Security By Design - Project

Digital Greenhouse Environment

Real-time CPS Simulation, Attack and Detection

Reddi Ashish (1009621)
Abhilasha Mohapatra (1009708)
Vidwaan Krishnan Siddharth (1009711)

Table of Contents

Introduction

A greenhouse is a controlled agricultural environment where factors such as temperature, humidity, light, and water are regulated to support plant growth. Unlike open-field farming, it requires precise monitoring and timely adjustments to maintain optimal conditions. They allow farmers to extend growing seasons, protect crops from extreme weather, and improve yield quality.

To achieve this, automation through Cyber-Physical Systems (CPS) provides an effective solution. By integrating sensors, programmable logic controllers (PLCs), and actuators, CPS can continuously scan environmental conditions, decide on corrective actions, and execute responses such as irrigation, heating, or lighting. This reduces reliance on manual intervention and enhances both efficiency and sustainability.

Our team selected the greenhouse as the focus of this project because it represents a real-world system where CPS can deliver high impact yet remains vulnerable to cyber threats. Some of the possible attacks on sensors or control logic can be:

- **Temperature**: A sustained deviation just 2-3 °C above the ideal range can stress plants, reduce flowering, or trigger premature fruit drop. At the other extreme, prolonged cooler conditions slow growth and can cause chilling injury.
- **Moisture**: Over-irrigation leads to root rot and nutrient leaching, while under-watering stunts plant growth and reduces yield size.
- **Light**: Excess artificial lighting may cause leaf burn and energy waste, whereas insufficient light reduces photosynthesis efficiency and delays harvest.
- **CO_2 levels**: Elevated CO_2 boosts photosynthesis up to a point, but excessive enrichment can harm plant physiology and make greenhouses unsafe for workers.

This project explores the design of a greenhouse Cyber-Physical System (CPS) with a focus on both functionality and security. By combining UPPAAL formal verification with a Python-based PLC simulation, we aimed to model, test, and validate the system's performance under normal and adversarial conditions. In doing so, the study highlights not only the core aspects of CPS design for agricultural automation but also the critical importance of resilience and security against potential attacks.

Methods

Implementation

To begin designing a Cyber-Physical System (CPS) for the greenhouse, we followed a step-by-step approach that balanced realism with practicality. The goal was to create a digital environment that captures sensor-actuator interactions, supports monitoring, and allows us to simulate both normal operations and attacks.

We started by breaking down the system into smaller programmable components, each representing a physical element of the greenhouse - such as temperature control, CO_2 regulation, lighting, and moisture. Each component was modelled as an individual PLC to mimic real-world modularity. This made it easier to simulate sensor behaviours and actuator responses without overcomplicating the initial design.

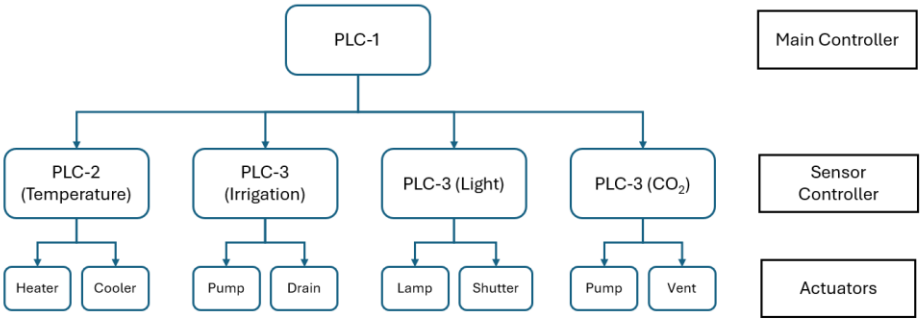

Fig 1 - Overview of System

At the same time, we considered future integration points: a centralized PLC to coordinate individual units, connections to a database for storing sensor data, and front-end dashboards for visualization. This incremental, modular approach let us validate each stage (simulation, data handling, visualization) before moving to more complex aspects like attacks and detection.

The implementation acted as a foundation for subsequent stages - UPPAAL modelling, Python PLCs, database pipelines, and UI dashboards. By starting simple and layering components step by step, we ensured that the CPS stayed manageable, flexible, and open to experimentation.

UPPAAL

To formally verify the behaviour of our greenhouse system, we built a model in UPPAAL. The goal was to design a complete system where the individual PLCs are working with each other, giving us a high-level understanding of it. It also captures how the central PLC interacts with individual PLCs (for water, temperature, light, and CO_2), along with their associated sensors and actuators.

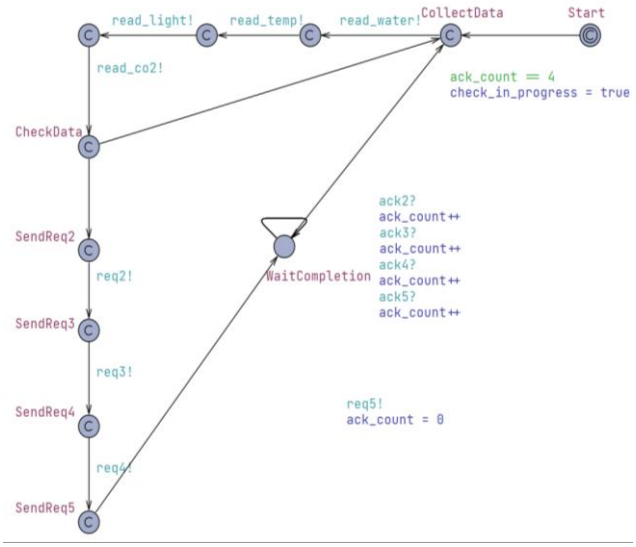

Fig 2 - UPPAAL main system model of PLC 1

Each PLC was modelled as a timed loop sub-system that receives requests, checks its parameter against thresholds, and either acknowledges or enters an adjusting state to bring the variable back into range. Sensors update values periodically, while actuators reset them to within the range when adjustments are triggered. The central PLC (PLC1) coordinates this process by collecting sensor data, checking conditions, and dispatching requests to other PLCs when needed.

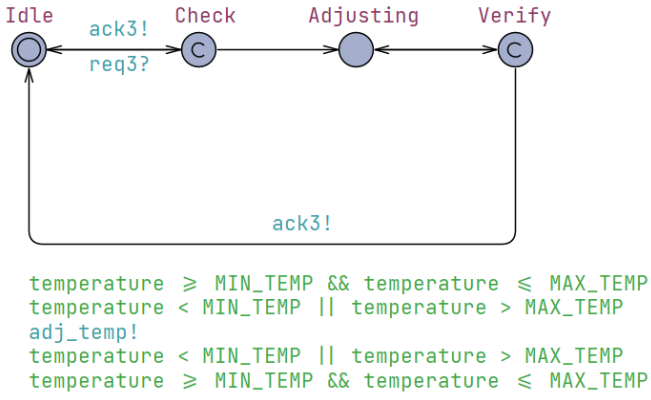

Fig 3 - UPPAAL system model of PLC 3

One key limitation we faced was parallelism. While in reality PLCs operate concurrently, our UPPAAL model only worked in a sequential manner. As a result, the system in UPPAAL cycles through each PLC one by one rather than running them all simultaneously. Despite this, the model was still useful for validating the logic flow and ensuring that all PLCs eventually return their variables within thresholds.

We used queries to check for correctness, focusing on three main aspects:

- **Threshold compliance** - water, temperature, light, and CO_2 values eventually return within the defined safe ranges.

- **Controller responsiveness** - each PLC can enter an adjusting state when its variable goes out of bounds.
- **Synchronization** - sensors remain active only during data collection phases, ensuring consistency.

This exercise gave us confidence in the overall design and highlighted where theoretical guarantees align or diverge from practical implementations.

PLC Implementation (Temperature Control on Training SKID)

To bridge the gap between simulation and a real-world setup, we implemented the temperature control loop on a Training SKID. This allowed us to observe how a PLC responds to live sensor input and actuator logic, and to compare theoretical behaviour with practical execution.

The PLC was configured with thresholds between 25-27 °C, with four actuator outputs:

- Low-power heater (Temp < ThresTemp1)
- High-power heater (Temp < ThresTemp1 - 1)
- Low-power cooler (Temp > ThresTemp2)
- High-power cooler (Temp > ThresTemp2 + 1)

Each scan cycle involved reading the sensor value, checking against thresholds, and activating the appropriate flag and actuator outputs. This design ensured stable temperature regulation while allowing us to observe realistic overshoot, undershoot, and actuator switching.

Demonstration Results

Here's a mapping of each button/output with the actuator:

Fig 4 - Allen-Bradley PLC Training SKID with Digital Outputs and Temperature Sensor

- D0_00 (Not Operational)
- **D0_01** - Below Threshold Indicator Flag
- **D0_02** - Low Power Heater (if Temp < ThresTemp1)
- **D0_03** - High Power Heater (if Temp < ThresTemp1 - 1)
- **D0_04** - Above Threshold Indicator Flag
- D0_05 (Not Operational)

- **D0_06** - Low Power Cooler (if Temp > ThresTemp2)
- **D0_07** - High Power Cooler (if Temp > ThresTemp2 + 1)

Fig 5 - Low-power actuator activation

When the temperature drifted slightly outside the defined range, the PLC flagged the deviation (D0_01 or D0_04) and activated the low-power heater/cooler (D0_02 or D0_06). This demonstrated normal corrective response.

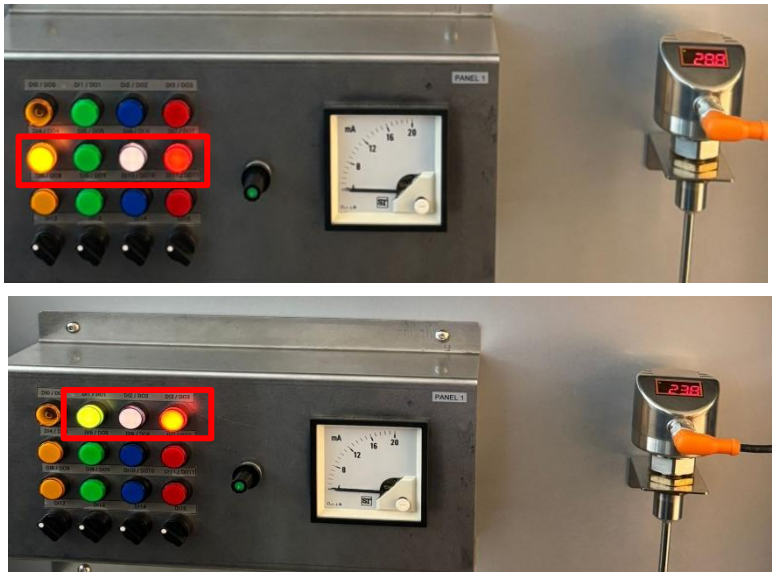

Fig 6 - High-power actuator engagement

At more extreme deviations (e.g., below ThresTemp1 - 1, around 22.8 °C), the PLC triggered both the low-power and high-power actuators (third button lit). This showed the system's ability to escalate responses when needed.

Fig 7 - Stable threshold condition

When the temperature was within 25-27 °C (e.g., at 25.0 °C), the PLC raised no flags and actuators remained idle. This steady-state condition represents the greenhouse's normal operation.

Attack Vector and Vulnerability

While the logic reliably managed thresholds under normal operation, it also revealed a vulnerability. Using the attack script, an adversary could overwrite PLC thresholds (ThresTemp1, ThresTemp2) via unsecured Modbus/TCP. This forces the PLC to interpret unsafe temperatures as "within range," disabling actuator responses. In practice, this means the same flags and actuators shown above would remain idle even under extreme conditions - making the system appear safe while crops are stressed or equipment overheats.

Our attack script here uses pycomm's ControlLogix driver to connect via IP, then pulls both the live temperature (Temp1) and the setpoint tags the PLC logic relies on.

```
spoof_thres1 = float(raw_input("Enter spoofed Thres1 value to write: "))
spoof_thres2 = float(raw_input("Enter spoofed Thres2 value to write: "))

# Overwrite the control thresholds
plc.write_tag('ThresTemp1', spoof_thres1, 'REAL')
plc.write_tag('ThresTemp2', spoof_thres2, 'REAL')
print("Spoofed thresholds applied:", spoof_thres1, spoof_thres2)
plc.close()
```

The PLC utilizes ThresTemp1 & 2 (which are declared Global Tags), and these are granted read/write permissions. This gives us access to overwrite their values easily.

By raising ThresTemp1 & 2 just beyond the current process value - and keeping them "reasonable" - the normal control routine won't trigger cooling/heating or alarms, even as the room drifts out of spec. The code targets the decision variables directly, which is stealthier than hammering.

Python Implementation

After validating our logic with UPPAAL and testing PLC behaviour on the Training SKID, we moved to a full Python-based implementation. The motivation was to build a flexible, software-driven CPS where we could simulate sensors, actuators, and controllers while also integrating databases, dashboards, and attack/detection mechanisms. Unlike UPPAAL, which ran sequentially, Python allowed us to model parallel execution of all PLCs, closer to real-world behaviour.

