META JACKET

Group 67

CAPSTONE PROJECT

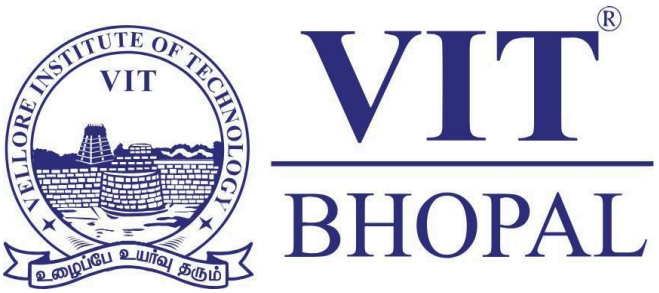
Phase – 2 Report

Submitted by

1. **20BAI10105 Ishaan Shukla**
2. **20BAI10127 Anagh Garg**
3. **20BAI10292 Varun Shukla**

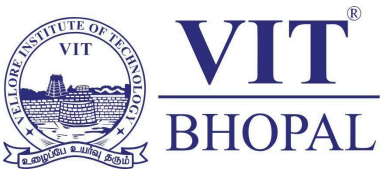
in partial fulfillment of the requirements for the

degree of Bachelor of Engineering and Technology



VIT Bhopal University
Bhopal
Madhya Pradesh

March, 2024



Bonafide Certificate

Certified that this project report titled “META JACKET” is the bonafide work of “20BAI100105 Ishaan Shukla, 20BAI10127 Anagh Garg, 20BAI10292 Varun Shukla” who carried out the project work under my supervision.

This project report (DSN4095-Capstone Project Phase-II) is submitted for the Project Viva-Voce examination held on 11/03/24

Supervisor

TABLE OF CONTENTS

INTRODUCTION

In the dynamic landscape of the Internet's rapid evolution, the pervasive threat of malware has emerged as a formidable challenge in the realm of cybersecurity. Malware, broadly defined as any software engineered to execute malicious actions, ranging from surreptitious information theft to covert espionage, poses a substantial risk to the integrity and security of computer systems. Kaspersky Labs (2017) succinctly characterizes malware as "a type of computer program designed to infect a legitimate user's computer and inflict harm on it in multiple ways."

Despite the continuous advancement of anti-virus scanners, the escalating diversity of malware strains surpasses the efficacy of conventional protection measures. This deficiency is glaringly evident in the alarming statistics reported by Kaspersky Labs (2016), revealing a staggering 6,563,145 hosts falling victim to attacks, with 4,000,000 distinct malware entities detected in 2015 alone. The consequences of these incursions extend beyond individual breaches, as Juniper Research (2016) projects a global surge in the cost of data breaches, anticipating a staggering $2.1 trillion by 2019.

Compounding this complex landscape is the diminishing barrier to entry for malware development. The proliferation of attack tools on the Internet has lowered the threshold for engaging in malicious activities. The ease of access to anti-detection techniques, coupled with the ability to purchase pre-engineered malware on the clandestine black market, has democratized the realm of cyber threats. Current empirical studies underscore a concerning trend: a rising number of attacks orchestrated by script-kiddies or automated processes, underscoring the evolving nature of cyber threats and the pressing need for innovative and adaptive cybersecurity measures.

MOTIVATION

In the dynamic landscape of cyberspace, where technology rapidly evolves, the threat of malware looms larger than ever. As India embraces the digital era, ensuring the security of our digital infrastructure is paramount. The MetaJacket project aims to tackle this challenge head-on, employing cutting-edge machine learning techniques to fortify our defenses against the ever-evolving landscape of malicious software.

Understanding the Urgency: India has witnessed a significant surge in cyber threats and attacks, with malware posing a substantial risk to individuals, businesses, and government entities alike. As we become more interconnected, the potential impact of a successful malware attack extends beyond mere inconvenience; it can disrupt critical services, compromise sensitive data, and pose a threat to national security. MetaJacket recognizes the urgency of the situation and strives to contribute to the nation's resilience against cyber threats.

Machine Learning as the Vanguard: Traditional antivirus solutions are becoming increasingly inadequate in the face of sophisticated malware variants. MetaJacket leverages the power of machine learning to enhance the efficacy of malware detection. By analyzing vast datasets and identifying patterns indicative of malicious behavior, our project seeks to provide a proactive and adaptive defense mechanism against the ever-mutating landscape of malware

A Call to Action: MetaJacket is more than a project; it is a call to action to fortify our digital defenses. By investing in this initiative, we contribute not only to the security of our digital infrastructure but also to the advancement of machine learning in the realm of cybersecurity. Together, let us build a safer, more resilient digital future for India.

Real Time Case Study

Throughout the annals of 2022, Mailchimp and its consortium of partners found themselves ensnared in the crosshairs of cyber adversaries, navigating a tumultuous landscape marked by a series of relentless attacks. The apex of vulnerability was reached in the inaugural month of 2023, where malevolent actors orchestrated a meticulously crafted phishing campaign that proved to be alarmingly successful. In an unfortunate turn of events, at least one unsuspecting Mailchimp employee succumbed to the stratagems employed by these digital malefactors, inadvertently divulging their login credentials.

The aftermath of this security breach reverberated ominously, resulting in the compromise of no fewer than 133 Mailchimp user accounts. Among the collateral damage were accounts of reputable businesses, including but not limited to WooCommerce, Statista, Yuga Labs, Solana Foundation, and FanDuel, casting a shadow of concern over the potential ramifications across diverse sectors.

The perpetrators of this insidious breach exhibited a strategic focus on the implementation of social engineering tactics, meticulously honing in on Mailchimp employees and contractors. Exploiting a perceived lapse in an employee's vigilance or perhaps their inherent challenge in discerning the nuanced intricacies of a sophisticated social engineering attack, the malevolent actors successfully infiltrated the sanctity of their user accounts. This incident stands as a stark testament to the realization that, in the realm of cybersecurity, the human factor remains a pivotal vulnerability that demands meticulous attention.

The profound repercussions of employee-induced data breaches, typified by this disconcerting incident, underscore the imperative of not underestimating the potency of phishing and other social engineering techniques. The call to action extends beyond the mere deployment of robust security software, advocating for a comprehensive approach that encompasses regular and rigorous cybersecurity training for both employees and partners. It is through the empowerment of the human element that organizations can aspire to create a resilient bulwark against the ever-evolving tactics employed by cyber adversaries.

Moreover, the incident underscores the efficacy of proactive measures such as the adoption of two-factor authentication (2FA). In this context, the implementation of a 2FA tool could have thwarted the malevolent actors from successfully leveraging the compromised credentials. The additional layer of security inherent in 2FA, requiring a secondary authentication factor, stands as a formidable deterrent against unauthorized access, thereby fortifying the integrity of user accounts.

In conclusion, the Mailchimp security breach of 2023 serves as a poignant reminder of the intricacies and vulnerabilities inherent in the contemporary cybersecurity landscape. It prompts a recalibration of defense strategies, emphasizing the paramount importance of ongoing education, training, and the integration of advanced security measures, including robust cybersecurity protocols and 2FA, to safeguard against the relentless ingenuity of cyber adversaries.

Novelty

- Biologically Inspired Intelligence:

The project's novelty lies in its pioneering approach of drawing inspiration from the intricacies of the human immune system. By incorporating principles of adaptive learning, behavioral analysis, and continuous evolution, the antivirus system transcends conventional cybersecurity paradigms, creating a digital entity that mirrors the sophistication of biological defense mechanisms.

- Dynamic Learning Ecosystem:

Unlike traditional antivirus solutions that rely on static databases, this project establishes a dynamic learning ecosystem. The system engages in a perpetual dance of acquiring knowledge from diverse datasets, enabling it to discern emerging threats and adapt its defenses in real-time. This continuous learning capability adds a layer of responsiveness unparalleled in the realm of cybersecurity.

