drones

MDPI

Article

A Study of the Data Security Attack and Defense Pattern in a Centralized UAV–Cloud Architecture

Gregorius Airlangga [1,2,*] and Alan Liu [2]

[1] Information System Department, Atma Jaya Catholic University of Indonesia, Jakarta 12930, Indonesia
[2] Electrical Engineering Department, National Chung Cheng University, Chiayi 621301, Taiwan; aliu@ee.ccu.edu.tw
* Correspondence: gregorius.airlangga@atmajaya.ac.id

Abstract: An unmanned aerial vehicle (UAV) is an autonomous flying robot that has attracted the interest of several communities because of its capacity to increase the safety and productivity of labor. In terms of software engineering, UAV system development is extremely difficult because the focus is not only on functional requirement fulfillment, but also on nonfunctional requirements such as security and safety, which play a crucial role in mission success. Consequently, architecture robustness is very important, and one of the most common architectures developed is based on a centralized pattern in which all UAVs are controlled from a central location. Even though this is a very important problem, many developers must expend a great deal of effort to adapt and improve security. This is because there are few practical perspectives of security development in the context of UAV system development; therefore, the study of attack and defense patterns in centralized architecture is required to fill this knowledge gap. This paper concentrates on enhancing the security aspect of UAV system development by examining attack and defense patterns in centralized architectures. We contribute to the field by identifying 26 attack variations, presenting corresponding countermeasures from a software analyst's standpoint, and supplying a *node.js* code template for developers to strengthen their systems' security. Our comprehensive analysis evaluates the proposed defense strategies in terms of time and space complexity, ensuring their effectiveness. By providing a focused and in-depth perspective on security patterns, our research offers crucial guidance for communities and developers working on UAV-based systems, facilitating the development of more secure and robust solutions.

Keywords: security; drone; UAV; Multi-UAV; Pattern Language; defense pattern; UML; class diagram; software architecture; centralized; quality attributes

Citation: Airlangga, G.; Liu, A. A Study of the Data Security Attack and Defense Pattern in a Centralized UAV–Cloud Architecture. *Drones* **2023**, *7*, 289. https://doi.org/10.3390/drones7050289

Academic Editors: Khair Ayman Al-Shamaileh and Naima Kaabouch

Received: 15 March 2023
Revised: 16 April 2023
Accepted: 18 April 2023
Published: 25 April 2023

1. Introduction

Unmanned aerial vehicles (UAVs), more generally known as drones, have gained popularity in recent years due to their versatility in a variety of applications, including surveillance, delivery, and search and rescue missions [1]. Currently, as the use of UAVs expands, safety and security issues are also growing [2]. UAVs pose risks such as crashes, interference with other aircraft, and cyberattacks that can lead to data theft or illegal control of the UAV [3]. The complexity of unmanned aerial vehicles necessitates a high level of software and hardware integration to enable their autonomous operation, flight dynamics, payload integration, real-time communication, and navigation. The dynamic contexts in which these systems operate can also contribute to their complexity [4]. In order to ensure the security, dependability, and efficacy of UAVs for their intended usage in cloud settings, it is essential to build and deploy a solid security strategy [5].

In the UAV–cloud deployment, decentralized and centralized architecture can be used to meet the aims of unmanned aerial vehicles [6]. A decentralized architecture may be more flexible; however, coordinating the actions of several UAVs might lead to accidents and conflicts [7]. In contrast, As potrayed in Figure 1, centralized design offers a high level of

system control and coordination, but it is prone to single points of failure [8]. It is essential to remember that centralized design is still susceptible to security concerns that can at any time interrupt the entire system. Many researchers have proposed several approaches in order to mitigate this problem; however, from a practical point of view, there is little practical work concerning the analysis of attack and defense patterns of UAV systems. It is also compounded by the fact that many UAV developers do not have special knowledge related to the security problems in UAVs. Therefore, a practical analysis related to attack and defense security patterns for UAV–cloud architecture is urgently needed in order to respond to the data security risks posed by UAVs [9]. These attack patterns are utilized by attackers to exploit system vulnerabilities, whereas the defense patterns are employed to prevent or minimize the impacts of these attacks [10].

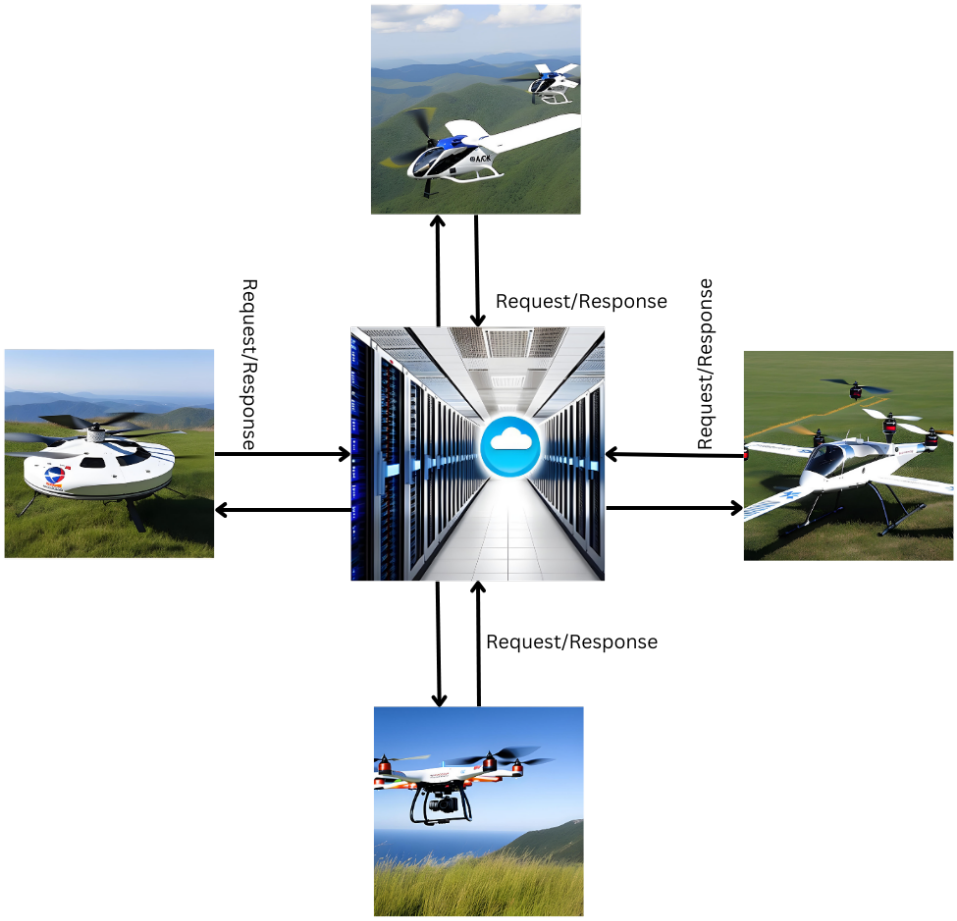

Figure 1. Centralized architecture of a multi-UAV system.

Our research will undertake an analysis of data security concerns in UAV–cloud architecture and provide protection methods to counteract these threats. We will identify potential vulnerabilities in the system that could be exploited by attackers and recommend mitigation strategies for these flaws. Our goal is to develop a defense architecture that is easily reusable, cohesive, loosely coupled, and, most importantly, has a high data security rate. Complexity analysis will be used to evaluate the efficacy of the suggested protection mechanisms against the detected attack patterns since the performance of the solution is a very important consideration for best security practice; higher performance can lead to a higher security rate and complicate the attacker's effort to destroy the entire system. In addition, we also provide a holistic pattern related to attack and defense in the context of drones, because developing a complete security architecture for UAVs operating in cloud settings is essential for improving the safety, dependability, and effectiveness of UAVs for their intended applications. The sections provided for this study include the

introduction, related work, defensive and attack patterns, analysis of complexity, and conclusion. Through this research, we intend to develop an all-encompassing strategy for addressing the data security vulnerabilities posed by UAVs in cloud environments.

2. Related Work

Unmanned aerial vehicles (UAVs) have been an emerging field of research and development due to their versatile applications such as surveillance, delivery, and search and rescue missions. With the increasing usage of UAVs, there is a growing concern for their safety and security. Potential risks include collisions, interference with other aircraft, and cyberattacks that can result in data theft or unauthorized control of the UAV. These concerns have led to a surge in research on the security and safety of UAVs. One of the earliest works in the field of UAV security was published in [11,12]. The paper presents an overview of the challenges and issues in securing UAVs, including the need for secure communication, data storage, and mission-critical decision making. The authors proposed models and recommendations for UAV security that include secure communication protocols, secure UAV control, and secure data storage.

In [13], which identified the security and privacy challenges of UAV communication in flying ad hoc networks, the authors presented a comprehensive survey of the existing security mechanisms, including authentication, confidentiality, integrity, and availability of data, and identified the limitations of these mechanisms. Similarly, the authors of [14] presented a survey of existing research on UAV security, including various types of attacks, vulnerabilities, and countermeasures. The authors highlighted the importance of securing UAVs against cyberattacks, such as jamming, eavesdropping, and spoofing.

In recent years, researchers have focused on the security of UAVs in cloud environments. In [15,16], they discussed the security challenges and threats for UAVs in cloud environments. The authors presented a comprehensive review of the state-of-the-art solutions for addressing these challenges and identified future research directions. Different from that, one of the unique aspects of our research is the emphasis on attack and defense patterns in UAV–cloud architecture. While existing research has identified the challenges and vulnerabilities of UAV security, our proposed attack and defense patterns provide a more practical and applicable approach to addressing these issues. These patterns are designed to help prevent and mitigate the effects of attacks, making them an essential component of a comprehensive security architecture for UAVs operating in cloud environments.

Furthermore, our research places a significant focus on the evaluation of the effectiveness of these defense mechanisms against the identified attack patterns. By using complexity analysis, we aim to provide a thorough assessment of the proposed defense mechanisms' capabilities and limitations. This will help validate the effectiveness of the proposed architecture in addressing the data security threats associated with UAVs. In conclusion, the research on the security and safety of UAVs has gained significant attention in recent years. The proposed research aims to provide a comprehensive approach to addressing the data security threats associated with UAVs in cloud environments. The proposed attack and defense patterns and the evaluation of their effectiveness will enable UAV developers and system analysts to develop a more robust security architecture that can help ensure the safety, dependability, and efficacy of UAVs for their intended uses.

3. Attack and Defense Pattern

Our explanation is based on a software engineering point of view, which is related to the UML diagram and software architecture knowledge base. We use a Javascript-based online tool to draw the diagrams, and in order to explain the pattern, we modify some guides from the best practice of the PLOP community [17–19]. The modification is conducted due to the fact that our discussions are more specific to the threat category compared to global. The pattern explanation consists of several sections such as definition, sequence diagram, and defense mechanism using the *node.js* programming language. The use of the *node.js* programming language is to help communities implement defense mechanisms in

cloud architecture; in addition, *node.js* is one of the most popular programming languages that runs in the cloud, and with some tweaking effort, the solution can also work in other *Javascript* environments.

3.1. Black Hole Attack

3.1.1. Definition

Consider the directed graph *G = (V, E)*, where *V* is the set of nodes and *E* is the set of directed edges connecting nodes. Each node *v* has its own IP address, shown by *IP(v)*. Let *S(V)* represent the network of UAV nodes and *C(V)* represent the cloud server. A black hole attack is a sort of denial of service attack in which a malicious node presents itself as having the quickest path to the cloud server *C(V)*, forcing other nodes to route their data via *v*. The malicious node *v* subsequently discards or disregards the incoming packets, interfering with the connection between the UAVs and the cloud server [20].

To protect against a black hole attack, a new route that bypasses the malicious node must be constructed. This may be accomplished by utilizing path planning such as *Dijkstra's*, *breadth first search*, or the *A** method to calculate the shortest path between the source and destination nodes. Let $P = (p_1, p_2, \ldots, p_m)$ represent the shortest path from a UAV node *s* to the cloud server *c*. If the malicious node *v* is present in *P*, a new route *P'* can be constructed by concatenating the nodes before *v* with those after *v*. To prevent the attacker node from discarding or ignoring incoming packets, it is necessary to restrict its traffic. This may be accomplished by adding a rule to the firewall that drops all traffic originating from IP address *IP (v)*.

M is the set of malicious nodes in the network, and *g* is a function that converts the size of *M* to a severity grade depending on the number of UAVs impacted, the type of data lost, and the mission criticality. To reduce the impact of black hole attacks on the UAV network, it is crucial to implement the necessary security measures to prevent and mitigate them. This may include frequent security audits, firewall setups, and intrusion detection systems to identify and respond to threats in real time.

3.1.2. Sequence Diagram

The sequence diagram in Figure 2 depicts a communication between UAV nodes (participant *S*) and the cloud server (participant *C*), with the possibility of malicious node interference (participant *V*). The *S→ C: Request* message indicates that the UAV nodes submit a request to the cloud server at the beginning of the sequence. As stated by the *C→ S: Response* message, the cloud server responds with a response message.

If a malicious node is available in the network, however, it may launch a black hole attack by pretending to have the quickest path to the cloud server. When UAV nodes submit a request to the malicious node (shown by *S -> V: Request*), the malicious node will not answer (marked by *V -> S: No Response*). When the UAV nodes submit a request to the cloud server (marked by *S -> C: Request*), the cloud server will answer with *C -> S: No Response*. The design contains an *alt* block to depict the alternate sequence flow dependent on the presence or absence of a malicious node. The *Malicious Node Attack* block is run if a malicious node is detected.

3.1.3. Defense Class Diagram

As shown in Figure 3, the class diagram depicts a network defense mechanism against black hole assaults. The system comprises several classes that collaborate to achieve this objective. The *PathPlanning* class, which offers methods for determining the shortest path between two nodes in a graph, is the core of the system. It has one private field, *graph*, which is an instance of the *Graph* class. The *Graph* class encapsulates a graph consisting of *nodes* and *edges*. The *nodes* and *edges* private fields are arrays of *Node* and *Edge* objects, respectively. The class includes methods for adding *nodes* and *edges* to the graph and retrieving the graph's *nodes* and *edges*. A *Node* object represents a graph node with a unique

identification, or id. The *Edge* class represents an edge between two nodes in a graph using the *from*, *to*, and *weight* private variables.

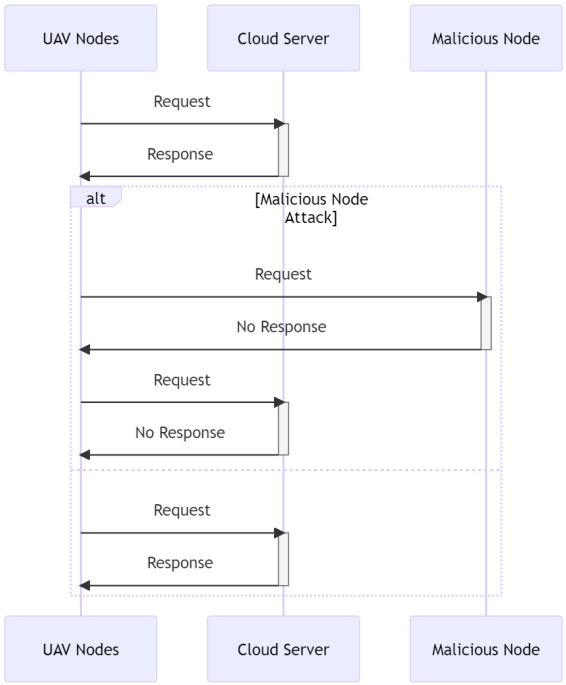

Figure 2. Black hole sequence diagram.

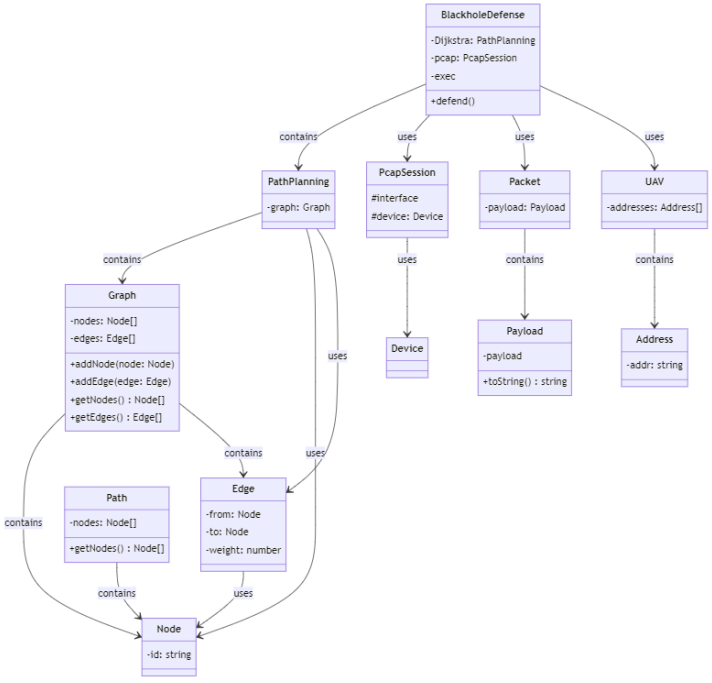

Figure 3. Black hole defense class diagram.

The *PcapSession* class defines a network session that enables the system to inject and receive packets from the network. It contains the protected fields interface and device, where device is an instance of the *Device* class. The *UAV* class represents an unmanned aerial vehicle (UAV) as a network client, and its private field addresses are an array of *Address* objects. The *Address* class represents a network address, with the *addr* field being private. The *BlackholeDefense* class contains techniques for protecting against black hole

assaults and is the primary class of the system. Private fields include *Dijkstra* as example of path planning method, and developers can consider other methods such as *A**, *artificial potential field*, *genetic algorithm*, *etc*. In addition, the methods of *pcap* and *exec* are added; *exec* is a reference to a system command execution library, *pcap* is a reference to a network packet capturing library, and *Dijkstra* is an instance of the *PathPlanning* class. The *Packet* class represents a network packet, and its private payload field is an instance of the *Payload* class.

The *Payload* class represents a network packet's payload and contains a private field named payload. It offers the *toString* function for stringifying the payload. *Path* is a class that depicts a path across a network, and its nodes field is an array of *Node* objects. It contains one public method, *getNodes*, for retrieving the path's nodes. Lastly, the graphic illustrates the class connections using arrows. *PathPlanning*, for instance, is related to *Graph*, *Node*, and *Edge*. *Graph* has connections with *Node*, *Edge*, etc.

Below is the potential scenario for implementing a defense mechanism against black hole attacks:

1. **Initialization**: Create instances of the *Graph*, *PathPlanning*, and *BlackholeDefense* classes. Populate the *Graph* object with the network topology, including *nodes* and *edges*. Then, it may also need to create instances of the *PcapSession* class for monitoring network traffic and instances of the UAV class for representing network clients.
2. **Monitor network traffic**: The *PcapSession* class is responsible for capturing packets from the network. By using the *pcap* library, the system captures network traffic and passes it to the *BlackholeDefense* class for analysis to detect potential black hole attacks.
3. **Analyze packets**: The *BlackholeDefense* class processes captured packets and uses the *Packet* and *Payload* classes to extract relevant information. The *Payload* class's *toString* method can be used to convert the payload into a string for further analysis.
4. **Detect black hole attacks**: The *BlackholeDefense* class leverages the *PathPlanning* class (with the *Dijkstra* algorithm or other methods) to find the shortest paths between nodes in the network. By comparing the actual network traffic and path information from the *PathPlanning* class, the defense mechanism can detect inconsistencies or suspicious activities that may indicate a black hole attack.
5. **Alert and respond**: If a black hole attack is detected, the defense mechanism can alert administrators or other network clients. It can also initiate countermeasures, such as updating routing tables to bypass the compromised node, isolating the malicious node, or informing other nodes about the attack.
6. **Adapt and learn**: In more advanced defense mechanisms, machine learning techniques can be applied to improve the detection and response capabilities. By learning from past attack patterns and continuously updating the detection algorithms, the defense system can become more effective in countering black hole attacks.

3.2. Collision Network

3.2.1. Definition

Let $G = (V, E)$ be an undirected graph that represents a collision network with n UAVs, where V is the set of n UAVs and E is the set of links between them. Each UAV is capable of transmitting and receiving data packets over the network, and a collision occurs when two or more devices transmit data packets simultaneously, leading to data loss or corruption. To prevent or resolve collisions, the devices can employ carrier sensing and collision detection approaches.

Carrier sensing requires each UAV to sense the network before transmitting data, which can be achieved by monitoring the network for existing traffic. The carrier sensing time for *UAVi*, denoted as c_i, represents the time it takes for *UAVi* to detect the availability of the network. If the network is busy, device i defers its transmission until the network is clear. The transmission time for *UAVi*, denoted as t_i, is calculated as $t_i = \max(c_i, t_j + d_{ji})$, where $j \in V$, $j \neq i$, d_{ji} is the propagation delay between *UAVi* and *UAVj*, t_j is the transmission time of *UAVj*, and the maximum operation ensures that *UAVi* defers its transmission until the network is available [21].

Collision detection is employed when multiple UAVs attempt to transmit data simultaneously, which may lead to a collision. In this case, each UAV monitors the network for potential collisions, and if a collision is detected, the device waits for a random backoff time before retransmitting its data packet. The backoff time for *UAVi*, denoted as b_i, is a random variable chosen from a uniform distribution with range [0, CW_i], where CW_i is the contention window size of *UAVi*. The retransmission time for *UAVi*, denoted as t'_i, is calculated as $t'_i = t_j + d_{ji} + b_i$, where $j \in V, j \neq i$, and the retransmission is scheduled to occur after the propagation delay and backoff time have elapsed.

3.2.2. Sequence Diagram

As depicted in Figure 4, two UAVs transmit data packets to the cloud in this sequence diagram. After requesting a route, the cloud responds with the requested path, and the UAVs begin transmitting data packets. The cloud acknowledges each data packet (*ACK)*. Nevertheless, one of the data packets is lost owing to a collision in the network and does not reach its destination. When it does not receive an *ACK* packet from the cloud, the defense class diagram of the UAV understands this and retransmits the lost packet. The retransmitted packet successfully reaches the cloud, and the *ACK* signal is received, indicating that the packet was sent successfully.

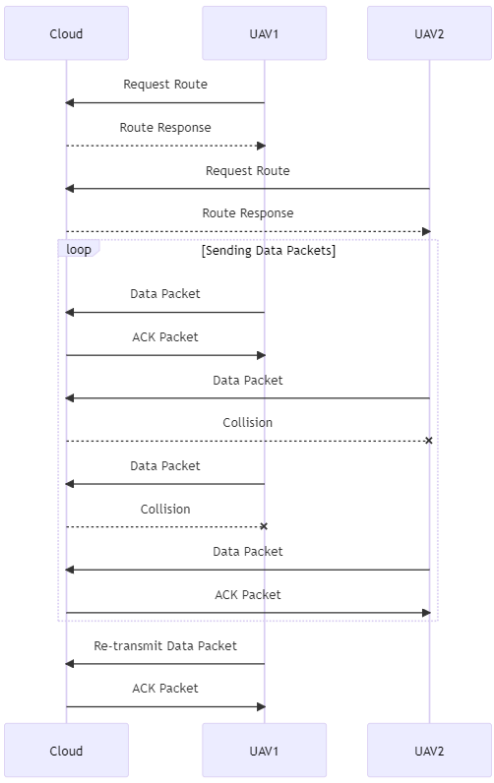

Figure 4. Collision network problem.

3.2.3. Defense Class Diagram

As depicted in Figure 5, the first class is called *EventEmitter.* This class represents an event emitter, which is an object that can emit named events and execute associated listeners. It has three methods: *addListener()* to register a listener for an event, *removeListener()* to remove a previously registered listener, and *emit()* to trigger an event with a set of arguments. The second class is *NetworkInterface*, which extends *EventEmitter*. It has two private properties: *name*, to store the name of the network interface, and *sessions*, which is an object to store the active sessions on this interface. Furthermore, it has two methods: *addSession(session)*, to add a new session to the interface, and *sendPacket(packet, src, dst)*, to

send a packet to a specified source and destination. When a packet is received on any of its sessions, it emits the packet event with the session and the decoded packet as arguments.

The third class is *CollisionDetector*, which has private properties to manage its internal state. This class also has an *interface* property to store the network interface that is associated with it, an *address* property to store its own address on the network, a *packet* property to store a buffer that will be sent to the network, an *intervalId* property to store the *ID* of the timer that is used to send packets at regular intervals, a *busy* property to indicate whether the network is busy, and a *backoff* property to store a value that is used in an exponential backoff algorithm. Then this class also has two methods: *start()* to start the timer and send packets, and *stop()* to stop the timer. Finally, the diagram shows the inheritance relationships between the classes using the –|> notation. Lastly, the *NetworkInterface* class inherits from *EventEmitter*, and *CollisionDetector* has a relationship with *NetworkInterface*, which means it uses an instance of *NetworkInterface* as a private property.

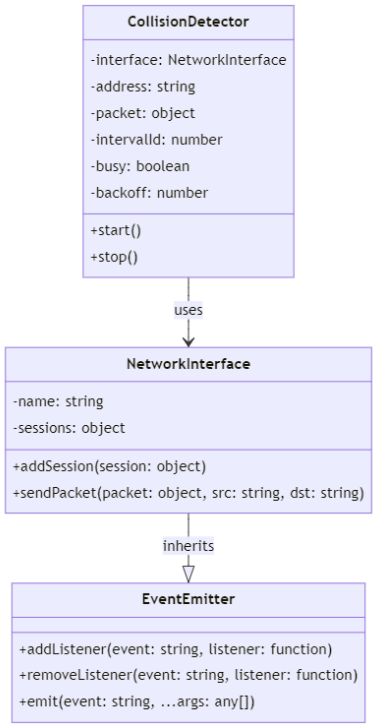

Figure 5. Collision defense class diagram.

Below is a detailed explanation about the mechanism process of using a collision network defense class diagram, the classes and their roles are explained in more detail to better understand how the system defends against collision network problems in a UAV–cloud communication setup.

1. **Initialization**: In this phase, the system administrator initializes and configures the network environment. Then, it creates an instance of the *NetworkInterface* class, specifying the network interface to be monitored. Subsequently, it creates a *CollisionDetector* instance, associating it with the *NetworkInterface* instance. The system administrator can also set additional parameters or fine-tune the detection mechanism as needed.
2. **Monitoring and event management**: The *CollisionDetector* continuously monitors the network interface by leveraging the *EventEmitter* capabilities inherited from the *NetworkInterface* class. The system listens for *packet* events emitted by the *NetworkInterface*, which occur when the interface receives packets from its sessions. As packets are received and decoded, the system triggers the *packet* event with the session and decoded packet information as arguments.

3. **Traffic analysis and collision detection**: Upon receiving packets, the *CollisionDetector* analyzes the packet information to detect potential collisions. It examines various packet attributes, such as source, destination, and traffic patterns, to determine if a collision has occurred or is likely to occur. If a potential collision is identified, the system may trigger additional events or raise alerts to notify administrators and initiate mitigation procedures.
4. **Adaptive collision detection algorithm**: To improve collision detection accuracy, the *CollisionDetector* may employ adaptive algorithms that adjust based on network conditions and traffic patterns. These algorithms may incorporate historical data, machine learning techniques, or other advanced methods to optimize collision detection and minimize false positives.
5. **Exponential backoff algorithm and traffic management**: In response to detected collisions or busy network conditions, the *CollisionDetector* uses an exponential *backoff* algorithm to manage packet retransmissions. The algorithm introduces a random delay before retransmitting packets, with the delay increasing exponentially after each failed attempt. This approach helps alleviate network congestion and reduces the likelihood of further collisions.
6. **Real-time collision mitigation**: Upon detecting a collision, the *CollisionDetector* can take real-time mitigation actions. These may include blocking malicious users, adjusting network parameters, or implementing traffic-shaping techniques to prevent future collisions. The system may also notify network administrators, allowing them to take further action if necessary.
7. **Data logging and reporting**: Throughout the monitoring process, the *CollisionDetector* can log relevant data, such as packet information, detected collisions, and mitigation actions. These data can be used for auditing, reporting, and further analysis to improve the defense mechanism and maintain network stability.
8. **Continuous monitoring and optimization**: The *CollisionDetector* persistently monitors the network interface, adapting its algorithms and adjusting its *backoff* mechanism as needed. This continuous process ensures that the system remains resilient against collision attacks while optimizing network performance.

The primary function of the *CollisionDetector* class is to monitor network traffic using the associated *NetworkInterface* instance. It listens for the *packet* event emitted by the *NetworkInterface* to track received packets. The *CollisionDetector* class is responsible for managing potential network collisions by adjusting packet transmission rates based on network conditions. This is performed using the *busy* property and the exponential *backoff* algorithm. By slowing down transmission rates when the network is busy, the system reduces the likelihood of collisions.

3.3. Data Tampering

3.3.1. Definition

Let D be a dataset consisting of n records stored in a cloud-based UAV system. Each record in D contains sensitive information, such as photographs or location data, which may be manipulated by an adversary. The goal is to design a secure system that minimizes the risk of data manipulation in the UAV–cloud system. To achieve this, the several measures can be taken, such as authentication, access control, data backups, and anomaly detection [22]. Authentication is a mechanism to implement a robust filtering system that ensures that only authorized users can access the data. Let A be the set of authorized users who have the necessary credentials to access the system. The authentication system can be represented by the function $f_{auth} : U \times P \rightarrow \{0, 1\}$, where U is the set of all possible usernames and P is the set of all possible passwords. The function returns 1 if the given username and password combination is valid and 0 otherwise.

Access control has a goal to restrict access to the data to those who require it for their responsibilities. Let R be the set of roles in the system management and let U_r be the set of users assigned to each role r in R. The access control system can be represented by the

function $f_{access} : U \times R \rightarrow \{0,1\}$, which returns 1 if the given user u has access to the data for their assigned role r and 0 otherwise. In addition, to mitigate the worst-case situation when data are already affected, the data backup mechanism must be conducted.

Data backup is a process to create frequent backups of the data to safeguard against data tampering. Let B be the set of backup copies of the data. The backup system can be represented by the function $f_{backup} : D \rightarrow B$, which creates a backup copy of the dataset D. In order to prevent the data tampering problem, anomaly detection can be conducted. Anomaly detection is a process to monitor the system for unusual activity to detect and prevent data tampering. Let M be the set of all system monitoring measures, such as intrusion detection systems, log analysis tools, and machine learning algorithms. The anomaly detection system can be represented by the function $f_{anomaly} : M \times D \rightarrow \{0,1\}$, which returns 1 if the system monitoring measures detect any unusual activity in the dataset D and 0 otherwise.

Therefore, the solution of data tampering can then be formulated as follows: Find a set of functions $\{f_{auth}, f_{access}, f_{backup}, f_{anomaly}\}$ that maximizes the security of the UAV–cloud system while minimizing the risk of data manipulation. The solution should ensure that only authorized users can access the data, access to the data is restricted to those who require it, frequent backups are created to safeguard against data tampering, and unusual activity is detected and prevented. The solution should also be cost-effective and scalable for large datasets.

3.3.2. Sequence Diagram

As depicted in Figure 6, in this sequence diagram, two unmanned aerial vehicles transmit data to the cloud for storage. Cloud computing accepts and stores data. Yet, the cloud also acknowledges a malicious data transmission sent by an attacker. The attacker then tampers with the cloud-stored data, perhaps altering them for their own objectives. When *UAV1* requests information from the cloud, it receives altered information, resulting in an incorrect response. This emphasizes the severity of data tampering in a UAV–cloud system and the significance of avoiding it.

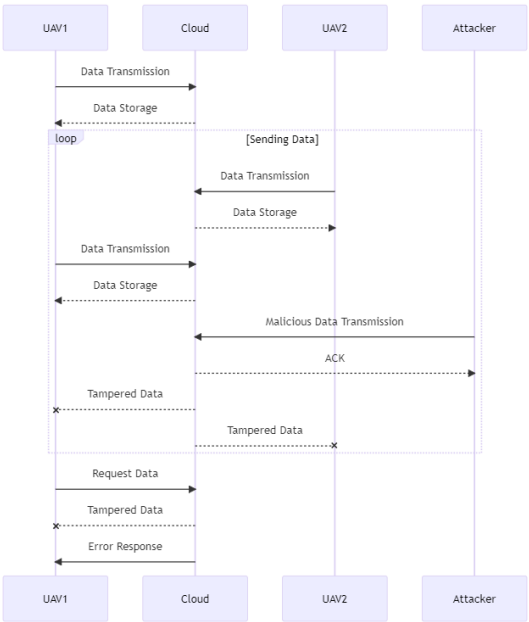

Figure 6. Data tampering.

3.3.3. Defense Class Diagram

As shown in Figure 7, the defense pattern class diagram of data tampering attack has the *crypto* module that can be used for defending purposes since it provides several

functions to encrypt data, such as calculating the *SHA-256* hash of a message. The *@textile/hub* module provides a client to access the *Textile Hub API*, as well as the *Client* and *PrivateKey* classes used to authenticate the client. The *DataTamperingDefense* class contains the *defendAgainstDataTampering()*, *addToLedger()*, and *detectTampering()* methods, which are used to defend against data tampering attacks. The *DataTamperingDefense* class interacts with the *crypto* module and the *@textile/hub* module to calculate hashes and store data in the distributed ledger. The *defendAgainstDataTampering()* method calls the *detectTampering()* method to check if the provided data have been tampered with.

If tampering has occurred, the method calls the *addToLedger()* method to add the tampered data to the distributed ledger. The *detectTampering()* method uses the *calculateHash()* function to calculate the hash of the provided data and compares it to the hash included with the data. If the hashes do not match, the data have been tampered with. The *addToLedger()* method uses the *@textile/hub* module to obtain a thread and push the provided data to the distributed ledger. If an error occurs, the method logs the error to the console.

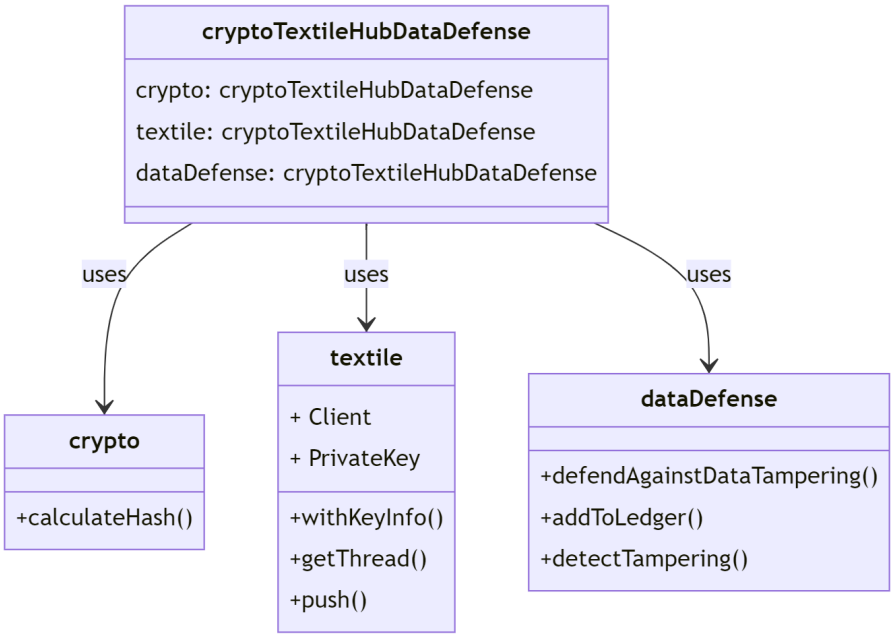

Figure 7. Defense data tampering class diagram.

Based on the class diagram in Figure 7 for defending against data tampering attacks, here is a potential scenario for implementing the defense mechanism:

1. **Initialization**: Instantiate and configure the *DataTamperingDefense* class, which will be responsible for managing the defense against data tampering attacks. Additionally, set up the necessary *cryptographic* module for hashing (e.g., SHA-256) and the *@textile/hub* module to interact with the *Textile Hub* API. Ensure that proper authentication and secure connections are established for the *@textile/hub* module.
2. **Data preparation and hashing**: Before transmitting or storing data, use the *cryptographic* module to calculate a secure hash of the data, which serves as their unique fingerprint. Depending on the sensitivity of the data, we can choose to implement additional security measures such as digital signatures, encryption, or message authentication codes (MACs) to further enhance data integrity and confidentiality.
3. **Secure data transmission and storage**: Transfer the data with their associated hash to the intended recipient or store them securely in a database or distributed ledger, using secure communication protocols such as TLS/SSL. Ensure proper access controls, encryption, and other security measures are in place to protect the data and their hash during transmission and storage.

