ASSIGNMENT 2

Part A: Conceptual Understanding

1. Spam Detection with Perceptron vs. SVM: A Comparative View

The Perceptron algorithm, one of the earliest machine learning models, offers a simple and efficient solution for this task. It is a linear classifier that learns a set of weights for input features, updating them iteratively based on classification errors.

Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks, with its most common use being in binary classification problems.

Below given is the comparative view:

Perceptron	Support Vector Machine (SVM)
Linear classifier	Margin-based classifier (can be linear or non-linear)
Online learning (updates weights on each misclassified sample)	Batch learning (finds optimal separating hyperplane)
Requires linearly separable data for best performance	Can handle both linearly and non-linearly separable data
Moderate; may misclassify if data is complex	Typically high due to margin maximization
Low; sensitive to noisy and overlapping data	High; aims for maximum margin, reducing impact of noise
No	Yes; supports kernel trick for non-linear separation
Low; fast training and prediction	Higher; especially with kernel functions
High; easy to understand weight updates	Medium; decision boundaries can be complex
Good for simple datasets	Excellent for complex, high-dimensional spam detection tasks
Lightweight filters in real-time applications	Email security systems requiring higher precision

2. Phishing detection using Logistic Regression and Decision Trees.

Phishing is a cyberattack where attackers trick users into revealing sensitive information by impersonating trusted entities.
Goal: Use machine learning to automatically detect phishing attempts from features extracted from URLs, emails, or websites.

Logistic Regression

- A linear classification algorithm.
- Predicts the probability of an instance being phishing or legitimate.
- Works well with linearly separable data.
- Fast and easy to implement.

- May struggle with non-linear data patterns.

Decision Trees

- A non-linear classification model.
- Splits data based on feature thresholds (e.g., URL length, use of IP address).
- Captures complex feature interactions.
- Easy to interpret due to decision paths.
- Prone to overfitting if not pruned properly.

3. How Naïve Bayes helps in Email Filtering?

Naïve Bayes is a probabilistic machine learning algorithm based on Bayes' Theorem. It is especially effective for email filtering tasks like spam detection due to its simplicity and efficiency.

Role in Email Filtering (Spam Detection)

- Naïve Bayes classifies emails into categories like spam and ham (not spam).
- It uses the frequency of words in an email (like "offer", "free", "win") as features.
- Each word contributes to the probability of an email being spam or ham.

Bayes' Theorem:

$$P(Spam|Email) = \frac{P(Email|Spam) \cdot P(Spam)}{P(Email)}$$

Naïve Assumption:

Assumes that all words in the email are independent (which is not always true, but works well in practice).

Advantages

- Fast and efficient on large datasets.
- Performs well even with limited training data.
- Easy to implement and interpret.
- Works well with text classification tasks like email filtering.

4. Malware detection and classification using Decision Trees

- To detect and classify software as malicious or benign using the Decision Tree machine learning algorithm.
- Input: Features extracted from software (static or dynamic analysis).
  Examples: file size, number of API calls, entropy, permissions, system behavior.
- Algorithm: Decision Tree learns decision rules from training data (if-else structure).
- Output: Class label — e.g., "Malware" or "Benign".

Implementation Steps

- Data Collection:
  Use labeled datasets from sources like Kaggle, VirusShare, or CIC-MalMem.
- Feature Extraction
  Static: Byte patterns, opcodes.
  Dynamic: API calls, system/network behavior.
- Preprocessing:
  Encode categorical data.

Normalize features if needed.

- Model Training:
  Fit the Decision Tree using sklearn.tree.DecisionTreeClassifier.
- Evaluation:
  Metrics: Accuracy, Precision, Recall, F1-score.
- Use confusion matrix for detailed analysis.

