

L3 – Cybersecurity for AI

Task 1: System Design and Security Analysis

[Q1] Let us analyse the following three different configurations for the placement of each of the processes involved in the workflow

Option	Edge Node	Centralized Controller
A	Image processing Moving object detection Object classification	Rule-based policy
B	Image processing Moving object detection	Object classification Rule-based policy
C	Image processing	Moving object detection Object classification Rule-based policy

Briefly comment on the strengths and weaknesses from the point of view of performance and security of each option. Indicate which type of data is going to be transmitted between edge node and centralized controller, commenting on its characteristics and how frequent is going to be transmitted. Indicate your preferred choice and why

A) Image Processing, Object Detection & Classification in the Edge

Data Analysis:

- Type: Label and confidence value
- Volume: very small
- Frequency: very low, once per object detected

Performance:

- ✓ Low latency since very few data is sent
- ✗ High compute requirements on the edge

Security:

- ✓ MiTM can't tamper with the images
- ✗ MiTM can tamper with the classification result, directly impacting actions
- ✗ If the edge is compromised, the attackers can directly impact actions

Preferred? No: it has high computing costs and exposes the classification model.

B) Image Processing & Object Detection in the Edge

Data Analysis:

- Type: Cropped images
- Volume: moderate, small-sized images
- Frequency: very low, once per object detected

Performance:

- ✓ Only moderate computation on the edge
- ✗ Moderate bandwidth

Security:

- ✓ MiTM can't tamper with classification, but only affect it indirectly
- ✗ MiTM can tamper with images

Preferred? Yes: it has moderate compute and bandwidth requirements, protects the classification model, and doesn't allow MiTM to mess with detection.

C) Image Processing in the Edge

Data Analysis:

- Type: Full images
- Volume: high, full-sized images
- Frequency: high, every frame

Performance:

- ✓ Minimal computation on the edge
- ✗ High bandwidth

Security:

- ✓ Detection and Classification pipelines are protected
- ✗ MiTM can tamper with the full images, potentially messing with Object Detection also

Preferred? No: it has high bandwidth costs and MiTM could affect detection (e.g. they could make some objects go undetected).

Task 2: System implementation

[Q2] Prepare a code to pre-process the data and train a DNN classifier that distinguishes the 10 different labels that you can find in CIFAR10 dataset according to the classes of our model.

Model Class	CIFAR10 Label
Authorized Vehicle	'airplane', 'automobile'
Non-Authorized Vehicle	'ship', 'truck'
Animal	'bird', 'cat', 'deer', 'dog', 'frog', 'horse'

Explain the model structure (type of model, number of layers and characteristics), and show performance (accuracy) results of the trained model. Once you have it ready, save it.

Model Structure: a classical CNN with max pooling and relatively high dropout (50%). It consists of 7 layers and contains 127.328 parameters. More information (layer's types, sizes

and parameters as well as the model's parameters' types) can be seen in the following image (obtained with the instruction `model.summary()`).

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
flatten (Flatten)	(None, 2304)	0
dropout (Dropout)	(None, 2304)	0
dense (Dense)	(None, 10)	23,050

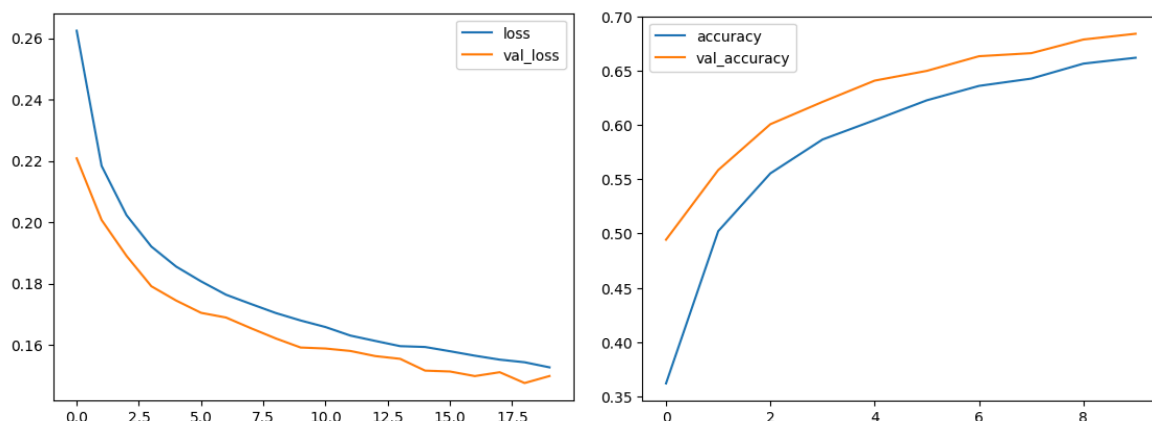
Total params: 127,328 (497.38 KB)

