MalDives AI-Assisted Malware Detection Tool

Prepared by Andy Wang, Jacqueline Chung

(Name In alphabetical order)

Information System Security

Southern Alberta Institute of Technology

July 31, 2023

Contents

Contents 2
1 Executive Summary 3
2 Methodology 4
2.1 Research 4
2.2 Obtaining and Understanding Datasets 6
2.3 Setting Up Development Environment 10
2.4 Algorithm Selection 11
2.5 Training 20
2.6 Design 22
2.7 Feature Extraction 23
2.8 Building the tool 29
2.9 Testing 31
3 Challenges 34
3.1 Start from Scratch 34
3.2 Open-Sourced Dataset 34
3.3 Time Management 34
3.4 Obtaining Malware Samples 35
3.5 Attempt to Set Up Cloud-Based Environment 35
4 Lessons Learned 36
4.1 Importance of Development Environment 36
4.2 Keeping Up to Date 36
4.3 Scalability 36
4.4 Better Feature Engineering 37
5 Future Development 38
5.1 Moving to Cloud 38
5.2 Integration of VirusTotal API 38
5.3 Self-training and Update 38
6 Conclusion 39
7 References 40
8 Appendix 41

1 Executive Summary

In an era where digital threats are both pervasive and evolving, there is an unyielding demand for innovative solutions. 'Maldives', an avant-garde project, was conceived with the vision to address this challenge, embodying the merger of traditional cybersecurity techniques with the dynamism of machine learning. As we delved into this project, our primary goal was to develop a tool capable of leveraging the latest advancements in AI to fortify malware detection and analysis.

Spanning a challenging yet enlightening four-month timeline, the development process was a testament to our commitment and ingenuity. The journey to creating 'Maldives' was neither linear nor devoid of obstacles. Right from the outset, we grappled with fundamental challenges such as building our machine learning proficiency from scratch, searching for accessible and relevant datasets, and maneuvering through budgetary constraints.

'Maldives', in its current iteration, is designed to analyze both static and dynamic features of potential malware, offering users insights into its type and family. The tool provides this classification based on learned patterns from historical data, showcasing the might of machine learning in real-world applications. Furthermore, it comes equipped with the capacity to adapt and grow, constantly updating its knowledge base, making it not only reactive but also proactive.

Another significant facet of our project was our pursuit of adaptability. We explored cloud solutions, considered the integration of APIs like VirusTotal, and delved deep into feature engineering, ensuring that 'Maldives' remains not just relevant, but also at the forefront of malware detection technology.

As we reflect on our journey, the 'Maldives' project emerges as a beacon of what is achievable when innovation meets determination. The tool is more than just a culmination of our efforts; it represents a promise for a safer digital future. With plans already outlined for its evolution, 'Maldives' is set to redefine standards in AI-assisted cybersecurity.

2 Methodology

2.1 Research

The beginning phase of our project involved an in-depth understanding of the domain we were venturing into - Machine Learning (ML) for malware detection. We embarked on this learning journey with the help of an online course from YouTube named "Machine Learning for Cyber Security," curated by Dr. Ricardo A. Calix, Ph.D., of Purdue University Northwest, Hammond, IN, USA.

The course was instrumental in introducing us to a wide spectrum of concepts related to ML and Deep Learning (DL), and their applications in Cybersecurity. It provided a wealth of practical code examples, which served as a solid foundation for the rest of our project.

2.1.1 Understanding Machine Learning and Deep Learning

Machine Learning is a subset of Artificial Intelligence (AI) that provides systems the ability to learn and improve from experience without being explicitly programmed. On the other hand, Deep Learning is a subfield of Machine Learning that uses neural networks with many layers (deep neural networks), facilitating the learning of complex patterns in large amounts of data.

The significant difference between the two lies in their approaches to learning from data. ML algorithms typically improve their performance as the number of samples increases, while DL algorithms excel at learning from large, complex datasets.

The table below offers a quick comparison:

	Machine Learning	**Deep Learning**
Data Dependencies	Performs well on small- to medium-sized datasets	Requires large datasets for effective learning
Hardware Dependencies	Can work on low-end machines	Requires powerful hardware (often GPUs)
Feature Engineering	Requires manual feature extraction	Automatic feature extraction
Interpretability	Models are often easily interpretable	Models can be challenging to interpret due to their complexity

2.1.2 Familiarization with Machine Learning Models

As part of our foundational study, we familiarized ourselves with various ML models, their unique characteristics, and applications:

- **Linear Regression**: Predicts a linear relationship between input variables (X) and a single output variable (Y).
- **Logistic Regression**: Useful for classification problems. It uses a logistic function to model a binary dependent variable.

- **Decision Trees**: A type of flowchart-like structure, where each internal node denotes a test on an attribute, each branch represents the outcome of a test, and each leaf node holds a class label.
- **Random Forest**: An ensemble learning method that operates by constructing multiple decision trees at training time and outputting the class that is the mode of the classes of the individual trees.
- **Support Vector Machines**: A set of supervised learning methods used for classification, regression, and outliers detection.
- **Neural Networks**: Computational systems inspired by the human brain, capable of learning from and interpreting data. Example models include Convolutional Neural Network(CNN) and Multilayer Perceptron models(MLP).

2.1.3 Introduction to Scikit-learn and TensorFlow

Throughout the course, we learned the importance of Python's sklearn and tensorflow libraries in ML and DL applications.

Sklearn, or Scikit-learn, is a Python library that provides a selection of efficient tools for machine learning and statistical modeling, including classification, regression, clustering, and dimensionality reduction via a consistent interface in Python.

On the other hand, Tensorflow is an open-source library developed by the Google Brain team. It's used for both ML and DL for tasks such as neural networks.

These libraries, along with their vast functionalities and easy-to-use interfaces, have been an integral part of our development process. They enabled us to build ML models, train, test, and evaluate them with ease and efficiency.

2.1.4 Introduction to Static and Dynamic Analysis

In the realm of cybersecurity, two primary methods are employed for malware analysis: static analysis and dynamic analysis. Both techniques offer their unique insights and, when used together, can provide a comprehensive understanding of the malware's behavior.

Static Analysis is conducted without executing the code of the program. It involves examining the code, structure, and resources of the malicious file. Techniques such as reading strings, API calls, disassembly (converting machine code into assembly code), decompiling (converting low-level code to high-level code), and reviewing other non-executable parts of a program are part of static analysis. This approach allows us to understand what the malware can potentially do if executed.

Dynamic Analysis, in contrast, involves observing the behavior of the malware during execution. It requires a controlled, isolated environment (like a sandbox) to prevent the malicious code from affecting non-laboratory systems. This analysis reveals how the malware interacts with the system, network, and other programs. It provides insights into the malware's runtime behavior and real-time operation, such as file modification, network traffic, and memory changes.

Our decision to adopt both static and dynamic analysis is rooted in the fact that each approach compensates for the limitations of the other. While static analysis can detect potential threats in the code, dynamic analysis can reveal the actual behavior of the malware, allowing us to validate and further explore the findings of static analysis. Also, static analysis can be beneficial when the malware uses advanced obfuscation techniques that make runtime analysis challenging.

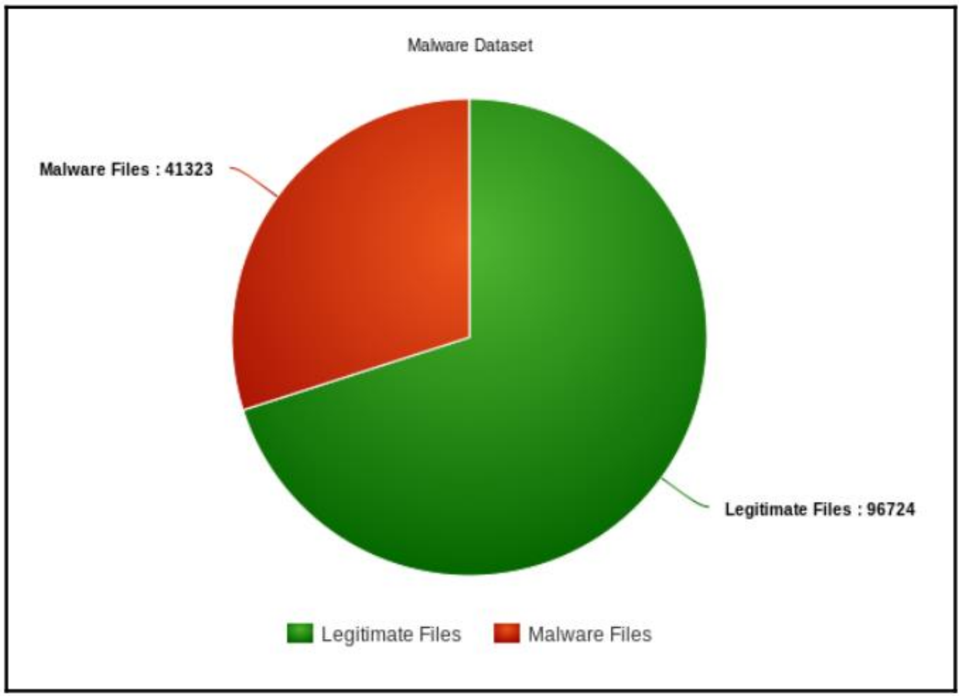
The course we took online provided a broad overview of these techniques. But the real impetus behind our decision was our previous coursework, where we delved into the theoretical aspects of static and dynamic malware analysis. We recognized this project as an excellent opportunity to apply those theoretical concepts into a practical setting, thereby solidifying our understanding and honing our skills.

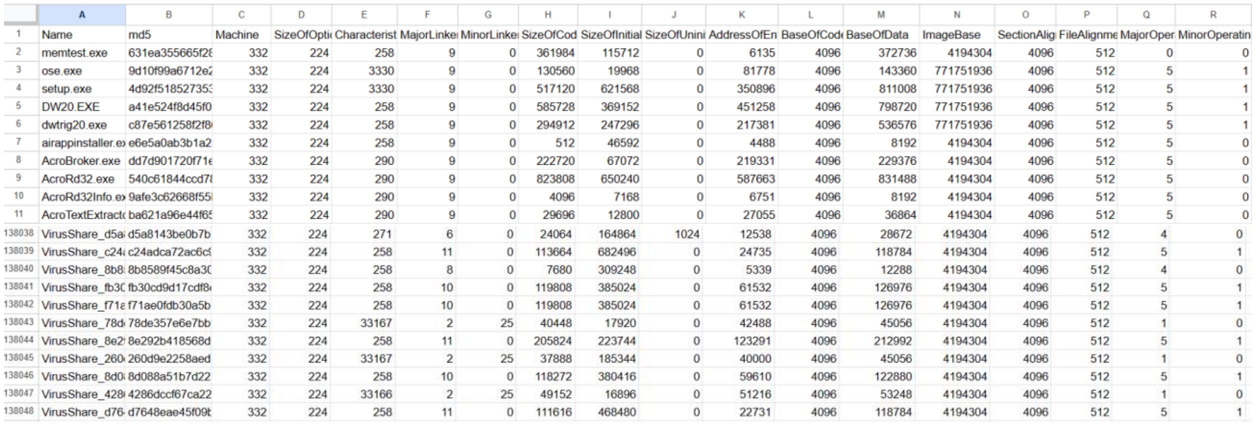
2.2 Obtaining and Understanding Datasets

2.2.1 Static Analysis Based on API Calls

For our project's static analysis, we utilized the PacktPublishing – GitHub Project. This resource provided an exemplary foundation for the creation and confirmation of our machine-learning models. The malware dataset contains features extracted from the following:

- 41,323 Windows binaries (executables .exe and .dlls), as legitimate files.
- 96,724 malware files downloaded from the VirusShare website. So, the dataset contains 138,048 lines, in total. The dataset is divided as follows:





	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Name	md5	Machine	SizeOfOpti	Characterist	MajorLinke	MinorLinke	SizeOfCod	SizeOfInitial	SizeOfUnini	AddressOfEn	BaseOfCod	BaseOfData	ImageBase	SectionAlig	FileAlignme	MajorOper	MinorOperatin
2	memtest.exe	631ea355665f28	332	224	258	9	0	361984	115712	0	6135	4096	372736	4194304	4096	512	0	0
3	ose.exe	9d10f99a6712e2	332	224	3330	9	0	130560	19968	0	81778	4096	143360	771751936	4096	512	5	1
4	setup.exe	4d92f51852735	332	224	3330	9	0	517120	621568	0	350896	4096	811008	771751936	4096	512	5	1
5	DW20.EXE	a41e524f8d45f0	332	224	258	9	0	585728	369152	0	451258	4096	798720	771751936	4096	512	5	1
6	dwtrig20.exe	c87e561258f2f8	332	224	258	9	0	294912	247296	0	217381	4096	536576	771751936	4096	512	5	1
7	airappinstaller.ex	e6e5a0ab3b1a2	332	224	258	9	0	512	46592	0	4488	4096	8192	4194304	4096	512	5	0
8	AcroBroker.exe	dd7d901720f71e	332	224	290	9	0	222720	67072	0	219331	4096	229376	4194304	4096	512	5	0
9	AcroRd32.exe	540c61844ccd7	332	224	290	9	0	823808	650240	0	587663	4096	831488	4194304	4096	512	5	0
10	AcroRd32Info.ex	9afe3c62668f55	332	224	290	9	0	4096	7168	0	6751	4096	8192	4194304	4096	512	5	0
11	AcroTextExtract	ba621a96e44f65	332	224	290	9	0	29696	12800	0	27055	4096	36864	4194304	4096	512	5	0
138038	VirusShare_d5a	d5a8143be0b7b	332	224	271	6	0	24064	164864	1024	12538	4096	28672	4194304	4096	512	4	0
138039	VirusShare_c24	c24adca72ac6c	332	224	258	11	0	113664	682496	0	24735	4096	118784	4194304	4096	512	5	1
138040	VirusShare_8b8	8b8589f45c8a30	332	224	258	8	0	7680	309248	0	5339	4096	12288	4194304	4096	512	4	0
138041	VirusShare_fb30	fb30cd9d17cdf8	332	224	258	10	0	119808	385024	0	61532	4096	126976	4194304	4096	512	5	1
138042	VirusShare_f71a	f71ae0fdb30a5b	332	224	258	10	0	119808	385024	0	61532	4096	126976	4194304	4096	512	5	1
138043	VirusShare_78d	78de357e6e7bb	332	224	33167	2	25	40448	17920	0	42488	4096	45056	4194304	4096	512	1	0
138044	VirusShare_8e2	8e292b418568d	332	224	258	11	0	205824	223744	0	123291	4096	212992	4194304	4096	512	5	1
138045	VirusShare_260	260d9e2258aed	332	224	33167	2	25	37888	185344	0	40000	4096	45056	4194304	4096	512	1	0
138046	VirusShare_8d0	8d088a51b7d22	332	224	258	10	0	118272	380416	0	59610	4096	122880	4194304	4096	512	5	1
138047	VirusShare_428	4286dccf67ca22	332	224	33166	2	25	49152	16896	0	51216	4096	53248	4194304	4096	512	1	0
138048	VirusShare_d76	d7648eae45f09t	332	224	258	11	0	111616	468480	0	22731	4096	118784	4194304	4096	512	5	1

Image: Snippet of the PacktPublishing dataset

This dataset consists of 56 features, encompassing several categories and features that collectively help us characterize and understand the behavior of malware. These columns contain features that can be applied to subsequent machine learning processes for malware detection and analysis. The most important features in this dataset are summarized as follows:

- **DllCharacteristics:** This feature describes various attributes related to Dynamic Link Libraries (DLLs). Analyzing DLL characteristics might reveal anomalies or specific behaviors associated with malicious actions.
- **Machine:** This represents the target machine's architecture. Malware often targets specific system architectures, so this feature might help identify particular attacks or malicious activities.
- **Characteristics:** This describes various characteristics and behaviors of the file. These may include execution levels, system compatibility, etc., which can reveal the specific intent or behavior of malicious files.
- **Subsystem:** The subsystem characteristic indicates the target environment of the code (e.g., Windows GUI, Console). Unusual subsystem choices might indicate attempts to conceal malicious activities.
- **VersionInformationSize:** The size of version information might involve metadata related to the file. Malware may tamper with or forge this information to conceal its true intent.
- **SectionsMaxEntropy:** The maximum entropy of sections might reveal encrypted or compressed parts of the code. High entropy is often associated with obfuscation techniques used by malware.
- **ImageBase:** This feature represents the preferred loading address of the program. Non-standard base addresses might be indicative of adversarial behavior, like attempts to bypass certain security measures.
- **ResourcesMaxEntropy:** The maximum entropy of resources can be used to detect whether resources have been encrypted or compressed. This might be indicative of malware attempting to hide its behavior.
- **MajorSubsystemVersion:** The major subsystem version may be related to specific operating systems or environments. Analyzing this information might reveal malicious attacks targeted at particular systems.
- **SizeOfOptionalHeader:** The size of the optional header might relate to the complexity of the file structure. Unconventional sizes might be indicative of malicious alterations.

