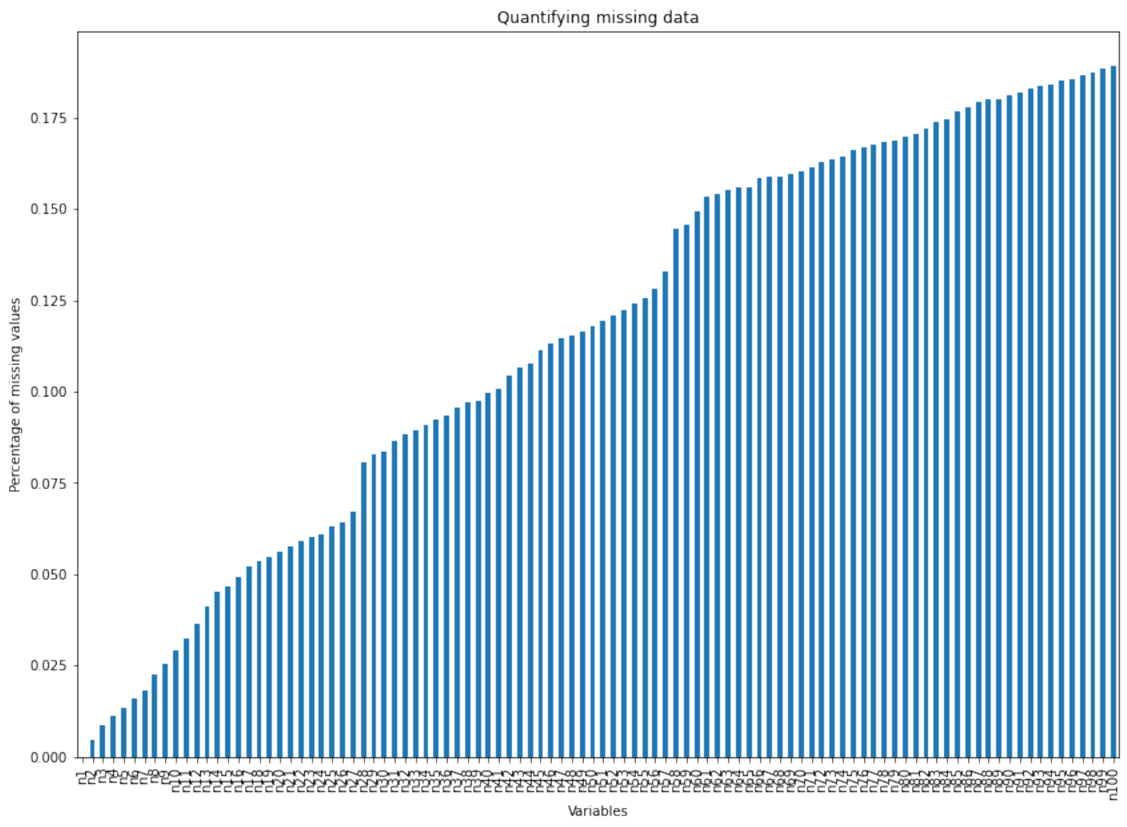


Figure 6.2. The bar plot displays the percentage of missing data continuously increasing by sequence length from 1 to 100.

importance, before the 20 attributes with the most significant impact were chosen. The details are depicted in Table 6.1. For instance, the first selected feature for ransomware is “`n1`”, while “`n68`” was for trojan. The opcodes could be any *mov, push, sub, movsx, jmp, std, daa, dec.*

Figure 6.3 presents a summary plot of the top Shapley values. According to their importance, the “`n1`” feature has the most impact on the model output, and “`n55`” is a shared feature that could be seen across all categories. For “`n7`”, “`n13`”, “`n31`”, “`n36`”, “`n58`”, “`n59`”, “`n63`”, “`n70`”, “`n66`”, “`n78`”, “`n86`”, “`n90`”, “`n95`”, “`n105`”, “`n309`”, “`n399`” features, they are rarely used, particularly the lowest number being in virus class. Additionally, the top 3 opcodes for “`n1`” and “`n55`” features are listed with the total number of occurrences per class in Table 6.2. Overall, “`push`” is discovered in all four families, followed by “`add`” instruction.

Table 6.1. The top 20 features from SHAP selection on opcode sequences for each category with "n1" and "n55" exist across all these categories.

Type	**Minimum/# of samples with 5,000 tokens**	**Total**	**SHAP Selected Features**
Ransom-ware	1/90	194	[**n1**, n95, n3, n44, n115, n134, n50, n23, n5, n120, n10, n4, n152, n105, n40, **n55**, n2, n12, n45, n14]
Trojan	1/426	2570	[n68, n50, n5, n23, n120, **n1**, n152, **n55**, n44, n399, n115, n31, n25, n78, n40, n59, n45, n14, n66, n4]
Miner	4/63	135	[n50, n12, n3, n44, n45, **n1**, **n55**, n4, n2, n56, n115, n14, n5, n309, n8, n13, n11, n17, n33, n15]
Virus	1/26	126	[n291, n230, n170, n131, n3, n25, n2, n4, n408, n141, n209, n29, n115, n32, n411, n93, n30, **n55**, **n1**, n181]
Rootkit	31/7	16	[n44, n45, n50, n58, **n55**, n68, **n1**, n4, n23, n53, n3, n2, n18, n8, n15, n115, n63, n10, n12, n56]
Backdoor	10/64	166	[**n1**, **n55**, n50, n5, n14, n23, n120, n33, n27, n7, n152, n18, n49, n53, n40, n36, n56, n25, n8, n6]
Benign	1/374	741	[n68, n50, n393, **n55**, n33, n44, n40, n14, **n1**, n70, n224, n2, n6, n52, n90, n27, n49, n56, n17, n86]
Other	1/17	52	[**n55**, n50, n393, n33, **n1**, n68, n40, n14, n23, n2, n152, n52, n49, n6, n3, n18, n56, n4, n27, n120]

6.1.2 API calls

In addition to opcodes, API call sequences are the most crucial part of the behavioral-based analysis. In the work of Catak [123], Windows 7 API calls from application execution in a Cuckoo sandbox were used, using 7107 malicious samples collected from various GitHub pages with the git command-line utility. Eight classes were tagged by VirusTotal [106] by searching the MD5 signatures. Five of them consisted of 1001 samples, including Worm, Virus, Trojan, Backdoor, and Downloader, while the other three classes were Spyware (832), Adware (379), and Dropper (891). The study's objective was to build a benchmark dataset for Windows API calls of metamorphic malware.

The changes in code signatures and environment recognition could conceal their malicious behavior through anti-analysis techniques for malware analysis [124]. Although this malware type with such capabilities hinders detection and classification, observing unnecessary API calls could help detect malware because of similar patterns.

The Windows API is an interface for developing applications on the Windows operating system, in which the application could make a call to request operating system services. Catak [123] converted their API calls to grayscale images, generated a 3-channel image representing them for the application, and then used CNN and LSTM as the classification algorithms to detect malware families. The unique 342 API calls were such as *ldrloaddll*, *ldrgetprocedureaddress*, and *regopenkeyexa* as a feature set. Regarding the results, the LSTM network could provide an accuracy of up to 98.5% for multi-class classification. With these public API calls, we perform SHAP selection on the dataset, and the top 20 features are represented in Figure 6.4.

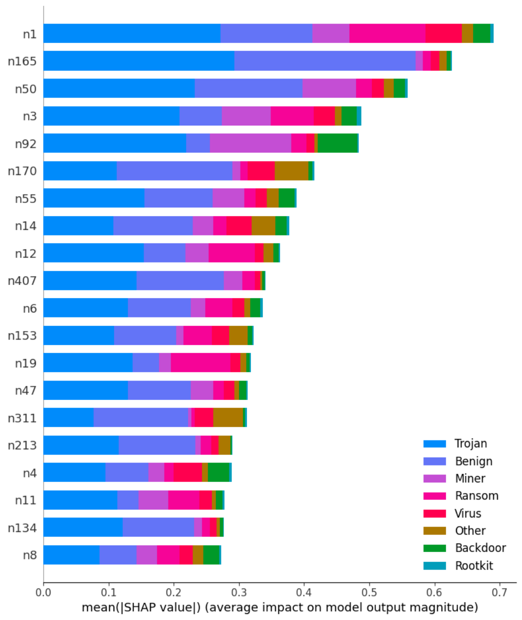


Figure 6.3. SHAP Summary Plot of the top 20 features of opcodes for the MLP.