- Behavioral Symphony in Cyberspace:

The orchestration of behavioral analysis within the antivirus system elevates its capabilities beyond mere threat detection. Like a symphony conductor interpreting the nuances of each instrument, the system deciphers the subtle notes of normalcy and dissonance in cyber behavior. This unique perspective enables it to anticipate and counteract novel threats with a level of sophistication previously unseen in the digital realm.

- User-Centric Transparency:

The project introduces a user interface that transcends the typical cryptic nature of cybersecurity tools. By providing transparent insights into the system's adaptive learning process, detected threats, and overall status, users are not only protected but also empowered. This user-centric approach fosters trust and collaboration, bridging the gap between complex technology and end-users in a novel and accessible manner.

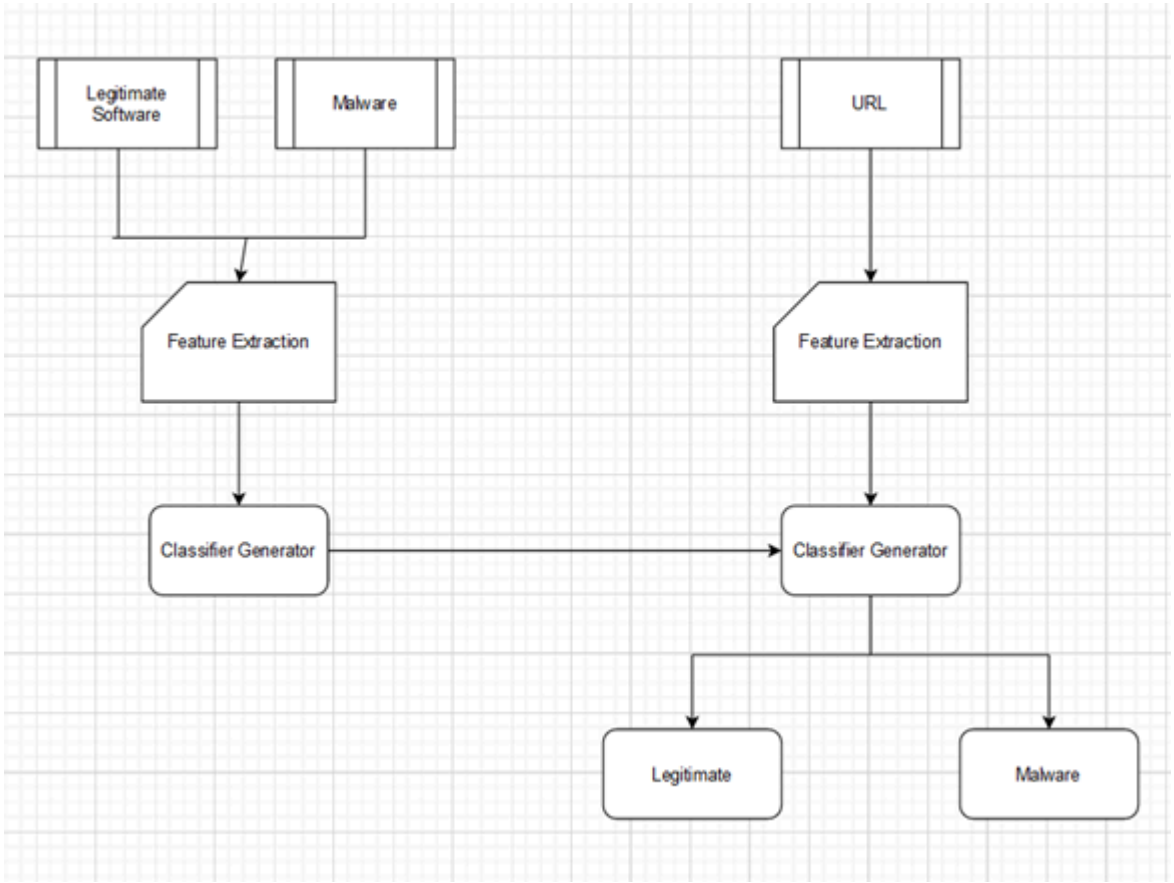
EXISTING WORK

Malware detectors that are based on signatures can perform well on previously-known malware, that was already discovered by some anti-virus vendors. However, it is unable to detect polymorphic malware, that has the ability to change its signatures, as well as new malware, for which signatures have not been created yet.

The accuracy of heuristics-based detectors is not always sufficient for adequate detection, resulting in a lot of false positives and false negatives. (Baskaran and Ralescu 2016). The need for the new detection methods is dictated by the high spreading rate of polymorphic viruses.

In his paper “Malware Detection Using Machine Learning” Dragos Gavrilut aimed for developing a detection system based on several modified perceptron algorithms. For different algorithms, he achieved the accuracy of 69.90%- 96.18%. It should be stated that the algorithms that resulted in best accuracy also produced the highest number of false-positives: the most accurate one resulted in 48 false positives. The most balanced algorithm with appropriate accuracy and the low false-positive rate had the accuracy of 93.01%. (Gavrilut,et al. 2009)
“A Static Malware Detection System Using Data Mining Methods” proposed extraction methods based on PE headers, DLLs and API functions and methods based on Naive Bayes, J48 Decision Trees, and Support Vector Machines. Highest overall accuracy was achieved with the J48 algorithm (99% with PE header feature type and hybrid PE header & API function feature type, 99.1% with API function feature type). (Baldangombo, Jambaljav and Horng 2013)

SYSTEM ARCHITECTURE



DATASET : Data Collection

In the initial phase of developing our groundbreaking adaptive antivirus system, our focus turns to the foundational step of data collection. This crucial stage forms the bedrock upon which the intelligence of our system will be built. Our approach is meticulous, aiming to assemble a diverse and comprehensive dataset that encapsulates the intricacies of various cyber threats. The following breakdown elucidates the intricacies of our data collection strategy:

1. Diverse Dataset Compilation:

Malware Samples: Our dataset encompasses a spectrum of malware samples, ranging from traditional viruses to sophisticated threats like trojans, ransomware, and polymorphic code. This diversity ensures that our adaptive antivirus system develops a nuanced understanding of the myriad manifestations of malicious code.

Attack Vectors: We categorize data based on different attack vectors, including email, web, and network. This categorization mirrors realistic scenarios, allowing our system to fortify its defenses against potential entry points and attack vectors.

Real-world Scenarios: To emulate the dynamic nature of cyber threats, we introduce datasets that simulate real-world scenarios. These scenarios include evolving threats, evasion techniques, and attempts to bypass conventional security measures.

2. Metadata and Context Inclusion:

File Attributes: We capture detailed file attributes, including size, type, and origin. This information assists our system in establishing patterns related to malicious files.

System Calls: Recording system calls generated during the execution of files provides a comprehensive understanding of behavior at the system level, aiding in the identification of anomalies indicative of malicious activity.

Network Activity: We monitor and record network activity associated with each file or scenario. This includes tracking communication patterns and connections to external entities, contributing to our system's comprehension of potential threats.

3. Threat Intelligence Integration:

External Threat Feeds: To enhance our dataset, we incorporate threat intelligence feeds. These feeds provide our system with real-time insights into the latest cyber threats and vulnerabilities, fostering adaptability.

Historical Data: Inclusion of historical data on past cyber incidents allows our system to recognize recurrent patterns and understand the evolution of threats over time.

4. Data Quality Assurance:

Validation and Cleaning: Rigorous validation processes are implemented to ensure the integrity and quality of our dataset. We meticulously remove redundancies, outliers, or inaccuracies that might compromise the learning process.

Balancing: We strive for a balanced representation of both benign and malicious instances within the dataset, recognizing that imbalances may affect the system's ability to generalize effectively.