- **MajorOperatingSystemVersion:** The major operating system version can reveal the OS version for which the file was compiled. This might be related to malicious activities targeting specific OS versions.
- **ResourcesMinEntropy:** The minimum entropy of resources might help detect whether resources have been specially processed, such as obfuscation or encryption.
- **SectionsMinEntropy:** The minimum entropy of sections can reveal the consistency and complexity of code or data sections. Low entropy might be associated with unencrypted, human-readable code, while high entropy might be indicative of obfuscation or encryption.

These features provide a comprehensive perspective on a file's structure and behavior, playing a crucial role in building an effective malware detection system.

2.2.2 Dynamic Analysis Based on Memory Dump

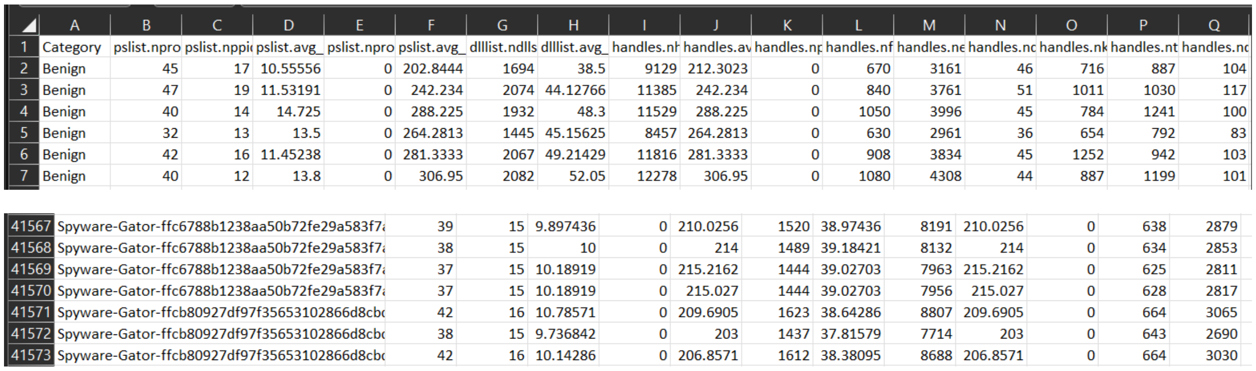
For our project's dynamic analysis, we utilized the Canadian Institute for Cybersecurity's Malware Memory Analysis dataset, also known as CIC-MalMem-2022. The dataset provided an ideal ground for developing and validating our machine learning models.

The dataset is balanced, composed of 50% malicious memory dumps and 50% benign memory dumps, totaling 58,596 records with 29,298 benign and 29,298 malicious. This balance helps prevent bias in the model towards either malicious or benign classifications, thus contributing to more accurate and reliable detection results.

Further, the dataset is also classified into various malware categories and families, as shown in the table below:

Malware category	Malware families	Sample Count
Trojan Horse	Zeus	195
	Emotet	196
	Refroso	200
	Scar	200
	Reconyc	157
Spyware	180Solutions	200
	Coolwebsearch	200
	Gator	200
	Transponder	241
	TIBS	141
Ransomware	Conti	200
	MAZE	195
	Pysa	171
	Ako	200
	Shade	220

This classification not only facilitates the detection of malware but also allows us to categorize the type and potentially the family of the detected malware. It's particularly valuable in handling sophisticated attacks where identifying the malware type can significantly assist in crafting an effective response or countermeasure.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	Category	pslist.npro	pslist.nppi	pslist.avg_	pslist.npro	pslist.avg_	dlllist.ndlls	dlllist.avg_	handles.nh	handles.av	handles.np	handles.nf	handles.ne	handles.nc	handles.nk	handles.nt	handles.nc
2	Benign	45	17	10.55556	0	202.8444	1694	38.5	9129	212.3023	0	670	3161	46	716	887	104
3	Benign	47	19	11.53191	0	242.234	2074	44.12766	11385	242.234	0	840	3761	51	1011	1030	117
4	Benign	40	14	14.725	0	288.225	1932	48.3	11529	288.225	0	1050	3996	45	784	1241	100
5	Benign	32	13	13.5	0	264.2813	1445	45.15625	8457	264.2813	0	630	2961	36	654	792	83
6	Benign	42	16	11.45238	0	281.3333	2067	49.21429	11816	281.3333	0	908	3834	45	1252	942	103
7	Benign	40	12	13.8	0	306.95	2082	52.05	12278	306.95	0	1080	4308	44	887	1199	101
41567	Spyware-Gator-ffc6788b1238aa50b72fe29a583f7a	39	15	9.897436	0	210.0256	1520	38.97436	8191	210.0256	0	638	2879				
41568	Spyware-Gator-ffc6788b1238aa50b72fe29a583f7a	38	15	10	0	214	1489	39.18421	8132	214	0	634	2853				
41569	Spyware-Gator-ffc6788b1238aa50b72fe29a583f7a	37	15	10.18919	0	215.2162	1444	39.02703	7963	215.2162	0	625	2811				
41570	Spyware-Gator-ffc6788b1238aa50b72fe29a583f7a	37	15	10.18919	0	215.027	1444	39.02703	7956	215.027	0	628	2817				
41571	Spyware-Gator-ffcb80927df97f35653102866d8cbc	42	16	10.78571	0	209.6905	1623	38.64286	8807	209.6905	0	664	3065				
41572	Spyware-Gator-ffcb80927df97f35653102866d8cbc	38	15	9.736842	0	203	1437	37.81579	7714	203	0	643	2690				
41573	Spyware-Gator-ffcb80927df97f35653102866d8cbc	42	16	10.14286	0	206.8571	1612	38.38095	8688	206.8571	0	664	3030				

Image: Snippet of the CIC-MalMem-2022 dataset

This dataset consists of several categories and features, all of which help us characterize and understand the behavior of malware. Despite lack of documentation, we found that these features have been extracted using Volatility, a powerful framework for volatile memory analysis through our research. While Volatility will be discussed in detail later in the document, we'll provide a brief overview of the columns and some features here:

- **Categories**: This column consists of the malware type, family and hash if it is malicious, otherwise it is labeled as "Benign"
- **pslist**: This module encapsulates features related to the list of running processes in the system.
- **dlllist**: This module covers features concerning the list of DLLs (Dynamic-link Libraries) loaded in each process.
- **handles**: This module contains features about various types of system handles, representing system resources.
- **ldrmodules**: This module covers features related to loaded modules.
- **malfind**: This module involves features specific to discovered malicious injections and related properties.
- **psxview**: This module pertains to features related to discrepancy-based detection of hidden processes.
- **modules**: This module consists of features about the system's loaded kernel modules.
- **svcscan**: This module covers features about the system's services.
- **callbacks**: This module includes features about system-wide and process-specific callback routines.
- **Class**: This is the target feature that indicates whether a memory dump is benign or malicious.

The extension of the above modules are the features that is extracted, for example:

- **pslist.nproc**: This feature indicates the number of processes running on the system. A significant deviation from the normal count could indicate malicious activity, as malware often starts additional processes to execute its tasks.
- **dlllist.ndlls**: This feature represents the total number of DLLs loaded across all processes. Unusually high numbers might suggest the presence of malicious DLLs injected into various processes.
- **handles.nhandles**: This feature counts the total number of handles in the system. Handles are system resources, and a high number of them could indicate a resource-hogging malware operation.

Understanding each feature is crucial as they provide the raw data used for training our machine learning model, and ultimately, they play a pivotal role in determining the effectiveness of our malware detection tool.

2.3 Setting Up Development Environment

Setting up a consistent and efficient development environment was one of our initial tasks. Our goal was to create a replicable, isolated setup that can maintain versioning for both Python and the required packages for the project.

We found the perfect tool for this task in Anaconda 3. Anaconda is a free, open-source distribution of Python and R programming languages. It is specifically designed for data science and machine learning applications. The key benefits of using Anaconda include its robust package management system, environment isolation capabilities, and pre-bundled scientific computing libraries.

Anaconda allows us to create separate environments for different projects, which can have their own versions of Python and installed packages. This ensures that the specific dependencies of one project do not interfere with the dependencies of another project.

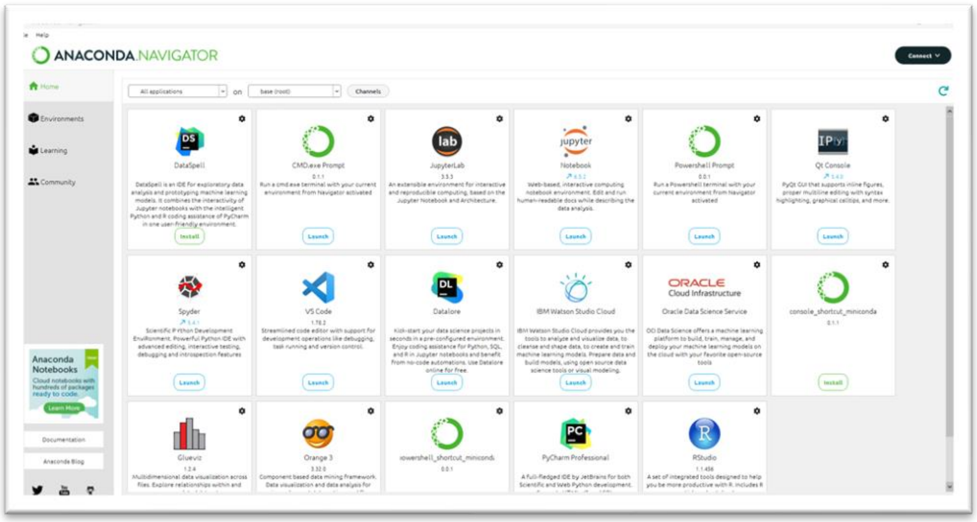
Here is the list of specific modules we required, along with their versions and a brief description:

- **Python 3.10.9**: The latest version of Python we used in this project. Python is a versatile, high-level programming language widely used for developing ML models.
- **scikit-learn 1.2.2**: A popular machine learning library in Python, offering various algorithms for classification, regression, clustering, etc., along with tools for model fitting, data preprocessing, model selection and evaluation.
- **keras 2.12.0**: An open-source neural-network library written in Python. It is user-friendly, modular, and extensible, and it can work on top of TensorFlow, providing a high-level, more intuitive API for building and training deep learning models.
- **joblib 1.3.1**: A set of tools to provide lightweight pipelining in Python. It leverages the power of multiprocessing but with a greater emphasis on computational efficiency. It's particularly useful for tasks that are heavy on large data and need to be distributed across multiple cores.
- **pandas 2.0.1**: A powerful, open-source data manipulation and analysis library for Python. It provides data structures and functions necessary to manipulate structured data, including

functionality for manipulating and aggregating data, handling missing data, merging data sets, and performing IO operations.

- **tensorflow 2.10.0**: An open-source library developed by Google Brain team. It's used for both machine learning and deep learning for tasks such as neural networks.

Using Anaconda 3, we could efficiently manage these packages and maintain an organized, controlled development environment, fostering productivity and minimizing issues related to package interdependencies or Python version conflicts.



DataSpell is an IDE for exploratory data analysis and prototyping machine learning models. It combines the interactivity of Jupyter notebooks with the intelligent Python and R coding assistance of PyCharm in one user-friendly environment.

Run a cmd.exe terminal with your current environment from Navigator activated

An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Run a Powershell terminal with your current environment from Navigator activated

PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Streamlined code editor with support for development operations like debugging, task running and version control.

Kick-start your data science projects in seconds in a pre-configured environment. Enjoy coding assistance for Python, SQL, and R in Jupyter notebooks and benefit from no-code automations. Use Datalore online for free.

IBM Watson Studio Cloud provides you the tools to analyze and visualize data, to cleanse and shape data, to create and train machine learning models. Prepare data and build models, using open source data science tools or visual modeling.

OCI Data Science offers a machine learning platform to build, train, manage, and deploy your machine learning models on the cloud with your favorite open-source tools

Multidimensional data visualization across files. Explore relationships within and

Component based data mining framework. Data visualization and data analysis for

A full-fledged IDE by JetBrains for both Scientific and Web Python development.

A set of integrated tools designed to help you be more productive with R. Includes R

Image 1: Anaconda 3 provides tools and modules that are ready for install and use for machine learning purpose.

2.4 Algorithm Selection

Algorithm selection is the bulk part of our project. We first embarked on a process of experimenting with various machine learning algorithms. Our goal was to identify the algorithm which delivered the best performance in terms of accuracy, interpretability, and computational efficiency. The dataset, already processed and formatted as a CSV file, allowed us to directly feed it into different algorithms for model training.

We initially explored four machine learning models - Random Forest (RF), Convolutional Neural Networks (CNN), Multi-Layer Perceptron (MLP), and Support Vector Machines (SVM), each of them with different parameter configurations.

