International Journal of Computer Science Trends and Technology (IJCST) – Volume 12 Issue 2, Mar - Apr 2024

RESEARCH ARTICLE OPEN ACCESS

Integrating Swarm Intelligence with Machine Learning Techniques for Android Malware Detection through API Call Analysis

Thottadi Lakshmi Prasanna [1], Madugula Murali Krishna [2]
[1] PG Scholar, Dept. of CSE, Sri Sivani College of Engineering, Srikakulam, AP - India.
[2] Assistant Professor, Dept. of CSE, Sri Sivani College of Engineering, Srikakulam, AP - India.

ABSTRACT
The increasing number of devices running Android has resulted in a rise in the complexity and variety of Android malware. Traditional signature-based detection techniques find it difficult to keep up with how these dangerous apps are changing. With an emphasis on API call analysis, this study presents an approach that combines machine learning and swarm intelligence to improve the detection accuracy of Android malware. Through the utilization of a swarm's collective decision-making power, this methodology maximizes the identification of relevant API calls linked to malware activity. The enhanced feature selection that follows helps to strengthen and improve malware detection systems, therefore tackling the urgent security issues brought on by contemporary Android threats.

Keywords — Gradient Boosting (GB), Support Vector Machines (SVM), and Logistic Regression (LR), API, CIA, KNN

I. INTRODUCTION

Information, networks, and system security are significantly at danger from malware. The proliferation of malware originating from diverse sources presents a noteworthy peril to the confidentiality, integrity, and availability (CIA) of persons and organizations alike. Defense-in-depth strategies must include both malware identification and mitigation [1]. Malware detection is accomplished using a range of techniques, such as static, dynamic, and hybrid approaches, depending on the particular platforms being used. One prominent source of malware is the mobile platform, particularly the popular Android platform. Desktop PCs are at serious danger from Windows viruses [2]. Researchers have been hard at work developing and putting into practice incredibly efficient methods for malware identification. This study looks at a number of malware detection solutions for Windows and Android and makes creative recommendations for improving the efficacy of malware detection systems for Windows Portable Executable (PE) and Android [3].

According to a poll that was just carried out, Android has become the most popular platform worldwide. Andy Rubin is largely acknowledged as the person who earned the moniker "Father of Android" because of his work on the "Camera" project, which outperformed the Symbian operating system. Google has been in charge of Android since August 2005, when Rubin turned over the reigns [4]. The Linux kernel serves as the foundation for the Android operating system (OS), which is distinguished by its open-source design. It is made especially to meet the requirements of touchscreen gadgets, namely tablets and smartphones. 2008 saw the release of the HTC Dream, which was the first Android device. According to the International Data Corporation (IDC) [5], 286 million smartphones were shipped to the global market in 2022.

Mobile smartphones have become an essential part of peoples' everyday lives. Users of these devices can take advantage of a number of digital services, such as email, social media sites like Facebook, online banking, and the ability to tweet [6]. Users are favoring mobile devices over home PCs due to the widespread availability of sophisticated mobile applications that come with features like GPS and maps. However, there are risks involved with using mobile devices, even with its convenience. Sensitive information such as credit card numbers, contact lists, and passwords are commonly stored on mobile devices, making them a desirable target for hackers attempting to gain unauthorized access. Unfortunately, it is necessary to provide adequate consideration to the security threats related to mobile devices, which means that hackers' activities must be given more attention [7].

A. Android Operating System

One of the most widely used mobile operating systems for smartphones and tablets is the Android Operating System (OS). The software in issue is distinguished by its Linux kernel basis and open-source nature [8]. It has a web browser, a Graphical User Interface (GUI), and end-user software download functionality. While the initial Android demos showed off the operating system's compatibility with QWERTY cellphones with large VGA displays, the core operating system was designed with low-cost mobile devices with conventional numeric keypads in mind [9]. A variety of operating system versions designed for different hardware platforms—such as gaming consoles and digital cameras—can be used with the

Apache v2 open-source license. Google Maps, YouTube, Chrome, Gmail, and other proprietary apps are pre-installed on most Android devices [10].

Fig. 1 depicts the android architecture, which is made up of a stack of software components assembled to fulfill the standards of mobile devices. An application, a runtime, a set of C/C++ libraries accessed through application framework services, and a Linux kernel make up the Android software stack [11].

The Android platform's central component, the Linux Kernel, gives mobile devices access to critical operating system functions, while the Dalvik Virtual Machine (DVM) is in charge of overseeing the execution of mobile apps.