Project's Model Description:

Technical Coding and code solutions

PE-header-based malware detection

The main part of this project is the machine learning model in which we used Random Forest classifier tree to classify the malware/benign files. The dataset that we are using contains 70.1% malwares and 29.9% benign files.

As per the splitting part goes, we divided the data into 70% training data and 30% testing data and then we selected the important features that are required for the classification using the extratrees.feature_importances_ function and then we compared the score of Decision Tree Classifier and Random Forest classifier and found that Random Forest Classifier's score (99.45%) is better than Decision Tree Classifier (99.04%), hence we selected it for training and then after training we saved it as Classifier.pkl and also saved the features that we found important as features.pkl to keep track of the required function while extracting it from any real file.

After the Machine Learning part comes the file extraction part in which we extract the features that are required for the classification. The major challenge that we faced while coding this project was to extract the features of the PE Header file and then storing it for the saved machine learning model. For extracting the content of the PE Header we used pefile library (https://pypi.org/project/pefile/). The selection of those features is done using the feature.pkl model that stores all the important features for the Random Forest Classifier. The extraction of the PE Header files is done using pefile library in python and then the selected features are then given to the classifier.pkl machine and it predicts the output.

Malicious URL detection

This part of the program also consists of two phases, Cleaning the data for the Logistic Regression and then training the machine to identify if it is malicious or not. The Model is sketched out in such a way that it can meaningfully understand the data from which it has to be trained upon and tried to develop a defined behaviour from the data-sets. Data-sets are the backbone of any model and hence it should be adequate and precise data for good as well as bad URLs present in the data for the model to be trained upon. The Machine

Learning for the malicious website detection will first involve in cleaning of our data within the data-sets. We used pandas and defined our own vectorizer to clear the data-sets and then used Logistic Regression to train our model.

Since the URLs are different from our normal text documents, a sanitization method is used to get the relevant data from raw URLs.

We will implement our sanitization function in python to filter the URLs. This will give us the desired URL data-set values to train the model and test it. The data-set will have two column structure one for URLs and other for labels (malicious or not).

Then we used Tf-idf machine learning text feature extraction approach from the sklearn python module. Before that we need to read the data-sets into data frames and matrix which can be understood by the vectorizer we prepared and later pass onto the term-frequency and inverse document frequency text extraction approach. We used pandas module in python for that task. As stated above we used logistic regression method to train our model but before that we passed the data to our custom vectorizer function using Tf-idf approach and after that we can train and test our ML model.

Still the model was not able to predict all the good URLs and hence club good URLs with bad URLs so we used the classical method of URL filtering with conjunction to the machine learning model i.e., Whitelist Filter. It is the list of websites that we know are good and at-least non-malicious and won't harm our users. So, we allow these particular websites through our internet traffic, the opposite is called as blacklisting. It's a very simple but efficient approach to segregate our network traffic, similarly we implemented that in our machine learning model

CODE DESCRIPTION (Backend)

In the backend we have 4 major files, 2 of each model, one for the PE header (internal computer files), another for URL's present on the internet(External links). Here is the description and explaination of the code .

1) Pe_main.py

Introduction
The Python code snippet provided encapsulates a sophisticated program tailored explicitly for the classification of Portable Executable (PE) files. By harnessing the capabilities of the pefile library, this program facilitates the extraction of crucial features inherent within PE files and subsequently applies advanced machine learning algorithms to classify them into distinct categories, namely malicious or legitimate.
Purpose

The overarching aim of this program is to automate the intricate process of classifying PE files based on their intrinsic features. By categorizing PE files, the program fulfills a critical role in identifying potentially malicious software, thus fortifying cybersecurity protocols. Furthermore, it empowers organizations to discern between benign and malicious software, thereby augmenting overall system security posture.

Feature Extraction

1. Entropy Calculation Function (get_entropy):
    - This function is instrumental in computing the entropy, a statistical measure of randomness, of data sequences encapsulated within PE files.
    - It plays a pivotal role in quantifying the randomness exhibited by various data sections within PE files, aiding in discerning anomalous patterns indicative of malicious behavior.
2. Resource Extraction Function (get_resources):
    - Designed to extract diverse resources embedded within PE files, including icons, bitmaps, etc.

- It meticulously computes the entropy and size attributes of each extracted resource, facilitating comprehensive feature extraction and subsequent analysis.

3. Version Information Extraction Function (get_version_info):
    - This function specializes in extracting version-related metadata embedded within PE files.
    - By retrieving details such as file version, product version, and associated flags, it provides critical insights into the origin and characteristics of the PE file.
4. Feature Extraction Function (extract_infos):
    - Leveraging the pefile library, this function orchestrates the extraction of an extensive array of features from PE files.
    - Features encompass a broad spectrum of attributes, spanning header information, section details, import/export counts, resource statistics, etc., thereby providing a comprehensive overview of the PE file's structure and composition.

Main Execution

1. Loading Classifier and Features:
    - The program initializes by loading pre-trained classifier models and feature datasets from serialized files (classifier.pkl and features.pkl, respectively).
    - These components serve as the backbone for the subsequent classification process, enabling efficient and accurate classification of PE files.
2. Extracting Features from PE File:
    - Upon receiving the path to a PE file as a command-line argument, the program leverages the extract_infos function to extract pertinent features from the specified PE file.
    - This step lays the foundation for subsequent classification, providing a comprehensive feature set essential for discerning between benign and malicious PE files.
3. Matching Features and Classification:
    - The extracted features are meticulously matched with the feature set utilized during the training phase of the classifier.
    - Leveraging the pre-trained classifier, the program predicts the classification label (malicious or legitimate) of the PE file based on the extracted features.

- The classification outcome, along with the pertinent features utilized during the process, is meticulously documented, providing stakeholders with actionable insights into the nature of the analyzed PE file.