2.4.1 Static Analysis

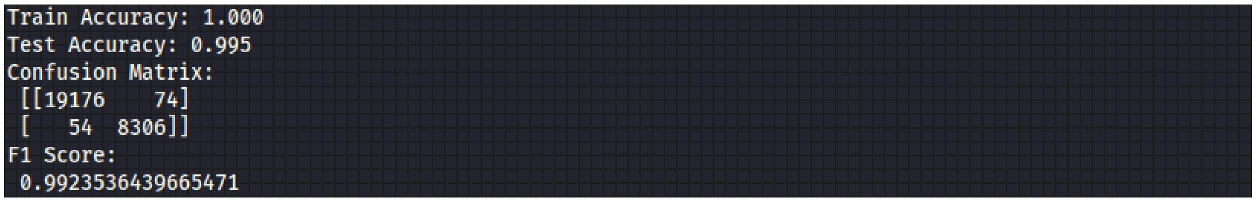
2.4.1.1 Direct Categorial Classification With Random Forest – Fixed Parameters

The static analysis we choice was the RF algorithm, aiming to predict the category of the malware directly with fixed parameters. In this approach, we utilized the RandomForestClassifier from the sklearn.ensemble library, initializing it with a fixed parameter of 100 decision trees (n_estimators=100).

The training-testing split ratio is 80:20, achieved through the test_size=0.2 parameter in the train_test_split function. The random_state=42 ensured the consistency and reproducibility of the split.

Parameter	**Value**	**Description**
n_estimators	100	It is a key parameter in the Random Forest classifier, which specifies the number of decision trees in the forest. n_estimators is set to 100, meaning that the Random Forest will consist of 100 decision trees. Having multiple trees enables the Random Forest to capture more complex patterns in the data, and the predictions from the individual trees are combined to produce a final prediction.
test_size	0.2	This parameter specifies the size of the testing set. 'test_size=0.2' means that the testing set will contain 20% of the original data, with the remaining 80% used for the training set.
random_state	42	This parameter is used to set the seed for the random number generator, ensuring the repeatability of the data split. By setting random_state=42, anyone using the same code and data can achieve the same split, thereby ensuring consistency in the results.

Based on the aforementioned parameters and the rigorous application of the Random Forest algorithm, the machine learning model was trained and evaluated. The subsequent analysis yielded the following conclusions, elucidating the model's proficiency in distinguishing between benign and malicious malware with remarkable accuracy and precision.



```
Train Accuracy: 1.000
Test Accuracy: 0.995
Confusion Matrix:
 [[19176    74]
 [   54  8306]]
F1 Score:
 0.9923536439665471
```

Interpretation of the results:

- Train Accuracy: 1.000

The training accuracy is the proportion of correct predictions made by the model on the training set. In this case, the training accuracy of 1.000 means that the model performed perfectly, with all training samples being classified correctly as either benign or malicious.

- Test Accuracy: 0.995

The test accuracy is the proportion of correct predictions made by the model on the test set. A test accuracy of 0.995 indicates that the model performed exceptionally well on unseen data, correctly classifying nearly all test samples as either benign or malicious.

- Confusion Matrix:

The confusion matrix is a table used to evaluate the performance of a classification model:

  - True Positives (TP): 8306, representing the number of malicious samples correctly identified.
  - True Negatives (TN): 19176, representing the number of benign samples correctly identified.
  - False Positives (FP): 74, representing the number of benign samples incorrectly identified as malicious.
  - False Negatives (FN): 54, representing the number of malicious samples incorrectly identified as benign.
- F1 Score: 0.9923536439665471

The F1 score is the harmonic mean of precision and recall, providing a single metric that balances both false positives and false negatives. In this case, the F1 score is close to 1, indicating that the model performs excellently in this regard.

These results indicate that the Random Forest model performs exceptionally well in classifying whether malware is benign or malicious in static analysis. The model's perfect training accuracy and near-perfect test accuracy, coupled with a high F1 score, demonstrate its effectiveness in this binary classification task. The confusion matrix further confirms its ability to distinguish between benign and malicious samples with high precision and recall.

2.4.2 Dynamic Analysis

2.4.2.1 Direct Categorial Classification With RF – Fixed Parameters

Our first attempt at dynamic analysis was with the RF algorithm, aiming to predict the category of the malware directly with fixed parameters. The target variable was structured as "MalwareType-MalwareFamily," such as "Spyware-CWS." The training-testing split ratio is 80:20.

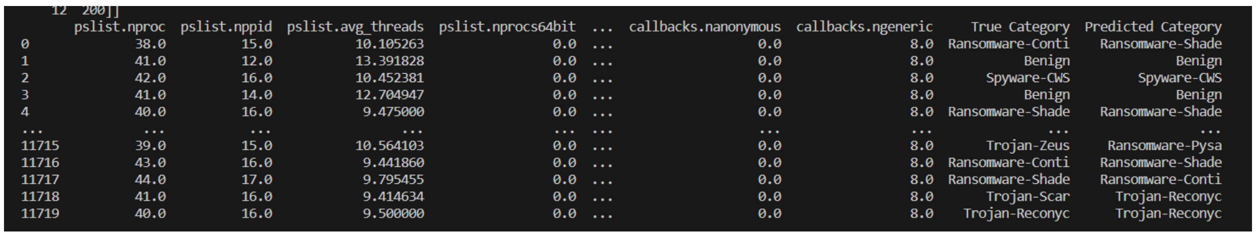
Below is a table that details some of the key parameters used in our Random Forest model and explains how they can affect the results:

Parameter	**Value**	**Description**
n_estimators	200	The number of trees in the forest. Increasing this number generally improves the model's predictive performance, but also increases computational demand.
random_state	42	This is used to guarantee that the split you generate is reproducible. It does this by defining the random number generation for dataset shuffling.
learning_rate	0.1	The learning rate shrinks the contribution of each tree. There is a trade-off between learning_rate and n_estimators.

However, this approach encountered some limitations:

- The RF model produced an accuracy of 75.73%, a figure that left room for substantial improvement.
- The model's structure did not allow for individual probabilities to be displayed for each possible label (malicious, benign, Trojan, Spyware, Ransomware, etc.), and more specifically, the probability of the dump being part of a particular malware family.

- In some instances, there were misclassifications for both type and family. For instance, the model predicted a sample to be "Ransomware-Pysa" when it was actually "Trojan-Zeus."



```
    12  200]]
       pslist.nproc  pslist.nppid  pslist.avg_threads  pslist.nprocs64bit  ...  callbacks.nanonymous  callbacks.ngeneric     True Category  Predicted Category
0              38.0          15.0           10.105263                 0.0  ...                   0.0                 8.0  Ransomware-Conti    Ransomware-Shade
1              41.0          12.0           13.391828                 0.0  ...                   0.0                 8.0            Benign              Benign
2              42.0          16.0           10.452381                 0.0  ...                   0.0                 8.0       Spyware-CWS         Spyware-CWS
3              41.0          14.0           12.704947                 0.0  ...                   0.0                 8.0            Benign              Benign
4              40.0          16.0            9.475000                 0.0  ...                   0.0                 8.0  Ransomware-Shade    Ransomware-Shade
...             ...           ...                 ...                 ...  ...                   ...                 ...               ...                 ...
11715          39.0          15.0           10.564103                 0.0  ...                   0.0                 8.0       Trojan-Zeus     Ransomware-Pysa
11716          43.0          16.0            9.441860                 0.0  ...                   0.0                 8.0  Ransomware-Conti    Ransomware-Shade
11717          44.0          17.0            9.795455                 0.0  ...                   0.0                 8.0  Ransomware-Shade    Ransomware-Conti
11718          41.0          16.0            9.414634                 0.0  ...                   0.0                 8.0       Trojan-Scar      Trojan-Reconyc
11719          40.0          16.0            9.500000                 0.0  ...                   0.0                 8.0    Trojan-Reconyc      Trojan-Reconyc
```

Image: The confusion matrix of the first model. Note that it predicted Ransomware-Pysa where the actual category was Trojan-Zeus. This approach did not allow us to understand specific probability of the sample being in specific category.

Upon reflection, we recognized the potential benefits of adopting a three-tiered model structure. The first tier would determine if the sample is malware or not. If it is, the second tier would classify the type of malware (e.g., Trojan, Spyware, Ransomware), and finally, the third tier would specify the malware's family within the previously classified type. This approach would enable a more granular understanding of each memory dump and allow the presentation of individual probabilities at each stage of the prediction.

2.4.2.2 Experimental: Sequential Categorial Classification With CNN – Fixed Parameters

While the Random Forest (RF) model is well-suited for the size of our dataset and the number of features present, it is essential to explore other potential modeling approaches to ensure we've adopted the optimal solution. Consequently, we decided to experiment with Convolutional Neural Networks (CNNs), a deep learning model typically used for analyzing visual imagery. The goal was to evaluate the performance of a CNN in this context and compare it against the performance of the RF model.

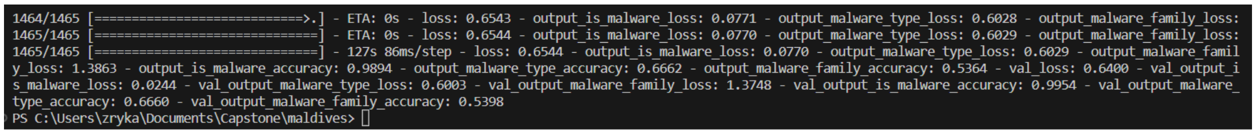
Although CNNs are commonly used for image classification tasks, they can also be effective for other types of data that have a grid-like topology, such as time-series data and other multi-dimensional numerical datasets like ours. The convolution operation allows CNNs to automatically and adaptively learn spatial hierarchies of features, which may potentially enhance the model's ability to recognize intricate patterns in the data.

For the CNN model, we employed various configurations of Dense Layers, set different epoch times, and utilized loss weights. The training-testing split ratio is 80:20. Below are some of the key parameters:

Parameter	**Value**	**Description**
Dense Layer	16	These are the regular deeply connected neural network layers. The number of Dense layers and the number of neurons in each layer can dramatically affect the model's learning capacity and computational demand.
Epoch Times	10	An epoch is one complete pass through the entire training dataset. The number of epochs can be set according to the complexity of the task and the capacity of the model. Too many epochs can lead to overfitting.

Loss Weights	0.2, 0.6, 0.2	These are used to give more importance to certain classes over others in the cost function that the network optimizes. In our case they refer to the loss weight of accuracy of is/is not malware, malware type and malware family and we emphasize the type of malware in this case

The experimental model we created utilized these parameters. However, despite achieving 99% accuracy for the 'is/is not malware' classification, the model performed poorly for malware type and family classifications, with accuracies of 66% and 54% respectively. We found that the loss weights didn't improve the model as much as we'd hoped, and we attributed the lackluster performance to this.



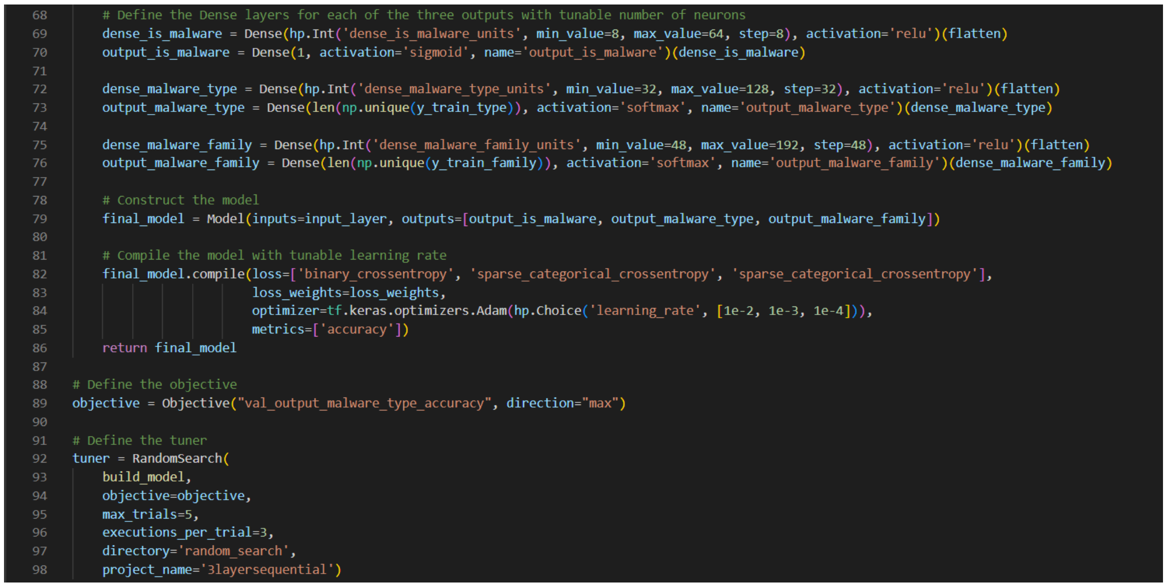
```
1464/1465 [===========================>.] - ETA: 0s - loss: 0.6543 - output_is_malware_loss: 0.0771 - output_malware_type_loss: 0.6028 - output_malware_family_loss:
1465/1465 [==============================] - ETA: 0s - loss: 0.6544 - output_is_malware_loss: 0.0770 - output_malware_type_loss: 0.6029 - output_malware_family_loss:
1465/1465 [==============================] - 127s 86ms/step - loss: 0.6544 - output_is_malware_loss: 0.0770 - output_malware_type_loss: 0.6029 - output_malware_famil
y_loss: 1.3863 - output_is_malware_accuracy: 0.9894 - output_malware_type_accuracy: 0.6662 - output_malware_family_accuracy: 0.5364 - val_loss: 0.6400 - val_output_i
s_malware_loss: 0.0244 - val_output_malware_type_loss: 0.6003 - val_output_malware_family_loss: 1.3748 - val_output_is_malware_accuracy: 0.9954 - val_output_malware_
type_accuracy: 0.6660 - val_output_malware_family_accuracy: 0.5398
PS C:\Users\zryka\Documents\Capstone\maldives> []
```

Image: The model showing accuracy for the 3 sequential predictions (is/is not malware, malware type, malware family) based on fixed parameter CNN model.

2.4.2.3 Experimental: Sequential Categorial Classification With CNN – Hyperparameter Tuner

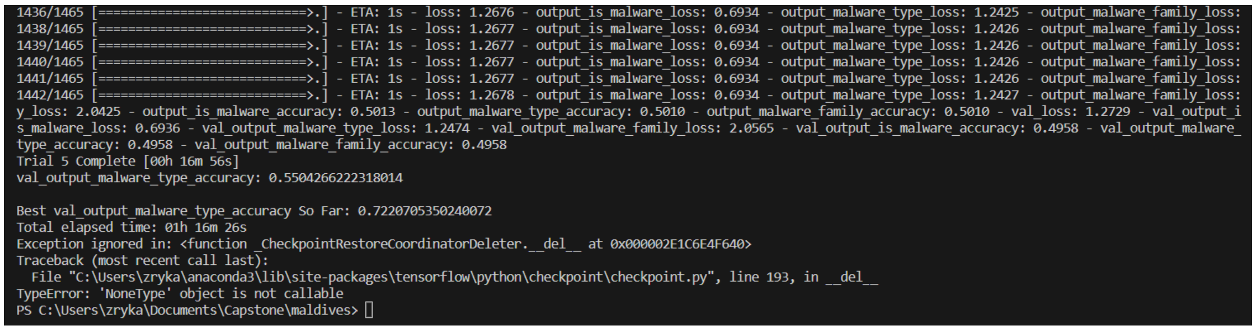
In this experiment, we used a hyperparameter tuner to let the model decide the best parameters, as opposed to manually setting them as done before. The tuner used here is a Random Search, which selects random combinations of hyperparameters to train the model and picks the best performing set.

