Incremental Security Enforcement of Cyber-Physical Systems

A Project Report submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF
TECHNOLOGY IN
COMPUTER SCIENCE AND ENGINEERING

Submitted By
TEAM P1-07
Aaditya Kumar Muktavarapu HU21CSEN0100580
Karthik Neriyanuri HU21CSEN0100607
Vijay Anirudh Vallabhu HU21CSEN0100611
Lalit Sai Srivatsa Narayanam HU21CSEN0101246

Under the esteemed guidance of
Dr. Sudeep K.S
Associate Professor

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
GITAM
(Deemed to be University)
HYDERABAD

2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
GITAM SCHOOL OF TECHNOLOGY
GITAM

(Deemed to be University)

DECLARATION

We, hereby, declare that the project report entitled "**Incremental Security Enforcement of Cyber-Physical Systems**" is an original work done in the Department of Computer Science and Engineering, GITAM School of Technology, GITAM (Deemed to be University), submitted in partial fulfillment of the requirements for the award of the degree of B.Tech. in Computer Science and Engineering. The work has not been submitted to any other college or University for the award of any degree or diploma.

Date: 20th March 2025

Registration No(s).	Name(s)	Signature(s)
HU21CSEN0100580	**Aaditya Kumar Muktavarapu**	
HU21CSEN0100607	**Karthik Neriyanuri**	
HU21CSEN0100611	**Vijay Anirudh Vallabhu**	
HU21CSEN0101246	**Lalit Sai Srivatsa Narayanam**	

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM SCHOOL OF TECHNOLOGY

GITAM

(Deemed to be University)

CERTIFICATE

This is to certify that the project report entitled "**Incremental Security Enforcement of Cyber-Physical Systems**" is a bonafide record of work carried out by **Aaditya Kumar Muktavarapu HU21CSEN0100580, Karthik Neriyanuri HU21CSEN0100607, Vijay Anirudh Vallabhu HU21CSEN0100611, Lalit Sai Srivatsa Narayanam HU21CSEN0101246**

Students submitted in partial fulfillment of the requirement for the award of the Bachelor of Technology in Computer Science and Engineering degree.

Project Guide	**Project Coordinator**	**Head of the Department**
Dr. Sudeep K.S	**Dr. A B Pradeep Kumar**	**Dr.Shaik Mahaboob Basha**
Associate Professor Dept. of CSE	**Assistant Professor Dept. of CSE**	**Professor & HOD Dept. of CSE**

ACKNOWLEDGEMENT

Our project report would not have been successful without the help of several people. We would like to thank the personalities who were part of our seminar in numerous ways, those who gave us outstanding support from the birth of the seminar.

We are very much obliged to our beloved **Prof. Sheik Mahboob Basha**, Head of the Department of Computer Science & Engineering, for providing the opportunity to undertake this seminar and encouragement in the completion of this seminar.

We hereby wish to express our deep sense of gratitude to **Dr. A B Pradeep Kumar,** Project Coordinator, Department of Computer Science and Engineering, School of Technology, and to our guide, **Dr. Sudeep K.S** Associate Professor, Department of Computer Science and Engineering, School of Technology for the esteemed guidance, moral support and invaluable advice provided by them for the success of the project report.

We are also thankful to all the Computer Science and Engineering department staff members who have cooperated in making our seminar a success. We would like to thank all our parents and friends who extended their help, encouragement, and moral support directly or indirectly in our seminar work.

Sincerely,

Aaditya Kumar Muktavarapu,
Karthik Neriyanuri,
Vijay Anirudh Vallabhu,
Lalit Sai Srivatsa Narayanam

TABLE OF CONTENTS

ABSTRACT

Ensuring the security and reliability of autonomous drone swarms is critical, particularly in the face of cyber threats such as GPS spoofing, command injection, and denial-of-service attacks. This project presents a compositional runtime enforcement framework for enforcing security policies on drones, comparing monolithic enforcement (where all policies are applied at once) and incremental enforcement (where policies are enforced sequentially).

The framework is implemented using MATLAB, leveraging real-time monitoring, policy validation, and enforcement mechanisms to detect and mitigate cyber attacks. A 3D visualization system is developed to demonstrate the effects of attacks, including drones deviating from intended paths, unauthorized commands, and policy violations. The framework dynamically enforces corrections, restoring compromised drones to safe operational states.

Through simulations, the project evaluates the trade-offs between monolithic and incremental enforcement, highlighting the impact on state space explosion, system efficiency, and violation recovery. The results show that incremental enforcement significantly reduces complexity while maintaining robust security. The visualization tools provide an intuitive way to observe attack consequences and enforcement effectiveness, making this framework valuable for real-world drone security applications.

Chapter 1
Introduction

Cyber-Physical Systems (CPS) are integrated systems where computational processes control physical processes. They are widely used in sectors like industrial automation, smart grids, autonomous vehicles, and healthcare.

With increased complexity and interconnectivity, CPS have become vulnerable to cyber-physical attacks (CP-attacks), which exploit both cyber and physical components to cause disruption.

Traditional security patches are ineffective for these systems, which require real-time security updates.

This project introduces an Incremental Security Enforcement Framework for CPS that allows for the dynamic addition of new security policies to mitigate evolving threats.

1.1 Problem Definition

Current CPS security solutions are inadequate for managing incremental policy updates as new threats emerge. Traditional monolithic approaches to runtime enforcement (RE) suffer from state space explosion and require re-certification each time a new policy is added. There is a need for a system that allows the incremental composition of enforcers to adapt to new policies without compromising existing security measures.

1.2 Objective

To develop a compositional runtime enforcement (RE) framework that incrementally applies new security policies to CPS. This framework must prevent state space explosion, ensure modularity, and support real-time enforcement, enabling CPS to adapt to emerging threats without a full system overhaul.

1.3 Limitations

- **Initial Set-Up Complexity**: Setting up initial policies requires defining the enforcers with precision to allow smooth incremental updates.
- **Non-Compatible Policies**: Some policies may not be compatible with others in a compositional enforcement structure, requiring careful management of interdependencies.

1.4 Applications

- **Industrial Control Systems**: Protects critical systems from cyber-attacks by allowing dynamic addition of safety policies.
- **Autonomous Vehicles**: Ensures real-time compliance with safety policies that protect passengers and pedestrians.
- **Healthcare Devices**: Enhances the security of medical devices by incrementally adding policies for new health threats or operational requirements

Chapter 2

Literature Review

Cyber-Physical Systems (CPS) are increasingly vulnerable to cyber threats due to their integration of computation, networking, and physical processes. Ensuring security in these systems requires scalable and incremental enforcement strategies to mitigate risks while maintaining system performance. This literature review examines three key papers that address security enforcement in CPS from different perspectives: incremental security enforcement, scalable security enforcement, and attack monitoring and defense.

Incremental Security Enforcement for Cyber-Physical Systems

Panda et al. (2021) propose an incremental security enforcement approach for CPS[1], emphasizing the need for adaptive security mechanisms. Traditional security enforcement methods often rely on monolithic strategies that may not scale efficiently with evolving threats. This paper introduces a framework that incrementally applies security policies based on system states and detected threats.

The authors utilize formal methods to model security policies and enforce them dynamically. The incremental approach ensures that security enforcement is computationally feasible, reducing the overhead associated with applying all policies simultaneously. Additionally, their approach enhances flexibility, allowing CPS to adapt to new threats without requiring complete system reconfiguration.

Experimental results demonstrate that incremental enforcement leads to improved system resilience with minimal performance degradation. The study highlights the significance of context-aware security enforcement, where security measures are applied based on real-time risk assessment rather than a static rule set. The findings suggest that incremental enforcement can be a viable solution for real-time CPS applications, such as industrial control systems and autonomous vehicles.

Scalable Security Enforcement for Cyber-Physical Systems

Building upon the incremental enforcement framework, Baird et al. (2022) explore scalable security enforcement mechanisms for CPS[2]. The paper addresses the challenges of applying security policies in large-scale CPS, where a high number of interconnected components must be protected efficiently.

The authors introduce a distributed security enforcement model that partitions security policies across different system layers. By leveraging hierarchical enforcement techniques, the proposed model ensures that security policies are applied at appropriate system levels, reducing computational burden and latency. The paper also discusses optimization strategies for minimizing redundant security checks, which further enhances system efficiency.

One of the key contributions of this study is the introduction of security enforcement scalability metrics. These metrics enable system designers to evaluate the trade-offs between security enforcement complexity and system performance. The research findings indicate that scalable enforcement techniques can significantly improve the security posture of CPS while maintaining operational efficiency.

Monitoring and Defense of Industrial Cyber-Physical Systems Under Typical Attacks

Jiang et al. (2023) focus on the monitoring and defense mechanisms for industrial CPS[3], addressing common cyber threats such as denial-of-service attacks, command injection, and spoofing. Unlike the previous two papers that primarily discuss enforcement mechanisms, this study emphasizes real-time attack detection and mitigation.

The authors propose a hybrid monitoring framework that combines anomaly detection with rule-based security policies. Machine learning techniques are employed to identify deviations from normal system behavior, allowing the system to detect potential threats proactively. The paper also introduces an adaptive defense mechanism that dynamically adjusts security policies based on detected threats.

Experimental evaluations show that the proposed framework effectively detects and mitigates cyber threats with high accuracy. The study underscores the importance of integrating monitoring with enforcement strategies to create a comprehensive security framework for CPS. Additionally, the paper discusses real-world applications in industrial automation, demonstrating how the proposed approach can enhance security resilience in practical scenarios.

Comparative Analysis

The three papers focus on different but complementary aspects of security enforcement in Cyber-Physical Systems (CPS). Panda et al. (2021) introduce an incremental security enforcement approach, which allows security policies to be applied dynamically based on system states, reducing computational overhead. Baird et al. (2022) build upon this by introducing scalable security enforcement, ensuring that security policies remain efficient in large-scale CPS with distributed enforcement mechanisms. Jiang et al. (2023) shift the focus to monitoring and defense, emphasizing real-time detection and mitigation of cyber threats using anomaly detection and machine learning techniques.

While the first two papers emphasize proactive security enforcement, the third focuses on reactive monitoring and response to cyber threats. Panda et al. and Baird et al. primarily use formal methods and optimization strategies to enhance policy application, whereas Jiang et al. integrate machine learning-based anomaly detection for threat identification. The studies collectively highlight the importance of adaptive security mechanisms that balance enforcement, scalability, and real-time defense for securing CPS against evolving cyber threats.

Chapter 3
Problem Analysis

Cyber-physical systems are exposed to sophisticated Cyber Physical attacks that require proactive, flexible, and scalable security mechanisms. The research identified that traditional enforcement models could not scale with the continuous addition of policies, creating bottlenecks and inefficiencies.

3.1 Problem Statement

To address the security challenges in CPS, this project seeks to develop a Runtime Enforcement framework that can incrementally compose enforcers for new policies, enhancing security adaptability without re-certifying the entire system.

3.2 Existing Methodologies

The conventional CPS security system uses monolithic enforcers, which combine all policies into a single-state machine. While effective for enforcing policies, this method suffers from significant limitations in scalability, adaptability, and efficiency.

3.2.1. Monolithic Enforcement

Monolithic enforcement consolidates all security policies into a single, unified state machine, aiming to handle multiple policies simultaneously within one overarching enforcer structure.

In this approach, each policy does not operate independently; instead, all policies are merged into a single enforcement mechanism that must evaluate and apply each policy in tandem.

While this unification allows for centralized control, it can lead to increased complexity as each policy’s conditions and requirements must be accounted for within a single system.

As policies grow in number and intricacy, the monolithic enforcer’s state machine must expand to represent every possible interaction, which often leads to what's known as "state space explosion"—a dramatic increase in the number of states the system must manage.

This not only strains memory and processing resources but also makes the system harder to modify, as any addition or adjustment to a policy can affect the overall structure.

Consequently, while monolithic enforcement provides a singular control point for policy application, it often sacrifices scalability and flexibility, making it less suitable for dynamic environments like Cyber-Physical Systems (CPS) that frequently need to adapt to evolving security demands.

- **Limitations**:
    - **State Space Explosion**: Each additional policy exponentially increases the state space, requiring more memory and computational resources, which impacts performance.
    - **Recertification Requirement**: Every time a new policy is added, the enforcer must be re-synthesized, which is a time-consuming and inefficient process. This re-certification disrupts the system, making it unsuitable for dynamic environments.
    - **Loss of Modularity**: Policies are interdependent, making it difficult to isolate or modify them individually. Any change to one policy can inadvertently impact others, reducing flexibility and increasing the likelihood of conflicts.

3.2.2 Runtime Verification (RV) Techniques

Runtime Verification (RV) techniques are centered on the passive observation of system behavior to ensure policy compliance, focusing on detecting deviations from expected norms without directly intervening in system operations.

In this approach, monitors are designed to observe the sequence of events and states that occur within a system in real-time, checking against predefined security or operational policies. Rather than actively enforcing rules, these monitors serve as a "watchdog" by tracking adherence and detecting violations as they happen.

Observation-based verification is the primary strategy within RV, where monitors operate in the background and continuously assess system actions for compliance. This passive approach allows RV to function with minimal disruption to system performance, as it does not alter the execution flow but instead raises alerts or logs incidents when non-compliant behaviors are detected.

This can be particularly valuable in high-stakes environments, such as industrial automation or autonomous systems, where it’s critical to maintain system integrity while minimizing intervention.

- **Limitations**:
    - **No Dynamic Enforcement**: RV methods are passive; they do not correct or enforce compliance dynamically but only report violations. Manual intervention is needed, which can delay response times.
    - **Inadequate for Reactive CPS**: RV techniques lack the real-time intervention capability required in reactive CPS (e.g., autonomous vehicles or medical devices), where immediate correction is essential for safety.
- **Applications**:
    - **Fault Detection**: Identifying when the system deviates from expected behavior.
    - **Anomaly Detection**: Recognizing unusual or potentially harmful patterns in system behavior.
    - **Policy Compliance Reporting**: Logging and reporting compliance status for auditing and analysis purposes.

3.2.3. Synchronous Programming Approaches

Synchronous programming is particularly well-suited for defining deterministic safety automata in Cyber-Physical Systems (CPS), as it emphasizes predictable, time-driven behavior where the system reacts in a tightly controlled, deterministic manner to each cycle or external event.

In a synchronous model, time is divided into discrete steps or cycles, with each cycle representing a moment where the system assesses inputs, updates its internal state, and produces outputs. This cycle-based approach is foundational for safety-critical applications, as it ensures the system's behavior remains consistent and easily predictable.

One of the primary benefits of synchronous programming in CPS is its real-time responsiveness, which is essential for systems where safety policies must be enforced without delay. Each event or input triggers an immediate response, enabling the system to continuously monitor and enforce security policies within each cycle.

For example, if a safety policy detects a potentially hazardous condition, synchronous programming allows for a swift response in the very next cycle, minimizing the risk of damage or security breaches.

- **Usage**:
    - **Deterministic Safety Automata**: Synchronous programming allows for the creation of predictable, cycle-based automata that enforce strict policies.
    - **Predictable Behavior**: Each response is consistent and crucial for CPS applications that demand reliability.
    - **Real-Time Responsiveness**: Ensures the system enforces policies immediately within each cycle, making it responsive to real-time demands.
- **Limitations**:
    - **No Incremental Policy Support**: Traditional synchronous approaches are static, meaning policies are defined at setup and cannot be easily modified or added later.
    - **Challenging Scalability**: It is costly and time-consuming to reconfigure or update policies, limiting the system's adaptability to new threats or requirements, which is a drawback for dynamically evolving CPS security needs.

3.3 Flaws and Disadvantages

- **State Space Explosion**: As policies increase, the state space grows exponentially.
- **Re-certification Needs**: Each policy addition requires re-validation.
- **Loss of Modularity**: Policies are tightly coupled, making modifications challenging.

3.4 Proposed System

The proposed system is a compositional RE framework that allows each policy to be enforced by an individual enforcer, added incrementally to the system. By serially composing enforcers, the system supports scalable and modular security enforcement for CPS.

