Windows Based Malware Prediction System using Deep Learning Techniques

Devika Anil
Department of Computer Science and Engineering,
Ramaiah Institute of Technology,
Bangalore,India

Aravind P Anil
Department of Computer Science and Engineering,
Ramaiah Institute of Technology,
Bangalore,India

Aatish Kayyath
Department of Computer Science and Engineering,
Ramaiah Institute of Technology,
Bangalore,India

Abishek Padaki
Department of Computer Science and Engineering,
Ramaiah Institute of Technology,
Bangalore,India

Dr. Rajarajeshwari S.
Department of Computer Science and Engineering,
Ramaiah Institute of Technology,
Bangalore,India

Abstract— Malware attack is a very large domain of cyber security attacks. Cryptanalysts across the world has been in a long drawn out battle which has intensified in the past decade with malware. With increase in the technological capabilities of computers, there is also a distinct sharp increase in the capabilities of what malware can do and more importantly, how a malware can prevent from being detected. This project aims at developing a model which accurately predicts the probability that Windows operating system will be hit by a malware. It works on an operating systems dataset that has over 7 million recorded operating systems and their various features, generated by combining heartbeat and threat reports collected by Microsoft's endpoint protection solution, Windows Defender. In order to understand the working of a wide variety of models on such a problem, three different models will be developed and assessed for its accuracy at predicting malware. Three different models under consideration are recurrent neural network, LightGBM technique and lastly a factorization model over a convolutional neural network known as XGBoost. This approach is assessing vulnerability of the system rather than the attacker. If the attacker is constantly evolving and learning new techniques against the system's defence, then efforts to defend against certain types of attacks are futile. Hence, predicting an attack in a more generic sense before it has even happened by assessing the system itself is the better alternative. Even though there are variants, a malware always targets a vulnerability or an exploit of the system to attack. If these weak points on the system are found and patched up before an attack happens, we can develop a very secure and malware proof security configuration.

Keywords—Malware; Windows Malware Predictor; LightGBM; Deep Learning; Prediction Systems; Performance Analysis; XDeepFM; Neural Network.

I. INTRODUCTION

Malware is software that is aimed at intentionally causing a system to behave in a way that it should not. Its main objective is to disrupt the system's normal functioning or cause damage to the system itself and its components or both. Malware comes in many different forms, but irrespective of the type, their objective is one - damage to a system. Malware attack is a very large domain of cyber security attacks. Cryptanalysts across the world has been in a long drawn out battle which has intensified in the past decade with malware. With increase in the technological capabilities of computers, there is also a distinct sharp increase in the capabilities of what malware can do and more importantly, how a malware can prevent from being detected.

There are two main types of defense against malware: malware detection and malware prevention. This work focus on the former and remove the problem before it even arises. There are over a billion potential systems in the world that potentially be affected by malware. Designing a technique to prevent attacks from all different types of malware can be extremely difficult as compared to detecting the causes of malware and predicting if a machine will be hit with malware.

If we check and analyze different types of malware that exist at this point in time, we may come up with a very efficient system to guard against attacks. However, this is very difficult as it requires documentation of every malware that has ever existed not to mention the very complicated and intricate system which requires different techniques against different types of malware. This makes malware detection essentially a time-series problem, but it is made complicated by the introduction of new machines, machines that come online and offline, machines that receive patches, machines that receive new operating systems, etc.

This leaves us in a predicament - stopping malware attacks by preparing ourselves against the different types of malware is practically impossible because it has to done exhaustively. Not doing so puts the system at great danger which defeats the purpose of defending a malware attack. Also, when technology grows leaps and bounds, so does the capability of malware. True exhaustive analysis of malware cannot be done in theory or practice. The list of malwares is constantly growing and evolving. True preparation against attacks takes a whole different approach.

This approach is assessing vulnerability of the system rather than the attacker. If the attacker is constantly evolving and learning new techniques against the system's defense, then efforts to defend against certain types of attacks are futile. Hence, we try to predict an attack in a more generic sense before it has even happened by assessing the system itself. This theory is mainly based on the assumption that malware is targeted. Even though there are variants, a malware always targets a vulnerability or an exploit of the system to attack. If these weak points on the system can be found out and patch them before an attack happens, a very secure and malware proof security configuration can be developed.

The rest of the paper contains the following contents. Section II discusses the related work, the existing systems and their working. Section III of this paper describes the proposed system. The technical considerations that underpin our experiments and the results obtained are given in Section IV. Section V has concluding remarks and further work.

II. RELATED WORK

A. Malware and Malware Obfuscation Techniques

Stuxnet [1] is one of the most advanced and sophisticated worms ever written. It was designed by the Intelligence agency in the United States and targeted at nuclear refinement plans in Iran. They released a worm through USB to the general public which latches on to any system and gathers administrative privileges for the system using zero-day exploits. This virus uses a total of 4 zero-day exploits. Then it searches for specific Siemens Centrifuge controllers connected to the system. If it does not exist, the worm destroys itself. Once it has control of the controller, it slowly varies the centrifuge speeds over a period of time so that they are all destroyed within a year. The also increase the gas pressure to form rocks within the centrifuges so that they degrade in quality – slowly but steadily. The worm is hailed as one of the most complicated codes ever written in the history of mankind. It floated on the internet for over a year without being detected by any machine or anti-virus.