This approach yielded an improvement in the classification accuracy for malware type, increasing it to 72%. Although this is a significant improvement compared to the fixed-parameter CNN model, it is evident that this is the limit for a CNN model given our dataset's characteristics. This experiment provided valuable insights into the relative strengths of different models when applied to our specific task. As a result, it has guided our decision-making for the final model selection and refinement.



```python
    # Define the Dense layers for each of the three outputs with tunable number of neurons
    dense_is_malware = Dense(hp.Int('dense_is_malware_units', min_value=8, max_value=64, step=8), activation='relu')(flatten)
    output_is_malware = Dense(1, activation='sigmoid', name='output_is_malware')(dense_is_malware)

    dense_malware_type = Dense(hp.Int('dense_malware_type_units', min_value=32, max_value=128, step=32), activation='relu')(flatten)
    output_malware_type = Dense(len(np.unique(y_train_type)), activation='softmax', name='output_malware_type')(dense_malware_type)

    dense_malware_family = Dense(hp.Int('dense_malware_family_units', min_value=48, max_value=192, step=48), activation='relu')(flatten)
    output_malware_family = Dense(len(np.unique(y_train_family)), activation='softmax', name='output_malware_family')(dense_malware_family)

    # Construct the model
    final_model = Model(inputs=input_layer, outputs=[output_is_malware, output_malware_type, output_malware_family])

    # Compile the model with tunable learning rate
    final_model.compile(loss=['binary_crossentropy', 'sparse_categorical_crossentropy', 'sparse_categorical_crossentropy'],
                        loss_weights=loss_weights,
                        optimizer=tf.keras.optimizers.Adam(hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4])),
                        metrics=['accuracy'])
    return final_model

# Define the objective
objective = Objective("val_output_malware_type_accuracy", direction="max")

# Define the tuner
tuner = RandomSearch(
    build_model,
    objective=objective,
    max_trials=5,
    executions_per_trial=3,
    directory='random_search',
    project_name='3layersequential')
```

Image: code snippet of the defined hyperparameters and tuner



```
1436/1465 [============================>.] - ETA: 1s - loss: 1.2676 - output_is_malware_loss: 0.6934 - output_malware_type_loss: 1.2425 - output_malware_family_loss:
1438/1465 [============================>.] - ETA: 1s - loss: 1.2677 - output_is_malware_loss: 0.6934 - output_malware_type_loss: 1.2426 - output_malware_family_loss:
1439/1465 [============================>.] - ETA: 1s - loss: 1.2677 - output_is_malware_loss: 0.6934 - output_malware_type_loss: 1.2426 - output_malware_family_loss:
1440/1465 [============================>.] - ETA: 1s - loss: 1.2677 - output_is_malware_loss: 0.6934 - output_malware_type_loss: 1.2426 - output_malware_family_loss:
1441/1465 [============================>.] - ETA: 1s - loss: 1.2677 - output_is_malware_loss: 0.6934 - output_malware_type_loss: 1.2426 - output_malware_family_loss:
1442/1465 [============================>.] - ETA: 1s - loss: 1.2678 - output_is_malware_loss: 0.6934 - output_malware_type_loss: 1.2427 - output_malware_family_loss:
y_loss: 2.0425 - output_is_malware_accuracy: 0.5013 - output_malware_type_accuracy: 0.5010 - output_malware_family_accuracy: 0.5010 - val_loss: 1.2729 - val_output_i
s_malware_loss: 0.6936 - val_output_malware_type_loss: 1.2474 - val_output_malware_family_loss: 2.0565 - val_output_is_malware_accuracy: 0.4958 - val_output_malware_
type_accuracy: 0.4958 - val_output_malware_family_accuracy: 0.4958
Trial 5 Complete [00h 16m 56s]
val_output_malware_type_accuracy: 0.5504266222318014

Best val_output_malware_type_accuracy So Far: 0.7220705350240072
Total elapsed time: 01h 16m 26s
Exception ignored in: <function _CheckpointRestoreCoordinatorDeleter.__del__ at 0x000002E1C6E4F640>
Traceback (most recent call last):
  File "C:\Users\zryka\anaconda3\lib\site-packages\tensorflow\python\checkpoint\checkpoint.py", line 193, in __del__
TypeError: 'NoneType' object is not callable
PS C:\Users\zryka\Documents\Capstone\maldives> 
```

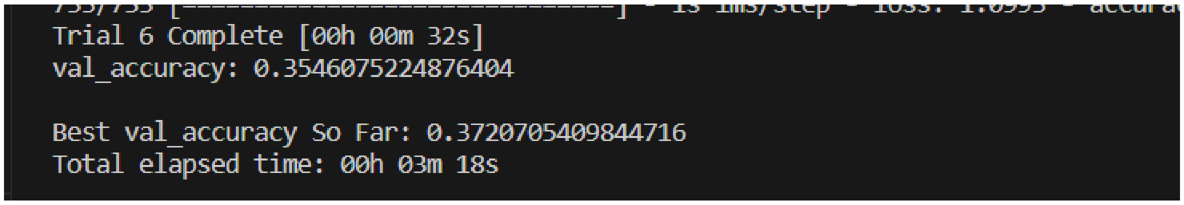
Image: training result of Sequential Categorial Classification with hyperparameter tuner

While the CNN model provided us with some performance improvements, it's important to address its limitations as well:

- **High computational cost**: Training a CNN model can be resource-intensive and time-consuming, especially when the dataset is large and the model architecture is complex.
- **High resource requirements**: Even with the use of a GPU, which significantly accelerates the training process, it still takes considerably longer to train a CNN model than a traditional machine learning model such as Random Forest.
- **Large model size**: CNN models, particularly those with deep architectures, tend to result in large model sizes. This decreases their portability and can be a limiting factor in situations where model deployment needs to be lightweight or transferred across different platforms.
- **Layered Complexity**: CNNs are deep models with multiple layers of neurons. Each layer learns different levels of abstractions of the data. For instance, in an image recognition task, lower layers might learn to detect edges and textures, while deeper layers could learn more abstract concepts like shapes or specific object parts. This hierarchy of learned features makes it challenging to directly interpret what the model has learned.
- **Black-Box Nature**: CNNs, like many deep learning models, are often referred to as "black boxes". This means that while we can clearly see the inputs and outputs of the model, the process in between – the way the model arrives at its decisions – is obscured.

2.4.2.4 Experimental: Categorial Classification With MLP – Hyperparameter Tuner

A Multilayer Perceptron (MLP) is a class of feedforward artificial neural network that consists of at least three layers of nodes: an input layer, a hidden layer, and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLPs are suitable for classification prediction problems where inputs are assigned to one of two or more mutually exclusive classes.



```
733/733 [==============================] - 1s 1ms/step - loss: 1.0993 - accura
Trial 6 Complete [00h 00m 32s]
val_accuracy: 0.3546075224876404

Best val_accuracy So Far: 0.3720705409844716
Total elapsed time: 00h 03m 18s
```

Image: result of MLP model, best accuracy we achieved was 37.20%

Despite the fact that MLPs are considered a robust tool for dealing with complex classification problems, the performance of our MLP model was quite low in this use case, only achieving an accuracy rate of 37.20%. There might be several reasons for this subpar performance:

- **Inadequate model architecture**: The MLP model's architecture might not have been sufficiently complex to capture the intricacies of the dataset. The number of hidden layers and the number of neurons in each hidden layer can greatly impact the performance of an MLP model, and we may not have found the optimal structure.
- **Overfitting and underfitting**: If the MLP model was too complex or if it was trained for too many epochs, it might have overfitted to the training data, performing poorly when tested on new, unseen data and vice versa
- **Data issues**: The dataset might not be suitable for MLPs. For instance, if the features in the dataset do not have clear linear separability or discernable patterns, MLPs may struggle to make accurate predictions.

These potential issues highlight the challenges associated with using MLPs and underline the need for careful model selection and hyperparameter tuning when dealing with complex classification problems.

Given the limited timeframe and the subpar performance of the MLP model in our initial experiments, we decided to shift our focus to other, more promising models rather than invest additional time in further refining and optimizing the MLP model.

2.4.2.5 Experimental: Categorial Classification With SVM – Hyperparameter Tuner

We then decided to employ Support Vector Machines (SVM), a supervised machine learning model that is often used in classification problems. SVM works by identifying the optimal hyperplane that best separates different classes within a given dataset. It defines decision boundaries using a subset of training points and maximizes the margin between different classes. SVM is versatile and can handle linear as well as non-linear data using kernels.

In our case, we used SVM for classification and experimented with a hyperparameter tuner to optimize the model's performance. Despite these efforts, the resulting precision and F1 score were 0.66 and 0.65 respectively, indicating that while the model was somewhat successful in categorizing the malware, there was still room for improvement. These scores suggest that our SVM model could benefit from further parameter tuning or perhaps a more in-depth feature selection process, which may enhance the model's ability to distinguish between various categories of malware more accurately.

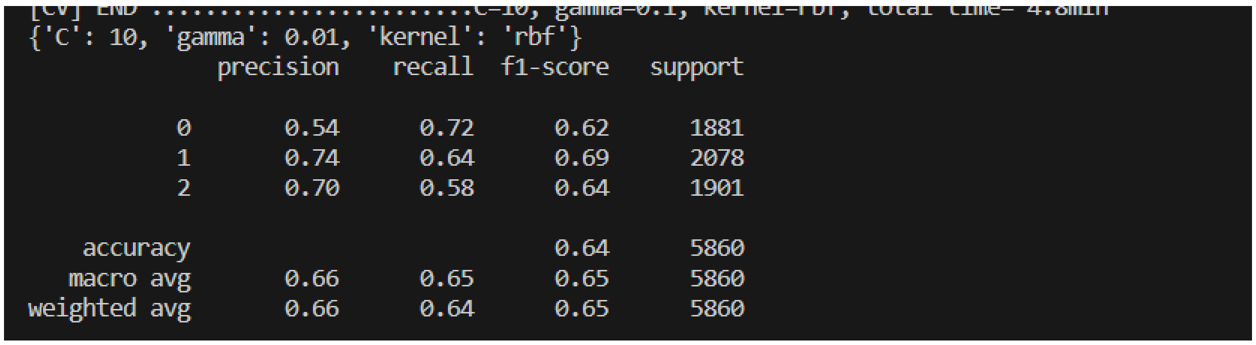

```
{'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}
              precision    recall  f1-score   support

           0       0.54      0.72      0.62      1881
           1       0.74      0.64      0.69      2078
           2       0.70      0.58      0.64      1901

    accuracy                           0.64      5860
   macro avg       0.66      0.65      0.65      5860
weighted avg       0.66      0.64      0.65      5860
```

Image: SVM model result on predicting the malware type

<u>2.4.2.6 Best Model: Sequential Categorial Classification With RF – Hyperparameter Tuner</u>

Random Forest (RF) was the most effective model in our experiments. Random Forest is an ensemble learning method that operates by constructing multiple decision trees during training and outputting the mode of the classes for classification or mean prediction for regression. Its strengths lie in its capability to handle a large number of features, resistance to overfitting, and robustness to outliers and non-linear data.

In our binary classification task of distinguishing between malicious and benign programs, RF achieved an accuracy of 99.99%, backed by a cross-validation score of 1.00. This outcome demonstrates the model's excellent ability to distinguish between benign and malicious entities on both trained data and unseen data.

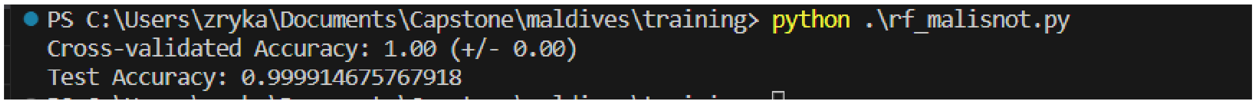

```
PS C:\Users\zryka\Documents\Capstone\maldives\training> python .\rf_malisnot.py
Cross-validated Accuracy: 1.00 (+/- 0.00)
Test Accuracy: 0.999914675767918
```

Image: Snippet of training result

For the categorical classification task, which involves determining the type of malware, the RF model achieved a weighted accuracy of 77%.

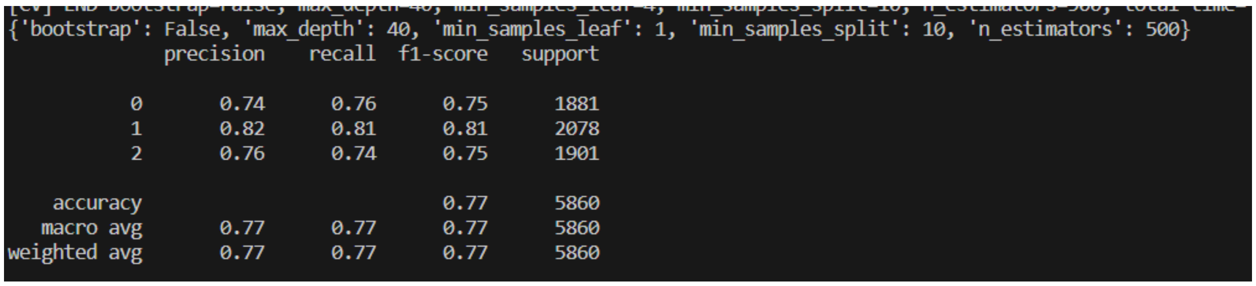
Within each type of malware, the model's success in correctly identifying the family varied.

The accuracies achieved were as follows:

Classification Level	Category	Accuracy
Binary Classification	Malicious or Not	99.99%
Categorical Classification	Malware Type	77%
----	Trojan	--

----	Spyware	--
----	Ransomware	--
Categorical Classification	Malware Family	
	Trojan Family	76%
----	Spyware Family	67%
----	Ransomware Family	59%

These differing levels of accuracy can be attributed to the characteristics of the malware types themselves. Some malware families within a specific type may have similar behaviors or patterns, which can make distinguishing between them more challenging.



```
{'bootstrap': False, 'max_depth': 40, 'min_samples_leaf': 1, 'min_samples_split': 10, 'n_estimators': 500}
              precision    recall  f1-score   support

           0       0.74      0.76      0.75      1881
           1       0.82      0.81      0.81      2078
           2       0.76      0.74      0.75      1901

    accuracy                           0.77      5860
   macro avg       0.77      0.77      0.77      5860
weighted avg       0.77      0.77      0.77      5860
```

Image: Snippet of training result for malware type classification. The hyperparameters tuner selected best parameters as shown in the first line.