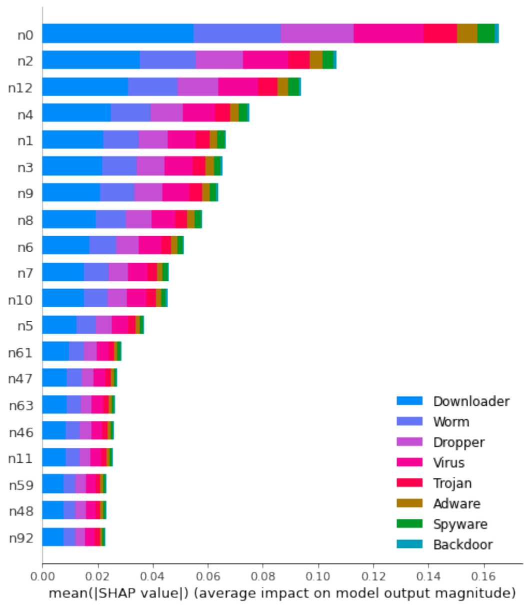


Figure 6.4. SHAP Summary Plot of the top 20 features of APIs for the MLP.

Category	**n1_1**	**n1_2**	**n1_3**	**n55_1**	**n55_2**	**n55_3**
Trojan	add (457)	mov (372)	push (337)	inc (190)	mov (190)	push (177)
Ransom	push (69)	mov (31)	add (23)	add (26)	mov (23)	push (21)
Miner	mov (40)	push (38)	add (8)	push (24)	add (16)	pop (14)
Benign	push (237)	nop (166)	add (134)	mov (207)	idiv (118)	inc (106)

Table 6.2. Top 3 opcodes for ransomware, trojan, miner, and benign class for selected n1 and n55 features.

6.1.3 Combination of Noriben Sequences and Opcode Sequences

We join Noriben sequences derived from chapter 5 with Opcode sequences by matching filenames and obtained 1900 samples with a length of 10000 sequences. The samples are classified as Trojan (945), Benign (722), Ransom (123), Miner (41), Backdoor (24), Virus (11), Rootkit (10), and Other (24). VotingClassifier from Scikit-learn [54] is applied to these sequences by assembling the predictions of multiple classifiers together and then voting on the result to improve the performance of a predictive model. There are two types of voting: (1) Hard voting predicts a class based on the highest majority of votes, and (2) Soft voting predicts a label based on the largest probability averaged by each classifier. In this study, we use soft voting, the average probability of all integrated classifiers to improve the classification performance of integration of Noriben and Opcode sequences, including Random Forest, XGBoost, LSTM, GRU, Transformers, MLP, and a hybrid architecture between GRU and Transformers.

6.2 Model architectures

This section introduces the seven machine learning algorithms employed for malware detection, i.e., RF, XGBoost, MLP, LSTM, GRU, and Transformers. Except for Random Forest and XGBoost, models are implemented using TensorFlow [53]. In order to obtain a good learning model, all hyperparameters are tuned by utilizing Keras Tuner [56] to perform RandomSearch. The learning rate is reduced during the hyperparameter search space when the accuracy metric had stopped improving for ten epochs (patience). The methodology adopted for automating the opcode extraction, analysis, and malicious behavior identification process is described. A 10-fold cross-validation approach is used to prevent overfitting. In the table describing hyperparameter tuning, model parameters are mainly represented in terms of a list. The elements in the list identify the units and regularizers consistent with the number of layers configured for each model. Their respective details are presented in the following subsections.

6.2.1 Random Forest

The Random Forest model is implemented using Scikit-Learn [54]. The model's hyperparameters are tuned using GridSearchCV with 10-fold cross-validation to obtain the desired model. Micro-F1 is used as the objective to maximize the grid search. The eight hyperparameters subject to the grid search, their respective ranges of possible values, and the respective selected values achieving the highest accuracy performance are presented in Table 6.3.

Table 6.3. Hyperparameter tuning for Random Forest Classifier on initial features

Name	**Initial Parameters**	**Tuning Ranges**	**Opcode Best Values**	**APIs Best Values**
n_estimators	200	range(100,650,50)	100	600
criterion	gini	[‘gini’, ‘entropy’]	gini	entropy
max_depth	14	range(6,20,2)	18	16
max_features	sqrt	[‘sqrt’, ‘log2’]	sqrt	sqrt
bootstrap	True	[True, False]	True	True
min_samples_leaf	2	range(2,12,2)	4	4
min_samples_split	2	range(2,12,2)	10	6
max_leaf_nodes	30	range(15,55,5)	50	50

6.2.2 XGBoost

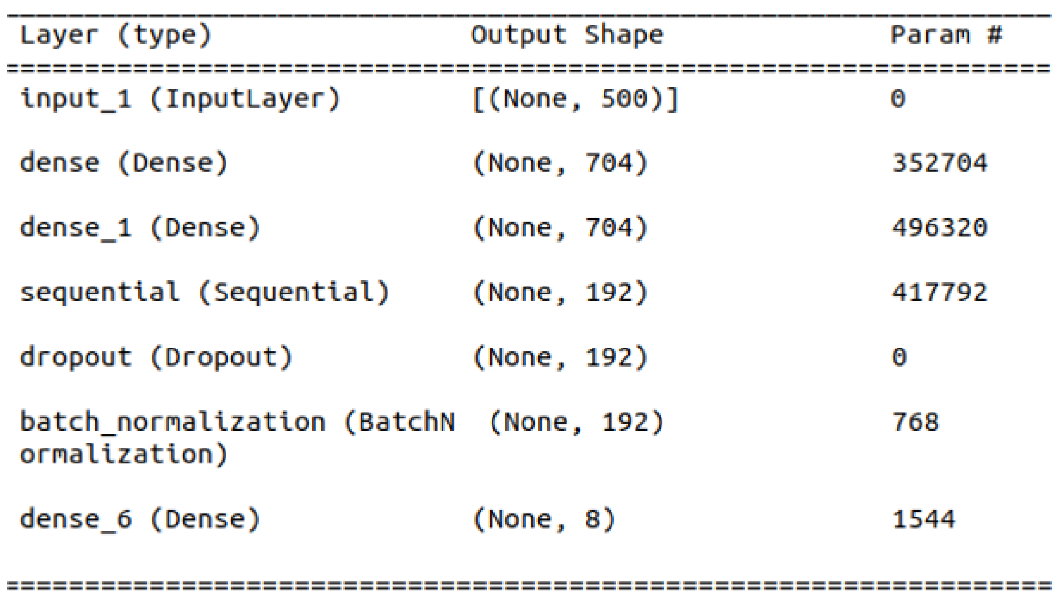
The XGBoost model is implemented using the XGBClassifier from Scikit-Learn [54]. A grid search from Ray Tune [111] is performed to determine a well-performing set of values for the hyperparameter search space. This search uses 'mlogloss' (default for multiclass problems) and 'AUC' as the evaluation metrics to measure the classification performance. Table 6.4 presents the five hyperparameters, their ranges of possible values, and the selected best-performing values.

Table 6.4. Hyperparameter tuning for XGBoost Classifier on initial features

Name	**Initial Parameters**	**Value**	**Opcode Best Values**	**APIs Best Values**
n_estimators	200	range(100,650,50)	250	450
max_depth	6	range(4,20,2)	14	16
learning_rate/eta	0.001	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.02	0.02
gamma	1.5	[0.8,1.0,1.5,2.0, 2.5,3.0,4.0,5.0]	1.5	3.0
min_child_weight	1	range(1,6,2)	1	1