Obfuscation [2] is by far the most dominant technique along with Signature methods used by malware for evasion and staying out of reach of anti-malware systems. There are various Obfuscation techniques and this paper analyses some of those methods. The simplest obfuscation technique is the dead code technique. It inserts harmless pieces of code at random to various parts of the payload but does not alter the behaviour in any way. The code is placed at clever locations that skip anti-malware scans. Another complementary method is Register Reassignment. Registers support legacy code. By switching from current standard to older generation of registers often, the Malware becomes very hard to find. Subroutine reordering is a very simple but effective Obfuscation technique. It generates n! variants where n is the number of subroutines. It randomly rearranges the subroutine while having minimal changes on the payload or the behavior of the malware. Instruction Substitution and Code Transposition are two other methods of obfuscation. Instruction Substitution splits a single instruction into multiple instructions while having the same effect. It effectively changes code with equivalent instructions. Code transposition reorders entire sequences of codes rather than just subroutines.

Private stream searching appears to be an entirely effective method for malware to surreptitiously search and exfiltrate email by resisting malware analysis techniques [3]. Malware designed to save and return messages on a specific sensitive topic will be able to do so without revealing the topic of interest upon analysis; all that will be determined is that it scans email in general. Furthermore, as the paper's implementation demonstrates, there is nothing to prevent these techniques from being used immediately. The example of PIR-based malware illustrates the more general possibility of malware employing public key obfuscation techniques to hide its behavior.

The traditional security systems like Intrusion Detection System/Intrusion Prevention System and Anti-Virus (AV) software are not able to detect unknown malware as they use signature-based methods. In order to solve this issue, static and dynamic malware analysis is being used along with machine learning algorithms for malware detection and classification. The main problems with these systems are that they have high false positive and false negative rate and the process of building classification model takes time (due to large feature set) which hinders the early detection of malware. Thus, the challenge is to select a relevant set of features, so that, the classification model can be built in less time with high accuracy. The proposal presents a system that addresses both the issues mentioned above. It uses an integration of both static and dynamic analysis features of malware binaries incorporated with machine learning process for detecting Zero-day malware [4]. The proposed model is tested and validated on a real-world corpus of malicious samples. The results show that the static and dynamic features considered together provide high accuracy for distinguishing malware binaries from clean ones and the relevant feature selection process can improve the model building time without compromising the accuracy of malware detection system.

B. Malware Detection Techniques

Malware sandboxes [5] are one of the most popular methods used by anti-virus testing engineers to find out if the effects of various malware in order to develop and create sophisticated counter measures. These are virtual testing playgrounds that are built with weak protective measures in order to get affected by a virus. However modern viruses have come up with a very new and innovative method against sandboxes in testing. The virus creators' aim is to make sure the virus or malware is not caught at the testing phase and will go under the radar of the sandbox. Hence, without a countermeasure, it can release fatal payloads onto real systems for adverse effects. The proposed methodology to evade sandboxes are by ensuring differential behavior in sandboxes. By detecting the environment, it releases payload conditionally. This publication provides a method for sandboxes to catch such malwares with advanced evasion techniques by introducing wear and tear to the malware. Essentially, the sandbox is made as close to a real-life system as possible – specifically an old system. This is the wear and tear being introduced. It can be event logs, recycle bin size, cache entries, network entries, registry, cookie count and so on.

An analysis on malware detection and removal techniques designed for android devices [6] shows worrisome results. Mobile Industry has grown exponentially at a very large rate in the last few years. This period of time is also marked with an increase in malware count designed for android. However, the detection rates stay the same for the most part. The analysis classifies various malware evasion techniques into three categories – Trivial, DSA or static analysis, and Undetectable. The malware is taken through one of these transformations and passed into a system with an anti-malware setup on it. The study came up with many findings. The major takeaway is that all anti-malware systems on android are vulnerable to many transformations and are not up to standard. The second finding is that android anti-malware works primarily based on code level artefacts – package names, asset names etc. However, at least 43% of malware don't use these techniques at all and hence have an easy path into the system. 90% of the malware designed did not need protection against static analysis of bytecode. The android security system does not bother implementing this vital layer on their systems and hence it is a waste of resource to enable evasion techniques for this method.

In the light of works done in regard to semantics aware methodology [7] for malware detection, aim for such technique is to understand and detect the presence of any malicious intent in a given program. Malware is known to evolve itself through the process of polymorphism or metamorphism in order to evade detection. This procedure is closely referred to as program obfuscation. Adding slightly new behavioral changes is said to modify the malware to a level where it may not be detected. The malware detection systems used a simple pattern matching technique that identified a certain sequence of instructions that is labelled malware using regular expressions. Understanding semantics of the program instruction will help overcome deficiencies brought on by the above mentioned basic technique. This also eliminates the need to frequently update the database used by the commercial virus scanners for all updates of the malware. Here, we learn the context of malicious behavior is found from the instruction sequence. When a template and an instruction sequence are executed from a state where the contents of the memory are the same, then after both the executions the state of the memory is the same. In other words, the malicious behavior specified by the template is demonstrated by the instruction sequence. Our malware detection algorithm AMD works by finding, for each template node, a matching node in the program. Nodes from the template and the program match if there exists an assignment to variables from the template node expression that unifies it with the program node expression. Once two matching nodes are found, we check whether the def-use relationships true between template nodes also hold true in the corresponding program (two nodes are related by a def-use relationship if one node's definition of a variable reaches the other node's use of the same variable). If all the nodes in the template have matching counterparts under these conditions, the algorithm has found a program fragment that satisfies the template and produces a proof of this relationship. Experimental evaluation of this algorithm has shown ability to detect all variants of certain malware, has no false positives, and is resilient to obfuscation transformations generally used by hackers.