5. Network Anomaly Detection: Techniques and Use Cases

Network Anomaly Detection (NAD) is the process of identifying abnormal patterns in network traffic that may indicate security threats such as intrusion attempts, malware activity, or data breaches. Unlike signature-based detection, which relies on known attack patterns, anomaly detection focuses on deviations from normal behavior, making it effective for detecting zero-day attacks and unknown threats.

Techniques Used in NAD

i. Statistical Method
These involve creating baseline profiles of normal network behavior (e.g., average packet size, connection frequency) and flagging deviations. They are simple and interpretable but often produce false positives in dynamic environments.

ii. Machine Learning Techniques
- Unsupervised learning (e.g., k-Means, DBSCAN) clusters normal and abnormal behaviors without labeled data.
- Supervised learning (e.g., Random Forest, SVM) requires labeled datasets to classify traffic as normal or malicious.
- Semi-supervised learning focuses on learning normal patterns and identifying outliers.

iii. Deep Learning Methods
Models such as autoencoders and LSTM networks capture complex temporal and spatial patterns in network traffic, offering high accuracy for large-scale systems.

Use Cases

i. Intrusion Detection Systems (IDS): Detect unauthorized access attempts.

ii. DDoS Attack Detection: Identify sudden spikes in traffic volume.

iii. Threat Monitoring: Detect unusual data transfers or privilege misuse.

iv. IoT Security: Identify abnormal communication patterns among IoT devices.

Part B: Research and Analysis

Research and answer the following (with proper references):

1. Identify and explain a real-world incident or attack from the category.

The Mirai botnet is one of the most infamous real-world botnet attacks. It first came to light in 2016 when it was used to launch one of the largest Distributed Denial of Service (DDoS) attacks in history. The attack targeted Dyn, a major DNS provider, causing widespread outages on websites like Twitter, Netflix, Reddit, and Spotify.

How It Worked:

1. Target Devices:
   Mirai infected Internet of Things (IoT) devices—such as home routers, IP cameras, DVRs, and smart appliances—that were running outdated firmware and using default usernames and passwords.
2. Infection Mechanism:
   The malware scanned the internet for vulnerable devices, logged in using a hardcoded list of default credentials, and infected them. Once a device was infected, it became part of the Mirai botnet.
3. Command and Control (C2):
   All infected devices (bots) were controlled by a remote server. The attacker could issue commands to launch DDoS attacks from thousands of compromised devices simultaneously.
4. Attack Execution:
   On October 21, 2016, Mirai flooded Dyn's DNS infrastructure with massive amounts of traffic. Since DNS is essential for routing web addresses, the attack crippled access to major websites across the U.S. and parts of Europe.

2. Describe how AI/ML models could have detected/prevented the threat.

AI and Machine Learning (ML) models are powerful tools in cybersecurity, especially for detecting and mitigating botnet-based threats like Mirai. Here's how they could have been effectively used:

1.Network Traffic Anomaly Detection

AI/ML models trained on normal network behavior could have flagged the unusual spike in traffic generated by infected IoT devices:

- Unsupervised ML models (e.g., Isolation Forest, Autoencoders) could detect outliers in traffic volume, port scanning, or unusual destination IPs.
- Time-series models (e.g., LSTM) could recognize patterns of periodic scanning or bursty DDoS-like behavior.
- Prevention: Early detection could have triggered rate limiting or traffic blocking rules before the full-scale DDoS attack was executed.

2. Device Behavior Profiling

ML models could learn typical behavior profiles for IoT devices:

- If a security camera suddenly began sending large volumes of DNS or HTTP requests, a behavior-based model would recognize it as abnormal.
- Reinforcement learning systems could dynamically adapt to evolving device usage patterns and adjust alerts or blocks accordingly.
- Prevention: Automatic isolation of misbehaving devices on the network.