We designed the system around five Python modules, each representing a PLC. Showcasing their behaviour and code snippet

PLC1 (Collector):

- Acts as the coordinator. It polls sensor values, compares them against thresholds, and issues requests to the respective PLCs.

```
if temperature < MIN_TEMP or temperature > MAX_TEMP:
    print("⚠ Temp out of range, requesting PLC3")
    # request climate control adjustment
```

PLC2 (Irrigation):

- Manages water levels by activating/deactivating irrigation actuators.
- Selected threshold values: 30 - 70%

```
if water_level < MIN_WATER:
    water_level += 1  # simulate irrigation
elif water_level > MAX_WATER:
    water_level -= 1  # reduce watering
```

PLC3 (Climate):

- Controls temperature using simulated heaters and coolers.
- Selected threshold values: 24-28 °C, Max Range: 0-50 °C

```
if temperature < MIN_TEMP:
    temperature += 1  # heater ON
elif temperature > MAX_TEMP:
    temperature -= 1  # cooler ON
```

PLC4 (Light):

- Adjusts artificial lighting to maintain required intensity.
- Selected threshold values: 400-600 lux, Max Range: 0-1000 lux

```
if light < MIN_LIGHT:
    light += 1  # turn lights ON
elif light > MAX_LIGHT:
    light -= 1  # dim/turn OFF
```

PLC5 (CO_2):

- Regulates CO_2 percentage through vents or injectors.
- Selected threshold values: 400-700 ppm, Max Range: 0-1000 ppm

```
if co2 < MIN_CO2:
    co2 += 1  # inject CO₂
```

```
elif co2 > MAX_CO2:
    co2 -= 1  # vent out
```

Each PLC was implemented as a Python script (temp_plc.py, irr_plc.py, light_plc.py, co2_plc.py, and plc1_collector.py) with its own update loop. They exchanged data via the collector module, creating a coordinated feedback system that mirrors an industrial greenhouse.

Override Mechanism

Alongside normal threshold-based behaviour, each PLC was extended with an override mechanism. This feature let us write over the actual sensor data into InfluxDB and directly force actuator outputs, either by supplying a constant value or selecting a random value between a range of values, shifting every cycle.

This feature parallels the manual override options found in real-life CPS, wherein an engineer would need to assume control of the system to possibly fix behaviour quickly, or to simulate attacks in real-time.

Taking Temperature as an example, we can see the implementation of the override system.

```
import random

if not override_active:
    # Normal operation
    if temperature < MIN_TEMP:
        temperature += 1   # heater ON
    elif temperature > MAX_TEMP:
        temperature -= 1   # cooler ON

else:
    # Override mode
    if override_mode == "constant":
        temperature = override_value
    elif override_mode == "range":
        temperature = random.uniform(override_min, override_max)

    print(f"[OVERRIDE] Forcing temperature to {temperature:.2f} °C")
```

In addition to this, the override system will be controlled via the front-end GUI.

InfluxDB & Grafana Integration

To make the CPS observable end-to-end, we streamed each PLC signal as well as alerts into InfluxDB which is a time-series database system and visualized it in Grafana.

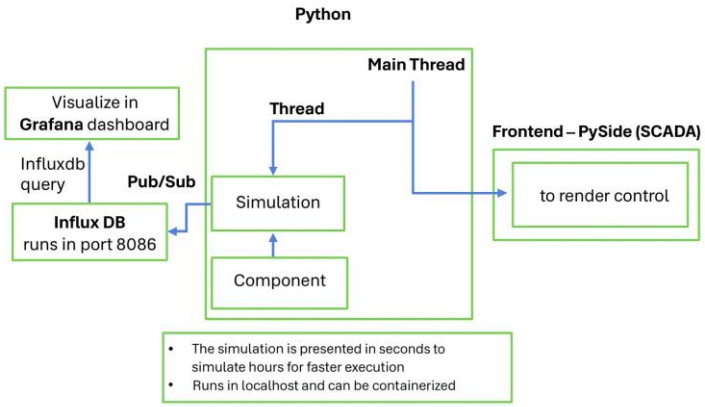

Fig 8 - Flow Diagram of Python Implementation

Benefits of using these include:

- live feedback during demos
- a persistent history for analysis/ML
- a single place to spot attacks or anomalies.

Data is written to InfluxDB using the following method:

```
from influxdb_client import InfluxDBClient, Point, WritePrecision
import os, time

client = InfluxDBClient(
    url=os.getenv("INFLUXDB_URL"),
    token=os.getenv("INFLUXDB_TOKEN"),
    org=os.getenv("INFLUXDB_ORG"),
)
write_api = client.write_api()

def write_signal(plc, signal, value, location="greenhouse_room_1"):
    p = (Point("greenhouse")
         .tag("plc", plc).tag("signal", signal).tag("location", location)
         .field("value", float(value))
         .time(time.time_ns(), WritePrecision.NS))
    write_api.write(bucket=os.getenv("INFLUXDB_BUCKET"), record=p)
```

- We used environment variables to load INFLUXDB_URL, INFLUXDB_ORG, INFLUXDB_BUCKET and a token (as set up in the InfluxDB initial setup)
- Each PLC loop writes the measured value(s) and tags for plc, signal, and location (which is greenhouse_room_1 in this case). We are also writing system alerts flagged by the collector PLC 1.
- We keep writes lightweight (one point per scan per signal) to ensure smooth dashboards and fast queries

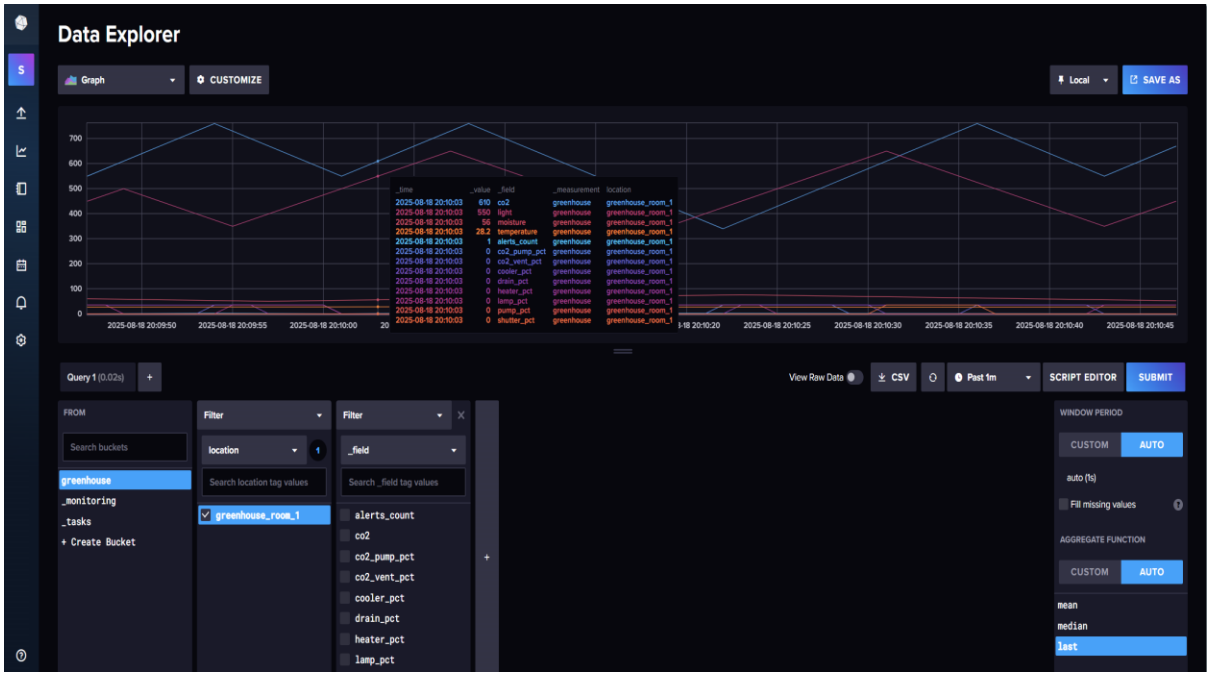

Fig 9 - InfluxDB Dashboard of Greenhouse and the values of sensors, actuators and alerts

We are able to view the writes to InfluxDB in real time using the console, where we can use the dashboard to view all recorded metrics. We have also built in a custom logging function that will record a snippet of sensor outputs for 5 or 10 minutes by duplicating the records from greenhouse_room_1 into greenhouse_room_2.

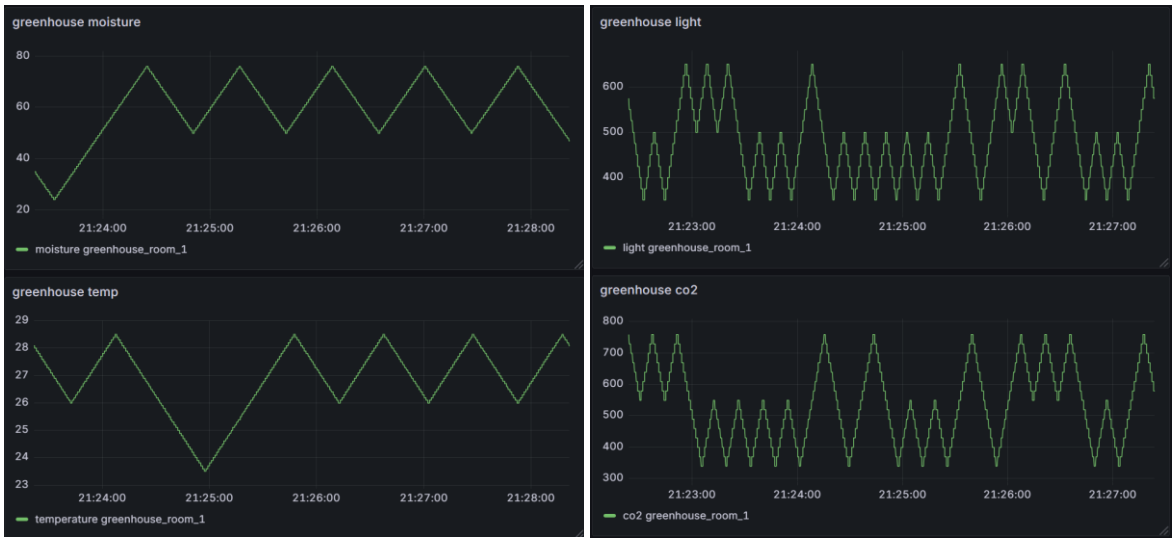

Fig 10 - Showing Grafana graphs

For Grafana, we then used InfluxDB as the data source. With this we set up time-based graph panels for each PLC signal (Temp, CO_2, Light, Irrigation). It is possible to set up alerts on the dashboard to highlight abnormal behaviour.

Building a Frontend (PySide)

For this project, our goal included building a robust and fully featured front end that offered the following functionalities:

- A dashboard offering oversight over all the PLCs, which allows monitoring sensor and actuator behaviour
- Ability to view system alerts that are currently detected
- An override function that allows us to manually override sensor output and affect actuator functions in real-time, in this case simulating an attack
- A subscribe/duplicate function that offers 5 or 10 minute capture of data streams to InfluxDB that is then duplicated into a separate location for easy access
- An attack detection mechanism that checks in real-time

We first attempted to build this using the Electron framework. However, it did not completely satisfy our requirements, especially due to the complexity and time spent troubleshooting to fix sync issues with InfluxDB. We therefore moved to PySide, a python-native implementation of the QT framework that allowed us to build a more responsive GUI.

With PySide, we were able to directly connect with our CPS python files instead of using a Node.js bridge, directly reducing performance overhead. We were also able to sync better with the system, refreshing the dashboard every second.

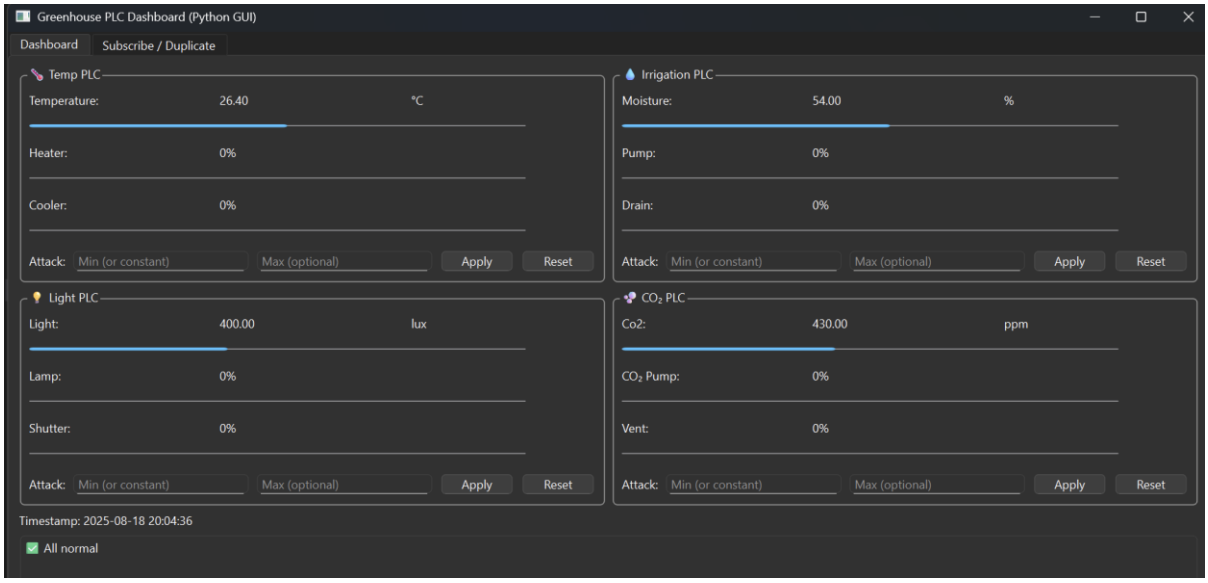

Fig 11 - PySide GUI Dashboard

With the Live Capture option, we can now start 5 or 10 minutes of data capture from the sensors via InfluxDB. Once the capture is started, data incoming into InfluxDB is cloned from the original location i.e. greenhouse_room_1 to greenhouse_room_2.