6.2.3 MLP

We employ nonlinear activations, BatchNormalization, and regularizers to avoid overfitting for the network architecture. First, a shape of sequences is provided to an input layer, followed by two dense layers with dropout and BatchNormalization layers. Then, consecutive multi-dense layers are inserted before the final output layer, which consists of eight neurons densely connected to the last batch normalization layer. In the network configuration, "swish" is an activation function for all layers. The model loss is calculated based on categorical cross-entropy. The accuracy and f1_score are computed as the evaluation. To optimize hyperparameters, the model is fine-tuned as provided in Table 6.5, and the best model obtained by RandomSearch is depicted in Figure 6.5.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 500)]             0

 dense (Dense)               (None, 704)               352704

 dense_1 (Dense)             (None, 704)               496320

 sequential (Sequential)     (None, 192)               417792

 dropout (Dropout)           (None, 192)               0

 batch_normalization (BatchN  (None, 192)              768
 ormalization)

 dense_6 (Dense)             (None, 8)                 1544

=================================================================
```

Figure 6.5. The best MLP as determined by RandomSearch in Tensorflow.

6.2.4 GRU and LSTM

GRU [62] and LSTM [118] are initiated with an embedding layer, followed by a convolutional layer and a max-pooling layer. The number of filters in the convolutional layer is 256, and the kernel size was 4 × 4 with no padding. Furthermore, Dropout is added between two bi-directional[3] CuDNNGRU and CuDNNLSTM to prevent overfitting and a softmax layer is used to output the malware family label. Figure 6.6 provides an overview of bidirectional GRU and LSTM used for multi-class malware classification. The information about hyperparameters, their ranges of possible values, and their selected values from the RandomSearch is provided in Table 6.6, and the best model is displayed in Figure 6.7.

6.2.5 Transformers

Transformers [55] is an attention-based encoder-decoder architecture focused on different tokens while generating words to model opcode sequences. The standard Transformers architecture described in [55] is applied for model configuration, and the setup by the RandomSearch with the selected best values is listed in Table 6.7.

Table 6.5. Hyperparameter tuning for MLP for Opcodes and API calls

Name	**Initial Value**	**Tuning Ranges**	**Opcode Best Value**	**APIs Best Value**
batch_size	15	ranges(10,50,5)	50	30
epochs	200	ranges(100,600,50)	200	500
optimizer	RMSprop	[‘Adam’,‘RMSprop’, ‘Adadelta’,‘Adagrad’]	Adam	RMSprop
learning_rate	0.0002	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.02	0.002
First Two Dense Layers	[512,256]	range(256,1024,32)	[704,704]	[768,800]
Hidden Layers in Sequential	5	range(2,10)	4	6
Hidden Neurons in Sequential	[256,256,256, 256,256]	range(128,512,32)	[320,224,288,192]	[448,256,480, 320,192,320]
bias_regularizer	[0.002, 0.001, 0.02]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.001,0.01, 0.002]	[0.02,0.0001, 0.002]
kernel_regularizer	[0.01,0.01, 0.02]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.02,0.01, 0.0001]	[0.001,0.0002, 0.0002]
kernel_constraint	[2.0,1.5,1.5]	range(1,3,0.5)	[2.5,1.5,2.0]	[1.0,3.0,3.0]
activity_regularizer	[0.0001, 0.001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0001, 0.02, 0.0001]	[0.0001,0.0001, 0.0001]

6.2.6 Combination of GRU and Transformers

This classifier is created as a combination of a GRU and a Transformers network. It is a concatenation of the outputs of the last BatchNormalization layer of the GRU and the final GlobalAveragePooling1D layer of the Transformers and passed to a final softmax layer. The tuning information for this combination is provided in Table 6.8.

6.3 Experimental Results

This section presents the results of our experiments for malware detection using opcode instructions for seven learning approaches. The evaluation metrics

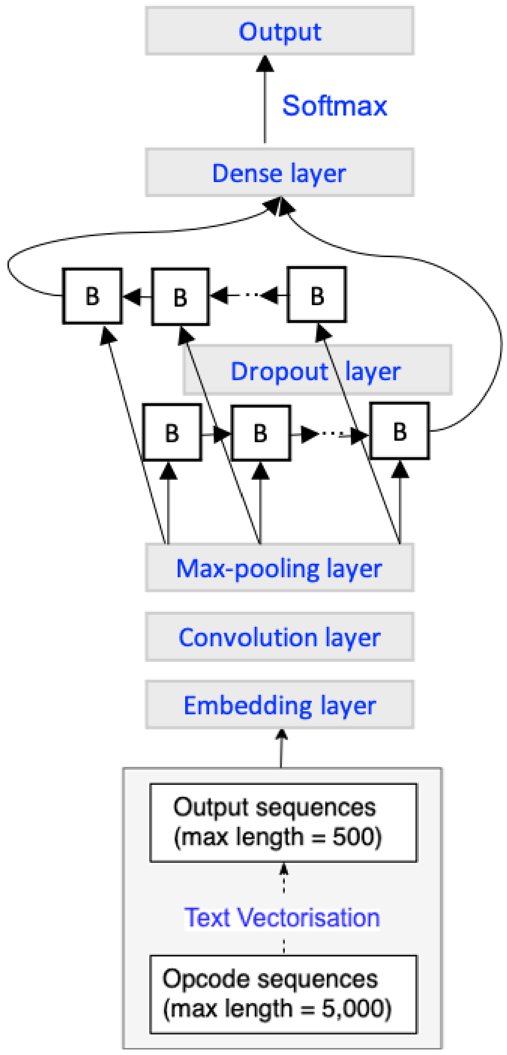
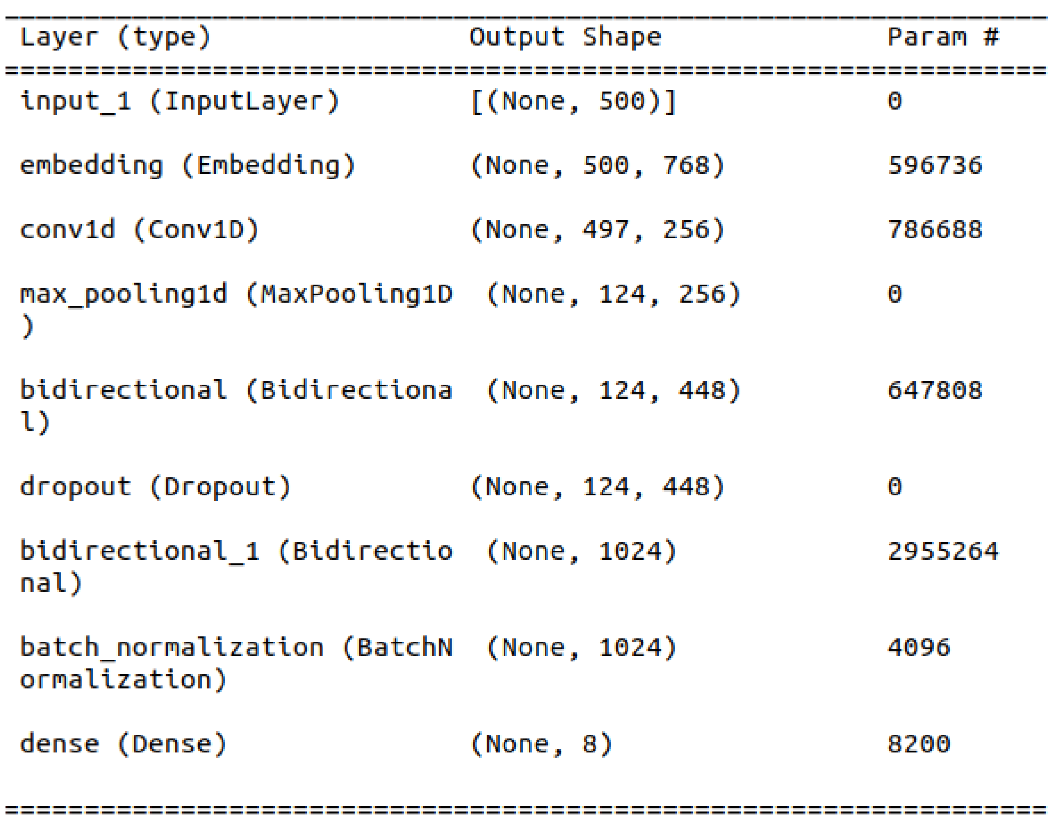


Figure 6.6. Flowchart for GRU and LSTM

are accuracy, F1-measure, Mean Absolute Error (MAE), and Area Under the receiver operating characteristic Curve (AUC) to measure the model performance.

Table 6.9 lists results for running all algorithms with their default parameter settings on static opcodes. Overall, XGB performed the best for all classifiers by achieving an accuracy of 74%. In contrast, Transformers provided the outstanding predictive result for dynamic API calls [125] with default model parameters as shown in Table 6.10. In addition to opcode and API sequences, the merging of Noriben and opcode sequences with their default parameters is presented in Table 6.13. XGB again yielded the best result, similar to Table 6.9.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 500)]             0

 embedding (Embedding)       (None, 500, 768)          596736

 conv1d (Conv1D)             (None, 497, 256)          786688

 max_pooling1d (MaxPooling1D  (None, 124, 256)         0
 )

 bidirectional (Bidirectiona  (None, 124, 448)         647808
 l)

 dropout (Dropout)           (None, 124, 448)          0

 bidirectional_1 (Bidirectio  (None, 1024)             2955264
 nal)

 batch_normalization (BatchN  (None, 1024)             4096
 ormalization)

 dense (Dense)               (None, 8)                 8200

=================================================================
```

Figure 6.7. The best GRU as determined by Random-Search in Tensorflow.