3. Mention at least two ML algorithms that are suitable for detecting this type of threat and justify your choice.

Two Suitable ML Algorithms for Detecting Mirai-like Botnet Threats

1. Isolation Forest

- Anomaly detection focused: Isolation Forest is designed specifically for identifying outliers or anomalies in large datasets—perfect for detecting unusual network behavior from infected IoT devices.
- Scalable and efficient: It works well in high-dimensional data environments like network traffic logs and is efficient enough for real-time detection.
- No need for labeled data: Since most network environments lack labeled "attack" data, Isolation Forest works well in unsupervised settings, learning from normal behavior to detect deviations.
- It could detect abnormal outbound traffic spikes or unusual scanning behavior from devices that previously showed low or stable activity.

2. Long Short-Term Memory (LSTM) Networks

- Sequential pattern learning: LSTMs are a type of recurrent neural network (RNN) that excel at learning temporal dependencies in time-series data, such as network traffic over time.
- Detects stealthy attacks: Can identify subtle but consistent changes in behavior, like periodic communication with a Command and Control (C2) server.
- Real-time monitoring: LSTMs can continuously learn and adapt to new behavior, making them useful in evolving botnet scenarios.
- LSTM could detect repetitive scanning and traffic bursts that gradually build up to a DDoS event, providing early warning.

Part C: Implementation Implement a simple simulation using Python and scikit-learn (or similar) based on any one of the following:

Ransomware detection using Random Forest:

- Dataset description (can use public datasets)

Importing the dataset

Source :

```
In [1]: import pandas as pd
        dataset = pd.read_csv('data.csv', sep='|')
```

About the dataset

```
In [2]: dataset.head()    #Top 5 row of the dataset
Out[2]:
```

ResourcesMinEntropy	ResourcesMaxEntropy	ResourcesMeanSize	ResourcesMinSize	ResourcesMaxSize	LoadConfigurationSize	VersionInformationSize	legitimate
2.568844	3.537939	8797.000000	216	18032	0	16	1
3.420744	5.080177	837.000000	518	1158	72	18	1
2.846449	5.271813	31102.272727	104	270376	72	18	1
2.669314	6.400720	1457.000000	90	4264	72	18	1
3.421598	5.190603	1074.500000	849	1300	72	18	1

```
In [3]: dataset.tail()    #Last 5 row of the dataset
Out[3]:
```

	Name	md5	Machine	SizeOfOptionalHeader	Characteristics	MajorLinkerVersion
138042	VirusShare_8e292b418568d6e7b87f2a32aee7074b	8e292b418568d6e7b87f2a32aee7074b	332	224	258	11
138043	VirusShare_260d9e2258aed4c8a3bbd703ec895822	260d9e2258aed4c8a3bbd703ec895822	332	224	33167	2
138044	VirusShare_8d088a51b7d225c9f5d11d239791ec3f	8d088a51b7d225c9f5d11d239791ec3f	332	224	258	10
138045	VirusShare_4286dccf67ca220fe67635388229a9f3	4286dccf67ca220fe67635388229a9f3	332	224	33166	2
138046	VirusShare_d7648eae45f09b3adb75127f43be6d11	d7648eae45f09b3adb75127f43be6d11	332	224	258	11