Next study involved understanding the working of a malware detection system used to understand and perform behavior based analysis [8] of malware on android systems called the crowdroid. In this system , a lightweight client is downloaded and used on smartphones as opposed to the security tools and mechanisms used in computers which are not feasible for applying on smartphones due to the excessive resource consumption and battery depletion. Then, the remote server will be in charge of parsing data, and creating a system call vector per each interaction of the users within their applications. Thus, a dataset of behavior data will be created for every application used. The more users using our Crowdroid application, the more complete and accurate will be our system. Finally, we cluster each dataset using a partitional clustering algorithm. This way we can differentiate between benign applications that demonstrate very similar system call patterns, and malicious Trojan applications that, even if having the same name and identifier, have a different behavior in terms of distance between example vectors. It is quite often seen that open(), read(), access(), chmod() and chown() are the most used system calls by malware. A benign application could make moderate or heavy use of those system calls and thus trigger false positives. We need to manage the perception of loss of privacy when supporting research community with their behavior information, against the benefit of having access to up-to-date behavioral-based detected malware statistics.

Normalized Compression Distance (NCD)[9] is a tool that uses compression algorithms to cluster and classify data in a wide range of applications. The author has demonstrated that several compression algorithms, lzma, bz2, zlib, and PPMZ, apparently fail to satisfy the properties of a normal compressor, and explored the implications of this on their capabilities for classifying malware with NCD. More generally, they have shown that file size is a factor that hampers the performance of NCD with these compression algorithms. Specifically, they have found that lzma performs best on this classification task when files are large (at least in the range we explored), but that bz2 performs best when files are sufficiently small. They have also found zlib to generally not be useful for this task. PPMZ, in spite of being the top performer in terms of idempotence, did not come close to the most accurate compressor in any case. Finally, we introduced two simple file combination techniques that improve the performance of NCD on large files with each of these compression algorithms. The author has explored the relationship between some of these properties and file size, demonstrated that this theoretical problem is actually a practical problem for classifying malware with large file sizes, and proposed some variants of NCD that mitigate this problem.

Malware detection systems have been driven by models learned from input features collected from real or simulated environments [10]. A potential malware sample, suspicious email is deemed malicious or non-malicious based on its similarity to the learned model at runtime. However, the training of the models has been historically limited to only those features available at runtime but in this paper the author has considered l an alternate learning approach that trains models using “privileged” information–features available at training time but not at runtime– to improve the accuracy and resilience of detection systems. In particular they have adapted and extended recent advances in knowledge transfer and lpmodel influence to enable the use of forensic or other data unavailable at runtime in a range of security domains. The evaluation shows that privileged information increases precision and recall over a system with no privileged information: we observe up to 7.7% relative decrease in detection error for fast-flux bot detection, 8.6% for malware traffic detection, 7.3% for malware classification.

The author introduce ITect[11], a scalable approach to malware similarity detection based on information theory. ITect targets file entropy patterns in different ways to achieve 100% precision with 90% accuracy but it could target 100% recall instead. It outperforms VirusTotal for precision and accuracy on combined Kaggle and VirusShare malware. ITect opens a new front in the arms race. Its level of abstraction makes it difficult to counter and it offers scalability advantages. The authors have demonstrated excellent precision and accuracy on a representative mixture of malware types drawn from the Kaggle malware data and VirusShare. They have demonstrated that both of its constituent detectors, EnTS and SLaMM, outperform previous information theoretic similarity measures. Indeed, ITect outperforms existing AntiVirus engines (as represented in VirusTotal) for accuracy and precision. Its time complexity is bounded above by the number of files to be classified. As an automated, execution agnostic, string-based similarity metric it offers wider scalability advantages beyond its time complexity class alone – reducing human effort and reducing the need for dynamic or static analysis.

C. The use of machine learning and deep learning in malware detection systems

With the advent of cheap computing power available freely for the general public, malware detection has become near impossible [12]. Malware uses various techniques such as Encryption, Oligomorphic, Polymorphism, Metamorphism, Obfuscation, Fragmentation, Session splicing, Protocol Violations, Code reuse Attacks and more. The most common malware detection techniques used are Signature method, Behavior method, and Heuristic method. However, these are simply not enough to catch the complex multi layered evasion techniques that most malwares use nowadays. The method being used here is a neural network to catch trojans. The suspected payloads are injected as weights into the neural network. The network catches the Trojan when the network receives a very specific type of data.

This next research work [13] gave us an insight into a couple of data mining and machine learning models used to understand malware and detect them prior to affecting the system. Although classification methods based on shallow learning architectures, such as Support Vector Machine (SVM), Naive Bayes (NB), Decision Tree (DT), and Artificial Neural Network (ANN), can be used to solve the above malware detection problem, deep learning has been demonstrated to be one of the most promising architectures for its superior layer wise feature learning models and can thus achieve comparable or better performance. Typical deep learning models include stacked AutoEncoders (SAEs), Deep Belief Networks with Restricted Boltzmann Machine, Convolutional Neural Networks etc. In this paper, we explore a deep learning architecture with SAEs model for malware detection. The SAE model is a stack of AutoEncoders, which are used as building blocks to create a deep network. An AutoEncoder, also called AutoAssociator, is an artificial neural network used for learning efficient codings. Architecturally, the form of an AutoEncoder is with an input layer, an output layer and one or more hidden layers connecting them. The goal of an AutoEncoder is to encode a representation of the input layer into the hidden layer, which is then decoded into the output layer, yielding the same (or as close as possible) value as the input layer. To form a deep network, an SAE model is created by daisy chaining AutoEncoders together, known as stacking. To use the SAEs for malware detection, a classifier needs to be added on the top layer. In this application, the SAEs and the classifier comprise the entire deep architecture model for malware detection.