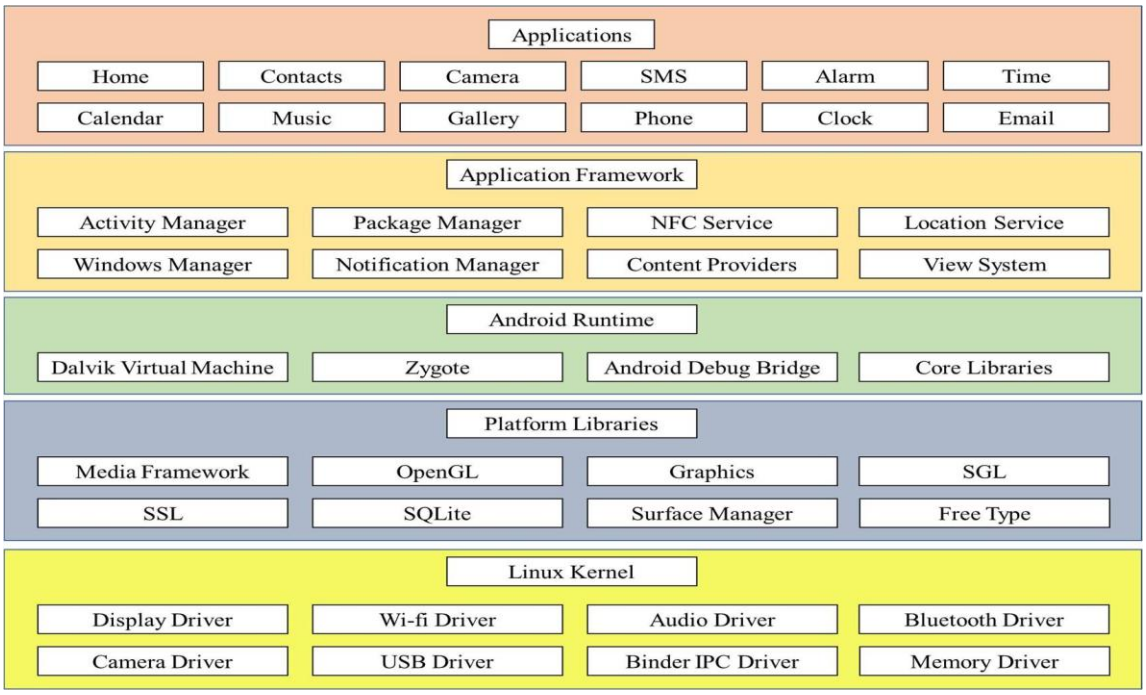
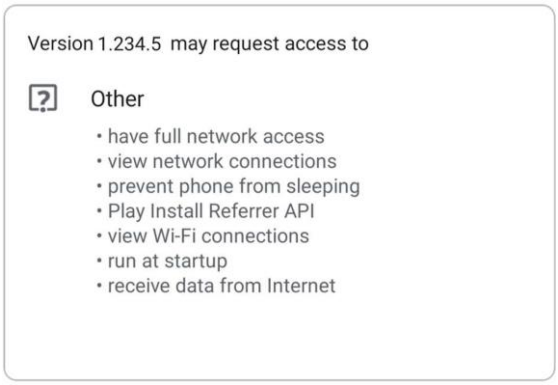


Fig. 1 - Android Architecture

If your program has install-time rights, it can access limited data within a particular bound or do activities that will not significantly harm the system or other apps [12]. The process by which an application store notifies a user for permission when they interact with an app's details page and install-time permissions are included in the app's declaration is illustrated in Fig. 2 The system instantly grants the app the necessary rights after installation [13].

Fig. 2 - Install-Time Permission



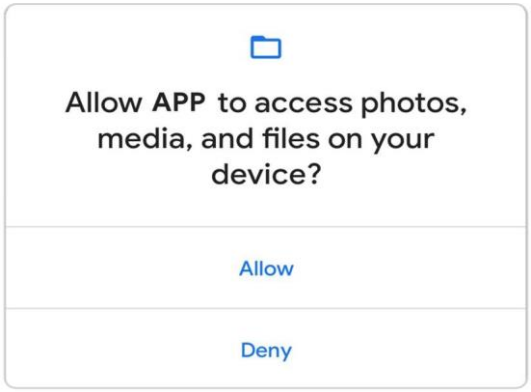
Standard permissions and signature permissions are two of the install-time permission subcategories that the Android operating system includes.

Runtime permissions, also known as hazardous permissions, give your program more authority to view restricted information or carry out prohibited actions that might seriously harm the system and other apps. Runtime rights must be requested from inside the program in order to access restricted data or carry out restricted activities. Prior to every access, it is essential to confirm and, if needed, get the relevant permissions [14]. Assuming that such permissions have already been given is not a prudent move. When an application requests runtime permission, the system responds with a runtime permission prompt, as seen in Fig. 3 There are several runtime rights that have the ability to access private user data, which is a different kind of restricted data that could contain sensitive data. Private user data includes a variety of personal data, including location and contact information [15].

Fig. 3 - Runtime Permission

II. METHODOLOGY

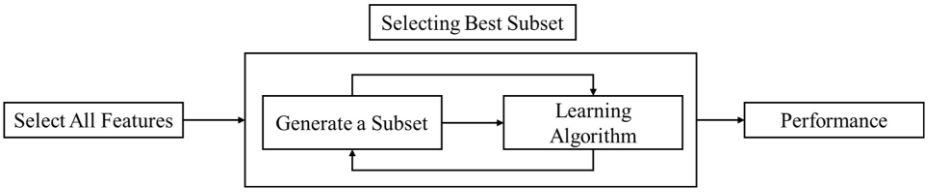
As seen in Fig. 4, characteristics are chosen for wrapper features according to search criteria, and their efficacy



is assessed by grouping, analyzing, and contrasting different feature combinations. By using clustering techniques, wrapper approaches may be used to find possible interactions between variables, which will enhance feature selection [16].

By choosing the most ideal characteristics, the feature selection algorithm in the wrapper approach functions as a wrapper around the predictive model algorithm. This method produces better outcomes, despite its high processing costs and vulnerability to overfitting. Typical techniques for machine learning feature selection

Fig. 4 - Feature Selection Using Wrapper Method



comprise a number of other strategies, such as Boruta feature selection, recursive feature removal, forward feature selection, and backward feature elimination [17].