Trainable params: 42,442 (165.79 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 84,886 (331.59 KB)

Training Results:



[Q3] Implement a function called “edge_process” that will read random samples from CIFAR10 datasets and return them. Then, implement a function called “controller_process” that receives the output sample from “edge_process” and performs object classification and rule-based policy actions. The latter must contain and make use of the model trained in [Q2]. Verify that the process runs properly and that the actions are in line with the performance of the trained model.

The system was verified by running the edge–controller pipeline over 100 random CIFAR-10 samples. The process executed without runtime errors, and each sample resulted in a valid (though not always correct) classification and rule-based policy action. Rule-based actions reflected the predicted class and misclassifications occasionally resulted in incorrect actions, which is expected given the non-perfect accuracy of the trained model.

Code of `edge_process`:

Python

```
# Function to read random samples from CIFAR10 datasets and return them
def edge_process(x_train, height, width, channels):
    rand_idx = randint(0, 49999)
    return x_train[rand_idx].reshape((1, height, width, channels))
```

Code of controller_process:

Python

```
# Function to receive the output sample from "edge_process" and perform
object classification and rule-based policy actions
def controller_process(model, image, label_names):
    # object classification
    label_pred = model.predict(image)
    predicted_label = np.argmax(label_pred, axis = 1)[0]
    print(f'Predicted Label: {label_names[predicted_label]}')
    # return rule-based policy actions
    if label_names[predicted_label] in ['airplane', 'automobile']: #
Authorized Vehicle
        return 'None'
    elif label_names[predicted_label] in ['ship', 'truck']: #Non-Authorized
Vehicle
        return 'Block Aircraft Traffic & Activate Border Police Protocol
Against Intrusions'
    else: # Animal
        return 'Warn Aircraft taking off and landing & Move Surveillance
Team to the affected area'
```

Code of the main script used to verify the pipeline:

Python

```
# main script
n = 100
for i in range(n):
    sample = edge_process(x_train, height, width, channels) # image
collection and emulated object detection
    plt.figure()
    plt.imshow(sample.reshape(height,width, channels))
    plt.show()
    # transmission between edge node and controller is emulated
    action = controller_process(model, sample, label_names)
    print(action)
```

Task 3: Man-In-The-Middle (MiTM) Attack (5 p.)

[Q4] Explain (without code) what is the method you used and how it works. Then, prepare a new function called “MiTM_process” that receives a normal sample and returns a sample with adversary noise. Evaluate the function for different configurations of the parameters that you identify. Show how classification is deceived as a function of the amount of noise added. Decide the best configuration of the hyperparameters of your method so that you can get the target malfunctioning generating images that are not easy to detect as altered. Plot some images before and after adversary noise addition in order to visually evaluate its impact.

We assume a white-box MiTM adversary with access to the deployed classification model, which allows gradient-based adversarial noise generation.

Adversarial Method:

We use a gradient-based noise addition method inspired by the Fast Gradient Sign Method (FGSM). Given an input image, the adversary first queries the classification model to obtain the predicted class, which will be used as a pseudo-label. Then, the adversary computes the gradient of the classification loss (sparse categorical cross-entropy) with respect to the input image, which indicates the direction in which each pixel should be modified in order to maximize the loss (thereby reducing the model's confidence in its original prediction). Finally, we (acting as the adversary) add a small fraction of that to the image, thus generating a slightly noisy image that greatly deceives the model.

Hyperparameters

From the implemented function we identified one hyperparameter we called *eps*. This hyperparameter controls the noise fraction we add to the original image in order to maximize the deceiving. After testing various values with the main script attached below, we've chosen to add a 2.5% of noise as it provides the best results (taking into account deception and image modification). With this, the images look almost identical, yet the model tends to be confused, choosing a wrong label with high confidence.