3.5 Functional Requirements

- **Dynamic Policy Addition**: Ability to add new policies without re-certifying the entire system.
- **Real-Time Monitoring**: Immediate enforcement of policies during system operation.
- **Modular Structure**: Independent enforcers for each policy, allowing isolated modifications

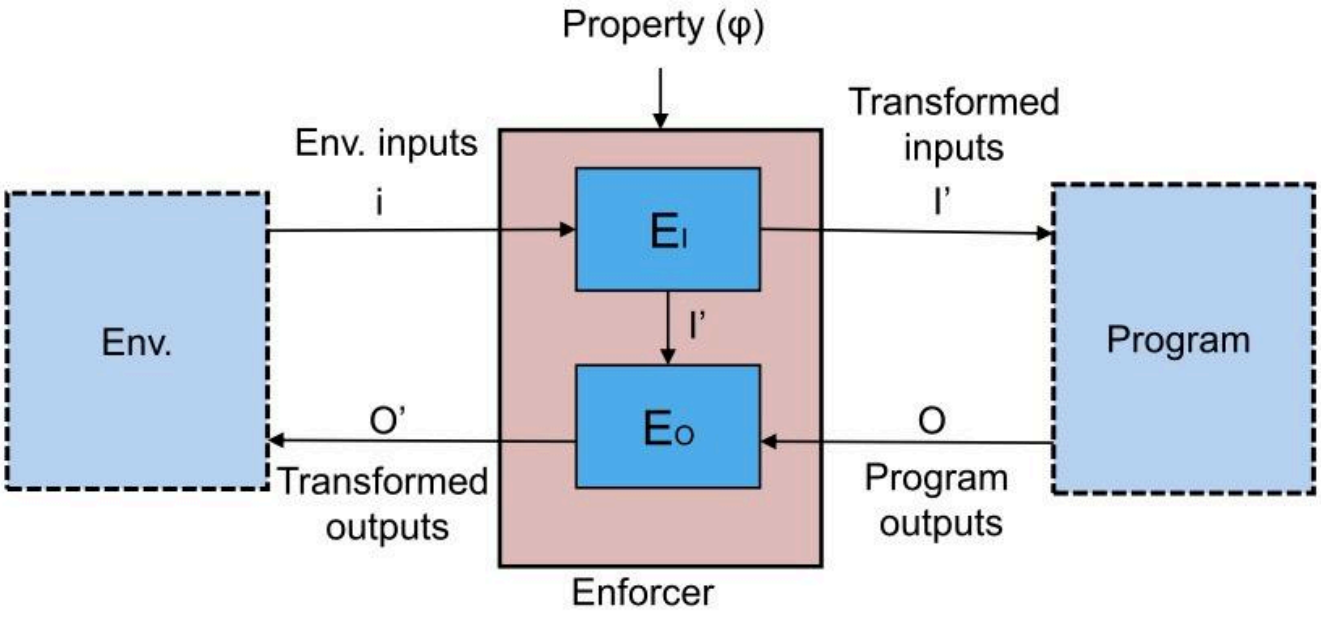
3.6 Non-Functional Requirements

- **Efficiency**: Minimize resource consumption while maintaining performance.
- **Reliability**: Ensure system adherence to policies under all conditions.
- **Scalability**: Support the addition of numerous policies without performance degradation.

Chapter 4

System Analysis

The design consists of a bi-directional enforcement model where input and output enforcers verify and adjust data flowing between the CPS and the environment. By intercepting and validating both inputs and outputs, this design prevents unsafe actions from reaching the CPS or the environment.

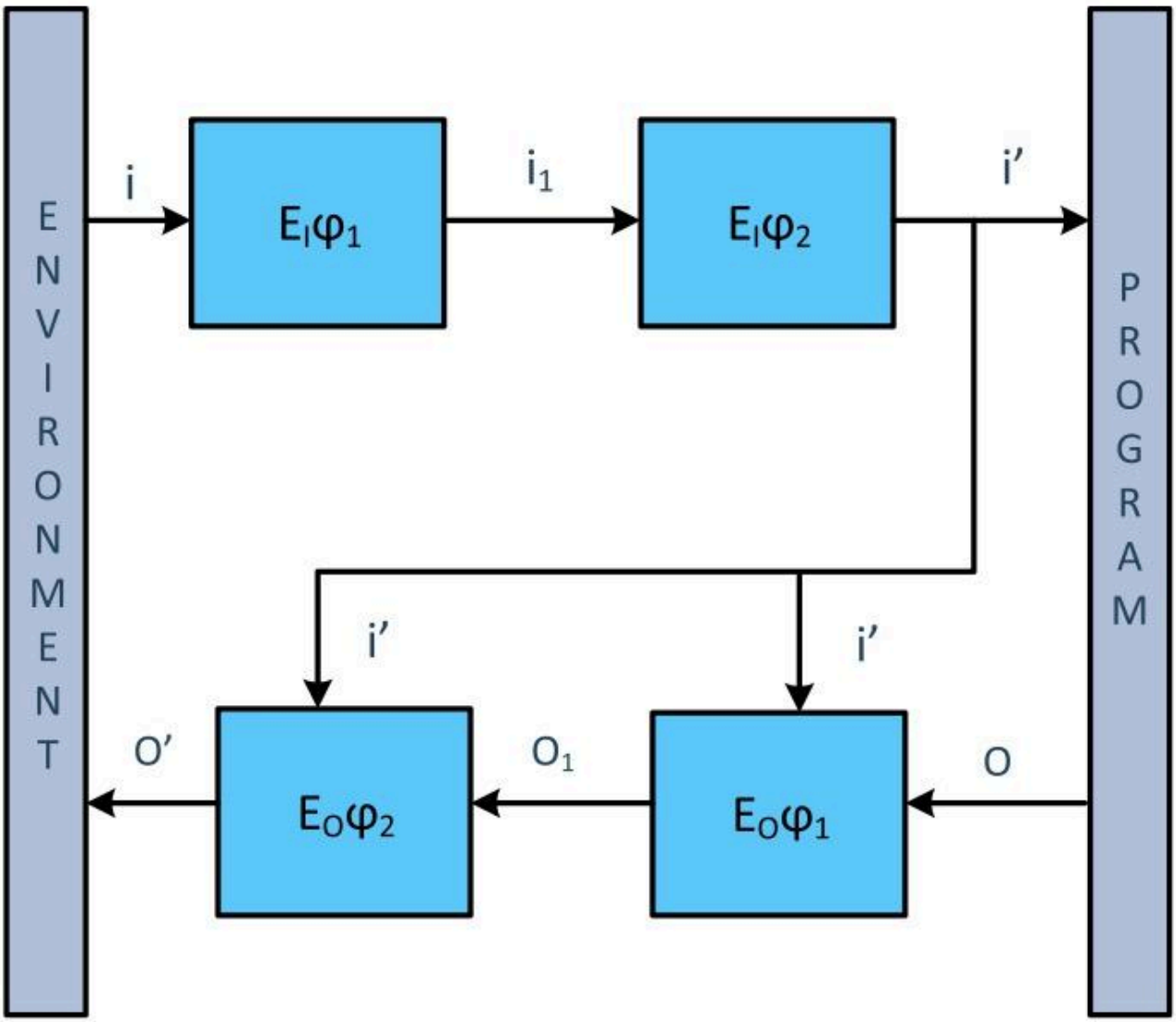


4.1 Proposed System Architecture

The methodology for this incremental security enforcement project begins with the development of a bi-directional runtime enforcement (RE) framework tailored to monitor and control both inputs and outputs in cyber-physical systems (CPS). This framework addresses the complex requirements of CPS, where new security policies must be incrementally added without triggering a complete overhaul of the system's enforcement structure. The project leverages Python for implementation, with custom tools such as the easy-rte-composition compiler, which extends the capabilities of the original easy-rte runtime enforcement engine to support serial policy composition.

To manage the complexity associated with enforcing multiple security policies, the project applies two approaches: a monolithic enforcement model, which combines all policies into one, and a serial composition model, which enforces policies individually in sequence.

The serial composition approach allows the addition of new policies without affecting the existing ones, thus preventing state-space explosion and maintaining efficient operation as the number of policies grows.



Chapter 5
Algorithms

The project introduces several key algorithms to implement the compositional runtime enforcement (RE) framework for Cyber-Physical Systems (CPS). These algorithms focus on enforcing policies on inputs and outputs, managing multiple policies using serial composition, and ensuring compliance by selecting compliant alternatives when violations occur. Below is a detailed overview of each algorithm.

5.1 Input and Output Enforcement Functions (EI and EO)

These functions enforce security policies on inputs (from the environment to the system) and outputs (from the system to the environment). They are essential for maintaining compliance with policies in real time by verifying and modifying events as needed.

Input Enforcement Function (EI)

- **Purpose**: To monitor and, if necessary, modify inputs from the environment before they are processed by the system.
- **Algorithm Steps**:
    1. **Receive Input**: For each new input event xxx from the environment, the function evaluates if it complies with a specified policy ϕ.
    2. **Compliance Check**: If xxx does not satisfy ϕ, it's deemed non-compliant.
    3. **Modification (Edit)**: The input is modified to a compliant version x′x'x′ by applying the **edit function**, which finds the closest alternative input that meets policy ϕ.
    4. **Pass to System**: The compliant input x′x'x′ is then passed to the system for processing.

Output Enforcement Function (EO)

- **Purpose**: To monitor and potentially modify outputs generated by the system before they are sent back to the environment.
- **Algorithm Steps**:
    1. **Receive Output**: Each output yyy generated by the system is evaluated against policy $\phi \backslash \phi\phi$.
    2. **Compliance Check**: If yyy violates ϕ, it is flagged as non-compliant.
    3. **Modification (Edit)**: The output is transformed into a compliant version y′y'y′ using the

edit function, ensuring it meets policy ϕ.

4. **Emit to Environment**: The compliant output y′y'y′ is sent to the environment, preserving system security and policy adherence.

5.2 Serial Composition of Enforcers

This algorithm allows the framework to enforce multiple policies by composing individual enforcers in series. Serial composition is crucial for handling incremental policy updates, as it prevents state space explosion and allows for modular policy management.

- **Purpose**: To enforce multiple policies incrementally by chaining enforcers, where each enforcer is responsible for a specific policy.
- **Algorithm Steps**:
    1. **Initialize Enforcers**: Define separate enforcers Eϕ1,Eϕ2,…,Eϕn ϕ1, ϕ_2,...ϕn Eϕ1,Eϕ2,…,Eϕn for each policy ϕ1,ϕ2,…,ϕn ϕ_1, ϕ_2, … ϕ_nϕ1,ϕ2,…,ϕn.
    2. **Compose Serially**: Arrange the enforcers in series. For any input or output event:
        - The event first passes through Eϕ1, which checks it against policy ϕ1.
        - If Eϕ1 modifies the event, the modified version is passed to Eϕ2, and so on, through Eϕn.
    3. **Final Output**: The final modified or validated event is output from the last enforcer in the chain.
    4. **Compliance Guarantee**: Each policy enforcer works independently, which ensures that each policy is enforced without causing interdependencies or exponential state growth.

5.3 Edit and Select Functions

These functions are responsible for finding and selecting policy-compliant alternatives when violations are detected. They minimize disruption to system behavior by selecting alternatives that closely match the original input or output.

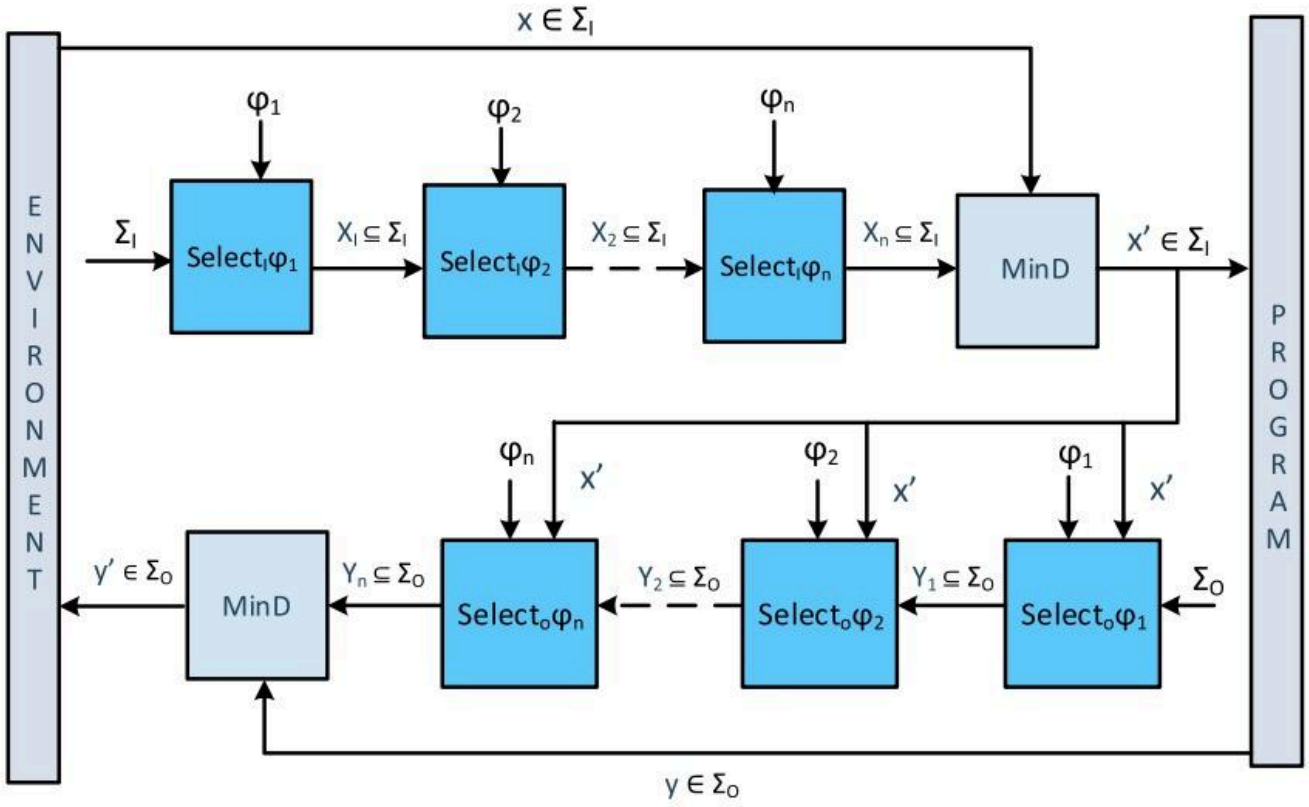
Edit Functions

- **editIϕI**: Handles inputs that violate policy, finding a compliant alternative that minimally deviates from the original input.
- **editOϕ**: Handles outputs that do not satisfy policy, selecting an alternative output that aligns

- with policy requirements.
- **Algorithm Steps for Edit Functions**:
    1. **Identify Violation**: When an input xxx or output yyy fails to comply with policy ϕ.
    2. **Generate Alternatives**: Compute a set of possible compliant alternatives X′X'X′ (for inputs) or Y′Y'Y′ (for outputs).
    3. **Select Closest Alternative**: Use the Select Function to choose the alternative x′x'x′ (or y′y'y′) that is closest to the original xxx (or yyy) while meeting policy ϕ.
    4. **Output Compliant Event**: The modified, policy-compliant event is then used as the new input/output for the system.

Select Functions

- **SelectIϕI**: Given a set of possible inputs X′X'X′, chooses a policy-compliant input that closely resembles the original input xxx.
- **SelectOϕ**: Given a set of possible outputs Y′Y'Y′, select a compliant output closest to the original output yyy.
- **Algorithm Steps for Select Functions**:
    1. **Input/Output Alternatives**: The function identifies all compliant alternatives for an input/output event.
    2. **Minimize Deviation**: Among the alternatives, choose the one with the minimal deviation from the original event, preserving system consistency.
    3. **Return Selected Event**: Return the closest compliant event for processing or emission.