Conclusion

In summation, this program represents a critical asset in the arsenal of cybersecurity professionals, offering a robust and efficient mechanism for automated analysis and classification of PE files. By seamlessly integrating feature extraction and classification functionalities, the program empowers organizations to proactively mitigate cybersecurity threats, thereby safeguarding critical assets and infrastructure against evolving security challenges.

```python
import pefile
import os
import array
import math
import pickle
import joblib
import sys
import argparse


#For calculating the entropy
def get_entropy(data):
    if len(data) == 0:
        return 0.0
    occurences = array.array('L', [0]*256)
    for x in data:
        occurences[x if isinstance(x, int) else ord(x)] += 1

    entropy = 0
    for x in occurences:
        if x:
            p_x = float(x) / len(data)
            entropy -= p_x*math.log(p_x, 2)

    return entropy

#For extracting the resources part
def get_resources(pe):
    """Extract resources :
    [entropy, size]"""
    resources = []
    if hasattr(pe, 'DIRECTORY_ENTRY_RESOURCE'):
        try:
            for resource_type in pe.DIRECTORY_ENTRY_RESOURCE.entries:
                if hasattr(resource_type, 'directory'):
                    for resource_id in resource_type.directory.entries:
                        if hasattr(resource_id, 'directory'):
                            for resource_lang in resource_id.directory.entries:
                                data = pe.get_data(resource_lang.data.struct.OffsetToData, resource_lang.data.struct.Size)
                                size = resource_lang.data.struct.Size
                                entropy = get_entropy(data)

                                resources.append([entropy, size])
        except Exception as e:
            return resources
    return resources

#For getting the version information
def get_version_info(pe):
    """Return version infos"""
    res = {}
    for fileinfo in pe.FileInfo:
        if fileinfo.Key == 'StringFileInfo':
            for st in fileinfo.StringTable:
                for entry in st.entries.items():
                    res[entry[0]] = entry[1]
        if fileinfo.Key == 'VarFileInfo':
            for var in fileinfo.Var:
                res[var.entry.items()[0][0]] = var.entry.items()[0][1]
    if hasattr(pe, 'VS_FIXEDFILEINFO'):
        res['flags'] = pe.VS_FIXEDFILEINFO.FileFlags
        res['os'] = pe.VS_FIXEDFILEINFO.FileOS
        res['type'] = pe.VS_FIXEDFILEINFO.FileType
        res['file_version'] = pe.VS_FIXEDFILEINFO.FileVersionLS
        res['product_version'] = pe.VS_FIXEDFILEINFO.ProductVersionLS
        res['signature'] = pe.VS_FIXEDFILEINFO.Signature
        res['struct_version'] = pe.VS_FIXEDFILEINFO.StrucVersion
    return res
```

```python
#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

    # Sections
    res['SectionsNb'] = len(pe.sections)
    entropy = list(map(lambda x:x.get_entropy(), pe.sections))
    res['SectionsMeanEntropy'] = sum(entropy)/float(len((entropy)))
    res['SectionsMinEntropy'] = min(entropy)
    res['SectionsMaxEntropy'] = max(entropy)
    raw_sizes = list(map(lambda x:x.SizeOfRawData, pe.sections))
    res['SectionsMeanRawsize'] = sum(raw_sizes)/float(len((raw_sizes)))
    res['SectionsMinRawsize'] = min(raw_sizes)
    res['SectionsMaxRawsize'] = max(raw_sizes)
    virtual_sizes = list(map(lambda x:x.Misc_VirtualSize, pe.sections))
    res['SectionsMeanVirtualsize'] = sum(virtual_sizes)/float(len(virtual_sizes))
    res['SectionsMinVirtualsize'] = min(virtual_sizes)
    res['SectionMaxVirtualsize'] = max(virtual_sizes)

    #Imports
    try:
        res['ImportsNbDLL'] = len(pe.DIRECTORY_ENTRY_IMPORT)
        imports = sum([x.imports for x in pe.DIRECTORY_ENTRY_IMPORT], [])
        res['ImportsNb'] = len(imports)
        res['ImportsNbOrdinal'] = 0
    except AttributeError:
        res['ImportsNbDLL'] = 0
        res['ImportsNb'] = 0
        res['ImportsNbOrdinal'] = 0
```

```
    #Exports
    try:
        res['ExportNb'] = len(pe.DIRECTORY_ENTRY_EXPORT.symbols)
    except AttributeError:
        # No export
        res['ExportNb'] = 0
    #Resources
    resources= get_resources(pe)
    res['ResourcesNb'] = len(resources)
    if len(resources)> 0:
        entropy = list(map(lambda x:x[0], resources))
        res['ResourcesMeanEntropy'] = sum(entropy)/float(len(entropy))
        res['ResourcesMinEntropy'] = min(entropy)
        res['ResourcesMaxEntropy'] = max(entropy)
        sizes = list(map(lambda x:x[1], resources))
        res['ResourcesMeanSize'] = sum(sizes)/float(len(sizes))
        res['ResourcesMinSize'] = min(sizes)
        res['ResourcesMaxSize'] = max(sizes)
    else:
        res['ResourcesNb'] = 0
        res['ResourcesMeanEntropy'] = 0
        res['ResourcesMinEntropy'] = 0
        res['ResourcesMaxEntropy'] = 0
        res['ResourcesMeanSize'] = 0
        res['ResourcesMinSize'] = 0
        res['ResourcesMaxSize'] = 0

    # Load configuration size
    try:
        res['LoadConfigurationSize'] = pe.DIRECTORY_ENTRY_LOAD_CONFIG.struct.Size
    except AttributeError:
        res['LoadConfigurationSize'] = 0


    # Version configuration size
    try:
        version_infos = get_version_info(pe)
        res['VersionInformationSize'] = len(version_infos.keys())
    except AttributeError:
        res['VersionInformationSize'] = 0
    return res


if __name__ == '__main__':

    #Loading the classifier.pkl and features.pkl
    clf = joblib.load('Classifier/classifier.pkl')
    features = pickle.loads(open(os.path.join('Classifier/features.pkl'),'rb').read())

    #extracting features from the PE file mentioned in the argument
    data = extract_infos(sys.argv[1])

    #matching it with the features saved in features.pkl
    pe_features = list(map(lambda x:data[x], features))
    print("Features used for classification: ", pe_features)

    #prediciting if the PE is malicious or not based on the extracted features
    res= clf.predict([pe_features])[0]
    print ('The file %s is %s' % (os.path.basename(sys.argv[1]),['malicious', 'legitimate'][res]))
```

2) url_main.py

Introduction

The provided Python code snippet encapsulates a robust program designed explicitly for the classification of Uniform Resource Locators (URLs) into distinct categories: malicious or legitimate. Leveraging advanced machine learning techniques, particularly logistic regression, this program enables the automatic categorization of URLs based on features extracted from their structures.

Purpose

The primary objective of this program is to streamline the process of URL classification, thereby facilitating the identification of potentially harmful websites and bolstering cybersecurity measures. By automating the classification task, the program empowers users to swiftly discern between benign and malicious URLs, enhancing the overall security posture of systems and networks.

Libraries Used

- pandas: This library is employed for efficient data manipulation and analysis, facilitating seamless handling of URL data and associated features.
- numpy: As a fundamental library for numerical computing, numpy provides essential support for mathematical functions and operations on arrays, crucial for various data processing tasks.
- sklearn: An indispensable machine learning library in Python, sklearn offers a plethora of functionalities for model selection, preprocessing, and evaluation, including support for logistic regression.
- model_selection: This submodule within sklearn provides essential tools for model selection and evaluation, enabling robust performance assessment of the logistic regression classifier.
- feature_extraction.text: Specifically, this submodule offers utilities to convert raw text data, such as URLs, into feature vectors suitable for input into machine learning models.
- linear_model: Within sklearn, linear_model implements a variety of linear models, including logistic regression, which is pivotal for URL classification tasks.
- pickle: Utilized for serializing and deserializing Python objects, pickle plays a critical role in storing and retrieving pre-trained logistic regression models and associated data structures.

URL Sanitization

- The program features a URL sanitization function meticulously crafted to

preprocess URLs before classification.
- Key preprocessing steps include converting URLs to lowercase, tokenizing based on specific delimiters (such as '-', '/', and '.'), sanitizing tokens, and removing duplicates and the 'com' token if present.

Input
- Users are prompted to input the URL(s) requiring classification, with the entered URLs stored in a list named urls for subsequent processing.

Whitelist Filtering
- A whitelist comprising known legitimate websites is defined to filter out URLs that are already classified as safe.
- This preemptive measure helps mitigate false positives, preventing legitimate URLs from being erroneously classified as malicious.
- URLs present in the whitelist are excluded from the list of URLs earmarked for classification (s_url).

Model Loading
- The program loads pre-trained logistic regression models and TF-IDF (Term Frequency-Inverse Document Frequency) vectorizers from serialized files (pickle_model.pkl and pickle_vector.pkl, respectively) using the pickle module.
- This deserialization process ensures that the logistic regression model and vectorizer are loaded into memory for subsequent use in URL classification.

Prediction
- Upon loading the TF-IDF vectorizer, sanitized URLs are transformed into feature vectors suitable for input into the logistic regression model.
- The logistic regression model then predicts the class label (malicious or legitimate) for each URL, facilitating swift and accurate classification.
- Predictions are exclusively made for URLs not present in the whitelist, ensuring that only unclassified URLs undergo classification.
- The classification result, indicating the malicious or legitimate nature of the entered domain, is printed for user reference.

Conclusion

In conclusion, this program represents a pivotal asset in the arsenal of cybersecurity professionals, offering a practical and efficient solution for automated URL classification. By harnessing the power of machine learning, it empowers users to identify and mitigate potential cybersecurity threats posed

by malicious websites, thereby fortifying overall cybersecurity defenses.