The involvement of deep learning architectures such as neural networks in the classification of malware [14] is the insight given by this next research work. Results show that deep learning indeed brings improvements in classification of malware system call traces. Combining convolutional and recurrent layers was the superior idea to achieve this improvement. This combination helps us obtain slightly better results than with simpler architectures with only feed forward or only convolutional layers. Using only LSTM recurrent network also does not achieve accuracy as high as we get with our architecture, which can be explained with the relatively short length of the malware execution traces. Although training time for neural network ranges from three to ten hours, on test time the classification is instantaneous. Therefore the neural network approach is very good in case where there is no common need to retrain the model. Overall, this approach exhibits better performance results when compared to previous malware classification approaches.

Malware detection based on machine learning techniques [15] is often treated as a problem specific to a particular malware family but classifying samples as malware or benign based on a single model would be far more efficient. However, such an approach is extremely challenging as extracting common features from a variety of malware families might result in a model that is too generic to be useful. The author in this paper has used four different machine learning techniques — support vector machines (SVM), χchi-squared test, k-NN, and random forests using n-grams as features to evaluate the effectiveness of generic malware models. To summarize, SVM, χ 2 , k-NN and random forests all performed well for individual malware families. The accuracy for all of these techniques dropped significantly when the models were more generic. Some of these techniques did better than others in the more generic cases. The random forest specifically was the strongest classifier with an accuracy of 88% for the most generic model.

PowerShell is increasingly used by cybercriminals as part of their attacks' tool chain, mainly for downloading malicious contents and for lateral movement. Indeed, a recent comprehensive technical report by Symantec dedicated to Power-Shell's abuse by cybercriminals reported on a sharp increase in the number of malicious PowerShell samples they received and in the number of penetration tools and frameworks that use PowerShell. This highlights the urgent need of developing effective methods for detecting malicious PowerShell commands. In the proposed paper, this challenge is addressed by implementing several novel detectors of malicious Power-Shell commands [16] and evaluating their performance. The proposal implements both "traditional" natural language processing (NLP) based detectors and detectors based on character-level convolutional neural networks (CNNs).

The method proposed [17] is an innovative method for detecting malware which uses combined features (static and dynamic) to classify whether a portable executable file is malicious or benign. The method employs two types of neural networks to fit distinct property of respective work pipelines. The first type of neural network used is recurrent neural network that is trained for extracting behavioral features of PE file, and the second type is convolutional neural network that is applied to classify samples. At the training stage, first the static information of a PE file is extracted and sandbox is used to record system API call sequences as dynamic behaviors. Then extraction of static features based on predefined rules and dynamic features out of the trained RNN model. Next they are combined and use well design algorithm to create images. Lastly, the concurrent classifier is trained and validated using images created in the previous steps labeled with 1(malicious) or 0(benign).

III. PROPOSED METHOD

This paper proposes three different algorithm strategies to solve the same problem at hand. The three different algorithms are LightGBM, xDeepFM and a recurrent neural network model. Feature Engineering performed to obtain maximum accuracy from the different types of algorithms have also been explained in detail. LightGBM, a decision tree based boosting algorithm, was chosen to understand the working of slow learners on this problem and its effects on large datasets. Recurrent neural network model would help portray the effect of strong deep learning algorithm. xDeepFM is a convolutional neural network teamed with factorization feature selection model to help out with sparse data. Detailed feature engineering has been performed prior to training the model so as to improve accuracy. All the three models had additional variables engineered from the dataset by performing time split validation. LightGBM and the recurrent neural network models involved feature based and one hot encoding to make training easier. xDeepFM did require encoding as it involves factorization method within it.

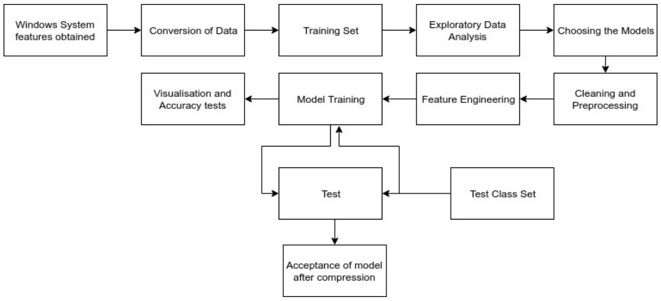
Windows System features obtained
Conversion of Data
Training Set
Exploratory Data Analysis
Choosing the Models
Cleaning and Preprocessing
Feature Engineering
Model Training
Visualisation and Accuracy tests
Test
Test Class Set
Acceptance of model after compression

Figure 3.1: Flow of data through the deep learning model

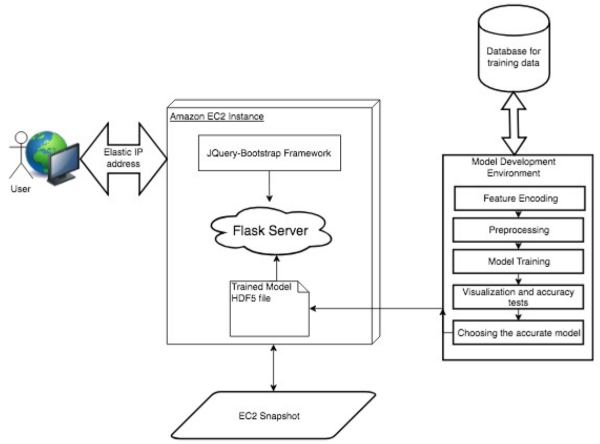
Database for training data
Amazon EC2 Instance
User
Elastic IP address
JQuery-Bootstrap Framework
Flask Server
Trained Model HDF5 file
Model Development Environment
Feature Encoding
Preprocessing
Model Training
Visualization and accuracy tests
Choosing the accurate model
EC2 Snapshot