5.4 Incremental Enforcement Function

This function defines the overall enforcement strategy by combining input and output enforcement functions to maintain compliance across both incoming and outgoing events in real-time.

- **Purpose**: To enforce policies incrementally by applying input and output enforcers sequentially, ensuring that all events comply without disrupting the entire system.
- **Algorithm Steps**:
    1. **Event Processing**: For each input/output event (x,y)(x, y)(x,y):
        - First, apply Input Enforcement (EI) to validate the input event.
        - After the system processes the input and produces an output, apply Output Enforcement (EO).
    2. **Modification for Compliance**: If either the input or output fails to comply, use the edit function to select the closest compliant alternative.
    3. **Causal Order Maintenance**: Ensure that each input is enforced before it affects output, maintaining causality and consistency in policy enforcement.
    4. **Emit Final Output**: The compliant input/output events are processed and emitted, ensuring system compliance with security policies in real-time.

5.5 Key Benefits of Using These Algorithms

The proposed compositional RE framework brings several notable advantages, addressing key challenges in securing Cyber-Physical Systems (CPS). This approach combines modularity, scalability, and real-time compliance, making it well-suited for dynamic, high-stakes environments where security policies need to be updated or added without disrupting system performance. Here's an in-depth look at each benefit:

5.5.1 Modularity

- **Independent Policy Management**: In traditional monolithic enforcement, all policies are bundled into a single enforcement structure, which makes modifying or isolating individual policies challenging. In contrast, the compositional RE framework uses serial composition to assign each policy its own enforcer. This design allows each policy to operate independently, without interdependencies or conflicts with other policies.
- **Flexible Policy Updates**: The modularity of the framework means new policies can be added, modified, or removed without needing to revalidate or redesign the entire enforcement system. This flexibility is essential in CPS environments where security policies may need regular updates in response to evolving threats.
- **Easier Debugging and Maintenance**: With each policy managed separately, debugging and maintenance are simplified. If an issue arises with a specific policy, engineers can isolate the corresponding enforcer without impacting others, reducing downtime and making troubleshooting more efficient.

5.5.2 Scalability

- **Avoids State Space Explosion**: A major limitation of monolithic approaches is the state space explosion problem, where the addition of new policies exponentially increases the state space, resulting in high memory and processing demands. By independently handling each policy enforcer, the compositional RE framework limits the state space to manageable levels. Each enforcer's state machine remains small and efficient, regardless of the number of policies, enabling the system to scale with minimal impact on performance.
- **Efficient Resource Usage**: As the framework does not merge policies into a single state machine, it uses computational resources more efficiently. This is particularly valuable for CPS with constrained resources, such as embedded systems in IoT devices, where memory and processing power are limited.

- **Linear Performance Scaling**: Testing has shown that the framework maintains a linear relationship between performance metrics (e.g., compile time, execution time) and the number of policies. This means that as new policies are added, system performance remains predictable and manageable, allowing CPS to scale securely.

5.5.3 Real-Time Compliance

- **Immediate Policy Enforcement**: The framework's Input (EI) and Output (EO) enforcement functions to ensure that policies are enforced on every incoming and outgoing event, maintaining continuous compliance with security policies in real-time. This capability is vital in CPS applications (e.g., autonomous vehicles and healthcare devices), where a delay in enforcement could lead to unsafe or unauthorized actions.
- **Dynamic Response to Threats**: By intercepting and verifying each input and output, the framework can dynamically adjust to security threats as they arise. If a policy violation is detected, the enforcer can modify the event to a compliant alternative in real time, preventing the propagation of unsafe behaviors throughout the system.
- **Consistent System Behavior**: The edit and select function**s** ensure that non-compliant inputs and outputs are replaced with the closest compliant alternatives, minimizing the deviation from intended system behavior. This preserves the system's functionality while ensuring security compliance, which is crucial for applications where reliability and predictability are essential.

5.5.4 Adaptability to Evolving Threats

- **Incremental Policy Integration**: With the compositional RE framework, policies can be added incrementally as new threats emerge without needing to overhaul or re-certify the entire system. This adaptability allows CPS to respond to a constantly evolving threat landscape, maintaining security compliance without compromising system uptime or performance.
- **Reduced Re-Certification Needs**: In safety-critical CPS applications, adding new policies typically requires re-certifying the entire system, which can be time-consuming and costly. The modular nature of this framework allows for seamless policy updates with minimal recertification, ensuring compliance while reducing operational disruptions.

5.5.5 Improved Reliability and Security

- **Enhanced Fault Isolation**: The independent enforcers isolate policy functions, meaning that any issue with one policy enforcer does not compromise the operation of others. This isolation enhances reliability by reducing the risk of cascading errors that could compromise the entire system.
- **Predictable and Transparent Operation**: Each enforcer operates in a deterministic, cycle-based fashion, producing predictable behavior. This predictability simplifies system auditing, as each policy enforcer behaves consistently, making it easier to verify that security policies are being enforced reliably.

CHAPTER 6

LIMITATIONS OF MONOLITHIC AND INCREMENTAL ENFORCEMENT

6.1 LIMITATIONS OF MONOLITHIC ENFORCEMENT

Monolithic enforcement presents significant challenges in modern cyber-physical systems due to its computational complexity, scalability issues, and high response time. As the number of policies increases, enforcement becomes exponentially difficult to manage, leading to inefficiencies in real-time applications.

6.1.1 Computational Complexity

One of the most significant drawbacks of monolithic enforcement is its exponential increase in computational complexity. The system must process all possible policy combinations simultaneously, making it infeasible for real-time operations.

- The number of states grows exponentially with the number of policies.
- Large-scale systems struggle with real-time enforcement due to excessive computational demands.

For example, a system with 50 security policies would require evaluating states, which is computationally impossible in real-time environments.

6.1.2 Scalability Challenges

Monolithic enforcement is not scalable because it requires complete recomputation whenever a new policy is added.

- In highly dynamic security environments, policies frequently change to adapt to emerging threats.
- Full recomputation delays policy updates, leaving the system vulnerable to new threats.

6.1.3 High Latency and Response Time

- Every policy must be evaluated before enforcement actions take effect.
- This leads to delayed enforcement, making monolithic approaches unsuitable for real-time applications like autonomous vehicles, drones, and industrial control systems.

For instance, if each policy check takes 1 second and there are 10 policies, enforcement takes:

This delay makes monolithic enforcement impractical for applications requiring instant decisions.

6.1.4 Limited Adaptability

- The rigid structure of monolithic enforcement makes integrating new security measures difficult.
- Every policy change requires a full system recertification, increasing administrative overhead.

6.1.5 Delayed Threat Response

- Security violations are corrected only after all policies have been evaluated, leading to delayed responses.
- Critical threats remain unmitigated until the entire policy set is processed.

6.1.6 High Computational Load

- Monolithic enforcement demands significant processing power and memory, reducing efficiency in embedded systems.
- Energy inefficiency leads to shorter battery life in battery-operated devices like drones and IoT sensors.

6.2 LIMITATIONS OF INCREMENTAL ENFORCEMENT

While incremental enforcement is a more scalable alternative to monolithic enforcement, it also has limitations that must be considered.

6.2.1 Dependency on Policy Structure

Incremental enforcement evaluates policies sequentially, meaning its efficiency depends on how policies are structured.

- Poorly defined policies can lead to inefficient enforcement sequences.
- Systems must be designed carefully to ensure policies are processed in the correct order.

6.2.2 Potential Policy Conflicts

- Independent enforcers may create policy conflicts if their interactions are not well-managed.
- For example, one policy may allow an action that another policy restricts, leading to inconsistent behavior.

6.2.3 Increased Complexity in Large Systems

- As the number of policies grows, debugging and policy interactions become increasingly complex.
- Systems require careful management to prevent policy overlap and unintended enforcement issues.

6.2.4 Limited Applicability in Some Scenarios

While incremental enforcement is highly effective for adaptive security policies, some high-security environments may still require monolithic enforcement.

- Systems requiring immediate, centralized control may not benefit from incremental processing.
- Some regulatory environments mandate strict security models that incremental enforcement may not fully support.

6.3 COMPARATIVE ANALYSIS

A direct comparison between monolithic and incremental enforcement reveals key differences in performance, efficiency, and adaptability. Monolithic enforcement excels in strict policy adherence, ensuring comprehensive validation of all policies before execution. However, its high computational complexity and latency make it impractical for real-time applications. Incremental enforcement, in contrast, offers faster response times and greater adaptability but requires structured policy sequencing to avoid inconsistencies. Monolithic enforcement offers a comprehensive approach to policy evaluation but struggles with scalability and high computational costs. Incremental enforcement, in contrast, provides flexibility and efficiency, particularly for real-time applications, but requires well-structured policies and careful management to prevent conflicts.

Performance benchmarks indicate that incremental enforcement significantly reduces computational overhead compared to monolithic enforcement. For instance, in a system with 50 policies, monolithic enforcement requires evaluating states, whereas incremental enforcement only processes policies sequentially, reducing complexity to $O(N \times S)$. This makes incremental enforcement more suitable for real-time cyber-physical systems, such as autonomous drones and IoT security frameworks. However, monolithic enforcement remains preferable in environments requiring exhaustive policy validation, such as financial transaction security and regulated industrial systems. Ultimately, the choice of enforcement mechanism should align with system goals. Hybrid enforcement approaches, combining aspects of both monolithic and incremental enforcement, could bridge the gap between comprehensive security and real-time efficiency, paving the way for more resilient cyber-physical systems.

CHAPTER 7

CYBER ATTACK SCENARIOS IN DRONE SYSTEMS

Drones are vulnerable to cyber-physical attacks that can manipulate their navigation, commands, and operational integrity. In this section, we analyze seven critical attack scenarios, their impact, and how incremental enforcement mitigates these threats.

7.1 GPS Spoofing Attack

7.1.1 Attack Description

- Attackers transmit fake GPS signals, tricking the drone into miscalculating its position.
- This forces the drone to change its route unknowingly.

7.1.2 Impact on Drone System

- Incorrect navigation, leading to flight into restricted zones.
- Increased risk of collisions with obstacles or other aircraft.
- Potential hijacking of drones for unauthorized missions.

7.1.3 Countermeasures Using Incremental Enforcement

- **GPS Integrity Check Policy**: Continuously compares GPS coordinates with onboard Inertial Measurement Unit (IMU) data.
- **Enforcement Action**: If a sudden jump in GPS location is detected, the drone ignores the new coordinates.
- Reverts to the last known valid GPS position.
- Switches to manual override mode if multiple GPS failures occur.

7.2. Command Injection Attack

7.2.1 Attack Description

- Attackers inject malicious commands into the drone's control system.
- This can trigger unauthorized actions like self-destruct, forced landing, or redirection to another location.

7.2.2 Impact on Drone System

- Loss of control, making the drone a potential weapon or intelligence leak.
- Can be used for ransomware attacks, demanding payment to regain control.
- Unauthorized deliveries or payload drops in hostile locations.

7.2.3 Countermeasures Using Incremental Enforcement

- **Command Authentication Policy**: Validates all incoming commands against a list of pre-approved actions.
- Enforcement Action:
    - Rejects unauthorized commands immediately.
    - Reverts to safe mode (hover, return to base).
    - If repeated injection attempts are detected, the drone activates emergency security protocols.

7.3. Battery Drain Attack

7.3.1 Attack Description

- Attackers manipulate power consumption by forcing:
    - Frequent takeoff-landing cycles.
    - Unnecessary sensor activation.
    - High-speed maneuvers that deplete the battery faster.

7.3.2 Impact on Drone System

- Mid-air power failure, causing drones to crash.
- Inability to complete mission objectives due to premature shutdown.
- Loss of communication with the command center.

7.3.3 Countermeasures Using Incremental Enforcement

- Battery Conservation Policy: Monitors power usage and optimizes flight efficiency.
- Enforcement Action:
    - Prioritizes Return to Base (RTB) over all other commands.
    - Disables non-essential sensors to conserve power.
    - Automatically reduces speed when the battery falls below a critical threshold

7.4. Denial of Service (DoS) Attack

7.4.1 Attack Description

- Attackers flood the drone's communication channels or processing unit with excessive requests.
- Causes high CPU load, leading to delayed responses or system crashes.

7.4.2 Impact on Drone System

- Drones become unresponsive, ignoring legitimate commands.
- Increased risk of mid-air failure due to processing overload.
- Loss of synchronization in drone swarms, affecting mission execution.

7.4.3 Countermeasures Using Incremental Enforcement

- Request Rate Limiting Policy: Monitors command frequency and detects abnormal request patterns.
- Enforcement Action:
- Blocks excessive command requests from untrusted sources.
- Prioritizes emergency actions like Return to Base (RTB).
- Drops redundant communication packets to maintain system stability.

7.5. Sensor Tampering Attack

7.5.1 Attack Description

- Attackers manipulate onboard sensors by:
    - Jamming radar signals to mislead obstacle detection.
    - Overloading cameras with bright light (optical attack).
    - Feeding false IMU data to affect navigation.

7.5.2 Impact on Drone System

- Faulty altitude readings, causing unexpected crashes.
- Navigation errors, leading to off-course flying.
- Inability to identify threats correctly in military drones.

7.5.3 Countermeasures Using Incremental Enforcement

- Sensor Redundancy Policy: Compares readings from multiple sensors before making decisions.
- Enforcement Action:
- Ignores outlier sensor data that deviates significantly from expected values.
- Switches to secondary sensor readings when anomalies are detected.
- If multiple sensors fail, the drone enters emergency landing mode.

7.6. Payload Tampering Attack (For Delivery Drones)

7.6.1 Attack Description

- Attackers attempt to steal, replace, or compromise drone payloads.
- This could involve:
    - Replacing high-value packages with illegal items.
    - Bypassing security checkpoints using drones.
    - Hijacking drones to steal medical or commercial supplies.

7.6.2 Impact on Drone System

- Loss of valuable shipments, leading to financial damage.
- Security risks, where drones carry unauthorized payloads.
- Reputational damage for delivery services and law enforcement agencies.

7.6.3 Countermeasures Using Incremental Enforcement

- Secure Payload Policy: Ensures continuous weight and integrity checks on transported goods.
- Enforcement Action:
- If a weight change is detected mid-flight, the drone returns to base.
- Locks payload compartments with biometric or encrypted access.
- Triggers onboard security cameras if tampering is detected.

7.7. Collision Induction Attack

7.7.1 Attack Description

- Attackers attempt to force drone collisions by:
    - Generating false obstacle alerts using electromagnetic signals.
    - Overloading sensors with conflicting data, forcing unnecessary detours.

7.2 Impact on Drone System

- Mid-air collisions with other drones or obstacles.
- Delays in mission execution due to unnecessary evasive actions.
- Disruption of coordinated drone swarms in surveillance and military operations.

7.3 Countermeasures Using Incremental Enforcement

- Collision Validation Policy: Compares obstacle detection data from multiple sources (Radar, LiDAR, Vision AI).
- Enforcement Action:
- Ignores single-source anomalies (e.g., false radar echoes).
- Uses AI-based obstacle recognition to prevent false alarms.
- Triggers manual override if continuous false alerts are detected.

CHAPTER 8

SOFTWARE DESIGN

8.1 OVERVIEW

The Compositional Runtime Enforcement Framework for Drone Swarm Security is designed to detect, mitigate, and prevent security violations in real time for drone swarms. This system ensures that drones adhere to predefined security policies and automatically correct deviations using enforcement mechanisms.

The framework handles various cyber attack scenarios, including:

- GPS Spoofing: Altering drone coordinates to mislead navigation.
- Command Injection: Unauthorized control commands leading to unsafe drone actions.
- Battery Drain Attacks: Forcing drones into inefficient operations to exhaust power.
- Denial of Service (DoS): Overloading drones with excessive commands to trigger timeouts.

To enforce security, the system provides two enforcement strategies:

- Monolithic Enforcement: Checks and enforces all policies at once, which can lead to computational overhead.
- Incremental Enforcement: Applies policies sequentially, improving efficiency and reducing state space explosion.