A. **Forward Selection:** The model is started with no attributes using the incremental procedure. The method of repeatedly improving the model entails adding the feature that results in the biggest boost in performance until adding a new variable no longer produces an increase in performance [18].
B. **Backward Elimination:** To improve the model's performance, we start the procedure with all the features present and then remove those that are thought to be less significant. Until the previously indicated characteristics are gone and no appreciable improvement or advancement is seen, the previously described procedure is repeated repeatedly [19].
C. **Recursive Feature Elimination (RFE):** The goal of this greedy optimization method is to find the feature subset with the best performance. Iteratively, the procedure creates models, ignoring the feature that performs the best or worse on each iteration. The leftover features are used to create the model after all of the accessible features have been used. The attributes are then listed in a hierarchical order according to how excluded they are [20].

An optimization problem requires the methodical selection of input values from a feasible set in order to optimize a particular function. This is followed by the computation of the associated function value. One notable area in applied mathematics is the extension of optimization theory and techniques to new formulations [21]. More broadly, optimization is the process of figuring out what values of an objective function are ideal within a given domain (or input). There is a wide range of target functions and areas where this approach can be applied.

To find the best solution, traditional optimization techniques for differentiable and continuous functions can be applied. To determine the ideal locations, these analytical techniques rely on differential procedures, which rely on differential calculus. The challenges posed by objective functions without continuity and/or differentiability limit the efficiency of traditional optimization techniques in real-world situations [22]. It is important to remember, nonetheless, that most numerical optimization techniques are derived from an analysis of calculus problem optimization techniques. The best solution for a single variable function, a multivariable function without constraints, a multivariable function with equality constraints, and a multivariable function with inequality constraints must be found using complex techniques [23]. Computational optimization is the study and solution of complicated optimization problems that might be discrete or continuous. The use of robust optimization strategies, interior-point approaches, and handling optimization issues in the face of uncertainty are all highly valued in this field of research [24]. The research includes fundamental methodological questions and real-world applications in a number of fields, including manufacturing, financial engineering, labor planning, healthcare systems, terminal operations, and weather forecasting [25]. Some of the current fields of research include the development of algebraic modeling software for mathematical programming, machine learning kernel approaches, network algorithms, and mathematical decomposition algorithms for large-scale decision issues.

Scientists and engineers have paid close attention to nature since it is a great source of inspiration for creating intelligent systems and algorithms. Careful observation reveals that many phenomena in our immediate surroundings are examples of optimization, including biological and various other systems. Because the natural world is infinite, it is conceivable to come up with an infinite number of computer optimization strategies that are inspired by different natural events and techniques. Recent studies have connected metaphors like ant colony optimization, simulated annealing, and genetic algorithms to different metaheuristic optimization techniques. Teaching-Learning-Based Optimization (TLBO), Swarm Optimization, Bat Optimization Method, Cuckoo Optimization Method, Honey Bee Algorithm, Firefly Optimization, Search Strategy in Harmony, Optimization of Water Evaporation, and Passing the Search Algorithm for Vehicles are just a few of the optimization algorithms that are examined in this study [26]. In the field of optimization, these algorithms are frequently employed to raise the efficacy and efficiency of search tactics. Several examples of metaheuristic optimization algorithms are the Fruit Fly Algorithm, Runner-Root Algorithm (RRA), Intelligent Water Drops (IWD) Algorithm, Tabu Search, Crow Search Algorithm, and Imperialist Competitive Algorithm.

The field of metaheuristic optimization revolves around the utilisation of metaheuristic methods to tackle optimization problems. The notion of optimization is widely applied in a variety of industries, including Internet routing, engineering, economics, and vacation scheduling. Because of the intrinsic limits of time, money, and resources, it is essential to make the most of what is available [27]. Most optimization problems that arise in real-world circumstances include several modes and nonlinearities, often combined with intricate restrictions. Conflicts arise frequently from divergent goals, when the best course of action could be exclusive to one goal. Finding an ideal or even less-than-ideal solution might provide serious challenges.

A relatively new family of optimization methods that use metaphors is called metaheuristic optimization techniques. In order to incorporate probabilistic methodologies, metaheuristic algorithms often use random numbers at different stages of the optimization cycle. The term "metaheuristic optimization" is frequently employed to characterize more recent metaheuristic optimization techniques, including simulated annealing, ant colony optimization, and genetic algorithms (GAs). These methods depart from conventional methods for nonlinear programming [28]. While approaches like simulated annealing, ant colony optimization, and genetic algorithms have shown convergence, the convergence of the most recent metaheuristic optimization techniques is yet unknown. While various approaches may have different names, some academics contend that the fundamental ideas behind the most recent metaheuristic optimization techniques don't change [29].

III. CODING AND IMPLEMENTATION

The Python programming language and Python Library were utilized in the Jupiter Notebook experiments on the machine learning techniques discussed in this work. It is compatible with NumPy, pandas, matplotlib, seaborn, and scipy, which are Python scientific and numerical libraries. It contains several methods like as K-NN, Naïve Bayes, and support vector machines for classification, regression, and clustering [29].

- Numpy :Base n-dimensional array package.
- Pandas : Data structures and analysis.
- Matplotlib : Comprehensive 2D/3D plotting.
- IPython :Enhanced interactive console
- Scipy : Fundamental library for scientific computing.
- Sympy : Symbolic mathematics.

