

AUTOMATION DETECTION OF MALWARE AND STENOGRAPHICAL CONTENT USING MACHINE LEARNING

Submitted in partial fulfillment of the requirements for
the award of
Bachelor of Engineering degree in Computer Science and Engineering

By

Henry D Samuel . (38110199)
Santhana Kumar M . (38110503)



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SCHOOL OF COMPUTING**

SATHYABAMA

**INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)**

**Accredited with Grade "A" by NAAC
JEPPIAAR NAGAR, RAJIV GANDHI SALAI,
CHENNAI - 600 119**

MARCH-2022



SATHYABAMA

**INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)**

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
BONAFIDE CERTIFICATE**

This is to certify that this Project Report is the bonafide work of Henry D Samuel (REG NO: 38110199), Santhana Kumar M. (REG NO: 38110503) who have done Project work as a team who carried out the project entitled "**AUTOMATION DETECTION OF MALWARE AND STENOGRAPHICAL CONTENT USING MACHINE LEARNING**" under my supervision from November 2021 to March 2022.

**Internal Guide
Ms. AISHWARYA R M.E.,**

**Head of the Department
Dr. L. Lakshmanan M.E., Ph.D.,
Dr.S.Vigneshwari M.E., Ph.D.,**

Submitted for Viva voce Examination held on _____

Internal Examiner

External Examiner

DECLARATION

We Henry D Samuel (REG NO: 38110199) and Santhana Kumar M. (REG NO: 38110503) hereby declare that the Project Report entitled **“AUTOMATION DETECTION OF MALWARE AND STENOGRAPHICAL CONTENT USING MACHINE LEARNING”** done by me under the guidance of **Ms. AISHWARYA R M.E.**, is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering degree in Computer Science and Engineering

DATE:

PLACE:

SIGNATURE OF THE CANDIDATE

ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to **Board of Management of SATHYABAMA** for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to **Dr. T. Sasikala M.E., Ph.D, Dean, School of Computing Dr. L. Lakshmanan M.E., Ph.D. , and Dr.S.Vigneshwari M.E., Ph.D. Heads** of the Department of Computer Science and Engineering for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide **Ms. Aishwarya R. M.E.**, for her valuable guidance, suggestions and constant encouragement paved way for the successful completion of my project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the **Department of Computer Science and Engineering** who were helpful in many ways for the completion of the project.

ABSTRACT

In recent times many malware attacks increasing in our society. Mainly image-based malware attacks are spreading worldwide and many people get harmful malware-based images through the technique called steganography. In the existing system, only open malware and files from the internet is identified.

The image-based malware cannot be identified and detected so many phishers make use of this technique and exploit the target. Social media platforms would be totally harmful to the users. To avoid these difficulties, by implementing Machine learning we can find the steganographic malware images(contents).

Our proposed methodology developing an Automation detection of malware and steganographic content using Machine Learning. Steganography is the field of hiding messages in apparently innocuous media (e.g., images), and steganalysis is the field of detecting this covert malware.

We propose a machine learning (ML) approach to steganalysis. In the existing system, only open malware and files from the internet are identified. But in recent times many people get harmful malware-based images through the technique called steganography. Social media platforms would be totally harmful to the users.

To avoid these difficulties, by implementing Machine learning we can find the steganographic malware images(contents). We use the steganalysis method using machine learning for logistic classification. By using this we can spot and get escape from the malware images sharing in social media like WhatsApp, Facebook without downloading it. It can be also used for all the photo-sharing sites such as google photos.

LIST OF FIGURES

Figure no.	Name of the Figure	Page no.
4.1	Input JPG image	41
4.2	Output image	41
4.3	Change in Output image	42
4.4	Malware Detection simulation	43
4.5	RGB Layer Identification Step	44
5.1	LSB Graph	50
5.2	False rate graph	50
5.3	Output image	51
5.4	Output image	51
5.5	Binary code image	52

TABLE OF CONTENT

CHAPTER NO.	TITLE	PAGE NO
1	INTRODUCTION	1
2	LITERATURE SURVEY	
	2.1 Survey Walk Through	2
	2.2 Tensor Flow	2
	2.3 Opencv	2
	2.4 keras	6
	2.5 Numpy	7
	2.6 Neural Networks	9
	2.7 Convolutional Neural Network	14
3	IMPLEMENTATION	
	3.1 Image Processing	19
	3.1.1 Digital Image Processing	19
	3.1.2 Pattern Recognition	20
	3.2 Basic approaches to malware detection	21

	3.3 Machine learning	22
4	3.4 Unsupervised Learning	22
METHODOLOGY	3.5 Supevised Learning	23
	4.3.6 Deep Learning	24
	13.7 Machine Learning Applications	24
	Methodology	29
	4.1.1 Training Model	29
	4.1.2 Segmentation	29
	4.2 Classification	30
	4.3 Testing	34
5	RESULT	
	5.1 Result	49
	5.2 Performance Analysis	52
6	CONCLUSION AND FUTURE SCOPE	
	6.1 Future Scope	54
	6.2 Conclusion	54
7	APPENDIX	
	a) Sample code	58

CHAPTER 1

INTRODUCTION

By definition, steganography is a technique or art of concealing a type of data within a different type of data. The word steganography derives from the Greek words stegano (sealed) and graph (writing), thus meaning "writing a sealed message. The technique was historically used by governments to hide sensitive information. One interesting form of steganography sends and receives secret messages publicly.

There is no way to discover the hidden message except by the sender and receiver. Because the secret message is embedded in the cover file, anyone observing it as an ordinary file does not notice that the cover file contains secret information, thus making steganography more secure.

The person who knows whether the cover file contains secret information is the only one who can attempt to steal it. Machine learning is the main domain used for modern steganography purposes. The major reason is the modern problem needs a modern solution. Machine learning powerful prediction algorithm helps to find out the stego content. It can be also useful for filtering the contents in the transmission area.

Image Steganography is a type of steganography. Common template is already programmed regarding the stego and the software identifies the text by matching the template.[5]

A review of LSB image steganography techniques is used for small types of text and URLs. It cannot find large-sized texts compared to the other techniques. It is mostly based on the LSB algorithm and its accuracy level is very low.[6]

Detection of LSB alternate and LSB identical Steganography Using Gray Level Run Length Matrix Using an old model system which is very useful for encrypting the system. Grayscale image recognition is very useful for encrypting the text alone and it is not useful for encrypting malware attacks.[7] Enhance security and ability for Arabic text steganography using 'Kashida' extensions. is very time-consuming for encrypting the texts.

CHAPTER 2

LITERATURE SURVEY

2.1 SURVEY WALKTHROUGH:

The domain analysis that we have done for the project mainly involved understanding the neural networks

2.2 TensorFlow:

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. It is used for both research and production at Google.

Features: TensorFlow provides stable Python (for version 3.7 across all platforms) and C APIs; and without API backwards compatibility guarantee: C++, Go, Java, JavaScript and Swift (early release). Third-party packages are available for C#, Haskell Julia, MATLAB,R, Scala, Rust, OCaml, and Crystal."New language support should be built on top of the C API. However, not all functionality is available in C yet." Some more functionality is provided by the PythonAPI.

Application: Among the applications for which TensorFlow is the foundation, are automated image-captioning software, suchas DeepDream.

2.3 Opencv:

OpenCV (Open Source Computer Vision Library) is a library of programming functions mainly aimed at real-time computer vision.[1] Originally developed by Intel, it was later supported by Willow Garage then Itseez (which was later acquired by Intel[2]). The library is cross-platform and free for use under the open-source BSD license.

OpenCV's application areas include:

- 2D and 3D feature toolkits
- Egomotion estimation
- Facial recognition system
- Gesture recognition
- Human–computer interaction (HCI)
- Mobile robotics
- Motion understanding
- Object identification
- Segmentation and recognition

Stereopsis stereo vision: depth perception from 2 cameras

- Structure from motion (SFM).
- Motion tracking
- Augmented reality

To support some of the above areas, OpenCV includes a statistical machine learning library that contains:

- Boosting
- Decision tree learning
- Gradient boosting trees
- Expectation-maximization algorithm
- k-nearest neighbor algorithm
- Naive Bayes classifier
- Artificial neural networks
- Random forest
- Support vector machine (SVM)
- Deep neural networks (DNN)

AForge.NET, a computer vision library for the Common Language Runtime (.NET Framework and Mono).

ROS (Robot Operating System). OpenCV is used as the primary vision package in ROS.

VXL, an alternative library written in C++.

Integrating Vision Toolkit (IVT), a fast and easy-to-use C++ library with an optional interface to OpenCV.

CVIPtools, a complete GUI-based computer-vision and image-processing software environment, with C function libraries, a COM-based DLL, along with two utility programs for algorithm development and batch processing.

OpenNN, an open-source neural networks library written in C++. List of free and open source software packages

- OpenCV Functionality
- Image/video I/O, processing, display (core, imgproc, highgui)
- Object/feature detection (objdetect, features2d, nonfree)
- Geometry-based monocular or stereo computer vision (calib3d, stitching, videostab)
- Computational photography (photo, video, superres)
- Machine learning & clustering (ml, flann)
- CUDA acceleration (gpu)

- Image-Processing:

Image processing is a method to perform some operations on an image, in order to get an enhanced image and or to extract some useful information from it.

If we talk about the basic definition of image processing then “Image

processing is the analysis and manipulation of a digitized image, especially in order to improve its quality”.

Digital-Image :

An image may be defined as a two-dimensional function $f(x, y)$, where x and y are spatial(plane) coordinates, and the amplitude of f at any pair of coordinates (x, y) is called the intensity or grey level of the image at that point.

In another word An image is nothing more than a two-dimensional matrix (3-D in case of coloured images) which is defined by the mathematical function $f(x, y)$ at any point is giving the pixel value at that point of an image, the pixel value describes how bright that pixel is, and what colour it should be.

Image processing is basically signal processing in which input is an image and output is image or characteristics according to requirement associated with that image. Image processing basically includes the following three steps :
Importing the image. Analysing and manipulating the image
Output in which result can be altered image or report that is based on image analysis

Applications of Computer Vision:

Here we have listed down some of major domains where Computer Vision is heavily used.

- Robotics Application
- Localization – Determine robot location automatically
- Navigation
- Obstacles avoidance
- Assembly (peg-in-hole, welding, painting)
- Manipulation (e.g. PUMA robot manipulator)
- Human Robot Interaction (HRI) – Intelligent robotics to interact with and serve people
- Medicine Application
- Classification and detection (e.g. lesion or cells classification and tumor

detection)

- 2D/3D segmentation
- 3D human organ reconstruction (MRI or ultrasound)
- Vision-guided robotics surgery
- Industrial Automation Application
- Industrial inspection (defect detection)
- Assembly
- Barcode and package label reading
- Object sorting
- Document understanding (e.g. OCR)
- Security Application
- Biometrics (iris, finger print, face recognition)
- Surveillance – Detecting certain suspicious activities or behaviors
- Transportation Application
- Autonomous vehicle
- Safety, e.g., driver vigilance monitoring

2.4 Keras:

Keras is an open-source neural-network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. It was developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System), and its primary author and maintainer is François Chollet, a Google engineer. Chollet also is the author of the Xception deep neural network model.

Features: Keras contains numerous implementations of commonly used neural-network building blocks such as layers, objectives, activation functions, optimizers, and a host of tools to make working with image and text data easier to simplify the

coding necessary for writing deep neural network code. The code is hosted on GitHub, and community support forums include the GitHub issues page, and a Slack channel.

In addition to standard neural networks, Keras has support for convolutional and recurrent neural networks. It supports other common utility layers like dropout, batch normalization, and pooling.

Keras allows users to productize deep models on smartphones (iOS and Android), on the web, or on the Java Virtual Machine. It also allows use of distributed training of deep-learning models on clusters of Graphics processing units (GPU) and tensor processing units (TPU) principally in conjunction with CUDA.

Keras applications module is used to provide pre-trained model for deep neural networks. Keras models are used for prediction, feature extraction and fine tuning. This chapter explains about Keras applications in detail.

Pre-trained models

Trained model consists of two parts model Architecture and model Weights. Model weights are large file so we have to download and extract the feature from ImageNet database. Some of the popular pre-trained models are listed below,

- ResNet
- VGG16
- MobileNet
- InceptionResNetV2
- InceptionV3

2.5 Numpy:

NumPy (pronounced /'nʌmpaɪ/ (NUM-py) or sometimes /'nʌmpi/ (NUM-pee)) is a library for the Python programming language, adding support for large, multi-

dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open- source software and has many contributors.

Features: NumPy targets the CPython reference implementation of Python, which is a non-optimizing bytecode interpreter. Mathematical algorithms written for this version of Python often run much slower than compiled equivalents. NumPy addresses the slowness problem partly by providing multidimensional arrays and functions and operators that operate efficiently on arrays, requiring rewriting some code, mostly inner loops using NumPy.

Using NumPy in Python gives functionality comparable to MATLAB since they are both interpreted, and they both allow the user to write fast programs as long as most operations work on arrays or matrices instead of scalars. In comparison, MATLAB boasts a large number of additional toolboxes, notably Simulink, whereas NumPy is intrinsically integrated with Python, a more modern and complete programming language. Moreover, complementary Python packages are available; SciPy is a library that adds more MATLAB-like functionality and Matplotlib is a plotting package that provides MATLAB-like plotting functionality. Internally, both MATLAB and NumPy rely on BLAS and LAPACK for efficient linear algebra computations.