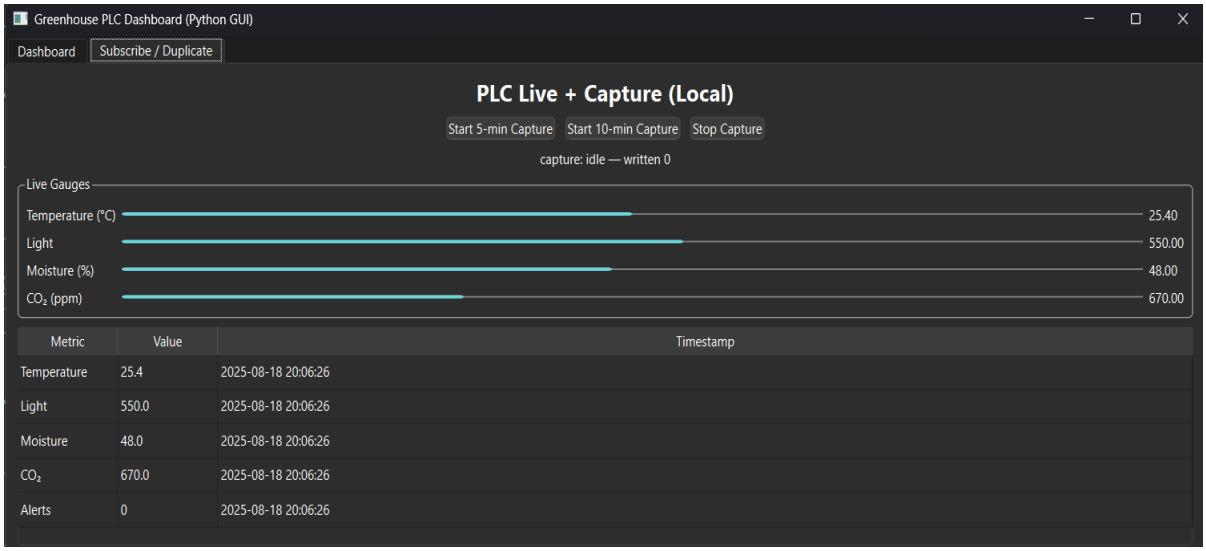

Fig 12 - Live Capture tab in GUI

Attack Method

For our attack demonstration, we leveraged the override functionality built directly into the Python PLC implementation and exposed through the PySide GUI. While originally intended as a testing feature, this mechanism effectively simulates how an attacker with access to the system could disrupt normal control logic.

The attack process:

- The attacker takes advantage of the override function in the following ways:
    - Force a constant value (e.g., lock temperature at 50 °C regardless of real fluctuations)

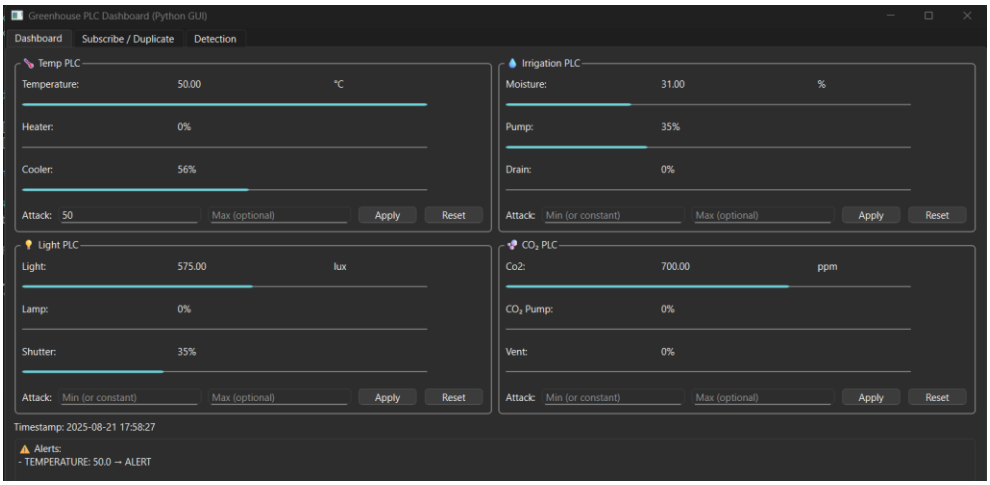

Fig 13 - Attack Method (constant value)

    - Define a range override (e.g., random values between 20-40 °C to mimic “normal” variability). In this example, temperature values fluctuate in and out of the threshold values. Below are examples of the values triggering varying actuator responses.

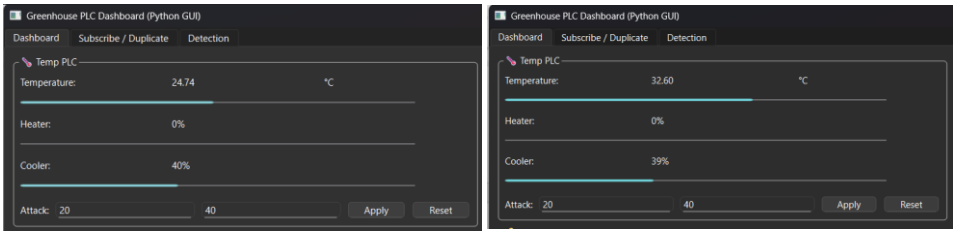

Fig 14 - Attack Method (random value within range)

- Once active, these overrides bypassed the PLC's threshold checks and actuator logic, replacing real sensor values with attacker-controlled inputs.

Detection

For our detection, we designed the detection to identify abnormal behaviour and potential cyber-physical attacks within the PLC-controlled environment. A machine learning model trained on collected sensor and actuator data continuously monitors the live data stream stored in InfluxDB. This model classifies each incoming instance as either normal or attack based on learned patterns of system operation. By integrating the detector into the data pipeline, the system is able to raise alerts in real-time whenever malicious activity is detected, while maintaining low latency in normal monitoring. The detection process ensures that deviations such as manipulated sensor readings or unusual timing behaviours are flagged. This enhances the resilience of the greenhouse CPS by complementing preventive security measures with active anomaly detection.

Training the detection model

We first started with building a dataset using realtime data from our system. Using our Python PLC implementation, we ran the system under both normal operation and attack scenarios. We use the override function to generate abnormal, "attack-type" behaviour in this case. This data is labelled as either Normal (0) or Attack (1). Using the collect_labeled_data.py script, we do it all and compile the data into a CSV file.

This dataset was used to train a Random Forest classifier because it is:

- Robust to noisy data and works well on structured features like sensor readings.
- Interpretable - we can easily check feature importance (e.g., which sensor values contribute most to detection).
- Proven - widely used in anomaly detection tasks with good generalization.

Using train_model.py, we generate a serialized model file that can be used in the detection process.

Performing Detection on System

Once trained, the model was integrated into a live detection pipeline (detect_attack.py). The detector queries recent windows of sensor data from InfluxDB, feeds them to the classifier, and prints out alerts in real time.

```python
# Batch prediction
for i in range(0, len(X_live), BATCH_SIZE):
    batch = X_live.iloc[i:i+BATCH_SIZE]
    preds = clf.predict(batch)
    for t, p in zip(batch_times, preds):
        if p == 1:
            print(f"[ATTACK] Detected at {t}")
        else:
            print(f"[NORMAL] at {t}")
```

- Normal behaviour is labelled and displayed as [NORMAL].
- Attack behaviour triggers [ATTACK] alerts, timestamped to the exact point in the sensor stream.

We then forward this output and display it in our PySide GUI under the Detection Tab.

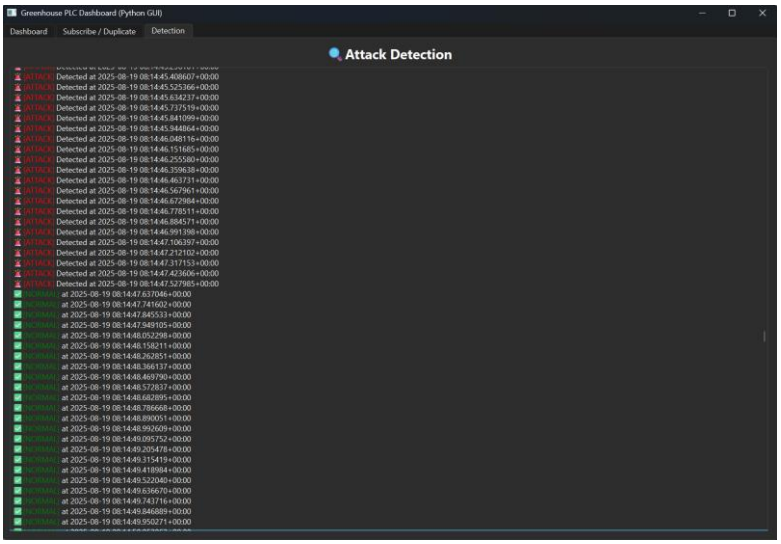

Fig 15 - Detection Tab in GUI

Analysis

Benefits of Formal Verification using UPPAAL

Modeling the greenhouse CPS in UPPAAL before implementing it in Python provided several advantages.

- UPPAAL allowed us to formally capture the system's timing, synchronization, and communication between multiple PLCs in a structured and verifiable way.
- Through model checking, we could verify important properties such as deadlock freedom, timely sensor-to-actuator responses, and correct fault-handling behaviour under different conditions.
- This early-stage verification helped identify design flaws and inconsistencies before investing effort in coding the full Python-based simulation. By starting with UPPAAL, we ensured that the system architecture was logically sound, and we gained confidence that the control flows and safety properties were preserved.
- We also were able to trace down any problems easily, allowing us to debug faster.
- Moving to Python afterwards enabled us to focus on realism, integration with InfluxDB, and live data handling, while building upon a validated and well-structured model.

While we weren't able to implement the UPPAAL model with a parallelized process cycle, instead relying on a sequential acknowledgement cycle, this proved to help us finalize the acceptance criteria and build a base template of the working PLCs. This resulted in a much more informed module-based python implementation.

Choosing PySide over Electron

Initially, Electron was used to develop the greenhouse dashboard due to its flexibility in creating modern web-based interfaces. However, as the project evolved, several limitations became apparent. Electron introduced significant overhead in terms of performance and resource usage, which was not ideal for a lightweight monitoring and control system. Furthermore, integration with the Python-based PLC simulations required additional layers of communication (such as WebSockets or APIs), which increased complexity and reduced reliability.

By shifting to PySide, we were able to build a native desktop interface directly in Python, ensuring tighter integration with the existing codebase, simpler communication with the PLC processes, and reduced system dependencies. This transition not only improved efficiency and maintainability but also made the dashboard easier to extend with features such as real-time detection logs and alert visualization.

Advantages of InfluxDB

InfluxDB was selected as the time-series database for the greenhouse project due to its efficiency in handling high-frequency sensor and actuator data. Unlike traditional relational databases, InfluxDB is optimized for time-stamped data, making it well-suited for continuously streaming measurements such as temperature, humidity, CO_2, and actuator states. Its lightweight write operations allow real-time data ingestion with minimal latency, while its query language (Flux) provides powerful tools for aggregation, anomaly detection, and trend analysis over time windows.

InfluxDB also supports retention policies and downsampling, enabling efficient long-term storage without overwhelming system resources. These features make it particularly advantageous for a cyber-physical system like the greenhouse, where accurate, scalable, and fast time-series data management is critical for both monitoring and security detection.

Feasibility of Attack Method

The feasibility of the attack method in this project lies in demonstrating how cyber-physical systems such as the greenhouse can be vulnerable to sensor override attacks. The simulated attack targets the data pipeline between the sensors and the PLCs, where readings are continuously transmitted at regular intervals. By overriding or replacing this stream with falsified but plausible values, an attacker can mask the true state of the environment and mislead the control logic.

In a real-world scenario, such an attack could be carried out by compromising a sensor node, reprogramming a PLC input routine, or injecting false values at the data collection point. For example, fixed temperature readings around 25 °C would prevent the system from detecting overheating or cooling events, causing the actuators to respond incorrectly and potentially leading to crop damage or system instability. The attack is therefore feasible because it exploits the inherent trust in sensor data, demonstrates how system integrity can be undermined, and highlights the need for data validation, redundancy, and anomaly detection in CPS environments.