4. **Data integrity verification**: When the data are retrieved or received, the *DataTamperingDefense* class's *defendAgainstDataTampering()* method should be invoked to verify the data's integrity. This method internally calls the *detectTampering()* function, which recalculates the hash of the received data using the same *cryptographic* hashing algorithm and compares it to the original hash attached to the data. If the hashes do not match, the data have been tampered with.
5. **Tampering detection and response**: If data tampering is detected, the *defendAgainstDataTampering()* method initiates the *addToLedger()* function to create an immutable record of the tampering attempt. This function uses the *@textile/hub* module to store the tampered data, their original hash, and any relevant metadata (e.g., timestamps, source/destination information) in a distributed ledger such as *Textile Hub*. The ledger should employ consensus algorithms and redundancy to ensure that the data stored cannot be altered or deleted by malicious actors.
6. **Alerting and incident response**: In case of a data tampering attempt, the defense mechanism can trigger alerts or notifications to inform administrators, relevant stakeholders, or other systems about the incident. This can initiate a coordinated response, including further investigation, containment, and recovery measures. Depending on the nature of the tampering and the affected data, responses may include revoking compromised keys or certificates, patching vulnerabilities, updating security policies, or taking legal action.
7. **Continuous monitoring, learning, and adaptation**: Regularly monitor the system for new instances of data tampering and adapt the defense mechanism as needed. This may involve updating hashing algorithms, adjusting detection thresholds, or incorporating additional security measures to further protect data integrity. Conduct regular security audits, threat modeling, and penetration testing to identify and address potential weaknesses in the system. Additionally, stay informed about the latest tampering techniques, attack patterns, and best practices in data security to continuously improve the defense mechanism.

3.4. Deauthentication

3.4.1. Definition

Let U and C denote the UAV and cloud-based system, respectively, which utilize a wireless communication channel for data exchange. Consider a deauthentication attack, D, initiated by an attacker, which sends n deauthentication frames to U and/or C with the aim of disconnecting U from C. The objective of the attacker is to disrupt the communication between U and C by successfully disconnecting U from C for a duration of t. The likelihood of a successful deauthentication attack is dependent on the signal strength of the attacker, the distance between the attacker and U/C, and the specific implementation of the wireless communication protocol. We denote the probability of a successful deauthentication attack by S, where S is a function of the signal strength, distance, and communication protocol used, given by S(*signal strength*, *distance*, *communication protocol*) = P(success). The attacker can optimize the probability of success by selecting values of signal strength and distance that maximize S and by exploiting potential vulnerabilities or weaknesses in U and C's implementation. The primary goal of the attacker is to disrupt the UAV–cloud system and deny authorized users access to the system, or to gain unauthorized access to the system [23].

3.4.2. Sequence Diagram

In the sequence diagram, as depicted in Figure 8, two UAVs are linked to the cloud and transfer data for storage to the cloud. The cloud recognizes the delivery of data by transmitting an acknowledgment packet (*ACK)* to each UAV, but an adversary sends a deauthentication packet to both UAVs, severing their connection to the cloud. Both *UAV1* and *UAV2* receive incorrect answers when requesting data from the cloud. This demonstrates the severity of a deauth attack in a UAV–cloud system and the significance of preventing it.

To avoid deauth attacks in a UAV–cloud system, the system must employ robust security features, including encryption, secure authentication systems, and access control. In addition, the system should watch for illegal deauth packets and take appropriate action to avoid their disruption. This may involve deploying intrusion detection systems or routinely monitoring system logs for indicators of an assault.

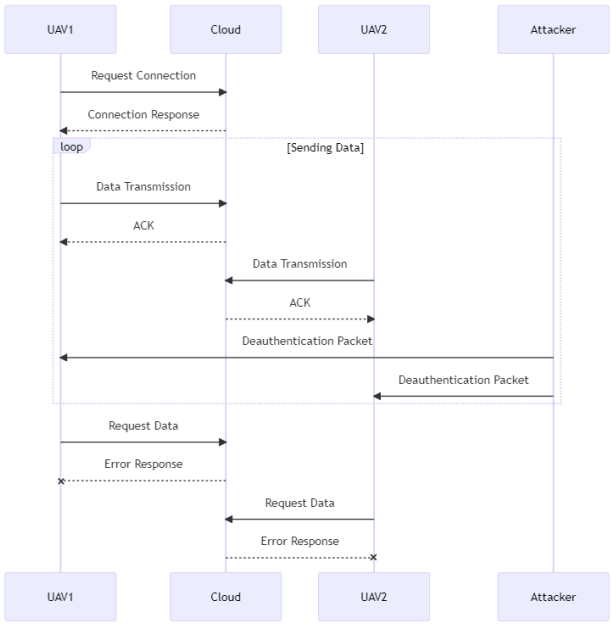

Figure 8. Deauth sequence diagram.

3.4.3. Defense Class Diagram

In Figure 9, *WifiInterface* is a class that represents a *Wi-Fi* interface. It has a *name* property, which is passed in as an argument to the constructor, and it can be used to block an *MAC* address and start a packet capture session. It also inherits from *EventEmitter*, which means that it can emit events. *DeauthDetector* is a class that represents a deauthentication detector. It takes a *WifiInterface* and a target *MAC* address as arguments in the constructor, and it listens for packets on the *Wi-Fi* interface. When it detects a deauthentication attack against the target, it emits a *deauth* event.

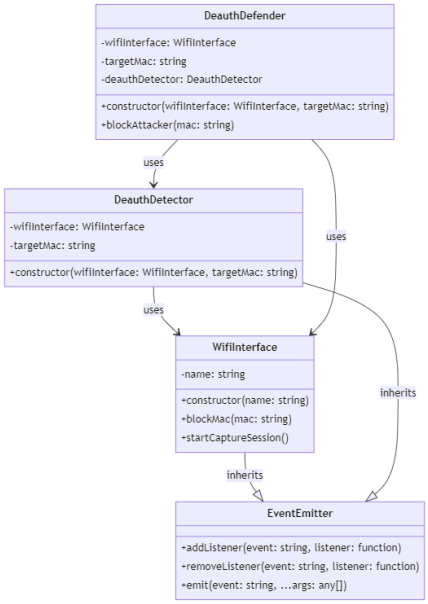

Figure 9. Deauth class diagram.

DeauthDefender is a class that represents a deauthentication defender. It takes a *WifiInterface* and a target *MAC* address as arguments in the constructor, and it creates a *DeauthDetector* to listen for deauthentication attacks. When a deauthentication attack is detected, it blocks the *MAC* address that is responsible for the attack. Finally, *EventEmitter* is a built-in *node.js* class that allows objects to emit events and register listeners for those events. Both *WifiInterface* and *DeauthDetector* inherit from *EventEmitter* so that they can emit and listen for events.

Based on the defense class diagram for deauth attacks, the following scenario illustrates how the defense mechanism operates:

1. **Initialization**: During the initialization phase, a *WifiInterface* instance is created to represent the *Wi-Fi* interface for the network being monitored. The name property is set to uniquely identify the interface, and the object is responsible for capturing packets, blocking MAC addresses, and managing other network activities. At this stage, the underlying network card may be set to promiscuous mode or monitor mode to allow packet capture of all traffic on the Wi-Fi network.
2. **DeauthDetector setup**: A *DeauthDetector* instance is created and initialized with the *WifiInterface* instance and a target MAC address as arguments. The *DeauthDetector* is responsible for monitoring packets on the Wi-Fi interface and detecting deauthentication attacks against the specified target device. To achieve this, the *DeauthDetector* analyzes packets captured by the *WifiInterface*, specifically looking for 802.11 management frames, such as deauthentication or disassociation frames, that indicate a potential attack.
3. **DeauthDefender activation**: A *DeauthDefender* instance is created and initialized with the *WifiInterface* instance and the target MAC address as arguments. The *DeauthDefender* leverages the *DeauthDetector* instance to monitor deauthentication attacks against the target device and takes action when an attack is detected. This class coordinates the overall defense mechanism and communicates with other system components, such as intrusion detection systems, firewalls, or network administrators, as needed.
4. **Deauthentication attack detection**: The *DeauthDetector* continuously analyzes packets captured by the *WifiInterface*, looking for patterns or anomalies indicative of a deauthentication attack, such as a high number of deauthentication frames sent to the target device within a short period. When the *DeauthDetector* identifies an attack pattern, it emits a *deauth* event containing information about the attack, such as the attacker's MAC address, the targeted device, and the timestamp of the attack.
5. **Deauth event handling**: The *DeauthDefender* listens for the *deauth* event emitted by the *DeauthDetector*. Upon receiving the *deauth* event, the *DeauthDefender* processes the event data and determines the appropriate response based on the attack severity, the attacker's identity, and the system's security policies.
6. **Attack mitigation**: After analyzing the *deauth* event, the *DeauthDefender* takes appropriate action to mitigate the attack. This can include blocking the attacker's MAC address using the *WifiInterface* instance, alerting network administrators, updating firewall rules, or even implementing dynamic channel switching or frequency hopping to evade the attack. In some cases, the *DeauthDefender* may also collect evidence of the attack for forensic analysis and reporting purposes.
7. **Ongoing monitoring and defense**: The *DeauthDefender* continuously monitors for deauthentication attacks, allowing it to detect and mitigate new threats in real time. The *WifiInterface*, *DeauthDetector*, and *DeauthDefender* instances work together in a coordinated manner to ensure that the target device remains protected from deauthentication attacks. This ongoing process allows the system to adapt to evolving threats and maintain a robust security posture in the face of changing attack techniques and tools.

3.5. DDoS and Slowloris

3.5.1. Definition

Let $U = \{u_1, u_2, \ldots, u_n\}$ be the set of n UAVs, and let $S = \{s_1, s_2, \ldots, s_m\}$ be the set of m cloud servers. Let $R = \{r_1, r_2, \ldots, r_k\}$ be the set of k resources that each u_i needs to access, such as data storage or processing capabilities. Let $T = \{t_1, t_2, \ldots, t_l\}$ be the set of l types of legitimate traffic that each web server s_j needs to handle. A DoS attack D is a binary function $D(S, t) -> \{0, 1\}$ that maps each cloud server s_j and type of traffic t_i to a binary value indicating whether a DoS attack is occurring. If $D(S, t) = 1$, then a DoS attack is occurring on server s_j with respect to traffic type t_i. Similarly, a Slowloris attack L is a binary function $L(S, t) -> \{0, 1\}$ that maps each cloud server s_j and type of traffic t_i to a binary value indicating whether a Slowloris attack is occurring. If $L(S, t) = 1$, then a Slowloris attack is occurring on server s_j with respect to traffic type t_i.

The impact of a DoS attack on the cloud server can be expressed: For each u_i, if there exist a server s_j and resource r_I such that $D(s_j, t) = 1$ and r_I is required by u_i, then the UAV is unable to access that resource, resulting in delays or complete failures in UAV operations. This can be expressed mathematically, as presented in Equation (1).

$$\exists\, s_j \in S,\, r_I \in R,\, t \in T : D(s_j, t) = 1 \wedge r_I \textit{ is required by } u_i \rightarrow \sim access(r_I, u_i) \quad (1)$$

where "$\exists$" represents "there exists", "$\in$" represents "belongs to", "$\wedge$" represents "logical AND", and "$\sim$" represents "not". The impact of a Slowloris attack on the web server can be expressed as follows: For each u_i, if there exist a server s_j and type of traffic t_i such that $L(s_j, t_i) = 1$ and the traffic type t_i is required by u_i, then the server becomes overwhelmed with incomplete HTTP requests [24], preventing it from handling legitimate traffic. This can be expressed mathematically, as shown in Equation (2).

$$\exists\, s_j \in S,\, t_i \in T : L(s_j, t_i) = 1 \wedge t_i \textit{ is required by } u_i \rightarrow \sim handle(t_i, s_j) \quad (2)$$

3.5.2. Sequence Diagram

As shown in Figure 10, two UAVs are connected to the cloud in this sequence diagram, and they transfer data to the cloud for storage. The cloud recognizes the delivery of data by transmitting an acknowledge packet (*ACK)* to each UAV, but an attacker floods the cloud with traffic and initiates a Slowloris attack, preventing the processing of legal traffic. Both *UAV1* and *UAV2* receive incorrect answers when requesting data from the cloud. This demonstrates the severity of a DDoS or Slowloris attack on a UAV–cloud system and the significance of avoiding it.

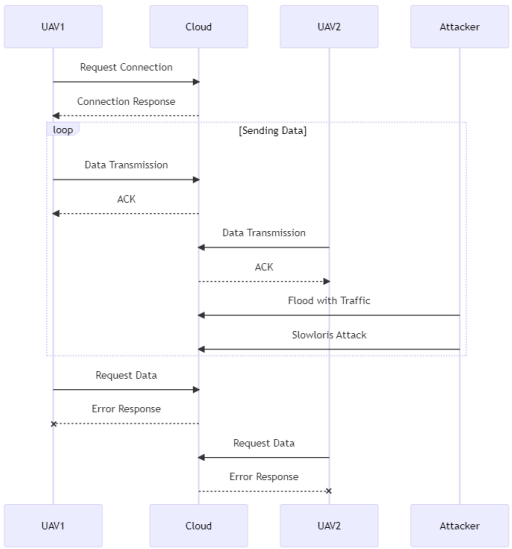

Figure 10. DDoS and Slowloris attack.

To avoid DDoS and Slowloris attacks in a UAV–cloud system, robust security mechanisms, including rate restriction, traffic filtering, and monitoring for anomalous traffic patterns, should be included. In addition, the system must be able to scale up or down to accommodate traffic spikes and prevent system overloads. Lastly, frequent security audits and penetration testing can assist detection and resolve any system vulnerabilities prior to their exploitation by attackers.

3.5.3. Defense Class Diagram

The class diagram in Figure 11 shows the relationships between the *DefendAgainstDDoSAndSlowLoris* class, the *Net*, *NetServer*, and *NetSocket* classes. *DefendAgainstDDoSAndSlowLoris* is the main class that has the responsibility of defending against DDoS and Slowloris attacks. It has private variables such as *target*, *port*, *MAX_CONNECTIONS*, *MAX_REQUESTS_PER_CONNECTION*, *requestData*, and *server*. The class has a constructor method that takes the *target* and *port* values as parameters, and a public method named *defendAgainstDDoSAndSlowLoris* that starts the server and handles incoming client connections.

Net is a class that represents the *node.js* net module and provides a method named *createServer* to create a new instance of *NetServer*. *NetServer* is a class that represents the server object created by calling the *net.createServer()* method. It has methods such as *listen*, on, and *close* to start the server, *add event listeners*, and close the server, respectively. *NetSocket* is a class that represents a client connection to the server. It has methods such as *on*, *write*, *destroy*, and *end* to handle incoming data, send data to the server, destroy the connection, and end the connection, respectively. *NetServer* has many *NetSocket* instances, as shown by the "has many" relationship arrow pointing from *NetServer* to *NetSocket*. *NetServer* creates an instance of *DefendAgainstDDoSAndSlowLoris*, as shown by the "creates" relationship arrow pointing from *NetServer* to *DefendAgainstDDoSAndSlowLoris*. *NetSocket* is used by *DefendAgainstDDoSAndSlowLoris* to handle incoming client connections, as shown by the "uses" relationship arrow pointing from *NetSocket* to *DefendAgainstDDoSAndSlowLoris*.

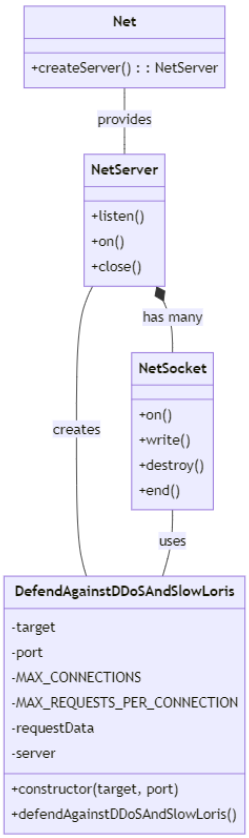

Figure 11. Class diagram of Slowloris and DDoS.

Based on the defense class diagram for DDoS and Slowloris attacks, the following defense mechanism scenario can be described:

1. **Initialization**: Instantiate the *DefendAgainstDDoSAndSlowLoris* class by providing target and port parameters to the constructor. The constructor initializes the server with the given target and port, setting up internal data structures such as *requestData* to track client connections and their request patterns. It also configures limits such as MAX_CONNECTIONS and MAX_REQUESTS_PER_CONNECTION to prevent abuse.
2. **Start defense**: Invoke the *defendAgainstDDoSAndSlowLoris* method to launch the defense mechanism. This method creates the server using the *Net* class's *createServer* method, binding the server to the specified target and port. It registers event listeners for incoming connections, invoking a callback function to handle each new connection.
3. **Handling connections**: Upon detecting a new client connection, the *NetServer* class generates a new *NetSocket* instance to manage the connection. This instance registers event listeners for incoming data, connection closures, and errors. Simultaneously, the *DefendAgainstDDoSAndSlowLoris* class processes each incoming connection, updating its internal *requestData* structure to track the number of active connections and requests per client.
4. **Monitoring connections**: The defense mechanism continuously assesses the *requestData* structure to evaluate the number of connections and requests from each client. If a client surpasses the MAX_CONNECTIONS or MAX_REQUESTS_PER_CONNECTION thresholds, the defense mechanism considers the connection suspicious, indicating a potential DDoS or Slowloris attack.
5. **Rate limiting and connection throttling**: To defend against Slowloris attacks, the defense mechanism may also implement rate limiting and connection throttling, ensuring that clients can only open a limited number of connections within a specific timeframe. This technique helps maintain server availability and prevents attackers from monopolizing resources.
6. **Connection termination and blacklisting**: Upon identifying a suspicious client connection, the *NetSocket* instance's destroy or end method is invoked to forcefully terminate or gracefully close the connection. In more advanced implementations, the defense mechanism may also blacklist the IP address of the suspicious client, preventing it from opening new connections for a specified duration.
7. **Logging and alerting**: The defense mechanism can incorporate logging and alerting capabilities, allowing system administrators to monitor server activity and receive notifications when potential attacks are detected. This feature enables prompt intervention and further analysis, helping to maintain server stability and security.
8. **Ongoing defense**: The defense mechanism remains vigilant, continuously monitoring and managing incoming connections to ensure the server's accessibility for legitimate clients while defending against DDoS and Slowloris attacks.

3.6. Flooding

3.6.1. Definition

Let N be the total number of devices that can be used for the flooding assault, let p_i be the probability that device i will be used in the flooding assault $for\ i\ in\ \{1,\ 2,\ \ldots,\ N\}$ and let the maximum traffic capacity that the cloud infrastructure can handle, t_i, be the traffic generated by device i in the cloud infrastructure $for\ i\ in\ \{1,\ 2,\ \ldots,\ N\}$. The objective is to maximize the total amount of traffic generated by the flooding assault subject to the maximum capacity of the cloud infrastructure, where we maximize function of $sum(p_i\ *\ t_i)\ for\ i\ in\ \{1,\ 2,\ \ldots,\ N\}$. This is subject to the following: $sum(p_i) = 1$ (exactly one device is used in the flooding assault), $sum(p_i\ *\ t_i) > T$ (the total amount of traffic generated by the devices in the flooding assault is greater than the maximum traffic capacity that the cloud infrastructure can handle), $p_i >= 0\ for\ i\ in\ \{1,\ 2,\ \ldots,\ N\}$

(the probability that each device is used in the flooding assault is non-negative), and $t_i >= 0 \; for \; i \; in \; \{1, 2, \ldots, N\}$ (the traffic generated by each device is non-negative)

The first constraint ensures that exactly one device is used in the flooding assault. The second constraint limits the total traffic generated by the devices to the maximum capacity of the cloud infrastructure. The third and fourth constraints ensure that the probabilities and traffic are non-negative. Note that this formulation assumes that the probability that each device is used in the flooding assault is independent of the traffic generated by the other devices [25].

3.6.2. Sequence Diagram

As shown in Figure 12, two UAVs are connected to the cloud in this sequence diagram, and they transfer data to the cloud for storage. The cloud recognizes the delivery of data by transmitting an acknowledge packet (*ACK)* to each UAV. Nevertheless, an adversary overwhelms the cloud's processing capabilities with traffic, rendering it unavailable. Both *UAV1* and *UAV2* receive incorrect answers when requesting data from the cloud. This demonstrates the severity of a flooding assault in a UAV–cloud system and the significance of avoiding it.

To prevent flooding assaults in a UAV–cloud system, the system should employ robust security mechanisms such as rate limitation, traffic filtering, and monitoring for anomalous traffic patterns. In addition, the system must be able to scale up or down to accommodate traffic spikes and prevent system overloads. Lastly, frequent security audits and penetration testing can assist detection and resolve any system vulnerabilities prior to their exploitation by attackers.

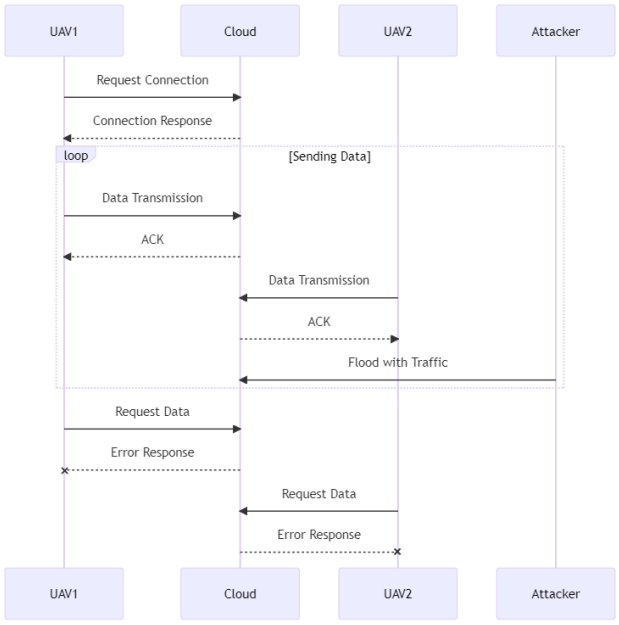

Figure 12. Flood with traffic.

3.6.3. Defense Class Diagram

As depicted on the Figure 13, the class diagram shows the relationships between five classes: *FloodDetector*, *FloodDefender*, *pcap*, *Set*, *console*, and *Object*. *FloodDetector* is a class that has private instance variables *interface*, *maxPacketsPerSecond*, *packetCounts*, *scores*, and *blockedIPs*, as well as a public constructor method *constructor* with *interface: string*, *maxPacketsPerSecond: number*. It uses the *pcap*, *Set*, *console*, and *Object* classes to implement its functionality. *FloodDefender* is a class that has private instance variables *interface* and *maxPacketsPerSecond*, as well as a public *constructor* method *constructor(interface: string, maxPacketsPerSecond: number)* and a public method *defend(): void*. It calls the *FloodDetector*

class to create a new object and start detecting flooding attacks. *pcap* is a class that has a public method, *decode*. *packet(rawPacket: string): object*, which decodes a raw packet from a network interface.

Set is a class that has a private instance variable *items* and public methods *constructor()*, *add(item: any): void*, and *has(item: any): Boolean*. It is used by *FloodDetector* to keep track of blocked IP addresses. *console* is a class that has a public method *log(message: string): void*, which logs a message to the console. It is used by *FloodDetector* to log detected flooding attacks. *Object* is a built-in JavaScript class that has a public method k*eys(obj: object): Array<string>*, which returns an array of the object's keys. It is used by *FloodDetector* to iterate over the *packetCounts* and scores objects. The arrows in the diagram show the relationships between the classes. The *FloodDetector* class uses *pcap*, *Set*, *console*, and *Object* classes, so it has four "uses" relationships with those classes. The *FloodDefender* class calls the *FloodDetector* class, so it has a "call" relationship with it.

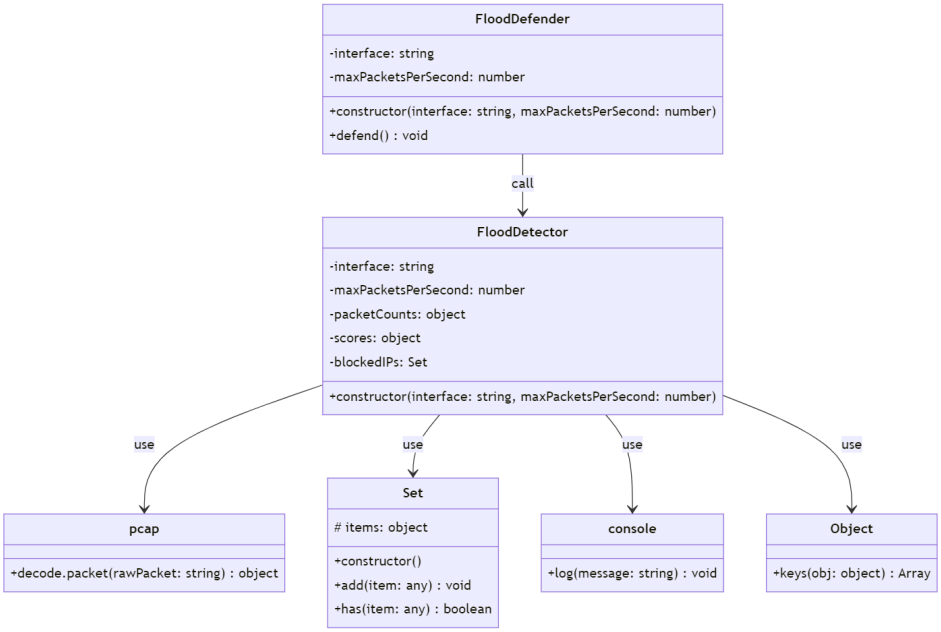

Figure 13. Flood defender class diagram.

Based on the defense class diagram of a flooding attack, here is a detailed implementation of the defense process:

1. **Initialization**: Instantiate the *FloodDefender* class by providing the network interface and *maxPacketsPerSecond* parameters to the constructor. This constructor initializes an instance of the *FloodDetector* class, passing along the same parameters.
2. **Start defense**: Invoke the *defend()* method of the *FloodDefender* class. This method starts the *FloodDetector* instance, which begins monitoring the network interface for potential flooding attacks.
3. **Packet capturing and decoding**: The *FloodDetector* class uses the *pcap* class to capture packets from the network interface. The *pcap* class decodes the packets and extracts relevant information, such as the source IP addresses.
4. **Track packet counts**: For each IP address, the *FloodDetector* maintains packet counts and scores in the *packetCounts* and scores objects. It updates these counts and scores as new packets are processed.
5. **Detect flooding attacks**: The *FloodDetector* continuously evaluates packet counts and scores for each IP address. If the number of packets received from an IP address surpasses the *maxPacketsPerSecond* threshold, the *FloodDetector* considers it a potential source of a flooding attack.

6. **Block malicious IP addresses**: When an IP address is identified as a source of a flooding attack, the *FloodDetector* adds it to the *blockedIPs* Set. This Set keeps track of blocked IP addresses, preventing further traffic from these addresses.
7. **Logging**: The *FloodDetector* uses the console class to log detected flooding attacks. It logs relevant information, such as the source IP address and the detected packet rate. System administrators can use this information to analyze the attack and take further action if necessary.
8. **Ongoing defense**: The defense mechanism remains active, continuously monitoring incoming packets and updating the packet counts and scores. It maintains a list of blocked IP addresses in the *blockedIPs* Set, providing ongoing defense against flooding attacks.

3.7. GPS Spoofing

3.7.1. Definition

Let $P = (p_1, p_2, p_3)$ be the true position of a GPS receiver in R^3, and let $S = (s_1, s_2, s_3)$ be the position that the GPS receiver thinks it is due to GPS spoofing. GPS spoofing is a technique that involves transmitting false GPS signals to a receiver, leading it to provide inaccurate position data. We can model this problem using the following mathematical formulation: Find a function $f: R -> R^3$ such that $f(P) = S$ and f is consistent with the actual GPS signals in the area. This means that for each $i = 1, 2, 3$, there exists a function $g_i: R^3 -> R$ that relates the spoofed GPS signal s_i to the actual GPS signal a_i, and the function f satisfies $f_i(P) = g_i(f(P))$ $for\ all\ i$.

The problem of GPS spoofing can then be formulated as an optimization problem, where the objective is to minimize the distance between the true position P and the spoofed position S, subject to the constraint that the function f is consistent with the actual GPS signals in the area. This can be expressed mathematically as follows: $minimize\ ||P - S||\ subject\ to\ f_i(P) = g_i(f(P))\ for\ all\ i$.

The optimal solution to this problem will depend on the specific objectives of the spoofing, as well as the technical and ethical constraints that are imposed. For example, to limit the extent of the spoofing, additional constraints can be added to the optimization problem, such as a constraint that restricts the range of the spoofing device. To ensure that the spoofing is used for legitimate purposes only, the optimization problem can be augmented with a set of ethical or legal constraints that must be satisfied by the solution [26].

3.7.2. Sequence Diagram

The sequence diagram in Figure 14 depicts the process of GPS spoofing. At the beginning, the attacker is continually spoofing the GPS signal and transmitting it to the GPS module of the UAVs. Then, the GPS module delivers the updated position of the UAVs to the cloud for processing based on the faked signal. At the same time, in a database, the cloud saves location information. In the meantime, three valid satellites are transmitting signals to the GPS module.

The first valid satellite transmits a signal to *UAV1*'s GPS module. The GPS module transmits *UAV1*'s current location to the cloud, where it is stored in a database. The second valid satellite transmits a signal to *UAV2*'s GPS module. The GPS module transmits *UAV2*'s current location to the cloud, where it is stored in a database. The third genuine satellite transmits signals to both *UAV1* and *UAV2*, causing the GPS modules on both UAVs to receive the same location update. Both drones' GPS units transmit their individual positions to the cloud, which saves them in a database. This graphic depicts, in further detail, the GPS spoofing process in a multi-UAV–cloud system, where an attacker may alter the GPS signals to control the location of the UAVs and disrupt the system's regular operation.

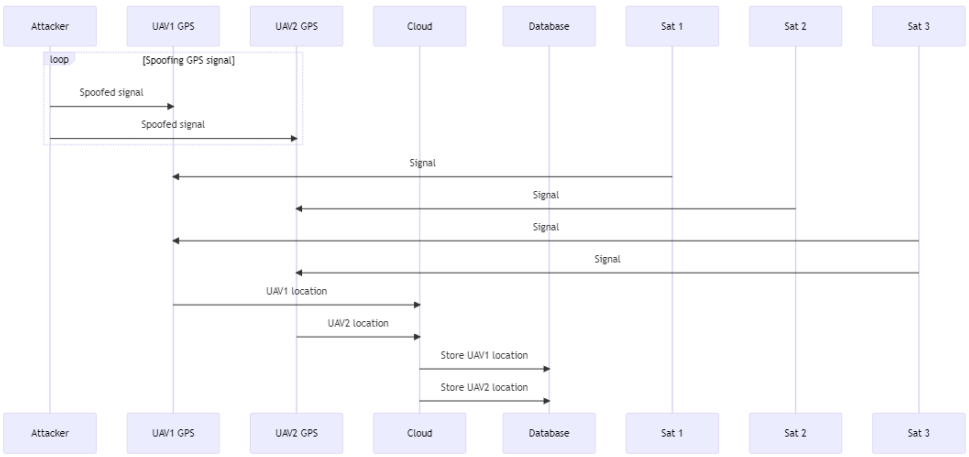

Figure 14. GPS spoofing sequence diagram.

3.7.3. Defense Class Diagram

As depicted on the Figure 15, the class diagram represents the relationships between four classes: *GpsSpoofingDetector*, *GpsSpoofingDefender*, *Position*, and *NetworkInterface*. *GpsSpoofingDetector* is responsible for detecting GPS spoofing by receiving *GPS* signals and analyzing them to determine if there is any indication of spoofing. It contains private instance variables *lastPosition* and *lastTime* which keep track of the last GPS position and time received by the detector. The *isValidPosition* method checks whether a given GPS position is valid or not, and the *checkForGpsSpoofing* method analyzes the position to detect spoofing. If spoofing is detected, the *blockSpoofing* method is called, which uses a *NetworkInterface* to block the connection from the IP address of the spoofing device. Finally, the *getDistance* method calculates the distance between two GPS positions.

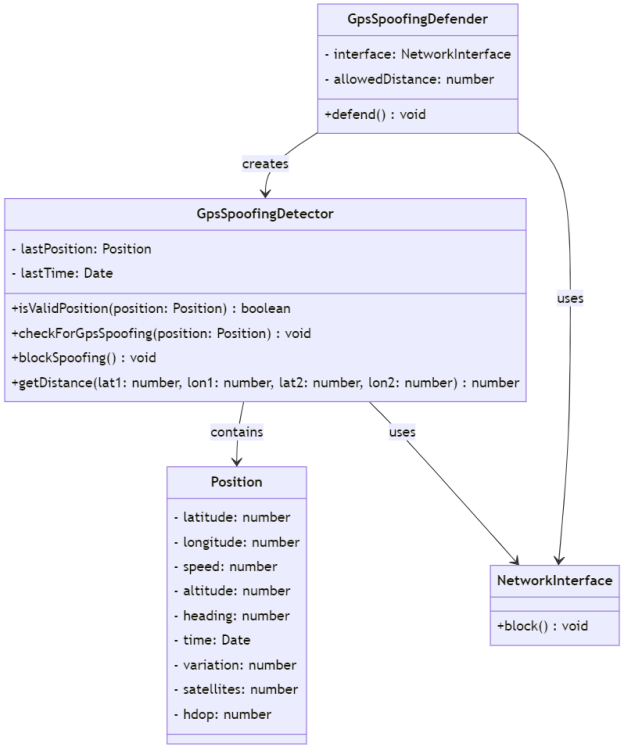

Figure 15. GPS spoofing detector class diagram.

GpsSpoofingDefender creates a *GpsSpoofingDetector* object and uses it to defend against GPS spoofing. It has private instance variables interface, which is a *NetworkInterface* object used to block the connection in case of GPS spoofing, and *allowedDistance*, which is the maximum allowed distance between the current and previous GPS positions. *Position* is a simple class that represents a GPS position with private instance variables *latitude*, *longitude*, *speed*, *altitude*, *heading*, *time*, *variation*, satellites, and *hdop*.

NetworkInterface is a class that represents a network interface and has a public method block, which blocks the connection from a specific IP address. The arrows in the class diagram represent the relationships between the classes. The *GpsSpoofingDetector* class contains a *Position* object, and uses a *NetworkInterface* object to block the connection. The *GpsSpoofingDefender* class creates a *GpsSpoofingDetector* object and uses a *NetworkInterface* object to block the connection. The –> and "uses" arrows indicate containment and dependency relationships, respectively.

Based on the defense class diagram of a GPS spoofing attack, here is a detailed implementation of the defense process:

1. **Initialization:** GpsSpoofingDefender initializes the defense system by creating an instance of *GpsSpoofingDetector* and providing it with a *NetworkInterface* object. This *NetworkInterface* object is responsible for network communication, enabling the blocking of connections from potential spoofing devices. The *GpsSpoofingDefender* also sets an *allowedDistance* threshold that acts as a safety measure to identify suspicious changes in GPS positions.
2. **Monitoring:** The *GpsSpoofingDetector* is responsible for monitoring incoming GPS signals continuously. This class analyzes these signals by comparing the received GPS positions with the previously recorded positions. It verifies the validity of the GPS signals by evaluating factors such as position changes, speed, heading, altitude, and time stamps. Moreover, it checks the number of satellites, variation, and the horizontal dilution of precision (hdop) values to determine the accuracy and reliability of the GPS signals.
3. **Detection process:** As part of the detection process, the GpsSpoofingDetector calculates the distance between consecutive GPS positions using the *getDistance* method. If the distance exceeds the *allowedDistance* threshold set by the *GpsSpoofingDefender*, the system raises an alarm or takes other defensive actions, such as switching to an alternative positioning system or alerting the user about the possible GPS spoofing attack.
4. **Take action:** If the *GpsSpoofingDetector* identifies a GPS spoofing attack, it proceeds to mitigate the attack by blocking the connection from the IP address of the suspected spoofing device. This is achieved by invoking the *blockSpoofing* method, which in turn calls the block method of the *NetworkInterface* object.
5. **GPS data management**: The Position class is used to store and manage GPS position data, making it easier to analyze and process the received signals for detecting GPS spoofing. It maintains information about latitude, longitude, speed, altitude, heading, time, variation, satellites, and hdop values, providing a comprehensive data structure for the analysis process.
6. **Monitor log**: Throughout the defense process, the *GpsSpoofingDefender* and *GpsSpoofingDetector* may log relevant information or events using a logging mechanism, such as the console class. This logged information can be useful for further analysis, system audits, or incident response actions.