Extract > url_main.py > ...

```python
'''

'''

import pandas as pd
import numpy as np
import random
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
import pickle


def sanitization(web):
    web = web.lower()
    token = []
    dot_token_slash = []
    raw_slash = str(web).split('/')
    for i in raw_slash:
        raw1 = str(i).split('-')
        slash_token = []
        for j in range(0,len(raw1)):
            raw2 = str(raw1[j]).split('.')
            slash_token = slash_token + raw2
        dot_token_slash = dot_token_slash + raw1 + slash_token
    token = list(set(dot_token_slash))
    if 'com' in token:
        token.remove('com')
    return token

urls = []
urls.append(input("Input the URL that you want to check (eg. google.com) : "))
#print (urls)

whitelist = ['hackthebox.eu','root-me.org','gmail.com']
s_url = [i for i in urls if i not in whitelist]

#Loading the model
file = "Classifier/pickel_model.pkl"
with open(file, 'rb') as f1:
    lgr = pickle.load(f1)
f1.close()
file = "Classifier/pickel_vector.pkl"
with open(file, 'rb') as f2:
    vectorizer = pickle.load(f2)
f2.close()

#predicting
x = vectorizer.transform(s_url)
y_predict = lgr.predict(x)

for site in whitelist:
    s_url.append(site)
#print(s_url)

predict = list(y_predict)
for j in range(0,len(whitelist)):
    predict.append('good')
print("\nThe entered domain is: ", predict[0])
```

3) pe.ipynb

1. Importing the Dataset
The initial step involves importing the dataset from a CSV file named 'data.csv' utilizing the pandas library. The CSV file is parsed using the specified separator '|' (pipe) to delineate the data fields.

2. About the Dataset
This section encompasses a thorough exploration of the dataset to gain insights into its structure and contents. Various methods such as head(), tail(), columns, describe(), and info() are employed to understand the dataset's characteristics, including its dimensions, data types, missing values, and summary statistics. Additionally, the value_counts() method is utilized to ascertain the distribution of malware (0) and benign (1) files within the dataset.

3. Visualization
To provide visual clarity on the distribution of malware and benign files within the dataset, the matplotlib library is imported for graphical representation. A pie chart is generated using the value_counts() method, visually depicting the proportion of malware and benign files in the dataset.

4. Feature Selection
The feature selection process is initiated by importing essential libraries such as os, pandas, numpy, pickle, pefile, sklearn.ensemble, SelectFromModel, joblib, DecisionTreeClassifier, confusion_matrix, svm, and sklearn.metrics. Irrelevant columns ('Name', 'md5', 'legitimate') are dropped from the dataset to prepare it for classification, as classification models only accept numerical features. The selected features are then stored in the variable X, while the target variable is stored in y. Feature importance is determined using the Extra Trees Classifier, and the SelectFromModel function is employed to select features based on their importance. The resultant dataset containing the selected features (X_new) is created, and the number of important features (nbfeatures) is computed.

5. Data Fitting and Choosing Important Variables
The Extra Trees Classifier is fitted to the dataset (X and y) to extract feature importance scores. Features are selected based on their importance using the SelectFromModel function, resulting in the creation of a transformed dataset (X_new) containing only the important features. The number of important

features is stored in nbfeatures for further analysis.

6. All the Required Features
A comprehensive list of the most important features and their corresponding importance scores is displayed. Features are ranked based on their importance, and the top features are printed along with their respective scores, providing insights into the key determinants influencing classification outcomes.

7. Testing Which Classifier Will Give Better Results
To evaluate the performance of different classifiers, namely Decision Tree and Random Forest, both models are trained on the training data (X_train, y_train) and evaluated on the testing data (X_test, y_test). The accuracy score of each classifier is computed and compared, with the classifier exhibiting the highest accuracy deemed as the winner.

8. Saving the Machine Learning Model and Features
The winning classifier is serialized and saved as 'classifier.pkl' using the joblib.dump function, ensuring preservation for future use. Similarly, the selected features are serialized and saved as 'features.pkl' using the pickle.dumps function, facilitating easy retrieval and deployment in subsequent analyses.

9. Loading the Classifier and Features
The saved classifier and features are loaded back into memory for future utilization. The classifier is loaded using the joblib.load function, while the features are loaded using the pickle.loads function, ensuring seamless integration into the classification pipeline.

10. Sample Output
A sample output is provided to illustrate the practical application of the trained classifier. The extracted features and the predicted label (malware/legitimate) for a sample file ('LoJaxSmallAgent.exe') are displayed, showcasing the classifier's efficacy in real-world scenarios.
This comprehensive report delineates the intricacies of dataset analysis, feature selection, classifier evaluation, and model serialization, laying the foundation for robust malware detection methodologies.

Importing the dataset

Source :

```
import pandas as pd
dataset = pd.read_csv('data.csv', sep='|')
```
[28]

About the dataset

```
dataset.head()    #Top 5 row of the dataset
```
[ ]

```
dataset.tail()     #Last 5 row of the dataset
```
[ ]

```
dataset.columns    # name of the columns
```
[ ]

```
dataset.describe(include="all")    # summary of numeric attributes
```
[ ]

```
dataset.info()    # info about the whole dataset
```
[ ]

```
dataset["legitimate"].value_counts()    # count of malware (0) and benign (1) files in dataset
```
[ ]

Visualization

```
import matplotlib.pyplot as plt

dataset["legitimate"].value_counts().plot(kind="pie",autopct="%1.1f%%")
plt.show()
```
[ ]

```
import os
import pandas
import numpy
import pickle
import pefile
import sklearn.ensemble as ek
from sklearn.feature_selection import SelectFromModel
import joblib
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix
from sklearn import svm
import sklearn.metrics as metrics
```
[50]

Feature Selection

```
# Feature
X = dataset.drop(['Name','md5','legitimate'],axis=1).values    #Droping this because classification model will not accept object type elements (float and int only)
# Target variable
y = dataset['legitimate'].values
```

Data Fitting and choosing the important variables

```
extratrees = ek.ExtraTreesClassifier().fit(X,y)
model = SelectFromModel(extratrees, prefit=True)
X_new = model.transform(X)
nbfeatures = X_new.shape[1]
```

```
#Number of important features
nbfeatures
```

```
#splitting the data (70% - training and 30% - testing)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_new, y ,test_size=0.29, stratify = y)
```

```
features = []
index = numpy.argsort(extratrees.feature_importances_)[::-1][:nbfeatures]
```

