Analysis of ML methods and algorithms for malware detection
(malware type - Trojan horses - detection from traffic flow records in a computer network)

INNOvative SOC/INNOSOC project

Contents

Introduction 3

Experiments and results 8

Datasets 8

Dataset: Trojan_Detection 8

Algorithms and methods used 10

Random forest (Random Forest) 10

AdaBoost and Gradient Boosting Classifier 11

Gaussian Naive Bayes 13

Deep Learning with Feedforward Neural Network 13

Preparation and calculation procedure 14

Settings and parameters of the selected methods 16

Random forest (Random Forest) 16

AdaBoost 16

Gaussian Naive Bayes 17

Deep Learning with Feedforward Neural Network 17

Evaluation of experiments 19

Confusion matrix 19

Accuracy 20

Precision 21

Recall 21

F1-Score 21

Results 22

Dataset: TrojanDetec (calculations on Google Colab - Processing time) 22

2025 12 05: calculations - screenshots 23

Summary 33

Introduction

Malware

The term malware (malicious software) is used to denote any program or code that has been created with the aim of damaging a computer system, for example to gain access to data, computer systems or networks without the user's consent, as well as to obtain financial gain for the creator of the software, disrupt operations, commit sabotage, and the like.

Malware can be classified on the basis of its

- behaviour
- method of propagation
- the objectives it pursues

There are many kinds of malware; the main types include the following.

Viruses

These are small programs that attach themselves to other legitimate files or programs. In order to be activated and spread further, they require user interaction, e.g. opening an infected file (invoice, image, ...) in an e-mail attachment, or clicking on a link in an e-mail message, opening multimedia files and programs downloaded from untrusted websites, connecting an infected USB stick, clicking on a malicious online advertisement, which may also appear on legitimate websites, clicking on a link on social networks, and others.

When the program is run, the virus is also launched without the user's knowledge, and it may infect other programs. The effects of viruses can vary in severity - from annoying messages to serious damage to files or the system. Protection against viruses requires a cautious approach to online activities and keeping the antivirus, the operating system and applications up to date.

Computer worms (worms)

These are standalone programs. Unlike viruses, they are designed to replicate and spread autonomously and do not require direct user interaction. They can exploit security vulnerabilities in software, the operating system or the network (in network protocols, applications and services, for example SMB file sharing. An example of the exploitation of a vulnerability in Microsoft Windows was the WannaCry worm. Another example is e-mail worms, which, after infecting one system, automatically send copies of themselves to the contacts in the user's address book. To spread worms, their creators often use social engineering, through which they try to induce the recipient to open the message.

The rapid spread of worms between systems can cause network capacity to be overwhelmed and lead to the exhaustion of system resources and the slowdown or failure of systems. Worms can also serve as a "carrier" for other malware. We prevent them by installing security patches for systems.

Trojan horses (trojans)

Trojan horses pretend to be legitimate and useful software (e.g. a game, a tool or an installation), but in reality they contain hidden malicious code that can be activated once a certain condition is met (a certain date occurs, e.g. New Year or Christmas).

Once launched, this hidden part of the program usually performs malicious activities, for example

- opening a "backdoor" (backdoor) into the system
- theft or modification of data
- deletion of files
- formatting of the disk
- installation of further malware

If a regular program is infected with a virus, it becomes its spreader and is also a potential Trojan horse.

Ransomware

It encrypts files on the compromised computer and makes them inaccessible. The perpetrator then demands a ransom (ransom) for the decryption key. It is one of the most dangerous and most frequent threats.

Spyware

It is designed to spy on the user's activities without their knowledge. It collects sensitive information such as login credentials, passwords, browsing history or financial data, and sends it to the attacker.

Adware (advertising-supported software)

This is software that automatically displays, plays or downloads advertising material, such as pop-up windows, banners or icons, or redirects the browser. It sometimes serves to finance the development of a program so that it is available to users free of charge. It is often associated with spyware, because many such programs monitor the user's activity without their knowledge and collect their data. Although it is usually not as destructive as other types of malware, it can compromise privacy, spy on the user, and slow down the system and make working with the internet and software unpleasant.

Rootkits (programs hiding malicious applications on a PC)

This is a set of tools that allow the attacker to gain complete control over the system while masking their presence and activities from the operating system and security software.

Keylogger (Keystroke Logger)

This is software or hardware that records keystrokes on the compromised device. The aim is to capture login credentials, credit card numbers and other confidential information.

Detection of Trojan horse malware

For the purposes of our innovative SOC solution, within pilot testing we focused only on Trojan horse malware (trojan) and on the methods of its detection that use network traffic monitoring in a computer network for analysis. First, we describe the ways in which Trojan horses spread and their properties that manifest themselves in network traffic and can be used to detect Trojan horses.

Ways in which Trojan horses spread

Trojan horses can spread in a computer network in various ways; their spread is often aided by social engineering – aimed at deceiving the user – or they exploit system vulnerabilities.

In social engineering, the Trojan horse poses as legitimate software and persuades the user to download and run it voluntarily. This may involve:

- **E-mail attachments and phishing** - one of the most common methods. Attackers send e-mails that appear trustworthy (e.g. as an invoice, a bank notice or a message from a colleague) and prompt the user to open a malicious attachment or click on a link.
- **Downloading software from untrusted sources** - users unknowingly download a Trojan horse when they download pirated software, cracks, games or other programs from unofficial or dubious websites.
- **Social networks and file-sharing services** – here, propagation may also take place via malicious links or files shared via chat applications, social media or peer-to-peer networks.

Examples of exploiting **system vulnerabilities**:

- **Exploitation of browser vulnerabilities (exploits)** - visiting an infected website can automatically trigger the download and installation of a Trojan horse without the user's knowledge, by exploiting security flaws in the web browser or its add-ons.
- **USB drives and removable media** - infected USB sticks can transfer a Trojan horse to another computer when they are connected and automatic execution (autorun) is enabled.

So, unlike viruses and worms, Trojan horses do not spread on their own. They require user interaction (e.g. opening a file, installing a program) or the exploitation of a specific vulnerability in the system in order to be installed and run.

How can a Trojan horse be detected by monitoring communication in a computer network?

Various methods and tools are used to detect Trojan horses in a computer network. Usually, this is a combination of network traffic monitoring, endpoint analysis and adherence to security best practices:

1. Use of security software

- **Antivirus and anti-malware programs** These programs are the basis of protection. They scan files and processes on endpoints and look for known signatures (digital fingerprints) of Trojan horses. Modern programs also use heuristic analysis, which can identify the suspicious behaviour of unknown malware.
- **Intrusion detection and prevention systems (IDS/IPS)** These systems monitor **network traffic** and look for patterns that could indicate the activity of a Trojan horse or an attacker, such as **unusual ports, unauthorized access attempts or communication with known malicious servers** (C&C servers - Command & Control servers - coordinate the actions of infected computers, provide instructions and updates, and may be used to upload stolen data).

2. Network traffic monitoring

- **Flow analysis (NetFlow, sFlow)** Network administrators can monitor the volume and type of data transferred. An unusual **increase in outgoing traffic to unknown destinations** may be a sign of data exfiltration by a Trojan horse.
- **DNS request monitoring** Many Trojan horses communicate **with their command servers via DNS**. Analysis of unusual or frequent requests to suspicious domains can help in their detection.

3. Behavioural and anomaly analysis

- **Behavioural analysis** Behaviour analysis tools (e.g. EDR - Endpoint Detection and Response) learn the normal behaviour patterns of users and applications in the network. Any significant deviation – for example, a program that suddenly starts accessing a large number of sensitive files or attempts to execute system commands – is flagged as suspicious activity.
- **Regular audits and logging** Regular review of system logs (event records) on servers and endpoints can reveal unauthorized configuration changes, the creation of new user accounts or the installation of unauthorized software.