3.8. Telemetry Spoofing

3.8.1. Definition

Let D be a set of telemetry data collected from a UAV–cloud system, and let D' be the set of spoofed telemetry data. The goal of the attacker is to generate a sequence of spoofed data points $d'_1, d'_2, \ldots, d'_T$ that can be inserted into the telemetry data stream without being detected, and that achieve a specific attack objective. Formally, let X be the set of features that describe the telemetry data, such as the UAV's *location*, *altitude*, *airspeed*,

and *sensor* readings. Let Y be the set of possible spoofed data points that can be generated by the attacker, and let $f : X \rightarrow Y$ be a function that maps telemetry data to spoofed data points.

The attacker's objective is to choose a sequence of inputs $x_1, x_2, \ldots, x_T$ that maximize a given objective function $R_T(1, y_2, \ldots, y_T)$, where $y_t = f(x_t)$ is the corresponding spoofed data point. The objective function R_T can be designed to reflect different attack goals, such as causing the UAV to deviate from its intended path, collide with another object, or perform unauthorized actions. To generate the spoofed data points, the attacker can use a variety of techniques, such as statistical modeling, machine learning, or signal processing. The spoofing algorithm can be trained on a dataset of telemetry data and corresponding spoofed data points, using supervised or unsupervised learning techniques. The training data can be generated through simulation or by collecting real telemetry data from a UAV–cloud system.

To evaluate the performance of the spoofing algorithm, the attacker can use metrics such as success rate, attack effectiveness, and computational complexity. Success rate is the percentage of spoofed data points that are accepted by the system without being detected, while attack effectiveness measures the impact of the attack on the UAV–cloud system's performance. Computational complexity is the amount of computational or energy resources needed to generate the spoofed data points, which is important in resource-constrained environments [27].

The problem of generating spoofed telemetry data can be formulated as an optimization problem: $maximize\ R_T(y_1, y_2, \ldots, y_T)$ subject to $y_t = f(x_t)\ for\ t = 1, 2, \ldots, T$ $D' \subseteq Y \cdot P(Detect = 1 \mid D \cup D') \leq \varepsilon$, where ε is a given threshold on the probability of detection, and $P(Detect = 1 \mid D \cup D')$ is the probability of detection given the entire set of telemetry data. The optimization problem can be solved using techniques such as convex optimization, dynamic programming, or reinforcement learning, depending on the complexity of the spoofing algorithm and the attack objective.

3.8.2. Sequence Diagram

Figure 16 shows the process of the telemetry spoofing problem; the adversary delivers forged telemetry data to the UAV, which subsequently transmits it to the cloud. The cloud verifies the data, determines that it is faked, and requests that the UAV verify the data. The UAV confirms that the telemetry data has been faked and then transmits a report to the cloud. Lastly, the cloud notifies the attacker with the spoofed telemetry.

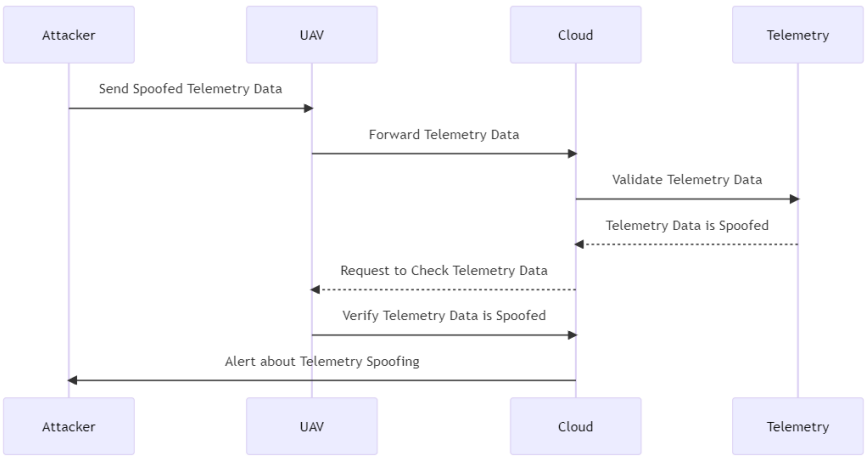

Figure 16. Telemetry spoofing problem sequence diagram.

3.8.3. Defense Class Diagram

In Figure 17, the *TelemetrySpoofingDetector* and *TelemetrySpoofingDefender* classes are shown as main classes, while the *Stats* class is an internal class. The *TelemetrySpoofingDetector*

has a composition relationship with the *Stats* class, as it uses an instance of Stats to store the calculated mean and standard deviation of altitude data. The *TelemetrySpoofingDefender* class has a composition relationship with the *TelemetrySpoofingDetector* class, as it holds an instance of *TelemetrySpoofingDetector* and delegates the start and stop methods to it.

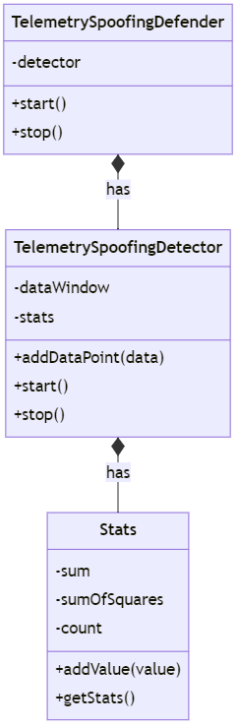

Figure 17. Telemetry spoofing defender class diagram.

Based on the defense class diagram of the telemetry spoofing attack, the defense mechanism can be explained as follows:

1. **Initialization**: An instance of the *TelemetrySpoofingDefender* class is created. During the instantiating process, the *TelemetrySpoofingDefender* constructs an instance of the *TelemetrySpoofingDetector* class, which is responsible for analyzing incoming telemetry data. The *TelemetrySpoofingDetector*, in turn, creates an instance of the *Stats* class to maintain and calculate essential statistical values related to the altitude data (e.g., mean and standard deviation).
2. **Telemetry data collection**: As the UAV receives telemetry data, it is passed to the *TelemetrySpoofingDetector* object using the *addDataPoint* method. The *TelemetrySpoofingDetector* maintains a *dataWindow*, which stores the collected telemetry data points over a sliding window of time or a fixed number of data points. This *dataWindow* allows for continuous analysis of the incoming telemetry data.
3. **Detection process initiation**: The *TelemetrySpoofingDefender* initiates the detection process by calling the start method on the *TelemetrySpoofingDetector* object. This sets up a continuous loop or an event-driven mechanism that analyzes the collected telemetry data at regular intervals or upon receiving new data points.
4. **Statistical analysis**: As part of the detection process, the *TelemetrySpoofingDetector* analyzes the collected telemetry data. It leverages the Stats object to compute the mean and standard deviation of the altitude data in the *dataWindow*. These statistical values are then used to assess the normalcy of the incoming telemetry data.
5. **Anomaly detection**: The *TelemetrySpoofingDetector* compares the computed mean and standard deviation values with predefined acceptable thresholds, which can be determined based on historical data, domain knowledge, or other techniques. If these

statistical values are found to be beyond the acceptable limits, it indicates the presence of a telemetry spoofing attack.

6. **Triggering response mechanisms**: Upon detecting a telemetry spoofing attack, the *TelemetrySpoofingDetector* can trigger various response mechanisms to mitigate the impact of the attack. These may include (i) alerting the UAV system operator to take manual corrective actions, (ii) automatically adjusting the UAV's control algorithm to disregard the spoofed telemetry data and rely on alternative navigation sources, (iii) initiating countermeasures to block or jam the source of the spoofed telemetry signals, and (iv) activating redundant systems or failsafe modes to ensure the safe operation of the UAV under attack.
7. **Stopping the detection process**: The *TelemetrySpoofingDefender* can halt the detection process at any moment by calling the stop method on the *TelemetrySpoofingDetector* object, which stops the continuous analysis loop or event-driven mechanism.

3.9. Gray Hole Attack

3.9.1. Definition

Let $G = (V, E)$ be a directed graph representing the communication network between UAVs and cloud servers, where V is the set of nodes representing UAVs and cloud servers, and E is the set of directed edges representing the communication links between them. Let t_e be the current traffic rate on edge e in E, and let c_e be the maximum capacity of edge e.

Suppose there is a malicious actor who gains access to a network device and launches a gray hole attack by selectively blocking or transmitting traffic on certain edges in E during a time interval $[0, T]$. Let $S(t)$ be the set of edges that the attacker selects for blocking or transmitting traffic at time t, and let $\rho(t)$ be the fraction of traffic that the attacker blocks or transmits on the selected edges at time t. Then, the attacker can reduce the total traffic rate on the network, causing congestion and disrupting legitimate traffic. The objective is to maximize the disruption caused by the attack during the time interval $[0, T]$, while minimizing the detection probability of the attack.

We can formulate this as a dynamic optimization problem, as follows: Maximize: $\int_0^T \sum_{e \in S(t)}^{2dt} (t_e(t) - \rho(t) * t_e(t))$ $Subject\ to\ t_e(t) <= c_e,\ for\ all\ e\ in\ E\ and\ t\ in\ [0,\ T]$ $\sum_{e \in S(t)} \rho(t) * t_{e(t)} <= \alpha,\ for\ all\ t\ in\ [0,\ T]\ P(D(t)) <= \beta$, for all t *in* $[0, T]$, where $t_{e(t)}$ is the traffic rate on edge e at time t, $S(t)$ is the set of edges selected by the attacker at time t, $\rho(t)$ is the fraction of traffic that the attacker blocks or transmits on the selected edges at time t, α is the maximum total traffic reduction that the attacker can achieve at any time, β is the maximum detection probability of the attack at any time, and $D(t)$ is the event that the attack is detected at time t.

The objective function maximizes the difference between the original traffic rate and the adjusted traffic rate on each selected edge over the time interval $[0, T]$. The first constraint ensures that the adjusted traffic rate does not exceed the capacity of each edge at any time. The second constraint limits the maximum total traffic reduction that the attacker can achieve at any time. The third constraint limits the maximum detection probability of the attack at any time, which could depend on factors such as the quality of the detection algorithm, the frequency of monitoring, and the noise level in the traffic data.

3.9.2. Sequence Diagram

As depicted in Figure 18, the adversary launches a gray hole assault against numerous UAVs in the system. Both *UAV1* and *UAV2* transmit incomplete data to the cloud, and the cloud demands the missing information. This request is intercepted by the attacker, who then transmits changed data to both *UAV1* and *UAV2*. When both UAVs transmit the altered data to the cloud, the cloud receives them, unaware that they have been altered, providing the attacker unauthorized access to the system. Lastly, the attacker uploads the altered data to the cloud.

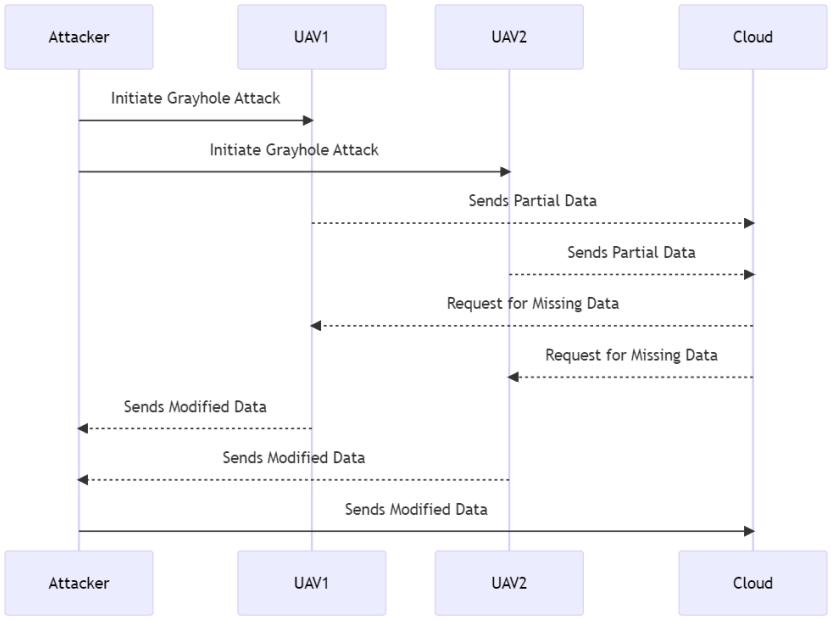

Figure 18. Gray hole sequence diagram.

3.9.3. Defense Class Diagram

In Figure 19, the *GrayholeDetector* class is responsible for detecting gray hole attacks on a UAV by monitoring the data that are being sent and received over the network. It has a *networkInterface* object, which is an instance of the *NetworkInterface* class, to listen for outgoing and incoming data and add them to the *outgoingDataQueue* and *incomingDataQueue*, respectively. It has a *monitorInterval* attribute to specify the time interval for monitoring the data queues. It also has three methods: *monitorDataQueues()* to check for any patterns in the data that may indicate a gray hole attack, *detectGrayholePattern(dataQueue)* to detect patterns in the data, and *blockGrayhole()* to block the gray hole attacker's IP address.

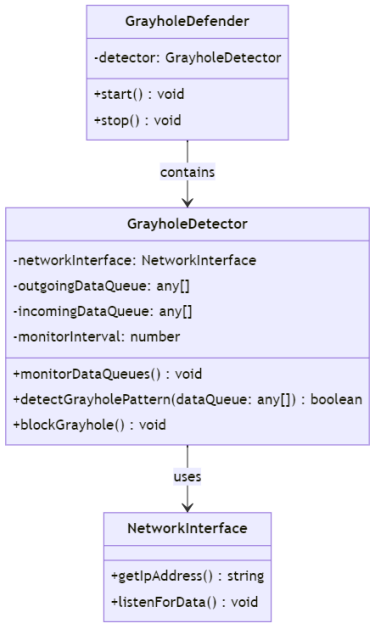

Figure 19. Grayhole class diagram.

The *GrayholeDefender* class is responsible for starting and stopping the *GrayholeDetector*. It has a detector attribute that is an instance of the *GrayholeDetector* class. It has two methods: *start()* to start the *GrayholeDetector* and *stop()* to stop the *GrayholeDetector*. The

NetworkInterface class is a mock class that provides the IP address of the *network interface* and a method to listen for outgoing and incoming data. It is used to simulate network traffic for testing purposes. The arrows in the diagram show the relationships between the classes. The *GrayholeDetector* class uses an instance of the *NetworkInterface* class to listen for outgoing and incoming data. The *GrayholeDefender* class contains an instance of the *GrayholeDetector* class.

Based on the defense gray hole attack class diagram, the defense mechanism scenario can be explained as follows:

1. **Initialization**: An instance of the *GrayholeDefender* class is created to manage the defense process. The *GrayholeDefender* initializes its detector attribute by creating an instance of the *GrayholeDetector* class. The *GrayholeDetector* initializes its *networkInterface* object with an instance of the *NetworkInterface* class, which is responsible for listening to incoming and outgoing network traffic, simulating a real-world environment for testing purposes.
2. **Commencement of monitoring**: The *GrayholeDefender* class invokes the *start()* method, which signals the *GrayholeDetector* to commence monitoring the network traffic. Utilizing its *networkInterface* object, the *GrayholeDetector* observes the network traffic, collecting and storing outgoing and incoming data in the *outgoingDataQueue* and *incomingDataQueue*, respectively. The *GrayholeDetector* employs a *monitorInterval* attribute to establish the frequency at which it checks the data queues for potential gray hole attack patterns.
3. **Data queue examination**: The *GrayholeDetector* periodically examines the contents of the *outgoingDataQueue* and *incomingDataQueue* based on the *monitorInterval*. It processes the data, searching for irregularities, inconsistencies, or patterns that could indicate the presence of a gray hole attack. The data analysis is performed by the *detectGrayholePattern(dataQueue)* method, which scrutinizes the queued data to discern any malicious activities.
4. **Attack detection and confirmation**: If the *detectGrayholePattern()* method identifies a suspicious pattern or anomaly that suggests a gray hole attack, the *GrayholeDetector* validates its findings before taking any further action. This validation step may involve comparing the detected pattern against known gray hole attack signatures or employing statistical analysis methods to ensure that the pattern is not a false positive.
5. **Attacker blocking and reporting**: Once the *GrayholeDetector* confirms the presence of a gray hole attack, it calls the *blockGrayhole()* method. This method leverages the *networkInterface* object to block the IP address of the gray hole attacker, effectively neutralizing the threat. In addition to blocking the attacker, the *GrayholeDetector* may also generate a report or alert containing details about the attack, such as the attacker's IP address, the timestamp of the attack, and the suspicious patterns observed. This information can be shared with network administrators or security teams for further investigation and potential improvements to the system's defenses.
6. **Monitoring termination and cleanup**: If necessary, the *GrayholeDefender* class can call the *stop()* method to cease the *GrayholeDetector's* monitoring activities. This may be useful in cases where the system is shutting down or when the network environment has been deemed safe. The *GrayholeDetector* then performs any necessary cleanup, such as deallocating memory or closing open connections.

3.10. Impersonation Attack

3.10.1. Definition

Let S be a set of sensitive data and resources involved in UAV–cloud systems. Let A be the set of possible attackers who may attempt to gain unauthorized access to the system through impersonation attacks. Let C be the set of cloud computing resources and UAVs used in the system. The problem is minimizing the risk of impersonation attacks in UAV–cloud systems while ensuring the security and integrity of the data being collected

and transmitted. This can be achieved by implementing effective security measures that prevent or detect impersonation attacks [28].

The objective function can be defined as minimize R, where R is the overall risk of impersonation attacks in the UAV–cloud system. The risk of impersonation attacks can be defined as a function of the probability of an attack occurring, the potential impact of the attack on the system, and the sensitivity of the data and resources involved. This can be mathematically formulated as $R = P \times I \times S$, where P is the probability of an impersonation attack occurring, I is the potential impact of the attack on the system, and S is the sensitivity of the data and resources involved.

3.10.2. Sequence Diagram

The scenario of attack is shown in Figure 20. The attacker intercepts the credentials supplied to the authentication system by both *UAV1* and *UAV2*. The attacker impersonates *UAV2* to *UAV1* and vice versa, deceiving each UAV into believing it is talking with the other. Each UAV transmits data to the cloud; however, the cloud requires identification from each UAV. The cloud accepts the forged authentication and provides the attacker unauthorized system access.

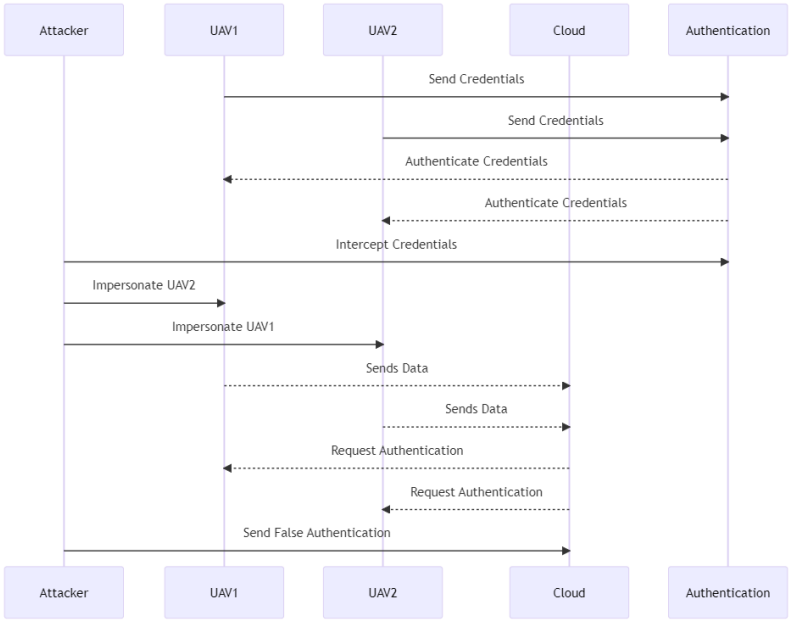

Figure 20. Impersonate sequence diagram.

3.10.3. Impersonation Class Diagram

As shown in Figure 21, it is the responsibility of the *ImpersonationDetector* class to identify impersonation assaults. It contains several methods, such as a constructor that initializes a *dgram* server to receive incoming telemetry data, a *connectToMongo()* method to connect to a *MongoDB* database to store telemetry data, a *onMessage()* method to handle incoming telemetry data, a *calculateStats()* method to calculate statistics of telemetry data, a *verifyAuthToken()* method to check if an authentication token is valid, and a *blockImpersonation()*. It also has the *onError()* and *close()* methods, which are used to handle server problems and end the service, respectively.

The class *ImpersonationDefender* is a wrapper for the class *ImpersonationDetector*. It has a constructor that creates an instance of *ImpersonationDetector*, a *start()* method for starting the server and connecting to *MongoDB*, and a *stop()* method for stopping the server and closing the *MongoDB* connection. The *ImpersonationDetector* class depends on the *MongoClient*, *dgram*, and *jwt* classes as external dependencies. The *ImpersonationDetector* class utilizes *MongoClient* to connect to a *MongoDB* database, *dgram* to establish a *UDP* server to accept

incoming telemetry data, and *jwt* to validate authentication tokens. As indicated by the "uses" arrows in the figure, the *ImpersonationDetector* class depends on these other classes.

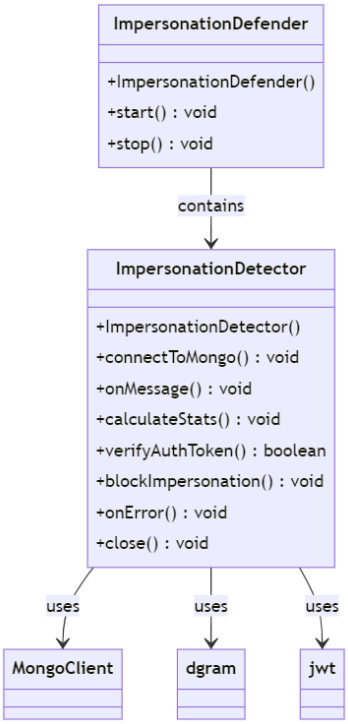

Figure 21. Impersonation defense class diagram.

In the defense scenario based on the impersonation attack class diagram, the system aims to protect against impersonation attacks by meticulously analyzing telemetry data, verifying authentication tokens, and deploying a multilayered defense approach. The process of implementation is as follows:

1. **Initialization**: The *ImpersonationDefender* class initializes an instance of the *ImpersonationDetector* class, which is responsible for detecting impersonation attacks. During this initialization, the *ImpersonationDetector* sets up a UDP server using the dgram class to receive incoming telemetry data.
2. **Store and analyze**: The *ImpersonationDetector* class connects to a *MongoDB* database using the *MongoClient* class to store and analyze telemetry data. This database stores historical and real-time data, allowing the system to compare incoming data with previously recorded patterns and identify anomalies.
3. **Processing**: When telemetry data are received, the *onMessage()* method in the *ImpersonationDetector* class processes the incoming data. This method parses the data and extracts relevant information such as the sender's IP address, device identifiers, and authentication tokens.
4. **Authentication**: To ensure the authenticity of the received data, the *verifyAuthToken()* method checks the authentication token using the *jwt* class. If the token is found to be invalid or expired, the system may immediately block the sender's IP address or flag the event for further investigation.
5. **Statistics measurement**: The *ImpersonationDetector* class calculates various statistics of the telemetry data, such as mean, standard deviation, and other relevant metrics, using the *calculateStats()* method. By comparing these statistics to historical data or predefined thresholds, the system can identify potential anomalies, which could be indicative of an impersonation attack.
6. **Take action**: If the system detects an impersonation attack based on the calculated statistics, invalid authentication tokens, or other suspicious patterns, the *blockImper-*

sonation() method is called to block the attacker's IP address, revoke authentication tokens, or take other appropriate countermeasures to mitigate the attack.

7. **Handle error**: To maintain system stability and handle unexpected errors, the *ImpersonationDetector* class employs the *onError()* method. This method logs errors, restarts the server if necessary, and informs system administrators of potential issues.
8. **Emergency shutdown**: The *ImpersonationDetector* can close the server connection and database connection using the *close()* method, which gracefully shuts down the system when needed.
9. **Start and stop mechanism**: The *ImpersonationDefender* class provides *start()* and *stop()* methods to control the defense mechanism by starting the server, connecting to the *MongoDB* database, or stopping the server and closing the database connection as needed.

3.11. Insider Attack

3.11.1. Definition

Optimize the allocation of security resources in a UAV–cloud network to minimize the risk of insider attacks while maintaining the continuity of the network's operations. The problem can be formulated as a constrained optimization problem, with the following mathematical formulation: Let N be the set of nodes in the UAV–cloud network, and let S be the set of available security resources. For each node $i \in N$, let s_i be the minimum level of security resources required to prevent potential vulnerabilities [29].

The objective function is to minimize the risk of insider attacks, which can be formulated as the sum of the probabilities of successful attacks on all nodes in N : $minimize \sum_{i \in N} P_i$, where Pi is the probability of a successful insider attack on node i. The constraints are as follows: the total budget for security resources must not exceed a given amount B : $\sum_{s \in S} c_s \leq B$ where c_s is the cost of security resource s.

Each node i must be assigned security resources that meet or exceed its minimum security requirement: $\sum_{s \in S} z_{si} \geq c_{si}$, where z_{si} is a binary decision variable indicating whether security resource s is assigned to node $z_{si} = 1$ or not ($z_{si} = 0$). The allocation of security resources must cover all potential attack vectors in the network: $\sum_{i \in N} \sum_{s \in S} x_{si} \sum_{j \in N}, c_{ij} \in C \; attack\_probability(i, j, c_{ij}) \geq 1$, where x_{si} is a binary decision variable indicating whether security resource s is assigned to any node in the network $x_{si} = 1$ or not ($x_{si} = 0$), and C is the set of possible attack types that an insider could carry out. The function $attack\_probability(i, j, c_{ij})$ returns the probability of an attack of type c_{ij} being successful from node i to node j.

The allocation of security resources must not impact the performance or reliability of the UAV–cloud network: $\sum_{i \in N} \sum_{s \in S} x_{si} \; impact(i, s) \leq T$, where $impact(i, s)$ is a function that measures the impact of assigning security resource s to node i, and T is a threshold value indicating the maximum allowable impact. The decision variables are the binary variables z_{si} and x_{si}, which determine the allocation of security resources to each node and across the network, respectively. The goal is to find the values of these decision variables that minimize the objective function while satisfying all constraints.

3.11.2. Sequence Diagram

As shown in Figure 22, the insider obtains access to *UAV1* and *UAV2* as well as the cloud in this scenario. The insider alters the data being transmitted by *UAV1* and then transmits the modified data to the cloud. The insider additionally obtains sensitive data from *UAV2* by requesting and obtaining them straight from the cloud. These acts are permissible due to the insider's allowed access to the system, yet they are detrimental to the business and may result in data breaches or other damage.

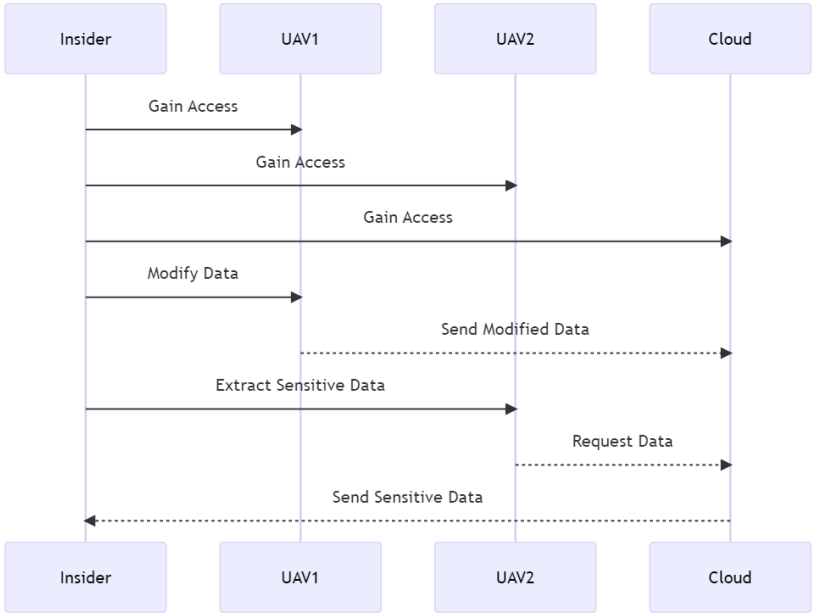

Figure 22. Insider sequence diagram.

3.11.3. Defense Class Diagram

As shown in Figure 23, the class diagram shows two main classes: *InsiderDetector* and *InsiderDefender*. The *InsiderDetector* class is responsible for detecting insider attacks on a UAV, and the *InsiderDefender* class is responsible for starting and stopping the *InsiderDetector*. The *InsiderDetector* class has a few private properties: *db*, *loginData*, *isModelTrained*, and model. It also has four public methods: *connectToMongo*, *trainModel*, *onLoginAttempt*, and *close*.

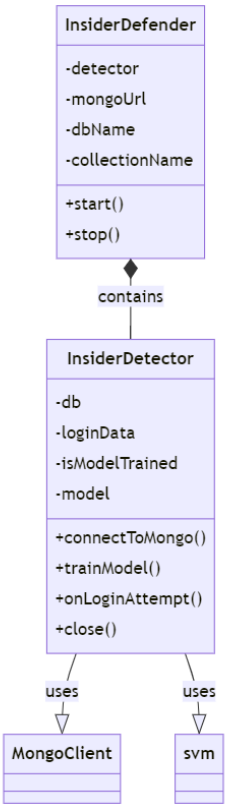

Figure 23. Insider defense class diagram.

The *connectToMongo* method connects to a *MongoDB* database and retrieves login data from a specified collection. The *trainModel* method prepares training data from the retrieved login data and trains a support vector machine (*SVM*) model using the *node-svm* library. The *onLoginAttempt* method checks for insider threats by predicting the outcome of a login attempt using the trained *SVM* model. If an unsuccessful login attempt is detected, the *takeAction* method is called to block the user's account. The *close* method closes the *MongoDB* connection.

The *InsiderDefender* class has three private properties: *detector*, *mongoUrl*, *dbName*, and *collectionName*. It has two public methods: *start* and *stop*. The *start* method creates a new instance of the *InsiderDetector* class, connects to the *MongoDB* database, and trains the *SVM* model. The *stop* method stops the *InsiderDetector* and *closes* the *MongoDB* connection. The *MongoClient* and *svm* classes are external dependencies used by the *InsiderDetector* class. The *InsiderDetector* class uses the *MongoClient* class to connect to the *MongoDB* database and the *svm* class to train the *SVM* model.

The defense mechanism against insider attacks, as described in the class diagram, operates in a comprehensive and systematic manner to identify and prevent potential threats. This process can be broken down into several steps and elaborated upon, as follows:

1. **Initialization**: The InsiderDefender class is responsible for managing the overall defense process. Upon initialization, it sets up the necessary configurations such as the MongoDB connection URL, database name, and collection name. These configurations are vital to ensuring the proper functioning of the defense mechanism.
2. **Data collection and preprocessing**: The InsiderDetector class, once instantiated by the InsiderDefender, connects to the MongoDB database and retrieves historical login data from the specified collection. These data serve as the foundation for training the SVM model. The login data may include various features such as timestamps, user IDs, IP addresses, and other relevant information that can help identify patterns in user behavior.
3. **Feature extraction and model training**: The InsiderDetector class preprocesses the collected data and extracts relevant features to create a dataset suitable for training the SVM model. During this process, data normalization, feature scaling, and other transformations may be applied to improve the model's accuracy and performance. Once the training dataset is prepared, the trainModel method is called to train the SVM model using the node-svm library.
4. **Real-time monitoring and prediction**: With the trained SVM model, the *InsiderDetector* class actively monitors login attempts in real time. For each login attempt, the onLoginAttempt method extracts the same features used during the training phase and feeds them into the SVM model. The model then predicts whether the login attempt is legitimate or potentially malicious based on the learned patterns in the historical data.
5. **Threat detection and mitigation**: If the SVM model predicts a malicious login attempt, it triggers the *takeAction* method, which can involve various security measures to block the user's account and prevent unauthorized access. This step may also include issuing alerts to security personnel, logging the event for further analysis, and implementing additional security measures to safeguard the system.
6. **Continuous improvement and adaptability**: The *InsiderDetector* can periodically update the SVM model by retraining it with new data, ensuring that the model stays up to date with the latest trends and patterns in user behavior. This continuous improvement process helps maintain the effectiveness of the defense mechanism over time.
7. **System shutdown and resource management**: The *InsiderDefender's stop* method can be called when the defense process is no longer needed or when the system is shutting down. This method ensures the proper termination of the *InsiderDetector* instance and the release of resources, such as closing the *MongoDB* connection.

3.12. Jamming Attack

3.12.1. Definition

Let S_t be the genuine signal on a wireless communication channel at time t, and let N_t be the noise signal at time t. The total signal at time t is then given by $X_t = S_t + N_t$. A jamming attack occurs when an attacker transmits a signal J_t on the same frequency as the targeted channel, resulting in a jamming signal J_t. The total signal at time t in the presence of a jamming attack is then given by $Y_t = S_t + J_t + N_t$. The goal of a jamming attack is to disrupt the normal operation of the wireless communication channel. This disruption can be quantified by a disruption function $D(Y_t)$, which measures the degree to which the jamming attack has disrupted the signal at time t. The attacker's objective is to maximize the disruption function, while the defender's objective is to minimize it. Jamming attacks can be directed at a single target, or they can be of a broader nature and impact a large geographical area. The extent of the attack can be quantified by a geographic function $G(Y_t)$, which measures the area over which the jamming attack is effective [30].

3.12.2. Sequence Diagram

As shown in Figure 24, the jammer provides a signal of interference to *UAV1* while it transmits data to the cloud. This interrupts the transport of data between *UAV1* and the cloud, preventing the cloud from receiving any data from *UAV1*. The jammer then transmits a signal of interference to *UAV2* when it is delivering data to the cloud, preventing the cloud from receiving any data from *UAV2*. Due to the jamming assaults, both UAVs are unable to send data to the cloud, and the cloud does not receive any data. This may result in a delay in data transmission or the complete loss of data, causing serious harm to the company.

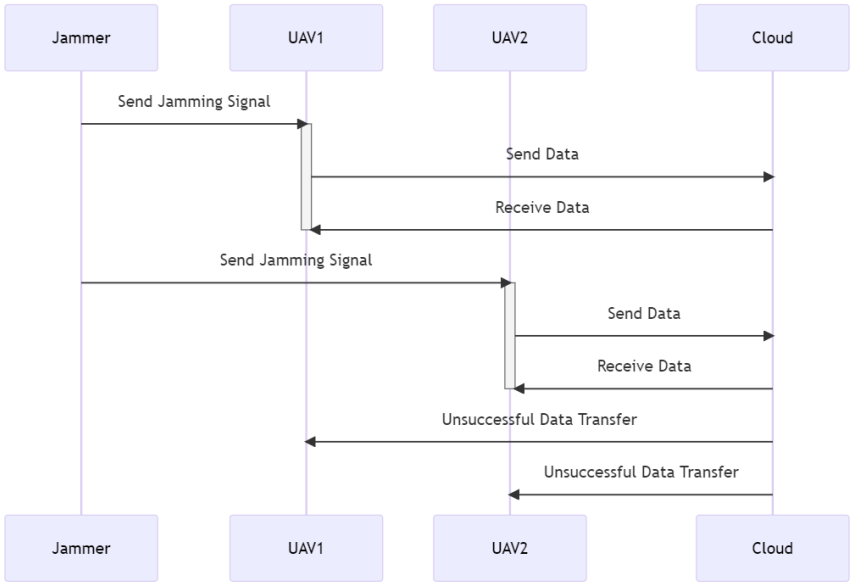

Figure 24. Jamming sequence diagram.

3.12.3. Defense Class Diagram

As shown in Figure 25, the class diagram depicts the relationships and attributes of the *JammingDetector* and *JammingDefender* classes, along with some related classes. The *JammingDetector* class has several private attributes, including a server attribute of type *dgram.Socket* and an *mqttClient* attribute of type *mqtt.Client*. It has several public methods, including *connectToCloudBroker()*, *onMessage()*, *calculateStats()*, *retrieveTelemetryData()*, *blockCommunication()*, *sendJammingAlert()*, *jamCommunication()*, *unblockCommunication()*, and *close()*.