Table 6.11 shows the improvements achieved by adequately tuning the essential hyperparameters. The overall performance increased for all classification models compared to the default performances from Table 6.9. With the same model configuration as the default parameters, the GRU exhibited the best performance results by reaching the maximum accuracy at 89%. The figure represented the correct class, but the absolute error was 0.036. So a low MAE implied higher average confidence.

In Table 6.12, hyperparameter tuning is conducted on dynamic API dataset [125] on the same architecture as the static opcodes. In general, the performance of all model approaches was obviously increased compared to Table 6.10. The GRU could provide the best predictive result among other classifiers by achieving an F1-score of 78%.

In addition, model parameters are tuned on the merged dataset. As shown in Table 6.14, GRU produced the best result compared to other algorithms. Moreover, this learning approach was the best classifier when trained on opcode

Table 6.6. GRU and LSTM hyperparameter tuning for API calls and Opcodes.

Name	Initial Value	Tuning Ranges	API calls (Best)		Opcodes (Best)	
			GRU	LSTM	GRU	LSTM
batch size	15	range(10,50,5)	25	20	30	50
epochs	200	range(100,650,50)	350	500	550	200
optimizer	RMSprop	[‘Adam’,‘RMSprop’, ‘Adadelta’, ‘Adagrad’]	Adam	Adadelta	Adam	RMSprop
learning rate	0.0001	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.0002	0.02	0.0001	0.0001
embedding	128	ranges(256,1024,64)	448	640	768	704
embedding regularizer	0.0001	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.0001	0.0001	0.0002	0.02
gru/lstm units	[512, 256]	ranges(128,512,32)	[416, 288]	[288, 384]	[224, 512]	[288, 448]
recurrent regularizer	[0.0001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.01, 0.01]	[0.0001, 0.002]	[0.02, 0.0001]	[0.002, 0.0002]
kernel regularizer	[0.001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]]	[0.0002, 0.002]	[0.0001, 0.02]	[0.001, 0.0002]	[0.01, 0.02]
bias regularizer	[0.01, 0.02]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.02, 0.01]	[0.002, 0.001]	[0.01, 0.01]	[0.0002, 0.02]
kernel constraint	[2.0, 1.5]	range(1,3,0.5)	[1.5, 2.0]	[2.0, 3.0]	[1.5, 1.5]	[1.5, 1.0]
activity regularizer	[0.0001, 0.0002]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.002, 0.02]	[0.001, 0.0001]	[0.01, 0.0002]	[0.01, 0.01]

sequences alone. However, we ran an additional experiment using a soft voting mechanism by integrating all classification models: RF, XGB, LSTM, GRU, Transformers, MLP, and concatenation GRU with Transformers, and could obtain better predictive performance with the highest accuracy at 96%.

We also analyze the detailed results of the classification report obtained by the hyperparameters on static opcodes, dynamic API calls, and a combination of Noriben and opcodes with soft voting. Their values are shown in Table 6.15. Remarkably, precision and recall on opcodes and a merged dataset were more than 90% for Trojan. Nevertheless, Rootkit had the maximum standard deviation compared to other categories, which meant data were more spread

Table 6.7. Hyperparameter tuning for Transformers for Opcodes and API calls

Name	Initial Value	Tuning Ranges	Opcode Best Value	APIs Best Value
batch_size	10	range(10,30,5)	15	15
epochs	100	range(100,600,50)	250	250
optimizer	Adam	[‘Adam’,‘RMSprop’, ‘Adadelta’,‘Adagrad’]	Adam	Adam
learning_rate	0.0001	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.0002	0.0002
trans_dim	512	ranges(512,2048,32)	832	640
num_heads	8	ranges(6,14,2)	12	12
feed_forward_dim	512	ranges(1024,3172,32)	1536	1408
enc_layer	5	ranges(2,10)	6	6
dec_layer	5	ranges(1,10)	4	4
embedding regularizer	[0.001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0002, 0.0002]	[0.0002, 0.0002]
kernel regularizer	[0.001, 0.001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.01, 0.002, 0.0002]	[0.01, 0.002, 0.0002]
bias regularizer	[0.01, 0.02, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.001, 0.001, 0.0002]	[0.001, 0.001, 0.0002]
kernel constraint	[2.0, 1.5, 1.0]	range(1,3,0.5)	[1.5, 2.5, 2.5]	[1.5, 2.5, 2.5]
activity regularizer	[0.001, 0.001, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0001, 0.01, 0.01]	[0.0001, 0.01, 0.01]

out in the dataset. Downloader had high precision and recall for API calls, and high standard deviation could be observed for Dropper and Spyware.

In addition to the reports of the evaluation metrics, we present the total elapsed time on the training default configuration and hyperparameter tuning with ten maximum trials for each classifier. We use stratified 10-Folds cross-validation using the GPU. Although RandomSearch considered not all possible combinations, Transformers were the most prolonged hours for optimization, as represented in Table 6.16.

Finally, Figure 6.8 is the confusion matrix that compared the predicted and expected values. The classification errors were less than 2% except for “Trojan” and “Other” class. We also generate ROC curves on opcodes as in

Table 6.8. Hyperparameter search space to tune the combining GRU and Transformers model.

Name	**Initial Value**	**Tuning Ranges**	**Best Value (Noriben+Opcode)**
batch_size	15	ranges(10,50,5)	15
epochs	100	ranges(100,600,50)	400
optimizer	RMSprop	[‘Adam’,‘RMSprop’, ‘Adadelta’,‘Adagrad’]	RMSprop
learning_rate	0.002	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	0.0002
embedding	256	range(256,1024,32)	800
embedding regularizer	[0.001,0.001,0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0002,0.002,0.02]
gru units	[512, 256]	ranges(128,512,32)	[320, 480]
recurrent regularizer	[0.0001,0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.0002,0.01]
kernel regularizer	[0.001,0.001,0.001, 0.001,0.01]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.01, 0.001, 0.001, 0.02, 0.0002]
bias regularizer	[0.01,0.02,0.01,0.01, 0.001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.02,0.02,0.0001, 0.01,0.002]
kernel constraint	[2.0, 1.5, 1.5, 2.0, 1.0]	range(1,3,0.5)	[2.5, 1.0, 2.5, 3.0, 2.5]
activity regularizer	[0.001,0.001,0.001, 0.001,0.0001]	[0.01, 0.02, 0.001, 0.002, 0.0001, 0.0002]	[0.002,0.0002,0.0001, 0.01,0.002]
trans_dim	128	ranges(512,1024,32)	704
num_heads	4	ranges(6,20,2)	8
feed_forward_dim	1024	ranges(1024,3172,128)	2176
enc_layer	3	ranges(2,10)	7
dec_layer	2	ranges(1,10)	2

Figure 6.9. It showed that all categories could yield an AUC value of almost 90%.

Table 6.9. Model Evaluation using default hyperparameter settings for opcodes

Algorithm	Avg. Test Acc	Avg. F1 Test	Avg. MAE	Avg.AUC
RF	0.736±0.01	0.662±0.01	0.066 ±0.01	0.641±0.01
XGB	**0.748±0.01**	0.706±0.01	**0.063±0.03**	0.697±0.01
LSTM	0.723±0.03	0.723±0.03	0.081±0.01	**0.921±0.01**
GRU	0.709±0.04	0.708±0.05	0.085±0.01	0.918±0.01
Transformers	0.731±0.02	**0.729±0.02**	0.081±0.01	0.918±0.01
MLP	0.615±0.02	0.602±0.03	0.123±0.01	0.878±0.01

Table 6.10. Model Evaluation using default hyperparameter settings for API calls [125]

Algorithm	Avg. Test Acc	Avg. F1 Test	Avg. MAE	Avg.AUC
RF	0.292±0.01	0.250±0.01	0.177±0.01	0.589±0.01
XGB	0.339±0.01	0.320±0.01	0.165±0.03	0.617±0.01
LSTM	0.449±0.02	0.461±0.02	0.146±0.01	0.802±0.01
GRU	0.426±0.02	0.444±0.03	0.148±0.01	0.775±0.01
Transformers	**0.461±0.02**	**0.466±0.02**	**0.143±0.01**	**0.813±0.01**
MLP	0.278±0.02	0.227±0.04	0.191±0.01	0.661±0.01

Table 6.11. Model Evaluation with average 10 fold cross-validation on hyperparameter tuning for opcodes