Feasibility of Detection Mechanism

The detection method is supported by the ability to monitor continuous greenhouse data streams and classify system states in real time. By training a machine learning model on both normal operation and attack scenarios, the detector is capable of distinguishing between legitimate sensor behaviour and overridden values that appear suspicious. Since the model is integrated directly into the InfluxDB pipeline, detection can be performed with minimal latency, making it practical for live monitoring.

With Weka, the accuracy of the detection was evaluated using the same train/test split, where performance metrics such as precision, recall, and F1-score indicated that the

model could reliably identify abnormal patterns without producing excessive false alarms. This balance between feasibility and accuracy ensures that the detection method is not only technically implementable but also effective at providing meaningful security insights in the greenhouse CPS environment.

Specifically, we created a dataset with normal and attack behaviour data collected from system runtimes. To avoid data leakage, we first removed duplicates from the dataset and ran a Random Forest Classifier with a train-test split of 80:20. This gave us the below results:

```
=== Run information ===

Scheme:       weka.classifiers.trees.RandomForest -P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1
Relation:     plc_labeled_data-weka.filters.unsupervised.attribute.NumericToNominal-Rlast-weka.filters.unsupervised.instance.RemoveDuplicates
Instances:    7548
Attributes:   5
              temperature
              moisture
              co2
              light
              label
Test mode:    split 80.0% train, remainder test

=== Classifier model (full training set) ===

RandomForest

Bagging with 100 iterations and base learner

weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Time taken to build model: 0.96 seconds

=== Evaluation on test split ===

Time taken to test model on test split: 0.03 seconds

=== Summary ===

Correctly Classified Instances        1507               99.8013 %
Incorrectly Classified Instances         3                0.1987 %
Kappa statistic                          0.996
Mean absolute error                      0.0023
Root mean squared error                  0.0407
Relative absolute error                  0.453  %
Root relative squared error              8.131  %
Total Number of Instances             1510

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.004    0.996      1.000    0.998      0.996    0.999     0.999     0
                 0.996    0.000    1.000      0.996    0.998      0.996    0.999     0.999     1
Weighted Avg.    0.998    0.002    0.998      0.998    0.998      0.996    0.999     0.999

=== Confusion Matrix ===

   a   b   <-- classified as
 743   0 |   a = 0
   3 764 |   b = 1
```

Fig 16 - Random Forest classifier analysis using Weka

From the above metrics, we are able to observe a very high accuracy of 99.8% with 3 false negatives detected. The ROC Area is also calculated at 0.999, which is very close to accurate and indicates an almost perfect distinction between Normal and Attack behaviour.

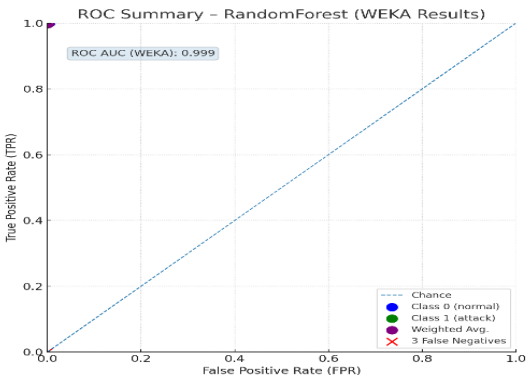

Fig 17 - ROC highlighting Area under Curve

Results

In this project, we evaluated the behaviour of our greenhouse CPS under both normal and adversarial conditions.

System Behavior

Under normal operation, the greenhouse maintained stable temperature cycles within the safe range of **24 °C to 28 °C**, with sensors and actuators working together to regulate the environment.

When a **sensor override attack** was introduced, the reported sensor value was fixed at 25 °C while the actual temperature continued to rise. Since the actuators relied on spoofed data, corrective actions were suppressed, creating a growing divergence between the true and reported system states.

Attack Success

The attack was initially successful in misleading the monitoring system and preventing appropriate actuator response. This led to overheating beyond the safe threshold, demonstrating that false data injection and spoofing can compromise system reliability.

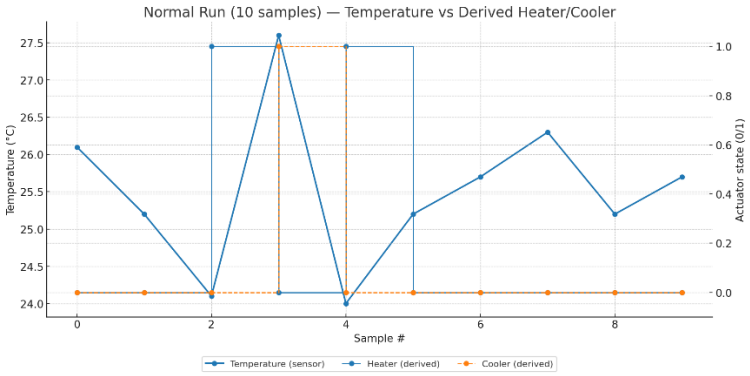

Fig 18 - Normal Sensor and Actuator Behaviour

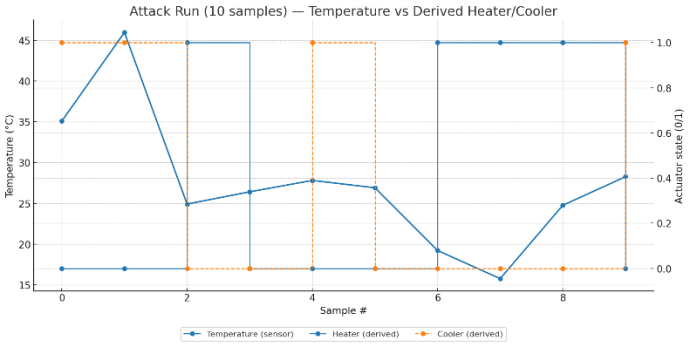

Fig 19 - Sensor and Actuator Behaviour during Attack

Detection Performance

The anomaly detection mechanism was able to identify the attack once the actual temperature exceeded the upper threshold of 28 °C, even though the spoofed readings appeared normal. This shows that while spoofing can temporarily evade detection, physical system drift eventually exposes abnormal conditions, allowing the monitoring logic to flag the issue.

Challenges

Below are some of the major challenges we found through the lifecycle of this project:

Electron Front-End GUI

Our initial dashboard was built in Electron, but it quickly became a bottleneck. We struggled with refresh rates that were either too fast, leading to jerky updates, or too slow, leaving the interface frozen. Debugging across Node.js and Python processes added friction, and small differences in environment setup caused inconsistent behaviour between team members. This led us to abandon Electron in favour of PySide, which integrated far more smoothly with the Python-based PLCs.

InfluxDB Integration

While InfluxDB gave us a robust time-series store, integration was not always straightforward. We ran into authentication issues (401 errors) until environment variables were configured consistently across machines. Querying also required care: raw data streams could be noisy, and pulling large ranges sometimes caused delays. We had to learn how to optimize Flux queries and manage buckets to keep dashboards responsive and training datasets consistent.

Machine Learning & Detection

The detection pipeline worked well for our demo dataset, but it highlighted the limits of a narrow training scope. Because our labels were only based on override attacks, the Random Forest model achieved near-perfect accuracy by treating anything outside thresholds as “attack” and everything inside as “normal.” While this was effective in our scenario, it is not enough for a general-purpose detector. Building a robust model would require more diverse training data - covering different attack strategies, sensor faults, actuator failures, and natural fluctuations. Another challenge was balancing real-time detection with minimal false positives; our current system works almost instantaneously, but its reliability under different greenhouse conditions remains untested.

General Integration

Finally, coordinating multiple moving parts - formal models, SKID PLCs, Python simulations, InfluxDB, Grafana, and PySide - was itself a challenge. Each component had different assumptions and interfaces. Getting them to work together required careful synchronization and often trial-and-error debugging.

Lessons Learned

These challenges reinforced the importance of incremental development and validation. By starting with UPPAAL and moving gradually through hardware and software stages, we were able to identify weaknesses early and adapt. They also highlighted that building a CPS is as much about managing the supporting tools and infrastructure as it is about the control logic itself. Most importantly, we learned that designing detection systems requires thinking beyond a single attack type - true robustness comes from anticipating variety and building layers of defence.

Future Improvements

In this project, we demonstrated how false data injection and sensor spoofing could disrupt greenhouse CPS operation. While our detection mechanism based on threshold monitoring successfully flagged anomalies once the physical temperature exceeded safe bounds, there are several areas where the system could be improved:

- **Resilience Enhancements**:
  Currently, detection relies on threshold-based anomaly triggers. In the future, we can integrate more advanced resilience strategies such as adaptive thresholds, redundant sensing, or cross-checking data from multiple sensor types. This would reduce reliance on a single data source and make spoofing attacks harder to succeed.
- **Scalability of Detection**:
  Our ML-based detection showed high accuracy within the simulated setup. However, scaling to larger greenhouse environments or multiple CPS deployments would require distributed monitoring and load balancing of detection models. Exploring lightweight ML models or edge-based detection can make the system more scalable in practice.
- **Real-World Integration**:
  We tested our concepts in simulation. Future work could involve deploying the model on actual hardware (Raspberry Pi or PLCs) and validating against real sensor-actuator feedback loops to bridge the gap between simulation and real-world CPS resilience.
- **Extended Security Coverage**:
  While we focused on false data injection and sensor spoofing, other attacks (e.g., denial-of-service on actuators, replay attacks, or communication hijacking) could be studied. This would provide a more complete evaluation of system robustness against diverse adversaries.

By addressing these areas, the system can evolve from a functional simulation into a more **resilient, scalable, and field-ready CPS** suitable for smart agriculture.

Conclusion

In this project, we designed and implemented a greenhouse Cyber-Physical System (CPS) using both UPPAAL formal verification and Python-based PLC simulation. We first established the system's baseline by simulating normal behaviour, where sensors and actuators worked together to maintain stable environmental conditions between the defined thresholds. This confirmed that our CPS design was functional and reliable under non-adversarial settings.

We then introduced adversarial scenarios, specifically false data injection and sensor spoofing attacks. By overriding sensor outputs with constant falsified values, we observed how the system was misled, preventing actuators from responding appropriately. This led to abnormal system behaviour, such as overheating, and highlighted the vulnerability of CPS when attackers target the sensor-to-monitoring pipeline.

To address this, we implemented detection mechanisms that identified anomalies once the physical state diverged from spoofed readings. Our experiments showed that although spoofed data initially evaded monitoring, physical divergence between actual and reported values eventually triggered anomaly flags. UPPAAL verified theoretical safety violations under attack, while Python simulations demonstrated their real-time impact and the effectiveness of detection.

Overall, this project allowed us to explore functional CPS design while uncovering key insights about resilience and security in agricultural automation. The results confirmed that while attacks can temporarily succeed, carefully designed monitoring and detection mechanisms can limit long-term damage. These findings emphasize the importance of integrating both verification and simulation approaches to build more robust and secure CPS for critical domains like smart agriculture.

Appendix

Project Repository

https://github.com/reddashi/SbD

References

1. Mo, Y., Kim, T. H.-J., Brancik, K., Dickinson, D., Lee, H., Perrig, A., & Sinopoli, B. (2019). A Stealthy Sensor Attack for Uncertain Cyber-Physical Systems
2. Dey, S., Paul, S., & Das, S. K. (2021). A Comprehensive Survey on Attacks and Countermeasures in Cyber-Physical Systems
3. Prathibha, S. R., Hongal, A., & Jyothi, M. P. (2017). IoT Based Monitoring System in Smart Agriculture. 2017 International Conference on Recent Advances in Electronics and Communication Technology (ICRAECT). IEEE. doi:10.1109/ICRAECT.2017.52
4. Barreto, L., & Amaral, A. (2018). Smart Farming: Cyber Security Challenges. 2018 International Conference on Intelligent Systems (IS). IEEE. doi:10.1109/IS.2018.871
5. Huang, M. (2023). Design of Intelligent Greenhouse Control System based on MCGS and PLC. Journal of Physics: Conference Series, 2510(1), 012022. IOP Publishing. doi:10.1088/1742-6596/2510/1/012022

Program Codes

Training SKID PLC Program

```
IF S:FS THEN
       ThresTemp1:= 25.0;
       ThresTemp2:= 27.0;

END_IF;

// Turn on heater DO_02 if temperature is below 25°C
IF Temp1 < ThresTemp1 THEN
 DO_01 := 1;
ELSE
 DO_01 := 0;
END_IF;

// Turn on cooler DO_04 if temperature is above 27°C
IF Temp1 > ThresTemp2 THEN
 DO_04 := 1;
ELSE
 DO_04 := 0;
END_IF;

// acting as a low power heater
IF Temp1 < ThresTemp1 THEN
 DO_02 := 1;
ELSE
 DO_02 := 0;
END_IF;
```

```
// acting as a high power heater
IF Temp1 < (ThresTemp1 - 1.0) THEN
 DO_03 := 1;
ELSE
 DO_03 := 0;
END_IF;

// acting as a low power cooler
IF Temp1 > ThresTemp2 THEN
  DO_06 := 1;
  ELSE
  DO_06 := 0;
END_IF;

// acting as a high power cooler
IF Temp1 > (ThresTemp2 + 1.0) THEN
  DO_07 := 1;
  ELSE
  DO_07 := 0;
END_IF;
```