In [ ]:
```
# Importing modules
import time
import numpy as np
import pandas as pd
from math import sqrt
from sklearn import metrics
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
```

In [ ]:
```
data  = pd.read_csv('StaticLayer_Permission_Bening&malware_Smaples.csv')
data  = data.values
feat  = np.asarray(data[:, 0:-1])
label = np.asarray(data[:, -1])

xtrain, xtest, ytrain, ytest = train_test_split(feat, label, test_size=0.2, stratify=label)
```

In [ ]:
```
# Define an objective function:
def Fx(solution):

  # Setting parameters
  alpha     = 0.99
  beta      = 1 - alpha

  fs = np.where(solution > 0.5)[0]
```

In [ ]:
```
!pip install mealpy==1.2.2 --quiet
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 308.3/308.3 kB 11.1 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.0/13.0 MB 111.1 MB/s eta 0:00:0000:0100:01
```

In [ ]:
```
# Importing modules
import time
import numpy as np
import pandas as pd
from math import sqrt
from sklearn import metrics
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
```

In [ ]:
```
data  = pd.read_csv('StaticLayer_Permission_Bening&malware_Smaples.csv')
data  = data.values
feat  = np.asarray(data[:, 0:-1])
label = np.asarray(data[:, -1])

xtrain, xtest, ytrain, ytest = train_test_split(feat, label, test_size=0.2, stratify=label)
```

In [ ]:
```
# Metaheuristic Whale Optimization & Firefly Optimization Algorithm
from mealpy.swarm_based.WOA import BaseWOA
from mealpy.swarm_based.ALO import BaseALO

# Setting parameters
ub1     = [1] * 4115    # Upper bound
lb1     = [0] * 4115    # Lower bound
obj_func = Fx       # This objective function come from "opfunu" library. You can design your own objective function like above
verbose = True     # Print out the training results
epoch = 50         # Number of iterations / generations / epochs
pop_size = 50       # Populations size (Number of individuals / Number of solutions)

md1 = BaseALO(obj_func, lb1, ub1, verbose, epoch, pop_size)
# md1 = BaseWOA(obj_func, lb1, ub1, verbose, epoch, pop_size)

start = time.time()
best_pos1, best_fit1, list_loss1 = md1.train()
end = time.time()
print("Time taken for feature reduction: ",end-start," Sec.")

rf = np.where(best_pos1 != 0.0)[0]
print('Reduced feature set size: ', len(rf))
print('Reduced features: ', rf)

np.savetxt(r'/content/drive/MyDrive/Colab_Notebooks/MCA_AMD/50A_50I_ALO_RF_C.txt', rf, fmt="%d", delimiter=',')

# Training classifier with reduced feature set
num_train = np.size(xtrain, 0)
num_valid = np.size(xtest, 0)
x_train   = xtrain[:, rf]
y_train   = ytrain.reshape(num_train)  # Solve bug
```

```
# Define an objective function:
def Fx(solution):

  # Setting parameters
  alpha     = 0.99
  beta      = 1 - alpha

  fs = np.where(solution > 0.5)[0]
  if (len(fs) == 0):
    return 1

  num_train = np.size(xtrain, 0)
  num_valid = np.size(xtest, 0)
  x_train   = xtrain[:, fs]
  y_train   = ytrain.reshape(num_train)  # Solve bug
  x_valid   = xtest[:, fs]
  y_valid   = ytest.reshape(num_valid)  # Solve bug

  # clf = DecisionTreeClassifier(random_state=0).fit(x_train, y_train) # Random Forest Classifier
  # clf = KNeighborsClassifier(n_neighbors = 5).fit(x_train, y_train)
  # clf = LogisticRegression().fit(x_train, y_train)
  # clf = SVC(kernel='linear', probability=True).fit(x_train, y_train)
  # clf = GradientBoostingClassifier(random_state=0).fit(x_train, y_train)
  clf = RandomForestClassifier(random_state=0).fit(x_train, y_train)
  # clf = NearestCentroid().fit(x_train, y_train)


  predictions = clf.predict(x_valid)

  acc       = np.sum(y_valid == predictions)  / num_valid
  cost      = (alpha * acc) + beta * (len(fs) / len(solution))

  return cost
```