The framework is implemented in MATLAB and Simulink for advanced simulations, along with 3D visualizations to track drone behaviors and attacks in real time.

8.2 Architectural Design

The software is built using a modular and layered architecture, ensuring flexibility, maintainability, and real-time enforcement.

8.2.1 Policy Management Service

The Policy Management Service is responsible for handling security policies dynamically.

- CRUD Operations: Policies can be created, read, updated, and deleted as needed.

- Serial Composition of Enforcers: Enables dynamic composition of multiple enforcement policies for layered security.
- Input/Output Validation: Ensures the correctness of drone state before enforcement is applied.

8.2.2 Runtime Enforcement Engine

The Runtime Enforcement Engine ensures real-time validation and correction of security violations.

- Modular Enforcers: Each security policy has a dedicated enforcement function.
- Algorithms for Input/Output Enforcement:
    - EI (Edit Input): Modifies incoming data before execution.
    - EO (Edit Output): Adjusts drone behavior after execution.
- Serial Composition of Enforcement Functions: Enforcement functions are applied in sequence to prevent cascading failures.

8.2.3 Monitoring and Alerting Service

This module provides real-time logging and visualization of policy violations.

- Log Non-Compliance: Records every detected violation for auditing.
- Real-Time Alerts: Notifies when a drone violates security policies.
- Visual Representation: Displays drone behaviors, attack effects, and corrections in a 3D simulation environment.

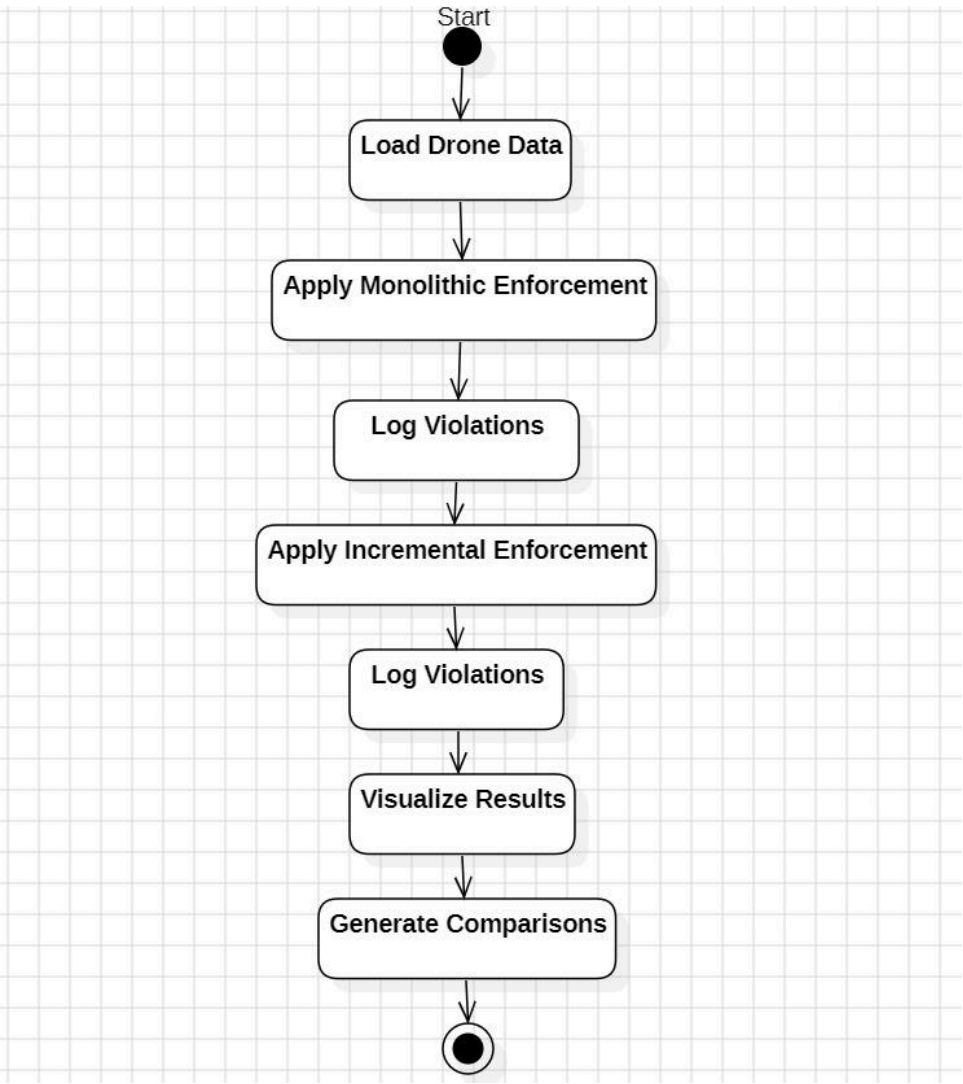
8.2.4 Storage System

The framework includes a database layer optimized for fast retrieval of policies and logs.

- Policies Database: Stores active security policies for enforcement.
- Violation Logs Database: Records violations for post-analysis and debugging.

8.3 Data Flow in the System

The data flow process of the enforcement framework follows a structured sequence, ensuring effective security enforcement and visualization of violations. The flow is depicted in the corresponding diagram and detailed below:



8.3.1 Start and Data Loading

The process begins with an initial state where the system initializes enforcement mechanisms and prepares for security checks.

The system then loads the real-time drone dataset, including:

- Drone position (x, y, z)
- Velocity
- Commands issued
- Battery levels

This step ensures that security policies are applied accurately.

8.3.2 Security Enforcement

8.3.2.1 Apply Monolithic Enforcement

- The monolithic enforcer applies all security policies at once.
- Ensures strict compliance but may cause unnecessary restrictions or inefficiencies.

8.3.2.2 Apply Incremental Enforcement

- The incremental enforcer applies security policies sequentially, addressing violations as they occur.
- This approach is more adaptive and efficient compared to monolithic enforcement.

8.3.2.3 Log Violations

Records any security violations such as:

- Boundary breaches
- Speed violations
- Unauthorized commands
- Stored in the Logger module for further analysis.

8.3.3 Analysis and Visualization

Visualize Results

- Uses the Visualizer module to graphically represent enforcement outcomes, including:
  - Drone movement in 3D space
  - Areas where enforcement was triggered
  - Timeline of security policy activations

Generate Comparisons

- Compares the following metrics:
  - Number of violations detected under each enforcement strategy
  - Performance impact (e.g., drone movement restrictions due to policies)
  - Efficiency of each method
- This helps determine whether incremental enforcement is more suitable for real-time operations.

Chapter 9

Tech Stack

9.1 Introduction to Matlab

MATLAB (short for matrix laboratory) is a high-level programming language and environment designed primarily for numerical computation, data analysis, and visualization. It is widely used in various fields such as engineering, physics, finance, and machine learning due to its powerful capabilities for handling matrix operations, solving mathematical problems, and visualizing data in 2D and 3D plots.

MATLAB allows users to write scripts and functions, perform data analysis, implement algorithms, and create graphical user interfaces (GUIs). It also offers extensive built-in functions for tasks like linear algebra, signal processing, image processing, and statistics, making it a popular tool for both academic research and industry applications.

9.2 Simulink

Simulink is an add-on product for MATLAB that provides a graphical environment for modeling, simulating, and analyzing multi domain dynamic systems. It allows users to design and simulate systems using block diagrams, making it particularly useful for engineers and researchers working in areas such as control systems, signal processing, communications, and embedded systems.

Simulink offers a wide range of pre-built blocks representing mathematical operations, signals, and system components, which can be easily connected to create models of complex systems. Users can simulate how their system will behave in real-world scenarios, and it also provides tools for testing, analyzing, and optimizing designs. Additionally, Simulink supports automatic code generation for real-time implementation on hardware, making it highly effective for developing embedded systems.

9.3 Simscape

Simscape is an add-on product for MATLAB and Simulink that provides a set of tools for modeling and simulating physical systems. It extends Simulink's capabilities by enabling the simulation of systems that involve mechanical, electrical, hydraulic, pneumatic, thermal, and other physical domains.

With Simscape, users can model physical components (such as gears, motors, batteries, and pumps)

using predefined blocks that represent real-world physical systems. These blocks are connected in a graphical environment, allowing for system-level modeling and simulation of complex, multi-domain systems. Simscape also supports the integration of different physical domains, making it ideal for applications like automotive systems, robotics, and energy systems.

Simscape's key advantage is its ability to model physical behavior with higher accuracy and in a more intuitive manner than traditional methods, making it an essential tool for designing and testing physical systems before implementing them in hardware.

9.4 Stateflow

It is an extension of Simulink that provides a graphical tool for designing and simulating state machines, flowcharts, and logic-based systems. It is used to model systems that require discrete logic, such as control systems, decision-making algorithms, event-driven systems, and policy-based systems. Stateflow uses states, transitions, and events to represent the behavior of a system in response to various inputs and conditions.

9.5 Control System Toolbox:

The Control System Toolbox is a MATLAB add-on product that provides a comprehensive set of tools for the analysis, design, and simulation of control systems. This toolbox is widely used in fields such as aerospace, automotive, robotics, and process control, where precise control over dynamic systems is required. The toolbox offers a rich set of functions for designing and tuning controllers, analyzing system stability, and simulating how systems will behave in response to various inputs and disturbances.

9.6 UAV Toolkit:

The UAV Toolbox in MATLAB is specifically designed for simulating, designing, and analyzing Unmanned Aerial Vehicles (UAVs), making it a powerful tool for working with Cyber-Physical Systems (CPS) such as drone swarms.

CHAPTER 10

STRUCTURE OF PROJECT

10.1 Cyber Attack

A cyber attack in a Cyber-Physical System (CPS) refers to an attack targeting the integration of computer-based control systems and physical processes in critical infrastructure. CPS are used in various sectors, such as manufacturing, healthcare, energy, transportation, and smart cities, where real-time data from sensors and actuators control and monitor physical systems.

10.1.1 Initialization (initializeAttack.m)

In MATLAB, the function initializeAttack.m suggests the initialization of some kind of cyber attack scenario, potentially for a simulation or testing purpose in a controlled environment, such as in a Cyber-Physical System (CPS) framework.

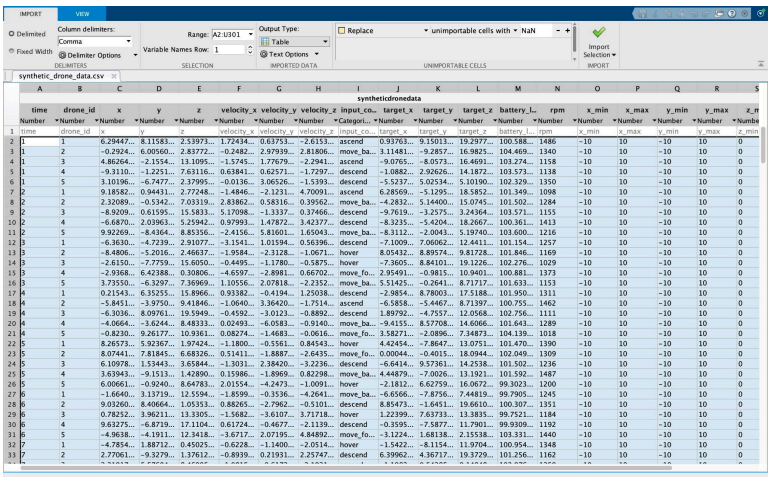
10.1.2 Simulation (simulateAttack.m)

The simulateAttack.m function in MATLAB would simulate a cyber attack on a target system, like a Cyber-Physical System (CPS). This function would typically involve running a simulation based on the attack parameters initialized in the initializeAttack.m function.

10.1.3 Data

In our project, we have used a dataset, which has been synthesized using matlab. The fields have been taken in accordance with the research papers.

10.1.3.1 synthetic_drone_data.csv



10.1.4 Enforcers

They are mechanisms or components that ensure the security, safety, and proper functioning of the system by enforcing policies, regulations, or constraints. They play a crucial role in protecting CPS from various threats, ensuring that the system operates correctly and securely, and maintaining system integrity despite potential adversities or failures.

They are edit functions that are implemented in real time.

10.1.4.1 Battery Enforcer (Enforce_battery.m)

This edit function is responsible for enforcing battery policies.

10.1.4.2 Boundary Enforcer (Enforce_boundary.m)

This edit function is responsible for boundary violations.

10.1.4.3 Conflict Enforer (Enforce_command_conflict.m)

Handles an attack where the drone is bombarded with multiple commands.

10.1.4.4 Timeout Enforcer (Enforce_timeout.m)

Responsible for ensuring that when the data packets don't reach the drone, the drone goes back to the selected height.

10.1.4.5 Velocity Enforcer (Enforce_velocity.m)

Responsible for ensuring that the drone does not cross the velocity limits, when an attack happens.

10.1.5 Logs

Primarily responsible in keeping track of all the violations of the policies that have occurred.

10.1.5.1 Incremental Framework Violations (incremental_violations.txt):

Maintains a record of violations that have taken place in the incremental enforcement framework.

10.1.5.2 Log Script (log_violations.m):

This file contains the code required for maintaining the log files for incremental framework violations and monolithic enforcement framework violations.

10.1.5.3 Monolithic Framework Violations (monolithic_violations.txt)

Maintains a record of violations that have taken place in the monolithic enforcement framework.

10.1.6 Policies

Policies in Cyber-Physical Systems (CPS) are crucial for ensuring the effective, secure, and reliable operation of systems that integrate physical processes with computational elements, such as sensors, actuators, and controllers.

They are select functions.

10.1.6.1 Battery Policy (policy_battery.m)

This select function is responsible for maintaining the battery levels in the drone, when an attack takes place.

10.1.6.2 Boundary Policy (policy_boundary.m)

This select function is responsible for adding the boundary limits such as the maximum altitude, perimeter the drone cannot cross.

10.1.6.3 Conflict Policy (policy_command_conflict.m)

This select function comes in when the attacker uses multiple commands simultaneously on the drone which can affect the movement of the drone. This is used to help mitigate from such attacks.

10.1.6.4 Timeout Policy (policy_timeout.m)

This policy kicks in when the drone is attacked in such a way where it does not receive the data packets from the user, the drone uses a timeout policy and proceeds to move to a predefined position.

10.1.6.5 Velocity Policy (policy_velocity.m)

This policy is responsible for describing the maximum and minimum velocity the drone should maintain.

10.1.7 security

10.1.7.1 apply_enforcement.m

It is responsible for ensuring all the enforcers are implemented.

10.1.7.2 detectAndMitigate.m

It is used to detect any cyberattacks and initiates the procedures to mitigate the cyber attacks accordingly.

10.1.7.3 initializeSecurityFramework.m:

Initializes the incremental security framework in matlab.

10.1.8 Visualization

10.1.8.1 animate_drones.m

creates a 2d simulation of the drones from the data set.

10.1.8.2 visualize_results.m

creates a 3d model of the drones in space

10.1.8.3 visualize_state_space.m

shows the difference between the state spaces of the monolithic and incremental enforcement techniques

10.1.8.4 visualizeSwarm.m

creates a 3d simulation of drones flying in 3d space. it shows when the drone is under attack and after the threat is mitigated

10.1.9 incremental_enforcement.m

demonstrates the modularity of this technique. calls the policies and enforces as needed

10.1.10 initializeSwarm.m

Initializes the UAV swarm by setting up their initial positions, trajectories, and security parameters.