All the required features

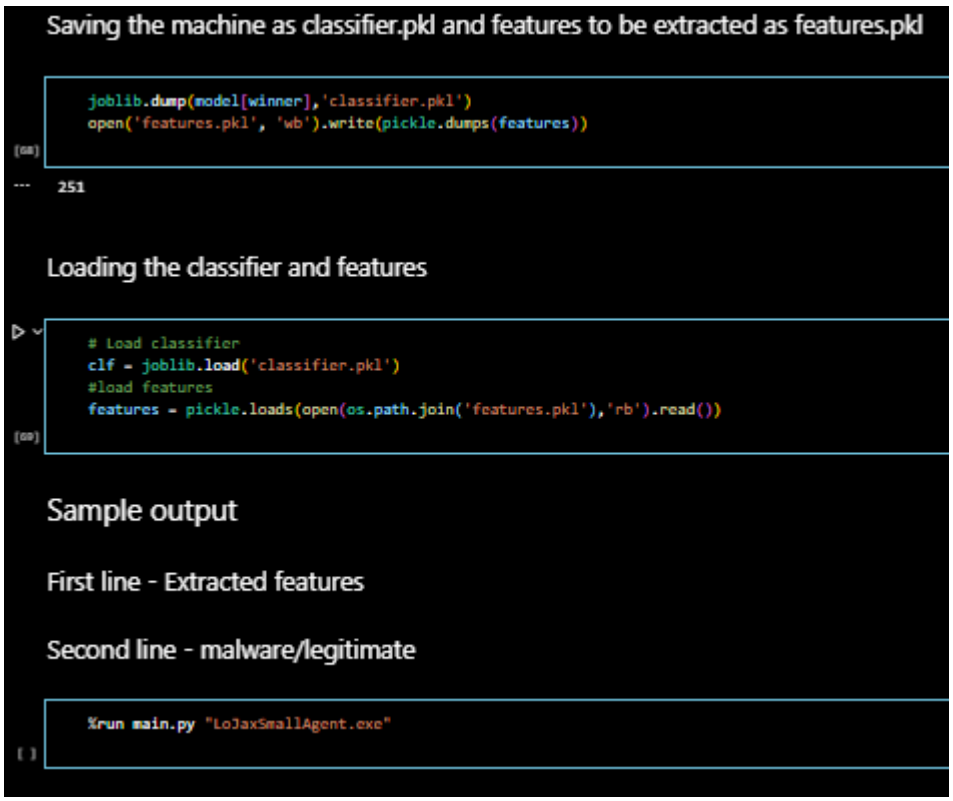
```
for f in range(nbfeatures):
    print("%d. feature %s (%f)" % (f + 1, dataset.columns[2+index[f]], extratrees.feature_importances_[index[f]]))
    features.append(dataset.columns[2+f])
```

Testing which Classifier will give better result

```
model = { "DecisionTree": DecisionTreeClassifier(max_depth=10),
          "RandomForest":ek.RandomForestClassifier(n_estimators=50)}
```

```
results = {}
for algo in model:
    clf = model[algo]
    clf.fit(X_train,y_train)
    score = clf.score(X_test,y_test)
    print ("%s : %s " %(algo, score))
    results[algo] = score
```

```
winner = max(results, key=results.get)# Selecting the classifier with good result
print("Using", winner, "for classification, with",len(features), 'features.')
```



4) url.ipynb

Project Report: URL Classification Using Logistic Regression
Introduction:
The project endeavors to employ machine learning techniques, specifically logistic regression, to classify URLs as either malicious or legitimate. By training a classification model on a dataset of URLs and utilizing TF-IDF vectorization for feature extraction, the project aims to provide an effective solution for cybersecurity measures. This report offers a comprehensive overview of the project implementation, covering data preprocessing, model training, and evaluation processes.

1. Importing Required Libraries:
The project begins by importing essential libraries such as pandas, numpy, random, pickle, train_test_split from sklearn.model_selection, TfidfVectorizer, and LogisticRegression. These libraries play pivotal roles in data manipulation, feature extraction, model training, and evaluation.

2. Importing the Dataset:
The dataset comprising URLs is imported from a CSV file named data_url.csv. Leveraging the Pandas library, the data is loaded into a DataFrame for seamless handling. Subsequently, the DataFrame is converted into a NumPy array for further processing. To mitigate bias during training, the data is shuffled to randomize the order.
3. Separating Data by Characteristics:
The URLs and their corresponding labels are segregated into two lists: urls and y. The urls list contains the URLs, while y encompasses the corresponding labels denoting malicious or legitimate URLs.

4. Sanitization of URLs:
Given the unique structure and special characters inherent in URLs, a sanitization method is devised. This function preprocesses URLs by converting them to lowercase and tokenizing them based on specific characters such as '-', '/', and '.'. Furthermore, duplicate tokens are removed, and the 'com' token is excluded if present.

5. Custom Vectorizer Function:
To ensure proper tokenization and transformation of URLs into feature vectors, a custom TF-IDF vectorizer function is initialized. This function utilizes the previously defined sanitization method as the tokenizer, enabling accurate feature extraction for model training.

6. Splitting the Dataset:
The dataset undergoes division into training and testing sets utilizing the train_test_split function from sklearn.model_selection. The training set constitutes 80% of the data, while the remaining 20% forms the testing set. This partition facilitates model training on a subset of the data while evaluating its performance on unseen data.

7. Model Training:
A logistic regression model is instantiated with the 'lbfgs' solver and a maximum iteration of 1000. Subsequently, the model is trained on the training data employing the fit method. The accuracy score of the model is computed using the testing data, with the result being printed for analysis.

8. Saving the Model and Vectorizer:
To enable seamless reuse of the trained logistic regression model and TF-IDF vectorizer, both entities are saved to disk utilizing the pickle module. This functionality streamlines the deployment process in production environments for real-time URL classification tasks, thereby enhancing cybersecurity measures.

Conclusion:
In summary, this project showcases the efficacy of logistic regression for URL classification, bolstered by TF-IDF vectorization for feature extraction. Through meticulous preprocessing and model training, the project furnishes an effective mechanism for discerning between malicious and legitimate URLs. The preserved model and vectorizer hold the potential for deployment in production environments, contributing significantly to bolstering cybersecurity measures.

Importing all the required libraries

```
import pandas as pd
import numpy as np
import random
import pickle
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
```

Importing the Dataset

Source :

```
url = 'data_url.csv'
url_csv = pd.read_csv(url,',',error_bad_lines=False)

#converting the data from csv to dataframe for easy handling
url_df = pd.DataFrame(url_csv)

#to convert into array
url_df = np.array(url_df)
random.shuffle(url_df)
```

Seperating the data according to it's characteristics

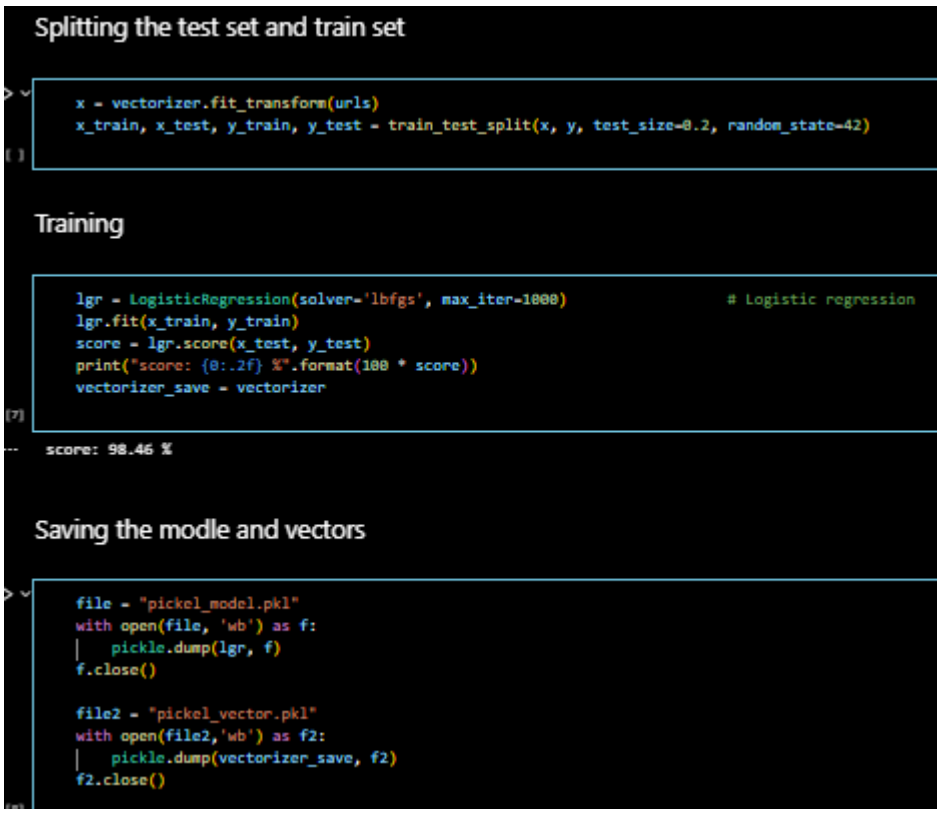
```
y = [d[1] for d in url_df]
urls = [d[0] for d in url_df]
```