Training SKID PLC Attack Code

```
#!/usr/bin/env python2
import time
from pycomm.ab_comm.clx import Driver as ClxDriver

PLC_IPS = {
    'greenhouse_plc': '192.168.1.151',
}

def spoof_temperature():
    plc = ClxDriver()
    if plc.open(PLC_IPS['greenhouse_plc']):
        # Read current Temp1
        temp1_val = plc.read_tag('Temp1')
        if temp1_val[0] is not None:
            print("Current Temp1 value: {:.2f}".format(temp1_val[0]))
        else:
            print("Failed to read Temp1")
            plc.close()
            return

        # Read current thresholds
        thres1 = plc.read_tag('ThresTemp1')[0]
        thres2 = plc.read_tag('ThresTemp2')[0]

        print("Current thresholds: ThresTemp1 = {:.2f}, ThresTemp2 =
{:.2f}".format(thres1, thres2))

        # Ask user for spoofed Temp value
        try:
            print("Temp is OUTSIDE thresholds.")
            spoof_thres1 = float(raw_input("Enter spoofed Thres1 value to write:
"))
            spoof_thres2 = float(raw_input("Enter spoofed Thres2 value to write:
"))
            print(plc.write_tag('ThresTemp1', spoof_thres1, 'REAL'))
            print(plc.write_tag('ThresTemp2', spoof_thres2, 'REAL'))
```

```
            print("Spoofed ThresTemp written:", spoof_thres1, " , ", spoof_thres2)

        except ValueError:
            print("Invalid input. Must be a number.")

        plc.close()
    else:
        print("Unable to connect to PLC")

if __name__ == '__main__':
    spoof_temperature()
```

PLC1 (Collector)

```
# plc1_collector.py
import threading
import time
import json
import sys
import os
import random

from influxdb_client import InfluxDBClient, Point, WritePrecision
from influxdb_client.client.write_api import SYNCHRONOUS

from temp_plc import TemperaturePLC, MIN_TEMP, MAX_TEMP
from light_plc import LightPLC, MIN_LIGHT, MAX_LIGHT
from irr_plc import IrrigationPLC, MIN_MOISTURE, MAX_MOISTURE
from co2_plc import CO2PLC, MIN_CO2, MAX_CO2

# --- InfluxDB setup ---
token  = os.environ.get("INFLUXDB_TOKEN") or os.environ.get("INFLUX_TOKEN")
org    = os.environ.get("INFLUXDB_ORG") or os.environ.get("INFLUX_ORG") or "SUTD"
bucket = os.environ.get("INFLUXDB_BUCKET") or os.environ.get("INFLUX_BUCKET") or
"greenhouse"
url    = os.environ.get("INFLUXDB_URL") or os.environ.get("INFLUX_URL") or
"http://localhost:8086"

client    = InfluxDBClient(url=url, token=token, org=org)
write_api = client.write_api(write_options=SYNCHRONOUS)

# --- Thresholds ---
THRESHOLDS = {
    "temperature": (MIN_TEMP, MAX_TEMP),
    "light":       (MIN_LIGHT, MAX_LIGHT),
    "moisture":    (MIN_MOISTURE, MAX_MOISTURE),
    "co2":         (MIN_CO2, MAX_CO2),
}

# Latest packets from each PLC
sensor_values = {"temp": None, "light": None, "irrigation": None, "co2": None}

# IMPORTANT: PLCs read this dict expecting NUMBERS. Keep it numeric.
overrides = {}            # e.g. {"temperature": 25.0}
range_overrides = {}      # e.g. {"temperature": (20.0, 25.0)}
_overrides_lock = threading.Lock()

# --- helpers ---
def _normalize_sensor_key(k: str) -> str:
```

```
    k = (k or "").lower()
    if k in ("temp", "temperature"):
        return "temperature"
    if k in ("moist", "moisture", "irrigation"):
        return "moisture"
    if k == "light":
        return "light"
    if k in ("co2", "carbon", "carbon_dioxide"):
        return "co2"
    return k

def _sender(slot_key):
    def send(packet: dict):
        sensor_values[slot_key] = packet
    return send

# --- stdin command listener (minimal change; adds override_range) ---
def stdin_listener():
    for line in sys.stdin:
        line = line.strip()
        if not line:
            continue
        try:
            cmd = json.loads(line)
        except Exception as e:
            print(f"STDIN JSON error: {e}", file=sys.stderr, flush=True)
            continue

        ctype = cmd.get("type")
        s_key = _normalize_sensor_key(cmd.get("sensor"))

        with _overrides_lock:
            if ctype == "override":
                # constant override stays numeric; clear any existing range
                try:
                    overrides[s_key] = float(cmd["value"])
                    range_overrides.pop(s_key, None)
                except Exception as e:
                    print(f"override parse error for {s_key}: {e}",
file=sys.stderr, flush=True)

            elif ctype == "override_range":
                # store the range and also set an initial numeric sample
                try:
                    vmin = float(cmd["min"])
                    vmax = float(cmd["max"])
                    if vmin > vmax:
                        vmin, vmax = vmax, vmin
                    range_overrides[s_key] = (vmin, vmax)
                    overrides[s_key] = random.uniform(vmin, vmax)  # numeric for
PLCs
                except Exception as e:
                    print(f"override_range parse error for {s_key}: {e}",
file=sys.stderr, flush=True)

            elif ctype == "clear_override":
                overrides.pop(s_key, None)
                range_overrides.pop(s_key, None)
            # else ignore unknown types
```

```
def check_alerts(sensors_payload: dict):
    alerts = {}
    for field, (low, high) in THRESHOLDS.items():
        val = sensors_payload.get(field)
        if val is None:
            continue
        try:
            fv = float(val)
        except Exception:
            continue
        if not (low <= fv <= high):
            alerts[field] = {"value": fv, "status": "ALERT", "target": f"PLC for
{field}"}
    return alerts

def run_once(plc):
    plc.run(cycles=1)

def main():
    # Start command reader
    threading.Thread(target=stdin_listener, daemon=True).start()

    # Instantiate PLCs with the numeric 'overrides' dict (unchanged contract)
    workers = [
        TemperaturePLC(sender=_sender("temp"),        overrides=overrides),
        LightPLC(      sender=_sender("light"),       overrides=overrides),
        IrrigationPLC( sender=_sender("irrigation"),  overrides=overrides),
        CO2PLC(        sender=_sender("co2"),         overrides=overrides),
    ]

    while True:
        # --- NEW: for any range overrides, resample a numeric value BEFORE each
cycle
        with _overrides_lock:
            for field, (vmin, vmax) in list(range_overrides.items()):
                overrides[field] = random.uniform(vmin, vmax)

        # Step each PLC once (same as original)
        threads = [threading.Thread(target=run_once, args=(w,)) for w in workers]
        for t in threads: t.start()
        for t in threads: t.join()

        # Build outgoing payload from latest PLC packets
        sensors = {
            "temperature": sensor_values["temp"]["temperature"]         if
sensor_values["temp"]        else 0.0,
            "light":       sensor_values["light"]["light"]              if
sensor_values["light"]       else 0.0,
            "moisture":    sensor_values["irrigation"]["moisture"]      if
sensor_values["irrigation"]  else 0.0,
            "co2":         sensor_values["co2"]["co2"]                  if
sensor_values["co2"]         else 0.0,
            "timestamp":   time.strftime("%Y-%m-%d %H:%M:%S"),
        }

        actuators = {
            "heater_pct":   sensor_values["temp"]["heater_pct"]         if
sensor_values["temp"]        else 0,
```

```
            "cooler_pct":   sensor_values["temp"]["cooler_pct"]          if
sensor_values["temp"]        else 0,
            "lamp_pct":     sensor_values["light"]["lamp_pct"]           if
sensor_values["light"]       else 0,
            "shutter_pct":  sensor_values["light"]["shutter_pct"]        if
sensor_values["light"]       else 0,
            "pump_pct":     sensor_values["irrigation"]["pump_pct"]      if
sensor_values["irrigation"]  else 0,
            "drain_pct":    sensor_values["irrigation"]["drain_pct"]     if
sensor_values["irrigation"]  else 0,
            "co2_pump_pct": sensor_values["co2"]["co2_pump_pct"]         if
sensor_values["co2"]         else 0,
            "co2_vent_pct": sensor_values["co2"]["co2_vent_pct"]         if
sensor_values["co2"]         else 0,
        }

        alerts = check_alerts(sensors)

        output = {"sensors": sensors, "actuators": actuators, "alerts": alerts}
        print(json.dumps(output), flush=True)

        # Influx write
        point = (
            Point("greenhouse")
            .tag("location", "greenhouse_room_1")
            .field("temperature",   float(sensors["temperature"]))
            .field("heater_pct",    float(actuators["heater_pct"]))
            .field("cooler_pct",    float(actuators["cooler_pct"]))
            .field("light",         float(sensors["light"]))
            .field("lamp_pct",      float(actuators["lamp_pct"]))
            .field("shutter_pct",   float(actuators["shutter_pct"]))
            .field("moisture",      float(sensors["moisture"]))
            .field("pump_pct",      float(actuators["pump_pct"]))
            .field("drain_pct",     float(actuators["drain_pct"]))
            .field("co2",           float(sensors["co2"]))
            .field("co2_pump_pct",  float(actuators["co2_pump_pct"]))
            .field("co2_vent_pct",  float(actuators["co2_vent_pct"]))
            .field("alerts_count",  int(len(alerts)))
            .time(time.time_ns(), WritePrecision.NS)
        )
        write_api.write(bucket=bucket, org=org, record=point)

        time.sleep(0.1)

if __name__ == "__main__":
    main()
```

PLC2 (Irrigation)

```
import time
import random
import threading

# Thresholds for soil moisture (in %)
MIN_MOISTURE = 30.0
MAX_MOISTURE = 70.0

# Extended moisture range (simulate drift)
MIN_MOISTURE_EXT = 0.0
```

```
MAX_MOISTURE_EXT = 100.0

MOISTURE_CHANGE_RATE = 1.0  # % per second

class IrrigationPLC:
    def __init__(self, sender=None, overrides=None):
        self.current_moisture = 50.0  # Start mid-range
        self.pump_pct = 0
        self.drain_pct = 0
        self.direction = random.choice([0, 1])  # 0 = drying, 1 = moistening
        self.sender = sender or (lambda data: None)
        self.running = True
        self.sensor_online = False
        self.overrides = overrides if overrides is not None else {}

        threading.Thread(target=self.live_loop, daemon=True).start()

    def live_loop(self):
        time.sleep(0.1)
        self.sensor_online = True

        middle_moisture = (MIN_MOISTURE + MAX_MOISTURE) / 2  # Target middle

        while self.running:
            if self.sensor_online:
                # Apply override if exists
                if "moisture" in self.overrides:
                    self.current_moisture = float(self.overrides["moisture"])
                else:
                    # Apply actuator effects
                    if self.pump_pct > 0:
                        self.current_moisture += MOISTURE_CHANGE_RATE
                        if self.current_moisture >= middle_moisture:
                            self.pump_pct = 0
                            self.direction = random.choice([0, 1])
                    elif self.drain_pct > 0:
                        self.current_moisture -= MOISTURE_CHANGE_RATE
                        if self.current_moisture <= middle_moisture:
                            self.drain_pct = 0
                            self.direction = random.choice([0, 1])
                    else:
                        # Drift naturally if no actuator active
                        if self.direction is None:
                            self.direction = random.choice([0, 1])
                        if self.direction == 0:
                            self.current_moisture -= MOISTURE_CHANGE_RATE
                        else:
                            self.current_moisture += MOISTURE_CHANGE_RATE

                # Actuator trigger logic
                if self.current_moisture < MIN_MOISTURE - 5:
                    diff = (MIN_MOISTURE - self.current_moisture)
                    self.pump_pct = min(100, diff + 29)
                    self.drain_pct = 0
                    self.direction = None
                elif self.current_moisture > MAX_MOISTURE + 5:
                    diff = self.current_moisture - (MAX_MOISTURE)
                    self.pump_pct = 0
                    self.drain_pct = min(100, diff + 29)
```

```
                    self.direction = None

                # Clamp values
                self.current_moisture = max(MIN_MOISTURE_EXT, min(MAX_MOISTURE_EXT,
self.current_moisture))

                # Send status
                self.sender({
                    "moisture": round(self.current_moisture, 2),
                    "pump_pct": self.pump_pct,
                    "drain_pct": self.drain_pct
                })
            else:
                # Sensor offline fallback
                self.sender({
                    "moisture": 0.0,
                    "pump_pct": 0,
                    "drain_pct": 0
                })

            time.sleep(1)

    def run(self, cycles=1):
        pass
```