In [ ]:
```
# Metaheuristic Whale Optimization & Firefly Optimization Algorithm
from mealpy.swarm_based.WOA import BaseWOA
from mealpy.swarm_based.ALO import BaseALO

# Setting parameters
ub1     = [1] * 4115    # Upper bound
lb1     = [0] * 4115    # Lower bound
obj_func = Fx       # This objective function come from "opfunu" library. You can design your own objective function like above
verbose = True     # Print out the training results
epoch = 50         # Number of iterations / generations / epochs
pop_size = 50       # Populations size (Number of individuals / Number of solutions)

md1 = BaseALO(obj_func, lb1, ub1, verbose, epoch, pop_size)
# md1 = BaseWOA(obj_func, lb1, ub1, verbose, epoch, pop_size)

start = time.time()
best_pos1, best_fit1, list_loss1 = md1.train()
end = time.time()
print("Time taken for feature reduction: ",end-start," Sec.")

rf = np.where(best_pos1 != 0.0)[0]
print('Reduced feature set size: ', len(rf))
print('Reduced features: ', rf)

np.savetxt(r'/content/drive/MyDrive/Colab_Notebooks/MCA_AMD/50A_50I_ALO_RF_C.txt', rf, fmt="%d", delimiter=',')

# Training classifier with reduced feature set
num_train = np.size(xtrain, 0)
num_valid = np.size(xtest, 0)
x_train   = xtrain[:, rf]
y_train   = ytrain.reshape(num_train)  # Solve bug
```

```
# Training classifier with reduced feature set
num_train = np.size(xtrain, 0)
num_valid = np.size(xtest, 0)
x_train   = xtrain[:, rf]
y_train   = ytrain.reshape(num_train)  # Solve bug
x_valid   = xtest[:, rf]
y_valid   = ytest.reshape(num_valid)  # Solve bug

# clf = DecisionTreeClassifier(random_state=0).fit(x_train, y_train)
# clf = KNeighborsClassifier(n_neighbors = 5).fit(x_train, y_train)
# clf = LogisticRegression().fit(x_train, y_train)
# clf = SVC(kernel='linear', probability=True).fit(x_train, y_train)
# clf = GradientBoostingClassifier(random_state=0).fit(x_train, y_train)
clf = RandomForestClassifier(random_state=0).fit(x_train, y_train)
# clf = NearestCentroid().fit(x_train, y_train)

predictions = clf.predict(x_valid)

precisionScore = metrics.precision_score(y_valid, predictions)
recallScore    = metrics.recall_score(y_valid, predictions)
f1Score        = metrics.f1_score(y_valid, predictions)
accuracyScore  = metrics.accuracy_score(y_valid, predictions)
meanSqrError   = metrics.mean_squared_error(y_valid, predictions)
rMeanSqrError  = sqrt(metrics.mean_squared_error(y_valid, predictions))

print('Precision: ', precisionScore)
print('Recall: ', recallScore)
print('F1-Score: ', f1Score)
print('Accuracy: ', accuracyScore)
print('MSR: ', meanSqrError)
print('RMSR: ', rMeanSqrError)
```

```
> Epoch: 38, Best fit: 0.726226337620983
> Epoch: 39, Best fit: 0.726226337620983
> Epoch: 40, Best fit: 0.726226337620983
> Epoch: 41, Best fit: 0.726226337620983
> Epoch: 42, Best fit: 0.726226337620983
> Epoch: 43, Best fit: 0.726226337620983
> Epoch: 44, Best fit: 0.726226337620983
> Epoch: 45, Best fit: 0.726226337620983
> Epoch: 46, Best fit: 0.726226337620983
> Epoch: 47, Best fit: 0.726226337620983
> Epoch: 48, Best fit: 0.726226337620983
> Epoch: 49, Best fit: 0.726226337620983
> Epoch: 50, Best fit: 0.726226337620983
Time taken for feature reduction:  610.9308748245239  Sec.
Reduced feature set size:  3336
Reduced features:  [   0    1    2 ... 4112 4113 4114]
Precision:  0.9285714285714286
Recall:  0.8024691358024691
F1-Score:  0.8609271523178809
Accuracy:  0.9341692789968652
MSR:  0.06583072100313479
RMSR:  0.2565749812494094
```

```
In [ ]:
```

IV. RESULTS AND DISCUSSIONS

One popular method for evaluating a machine learning system is to split the dataset into three separate sets: the test dataset (15%), the validation dataset (20%), and the training dataset (65%). To make sure that each set is reflective of the whole data distribution, the dataset must be randomly divided before being divided into smaller sets. Furthermore, in order to minimize any kind of data leakage that might provide skewed findings, it is imperative to refrain from including the validation and test datasets into the training dataset.

The machine learning model is trained using the training dataset, which enables it to identify patterns and correlations in the data. The model's performance is then assessed using the validation dataset, and any necessary hyperparameter adjustments, such changing the regularization strength or learning rate, are made. By doing this, overfitting is prevented, a condition in which a model performs well on training data but is unable to generalize to new, untested data. Ultimately, the test dataset functions as a stand-alone collection to evaluate the overall performance of the model following training and validation. As seen in fig. 5, this final evaluation offers an objective assessment of the model's expected performance on fresh, untested data in real-world circumstances.

Researchers and practitioners may make sure their machine learning models are thoroughly tested and have good cross-validation and new data generalization capabilities by using this strategy and keeping a distinct division between the training, validation, and test datasets.

Fig. 5- Partition of dataset

The model will be trained when the datasets have been defined. Following training, the model is assessed using the validation dataset. This is iterative and may accommodate any modifications or adjustments required for a model depending on outcomes that can be carried out and reassessed. By doing this, it is ensured that the test dataset is not wasted and may be utilized to test an assessed model.

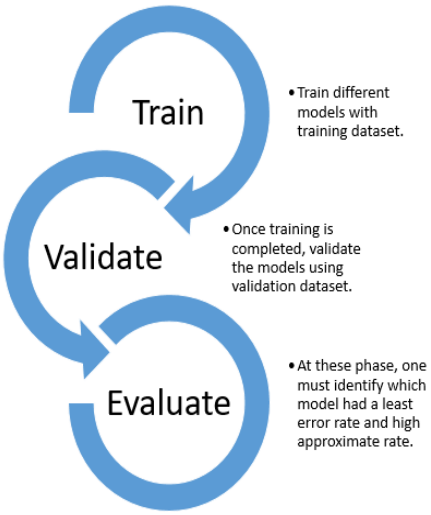


Fig. 6- Iterative process to evaluate machine learning model

Following the completion of the model evaluation, the model with the lowest error rate and highest accuracy in approximating outcomes is chosen for additional testing with a different test dataset. This phase is essential to guarantee that the model's performance stays stable and is consistent with the findings from the validation dataset, as Fig. 6 illustrates.

Verifying that the chosen model's performance with the test dataset is still robust and equivalent to its performance with the validation dataset is the main goal of testing it. This procedure acts as a last check to validate the extension of the model to new, untested data and its dependability.