Algorithm	Avg. Test Acc	Avg. F1 Test	Avg. MAE	Avg.AUC
RF	0.747±0.01	0.684±0.01	0.063±0.01	0.664±0.01
XGB	0.745±0.01	0.709±0.01	0.064±0.03	0.705±0.01
LSTM	0.858±0.04	0.856±0.04	0.054±0.01	0.977±0.02
GRU	**0.891±0.05**	**0.890±0.05**	**0.036±0.01**	**0.984±0.02**
Transformers	0.818±0.03	0.807±0.03	0.077±0.01	0.971±0.02
MLP	0.765±0.02	0.768±0.02	0.087±0.01	0.942±0.01

Table 6.12. Model Evaluation with average 10 fold cross-validation on hyperparameter tuning on API calls [125]

Algorithm	Avg. Test Acc	Avg. F1 Test	Avg. MAE	Avg.AUC
RF	0.318±0.01	0.280±0.01	0.171±0.01	0.604±0.01
XGB	0.389±0.02	0.377±0.02	0.153±0.01	0.647±0.01
LSTM	0.635±0.06	0.642±0.07	0.117±0.01	0.915±0.04
GRU	**0.736±0.09**	**0.783±0.11**	**0.075±0.02**	**0.945±0.05**
Transformers	0.626±0.09	0.691±0.10	0.102±0.02	0.905±0.05
MLP	0.681±0.13	0.723±0.15	0.089±0.03	0.917±0.09

6.4 Discussion

We used open-source tools according to different compression methods to extract opcode sequences from a Windows PE file and then used developed models to predict categories. We initialized the extraction process by unpacking such PE files and disassembling them. From the results of our experiments, the GRU was the best classifier for malware classification, using only static

Table 6.13. Model Evaluation using default hyperparameter settings for a combination of Noriben and Opcode Sequences

Algorithm	Avg. Test Acc	Avg. F1 Test	Avg. MAE	Avg.AUC
RF	0.770±0.02	0.721±0.02	0.057±0.01	0.794±0.01
XGB	**0.803±0.03**	**0.784±0.03**	**0.049±0.01**	0.831±0.02
LSTM	0.716±0.03	0.718±0.02	0.075±0.01	0.931±0.01
GRU	0.733±0.03	0.732±0.03	0.070±0.01	0.940±0.01
Transformers	0.665±0.07	0.666±0.07	0.089±0.02	0.903±0.02
MLP	0.739±0.03	0.725±0.05	0.086±0.01	0.944±0.01
GRU+Trans	0.749±0.02	0.750±0.01	0.067±0.01	**0.945±0.01**

Table 6.14. Model Evaluation with average 10 fold cross-validation on hyperparameter tuning on a combination of Noriben and Opcode Sequences

Algorithm	Avg. Test Acc	Avg. F1 Test	Avg. MAE	Avg.AUC
RF	0.779±0.02	0.735±0.02	0.055±0.01	0.801±0.01
XGB	0.807±0.03	0.790±0.03	0.048±0.01	0.837±0.03
LSTM	0.950±0.07	0.947±0.08	0.016±0.02	0.991±0.02
GRU	0.952±0.06	0.951±0.06	0.017±0.02	0.991±0.01
Transformers	0.948±0.06	0.948±0.06	0.017±0.02	0.991±0.01
MLP	0.899±0.06	0.901±0.06	0.038±0.01	0.982±0.02
GRU+Trans	0.949±0.07	0.948±0.07	0.015±0.02	**0.993±0.01**
Soft Voting	**0.966±0.06**	**0.966±0.06**	**0.009±0.01**	0.981±0.03

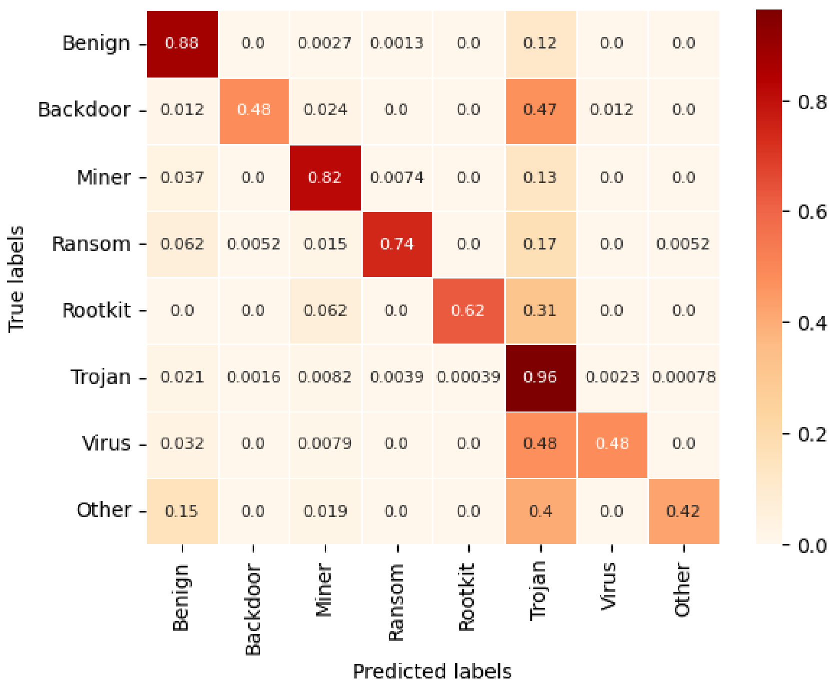
opcodes and dynamic API calls and a combination of static opcodes with dynamic Noriben sequences. We also compared opcodes and API calls using the same model parameters as default and performing hyperparameter tuning. The predictive result of APIs was lower than opcodes for all algorithms; even though the samples of public dynamic API calls were greater than opcodes, they were similar to the imbalanced dataset. Furthermore, we joined Noriben sequences as described in chapter 5 with opcode sequences. Sequence models (LSTM, GRU, and Transformers) could achieve higher evaluation metrics than Random Forest and XGBoost, at approximately 95%. An increase in performance for the combined dataset was the highest by integrating all classifiers and then using a soft voting mechanism to average the predictions. However, due to programming languages and advanced metamorphic and polymorphic malware techniques, opcodes may not be extracted by open-source disassem-

Table 6.15. Precision, recall, and F1 score per class for 10-fold cross-validation of opcodes and API calls using GRU and a soft voting classifier for a combined Noriben and Opcode.

	Static Opcodes				Dynamic API calls				Combined Noriben and Opcode		
	Precision	Recall	f1_score		Precision	Recall	f1_score		Precision	Recall	f1_score
Benign	0.90±0.04	0.87±0.07	0.88±0.04	Adware	0.99±0.02	0.67±0.07	0.80±0.06	Benign	0.97±0.05	0.97±0.04	0.97±0.05
Backdoor	0.81±0.05	0.69±0.16	0.74±0.10	Backdoor	0.92±0.12	0.68±0.10	0.78±0.10	Backdoor	0.96±0.07	1.00±0.00	0.98±0.04
Miner	0.80±0.17	0.81±0.14	0.80±0.14	Downloader	0.94±0.08	0.82±0.10	0.88±0.09	Miner	0.97±0.15	0.85±0.24	0.91±0.21
Ransom	0.92±0.25	0.74±0.23	0.82±0.23	Dropper	0.94±0.11	0.71±0.14	0.81±0.13	Ransom	0.97±0.09	0.90±0.24	0.94±0.21
Rootkit	0.90±0.46	0.56±0.33	0.69±0.36	Spyware	0.88±0.16	0.64±0.11	0.74±0.12	Rootkit	1.00±0.40	0.80±0.40	0.89±0.40
Trojan	0.90±0.04	0.95±0.03	0.93±0.03	Trojan	0.90±0.13	0.66±0.10	0.76±0.11	Trojan	0.97±0.05	0.99±0.02	0.98±0.04
Virus	0.88±0.27	0.48±0.18	0.62±0.20	Worm	0.96±0.06	0.64±0.09	0.77±0.08	Virus	0.64±0.32	0.82±0.32	0.72±0.31
Other	0.88±0.40	0.42±0.12	0.57±0.18	Virus	0.92±0.14	0.61±0.10	0.73±0.11	Other	1.00±0.30	0.71±0.31	0.83±0.29

Table 6.16. Training time for each algorithm on Static Opcodes and Dynamic API calls

Algorithm	Default Opcode Training	Opcode Tuning	No.of tuning parameters	Default API Training	API Tuning	No.of tuning parameters	Default Nori+OPS Training	Nori+OPS Tuning	No.of tuning parameters
RF	28m	43h:32m	8 (grid)	1m	53h:25m	8 (grid)	2m	49h:50m	8 (grid)
XGB	16m	7h:29m	5 (grid)	1m	17h:19m	5 (grid)	4m	14h:47m	5 (grid)
LSTM	3h:20m	4h:52m	12 (random)	17h:48m	20h:16m	12 (random)	2h:24m	3h:39m	12 (random)
GRU	3h:04m	4h:39m	12 (random)	17h:25m	19h:48m	12 (random)	2h:22m	3h:24m	12 (random)
Transformers	35h:41m	47h:30m	14 (random)	27h:37m	44h:41m	14 (random)	13h:10m	52h:48m	14 (random)
MLP	1h:08m	2h:49m	11 (random)	2h:25m	4h:18m	11 (random)	32m	1h:29m	11 (random)