Figure 3.2: Complete end-to-end system architecture

The deep learning flowchart has been depicted in figure 3.1. Figure 3.1 shows the system build to analyze an OS vulnerability to malware. The final product has to be available publicly to anybody. Hence, rather than running it off a local system, the model and the GUI run on an EC2 instance..The model is connected to the frontend via a server whose routes and API endpoints are created using a flask server. The server fetches the trained model details from an h5 file which stores the model itself. We also store a small sample of the dataset within the EC2 instance

A. Feature Engineering

Time split validation, feature encoding, and elimination of features by variable importance and cross-validation techniques are the secret ingredients to improving accuracy that is used in all the models under this study.

a. Time Split Validation

The following variables were discovered by trying engineered variables, based on the knowledge of operating systems and malware, to see which ones increased Time Split Validation. Each variable was added to the model one at a time and validation score was recorded. Only the follow 5 variables increased validation score.

- AppVersion2 indicates whether your Windows Defender is up to date. This is the second number from AppVersion. Regardless of your operating system and version, you can always have AppVersion with second number equal 18. For example, you should have 4.18.1807.18075 instead of 4.12.xx.xx.
- Lag1 is the difference between AvSigVersion_Date and Census_OSVersion_Date. Since AvSigVersion is the virus definitions for Windows Defender, this variable indicates whether Windows Defender is out-of-date by comparing its last install with the date of the operating system. Out-of-date antivirus indicates that a user either has better antivirus or they don't use their computer often. In either case, they have less HasDetections.
- Lag5 is the difference between AvSigVersion_Date and July 26, 2018. The first observation in Microsoft's training data is July 26, 2018. Therefore if a computer has AvSigVersion_Date before this then their antivirus is out-of-date. (The first observation in the test data is September 27, so you use this difference when encoding the test data.)
- driveA is the ratio of hard drive partition used for the operating system with the total hard drive. Savy users install multiple operating systems and have a lower ratio. Savy users have reduced HasDetections.
- driveB is the difference between harddrive partition used for the operating system and total hard drive. Responsible users manager their hard drives well. Responsible users have reduced HasDetections.

b. Feature Encoding

In many practical data science activities, the data set will contain categorical variables. Since machine learning is based on mathematical equations, it would cause a problem when we keep categorical variables as is.

The following types of frequency encoding has been performed on the categorical variables made available to us by Microsoft:

- Frequency encoding:

It is a way to utilize the frequency of the categories as labels. In the cases where the frequency is related somewhat with the target variable, it helps the model to understand and assign the weight in direct and inverse proportion, depending on the nature of the data. There are a lot of time dependent variables that exist like (EngineVersion, AvSigVersion, AppVersion, Census_OSVersion, Census_OSBuildRevision) .

And the test set would have variables from versions from beyond the time, the train and test variables are frequency encoded separately.

- One Hot Encoding:

In this method, we map each category to a vector that contains 1 and 0 denoting the presence of the feature or not. The number of vectors depends on the categories which we want to keep.

Among all our category variables, there are a combined 211,562 values. It is tedious to one-hot-encode all.

Then for each value, we will test the following hypotheses

H0: Prob (HasDetections=1 given value is present) =0.5

HA: Prob (HasDetections=1 given value is present) ≠0.5

The test statistic z-value equals $\hat{p}$, the observed HasDetections rate given value is present, minus 0.5 divided by the standard deviation of $\hat{p}$. The Central Limit Theorem tells us

$$Z\text{-value}=\hat{p} -0.5SD (\hat{p})$$

If the absolute value of z is greater than 2.0, we are 95% confident that Prob (HasDetections=1 given value is present) is not equal 0.5 and we will include a Boolean for this value in our model. Actually, we'll use a threshold of 5.0 and require $10-7n>0.005$. This adds lesser new Boolean variables.

Note that these feature encoding methods have only been used in LightGBM and recurrent neural network models. XDeepFM has its own factorization module.

B. Light Gradient Boosting Method

Boosting is a technique of making weak learners perform better by making modifications. LGBM grows tree leaf-wise while other algorithm grows level-wise. Leaf-wise algorithm can reduce more loss than a level-wise algorithm. It can handle the large size of data and takes lower memory to run. It focuses on accuracy of results and supports GPU learning. It is sensitive to overfitting. The training dataset is then converted into sparse matrix because of the huge number of null variables present in our dataset and it performs better in training.

After this we perform K fold cross validation where the dataset is divided into k number of sets and then randomly it's trained and tested. For our dataset we have taken k = 10.