Selecting RF as our final model choice is backed by several reasons:

- **Robustness**: Random Forest is resistant to overfitting, which is a significant advantage in a field like cybersecurity, where new data is continuously being generated.
- **Feature Handling**: Random Forest can effectively manage large feature spaces, making it suitable for our dataset with a considerable number of features.
- **Versatility**: Random Forest can handle both categorical and numerical data, enabling us to effectively use all of the data provided in the dataset.
- **Efficiency**: Compared to deep learning models, RF is less computationally intensive, making it a more practical choice for systems with limited computational resources.
- **Interpretability**: While not as interpretable as a single decision tree, RF still retains a level of interpretability. Feature importance can be derived, which can offer insights into what features are driving the model's decisions.

Given these factors, Random Forest was the clear choice for our final model.

2.5 Training

The training phase is a critical stage in the machine learning pipeline. This phase involves passing the input data to a machine learning algorithm which 'learns' from this data by adjusting its internal parameters. For our study, we chose the Random Forest Classifier model due to its robustness and excellent performance on similar tasks. To fine-tune the performance of the model, we employed a method known as Grid Search Cross Validation.

2.5.1 Hyperparameters Setting

For optimization, we specify several hyperparameters. Hyperparameters are variables that define the model structure or indicate how the model is supposed to learn. These parameters are set before the learning process begins.

In this project, the parameters being optimized are:

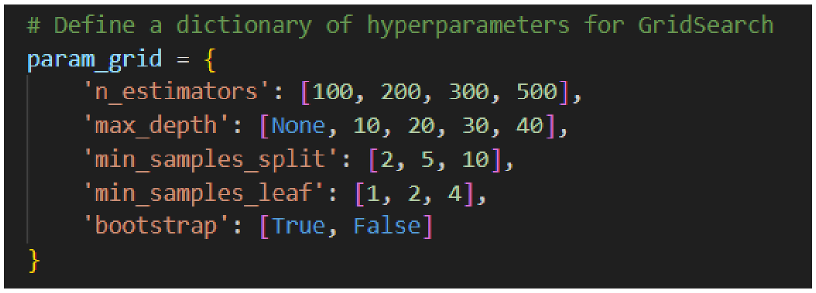
- **n_estimators**: The number of trees in the forest.
- **max_depth**: The maximum depth of the tree.
- **min_samples_split**: The minimum number of samples required to split an internal node.
- **min_samples_leaf**: The minimum number of samples required to be at a leaf node.
- **bootstrap**: Whether bootstrap samples are used when building trees.

The following table outlines the hyperparameters we targeted in our optimization process for detecting malware types:

Hyperparameter	**Description**	**Values**
n_estimators	The number of trees in the forest.	100, 200, 400, 500
max_depth	The maximum depth of the tree.	None, 10, 20, 40
min_samples_split	The minimum number of samples required to split an internal node.	2, 10, 20, 30
min_samples_leaf	The minimum number of samples required to be at a leaf node.	1, 2
bootstrap	Whether bootstrap samples are used when building trees.	True, False

Recognizing that different malware families might possess unique characteristics and behaviors, we made the strategic decision to optimize the hyperparameters for each malware family independently. This approach is based on the premise that the optimal model configuration may vary according to the specific patterns and structures present in different malware families. For instance, Trojan, Spyware, and Ransomware may each require a unique combination of n_estimators, max_depth, min_samples_split, and min_samples_leaf to achieve the best classification performance.

By iteratively testing and refining our hyperparameters for each malware family, we fine-tuned our model to capture the distinct characteristics of Trojan, Spyware, and Ransomware more effectively. This hyperparameter tuning process was both systematic and rigorous, involving numerous training rounds and careful monitoring of the model's performance at each stage. As a result of this meticulous optimization, our model's ability to identify and classify each malware family was significantly enhanced. This underlines the importance of a targeted, flexible approach when configuring machine learning models for complex, multifaceted tasks like malware classification.



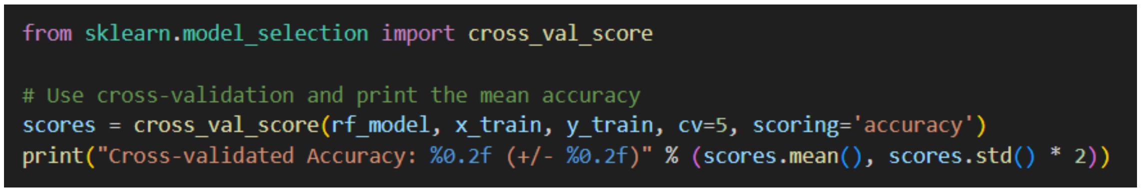
```python
# Define a dictionary of hyperparameters for GridSearch
param_grid = {
    'n_estimators': [100, 200, 300, 500],
    'max_depth': [None, 10, 20, 30, 40],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}
```

Image: Code snippet of hyperparameters for ransomware family detection

2.5.3 Grid Search Cross Validation

We use Grid Search to optimize these hyperparameters. Grid Search is a tuning technique that exhaustively tries every combination of the provided hyperparameter values to find the best model. To apply Grid Search, we use the GridSearchCV() function from the Scikit-learn library.

This function performs a cross-validated grid-search over a parameter grid. We specify the number of folds for cross-validation via cv=3, meaning the training set is split into 3 subsets and the model is trained and tested 3 times, each time with a different subset held out as the test set.



```python
from sklearn.model_selection import cross_val_score

# Use cross-validation and print the mean accuracy
scores = cross_val_score(rf_model, x_train, y_train, cv=5, scoring='accuracy')
print("Cross-validated Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

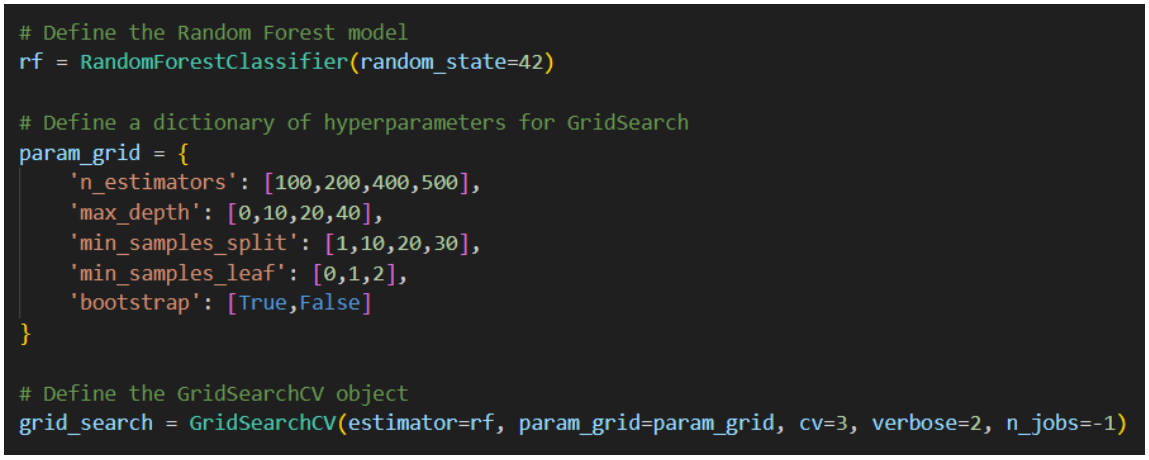
Image: Code snippet of cross validation

2.5.4 Model Training

Finally, the model is trained using the fit() function. This function adjusts the internal parameters of the model, in this case, the Random Forest, based on the input data and the selected hyperparameters. The training process uses cross-validation to evaluate the performance of each set of hyperparameters on the training data.

Ultimately, the best set of hyperparameters, i.e., those that produce the highest performing model on the training data, is selected and retained for the final model.

Through this exhaustive and systematic approach, we ensure the optimal performance of our model by selecting the most fitting hyperparameters.



```python
# Define the Random Forest model
rf = RandomForestClassifier(random_state=42)

# Define a dictionary of hyperparameters for GridSearch
param_grid = {
    'n_estimators': [100,200,400,500],
    'max_depth': [0,10,20,40],
    'min_samples_split': [1,10,20,30],
    'min_samples_leaf': [0,1,2],
    'bootstrap': [True,False]
}