It is essential to remember that, should the model demonstrate high accuracy in the testing phase, it is crucial to confirm that the test and validation datasets were not accidentally incorporated into the training dataset. If these datasets are accidentally leaked during training, the model's performance measures may be falsely inflated, which would provide a false impression of the model's actual predictive power. As a result, as seen in fig. 7, maintaining rigorous separation between training, validation, and test datasets is essential for getting trustworthy and dependable results from the machine learning model.

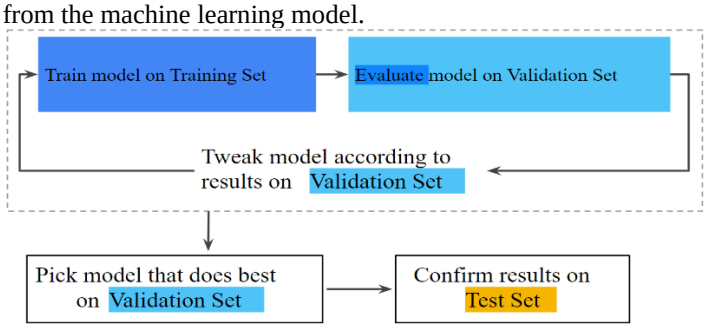


Fig .7-An iterative workflow of training, evaluating, and testing of ML models

A. Performance Comparison

Six classification algorithms were used: Decision Trees (DT), Random Forest (RF), k-Nearest Neighbors (KNN), Gradient Boosting (GB), Support Vector Machines (SVM), and Logistic Regression (LR) to evaluate the models' classification performance using the ALO and WOA wrapper-based feature selection algorithms. Table 1 presents the detailed results of the trials carried out for each of these methods, and Fig. 8 provides the related visual representations. The ideal hyperparameters for the ALO and WOA algorithms are also included in Table 2, which was essential in obtaining the best outcomes in all of the tests. This extensive assessment makes it possible to fully comprehend how various classification algorithms function in conjunction with the ALO and WOA feature selection methods, offering insightful information for additional research and comparison.

TABLE I

Hyper parameters of ALO & WOA

WOA	ALO
epsilon = 0.001	epsilon = 0.001
beta = 0.95	beta = 0.95
threshold = 0.5	alpha = 0.6
no_whales = 10	no_ants = 10
no_iter = 10	no_iter = 10

TABLE 2
Accuracy comparison of DT, RF, KNN, GB, SVM & LR with ALO & WOA

S. No	Classifier	Feature Selection Method	Acc. Before Feature Selection	Acc. After Feature Selection	% Change in Accuracy	Features Selected	% Decrease in Features	Time Taken in Sec.
1	DT	WOA	85.01%	85.89%	1.04%	613	85.10%	17.61
2		ALO		91.54%	7.67%	3709	9.87%	235.98
3	GB	WOA	81.50%	83.29%	2.20%	362	91.20%	864.84
4		ALO		88.40%	8.46%	3336	18.93%	832.42
5	KNN	WOA	81.28%	83.70%	2.97%	1332	67.63%	102.03
6		ALO		88.28%	8.61%	4000	2.79%	358.85
7	LR	WOA	80.03%	82.45%	3.02%	609	85.20%	170.63
8		ALO		84.34%	5.39%	3102	24.62%	355.67
9	**RF**	**WOA**	**86.45%**	**94.04%**	**8.79%**	**73**	**98.23%**	371.86
10		ALO		91.42%	5.75%	659	83.99%	610.93
11	SVM	WOA	82.40%	86.83%	5.38%	1357	67.02%	1727.1
12		ALO		81.50%	1.09%	1659	59.68%	1170.5

The wrapper-based ant lion optimized feature selection method effectively shrunk the feature search space, according to an analysis of the experiment findings. By using the suggested deep neural classifier, this decrease was made without sacrificing the classification accuracy. Fig. 9 provides a thorough description of the evaluation metrics for the Whale Optimization Algorithm (WOA) using the Random Forest classifier. These measures offer a thorough grasp of the WOA's performance when used in tandem with the Random Forest classifier, illuminating how well it manages feature selection to preserve classification accuracy.