10.1.11 loadDataset.m

loads the drone data set for simulation

10.1.12 main.m

main script that runs the framework and simulation

10.1.13 monolithic_enforcement.m

is an aggregation of all the different security policies

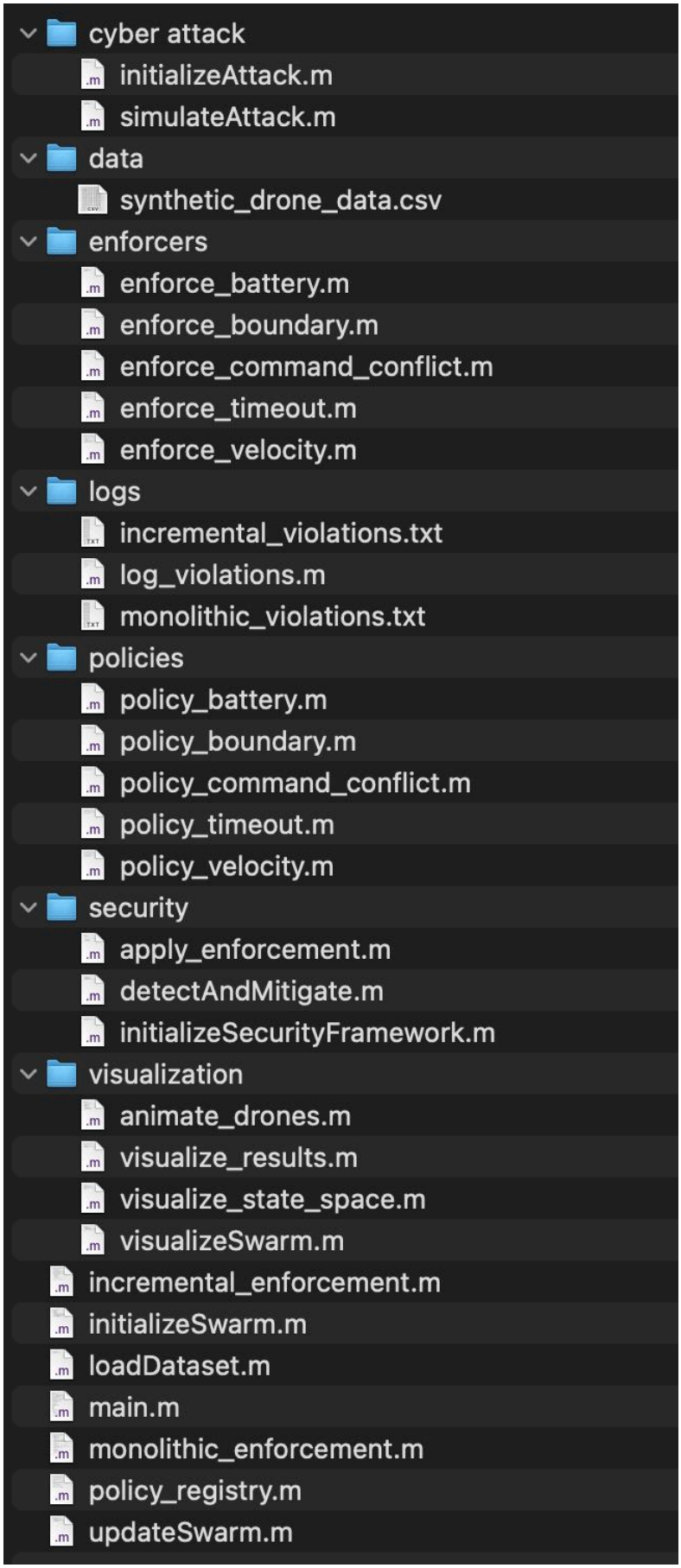
10.1.14 policy_registry.m

keeps track of each security policy and can be updated when new policies are added

10.1.15 updateSwarm.m

Updates the UAV positions, applies security enforcement, and refreshes the 3D visualization at each simulation step.

Project structure



Chapter 11

Code Implementation

11.1 Monolithic Enforcement

```
% Enforce Boundary Policy
if droneState.x < boundaries.x_min || droneState.x > boundaries.x_max || ...
   droneState.y < boundaries.y_min || droneState.y > boundaries.y_max || ...
   droneState.z < boundaries.z_min || droneState.z > boundaries.z_max
    violations{end+1} = sprintf('Boundary violation for Drone %d at time %d', droneState.drone_id, droneState.time);
    droneState.x = min(max(droneState.x, boundaries.x_min), boundaries.x_max);
    droneState.y = min(max(droneState.y, boundaries.y_min), boundaries.y_max);
    droneState.z = min(max(droneState.z, boundaries.z_min), boundaries.z_max);
    hasViolation = true;
end

% Enforce Timeout Policy
if droneState.timeout > maxTimeout
    violations{end+1} = sprintf('Timeout violation for Drone %d at time %d', droneState.drone_id, droneState.time);
    droneState.input_command = 'hover';
    hasViolation = true;
end

% Enforce Velocity Policy
velocity = sqrt(droneState.velocity_x^2 + droneState.velocity_y^2 + droneState.velocity_z^2);
if velocity > maxVelocity
    violations{end+1} = sprintf('Velocity violation for Drone %d at time %d', droneState.drone_id, droneState.time);
    scale = maxVelocity / velocity;
    droneState.velocity_x = droneState.velocity_x * scale;
    droneState.velocity_y = droneState.velocity_y * scale;
    droneState.velocity_z = droneState.velocity_z * scale;
    hasViolation = true;
end

% Enforce Battery Conservation Policy
if droneState.battery_level <= 10
    violations{end+1} = sprintf('Battery violation for Drone %d at time %d', droneState.drone_id, droneState.time);
    droneState.input_command = 'hover';
```

all the policies are aggregated into one script which is the cause of state space explosion

11.2 Incremental enforcement

```
% Apply policies incrementally
for j = 1:length(policies)
    if ~policies{j}(droneState)
        % Log the violation
        violations{end+1} = sprintf('Policy %d violation for Drone %d at time %d', ...
                                    j, droneState.drone_id, droneState.time);
        % Enforce the policy
        droneState = enforcers{j}(droneState);
    end
end
```

the modular policies are added as per requirement

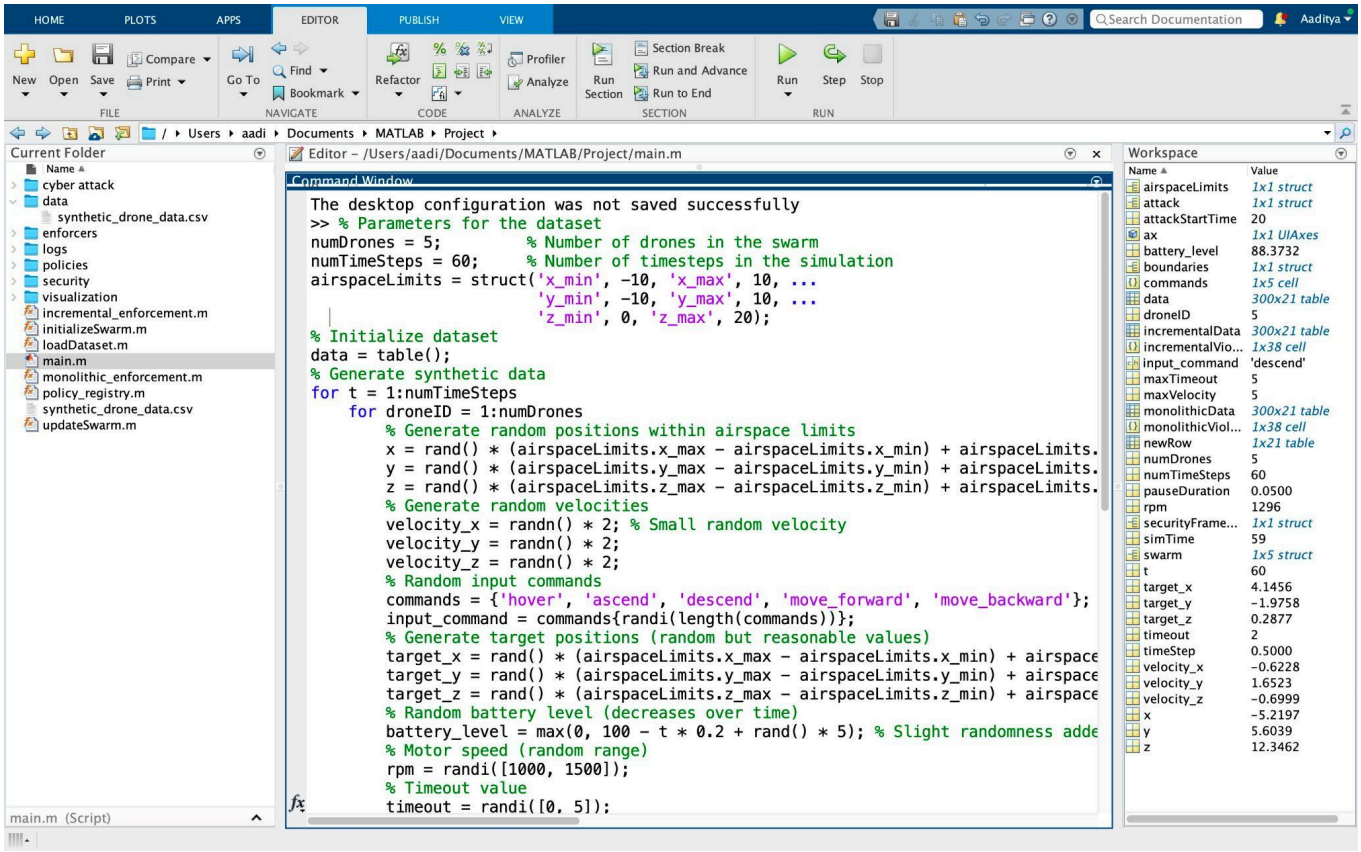
11.3 Policy registry

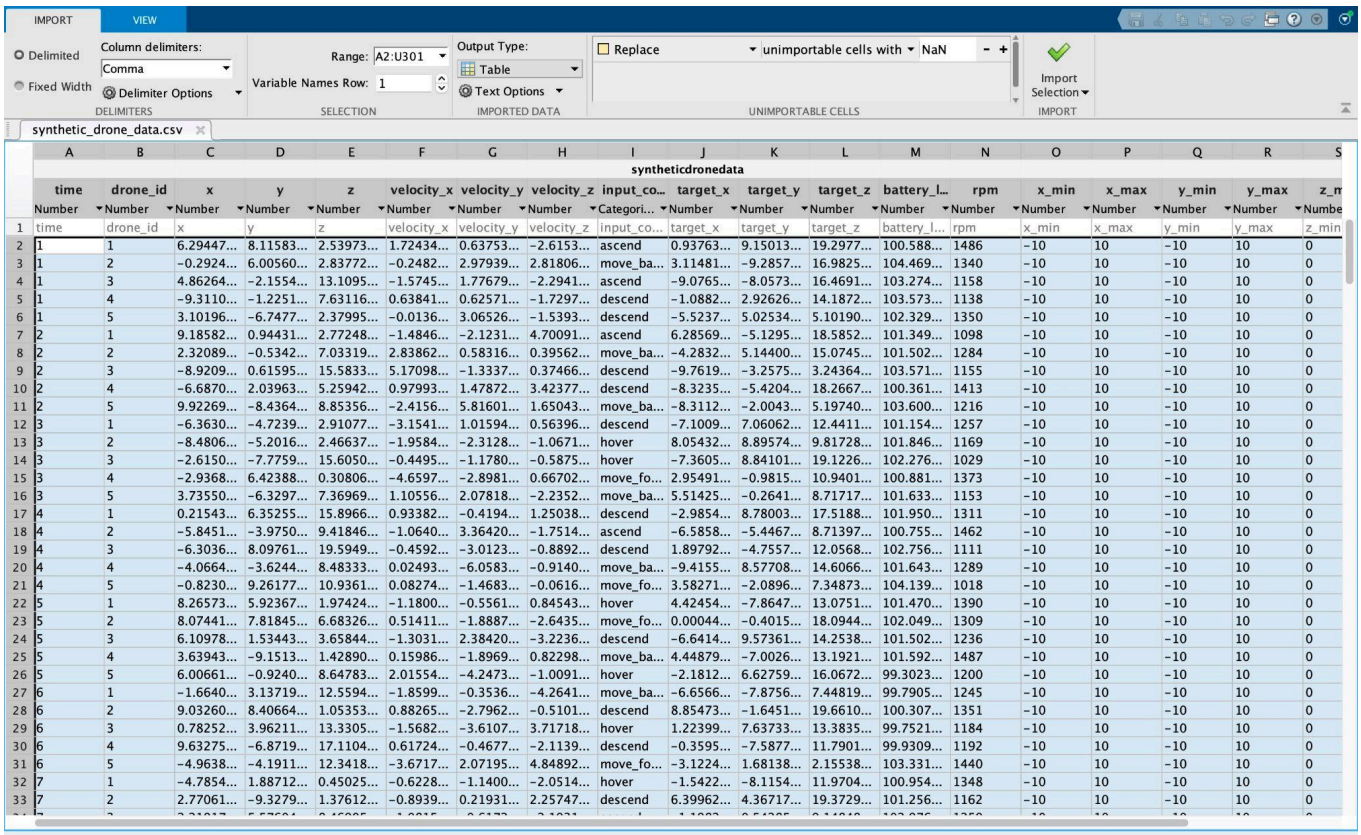
```
% Initialize policies and enforcers
policies = {};
enforcers = {};

% 1. Boundary Policy
policies{end + 1} = @(state) policy_boundary(state, boundaries);
enforcers{end + 1} = @(state) enforce_boundary(state, boundaries);

% 2. Command Conflict Policy
policies{end + 1} = @(state) policy_command_conflict(state);
enforcers{end + 1} = @(state) enforce_command_conflict(state);

% 3. Battery Conservation Policy
policies{end + 1} = @(state) policy_battery(state);
enforcers{end + 1} = @(state) enforce_battery(state);

% 4. Timeout Policy
policies{end + 1} = @(state) policy_timeout(state, maxTimeout);
enforcers{end + 1} = @(state) enforce_timeout(state);

% 5. Velocity Policy
policies{end + 1} = @(state) policy_velocity(state, maxVelocity);
enforcers{end + 1} = @(state) enforce_velocity(state, maxVelocity);
```