	Benign	Backdoor	Miner	Ransom	Rootkit	Trojan	Virus	Other
Benign	0.88	0.0	0.0027	0.0013	0.0	0.12	0.0	0.0
Backdoor	0.012	0.48	0.024	0.0	0.0	0.47	0.012	0.0
Miner	0.037	0.0	0.82	0.0074	0.0	0.13	0.0	0.0
Ransom	0.062	0.0052	0.015	0.74	0.0	0.17	0.0	0.0052
Rootkit	0.0	0.0	0.062	0.0	0.62	0.31	0.0	0.0
Trojan	0.021	0.0016	0.0082	0.0039	0.00039	0.96	0.0023	0.00078
Virus	0.032	0.0	0.0079	0.0	0.0	0.48	0.48	0.0
Other	0.15	0.0	0.019	0.0	0.0	0.4	0.0	0.42

Figure 6.8. Confusion matrix among different malware families for opcodes with a GRU

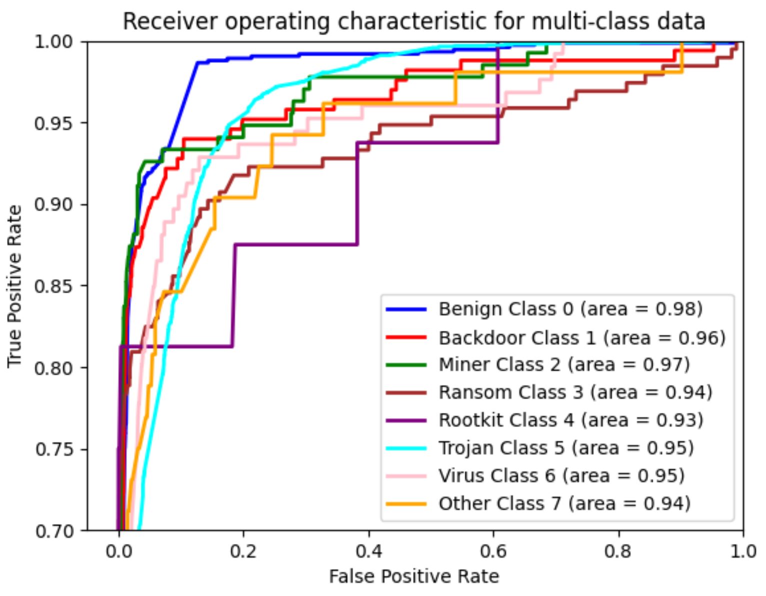


Figure 6.9. ROC Curve for a GRU on opcodes

bling tools. Some codes could be missing, and some parts were not interpreted as functions, so decompiling tools could not detect them. Junk codes appended in the code section or added in the packer could render the decompiler as broken. Also, encryption application code could be an obstacle to static analysis. Consequently, the obtained opcodes could be incomplete data and contain false positives. Our analysis used consecutive static opcode sequences only to classify categories and not include discovering malware patterns. To be more specific, the top three opcodes from feature selection indicated each type's frequency of occurrence from our samples.

Chapter 7

Conclusion

This thesis began by providing information about utilizing machine learning techniques in malware and considered predicting labels as a binary and multi-class classification problem. The main contributions are using data collection from static and dynamic analysis to characterize and predict class labels by preprocessing data to create feature sets and benchmark different machine learning approaches using only static and dynamic features, their combination, and sequences. We conducted several experiments to achieve two main goals:

- Improving standard classifiers for predicting classes from static and dynamic artifacts as a binary classification problem.

- Improving the accuracy of predictions of malware categories for long Noriben and Opcode sequences.

Regarding the first point, we collected data from static and dynamic analysis and arranged them into three datasets: (1) static data, (2) dynamic data, and (3) combined data. We used a Python "`pefile`" [30] library to carve out static attributes and acquire dynamic data from HybridAnalysis [47] augmented with memory analysis. Moreover, SHAP feature selection [52] is applied to a full set of static and dynamic attributes. All three datasets used two experiments–with and without the SHAP selection method. A neural network was the best classifier that provided accuracy at approximately 97% for both experiments. Combining static and dynamic artifacts could improve classification performance instead of providing only static and dynamic datasets as model inputs. For feature selection, the predictive result was rather close to using the full set of the dynamic and combined datasets.

We had two datasets to achieve the second goal: (1) static opcodes and (2) dynamic Noriben sequences. On the one hand, three compression algorithms were considered to extract information from static data. On the other hand, Noriben [51] script was placed on the victim machine to capture application activities and extract sequences from log files. The aim was to categorize these sequences into eight classes. Regarding the results, a GRU outperformed the other algorithms for the first dataset, while a hybrid approach between GRU and Transformers was the best for the second dataset.

7.1 Future Research Directions

The research presented here suggests a number of directions for future works. This section discusses some limitations of the thesis and possible research extensions. There are still some things that need to be addressed in future work. Most importantly, some ransomware applications encrypted Noriben's output during runtime, making inaccessible application operations. To obtain a comprehensive view of all malicious functionality, examine network connection properties and contents to help discover how it is spreading, such as the C2 mechanics or downloader details, and payload delivery would need to be added for investigation. We also intend to collect more malware samples to apply our methods to a larger dataset to verify model effectiveness. The directions of future works can be described below.

- **Multi-Label classification**: A sample may belong to multiple labels when making predictions instead of one class during the classification task. For example, one sample can belong to trojan and spyware. Multi-labels could be assigned to a given malware in a malware category, such as using common antivirus engines by uploading binaries and obtaining different labels as return values. Different engines could provide distinct labels. For example, Trojan's family could be named Emolet, Multi.Accesstr, and Downloader. One malware could have similar characteristics to others in terms of causing damage to the system or way of infection. That means one sample could be associated with several families. However, providing accurate label outputs requires sophisticated knowledge from various security engines and applying learning approaches to categorize them.

- **Graph Neural Network**: As a sequence extracted from binaries, we could use these sequences to observe behavioral graphs. In the work of [114], the DGCNN architecture can be replicated to apply to our multiclass malware classification. Based on many open-source libraries for graph neural networks, such as the Spektral Python library [126], GNNs [127] could be built to perform graph classification to make predictions at the level of nodes, edges, or entire graphs. GNNs could discover insights in a network without creating false connections by analyzing relationships between features (nodes). A feature matrix (N*d) is formed by stacking feature vectors to create nodes. The value of "`N`" represented the number of rows (samples), while the value of "`d`" represented the number of features. One-hot encoding replaced categorical features with binary values to generate an adjacency matrix (edges) and then generated an undirected graph.

- **Reinforcement Learning**: In general, reinforcement learning [128] is a framework for sequential decision-making (e.g. gameplay, machine translation) and learning from successive experiments. However, reinforcement learning could be applied to classification problems by enabling an agent to take suitable actions in an environment to maximize cumulative rewards. For instance, Q-Learning [129] is a model-free reinforcement learning algorithm that used the current state to make predictions. It estimated an action taken in a state directly from the interaction between the agent and the environment. To be more specific, the environment could be customized and allowed the agent to learn and solve the problem. A comparison between action from the epsilon-greedy policy [130] and the expected action is conducted to obtain the reward for the current state. The reward is given to one when correctly predicting the class (identical between predicted and expected value); otherwise, a minus one is assigned for failure (unidentical between predicted and expected value). So, the agent would take a series of actions to achieve the maximum cumulative reward.

- **Time Series Forecasting**: We could expand collecting data samples by time series and use them as inputs for sequence models. Given time-based data, machine learning could analyze and generate good forecasts. For the static analysis, the binaries could be sorted by creation time (timestamp) or file modification time by using ExifTool [98] to extract file metadata and then flag it as malware. At the same time, the data could be a sequence of command lines and scripts extracted from Windows Event log entries for dynamic analysis.

References

[1] Ken Kizzee. Cyber attack statistics to know in 2023, Apr 2023. `https://parachute.cloud/cyber-attack-statistics-data-and-trends/`.