The *JammingDefender* class has a private detector attribute of type *JammingDetector*, and it has two public methods, *start()* and *stop()*. The class diagram also includes classes for *dgramSocket*, *mqttClient*, and *child_process*, which are related to the *JammingDetector* class. The *dgramSocket* and *mqttClient* classes have public methods for communication with the server and broker, respectively. The *child_process* class has a public method for executing shell commands.

The relationships between classes are indicated by the arrows in the diagram. The *JammingDetector* class has a composition relationship with the *dgramSocket* and *mqttClient* classes, indicating that it owns instances of these classes. It also has a dependency relationship with the *child_process* class, indicating that it relies on this class to execute shell commands. The *JammingDetector* class is associated with the *JammingDefender* class through the detector attribute, which holds a reference to an instance of the *JammingDetector* class. The *JammingDefender* class has an aggregation relationship with the *JammingDetector* class, indicating that it holds a reference to an instance of the *JammingDetector* class. Finally, the *JammingDefender* class has an association relationship with the *JammingDetector* class, indicating that it interacts with the *JammingDetector* class to start and stop the jamming detection process.

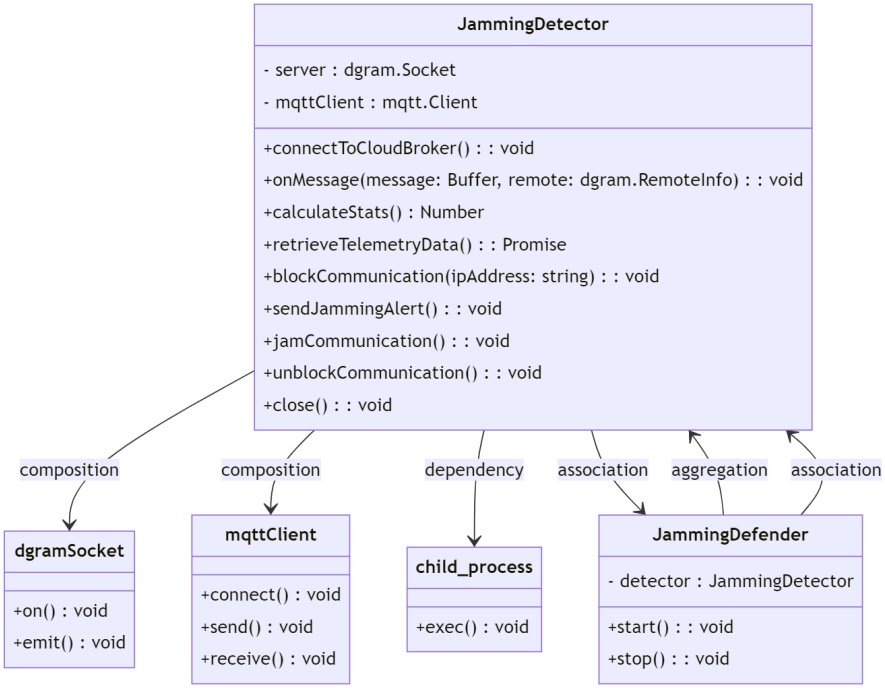

Figure 25. Jamming class diagram.

The defense mechanism against jamming attacks in the class diagram consists of several steps that involve the collaboration of the *JammingDetector* and *JammingDefender* classes, as well as some related classes, such as *dgramSocket*, *mqttClient*, and *child_process*. Here is a more detailed and comprehensive explanation of the defense mechanism:

1. **Initialization and starting the defense process**: The defense mechanism is initialized and started by the *JammingDefender* class, which creates an instance of the *JammingDetector* class. When the *start()* method of the *JammingDefender* class is called, it sets up the *JammingDetector* instance to begin the detection and defense process against jamming attacks.
2. **Connecting to the cloud broker**: The *JammingDetector* class establishes a connection to the cloud broker by invoking the *connectToCloudBroker()* method. This connection enables the *JammingDetector* to send and receive messages from an *MQTT broker*, which is essential for monitoring the communication network and detecting jamming attacks.

3. **Listening for incoming messages**: The *JammingDetector* class listens for incoming messages from the *MQTT* broker by implementing the *onMessage()* method. This method processes the received messages and extracts the relevant telemetry data, which are used to analyze the communication network's health and identify any anomalies that may indicate a jamming attack.
4. **Analyzing telemetry data**: The *JammingDetector* class periodically analyzes the telemetry data by calling the *calculateStats()* method. This method computes various statistics from the data, such as packet loss, latency, and signal strength, to identify potential jamming attacks. If the calculated statistics show a significant deviation from the expected values, it may suggest the presence of a jamming attack on the communication network.
5. **Detecting jamming patterns**: In addition to analyzing telemetry data, the *JammingDetector* class also uses the *detectGrayholePattern(dataQueue)* method to detect patterns in the data that may be indicative of a jamming attack. This method looks for specific patterns, such as a sudden increase in packet loss or a significant drop in signal strength, to determine whether a jamming attack is in progress.
6. **Responding to jamming attacks**: If a jamming attack is detected, the JammingDetector class takes several actions to mitigate the attack and protect the communication network. It blocks the communication of the suspected jammer by calling the *blockCommunication()* method, sends a jamming alert to the relevant authorities using the *sendJammingAlert()* method, and jams the attacker's communication by invoking the *jamCommunication()* method. These actions help minimize the impact of the jamming attack on the network and allow the system to continue operating despite the attack.
7. **Restoring normal communication**: Once the jamming attack has been mitigated, the *JammingDetector* class can restore normal communication by calling the *unblockCommunication()* method. This method unblocks the previously blocked communication, allowing the network to resume its normal operation.
8. **Stopping the defense process**: The *JammingDefender* class can stop the defense process by calling its *stop()* method. This method stops the *JammingDetector* instance, which in turn stops monitoring the communication network and releases the resources associated with it.
9. **Cleanup and closing connections**: The *JammingDetector* class has a *close()* method that can be used to close the server and *MQTT* client connections and release any resources associated with them. This method ensures proper cleanup of the *JammingDetector* instance when the defense mechanism is no longer needed.

3.13. Eavesdropping

3.13.1. Definition

An eavesdropping attack is a type of security threat in which an attacker secretly intercepts and eavesdrops on a private communication. This type of attack can take several forms, including wiretapping phone calls, intercepting wireless network communications, and even eavesdropping on face-to-face conversations. Often, the goal of an eavesdropping attack is to obtain sensitive information or access to confidential data. Often, it is difficult to identify eavesdropping attacks since they occur without the parties' knowledge. Utilizing encrypted messaging or secure voice-over-IP (VoIP) services can lessen the risk of eavesdropping attacks. In addition, the use of end-to-end encryption protects against efforts to eavesdrop by guaranteeing that only the intended recipients can read or access the sent data.

In the context of a UAV–cloud system, an eavesdropping attack aims to intercept and decode confidential information exchanged between the unmanned aerial vehicles (UAVs) and the cloud infrastructure. The attacker's objective is to maximize the amount of successfully intercepted and decoded information while overcoming the challenges posed by encryption mechanisms, secure communication protocols, physical security controls, and access restrictions. Here is a comprehensive narrative of the mathematical

problem formulation from the attacker's perspective. The attacker's primary objective is to maximize the amount of intercepted and decoded information (*I**). This objective can be represented mathematically as a function, $I* = g(I, U, C, E_{inv}, P_{inv})$, where I represents the information exchanged between a set of UAVs (U) and the cloud infrastructure (C). E_{inv} and P_{inv} denote the attacker's knowledge of the inverse functions for encryption and secure communication protocols, respectively.

In order to successfully carry out the eavesdropping attack, the attacker must overcome several constraints: (i) the attacker must intercept the communication between the UAVs and the cloud infrastructure. This constraint can be represented mathematically as a function, $f(U, C) = I$, which indicates that the attacker's ability to intercept the communication depends on the properties of the UAVs and the cloud infrastructure. (ii) The attacker must have the knowledge or capability to break the encryption and secure communication protocols used in the UAV–cloud system. This constraint can be represented by two separate mathematical equations, $E_{inv}(E(I)) = I$ and $P_{inv}(P(I)) = I$, where $E(I)$ and $P(I)$ are the encrypted and secure communication protocol-applied versions of the information, respectively. (iii) The attacker must also overcome the physical security controls and access restrictions implemented within the UAV–cloud system. These constraints can be represented by a set of mathematical equations, $S_{inv}(S(U)) = U$ and $S_{inv}(S(C)) = C$, which denote that the attacker needs to bypass the security measures protecting the UAVs and the cloud infrastructure. Additionally, $R_{inv}(R(U, C)) = U \times C$ represents the attacker's ability to overcome access restrictions placed on the interactions between UAVs and the cloud infrastructure.

Usage of physical security controls, such as securing doors and shielding computer equipment, and installation of access restrictions, such as user identification and authorization, can also help against eavesdropping attacks. In addition, it is vital to regularly assess and revise security policies and processes to ensure they account for the most recent risks and dangers [31].

3.13.2. Sequence Diagram

As shown in Figure 26, in this sequence diagram, the attacker starts the process of network eavesdropping. They intercept and store data transmitted between UAVs and the cloud. Next, the attacker examines the intercepted traffic, identifies user data, and requests them from the cloud. They intercept and keep the user information. The attacker then uses the intercepted data to transmit malicious orders to the drones across the network, forcing the cloud to execute and respond to those commands. In the end, the attacker is successful in attacking the system.

3.13.3. Defense Class Diagram

As shown in Figure 27, the class diagram describes a program that includes an EavesdroppingDefender class, which uses other classes such as *mqttClient*, *mongodb*, and *crypto* to perform its tasks. The *EavesdroppingDefender* class has three private instance variables: *uavClient*, *cloudClient*, and *collection*, which are instances of the *mqttClient* and *mongodbCollection* classes, respectively. The class includes a constructor method that creates instances of the *mqttClient* class, subscribes to topics, and sets up message handlers. The *onUAVMessage* and *onCloudMessage* methods handle incoming messages from the UAV and cloud brokers, respectively. The *encryptMessage* and *decryptMessage* methods use the *crypto* class to perform AES-256-CBC encryption and decryption. The *storeDecryptedMessage* and *storeEncryptedMessage* methods store messages in the *MongoDB* database. The close method unsubscribes from topics and ends the *MQTT* connections.

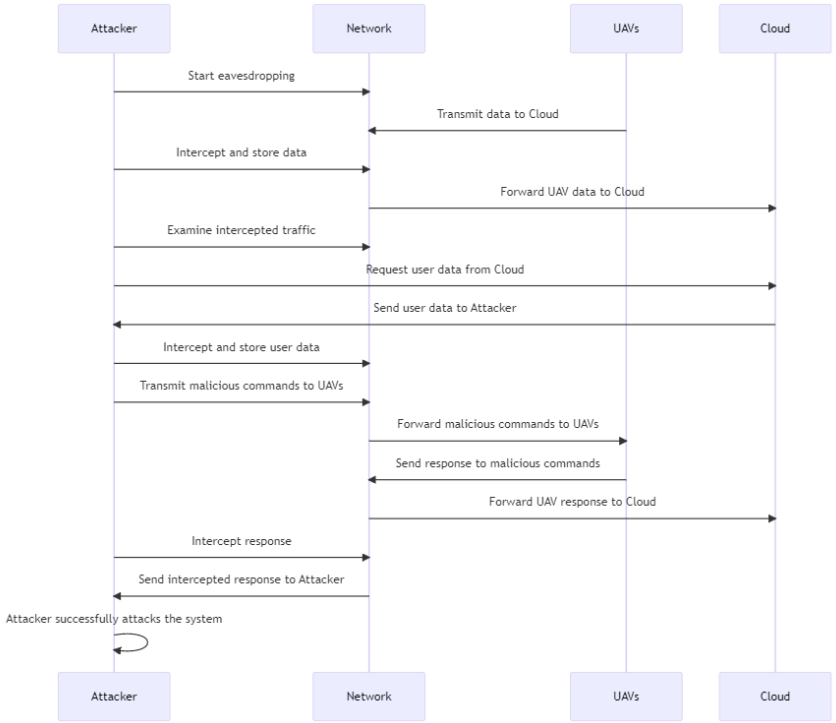

Figure 26. Eavesdropping sequence diagram.

The *mqttClient* class is used to connect to *MQTT* brokers, subscribe to and publish messages on topics, and handle incoming messages. The *mqttClientOptions* class is used to set options such as a username and password for connecting to a broker. The *crypto* class provides methods for generating random bytes, creating ciphers and deciphers for encryption and decryption. The *mongodb* class provides a method for inserting documents into the MongoDB database and a db method that returns an instance of the *mongodbCollection* class. The arrows in the diagram indicate the relationships between the classes. The *EavesdroppingDefender* class uses instances of the *mqttClient*, *mongodb*, and *crypto* classes to perform its tasks. The *mqttClient* class has an instance of the *mqttClientOptions* class. The *mongodb* class connects to the *MongoDB* database and returns an instance of the *mongodbCollection* class.

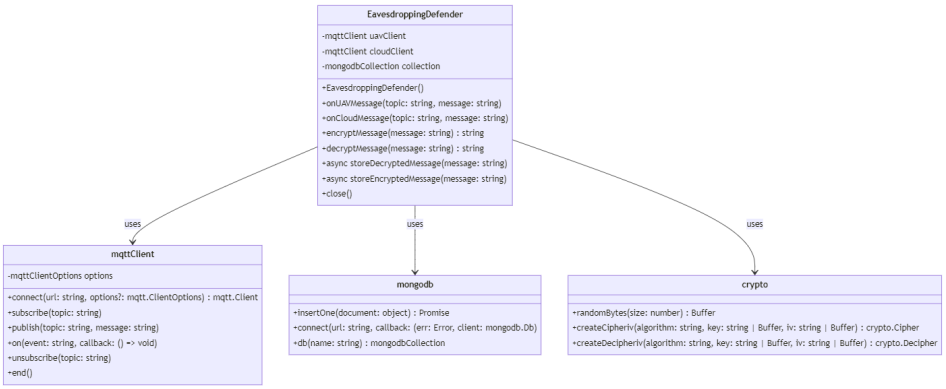

Figure 27. Eavesdropping defense class diagram.

The class diagram depicts a defense mechanism against eavesdropping attacks within a UAV–cloud system, using encryption, secure communication, and data storage techniques. The primary class managing the secure communication is the *EavesdroppingDefender* class, which collaborates with instances of *mqttClient*, *mongodb*, and *crypto* classes to safeguard the data transmission and storage. Below is the implementation process:

1. **Initialization**: The *EavesdroppingDefender* class initiates the defense process by creating instances of *mqttClient* and subscribing to appropriate topics. The class sets up message handlers through *onUAVMessage* and *onCloudMessage* methods, which are responsible for managing incoming messages from the UAV and the cloud, respectively.
2. **Encryption**: Upon receiving a message from the UAV, the *EavesdroppingDefender* class uses the *encryptMessage* method to apply *AES-256-CBC* encryption, provided by the *crypto* class, to the message. The encrypted message is then securely transmitted to the cloud through *mqttClient*, ensuring that potential eavesdropping attackers cannot decipher the message content.
3. **Decryption**: Similarly, when the cloud sends a message, the *EavesdroppingDefender* class intercepts it, decrypts the message using the *decryptMessage* method, and securely stores the decrypted data in the *MongoDB* database with the *storeDecryptedMessage* method.
4. **Storing**: Moreover, the *EavesdroppingDefender* class has *storeEncryptedMessage* and *close* methods. The *storeEncryptedMessage* method allows for storing encrypted messages in the *MongoDB* database, adding an extra layer of security. The close method takes care of unsubscribing from topics and terminating the *MQTT* connections, reducing the likelihood of eavesdropping attacks during inactive periods.
5. **Handling incoming messages**: The *mqttClient* class facilitates connecting to *MQTT* brokers and handling incoming messages, while the *mqttClientOptions* class sets up necessary options such as username and password for secure broker connections. The *crypto* class offers various methods for generating random bytes and creating ciphers and deciphers for encryption and decryption. The *mongodb* class enables the insertion of documents into the *MongoDB* database and returns an instance of the *mongodbCollection* class for additional database operations.

3.14. Poor Link Quality and High Latency

3.14.1. Definition

Poor link quality and excessive latency are two prevalent network connection concerns. Poor connection quality can be caused by a number of causes, including physical distance between devices, interference from other devices, environmental variables such as walls or obstructions, or network hardware or software problems. Low link quality can lead to sluggish data transfer, lost connections, and other performance problems.

In contrast, high latency refers to delays in the time it takes for data to be transferred and received across network devices. This delay may be caused by physical distance, network congestion, or other technical or software difficulties with the network. Excessive latency can lead to sluggish reaction times, poor network performance, and other issues that might negatively influence the user experience.

To reduce these challenges, it is essential to identify the underlying source of the problem and take the necessary measures to remedy it. If the problem is caused by interference or environmental conditions, for instance, it may be necessary to relocate equipment or modify the network configuration in order to improve the connection. If the problem is caused by network congestion, it may be essential to increase network bandwidth or employ traffic management techniques to prioritize specific types of traffic. Frequent monitoring and performance testing may also assist in identifying and resolving any issues before they have an effect on the user experience [32].

3.14.2. Sequence Diagram

As shown in Figure 28, the attacker sends a large quantity of data to *UAV1*, causing the data queue to become full. The *UAV1* then transmits the data to the cloud, which confirms its reception. The cloud then transmits the data to *UAV2*, which confirms receipt of the data.

Due to the large volume of data being communicated, however, the communication between *UAV1* and the cloud is extremely delayed. The network reacts to *UAV1's* request

for extra bandwidth by raising the capacity. With the increased bandwidth, latency is reduced, and *UAV1* is able to transfer data to the cloud and *UAV2* more effectively.

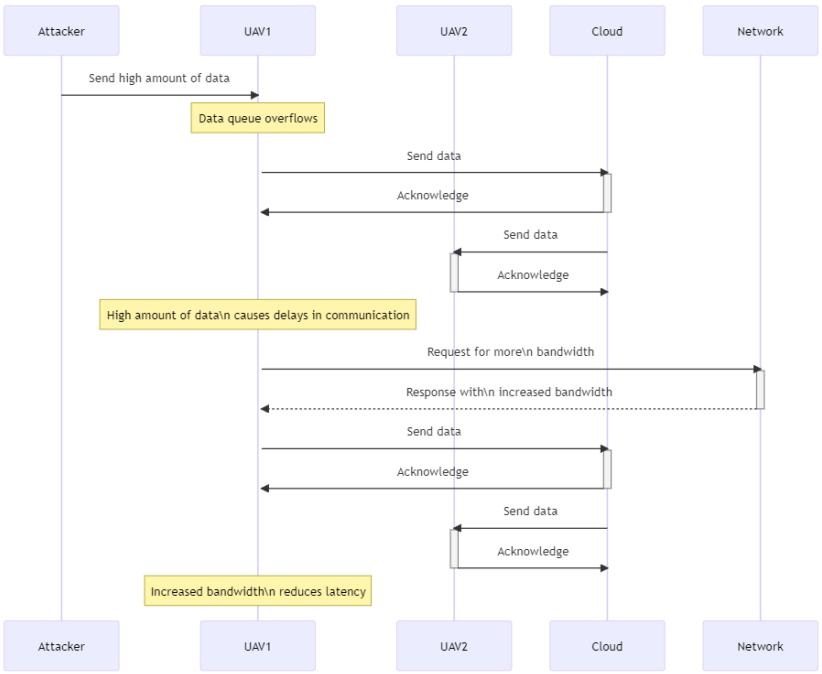

Figure 28. High latency due to attacker.

3.14.3. Defense Class Diagram

The class diagram in Figure 29 depicts a system for monitoring and maintaining link quality between an unmanned aerial vehicle (UAV) and a cloud-based server. The system is designed to handle message transmission and reception, prioritize messages based on their latency, compress messages to reduce bandwidth usage, and store latency data in a *MongoDB* database.

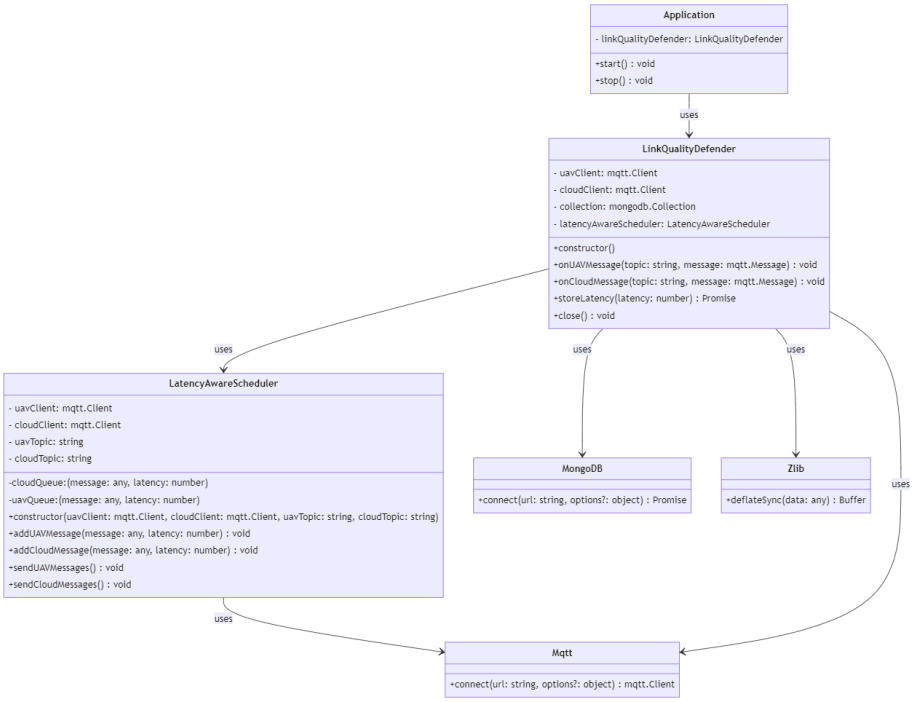

Figure 29. Latency and link quality defender.

The defense mechanisms in the class diagram focus on monitoring and maintaining link quality between an unmanned aerial vehicle (UAV) and a cloud-based server by managing message latency, preserving communication quality, and ensuring system reliability. Below is the implementation process explanation of the class diagram:

1. **Latency-based message scheduling and prioritization**: The *LatencyAwareScheduler* class plays a crucial role in managing the communication between the UAV and the cloud by scheduling messages based on their latency. It maintains separate message queues for both the UAV and the cloud to prioritize messages with lower latency. By performing this, the system ensures that critical information is transmitted in a timely manner, even when faced with network congestion or other potential issues affecting communication quality.
2. **Continuous monitoring and assessment of link quality**: The *LinkQualityDefender* class is responsible for handling incoming messages from the UAV and cloud, managing *MQTT* clients, and storing latency data in a *MongoDB* database. By continuously monitoring the latency of messages and the overall quality of the communication link, the system can detect potential communication issues, such as network congestion, interference, or even attacks targeting the link. As a result, the system can respond proactively to maintain or improve the link quality by taking appropriate actions, such as adjusting message priorities, rescheduling transmissions, or even triggering alerts.
3. **Efficient message compression for improved transmission**: The *Zlib* class is used by the *LinkQualityDefender* class to compress messages before transmission. By compressing the data, the system effectively reduces the size of messages, improving transmission efficiency and reducing the chances of communication delays or packet loss. This efficiency is particularly important for maintaining high-quality communication in scenarios where network bandwidth is limited or under attack.
4. **Resilient and reliable communication through MQTT**: The *Mqtt* class represents the *MQTT* client library used by both the *LinkQualityDefender* and *LatencyAwareScheduler* classes. *MQTT* is a lightweight messaging protocol designed specifically for situations where low-bandwidth, high-latency, or unreliable networks are expected. By employing *MQTT* as the communication protocol, the system ensures that the communication link between the UAV and the cloud remains robust and reliable, even in challenging network conditions or under attack.
5. **Centralized control and management of defense mechanisms**: The Application class provides a central interface for starting and stopping the *LinkQualityDefender* instance. This centralized control simplifies the management and monitoring of the system's performance and allows operators to easily enable or disable the defense mechanisms as needed. This adaptability is crucial in ensuring that the system can respond effectively to a wide range of threats or communication challenges.

3.15. Man-In-The-Middle Attack

3.15.1. Definition

An attacker intercepts and modifies communication between two parties in a man-in-the-middle (MITM) cyberattack. In a conventional man-in-the-middle (MITM) assault, the attacker inserts themselves between the two parties and relays messages, making it look as though the communication is coming straight from the targeted party. The attacker can then use this position to obtain access to sensitive information or influence the connection. They may, for instance, steal login passwords, reroute financial transactions to their own accounts, or alter the substance of transmitted communications. MITM attacks can be executed in a number of ways, including by intercepting communication over an unprotected wireless network, by breaking into network infrastructure or servers, or by utilizing phishing or other social engineering techniques to deceive users into downloading malware on their devices [33].

3.15.2. Sequence Diagram

As shown in Figure 30, the attacker is employing a man-in-the-middle assault to intercept communications between UAVs and the ground control station. The adversary sends a faked beacon message to both UAVs, which causes them to associate with the adversary's false access point. The attacker then sends a faked probe request and obtains a probe response. The attacker sends a faked probe response to both UAVs, which causes them to associate with his phony access point.

After the UAVs are linked with the adversary's bogus access point, the adversary intercepts the telemetry data transmitted from *UAV1* to the ground control station. The attacker then transmits a faked order to the ground control station, which is subsequently performed by the station and transmitted to *UAV1*. *UAV1* then carries out the instruction.

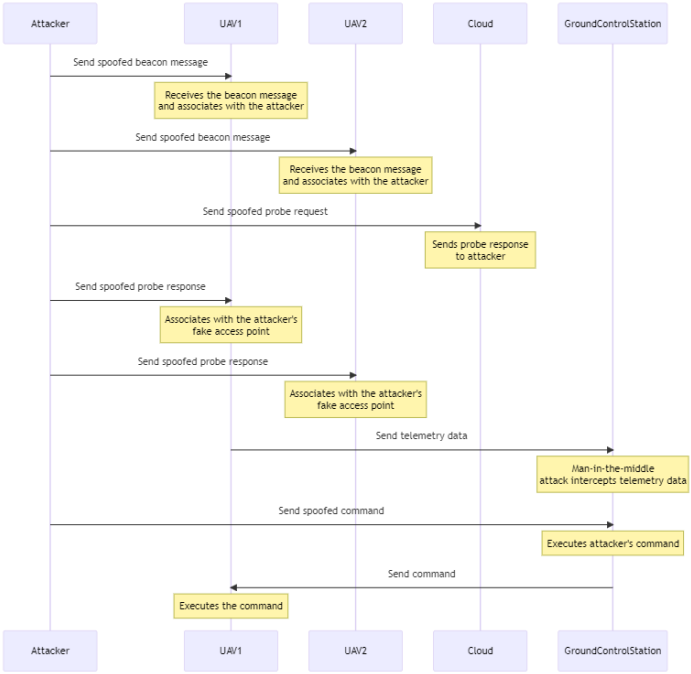

Figure 30. Man-in-the-middle attack.

3.15.3. Defense Class Diagram

As shown in Figure 31, the class diagram illustrates the connection between the *Defender*, *mqtt.Client*, and *mongodb.Collection* classes. The *Defender* class is the primary class that represents an object's defensive capabilities. It contains the *uavClient*, *cloudClient*, and *collection* instance variables. These instances of the *mqtt.Client* and *mongodb.Collection* classes are used to connect to *MQTT brokers* and the *MongoDB* database, respectively. This class defines the *MAX LATENCY*, *HMAC SECRET*, and *IV LENGTH constants*.

This class also represents the *MQTT* client that connects to *MQTT* brokers. *subscribe()*, *publish()*, *on()*, *unsubscribe()*, and end are its five public methods.These methods are used, respectively, to subscribe to topics, publish messages, add event listeners, unsubscribe from topics, and close the connection to the broker. In support of amore comprehensive defensive mechanism, we also represent the *MongoDB* collection used to hold latency data. It possesses a single public method, *insertOne()*, which is used to add a document to the collection.

Many public methods of the *Defender* class describe the behavior of the object. The *constructor()* function initializes and subscribes to the corresponding topics for the *uavClient*, *cloudClient*, and collection instances. The *onUAVMessage()* and *onCloudMessage()* methods are event listeners for UAV and cloud messages, respectively. They encrypt and decrypt messages, validate *HMAC* signatures, distribute messages, and store latency statistics in the *MongoDB* collection. Message encryption, *HMAC* signature formation and verification, and

decryption are handled by the *encryptMessage()*, *createHmac()*, *verifyHmac()*, and *decryptMessage()* functions. The *storeLatency()* function stores latency data in a MongoDB collection. The *close()* function is used to unsubscribe from topics and disconnect from *MQTT* brokers.

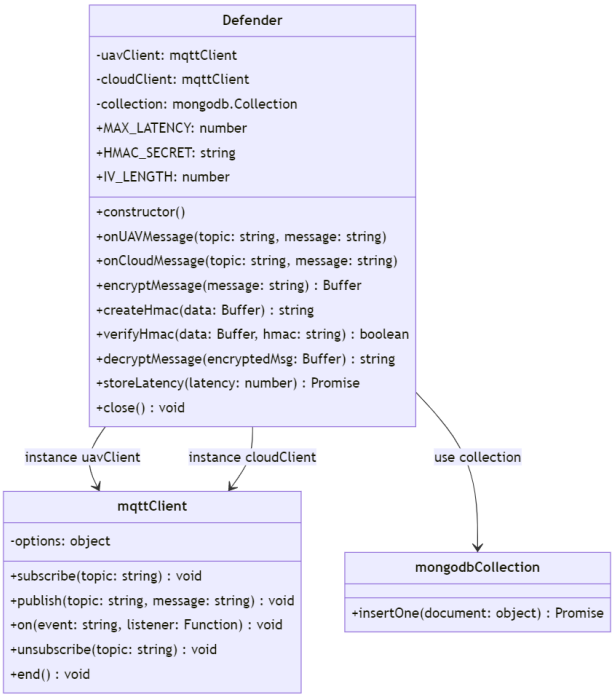

Figure 31. MITM defense class diagram.

The defense mechanisms are designed to protect the communication between the UAV and the cloud server from unauthorized interception and tampering. Here is an explanation of the defense scenarios based on the class diagram:

1. **Initializing the *Defender* class**: When the system starts, the *Defender* class is initialized, and its constructor method is responsible for creating instances of the *mqtt.Client* and *mongodb.Collection* classes. The instances are used to connect to *MQTT* brokers and the *MongoDB* database, respectively.
2. **Secure MQTT communication setup**: The *mqtt.Client* class establishes secure connections to the *MQTT* brokers using security features such as TLS/SSL and username/password authentication. This ensures that the communication channel between the UAV and the cloud server is protected against unauthorized access and MITM attacks.
3. **Subscribing to topics and handling messages**: Once secure connections are established, the *Defender* class subscribes to the relevant topics using the *mqtt.Client* instances. The *onUAVMessage()* and *onCloudMessage()* methods are set as event listeners for incoming messages from the UAV and cloud brokers, respectively.
4. **Encrypting and decrypting messages**: When a message is sent between the UAV and the cloud server, the *Defender* class uses the *encryptMessage()* method to encrypt the message using strong encryption techniques (e.g., AES-256-CBC). This ensures that the contents of the message are protected and unreadable to potential attackers. Upon receiving a message, the *Defender* class uses the *decryptMessage()* method to decrypt and process the message securely.
5. **Ensuring message integrity and authenticity**: Before sending an encrypted message, the *Defender* class generates an HMAC signature using the *createHmac()* method. The HMAC signature is added to the message to ensure its integrity and authenticity. When a message is received, the *Defender* class uses the *verifyHmac()* method to verify

the HMAC signature, confirming that the message has not been tampered with and originates from a trusted source.

6. **Handling and distributing messages**: Once the message is decrypted and its HMAC signature is verified, the *Defender* class processes the message and distributes it to the appropriate components in the system.
7. **Monitoring and storing latency data**: Throughout the communication process, the *Defender* class uses the *storeLatency()* method to collect and store latency data in the *MongoDB* collection. This continuous monitoring of latency helps detect potential communication anomalies or suspicious patterns that may indicate a MITM attack or other network issues.
8. **Closing the MQTT connections**: When the communication process is completed, or the system needs to be shut down, the *Defender* class uses the *close()* method to unsubscribe from topics and disconnect from the *MQTT* brokers securely.

3.16. Modification Attack

3.16.1. Definition

A modification attack is a sort of cyberattack in which an attacker modifies or corrupts data during transmission over a network. This can be accomplished by intercepting and altering data packets or by inserting malicious malware into a network. Often, the objective of a modification attack is to disrupt or undermine the integrity of sent data and obtain unauthorized access to sensitive data. This is especially dangerous when the manipulated data are utilized for vital purposes, such as in financial transactions or industrial control systems. Modification attacks can be executed in a variety of methods, such as via intercepting communication across unprotected networks, exploiting weaknesses in network architecture or software, or employing social engineering techniques to persuade users into downloading malware on their devices [34] .

3.16.2. Sequence Diagram

As shown in Figure 32, this sequence diagram illustrates a modification assault mechanism scenario in the context of a multi-UAV–cloud system. The graphic depicts two possible scenarios: when the hacker alters the communication and when he intercepts it. In the first scenario, the hacker transmits a request for a message to *UAV1*. *UAV1* requests a message from the cloud, and the cloud answers with the requested message. The message is returned to *UAV1* and is subsequently forwarded to the hacker. The hacker edits the message and transmits it to *UAV1* again. The amended message is transmitted by *UAV1* and acknowledged by the cloud. The acknowledgment is then transmitted back to *UAV1* before being relayed to the hacker.

In the second scenario, the hacker intercepts the transmission by transmitting a request for a message to *UAV1*. *UAV1* requests a message from the cloud, and the cloud answers with the requested message. The message is returned to *UAV1* and is subsequently forwarded to the hacker. Moreover, the hacker sends a communication request to *UAV2*. *UAV2* transmits the message request to the cloud, which then returns the message. The message is returned to *UAV2* and is subsequently forwarded to the hacker. The hacker edits the message and transmits it to *UAV1* again. The amended message is transmitted by *UAV1* and acknowledged by the cloud. The acknowledgment is then transmitted back to *UAV1* before being relayed to the hacker.

3.16.3. Defense Class Diagram

As shown in Figure 33, the class diagram is a visual representation of the classes, their attributes, and their relationships in the code provided. In this case, the class diagram describes the class structure of the *ModificationDefender* module, as well as the relationships between this class and other modules it uses. The *ModificationDefender* class is the central class of the module; it handles the main functionality of detecting and preventing modification attacks on data transmitted over a UAV–cloud network. This class has several private

properties, including *uavClient* and *cloudClient*, which are instances of the *mqtt.Client* class used to connect to the *MQTT* brokers on the UAV and cloud sides, respectively. It also has a collection property, which is an instance of the *mongodb.Collection* class used to store latency data in the *MongoDB* database. Additionally, the *ModificationDefender* class has a private *lastHash* property, which is used to store the hash of the last received message to check for modification attacks.

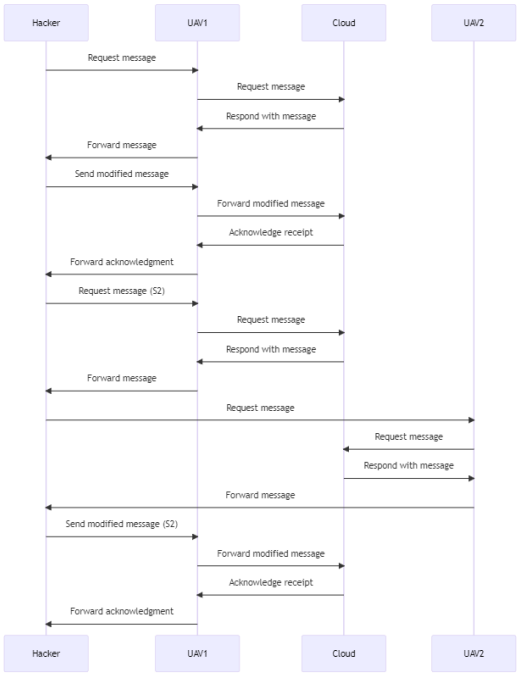

Figure 32. Sequence diagram of modification attack.