4. Best practices

- **Firewall** A correctly configured firewall restricts unauthorized access to the network
  and blocks communication on non-standard ports, making it more difficult for Trojan horses to communicate with the outside world.
- **User education** Social engineering is a common method of spreading Trojan horses. Training employees to recognize phishing e-mails and dangerous websites significantly reduces the risk of infection.
- **Regular updates** Keeping operating systems and software up to date helps to patch security vulnerabilities that Trojan horses could exploit to infiltrate the network.

By combining these techniques and tools, the chances of early detection and neutralization of Trojan horses in the network can be significantly increased.

Trojan horses can generate network traffic, and we want to detect them on that basis

As stated above, Trojan horses may also manifest themselves by generating suspicious network activity or traffic that may indicate a compromised device.

Examples of such communication generated by Trojan horses are:

Unauthorized communication - a Trojan horse may communicate with a remote command server controlled by the attacker, from which it receives instructions or to which it sends stolen data.

DDoS attacks - some trojans turn the compromised device into a "zombie computer" that is part of a botnet. This network of infected computers can then be used to flood a server with a large volume of traffic – for a DDoS (Distributed Denial of Service) attack.

Downloading further malware – a common type of trojan (downloader trojan) uses the internet connection to download and install further malicious (malicious) programs onto the infected system.

Increased bandwidth usage:

Trojan activities, such as sending a large volume of stolen data or participating in a botnet, can lead to unexpected spikes in network activity and slow down the network connection for the user.

Since the activity of Trojan horses can manifest itself in unusual network traffic, we investigated machine learning models that detect trojans on the basis of network traffic data.

Experiments and results

Datasets

Dataset: Trojan_Detection

The Trojan_Detection dataset was used to train models for detecting "Trojan horse" (trojan horse) malware on the basis of network traffic data:

github.com/Shreekavinmr/Trojan-horse-detection/blob/main/dataset.pdf, www.kaggle.com/datasets/subhajournal/trojan-detection/data – Download.

These data originally come from the web portal of the Canadian Institute for Cybersecurity (CIC) www.unb.ca/cic/datasets/index.html.

The Trojan_Detection dataset has an uncompressed size of 93.89 MB and contains records of flows in a computer network. A packet flow here denotes a series of packets in the communication between two applications - these share the same five-tuple of data – source and destination IP address, source and destination port, and protocol type.

The dataset contains records of two types (malicious – Trojan and benign - Benign):

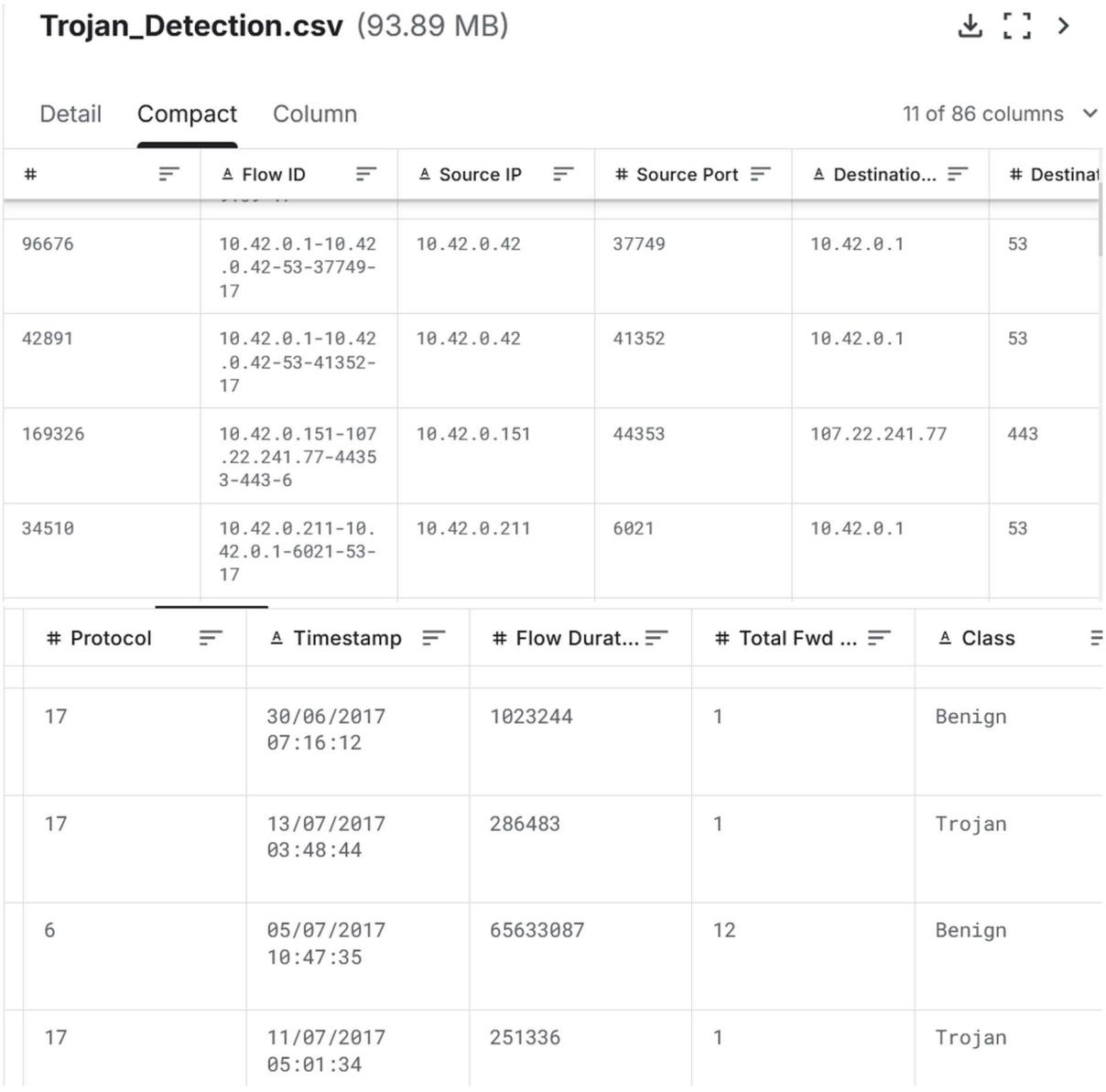
- Trojan Data - 90683 observations – malicious flows generated by Trojan horses
- Benign Data - 86799 observations – flows of benign network traffic

In the dataset, 86 parameters are recorded for each flow:

Flow ID	Source IP	Source Port
Destination IP	Destination Port	Protocol
Timestamp	Flow Duration	Total Fwd Packets
Total Backward Packets	Total Length of Fwd Packets	Total Length of Bwd Packets
Fwd Packet Length Max	Fwd Packet Length Min	Fwd Packet Length Mean
Fwd Packet Length Std	Bwd Packet Length Max	Bwd Packet Length Min
Bwd Packet Length Mean	Bwd Packet Length Std	Flow Bytes/s
Flow Packets/s	Flow IAT Mean	Flow IAT Std
Flow IAT Max	Flow IAT Min	Fwd IAT Total
Fwd IAT Mean	Fwd IAT Std	Fwd IAT Max
Fwd IAT Min	Bwd IAT Total	Bwd IAT Mean
Bwd IAT Std	Bwd IAT Max	Bwd IAT Min
Fwd PSH Flags	Bwd PSH Flags	Fwd URG Flags
Bwd URG Flags	Fwd Header Length	Bwd Header Length
Fwd Packets/s	Bwd Packets/s	Min Packet Length
Max Packet Length	Packet Length Mean	Packet Length Std
Packet Length Variance	FIN Flag Count	SYN Flag Count
RST Flag Count	PSH Flag Count	ACK Flag Count
URG Flag Count	CWE Flag Count	ECE Flag Count