Python bindings of the widely used computer vision library OpenCV utilize NumPy arrays to store and operate on data. Since images with multiple channels are simply represented as three-dimensional arrays, indexing, slicing or masking with other arrays are very efficient ways to access specific pixels of an image. The NumPy array as universal data structure in OpenCV for images, extracted feature points, filter kernels and many more vastly simplifies the programming workflow and debugging.

Limitations: Inserting or appending entries to an array is not as trivially

possible as it is with Python's lists. The `np.pad(...)` routine to extend arrays actually creates new arrays of the desired shape and padding values, copies the given array into the new one and returns it. NumPy's `np.concatenate([a1,a2])` operation does not actually link the two arrays but returns a new one, filled with the entries from both given arrays in sequence. Reshaping the dimensionality of an array with `np.reshape(...)` is only possible as long as the number of elements in the array does not change. These circumstances originate from the fact that NumPy's arrays must be views on contiguous memory buffers. A replacement package called Blaze attempts to overcome this limitation.

Algorithms that are not expressible as a vectorized operation will typically run slowly because they must be implemented in "pure Python", while vectorization may increase memory complexity of some operations from constant to linear, because temporary arrays must be created that are as large as the inputs. Runtime compilation of numerical code has been implemented by several groups to avoid these problems; open source solutions that interoperate with NumPy include `scipy.weave`, `numexpr` and `Numba`. `Cython` and `Pythran` are static-compiling alternatives to these.

2.6 Neural Networks:

A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. In this sense, neural networks refer to systems of neurons, either organic or artificial in nature. Neural networks can adapt to changing input; so the network generates the best possible result without needing to redesign the output criteria. The concept of neural networks, which has its roots in artificial intelligence, is swiftly gaining popularity in the development of trading systems.

A neural network works similarly to the human brain's neural network. A "neuron" in a neural network is a mathematical function that collects and

classifies information according to a specific architecture. The network bears a strong resemblance to statistical methods such as curve fitting and regression analysis.

A neural network contains layers of interconnected nodes. Each node is a perceptron and is similar to a multiple linear regression. The perceptron feeds the signal produced by a multiple linear regression into an activation function that may be nonlinear.

In a multi-layered perceptron (MLP), perceptrons are arranged in interconnected layers. The input layer collects input patterns. The output layer has classifications or output signals to which input patterns may map. Hidden layers fine-tune the input weightings until the neural network's margin of error is minimal. It is hypothesized that hidden layers extrapolate salient features in the input data that have predictive power regarding the outputs. This describes feature extraction, which accomplishes a utility similar to statistical techniques such as principal component analysis.

Areas of Application

Followings are some of the areas, where ANN is being used. It suggests that ANN has an interdisciplinary approach in its development and applications.

Speech Recognition

Speech occupies a prominent role in human-human interaction. Therefore, it is natural for people to expect speech interfaces with computers. In the present era, for communication with machines, humans still need sophisticated languages which are difficult to learn and use. To ease this communication barrier, a simple solution could be, communication in a spoken language that is possible for the machine to understand.

Great progress has been made in this field, however, still such kinds of systems are facing the problem of limited vocabulary or grammar along with the issue of retraining of the system for different speakers in different

conditions. ANN is playing a major role in this area. Following ANNs have been used for speech recognition –

Multilayer networks

Multilayer networks with recurrent connections Kohonen self-organizing feature map The most useful network for this is Kohonen Self-Organizing feature map, which has its input as short segments of the speech waveform. It will map the same kind of phonemes as the output array, called feature extraction technique. After extracting the features, with the help of some acoustic models as back-end processing, it will recognize the utterance.

Character Recognition

It is an interesting problem which falls under the general area of Pattern Recognition. Many neural networks have been developed for automatic recognition of handwritten characters, either letters or digits. Following are some ANNs which have been used for character recognition – Multilayer neural networks such as Backpropagation neural networks.

Neocognitron

Though back-propagation neural networks have several hidden layers, the pattern of connection from one layer to the next is localized. Similarly, neocognitron also has several hidden layers and its training is done layer by layer for such kind of applications.

Signature Verification Application

Signatures are one of the most useful ways to authorize and authenticate a person in legal transactions. Signature verification technique is a non-vision based technique.

For this application, the first approach is to extract the feature or rather the

geometrical feature set representing the signature. With these feature sets, we have to train the neural networks using an efficient neural network algorithm. This trained neural network will classify the signature as being genuine or forged under the verification stage.

Human Face Recognition

It is one of the biometric methods to identify the given face. It is a typical task because of the characterization of “non-face” images. However, if a neural network is well trained, then it can be divided into two classes namely images having faces and images that do not have faces.

First, all the input images must be preprocessed. Then, the dimensionality of that image must be reduced. And, at last it must be classified using neural network training algorithm. Following neural networks are used for training purposes with preprocessed image –

Fully-connected multilayer feed-forward neural network trained with the help of back- propagation algorithm.

For dimensionality reduction, Principal Component Analysis PCA is used.

Deep Learning:

Deep-learning networks are distinguished from the more commonplace single-hidden- layer neural networks by their depth; that is, the number of node layers through which data must pass in a multistep process of pattern recognition.

Earlier versions of neural networks such as the first perceptrons were shallow, composed of one input and one output layer, and at most one hidden layer in between. More than three layers (including input and output) qualifies as “deep” learning. So deep is not just a buzzword to make algorithms seem like they read Sartre and listen to bands you haven’t heard of yet. It is a strictly defined term that means more than one hidden layer.

In deep-learning networks, each layer of nodes trains on a distinct set of features based on the previous layer's output. The further you advance into the neural net, the more complex the features your nodes can recognize, since they aggregate and recombine features from the previous layer.

This is known as feature hierarchy, and it is a hierarchy of increasing complexity and abstraction. It makes deep-learning networks capable of handling very large, high- dimensional data sets with billions of parameters that pass through nonlinear functions.

Above all, these neural nets are capable of discovering latent structures within unlabeled, unstructured data, which is the vast majority of data in the world. Another word for unstructured data is raw media; i.e. pictures, texts, video and audio recordings. Therefore, one of the problems deep learning solves best is in processing and clustering the world's raw, unlabeled media, discerning similarities and anomalies in data that no human has organized in a relational database or ever put a name to.

For example, deep learning can take a million images, and cluster them according to their similarities: cats in one corner, ice breakers in another, and in a third all the photos of your grandmother. This is the basis of so-called smart photo albums.

Deep-learning networks perform automatic feature extraction without human intervention, unlike most traditional machine-learning algorithms. Given that feature extraction is a task that can take teams of data scientists years to accomplish, deep learning is a way to circumvent the chokepoint of limited experts. It augments the powers of small data science teams, which by their nature do not scale.

When training on unlabeled data, each node layer in a deep network learns features automatically by repeatedly trying to reconstruct the input from which it draws its samples, attempting to minimize the difference between the network's guesses and the probability distribution of the input data itself.

Restricted Boltzmann machines, for examples, create so-called reconstructions in this manner.

In the process, these neural networks learn to recognize correlations between certain relevant features and optimal results – they draw connections between feature signals and what those features represent, whether it be a full reconstruction, or with labeled data.

A deep-learning network trained on labeled data can then be applied to unstructured data, giving it access to much more input than machine-learning nets.

2.7 Convolution neural network:

Convolutional neural networks (CNN) is a special architecture of artificial neural networks, proposed by Yann LeCun in 1988. CNN uses some features of the visual cortex. One of the most popular uses of this architecture is image classification. For example Facebook uses CNN for automatic tagging algorithms, Amazon — for generating product recommendations and Google — for search through among users' photos.

Instead of the image, the computer sees an array of pixels. For example, if image size is 300 x 300. In this case, the size of the array will be 300x300x3. Where 300 is width, next 300 is height and 3 is RGB channel values. The computer is assigned a value from 0 to 255 to each of these numbers. This value describes the intensity of the pixel at each point.

To solve this problem the computer looks for the characteristics of the baselevel. In human understanding such characteristics are for example the trunk or large ears. For the computer, these characteristics are boundaries or curvatures. And then through the groups of convolutional layers the computer constructs more abstract concepts. In more detail: the image is passed through a series of convolutional, nonlinear, pooling layers and fully connected layers, and then generates the output.

Applications of convolution neural network:

Decoding Facial Recognition:

Facial recognition is broken down by a convolutional neural network into the following major components -

- Identifying every face in the picture
- Focusing on each face despite external factors, such as light, angle, pose, etc.
- Identifying unique features

Comparing all the collected data with already existing data in the database to match a face with a name.

A similar process is followed for scene labeling as well. Analyzing Documents: Convolutional neural networks can also be used for document analysis. This is not just useful for handwriting analysis, but also has a major stake in recognizers. For a machine to be able to scan an individual's writing, and then compare that to the wide database it has, it must execute almost a million commands a minute. It is said with the use of CNNs and newer models and algorithms, the error rate has been brought down to a minimum of 0.4% at a character level, though it's complete testing is yet to be widely seen.

The SGD algorithm is good at identifying harmful websites, but it can make mistakes when evaluating which ones are safe, labelling them as risky

[1]The web mining algorithm will extract textual information from web pages and identify those associated with terrorism. A system whose main purpose is to create a website where people may inspect any webpage or website for any evidence of terrorist activity.

[2]The persuading concept is to see if the feature's equivalent word appears in the mail. When a good classifier is employed to create the classification model, the

experimental results of this method have high TPR and Precision values, and the false positive rate is regulated within an acceptable range

[3]Linguistic characteristics are critical for distinguishing across users' written styles. Sentiment analysis is most effective with content that has a subjective context, such as a suicide note. These characteristics can be derived explicitly from the user profile or inferred implicitly using various data mining tools and methodologies

[4]Using a document embedding, A decision tree model with gradient boosting predicts dangerous categories of gathered web pages.

[5] The ROC curve was used to comprehend a performance measurement for a classification task at various thresholds. The false-positive percentage should be kept as low as possible, whereas the true positive rate should be maximized.

[6]Hierarchical spatial scaling's analytic bias improves the model's ability to handle detection problems in documents of possibly changing sizes. A feature extractor is a programme that parses a neural network model that distinguishes a series of tokens from HTML pages judgments, in this approach.

[7]Cross-channel scripting defence methods follow a website's whole path, consisting of sustained storage systems If the soiled information is not cleansed, an alert is generated. The adversary can use this threat to insert inappropriate material into the user's embedded system. causing web applications to malfunction and information to be leaked.

[8]Clustering methods lustring (e.g., k-means, DBSCAN) To detect malicious domains, unsupervised separation instances evaluate data and derive necessary details from it.

[9]The Dirichlet latent allocation topic model proposed a pattern that determines that tweets about crime are more likely to be positive. However, A stash of tweets is available; however, a series of old tweets is either impossible or prohibitively expensive.

The key insight underlying our work is that an infected application run will modify the control-flow/data structures compared to a benign run. This will be reflected in its memory access pattern. This is obvious for the important class of memory corruption vulnerabilities for code in memory-unsafe programming languages such

as C/C++. The same is true for another important class of malware, kernel rootkits, which modify control flow in the operating system.

While these two classes are used extensively in this paper, control-flow and/or data structure modification are intrinsic characteristics of malware. Thus, we propose hardware monitoring of memory accesses for classifying individual application runs as being malicious or benign. Since virtual addresses provide for a more consistent signature than physical addresses, we propose obtaining the virtual address trace through in-processor monitoring.

A major challenge in monitoring memory accesses is the sheer volume of the data. Our framework addresses this by dividing accesses into epochs, summarizing the memory access patterns of each epoch into features which are then fed to a machine learning classifier. Experiments show this framework is effective in detecting diverse classes of malware. The contributions of this paper are as follows:

- We target application-run-specific malware detection.
- We introduce a framework for malware detection that is based on online analysis of virtual memory access patterns in contrast to physical memory access patterns.
- We introduce novel summarization and feature extraction techniques for function/syscall memory access patterns.
- We demonstrate the feasibility of our approach with experiments that consider both kernel-level and user-level malware. We demonstrate its efficacy through extremely low false-positive and false-negative rates. While the proposed classification methodology is intended to be realized in hardware through in-processor monitoring and classification, the details of hardware design are beyond the scope of this paper. The focus of this paper is demonstrating the value of such a framework, and addressing the data volume concerns in its design.

This paper also focuses on offline learning but with recent breakthroughs in the development of machine learning cores, e.g. [21], we believe even online learning of the detection model is realizable in hardware.

II. Malware and Memory Access Patterns We now describe certain common types of kernel and userlevel malware and how they affect memory accesses.

Kernel Rootkits: Kernel rootkits modify kernel data structures to redirect control flow in system calls to malevolent code. The two most common ways are: system call table modification, which changes a function pointer in the syscall table, and virtual file system (VFS) function pointer hooking, which replaces function pointers in the VFS file operation structure.

User-level Malware: User-level malware primarily exploits memory vulnerabilities: buffer/heap overflow, return-oriented programming, etc. As with kernel-level rootkits, user-level malware introduces anomalous control flow; e.g. in return oriented programming (ROP), the attack executes a sequence of “gadgets” which are carefully chosen from an existing code base, usually from library functions and chained to implement the malicious objective. In this example, the “signature” is the anomalous control jumps to library functions.

III. Monitoring Memory Accesses: Challenges and Solutions As contemporary processors execute billions of instructions/second, storing and analysing all memory accesses is not feasible for online monitoring due to the sheer volume of data. The memory access trace needs to be summarized while retaining essential characteristics that enable malware detection. Another challenge is the lack of delimiters in the raw memory access trace. In most malware infected program runs, the malicious behavior only occurs at certain phases of the execution, the other phases are normal program behavior. Effective identification of these phases normally requires human expert analysis. We address these as follows.