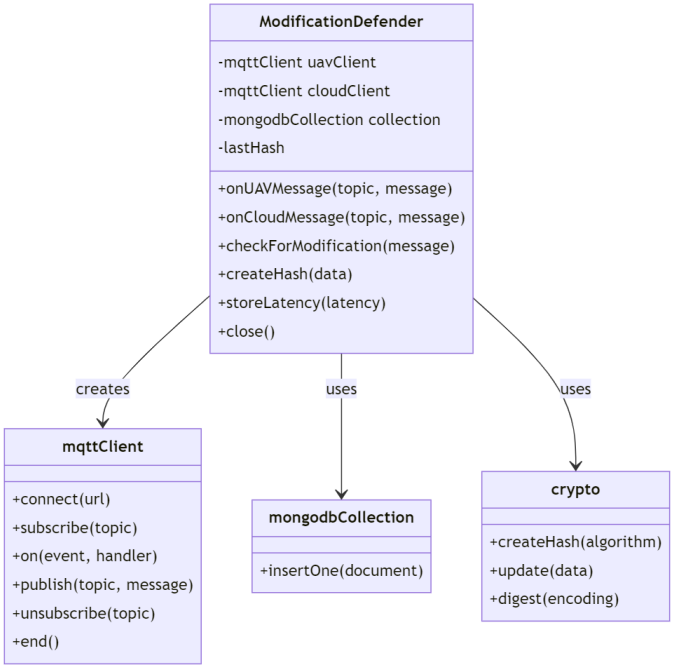

Figure 33. Modification defender class diagram.

The *ModificationDefender* class has several public methods, including *onUAVMessage*, *onCloudMessage*, *checkForModification*, *createHash*, *storeLatency*, and *close*. These methods are used to handle incoming messages from the UAV and cloud sides, check for message

modification, create message hashes, store latency data, and close the *MQTT* connections when needed. The class diagram also includes three external modules that are used by the *ModificationDefender* class: *mqtt*, *mongodb*, and *crypto*. The mqtt module provides the Client class, which is used to establish *MQTT* connections with the brokers. The *mongodb* module provides the Collection class, which is used to store latency data in the MongoDB database. The *crypto* module provides various *cryptographic* functions used by the *ModificationDefender* class to create and verify *HMAC* signatures, encrypt and decrypt messages, and create message hashes.

The defense mechanism scenario for the modification attack can be explained sequentially as follows:

1. **Initialization**: The system initializes the defense mechanism by creating an instance of the *ModificationDefender* class. During this process, the system establishes connections to the *MQTT* brokers for UAV and cloud communication using the *uavClient* and *cloudClient* properties. Additionally, the system connects to the *MongoDB* database using the collection property to store latency data.
2. **Subscription**: Upon instantiating, the *ModificationDefender* class subscribes to the relevant *MQTT* topics for both the UAV and the cloud. The class sets up event listeners for incoming messages from the UAV and the cloud by implementing the *onUAVMessage* and *onCloudMessage* methods, respectively.
3. **Event listener**: When a message is received from the UAV or the cloud, the corresponding event listener (*onUAVMessage* for UAV messages or *onCloudMessage* for cloud messages) is triggered. These methods decrypt the incoming messages, verify their HMAC signatures, and pass the decrypted messages to the *checkForModification* method for further examination.
4. **Modification checking**: The *checkForModification* method compares the hash of the incoming message with the stored *lastHash* property to determine if the message has been modified during transmission. If the *checkForModification* method detects a modification, it initiates an appropriate response. This might include discarding the altered message, notifying the system administrator of a potential breach, or implementing additional countermeasures to protect the system from future attacks. Otherwise, if the method does not detect any modifications, the message is considered valid, and the process proceeds to the next step.
5. **Hash creation**: Upon validation, the *createHash* method generates a new hash for the message. This new hash is then stored in the *lastHash* property, overwriting the previous value for subsequent message comparisons.
6. **Latency logging**: The *storeLatency* method logs latency data related to the message transmission in the *MongoDB* database, utilizing the collection property. This information can be analyzed later to identify potential bottlenecks in the system or to optimize the performance of the UAV–cloud communication system.
7. **Monitoring and validating**: The *ModificationDefender* class continuously monitors and validates messages transmitted between the UAV and the cloud, ensuring the integrity of the communication system and detecting potential modification attacks.
8. **Encryption and unsubscribe**: The encryption and HMAC signature generation process is conducted before transmitting the message to its destination, ensuring the message's integrity and authenticity during transmission. When the defense mechanism is no longer needed or must be terminated, then the close method is called. This method unsubscribes the ModificationDefender instance from the MQTT topics and disconnects from the MQTT brokers, effectively ending the message monitoring and validation process.

3.17. Replay Attack

3.17.1. Definition

Let *P(success)* denote the probability that an attacker is able to successfully intercept and replay the communication between a UAV and the cloud server. Then, we can model

this probability as $P(success) = P(interception) * P(replay) * P(accept)$, where $P(interception)$ is the probability that the attacker is able to intercept the communication, $P(replay)$ is the probability that the attacker is able to replay the intercepted communication, and $P(accept)$ is the probability that the cloud server accepts the replayed communication.

To mitigate the risk of a replay attack, the security developer can focus on reducing each of these probabilities. For example, encryption and authentication protocols to reduce the probability of interception and replay can be used. In addition, strict validation checks on the cloud server to reduce the probability of acceptance of a replayed communication also can be implemented. In addition, we can also consider the potential consequences of a successful replay attack by mathematical formulation: Let C denote the potential cost or harm caused by a successful attack. Then, a model of the expected cost of a replay attack can be written as $E(C) = P(success) * C$. This formulation helps us to quantify the potential impact of a replay attack and can guide our efforts to implement effective security measures to prevent such attacks [35].

3.17.2. Sequence Diagram

As depicted in Figure 34, there are four participants depicted in the diagram: a hacker, a cloud, and two UAVs (*UAV1* and *UAV2*). The graphic begins with a hacker intercepting a communication transmitted to the cloud by *UAV1*. The communication contains data that the hacker intends to transmit to the cloud again. The hacker sends a request for a message to both *UAV2* and the cloud. After receiving the message request, the cloud transmits the message to *UAV2*. The message is received by *UAV2* and is transmitted back to the cloud. The cloud returns the message to *UAV1*, which then forwards it to the hacker.

The hacker has now successfully transmitted the message to the cloud. The message is a repetition of the original message, not a fresh one. This can result in a variety of security risks, including unwanted access and data corruption. Of importance in this situation is a protection system that inhibits replay assaults. Using timestamping or nonce values to verify that each communication is unique and cannot be replayed is one possible method.

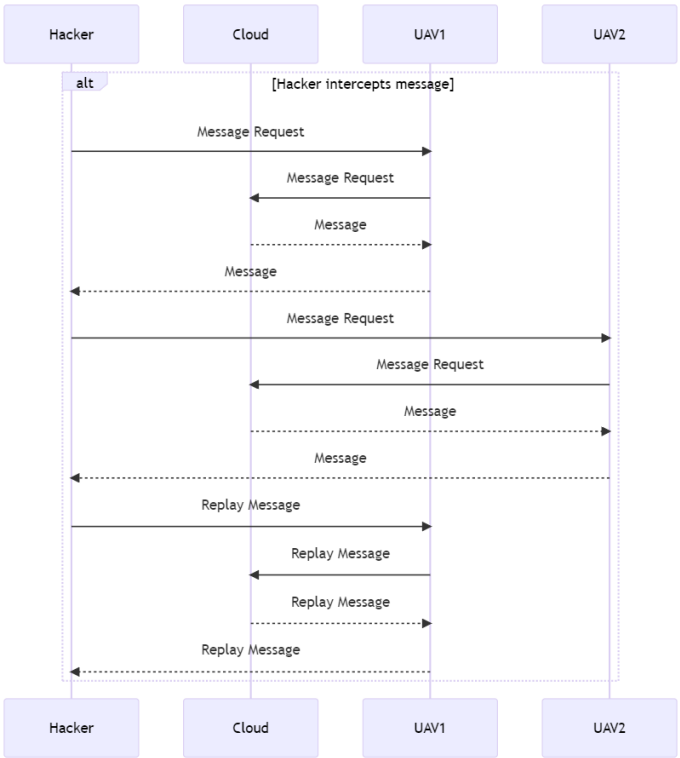

Figure 34. Sequence diagram of replay attack.

3.17.3. Defense Class Diagram

As shown in Figure 35, the *ReplayDefender* class uses instances of the *mqtt.Client* and *mongodb.Collection* classes to handle the communication with the *MQTT* brokers and the *MongoDB* database, respectively. The *uavClient* and *cloudClient* fields in the *ReplayDefender* class are instances of the *mqtt.Client* class, and the collection field is an instance of the *mongodb.Collection* class. Furthermore, the *ReplayDefender* class has a number of methods that use the *mqtt.Client* and *mongodb.Collection* classes, such as *connect()*, *subscribe()*, *on()*, *insertOne()*, *findOne()*, *updateOne()*, *deleteOne()*, and *end()*. These methods allow the *ReplayDefender* class to communicate with the *MQTT* brokers and the *MongoDB* database to perform the necessary tasks to detect and prevent replay attacks.

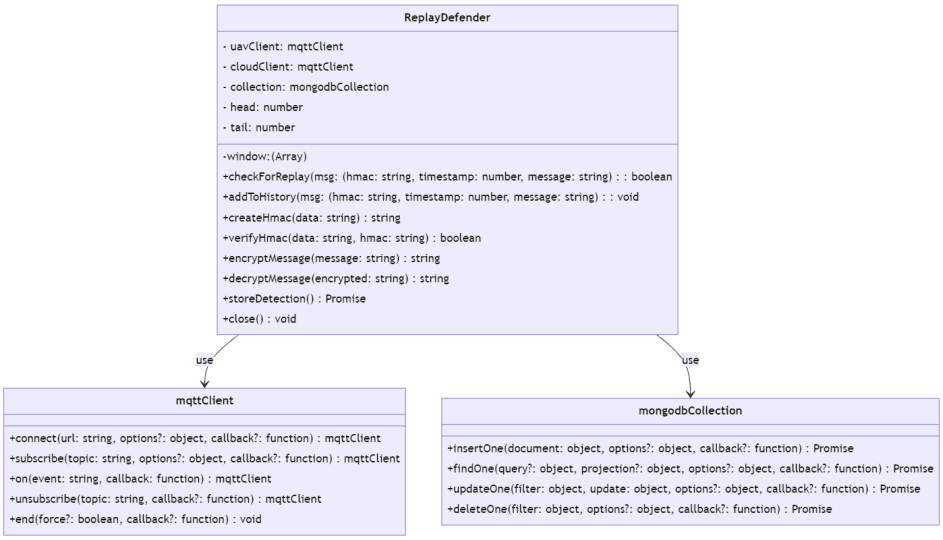

Figure 35. Replay defender class diagram.

Below is a more detailed explanation of the defense mechanism against replay attacks based on the replay defender class diagram:

1. **Initialization**: The *ReplayDefender* class is initialized, creating an instance that connects to the *MQTT* brokers for UAV and cloud communication through the *uavClient* and *cloudClient* properties. Additionally, the collection property is used to connect to the *MongoDB* database for storing and managing message data.
2. **Subscription**: The *ReplayDefender* subscribes to relevant *MQTT* topics for both the UAV and the cloud, setting up event listeners for incoming messages from both sides. This is achieved by calling the *subscribe()* method on the *uavClient* and *cloudClient* instances.
3. **Message listener**: When a message is received from the UAV or the cloud, the event listeners trigger the *on()* method. This method processes the incoming message and checks for potential replay attacks: (i) The *on()* method calls the *findOne()* method to search for the received message in the *MongoDB* database. This checks if the message has been previously recorded. (ii) If the *findOne()* method returns a result, the message has been previously recorded, indicating a replay attack. The defense mechanism can initiate an appropriate response, such as discarding the message or notifying the system administrator. (iii) If the *findOne()* method does not return any results, the message is considered new and valid. The process moves to the next step.
4. **Insert into database**: The *insertOne()* method adds the new, valid message to the *MongoDB* database. The message is stored with a timestamp or a unique identifier, which helps differentiate it from future messages and detect potential replay attacks.
5. **Monitoring**: With continuous monitoring, the *ReplayDefender* class validates messages transmitted between the UAV and the cloud. If a message is identified as a replay, the

updateOne() or *deleteOne()* methods can be employed to manage the recorded messages in the *MongoDB* database, keeping the record updated and optimized.

6. **Detects and prevents**: The *ReplayDefender* class detects and prevents replay attacks in the communication system by continuously validating messages transmitted between the UAV and the cloud, ensuring the security and reliability of the communication system.
7. **Unsubscription**: When the defense mechanism is no longer needed, the *close()* method is called. This method unsubscribes the *ReplayDefender* instance from the *MQTT* topics and disconnects from the *MQTT* brokers, effectively ending the message monitoring and validation process.

3.18. Rushing Cyberattack

3.18.1. Definition

A rushing cyber assault is a form of cyberattack in which the attacker tries to obtain access to a target's network or system before the victim can identify or respond to the attack. This sort of attack can take a variety of forms, including brute force assaults, phishing, and malware, and frequently exploits known vulnerabilities or social engineering techniques to obtain early access. A rushing cyber assault is particularly harmful since it might allow the attacker to acquire a foothold in the victim's network before being recognized, making it more challenging to limit and remedy the attack.

3.18.2. Sequence Diagram

As shown in Figure 36, the sequence diagram depicts the processes of a cyber assault in a multi-UAV–cloud environment. A hacker, a cloud, two UAVs, a server, and the attacker are among the participants in the diagram. Beginning the procedure, the hacker registers numerous UAVs with the cloud. The cloud returns the UAV IDs in response. The attacker then connects to *UAV1*, requests data, obtains the requested data, and disconnects. This method is repeated for every UAV.

The attacker then demands target data from the server, which the server provides. The attacker then uploads malicious material to the cloud. The attacker then iterates through each UAV once again, connects to *UAV2*, and uploads fabricated data to the cloud. As soon as the server detects the target, the attacker disconnects every UAV from the cloud. This sequence diagram depicts the process for a rush cyber assault in the setting of a multi-UAV–cloud.

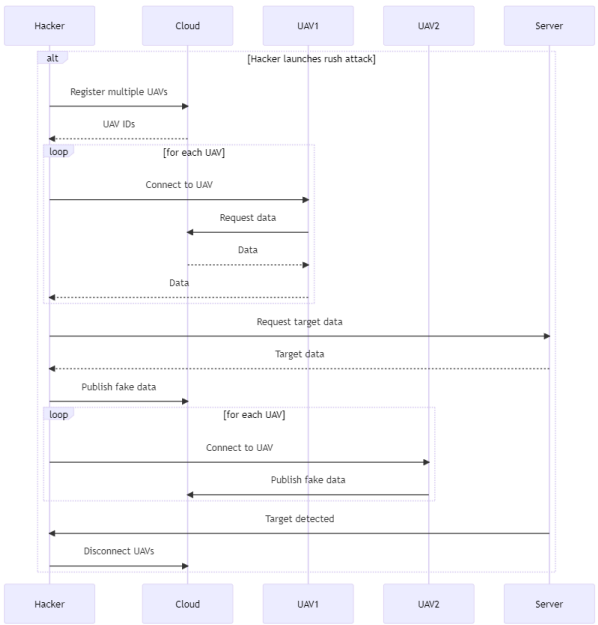

Figure 36. Sequence diagram rush attack.

3.18.3. Defense Class Diagram

As shown in Figure 37, the *RushDefender* class represents the main logic of the system. It has several private properties, including *uavClient*, *cloudClient*, *serverClient*, *uavs*, *collection*, *window*, *head*, and *tail*. It also has several public methods, including *onUAVMessage(topic, message)*, *onCloudMessage(topic, message)*, *onServerMessage(topic, message)*, *defend()*, *publishAlert(target, timestamp)*, *disconnectUavs()*, *and close()*. In addition, it has several private helper methods, including *isTargeting(uav, target)*, *encryptMessage(message)*, *createHmac(message)*, *addToHistory(message)*, *checkForReplay(message)*, *verifyHmac(encryptedMsg, hmac)*, *decryptMessage(encryptedMsg)*, and *disconnectUAV(uav)*. The *RushDefender* class uses the *Mqtt* and *MongoCollection* classes, which are represented by the dashed lines that connect the *RushDefender* class to the other classes.

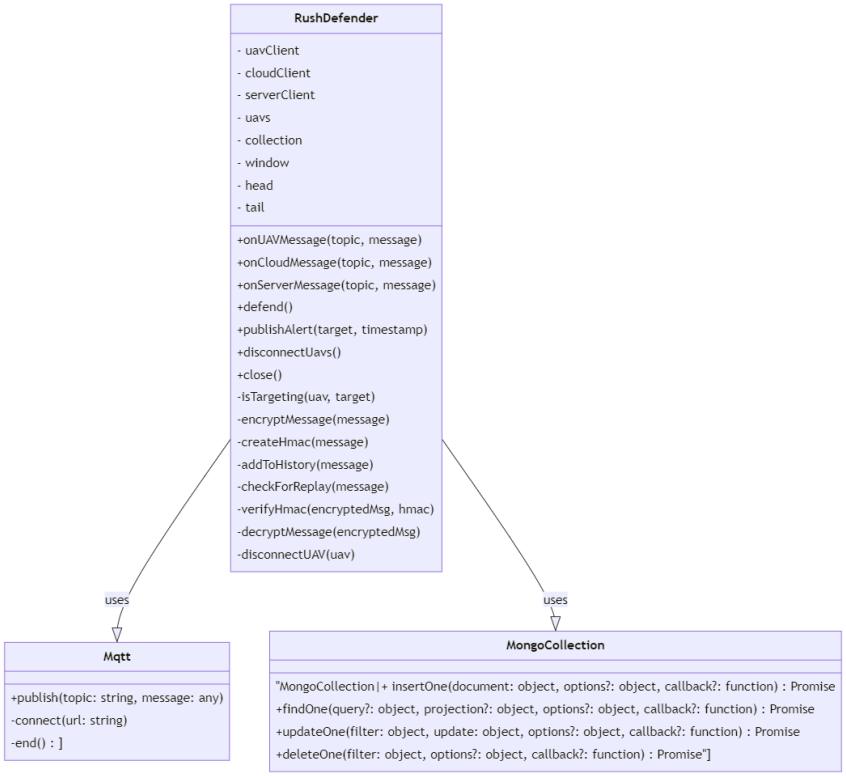

Figure 37. Rush cyber defense.

The *Mqtt* class represents the *MQTT* client that is used by the *RushDefender* class to communicate with the *MQTT* broker. It has one private method, *connect(url: string)*, and one public method, *publish(topic: string, message: any)*. The *MongoCollection* class represents a collection within a *MongoDB* database and has several methods that contain operation methods of the database. The *RushDefender* class uses the *Mqtt* class to communicate with the *MQTT* broker, and the *MongoCollection* class to interact with the *MongoDB* database.

Below is a more detailed explanation of the process in sequential order, with each step connected to the next to illustrate the defense mechanism against rush cyberattacks:

1. **Initialization**: The *RushDefender* class is initialized, which sets up instances of *MQTT* clients (*uavClient*, *cloudClient*, and *serverClient*) for communication with the UAVs, cloud, and server. It also initializes a *MongoDB* database instance (collection) to store message history and manage rush attack detection.
2. **Subscription**: The *RushDefender* class subscribes to relevant *MQTT* topics for the UAV, cloud, and server. Event listeners for incoming messages from each component are set up using *onUAVMessage*, *onCloudMessage*, and *onServerMessage* methods.
3. **Message processing**: Upon receiving a message from any component (UAV, cloud, or server), the respective event listener is triggered to process the message and detect

potential rush attacks: (i) The message is encrypted with the *encryptMessage()* method. (ii) An HMAC signature is created using the *createHmac()* method. (iii) The message is added to the history buffer with the *addToHistory()* method. The message history buffer uses a sliding window approach, managed by the properties head, tail, and window. (iv) The *checkForReplay()* method searches the message history within the sliding window to identify potential rush attacks by detecting repeated messages.

4. **Detect rush cyberattack**: If a rush attack is detected by the *checkForReplay()* method, the *defend()* method initiates an appropriate response: (i) An alert message is sent to the targeted component (UAV or cloud) using the *publishAlert()* method, which includes information about the detected attack and its timestamp. (ii) The affected UAVs are disconnected from the communication network with the *disconnectUavs()* method, preventing further rush attacks. (iii) The *disconnectUAV()* private helper method disconnects a specific UAV from the network.
5. **Monitoring**: The system continuously monitors messages exchanged between the UAV, cloud, and server to detect and defend against rush cyberattacks. Steps 3 and 4 are repeated for each incoming message to ensure ongoing protection.
6. **Unsubscription**: When the defense mechanism is no longer required, the *close()* method is called to unsubscribe the *RushDefender* instance from *MQTT* topics and disconnect the *MQTT* clients, ending the rush attack monitoring and defense process.

3.19. Selfishness Attack

3.19.1. Definition

Let us consider a scenario in which there are n drones in a UAV–cloud network, each with a certain amount of bandwidth and battery power. We want to model the behavior of a selfish drone that consumes more resources than necessary to optimize its own performance at the expense of other drones in the network. We can represent the amount of bandwidth consumed by each *UAVi* as b_i and the amount of battery power consumed by each *UAVi* as p_i. We can also represent the collective benefit of the network as B, which depends on the total amount of resources consumed by all drones in the network.

We can formulate the problem of selfish behavior in the UAV–cloud network as follows: Maximize $b_k + p_k$ *Subject to*, $b_k + \sum_{i=1,\, i\neq k}^{n} b_i \leq B$, $p_k + \sum_{i=1,\, i\neq k}^{n} p_i \leq P, b_k,\ p_k \geq 0$, where k is the index of the selfish drone, and P is the total amount of battery power available in the network. The objective function represents the selfish drone's desire to maximize its own benefit by consuming more resources than necessary. The first constraint ensures that the total amount of bandwidth consumed by all drones in the network, except for the selfish drone, does not exceed the collective benefit of the network. The second constraint ensures that the total amount of battery power consumed by all drones in the network, except for the selfish drone, does not exceed the total amount of battery power available. The third constraint ensures that the amount of resources consumed by the selfish drone is non-negative [36].

3.19.2. Sequence Diagram

As shown in Figure 38, in this multi-UAV–cloud scenario, the cloud server is responsible for assigning tasks to the unmanned aerial vehicles (UAVs) and monitoring their progress. There are two UAVs in this particular case: *UAV1*, which exhibits selfish behavior, and *UAV2*, which behaves as expected. Initially, the cloud server assigns distinct missions to each UAV. To begin their respective tasks, *UAV1* requests *Task1* from the cloud server, while *UAV2* requests *Task2*. The cloud server acknowledges these requests and delivers *Task1* to *UAV1* and *Task2* to *UAV2* accordingly.

As the UAVs start working on their tasks, they report their progress to the cloud server. *UAV1*, however, demonstrates selfish behavior. After reaching 50% progress on *Task1*, it ceases to send any further updates to the cloud server, contrary to the expectations of a normal UAV. In contrast, *UAV2* continues to send regular progress updates on *Task2* to the cloud server as it diligently works towards task completion. At some point, *UAV1* decides

to complete *Task1*. To perform this, it sends a request for *Task1* completion to the cloud server. In response, the cloud server provides the necessary completion details for *Task1* to *UAV1*. Meanwhile, *UAV2* reaches the end of *Task2* and sends a request for *Task2* completion to the cloud server. Recognizing the completion of the task, the cloud server delivers the appropriate completion details to *UAV2*.

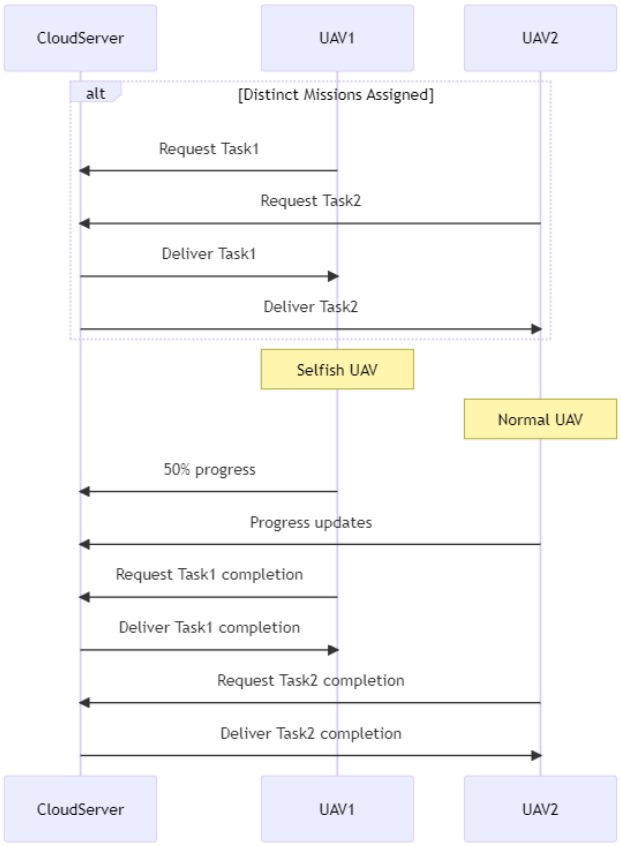

Figure 38. Selfishness sequence diagram.

3.19.3. Defense Class Diagram

As shown in Figure 39, *SelfishDefender* is the main class that represents the defender, which has *MQTT* clients for UAVs, cloud, and server, and a map of registered UAVs with their assigned tasks and progress. The class has a constructor, which initializes the *MQTT* clients and the UAV map, and methods *onUAVMessage()*, *onCloudMessage()*, *onServerMessage()*, and *defend()* that handle incoming messages, assign tasks to UAVs, and register UAVs.

The *MQTTClient* class is an abstract class that represents the *MQTT* client used by the *SelfishDefender* class to connect to the *MQTT* brokers. The *CloudClient*, *ServerClient*, and *UAVClient* classes are concrete subclasses of the *MQTTClient* class that represent the *MQTT* clients for cloud, server, and UAVs, respectively. The *UAVData* class represents the data structure that holds the task and progress of a UAV. This class is contained in the *SelfishDefender* class using a one-to-many relationship. Note that the class diagram assumes the existence of a standard *mqtt* package with *connect()*, *subscribe()*, *publish()*, and *on()* methods for *MQTT* clients.

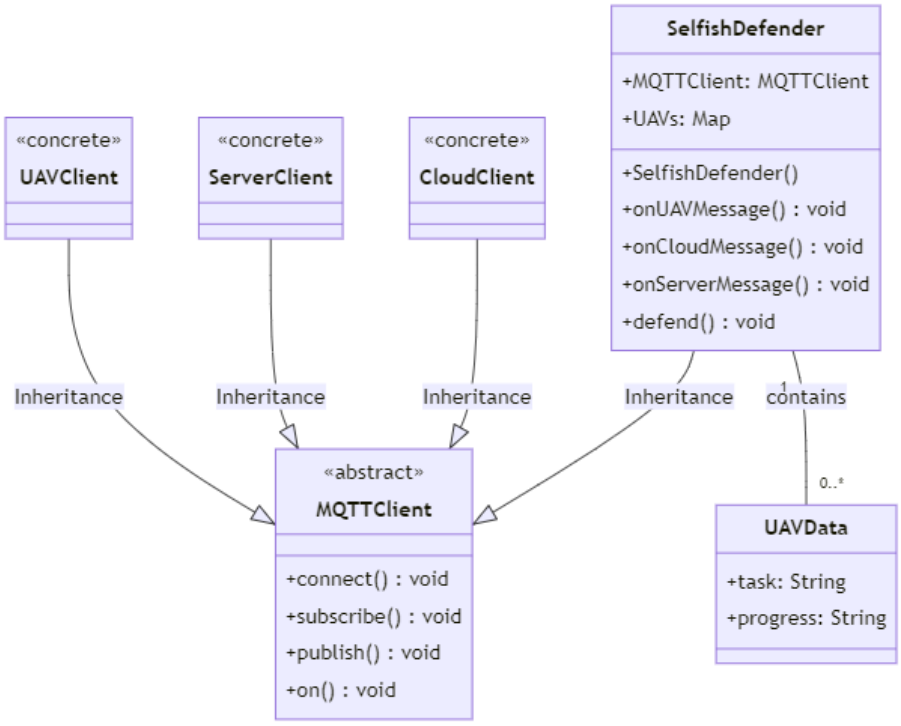

Figure 39. Selfishness defender class diagram.

Below is the process of detecting and defending against selfishness cyberattacks using the SelfishDefender class in the context of a multi-UAV–cloud system.

1. **Initialization**: When the *SelfishDefender* class is instantiated, it initializes the *MQTT* clients for the UAVs, cloud, and server. It also initializes the UAV map, which will store the registered UAVs and their corresponding assigned tasks and progress.
2. **Establishing communication**: The *SelfishDefender* class connects to the *MQTT* brokers using the *MQTT* clients. By subscribing to relevant topics, it ensures that it can receive messages from the UAVs, cloud, and server, allowing it to monitor the progress of the UAVs and detect any selfish behavior.
3. **UAV registration**: The *SelfishDefender* class uses the *onServerMessage()* method to listen for UAV registration messages from the server. When a new UAV registers, it is added to the UAV map, along with its assigned task and initial progress information.
4. **Task assignment**: Using the *onCloudMessage()* method, the SelfishDefender class processes task assignment messages from the cloud server. It assigns tasks to the UAVs and updates the UAV map with this information.
5. **UAV progress updates**: The *onUAVMessage()* method is used by the *SelfishDefender* class to process progress updates from the UAVs. The UAV map is updated with the latest progress information for each UAV.
6. **Analyzing UAV progress**: As the *SelfishDefender* class receives progress updates from the UAVs, it continuously analyzes the progress data in the UAV map. It compares the reported progress of each UAV with its assigned task, allowing it to detect any selfish behavior.
7. **Detecting selfish behavior**: If a UAV exhibits selfish behavior, such as not reporting progress or not completing tasks, the *SelfishDefender* class identifies it based on the analysis in step 6.
8. **Initiating defense mechanisms**: Once selfish behavior is detected, the *SelfishDefender* class triggers the *defend()* method. Depending on the severity and impact of the selfish behavior, this method initiates various defense mechanisms, such as (i) reassigning the task, which means the *SelfishDefender* class can reassign the task to another UAV to ensure the mission's success, despite the selfish UAV's behavior; (ii) disabling the

selfish UAV, which means that if necessary, the *SelfishDefender* class can temporarily disable the selfish UAV to prevent it from causing further harm to the overall mission; (iii) notifying relevant entities, which means the *SelfishDefender* class can also notify the cloud server or other relevant entities about the selfish UAV. This allows these entities to take appropriate action, such as reassigning tasks or penalizing the selfish UAV.

9. **Updating the UAV map**: As tasks are reassigned or selfish UAVs are disabled, the *SelfishDefender* class updates the UAV map to reflect the current status of the UAVs and their tasks.
10. **Ongoing monitoring and communication**: Throughout the process, the *SelfishDefender* class continues listening for incoming messages from the UAVs, cloud, and server, ensuring it remains connected to the *MQTT* brokers and responds appropriately to incoming messages.
11. **Closing communication channels**: Once the mission is completed or the *SelfishDefender* class is no longer needed, it closes the *MQTT* connections to the brokers, effectively ending its role in the system.

3.20. SPOF Cyberattack

3.20.1. Definition

In the given problem formulation, the network of n UAVs and m cloud-based services is represented by a graph *G=(V,E)*, where V is the set of nodes and E is the set of edges. Each node in V represents either a UAV or a cloud-based service, and each edge in E represents a connection between a UAV and a cloud-based service. Each UAV is connected to k cloud-based services, and an SPOF cyber assault can disable any k of these services. This means that if a UAV is connected to only k services, and any one of these services is disabled in an SPOF attack, the UAV will be unable to perform its intended function. Therefore, the problem seeks to identify the optimal placement of redundancy measures to mitigate the impact of SPOF cyber assaults on the UAV–cloud network [37].

To solve this problem, the set of nodes V is partitioned into two subsets: the set of UAV nodes $U = \{v_1, v_2, \ldots, v_n\}$, and the set of cloud-based service nodes $S = \{s_1, s_2, \ldots, s_m\}$. Each cloud-based service node s_i has a capacity c_i, which represents the maximum number of UAVs that can be connected to it. The problem can be formulated as an optimization problem, where the objective is to minimize the number of UAVs that are unable to perform their intended function in the event of an SPOF cyber assault, subject to the constraints that each UAV is connected to at least *k+1* cloud-based services, and that the total capacity of the redundant cloud-based services is minimized. The optimal placement of redundancy measures can be found by adding redundant cloud-based services to the network such that each UAV is connected to at least *k+1* services, including at least one redundant service. The number of redundant services to be added and their placement in the network should be determined in such a way that the total capacity of the redundant services is minimized.

3.20.2. Sequence Diagram

As shown in Figure 40, Phase 1 of this process diagram depicts the Attacker acquiring knowledge about the SPOF system, followed by scanning and vulnerability identification in Phase 2. In Phase 3, the Attacker exploits a weakness to obtain access to the SPOF system; in Phase 4, the attacker installs a backdoor; and in Phase 5, the SPOF system is attacked. Due to the failure of the SPOF system in Phase 6, the Victim transfers to the backup system, which the Attacker continues to target in Phase 7. In Phase 8, the Attacker causes damage or steals sensitive data, and in Phase 9, they conceal their traces on both systems. It is vital to highlight that this is an unlawful and immoral behavior, and if detected, the Attacker will face severe penalties. It is usually preferable to use one's skills and abilities towards beneficial and lawful endeavors.

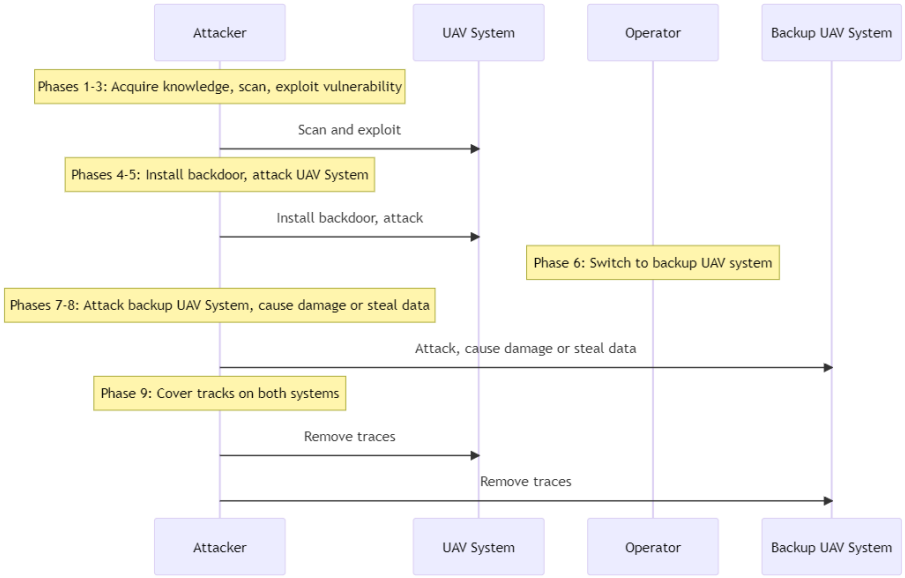

Figure 40. SPOF pattern.

3.20.3. SPOF Class Diagram

As shown in Figure 41, the class diagram represents a system for monitoring and detecting single points of failure (SPOFs) in a network of unmanned aerial vehicles (UAVs). The following is an explanation of what is happening in this system. The *SpofDefender* class is the main class that connects to the *MQTT* broker for both the cloud and UAVs, and sets up event listeners to receive messages from them. It also initializes a *Map* object called *uavs* that stores information about each UAV, including its *ID*, *last data* and *publish* request times, and *idle timeout*. The *UAV* class represents a single UAV in the network. It has a unique ID, and tracks the time when it last sent a data request, publish request, and when it should be considered idle. It also has methods to update these attributes and reset the idle timeout.

The *DataRequest* and *PublishRequest* classes are used to represent requests for data and publishing messages that the UAVs can execute. These classes store information about the UAV and the number of requests to execute, and have an *execute()* method that sends the corresponding message to the UAV. The *SpofEvent* class represents a single point of failure event, which is detected by the *detectSpofEvent* method in the *SpofDefender* class. The *SpofDefender* class has several methods: *onCloudMessage* is called when a message is received from the cloud broker. It checks for SPOF events in the data and publishes an alert message to the cloud broker if necessary. *onUAVMessage* is called when a message is received from a UAV.