Down/Up Ratio	Average Packet Size	Avg Fwd Segment Size
Avg Bwd Segment Size	Fwd Header Length.1	Fwd Avg Bytes/Bulk
Fwd Avg Packets/Bulk	Fwd Avg Bulk Rate	Bwd Avg Bytes/Bulk
Bwd Avg Packets/Bulk	Bwd Avg Bulk Rate	Subflow Fwd Packets
Subflow Fwd Bytes	Subflow Bwd Packets	Subflow Bwd Bytes
Init_Win_bytes_forward	Init_Win_bytes_backward	act_data_pkt_fwd
min_seg_size_forward	Active Mean	Active Std
Active Max	Active Min	Idle Mean
Idle Std	Idle Max	Idle Min
Class – Trojan/Benign		

The dataset can be viewed at www.kaggle.com/code/proloypal/trojan-detection-cnn-97-accuracy/data (11 of the 86 columns were selected for the figure):



Trojan_Detection.csv (93.89 MB)

Detail | Compact | Column — 11 of 86 columns

#	Flow ID	Source IP	Source Port	Destinatio...	Destinat
96676	10.42.0.1-10.42.0.42-53-37749-17	10.42.0.42	37749	10.42.0.1	53
42891	10.42.0.1-10.42.0.42-53-41352-17	10.42.0.42	41352	10.42.0.1	53
169326	10.42.0.151-107.22.241.77-44353-443-6	10.42.0.151	44353	107.22.241.77	443
34510	10.42.0.211-10.42.0.1-6021-53-17	10.42.0.211	6021	10.42.0.1	53

# Protocol	Timestamp	# Flow Durat...	# Total Fwd ...	Class
17	30/06/2017 07:16:12	1023244	1	Benign
17	13/07/2017 03:48:44	286483	1	Trojan
6	05/07/2017 10:47:35	65633087	12	Benign
17	11/07/2017 05:01:34	251336	1	Trojan

Algorithms and methods used

Random forest (Random Forest)

The "random forest" algorithm is used to classify unknown objects into classes. For example, it is to decide whether or not a bank client will be able to repay a loan, or whether a packet flow in the network belongs to benign regular traffic or is an unusual - anomalous flow that may be caused by malware.

The random forest has the CART method build a chosen number of decision trees and classifies unknown objects by having these trees vote on which class the individual objects belong to.

The algorithm is based on training data that contain information about objects - parameters (predictors) and class membership (the predicted variable). The parameters may be, for example, the client's age, income, number of children, nationality, or in the case of network traffic – the source IP address, source port, protocol, and others.

We train the "random forest" model by choosing the number of trees (n_estimators) that the forest is to contain and choosing the number of predictors to be used each time a tree is grown. Then, to generate each tree, a selection of records from the training set is randomly prepared (using the bootstrap method - sampling with replacement), and the branching of the tree is repeated in the following steps:

- from the set of predictors, we randomly select the required number of predictors
- we use the CART method (see below) to find the best decision rule for splitting the selected records
- the construction of the decision tree ends when the stopping criterion is met (maximum tree depth, minimum number of cases in a terminal leaf of the tree, or the leaf contains only elements of the same class)

The CART method used is characterized below.

CART (Classifi-cation And Regression Trees)

The CART algorithm is used to solve the classification task. It belongs to the decision trees described in the chapter Analysis of ML methods and algorithms for the detection of spam e-mail messages. The main idea of CART is to train a binary tree – which at each splitting step selects, from all possible decision classification conditions, the classification condition that is the "best" in terms of the so-called "purity" of the sets obtained by splitting.

We will explain the construction of the classification conditions of the CART algorithm using the example of bank clients about whom we know the so-called predictors - number of children (0, 1, 2, 3) and nationality (Slovak, Czech and Hungarian).

Furthermore , we know whether or not they were able to repay a loan, so we have two classes that we want to predict for new loan applicants.

Here we have two kinds of predictors, each of which allows several branching conditions to be constructed:

- ordinal variables – number of children (0, 1, 2, 3) – here we can construct the branching conditions "number of children" < 1, < 2, < 3, 3
- categorical variables – Slovak, Czech, Hungarian – here the possible branchings are
    - Slovak – (Czech, Hungarian), Czech – (Hungarian, Slovak), Hungarian – (Slovak, Czech)

Then, at each step where the CART method decides on a split, all splitting conditions are examined and the “best” condition is selected.

To determine the “best” splitting condition, the “Gini index” or “entropy” functions are most commonly used; these criteria measure the “homogeneity” of the elements of the individual classes in the sets produced by the split. The term “impurity” is also used for this property. The aim is to select the split that reduces the “impurity” as much as possible.

More information can be found in the supporting document:

- KREDITNÝ SKÓRING POMOCOU NÁHODNÝCH LESOV.pdf
- DECISION TREES - CLASSIFICATION AND REGRESSION TREES – CART.pdf

AdaBoost and Gradient Boosting Classifier

AdaBoost (Adaptive Boosting) is a popular machine learning method belonging to the category of ensemble methods (methods combining sets of models). Its main goal is to create a strong classifier by combining several weak classifiers (so-called "weak learners"), which are usually simple models, for example decision trees with a depth of 1.

The principle of the AdaBoost method lies in an adaptive approach, where the models are trained sequentially in a series of steps, with each subsequent model focusing on the errors made by the preceding models.

The AdaBoost method comprises the following steps:

1. Initialization of data (record) weights

At the start, each training sample in the dataset is assigned the same weight. This means that all samples are considered equally important for the first training.

2. Iterative training of weak classifiers

Training takes place in cycles. Each iteration (step) comprises:

Training: A new weak classifier is trained with the aim of minimizing the error on the currently reweighted training data.

Error calculation: The error rate (weighted error) of this new model is calculated.

Classifier weight calculation: A model that achieved a lower error rate receives a higher weight in the final ensemble of models (i.e. its vote will be more important when classifying unknown objects). A model with a high error receives a low weight.

3. Adaptive reweighting of the data

We increase the weights of the samples that the current classifier misclassified. Samples that were classified correctly will have their weight reduced. This weighting of the samples ensures that the next weak classifier concentrates primarily on the samples that were misclassified.

4. Combination (Aggregation)

Once all iterations have been completed, the final strong classifier is created, which is a weighted linear combination of all the trained weak classifiers. In prediction, the predictions of the individual models are summed, with each vote weighted according to its accuracy (the classifier weight from step 2).

Advantages of AdaBoost:

- simplicity - it is easy to implement and often uses simple weak classifiers
- high accuracy – it can achieve very high classification accuracy
- resistance to overfitting - when used with weak classifiers, it usually does not suffer from overfitting as much as other methods.

The AdaBoost method is a robust and efficient algorithm that forms the basis of many more advanced boosting techniques (such as Gradient Boosting Machines or XGBoost) and is frequently used in practice.

Gradient Boosting Classifier is a powerful machine learning algorithm that combines many "weak" models (usually decision trees) into a single "strong" one, progressively correcting the errors of the preceding models in order to achieve accurate predictions in classification tasks. Each new tree learns to predict the errors of the combined model from the previous iterations. The final prediction model is the sum of the predictions of all these weak models. The gradient is used for optimization and for "stepping" towards minimizing the error function, similarly to neural networks.