5 rows × 57 columns

```
In [4]: dataset.columns    # name of the columns

Out[4]: Index(['Name', 'md5', 'Machine', 'SizeOfOptionalHeader', 'Characteristics',
       'MajorLinkerVersion', 'MinorLinkerVersion', 'SizeOfCode',
       'SizeOfInitializedData', 'SizeOfUninitializedData',
       'AddressOfEntryPoint', 'BaseOfCode', 'BaseOfData', 'ImageBase',
       'SectionAlignment', 'FileAlignment', 'MajorOperatingSystemVersion',
       'MinorOperatingSystemVersion', 'MajorImageVersion', 'MinorImageVersion',
       'MajorSubsystemVersion', 'MinorSubsystemVersion', 'SizeOfImage',
       'SizeOfHeaders', 'CheckSum', 'Subsystem', 'DllCharacteristics',
       'SizeOfStackReserve', 'SizeOfStackCommit', 'SizeOfHeapReserve',
       'SizeOfHeapCommit', 'LoaderFlags', 'NumberOfRvaAndSizes', 'SectionsNb',
       'SectionsMeanEntropy', 'SectionsMinEntropy', 'SectionsMaxEntropy',
       'SectionsMeanRawsize', 'SectionsMinRawsize', 'SectionMaxRawsize',
       'SectionsMeanVirtualsize', 'SectionsMinVirtualsize',
       'SectionMaxVirtualsize', 'ImportsNbDLL', 'ImportsNb',
       'ImportsNbOrdinal', 'ExportNb', 'ResourcesNb', 'ResourcesMeanEntropy',
       'ResourcesMinEntropy', 'ResourcesMaxEntropy', 'ResourcesMeanSize',
       'ResourcesMinSize', 'ResourcesMaxSize', 'LoadConfigurationSize',
       'VersionInformationSize', 'legitimate'],
      dtype='object')
```

• Data Preprocessing

```
In [5]:  dataset.describe(include="all")     # summary of numeric attributes
Out[5]:
```

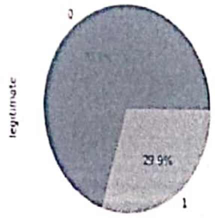
	Name	md5	Machine	SizeOfOptionalHeader	Characteristics	MajorLinkerVersion	MinorLinkerVersion	SizeOfCod
count	138047	138047	138047.000000	138047.000000	138047.000000	138047.000000	138047.000000	1.380470e+0
unique	107488	138047	NaN	NaN	NaN	NaN	NaN	Na
top	mshtml.dll	b6107a498e6a02aa3f6b9af1d2e8628	NaN	NaN	NaN	NaN	NaN	Na
freq	187	1	NaN	NaN	NaN	NaN	NaN	Na
mean	NaN	NaN	4259.069274	225.845632	4444.145994	8.619774	3.819286	2.425960e+0
std	NaN	NaN	10880.347245	5.121399	8186.782524	4.088757	11.862675	5.754485e+0
min	NaN	NaN	332.000000	224.000000	2.000000	0.000000	0.000000	0.000000e+0
25%	NaN	NaN	332.000000	224.000000	258.000000	8.000000	0.000000	3.020800e+0
50%	NaN	NaN	332.000000	224.000000	258.000000	9.000000	0.000000	1.138640e+0
75%	NaN	NaN	332.000000	224.000000	8226.000000	10.000000	0.000000	1.203200e+0
max	NaN	NaN	34404.000000	352.000000	49551.000000	255.000000	255.000000	1.818587e+0