The policy registry is updated as and when new policies are added

11.4 Enforcers

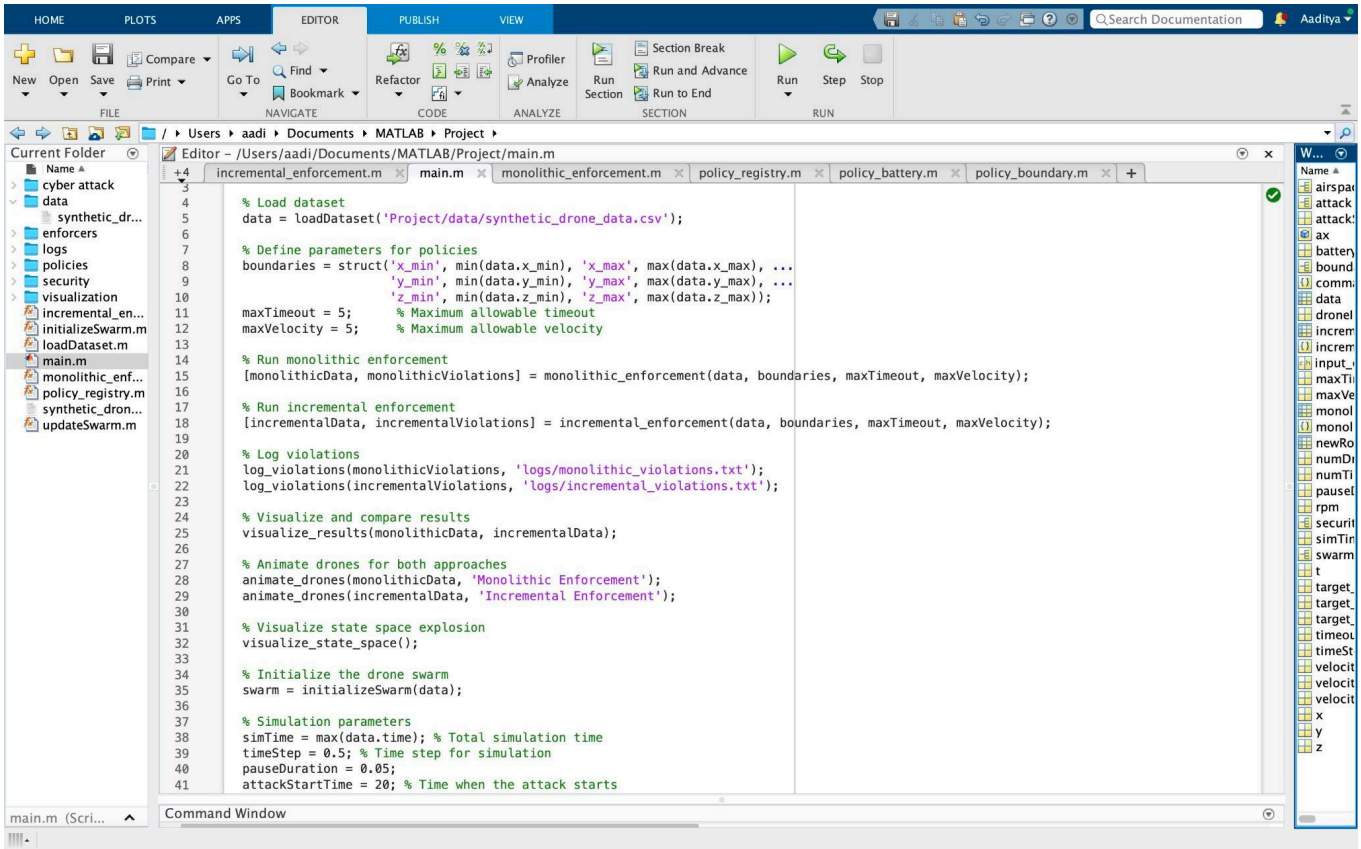
11.5 Data set synthesis

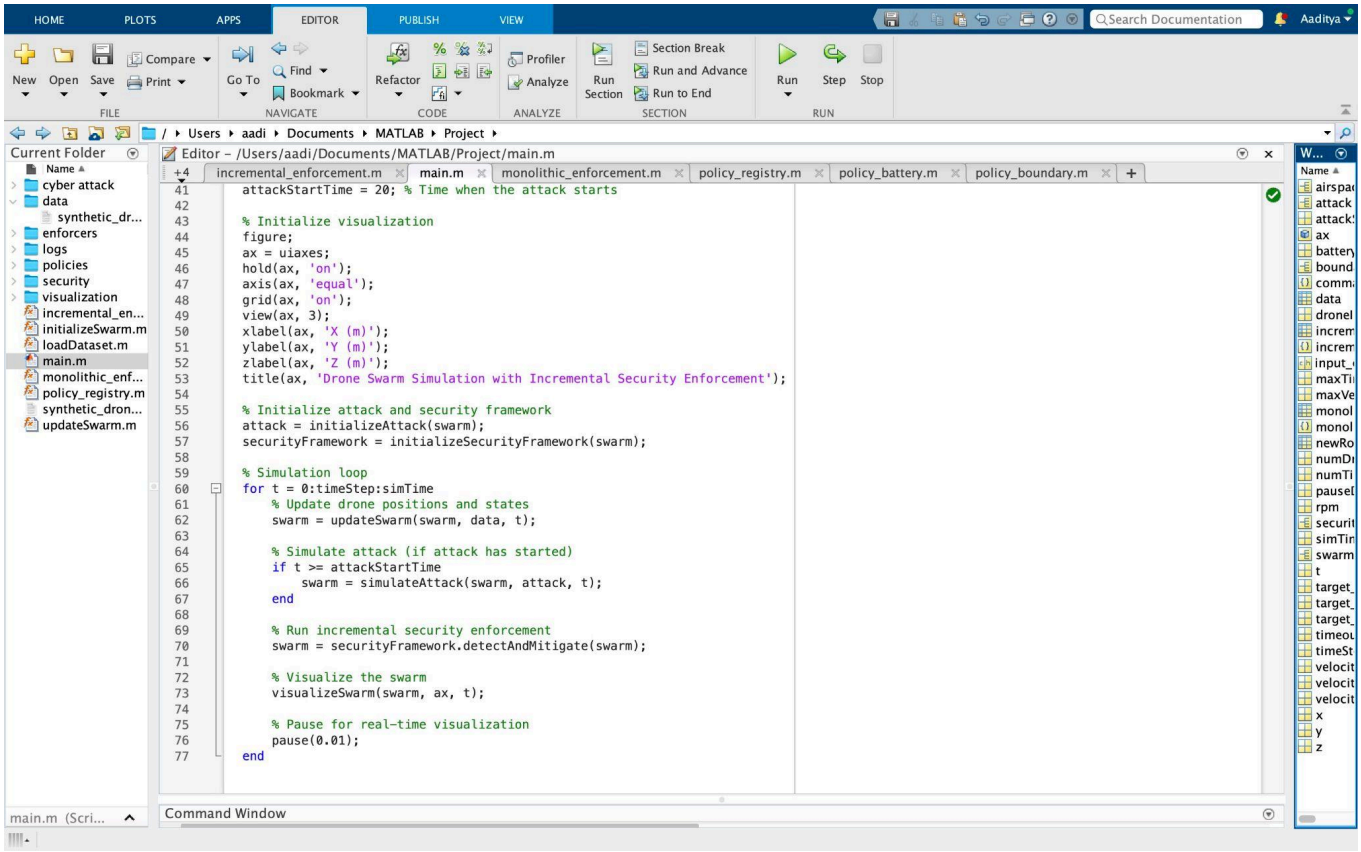




	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
	time	drone_id	x	y	z	velocity_x	velocity_y	velocity_z	input_co...	target_x	target_y	target_z	battery_l...	rpm	x_min	x_max	y_min	y_max	z_m
	Number	Number	Number	Number	Number	Number	Number	Number	Categori...	Number	Number	Number	Number	Number	Number	Number	Number	Number	Numbe
1	time	drone_id	x	y	z	velocity_x	velocity_y	velocity_z	input_co...	target_x	target_y	target_z	battery_l...	rpm	x_min	x_max	y_min	y_max	z_min
2	1	1	6.29447...	8.11583...	2.53973...	1.72434...	0.63753...	-2.6153...	ascend	0.93763...	9.15013...	19.2977...	100.588...	1486	-10	10	-10	10	0
3	1	2	-0.2924...	6.00560...	2.83772...	-0.2482...	2.97939...	2.81806...	move_ba...	3.11481...	-9.2857...	16.9825...	104.469...	1340	-10	10	-10	10	0
4	1	3	4.86264...	-2.1554...	13.1095...	-1.5745...	1.77679...	-2.2941...	ascend	-9.0765...	-8.0573...	16.4691...	103.274...	1158	-10	10	-10	10	0
5	1	4	-9.3110...	-1.2251...	7.63116...	0.63841...	0.62571...	-1.7297...	descend	-1.0882...	2.92626...	14.1872...	103.573...	1138	-10	10	-10	10	0
6	1	5	3.10196...	-6.7477...	2.37995...	-0.0136...	3.06526...	-1.5393...	descend	-5.5237...	5.02534...	5.10190...	102.329...	1350	-10	10	-10	10	0
7	2	1	9.18582...	0.94431...	2.77248...	-1.4846...	-2.1231...	4.70091...	ascend	6.28569...	-5.1295...	18.5852...	101.349...	1098	-10	10	-10	10	0
8	2	2	2.32089...	-0.5342...	7.03319...	2.83862...	0.58316...	0.39562...	move_ba...	-4.2832...	5.14400...	15.0745...	101.502...	1284	-10	10	-10	10	0
9	2	3	-8.9209...	0.61595...	15.5833...	5.17098...	-1.3337...	0.37466...	descend	-9.7619...	-3.2575...	3.24364...	103.571...	1155	-10	10	-10	10	0
10	2	4	-6.6870...	2.03963...	5.25942...	0.97993...	1.47872...	3.42377...	descend	-8.3235...	-5.4204...	18.2667...	100.361...	1413	-10	10	-10	10	0
11	2	5	9.92269...	-8.4364...	8.85356...	-2.4156...	5.81601...	1.65043...	move_ba...	-8.3112...	-2.0043...	5.19740...	103.600...	1216	-10	10	-10	10	0
12	3	1	-6.3630...	-4.7239...	2.91077...	-3.1541...	1.01594...	0.56396...	descend	-7.1009...	7.06062...	12.4411...	101.154...	1257	-10	10	-10	10	0
13	3	2	-8.4806...	-5.2016...	2.46637...	-1.9584...	-2.3128...	-1.0671...	hover	8.05432...	8.89574...	9.81728...	101.846...	1169	-10	10	-10	10	0
14	3	3	-2.6150...	-7.7759...	15.6050...	-0.4495...	-1.1780...	-0.5875...	hover	-7.3605...	8.84101...	19.1226...	102.276...	1029	-10	10	-10	10	0
15	3	4	-2.9368...	6.42388...	0.30806...	-4.6597...	-2.8981...	0.66702...	move_fo...	2.95491...	-0.9815...	10.9401...	100.881...	1373	-10	10	-10	10	0
16	3	5	3.73550...	-6.3297...	7.36969...	1.10556...	2.07818...	-2.2352...	move_ba...	5.51425...	-0.2641...	8.71717...	101.633...	1153	-10	10	-10	10	0
17	4	1	0.21543...	6.35255...	15.8966...	0.93382...	-0.4194...	1.25038...	descend	-2.9854...	8.78003...	17.5188...	101.950...	1311	-10	10	-10	10	0
18	4	2	-5.8451...	-3.9750...	9.41846...	-1.0640...	3.36420...	-1.7514...	ascend	-6.5858...	-5.4467...	8.71397...	100.755...	1462	-10	10	-10	10	0
19	4	3	-6.3036...	8.09761...	19.5949...	-0.4592...	-3.0123...	-0.8892...	descend	1.89792...	-4.7557...	12.0568...	102.756...	1111	-10	10	-10	10	0
20	4	4	-4.0664...	-3.6244...	8.48333...	0.02493...	-6.0583...	-0.9140...	move_ba...	-9.4155...	8.57708...	14.6066...	101.643...	1289	-10	10	-10	10	0
21	4	5	-0.8230...	9.26177...	10.9361...	0.08274...	-1.4683...	-0.0616...	move_fo...	3.58271...	-2.0896...	7.34873...	104.139...	1018	-10	10	-10	10	0
22	5	1	8.26573...	5.92367...	1.97424...	-1.1800...	-0.5561...	0.84543...	hover	4.42454...	-7.8647...	13.0751...	101.470...	1390	-10	10	-10	10	0
23	5	2	8.07441...	7.81845...	6.68326...	0.51411...	-1.8887...	-2.6435...	move_fo...	0.00044...	-0.4015...	18.0944...	102.049...	1309	-10	10	-10	10	0
24	5	3	6.10978...	1.53443...	3.65844...	-1.3031...	2.38420...	-3.2236...	descend	-6.6414...	9.57361...	14.2538...	101.502...	1236	-10	10	-10	10	0
25	5	4	3.63943...	-9.1513...	1.42890...	0.15986...	-1.8969...	0.82298...	move_ba...	4.44879...	-7.0026...	13.1921...	101.592...	1487	-10	10	-10	10	0
26	5	5	6.00661...	-0.9240...	8.64783...	2.01554...	-4.2473...	-1.0091...	hover	-2.1812...	6.62759...	16.0672...	99.3023...	1200	-10	10	-10	10	0
27	6	1	-1.6640...	3.13719...	12.5594...	-1.8599...	-0.3536...	-4.2641...	move_ba...	-6.6566...	-7.8756...	7.44819...	99.7905...	1245	-10	10	-10	10	0
28	6	2	9.03260...	8.40664...	1.05353...	0.88265...	-2.7962...	-0.5101...	descend	8.85473...	-1.6451...	19.6610...	100.307...	1351	-10	10	-10	10	0
29	6	3	0.78252...	3.96211...	13.3305...	-1.5682...	-3.6107...	3.71718...	hover	1.22399...	7.63733...	13.3835...	99.7521...	1184	-10	10	-10	10	0
30	6	4	9.63275...	-6.8719...	17.1104...	0.61724...	-0.4677...	-2.1139...	descend	-0.3595...	-7.5877...	11.7901...	99.9309...	1192	-10	10	-10	10	0
31	6	5	-4.9638...	-4.1911...	12.3418...	-3.6717...	2.07195...	4.84892...	move_fo...	-3.1224...	1.68138...	2.15538...	103.331...	1440	-10	10	-10	10	0
32	7	1	-4.7854...	1.88712...	0.45025...	-0.6228...	-1.1400...	-2.0514...	hover	-1.5422...	-8.1154...	11.9704...	100.954...	1348	-10	10	-10	10	0
33	7	2	2.77061...	-9.3279...	1.37612...	-0.8939...	0.21931...	2.25747...	descend	6.39962...	4.36717...	19.3729...	101.256...	1162	-10	10	-10	10	0

11.6 main.m





11.7 SECURITY

11.7.1 apply enforcement.m

```
function apply_enforcement()
    global uavList;

    for i = 1:length(uavList)
        drone = uavList(i).uav;
        [position, orientation, isValid] = uavList(i).trajectory();

        if ~isValid
            continue;
        end

        state = struct('x', position(1), 'y', position(2), 'z', position(3), 'drone_id', i);

        if detect_gps_spoofing(state)
            enforce_gps_recovery(state);
        elseif detect_command_injection(state)
            enforce_command_safety(state);
        elseif detect_battery_attack(state)
            enforce_battery_conservation(state);
        end
    end
end
```

11.7.2 detectAndMitigate.m

```
function swarm = detectAndMitigate(swarm)
    for i = 1:length(swarm)
        % Example: Detect boundary breach
        pos = swarm(i).position;
        bounds = swarm(i).bounds;
        if pos(1) < bounds(1) || pos(1) > bounds(2) || ...
           pos(2) < bounds(3) || pos(2) > bounds(4) || ...
           pos(3) < bounds(5) || pos(3) > bounds(6)
            swarm(i).state = 'mitigated';
            swarm(i).velocity = -swarm(i).velocity; % Reverse direction to mitigate
        end
    end
end
```

11.7.3 initializeSecurityFramework.m

```
function securityFramework = initializeSecurityFramework(swarm)
    % Initialize the security framework
    securityFramework = struct();
    securityFramework.detectAndMitigate = @detectAndMitigate; % Link to the enforcement function
end
```

11.7.4 initializeAttack.m

```
function attack = initializeAttack(swarm)
    % Define the attack scenario
    attack = struct();
    attack.type = 'boundaryBreach'; % Type of attack
    attack.targetDrone = 1; % Target the first drone in the swarm
end
```

```
function swarm = simulateAttack(swarm, attack, time)
    % Example: Boundary breach attack
    if strcmp(attack.type, 'boundaryBreach')
        targetDrone = attack.targetDrone;
        swarm(targetDrone).position = swarm(targetDrone).position + [10, 0, 0];
        % Force the drone to breach the boundary
        swarm(targetDrone).state = 'underAttack';
    end
end
```

11.8 log_violations.m

```
function log_violations(violations, logFilePath)
    % LOG_VIOLATIONS: Logs policy violations into a specified text file.

    % Open the file for appending
    fid = fopen(logFilePath, 'a');
    if fid == -1
        error('Error opening the log file: %s', logFilePath);
    end

    % Log each violation
    for i = 1:length(violations)
        fprintf(fid, '%s\n', violations{i});
    end

    % Close the file
    fclose(fid);
end
```

11.9 Visualizations

11.9.1 animate_drones

```
function animate_drones(data, titleText)
    figure;
    hold on;

    numDrones = length(unique(data.drone_id));
    scatterHandles = gobjects(numDrones, 1);
    for i = 1:numDrones
        scatterHandles(i) = scatter3(NaN, NaN, NaN, 100, 'filled');
    end

    xlim([min(data.x), max(data.x)]);
    ylim([min(data.y), max(data.y)]);
    zlim([min(data.z), max(data.z)]);
    xlabel('X'); ylabel('Y'); zlabel('Z');
    title(titleText);

    uniqueTimes = unique(data.time);
    for t = uniqueTimes'
        currentData = data(data.time == t, :);
        for i = 1:numDrones
            droneData = currentData(currentData.drone_id == i, :);
            scatterHandles(i).XData = droneData.x;
            scatterHandles(i).YData = droneData.y;
            scatterHandles(i).ZData = droneData.z;
        end
        pause(0.2);
    end
end
```