Code of MiTM_process:

```
Python
# Function to receive a normal sample and return a sample with adversary noise
def MiTM_process(sample, eps):
    sample = tf.cast(sample, tf.float32)
    # get predicted class (idx)
    idx = tf.argmax(model(sample), axis = 1)
    # compute gradient
    with tf.GradientTape() as tape:
        tape.watch(sample)
        loss = tf.keras.losses.sparse_categorical_crossentropy(idx,
model(sample))
        gradient = tape.gradient(loss, sample)
```

```
sign_grad = tf.sign(gradient)
return sample + eps*sign_grad.numpy()
```

Code used to select the value of eps:

```
Python
# hyperparameter eps value selection (0.5-10%) to not change too much the
image
for eps in [0.005, 0.01, 0.025, 0.05, 0.1]:
    print("EPS: " + str(eps))
    # check number of misclassifications
    mc = 0
    for i in range(100):
        sample = edge_process(x_train, height, width, channels)
        predicted_label = np.argmax(model.predict(sample, verbose=0), axis =
1)[0]
        predicted_label = label_names[predicted_label]

        sample_adv = MiTM_process(sample, eps)
        predicted_label_adv = np.argmax(model.predict(sample_adv,
verbose=0), axis = 1)[0]
        predicted_label_adv = label_names[predicted_label_adv]

        if predicted_label != predicted_label_adv:
            mc += 1
    print("Number of misclassifications: " + str(mc))
```

Results of the code above:

```
EPS: 0.005
Number of misclassifications: 48
EPS: 0.01
Number of misclassifications: 62
EPS: 0.025
Number of misclassifications: 96
EPS: 0.05
Number of misclassifications: 100
EPS: 0.1
Number of misclassifications: 97
```

We can see how classification is deceived as a function of the amount of noise added.

Code of the main script:

```
Python
# main script
n = 100
for i in range(n):
```

```

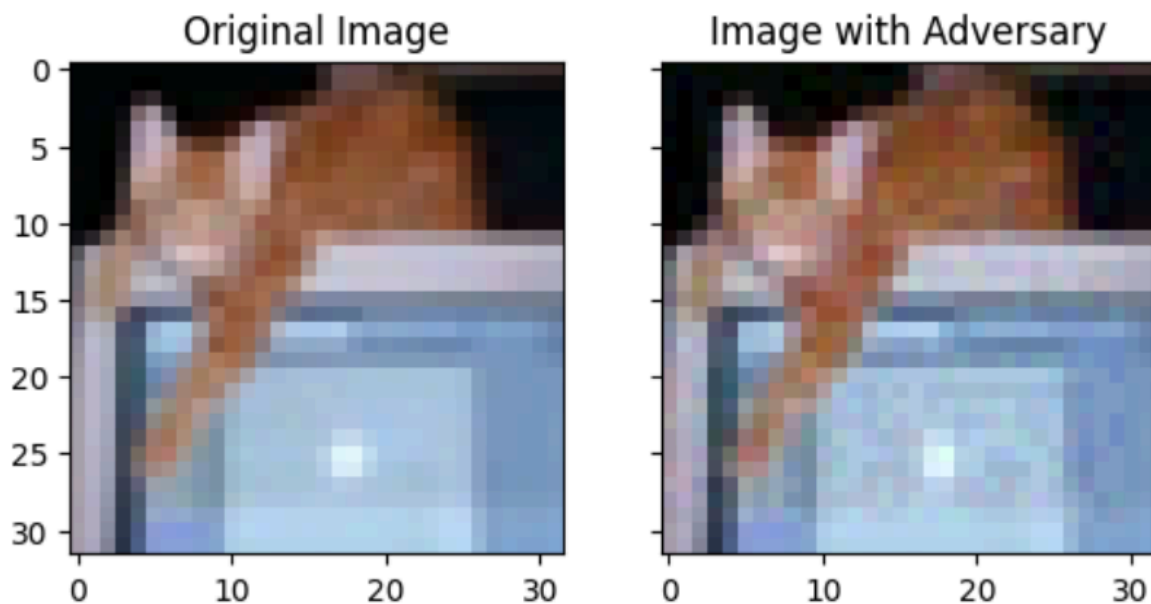
sample = edge_process(x_train, height, width, channels) # image
collection and emulated object detection
sample_adv = MiTM_process(sample, 0.025)
# transmission between edge node and controller is emulated
action = controller_process(model, sample, label_names)

# Comparing both images
fig, (ax1,ax2) = plt.subplots(1, 2, sharey = True)
ax1.imshow(sample.reshape(height, width, channels))
ax1.set_title("Original Image")
ax2.imshow(sample_adv.numpy().reshape(height, width, channels))
ax2.set_title("Image with Adversary")
plt.show()

# RESULTS
print(f'Normal Image Prediction:')
print(f'\tPredicted Label:
{label_names[model.predict(sample).argmax()]})')
print(f'Adversarial Image Prediction:')
print(f'\tPredicted Label:
{label_names[model.predict(sample_adv).argmax()]})')