Epoch-based Monitoring: Program execution is divided into epochs and epochs are separated by inserting epoch markers in the memory access stream. We investigate three choices of epoch markers: (i) system calls, (ii) function calls, and (iii) the complete program run. We show experimentally that system calls are

effective epoch markers for kernel-level malware. Function calls are effective epoch markers for userlevel malware. Using the entire program run as an epoch is not feasible for continuous running programs such as web browsers but can be effective for small programs. It still has limitations and this is discussed further in §VI.

CHAPTER 3

3.1 IMAGE PROCESSING

Image processing is a method to perform some operations on an image, in order to get an enhanced image or to extract some useful information from it. It is a type of signal processing in which input is an image and output may be image or characteristics/features associated with that image. Nowadays, image processing is among rapidly growing technologies. It forms core research area within engineering and computer science disciplines too.

Image processing basically includes the following three steps:

- Importing the image via image acquisition tools.
- Analysing and manipulating the image.
- Output in which result can be altered image or report that is based on image analysis.

There are two types of methods used for image processing namely, analogue and digital image processing. Analogue image processing can be used for the hard copies like printouts and photographs. Image analysts use various fundamentals of interpretation while using these visual techniques.

Digital image processing techniques help in manipulation of the digital images by using computers. The three general phases that all types of data have to undergo while using digital technique are pre- processing, enhancement, and display, information extraction.

3.1.1 Digital image processing:

Digital image processing consists of the manipulation of images using digital computers. Its use has been increasing exponentially in the last decades. Its applications range from medicine to entertainment, passing by geological processing and remote sensing. Multimedia systems, one of the pillars of the modern information society, rely heavily on digital image processing.

Digital image processing consists of the manipulation of those finite precision numbers. The processing of digital images can be divided into several classes: image enhancement, image restoration, image analysis, and image compression. In image enhancement, an image is manipulated, mostly by heuristic techniques, so that a human viewer can extract useful information from it.

Digital image processing is to process images by computer. Digital image processing can be defined as subjecting a numerical representation of an object to a series of operations in order to obtain a desired result. Digital image processing consists of the conversion of a physical image into a corresponding digital image and the extraction of significant information from the digital image by applying various algorithms.

3.1.2 Pattern recognition:

On the basis of image processing, it is necessary to separate objects from images by pattern recognition technology, then to identify and classify these objects through technologies provided by statistical decision theory. Pattern recognition is the automated recognition of patterns and regularities in data. It has applications in statistical data analysis, signal processing, image analysis, information retrieval, bioinformatics, data compression, computer graphics and machine learning. Pattern recognition has its origins in statistics and engineering; some modern approaches to pattern recognition include the use of machine learning,

due to the increased availability of big data and a new abundance of processing power. These activities can be viewed as two facets of the same field of application, and they have undergone substantial development over the past few decades.

Pattern recognition systems are commonly trained from labeled "training" data. When no labeled data are available, other algorithms can be used to discover previously unknown patterns. KDD and data mining have a larger focus on unsupervised methods and stronger connection to business use. Pattern recognition focuses more on the signal and also takes acquisition and Signal Processing into consideration. It originated in engineering, and the term is popular in the context of computer vision: a leading computer vision conference is named Conference on Computer Vision and Pattern Recognition.

In machine learning, pattern recognition is the assignment of a label to a given input value. In statistics, discriminant analysis was introduced for this same purpose in 1936. An example of pattern recognition is classification, which attempts to assign each input value to one of a given set of *classes* (for example, determine whether a given email is "spam" or "non-spam"). Pattern recognition is a more general problem that encompasses other types of output as well. Other examples are regression, which assigns a real-valued output to each input;^[1] sequence labeling, which assigns a class to each member of a sequence of values^[2] (for example, part of speech tagging, which assigns a part of speech to each word in an input sentence); and parsing, which assigns a parse tree to an input sentence, describing the syntactic structure of the sentence.^[3]

Pattern recognition algorithms generally aim to provide a reasonable answer for all possible inputs and to perform "most likely" matching of the inputs, taking into account their statistical variation. This is opposed to *pattern matching* algorithms, which look for exact matches in the input with pre-existing patterns. A common example of a pattern-matching algorithm is regular expression matching, which looks for patterns of a given sort in textual data and is included in the search capabilities of many text editors and word processors.

3.2 Basic approaches to malware detection:

An efficient, robust and scalable malware recognition module is the key

component of every cybersecurity product. Malware recognition modules decide if an object is a threat, based on the data they have collected on it. This data may be collected at different phases: – Pre-execution phase data is anything you can tell about a file without executing it. This may include executable file format descriptions, code descriptions, binary data statistics, text strings and information extracted via code emulation and other similar data. – Post-execution phase data conveys information about behavior or events caused by process activity in a system. In the early part of the cyber era, the number of malware threats was relatively low, and simple manually created pre-execution rules were often enough to detect threats. The rapid rise of the Internet and the ensuing growth in malware meant that manually created detection rules were no longer practical - and new, advanced protection technologies were needed. Anti-malware companies turned to machine learning, an area of computer science that had been used successfully in image recognition, searching and decision-making, to augment their malware detection and classification. Today, machine learning boosts malware detection using various kinds of data on host, network and cloud-based anti-malware components. Machine learning: concepts and definitions

3.3 Machine learning: concepts and definitions

According to the classic definition given by AI pioneer Arthur Samuel, machine learning is a set of methods that gives computers “the ability to learn without being explicitly programmed”. In other words, a machine learning algorithm discovers and formalizes the principles that underlie the data it sees. With this knowledge, the algorithm can ‘reason’ the properties of previously unseen samples. In malware detection, a previously unseen sample could be a new file. Its hidden property could be malware or benign. A mathematically formalized set of principles underlying data properties is called the model. Machine learning has a broad variety of approaches that it takes to a solution rather than a single method. These approaches have different capacities and different tasks that they suit best.

3.4 Unsupervised Learning:

One machine learning approach is unsupervised learning. In this setting, we are given only a data set without the right answers for the task. The goal is to discover the structure of the data or the law of data generation. One important example is

clustering. Clustering is a task that includes splitting a data set into groups of similar objects. Another task is representation learning – this includes building an informative feature set for objects based on their low-level description (for example, an autoencoder model). Machine Learning Methods for Malware Detection In this paper, we summarize our extensive experience using machine learning to build advanced protection for our customers. Unsupervised learning 2 Large unlabeled datasets are available to cybersecurity vendors and the cost of their manual labeling by experts is high – this makes unsupervised learning valuable for threat detection. Clustering can help to optimize efforts for the manual labeling of new samples. With informative embedding, we can decrease the number of labeled objects needed for the next machine learning approach in our pipeline: supervised learning. Supervised learning is a setting that is used when both the data and the right answers for each object are available. The goal is to fit the model that will produce the right answers for new objects.

3.5 Supervised learning:

Supervised learning consists of two stages: • Training a model and fitting a model to available training data. • Applying the trained model to new samples and obtaining predictions. The task: • we are given a set of objects • each object is represented with feature set X • each object is mapped to the right answer or labeled as Y This training information is utilized during the training phase, when we search for the best model that will produce the correct label Y for previously unseen objects given the feature set X . In the case of malware detection, X could be some features of file content or behavior, for instance, file statistics and a list of used API functions. Labels Y could be malware or benign, or even a more precise classification, such as a virus, Trojan-Downloader or adware. In the training phase, we need to select a family of models, for example, neural networks or decision trees. Usually, each model in a family is determined by its parameters. Training means that we search for the model from the selected family with a particular set of parameters that gives the most accurate answers for the trained model over the set of reference objects according to a particular metric. In other words, we 'learn' the optimal parameters that define valid mapping from X to Y . After we have trained a model and verified its quality, we are ready for the next phase – applying

the model to new objects. In this phase, the type of the model and its parameters do not change. The model only produces predictions. In the case of malware detection, this is the protection phase. Vendors often deliver a trained model to users where the product makes decisions based on model predictions autonomously. Mistakes can cause devastating consequences for a user – for example, removing an OS driver. It is crucial for the vendor to select a model family properly. The vendor must use an efficient training procedure to find the model with a high detection rate and a low false positive rate.

3.6 Deep Learning:

Deep learning is a special machine learning approach that facilitates the extraction of features of a high level of abstraction from low-level data. Deep learning has proven successful in computer vision, speech recognition, natural language processing and other tasks. It works best when you want the machine to infer high-level meaning from low-level data. For image recognition challenges, like ImageNet, deep learning-based approaches already surpass humans. It is natural that cybersecurity vendors tried to apply deep learning for recognizing malware from low-level data. A deep learning model can learn complex feature hierarchies and incorporate diverse steps of malware detection pipeline into one solid model that can be trained end-to-end, so that all of the components of the model are learned simultaneously.

3.7 Machine learning application specifics in cybersecurity

User products that implement machine learning make decisions autonomously. The quality of the machine learning model impacts the user system performance and its state. Because of this, machine learning-based malware detection has specifics.

Large representative datasets are required

It is important to emphasize the data-driven nature of this approach. A created model depends heavily on the data it has seen during the training phase to determine which features are statistically relevant for predicting the correct label. Let's look at why making a representative data set is so important. Imagine we

collect a training set, and we overlook the fact that occasionally all files larger than 10 MB are all malware and not benign (which is certainly not true for real world files). While training, the model will exploit this property of the dataset, and will learn that any file larger than 10 MB is malware. It will use this property for detection. When this model is applied to real world data, it will produce many false positives. To prevent this outcome, we needed to add benign files with larger sizes to the training set. Then, the model will not rely on an erroneous data set property. Generalizing this, we must train our models on a data set that correctly represents the conditions where the model will be working in the real world. This makes the task of collecting a representative dataset crucial for machine learning to be successful.

The trained model has to be interpretable

Most of the model families used currently, like deep neural networks, are called black box models. Black box models are given the input X , and they will produce Y through a complex sequence of operations that can hardly be interpreted by a human. This could pose a problem in real-life applications. For example, when a false alarm occurs, and we want to understand why it happened, we ask whether it was a problem with a training set or the model itself. The interpretability of a model determines how easy it will be for us to manage it, assess its quality and correct its operation.

False positive rates must be extremely low

False positives happen when an algorithm mistakes a malicious label for a benign file. Our aim is to make the false positive rate as low as possible, or zero. This is not typical for a machine learning application. This is important, because even one false positive in a million benign files can create serious consequences for users. This is complicated by the fact that there are lots of clean files in the world, and they keep appearing. To address this problem, it is important to impose high requirements for both machine learning models and metrics that will be optimized during training, with the clear focus on low false positive rate (FPR) models. This is still not enough, because new benign files that go unseen earlier may occasionally be falsely detected. We take this into account and implement a flexible design of a

model that allows us to fix false-positives on the fly, without completely retraining the model. Examples of this are implemented in our pre- and post-execution models, which are described in the following sections.

Algorithms must allow us to quickly adapt them to malware writers' counteractions

Outside the malware detection domain, machine learning algorithms regularly work under the assumption of fixed data distribution, which means that it doesn't change with time. When we have a training set that is large enough, we can train the model so that it will effectively reason any new sample in a test set. As time goes on, the model will continue working as expected. After applying machine learning to malware detection, we have to face the fact that our data distribution isn't fixed:

- Active adversaries (malware writers) constantly work on avoiding detections and releasing new versions of malware files that differ significantly from those that have been seen during the training phase.
- Thousands of software companies produce new types of benign executables that are significantly different from previously known types. The data on these types was lacking in the training set, but the model, nevertheless, needs to recognize them as benign.

This causes serious changes in data distribution and raises the problem of detection rate degradation over time in any machine learning implementation. Cybersecurity vendors that implement machine learning in their antimalware solutions face this problem and need to overcome it. The architecture needs to be flexible and has to allow model updates 'on the fly' between retraining. Vendors must also have effective processes for collecting and labeling new samples, enriching training datasets and regularly retraining models.

Detecting new malware in pre-execution with similarity hashing

At the dawn of the antivirus industry, malware detection on computers was based on heuristic features that identified particular malware files by:

- code fragments
- hashes of code fragments or the whole file
- file properties
- and combinations of these features.

The main goal was to create a reliable fingerprint—a combination of features – of a malicious file that could be checked quickly. Earlier, this workflow required the manual creation of detection rules, via the careful selection of a

representative sequence of bytes or other features indicating malware. During the detection, an antiviral engine in a product checked the presence of the malware fingerprint in a file against known malware fingerprints stored in the antivirus database. However, malware writers invented techniques like server-side polymorphism. This resulted in a flow of hundreds of thousands of malicious samples being discovered every day. At the same time, the fingerprints used were sensitive to small changes in files. Minor changes in existing malware took it off the radar. The previous approach quickly became ineffective because:

- Creating detection rules manually couldn't keep up with the emerging flow of malware.
- Checking each file's fingerprint against a library of known malware meant that you couldn't detect new malware until analysts manually create a detection rule.

We were interested in features that were robust against small changes in a file. These features would detect new modifications of malware, but would not require more resources for calculation. Performance and scalability are the key priorities of the first stages of anti-malware engine processing. To address this, we focused on extracting features that could be:

- calculated quickly, like statistics derived from file byte content or code disassembly
- directly retrieved from the structure of the executable, like a file format description.