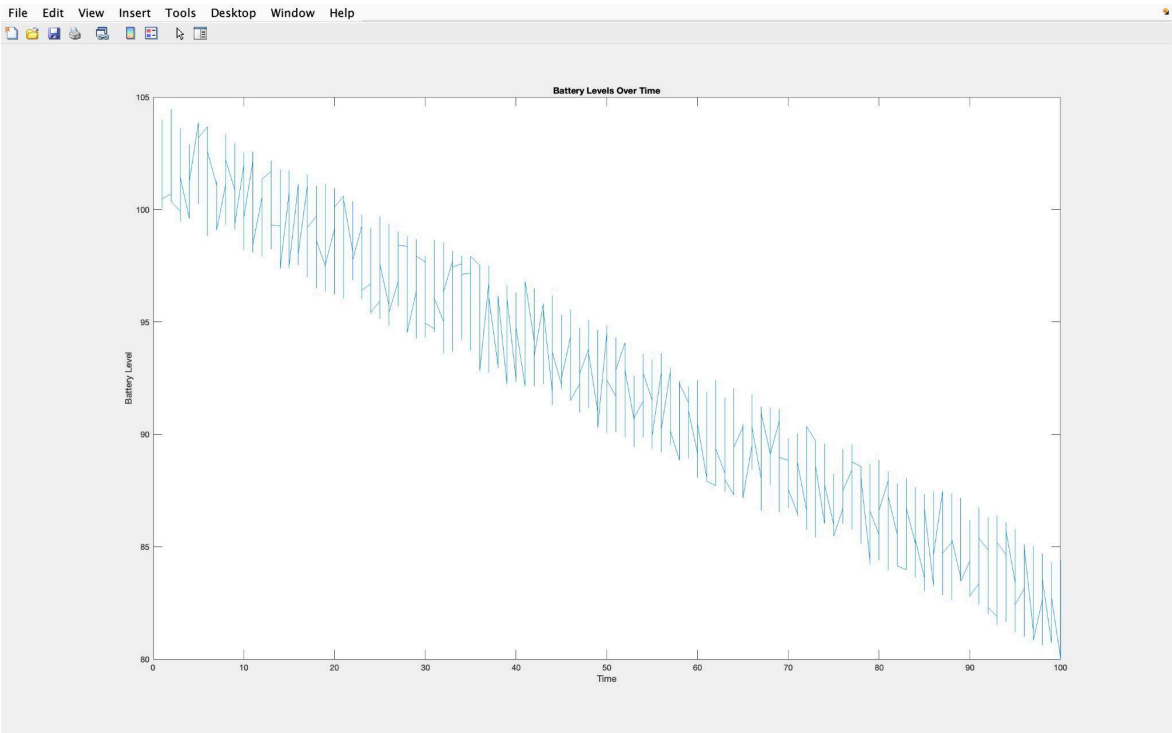
```
function visualize_results(monolithicData, incrementalData)
    figure;

    subplot(1, 2, 1);
    scatter3(monolithicData.x, monolithicData.y, monolithicData.z, 10, monolithicData.drone_id, 'filled');
    title('Monolithic Enforcement');
    xlabel('X'); ylabel('Y'); zlabel('Z');
    grid on;

    subplot(1, 2, 2);
    scatter3(incrementalData.x, incrementalData.y, incrementalData.z, 10, incrementalData.drone_id, 'filled');
    title('Incremental Enforcement');
    xlabel('X'); ylabel('Y'); zlabel('Z');
    grid on;
end
```

```
function visualize_state_space()
    numPolicies = 1:10;
    monolithicStates = 2.^numPolicies; % Exponential growth
    incrementalStates = numPolicies;  % Linear growth

    figure;
    plot(numPolicies, monolithicStates, '-o', 'DisplayName', 'Monolithic');
    hold on;
    plot(numPolicies, incrementalStates, '-o', 'DisplayName', 'Incremental');
    xlabel('Number of Policies');
    ylabel('Number of States');
    title('State Space Explosion: Monolithic vs Incremental');
    legend;
    grid on;
end
```

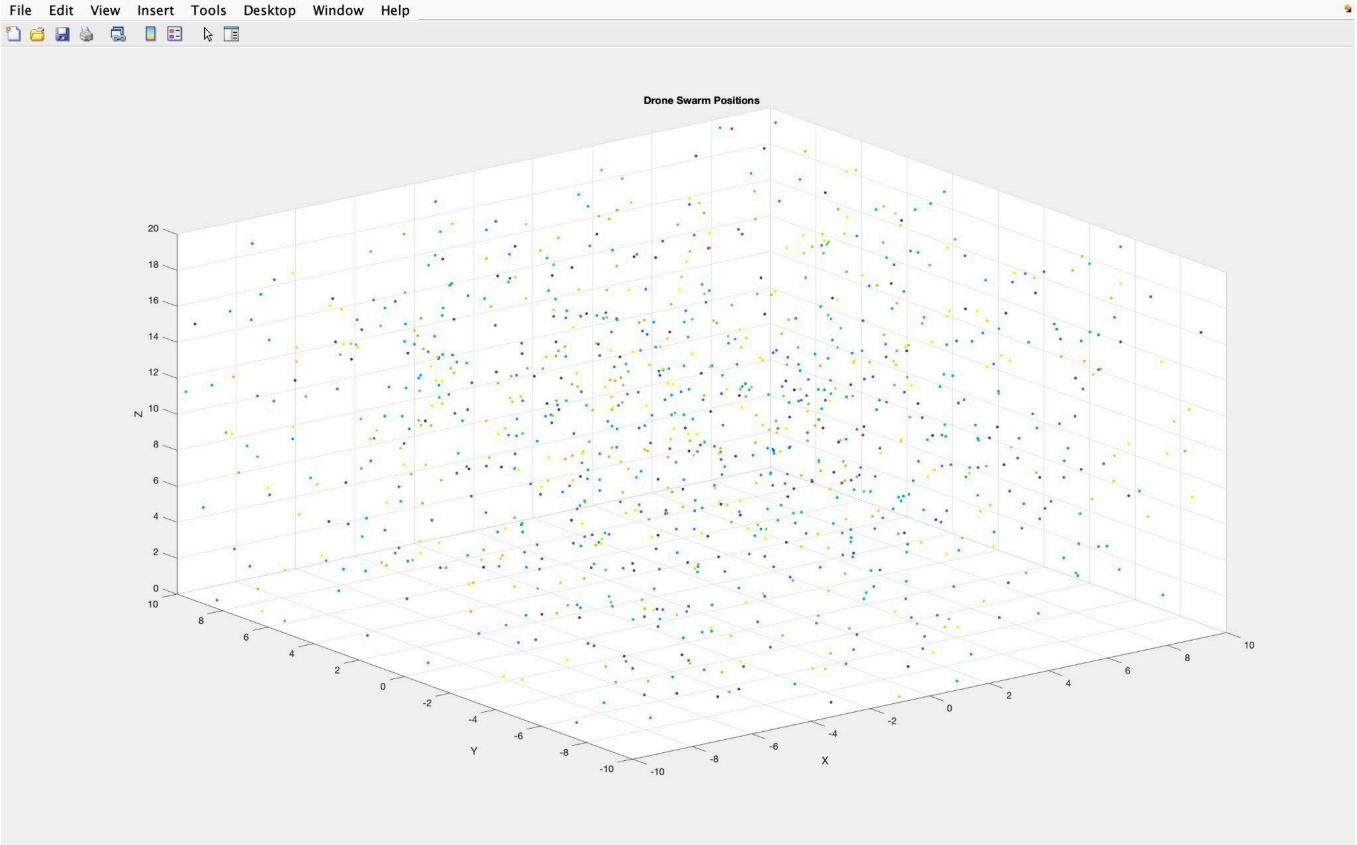
```
function visualizeSwarm(swarm, ax, time)
    cla(ax); % Clear the axes
    for i = 1:length(swarm)
        pos = swarm(i).position;
        if strcmp(swarm(i).state, 'normal')
            scatter3(ax, pos(1), pos(2), pos(3), 'filled', 'b'); % Blue for normal drones
        elseif strcmp(swarm(i).state, 'underAttack')
            scatter3(ax, pos(1), pos(2), pos(3), 'filled', 'r'); % Red for attacked drones
        elseif strcmp(swarm(i).state, 'mitigated')
            scatter3(ax, pos(1), pos(2), pos(3), 'filled', 'g'); % Green for mitigated drones
        end
    end
    title(ax, sprintf('Drone Swarm Simulation (Time = %.1f s)', time));
    drawnow;
end
```

```
Velocity violation for Drone 1 at time 2
Velocity violation for Drone 1 at time 4
Velocity violation for Drone 8 at time 4
Velocity violation for Drone 6 at time 5
Velocity violation for Drone 1 at time 9
Velocity violation for Drone 3 at time 9
Velocity violation for Drone 5 at time 10
Velocity violation for Drone 6 at time 11
Velocity violation for Drone 8 at time 13
```

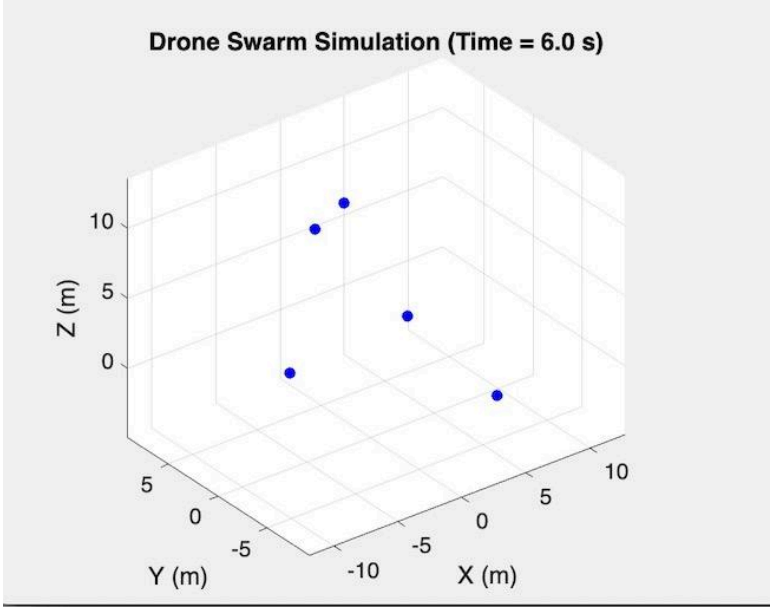
Battery Fluctuations in the drone due to attack