11 rows × 57 columns

```
In [6]:  dataset.info()     # info about the whole dataset

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 138047 entries, 0 to 138046
Data columns (total 57 columns):
Name                            138047 non-null object
md5                             138047 non-null object
Machine                         138047 non-null int64
SizeOfOptionalHeader            138047 non-null int64
Characteristics                 138047 non-null int64
MajorLinkerVersion              138047 non-null int64
MinorLinkerVersion              138047 non-null int64
SizeOfCode                      138047 non-null int64
SizeOfInitializedData           138047 non-null int64
SizeOfUninitializedData         138047 non-null int64
AddressOfEntryPoint             138047 non-null int64
BaseOfCode                      138047 non-null int64
BaseOfData                      138047 non-null int64
ImageBase                       138047 non-null float64
SectionAlignment                138047 non-null int64
FileAlignment                   138047 non-null int64
MajorOperatingSystemVersion     138047 non-null int64
MinorOperatingSystemVersion     138047 non-null int64
MajorImageVersion               138047 non-null int64
MinorImageVersion               138047 non-null int64
MajorSubsystemVersion           138047 non-null int64
MinorSubsystemVersion           138047 non-null int64
SizeOfImage                     138047 non-null int64
SizeOfHeaders                   138047 non-null int64
CheckSum                        138047 non-null int64
Subsystem                       138047 non-null int64
DllCharacteristics              138047 non-null int64
SizeOfStackReserve              138047 non-null int64
SizeOfStackCommit               138047 non-null int64
SizeOfHeapReserve               138047 non-null int64
SizeOfHeapCommit                138047 non-null int64
LoaderFlags                     138047 non-null int64
NumberOfRvaAndSizes             138047 non-null int64
SectionsNb                      138047 non-null int64
SectionsMeanEntropy             138047 non-null float64
SectionsMinEntropy              138047 non-null float64
SectionsMaxEntropy              138047 non-null float64
SectionsMeanRawsize             138047 non-null float64
SectionsMinRawsize              138047 non-null int64
SectionMaxRawsize               138047 non-null int64
SectionsMeanVirtualsize         138047 non-null float64
SectionsMinVirtualsize          138047 non-null int64
SectionMaxVirtualsize           138047 non-null int64
ImportsNbDLL                    138047 non-null int64
ImportsNb                       138047 non-null int64
ImportsNbOrdinal                138047 non-null int64
ExportNb                        138047 non-null int64
ResourcesNb                     138047 non-null int64
ResourcesMeanEntropy            138047 non-null float64
ResourcesMinEntropy             138047 non-null float64
ResourcesMaxEntropy             138047 non-null float64
ResourcesMeanSize               138047 non-null float64
ResourcesMinSize                138047 non-null int64
ResourcesMaxSize                138047 non-null int64
LoadConfigurationSize           138047 non-null int64
VersionInformationSize          138047 non-null int64
legitimate                      138047 non-null int64
dtypes: float64(10), int64(45), object(2)
memory usage: 60.0+ MB
```

```
In [7]:  dataset['legitimate'].value_counts()     # count of malware (0) and benign (1) files in dataset

Out[7]:  0    96724
         1    41323
         Name: legitimate, dtype: int64
```

```
In [51]: import matplotlib.pyplot as plt

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



• **Model building and evaluation**

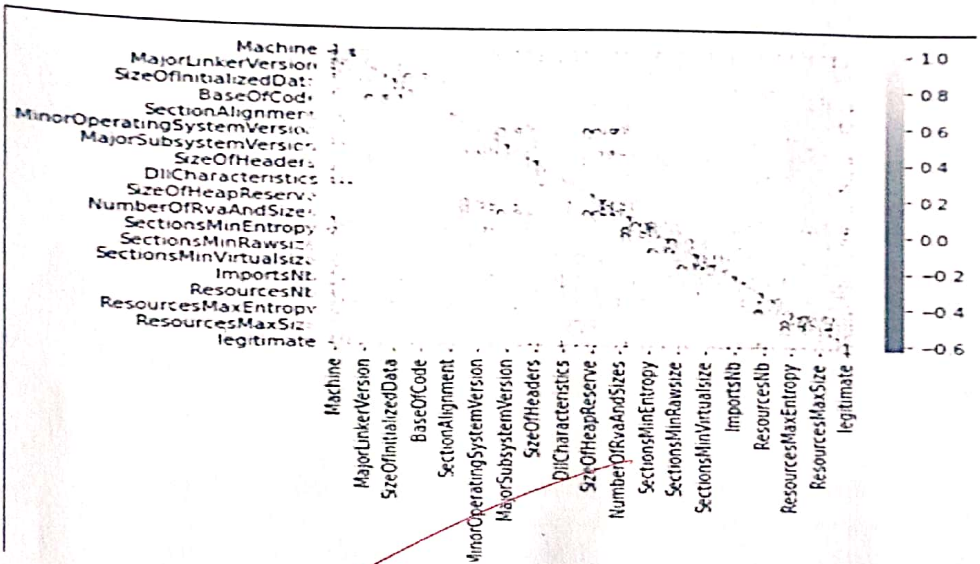
```
In [75]: import os
         import pandas
         import numpy
         import pickle
         import pefile
         import sklearn.ensemble as ek
         from sklearn.feature_selection import SelectFromModel
         import joblib
         from sklearn.tree import DecisionTreeClassifier
         from sklearn.metrics import confusion_matrix
         from sklearn import svm
         import sklearn.metrics as metrics
         # Feature
         X = dataset.drop(['Name','md5','legitimate'],axis=1).values    #Droping this because classification model will not accept object
         # Target variable
         y = dataset['legitimate'].values
         from sklearn.ensemble import ExtraTreesClassifier
         from sklearn.feature_selection import SelectFromModel