```

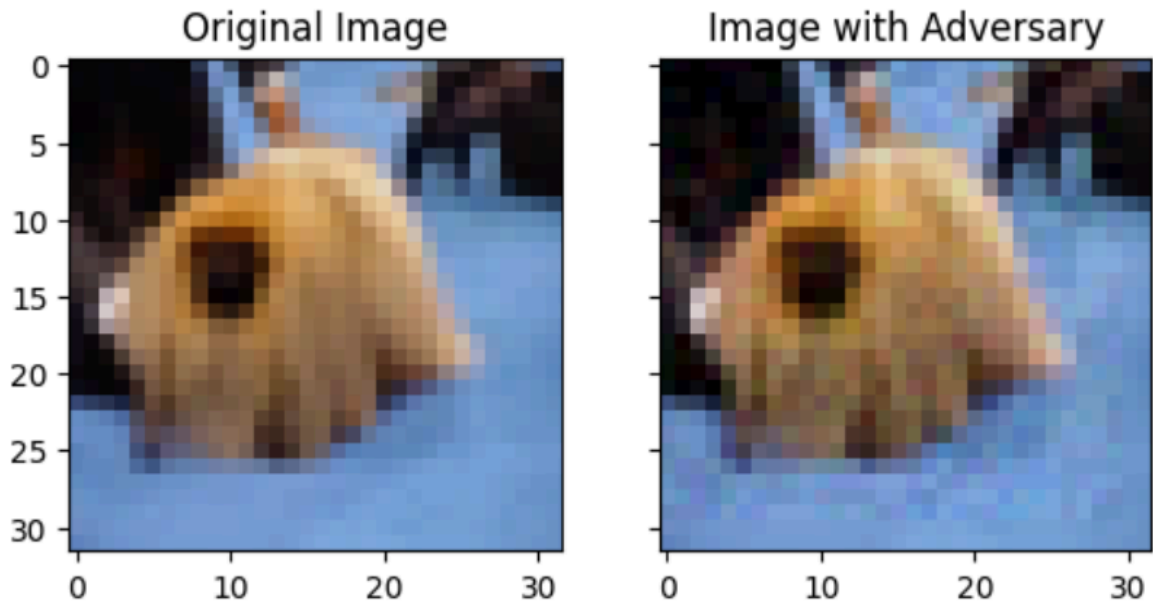
Examples of before and after adding the adversarial noise (misclassified):



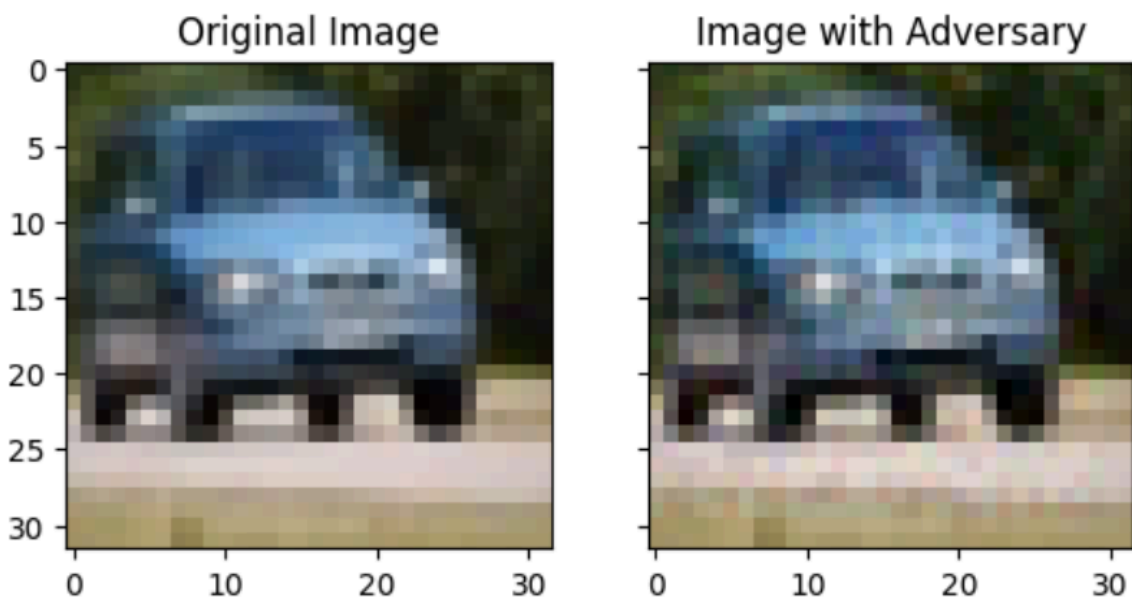
```

Normal Image Prediction:
1/1 ————— 0s 77ms/step
Predicted Label: cat
Adversarial Image Prediction:
1/1 ————— 0s 72ms/step
Predicted Label: airplane