Using this data, we calculated a specific type of hash functions called locality-sensitive hashes (LSH). Regular cryptographic hashes of two almost identical files differ as much as hashes of two very different files. There is no connection between the similarity of files and their hashes. However, LSHs of almost identical files map to the same binary bucket – their LSHs are very similar – with high probability. LSHs of two different files differ substantially. But we went further. The LSH calculation was unsupervised. It didn't take into account our additional knowledge of each sample being malware or benign. Having a dataset of similar and non-similar objects, we enhanced this approach by introducing a training phase. We implemented a similarity hashing approach. It's similar to LSH, but it's supervised and capable of utilizing information about pairs of similar and non-similar objects. In this case:

- Our training data X would be pairs of file feature representations $[X1, X2]$
- Y would be the label that would tell us whether the objects were actually semantically similar or not.
- During training, the algorithm fits parameters of hash mapping $h(X)$ to maximize the number of pairs from the training set, for which $h(X1)$ and $h(X2)$ are identical for similar objects and different otherwise.

This algorithm that is being

applied to executable file features provides specific similarity hash mapping with useful detection capabilities. In fact, we train several versions of this mapping that differ in their sensitivity to local variations of different sets of features. For example, one version of similarity hash mapping could be more focused on capturing the executable file structure, while paying less attention to the actual content. Another could be more focused on capturing the ASCII-strings of the file. This captures the idea that different subsets of features could be more or less discriminative to different kinds of malware files. For one of them, file content statistics could reveal the presence of an unknown malicious packer. For the others, the most important piece of information regarding potential behavior is concentrated in strings representing used OS API, created file names, accessed URLs or other feature subsets. For more precise detection in products, the results of a similarity hashing algorithm are combined with other machine learning-based detection methods.

CHAPTER 4

4.1 METHODOLOGY

4.1.1 TRAINING MODULE:

Supervised machine learning: It is one of the ways of machine learning where the model is trained by input data and expected output data. To create such model, it is necessary to go through the following phases:

1. model construction
2. model training
3. model testing
4. model evaluation

Model construction: It depends on machine learning algorithms. In this project case, it was neural networks. Such an algorithm looks like:

1. begin with its object: `model = Sequential()`
2. then consist of layers with their types: `model.add(type_of_layer())`
3. after adding a sufficient number of layers the model is compiled. At this moment Keras communicates with TensorFlow for construction of the model. During model compilation it is important to write a loss function and an optimizer algorithm. It looks like: `model.compile(loss= 'name_of_loss_function',`

optimizer= 'name_of_optimizer_alg') The loss function shows the accuracy of each prediction made by the model.

Before model training it is important to scale data for their further use.

4.1.2 SEGMENTATION

Image segmentation is the process of partitioning a digital image into multiple segments(sets of pixels, also known as image objects). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyse. Modern image segmentation techniques are powered by deep learning technology. Here are several deep learning architectures used for segmentation:

Why does Image Segmentation even matter?

If we take an example of Autonomous Vehicles, they need sensory input devices like cameras, radar, and lasers to allow the car to perceive the world around it, creating a digital map. Autonomous driving is not even possible without object detection which itself involves image classification/segmentation.

How Image Segmentation works

Image Segmentation involves converting an image into a collection of regions of pixels that are represented by a mask or a labeled image. By dividing an image into segments, you can process only the important segments of the image instead of processing the entire image. A common technique is to look for abrupt discontinuities in pixel values, which typically indicate edges that define a region. Another common approach is to detect similarities in the regions of an image. Some techniques that follow this approach are region growing, clustering, and thresholding. A variety of other approaches to perform image segmentation have been developed over the years using domain-specific knowledge to effectively solve segmentation problems in specific application areas.

4.2 CLASSIFICATION :CONVOLUTION NEURAL NETWORK

Image classification is the process of taking an input (like a picture) and outputting its class or probability that the input is a particular class. Neural networks are applied in the following steps:

1. One hot encode the data: A one-hot encoding can be applied to the integer representation. This is where the integer encoded variable is removed and a new binary variable is added for each unique integer value.
2. Define the model: A model said in a very simplified form is nothing but a function that is used to take in certain input, perform certain operation to its best on the given input (learning and then predicting/classifying) and produce the suitable output.
3. Compile the model: The optimizer controls the learning rate. We will be using 'adam' as our optimizer. Adam is generally a good optimizer to use for many cases. The adam optimizer adjusts the learning rate throughout training. The learning rate determines how fast the optimal weights for the model are calculated. A smaller learning rate may lead to more accurate weights (up to a certain point), but the time it takes to compute the weights will be longer.
4. Train the model: Training a model simply means learning (determining) good values for all the weights and the bias from labeled examples. In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called empirical risk minimization.
5. Test the model

A convolutional neural network convolves learned features with input data and uses 2D convolution layers.

Convolution Operation:

In purely mathematical terms, convolution is a function derived from two given functions by integration which expresses how the shape of one is modified by the other.

Convolution formula:

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau$$

Here are the three elements that enter into the convolution operation:

- Input image
- Feature detector
- Feature map

Steps to apply convolution layer:

- You place it over the input image beginning from the top-left corner within the borders you see demarcated above, and then you count the number of cells in which the feature detector matches the input image.
- The number of matching cells is then inserted in the top-left cell of the feature map
- You then move the feature detector one cell to the right and do the same thing. This movement is called a stride and since we are moving the feature detector one cell at a time, that would be called a stride of one pixel.
- What you will find in this example is that the feature detector's middle-left cell with the number 1 inside it matches the cell that it is standing over inside the input image. That's the only matching cell, and so you write "1" in the next cell in the feature map, and so on and so forth.
- After you have gone through the whole first row, you can then move it over to the next row and go through the same process.

There are several uses that we gain from deriving a feature map. These are the most important of them: Reducing the size of the input image, and you should know that the larger your strides (the movements across pixels), the smaller your feature map.

Relu Layer:

Rectified linear unit is used to scale the parameters to non negative values. We get pixel values as negative values too. In this layer we make

them as 0's. The purpose of applying the rectifier function is to increase the non-linearity in our images. The reason we want to do that is that images are naturally non-linear. The rectifier serves to break up the linearity even further in order to make up for the linearity that we might impose on an image when we put it through the convolution operation. What the rectifier function does to an image like this is remove all the black elements from it, keeping only those carrying a positive value (the grey and white colors). The essential difference between the non-rectified version of the image and the rectified one is the progression of colors. After we rectify the image, you will find the colors changing more abruptly. The gradual change is no longer there. That indicates that the linearity has been disposed of.

Pooling Layer:

The pooling (POOL) layer reduces the height and width of the input. It helps reduce computation, as well as helps make feature detectors more invariant to its position in the input. This process is what provides the convolutional neural network with the “spatial variance” capability. In addition to that, pooling serves to minimize the size of the images as well as the number of parameters which, in turn, prevents an issue of “overfitting” from coming up. Overfitting in a nutshell is when you create an excessively complex model in order to account for the idiosyncracies we just mentioned. The result of using a pooling layer and creating down sampled or pooled feature maps is a summarized version of the features detected in the input. They are useful as small changes in the location of the feature in the input detected by the convolutional layer will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model's invariance to local translation.

Fully Connected Layer:

The role of the artificial neural network is to take this data and combine the features into a wider variety of attributes that make the convolutional network more capable of classifying images, which is the whole purpose from creating a convolutional neural network. It has neurons linked to each other, and activates if it identifies patterns and sends signals to

output layer .the outputlayer gives output class based on weight values, For now, all you need to know is that the loss function informs us of how accurate our network is, which we then use in optimizing our network in order to increase its effectiveness. That requires certain things to be altered in our network. These include the weights (the blue lines connecting the neurons, which are basically the synapses), and the feature detector since the network often turns out to be looking for the wrong features and has to be reviewed multiple times for the sake of optimization.This full connection process practically works as follows:

- The neuron in the fully-connected layer detects a certain feature; say, a nose.
- It preserves its value.
- It communicates this value to the classes trained images.

4.3 TESTING

The purpose of testing is to discover errors. Testing is a process of trying to discover every conceivable fault or weakness in a work product.It provides a way to check the functionality of components, sub assemblies, assemblies and/or a finished product. It is the process of exercising software with the intent of ensuring that the software system meets its requirements and user expectations and does not fail in an unacceptable manner.

Software testing is an important element of the software quality assurance and represents the ultimate review of specification, design and coding. The increasing feasibility of software as a system and the cost associated with the software failures are motivated forces for well planned through testing.

Testing Objectives:

There are several rules that can serve as testing objectives they are:

- Testing is a process of executing program with the intent of finding an error.
- A good test case is the one that has a high probability of finding an undiscovered error.

Types of Testing:

In order to make sure that the system does not have errors, the different levels of testing strategies that are applied at different phases of software development are :

Unit Testing:

Unit testing is done on individual models as they are completed and becomes executable. It is confined only to the designer's requirements. Unit testing is different from and should be preceded by other techniques, including:

- Inform Debugging
- Code Inspection

Black Box testing

In this strategy some test cases are generated as input conditions that fully execute all functional requirements for the program.

This testing has been used to find error in the following categories: Incorrect or missing functions

- Interface errors
- Errors in data structures are external database access
- Performance error
- Initialisation and termination of errors

- In this testing only the output is checked for correctness
- The logical flow of data is not checked

White Box testing

In this the test cases are generated on the logic of each module by drawing flow graphs of that module and logical decisions are tested on all the cases.

It has been used to generate the test cases in the following cases:

- Guarantee that all independent paths have been executed
- Execute all loops at their boundaries and within their operational bounds.
- Execute internal data structures to ensure their validity.

Integration Testing

Integration testing ensures that software and subsystems work together a whole. It test the interface of all the modules to make sure that the modules behave properly when integrated together. It is typically performed by developers, especially at the lower, module to module level. Testers become involved in higher levels

System Testing

Involves in house testing of the entire system before delivery to the user. The aim is to satisfy the user the system meets all requirements of the client's specifications. It is conducted by the testing organization if a company has one. Test data may range from and generated to production.

Requires test scheduling to plan and organize:

- Inclusion of changes/fixes.
- Test data to use

One common approach is graduated testing: as system testing progresses and (hopefully) fewer and fewer defects are found, the code is frozen for testing for increasingly longer time periods.

Acceptance Testing

It is a pre-delivery testing in which entire system is tested at client's site on real world data to find errors.

User Acceptance Test (UAT)

“Beta testing”: Acceptance testing in the customer environment.

Requirements traceability:

- Match requirements to test cases.
- Every requirement has to be cleared by at least one test case.
- Display in a matrix of requirements vs. test cases.

Model training:

After model construction it is time for model training. In this phase, the model is trained using training data and expected output for this data. It's look this way: `model.fit(training_data, expected_output)`. Progress is visible on the console when the script runs.

At the end it will report the final accuracy of the for training the first level classifier and 200 attacks and corresponding benign versions for the second level training. The remaining 200 pairs were used in testing. MAP only needs one training phase. Thus, 400 attacks and corresponding benign versions were used for training and the remaining 351 pairs are used for testing. We developed a “pintool” for Pin version .

Function Call vs Entire-Program Epoch: Since using the entire program run as an epoch gives reasonable results with MAP's feature sets, it is worth comparing this with the function call epoch. The main problem with using the entireprogram-run epoch is that several applications run for an indeterminate amount of time (e.g. web browser). Summarizing over the entire program run would require the program to finish which, if not making the training phase impossible, would 2017 Design, Automation and Test in Europe (DATE) 173 likely add error as the malicious part could be a small part of the run. In contrast, by summarizing over a function call, the training data is easier to collect and more accurate. For applications that do not execute continuously, summarizing over the entire run is feasible.

RIPE has this characteristic. To evaluate the value of using function call as epoch for it, we built the memory access histograms for the entire program run, and trained them using different classifiers.

[14] to collect memory access patterns. As in the case of the rootkit experiments, we expect data gathering and detection will eventually be performed in hardware. During the testing phase the machine learning algorithm attempts to detect attacks that have not been seen before based on other (somewhat similar) attacks it has seen. Detection using MAP's Feature Sets: Table II shows the feature sets used by MAP.

There are three kinds of features based on: (i) architectural events, (ii) memory addresses, and (iii) the instruction mix. MAP collects data for every 10K instructions (its epoch) to form feature vectors and each feature vector is labeled as malicious/benign based on the program being executed. The detection model is then trained to label these 10K-instruction epochs as malicious/benign.

Model Testing:

During this phase a second set of data is loaded. This data set has never been seen by the model and therefore it's true accuracy will be verified. After the

model training is complete, and it is understood that the model shows the right result, it can.

be saved by: `model.save("name_of_file.h5")`. Finally, the saved model can be used in the real world. The name of this phase is model evaluation. This means that the model can be used to evaluate new data.

Although we envision memory accesses will be collected using specialized hardware, this initial evaluation gathers memory accesses using an instrumented version of QEMU 2.2.0 [2].

QEMU was used to execute a target machine which ran Debian "Squeeze" with Linux kernel v2.6.32. In a rootkit infected system, the behavior of a system utility is changed to meet the purpose of the rootkit, e.g., `ps` may hide malicious processes.

Both benign and malicious traces were collected by running the following system utilities: `ls`, `ps`, `lsmod`, `netstat`. The utilities were executed with different current directories, background processes, arguments, etc. for a total of 50 runs for each rootkit. Both benign and malicious memory traces are summarized using the system call epoch.