Since the urls are different from our normal text documents, we need to use a sanitization method to get the relevant data from raw urls.

```
def sanitization(web):
    web = web.lower()
    token = []
    dot_token_slash = []
    raw_slash = str(web).split('/')
    for i in raw_slash:
        # removing slash to get token
        raw1 = str(i).split('-')
        slash_token = []
        for j in range(0,len(raw1)):
            # removing dot to get the tokens
            raw2 = str(raw1[j]).split('.')
            slash_token = slash_token + raw2
        dot_token_slash = dot_token_slash + raw1 + slash_token
    # to remove same words
    token = list(set(dot_token_slash))
    if 'com' in token:
        #remove com
        token.remove('com')
    return token
```

We will have to pass the data to our custom vectorizer function using Tf-idf approach

```
# term-frequency and inverse-document-frequency
vectorizer = TfidfVectorizer(tokenizer=sanitization)
```



5) Main.py

Overview:
The provided code snippet delineates an interface for a sophisticated Malware Detection system. It offers users the flexibility to conduct scans on Portable Executable (PE) files and URLs, empowering them to safeguard their systems against potential threats effectively.
Functions:

1. run_PE():
    - This function facilitates user input for the path and name of a file.
    - Subsequently, it invokes the execution of a designated Python script, PE_main.py, located within the Extract directory, passing the provided file path as a crucial parameter.
2. run_URL():
    - Initiates the execution of another crucial Python script, url_main.py, situated within the Extract directory, ostensibly designed to handle URL scanning operations.
3. exit():
    - Executes a system command to gracefully terminate the program, ensuring clean closure and resource release.

4. start():
    - Presents users with a sophisticated welcome message, elegantly rendered using ASCII art, under the distinctive title "Malware Detector."
    - Offers users a selection of actions through a meticulously crafted menu, comprising options for PE scanning, URL scanning, or program exit.
    - Captures user input to discern the chosen action, seamlessly directing program flow to the corresponding functionality (run_PE(), run_URL(), or exit()).
    - Upon completion of the selected action, prompts the user to ascertain their intent to conduct another search, thereby enhancing user engagement and iterative usage of the application.

User Interaction:

- The program commences with an aesthetically pleasing ASCII art title, setting a professional tone for user engagement.
- Users are then presented with a meticulously designed menu, affording them intuitive access to essential functionalities.
- Precise error handling mechanisms ensure a smooth and graceful exit in case of invalid user input, enhancing user experience and program robustness.
- Post-action execution, users are gracefully prompted to determine their inclination towards conducting additional searches, fostering iterative usage and user satisfaction.

Execution Flow:

1. The program initializes by invoking the start() function, ushering users into the application interface.
2. Users are empowered to select an action from the menu, steering program flow towards the chosen functionality.
3. The selected action is executed seamlessly, leveraging the underlying system architecture and designated scripts.
4. Upon action completion, users are gracefully prompted to discern their intention to continue with additional searches or gracefully exit the program.
5. Depending on user input, the program seamlessly transitions between functionalities or concludes its operation, ensuring a fluid and intuitive user experience.

main.py > ...

```python
import os
import time
import pyfiglet

def run_PE():
    file = input("Enter the path and name of the file : ")
    os.system("python Extract/PE_main.py {}".format(file))

def run_URL():
    os.system('python Extract/url_main.py')

def exit():
    os.system('exit')

def start():
    print(pyfiglet.figlet_format("Malware Detector"))
    print(" Welcome to antimalware detector \n")
    print(" 1. PE scanner")
    print(" 2. URL scanner")
    print(" 3. Exit\n")

    select = int(input("Enter your choice : "))

    if (select in [1,2,3]):

        if(select == 1):
            run_PE()
            choice = input("Do you want to search again? (y/n)")
            if(choice not in ['Y','N','n','y']):
                print("Bad input\nExiting...")
                time.sleep(3)
                exit()
            else:
                if(choice == 'Y' or 'y'):
                    start()
                elif(choice == 'N' or 'n'):
                    exit()


        elif(select == 2):
            run_URL()
            choice = input("Do you want to search again? (y/n)")
            if(choice not in ['Y','N','n','y']):
                print("Bad input\nExiting...")
                time.sleep(3)
                exit()
            else:
                if(choice == 'Y' or 'y'):
                    start()
                else:
                    exit()

        else:
            exit()
    else:
        print("Bad input\nExiting...")
        time.sleep(3)
        exit()

start()
```

Testing and Evaluation

With the UI and Reporting elements in place, the focus transitions to Testing and Evaluation, a critical phase to assess the effectiveness of the adaptive antivirus system in diverse scenarios. Rigorous testing ensures that the system aligns with predefined objectives and operates seamlessly. The systematic approach is outlined below:

1. Diverse Dataset Testing:

   Scenario-Based Testing: Simulate real-world conditions through diverse dataset testing, incorporating known and novel threats across various attack vectors.

   Effectiveness Evaluation: Evaluate the system's effectiveness in detecting and mitigating threats, emphasizing its real-world applicability.

2. Performance Metrics:

   Detection Accuracy: Assess the system's accuracy in correctly identifying malicious entities while minimizing false positives.

   Response Time: Measure the system's responsiveness in adapting to and mitigating threats in a timely manner.

   Resource Utilization: Evaluate the impact of the adaptive antivirus on system resources to ensure efficiency.

3. Scalability Assessment:

   System Scalability: Assess the system's scalability to accommodate varying data volumes and user loads.

   Adaptability to Growth: Ensure the adaptive antivirus system can scale seamlessly with evolving cybersecurity requirements.

4. User Experience Testing:

Usability Assessment: Gauge the usability of the UI through user experience testing, considering factors such as intuitiveness and navigation.

User Feedback Incorporation: Integrate user feedback from the UI into the evaluation process, ensuring alignment with user expectations.

5. Security Testing:

   Vulnerability Assessment: Conduct security testing to identify and address potential vulnerabilities within the adaptive antivirus system.

   Resilience to Attacks: Evaluate the system's resilience to potential cyber-attacks, ensuring its robustness in the face of adversarial scenarios.

6. Real-World Deployment Simulation:

   Pilot Deployment: Simulate a pilot deployment in real-world environments to validate the system's performance and adaptability.

   Monitoring and Feedback Integration: Incorporate monitoring mechanisms to gather user feedback during the pilot deployment, facilitating continuous improvement.

7. Iterative Refinement:

Feedback Utilization: Utilize feedback from testing and evaluation phases to iteratively refine the adaptive antivirus system.

Continuous Improvement Loop: Establish a continuous improvement loop, ensuring that the system evolves dynamically in response to emerging threats and user needs

CONCLUSION

In the initial stages of developing an adaptive antivirus system, the focus revolves around meticulous data collection and the creation of a diverse dataset capturing various cyber threats. This comprehensive approach includes malware samples ranging from traditional viruses to sophisticated threats like trojans and ransomware. The dataset is categorized based on different attack vectors, real-world scenarios, and enriched with metadata and context, including file attributes, system calls, and network activity. The integration of threat intelligence further enhances the dataset's richness. Rigorous data quality assurance processes ensure validation, cleaning, and balancing for optimal learning during subsequent phases. Feature engineering follows, identifying critical features such as file attributes, system calls, and network activity. Advanced explorations include API calls, code execution patterns, and payload analysis, all informed by collaboration with cybersecurity experts and the incorporation of threat intelligence for a nuanced understanding of evolving threats.