Now for LGBM firstly the parameters are decided by hyper parameter tuning.

a. Pseudocode

```
Input: Labeled dataset (consisting of OS environment variables)
Output: Probability of that particular system OS build being affected by malware
Frequency_encoding:
Count the frequency of all the variables separately and store the values
Code <- Divide the count by the maximum count to generalise
Map the variable the to the code now
Return code
One_Hot_Encoding:
category<-Choose category values that comprise more than 50% of and has significance greater than "Z-value"
For i in category:
Return Separate the categorical variable 'i' as new columns according to the same naive OHE method
Preprocessing_Dataset:
For i in Time_Series_Attributes
Frequency_encoding(i)
For i in every_other_data:
One_Hot_Encoding()
K <- 5
Training_Dataset, Test_Dataset ← Split Tokenized training and test dataset using K-fold technique
File.h5 ← Store Training_Dataset, Test_Dataset as an h5 file to be used by all models
Create the model with max_depth=1, n_estimators=30000, learning_rate=0.05,
Fit the Model on Training_Dataset
Training_history ← Save the training history as csv file
Get predictions for the test set and prepare a submission CSV
```

C. Recurrent Neural Network

Recurrent Neural Networks or RNN are robust and powerful types of neural networks which belong to the most promising algorithms present in today's world because they are the only ones with an internal memory. Because of their internal memory, RNN's are able to remember important things about the input they received, which enables them to be very precise in predictions. Recurrent Neural Networks produce predictive results in sequential data that other algorithms cannot predict.

Unlike the Light Gradient Boosting framework model (LGBM), RNN is a strong learner. The RNN model used consists of a three-layer fully connected network with 100 neurons in each layer. The activation function used is ReLU. The Backpropagation dropout in the model used is forty percent. Like the other two models, the pre-processing for the RNN model is done using One Hot Encoding and K-Fold Cross Validation. As per the accuracy scores between the three models, the RNN model was found to be the most accurate with a Training Accuracy of 0.68 (epoch 15) and a Validation Accuracy of 0.6 (Metric Used: ROC-AUC scores).

The disadvantage of this model is the high possibility of gradient vanishing or explosion when using the activation function.

a. Pseudocode

```
Input: Labeled dataset (consisting of OS environment variables)
Output: Probability of that particular system OS build being affected by malware
Frequency_encoding:
Count the frequency of all the variables separately and store the values
Code <- Divide the count by the maximum count to generalize
Map the variable the to the code now
Return code
One_Hot_Encoding:
category<-Choose category values that comprise more than 50% of and has significance greater than "Z-value"
For i in category:
Return Separate the categorical variable 'i' as new columns according to the same naive OHE method
Preprocessing_Dataset:
For i in Time_Series_Attributes
Frequency_encoding(i)
For i in every_other_data:
One_Hot_Encoding()
K <- 5
Training_Dataset, Test_Dataset ← Split Tokenized training and test dataset using K-fold technique
File.h5 ← Store Training_Dataset, Test_Dataset as an h5 file to be used by all models
Create the model :
Model ← Add two layers of Recurrent Neural Network to the model
Model ← Add necessary dropout values to ensure overfitting does not take place
Model ← Add categorical_crossentropy as loss function and AUC scores as the Accuracy measurement
Fit the Model on Training_Dataset
Training_history ← Save the training history as csv file
Get predictions for the test set and prepare a submission CSV
```

D. XDeepFM

Extreme Deep Factorization Model utilizes deep learning along with a factorization machine. The model provided the highest accuracy among all the implemented models. It combines feature interactions, both implicit and explicit.

The model was primarily used for Recommender Systems. In recommender systems with a very large number of features to train, the results can get very skewed due to the huge number of influences the features have. In certain data sets, the dimension of the features is huge but the input features or actual usable features are very sparse. Other alternatives for classic recommender systems with sparse input features is logistic regression with "Follow the Regularized Leader". XDeepFM provides a very high accuracy when it comes to cross features or multi-way features. These are features that combine categorical raw features to give very specific training scenarios. However, cross feature engineering has disadvantages. Getting significant information or trainable data from cross features is accompanied by a high cost. The computation is heavy and the exploratory data analysis needs to find out specific patterns to choose the features to enable cross features. This is specific to each dataset and data scientists have their work cut out. In massive datasets with hundreds of features, it is not possible to extract all cross features manually to test for accuracy. Finally, the patterns that data scientists pick out for cross featuring can miss out on many hidden or unseen features which may have been actually significant to improving the accuracy.

Due to these problems, we present with a factorization machine. The machine embeds each feature in a latent vector to create pairwise interactions of every feature.

a. Factorization Model

The factorization machine is a vital part of XDeepFM. It helps us consolidate important features from the model rather than letting data scientists do EDA and hand pick the features. The machine compensates for scarcity of the input model and dataset. The machine works by creating a matrix of n dimensions where n is the number of user inputs. However, majority of the matrix is zeroes as the input is scarce. If k is the number of features we need to consolidate, we cross multiply the n matrices to get the final matrix with embedded feature values.

In our model, an embedding layer is applied on the raw input to compress the sparse matrix to a dense low dimensional matrix. This reduces the cost of training by a significant margin.

b. Compressed Interaction Network

While the factorization machine learns features implicitly, the Compressed Interaction Network learns other features explicitly and the extent of growth grows with the depth of the network. The interactions for the CIN will stay at a constant level of complexity and will not grow with interaction depth. The Compressed Interaction Network takes features from both Convolutional Neural Networks and Recurrent Neural Networks. Convolutional Neural Networks work based on moving a filter along the network layer. By taking the dot product, it consolidates the entire layer into a required dimension that can be controlled by the sliding window filter. When it is passed upon multiple layers, we get multiple outputs that can be stacked on one another. These outputs are called single depth slice. The CNN then has a very cost-effective method called pooling. Essentially, it takes a single depth layer and applies a logical relationship over an entire section of the depth. This ensures a lower dimension and easier calculation. It also reduces the number of parameters required.

In our model, the CIN does pooling at each logical layer to reduce the number of parameters. After the pooling is done, it passes the output as the input to the next layer which encompasses the basic structure of a Recurrent Neural Network.