It checks for SPOF events in the data and publishes an alert message to the cloud broker if necessary. It also resets the idle timeout for the *UAV.detectSpofEvent* that is called periodically to check for SPOF events in the data received from the UAVs. If an SPOF event is detected, it creates a *SpofEvent* object with the type of event and the data, and returns it. *publishAlert* is called to publish an alert message to the cloud broker with the details of the SPOF.*event.startEventInterval* is called to start an interval to periodically call *detectSpofEvent*. Overall, this system uses *MQTT* to communicate with the cloud and UAVs, and *MongoDB* to store SPOF event data. The *SpofDefender* class acts as a central control point that periodically checks for SPOF events and publishes alerts when necessary. The UAV class tracks the state of each UAV in the network, and the *DataRequest* and *PublishRequest* classes are used to send messages to the UAVs. The *SpofEvent* class represents a single point of failure event that can occur in the network.

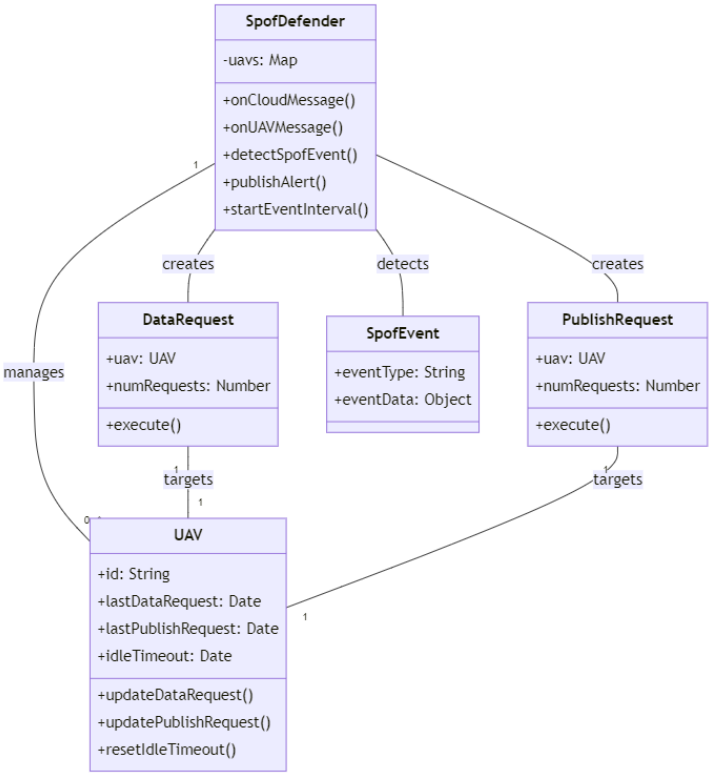

Figure 41. Defense class diagram.

Here is a detailed explanation of the defense mechanisms for SPOF attacks, focusing on how each step relates to the next:

1. **Continuous monitoring**: The *SpofDefender* class continually monitors the UAV network, subscribing to messages from the cloud broker and UAVs. This constant monitoring helps detect any irregularities or vulnerabilities in the system. Continuous monitoring serves as the foundation for the subsequent steps in the defense process.
2. **SPOF event detection**: Building upon the continuous monitoring, the *detectSpofEvent* method in the *SpofDefender* class is periodically called to analyze data received from UAVs. This analysis is essential in identifying potential SPOF events. If an SPOF event is detected, a *SpofEvent* object is created, containing information about the event and relevant data.
3. **Alert generation**: Once an SPOF event is detected and a *SpofEvent* object is created, the *SpofDefender onCloudMessage* and *onUAVMessage* methods are responsible for publishing an alert message to the cloud broker. This step is crucial for notifying system administrators or other relevant entities about the event, allowing for timely intervention and remediation of the issue.
4. **Timely intervention**: As a direct response to the alert generated in step 3, operators can take appropriate actions to address detected SPOF events. This step is essential for reducing the impact of the attack on the UAV network. Actions may involve repairing vulnerabilities, patching security holes, or isolating affected systems to prevent the spread of the attack.
5. **Periodic updates and maintenance**: To maintain the effectiveness of the *SpofDefender* system in detecting and defending against SPOF cyberattacks, regular updates and maintenance must be performed. This ongoing process ensures that the system is always up to date and ready to counter new threats. Updates include refining the detection algorithms and methods, as well as updating software and hardware components.
6. **Redundancy and failover systems**: As a long-term defense strategy, redundancy and failover systems should be implemented alongside the *SpofDefender* system. These

systems provide backup resources and capabilities, ensuring that the UAV network can continue to function even when one or more components fail. This final step enhances the overall resilience of the UAV network against SPOF cyberattacks.

3.21. Sybil Attack

3.21.1. Definition

Given a set of UAVs, represented as nodes in a graph, and a set of connections between them, represented as edges, the objective is to identify Sybil nodes in the network, i.e., nodes that have been created by an attacker to gain control or disrupt the network's operation, and maximize their number. The Sybil nodes can be created by the attacker in two ways: by creating virtual machines on the cloud platform or by manipulating the UAVs' communication protocols. The problem can be formulated as an optimization problem, where the goal is to maximize the number of Sybil nodes in the network while ensuring that the network remains connected and functional [38].

The problem can be modeled using graph theory and can be solved using various algorithms, such as the maximum flow algorithm or the maximum matching algorithm. Formally, let $G = (V, E)$ be a directed graph representing the UAV–cloud network, where V is the set of nodes and E is the set of edges. Let S be the set of Sybil nodes in the network, and let C be the set of UAVs' decision-making algorithms that can be manipulated. The objective is to find a set of nodes S' such that S' is a subset of V | S' | is maximized. The connectivity of the network is measured by the maximum flow or maximum matching.

3.21.2. Sequence Diagram

As shown in Figure 42, the sequence diagram depicts the process of a Sybil attack. Firstly, the Attacker first finds Target Nodes in several UAV–cloud systems in Phase 1, and then in Phase 2 creates phony identities and launches a Sybil assault against the Target Nodes. In Phase 3, the phony identities enter Legitimate Nodes' UAV–cloud systems, which accept them as valid nodes.

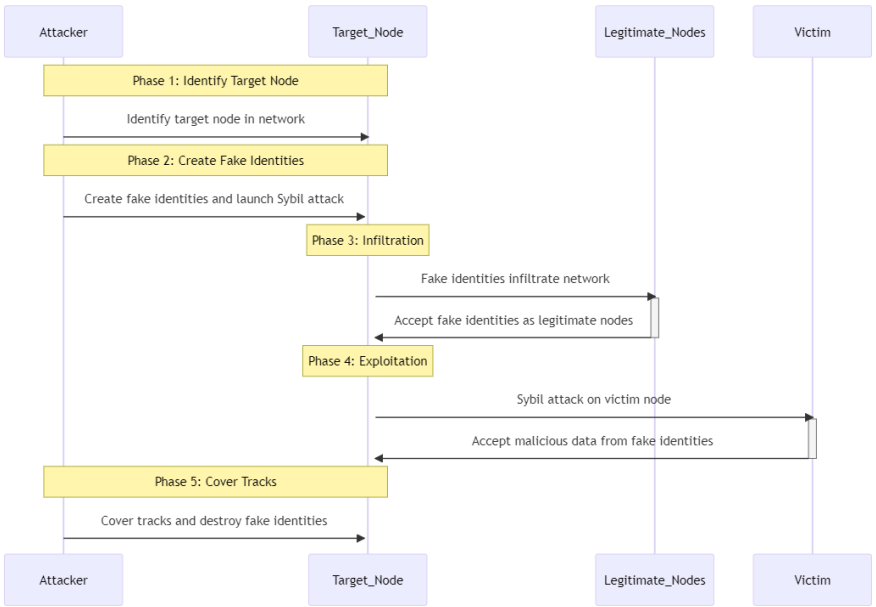

Figure 42. Sybil attack.

Once the UAV–cloud systems have been infiltrated, the Attacker initiates the Sybil attack against the UAV–cloud system of the Victim in Phase 4, and the Victim accepts harmful data from the phony identities. Phase 5 concludes with the Attacker covering their traces and destroying their bogus identities to avoid being discovered. It is vital to highlight

that this is an unlawful and immoral behavior, and if detected, the Attacker will face severe penalties. It is usually preferable to use one's skills and abilities towards beneficial and lawful endeavors.

3.21.3. Defense Class Diagram

Figure 43 shows the defense class diagram to mitigating a Sybil attack. The *mqtt* and *MongoClient* classes are not explicitly shown in the diagram, as they are external classes used by the *SybilDefender* class. The *SybilDefender* class has several properties, including two *MQTT* client instances (*uavClient* and *cloudClient*), two sets of UAVs (*uavs* and *sybilUAVs*), and various configuration parameters (*MAX_UAVS, DEVIATION_MULTIPLIER, and DETECTION_INTERVAL*).

It also has several methods, including event handlers for *MQTT* messages (*onCloudMessage* and *onUAVMessage*), a method for detecting Sybil UAVs (*detectSybilUAVs*), and a method for publishing alerts to the cloud and storing them in a *MongoDB* database (*publishAlert*). The *defend* method is the main entry point for the class, and it sets up periodic checks for idle UAVs and Sybil UAVs. Overall, the class diagram shows the relationships between the various properties and methods of the *SybilDefender* class, as well as its external dependencies on the *mqtt* and *MongoClient* classes.

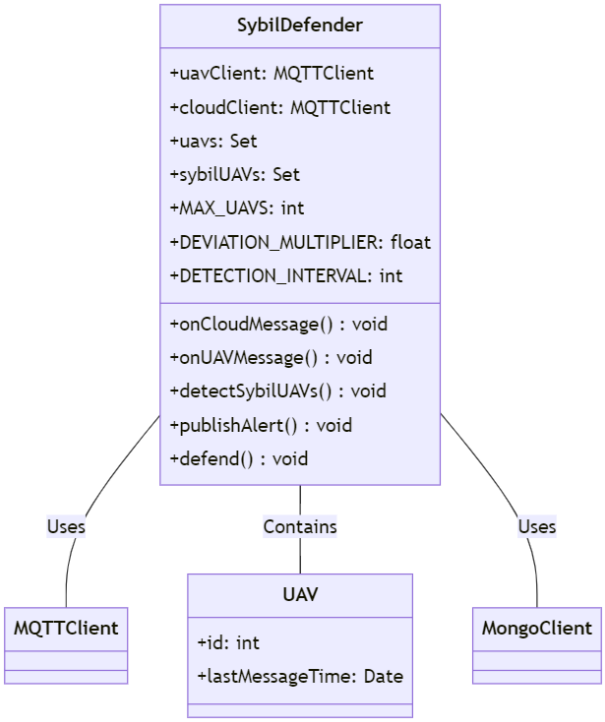

Figure 43. Defense from Sybil attack.

The defense mechanism for Sybil cyberattacks in the context of a UAV–cloud system can be explained as follows:

1. **Initialization**: The *SybilDefender* class initializes and establishes connections with the *MQTT* broker for UAVs and the cloud. It also initializes an instance of the *MongoClient* class to interact with the *MongoDB* database. The *SybilDefender* sets up initial configuration parameters, such as MAX_UAVS, DEVIATION_MULTIPLIER, and DETECTION_INTERVAL.
2. **UAV communication**: The UAVs in the network communicate with the *SybilDefender* class by sending messages through the *MQTT* broker. These messages may include telemetry data, status updates, or other relevant information.

3. **Message processing**: The *SybilDefender* class listens for incoming messages from the UAVs and the cloud. It processes messages from the UAVs using the *onUAVMessage()* method and messages from the cloud using the *onCloudMessage()* method.
4. **Periodic detection**: The *SybilDefender* class periodically runs the *detectSybilUAVs()* method to analyze the messages received from the UAVs and identify any potential Sybil UAVs. This method compares the received messages against predefined thresholds (e.g., MAX_UAVS and DEVIATION_MULTIPLIER).
5. **Sybil UAV identification**: If the *SybilDefender* class identifies any Sybil UAVs, it adds them to the *sybilUAVs* set. This allows the system to keep track of the malicious UAVs.
6. **Taking action**: After identifying the Sybil UAVs, the *SybilDefender* class takes appropriate action against them, such as isolating or blocking the malicious UAVs from the network. This step ensures that the impact of the Sybil attack is mitigated.
7. **Alert generation**: The *SybilDefender* class uses the *publishAlert()* method to send alerts to the cloud about the identified Sybil UAVs and any actions taken by the *SybilDefender* class to mitigate the attack.
8. **Storing alert information**: In addition to sending alerts, the SybilDefender class stores the alert information in the MongoDB database using the MongoClient instance. This helps maintain a record of detected Sybil UAVs and corresponding defensive actions.
9. **Defend method**: The *defend()* method serves as the main entry point for the *SybilDefender* class, setting up the periodic checks for idle and Sybil UAVs. It ensures that the *SybilDefender* class continuously monitors and defends the UAV–cloud system against Sybil attacks.
10. **Continuous monitoring**: As the *defend()* method runs periodically, the *SybilDefender* class continues to monitor and analyze the communication between the UAVs and the cloud, detecting and defending against any new Sybil attacks.

3.22. SYN Flood Attack

3.22.1. Definition

Let S be the number of *SYN* requests sent by the attacker during the attack. Let T be the time interval during which the attack occurs. Let R be the rate at which *SYN* requests are sent, i.e., $R = S / T$. Let A be the number of available resources, such as memory and CPU, on the target system. Let U be the rate at which resources are consumed by each SYN request, i.e., $U = 1 / RTT$, where *RTT* is the round-trip time for the three-way handshake. The objective of the attacker is to consume all available resources on the target system, i.e., to make $A = 0$.

The attacker aims to achieve this by sending a large number of SYN requests (i.e., a high value of S) and not responding to the *SYN-ACK* replies, which causes the target system to use resources while waiting for a response. The attacker's success in consuming resources depends on the rate at which resources are consumed by each SYN request (i.e., U) and the rate at which SYN requests are sent (i.e., R). The impact of a SYN flood attack on the target system can be quantified by the number of legitimate requests that are denied due to the attack. Let L be the rate at which legitimate requests are received by the target system. If the rate of SYN requests exceeds the rate of legitimate requests (i.e., $R > L$), the target system will become overloaded and unable to respond to legitimate requests, resulting in a denial of service [39] .

3.22.2. Sequence Diagram

As shown in Figure 44, in Phase 1 of this sequence diagram, the Attacker selects Target Nodes in several UAV–cloud systems before delivering a large number of SYN requests to the Target Nodes in Phase 2. In Phase 3, the Target Nodes react with SYN-ACK packets. In Phase 4, the Attacker drops or responds to these packets with RST packets. In Phase 5, the Attacker repeatedly repeats the attack until the Target Nodes become inactive. In Phase 6, the SYN flood assault causes the UAV–cloud systems to become unresponsive, culminating in system collapse.

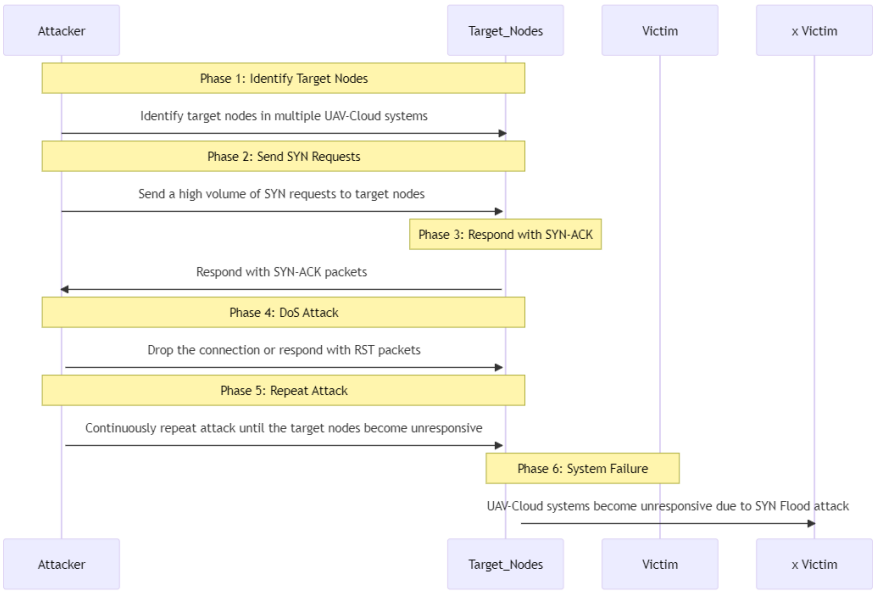

Figure 44. SYN flood attack.

3.22.3. Defense Class Diagram

As presented in Figure 45, *SynFloodDefender* is a class that has *mqtt.Client* as its private members. It has five public methods: *onCloudMessage*, *onUAVMessage*, *detectSynFlood*, *publishAlert*, and *defend*. *mqtt.Client* is a class used to create *MQTT* clients to connect to brokers. It has six public methods: *connect*, *on*, *subscribe*, *publish*, *once*.

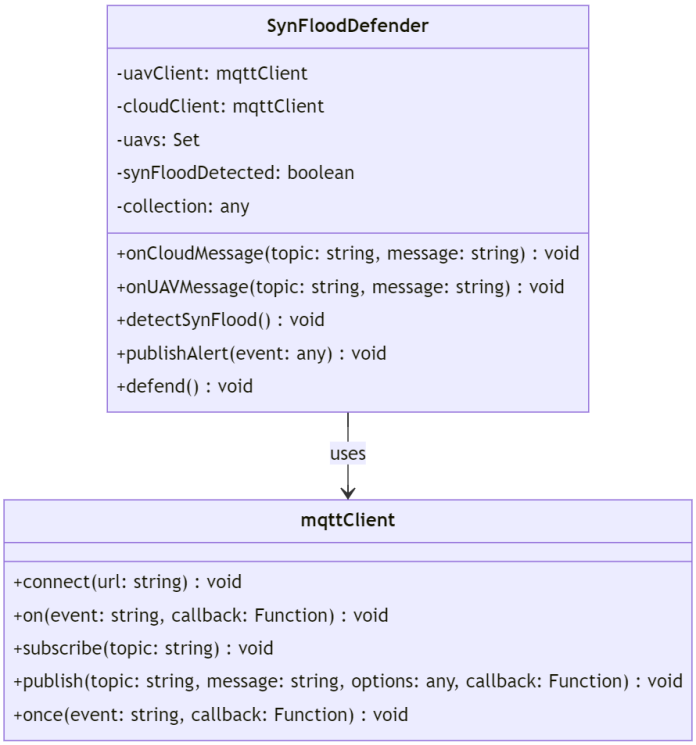

Figure 45. Defense from SYN flood attack.

The *SynFloodDefender* class uses the *mqtt.Client* class to connect to *MQTT* brokers and subscribe to topics. It also has a Set object to keep track of registered UAVs and a Boolean flag to indicate if a SYN flood attack has been detected. The *detectSynFlood* method collects

data from UAVs, analyzes them for SYN flood attack, and publishes an alert if an attack is detected. The *publishAlert* method saves the event to *MongoDB* and publishes an alert to the cloud. The *defend* method is used to periodically check for SYN flood attacks. The *mqtt.Client* class is used by *SynFloodDefender* to connect to brokers, subscribe to topics, publish messages, and listen to events.

The defense mechanism against SYN flood attacks in the context of a UAV–cloud system can be explained as follows:

1. **Initializing the *SynFloodDefender* class**: The system initializes the *SynFloodDefender* class, establishing connections with the *MQTT* broker for UAVs and the cloud. It sets up a Set object to track registered UAVs and a Boolean flag to indicate if a SYN flood attack has been detected.
2. **Connecting and communicating with UAVs**: The UAVs in the network connect to the *MQTT* broker and start sending messages containing telemetry data, status updates, or other relevant information to the *SynFloodDefender* class.
3. **Subscribing to message topics**: The *SynFloodDefender* class subscribes to the necessary *MQTT* topics to receive incoming messages from the UAVs and the cloud.
4. **Processing messages from UAVs and cloud**: As the *SynFloodDefender* class receives messages from the UAVs, it processes them using the *onUAVMessage()* method. Similarly, messages from the cloud are processed using the *onCloudMessage()* method.
5. **Periodically checking for SYN flood attacks:** The *SynFloodDefender* class periodically executes the *detectSynFlood()* method, which analyzes the messages received from the UAVs to identify any potential SYN flood attacks.
6. **Analyzing UAV communication patterns**: During the execution of the *detectSynFlood()* method, the SynFloodDefender class examines the UAV communication pattern and compares it against predefined thresholds to identify any abnormal behavior indicative of a SYN flood attack.
7. **Identifying a SYN flood attack**: If the *SynFloodDefender* class detects a SYN flood attack, it sets the Boolean flag accordingly to indicate that a SYN flood attack has been identified.
8. **Generating and sending alerts**: Upon detecting a SYN flood attack, the *SynFloodDefender* class uses the *publishAlert()* method to send alerts to the cloud, informing it about the detected SYN flood attack and any actions taken by the SynFloodDefender class to mitigate the attack.
9. **Storing alert information in MongoDB**: In addition to sending alerts, the *SynFloodDefender* class stores the alert information in the *MongoDB* database to maintain a record of detected SYN flood attacks and corresponding defensive actions.
10. **Running the defend() method**: The *defend()* method serves as the main entry point for the *SynFloodDefender* class, setting up the periodic checks for SYN flood attacks and ensuring continuous monitoring and defense against SYN flood attacks in the UAV–cloud system.
11. **Continuously monitoring and defending against SYN flood attacks**: As the *defend()* method runs periodically, the *SynFloodDefender* class keeps monitoring and analyzing the communication between the UAVs and the cloud, detecting and defending against any new SYN flood attacks.

3.23. Wormhole Attack

3.23.1. Definition

Let $G = (V, E)$ denote a wireless mesh network with N nodes, where each node v in V can transmit and receive data packets. Let D be the set of all possible data packets that can be transmitted across the network, and let S be the set of all possible sources of data packets. A wormhole attack is represented as a directed virtual link between two nodes u and v, denoted by (u, v), which is established by the attacker to intercept and reroute network traffic [40].

Let $H = (V, E', C)$ denote a directed graph, where E′ is the set of virtual links established by the attacker, and C is the cost of each virtual link. The objective of the attacker is to maximize the amount of sensitive data intercepted or the number of network connections disrupted. This can be represented mathematically as follows: $Maximize \sum_{d \in D} \sum_{s \in S} X_{d,s}$, where $X_{d,s}$ is a binary decision variable that indicates whether a data packet d from source s has been intercepted.

3.23.2. Sequence Diagram

Figure 46 shows the process of a wormhole cyberattack. In Phase 1 of this sequence diagram, the Attacker selects Target Nodes in several UAV–cloud systems before delivering a large number of SYN requests to the Target Nodes in Phase 2. In Phase 3, the Target Nodes react with SYN-ACK packets. In Phase 4, the Attacker drops or responds to these packets with RST packets. In Phase 5, the Attacker repeatedly repeats the attack until the Target Nodes become inactive. In Phase 6, the SYN flood assault causes the UAV–cloud systems to become unresponsive, culminating in system collapse.

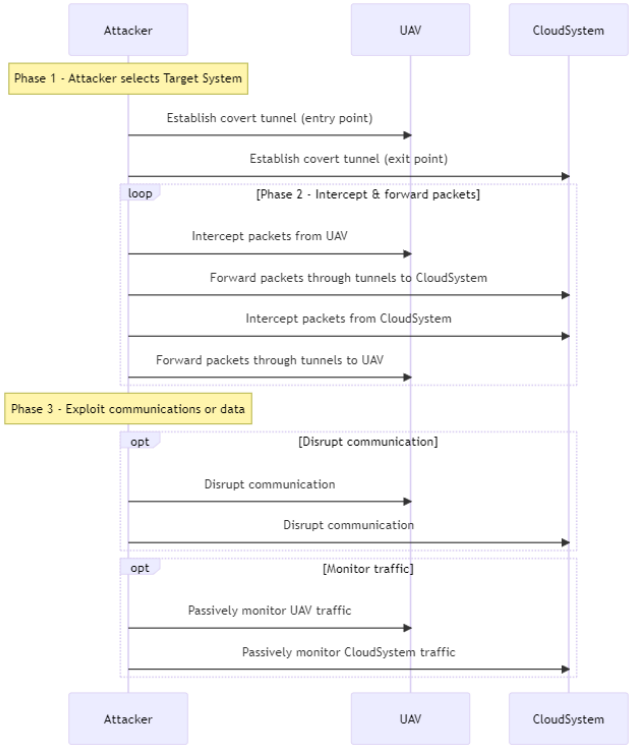

Figure 46. Wormhole cyberattack.

3.23.3. Defense Class Diagram

As shown in Figure 47, the *WormholeDefender* class has a one-to-many relationship with the *mqtt* class through its *uavClient* and *cloudClient* properties, indicating that it uses instances of the *mqtt* class to connect to the UAV and cloud brokers. Similarly, the *WormholeDefender* class has a one-to-one relationship with the *MongoClient* class through its collection property, indicating that it uses an instance of the *MongoClient* class to connect to the *MongoDB* database. The *WormholeDefender* class has three public methods: *calculateDistance*, *detectWormhole*, and *publishAlert*, and one private method: the *constructor*. The *defend* method is also public but it is not used in the code provided. The *onCloudMessage* and *onUAVMessage* methods are public and are used as event handlers for messages received from the cloud and UAV brokers, respectively.

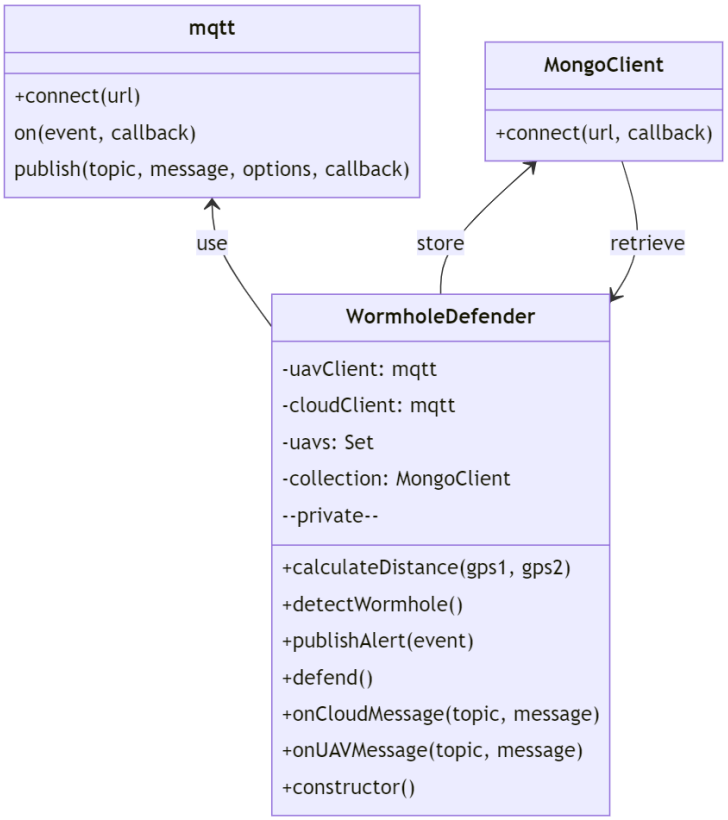

Figure 47. Defense from wormhole attack.

In the context of a network communication system involving UAVs and cloud brokers, the *WormholeDefender* class aims to protect the system from potential wormhole attacks. Below is a step-by-step explanation of the defense mechanisms:

1. **Initialization**: An instance of the *WormholeDefender* class is created, establishing connections to the UAV and cloud brokers using the *mqtt* class instances (*uavClient* and *cloudClient*) and a connection to the *MongoDB* database using the *MongoClient* instance (collection).
2. **Monitoring communication**: The *WormholeDefender* class listens for incoming messages from the UAV and cloud brokers using the *onUAVMessage* and *onCloudMessage* event handlers, respectively. These methods are responsible for processing the messages and collecting relevant data for further analysis.
3. **Distance calculation**: The *calculateDistance* method is used to compute the distance between two communicating nodes (UAVs or cloud brokers) based on their location information. This method helps in determining if the communication distance is within the expected range.
4. **Wormhole detection**: The *detectWormhole* method analyzes the collected data to identify potential wormhole attacks. This method checks for anomalies in the communication patterns, such as unexpected delays or significantly longer communication distances than expected, which could indicate the presence of a wormhole attack.
5. **Alert generation**: If a wormhole attack is detected, the *WormholeDefender* class generates an alert using the *publishAlert* method. This method saves the event to the *MongoDB* database and publishes an alert message to the cloud broker, notifying the network administrator or other security mechanisms about the detected attack.
6. **Defense strategy**: Upon receiving the alert, the network administrator or automated security systems can take appropriate defensive actions, such as isolating the affected nodes, reconfiguring the network topology, or implementing additional security measures to prevent further attacks.

3.24. Brute Force Attack

3.24.1. Definition

In the context of UAV–cloud systems, brute force attacks on user or system accounts pose a significant threat as they may allow attackers to gain unauthorized access to sensitive data or systems. Attackers can use specialized software or tools to attempt a large number of passwords or passphrases until the correct one is discovered, potentially leading to the installation of malware, ransomware, or other forms of damage.

If a brute force attack is successful, the attacker may also be able to access other accounts or systems if they have administrator credentials, which could lead to even more damage. To prevent such attacks, it is essential to use strong passwords or passphrases, enable multifactor authentication, and set up account lockout restrictions that prevent attackers from continuously attempting different passwords. By implementing these measures, the risk of brute force attacks can be significantly reduced in the UAV–cloud environment [41].

3.24.2. Sequence Diagram

As shown in Figure 48, the sequence diagram portrays the process of a brute force attack, illustrating the interaction between an Attacker and a Target System. Initially, the Attacker identifies a system that they want to target for the attack. Subsequently, the Attacker initiates a series of attempts to test various username and password combinations in an effort to gain unauthorized access to the Target System. During this process, the Attacker sends a test username and password combination to the Target System, which then responds, indicating whether the authentication attempt was successful or unsuccessful.

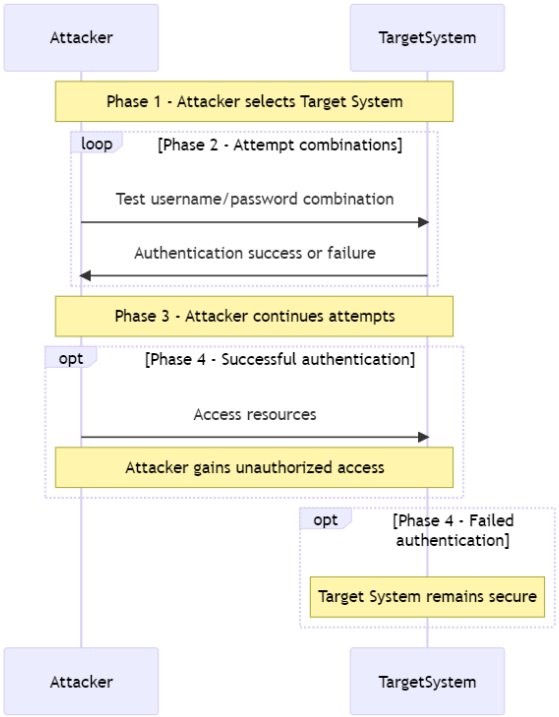

Figure 48. Brute force attack.

As a result, the Attacker continues trying different combinations, persisting until they either succeed in gaining access or exhaust all possible combinations. Consequently, if the Attacker discovers a successful combination, they proceed to access the resources within the Target System, thereby gaining unauthorized access. On the other hand, if the Attacker fails to find a successful combination, the Target System remains secure, and the attack is ultimately unsuccessful.

3.24.3. Defense Class Diagram

As shown in Figure 49, *BruteForceDefender* is the primary class in the system. It is responsible for connecting to an *MQTT* message broker, receiving messages from linked drones, detecting brute force assaults, and publishing alarms to a *MongoDB* database. The class has secret methods for processing messages received from the cloud and unmanned aerial vehicles, detecting brute force assaults, and issuing database alerts. It also contains a public function *defend()* that starts the detection of brute force attempts. The *mqtt* class represents the *MQTT* message broker and offers methods for connecting to the broker, subscribing to topics, publishing messages, and registering callbacks for message events.

The *mongodb* class represents the *MongoDB* database and offers methods for establishing a new database instance and connecting to the database. The *insertOne()* function of the Collection class, a child class of *mongodb*, is used to insert a new document into the database collection. The diagram's arrows depict the connections between the classes. For example, the *BruteForceDefender* class communicates with the *MQTT* message broker and *MongoDB* database via instances of the *mqtt* and *mongodb* classes, respectively.

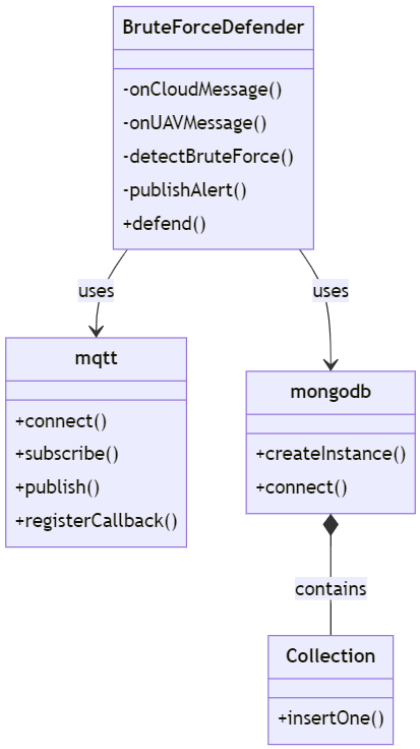

Figure 49. Defense class diagram of brute force.

3.25. Leaks of Data Due to Human Mistakes

3.25.1. Definition

Let *X* be the set of all sensitive data collected by UAVs and transmitted to a cloud-based system for processing and analysis. Let *Y* be the set of all workers or other persons with access to the sensitive data. Let *Z* be the set of all possible ways in which the sensitive data can be leaked due to human error. The challenge is to minimize the risk of a data breach, given the following constraints: *For all x in X*, ensure that only authorized personnel have access to the data and that the data are transmitted securely. *For all y in Y*, ensure that they are aware of the importance of data security and are trained to handle sensitive data appropriately. *For all z in Z*, ensure that appropriate security measures are in place to prevent inadvertent exposure of sensitive data, including the use of encryption, multifactor authentication, and access control [42].

The objective is to minimize the probability of a data breach, which can be defined as the likelihood that sensitive data are disclosed to unauthorized individuals or exploited by cybercriminals. This can be expressed as a function of the probability of each possible leakage scenario in Z, weighted by the severity of the consequences of a data breach. A possible mathematical model to achieve this objective is a stochastic optimization model that incorporates the probabilities of different leakage scenarios and their associated consequences. The model can be solved using techniques such as *Monte Carlo* simulation or scenario analysis to evaluate the risk of a data breach and identify the optimal security measures to mitigate this risk.

3.25.2. Sequence Diagram

As shown in Figure 50, in Phase 1 of this sequence diagram, the *Cloud Administrator* configures the UAV–cloud system's security parameters. In Phase 2, the UAV obtains data from the victim UAV–cloud system. In Phase 3, the Attacker gains access to the security settings using social engineering. In Phase 4, the Cloud Administrator then incorrectly configures the security settings, resulting in a data breach.

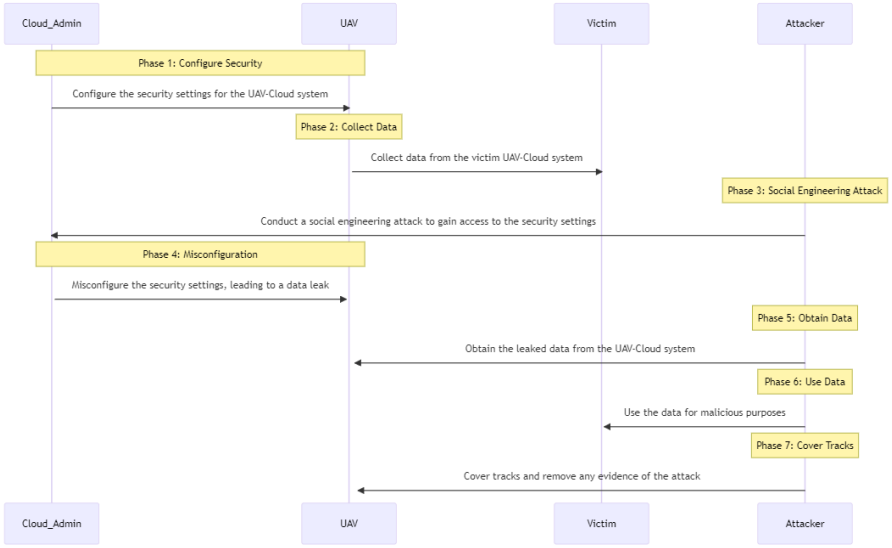

Figure 50. Leak data due to human error.