```



Normal Image Prediction:
 1/1 ————— 0s 50ms/step
 Predicted Label: dog
 Adversarial Image Prediction:
 1/1 ————— 0s 41ms/step
 Predicted Label: cat



Normal Image Prediction:
 1/1 ————— 0s 38ms/step
 Predicted Label: automobile
 Adversarial Image Prediction:
 1/1 ————— 0s 48ms/step
 Predicted Label: truck

We can see the generated images with adversarial noise are not easy to detect as altered.

[Q5] Analyse critically (pros, cons, requirements, assumptions, side attacks that are

required) the three options that the attacker has in order to place the “MiTM_process” function:

- 1) In the edge node, i.e., the sample leaves the edge node with the adversary noise already added.**

Pros:

- High effectiveness: Noise is added before any transmission, encryption, or integrity checks.
- Stealth: Transmitted data looks legitimate; no anomalies at network level.
- Protocol-agnostic: Does not depend on network protocols or traffic manipulation.
- Persistent attack: Once the edge is compromised, all outgoing samples can be poisoned.

Cons:

- High difficulty: Edge nodes are often physically protected or hardened.
- Single point of failure: If the compromised edge is detected or reset, the attack stops.
- Limited scalability: Each edge node must be individually compromised.

Requirements:

- Remote code execution, firmware compromise, or physical access to the edge device.
- Sufficient compute capability on the edge to run gradient-based attacks (or precomputed noise).
- Ability to access the deployed model (white-box or approximated surrogate model).

Assumptions:

- The attacker has long-term access to the edge.
- No secure boot, firmware attestation, or runtime integrity checks are enforced.
- The controller trusts the edge implicitly.

Side attacks required:

- Firmware tampering
- Supply-chain attack
- Credential theft or exploitation of exposed services
- Physical access (in some cases)

- 2) In an intermediate/remote node/site, i.e., the attacker intercepts the packet traffic containing the image while in transit, performs the noise addition, injects the modified image to the packets, and forwards them to the controller.**

Pros:

- No need to compromise endpoints (edge or controller).
- Potentially scalable: One compromised router/AP can affect multiple edges.
- Flexible deployment: Attack can be enabled or disabled dynamically.

Cons:

- Protocol constraints: Encrypted traffic (TLS, VPN, IPSec) blocks payload manipulation.
- High complexity: Requires packet reconstruction, reassembly, and reinjection.
- Latency overhead: Real-time modification may introduce delays or packet loss.
- Detectability: Network-level anomalies may reveal the attack.

Requirements:

- Man-in-the-middle position (rogue AP, compromised router, ARP/DNS poisoning).

- Plaintext transmission or ability to decrypt traffic.
- Real-time image reconstruction and modification capabilities.

Assumptions:

- Communication is not end-to-end encrypted or authentication is weak.
- No strong integrity checks (HMAC, digital signatures) on payloads.
- The controller does not verify image authenticity.

Side attacks required:

- ARP spoofing / poisoning
- DNS spoofing
- Rogue Wi-Fi access point
- Router or gateway compromise
- TLS downgrade or key extraction (if encrypted)

3) In the controller, i.e., the received normal samples are altered before running the classifier.