A detection model is was trained for each kind affected syscall. `2knark s` and `override` had to be modified to run on our target system. The training set contained the rootkits: `avg coder`, `adore-ng`, `kbeast` and `AFkit` (bold in Table I). We then test the ability of the learned model to distinguish between infected and benign systems on the remaining rootkits. The machine learning algorithm is trained on the 4 rootkits, but is asked to detect 6 rootkits it has never seen before. These experiments demonstrate our framework can detect new malware.

Detection Results: Figure 3 shows the rootkit detection results using our detection framework as a Receiver-Operating Characteristic (ROC) graph. The x-axis shows the false positive rate and the y-axis shows true positive rate. Points on the graph show the achieved detection rate at a certain false positive rate.

The graphs show the classification results for `sys read` and `sys getdents` using different machine learning classifiers with a 4k histogram bin size. Detection

performance is not sensitive to histogram bin size: 1k and 16k bin sizes yield similar results. For both system calls, the best performing classifier (random forest) reaches 100% true positive rate, i.e., detects all attacks, at < 1% false positive rate.

B. Case Study: User Level Memory Corruption Malware In this section, we focus on user-level applications. We provide a direct comparison with Malware-Aware Processors (MAP)[17] that also targets user level programs.

As mentioned in §I, MAP's detection scenario is different from ours. However, their feature sets can still be used in our detection scenario, and provide a good comparison to our feature set of virtual memory access patterns.

Benchmark Suite: We use the RIPE [22] benchmark suite for our experiments. RIPE is a synthetic benchmark which contains a total of 850 different memory corruption attacks in various forms including “modern” attacks such as return-to-libc attacks, return-oriented-programming, etc.

This suite was executed on a Linux system running Ubuntu 6.06 distribution with kernel version 2.6.15. We also created a “benign” version of RIPE where each of the attack targets is patched.

Methodology: Among the 850 RIPE attacks, 751 were successful on our target system. For our two-level classification mode, we used 351 attacks and corresponding benign versions

The program is developed using python and Machine learning algorithms. A stereogram embeds a word in the image pixels. There are numerous ways of embedding the word in the image pixels. Among them are: Substitution method, distortion method. We are going to use the substitution method using a machine learning algorithm. Each image is stored as a significant bit. It is also converted into binary form.

Our first step is to convert the normal image into a binary representation. If some kind of stego content is available we can easily identify using the machine learning algorithm. After this step, if it is not passed this test, it would undergo the next step of the process which extracting to form the hex code

, RGB values of pixels in the cover picture is converted into its corresponding octal values. As a cover for the payload, different image formats were used and analytic values were calculated for the red, green, and blue channels of the input image and the resulting image, as well as the difference among the average pixels for both images.

Our algorithm next converts the images to grayscale for creating a separate channel for each image. The stego content image is encrypted with different types of algorithms. So, the message cannot be decoded but we can identify the malware content. Many hazardous contents can be sent with the image file and can hack anyone's personal system. The algorithm identifies the individual channel of the RGB layer and identifies the spyware.

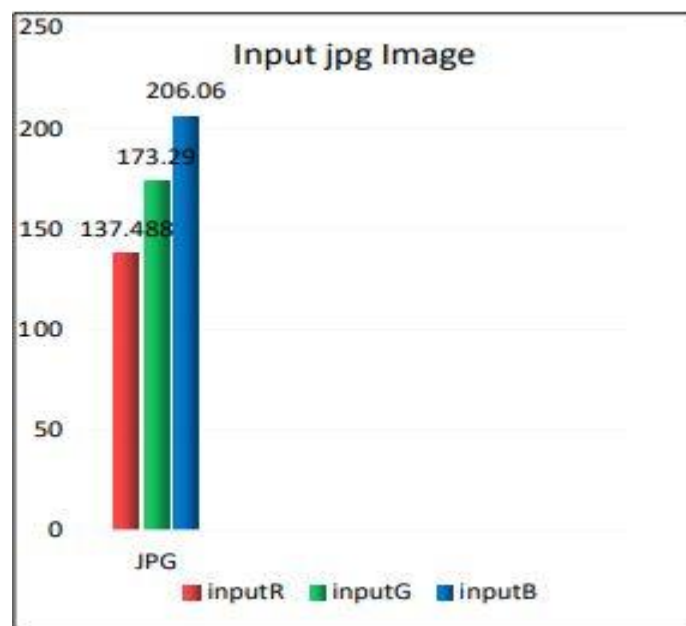


Fig4.1. Input JPG image

In Fig. 1, the picture is represented by the red, green, and blue channels of the image. The value of the red channel is 137.488, the value of the green channel is 173.29, and the value of the blue channel is 206.06.

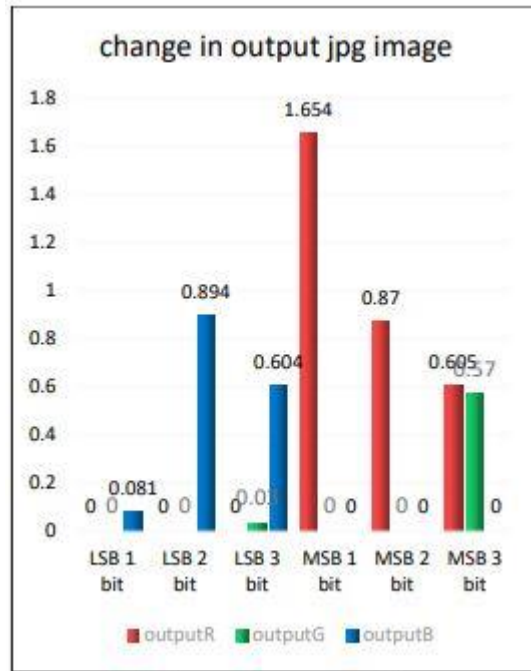


Fig4.2. Output image

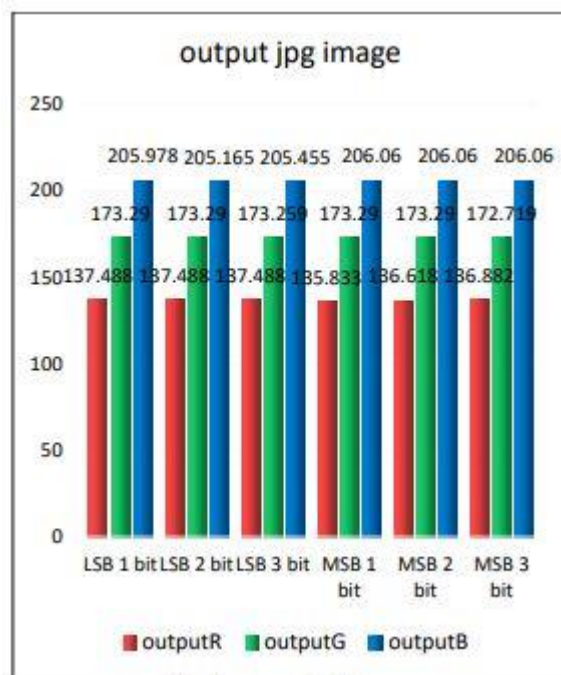


Fig4.3.Change in Output image

Display the changes in the image due to hiding messages, the amount of change depends on the size of the bits (1, 2, or 3) used and the technique (LSB or MSB)

used. A difference of 1.654% is shown using only 1-bit substitution on the red channel, and a difference of 0.605 and 0.57 on the two channels (red and green).

Therefore, if LSB is used with a one-bit substitution, the least amount will appear on only one channel (blue). With the significant change in the bits of the image, we can able find the stego content occupied in the image.

This algorithm used a classified method that is based on logistic classification and linear classification. This system allows separating the files which contain steno content. This algorithm is based on the steganalysis method.

The substitution method is made the problem easier to solve. The change in the least significant bit in the image can help us to intimate that the file contains more stego content. This RGB layer identification method can be classified into the following steps shows in figure 5. (More clarity definition about fig 4 and fig 5

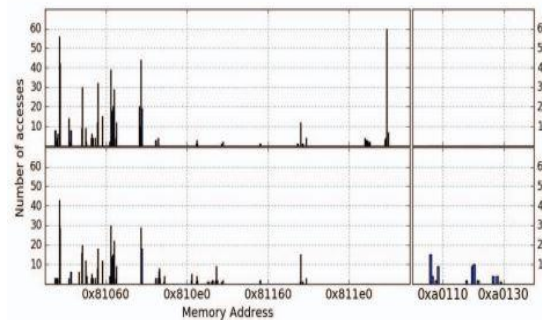


Fig 4.4: Malware Detection simulation

Pseudocode for this process

Step 1: For i 1 to k

Step 2: for each training data instance d_i :

Step 3: Set the target value for the regression to

$$Y_j - P(1|d_j)$$

$$[p(1|d_j) \cdot (1 - P(d_j))]$$

Step 4: initialize the weight of instance d_j to $P(1|d_j)$.

(1-P) (1|

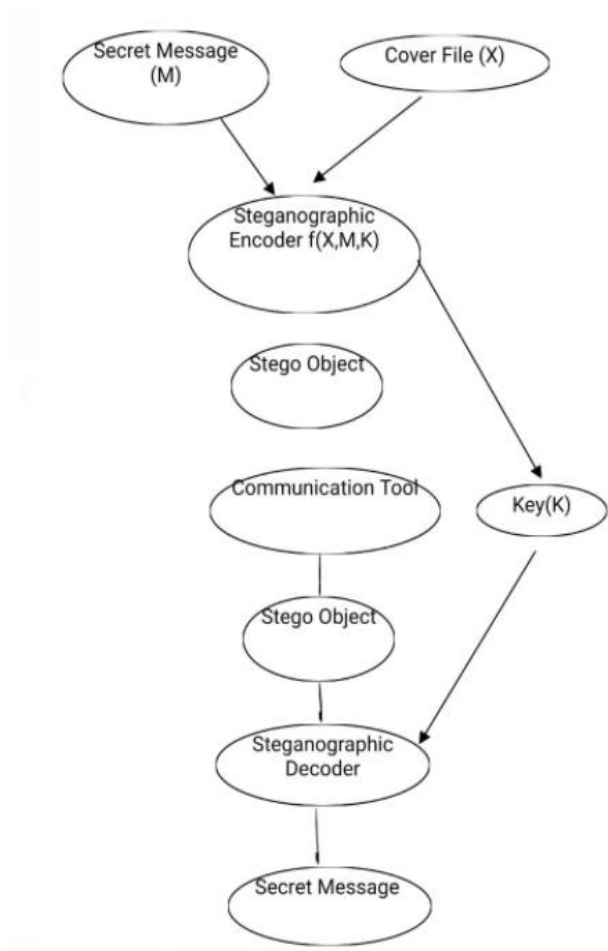


Fig 4.5: RGB Layer Identification Step

Three commonly used classifiers: logistic regression, SVM and random forest are used by our work. We consider two design points for the classifiers. Direct Classification of Memory Access Histograms: This performs binary classification directly on the summary histograms computed in each epoch. Empirically, this was effective in detecting kernel-level malware. During the training phase, each system call affected by a rootkit is labelled as malicious, while other system calls are labelled benign.¹ Since rootkits corrupt syscall execution, the classifier then learns to distinguish between benign and malicious syscalls.

Weighted Classification of Memory Access Histograms: For user-level malware the epoch boundaries correspond to function calls, which unlike system calls are not limited in number and are different across programs. Therefore, detecting malware by analyzing the histogram of a single function is hard.

Our solution is to classify the different functions in a program separately and consider a weighted sum of classification results. Each execution is labeled as either malicious or benign and in the first phase classifiers are trained to recognize these labels. These labels are then assigned to every function in the execution, which may result in a function that was not infected being labeled as malicious. It is precisely this noncorrelation that is addressed in the second training phase.

Using the MAP Features in Our Detection Scenario: MAP provides us with potential features for malware detection. However, their framework is not applicable to our detection scenario as it is designed to label different programs while ours detects whether one particular application is infected by malware. In most malware infected program runs, the malicious behavior only occurs at certain phases of the execution, the other phases are normal program behavior.

This makes it hard to label the 10K-instruction epochs correctly without manual intervention. If we label all the epochs in a malware infected run as malicious, most epochs that reflect normal program behavior would be wrongly labeled, resulting in huge training error. Therefore, instead of using the 10K-instruction epoch, we use the entire program run as the epoch. Although it is hard to correctly label the 10K-instruction epochs, any malicious behavior is reflected in the feature vectors of entire malicious runs, so labels can be correctly applied. These feature vectors are collected over the entire run and the classifier is trained to label the entire run. Data collection for the MAP features was also done using our “pin tool”.

The second training phase uses a different training dataset. Assume we have n models for the n function calls and each 1 In this scenario we need some human input to identify the system calls affected by each rootkit used in training. This is relatively easy compared to the manual analysis required for the techniques in [19], [15], [12]. model m_1 to m_n provides its classification result. An accuracy rate

is assigned to each model. We discard weak models with accuracy below 60% as they have poor correlation between the run being malicious and the function being malicious.

Assume that k models, M_1 to M_k are left, and they have accuracy rate r_1 to r_k . We assign a weight: $w_i = r_i - 0.5$ ($r_j - 0.5$) to each model. Assume the classification result from each M_i is c_i (0 for benign, 1 for malicious, if f_i is called multiple times, c_i is the average value of all the classification results). The classifier result is defined as the weighted sum of each model $C = \sum c_i w_i$. If C is above a threshold T , decided in the second phase, it is classified as malicious and otherwise it is classified as benign.

This classification emphasizes functions whose infection correlates strongly with infected executions. It is important to note that this is done by the algorithm without any expert human analysis.