In Phase 5, the Attacker acquires the leaked data from the UAV–cloud system, and in Phase 6, he utilizes them for nefarious purposes. In Phase 7, the *Attacker* conceals their tracks and eliminates all traces of the attack. It is vital to remember that data leaking due to administrative errors poses a substantial danger to security and privacy, and can have severe repercussions for the impacted persons and organizations. It is crucial to establish security settings correctly and teach staff to spot and withstand social engineering attempts.

3.25.3. Defense Class Diagram

As shown in Figure 51, the *crypto* class provides encryption and decryption functionality, including the *createCipher* and *createDecipher* methods, which are used for encrypting and decrypting data. The *speakeasy* class provides multifactor authentication functionality, including the *generateSecret*, *totp* and *totp*.verify methods, which are used for generating authentication tokens and verifying them.

The *AccessControl* class provides access control functionality, including the can method, which is used for checking if a user has access to a particular resource. The *Server* class represents an *HTTPS* server that listens on a given port and handles requests from clients.

The *Logger* class provides logging functionality, including the info method, which is used for logging messages.

The *MultiFactor* class provides multifactor authentication functionality, including the *generateSecret* and *verifyToken* methods, which are used for generating authentication tokens and verifying them. The *Main* class is the main class of the application, and it includes private properties for data, key, user role, action, resource, token, secret, encrypted data, decrypted data, access granted, server, logger, eslintCli, and eslintReport, and instances of the Encrypt, MultiFactor, crypto, *AccessControl*, and *speakeasy* classes. The *Main* class also contains public methods for setting up the *HTTPS* server, logging, static code analysis and testing, and for running the application. The *Main* class interacts with the other classes in the system by calling their methods, as indicated by the arrows in the diagram.

Monitoring and logging (Server and Logger classes) is the Server class that handles incoming client requests, while the Logger class logs relevant events. By carefully monitoring and logging user activities, it is possible to detect potential data leaks and address them before significant damage occurs. Logging can also help in tracing the source of a data leak and identifying the cause of the human error. Furthermore, the Main class manages the overall application and interacts with other classes to enforce security measures. By having a central class that oversees the entire process, it is easier to ensure that all security mechanisms are in place and functioning correctly.

The Crypto class provides encryption and decryption functionality, which can be used to protect sensitive data from being accidentally leaked by human error. By ensuring that sensitive data are encrypted before they are stored or transmitted, the risk of data leakage due to human error is significantly reduced. If someone inadvertently shares or exposes encrypted data, they would still be unreadable without the correct decryption key.

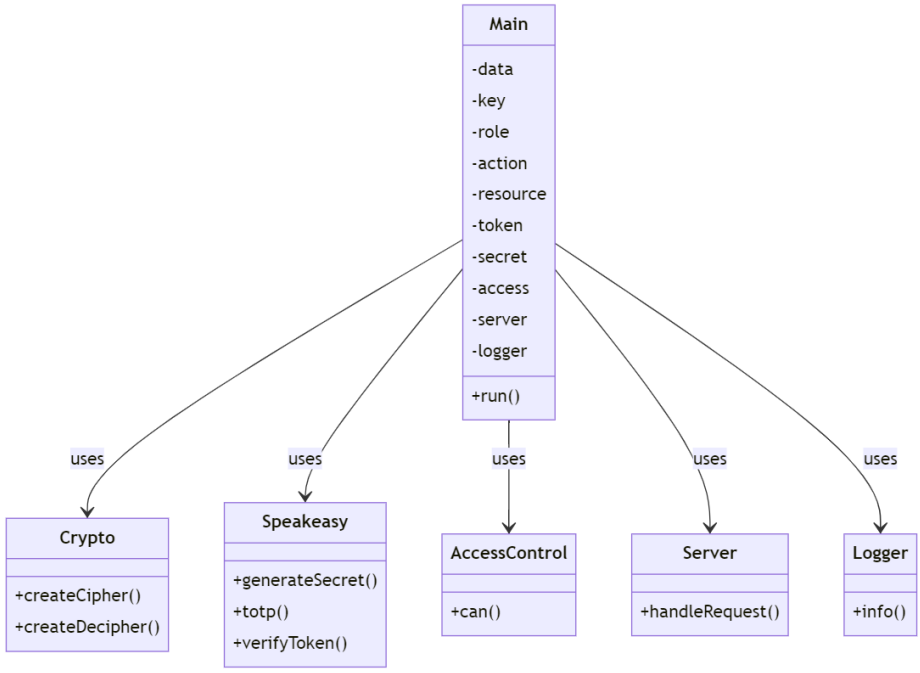

Figure 51. Defense class solution.

3.26. Data Loss Due to System Crash

3.26.1. Definition

Let D denote the set of all data stored by an organization, and let S be the set of storage devices used to store D. Let B be the set of backup devices used to make copies of D for recovery purposes. Let C: $D \times S \rightarrow \{0,1\}$ be a function that indicates whether a particular storage device correctly stores a particular datum. If $C(d,s) = 1$, it means that datum d is correctly stored on device s, and if $C(d,s) = 0$, it means that datum d is not correctly stored

on device s. Let *R:D* × *B* → *{0,1}* be a function that indicates whether a particular backup device correctly stores a copy of a particular datum. If *R(d,b) = 1*, it means that a copy of datum d is correctly stored on backup device *b*, and if *R(d,b) = 0*, it means that a copy of datum *d* is not correctly stored on backup device *b*. Let *P: D -> {0,1}* be a function that indicates whether a particular datum is sensitive or vital. If *P(d) = 1*, it means that datum *d* is sensitive or vital, and if *P(d) = 0*, it means that datum *d* is not sensitive or vital [43].

The problem is to minimize the risk of inadvertent data loss, subject to the constraints that: *For all data d in D*, there exists at least one storage device *s in S* such that *C(d,s) = 1*. *For all data d in D*, there exists at least one backup device *b in B* such that *R(d,b) = 1*. For all sensitive or vital data *d in D*, there exists at least one backup device *b in B* such that *R(d,b) = 1*. The cost of storing and backing up data is minimized.

3.26.2. Sequence Diagram

As shown in Figure 52, this sequence diagram represents a communication sequence between a UAV (unmanned aerial vehicle) and a cloud system, involving the storage and retrieval of data. The diagram is split into two parts: data storage and data retrieval. Below is a step-by-step explanation of each part:

1. **Data storage**: The sequence starts with the UAV sending a request to the cloud for data storage. Then, the cloud receives the request and verifies the UAV's privileges to access the cloud system. After that, it sends a message to the UAV requesting the data to be stored. The UAV sends the data to the cloud. Furthermore, the cloud receives the data and verifies their integrity. If the data are corrupt, the cloud sends an error message to the UAV and discards the data. If the data are valid, the cloud stores the data in multiple devices and creates backups in multiple devices. Then, the cloud sends a message to the UAV indicating that the data have been successfully stored. After that, the UAV acknowledges the message. If the data are sensitive or vital, the cloud performs additional encryption and access control measures to ensure data security.
2. **Data retrieval**: The sequence starts with the UAV sending a request to the cloud for data retrieval. Then, the cloud receives the request and verifies the UAV's privileges to access the cloud system. After that, it sends a message to the UAV requesting the data to be retrieved. The cloud retrieves the data from the storage and backup devices, and sends them to the UAV. In the next step, the UAV receives the data and verifies their integrity using a checksum. If the data are corrupt, the UAV sends an error message to the cloud and requests the data to be retrieved again. If the data are valid, the UAV acknowledges the message and uses the data as needed. Overall, this sequence diagram shows the flow of data between the UAV and the cloud, as well as the data integrity checks and access control measures taken by the cloud system.

3.26.3. Defense Class Diagram

As shown in Figure 53, this class diagram defines five classes: *AWS*, *S3*, *StoreService*, *CryptoService*, and *DataEntity*. The arrows between the classes represent method calls and indicate the relationships between the classes. Here is a brief explanation of each class: *AWS* represents the Amazon Web Services SDK, and has a private property called options, which represents the *AWS* configuration options. Then, S3 represents the *S3* storage service, and provides public methods *putObject()* and *getObject()* to interact with the *S3* storage service. Furthermore, *StoreService* represents a service that uses the *S3* class to store, backup, and retrieve data from an *S3* bucket, and also uses the *CryptoService* class to encrypt and decrypt data.

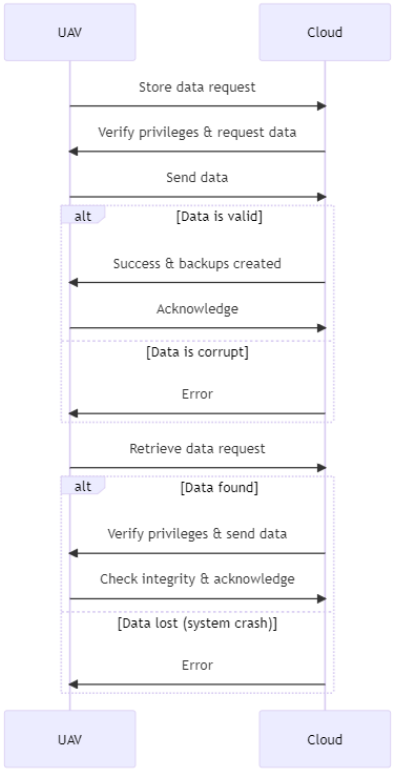

Figure 52. UAV–cloud data loss when system crashes.

CryptoService represents a service that provides methods to encrypt and decrypt data using AES-256 encryption. Then, *DataEntity* represents an entity for storing and retrieving data from an S3 bucket. It has private properties for the bucket name and the *S3* instance, and provides public methods to store and retrieve data. The arrows between the classes indicate that the *StoreService* class uses the *S3* class to interact with the *S3* storage service, and also uses the *CryptoService* class to encrypt and decrypt data. The *AWS* class also calls methods on the *S3* class to interact with the *S3* storage service.

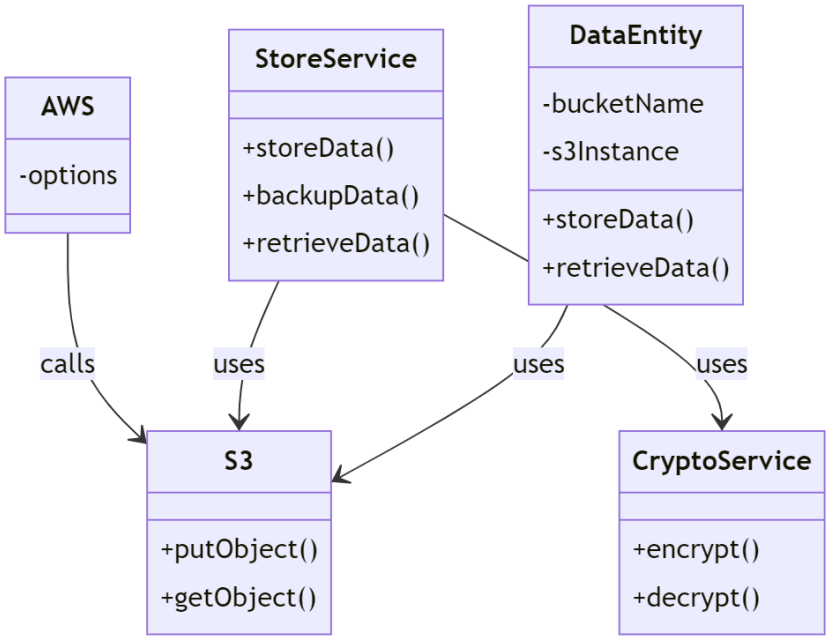

Figure 53. Defense class diagram to avoid system crash.

Below is a sequential explanation of the defense mechanisms against data leaks due to a system crash:

1. **Data storage request**: The process begins with a data storage request initiated by a user or a system component. The request is sent to the *StoreService* class, which is responsible for handling the storage and retrieval of data.
2. **Access control**: The *StoreService* class interacts with the *AWS* class to verify the access privileges of the requestor. This step ensures that only authorized users or services can store or access the data, providing an essential security measure against unauthorized data access.
3. **Data encryption**: Before storing the data, the *StoreService* class calls the *CryptoService* class to encrypt sensitive data using AES-256 encryption. This process ensures that the data are protected and unreadable, even if they are accessed by unauthorized parties during a system crash.
4. **Redundant storage**: The *StoreService* class then interacts with the *S3* class to store the encrypted data in an *S3* bucket. Data are stored across multiple devices, providing redundancy and reducing the risk of data loss due to a single device or system failure.
5. **Data backup**: In addition to storing the data, the *StoreService* class also creates multiple backups of the data across different devices. This further enhances redundancy and reduces the likelihood of data loss during a system crash.
6. **Data retrieval request**: When a user or system component requests to retrieve data, the *StoreService* class once again verifies the access privileges of the requestor using the AWS class.
7. **Data decryption**: If the requestor is authorized, the *StoreService* class retrieves the encrypted data from the *S3* storage service and uses the *CryptoService* class to decrypt them. This ensures that the data are readable only by authorized parties.
8. **Data integrity check**: After the data have been decrypted, an integrity check can be performed to ensure that the retrieved data are not corrupt. If the data are found to be corrupt, an error message is sent, and the data retrieval process is repeated until a valid copy of the data is obtained.

3.27. Difference of Security Attack and Defense Pattern between Centralized vs. Decentralized

The security landscape of centralized and decentralized architectures varies significantly due to their inherent design and communication patterns. Understanding the differences in attack mechanisms and defense strategies between these architectures is crucial for designing secure systems. In this section, we explain the main difference of security attack and defense pattern between centralized and decentralized architecture.

3.27.1. Attack Pattern Comparison

Black Hole and Gray Hole Attacks

In a centralized architecture, an attacker who successfully compromises the central server can potentially gain the ability to discard or selectively forward data, thereby affecting the entire network. The consequences of such an attack are more far-reaching, as the compromised central server acts as the core communication hub for the entire system. On the other hand, in a decentralized architecture, an attacker can target individual nodes to execute black hole or gray hole attacks. However, the impact of these attacks is more localized, and the overall network remains largely unaffected. To cause widespread disruption in a decentralized system, the attacker would need to coordinate a simultaneous attack on multiple nodes, which is more complex and challenging to achieve.

Collision Network

In a centralized architecture, a high volume of communication with the central server can lead to an increased risk of collisions. These collisions, in turn, may cause network bottlenecks and impact the overall system performance. In such scenarios, the central server becomes a point of congestion, as all nodes attempt to communicate with it simultaneously,

making the system more vulnerable to slowdowns and potential disruptions. Conversely, in a decentralized architecture, the distributed nature of the system inherently reduces the likelihood of network collisions. However, this does not entirely eliminate the possibility of collisions. A high volume of communication between nodes can still lead to local disruptions within specific areas of the network. Despite this, the impact of such disruptions is generally more confined, as the decentralized design of the system prevents a single point of failure from causing widespread issues.

Data Tampering, Modification, and Replay Attacks

In the context of data tampering, modification, and replay attacks, centralized and decentralized architectures face distinct challenges and vulnerabilities. In a centralized architecture, an attacker who successfully compromises the central server can potentially manipulate data across the entire network. This is because the central server acts as the primary repository and communication hub for all nodes, making it a high-value target for attackers seeking to cause widespread disruption.

On the other hand, in a decentralized architecture, an attacker might attempt to tamper with data on specific nodes. However, the impact of such an attack is generally limited due to the distributed nature of the system. Decentralized architectures often employ consensus algorithms and redundancy measures to validate and cross-verify data across multiple nodes. These mechanisms help to detect and prevent data tampering, modification, and replay attacks by ensuring that maliciously altered data are not accepted as valid by the majority of the network.

Deauthentication, DDoS, Slowloris, Flooding, and SYN Flood Attacks

In a centralized architecture, these attacks target the central server, potentially causing disruptions across the entire network. For instance, a deauthentication attack can forcibly disconnect nodes from the central server, leading to loss of communication and control. A DDoS or Slowloris attack can inundate the server with excessive traffic, rendering it unable to process legitimate requests, leading to a system-wide outage. Flooding and SYN flood attacks can also saturate the central server's resources, causing network congestion and severely impacting overall system performance. Since the central server is the primary communication hub and data repository for all nodes, successful attacks on it can have far-reaching consequences for the system's functionality and performance.

In a decentralized architecture, these attacks can target multiple nodes simultaneously. However, the impact of these attacks is generally more localized. For example, a deauthentication attack might only affect a few nodes within the network, while the rest continue to operate normally. Similarly, a DDoS or Slowloris attack on specific nodes might cause temporary disruptions but not compromise the entire system. Flooding and SYN flood attacks can still lead to localized network congestion and performance issues, but the decentralized nature of the system helps prevent these issues from propagating across the entire network. Nevertheless, if a significant portion of the network becomes overwhelmed by these attacks, the entire system could still be adversely affected. In such cases, the decentralized architecture's resilience depends on the distribution of the nodes and the ability to maintain functionality even when some nodes are compromised.

GPS Spoofing and Telemetry Spoofing

In a centralized architecture, the entire network is more susceptible to GPS spoofing and telemetry spoofing if the central server is compromised. In this case, an attacker can manipulate the central server to collect or relay false location and telemetry data to the nodes. Since the central server is responsible for processing and disseminating this information, a successful attack can have significant consequences for the entire system, causing UAVs to follow incorrect paths, make erroneous decisions, or fail to complete their missions.

On the other hand, in a decentralized architecture, the impact of GPS spoofing and telemetry spoofing can be mitigated by using a consensus-based approach to verify location and telemetry data. In this approach, multiple nodes within the network share and cross-verify the data they receive, ensuring that a majority of the nodes agree on the data's accuracy before accepting them as valid. This method helps to detect and counteract false data introduced by an attacker, as the malicious information is unlikely to achieve consensus among the nodes. In such a scenario, the impact of the attack is more localized, and the decentralized system's resilience is bolstered by its distributed nature and the ability to validate information collectively.

Impersonation and Man-In-The-Middle Attacks

In a centralized architecture, unauthorized access to the central server through impersonation or MITM attacks can have far-reaching consequences for the entire network. For example, an attacker impersonating a legitimate node can send malicious commands or falsified data to the central server, which then propagates the false information across the network. In an MITM attack, the attacker can intercept and modify communications between the central server and the nodes, potentially causing widespread disruptions or unauthorized access to sensitive information.

Conversely, in a decentralized architecture, the impact of impersonation and MITM attacks can be limited through robust node authentication mechanisms and secure communication channels. For instance, employing public-key cryptography can help ensure the identity of the communicating nodes, making it more challenging for an attacker to impersonate a legitimate node. However, attackers can still target specific nodes or exploit weak points in the network. For example, if an attacker successfully impersonates a node within the decentralized network, they may be able to introduce false data or malicious commands to a limited number of neighboring nodes. Similarly, an MITM attack in a decentralized system might compromise the communications between targeted nodes, but the damage is likely to be more localized due to the distributed nature of the system.

Insider and Selfishness Attacks

In a centralized architecture, malicious insiders can cause more significant damage by compromising the central server. For instance, a malicious employee with access to the central server can manipulate the system by injecting false data, modifying critical configurations, or even deleting essential information. The central server's role as the primary communication hub and data repository for all nodes means that successful insider attacks can have far-reaching consequences for the system's overall functionality and performance.

On the other hand, in a decentralized architecture, the impact of insider and selfishness attacks is more localized. For example, a malicious insider might compromise an individual node and manipulate the data it shares with neighboring nodes, but the overall impact on the network is limited due to its distributed nature. However, attackers can still exploit individual nodes or form colluding groups to undermine the system. In a selfishness attack, a group of nodes may collaborate to consume network resources disproportionately, such as bandwidth or computational power, resulting in degraded performance for other nodes in the network.

Jamming Attack

In a centralized architecture, a jamming attack can have severe consequences if the communication between the central server and nodes is jammed. Since the central server is the primary communication hub for all nodes, disrupting this link can paralyze the entire system, preventing data exchange and coordination between nodes. For example, an attacker could deliberately generate radio frequency interference in the communication channel between the central server and the UAVs, making it impossible for them to receive commands or transmit essential data, effectively rendering the entire system inoperable.

Conversely, in a decentralized architecture, the system is more resilient against jamming attacks. If communication between nodes is disrupted, the system can reroute traffic and continue functioning, though the impact depends on the number of nodes affected and the network topology. For instance, in a mesh topology where nodes have multiple connections to neighboring nodes, a jamming attack may affect only a portion of the network. The unaffected nodes can still communicate with each other and maintain system functionality by bypassing the jammed nodes, albeit with reduced performance. However, if a significant number of nodes are affected or if the network topology is such that jammed nodes isolate certain parts of the network, the overall system functionality could be significantly impacted.

Eavesdropping: Centralized

In a centralized architecture, the system is more susceptible to eavesdropping attacks if the central server's communication is intercepted. Since the central server serves as the primary communication hub and data repository for all nodes, an attacker eavesdropping on this communication can potentially gain access to sensitive information for the entire network. For example, an attacker could intercept the transmission of flight plans, sensor data, or control commands between the central server and UAVs, potentially jeopardizing mission objectives, privacy, or even safety.

Conversely, in a decentralized architecture, encrypted communication channels between nodes can limit the damage caused by eavesdropping attacks. Since communication is distributed across multiple nodes, an attacker would have to intercept several communication channels to obtain comprehensive information about the system. However, attackers can still target individual nodes or exploit weak points in the network. For instance, if a particular node has weak encryption or an outdated security protocol, an attacker could intercept its communication with other nodes, gaining access to a portion of the system's sensitive data.

Poor Link Quality and High Latency

In a centralized architecture, the entire network can be affected if the central server experiences poor link quality or high latency, whether due to natural factors or an attacker's intervention. As the central server is the primary communication hub and data repository for all nodes, any performance issues it faces can have a ripple effect throughout the entire network. An attacker could intentionally degrade the central server's connection, causing delayed or lost data, which in turn reduces the effectiveness of surveillance or reconnaissance missions. For example, a cyberattacker might employ jamming or flooding techniques to disrupt the communication link between the UAVs and the central server, resulting in poor link quality or high latency.

Conversely, in a decentralized architecture, the system can leverage alternative paths and distribute the load to overcome issues related to poor link quality or high latency. Since communication is distributed across multiple nodes, the system can dynamically route data through the network, bypassing congested or slow paths to minimize latency and maintain performance. However, if a significant portion of the network is affected by poor link quality or high latency, either due to natural factors or an attacker's intervention, the entire system can still be impacted. An attacker might target multiple nodes simultaneously, aiming to disrupt the network's overall performance. For instance, if an attacker interferes with multiple nodes in the network by jamming their communication channels or flooding them with malicious traffic, the system may struggle to route data efficiently, leading to an overall degradation in performance and responsiveness.

Rushing Cyberattack

This attack occurs when an attacker node quickly forwards routing packets, attempting to gain control over network routes and disrupt the normal flow of data. In a centralized architecture, a successful rushing attack on the central server could disrupt the entire

network, as it is responsible for managing all communication routes. For example, an attacker might manipulate routing information to redirect traffic through a compromised node or create a network bottleneck, degrading overall performance. On the other hand, decentralized architectures are more resilient to rushing attacks, as they rely on multiple nodes for routing decisions. However, if a significant portion of nodes is compromised, the attacker could still control key network routes, causing performance issues or enabling data interception.

Sybil Attack

An attacker creates multiple fake identities to subvert the trust-based mechanisms in a network, often to manipulate voting or consensus processes. If the central server in a centralized architecture is compromised by a Sybil attack, the entire network could be affected. For example, an attacker might use fake identities to influence decisions made by the central server, such as granting unauthorized access to sensitive data. Conversely, decentralized architectures are more resilient to Sybil attacks due to their consensus mechanisms and trust-based systems. However, if a significant portion of nodes is compromised or the consensus process is subverted, the attacker could still severely impact the system. For instance, an attacker might launch a Sybil attack to manipulate voting results in a decentralized consensus algorithm, undermining the network's integrity.

Wormhole Attack

An attacker creates a low-latency tunnel between two points in the network, effectively "short-circuiting" the normal routing process and capturing or altering the data transmitted through the tunnel. In a centralized architecture, a successful wormhole attack on the central server could severely impact the entire network, as the attacker could intercept and modify data transmitted between the central server and the connected nodes. For example, an attacker could capture sensitive information or inject malicious data into the network. Conversely, decentralized architectures are generally more resilient to wormhole attacks due to their distributed nature. However, if a wormhole attack targets multiple nodes or key network routes, the system could still be severely impacted. For instance, an attacker might compromise several nodes and establish a wormhole to intercept or alter data transmitted between them, disrupting network communication and data integrity.

SPOF Attack

In a centralized architecture, the system is inherently more vulnerable to SPOF attacks due to the reliance on a single central server for data storage, processing, and communication. If an attacker successfully targets and compromises the central server, it can lead to the failure of the entire network. For example, consider a centralized UAV control system where all the UAVs are controlled by a single ground control station. If the ground control station is taken offline by an SPOF attack, all the UAVs may lose their ability to communicate, leading to a potential loss of control or a mission failure.

On the other hand, decentralized architectures are generally more resilient to SPOF attacks because they distribute data and functionality across multiple nodes, reducing the likelihood of a single point of failure. However, key nodes or infrastructure components can still be targeted, and if they are compromised, it may affect the overall system performance. For instance, in a decentralized UAV control system, an attacker might target a specific node responsible for relaying critical information between other nodes. If this node is compromised or taken offline, it could cause a disruption in the communication flow and degrade the performance of the entire system.

Brute Force Attack

In a centralized architecture, the system is more susceptible to brute force attacks if the central server's security measures are breached. Since the central server is the primary communication hub and data repository for all nodes, a successful brute force attack can

have severe consequences for the system's overall functionality and security. For example, if an attacker gains unauthorized access to the central server in a UAV control system, they may be able to manipulate UAV flight paths, intercept sensitive data, or cause disruptions in UAV operations.

On the other hand, decentralized architectures are better at isolating the impact of brute force attacks, as they distribute data and functionality across multiple nodes. However, attackers can still target individual nodes or exploit weak points in the network. For instance, if an attacker successfully brute-forces a node's authentication credentials in a decentralized UAV control system, they might be able to manipulate the data or commands specific to that node. While the impact of this attack may be more localized compared to a centralized system, it could still cause disruptions or security breaches within the affected area.

Leaks of Data Due to Human Mistakes and Data Loss Due to System Crashes

In a centralized architecture, the consequences of data leaks or system crashes can be more significant if the central server is affected. Since the central server serves as the primary communication hub and data repository for all nodes, any data leaks or crashes can potentially impact the entire network. For example, in a UAV control system, a data leak at the central server could expose sensitive information about UAV flight paths or reconnaissance data, while a system crash could lead to a temporary or permanent loss of control over the UAV fleet. The attacker would focus on targeting the central server to maximize the impact. If the attacker can exploit human mistakes, such as exploiting weak passwords, social engineering, or phishing attacks, they can gain unauthorized access to sensitive data or even control over the entire system. Furthermore, by inducing a system crash, the attacker can disrupt the network, causing downtime or loss of functionality, affecting all nodes that rely on the central server.

On the other hand, decentralized architectures can limit the impact of data leaks and system crashes through redundancy and consensus mechanisms. With multiple nodes storing data and processing information, the network can recover from individual node failures or data leaks more easily. However, the system can still be affected if a significant number of nodes or data replicas are compromised. For instance, if a large portion of the nodes in a decentralized UAV control system experiences data leaks or crashes, the overall system performance, data integrity, and security may be compromised, potentially leading to disruptions in UAV operations. From an attacker's perspective, decentralized architectures require more effort and resources to exploit, as the attacker would need to target multiple nodes to cause significant damage. However, both architectures can be made more secure by implementing robust security measures, regular monitoring, and proper access controls to minimize the risks associated with human mistakes and system crashes.

3.27.2. Defense Pattern Comparison

The defense mechanisms and solutions for attacks in centralized and decentralized architectures differ in their approaches and effectiveness based on the unique characteristics of each architecture.

Black Hole and Gray Hole Attacks

From a security developer's perspective, defending against black hole and gray hole attacks in centralized and decentralized architectures involves implementing various proactive and reactive measures to minimize the impact of these attacks and maintain the system's overall security and performance. In a centralized architecture, the security developer should focus on protecting the central server, as it is the primary communication hub and data repository for all nodes. To defend against black hole and gray hole attacks, they can implement the following strategies:

1. Strict access controls: Employ robust authentication mechanisms, such as multifactor authentication, to ensure that only authorized personnel can access the central server.

Additionally, implement the principle of least privilege, granting users the minimum access necessary to perform their tasks.
2. Intrusion detection systems: Deploy intrusion detection systems (IDSs) to monitor network traffic for signs of malicious activity, such as unauthorized access attempts, traffic anomalies, or suspicious patterns indicative of black hole or gray hole attacks.
3. Continuous monitoring: Regularly monitor the central server for signs of compromise, including unexpected changes in system behavior, unexplained data loss, or unauthorized access attempts. Implement real-time alerting systems to notify security personnel of potential incidents.

In a decentralized architecture, the security developer should focus on detecting and isolating malicious nodes, as these attacks target individual nodes within the network. To defend against black hole and gray hole attacks in a decentralized system, they can implement the following strategies:

1. Reputation-based systems: Develop and employ a reputation-based system that evaluates the trustworthiness of nodes based on their past behavior and interactions with other nodes. This system can help identify and isolate malicious nodes, minimizing the impact of black hole and gray hole attacks.
2. Consensus algorithms: Utilize consensus algorithms that require nodes to agree on the validity of data and transactions before they are added to the network. This approach helps detect and mitigate the impact of malicious nodes attempting to propagate false or manipulated data.
3. Node monitoring and isolation: Continuously monitor individual nodes for signs of black hole or gray hole attacks, such as sudden changes in data-forwarding behavior or unexpected drops in network performance. If a node is suspected of being malicious, isolate it from the network to prevent further harm.

Collision Network Attack

From a security developer's perspective, defending against collision network attacks in both centralized and decentralized architectures involves implementing various strategies to optimize communication, manage network traffic, and prevent collisions from impacting the system's overall performance and reliability. In a centralized architecture, the security developer should focus on enhancing the communication between the central server and nodes to minimize the chance of collisions. To defend against collision network attacks in a centralized system, they can implement the following strategies:

1. Optimize communication protocols: Review and optimize communication protocols to ensure that they efficiently manage the flow of data between the central server and nodes. This may include implementing error detection and correction mechanisms, as well as prioritizing critical data transmissions.
2. Load balancing: Introduce load-balancing solutions to distribute network traffic evenly across multiple communication channels, preventing bottlenecks and reducing the likelihood of collisions. Load balancing can also help ensure that the central server's resources are used efficiently, maintaining optimal performance.
3. Traffic management: Implement traffic management solutions that monitor and control the flow of data between the central server and nodes. This may involve using techniques such as traffic shaping, congestion control, and quality of service (QoS) policies to prioritize and manage network traffic, reducing the risk of collisions.

In a decentralized architecture, the security developer should focus on improving communication between nodes and distributing network traffic across multiple channels. To defend against collision network attacks in a decentralized system, they can implement the following strategies:

1. Efficient routing algorithms: Develop and employ routing algorithms that efficiently route data between nodes, taking into account factors such as network topology, latency, and node availability. This can help minimize the chance of collisions by

avoiding congested communication paths and ensuring that data are transmitted along the most efficient routes.
2. Multiple communication channels: Utilize multiple communication channels to distribute network traffic, preventing bottlenecks and reducing the likelihood of collisions. This can be achieved by employing different communication technologies or leveraging multiple frequency bands, ensuring that the network can continue to operate effectively even when one channel is experiencing congestion or interference.

Data Tampering, Modification, and Replay Attacks

From a security developer's perspective, defending against data tampering, modification, and replay attacks requires the implementation of robust security measures to protect data in both centralized and decentralized architectures. The strategies employed in each system type will vary to address their unique challenges. In a centralized architecture, the security developer should focus on securing data in transit and at rest, as well as ensuring the integrity of communication between the central server and nodes. To defend against data tampering, modification, and replay attacks in a centralized system, they can implement the following strategies:

1. Encryption: Employ strong encryption techniques to protect data, both in transit between the central server and nodes, and at rest within the server itself. This can help ensure that even if an attacker intercepts the data, they cannot read or modify them without the appropriate decryption keys.
2. Digital signatures: Utilize digital signatures to verify the authenticity and integrity of data transmitted between the central server and nodes. This can help prevent attackers from tampering with or modifying the data without detection.
3. Secure communication protocols: Implement secure communication protocols, such as transport layer security (TLS) or secure shell (SSH), to establish encrypted communication channels between the central server and nodes. This can help protect against eavesdropping and tampering attacks during data transmission.

In a decentralized architecture, the security developer should focus on ensuring data integrity across all nodes and preventing tampering or modification by malicious nodes. To defend against data tampering, modification, and replay attacks in a decentralized system, they can implement the following strategies:

1. Consensus algorithms: Employ consensus algorithms that require a majority of nodes to agree on the validity and integrity of data before they are accepted into the system. This can help detect and reject tampered or modified data, preventing malicious nodes from manipulating the network.
2. Redundancy measures: Implement redundancy measures, such as data replication and erasure coding, to store multiple copies of data across different nodes. This can help ensure data integrity and prevent the loss or corruption of data due to tampering or modification.
3. Cryptographic techniques: Utilize cryptographic techniques, such as cryptographic hashing and digital signatures, to verify the authenticity and integrity of data transmitted between nodes. This can help prevent attackers from tampering with or modifying the data without detection.

Deauthentication, DDoS, Slowloris, Flooding, and SYN Flood Attacks

For centralized architectures, where the central server is the primary target, security developers should focus on safeguarding the server from these types of attacks. They can implement the following strategies:

1. Firewalls: Deploy advanced firewalls to filter incoming traffic and protect the central server from unauthorized access or malicious traffic patterns associated with DDoS, Slowloris, flooding, and SYN flood attacks.

2. Rate limiting: Implement rate limiting to control the volume of incoming traffic and prevent the central server from being overwhelmed by excessive requests. This can help mitigate the impact of DDoS and SYN flood attacks.
3. Intrusion detection and prevention systems (IDPSs): Employ IDPSs to continuously monitor the network for signs of malicious activity, such as deauthentication or DDoS attacks, and take automated actions to block or mitigate detected threats.

For decentralized architectures, security developers should focus on leveraging the distributed nature of the system to mitigate the impact of these attacks. They can implement the following strategies:

1. Distributed network monitoring: Utilize distributed network monitoring tools to detect unusual traffic patterns or signs of attacks across all nodes in the network. This can help identify potential threats and respond quickly to mitigate their impact.
2. Cooperative filtering: Implement cooperative filtering mechanisms, in which nodes share information about malicious traffic or attackers, allowing the system to collaboratively block or mitigate the attacks. This can help prevent the spread of DDoS, Slowloris, flooding, and SYN flood attacks within the decentralized network.
3. Load balancing: Employ load-balancing techniques to distribute incoming traffic across multiple nodes, preventing any single node from being overwhelmed by malicious requests. This can help mitigate the impact of DDoS and SYN flood attacks on the decentralized system.

GPS Spoofing and Telemetry Spoofing

For centralized architectures, where the central server is responsible for processing and validating location and telemetry data, security developers can implement the following strategies:

1. Cryptographic verification: Use cryptographic techniques, such as digital signatures, to verify the authenticity and integrity of location and telemetry data received by the central server. This ensures that the data have not been tampered with and come from a trusted source.
2. Multisource data fusion: Employ multisource data fusion techniques, which involve combining information from multiple independent sources, such as GPS, GLONASS, and Galileo. This allows the central server to cross-validate location and telemetry data, reducing the likelihood of accepting spoofed data.

For decentralized architectures, where nodes rely on distributed data processing and validation, security developers can implement the following strategies:

1. Consensus-based approach: Utilize a consensus-based approach to verify location and telemetry data by leveraging multiple independent sources. Nodes in the network can share and validate data with their peers, allowing the system to collaboratively determine the accuracy and reliability of the information. This can help mitigate the impact of GPS spoofing and telemetry spoofing attacks by making it more difficult for attackers to manipulate data across the entire network.
2. Redundant data sources: Encourage the use of redundant data sources in the decentralized network, such as multiple GNSS constellations or alternative positioning systems such as LORAN, eLoran, or local positioning systems. This increases the resilience of the network against GPS spoofing and telemetry spoofing attacks by providing additional data sources for validation and cross-referencing.