More information can be found in the supporting document:

- Ensemble methods in general: Bagging - random forests - Boost.pdf.

Gaussian Naive Bayes

The Gaussian Naive Bayes method is based on Bayes' theorem and two key assumptions: the "naive" independence of attributes, and that continuous data have a Gaussian (normal) distribution.

The algorithm uses Bayes' theorem to calculate the conditional probability that a data point belongs to a particular class, given its properties (features) - it tries to determine which class is most likely to generate points with such attributes.

The method assumes that all explanatory variables (features) are mutually independent. This means that the presence of one feature does not affect the presence of any other feature within a given class. This assumption is often a simplification for real data (hence "naive"), but in practice the algorithm nevertheless achieves good results.

For Gaussian Naive Bayes, it is also assumed that the values of each continuous feature are distributed according to a Gaussian (normal) distribution.

- the algorithm estimates the parameters of this distribution (mean and variance) for each feature within each class
- then, during prediction, the probability of observing a specific feature value for a given class is calculated on the basis of these parameters and the probability density.

Classification is performed in the following steps:

1. Parameter estimation: During training, the mean and variance are calculated for each class and each feature.
2. Probability calculation: For a new, unknown data point (record), the probability of its features is calculated for each of the possible classes using the Gaussian density (the probability that, if the object belongs to the given class, its feature takes such a value).
3. Application of Bayes' theorem: These probabilities (of the attribute values) are combined using Bayes' theorem (thanks to the independence assumption, they are multiplied, and the result is also multiplied by the probability of occurrence of the given class) - the overall posterior probability that the data point belongs to each class is calculated.
4. Prediction: The class with the highest calculated probability is determined as the resulting prediction for the given data point.

Deep Learning with Feedforward Neural Network

Deep Learning with Feedforward Neural Network (deep learning with a feedforward neural network) is one of the machine learning methods and is regarded as the next step in the field of neural networks. Deep neural networks have multilayer neuron architectures.

The principle of deep learning with feedforward neural networks is that the input data pass through several layers of neurons, where each layer processes the information and the output of one layer serves as the input for the next. These networks usually consist of an input layer, one or more hidden layers and an output layer. Each neuron in one layer is connected to the neurons in the next layer, and each connection is assigned a weight that is adjusted during training. The network is trained by passing input samples through the network and comparing the output with the expected output. The difference between these values is used to adjust the weights by means of the backpropagation algorithm, which calculates the gradient of the error with respect to each weight. This gradient indicates the direction in which the error increases the most, and the weights are therefore adjusted in the opposite direction, which leads to a decrease in the error. This learning process is repeated iteratively for different data, which is called stochastic gradient descent (SGD - Stochastic Gradient Descent).

If the input data contain noise, the network may also try to learn undesirable incorrect patterns. To prevent overfitting, where the network learns the noise instead of the actual patterns, regularization is used, for example L2 regularization, which limits the size of the weights so that the resulting decision boundary is smoother and more realistic. This method helps the network learn “nice” decision boundaries that are less sensitive to noise in the data.

The advantage of deep networks

is their ability to automatically learn their own patterns from data, which is advantageous for complex tasks such as image classification.

The advantage of deep neural networks with a feedforward architecture is that they are able to automatically learn complex non-linear patterns from data and are effective in many areas, including image classification, malware detection and medical image analysis.

Preparation and calculation procedure

For detection, we used the models written in the “Trojan-horse-detection” script from the portal: https://github.com/Shreekavinmr/Trojan-horse-detection/blob/main/README.md .

Work with the models consisted of the following steps.

1. Data preprocessing

- Loading the dataset for Trojan detection using the pandas library.
- Data cleaning: omitting irrelevant and duplicate records and, where necessary, filling in missing data.

- Label encoding of categorical data: conversion of categorical data such as the flow identifier (Flow ID), source and destination IP address (Source IP, Destination IP) and timestamp (Timestamp) – into numerical values.
- Class Label Encoding: Conversion of the class labels ("Trojan" and "Benign") into binary form (1 and 0, respectively).
- Feature Selection: Use mutual information to select the features that are most important for classification

2. Splitting the data into training and test data

Splitting the data into a training set and a test set (typically 80% training and 20% test data). The function using train_test_split from the scikit-learn library was used for the split.

3. Calculations using machine learning algorithms: Random Forest, AdaBoost (**a**daptive **b**oosting), Gaussian Naive Bayes and deep learning with a feedforward neural network. A description of the machine learning algorithms used follows.

Random forest (Random Forest)

Training the Random Forest model: Initialization and training of the Random Forest Classifier with a suitable number of estimators (50 trees). Evaluation of the model's accuracy on the test set.

Hyperparameter tuning: Use different numbers of estimators – find the optimal configuration.

AdaBoost

Training the AdaBoost model: Initialize and train the AdaBoost classifier (e.g. with 100 estimators). Evaluate the model's accuracy on the test set.

Gaussian Naive Bayes

Model training: A Gaussian naive Bayes classifier is initialized and trained.

Cross-validation: Cross-validation is applied to obtain a more robust estimate of the models' performance.

Deep Learning with Feedforward Neural Network

Library installation: Install the TensorFlow, scikit-learn and pandas libraries.

Data preprocessing for the neural network: Standardize the input data using StandardScaler. Define the architecture of the feedforward neural network with suitable activation functions and layers.

Neural network training: Compiling and training the neural network on the preprocessed data for a specified number of epochs.

Settings and parameters of the selected methods

Random forest (Random Forest)

number of estimators: 50 trees

hyperparameter tuning: 10, 20, 30, 40, 50, 60, 70, 80, 90 trees

Parameter settings:

```
clf.get_params(deep=True)
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'sqrt',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'monotonic_cst': None,
 'n_estimators': 50,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

Default parameter settings according to

Parameter settings:

scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

AdaBoost

number of estimators: 100 trees

Parameter settings:

```
Aclf = AdaBoostClassifier(n_estimators=100)
Aclf.get_params(deep=True)
{'algorithm': 'deprecated',
 'estimator': None,
 'learning_rate': 1.0,
 'n_estimators': 100,
 'random_state': None}

Gclf = GradientBoostingClassifier(n_estimators=100,
learning_rate=1.0,max_depth=1, random_state=0).fit(X_train, y_train)
Gclf.get_params(deep=True)
```

```
{'ccp_alpha': 0.0,
 'criterion': 'friedman_mse',
 'init': None,
 'learning_rate': 1.0,
 'loss': 'log_loss',
 'max_depth': 1,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_iter_no_change': None,
 'random_state': 0,
 'subsample': 1.0,
 'tol': 0.0001,
 'validation_fraction': 0.1,
 'verbose': 0,
 'warm_start': False}
```

scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

Gaussian Naive Bayes

Model training: A Gaussian naive Bayes classifier is initialized and trained.

Cross-validation: Cross-validation is applied to obtain a more robust estimate of the models' performance.

Parameter settings:

```
gnb = GaussianNB()
gnb.get_params(deep=True)
{'priors': None, 'var_smoothing': 1e-09}
```

https://scikit-learn.org/0.22/modules/generated/sklearn.naive_bayes.GaussianNB.html

Deep Learning with Feedforward Neural Network

Library installation: Install the TensorFlow, scikit-learn and pandas libraries.

Data preprocessing for the neural network: Standardize the input data using StandardScaler. Define the architecture of the feedforward neural network with suitable activation functions and layers.

Neural network training: Compiling and training the neural network on the preprocessed data for a specified number of epochs.

The neural network used has the following architecture:

```
model = tf.keras.Sequential([
```

```
    tf.keras.layers.Dense(64, activation='relu',
input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')  # Output layer with
sigmoid activation for binary classification
])

model.get_config()
{'name': 'sequential',
 'trainable': True,
 'dtype': {'module': 'keras',
  'class_name': 'DTypePolicy',
  'config': {'name': 'float32'},
  'registered_name': None},
 'layers': [{'module': 'keras.layers',
   'class_name': 'InputLayer',
   'config': {'batch_shape': (None, 84),
    'dtype': 'float32',
    'sparse': False,
    'ragged': False,
    'name': 'input_layer'},
   'registered_name': None},
  {'module': 'keras.layers',
   'class_name': 'Dense',
   'config': {'name': 'dense',
    'trainable': True,
    'dtype': {'module': 'keras',
     'class_name': 'DTypePolicy',
     'config': {'name': 'float32'},
     'registered_name': None},
    'units': 64,
    'activation': 'relu',
    'use_bias': True,
    'kernel_initializer': {'module': 'keras.initializers',
     'class_name': 'GlorotUniform',
     'config': {'seed': None},
     'registered_name': None},
    'bias_initializer': {'module': 'keras.initializers',
     'class_name': 'Zeros',
     'config': {},
     'registered_name': None},
    'kernel_regularizer': None,
    'bias_regularizer': None,
    'kernel_constraint': None,
    'bias_constraint': None},
   'registered_name': None,
   'build_config': {'input_shape': (None, 84)}},
  {'module': 'keras.layers',
   'class_name': 'Dense',
   'config': {'name': 'dense_1',
    'trainable': True,
    'dtype': {'module': 'keras',
     'class_name': 'DTypePolicy',
     'config': {'name': 'float32'},
     'registered_name': None},
    'units': 32,
    'activation': 'relu',
    'use_bias': True,
    'kernel_initializer': {'module': 'keras.initializers',
```

```
      'class_name': 'GlorotUniform',
      'config': {'seed': None},
      'registered_name': None},
     'bias_initializer': {'module': 'keras.initializers',
      'class_name': 'Zeros',
      'config': {},
      'registered_name': None},
     'kernel_regularizer': None,
     'bias_regularizer': None,
     'kernel_constraint': None,
     'bias_constraint': None},
    'registered_name': None,
    'build_config': {'input_shape': (None, 64)}},
   {'module': 'keras.layers',
    'class_name': 'Dense',
    'config': {'name': 'dense_2',
     'trainable': True,
     'dtype': {'module': 'keras',
      'class_name': 'DTypePolicy',
      'config': {'name': 'float32'},
      'registered_name': None},
     'units': 1,
     'activation': 'sigmoid',
     'use_bias': True,
     'kernel_initializer': {'module': 'keras.initializers',
      'class_name': 'GlorotUniform',
      'config': {'seed': None},
      'registered_name': None},
     'bias_initializer': {'module': 'keras.initializers',
      'class_name': 'Zeros',
      'config': {},
      'registered_name': None},
     'kernel_regularizer': None,
     'bias_regularizer': None,
     'kernel_constraint': None,
     'bias_constraint': None},
    'registered_name': None,
    'build_config': {'input_shape': (None, 32)}}],
  'build_input_shape': (None, 84)}
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
```

Training ran for 10 epochs, with batch size = 64:

```
model.fit(X_train, y_train, epochs=10, batch_size=64,
validation_split=0.2)
```

Evaluation of experiments

The following metrics were used to evaluate the experiments.

Confusion matrix

The confusion matrix is one of the most popular ways of evaluating any classification model. Although the matrix itself is not a metric, the matrix representation can be used to define various metrics, all of which become important in some specific case or

scenario. A confusion matrix can be created for binary classification as well as for a classification model with multiple classes.

A confusion matrix is created by comparing the predicted class label of a data point with its actual class label. This comparison is repeated for the entire dataset (dataset), and the results of this comparison are compiled in a matrix or tabular format. This resulting matrix represents the confusion matrix (fig. below).

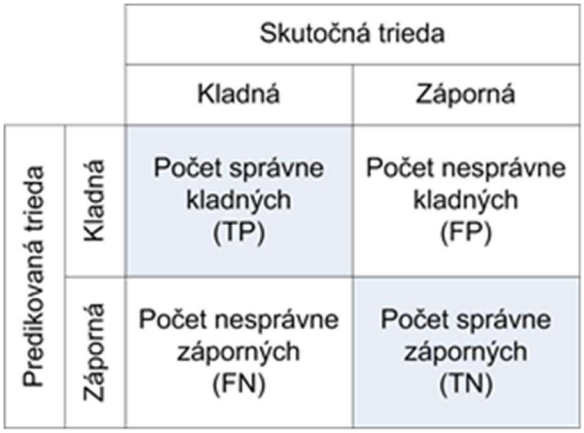


Fig. 3 Confusion matrix

In general, we usually have 2 classes – the positive class and the negative class. Based on this structure, we can clearly see four terms of importance.

• True Positive (TP) – Number of true positives: This is the sum of the total number of examples (data points from the dataset) from the positive class where the actual class label was equal to the predicted class label, i.e. the total number of cases in which we correctly predicted the positive class label using our model.

• False positive (FP) – Number of false positives: This is the number of total cases from the negative class that the model incorrectly classified by predicting them as positive. Hence the name, false positive.

• True Negative (FN) – Number of false negatives: This is the number of the total number of examples from the negative class where the actual class label was equal to the predicted class label, i.e. the total number of cases in which we correctly predicted the negative class label using our model.

• False negative (FN) – Number of false negatives: This is the number of the total number of cases from the positive class that our model incorrectly classified by predicting them as negative. Hence the name, false negative.

Accuracy

This metric is one of the most popular ways of measuring or evaluating the effectiveness of any classifier. It is defined as the overall accuracy or the proportion of correct

predictions of the model. The formula for calculating it from the confusion matrix is as follows:

$$A = \frac{TP + TN}{TP + FP + TN + FN}$$

Precision

Precision (precision), also known as positive predictive value, is another metric that can be derived from the confusion matrix. This metric is defined as the number of predictions that are actually correct or relevant out of all predictions based on the positive class. The formula for this metric is as follows:

$$P = \frac{TP}{TP + FP}$$

A model with a high precision value identifies a higher proportion of the positive class compared with a model with lower precision. This metric becomes important in cases where we are more interested in finding the maximum number of positive classes, even if the overall accuracy decreases.

Recall

Recall (recall) is a model metric for identifying the percentage of relevant data points. It is defined as the number of instances of the positive class that were correctly predicted. It is also referred to as the coverage rate or sensitivity. The formula for calculating this metric is as follows:

$$R = \frac{TP}{TP + FN}$$

This metric becomes an important indicator of classifier effectiveness in scenarios in which we want to capture the largest possible number of cases of a particular class, even if this causes an increase in false detections/predictions. As an example, in the case of bank fraud, a model with a high recall value will provide us with a higher number of potential fraud cases. However, it will equally increase the triggering of alarms for most suspicious cases.

F1-Score

There are cases in which we want to achieve a balanced optimization of precision and recall. The F1-Score is a metric that is the harmonic mean of the precision and recall metrics and helps us to optimize the classifier for balanced precision and recall of prediction. The formula for calculating the F1-Score is as follows:

$$F = \frac{2.P.R}{P + R}$$

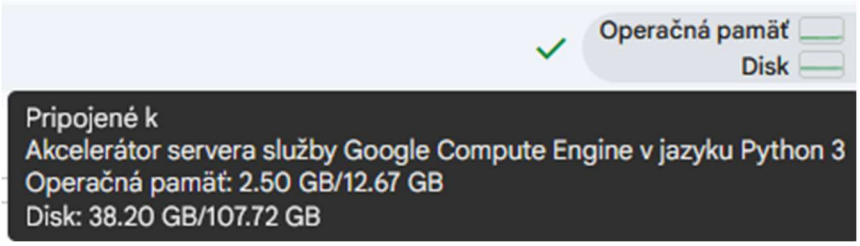
Results

Dataset: TrojanDetec (calculations on Google Colab - Processing time)

Method	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	Processing time (sec.)
Random Forest classifier (50 estimators)	0.999967	0.999967	0.999967	0.999967	9 sec
trying different numbers of trees: 10	99.990172%				additional, in total 60 sec.
20	99.993448%				
30	99.996724%				
40	100.000000%				
50	99.993448%				
60	99.996724%				
70	99.996724%				
80	99.996724%				
90	99.993448%				
AdaBoostClassifier (100 estimators)	0.9996396278338356				20 sec.
cross-validation to obtain a more robust estimate	0.9996265210301141				60 sec.
GaussianNB Classifier					20
gnb.score	91.498493%				
cross_val_score	90.108052%				
using reciver operating characteristic score	92.183155%				
weighted avg		0.921035	0.914985	0.914487	
Deep learning (Feed forward neural network) – 10 epoch	0.983947				55 sec. training alone

weighted avg		0.983953	0.983947	0.983947	

2025 12 05: calculations - screens



Random Forest classifier

[29] ✓ 0 s