Let the standard for good detection be $> 95\%$ true positive rate with false positives $< 5\%$. Two feature sets/detection methods meet these criteria. Ours using two-level classification of memory accesses in the function call epoch not only meets the standard but also has the highest accuracy under any false positive rate. Among the MAP feature sets, only INS2 with the logistic regression classifier is slightly above the standard.

Table III shows the comparison between our method ("FUNC") and "COMB", "INS2" and "MEM1" which are the most successful among the MAP features. We see that for each false positive rate (FP), our method has best true positive rate (TP). Also, as the allowed false positive rate decreases, our method maintains a good detection rate while others drop quickly. These results show the strength of our method, especially in the very low false positive rate regime.

The results are presented using ROC graphs. From the graphs, we see that when using small histogram bin sizes, the two epochs perform similarly. But when the bin size is larger, the results of the function epoch stay mostly the same while the detection rate using the entire program epoch deteriorates.

Thus, the function call epoch is resilient to changes in histogram bin size, while with the entire-program epoch, the histogram bin size needs to be chosen carefully. This may need human input and may possibly differ between applications.

Summary Histograms for Epochs Offline Histogram Storage Train Classifier
Trained Model Program (a) Training exec Verify Binary Signature Load Model to
HW Classifier HW Execution Monitor Authenticated Handler Malware detected (b)
Operation Example Epoch Fig. 2:

Overview of the Complete Framework Figure 2 shows the complete framework. The upper half shows training and the lower half monitoring/detection. Data Collection and Training: During training, the program is executed and the summary histograms computed and stored for offline analysis.

Once the data is collected, each histogram is labeled either “benign” or “malicious” and a classifier is trained to learn these labels. This classifier model is distributed along with the executable for the application/operating system.

Monitoring and Malware Detection: Our application-specific malware detection aims to ensure the integrity of commonly used applications (e.g. web browser, email) and the system kernel. There are only a few frequently used applications and kernel system calls, so the model storage space is not large. The classification model for each application is retrieved from the executable and used to program the hardware classifier at load time.

Runtime authentication of the malware model and binary is necessary to protect it from malware. When the classifier detects potential malware, it raises an exception that is processed by an authenticated software handler which executes in a hardware-enforced sandbox and performs detailed analysis of the executing program.

Address Space Layout Randomization (ASLR) introduces noise in the virtual memory traces because it shifts the code and data segments by adding a

randomized offset to their initial addresses. Since this offset is known to the operating system, ASLR can be “de-randomized”.

Granularity of Memory Block Size: An important design parameter is the size of each bin in the summary histograms. There is a trade-off here: smaller bins provide more detailed memory access information than larger bins, but require larger net storage. However, from a machine learning perspective, histograms with smaller bins may not always make a better feature vector. Larger bins may be more preferable because the classifier may avoid “confusion” due to unimportant smallscale variations. The experimental evaluation considers bin sizes of 1KB, 4KB and 16KB.

CHAPTER 5

5.1 RESULT

In this concept design study, we suggested a technique for creating a classification model for precedent suicidal ideation among adolescents using natural language processing and machine learning of clinical narratives from health record data prior to admission. This is the first time that unstructured data has been used to up skill a machine-learning classification system in a teenage inpatient population using NLP as far as we know.

. This framework was applied to the application-specific malware detection scenario which targets detecting malware infected runs of known applications. We addressed the challenge of online memory data collection using a system/function-call epoch-based memory access summary.

We experimentally covered both kernel and user level threats and demonstrated very high detection accuracy against kernel level rootkits (100% detection rate with less than 1% false positives) and user level memory corruption attacks (99.0% detection rate with less than 5% false positives).

A key value of the proposed methodology is using machine learning to determine malware signatures for classification in contrast to the traditional reliance on human insight – a major step in automating this critical analysis problem.

This first success signal based on a limited sample size needs to be confirmed in bigger datasets, The method demonstrates how EHR notes for adolescent suicide attempts can be enhanced with clinically relevant information to identify children with a history of suicide risk, This can aid in patient's health organizing during a

highly vulnerable period for this high-risk demographic

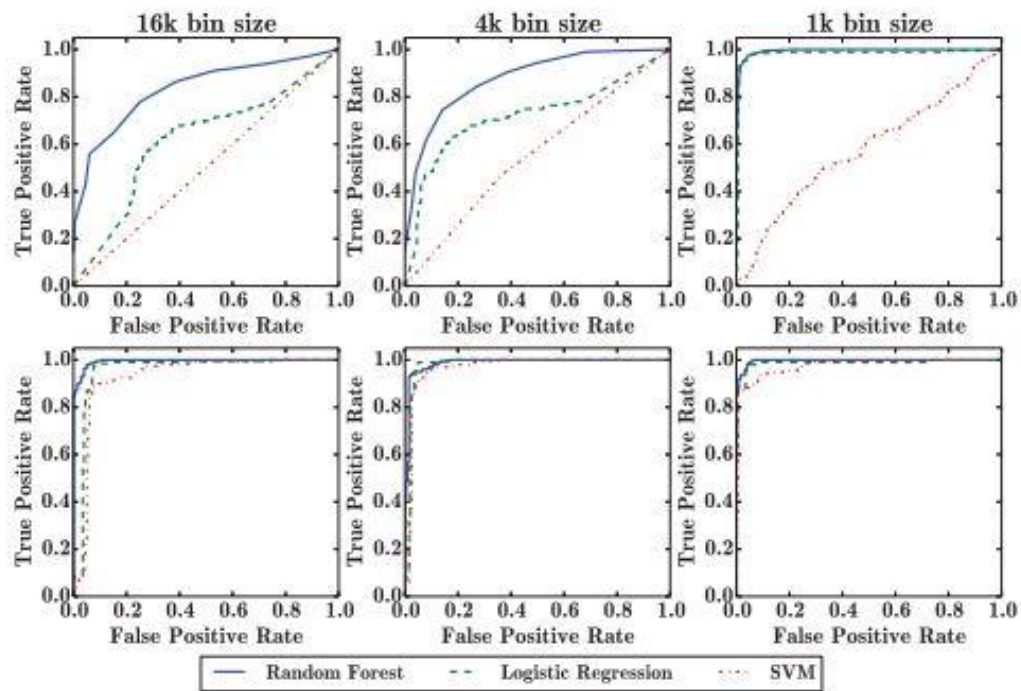


Fig5.1.LSB Graph

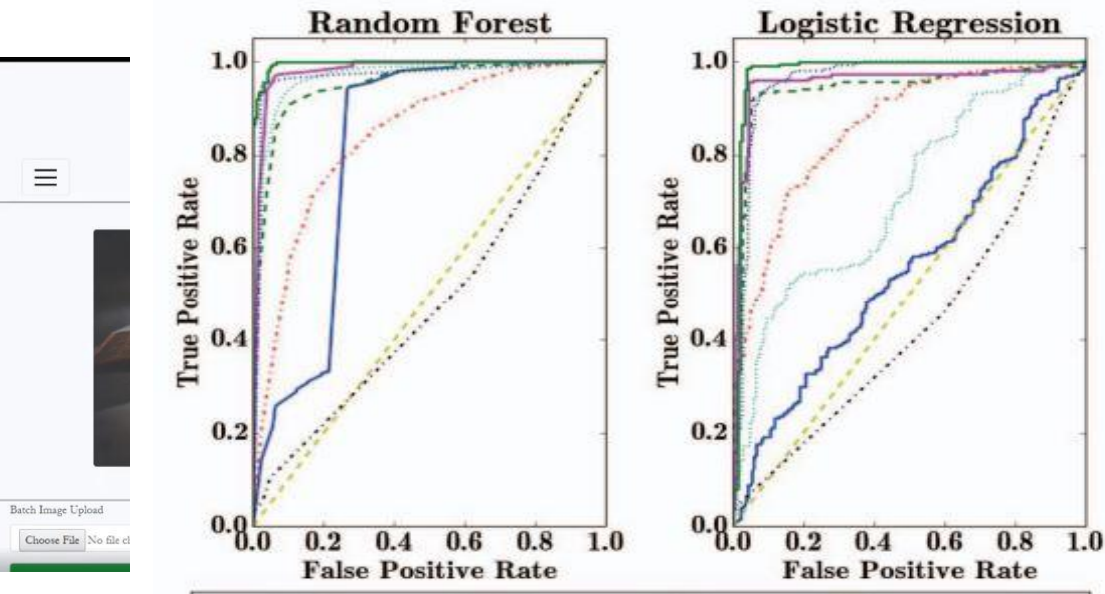


Fig5.3. Output image

Fig5.2. False rate Graph

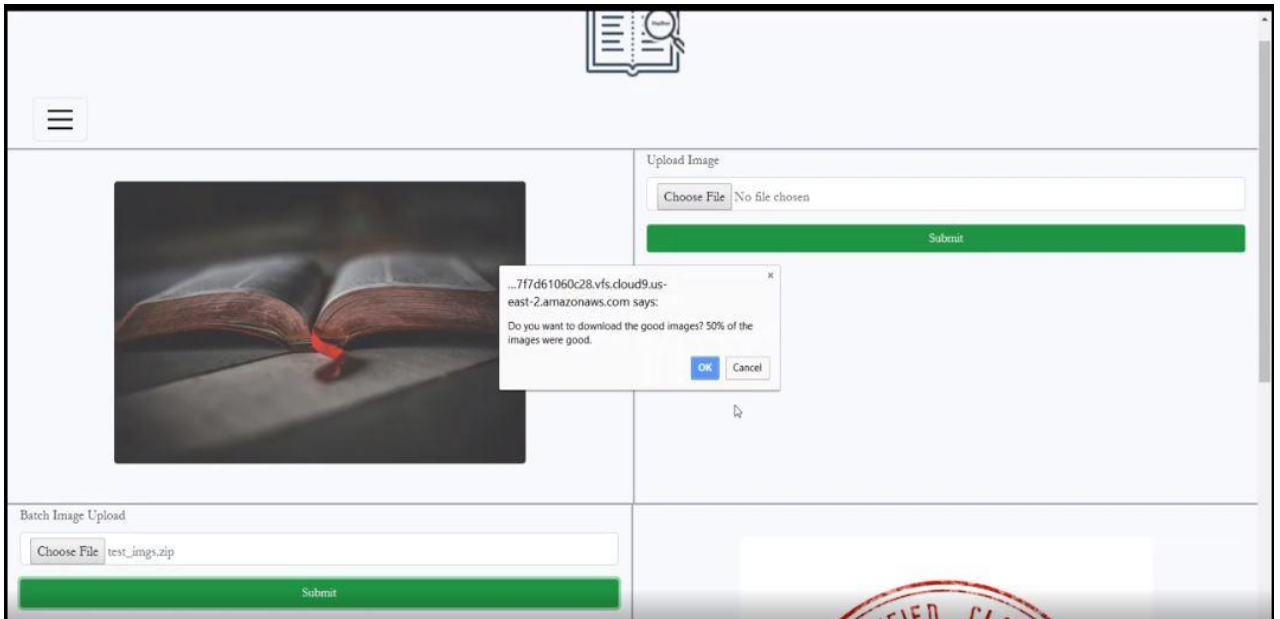


Fig5.4. Output image

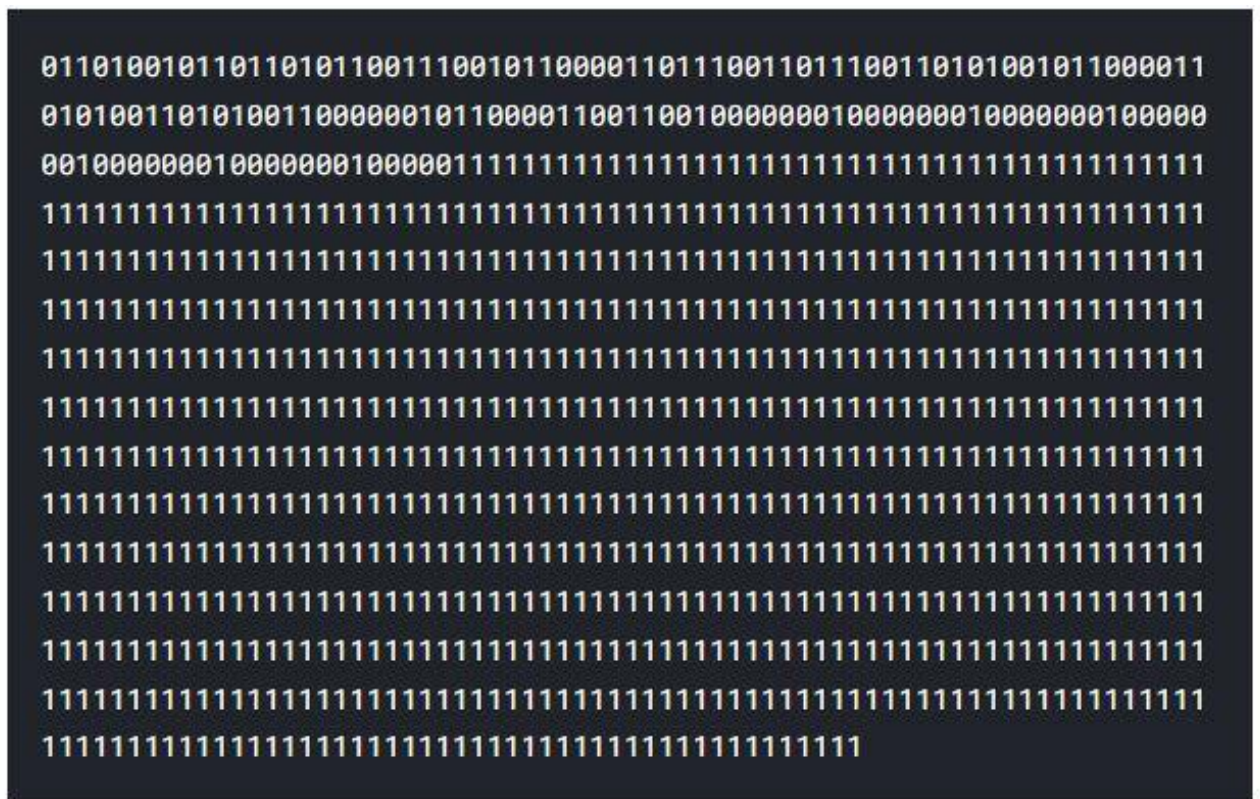


Fig5.5. Binary code image

5.2 PERFORMANCE ANALYSIS

Performance analysis			
Algorithms	Functions	Limitation	Performance analysis
Linguistic Steganography Detection Algorithm Using Statistical Language Model	Statistical Language Model	When the text size is small it cannot detect the malware content. It can only detect the text.	93.9% when the text size is 2KB
Detection of Steganography Inserted by OutGuess and means of Neural Networks	Neural Networks Huffman's coding	The response time is longer	Accuracy is very low
Detection of LSB Steganography Based on Distribution of Pixel Differences in Natural Images	Statistical Distribution of Pixel Differences	It can detect only using gray scale images	It can detect gray scale with high accuracy but not stego content.
An analysis of LSB Based Image Steganography Technique	LSB Replacement algorithm	It cannot include more sized images.	Not mentioned
Detection of LSB Replacement and LSB Matching Steganography Using Gray Level Run Length Matrix	LSB steganography methods	It can only identify by converting gray scale images	82% stego accuracy
Analysis of Image Steganography Techniques for Different Image Format	Substitution method Distortion method	Cannot identify the stego content	95 % accuracy for stego contents