# Define the GridSearchCV object
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=3, verbose=2, n_jobs=-1)
```

Image: Code snippet of defining RF model with grid search.

2.6 Design

The design of our MalDives tool was primarily driven by the dual goals of efficiency and user-friendliness, while also considering the constraints of time and budget. The architecture of the tool allows users to perform both dynamic and static analysis with the same command-line interface, offering flexibility and convenience.

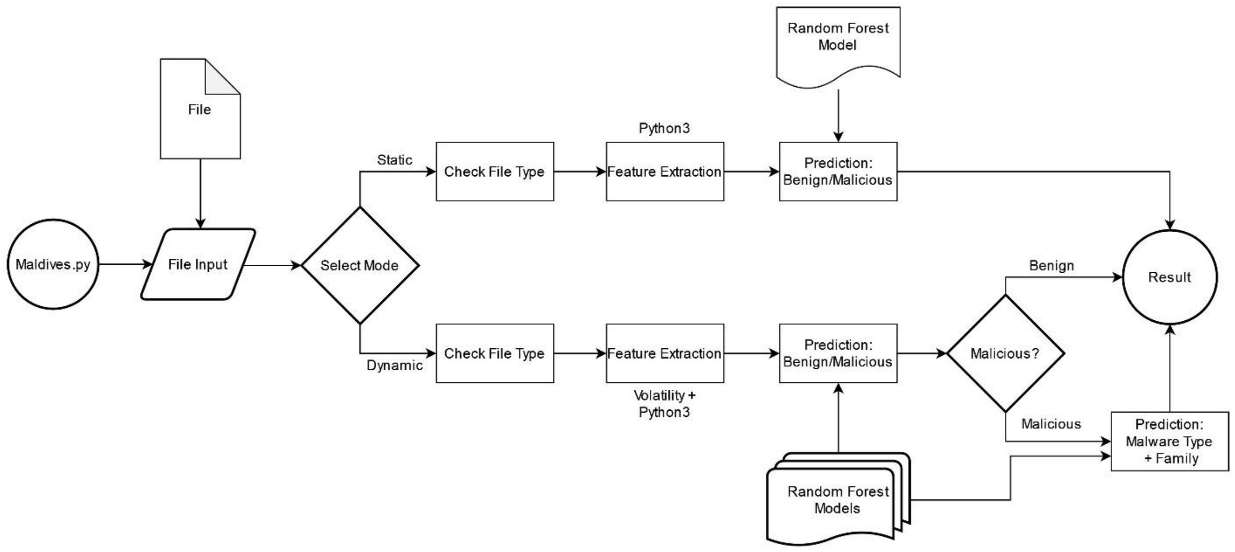


Image: Design flow chart of MalDives

The user initiates the process by executing the MalDives.py script and supplying a file path as an argument, along with a flag to denote the mode of analysis. For instance, the **command python3 MalDives.py -d /path/to/image/file** triggers dynamic analysis, while replacing **-d** with **-s** initiates static analysis. This design choice simplifies the process of selecting the type of analysis and provides a straightforward, consistent user experience.

Upon receiving the user's input, the tool checks the type of the provided file as a first step to minimize errors. This built-in validation mechanism ensures that the file is suitable for the selected mode of analysis and helps prevent unnecessary computation or inaccurate results.

If the user selects static analysis, the tool performs feature extraction on the given file and uses the trained model to predict whether the file is malicious or benign. In contrast, dynamic analysis not only determines if the memory dump shows infected sign, but also identifies the specific type of malware and its potential family. This distinction provides users with more detailed, valuable information about the potential threat, aiding in further investigation or remediation efforts.

The tool then displays the results, along with the corresponding probabilities, providing the user with a quantified level of confidence in the prediction. This approach prioritizes transparency and helps users to make informed decisions based on the output of the tool.

The design of the MalDives tool reflects our intent to create a user-friendly and reliable system capable of complex malware analysis tasks, despite constraints in time and budget. By balancing simplicity with comprehensive functionality, we aim to provide a powerful yet accessible tool that can prove the concept of effective and efficient malware detection and classification.

2.7 Feature Extraction

Feature extraction is a critical aspect of machine learning, especially in the domain of malware analysis where understanding the intricacies of data holds paramount importance. By extracting relevant features from a given dataset, we can transform raw data into an informative set that aids in accurate model training and prediction. The goal is to capture the essence of the data, drawing out attributes that define the characteristics of malware, while ignoring noise or irrelevant information.

For our tool, we've adopted a two-pronged approach, focusing on both static and dynamic analyses.

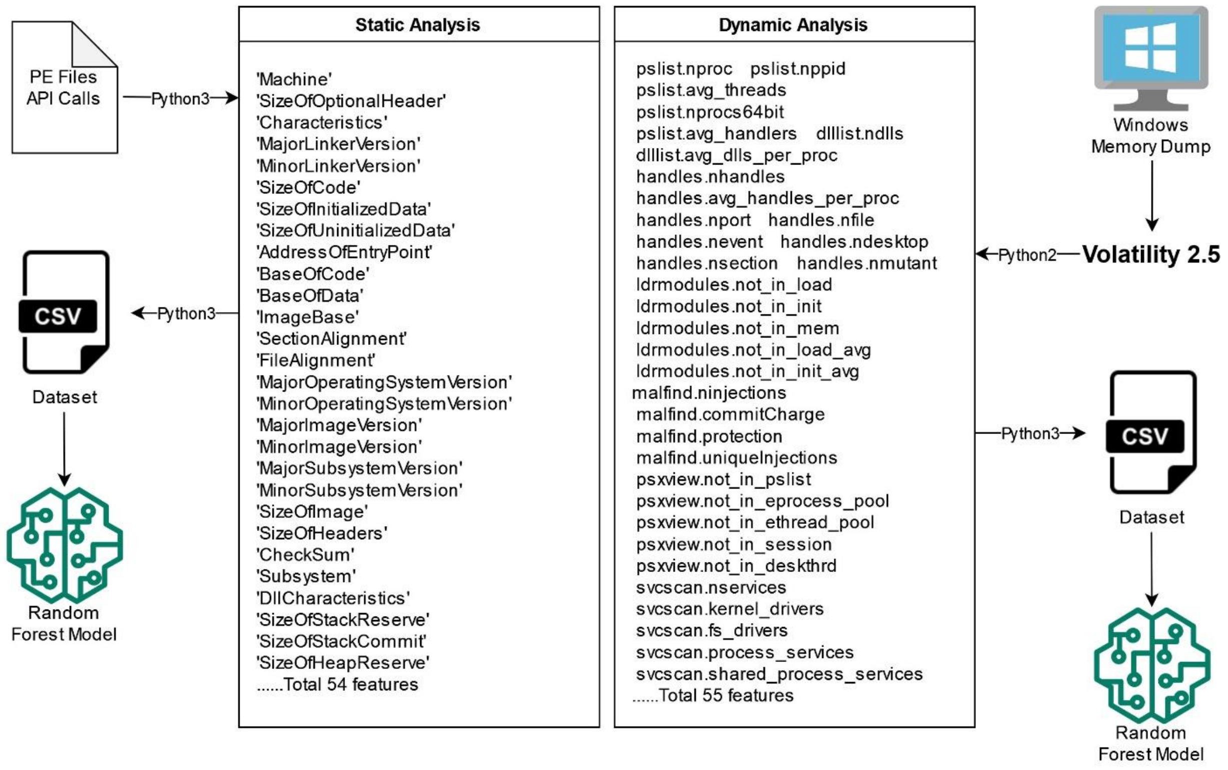


Image: Chart showing how features are extracted and used to train the models

2.7.1 *Static Analysis*

Static feature extraction pertains to analyzing the attributes of a program without executing it. For our purposes, a total of 54 API calls were extracted from the win32 winnt.h header. This set includes key identifiers such as the major and minor linker version, major and minor image version, size of stack (both reserve and commit), heap size, and more. Each of these API calls can be seen as a building block, capable of performing a variety of system-level operations.

These operations can span a wide range of tasks — from accessing hardware resources and modifying system settings to reading and writing files. Malicious actors often exploit these operations to craft sophisticated malware that can inject malicious code, escalate privileges, maintain persistence in a system, and even evade traditional detection methods. Hence, understanding and extracting these static features provides a glimpse into the potential actions a program might undertake, without the need to execute it.

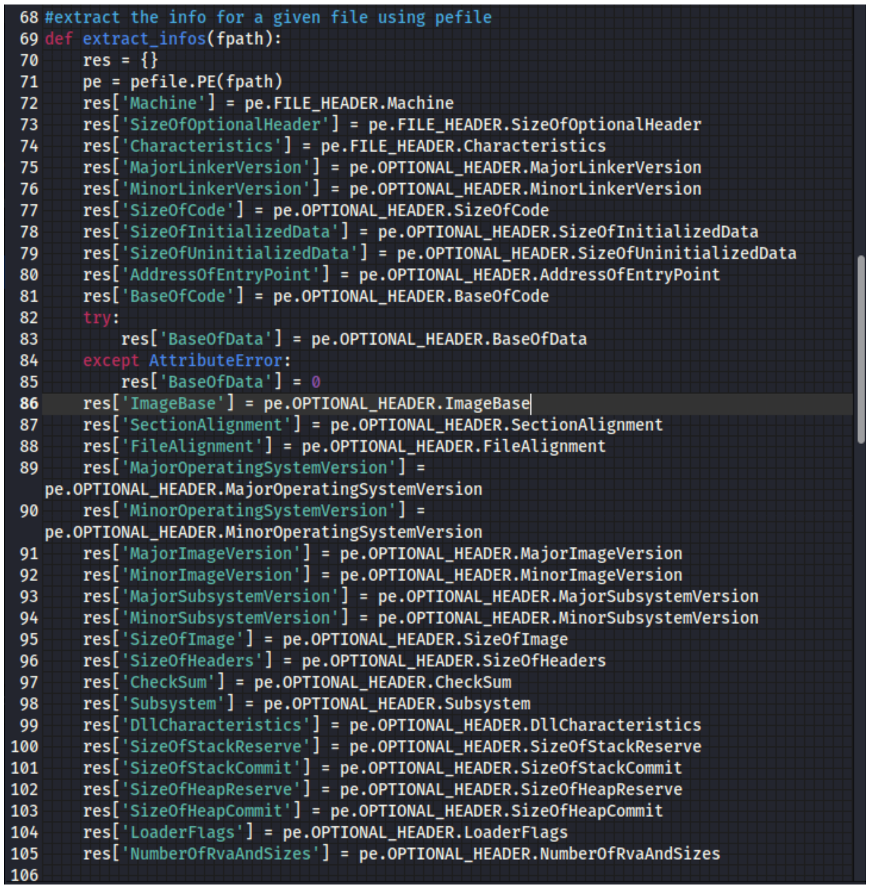
The table below offers a concise overview of the features we extracted:

Feature	Description
Machine	Specifies the architecture type of the computer
SizeOfOptionalHeader	The size of the optional header

Characteristics	A set of flags describing file characteristics
MajorLinkerVersion	Major version numbers of the linker
MinorLinkerVersion	Minor version numbers of the linker
SizeOfCode	The size of the code section
SizeOfInitializedData	The size of initialized data
SizeOfUninitializedData	The size of uninitialized data
AddressOfEntryPoint	The address of the entry point where program execution begins
BaseOfCode	Base addresses of the code segments
BaseOfData	Base addresses of the data segments
ImageBase	The preferred load address of the executable in memory
SectionAlignment	Alignment of sections in memory
FileAlignment	Alignment of sections in file
MajorOperatingSystemVersion	Major version numbers of the operating system
MinorOperatingSystemVersion	Minor version numbers of the operating system
MajorImageVersion	Major version numbers of the image
MinorImageVersion	Minor version numbers of the image
MajorSubsystemVersion	Major version numbers of the subsystem
MinorSubsystemVersion	Minor version numbers of the subsystem
SizeOfImage	The size of the image
SizeOfHeaders	The size of all headers
CheckSum	Checksum
Subsystem	The subsystem required to execute the executable
DllCharacteristics	The subsystem required to execute the executable
SizeOfStackReserve	Reserve size of the stack
SizeOfStackCommit	Commit size of the stack
SizeOfHeapReserve	Reserve size of the heap
SizeOfHeapCommit	Commit size of the heap
LoaderFlags	Flags related to the loader
NumberOfRvaAndSizes	Number of RVAs and sizes
SectionsNb	Number of RVAs and sizes
SectionsMeanEntropy	Entropy statistics of sections (Mean)
SectionsMinEntropy	Entropy statistics of sections (Min)
SectionsMaxEntropy	Entropy statistics of sections (Max)
SectionsMeanRawsize	Raw size statistics of sections (Mean)
SectionsMinRawsize	Raw size statistics of sections (Min)
SectionMaxRawsize	Raw size statistics of sections (Max)
SectionsMeanVirtualsize	Raw size statistics of sections (Mean)
SectionsMinVirtualsize	Raw size statistics of sections (Min)
SectionMaxVirtualsize	Raw size statistics of sections (Max)

ImportsNbDLL	Number of imported DLLs
ImportsNb	Number of imported functions
ImportsNbOrdinal	Number of imported ordinals
ExportNb	Number of exported functions
ResourcesNb	Number of resources
ResourcesMeanEntropy	Entropy statistics of resources (Mean)
ResourcesMinEntropy	Entropy statistics of resources (Min)
ResourcesMaxEntropy	Entropy statistics of resources (Max)
ResourcesMeanSize	Size statistics of resources (Mean)
ResourcesMinSize	Size statistics of resources (Min)
ResourcesMaxSize	Size statistics of resources (Max)
LoadConfigurationSize	Size of load configuration
VersionInformationSize	Size of version information

These features can be used for various purposes, such as executable file analysis, reverse engineering, and malware detection. Particularly in malware detection, these features can aid a classifier in recognizing the different behaviors and characteristics of malicious and benign samples.



```
#extract the info for a given file using pefile
def extract_infos(fpath):
    res = {}
    pe = pefile.PE(fpath)
    res['Machine'] = pe.FILE_HEADER.Machine
    res['SizeOfOptionalHeader'] = pe.FILE_HEADER.SizeOfOptionalHeader
    res['Characteristics'] = pe.FILE_HEADER.Characteristics
    res['MajorLinkerVersion'] = pe.OPTIONAL_HEADER.MajorLinkerVersion
    res['MinorLinkerVersion'] = pe.OPTIONAL_HEADER.MinorLinkerVersion
    res['SizeOfCode'] = pe.OPTIONAL_HEADER.SizeOfCode
    res['SizeOfInitializedData'] = pe.OPTIONAL_HEADER.SizeOfInitializedData
    res['SizeOfUninitializedData'] = pe.OPTIONAL_HEADER.SizeOfUninitializedData
    res['AddressOfEntryPoint'] = pe.OPTIONAL_HEADER.AddressOfEntryPoint
    res['BaseOfCode'] = pe.OPTIONAL_HEADER.BaseOfCode
    try:
        res['BaseOfData'] = pe.OPTIONAL_HEADER.BaseOfData
    except AttributeError:
        res['BaseOfData'] = 0
    res['ImageBase'] = pe.OPTIONAL_HEADER.ImageBase
    res['SectionAlignment'] = pe.OPTIONAL_HEADER.SectionAlignment
    res['FileAlignment'] = pe.OPTIONAL_HEADER.FileAlignment
    res['MajorOperatingSystemVersion'] = 
pe.OPTIONAL_HEADER.MajorOperatingSystemVersion
    res['MinorOperatingSystemVersion'] = 
pe.OPTIONAL_HEADER.MinorOperatingSystemVersion
    res['MajorImageVersion'] = pe.OPTIONAL_HEADER.MajorImageVersion
    res['MinorImageVersion'] = pe.OPTIONAL_HEADER.MinorImageVersion
    res['MajorSubsystemVersion'] = pe.OPTIONAL_HEADER.MajorSubsystemVersion
    res['MinorSubsystemVersion'] = pe.OPTIONAL_HEADER.MinorSubsystemVersion
    res['SizeOfImage'] = pe.OPTIONAL_HEADER.SizeOfImage
    res['SizeOfHeaders'] = pe.OPTIONAL_HEADER.SizeOfHeaders
    res['CheckSum'] = pe.OPTIONAL_HEADER.CheckSum
    res['Subsystem'] = pe.OPTIONAL_HEADER.Subsystem
    res['DllCharacteristics'] = pe.OPTIONAL_HEADER.DllCharacteristics
    res['SizeOfStackReserve'] = pe.OPTIONAL_HEADER.SizeOfStackReserve
    res['SizeOfStackCommit'] = pe.OPTIONAL_HEADER.SizeOfStackCommit
    res['SizeOfHeapReserve'] = pe.OPTIONAL_HEADER.SizeOfHeapReserve
    res['SizeOfHeapCommit'] = pe.OPTIONAL_HEADER.SizeOfHeapCommit
    res['LoaderFlags'] = pe.OPTIONAL_HEADER.LoaderFlags
    res['NumberOfRvaAndSizes'] = pe.OPTIONAL_HEADER.NumberOfRvaAndSizes
```

Image: Code snippet of extracting various static features from a Portable Executable (PE) file using the pefile library

2.7.2 Dynamic Analysis

While static analysis provides insight into what a program could do, dynamic analysis reveals what a program does when it is executed. For our dynamic feature extraction, we've incorporated 55 features, sourced using Volatility. As a potent open-source memory forensics framework, Volatility offers a plethora of plugins — from pslist, dlllist, and handles to service scans. Each of these plugins grants a unique lens to view and understand the captured state of memory.

Malware, upon execution, invariably loads into the system's memory. This trait makes memory a gold mine for investigators as any action the malware undertakes — be it benign or malicious — leaves traces here. By leveraging the power of Volatility, we can capture the behavioral footprint of malware, documenting its actions and intentions, thereby enhancing our ability to detect and classify it.

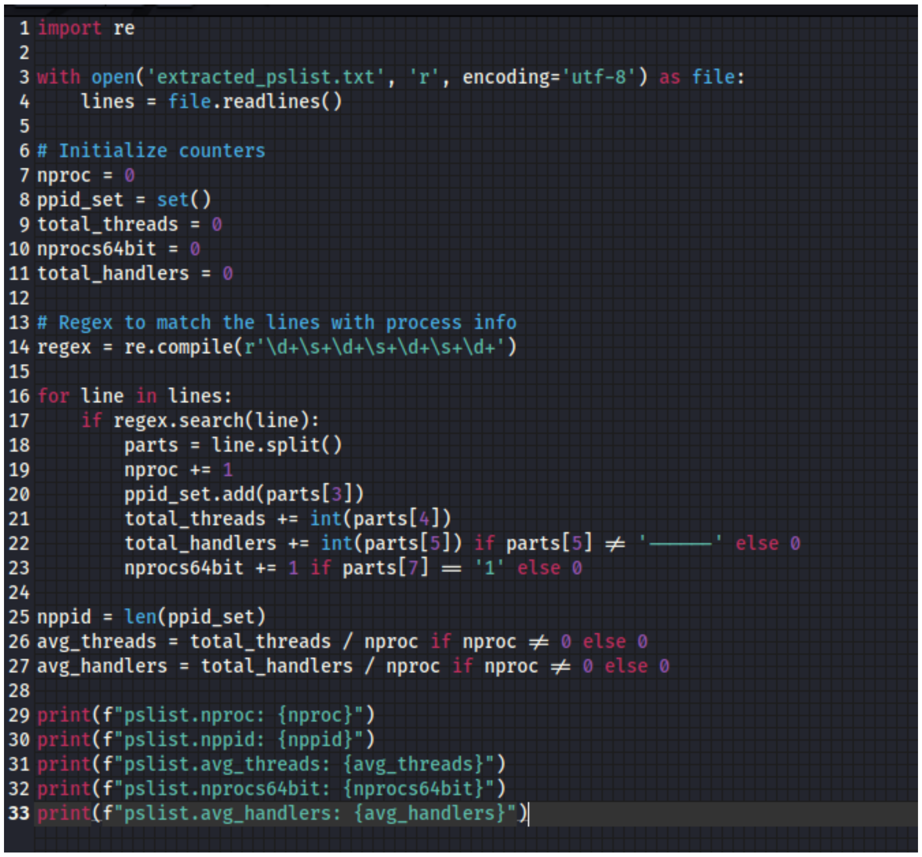
To render our dynamic analysis truly actionable, we leaned on the capabilities of Volatility and its plethora of plugins. However, merely obtaining the raw data from these plugins wouldn't suffice. For our models to work effectively, we needed quantifiable metrics. Consequently, we developed Python scripts tailored to extract specific, quantifiable features from each of the plugins. This streamlined extraction process ensured that our dataset was both consistent and informative.

The table below offers a concise overview of the features we extracted:

Feature	Description
pslist.nproc	Number of processes
pslist.nppid	Number of parent process IDs
pslist.avg_threads	Average number of threads
pslist.nprocs64bit	Number of 64-bit processes
pslist.avg_handlers	Average number of handlers
dlllist.ndlls	Number of dynamic link libraries
dlllist.avg_dlls_per_proc	Average DLLs per process
handles.nhandles	Total handles
handles.avg_handles_per_proc	Average handles per process
handles.nport	Number of port handles
handles.nfile	Number of file handles
handles.nevent	Number of event handles
handles.ndesktop	Number of desktop handles
handles.nkey	Number of key handles
handles.nthread	Number of thread handles
handles.ndirectory	Number of directory handles
handles.nsemaphore	Number of semaphore handles
handles.ntimer	Number of timer handles
handles.nsection	Number of section handles
handles.nmutant	Number of mutant handles
ldrmodules.not_in_load	Modules not in load count
ldrmodules.not_in_init	Modules not in initialization count
ldrmodules.not_in_mem	Modules not in memory count
ldrmodules.not_in_load_avg	Average modules not in load
ldrmodules.not_in_init_avg	Average modules not in initialization

ldrmodules.not_in_mem_avg	Average modules not in memory
malfind.ninjections	Number of injections
malfind.commitCharge	Commit charge of injections
malfind.protection	Protection level of injections
malfind.uniqueInjections	Number of unique injections
psxview.not_in_pslist	Not in process list count
psxview.not_in_eprocess_pool	Not in eprocess pool count
psxview.not_in_ethread_pool	Not in ethread pool count
psxview.not_in_pspcid_list	Not in PspCid list count
psxview.not_in_csrss_handles	Not in csrss handles count
psxview.not_in_session	Not in session count
psxview.not_in_deskthrd	Not in deskthrd count
psxview.not_in_pslist_false_avg	Average false negatives in process list
psxview.not_in_eprocess_pool_false_avg	Average false negatives in eprocess pool
psxview.not_in_ethread_pool_false_avg	Average false negatives in ethread pool
psxview.not_in_pspcid_list_false_avg	Average false negatives in PspCid list
psxview.not_in_csrss_handles_false_avg	Average false negatives in csrss handles
psxview.not_in_session_false_avg	Average false negatives in session
psxview.not_in_deskthrd_false_avg	Average false negatives in deskthrd
modules.nmodules	Total modules
svcscan.nservices	Total services identified
svcscan.kernel_drivers	Kernel drivers count
svcscan.fs_drivers	File system drivers count
svcscan.process_services	Services running as individual processes
svcscan.shared_process_services	Shared process services count
svcscan.interactive_process_services	Interactive process services count
svcscan.nactive	Number of active services
callbacks.ncallbacks	Total callbacks
callbacks.nanonymous	Number of anonymous callbacks

Each of these features paints a segment of the larger picture — the state and behavior of a program loaded into the memory. By effectively extracting these metrics, we set a robust foundation for our models to understand, classify, and predict the nature and intent of the loaded program.