PLC3 (Temperature/Climate)

```
import time
import random
import threading

# Threshold range (target values to maintain)
MIN_TEMP = 24.0
MAX_TEMP = 28.0

# Extended simulation range
MIN_TEMP_EXT = 0.0
MAX_TEMP_EXT = 50.0

TEMP_CHANGE_RATE = 0.1

class TemperaturePLC:
    def __init__(self, sender=None, overrides=None):
        self.current_temp = 26.0
        self.heater_pct = 0
        self.cooler_pct = 0
        self.direction = random.choice([0, 1])
        self.sender = sender or (lambda data: None)
        self.running = True
        self.sensor_online = False
        self.overrides = overrides if overrides is not None else {}

        threading.Thread(target=self._live_loop, daemon=True).start()

    def _live_loop(self):
        time.sleep(0.1)
        self.sensor_online = True
        target_temp = (MIN_TEMP + MAX_TEMP) / 2
```

```
        while self.running:
            if self.sensor_online:
                # Apply override if exists
                if "temperature" in self.overrides:
                    self.current_temp = float(self.overrides["temperature"])
                else:
                    # Normal actuator/environment logic
                    if self.heater_pct > 0:
                        self.current_temp += TEMP_CHANGE_RATE
                        if self.current_temp >= target_temp:
                            self.heater_pct = 0
                            self.direction = random.choice([0, 1])
                    elif self.cooler_pct > 0:
                        self.current_temp -= TEMP_CHANGE_RATE
                        if self.current_temp <= target_temp:
                            self.cooler_pct = 0
                            self.direction = random.choice([0, 1])
                    else:
                        if self.direction == 0:
                            self.current_temp -= TEMP_CHANGE_RATE
                        else:
                            self.current_temp += TEMP_CHANGE_RATE

                # Actuator logic still applies even if overridden
                if self.current_temp < MIN_TEMP - 0.5:
                    diff = (MIN_TEMP - self.current_temp)
                    self.heater_pct = min(100, diff + 34.5)
                    self.cooler_pct = 0
                    self.direction = None
                elif self.current_temp > MAX_TEMP + 0.5:
                    diff = self.current_temp - (MAX_TEMP)
                    self.cooler_pct = min(100, diff + 34.5)
                    self.heater_pct = 0
                    self.direction = None

                self.current_temp = max(MIN_TEMP_EXT, min(MAX_TEMP_EXT,
self.current_temp))

                self.sender({
                    "temperature": round(self.current_temp, 2),
                    "heater_pct": round(self.heater_pct, 1),
                    "cooler_pct": round(self.cooler_pct, 1)
                })

            else:
                self.sender({
                    "temperature": 0.0,
                    "heater_pct": 0,
                    "cooler_pct": 0
                })

            time.sleep(1)

    def run(self, cycles=1):
        pass
```

PLC4 (Light)

```
import time
import random
import threading

# Threshold range (target light levels in lux)
MIN_LIGHT = 400.0
MAX_LIGHT = 600.0

# Extended simulation range (can drift beyond)
MIN_LIGHT_EXT = 0.0
MAX_LIGHT_EXT = 1000.0

# Lux change per second
LIGHT_CHANGE_RATE = 25.0

class LightPLC:
    def __init__(self, sender=None, overrides=None):
        self.current_light = 500.0  # Start in the middle
        self.lamp_pct = 0           # 0-100% power for lamp
        self.shutter_pct = 0        # 0-100% for blocking light
        self.direction = random.choice([0, 1])  # 0 = down, 1 = up
        self.sender = sender or (lambda data: None)
        self.running = True
        self.sensor_online = False
        self.overrides = overrides if overrides is not None else {}

        threading.Thread(target=self._live_loop, daemon=True).start()

    def _live_loop(self):
        time.sleep(0.1)  # Sensor warm-up
        self.sensor_online = True
        target_light = (MIN_LIGHT + MAX_LIGHT) / 2  # = 500 lux

        while self.running:
            if self.sensor_online:
                # Apply override if exists
                if "light" in self.overrides:
                    self.current_light = float(self.overrides["light"])
                else:
                    # ACTUATOR CONTROL: return to middle if out of range
                    if self.lamp_pct > 0:
                        self.current_light += LIGHT_CHANGE_RATE
                        if self.current_light >= target_light:
                            self.lamp_pct = 0
                            self.direction = random.choice([0, 1])

                    elif self.shutter_pct > 0:
                        self.current_light -= LIGHT_CHANGE_RATE
                        if self.current_light <= target_light:
                            self.shutter_pct = 0
                            self.direction = random.choice([0, 1])

                    else:
                        # No actuator: simulate environment drift
                        if self.direction == 0:
                            self.current_light -= LIGHT_CHANGE_RATE
                        else:
```

```
                            self.current_light += LIGHT_CHANGE_RATE

                # OUT OF THRESHOLD: activate actuator
                if self.current_light < MIN_LIGHT - 25:
                    diff = (MIN_LIGHT - self.current_light)
                    self.lamp_pct = min(100, diff - 15)
                    self.shutter_pct = 0
                    self.direction = None
                elif self.current_light > MAX_LIGHT + 25:
                    diff = self.current_light - (MAX_LIGHT)
                    self.shutter_pct = min(100, diff - 15)
                    self.lamp_pct = 0
                    self.direction = None

                # Clamp to sim limits
                self.current_light = max(MIN_LIGHT_EXT, min(MAX_LIGHT_EXT,
self.current_light))

                # Send updated sensor + actuator values
                self.sender({
                    "light": round(self.current_light, 0),
                    "lamp_pct": round(self.lamp_pct, 0),
                    "shutter_pct": round(self.shutter_pct, 0)
                })

            else:
                # If sensor offline
                self.sender({
                    "light": 0.0,
                    "lamp_pct": 0,
                    "shutter_pct": 0
                })

            time.sleep(1)

    def run(self, cycles=1):
        pass

__all__ = ['LightPLC', 'MIN_LIGHT', 'MAX_LIGHT', 'MIN_LIGHT_EXT', 'MAX_LIGHT_EXT']
```

PLC5 (CO_2)

```
import time
import random
import threading

# Thresholds for CO2 (in ppm)
MIN_CO2 = 400
MAX_CO2 = 700

# Extended CO2 range (simulate drift)
MIN_CO2_EXT = 0
MAX_CO2_EXT = 1000

CO2_CHANGE_RATE = 30  # ppm per second

class CO2PLC:
    def __init__(self, sender=None, overrides=None):
        self.current_co2 = 550  # Start mid-range
```

```
        self.co2_pump_pct = 0
        self.co2_vent_pct = 0
        self.direction = random.choice([0, 1])  # 0 = dropping CO2, 1 = increasing
CO2
        self.sender = sender or (lambda data: None)
        self.running = True
        self.sensor_online = False
        self.overrides = overrides if overrides is not None else {}

        threading.Thread(target=self.live_loop, daemon=True).start()

    def live_loop(self):
        time.sleep(0.1)
        self.sensor_online = True

        middle_co2 = (MIN_CO2 + MAX_CO2) / 2  # Target mid CO2

        while self.running:
            if self.sensor_online:
                # Apply override if exists
                if "co2" in self.overrides:
                    self.current_co2 = float(self.overrides["co2"])
                else:
                    # Actuator effects
                    if self.co2_pump_pct > 0:
                        self.current_co2 += CO2_CHANGE_RATE
                        if self.current_co2 >= middle_co2:
                            self.co2_pump_pct = 0
                            self.direction = random.choice([0, 1])
                    elif self.co2_vent_pct > 0:
                        self.current_co2 -= CO2_CHANGE_RATE
                        if self.current_co2 <= middle_co2:
                            self.co2_vent_pct = 0
                            self.direction = random.choice([0, 1])
                    else:
                        # Natural drift
                        if self.direction is None:
                            self.direction = random.choice([0, 1])
                        if self.direction == 0:
                            self.current_co2 -= CO2_CHANGE_RATE
                        else:
                            self.current_co2 += CO2_CHANGE_RATE

                # Actuator trigger
                if self.current_co2 < MIN_CO2 - 30:
                    diff = (MIN_CO2 - self.current_co2)
                    self.co2_pump_pct = min(100, diff - 25)
                    self.co2_vent_pct = 0
                    self.direction = None
                elif self.current_co2 > MAX_CO2 + 30:
                    diff = self.current_co2 - (MAX_CO2)
                    self.co2_pump_pct = 0
                    self.co2_vent_pct = min(100, diff - 25)
                    self.direction = None

                # Clamp
                self.current_co2 = max(MIN_CO2_EXT, min(MAX_CO2_EXT,
self.current_co2))
```

```
                self.sender({
                    "co2": round(self.current_co2, 2),
                    "co2_pump_pct": self.co2_pump_pct,
                    "co2_vent_pct": self.co2_vent_pct
                })
            else:
                self.sender({
                    "co2": 0.0,
                    "co2_pump_pct": 0,
                    "co2_vent_pct": 0
                })

            time.sleep(1)

    def run(self, cycles=1):
        pass
```