[2] Muchammad Naseer, Jack Febrian Rusdi, Nuruddeen Musa Shanono, Sazilah Salam, Zulkiflee Bin Muslim, Nur Azman Abu, and Iwan Abadi. Malware detection: issues and challenges. In *Journal of Physics: Conference Series*, volume 1807, page 012011. IOP Publishing, 2021.

[3] Ferhat Ozgur Catak, Ahmet Faruk Yazı, Ogerta Elezaj, and Javed Ahmed. Deep learning based sequential model for malware analysis using windows exe api calls. *PeerJ Computer Science*, 6:e285, 2020.

[4] Danish Vasan, Mamoun Alazab, Sobia Wassan, Hamad Naeem, Babak Safaei, and Qin Zheng. Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks*, 171:107138, 2020.

[5] Miles Q Li, Benjamin CM Fung, Philippe Charland, and Steven HH Ding. I-mad: Interpretable malware detector using galaxy transformer. *Computers & Security*, 108:102371, 2021.

[6] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* No Starch Press, USA, 1st edition, 2012.

[7] Prabhu Seshagiri, Anu Vazhayil, and Padmamala Sriram. Ama: static code analysis of web page for the detection of malicious scripts. *Procedia Computer Science*, 93:768–773, 2016.

[8] Doaa Wael, Samir G Sayed, and Nashwa AbdelBaki. Enhanced approach to detect malicious vbscript files based on data mining techniques. *Procedia Computer Science*, 141:552–558, 2018.

[9] Peyman Khodamoradi, Mahmood Fazlali, Farhad Mardukhi, and Masoud Nosrati. Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms. In *2015*

18th CSI International Symposium on Computer Architecture and Digital Systems (CADS), pages 1–6. IEEE, 2015.

[10] Alireza Khalilian, Amir Nourazar, Mojtaba Vahidi-Asl, and Hassan Haghighi. G3md: Mining frequent opcode sub-graphs for metamorphic malware detection of existing families. *Expert Systems with Applications*, 112:15–33, 2018.

[11] James B Fraley and Marco Figueroa. Polymorphic malware detection using topological feature extraction with data mining. In *SoutheastCon 2016*, pages 1–7. IEEE, 2016.

[12] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13:1–12, 2017.

[13] Favio Vázquez. The data fabric for machine learning. part 1., Jan 2019.

[14] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil DB Bruce, Yang Wang, and Farkhund Iqbal. Malware classification with deep convolutional neural networks. In *2018 9th IFIP international conference on new technologies, mobility and security (NTMS)*, pages 1–5. IEEE, 2018.

[15] 5fingers Yuval Nativ, Lahad Ludar. thezoo - a live malware repository, Jan 2015.

[16] Richard Harang and Ethan M Rudd. Sorel-20m: A large scale benchmark dataset for malicious pe detection. *arXiv preprint arXiv:2012.07634*, 2020.

[17] NetMarketShare. Net marketshare: Operating system market share, Jun 2022.

[18] Paloalto. Cryptominers.

[19] Cybersecurity and Infrastructure Security Agency (CISA). Protecting sensitive and personal information from ransomware-caused data breaches, 2021.

[20] Veracode. Rootkit: What is a rootkit?

[21] Malwarebytes. Trojan horse – virus or malware?

[22] Malak Abdullah I Almarshad, Mohssen MZE Mohammed, and Al-Sakib Khan Pathan. Detecting zero-day polymorphic worms with jaccard similarity algorithm. *International Journal of Communication Networks and Information Security*, 8(3):203, 2016.

[23] TechTarget. backdoor (computing).

[24] Kyoung Soo Han, Jae Hyun Lim, Boojoong Kang, and Eul Gyu Im. Malware analysis using visualized images and entropy graphs. *International Journal of Information Security*, 14(1):1–14, 2015.

[25] Aziz Makandar and Anita Patrot. Malware analysis and classification using artificial neural network. In *2015 International conference on trends in automation, communications and computing technology (I-TACT-15)*, pages 1–6. IEEE, 2015.

[26] Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen. Amal: high-fidelity, behavior-based automated malware analysis and classification. *computers & security*, 52:251–266, 2015.

[27] Reinaldo Jose Mangialardo and Julio Cesar Duarte. Integrating static and dynamic malware analysis using machine learning. *IEEE Latin America Transactions*, 13(9):3080–3087, 2015.

[28] Matt Pietrek. An in-depth look into the win32 portable executable file format, part 2. *MSDN Magazine, March*, 2002.

[29] M Zubair Shafiq, S Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *International workshop on recent advances in intrusion detection*, pages 121–141. Springer, 2009.

[30] Ero Carrera. Python pe parsing module, May 2022.

[31] Ajit Kumar. *A framework for malware detection with static features using machine learning algorithms*. PhD thesis, Department of Computer Science, Pondicherry University, 2017.

[32] Hyungjoon Koo (Kevin). Kevin's attic for security research. `https://dandylife.net/blog/archives/388`, 2015. Accessed: 2019-06-06.

[33] André Ricardo Abed Grégio, Paulo Lício de Geus, Christopher Kruegel, and Giovanni Vigna. Tracking memory writes for malware classification and code reuse identification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 134–143. Springer, 2012.

[34] Jakub Palša, Norbert Ádám, Ján Hurtuk, Eva Chovancová, Branislav Madoš, Martin Chovanec, and Stanislav Kocan. Mlmd—a malware-detecting antivirus tool based on the xgboost machine learning algorithm. *Applied Sciences*, 12(13):6672, 2022.

[35] Joshua Saxe and Konstantin Berlin. expose: A character-level convolutional neural network with embeddings for detecting malicious urls, file paths and registry keys. *arXiv preprint arXiv:1702.08568*, 2017.

[36] Azadeh Jalilian, Zahra Narimani, and Ebrahim Ansari. Static signature-based malware detection using opcode and binary information. In *The 7th International Conference on Contemporary Issues in Data Science*, pages 24–35. Springer, 2019.

[37] Bojan Kolosnjaji, Apostolis Zarras, Tamas Lengyel, George Webster, and Claudia Eckert. Adaptive semantics-aware malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 419–439. Springer, 2016.

[38] Mamoru Mimura. Evaluation of printable character-based malicious pe file-detection method. *Internet of Things*, 19:100521, 2022.

[39] Gianni Amato. Statically analyze pe and microsoft office files. `https://libraries.io/pypi/peframe-ds`, 2021. Accessed: 2021-02-01.

[40] László Molnár Markus F.X.J. Oberhumer and John F. Reiser. Upx the ultimate packer for executables. `https://upx.github.io/`.

[41] Python Software Foundation. zlib — compression compatible with gzip. `https://docs.python.org/3/library/zlib.html`.

[42] Mono Project. Dis/assembling cil code. `https://www.mono-project.com/docs/tools+libraries/tools/monodis/`.

[43] Michael Kerrisk. objdump — linux manual page. `https://man7.org/linux/man-pages/man1/objdump.1.html`.

[44] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. Neurlux: dynamic malware analysis without feature engineering. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 444–455, 2019.

[45] Yu Wang, Jack W Stokes, and Mady Marinescu. Neural malware control with deep reinforcement learning. In *MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM)*, pages 1–8. IEEE, 2019.

[46] Yu Wang, Jack Stokes, and Mady Marinescu. Actor critic deep reinforcement learning for neural malware control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1005–1012, 2020.

[47] CrowdStrike. Falcon sandbox, May 2019.

[48] Aaron Walters. Volatility framework, Dec 2016.

[49] Sonali Yadav. Cyber forensics: Its importance, cyber forensics techniques, and tools. In *Critical Concepts, Standards, and Techniques in Cyber Forensics*, pages 1–15. IGI Global, 2020.

[50] VMware. Vmware workstation player. `https://www.vmware.com/products/workstation-player.html`.

[51] Brian Baskin. Noriben malware analysis sandbox.

[52] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

[53] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[54] Fabian Pedregosa, Gaël 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. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[56] Tom O'Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. Keras Tuner, 2019.

[57] Jason Huang. Rmsprop, 2020.

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

[59] L Breiman. Random forests. *Machine Learning*, 45:5–32, 10 2001.

[60] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[61] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[62] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[63] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[64] Yu Gu, Robert Tinn, Hao Cheng, Michael Lucas, Naoto Usuyama, Xiaodong Liu, Tristan Naumann, Jianfeng Gao, and Hoifung Poon. Domain-specific language model pretraining for biomedical natural language processing. *ACM Transactions on Computing for Healthcare (HEALTH)*, 3(1):1–23, 2021.

[65] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[66] Mohammad Taher Pilehvar and Jose Camacho-Collados. Embeddings in natural language processing: Theory and advances in vector representations of meaning. *Synthesis Lectures on Human Language Technologies*, 13(4):1–175, 2020.