         # Train ExtraTrees model with explicit n_estimators
         extratrees = ExtraTreesClassifier(n_estimators=100).fit(X, y)
         model = SelectFromModel(extratrees, prefit=True)

         X_new = model.transform(X)
         nbfeatures = X_new.shape[1]

         print(f"Number of selected features: {nbfeatures}")
         nbfeatures
         from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X_new, y ,test_size=0.29, stratify = y)
         features = []
         index = numpy.argsort(extratrees.feature_importances_)[::-1][:nbfeatures]
         for f in range(nbfeatures):
             print("%d. feature %s (%f)" % (f + 1, dataset.columns[2+index[f]], extratrees.feature_importances_[index[f]]))
             features.append(dataset.columns[2+f])

         Number of selected features: 13
         1. feature DllCharacteristics (0.153561)
         2. feature Characteristics (0.133433)
         3. feature Machine (0.099254)
         4. feature Subsystem (0.068109)
         5. feature VersionInformationSize (0.062444)
         6. feature SectionsMaxEntropy (0.056239)
         7. feature ImageBase (0.046030)
         8. feature SizeOfOptionalHeader (0.044028)
         9. feature ResourcesMaxEntropy (0.038746)
         10. feature MajorSubsystemVersion (0.038138)
         11. feature ResourcesMinEntropy (0.037628)
         12. feature MajorOperatingSystemVersion (0.022191)
         13. feature SectionsMinEntropy (0.019408)
```

Testing which Classifier will give better result

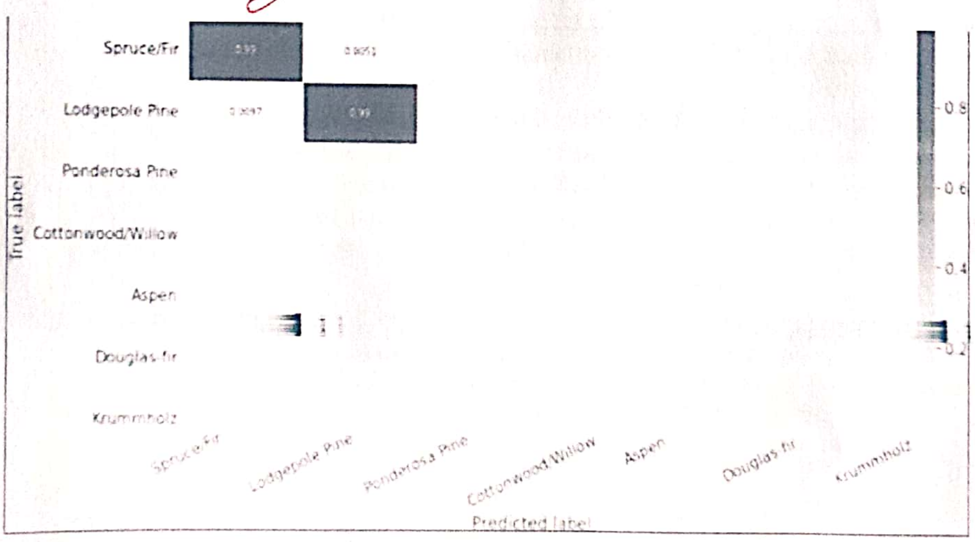
```
In [83]: model = { "DecisionTree": DecisionTreeClassifier(max_depth=10),
                  "RandomForest":ek.RandomForestClassifier(n_estimators=50)}