PySide GUI

```
# pyside.py
import json
import os
import sys
import time
from pathlib import Path

from PySide6.QtCore import Qt, QByteArray, Signal, Slot, QTimer, QProcess
from PySide6.QtGui import QCloseEvent
from PySide6.QtWidgets import (
    QApplication, QWidget, QVBoxLayout, QHBoxLayout, QLabel, QProgressBar,
    QGroupBox, QPushButton, QLineEdit, QTextEdit, QGridLayout, QMessageBox,
    QMainWindow, QTabWidget, QTableWidget, QTableWidgetItem
)

APP_TITLE = "Greenhouse PLC Dashboard (Python GUI)"
ROOT = Path(__file__).resolve().parent
COLLECTOR = str(ROOT / "plc1_collector.py")    # must exist next to this file


# ---------- small helpers ----------
def pct_bar(max_val, value):
    if max_val <= 0:
        return 0
    try:
        return max(0, min(100, int((float(value) / float(max_val)) * 100)))
    except Exception:
        return 0


# ---------- Influx duplication (no extra port) ----------
class InfluxDuplicator(QWidget):
    """
    Thin wrapper around influxdb_client that:
      - starts a timed 'capture window'
      - on each new data packet, writes a duplicate row to greenhouse_room_2
    Reads env vars: INFLUXDB_URL, INFLUXDB_ORG, INFLUXDB_BUCKET, INFLUXDB_TOKEN
    """
    statusChanged = Signal(dict)  # {active:bool, remaining:int, reason:str,
written:int, label:str}
```

```
    def __init__(self, parent=None):
        super().__init__(parent)
        self._active = False
        self._deadline = 0.0
        self._label = ""
        self._reason = ""
        self._written = 0

        self._client = None
        self._write_api = None
        self._bucket = None
        self._org = None
        self.Point = None
        self._load_influx()

        # status ticker (for countdown + surfacing errors)
        self._ticker = QTimer(self)
        self._ticker.setInterval(1000)
        self._ticker.timeout.connect(self._tick)
        self._ticker.start()

    def _load_influx(self):
        try:
            from influxdb_client import InfluxDBClient, Point
            from influxdb_client.client.write_api import SYNCHRONOUS
            self.Point = Point
        except Exception:
            self._reason = "influxdb_client not installed (pip install influxdb-
client)"
            return

        token  = os.environ.get("INFLUXDB_TOKEN") or os.environ.get("INFLUX_TOKEN")
        org    = os.environ.get("INFLUXDB_ORG") or os.environ.get("INFLUX_ORG") or
"SUTD"
        bucket = os.environ.get("INFLUXDB_BUCKET") or
os.environ.get("INFLUX_BUCKET") or "greenhouse"
        url    = os.environ.get("INFLUXDB_URL") or os.environ.get("INFLUX_URL") or
"http://localhost:8086"

        if not (url and org and bucket and token):
            self._reason = "Influx env vars missing
(INFLUXDB_URL/ORG/BUCKET/TOKEN)"
            return

        try:
            self._client = InfluxDBClient(url=url, token=token, org=org)
            # Synchronous -> no buffering; points appear immediately
            self._write_api = self._client.write_api(write_options=SYNCHRONOUS)
            self._bucket = bucket
            self._org = org
            self._reason = ""
        except Exception as e:
            self._reason = f"Influx init failed: {e}"

    def start(self, duration_sec: int, label: str = ""):
        if self._client is None or self._write_api is None:
            # lazy retry if env added later
            self._load_influx()
```

```
        if self._client is None or self._write_api is None:
            self._active = False
            self._deadline = 0
            self._label = ""
            self._emit_status()
            return

        self._active = True
        self._deadline = time.time() + int(duration_sec)
        self._label = label or ""
        self._written = 0
        self._emit_status()

    def stop(self):
        self._active = False
        self._deadline = 0
        self._label = ""
        try:
            if self._write_api is not None and hasattr(self._write_api, "flush"):
                self._write_api.flush()  # just in case
        except Exception as e:
            self._reason = f"flush failed: {e}"
        self._emit_status()

    def _tick(self):
        if not self._active:
            self._emit_status()
            return
        remaining = int(max(0, self._deadline - time.time()))
        if remaining == 0:
            self.stop()
        else:
            self._emit_status()

    def _emit_status(self):
        remaining = int(max(0, self._deadline - time.time())) if self._active else
0
        self.statusChanged.emit({
            "active": self._active,
            "remaining": remaining,
            "label": self._label,
            "reason": self._reason,
            "written": self._written,
        })

    def write_duplicate(self, sensors: dict, actuators: dict):
        """Call this on every new packet while active."""
        if not self._active or self._write_api is None or self.Point is None:
            return

        try:
            p = (
                self.Point("greenhouse")  # keep single measurement; separate by
tag
                .tag("location", "greenhouse_room_2")
                .field("temperature", float(sensors.get("temperature", 0.0)))
                .field("light", float(sensors.get("light", 0.0)))
                .field("moisture", float(sensors.get("moisture", 0.0)))
                .field("co2", float(sensors.get("co2", 0.0)))
```

```
            )
            # mirror actuators to match schema
            for k, v in (actuators or {}).items():
                try:
                    p = p.field(k, float(v))
                except Exception:
                    pass

            if self._label:
                p = p.tag("label", self._label).tag("source", "duplicate")

            # pass org explicitly for robustness
            self._write_api.write(bucket=self._bucket, org=self._org, record=p)
            self._written += 1
        except Exception as e:
            self._reason = f"write failed: {e}"


# ---------- PLC panels & dashboard ----------
class PlcPanel(QGroupBox):
    # sensor_key, payload ({"type":"range","min":...,"max":...} or
{"type":"constant","value":...})
    applyOverride = Signal(str, dict)
    clearOverride = Signal(str)

    def __init__(self, title, sensor_key, value_unit, gauge_max,
                 actuator_specs, parent=None):
        super().__init__(title, parent)
        self.sensor_key = sensor_key
        self.value_unit = value_unit
        self.gauge_max = gauge_max
        self.actuator_specs = actuator_specs  # [('Heater','heater_pct'), ...]

        self.value_lbl = QLabel("--")
        self.gauge = QProgressBar()
        self.gauge.setRange(0, 100)
        self.gauge.setFormat("")

        grid = QGridLayout()
        row = 0

        grid.addWidget(QLabel(f"{sensor_key.capitalize()}: "), row, 0,
Qt.AlignLeft)
        grid.addWidget(self.value_lbl, row, 1, Qt.AlignLeft)
        grid.addWidget(QLabel(self.value_unit), row, 2, Qt.AlignLeft)
        row += 1
        grid.addWidget(self.gauge, row, 0, 1, 3)
        row += 1

        self.act_rows = []
        for label_text, key in self.actuator_specs:
            name_lbl = QLabel(f"{label_text}:")
            val_lbl = QLabel("--%")
            bar = QProgressBar()
            bar.setRange(0, 100)
            bar.setFormat("")
            grid.addWidget(name_lbl, row, 0, Qt.AlignLeft)
            grid.addWidget(val_lbl, row, 1, Qt.AlignLeft)
            row += 1
```

```
            grid.addWidget(bar, row, 0, 1, 3)
            row += 1
            self.act_rows.append((key, val_lbl, bar))

        # --- override inputs: range (min, max) OR constant (min only) ---
        self.override_min = QLineEdit()
        self.override_min.setPlaceholderText("Min (or constant)")
        self.override_max = QLineEdit()
        self.override_max.setPlaceholderText("Max (optional)")

        self.apply_btn = QPushButton("Apply")
        self.reset_btn = QPushButton("Reset")

        o_row = QHBoxLayout()
        o_row.addWidget(QLabel("Attack:"))
        o_row.addWidget(self.override_min)
        o_row.addWidget(self.override_max)
        o_row.addWidget(self.apply_btn)
        o_row.addWidget(self.reset_btn)

        v = QVBoxLayout()
        v.addLayout(grid)
        v.addLayout(o_row)
        self.setLayout(v)

        self.apply_btn.clicked.connect(self._apply_clicked)
        self.reset_btn.clicked.connect(self._reset_clicked)

    @Slot()
    def _apply_clicked(self):
        try:
            vmin = float(self.override_min.text()) if self.override_min.text() else
None
            vmax = float(self.override_max.text()) if self.override_max.text() else
None
        except ValueError:
            QMessageBox.warning(self, "Invalid override", "Enter numeric values.")
            return

        if vmin is not None and vmax is not None:
            if vmin > vmax:
                QMessageBox.warning(self, "Invalid range", "Min cannot be greater
than Max.")
                return
            payload = {"type": "range", "min": vmin, "max": vmax}
        elif vmin is not None:
            payload = {"type": "constant", "value": vmin}
        else:
            QMessageBox.warning(self, "No value", "Enter at least a Min value.")
            return

        self.applyOverride.emit(self.sensor_key, payload)

    @Slot()
    def _reset_clicked(self):
        self.clearOverride.emit(self.sensor_key)

    def update_view(self, sensors: dict, actuators: dict):
        sv = sensors.get(self.sensor_key)
```

```
        if sv is not None:
            try:
                self.value_lbl.setText(f"{float(sv):.2f}")
            except Exception:
                self.value_lbl.setText(str(sv))
            self.gauge.setValue(pct_bar(self.gauge_max, sv))

        for key, val_lbl, bar in self.act_rows:
            try:
                p = float(actuators.get(key, 0))
            except Exception:
                p = 0.0
            val_lbl.setText(f"{p:.0f}%")
            bar.setValue(int(max(0, min(100, p))))


class DashboardWidget(QWidget):
    """
    Dashboard tab:
      - spawns plc1_collector.py
      - emits 'newPacket' with parsed JSON
    """
    newPacket = Signal(dict)

    def __init__(self, parent=None):
        super().__init__(parent)
        self.proc = QProcess(self)
        self.proc.setProgram(sys.executable)
        self.proc.setArguments(["-u", COLLECTOR])
        self.proc.setProcessChannelMode(QProcess.MergedChannels)

        # forward parent env (so Influx creds reach the collector)
        env = self.proc.processEnvironment()
        for k in ("INFLUXDB_URL", "INFLUXDB_ORG", "INFLUXDB_BUCKET",
"INFLUXDB_TOKEN",
                  "INFLUX_URL", "INFLUX_ORG", "INFLUX_BUCKET", "INFLUX_TOKEN"):
            v = os.environ.get(k)
            if v:
                env.insert(k, v)
        self.proc.setProcessEnvironment(env)

        self.temp_panel = PlcPanel("🔧 Temp PLC", "temperature", "°C", 50.0,
                                   [("Heater", "heater_pct"), ("Cooler",
"cooler_pct")])
        self.moist_panel = PlcPanel("💧 Irrigation PLC", "moisture", "%", 100.0,
                                    [("Pump", "pump_pct"), ("Drain", "drain_pct")])
        self.light_panel = PlcPanel("💡 Light PLC", "light", "lux", 1000.0,
                                    [("Lamp", "lamp_pct"), ("Shutter",
"shutter_pct")])
        self.co2_panel = PlcPanel("🫧 CO₂ PLC", "co2", "ppm", 1000.0,
                                  [("CO₂ Pump", "co2_pump_pct"), ("Vent",
"co2_vent_pct")])

        for panel in (self.temp_panel, self.moist_panel, self.light_panel,
self.co2_panel):
            panel.applyOverride.connect(self.send_override)
            panel.clearOverride.connect(self.clear_override)
```

```
        self.timestamp_lbl = QLabel("Timestamp: --")
        self.alerts_txt = QTextEdit()
        self.alerts_txt.setReadOnly(True)
        self.alerts_txt.setMinimumHeight(140)

        grid = QGridLayout()
        grid.addWidget(self.temp_panel, 0, 0)
        grid.addWidget(self.moist_panel, 0, 1)
        grid.addWidget(self.light_panel, 1, 0)
        grid.addWidget(self.co2_panel, 1, 1)

        info = QVBoxLayout()
        info.addWidget(self.timestamp_lbl)
        info.addWidget(self.alerts_txt)

        root = QVBoxLayout()
        root.addLayout(grid)
        root.addLayout(info)
        self.setLayout(root)

        self.proc.readyReadStandardOutput.connect(self.read_proc)
        self.proc.finished.connect(self.proc_finished)
        self.proc.errorOccurred.connect(self.proc_error)
        self.proc.start()

        if os.environ.get("INFLUXDB_TOKEN") in (None, "") and
os.environ.get("INFLUX_TOKEN") in (None, ""):
            self.alerts_txt.append("⚠ INFLUX token not set; collector may log auth
errors.")

    @Slot()
    def read_proc(self):
        while self.proc.canReadLine():
            line: QByteArray = self.proc.readLine()
            text = bytes(line).decode(errors="ignore").strip()
            if not text:
                continue
            try:
                data = json.loads(text)
                self.update_dashboard(data)
                self.newPacket.emit(data)  # give to other tabs (e.g., duplicator)
            except json.JSONDecodeError:
                self.alerts_txt.append(text)

    @Slot(int, QProcess.ExitStatus)
    def proc_finished(self, code, status):
        self.alerts_txt.append(f"Collector exited (code={code},
status={int(status)}).")

    @Slot("QProcess::ProcessError")
    def proc_error(self, err):
        self.alerts_txt.append(f"Process error: {err}")

    def update_dashboard(self, data: dict):
        sensors = data.get("sensors", {})
        actuators = data.get("actuators", {})
        alerts = data.get("alerts", {})
        ts = sensors.get("timestamp", "--")
        self.timestamp_lbl.setText(f"Timestamp: {ts}")
```

```
        self.temp_panel.update_view(sensors, actuators)
        self.moist_panel.update_view(sensors, actuators)
        self.light_panel.update_view(sensors, actuators)
        self.co2_panel.update_view(sensors, actuators)

        if not alerts:
            self.alerts_txt.setPlainText("☑ All normal")
        else:
            out_lines = ["⚠ Alerts:"]
            for k, info in alerts.items():
                out_lines.append(f"- {k.upper()}: {info.get('value')} →
{info.get('status')}")
            self.alerts_txt.setPlainText("\n".join(out_lines))

    @Slot(str, dict)
    def send_override(self, sensor_key: str, payload: dict):
        # Normalize to collector message shapes
        if payload.get("type") == "range":
            cmd = {"type": "override_range", "sensor": sensor_key,
                   "min": payload["min"], "max": payload["max"]}
        else:
            cmd = {"type": "override", "sensor": sensor_key,
                   "value": payload["value"]}
        self._write_cmd(cmd)

    @Slot(str)
    def clear_override(self, sensor_key: str):
        cmd = {"type": "clear_override", "sensor": sensor_key}
        self._write_cmd(cmd)

    def _write_cmd(self, obj: dict):
        if self.proc.state() != QProcess.Running:
            QMessageBox.warning(self, "Not running", "Collector process is not
running.")
            return
        payload = (json.dumps(obj) + "\n").encode()
        written = self.proc.write(payload)
        if written == -1:
            self.alerts_txt.append("Failed to write to collector stdin.")
            return
        if not self.proc.waitForBytesWritten(500):
            self.alerts_txt.append("Timed out waiting for command to be written.")

    def shutdown(self):
        if self.proc and self.proc.state() == QProcess.Running:
            self.proc.kill()
            self.proc.waitForFinished(2000)


# ---------- Local Subscribe / Duplicate tab (no HTTP) ----------
class LocalSubscribeTab(QWidget):
    """
    Mirrors the 'Subscribe/Duplicate' page but uses the local InfluxDuplicator
    and the live packets from DashboardWidget.newPacket (no network, no ports).
    """
    def __init__(self, duplicator: InfluxDuplicator, parent=None):
        super().__init__(parent)
```

```
        self.dupe = duplicator

        title = QLabel("PLC Live + Capture (Local)")
        f = title.font()
        f.setPointSize(16)
        f.setBold(True)  # portable across PySide6 versions
        title.setFont(f)
        title.setAlignment(Qt.AlignHCenter)

        self.btn5 = QPushButton("Start 5-min Capture")
        self.btn10 = QPushButton("Start 10-min Capture")
        self.btnStop = QPushButton("Stop Capture")

        top = QHBoxLayout()
        top.addStretch(1)
        top.addWidget(self.btn5)
        top.addWidget(self.btn10)
        top.addWidget(self.btnStop)
        top.addStretch(1)

        self.capStatus = QLabel("capture: idle")
        self.capStatus.setAlignment(Qt.AlignHCenter)

        gbox = QGroupBox("Live Gauges")
        grid = QGridLayout(gbox)

        self.pbTemp = self._mk_bar(0, 50)
        self.pbLight = self._mk_bar(0, 1000)
        self.pbMoist = self._mk_bar(0, 100)
        self.pbCO2 = self._mk_bar(0, 1000)

        self.lblTemp = QLabel("0.00")
        self.lblLight = QLabel("0.00")
        self.lblMoist = QLabel("0.00")
        self.lblCO2 = QLabel("0.00")

        r = 0
        grid.addWidget(QLabel("Temperature (°C)"), r, 0);
grid.addWidget(self.pbTemp, r, 1); grid.addWidget(self.lblTemp, r, 2); r += 1
        grid.addWidget(QLabel("Light"),           r, 0);
grid.addWidget(self.pbLight, r, 1); grid.addWidget(self.lblLight, r, 2); r += 1
        grid.addWidget(QLabel("Moisture (%)"),    r, 0);
grid.addWidget(self.pbMoist, r, 1); grid.addWidget(self.lblMoist, r, 2); r += 1
        grid.addWidget(QLabel("CO₂ (ppm)"),       r, 0); grid.addWidget(self.pbCO2,
r, 1); grid.addWidget(self.lblCO2,   r, 2); r += 1

        self.tbl = QTableWidget(5, 3)
        self.tbl.setHorizontalHeaderLabels(["Metric", "Value", "Timestamp"])
        self.tbl.verticalHeader().setVisible(False)
        self.tbl.horizontalHeader().setStretchLastSection(True)
        for i, name in enumerate(["Temperature", "Light", "Moisture", "CO₂",
"Alerts"]):
            self.tbl.setItem(i, 0, QTableWidgetItem(name))
            self.tbl.setItem(i, 1, QTableWidgetItem("—"))
            self.tbl.setItem(i, 2, QTableWidgetItem("—"))

        root = QVBoxLayout(self)
        root.addWidget(title)
        root.addLayout(top)
```

```
        root.addWidget(self.capStatus)
        root.addWidget(gbox)
        root.addWidget(self.tbl)
        root.addStretch(1)

        # wire buttons
        self.btn5.clicked.connect(lambda: self.dupe.start(300, "block-5min"))
        self.btn10.clicked.connect(lambda: self.dupe.start(600, "block-10min"))
        self.btnStop.clicked.connect(self.dupe.stop)
        self.dupe.statusChanged.connect(self._on_status)

    def _mk_bar(self, lo, hi):
        bar = QProgressBar()
        bar.setRange(lo, hi)
        bar.setTextVisible(False)
        bar.setFixedHeight(18)
        return bar

    @Slot(dict)
    def _on_status(self, j):
        # Show errors or live countdown + written counter
        if j.get("reason"):
            self.capStatus.setText(f"capture: {('ACTIVE' if j.get('active') else
'idle')} — {j['reason']}")
            return
        w = j.get("written", 0)
        if j.get("active"):
            rem = j.get("remaining", 0)
            label = j.get("label") or ""
            self.capStatus.setText(f"capture: ACTIVE{f' (label={label})' if label
else ''} — remaining {rem}s — written {w}")
        else:
            self.capStatus.setText(f"capture: idle — written {w}")

    # called by MainWindow when new packets arrive
    def update_live(self, sensors: dict, alerts_count: int, ts: str):
        temp = float(sensors.get("temperature") or 0.0)
        light = float(sensors.get("light") or 0.0)
        moist = float(sensors.get("moisture") or 0.0)
        co2  = float(sensors.get("co2") or 0.0)

        self.pbTemp.setValue(int(temp))
        self.pbLight.setValue(int(light))
        self.pbMoist.setValue(int(moist))
        self.pbCO2.setValue(int(co2))

        self.lblTemp.setText(f"{temp:.2f}")
        self.lblLight.setText(f"{light:.2f}")
        self.lblMoist.setText(f"{moist:.2f}")
        self.lblCO2.setText(f"{co2:.2f}")

        rows = [
            ("Temperature", temp, ts),
            ("Light",       light, ts),
            ("Moisture",    moist, ts),
            ("CO₂",         co2,   ts),
            ("Alerts",      alerts_count, ts),
        ]
        for i, (_, val, t) in enumerate(rows):
```

```
            self.tbl.setItem(i, 1, QTableWidgetItem(f"{val}"))
            self.tbl.setItem(i, 2, QTableWidgetItem(t))


# ---------- Detection tab ----------
class DetectionTab(QWidget):
    """
    Runs detect_attack.py in the background and streams detection results here.
    """
    def __init__(self, parent=None):
        super().__init__(parent)

        title = QLabel("🔍Attack Detection")
        f = title.font()
        f.setPointSize(16)
        f.setBold(True)
        title.setFont(f)
        title.setAlignment(Qt.AlignHCenter)

        self.log = QTextEdit()
        self.log.setReadOnly(True)

        root = QVBoxLayout(self)
        root.addWidget(title)
        root.addWidget(self.log)

        # --- start attack detector as subprocess ---
        self.proc = QProcess(self)
        self.proc.setProgram(sys.executable)
        self.proc.setArguments(["-u", str(ROOT / "detect_attack.py")])
        self.proc.setProcessChannelMode(QProcess.MergedChannels)

        # forward env (so it has Influx credentials)
        env = self.proc.processEnvironment()
        for k in ("INFLUXDB_URL", "INFLUXDB_ORG", "INFLUXDB_BUCKET",
"INFLUXDB_TOKEN",
                  "INFLUX_URL", "INFLUX_ORG", "INFLUX_BUCKET", "INFLUX_TOKEN"):
            v = os.environ.get(k)
            if v:
                env.insert(k, v)
        self.proc.setProcessEnvironment(env)

        self.proc.readyReadStandardOutput.connect(self._on_output)
        self.proc.finished.connect(self._on_finished)
        self.proc.start()

    @Slot()
    def _on_output(self):
        while self.proc.canReadLine():
            line = bytes(self.proc.readLine()).decode(errors="ignore").strip()
            if line:
                # Escape HTML characters
                safe_line = line.replace("&", "&amp;").replace("<",
"&lt;").replace(">", "&gt;")

                if safe_line.startswith("[ATTACK]"):
                    # color only the label
                    html_line = safe_line.replace("[ATTACK]", '<span
```

```
style="color:red;">🚨[ATTACK]</span>', 1)
                elif safe_line.startswith("[NORMAL]"):
                    html_line = safe_line.replace("[NORMAL]", '<span
style="color:green;">☑[NORMAL]</span>', 1)
                else:
                    html_line = safe_line

                self.log.append(html_line)

    @Slot(int, QProcess.ExitStatus)
    def _on_finished(self, code, status):
        self.log.append(f"[INFO] detect_attack.py exited (code={code},
status={status})")

    def shutdown(self):
        if self.proc and self.proc.state() == QProcess.Running:
            self.proc.kill()
            self.proc.waitForFinished(2000)

# ---------- Main window ----------
class MainWindow(QMainWindow):
    """Hosts tabs: Dashboard and Local Subscribe/Duplicate."""
    def __init__(self):
        super().__init__()
        self.setWindowTitle(APP_TITLE)

        self.tabs = QTabWidget()
        self.dashboard = DashboardWidget()
        self.duplicator = InfluxDuplicator()
        self.subscribe = LocalSubscribeTab(self.duplicator)
        self.detection = DetectionTab()

        self.tabs.addTab(self.dashboard, "Dashboard")
        self.tabs.addTab(self.subscribe, "Subscribe / Duplicate")
        self.tabs.addTab(self.detection, "Detection")

        self.setCentralWidget(self.tabs)
        self.resize(1200, 800)

        # bridge dashboard packets -> duplicator + local tab
        self.dashboard.newPacket.connect(self._on_packet)

    @Slot(dict)
    def _on_packet(self, data: dict):
        sensors = data.get("sensors", {}) or {}
        actuators = data.get("actuators", {}) or {}
        alerts = data.get("alerts", {}) or {}
        ts = sensors.get("timestamp", "")
        # feed duplicator (only writes when active)
        self.duplicator.write_duplicate(sensors, actuators)
        # update the local subscribe UI
        self.subscribe.update_live(sensors, len(alerts), ts)

    def closeEvent(self, event: QCloseEvent):
        if hasattr(self, "dashboard") and isinstance(self.dashboard,
DashboardWidget):
            self.dashboard.shutdown()
        if hasattr(self, "detection") and isinstance(self.detection, DetectionTab):
```

```
            self.detection.shutdown()
        event.accept()


def main():
    app = QApplication(sys.argv)
    w = MainWindow()
    w.show()
    sys.exit(app.exec())


if __name__ == "__main__":
    main()
```