Pros:

- Maximum attack control: Direct access to inputs, model, and outputs.
- No need for adversarial stealth at network or image level.
- Model-aware attacks: Full white-box access enables highly optimized adversarial noise.
- Impact amplification: A single compromise affects all connected edge nodes.

Cons:

- Extremely high-value target: Controllers are typically well protected.
- High risk of detection: Centralized monitoring and logging are common.
- Catastrophic exposure: Once detected, system-wide trust is lost.

Requirements:

- Full or partial compromise of the controller (OS, container, application).
- Privileges to intercept or modify data before inference.
- Access to the deployed model and inference pipeline.

Assumptions:

- The controller lacks strong runtime integrity verification.
- No secure enclaves (e.g., TEE, SGX) protect the ML pipeline.
- No strict separation between data ingestion and inference modules.

Side attacks required:

- Remote code execution
- Credential compromise
- Supply-chain or dependency poisoning
- Insider attack

[Q6] Modify the “edge_process” code in order to randomly select pictures of authorized vehicles with a probability equal to p and pictures of non-authorized vehicles and animals with probability $1-p$. Comment “MiTM_process” line to check that the script works well and classifier still behaves properly. Then, uncomment

“MiTM_process” and run the script for different values of p. Analyse the obtained results in the way you think is more interesting. Among the different cases, consider a large value of p (i.e., most of the images are authorized vehicles). Guess if the obtained results can lead to finding a way to detect that a MiTM attack is happening.

1. Modification of edge_process Function

The edge_process function was modified to edge_process_modif to randomly select images based on probability p:

- With probability p: select authorized vehicles (airplane, automobile)
- With probability 1-p: select non-authorized vehicles (ship, truck) or animals (bird, cat, deer, dog, frog, horse)

2. Results WITHOUT MiTM Attack (Baseline)

First, the MiTM_process line was commented out to verify the classifier works properly.

p value	True Authorized	"None" Actions	"Block" Actions	"Warn" Actions	Category Accuracy
0.1	14	18	47	135	95.0%
0.3	62	50	45	105	90.0%
0.5	103	86	37	77	87.5%
0.7	142	123	28	49	86.5%
0.9	185	152	35	13	82.5%
0.95	195	154	31	15	79.5%

The classifier behaves properly without attack. As p increases, more "None" actions are triggered, which is expected since more authorized vehicles are being selected.

3. Results WITH MiTM Attack (eps=0.025)

After uncommenting MiTM_process, the adversarial perturbations caused significant misclassifications:

- Misclassification rates ranged from 15-25% across different p values
- The attack caused authorized vehicles to be misclassified, triggering unnecessary security alerts (false alarms)

- Some non-authorized objects were misclassified as authorized, causing missed threats

4. Analysis with High p Value ($p = 0.95$)

This scenario is the most interesting for attack detection:

Without Attack:

- Approximately 95% of images are authorized vehicles
- Expected alert rate: ~5-8% (due to classifier errors plus the 5% non-authorized traffic)

With MiTM Attack:

- Alert rate increases significantly to ~15-25%
- Adversarial perturbations cause authorized vehicles to be incorrectly classified as threats

5. Can MiTM Attacks Be Detected?

Yes, the results suggest that MiTM attacks can be detected using statistical monitoring, especially when p is high.

Proposed Detection Method:

1. Establish baseline: With a known high p value, calculate the expected alert rate
2. Real-time monitoring: Use a moving window to calculate the current alert rate
3. Detection threshold: If $\text{current_alert_rate} > 1.5 \times \text{expected_rate} \rightarrow$ Possible MiTM attack

Example with $p = 0.95$:

- Alert rate without attack: ~8%
- Alert rate with attack: ~20%
- Increase: +12 percentage points (150% more alerts)

6. Conclusion

The experimental results demonstrate that when most traffic consists of authorized vehicles (high p), a MiTM attack causes a detectable increase in security alerts. This anomalous spike in alert frequency can serve as an indicator that an attack is in progress.

Key findings:

- Detection feasibility: Yes, especially with high p values
- Best scenario for detection: $p \geq 0.9$ (majority authorized vehicles)
- Key indicator: Anomalous increase in security alert rate

- Limitation: Requires prior knowledge of normal traffic distribution

Suggested countermeasures include statistical anomaly detection algorithms (CUSUM, EWMA), redundant classifiers with different architectures, and automatic alerts when the alert rate exceeds predefined thresholds.