CHAPTER 6

6.1 FUTURE SCOPE

To curb the menace of terrorism and to destroy the online presence of dangerous terrorist organizations like ISIS and other radicalization websites. We need a proper system to detect and terminate websites which are spreading harmful content used to radicalizing youth and helpless people. We analysed the usage of Online Social Networks (OSNs) in the event of a terrorist attack. We used different metrics like number of tweets, whether users in developing countries tended to tweet, re-tweet or reply, demographics, geo location and we defined new metrics (reach and impression of the tweet) and presented their models. While the developing countries are faced by many limitations in using OSNs such as unreliable power and poor Internet connection, still the study finding challenges the traditional media of reporting during disasters like terrorist's attacks. We recommend centers globally to make full use of the OSNs for crisis communication in order to save more lives during such

6.2 CONCLUSION

We concluded this paper the ways of negotiating for a data steganography system

Error! Reference sources not found and present the results obtained from them in the form of bar graphs. These graphs are collated according to the results produced from colorful methods.

Based on our evaluation, we have made it evident that to keep the quantum performing picture, we must use JPG format as the cover for the container, and that we must cover only one last substantial piece since in our trail showed the difference of 0.081 among its average value and the cover. There is no significant difference between LSB and MS for other picture format combinations and the number of pieces to cover.

In this paper, we presented a framework for detecting malware that uses online analysis of steganalysis with machine learning to identify Steno contents with a precision of 98 per cent. Our framework has been applied to detecting infected malware from well-known applications via a case study that focuses on detecting application-specific malicious software.

The purpose of this study is to gather online memory data with the aid of a summary of memory accesses based on the system call epoch. Both kernel and user-level threats were covered in experiments that shows high detection accuracy against stego contents (100% detection rate with less than 1% false positives) as well as memory steganalysis attacks (99.0% detection rate with less than 4 percentage false positives). As part of the proposed methodology, machine learning is used to identify malware signatures for classification rather than relying on human handy works - a crucial step to automate this analytical problem.

REFERENCES

- [1] Idika, N., & Mathur, A. P. (2007). A survey of malware detection techniques. *Purdue University*, 48
- (2). Bryant, R. E. (2005, May). Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)* (pp. 32-46). IEEE.
- [3] Moser, A., Kruegel, C., & Kirda, E. (2007, December). Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)* (pp. 421-430). IEEE.
- [4] Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011, October). Crowddroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (pp. 15-26).
- [5] Ye, Y., Wang, D., Li, T., & Ye, D. (2007, August). IMDS: Intelligent malware detection system. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 1043-1047).
- [6] Vinod, P., Jaipur, R., Laxmi, V., & Gaur, M. (2009, March). Survey on malware detection methods. In *Proceedings of the 3rd Hackers' Workshop on computer*

and internet security (IITKHACK'09) (pp. 74-79).

[7] Sahs, J., & Khan, L. (2012, August). A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference* (pp. 141-147). IEEE.

[8] McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., ... & Joon Ahn, G. (2017, March). Deep android malware detection. In *Proceedings of the seventh ACM on conference on data and application security and privacy* (pp. 301-308).

[9] Ye, Y., Li, T., Adjeroh, D., & Iyengar, S. S. (2017). A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*, 50(3), 1-40.

[10] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X. Y., & Wang, X. (2009, August). Effective and efficient malware detection at the end host. In *USENIX security symposium* (Vol. 4, No. 1, pp. 351-366).

[11] Yuan, Z., Lu, Y., Wang, Z., & Xue, Y. (2014, August). Droid-sec: deep learning in android malware detection. In *Proceedings of the 2014 ACM conference on SIGCOMM* (pp. 371-372).

[12] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., & Weiss, Y. (2012). "Andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1), 161-190.

[13] Preda, M. D., Christodorescu, M., Jha, S., & Debray, S. (2007). A semantics-based approach to malware detection. *ACM SIGPLAN Notices*, 42(1), 377-388.

[14] Bazrafshan, Z., Hashemi, H., Fard, S. M. H., & Hamzeh, A. (2013, May). A survey on heuristic malware detection techniques. In *The 5th Conference on Information and Knowledge Technology* (pp. 113-120). IEEE.

[15] Zarni Aung, W. Z. (2013). Permission-based android malware detection. *International Journal of Scientific & Technology Research*, 2(3), 228-234.

7.APPENDIX

a) SAMPLE CODE

```
import subprocess
import sys

subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'requests'])
subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'bs4'])
subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'ttkthemes'])
subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'lxml'])
subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'tkinter'])
subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'PIL'])
import model
import preprocess as pp

shouldPreprocess = False

if __name__ == '__main__':
    if shouldPreprocess:
        data = pp.getDataset('RedditDataSentimental.xlsx')
        data = pp.cleanDataset(data, True, True)

    # Get preprocessed dataset
    data = pp.getDataset('dataset.csv')
    data = model.minSubSample(data, False)

    # Indication for Logistic Regression model
```

```

if False:
    model.validateModel(data) # not executed except for testing due to time
                                required to evaluate

    # Parameter tweaking to trade accuracy for greater positive recall / sensitivity
    if False:
        model.gridSearchParameters(data) # not executed except for testing due to
        time required to evaluate

    # Final training
    model.trainFinalModel(data)
import math

from sklearn.metrics import accuracy_score
from sklearn.svm import LinearSVC

import pipeline as pl
import preprocess as pp
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from pprint import pprint
from sklearn.utils import shuffle
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn import metrics

def minSubSample(data, shouldShuffle=False):
    """
    minSubSample - prepares DataFrame by taking ?shuffled subsample from data, based on
    minimum size of groups, to balance classes

    :param data: pandas DataFrame
    :param shouldLemmatise: boolean, defaults False - determines whether groups are
    shuffled
    :return: returns DataFrame
    """
    # Get smallest sample group index
    minIndex = np.argmin(data['Suicidal'].value_counts())

    # Separate into distinct groups
    suicidalGroup = data[data['Suicidal'] == 1]
    notSuicidal = data[data['Suicidal'] == 0]
    suicidalGroup = suicidalGroup.reset_index(drop=True)
    notSuicidal = notSuicidal.reset_index(drop=True)

```

```

# Shuffle if required
if shouldShuffle:
    suicidalGroup = shuffle(suicidalGroup)
    notSuicidal = shuffle(notSuicidal)

# Sample groups by smallest group size
numSamples = len([notSuicidal, suicidalGroup][minIndex])
notSuicidal = notSuicidal[:numSamples]
suicidalGroup = suicidalGroup[:numSamples]

# Merge groups of data and shuffle
data = shuffle(pd.concat([suicidalGroup, notSuicidal])).reset_index(drop=True)

return data

def validateModel(data):
    """
    validateModel - tests validity and prints accuracy score of Logistic Regression model

    :param data: pandas DataFrame
    :return: None
    """
    # Tfidf transformation
    tfidfVect = TfidfVectorizer()
    trainContentTfidf = tfidfVect.fit_transform(data['Sentence'])

    # Fit model
    skf = StratifiedKFold(n_splits=5, shuffle=True)
    skf.get_n_splits(trainContentTfidf, data['Suicidal'])

    LR = LogisticRegression()
    scores = []
    for train_index, test_index in skf.split(trainContentTfidf, data['Suicidal']):
        x_train, x_test = trainContentTfidf[train_index], trainContentTfidf[test_index]
        y_train, y_test = data.iloc[train_index, 1], data.iloc[test_index, 1]

        LR.fit(x_train, y_train)

        y_pred = LR.predict(x_test)
        scores.append(round(metrics.accuracy_score(y_test, y_pred) * 100, 2))
        # print(metrics.classification_report(y_test, y_pred, target_names=['Not Suicidal',
        'Suicidal']))

    # Print scores
    scores = [{'Split': i, 'Accuracy': x} for i, x in enumerate(scores)]
    print('____ LR indication ____\n')
    pprint(scores)
    print('_____')

def evaluateModel(pipeline, train_data, train_labels):

```

```

"""
evaluateModel - evaluate pipeline model from pl

:param pipeline: pipeline referring to MODEL_PIPELINES index
:param train_data: x_train data
:param train_labels: y_train data
:return: return GridSearchCV dataframe
"""

# Get the current pipeline and its parameters
curPipeline = pl.MODEL_PIPELINES[pipeline]['Pipeline']
curPipelineParams = pl.MODEL_PIPELINES[pipeline]['Parameters']

# Grid search cv to evaluate the models for scorers
gs_eval = GridSearchCV(
    curPipeline,
    curPipelineParams,
    scoring=pl.MODEL_SCORERS,
    refit=pl.REFIT_SCORE,
    error_score='raise',
    cv=StratifiedKFold(n_splits=10, random_state=None),
    n_jobs=-1
)
gs_eval = gs_eval.fit(train_data, train_labels)

print(f'Best params for {pl.REFIT_SCORE}')
print(gs_eval.best_params_)

return gs_eval

def evaluateModels(train_data, train_labels):
    """
    evaluateModels - evaluate all models and parameters within pl.MODEL_PIPELINES

    :param train_data: x_train data
    :param train_labels: y_train data
    :return: array (names of pipelines), pd DataFrame (cv_results_), array (best_params_)
    """
    names = [ ]; results = [ ]; params = [ ]
    for curPipeline in pl.MODEL_PIPELINES:
        gs = evaluateModel(curPipeline, train_data, train_labels)
        names.append(curPipeline)
        results.append(pd.DataFrame(gs.cv_results_))
        params.append(gs.best_params_)

    return names, results, params

def gridSearchParameters(data):
    """
    gridSearchParameters - hyper parameter tweaking as per pl model pipelines to trade accuracy for higher sensitivity

```

```

:param data: pandas DataFrame
:return: None
"""

names, results, params = evaluateModels(data.iloc[:, 0], data.iloc[:, 1])

for i, x in enumerate(names):
    print("Test -->' + x)
    print("Results -->")
    print(results[i].head())

def trainFinalModel(data):
    """
    trainFinalModel - final training model with best parameters

    :param data: pandas DataFrame
    :return: None
    """
    # Transform
    tfidfVect = TfidfVectorizer(ngram_range=(1, 3), norm='l1')
    trainContentTfidf = tfidfVect.fit_transform(data['Sentence'])

    x_train, x_test, y_train, y_test = train_test_split(trainContentTfidf, data.iloc[:,1],
test_size=0.2)

    # Fit model
    LR = LogisticRegression(C=4, max_iter=1000, penalty='l2')
    LR.fit(x_train, y_train)

    # Predict
    y_pred = LR.predict(x_test)
    print(f' Accuracy of LR : {round(metrics.accuracy_score(y_test, y_pred) * 100, 2)}%')
    print(metrics.classification_report(y_test, y_pred))
    LogisticRegression_score = metrics.accuracy_score(y_test, y_pred)

    # Visualise model prediction vs truth
    sns.heatmap(metrics.confusion_matrix(y_test, y_pred), annot=True,
fmt='d', cmap='YlGnBu')
    plt.title('Confusion matrix of logistic regression', y=1.1)
    plt.xlabel('Predicted')
    plt.ylabel('Truth')
    plt.show()

    # Fit model
    classifier = LinearSVC(C=4, max_iter=1000, penalty='l2')
    classifier.fit(x_train, y_train)

    # Predict
    y_prediction = classifier.predict(x_test)
    print(f' Accuracy Score of SVC: {round(metrics.accuracy_score(y_test, y_prediction) *