```
import numpy as np  # Import NumPy
```

[30] ✓ 9 s

```
from sklearn.ensemble import RandomForestClassifier
# Setup random seed
np.random.seed(42)
clf = RandomForestClassifier(n_estimators = 50) #50 trees (estimators). The n_estimators parameter controls how many t
clf.fit(X_train,y_train)
clf.score(X_test,y_test)
```

```
0.9999672388939851
```

[31] ✓ 0 s

```
y_preds = clf.predict(X_test)
y_preds
```

```
array([0, 1, 1, ..., 1, 0, 1])
```

[32] ✓ 0 s

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, precision_recall_fscore_support
prec_rec = precision_recall_fscore_support(y_test, y_preds, average='weighted')
# print(f"Accuracy:{accuracy_score(y_test, y_preds):.4f}")
print(f"Accuracy:{accuracy_score(y_test, y_preds):.6f}")
# print(f"Precision:{prec_rec[0]:.4f} Recall:{prec_rec[1]:.4f} F1-score:{prec_rec[2]:.4f}")
print(f"Precision:{prec_rec[0]:.6f} Recall:{prec_rec[1]:.6f} F1-score:{prec_rec[2]:.6f}")
print(classification_report(y_test, y_preds, digits=6))
```

```
Accuracy:0.999967
Precision:0.999967 Recall:0.999967 F1-score:0.999967
              precision    recall  f1-score   support

           0   0.999932  1.000000  0.999966     14754
           1   1.000000  0.999937  0.999968     15770

    accuracy                       0.999967     30524
   macro avg   0.999966  0.999968  0.999967     30524
weighted avg   0.999967  0.999967  0.999967     30524
```

[33] ✓ 0 s

```
print(f"Accuracy:{accuracy_score(y_test, y_preds):.6f}")
```

```
Accuracy:0.999967
```

[34]
✓ 1 min

```
# 1 min.
# Imporve model.
# Try different amount on n_estimators
import numpy as np
np.random.seed(42)
for i in range(10,100,10):
    print(f"Trying model with {i} estimators...")
    clf = RandomForestClassifier(n_estimators=i).fit(X_train, y_train)
    #print(f"Model accuracy on test set : {clf.score(X_test,y_test) * 100:.2f}%")
    print(f"Model accuracy on test set : {clf.score(X_test,y_test) * 100:.6f}%")
    print(" ")
```

```
Trying model with 10 estimators...
Model accuracy on test set : 99.990172%

Trying model with 20 estimators...
Model accuracy on test set : 99.993448%

Trying model with 30 estimators...
Model accuracy on test set : 99.996724%

Trying model with 40 estimators...
Model accuracy on test set : 100.000000%

Trying model with 50 estimators...
Model accuracy on test set : 99.993448%

Trying model with 60 estimators...
Model accuracy on test set : 99.996724%

Trying model with 70 estimators...
Model accuracy on test set : 99.996724%

Trying model with 80 estimators...
Model accuracy on test set : 99.996724%

Trying model with 90 estimators...
Model accuracy on test set : 99.993448%
```

[35]
✓ 1 min

```
# 1 min.
from sklearn.model_selection import cross_val_score
cross_val_score = cross_val_score(clf, X, y)
np.mean(cross_val_score)
```

```
np.float64(0.9998427458324681)
```

AdaBoost classifier

[36] ✓ 15 s

```
# 20 sek.
#This code is following a conventional train-test split approach. It fits the model to
#a portion of the data (training set) and then assesses its performance on a separate, previously unseen
#portion of the data (test set).
from sklearn.ensemble import AdaBoostClassifier

# Setup random seed
np.random.seed(42)
Aclf = AdaBoostClassifier(n_estimators=100)
Aclf.fit(X_train,y_train)
Aclf.score(X_test,y_test)
```

```
0.9996396278338356
```

[37] ✓ 1 min

```
# 1 min.
#using cross-validation to obtain a more robust estimate of the model's
#performance by systematically testing it on different subsets of the data.
from sklearn.model_selection import cross_val_score
cross_val_score = cross_val_score(Aclf, X, y)
np.mean(cross_val_score)
```

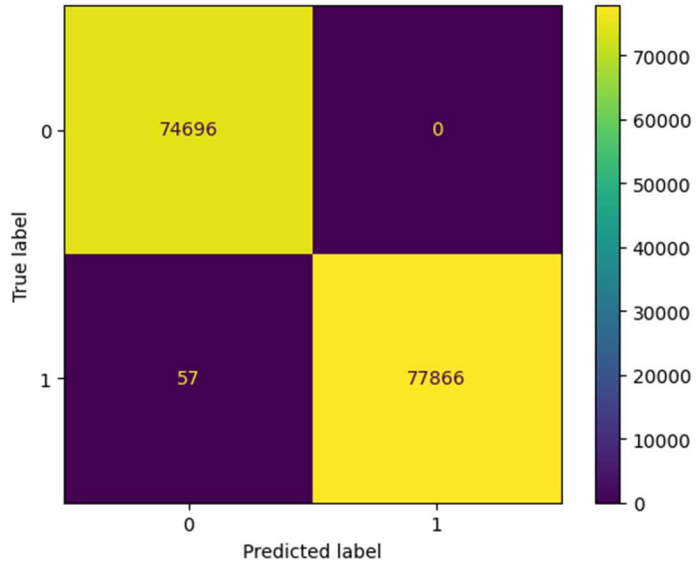
```
np.float64(0.9996265210301141)
```

[38] ✓ 1 s

```
from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_estimator(estimator=Aclf, X=X,y=y)
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x781430cc32c0>
```



[39] ✓ 15 s

```
# 19 sek.
from sklearn.ensemble import GradientBoostingClassifier
Gclf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,max_depth=1, random_state=0).fit(X_train, y_train)
Gclf.score(X_test,y_test)
```

```
1.0
```

[40] ✓ 0 s

```
print(f"Model accuracy on test set : {Gclf.score(X_test,y_test) * 100:.6f}%")
```

```
Model accuracy on test set : 100.000000%
```

GaussianNB Classifier

Fitting the data into the model.