```python
import re

with open('extracted_pslist.txt', 'r', encoding='utf-8') as file:
    lines = file.readlines()

# Initialize counters
nproc = 0
ppid_set = set()
total_threads = 0
nprocs64bit = 0
total_handlers = 0

# Regex to match the lines with process info
regex = re.compile(r'\d+\s+\d+\s+\d+\s+\d+')

for line in lines:
    if regex.search(line):
        parts = line.split()
        nproc += 1
        ppid_set.add(parts[3])
        total_threads += int(parts[4])
        total_handlers += int(parts[5]) if parts[5] ≠ '———' else 0
        nprocs64bit += 1 if parts[7] == '1' else 0

nppid = len(ppid_set)
avg_threads = total_threads / nproc if nproc ≠ 0 else 0
avg_handlers = total_handlers / nproc if nproc ≠ 0 else 0

print(f"pslist.nproc: {nproc}")
print(f"pslist.nppid: {nppid}")
print(f"pslist.avg_threads: {avg_threads}")
print(f"pslist.nprocs64bit: {nprocs64bit}")
print(f"pslist.avg_handlers: {avg_handlers}")
```

Image: Code snippet of processing Volatility pslist plugin output into quantifiable feature used for generating the dataset.

2.8 Building the tool

Drawing from our design in section 2.6, the development of "Maldives" commenced with an emphasis on ensuring it was user-friendly and robust.

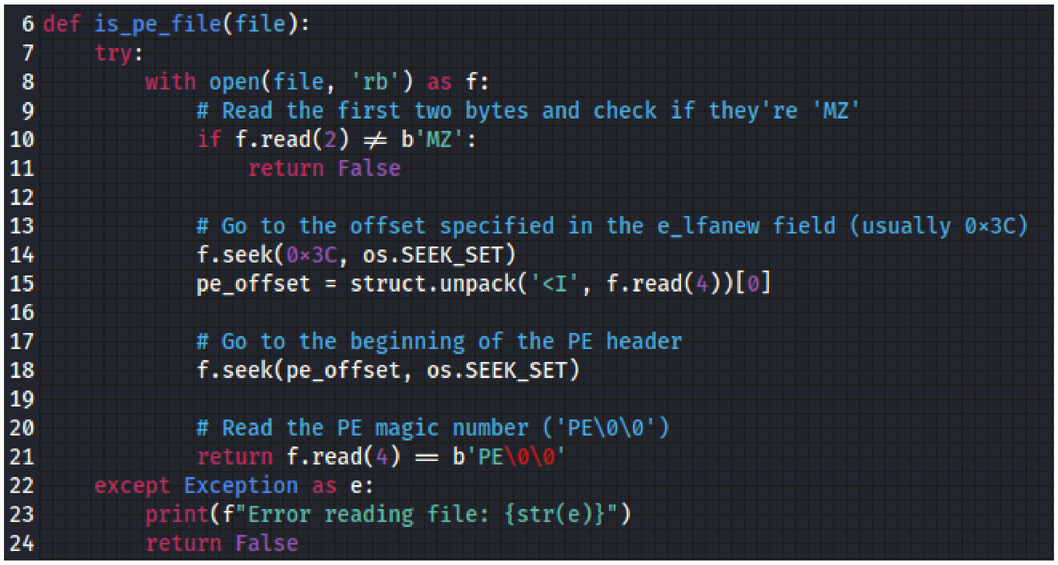
2.8.1 Platform Selection and Trade-offs

Initially, we considered utilizing Amazon Web Services (AWS) to construct an online sandbox environment. The motivation behind this approach was to enhance the security of the tool, ensuring that any malicious samples would be isolated and could not compromise the analyst's machine. An online sandbox also offers scalability and easier distribution of the tool across teams without requiring local installations. However, after evaluating the costs and weighing it against our project budget, we decided to forego this direction. Furthermore, the potential consequences of executing malware in a cloud environment, coupled with the uncertainties surrounding the impact on the cloud infrastructure, made the option less attractive.

2.8.2 File Type Validation

To ensure that our tool processes valid input and minimizes errors, we implemented rigorous file type checks. This approach not only guarantees that the tool operates with the appropriate data but also that analysts are promptly informed when they provide unsuitable input.

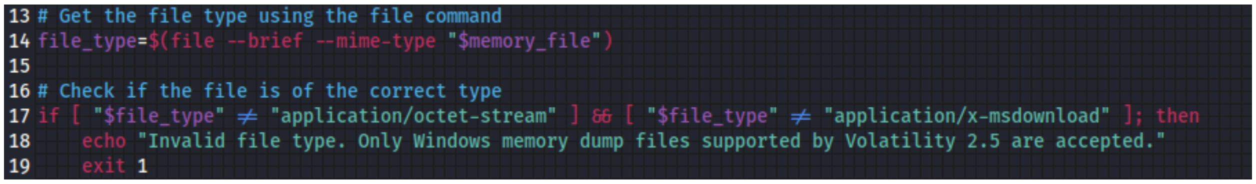
The script uses the file command to deduce the MIME type of the provided file, ensuring that it's a legitimate Windows memory dump.



```python
def is_pe_file(file):
    try:
        with open(file, 'rb') as f:
            # Read the first two bytes and check if they're 'MZ'
            if f.read(2) ≠ b'MZ':
                return False

            # Go to the offset specified in the e_lfanew field (usually 0×3C)
            f.seek(0×3C, os.SEEK_SET)
            pe_offset = struct.unpack('<I', f.read(4))[0]

            # Go to the beginning of the PE header
            f.seek(pe_offset, os.SEEK_SET)

            # Read the PE magic number ('PE\0\0')
            return f.read(4) == b'PE\0\0'
    except Exception as e:
        print(f"Error reading file: {str(e)}")
        return False
```

Image: The snippet how we ensure that only valid memory dump files compatible with Volatility 2.5 are processed

For static analysis, our goal was to process only valid Portable Executable (PE) files. PE files are the common format for executables, dynamic-link libraries, and even some system files on Windows. Ensuring the provided file is a PE helps to filter out irrelevant or incorrectly provided files.



```bash
# Get the file type using the file command
file_type=$(file --brief --mime-type "$memory_file")

# Check if the file is of the correct type
if [ "$file_type" ≠ "application/octet-stream" ] && [ "$file_type" ≠ "application/x-msdownload" ]; then
    echo "Invalid file type. Only Windows memory dump files supported by Volatility 2.5 are accepted."
    exit 1
```

Image: Snippet checking for the magic numbers.

The script specifically checks for the magic numbers typical of PE files (MZ and PE\0\0). These unique sequences guarantee that the file under inspection is a Windows executable or library, which is critical for the accuracy and reliability of the static analysis process.

In conclusion, building "Maldives" was a delicate balance of maintaining functionality while ensuring user-friendliness and accuracy. Through robust validation and thoughtful design considerations, we've tried to provide analysts with a powerful yet easy-to-use tool for malware assessment.

2.9 Testing

The success of any tool not only depends on its design and build but significantly on its rigorous testing. To ensure the efficacy of our tool, we needed real-world malware samples against which "Maldives" could be evaluated.

2.9.1 Acquiring Malware Samples

Our first instinct was to utilize the well-known VirusTotal API for sourcing malware samples. VirusTotal, being a repository of malware signatures and samples, would have provided a comprehensive set of data to test our tool. However, we found that accessing bulk samples from VirusTotal requires a subscription which exceeded our project's budget.

Given this constraint, we turned to alternative platforms: MalwareBazaar and Virus Share. Both platforms have become invaluable resources for researchers, offering a variety of malware samples free of charge. Our focus was primarily on acquiring trojan, spyware, and ransomware samples as they were representative of the families in our training dataset.

The process involved:

Registration & Verification: Both platforms usually require users to create an account and undergo some level of verification to ensure the responsible use of the data.

Browsing & Selection: We filtered and browsed through their collections to find samples that matched our requirements.

Secure Download: Samples were downloaded securely, ensuring that they were stored and handled in isolated environments to prevent any unintentional execution.

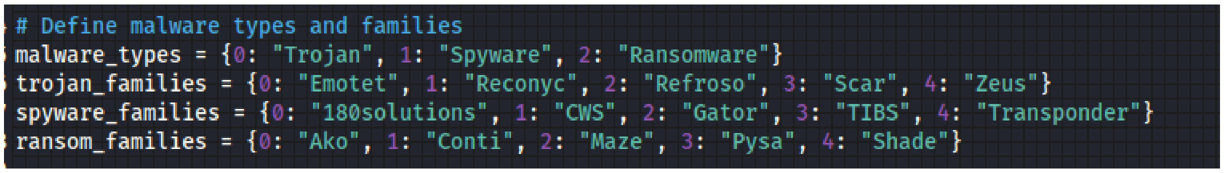
Documentation: Every sample was documented with its source, type, and other relevant metadata for reference during testing. See Appendix for the list of malware sample hashes.

2.9.2 Feature Extraction & Prediction

With the samples in hand, the next step was to prepare them for prediction. Leveraging the scripts we developed earlier, we efficiently extracted the necessary features from each sample, ensuring they matched the input format expected by our models.

After feature extraction, the processed data was passed onto our trained models for prediction. Machine learning models, especially when working with non-numeric data, rely on techniques like label encoding to transform categorical data into a format they can understand — numbers. For instance, if our dataset had malware families like "trojan", "spyware", and "ransomware", label encoding might represent them as "1", "2", and "3" respectively.

When our models make a prediction, they don't exactly say "trojan" or "spyware"; instead, they provide the encoded labels like "1" or "2". As a result, to make sense of these predictions, we need to decode them back into their original categories. This is an essential step to ensure the results are interpretable and actionable for users.



```
# Define malware types and families
malware_types = {0: "Trojan", 1: "Spyware", 2: "Ransomware"}
trojan_families = {0: "Emotet", 1: "Reconyc", 2: "Refroso", 3: "Scar", 4: "Zeus"}
spyware_families = {0: "180solutions", 1: "CWS", 2: "Gator", 3: "TIBS", 4: "Transponder"}
ransom_families = {0: "Ako", 1: "Conti", 2: "Maze", 3: "Pysa", 4: "Shade"}
```

Image: code defining the dictionary of malware types and families labels.

2.9.3 Prediction Probabilities and Their Importance

In our dynamic analysis, along with the prediction results indicating if a file is malicious or benign, we also display the associated prediction probabilities. These probabilities represent the model's confidence in its prediction.

Functionality:

1. **Broad Categorization**: Firstly, the tool provides a broad categorization, indicating whether the given file is deemed malicious or non-malicious based on its probability. The distinction is based on a set threshold, and the model's confidence is reflected in how close the probability is to either end (0 for benign or 1 for malicious).
2. **Uncertainty Handling**: For scenarios where the model is less certain (probabilities lying between 0.45 and 0.55), we have implemented a mechanism to inform the user that the tool is "uncertain". This range is specifically chosen because it's close to the 0.5 mark, which is a point of complete uncertainty.
3. **Malware Typing**: If the file is detected as malicious, we go a step further by trying to classify its type, i.e., whether it's a Trojan, Spyware, or Ransomware.
4. **Malware Family Classification**: To provide a more granular analysis, once the malware type is determined, the tool further predicts its specific family, such as "Zeus" for Trojans or "Maze" for Ransomware, accompanied by the associated probability.

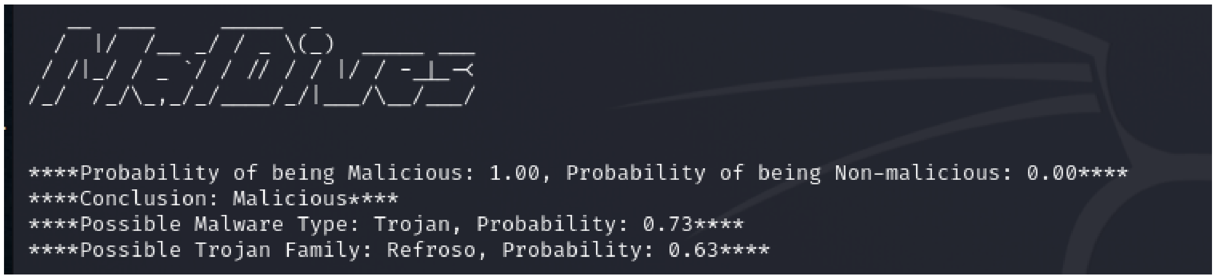
We include probability for the following reason:

Transparency: Providing probabilities offers a degree of transparency. Instead of a black-and-white answer, users understand the model's level of confidence in its predictions.

Informed Decision-making: By understanding the model's confidence level, analysts or IT professionals can make more informed decisions. A 90% confidence in a file being malicious might be treated differently than a 51% confidence.

Handling Ambiguities: Real-world data is often noisy and ambiguous. Providing a probability range of uncertainty allows users to take potential additional actions like further manual analysis or seeking second opinions.

Improving Trust: Knowing that the system can express uncertainty (instead of always acting overly confident) can bolster user trust in the tool. They are assured that the tool will highlight when it's less certain, prompting a more cautious approach.