Subsequently, the machine learning model deployment phase employs supervised learning for classification, unsupervised learning for anomaly detection, and reinforcement learning for adaptive responses. The integration of advanced techniques like transfer learning and ensemble methods, along with validation and hyperparameter tuning, ensures robust model performance. The continuous learning framework facilitates periodic retraining and adaptation to emerging threats. The behavioral analysis and adaptive response mechanisms phase emphasizes the development of algorithms for understanding system behavior, anomaly detection, and dynamic responses. This involves statistical approaches, machine learning models, and user-centric feedback loops, creating a system that continuously learns, adapts, and improves. The final phases include the creation of a user interface and reporting system for transparent interaction and feedback, comprehensive testing and evaluation across diverse scenarios, and iterative refinement based on user feedback, ensuring the adaptive antivirus system evolves dynamically to address emerging cyber threats and user needs.

6.1 Reference:

1. U. Tatar, B. Nussbaum, Y. Gokce, and O. F. Keskin, “Digital force majeure: the mondelez case, insurance, and the (un) certainty of attribution in cyberattacks,” *Business Horizons*, vol. 64, 2021.
   View at: Publisher Site | Google Scholar
2. S. K. Sahay, A. Sharma, and H. Rathore, “Evolution of malware and its detection techniques,” in *Information and Communication Technology for Sustainable Development*, pp. 139–150, Springer, Singapore, 2020.
   View at: Publisher Site | Google Scholar
3. W. Huang and J. W. Stokes, “Mtnet: a multi-task neural network for dynamic malware classification,” in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment-Volume 9721, DIMVA 2016*, pp. 399–418, Springer-Verlag New York, Inc., New York, NY, USA, June 2016.
   View at: Google Scholar
4. W. Hu and Y. Tan, “Black-box attacks against rnn based malware detection algorithms,” in *Proceedings of the Workshops at the mThirty-Second AAAI Conference on Artificial Intelligence*, Peking University, Beijing, China, February 2018.
   View at: Google Scholar
5. Y. Chen, Z. Shan, F. Liu et al., “A gene-inspired malware detection approach,” in *Journal of Physics: Conference Series*, vol. 1168, IOP Publishing, 2019.
   View at: Publisher Site | Google Scholar
6. M. Ijaz, M. H. Durad, and M. Ismail, “Static and dynamic malware analysis using machine learning,” in *Proceedings of the 2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pp. 687–691, IEEE, Islamabad, Pakistan, January 2019.
   View at: Google Scholar
7. I. You and K. Yim, “Malware obfuscation techniques: a brief survey,” in *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pp. 297–300, IEEE, Fukuoka, Japan, November 2010.
   View at: Google Scholar
8. B. Kolosnjaji, A. Demontis, B. Biggio et al., “Adversarial malware binaries: evading deep learning for malware detection in executables,” in

Proceedings of the 2018 26th European Signal Processing Conference (EUSIPCO), pp. 533–537, IEEE, Rome, Italy, September 2018.
View at: Google Scholar

9. D. Carlin, P. O'Kane, and S. Sezer, "A cost analysis of machine learning using dynamic runtime opcodes for malware detection," *Computers & Security*, vol. 85, pp. 138–155, 2019.
View at: Publisher Site | Google Scholar
10. T. Shibahara, T. Yagi, M. Akiyama, D. Chiba, and T. Yada, "Efficient dynamic malware analysis based on network behavior using deep learning," in *Proceedings of the 2016 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–7, IEEE, Washington, DC, USA, December 2016.
View at: Google Scholar
11. M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *Computers & Security*, vol. 77, pp. 578–594, 2018.
View at: Publisher Site | Google Scholar
12. J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proceedings of the 2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 11–20, IEEE, Fajardo, PR, USA, October 2015.
View at: Google Scholar
13. F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: eliminating experimental bias in malware classification across space and time," in *Proceedings of the 28th {USENIX} Security Symposium ({USENIX} Security 19), {USENIX} Association*, pp. 729–746, Santa Clara, CA, USA, September 2019.
View at: Google Scholar
14. K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," 2016, https://arxiv.org/abs/1606.04435 arXiv preprint arXiv:1606.04435.
View at: Google Scholar
15. H. Sayadi, N. Patel, S. M. Pd, A. Sasan, S. Rafatirad, and H. Homayoun, "Ensemble learning for effective run-time hardware-based malware detection: a comprehensive analysis and classification," in

Proceedings of the 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), pp. 1–6, IEEE, San Francisco, CA, USA, June 2018.
View at: Google Scholar

16. N. Usman, S. Usman, F. Khan et al., "Intelligent dynamic malware detection using machine learning in ip reputation for forensics data analytics," *Future Generation Computer Systems*, vol. 118, pp. 124–141, 2021.
View at: Publisher Site | Google Scholar
17. P. Burnap, R. French, F. Turner, and K. Jones, "Malware classification using self organising feature maps and machine activity data," *Computers & Security*, vol. 73, pp. 399–410, 2018.
View at: Publisher Site | Google Scholar
18. M. Rhode, L. Tuson, P. Burnap, and K. Jones, "Lab to soc: robust features for dynamic malware detection," in *Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Industry Track*, pp. 13–16, IEEE, Portland, OR, USA, June 2019.
View at: Google Scholar
19. H. Sayadi, A. Houmansadr, S. Rafatirad, H. Homayoun, and P. D. Sai Manoj, "Comprehensive assessment of run-time hardware-supported malware detection using general and ensemble learning," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pp. 212–215, ACM, Ischia, Italy, May 2018.
View at: Google Scholar
20. S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: towards efficient real-time protection against malware," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 2, pp. 289–302, 2016.
View at: Publisher Site | Google Scholar
21. M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: a framework for efficient online malware detection," in *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 651–661, IEEE, Burlingame, CA, USA, February. 2015.
View at: Google Scholar
22. X. Yuan, "Phd forum: deep learning-based real-time malware detection with multi-stage analysis," in *Proceedings of the 2017 IEEE*

International Conference on Smart Computing (SMARTCOMP), pp. 1-2, IEEE, Hong Kong, China, May 2017.
View at: Google Scholar

23. R. Sun, X. Yuan, P. He et al., "Learning fast and slow: PROPEDEUTICA for real-time malware detection," *CoRR*, vol. abs/1712, Article ID 01145, 2017.
View at: Google Scholar

24. N. Scaife, H. Carter, P. Traynor, and K. R. Butler, "Cryptolock (and drop it): stopping ransomware attacks on user data," in *Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pp. 303–312, IEEE, Nara, Japan, June 2016.
View at: Google Scholar

25. GlobalStats, "Market share of windows operating system versions," 2018, http://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide#monthly-201708-201709-bar.
View at: Google Scholar

26. J. Benda, A. Longtin, and L. Maler, "Spike-frequency adaptation separates transient communication signals from background oscillations," *Journal of Neuroscience*, vol. 25, no. 9, pp. 2312–2321, 2005.
View at: Publisher Site | Google Scholar

27. H.-D. Huang, C.-S. Lee, H.-Y. Kao, Y.-L. Tsai, and J.-G. Chang, "Malware behavioral analysis system: Twman," in *Proceedings of the 2011 IEEE Symposium on Intelligent Agent (IA)*, pp. 1–8, IEEE, Paris, France, April 2011.
View at: Google Scholar

28. T. Kim, B. Kang, and E. G. Im, "Runtime detection framework for android malware," *Mobile Information Systems*, vol. 2018, 2018.
View at: Publisher Site | Google Scholar

29. R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, vol. 135, MIT Press Cambridge, Cambridge, MA, USA, 1998.

30. C. J. C. H. Watkins, "Learning from delayed rewards," King's University, London, UK, 1989, PhD thesis.
View at: Google Scholar

31. C. J. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
View at: Publisher Site | Google Scholar