[52] ✓ 0 s

```
from sklearn.naive_bayes import GaussianNB
# Setup random seed
np.random.seed(42)
gnb = GaussianNB()
gnb.fit(X_train, y_train)
```

```
GaussianNB
GaussianNB()
```

Evaluating the model using `score()` method

[54] ✓ 0 s

```
print(f"Model accuracy on test set using score method : {gnb.score(X_test,y_test) * 100:.6f}%")
```

```
Model accuracy on test set using score method : 91.498493%
```

[55] ✓ 0 s

```
y_preds = gnb.predict(X_test)
y_preds
```

```
array([1, 1, 1, ..., 1, 0, 1])
```

Evlauating the model using Cross-validation method to avoid overfitting

[56] ✓ 0 s

```
from sklearn.model_selection import cross_val_score
cross_val_score = cross_val_score(gnb, X, y)
print(f"Model accuracy on test set using Cross-validation method : {np.mean(cross_val_score) * 100:.6f}%")
```

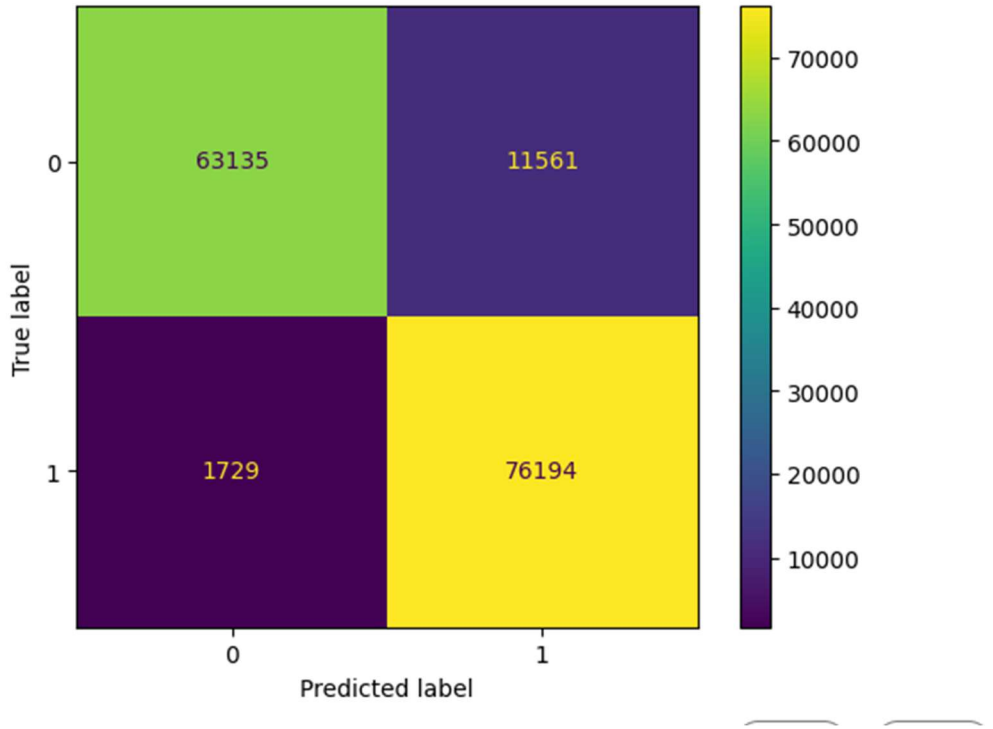
```
Model accuracy on test set using Cross-validation method : 90.108052%
```

Confusion matrix

[57] ✓ 0 s

```
from sklearn.metrics import ConfusionMatrixDisplay
ConfusionMatrixDisplay.from_estimator(estimator=gnb, X=X,y=y)
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x781430570a40>
```



Model Evaluation: Area under the reciever operating characteristic curve (AUC/ROC)

- Area under curve (AUC)
- ROC curve
- ROC curves are a comparison of a model's true positive rate (tpr) versuss a models false positive rate (fpr).

- True possitve = model predicts 1 when truth is 1.
- False possitive = model predicts 1 when truth is 0.
- True negative = model predicts 0 when truth is 0.
- False negative = model predicts 0 when truth is 1

[58]
✓ 0 s

```
# Make predictions with probabilities
y_probs = gnb.predict_proba(X_test)
y_probs[:10]
```

```
array([[0.00846286, 0.99153714],
       [0.35646839, 0.64353161],
       [0.02987006, 0.97012994],
       [0.57359444, 0.42640556],
       [0.58218822, 0.41781178],
       [0.0158397 , 0.9841603 ],
       [0.00721634, 0.99278366],
       [0.30745915, 0.69254085],
       [0.02352071, 0.97647929],
       [0.77689046, 0.22310954]])
```

[59]
✓ 0 s

```
y_probs_positive = y_probs[:,1]
y_probs_positive[:10]
```

```
array([0.99153714, 0.64353161, 0.97012994, 0.42640556, 0.41781178,
       0.9841603 , 0.99278366, 0.69254085, 0.97647929, 0.22310954])
```

[60]
✓ 0 s

```
# Claculate fpr, tpr and thresholds.
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, y_probs_positive)

# Check the false positive rates
fpr
```

```
array([0.00000000e+00, 6.77782296e-05, 2.71112919e-04, ...,
       9.99051105e-01, 9.99186661e-01, 1.00000000e+00])
```

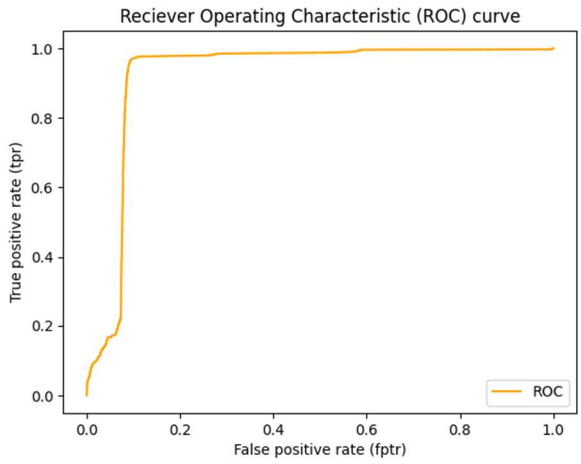
[61]
✓ 0 s

```
import matplotlib.pyplot as plt

def plot_roc_curve(fpr, tpr):
    """
    Plots a ROC curve given the false positive rate (fpr)
    and true positive rate (tpr) of a model
    """
    # Plot ROC curve
    plt.plot(fpr, tpr, color = "orange", label="ROC")

    #Customize the plot.
    plt.xlabel("False positive rate (fptr)")
    plt.ylabel("True positive rate (tpr)")
    plt.title("Reciever Operating Characteristic (ROC) curve")
    plt.legend()
    plt.show()

plot_roc_curve(fpr, tpr)
```



[50]
✓ 0 s

```
from sklearn.metrics import roc_auc_score
print(f"Model accuracy on test set using reciver operating characteristic score method : {roc_auc_score(y_test, y_probs_positive) * 100:.6f
```

```
Model accuracy on test set using reciver operating characteristic score method : 92.183155%
```

Classification Report for **GaussianNB Classifier**

[62] ✓ 0 s

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_preds, digits=6))
```

```
              precision    recall  f1-score   support

           0   0.971974  0.848583  0.906097     14754
           1   0.873378  0.977108  0.922336     15770

    accuracy                       0.914985     30524
   macro avg   0.922676  0.912846  0.914216     30524
weighted avg   0.921035  0.914985  0.914487     30524
```

Deep learning (Feed forward neural network)

+ Kód + Text

[63] ✓ 6 s