Dataset Builder

```
# collect_labeled_data.py
import pandas as pd
import time
import os
from influxdb_client import InfluxDBClient

# ==== USER SETTINGS ====
token  = os.environ.get("INFLUXDB_TOKEN") or os.environ.get("INFLUX_TOKEN")
org    = os.environ.get("INFLUXDB_ORG") or os.environ.get("INFLUX_ORG") or "SUTD"
bucket = os.environ.get("INFLUXDB_BUCKET") or os.environ.get("INFLUX_BUCKET") or
"greenhouse"
url    = os.environ.get("INFLUXDB_URL") or os.environ.get("INFLUX_URL") or
"http://localhost:8086"

# ==== INPUT LABEL ====
label = int(input("Enter label (0 = normal, 1 = attack): "))
filename = "plc_labeled_data.csv"

# ==== CONNECT ====
client = InfluxDBClient(url=url, token=token, org=org)
query_api = client.query_api()

print(f"📡 Collecting data with label {label}... Press Ctrl+C to stop.")

try:
    while True:
        query = f'''
        from(bucket: "{bucket}")
          |> range(start: -5s)
          |> filter(fn: (r) => r["_measurement"] == "greenhouse")
          |> pivot(rowKey:["_time"], columnKey:["_field"], valueColumn:"_value")
        '''
        result = query_api.query_data_frame(query)

        # Handle if result is a list of DataFrames
        if isinstance(result, list):
            if len(result) == 0:
                time.sleep(1)
                continue
            df = pd.concat(result)
        else:
            df = result
```

```
        # Ensure it's not empty
        if not df.empty:
            # Only keep sensor columns that exist
            cols = [c for c in ["temperature", "moisture", "co2", "light"] if c in
df.columns]
            if cols:
                df = df[cols].dropna()
                df["label"] = label
                df.to_csv(filename, mode='a', header=not os.path.exists(filename),
index=False)
                print(f"💾 Saved {len(df)} rows with label {label}")

        time.sleep(1)

except KeyboardInterrupt:
    print(f"☑ Data collection stopped. Saved to {filename}")
```

Model Training

```
# train_model.py
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
import joblib

# Load collected data
df = pd.read_csv("plc_labeled_data.csv")

X = df[["temperature", "moisture", "co2", "light"]]
y = df["label"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)

print(f"☑ Accuracy: {clf.score(X_test, y_test) * 100:.2f}%")

joblib.dump(clf, "plc_detector.pkl")
print("💾 Model saved as plc_detector.pkl")
```

Attack Detection

```
import pandas as pd
import time
import joblib
from influxdb_client import InfluxDBClient
import os

# Load trained model
clf = joblib.load("plc_detector.pkl")

# InfluxDB environment
token  = os.environ.get("INFLUXDB_TOKEN") or os.environ.get("INFLUX_TOKEN")
org    = os.environ.get("INFLUXDB_ORG") or os.environ.get("INFLUX_ORG") or "SUTD"
bucket = os.environ.get("INFLUXDB_BUCKET") or os.environ.get("INFLUX_BUCKET") or
"greenhouse"
url    = os.environ.get("INFLUXDB_URL") or os.environ.get("INFLUX_URL") or
```

```
"http://localhost:8086"

client = InfluxDBClient(url=url, token=token, org=org, bucket=bucket)
query_api = client.query_api()

WINDOW = 10      # seconds for Influx query
BATCH_SIZE = 5   # batch prediction

print("[ALERT] Real-time PLC Attack Detector Running...")

last_seen = None  # track most recent timestamp

while True:
    try:
        query = f'''
        from(bucket: "{bucket}")
          |> range(start: -{WINDOW}s)
          |> filter(fn: (r) => r["_measurement"] == "greenhouse")
          |> pivot(rowKey:["_time"], columnKey:["_field"], valueColumn:"_value")
        '''
        result = query_api.query_data_frame(query)

        # Flatten result
        if isinstance(result, list):
            dfs = [r.dropna(axis=1, how="all") for r in result if isinstance(r,
pd.DataFrame) and not r.empty]
            if len(dfs) == 0:
                time.sleep(1)
                continue
            df = pd.concat(dfs, ignore_index=True)
        else:
            if result.empty:
                time.sleep(1)
                continue
            df = result.dropna(axis=1, how="all")

        # Ensure required features exist
        required_cols = ["temperature", "moisture", "co2", "light"]
        for col in required_cols:
            if col not in df.columns:
                df[col] = 0

        # Only keep rows with timestamp > last_seen
        if last_seen is not None:
            df = df[df["_time"] > last_seen]

        if df.empty:
            time.sleep(1)
            continue

        # Update last_seen
        last_seen = df["_time"].max()

        X_live = df[required_cols]
        times = df["_time"].tolist()

        # Batch prediction
        for i in range(0, len(X_live), BATCH_SIZE):
            batch = X_live.iloc[i:i+BATCH_SIZE]
```

```
            batch_times = times[i:i+BATCH_SIZE]
            preds = clf.predict(batch)

            for t, p in zip(batch_times, preds):
                if p == 1:
                    print(f"[ATTACK] Detected at {t}")
                else:
                    print(f"[NORMAL] at {t}")

        time.sleep(0.5)

    except Exception as e:
        print(f"[ERROR] {e}")
        time.sleep(2)
```