In [84]: results = {}
         for algo in model:
             clf = model[algo]
             clf.fit(X_train,y_train)
             score = clf.score(X_test,y_test)
             print ("%s : %s " %(algo, score))
             results[algo] = score
             winner = max(results, key=results.get)# Selecting the classifier with good result
             print("Using", winner, "for classification, with",len(features), 'features.')

         DecisionTree : 0.9902333016935605
         Using DecisionTree for classification, with 13 features.
         RandomForest : 0.9942548833491532
         Using RandomForest for classification, with 13 features.
```

- **Performance metrics**

Correlation matrix:



Confusion matrix



Part D:Reflection And Innovation

Major Challenge: Class Imbalance in Random Forest Detection

A common issue in malware and intrusion detection is class imbalance. There are many more benign samples than malicious ones. When a Random Forest is trained on data that is heavily skewed, it tends to favor the majority (benign) class and often misses many attacks. For instance, mobile malware datasets usually have very few malicious apps compared to benign ones. In practice, a Random Forest can achieve high overall accuracy but still overlook rare threats. To address this, researchers use techniques to balance the data. Oversampling, like SMOTE, or under sampling can help rebalance the classes before training. Adjusting class weights in the Random Forest can also penalize mistakes in identifying malware more heavily. One recent study applied a balancing method in the Random Forest ensemble, training each tree on a different balanced subset. This approach improved the detection of minority class (malware) when compared to unbalanced training. In summary, balancing strategies, either through data sampling or weighted training, are essential to counteract the bias caused by class imbalance when using Random Forests in cybersecurity..

Idea: Federated Learning for Collaborative Threat Detection

A promising approach is to use federated learning (FL) to enhance detection accuracy while maintaining data privacy. In this method, multiple organizations (or network segments) each train a local threat-detection model on their own logs. They then share only model updates with a central server. The server uses federated averaging to create a global model that reflects patterns from all participants without exchanging raw data. This approach utilizes a much larger and more varied dataset of attacks than any single site can provide, improving the model's ability to generalize to new or rare threats. For example, an organization that has never encountered a specific malware strain could benefit if another participant's local model has learned it.

This federated IDS can adjust to evolving threats by consistently integrating fresh local updates. Importantly, it maintains privacy and compliance since sensitive logs remain under each host's control. Recent studies show that FL-based IDS can match or surpass traditional centralized models. One study found that a federated IDS achieved better accuracy and lower loss than a standard deep-learning IDS, particularly in situations where privacy is crucial. In other words, FL contributes to building an effective global detector, even in IoT or enterprise networks, while reducing the risk of data breaches during model training.

In practice, this idea could be put into action by developing a common AI threat-detection model, like a neural classifier for network anomalies, that is trained across firms or departments using federated learning. Each participant would periodically download the current model, train it on local logs, and upload model gradients. The server would then combine these updates into an improved model. Over time, the global model would learn a richer set of attack signatures and behavioral patterns. This collaborative, decentralized training method is effective because it leverages much more diverse data (enhancing detection of new attacks) and has been shown to improve IDS performance, all while ensuring that raw security data remains local. Thus, federated learning offers a practical way to enhance AI-driven threat detection by merging intelligence from multiple sources.

Sources: Recent studies and reviews support these points. For instance, Shanmugam et al.

highlight that imbalanced traffic (benign far exceeding malicious) in IDS data causes ML models to excel on normal traffic but fail on rare attacks. Alazab et al. demonstrate that federated learning allows multiple parties to train a joint IDS model without sharing data, resulting in better accuracy and robust detection while preserving privacy. These and other works inform the analysis and proposed solutions outlined above.