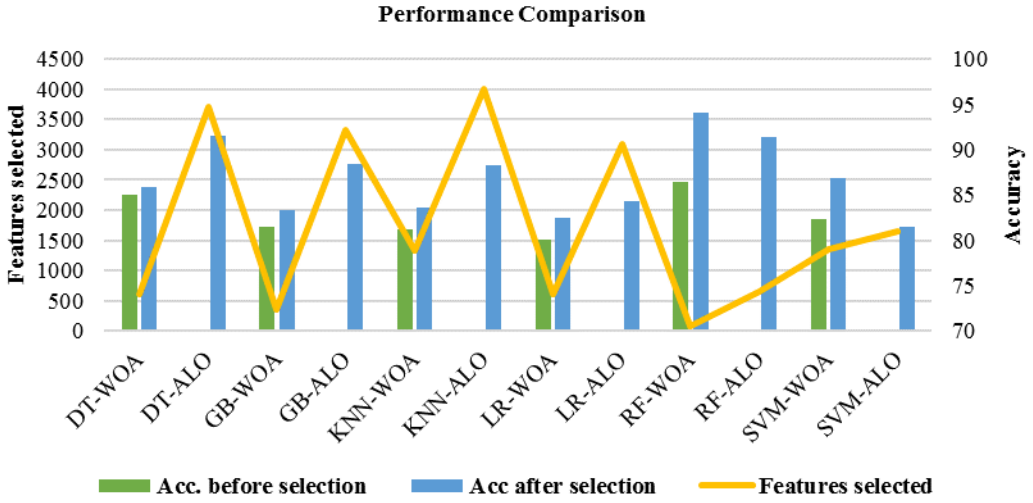


Fig. 8- Performance comparison of six different classifiers with ALO & WOA

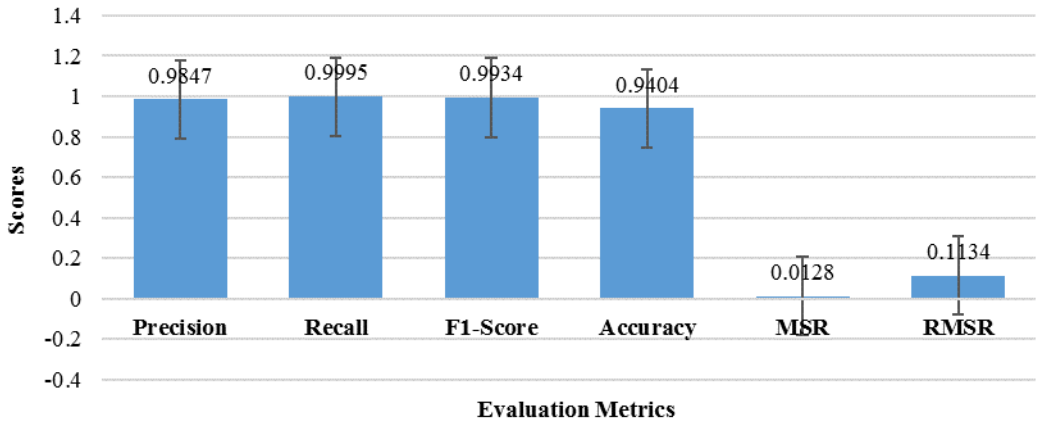


Fig. 9-Evaluation metrics of WOA with Random Forest

V. CONCLUSION

Smartphones are becoming a necessary component of everyday life and are used for a wide range of tasks. The security of Android smartphones has grown increasingly important due to the growing reliance on them. As a result of the dynamic Android operating system's constant introduction of new features and APIs, machine learning-based approaches have become indispensable instruments in the detection of Android malware.

The sophisticated feature selection method used in this research study makes API calls to differentiate between Android apps that are harmful and those that are not. Wrapper-based swarm intelligence algorithms, including Ant Lion Optimizer (ALO) and Whale Optimization Algorithm (WOA), are utilized to optimize this feature selection procedure. Then, to classify apps as either malicious or benign, the chosen features are combined with a variety of classification algorithms, such as Decision Trees (DT), Random Forest (RF), K-Nearest Neighbors (KNN), Gradient Boosting (GB), Support Vector Machines (SVM), and Logistic Regression (LR).

The Random Forest classifier performs the best out of all the classifiers evaluated, with accuracy of 94.04% with WOA and 91.42% with ALO. This classifier performs noticeably better than the others, with noticeably higher accuracy. Notably, the wrapper-based WOA swarm intelligence optimizer is able to achieve a 98% decrease in the search space dimensionality by successfully lowering the number of features from 4115 to 73. The Random Forest classifier's increased accuracy is mostly due to this feature reduction.

To sum up, this research study offers an enhanced method for detecting Android malware. It selects features using wrapper-based swarm intelligence techniques and achieves impressive outcomes with the Random Forest classifier. The results of the study demonstrate how these methods may be used to enhance the precision and effectiveness of Android malware detection systems while taking into account the dynamic nature of threats in today's smartphone environment.

VI. FUTURE WORK

In order to increase the effectiveness of detecting and categorizing Android malware, research is concentrated on creating and utilizing hybrid architectures that include cutting-edge deep learning algorithms. The purpose of these hybrid designs is to improve the precision and speed of malware detection on Android devices by utilizing the advantages of several deep learning techniques. In addition, future research will examine the application of different optimization strategies to systematically lower the feature count in high-dimensional feature spaces. The objective of this endeavor is to enhance the detection system's overall effectiveness and expedite the process of identifying malware. The project aims to improve the state-of-the-art in Android malware detection and contribute to the creation of more reliable and efficient cybersecurity solutions for mobile devices by fusing deep learning with optimization techniques.

REFERENCES

[1] Dhalaria and Gandotra, "A Framework for Detection of Android Malware using Static Features," in 2020 IEEE 17th *India Council International Conference (INDICON)*, New Delhi, India, 2020, pp. 1-7.

[2] Jiang, Baolei, Guan, and Huang, "Android Malware Detection Using Fine-Grained Features," *Hindawi*, vol. 2020, Jan 2020.

[3] Jung, Kim, Shin, Lee, Hyunjae, Cho, Kyoungwon, "Android Malware Detection Based on Useful API Calls and Machine Learning," in 2018 *IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, Laguna Hills, CA, USA, pp. 175-178, 2018.

[4] Jayakrishna, M., Selvakumar, V., Kumar, A., Dilip, S. M., & Maaliw, R. R. (2023, February). Multi-scale Memory Residual Network Based Deep Learning Model for Network Traffic Anomaly Detection. In International Conference on Intelligent

Computing and Networking (pp. 475-482). Singapore: Springer Nature Singapore.