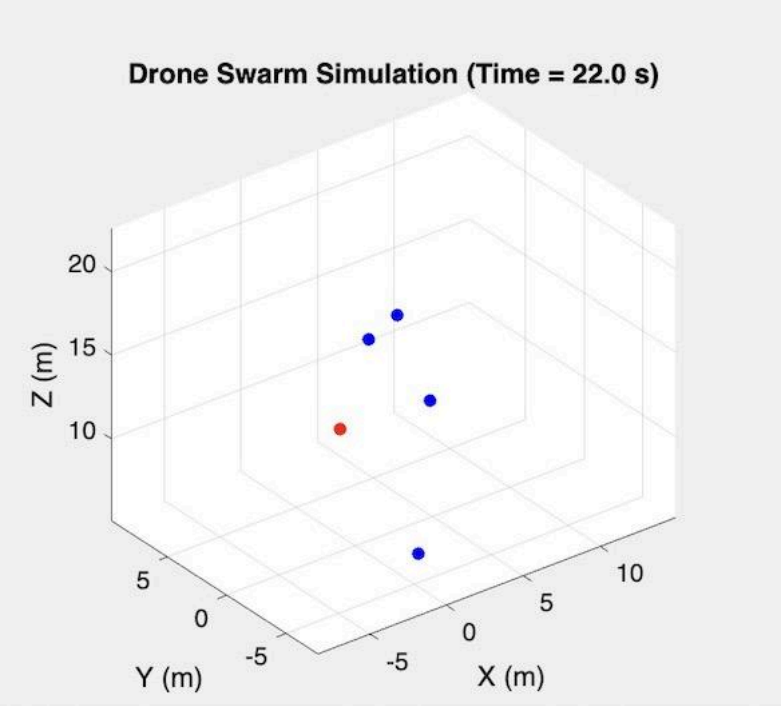
Drone swarm in a 3d space



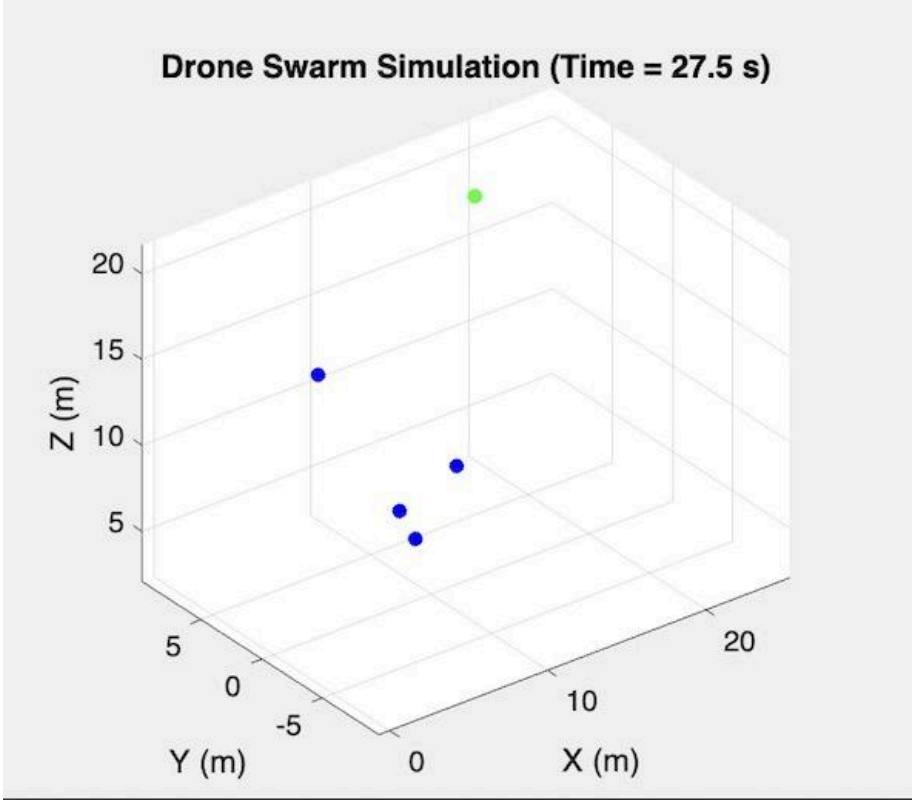
Drones flying



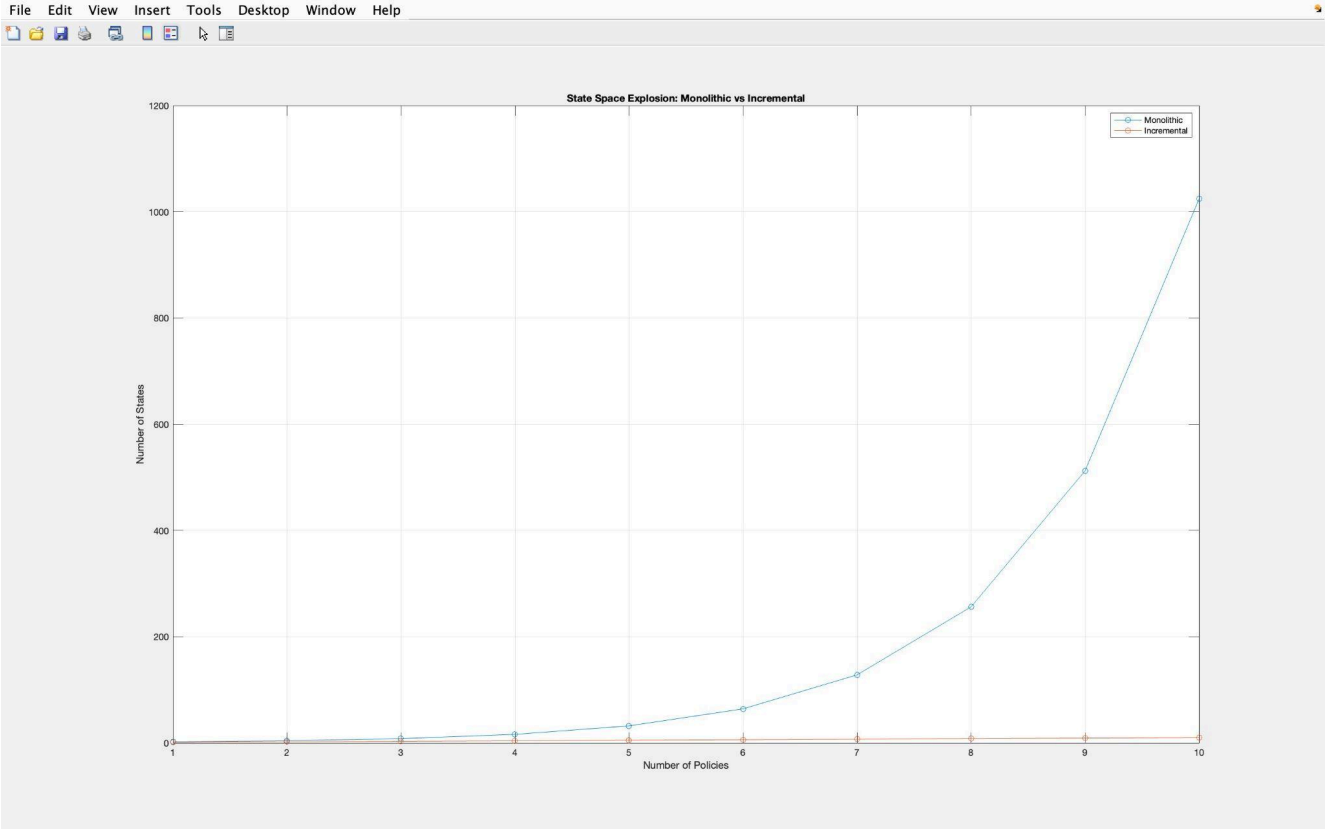
Drone under attack



Cyber attack mitigated

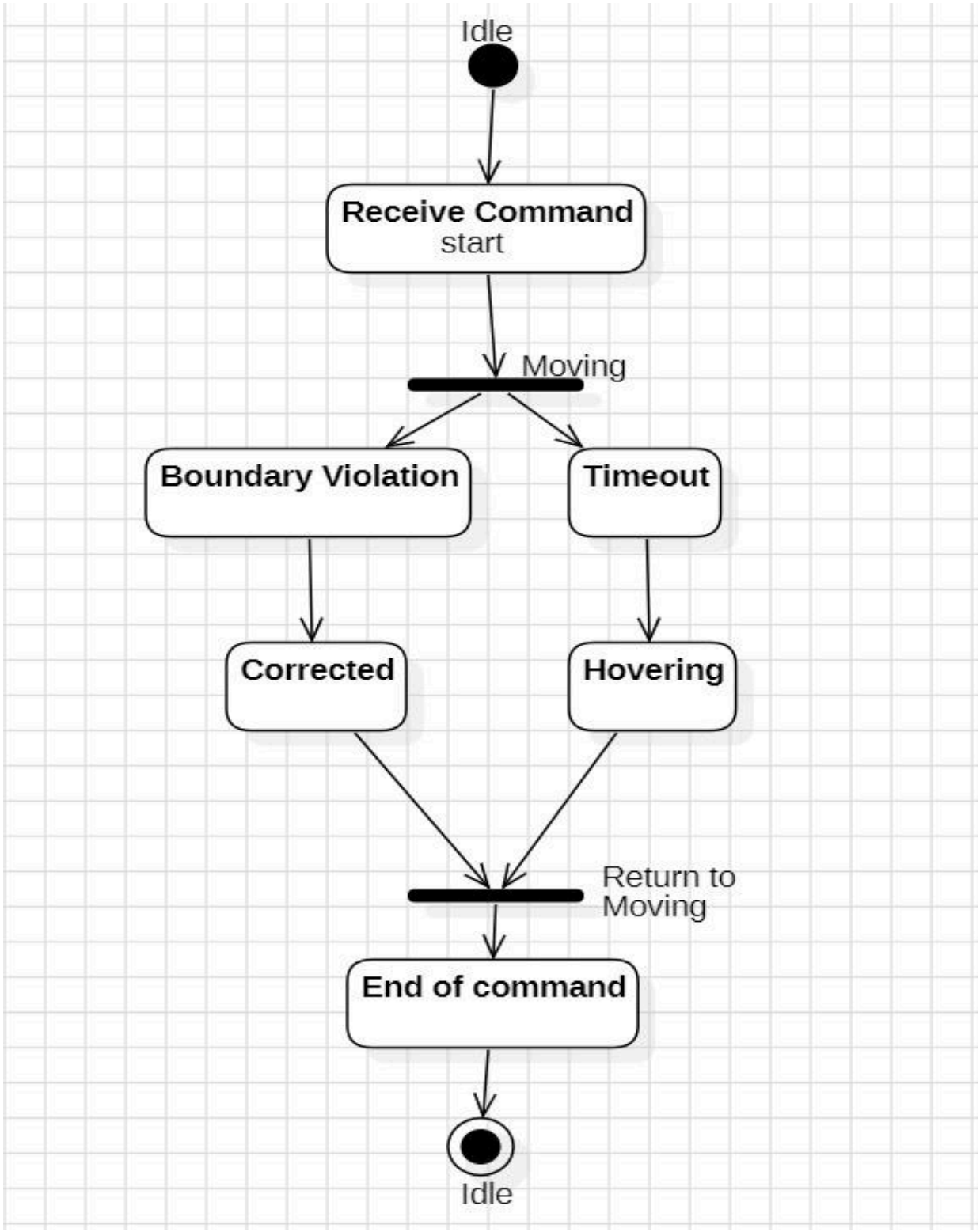


Difference in state space between monolithic and incremental enforcement techniques



UML DIAGRAMS:

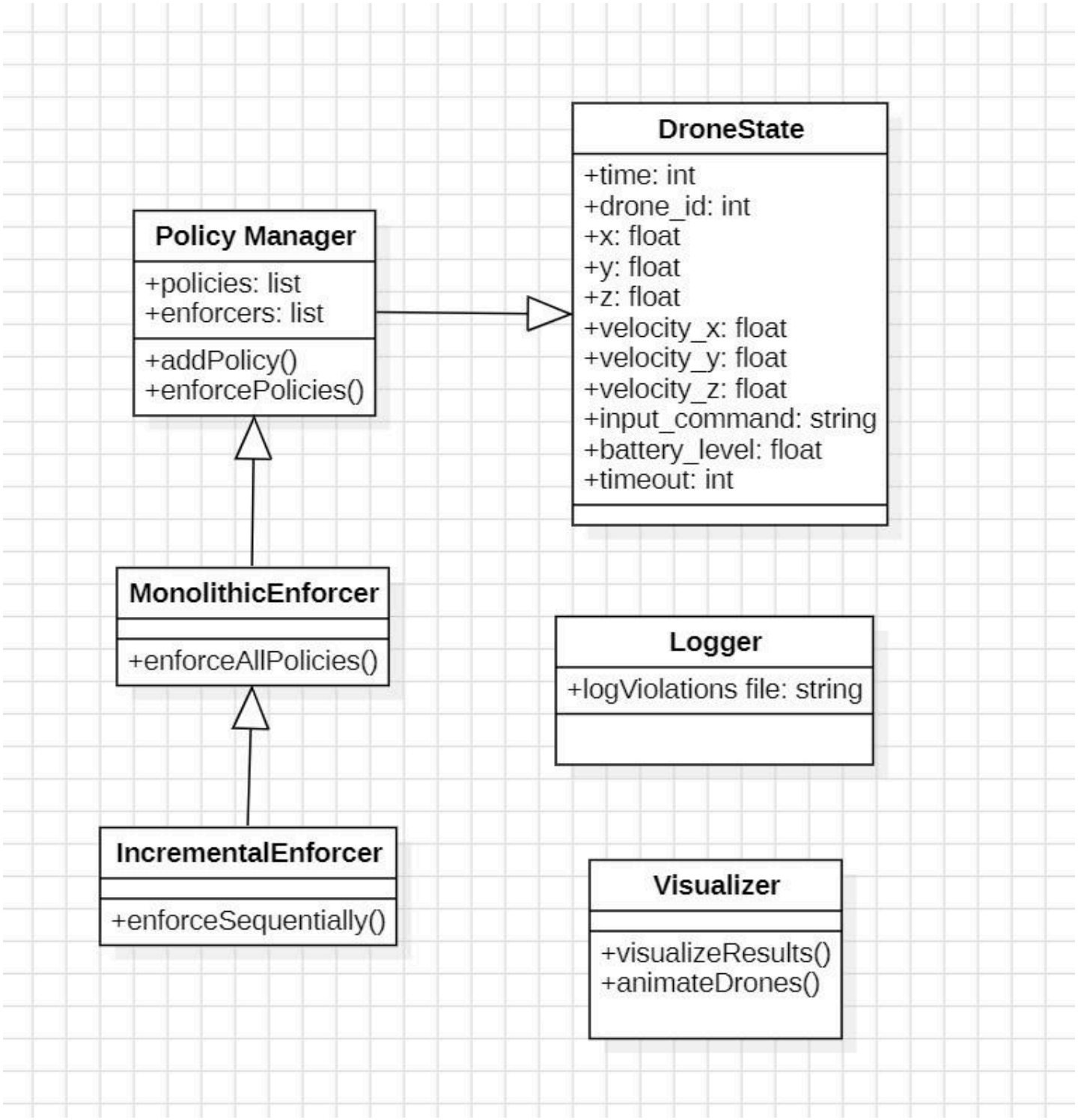
Activity Diagram



Key Components & Explanation:

1. **Idle State (Start & End)**
    - The drone starts in an Idle state, awaiting commands.
    - After completing its task, it returns to the idle state.
2. **Receive Command (Start)**
    - The drone receives a command, initiating movement.
    - It transitions to the moving state.
3. **Moving State (Main Operation)**
    - The drone is actively flying and executing its mission.
    - This is a key operational state where security enforcement is checked.
4. **Boundary Violation (Security Enforcement)**
    - If the drone crosses predefined boundaries, it triggers a boundary violation.
    - The Enforcement Framework steps in to correct the position.
    - Once corrected, the drone resumes operation.
5. **Timeout (Another Security Measure)**
    - If a drone is unresponsive or fails to receive further commands, a timeout occurs.
    - The drone enters a hovering state to wait for new instructions.
6. **Return to Moving**
    - After a correction (boundary enforcement) or timeout (hovering), the drone returns to moving.
    - This ensures smooth integration of security policies.
7. **End of Command (Final State)**
    - The drone finishes its assigned task and transitions back to the idle state.

Class Diagram



Key Classes & Explanation:

- **DroneState (Core Data Structure)**
  - Stores real-time state information of a drone.
  - Attributes include:
    - Position (x, y, z) – 3D coordinates.
    - Velocity (velocity_x, velocity_y, velocity_z) – Speed in different directions.
    - Command Input (input_command) – The command the drone is currently following.
    - Battery Level (battery_level) – Monitors energy consumption.
    - Timeout (timeout) – Tracks if the drone becomes unresponsive.
  - This class is central to the security framework, as policies are enforced based on its attributes.
- **Policy Manager (Security Policy Controller)**
  - Manages the list of policies and enforcers.
  - Responsibilities:
    - addPolicy() – Adds new security policies.
    - enforcePolicies() – Applies policies to the drone's state.
  - Acts as an interface between enforcers (Monolithic & Incremental) and the drone's real-time data.
- **MonolithicEnforcer (Strict Security Implementation)**
  - Implements enforceAllPolicies() which applies all security policies at once.
  - Can cause performance overhead due to its rigid approach.
  - Suitable for scenarios where strict and immediate enforcement is needed.
- **IncrementalEnforcer (Flexible Security Implementation)**
  - Implements enforceSequentially() which applies policies in a step-by-step manner.
  - More adaptive and efficient, activating only necessary policies.
  - This approach is better for real-time systems like drones.
- **Logger (Violation Tracking)**
  - Stores log files that record security violations.
  - Attribute:
    - logViolations file: string – Specifies where violations are logged.
  - Helps in analyzing policy effectiveness and debugging security issues.
- **Visualizer (Graphical Representation)**
  - Implements:
    - visualizeResults() – Plots security enforcement efficiency.
    - animateDrones() – Simulates drone movements and security responses.
  - Useful for presenting the effectiveness of Monolithic vs. Incremental enforcement in a simulation.

Conclusion

Conclusion & Future Scope

The compositional runtime enforcement framework developed in this project addresses key security challenges in Cyber-Physical Systems (CPS) by providing a scalable, modular, and adaptive security solution. Unlike traditional monolithic enforcement approaches, which suffer from state space explosion and lack flexibility, our framework employs incremental enforcement and serial composition of policies, ensuring real-time compliance without excessive computational overhead.

By implementing modular enforcers and dynamic policy updates, our approach allows incremental policy additions without requiring system-wide re-certification, making it highly adaptable to evolving security threats. The framework ensures that policy violations are handled efficiently, avoiding unnecessary enforcement while still maintaining robust security guarantees.

This project has provided valuable insights into how security enforcement can be integrated into dynamic environments, making CPS more resilient to cyber threats. Our implementation in UAV swarms demonstrates how compositional enforcement can be applied to autonomous systems, industrial automation, and real-time decision-making applications.

Future Scope

While the current framework effectively detects and mitigates security violations in UAV swarms, future work can enhance its capabilities in several ways:

1. **Machine Learning for Adaptive Enforcement** – Implementing AI-driven anomaly detection to dynamically adjust security policies based on evolving attack patterns.
2. **Integration with Blockchain** – Using decentralized security models to ensure tamper-proof enforcement across distributed CPS networks.
3. **Enhanced 3D Visualization** – Improving the real-time UAV swarm simulation with better UI/UX, AI-based path prediction, and dynamic attack visualization.
4. **Scalability Testing on Large CPS Networks** – Extending the framework to handle hundreds or thousands of autonomous agents with real-world sensor data.
5. **Multi-Agent Collaboration & Attack Response** – Implementing coordinated security responses where UAVs can share security alerts and react collectively to cyber threats.
6. **Hardware-in-the-Loop (HIL) Testing** – Deploying the framework on physical UAVs to evaluate real-world performance and robustness.

By expanding in these areas, the framework can be generalized beyond UAV swarms, paving the way for more secure and intelligent Cyber-Physical Systems in high-security applications.

References

[1]**Incremental Security Enforcement for Cyber-Physical Systems** by Abhinandan Panda, Alex Baird, Srinivas Pinisetty, and Partha Roop
IEEE Access 2023

[2]**Scalable Security Enforcement for Cyber-Physical Systems** by Alex Baird, Abhinandan Panda, Hammond Pearce, Srinivas Pinisetty, and Partha Roop
IEEE Access 2024

[3]**Monitoring and Defense of Industrial Cyber-Physical Systems Under Typical Attacks** by Yuchen Jiang, Shimeng Wu, Renjie Ma, Ming Liu, Hao Luo, and Okyay Kaynak.
IEEE TRANSACTIONS ON INDUSTRIAL CYBER-PHYSICAL SYSTEMS, VOL. 1, 2023