With our dataset, the hidden layer size is a 128 x 128 matrix and the cross layer is a 3D matrix with the same dimensions. We have set the learning rate to 0.001 so as to not overfit with just 1 epoch. This ensures efficiency of the model and at the same time, does not spend additional resources like LGBM on Has Detections.

c. Pseudocode

```
Input: Labeled dataset (consisting of OS environment variables)
Output: Probability of that particular system OS build being affected by malware
Frequency_encoding:
Count the frequency of all the variables separately and store the values
Code <- Divide the count by the maximum count to generalise
Map the variable the to the code now
Return code
One_Hot_Encoding:
category<-Choose category values that comprise more than 50% of and has significance greater than "Z-value"
For i in category:
Return Separate the categorical variable 'i'  as  new columns according to the same naive OHE method
Preprocessing_Dataset:
For  i in Time_Series_Attributes
Frequency_encoding(i)
For i in every_other_data:
One_Hot_Encoding()
K <- 5
Training_Dataset, Test_Dataset ← Split Tokenized training and test dataset using K-fold technique
Factorization Machine - Multiplicative factorization of necessary features with dimension n x k where n is features and k is required features
File.h5 ← Store Training_Dataset, Test_Dataset as an h5 file to be used by all models
Create the model :
Model ← Add two layers of convolutional  Neural Network to the model
Model ← Add necessary dropout values to ensure overfitting does not take place
Model ← Add categorical_crossentropy as loss function and AUC scores as the Accuracy measurement
Fit the Model on Training_Dataset
Training_history ← Save the training history as csv file
Get predictions for the test set and prepare a submission CSV
```

IV. EXPERIMENT AND RESULTS

A. Dataset and Experimental Setting

The goal of this competition is to predict a Windows machine's probability of getting infected by various families of malware, based on different properties of that machine. The telemetry data containing these properties and the machine infections was generated by combining heartbeat and threat reports collected by Microsoft's endpoint protection solution, Windows Defender. Each row in this dataset corresponds to a machine, uniquely identified by a MachineIdentifier. HasDetections is the ground truth and indicates that malware was detected on the machine. Using the information and labels in train.csv, you must predict the value for HasDetections for each machine in test.csv. There are upto 82 columns that showcase various OS build information and upto 89,21,483 rows of values. This information has been obtained from Microsoft. Due to the physical limitations of RAM and CPU, only one-tenth of the data was used for training purposes. Experimental setting used Python based libraries to implement the three deep learning models.

B. Evaluation Metrics

The trained model is evaluated using three evaluation metrics - Validation Accuracy, Validation Loss, Training accuracy and Training Loss, calculated using ROC-AUC metric.

C. Experimental Results

Fold	Training Loss	Training AUC	Validation Loss	Validation AUC
1	0.692329	0.515657	0.693036	0.515532
2	0.693624	0.50221	0.693088	0.50213
3	0.692962	0.508201	0.693061	0.508194
4	0.692039	0.520162	0.693026	0.520123
5	0.693034	0.516667	0.693034	0.516452

Table 4.1: LightGBM Results

Table 4.1 shows the results obtained by LightGBM on the dataset. The model was trained for 5 folds. The accuracy is found to remain the same from the 1st fold till the 3rd fold. The maximum accuracy is obtained at the 4th epoch at a value of 0.520162. The least loss is also obtained at the 4th fold.

Fold	Validation Loss	Validation AUC	Training Loss	Training AUC
1	0.5981	0.7373	0.6225	0.718
2	0.6065	0.7276	0.5981	0.737
3	0.6213	0.7215	0.598	0.7383
4	0.6238	0.721	0.597	0.7389

Table 4.2: XDeepFM Results

Table 4.2 shows the results obtained by XDeepFM on the dataset. The XDeepFM was trained for 5 folds. The accuracy is found to have a steady growth throughout. The maximum accuracy is obtained at the 4th fold at a value of 0.7389. The least Training loss is obtained at the 4th fold too.

Epoch	Loss	Accuracy	Validation Accuracy	Validation Loss	Training AUC	Validation AUC
1	0.6899	0.5736	0.6563	0.587	0.6654	0.6369
2	0.6596	0.598	0.6549	0.5976	0.667	0.6486
3	0.6608	0.5954	0.6545	0.6022	0.6697	0.6478
4	0.6557	0.6028	0.6517	0.602	0.6669	0.6444
5	0.652	0.6074	0.6518	0.6012	0.671	0.6509
6	0.6524	0.6086	0.6565	0.5902	0.6716	0.6494
7	0.6541	0.6108	0.6567	0.603	0.6751	0.6519
8	0.6508	0.6184	0.653	0.606	0.6753	0.6516
9	0.6503	0.619	0.6519	0.6066	0.6755	0.6499
10	0.6498	0.615	0.653	0.6082	0.6765	0.6512
11	0.6479	0.6116	0.6522	0.6098	0.6773	0.6543
12	0.6473	0.623	0.6574	0.6074	0.6771	0.6516
13	0.6516	0.6128	0.6554	0.6082	0.6772	0.6514
14	0.6515	0.614	0.6519	0.6102	0.6768	0.6502
15	0.6492	0.6232	0.6517	0.6068	0.6789	0.6514
16	0.6447	0.6266	0.6537	0.6068	0.6812	0.6519
17	0.6492	0.6126	0.6567	0.608	0.6796	0.6494
18	0.648	0.616	0.6519	0.6096	0.6787	0.6497
19	0.6441	0.6216	0.651	0.608	0.6835	0.6528
20	0.6472	0.62	0.6532	0.6028	0.6827	0.6505

Table 4.3: Recurrent Neural Network results

Table 4.3 shows the results obtained by the recurrent neural network on the dataset. This model was trained for 20 epochs. The accuracy is found to have a steady growth till the 13th epoch after which it drops. The maximum accuracy is obtained at the 19th epoch at a value of 0.6835.