[5] R. B. Hadiprakoso, I. K. S. Buana, and Y. R. Pramadi, "Android Malware Detection Using Hybrid-Based Analysis & Deep Neural Network," in 2020 3rd *International Conference on Information and Communications Technology (ICOIACT)*, Yogyakarta, Indonesia, pp. 252-256, 2021.

[6] Q. Han, V. S. Subrahmanian and Y. Xiong, "Android Malware Detection," IEEE Transactions on Information Forensics and Security, vol. 15, pp. 3511-3525, 2020.

[7] A. D. Lorenzoa, F. Martinellib, E. Medveta, F. Mercaldobc and A. Santone, "Visualizing the outcome of dynamic analysis of Android malware with VizMal," Journal of Information Security and Applications, vol. 50, pp. 1-8, 2020.

[8] J. Xu, Y. Li, R.Deng and K. Xu, "SDAC: A slow-gging solution for android malware detection using semantic distance based API clustering," IEEE Transactions on Dependable and Secure Computing, vol. 19, no. 3, pp. 1-15, 2020.

[9] Jawarneh, M., Jayakrishna, M., Davuluri, S. K., Ramanan, S. V., Singh, P. P., & Joseph, J. A. (2023, February). Energy Efficient Lightweight Scheme to Identify Selective Forwarding Attack on Wireless Sensor Networks. In International Conference on Intelligent Computing and Networking (pp. 425-436). Singapore: Springer Nature Singapore.

[10] H. Hasan, B. T. Ladani and B. Zamani, "MEGDroid: A model-driven event generation framework for dynamic android malware," Information and Software Technology, vol. 135, no. 106569, pp. 1-16, 2021.

[11] X. Liu, X. Du, Q. Lei and K. Liu, "Multifamily Classification of android malware with a fuzzy strategy to resist polymorphic familial variants," IEEE Access, vol. 8, pp. 156900-156914, 2020.

[12] S. I. Hani and N. M. Sahib, "Detection of malware under android mobile application," in 2020 3rd International Conference on Engineering Technology and its Applications, pp. 179-184, 2020.

[13] J. Jiang, S. Li, M. Yu, G. Li, C. Liu et al., "Android malware family classification based on sensitive opcode," in IEEE Symposium on Computers and Communications (ISCC), pp. 1-7, 2019.

[14] W. Wang, Y. T. Li, T. Zou, X. Wang, J. Y. You et al., "A novel image classification approach via Dense-MobileNet models," Mobile Information Systems, https://doi.org/10.1155/2020/7602384, 2020.

[15] N. Daoudi, K. Allix, T. F. Bissyandé and J. J. Klein, "Lessons learnt on reproducibility in machine learning based android malware detection," Empirical Software Engineering, vol. 74, pp. 1-53, 2021.

[16] Z. H. Qaisar and R. R. Li, "Multimodal information fusion for android malware detection using lazy learning," Multimed Tools Appl, vol.81, pp. 12077–12091, 2021.

[17] H. Rathore, S. K. Sahay, P. Nikam and M. Sewak, "Robust android malware detection system against adversarial attacks using q-learning," Information Systems Frontiers, vol. 23, pp. 867-882, 2021.

[18] D. Tehrany and A. Rasoolzadegan, "A new machine learning-based method for android malware detection on imbalanced dataset," Multimed Tools Appl, vol.80, pp. 24533–24554, 2021.

[19] V. P. Dharmalingam and P. Visalakshi, "A novel permission ranking system for android malware detection-the permission grader," Journal of Ambient Intelligence and Humanized Computing, vol. 12, pp. 5071–5081, 2021.

[20] O. Yildiz and I. A. Doğru, "A novel permission-based Android malware detection system using feature selection based on linear regression," International Journal of Software Engineering and Knowledge Engineering, vol. 29, no. 2, pp. 245-262, 2019.

[21] N. A. Sarah, F. Y. Rifat, Md. S. Hossain and H. S. Narman, "An Efficient android malware prediction using ensemble machine learning algorithms," Procedia Computer Science, vol. 191, pp. 184-191, 2021.

[22] O. N. Elayan and A. M. Mustafa, "Android malware detection using deep learning," Procedia Computer Science, vol. 184, pp. 847-852, 2021.

[23] J. M. Arif, M. F. A. Razak, S. R. T.Mat, S. Awang and N. S. N. Ismail, "Android mobile malware detection using fuzzy AHP," Journal of Information Security and Applications, vol. 61, pp. 1-35, 2021.

[24] W. Wang, J. Wei, S. Zhang and X. Luo, "LSCDroid: Malware detection based on local sensitive API invocation sequences," IEEE Transactions on Reliability, vol. 69, no. 1, pp. 174-187, 2019.

[25] H. Gao, S. Cheng and W. Zhang, "GDroid: Android malware detection and classification with graph convolutional network," Computers & Security, vol. 106, no. 102264, pp. 1-14, 2021.

[26] A. A. Taha and S. S. J. Malebary, "Hybrid classification of Android malware based on fuzzy clustering and the gradient boosting machine," Neural Computing and Applications, vol. 33, pp. 6721–6732, 2021.

[27] M. Seyedali, "The Ant Lion Optimizer,"

Advances in Engineering Software, vol. 83, pp. 80-98, 2015.

[28] X. S. Yang and S. Deb, "Cuckoo search via levy flights," in World Congress on Nature & Biologically Inspired Computing (NaBIC), pp.210-214, 2009.

[29] G. Lindfield and J.Penny," Nature-inspired optimization algorithms," Academic Press, pp. 85-100, 2017.