```
   __  ___     _____  _
  /  |/  /__ _/ / _ \(_)  _____ _______
 / /|_/ / _ `/ / // / / |/ / -_) __(_-<
/_/  /_/\_,_/_/____/_/|___/\__/_/ /___/


****Probability of being Malicious: 1.00, Probability of being Non-malicious: 0.00****
****Conclusion: Malicious****
****Possible Malware Type: Trojan, Probability: 0.73****
****Possible Trojan Family: Refroso, Probability: 0.63****
```

Image: Sample run of a trojan sample.

3 Challenges

Throughout the development of 'Maldives', our team confronted an array of challenges ranging from the intricacies of machine learning to constraints in time and budget. Here's an in-depth look at some of the main challenges we faced.

3.1 Start from Scratch

Our endeavor into the world of machine learning and malware detection began on a blank slate. We embarked on this ambitious project with little to no prior knowledge in the domain of machine learning. This absence of a foundational understanding meant diving deep into self-study, investing countless hours in grasping fundamental concepts, reading relevant research, and experimenting with models and algorithms. The journey, while arduous, was one of immense learning and growth.

3.2 Open-Sourced Dataset

The quest to locate an apt, open-sourced dataset was a complicated and time-consuming task. The landscape of available datasets was vast, yet many were either excessively intricate for our intended application or had prohibitive cost implications. Our tight budget further limited our options. Eventually, our dependence on the chosen datasets meant that we had to adjust and adapt our methodologies to suit their intricacies.

For instance, our dynamic analysis was heavily influenced by our dataset. We meticulously constructed a sandbox environment, attempting to mirror the conditions under which the original dataset was curated. After a myriad of trials with various Windows versions, we managed to emulate an environment closely resembling the original. We sought guidance and documentation from the dataset's originator, a professor at the University of New Brunswick, however unfortunately, all associated documentation was unavailable with the closure of the project in 2022. This compelled us to rely on our resourcefulness and ingenuity.

3.3 Time Management

Time was perpetually against us. With a window of merely four months, every phase, from learning and understanding to experimentation and implementation, had to be expedited. The pressure to produce reliable results within this brief timeframe further compounded the challenge.

At the outset of our project, we recognized the importance of an airtight plan. Every phase, every task, from initial research to final touches, was demarcated with clear timelines. This preemptive approach not only provided us with a roadmap to navigate the complexities of our project but also ensured that no valuable time was lost in indecision or ambiguity.

Moreover, this predetermined timeline acted as our north star, guiding us when the weight of the challenges threatened to derail our progress. We consistently checked in with our schedule, adjusting our pace, prioritizing tasks, and recalibrating our approach when necessary. The result was not just adherence to our planned trajectory but also a systematic approach that prevented oversight or omission.

3.4 Obtaining Malware Samples

Access to genuine malware samples was pivotal to our project, yet it turned out to be a significant bottleneck. While platforms like VirusTotal offered a treasure trove of samples, their access was gated behind a paywall. Our salvation came in the form of platforms like MalwareBazaar and VirusShare. However, their scope was limited. The samples available often lacked proper tagging, rendering the task of pinpointing specific malware types immensely challenging.

3.5 Attempt to Set Up Cloud-Based Environment

A pivotal decision we grappled with was the platform for our sandbox and tool. We initially thought of the idea of leveraging cloud platforms like AWS to bolster security and accessibility. However, a careful examination of costs made it abundantly clear that such an endeavor would significantly overshoot our budget. Consequently, we pivoted back to our initial strategy, focusing our energies on crafting a robust proof-of-concept instead.

In conclusion, the myriad challenges that 'Maldives' threw our way, while daunting, were instrumental in shaping our understanding and approach. They forced us to think outside the box, to be adaptable, and to persevere. Each hurdle, in retrospect, was a lesson in disguise, enriching our experience and knowledge.

4 Lessons Learned

During the course of the 'Maldives' project, we encountered a range of experiences that solidified several key lessons for us. These insights are not only pivotal for this project but also offer a broader perspective that could benefit future endeavors in similar domains.

4.1 Importance of Development Environment

The consistency and reproducibility of our development environment emerged as foundational elements. Machine learning models, given their inherent complexity and reliance on specific versions and dependencies, are highly sensitive to even minor changes in the environment. We discovered that training a model with a particular version of Python or a specific library, and then deploying it under a different set-up, often led to unexpected challenges or even performance degradations. To mitigate these risks and ensure that our tool performed reliably across different platforms, we saw the need for meticulously replicating our development environment. The creation of a requirements.txt file was instrumental in this context. It provided users with a hassle-free mechanism to install all the necessary dependencies, ensuring the tool's consistent and optimal performance.

4.2 Keeping Up to Date

The rapidly evolving landscape of cybersecurity further underscored the importance of remaining updated. As new malware variants spring up and older ones evolve with new techniques, it becomes imperative that our tools and datasets are not left behind. For 'Maldives' to retain its relevance and efficacy, we recognized the need for periodic updates to our datasets. Similarly, the sandbox, designed as a controlled simulation of real-world computing environments, needs to be periodically refreshed to mirror the latest operating systems, applications, and configurations.

4.3 Scalability

One of the enlightening takeaways from our project was the realization that capturing malicious activities is not always straightforward. A single memory dump might not be comprehensive enough, given that malware can employ various tactics to stay dormant or evade detection. To effectively corner such elusive threats, it became clear that a more dynamic approach was needed. This would involve taking multiple memory dumps at varied intervals. Our vision for the future revolves around a scalable solution anchored in the cloud. Here, a multitude of instances could operate within a virtual network, with automation managing the intricacies of memory dumps, training models, and making predictions. Such an architecture, marked by its scalability and automation, promises a tool that's not only robust but also immensely practical for real-world applications.

4.4 Better Feature Engineering

Our deep dive into the 'Maldives' project also shed light on the nuanced art of feature engineering. As we progressed, the relevance and weight of different features in malware detection became more evident. Not all features were equally impactful. Some offered crucial insights into the malware's behavior, while others seemed to add little more than noise. This accentuated the importance of continuous refinement in feature engineering. The process requires a careful blend of domain expertise, rigorous data analysis, and iterative refinements. We came to terms with the possibility that our dataset might not be perfect—it could benefit from pruning irrelevant features, introducing new insightful ones, and transforming existing ones to better encapsulate the underlying malware patterns.

In essence, the journey with 'Maldives', filled with its unique challenges and triumphs, was a profound learning experience. These lessons, ranging from the rudiments of the development environment to the intricacies of feature engineering, are set to inform and shape our future projects.

5 Future Development

As the chapters of our 'Maldives' project unfold, it's essential not only to reflect on our accomplishments but also to anticipate the avenues of growth awaiting us. Each challenge faced and lesson learned shines a light on the myriad possibilities for 'Maldives' to evolve, adapt, and thrive in the ever-changing world of cybersecurity.

5.1 Moving to Cloud

A significant leap we envision is making a strategic transition to cloud platforms. The potential benefits are multifold. By migrating to the cloud, we not only anticipate bolstered security measures, but also expect a substantial enhancement in scalability. A cloud-based architecture holds the promise of seamless processing and analysis of vast swathes of data, all while assuring the highest levels of security. Furthermore, a cloud-based sandbox would serve as an ideal playground, allowing us to emulate and study the behaviors of the latest malware iterations in a safe, controlled environment.

5.2 Integration of VirusTotal API

The integration of the VirusTotal API is another advancement we are keenly exploring. This addition would elevate 'Maldives' from a manually operated tool to a largely automated malware detection system. With the VirusTotal API at its core, our tool could automatically acquire new malware samples, comprehensively extract their features, and consequently, craft new datasets. This continuous cycle of data ingestion and analysis would enable our models to be perpetually trained on the latest data. The result? 'Maldives' would be in a perpetual state of learning, its detection algorithms always attuned to the most recent malware threats and techniques.

5.3 Self-training and Update

With the foundations laid, a natural progression for 'Maldives' is towards self-sufficiency. We aim to imbue the tool with the ability to self-train. This means that, with every new piece of malware data it encounters, the tool will have the capability to learn, adapt, and refine its detection mechanisms autonomously. Such a dynamic approach ensures that 'Maldives' remains agile and responsive to emerging threats. This self-training capability, complemented by regular updates, ensures that our tool remains a step ahead, always ready to tackle the newest cybersecurity challenges.

To conclude, the roadmap ahead for 'Maldives' is replete with promise and potential. The envisioned enhancements and additions are not mere iterations but transformative changes that can redefine the way AI-assisted malware detection is approached. As we continue our journey, we remain committed to refining and elevating 'Maldives', ensuring it remains at the vanguard of cutting-edge malware detection solutions.

6 Conclusion

The journey of developing 'Maldives' was one of exploration, learning, and resilience. From grappling with the intricacies of machine learning to navigating the nuances of malware detection, our venture was both challenging and rewarding. Throughout, our primary focus remained on building a reliable, efficient, and user-friendly tool that could harness the capabilities of AI for malware detection.

Despite initial setbacks related to dataset acquisition, understanding malware samples, and budgetary constraints, our team demonstrated adaptability, ultimately producing a tool that showcases the potential of combining traditional cybersecurity practices with innovative AI methodologies. The lessons learned throughout this process, from the importance of a consistent development environment to the value of scalable solutions, have not only enriched our understanding but have paved the way for future advancements.

Looking forward, the blueprint for 'Maldives' encompasses exciting expansions like cloud migration, API integration, and the prospect of a continually evolving, self-learning tool. While we have achieved a significant milestone with the current iteration of 'Maldives', the horizon beckons with opportunities for refinement, enhancement, and innovation.

7 References

1. E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2014.

2. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

3. "Machine Learning for Cyber Security," YouTube. [Online]. Available: https://www.youtube.com/@machinelearningforcybersec5515. [Accessed: May-2023].

4. Coelho, *Getting Started with Deep Learning: Programming and Methodologies using Python*. CreateSpace Independent Publishing Platform, 1st edition, 2017. ISBN-10: 1542567092.

5. T. Carrier, "Canadian Institute for Cybersecurity's Malware Memory Analysis dataset," 2022. [Online]. Available: https://www.unb.ca/cic/datasets/malmem-2022.html. [Accessed: June-2023].

6. H. S. Anderson and P. Roth, "EMBER: An open dataset for training static PE malware machine learning models," arXiv preprint arXiv:1804.04637, 2018.

7. Packt Publishing, "Mastering Machine Learning for Penetration Testing," [Online]. Available: https://github.com/PacktPublishing/Mastering-Machine-Learning-for-Penetration-Testing/blob/master/Chapter03/MalwareData.csv.gz. [Accessed:June-2023].

8 Appendix

Environment & Dependencies for 'Maldives' Project

1. Operating System:

- Recommended: Linux-based OS (Ubuntu, Kali Linux, etc.), Windows (Windows 10 or Windows Server 2016+ for container support)

2. Language & Frameworks:

- Python 3 (specifically for model training and predictions):
  Version: Python 3.7+ (considering the libraries' versions you mentioned)
- Python 2.7 (specific utilities):
  Version: Python 2.7.x

3. Python 3 Libraries & Modules:

- `scikit-learn==1.2.2`: For machine learning models and utilities
- `keras==2.12.0`: Deep learning framework
- `joblib==1.3.1`: Saving and loading models
- `pandas==2.0.1`: Data handling and manipulation
- `scipy==1.10.1`: Scientific computing
- `pyfiglet==0.8.post1`: Text-based representation
- `pefile==2022.5.30`: PE file analysis

4. Python 2.7 Libraries & Modules:

- `cffi==1.14.0`: Foreign Function Interface for Python calling C code
- `distorm3==3.5.2`: Disassembler tool for binary files
- `pycryptodome==3.18.0`: Cryptographic and hash functions

5. Feature Extraction & Malware Analysis Tools:

- Win7 32bit SP1 Sandbox for dynamic malware analysis
- Volatility 2.5 memory forensics and extraction in line with Python 2.7 requirements

6. Virtualization & Containerization:

- VirtualBox or VMware for sandbox setups
- Anaconda 3: Used for creating isolated Python environments for training and utility purposes

7. Source Control & Collaboration:

- Git (with platforms like GitHub or GitLab) for version control and collaboration

List of Malware Sample Tested:

Sample	Hashes
Ransomware-Maze	6666c8ec8a82e24894a183945ea1051230ca6434d8fbdd7cba77b624f71bc0f9
Ransomware-Maze	73f475dd2c38b10ee049fa873735400d2c3ad61f19342eb864d869890a2cff39
Trojan-Refroso	2d998ca396afd67a498a8d6ea52bf28c3538ec28212eb2fb572dd747a87d2d03
Trojan-Refroso	ec26e1ba8a6779081dde986d34e1b2e9c26845f661714f3c4815f1746885e1fa
Trojan-Zeus	2a36f829f8f83ea131503f6b3c6e164277a9f648f925db3e78be6b5fe4154f51
Trojan-Zeus	1abebc7abf04169dc87384f32063393e8ae9c1fe714a226ef139b8f8b70b7c49
Trojan-Scar	20f8b533ae63fea2bae3631f40fa0910518fb4eee4283f2a7c5511c495bef0a6
Spyware-CWS	0d5f6f5371ead0e0e892dc2209eabe853f3a64b10530630b73f5a6e3e97cdc1f
Spyware-CWS	0c7f38bb973664364d0333459a5a7058474a21915f13388bbaa6696fa7185b02
Spyware-Transponder	00af9e735822c30f5d7f20f11be722ccfea2fcc418f8d1e65cb9cc5f7c9f526c
Ransomware-Ako	0afc043fc760b4e7e6bab1f21e3a88d341522f63da309f49ffe17ba0908c4771
Spyware-Gator	00a316e53de939f82885157ce057f134eb26ffbdaed314aa50c0ef19318bcdf4
Trojan-Zeus	7c4ecadd70e6c4f82dd598949634e1b6a50bebd658cec4b8489a9302a95c03fb

List of Keywords:

Anaconda 3

API Calls

Automation

Cloud-based Environment

Cloud Costs

Containerization

Cybersecurity

Data sets

Deep Learning

Dependency Management

Domain Knowledge

Dynamic Analysis

Environment Setup

Feature Engineering

Feature Extraction

GitHub

Joblib

Keras

Label Encoding

Machine Learning

Maldives

Malware Analysis

Malware Detection

Malware Samples

Memory Dump

Model Training

Multilayer Perceptron

Open-Source

Pandas

Penetration Testing

Prediction

Probability

Proof-of-Concept

Python

Random Forests

Ransomware

Real-world Threats

Sandbox

Scalability

Scikit-learn

Scipy

Spyware

Static Analysis

Support Vector Machines

Tensorflow

Threat Intelligence

Trojan

Virtualization

VirusShare

VirusTotal

Volatility

End of Report