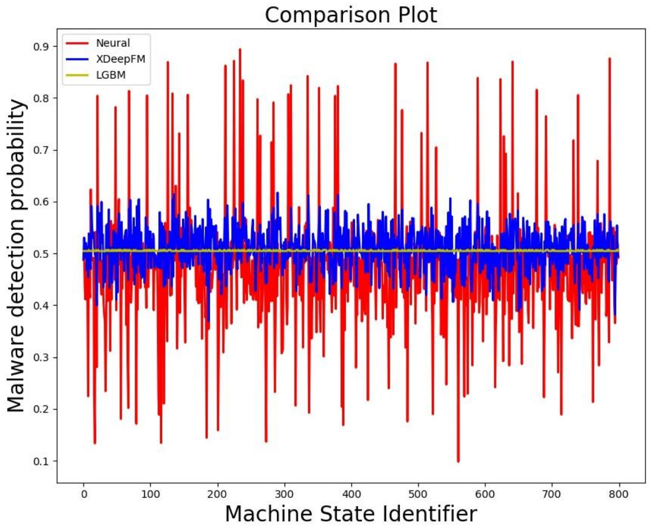

Figure 4.1 Model Behavior Comparison Plot

The above graph depicts the behavior of the three different models of 1000 test data values. It is clearly observable that the LGBM model is incapable of differentiate between the vulnerable and least vulnerable machines and plays it safe by predicting all values around 50 percent. Neural network model is able to understand the extreme cases well. XDeepFM also identifies a similar pattern but with much more efficiency.

V. CONCLUSION AND FUTURE WORK

We intend to build a model for predicting malware existence in operating systems with an impeccable accuracy. The outcome will also contain a hassle free user interface for the same as well. Future work could comprise of combining the works of detecting malware from the system codes written by understanding their semantics to help predict. We could also classify being detected along with predictions. Malware can be very devastating depending on the target entity. Key-loggers can gain confidential information on entire organizations which can have monetary impact of billions of dollars Viruses like Stuxnet can change the fate of an entire country by altering war. Catching a virus before it hits a system will improve efficiency and simplify code base of various anti-virus deployments.

VI. REFERENCES

[1] Baezner, Marie, and Patrice Robin. Stuxnet. No. 4. ETH Zurich, 2017.
[2] You, Ilsun, and Kangbin Yim. "Malware obfuscation techniques: A brief survey." In 2010 International conference on broadband, wireless computing, communication and applications, pp. 297-300. IEEE, 2010.
[3] Bethencourt, John, Dawn Xiaodong Song and Brent Waters.“Analysis-Resistant Malware”. In: NDSS (2008)
[4] Ekta Gandotra, Divya Bansal, Sanjeev Sofat (2016) “Zero-Day Malware Detection”. In: 2016 Sixth International Symposium on Embedded Computing and System Design (ISED).
[5] Miramirkhani, Najmeh, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts." In 2017 IEEE Symposium on Security and Privacy (SP), pp. 1009-1024. IEEE, 2017.
[6] Rastogi, Vaibhav, Yan Chen, and Xuxian Jiang. "Catch me if you can: Evaluating android anti-malware against transformation attacks." IEEE Transactions on Information Forensics and Security 9, no. 1 (2014): 99-108.
[7] Christodorescu, Mihai, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. "Semantics-aware malware detection." In 2005 IEEE Symposium on Security and Privacy (S&P'05), pp. 32-46. IEEE, 2005.
[8] Burguera, Iker, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: 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. ACM, 2011.
[9] Borbely, Rebecca Schuller. "On normalized compression distance and large malware." Journal of Computer Virology and Hacking Techniques 12, no. 4 (2016): 235-242.
[10] Burnaev, Evgeny, and Dmitry Smolyakov. "One-class SVM with privileged information and its application to malware detection." In 2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW), pp. 273-280. IEEE, 2016.
[11] Bhattacharya, Sukriti, Héctor D. Menéndez, Earl Barr, and David Clark. "Itect: Scalable information theoretic similarity for malware detection." arXiv preprint arXiv:1609.02404 (2016).
[12] Jhonattan J Barriga, Sang Guun Yoo. “Malware Detection and Evasion with Machine Learning Techniques: A Survey” International Journal of Applied Research 2017
[13] Hardy, William, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin Li. "DL4MD: A deep learning framework for intelligent malware detection." In Proceedings of the International Conference on Data Mining (DMIN), p. 61. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016.
[14] Kolosnjaji, Bojan, Apostolis Zarras, George Webster, and Claudia Eckert. "Deep learning for classification of malware system call sequences." In Australasian Joint Conference on Artificial Intelligence, pp. 137-149. Springer, Cham, 2016.
[15] Bagga, Naman. "Measuring the Effectiveness of Generic Malware Models." (2017).

[16] Danny Hendler, Shay Kels and Amir Rubin (2018) “Detecting Malicious PowerShell Commands using Deep Neural Networks”. In: arXiv:1804.04177, Cornell University, 2018.

[17] Zhou H. (2019) Malware Detection with Neural Network Using Combined Features. In: Yun X. et al. (eds) Cyber Security. CNCERT 2018. Communications in Computer and Information Science, vol 970. Springer, Singapore