```

```

100, 2})}')
print(metrics.classification_report(y_test, y_prediction))
ACC1=metrics.accuracy_score(y_test, y_prediction)
# Visualise model prediction vs truth
sns.heatmap(metrics.confusion_matrix(y_test, y_prediction), annot=True, fmt='d',
cmap='YlGnBu')
plt.xlabel('Predicted')
plt.ylabel('Truth')
plt.title('Confusion matrix of SVC', y=1.1)
plt.show()

labels = ["SVC", "LogisticRegression"]
usages = [ACC1, LogisticRegression_score]
y_positions = range(len(labels))
plt.bar(y_positions, usages)
plt.xticks(y_positions, labels)
plt.ylabel("Accuracy")
plt.title("best model selection")
plt.show()
from textthero import preprocessing as pp
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import make_scorer, recall_score, accuracy_score, precision_score

# File directory for training input
INPUT_DIRECTORY = './input/'

# Preprocessing pipeline for cleaning text
PREPROCESSING_PIPELINE = [
    pp.remove_html_tags,
    pp.remove_urls,
    pp.lowercase,
    pp.remove_digits,
    pp.remove_punctuation,
    pp.remove_diacritics,
    pp.remove_stopwords,
    pp.remove_whitespace,
    pp.remove_brackets
]

# Model scorers
REFIT_SCORE = 'recall_score'
MODEL_SCORERS = {
    'precision_score': make_scorer(precision_score),
    'recall_score': make_scorer(recall_score),
    'accuracy_score': make_scorer(accuracy_score)
}

```

```

# GridSearchCV pipelines
MODEL_PIPELINES = {
    'fe=tfidf+bigram+default_c=LR' : {
        'Pipeline' : Pipeline([
            ('tfidf', TfidfVectorizer()),
            ('clf', LogisticRegression())
        ]),
        'Parameters' : {
            'tfidf__norm' : ['l1', 'l2'],
            'tfidf__ngram_range': [(1, 2)],
            'clf__penalty': ['l2'],
            'clf__max_iter': [1000],
            'clf__C' : [0.25, 0.5, 1, 2, 4]
        }
    },
    'fe=tfidf+trigram+default_c=LR' : {
        'Pipeline' : Pipeline([
            ('tfidf', TfidfVectorizer()),
            ('clf', LogisticRegression())
        ]),
        'Parameters' : {
            'tfidf__norm' : ['l1', 'l2'],
            'tfidf__ngram_range': [(1, 3)],
            'clf__penalty': ['l2'],
            'clf__max_iter': [1000],
            'clf__C' : [0.25, 0.5, 1, 2, 4]
        }
    }
}

import os
import emoji
import nltk
import pipeline as pl
import texthero as hero
import pandas as pd
import regex as re
from nltk.stem import WordNetLemmatizer

# Utilities
def getDataset(filename):
    """
    getDataset - gets and reads the specified file from the input directory,
    drops null rows and columns excluded from our analysis.

    :param filename: in format of {filename.extension}
    :return: returns pandas DataFrame
    """
    # Read file
    filename = pl.INPUT_DIRECTORY + filename
    split = os.path.splitext(filename)

```

```

data = None
if 'csv' in split[1]:
    data = pd.read_csv(filename)
elif 'xlsx' in split[1]:
    data = pd.read_excel(filename)

# Select only Suicidal and Sentence columns
data = data[['Sentence', 'Suicidal']]

# Remove null rows
data = data.dropna()

return data

def removeHandles(text):
    """
    removeHandles - removes @alphanumeric handles from text

    :param text: string
    :return: returns cleansed string
    """
    text = ''.join(re.sub('@\w+', '', text).split())
    return text

def removeHashtag(text):
    """
    removeHashtag - removes #hashtag from text

    :param text: string
    :return: returns cleansed string
    """
    text = ''.join(re.sub('\#[\w\_]+', '', text).split())
    return text

def removeEmoji(text):
    """
    removeEmoji - removes all instances of emoji in the string passed.

    :param text: string
    :return: returns cleansed string
    """
    text = emoji.get_emoji_regexp().sub('', text)
    return text

def cleanSentences(text, lemmatise=True):
    """
    cleanSentences - pre-processes text for analysis, optional lemmatisation (defaults to
    True)

    :param text: array of strings

```

```

:param lemmatise: boolean, defaults True - determines whether function performs
lemmatisation
:return: returns array of cleaned strings
"""
# Remove emoji
text = text.apply(lambda x: ' '.join([removeEmoji(word) for word in x.split()]))

# Remove hashtag
text = text.apply(lambda x: ' '.join([removeHashtag(word) for word in x.split()]))

# Remove handles
text = text.apply(lambda x: ' '.join([removeHandles(word) for word in x.split()]))

# Clean text as per the pipeline
text = hero.clean(text, pipeline=pl.PREPROCESSING_PIPELINE)

# Lemmatisation
if lemmatise:
    nltk.download('wordnet')
    text = text.apply(lambda x: ' '.join([WordNetLemmatizer().lemmatize(word) for word in
x.split()]))

# Remove words with < 2 len and/or aren't alphabetical
text = text.apply(lambda x: ' '.join([word for word in x.split() if len(word) > 2 and
word.isalpha()]))

return text

def cleanDataset(data, shouldLemmatise=True, shouldSave=False):
    """
    cleanDataset - prepares DataFrame for analysis

    :param data: pandas DataFrame
    :param shouldLemmatise: boolean, defaults to True - passed to cleanSentences if
lemmatisation required
    :param shouldSave: boolean, defaults to False - determines whether we save the cleaned
DF
                                as a csv to ./inputs/dataset.csv
    :return: returns cleansed DataFrame
    """
    # Replace yes/no string datapoint as binary representation
    data.loc[data['Suicidal'] == 'Yes', 'Suicidal'] = 1
    data.loc[data['Suicidal'] == 'No', 'Suicidal'] = 0

    # Preprocess string data
    data['Sentence'] = cleanSentences(data['Sentence'], shouldLemmatise)

    # Save input if required
    if shouldSave:
        data.to_csv(pl.INPUT_DIRECTORY + 'dataset.csv')

```

```

    return data
import json # to print list/dict in textbox
import tkinter as tk # root GUI module
import tkinter.scrolledtext as scrolledtext # module for scrollable text widget
import tkinter.ttk as ttk # themed GUI module
from tkinter.filedialog import askopenfile # module to read file

import requests # module to get all contents of a website
from bs4 import BeautifulSoup # module to get only text from a website
from PIL import Image, ImageTk # module to open and load a image
from ttkthemes import ThemedStyle # module to use in-built GUI themes

# class to get all frames together
class MyApp(tk.Tk):

    def __init__(self, *args, **kwargs, ):
        tk.Tk.__init__(self, *args, **kwargs)
        container = tk.Frame(self)
        container.pack(side="top", fill="both", expand=True)

        container.grid_rowconfigure(0, weight=1)
        container.grid_columnconfigure(0, weight=1)

        self.frames = {}

        menu = tk.Menu(container)

        ex = tk.Menu(menu, tearoff=0)
        menu.add_cascade(menu=ex, label="Exit")
        ex.add_command(label="Exit",
                        command=self.destroy)

        tk.Tk.config(self, menu=menu)

        for F in (Startpage, PageOne, PageTwo):
            frame = F(container, self)
            self.frames[F] = frame
            frame.grid(row=0, column=0, sticky="nsew")

        self.show_frame(Startpage)

    def show_frame(self, cont):
        frame = self.frames[cont]
        frame.tkraise()

# Home page
class Startpage(ttk.Frame):

```

```

def __init__(self, parent, controller):
    ttk.Frame.__init__(self, parent)

    label = ttk.Label(self, text="Detection of suicidal content", font=("Simplifica", 22)) #
page heading
    label.pack(pady=5, padx=5)

    ttk.Label(self, text="").pack()

    button1 = ttk.Button(self, text="Detect",
                        command=lambda: controller.show_frame(PageOne)) # got to detect
page
    button1.pack()

    ttk.Label(self, text="").pack()

    button2 = ttk.Button(self, text="About",
                        command=lambda: controller.show_frame(PageTwo)) # got to about
page
    button2.pack()

    ttk.Label(self, text="").pack()

    img = ImageTk.PhotoImage(Image.open(r'wallpaper.jpg').resize((1200, 700))) # set
the home page image
    img.image = img
    ttk.Label(self, image=img).pack()

# ***** PAGES *****
# Detect page
class PageOne(ttk.Frame):

    def __init__(self, parent, controller):
        ttk.Frame.__init__(self, parent)
        label = ttk.Label(self, text="Detect", font=("Simplifica", 22)) # page heading
        label.pack(pady=5, padx=5)

        ttk.Label(self, text="\n").pack()

        ttk.Label(self, text="Enter a webpage", font=(18)).pack()
        text = tk.Entry(self, font=(26), width=70, bg="lightgray") # textbox to enter a website
        text.pack()

        ttk.Label(self, text="").pack()

        # load the file containing fixed keywords
        j = []
        f = open(r'keywords.txt')

```

```

for line in f:
    j.append(line.strip())
f.close()
d = dict.fromkeys(j, 0)

# code to scan the website given in textbox
def scan():
    count = 0
    url = text.get()
    text.delete(0, "end")
    result = requests.get(url.strip())
    soup = BeautifulSoup(result.content, 'lxml')
    for i in soup.get_text().split():
        if (i.lower() in j):
            count += 1
        if i.lower() in d:
            d[i.lower()] += 1
    l3.config(state=tk.NORMAL)
    l3.delete('1.0', "end")
    di = dict(sorted(d.items(), reverse=True, key=lambda item: item[1]))
    lis = [(k, v) for k, v in di.items() if v >= 1]
    l3.insert(tk.END, url.strip() + " = " + str(count) + "\n\nKeywords matched: \n" +
json.dumps(lis))
    l3.config(state=tk.DISABLED)

b2 = ttk.Button(self, text="Scan", command=scan)
b2.pack()

ttk.Label(self, text="").pack()

# code to open and scan the list of websites given in a text file
def open_n_scan():
    files = askopenfile(mode='r', filetypes=[("Text File", "*.txt")])
    l3.config(state=tk.NORMAL)
    l3.delete('1.0', "end")
    for url in files:
        count = 0
        result = requests.get(url.strip())
        soup = BeautifulSoup(result.content, 'lxml')
        for i in soup.get_text().split():
            if (i.lower() in j):
                count += 1
        l3.insert(tk.END, url.strip() + " = " + str(count) + "\n")
    l3.config(state=tk.DISABLED)

ttk.Label(self, text="Select your text file containing urls", font=(18)).pack()

b1 = ttk.Button(self, text="Open and Scan", command=open_n_scan)
b1.pack()

```

```

        ttk.Label(self, text="").pack()

        l3 = scrolledtext.ScrolledText(self, font=(18), height=10, width=70, bg="lightgray",
                                         state=tk.DISABLED) # multiline textbox
        l3.pack()

        ttk.Label(self, text="").pack()

        button1 = ttk.Button(self, text="Back to Home",
                              command=lambda: controller.show_frame(Startpage)) # go to home
page
        button1.pack()

        ttk.Label(self, text="").pack()

        button2 = ttk.Button(self, text="About",
                              command=lambda: controller.show_frame(PageTwo)) # got to about
page
        button2.pack()

# About page
class PageTwo(ttk.Frame):

    def __init__(self, parent, controller):
        ttk.Frame.__init__(self, parent)
        label = ttk.Label(self, text="About", font=("Simplifica", 22)) # page heading
        label.pack(pady=5, padx=5)

        ttk.Label(self, text="").pack()

        button1 = ttk.Button(self, text="Back to Home",
                              command=lambda: controller.show_frame(Startpage)) # got to home
page
        button1.pack()

        ttk.Label(self, text="").pack()

        button2 = ttk.Button(self, text="Detect",
                              command=lambda: controller.show_frame(PageOne)) # got to detect
page
        button2.pack()

        ttk.Label(self, text="").pack()

        tk.Message(self, relief="sunken", bd=4, font=(20), width=1100,
                    text="Suicide is a serious public health problem that can have long-lasting
effects on individuals, families, and communities. The good news is that suicide is
preventable. Preventing suicide requires strategies at all levels of society. This includes
prevention and protective strategies for individuals, families, and communities. Everyone

```

```

can help prevent suicide by learning the warning signs, promoting prevention and
resilience, and a committing to social change.")).pack()
    # Info on suicide
    ttk.Label(self, text="").pack()

    tk.Message(self, relief="sunken", bd=4, font=(20), width=1100,
                text=" Contact the National Suicide Prevention Lifeline Call 1-800-273-TALK
(1-800-273-8255) You'll be connected to a skilled, trained counselor in your area. For
more information, visit the National Suicide Prevention Lifelineexternal icon").pack()
    # Info on suicide content
    ttk.Label(self, text="").pack()

    tk.Message(self, relief="sunken", bd=4, font=(20), width=1100,
                text=" Coping and problem-solving skills, Cultural and religious beliefs that
discourage suicide, Connections to friends, family, and community support, Supportive
relationships with care providers, Availability of physical and mental health care, Limited
access to lethal means among people at risk").pack()
    # About
    ttk.Label(self, text="").pack()

    ttk.Label(self, text="", font=(20)).pack() # copyright

app = MyApp()

# set default app theme
style = ThemedStyle(app)
style.set_theme("plastik")

# set app icon
icon = ImageTk.PhotoImage(Image.open(r'icon.jpg'))
app.iconphoto(True, icon)

app.resizable(0, 0)
app.title("Detect web pages with suicidal content") # app title
app.state('zoomed') # maximized app by default
app.mainloop()

```