[67] ZXS107020. Multi-class classification using xgboost, 2018.

[68] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 38–49. IEEE, 2000.

[69] Rajesh Kumar and S Geetha. Malware classification using xgboost-gradient boosted decision tree. *Adv. Sci. Technol. Eng. Syst*, 5:536–549, 2020.

[70] Hyrum S Anderson and Phil Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[71] Li Chen, Ravi Sahita, Jugal Parikh, and Marc Marino. Stamina: scalable deep learning approach for malware classification. *Intel Labs Whitepaper*, 2020.

[72] Abdurrahman Pektaş and Tankut Acarman. Malware classification based on api calls and behaviour analysis. *IET Information Security*, 12(2):107–117, 2018.

[73] Jun Chen, Shize Guo, Xin Ma, Haiying Li, Jinhong Guo, Ming Chen, and Zhisong Pan. Slam: A malware detection method based on sliding local attention mechanism. *Security and Communication Networks*, 2020, 2020.

[74] Alibaba. Alitianchi contest, 2021. https://tianchi.aliyun.com/competition/introduction.htm?spm=5176.11409106.5678.1.4354684c I0fYC1? raceId=231668s.

[75] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[76] Shiqi Luo, Zhiyuan Liu, Bo Ni, Huanhuan Wang, Hua Sun, and Yong Yuan. Android malware analysis and detection based on attention-cnn-lstm. *Journal of Computers*, 14(1):31–44, 2019.

[77] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[78] VX Heaven. Vx heaven virus collection 2010-05-18. 2010-05-18.

[79] Ben Athiwaratkun and Jack W Stokes. Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 2482–2486. IEEE, 2017.

[80] Justin Seitz. *Gray Hat Python: Python programming for hackers and reverse engineers.* no starch press, 2009.

[81] Qipeng Guo, Xipeng Qiu, Pengfei Liu, Yunfan Shao, Xiangyang Xue, and Zheng Zhang. Star-transformer. *arXiv preprint arXiv:1902.09113*, 2019.

[82] Hex-Rays. A powerful disassembler and a versatile debugger. `https://www.hex-rays.com/ida-pro/`.

[83] Raymond Canzanese, Moshe Kam, and Spiros Mancoridis. Toward an automatic, online behavioral malware classification system. In *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 111–120. IEEE, 2013.

[84] Sumaya Saadat and V Joseph Raymond. Malware classification using cnn-xgboost model. In *Artificial Intelligence Techniques for Advanced Computing Applications*, pages 191–202. Springer, 2021.

[85] Kaggle. Malimg dataset, 2019.

[86] Zhihua Cui, Fei Xue, Xingjuan Cai, Yang Cao, Gai-ge Wang, and Jinjun Chen. Detection of malicious code variants based on deep learning. *IEEE Transactions on Industrial Informatics*, 14(7):3187–3196, 2018.

[87] Xin-She Yang. A new metaheuristic bat-inspired algorithm. In *Nature inspired cooperative strategies for optimization (NICSO 2010)*, pages 65–74. Springer, 2010.

[88] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and Bangalore S Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, pages 1–7, 2011.

[89] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian joint conference on artificial intelligence*, pages 137–149. Springer, 2016.

[90] Corvus Forensics. Virusshare, Jun 2020.

[91] Kyle Maxwell. Maltrieve: A tool to retrieve malware directly from the source for security researchers., 2015. https://github.com/krmaxwell/maltrieve.

[92] Mingdong Tang and Quan Qian. Dynamic api call sequence visualisation for malware classification. *IET Information Security*, 13(4):367–377, 2019.

[93] Ben Athiwaratkun and Jack W Stokes. Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 2482–2486. IEEE, 2017.

[94] Hanqi Zhang, Xi Xiao, Francesco Mercaldo, Shiguang Ni, Fabio Martinelli, and Arun Kumar Sangaiah. Classification of ransomware families with machine learning based on n-gram of opcodes. *Future Generation Computer Systems*, 90:211–221, 2019.

[95] Wen Zhang, Taketoshi Yoshida, and Xijin Tang. A comparative study of tf* idf, lsi and multi-words for text classification. *Expert systems with applications*, 38(3):2758–2765, 2011.

[96] Sanchit Gupta, Harshit Sharma, and Sarvjeet Kaur. Malware characterization using windows api call sequences. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 271–280. Springer, 2016.

[97] Anishka Singh, Rohit Arora, and Himanshu Pareek. Malware analysis using multiple api sequence mining control flow graph. *arXiv preprint arXiv:1707.02691*, 2017.

[98] others Kevin M (sylikc), Sven Marnach. Python wrapper for exiftool, Mar 2022.

[99] George D Webster, Bojan Kolosnjaji, Christian von Pentz, Julian Kirsch, Zachary D Hanif, Apostolis Zarras, and Claudia Eckert. Finding the needle: A study of the pe32 rich header and respective malware triage. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 119–138. Springer, 2017.

[100] Capstone. Capstone the ultimate disassembler. `https://www.capstone-engine.org/lang_python.html`.

[101] Adam Kramer. Acquiring a memory dump from fleeting malware, Nov 2017.

[102] Joachim Metz. Python bindings module for libscca. `https://pypi.org/project/libscca-python/`.

[103] Willi Ballenthin. Pure python parser for recent windows event log files (.evtx). `https://pypi.org/project/python-evtx/`.

[104] Ed Gordon. Zip for windows. `http://gnuwin32.sourceforge.net/packages/zip.htm`.

[105] Thomas Kluyver. Read and write zip files - backport of the zipfile module from python 3.6. `https://pypi.org/project/zipfile36/`.

[106] VirusTotal. Virustotal, June 2020.

[107] VirusSign. Virussign, Jun 2020.

[108] FileHorse. Filehorse, Jun 2020.

[109] Scott Lundberg. Shap, Jan 2021.

[110] Didier Stevens. New tool: Xorsearch.py, August 2020.

[111] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[112] François Chollet et al. Keras, 2015.

[113] Claudio Filipi Gonçalves Dos Santos and João Paulo Papa. Avoiding overfitting: A survey on regularization methods for convolutional neural networks. *ACM Computing Surveys (CSUR)*, 54(10s):1–25, 2022.

[114] Angelo Oliveira and R Sassi. Behavioral malware detection using deep graph convolutional neural networks. *TechRxiv.[link]*, 2019.

[115] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[116] Angelo Oliveira. Malware analysis datasets: Api call sequences, 2019.

[117] Abir Rahali and Moulay A Akhloufi. Malbert: Using transformers for cybersecurity and malicious software detection. *arXiv preprint arXiv:2103.03806*, 2021.

[118] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[119] TensorFlow. Module:tf.compat.v1.keras.layers.

[120] Saja Alqurashi and Omar Batarfi. Static and dynamic malware analysis to extract opcode sequences and api call sequences. 2018.

[121] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K Penya, Borja Sanz, Carlos Laorden, and Pablo G Bringas. Idea: Opcode-sequence-based malware detection. In *Engineering Secure Software and Systems: Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings 2*, pages 35–43. Springer, 2010.

[122] ABUSE. Urlhaus database, Jun 2020. `https://urlhaus.abuse.ch/`.

[123] Ferhat Ozgur Catak, Javed Ahmed, Kevser Sahinbas, and Zahid Hussain Khand. Data augmentation based malware detection using convolutional neural networks. *PeerJ Computer Science*, 7:e346, 2021.

[124] Sumith Maniath, Aravind Ashok, Prabaharan Poornachandran, VG Sujadevi, Prem Sankar AU, and Srinath Jan. Deep learning LSTM based ransomware detection. In *2017 Recent Developments in Control, Automation & Power Engineering (RDCAPE)*, pages 442–446. IEEE, 2017.

[125] Ferhat Ozgur Catak, Ahmet Faruk Yazı, Ogerta Elezaj, and Javed Ahmed. Deep learning based sequential model for malware analysis using windows exe api calls. *PeerJ Computer Science*, 6:e285, 2020.

[126] Daniele Grattarola and Cesare Alippi. Graph neural networks in tensorflow and keras with spektral [application notes]. *IEEE Computational Intelligence Magazine*, 16(1):99–106, 2021.

[127] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[128] Yaser Keneshloo, Tian Shi, Naren Ramakrishnan, and Chandan K. Reddy. Deep reinforcement learning for sequence to sequence models, 2019.

[129] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[130] Alexandre dos Santos Mignon and Ricardo Luis de Azevedo da Rocha. An adaptive implementation of ε-greedy in reinforcement learning. *Procedia Computer Science*, 109:1146–1151, 2017.