```
pip install tensorflow
```

```
Requirement already satisfied: tensorflow in /usr/local/lib/python3.12/dist-packages (2.19.0)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (25.9.23)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (0.6.0)
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (0.2.0)
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (18.1.1)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (3.4.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.12/dist-packages (from tensorflow) (25.0)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3 in /usr/local/lib/python3.12
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (2.32.4)
Requirement already satisfied: setuptools in /usr/local/lib/python3.12/dist-packages (from tensorflow) (75.2.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (1.17.0)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (3.2.0)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (4.15.0)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (2.0.1)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (1.76.0)
Requirement already satisfied: tensorboard~=2.19.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (2.19.0)
Requirement already satisfied: keras>=3.5.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (3.10.0)
Requirement already satisfied: numpy<2.2.0,>=1.26.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (2.0.2)
Requirement already satisfied: h5py>=3.11.0 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (3.15.1)
Requirement already satisfied: ml-dtypes<1.0.0,>=0.5.1 in /usr/local/lib/python3.12/dist-packages (from tensorflow) (0.5.4)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.12/dist-packages (from astunparse>=1.6.0->tensorflow) (0.45.1)
Requirement already satisfied: rich in /usr/local/lib/python3.12/dist-packages (from keras>=3.5.0->tensorflow) (13.9.4)
Requirement already satisfied: namex in /usr/local/lib/python3.12/dist-packages (from keras>=3.5.0->tensorflow) (0.1.0)
Requirement already satisfied: optree in /usr/local/lib/python3.12/dist-packages (from keras>=3.5.0->tensorflow) (0.18.0)
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests<3,>=2.21.0->tensorflow) (3
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests<3,>=2.21.0->tensorflow) (3.11)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests<3,>=2.21.0->tensorflow) (2.5.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests<3,>=2.21.0->tensorflow) (2025.11
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.12/dist-packages (from tensorboard~=2.19.0->tensorflow) (3.10)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local/lib/python3.12/dist-packages (from tensorboard~=2.19.0->t
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from tensorboard~=2.19.0->tensorflow) (3.1.3)
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.12/dist-packages (from werkzeug>=1.0.1->tensorboard~=2.19.0->tens
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.12/dist-packages (from rich->keras>=3.5.0->tensorflow) (4.0.0
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.12/dist-packages (from rich->keras>=3.5.0->tensorflow) (2.1
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.12/dist-packages (from markdown-it-py>=2.2.0->rich->keras>=3.5.0->tensor
```

[64] ✓ 18 s

```
pip install scikit-learn
```

```
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.12/dist-packages (1.6.1)
Requirement already satisfied: numpy>=1.19.5 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (2.0.2)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (1.16.3)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (1.5.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn) (3.6.0)
```

[65] ✓ 17 s

```
pip install pandas
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.12/dist-packages (2.2.2)
Requirement already satisfied: numpy>=1.26.0 in /usr/local/lib/python3.12/dist-packages (from pandas) (2.0.2)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.12/dist-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/dist-packages (from pandas) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12/dist-packages (from pandas) (2025.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.8.2->pandas) (1.17.0)
```

[66] ✓ 28 s

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
```

[67] ✓ 3 s

```
dataset_path = Trojan_Detection_csv
df = pd.read_csv(dataset_path, sep=',', skipinitialspace=True)
df.drop(columns=df.columns[0], axis=1, inplace=True)
```

[68] ✓ 0 s

```
df = df.dropna()
df = df.replace("Trojan", 1)
df = df.replace("Benign", 0)
```

```
/tmp/ipython-input-1200159886.py:3: FutureWarning: Downcasting behavior in `replace` is deprecated and will be removed in a future version.
  df = df.replace("Benign", 0)
```

[69] ✓ 0 s

```
from sklearn.preprocessing import LabelEncoder
number = LabelEncoder()
df["Flow ID"] = number.fit_transform(df["Flow ID"])
df["Source IP"] = number.fit_transform(df["Source IP"])
df["Destination IP"] = number.fit_transform(df["Destination IP"])
df["Timestamp"] = number.fit_transform(df["Timestamp"])
```

[70] ✓ 0 s

```
X = df.drop("Class", axis=1)
y = df["Class"]
```

[71] ✓ 0 s

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

[72] ✓ 0 s

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

[73] ✓ 0 s

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')  # Output layer with sigmoid activation for binary classification
])
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` argument
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[74] ✓ 0 s

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

[75] ✓ 55 s

```
model.fit(X_train, y_train, epochs=10, batch_size=64, validation_split=0.2)
```

```
Epoch 1/10
1527/1527 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8827 - loss: 0.3230 - val_accuracy: 0.9656 - val_loss: 0.1243
Epoch 2/10
1527/1527 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9663 - loss: 0.1177 - val_accuracy: 0.9740 - val_loss: 0.0933
Epoch 3/10
1527/1527 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9728 - loss: 0.0919 - val_accuracy: 0.9764 - val_loss: 0.0828
Epoch 4/10
1527/1527 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9755 - loss: 0.0805 - val_accuracy: 0.9776 - val_loss: 0.0723
Epoch 5/10
1527/1527 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9777 - loss: 0.0704 - val_accuracy: 0.9760 - val_loss: 0.0739
Epoch 6/10
1527/1527 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9792 - loss: 0.0636 - val_accuracy: 0.9815 - val_loss: 0.0584
Epoch 7/10
1527/1527 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9818 - loss: 0.0554 - val_accuracy: 0.9823 - val_loss: 0.0562
Epoch 8/10
1527/1527 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9831 - loss: 0.0528 - val_accuracy: 0.9817 - val_loss: 0.0565
Epoch 9/10
1527/1527 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9840 - loss: 0.0505 - val_accuracy: 0.9819 - val_loss: 0.0540
Epoch 10/10
1527/1527 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9843 - loss: 0.0478 - val_accuracy: 0.9834 - val_loss: 0.0496
<keras.src.callbacks.history.History at 0x7813bf6fb650>
```

[76] ✓ 1 s

```
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test loss: {loss:.6f}, Test accuracy: {accuracy:.6f}")
```

```
954/954 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9842 - loss: 0.0493
Test loss: 0.050193, Test accuracy: 0.983947
```

[77] ✓ 1 s

```
y_pred = (model.predict(X_test) > 0.5).astype(int)
```

```
954/954 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
```

[78] ✓ 0 s

```
# Print classification report
#print(classification_report(y_test, y_pred))
print(classification_report(y_test, y_pred, digits = 6))
```

```
                precision    recall  f1-score   support

           0   0.982201  0.985449  0.983822     15119
           1   0.985672  0.982473  0.984070     15405

    accuracy                       0.983947     30524
   macro avg   0.983936  0.983961  0.983946     30524
weighted avg   0.983953  0.983947  0.983947     30524
```

Summary

On the basis of the experiments performed, it can be concluded that the most suitable methods
for detecting “Trojan horse” type malware from network traffic are the methods

- Random Forest: success rate 99.99%, evaluation time ~9 s
- AdaBoost: success rate 99.96%, evaluation time ~19 s
- DL FFNN: success rate 98.39%, evaluation time ~15 sec.

Of these, Random Forest performed the calculations twice as fast. The next in order, Deep learning (Feed forward neural network), still has a high success rate (98.39%), but the network training time was longer (55 sec.). The fastest method was GaussianNB Classifier – however, it achieved the worst results (91.49%). We consider the **Random Forest** algorithm to be the most suitable method for detecting “Trojan horse” type malware by analysing traffic.