Impersonation and Man-In-The-Middle Attacks

In order to defend against impersonation and man-in-the-middle attacks, security developers need to adopt different strategies based on the architecture of the system, either centralized or decentralized. For centralized architectures, where a single central server manages and controls the system, security developers can implement the following defense strategies:

1. Strong authentication: Employ robust authentication mechanisms, such as multifactor authentication (MFA), to verify the identity of users, devices, or nodes connecting to the central server. This helps prevent unauthorized access and reduces the risk of impersonation attacks.
2. Authorization mechanisms: Implement granular access control policies to ensure that users, devices, or nodes have the appropriate permissions to access resources or execute actions within the system. This can help limit the impact of a successful impersonation attack.
3. Secure communication channels: Use secure communication protocols, such as SSL/TLS, to protect data transmitted between the central server and its clients. This helps prevent man-in-the-middle attacks by ensuring the confidentiality and integrity of the transmitted data.

For decentralized architectures, where the system relies on multiple nodes working together, security developers can implement the following defense strategies:

1. Node authentication: Utilize robust node authentication mechanisms to verify the identity of nodes within the network. This can help prevent malicious nodes from joining the network and carrying out impersonation attacks.
2. Secure communication channels: Implement secure communication protocols, such as SSL/TLS, between nodes to protect the confidentiality and integrity of the data exchanged within the network. This can help prevent man-in-the-middle attacks by ensuring that attackers cannot intercept or tamper with the transmitted data.
3. Trust-based systems: Employ trust-based systems, such as reputation systems or blockchain technology, to build a secure and resilient network. These systems can help nodes collectively identify and isolate malicious or compromised nodes, reducing the risk of impersonation and man-in-the-middle attacks.

Insider and Selfishness Attacks

To defend against insider and selfishness attacks, security developers need to consider different strategies based on the system architecture, whether centralized or decentralized. For centralized architectures, where a single central server manages and controls the system, security developers can implement the following defense strategies:

1. Strict access controls: Establish granular access control policies to ensure that users, devices, or nodes have the appropriate permissions to access resources or execute actions within the system. This can help limit the impact of insider attacks by restricting the actions and resources available to potentially malicious insiders.
2. User monitoring: Implement user activity monitoring to track and analyze actions performed by users within the system. This can help detect unusual or malicious behavior, such as unauthorized access attempts, data exfiltration, or other signs of insider threats.
3. Anomaly detection systems: Employ advanced anomaly detection systems, such as machine learning algorithms, to identify suspicious patterns or deviations from normal behavior. These systems can help detect potential insider threats and trigger alerts for further investigation or remediation.

For decentralized architectures, where the system relies on multiple nodes working together, security developers can implement the following defense strategies:

1. Reputation-based systems: Use reputation-based systems to build a secure and resilient network. These systems can help nodes collectively identify and isolate malicious or selfish nodes, reducing the risk of insider and selfishness attacks.
2. Collaborative monitoring: Implement collaborative monitoring mechanisms, where nodes within the network share information about the behavior and performance of other nodes. This can help detect selfish or malicious nodes that may be attempting to exploit the system.

3. Consensus algorithms: Employ consensus algorithms, such as proof-of-work or proof-of-stake, to ensure the integrity and consistency of the network. These algorithms can help prevent malicious nodes from dominating or subverting the decision-making process within the network, limiting the impact of insider and selfishness attacks.

Jamming Attack

For centralized architectures, where a single central server manages and controls the system, security developers can implement the following defense strategies:

1. Frequency hopping: Implement frequency-hopping techniques to rapidly switch between multiple frequency channels during communication, making it difficult for an attacker to jam the entire communication link between nodes and the central server.
2. Spread spectrum techniques: Employ spread spectrum techniques, such as direct-sequence spread spectrum (DSSS) or frequency-hopping spread spectrum (FHSS), to distribute the signal across a broader frequency range, making it more resistant to jamming attempts.
3. Alternative communication channels: Establish alternative communication channels or backup links between the central server and nodes to maintain connectivity in case the primary communication channel is jammed.

For decentralized architectures, where the system relies on multiple nodes working together, security developers can implement the following defense strategies:

1. Network topology: Leverage the network's topology to identify alternative communication paths that can bypass the jammed areas. This can help maintain connectivity and ensure the continuous flow of information within the network.
2. Alternative communication paths: Utilize multiple communication paths between nodes to ensure that if one path is jammed, others can still be used to transmit data. This can enhance the network's resilience to jamming attacks.
3. Adaptive routing: Implement adaptive routing algorithms that can dynamically adjust to the network conditions and route data through the network, bypassing jammed areas or congested paths. This can help maintain network performance and minimize the impact of jamming attacks.

Eavesdropping

For centralized architectures, where a single central server manages and controls the system, security developers can implement the following defense strategies:

1. Encryption: Encrypt data transmitted between the central server and nodes to ensure that even if an attacker intercepts the communication, they cannot decipher the information. This can include symmetric encryption algorithms, such as AES, or asymmetric encryption algorithms, such as RSA.
2. Secure communication protocols: Implement secure communication protocols such as TLS or SSL to establish a secure channel between the central server and nodes. These protocols provide encryption, authentication, and data integrity, protecting data from eavesdropping and tampering.

For decentralized architectures, where the system relies on multiple nodes working together, security developers can implement the following defense strategies:

1. Encrypted communication channels: Use encrypted communication channels to protect data exchange between nodes. This can be achieved by employing secure communication protocols such as TLS or end-to-end encryption techniques to ensure that data remain confidential during transmission.
2. Public-key cryptography: Implement public-key cryptography, such as RSA or elliptic curve cryptography (ECC), to enable secure key exchange and data encryption between nodes. Public-key cryptography allows nodes to exchange keys securely, even

in the presence of eavesdroppers, ensuring that data can be encrypted and decrypted only by the intended recipients.

Poor Link and High Latency

For centralized architectures, where a single central server manages and controls the system, security developers can implement the following defense strategies:

1. Optimize network infrastructure: Ensure that the network infrastructure is well designed and adequately maintained to provide reliable and high-performance connectivity. This may involve using high-quality equipment, regular maintenance checks, and ensuring proper network topology.
2. Load balancing: Distribute network traffic across multiple resources or servers to prevent overloading the central server and to ensure smooth communication between nodes. Load-balancing techniques can help manage network congestion and reduce latency, leading to improved system performance.
3. Traffic management solutions: Implement traffic management solutions such as quality of service (QoS) policies to prioritize critical data transmissions and allocate resources effectively. This approach can help reduce latency and maintain optimal network performance.

For decentralized architectures, where the system relies on multiple nodes working together, security developers can implement the following defense strategies:

1. Leverage multiple paths: Utilize the distributed nature of the network to route data through multiple paths, bypassing congested or slow routes. This can help reduce latency and maintain optimal system performance, even when some nodes experience poor link quality or high latency.
2. Distribute traffic: Distribute network traffic across various nodes to avoid overloading specific nodes and to maintain a balanced load across the network. This can help minimize the impact of poor link quality and high latency on the overall system performance.
3. Adaptive routing algorithms: Implement adaptive routing algorithms that can dynamically adjust to changing network conditions, such as congestion or node failures. These algorithms can reroute data packets through alternative paths to minimize latency and maintain optimal network performance.

Rushing Cyberattack, Sybil Attack, and Wormhole Attack

For centralized architectures, where a single central server manages and controls the system, security developers can implement the following defense strategies:

1. Strict access controls: Implement strong authentication and authorization mechanisms to ensure that only authorized users can access the central server and its resources. This can help prevent unauthorized users from launching attacks against the system.
2. Intrusion detection systems: Deploy intrusion detection systems (IDSs) that monitor the network for suspicious activities and detect potential attacks. IDSs can help identify rushing cyberattacks, Sybil attacks, and wormhole attacks in their early stages, allowing security teams to take appropriate countermeasures.
3. Continuous monitoring: Establish continuous monitoring processes to track and analyze the network's performance, security, and overall health. Regular monitoring can help identify and respond to potential threats, including rushing cyberattacks, Sybil attacks, and wormhole attacks, before they can cause significant damage.

For decentralized architectures, where the system relies on multiple nodes working together, security developers can implement the following defense strategies:

1. Reputation-based systems: Implement reputation-based systems that assign trust scores to nodes based on their past behavior and contributions to the network. This

can help identify and isolate malicious nodes involved in rushing cyberattacks, Sybil attacks, and wormhole attacks.
2. Trust-based systems: Employ trust-based systems that evaluate the reliability and trustworthiness of nodes before allowing them to participate in network activities. This can help prevent malicious nodes from launching attacks or influencing the network's operations.
3. Secure routing protocols: Use secure routing protocols that are designed to detect and mitigate attacks such as rushing cyberattacks, Sybil attacks, and wormhole attacks. These protocols typically incorporate mechanisms to verify the authenticity and integrity of routing information, ensuring that data are transmitted securely and reliably across the network.

SPOF Attack

Defending against single point of failure (SPOF) attacks requires different approaches depending on whether the system architecture is centralized or decentralized. For centralized architectures, where a single central server manages and controls the system, security developers can implement the following defense strategies:

1. Redundancy: Introduce redundancy by creating multiple instances of critical system components or services. This can help ensure that if one component fails or is targeted in an attack, the others can continue to function, minimizing the impact of a single point of failure.
2. Backup systems: Establish backup systems to store and maintain copies of essential data and configurations. Regularly updating and testing these backups can help quickly restore the system to normal operation in case of a failure or an attack.
3. Load balancing: Employ load-balancing techniques to distribute network traffic across multiple servers or resources. This can help prevent overloading a single point in the system and increase its overall reliability and performance.

For decentralized architectures, where the system relies on multiple nodes working together, security developers can implement the following defense strategies:

1. Inherent resilience: Decentralized systems are inherently more resilient against SPOF attacks due to their distributed nature. Each node operates independently, ensuring that the failure or compromise of a single node has a limited impact on the overall system.
2. Protect key infrastructure components: Although decentralized systems are more resilient, it is crucial to ensure that key infrastructure components, such as critical nodes or communication links, are protected and secure. Implement strong access controls, encryption, and intrusion detection systems to safeguard these components against potential attacks.
3. Regularly assess and update security measures: Continuously evaluate and update the security measures in place, considering the evolving threat landscape and the unique challenges of decentralized systems. Regular assessments can help identify and address potential vulnerabilities before they can be exploited by attackers.

Brute Force Attack

For centralized architectures, where a single central server manages and controls the system, security developers can implement the following defense strategies:

1. Strong encryption: Employ strong encryption algorithms to protect data stored on the central server, as well as data transmitted between the server and the nodes. This can help safeguard sensitive information and prevent unauthorized access.
2. Complex passwords: Use complex, unique passwords for the central server and all administrative accounts to make it more challenging for attackers to guess or crack the passwords using brute force techniques.

3. Account lockout policies: Implement account lockout policies to limit the number of failed login attempts within a specific timeframe. This can help protect the central server from brute force attacks by temporarily locking out accounts after a certain number of unsuccessful login attempts, reducing the likelihood of an attacker gaining unauthorized access.

For decentralized architectures, where the system relies on multiple nodes working together, security developers can implement the following defense strategies:

1. Strong authentication mechanisms: Utilize strong authentication mechanisms, such as multifactor authentication (MFA) or public-key cryptography, to protect individual nodes from unauthorized access. These mechanisms can help ensure that only authorized users can access the nodes, making brute force attacks more difficult.
2. Secure communication channels: Implement secure communication channels between nodes using encryption and secure protocols, such as SSL/TLS. This can help protect data exchanged between nodes and prevent unauthorized access or eavesdropping by attackers.
3. Regular security assessments: Continuously evaluate and update security measures to address the evolving threat landscape and the unique challenges of decentralized systems. Regular assessments can help identify and mitigate potential vulnerabilities, such as weak passwords or outdated encryption algorithms, before they can be exploited by attackers.

Leaks of Data Due to Human Mistakes and Data Loss Due to System Crash

For centralized architectures, where a single central server manages and controls the system, security developers can implement the following defense strategies:

1. Strict access controls: Enforce strict access controls to limit the number of users who can access sensitive data on the central server. By granting access only to those who require it, the risk of data leaks due to human mistakes can be minimized.
2. User monitoring: Implement user monitoring tools to track user activities and detect potential data leaks or unauthorized access. This can help identify unusual behavior or potential breaches and allow for swift remediation.
3. Data backup and recovery solutions: Develop robust data backup and recovery solutions to protect against data loss due to system crashes. Regularly scheduled backups and reliable recovery mechanisms can help ensure data continuity and minimize downtime in the event of a crash.

For decentralized architectures, where the system relies on multiple nodes working together, security developers can implement the following defense strategies:

1. Redundancy measures: Employ redundancy measures, such as data replication and distributed storage, to ensure that data are stored across multiple nodes. This can help prevent data loss due to system crashes and minimize the impact of human mistakes on the overall system.
2. Consensus algorithms: Implement consensus algorithms to maintain data integrity and consistency across nodes. By ensuring that all nodes agree on the state of the data, it becomes more difficult for individual nodes to introduce errors or tamper with the data.
3. Collaborative monitoring: Use collaborative monitoring tools to track user activities and detect potential data leaks or unauthorized access across the decentralized network. By monitoring the activities of users across the entire system, security developers can more effectively detect and respond to potential data leaks and breaches.

4. Qualitative Analysis of Security Attack

This section presents a qualitative analysis of various cybersecurity attacks and their potential impacts on UAV–cloud centralized systems, focusing on data loss, sensitivity,

latency, delay, network availability, authentication, integrity, and confidentiality. By examining the characteristics and consequences of each attack, we assess their effects on these attributes, offering valuable insights for prioritizing security measures and countermeasures in the context of UAV–cloud systems.

4.1. Delphi Method

In order to implement qualitative analysis, the Delphi method is used for assessing latency and other network attributes of 26 cybersecurity attacks in the context of UAV–cloud systems. The main objective of the study is to obtain expert consensus on the impact of each attack on the selected attributes and assign qualitative values (high, medium, and low) to each. There are several processes:

1. **Define the problem**: The main research question is, "What are the impacts of 26 cybersecurity attacks on latency and other network attributes in UAV–cloud systems, and how can they be qualitatively assessed?"
2. **Select the experts**: We assemble a diverse group of 20 experts with experience in cybersecurity, UAV systems, and cloud computing. The experts are a mix of professionals from academia, industry, and government to cover different perspectives. Each expert has a background in at least one of the following areas: cybersecurity, UAV systems, or cloud computing, and is familiar with network performance metrics.
3. **Develop the questionnaire**: After that, we create a series of questions that allow experts to provide their opinions on the impacts of each cybersecurity attack on the selected network attributes, such as latency, data loss, sensitivity, authentication, integrity, and confidentiality. The purpose of the questionnaire is to obtain qualitative assessments (high, medium, low) for each attribute using rating scales or other mechanisms to capture expert opinions effectively.
4. **First round**: In this step, we distribute the questionnaire to the experts, asking them to assess the impact of each attack on the network attributes in the context of UAV–cloud systems. We also ensure anonymity to promote unbiased opinions.
5. **Discuss and analyze the responses**: In this phase, we discuss the response from the collected first-round responses and analyze them. After that, we summarize the expert opinions on the impacts of the attacks on each network attribute.
6. **Second round**: After analyzing the first-round step, we develop a new questionnaire based on the results from the first round, presenting the summarized assessments and asking experts to reconsider their initial opinions in light of the collective input. Then, we encourage experts to provide any additional insights or explanations to support their assessments.
7. **Iterate**: We continue to analyze the responses and develop new questionnaires for additional rounds, repeating the process until a consensus is reached or diminishing returns are observed. A consensus can be considered achieved when at least 75% of experts agree on the impact assessments. Diminishing returns can be indicated by minimal changes in assessments between rounds.
8. **Compile results**: We summarize the final results, detailing the consensus on the impacts of each cybersecurity attack on latency and the other network attributes in UAV–cloud systems. Furthermore, we also highlight the key differences between attacks and their implications for system security and performance. Table 1 presents a comprehensive overview of the rate of impacts of individual cyberattacks on various network attributes.

Table 1. Impact of a security attack on various attributes.

Attack	Data Loss	Sensitivity	Latency and Delay	Network Availability	Authentication and Integrity	Confidentiality
Black hole	high	medium	high	high	low	low
Collision network	medium	low	high	medium	low	low
Data tampering	medium	high	low	low	high	high
Deauthentication	medium	medium	high	high	medium	medium
DDoS and Slowloris	medium	low	high	high	low	low
Flooding	medium-high	low	high	high	low	low
GPS spoofing	low	medium	low	low	medium	low
Telemetry spoofing	low	high	low	low	high	medium
Gray hole	high	medium	high	high	medium	low
Impersonation	low	high	low	low	high	high
Insider	high	high	low-medium	medium-high	high	high
Jamming	high	medium	high	high	low	low
Eavesdropping	low	high	low	low	low	high
Poor link and high latency	medium	low	high	medium	low	low
Man-in-the-middle	low	high	medium	low	high	high
Modification	low	high	medium	low	high	medium
Replay	low	medium	medium	low	high	low
Rushing cyber	medium	low	medium	medium	low	low
Selfishness	medium	low	medium	medium	low	low
SPOF	high	medium	high	high	medium	medium
Sybil	low	medium	low	medium	high	medium
SYN flood	low	low	high	high	low	low
Wormhole	medium	high	medium	medium	high	high
Brute force	low	high	low	low	high	high
Leaks data (human)	low	high	low	low	high	high
Data loss (system crash)	high	medium	high	high	medium	medium

4.2. Results and Analysis

4.2.1. Black Hole Attack

Data loss: The consensus among experts is that the data loss impact is high due to the intentional dropping of all received packets, which can severely affect the performance and mission success of the UAV–cloud system. **Sensitivity**: The panel agreed that the sensitivity impact is medium, as the attack mainly targets data loss and not the content or confidentiality of the data. However, experts highlighted that the loss of sensitive data could have severe consequences, depending on the specific data type and mission context.

Latency and delay: Experts concluded that latency and delay impacts are high, as dropped packets cause retransmissions and increase the time it takes to transmit data, directly affecting the real-time control and responsiveness of the UAV. **Network availability**: The panel reached a consensus that network availability is highly impacted, as the attack can disrupt communication between the UAV and the cloud, leading to a potential loss of control and system unavailability.

Authentication and integrity: The experts agreed that the impact on authentication and integrity is low, as the attack primarily focuses on dropping packets rather than altering or forging them. However, some experts noted that a sophisticated attacker could potentially combine this attack with other techniques to compromise authentication or integrity. **Confidentiality**: The panel concluded that the confidentiality impact is low, as the attack does not directly target the content or confidentiality of the data. Nevertheless, experts emphasized that this assessment assumes proper encryption and security measures are in place to protect sensitive data.

4.2.2. Collision Network Attack

Data loss: The impact on data loss is considered medium, as the attacker's primary goal is to generate collisions, leading to retransmissions, and the possibility of data loss. However, the extent of data loss depends on the severity and frequency of the collisions. **Sensitivity**: The sensitivity impact is low, as the attack mainly targets data transmission rather than the content or confidentiality of the data. This means that while the delivery of data may be affected, the data themselves remain unaltered and secure.

Latency and delay: The latency and delay impact is high, as collisions result in retransmissions, which can significantly increase the time it takes to transmit data. This can adversely affect the real-time control and responsiveness of UAV systems, hindering mission success. **Network availability**: The impact on network availability is medium, as the attack can disrupt communication, causing intermittent connectivity issues. However, it may not necessarily render the entire UAV–cloud system unavailable, depending on the severity of the attack and the system's resilience.

Authentication and integrity: The impact on authentication and integrity is low, as the attack primarily focuses on causing collisions rather than altering or forging data packets. This means that, while data transmission may be disrupted, the integrity and authentication of the data remain intact. **Confidentiality**: The confidentiality impact is low, as the attack does not directly target the content or confidentiality of the data. Assuming that proper encryption and security measures are in place, the sensitive information within the UAV–cloud system should remain protected even in the face of collision network attacks.

4.2.3. Data Tampering

Data loss: The impact of data loss is considered medium, as the attacker alters the data, potentially rendering them useless and leading to data loss. This can disrupt the functionality and decision-making capabilities of the UAV–cloud system. **Sensitivity**: The sensitivity impact is high, as the attacker can change sensitive information, compromising the confidentiality of the data. This can lead to severe consequences, such as unauthorized access to mission details or the exposure of critical system vulnerabilities.

Latency and delay: The latency and delay impact is low, as data tampering does not directly impact the time it takes to transmit data. However, depending on the severity of

the tampering, additional processing or error-checking mechanisms may slightly increase latency. **Network availability**: The impact on network availability is low, as the attack does not affect the communication link between the UAV and the cloud. The system remains operational, but the integrity and confidentiality of the data are compromised.

Authentication and integrity: The impact on authentication and integrity is high, as the attack aims to compromise the integrity of the data. This can undermine the trustworthiness of the UAV–cloud system and lead to incorrect decision making or unintended consequences. **Confidentiality**: The confidentiality impact is high, as the attack directly targets the content and confidentiality of the data. By tampering with sensitive information, the attacker can gain unauthorized access to critical system components or manipulate the UAV–cloud system for malicious purposes.

4.2.4. Deauthentication Attack

Data loss: The impact of data loss is considered medium, as the attacker forces legitimate users to disconnect, potentially leading to data loss. This can hinder the overall functionality of the UAV–cloud system, affecting both control and data collection. **Sensitivity**: The sensitivity impact is medium, as the attack can disrupt access to sensitive data but does not directly target its content or confidentiality. Although sensitive data may not be directly compromised, the loss of access can have severe consequences depending on the mission context and the nature of the data.

Latency and delay: The impact on latency and delay is high, as the attack can disrupt communication, causing delays and affecting the responsiveness of the UAV. This can result in reduced mission effectiveness and increased operational risks. **Network availability**: The network availability impact is high, as the attack can render the system unavailable to legitimate users. By forcing disconnections, the attacker can effectively disrupt the operation of the UAV–cloud system and its communication with authorized personnel.

Authentication and integrity: The impact on authentication and integrity is medium, as the attack targets the authentication process but does not directly impact data integrity. Although the data remain intact, the disruption of the authentication process can prevent legitimate users from accessing and verifying the information. **Confidentiality**: The confidentiality impact is medium, as the attack can disrupt access to sensitive data but does not directly target their content or confidentiality. The primary concern is the loss of access to sensitive data, which can hinder mission objectives and decision-making processes.

4.2.5. DDos and Slowloris Attack

Data loss: The impact of data loss is considered medium, as the attack aims to exhaust server resources, potentially causing data loss due to congestion and system unavailability. This can hinder the overall functionality of the UAV–cloud system, affecting both control and data collection. **Sensitivity**: The sensitivity impact is low, as the attack does not target the content or confidentiality of the data. The primary concern with this attack is the potential disruption of data transmission, rather than the compromise of sensitive information.

Latency and delay: The impact on latency and delay is high, as the Slowloris attack can cause significant delays in data transmission, affecting the responsiveness of the UAV. This can result in reduced mission effectiveness and increased operational risks. **Network availability**: The network availability impact is high, as the attack can render the system unavailable to legitimate users. By exhausting server resources, the attacker can effectively disrupt the operation of the UAV–cloud system and its communication with authorized personnel.

Authentication and integrity: The impact on authentication and integrity is low, as the attack focuses on exhausting server resources rather than altering or forging data packets. While the attack does not directly impact data integrity, the resulting congestion can hinder legitimate users' access to the system and the verification of information. **Confidentiality**: The confidentiality impact is low, as the Slowloris attack does not directly target the

content or confidentiality of the data. The primary concern is the potential disruption of data transmission and the system's unavailability, rather than the compromise of sensitive information.

4.2.6. Flooding Attack

Data loss: The impact of data loss ranges from medium to high, as the attacker aims to overwhelm the UAV–cloud system with excessive traffic, potentially causing data loss due to congestion and system unavailability. This disruption can affect both the control and data collection capabilities of the UAV–cloud system. **Sensitivity**: The sensitivity impact is low, as the attack does not target the content or confidentiality of the data. The primary concern with this attack is the potential disruption of data transmission, rather than the compromise of sensitive information.

Latency and delay: The impact on latency and delay is high, as the excessive traffic generated during the attack can significantly increase the time it takes to transmit data. This can result in reduced mission effectiveness and increased operational risks for the UAV–cloud system. **Network availability**: The network availability impact is high, as the attack aims to overwhelm the system, making it unavailable to legitimate users. By flooding the UAV–cloud system with traffic, the attacker can effectively disrupt the operation and communication with authorized personnel.

Authentication and integrity: The impact on authentication and integrity is low, as the attack primarily focuses on flooding the system with traffic rather than altering or forging data packets. While the attack does not directly impact data integrity, the resulting congestion can hinder legitimate users' access to the system and the verification of information. **Confidentiality**: The confidentiality impact is low, as the flooding attack does not directly target the content or confidentiality of the data. The primary concern is the potential disruption of data transmission and the system's unavailability, rather than the compromise of sensitive information.

4.2.7. GPS Spoofing Attack

Data loss: The impact on data loss is low since the attack targets the UAV's navigation system rather than causes data loss or corruption in the communication link. The primary focus of GPS spoofing is to manipulate the UAV's position, which does not directly result in the loss of data being transmitted between the UAV and the cloud. **Sensitivity**: The attack has a medium impact on sensitivity, as the attacker can manipulate the UAV's position, potentially compromising the mission and revealing sensitive information about the UAV's location and trajectory. The risk stems from the possibility that the attacker may gain insights into the UAV's operations or hinder its ability to complete its mission successfully.

Latency and delay: The impact on latency and delay is low, as GPS spoofing does not affect the time it takes to transmit data or the responsiveness of the UAV. Although the attack primarily focuses on the UAV's navigation system, it does not introduce additional delays in data transmission between the UAV and the cloud. **Network availability**: The attack has a low impact on network availability since it does not directly affect the communication link between the UAV and the cloud. The objective of GPS spoofing is to deceive the UAV's navigation system, which is separate from the communication link responsible for network availability.

Authentication and integrity: GPS spoofing attack has a medium impact on authentication and integrity, as it primarily focuses on manipulating the UAV's navigation data rather than altering or forging data packets. A successful GPS spoofing attack can undermine the integrity of the UAV's position and mission data, potentially leading to incorrect decisions or actions based on inaccurate location information. **Confidentiality**: The attack has a low impact on confidentiality, as it does not directly target the content or confidentiality of the data transmitted between the UAV and the cloud. While the attacker might gain insights into the UAV's location and trajectory, the content of data

being transmitted remains unaffected, and proper encryption and security measures can help maintain confidentiality.

4.2.8. Telemetry Spoofing Attack

Data loss: The impact on data loss is low, as the attack primarily focuses on altering or forging telemetry data rather than causing data loss. The attacker's goal is to manipulate the telemetry data, not to disrupt data transmission or delete critical data. **Sensitivity**: The sensitivity impact is high, as telemetry data are critical for UAV operations, and their manipulation can compromise the mission and reveal sensitive information about the UAV's status and control. Successful telemetry spoofing attacks can jeopardize the safety and success of the operation, making it crucial to protect these data from malicious actors.

Latency and delay: The impact on latency and delay is low, as telemetry spoofing does not directly impact the time it takes to transmit data. The attack specifically targets the UAV's telemetry data, without directly affecting data transmission and communication processes. **Network availability**: The network availability impact is low, as the attack does not directly affect the communication link between the UAV and the cloud. The primary focus of the attacker is on manipulating the UAV's telemetry data, rather than disrupting the overall network availability.

Authentication and integrity: The impact on authentication and integrity is high, as the attacker aims to compromise the integrity of the telemetry data by altering or forging them. Successfully manipulated telemetry data can result in incorrect or misleading information being used for UAV control and decision making, potentially leading to mission failures or unexpected operational outcomes. **Confidentiality**: The confidentiality impact is medium, as the attacker directly targets the content of the telemetry data, potentially revealing sensitive information. This could lead to unauthorized access or exposure of sensitive UAV operational details, making it essential to secure the confidentiality of telemetry data.

4.2.9. Gray Hole Attack

Data loss: The impact on data loss is high, as the attacker selectively drops packets, leading to data loss and communication disruption. This can severely affect the performance and mission success of the UAV–cloud system. **Sensitivity**: The sensitivity impact is medium, as the attack targets data loss rather than the content or confidentiality of the data. However, the loss of sensitive data could have severe consequences, depending on the specific data type and mission context.

Latency and delay: The impact on latency and delay is high, as selectively dropped packets cause retransmissions and increase the time it takes to transmit data. This directly affects the real-time control and responsiveness of the UAV. **Network availability**: The network availability impact is high, as the attack can disrupt communication and render the system unavailable. This can lead to a loss of control and system unavailability, severely hindering the UAV–cloud system's operation.

Authentication and integrity: The impact on authentication and integrity is medium, as the attacker selectively drops packets rather than altering or forging them. However, the attacker could potentially combine this attack with other techniques to compromise authentication or integrity, posing a more significant threat to the UAV–cloud system. **Confidentiality**: The confidentiality impact is low, as the attack does not directly target the content or confidentiality of the data. However, this assessment assumes that proper encryption and security measures are in place to protect sensitive data from unauthorized access and exposure.

4.2.10. Impersonation Attack

Data loss: The impact on data loss is low, as the attacker primarily focuses on pretending to be a legitimate entity rather than causing data loss. However, the unauthorized access gained through this attack could potentially lead to data loss if the attacker decides

to delete or tamper with the data. **Sensitivity**: The sensitivity impact is high, as the attacker can gain unauthorized access to sensitive data and resources by impersonating a legitimate entity. This can lead to severe consequences, such as exposing the UAV's location, mission details, or other sensitive information.

Latency and delay: The impact on latency and delay is low, as impersonation attacks do not directly impact the time it takes to transmit data. However, indirect consequences of unauthorized access might affect the responsiveness of the UAV–cloud system. **Network availability**: The network availability impact is low, as the attack does not directly affect the communication link between the UAV and the cloud. Nonetheless, the unauthorized access could potentially lead to system disruptions or resource unavailability.

Authentication and integrity: The impact on authentication and integrity is high, as the attacker aims to bypass the authentication process and compromise the integrity of the system. This can undermine the overall security and trustworthiness of the UAV–cloud system. **Confidentiality**: The confidentiality impact is high, as the attacker can gain unauthorized access to sensitive data by impersonating a legitimate entity. This can lead to unauthorized disclosure of confidential information, placing the UAV–cloud system and its mission at risk.

4.2.11. Insider Attack

Data loss: The impact on data loss is high, as the attacker, being an insider, has privileged access to the system and can potentially cause significant data loss and damage. This could severely affect the performance and mission success of the UAV–cloud system. **Sensitivity**: The sensitivity impact is high, as the attacker has privileged access to sensitive data and can compromise its confidentiality. This can lead to unauthorized disclosure of confidential information, placing the UAV–cloud system and its mission at risk.

Latency and delay: The impact on latency and delay is low to medium, as it depends on the attacker's actions. An insider attacker could potentially disrupt communication or cause delays in data transmission, affecting the responsiveness of the UAV–cloud system. **Network availability**: The network availability impact is medium to high, as the attacker can potentially disrupt the communication link and render the system unavailable. This could result in a loss of control and system unavailability, which can have serious consequences on the UAV–cloud system's operations.

Authentication and integrity: The impact on authentication and integrity is high, as the attacker already has authenticated access to the system and can compromise its integrity by altering or deleting data. This can undermine the overall security and trustworthiness of the UAV–cloud system. **Confidentiality**: The confidentiality impact is high, as the attacker has privileged access to sensitive data and can exploit them for malicious purposes. This can lead to unauthorized disclosure of confidential information, placing the UAV–cloud system and its mission at risk.

4.2.12. Jamming Attack

Data loss: The impact on data loss is high, as the attacker seeks to disrupt the communication link by flooding the network with noise, causing significant data loss. This can severely impact the performance and mission success of the UAV–cloud system. **Sensitivity**: The sensitivity impact is medium, as the attack primarily targets data loss rather than the content or confidentiality of the data. However, the loss of sensitive data could have severe consequences, depending on the specific data type and mission context.

Latency and delay: The impact on latency and delay is high, as jamming disrupts communication links and causes retransmissions, increasing the time it takes to transmit data. This directly affects the real-time control and responsiveness of the UAV–cloud system. **Network availability**: The network availability impact is high, as the attack can render the communication link between the UAV and the cloud unavailable. This may lead to a loss of control and system unavailability, which can have serious consequences for the UAV–cloud system's operations.

Authentication and integrity: The impact on authentication and integrity is low, as the attack primarily focuses on disrupting the communication link rather than altering or forging data packets. However, some experts noted that a sophisticated attacker could potentially combine this attack with other techniques to compromise authentication or integrity. **Confidentiality**: The confidentiality impact is low, as the attack does not directly target the content or confidentiality of the data. Nevertheless, experts emphasized that this assessment assumes proper encryption and security measures are in place to protect sensitive data.

4.2.13. Eavesdropping Attack

Data loss: The impact on data loss is low, as the attacker primarily focuses on intercepting data rather than causing data loss. Eavesdropping attackers seek to access information being transmitted between the UAV and the cloud, not to disrupt or corrupt the data themselves. **Sensitivity**: The sensitivity impact is high, as the attacker can access sensitive information transmitted between the UAV and the cloud. Unauthorized access to sensitive data can compromise the mission and have serious consequences, depending on the specific data type and mission context.

Latency and delay: The impact on latency and delay is low, as eavesdropping does not directly impact the time it takes to transmit data. The primary objective of an eavesdropping attacker is to monitor the data being transmitted, not to introduce delays or disrupt the communication process. **Network availability**: The network availability impact is low, as the attack does not directly affect the communication link between the UAV and the cloud. Eavesdropping attackers are primarily interested in intercepting data, not in causing disruptions to the communication link or network availability.

Authentication and integrity: The impact on authentication and integrity is low, as the attacker aims to intercept and monitor the data rather than altering or forging them. While eavesdropping attacks do not inherently compromise data integrity, they highlight the importance of robust authentication mechanisms to protect against unauthorized access. **Confidentiality**: The confidentiality impact is high, as the attacker can gain unauthorized access to sensitive data by intercepting them during transmission. The primary goal of an eavesdropping attack is to access confidential information, which can significantly compromise the data's confidentiality and lead to severe consequences for the UAV–cloud system and its operations.

4.2.14. Poor Link Quality and High Latency Attack

Data loss: This attack has a medium impact on data loss, as poor link quality can lead to packet loss and data corruption. The degraded communication quality can result in incomplete or lost data, which could affect the performance and mission success of the UAV–cloud system. **Sensitivity**: The sensitivity impact is low, as the attack does not directly target the content or confidentiality of the data. The primary objective of this attack is to disrupt the communication link, not to compromise sensitive information.

Latency and delay: The impact on latency and delay is high, as poor link quality increases the time it takes to transmit data due to retransmissions and delays. This can affect the overall responsiveness and efficiency of the UAV–cloud system, which might have adverse consequences for mission-critical operations. **Network availability**: The network availability impact is medium, as the attack can degrade the communication link between the UAV and the cloud but not render it completely unavailable. The degraded link can result in reduced system performance and challenges in maintaining system availability.

Authentication and integrity: The impact on authentication and integrity is low, as the attack primarily focuses on degrading the communication link rather than altering or forging data packets. The objective is to disrupt the communication process rather than compromise the data's integrity. **Confidentiality**: The confidentiality impact is low, as the attack does not directly target the content or confidentiality of the data. The focus of the

attack is on disrupting the communication link, not on accessing or compromising sensitive information during transmission.

4.2.15. Man-In-the-Middle Attack

Data loss: The impact on data loss is low, as the attacker primarily focuses on intercepting and potentially altering data rather than causing data loss. The primary objective of this attack is to gain unauthorized access to the data transmitted between the UAV and the cloud, rather than to delete or corrupt the data. **Sensitivity**: The attack has a high impact on sensitivity, as the attacker can access and potentially manipulate sensitive information transmitted between the UAV and the cloud. The interception and possible tampering of sensitive data can lead to severe consequences, depending on the specific data type and mission context.

Latency and delay: The impact on latency and delay is medium, as the attacker might introduce additional delay when intercepting and forwarding data. The presence of the attacker in the communication process can cause performance degradation and slower response times, which may affect the overall efficiency of the UAV–cloud system. **Network availability**: The attack has a low impact on network availability as it does not directly affect the communication link between the UAV and the cloud. The primary goal of the attacker is to intercept and manipulate data, not to disrupt the communication link itself.

Authentication and integrity: Man-in-the-middle attacks have a high impact on authentication and integrity, as the attacker aims to intercept, monitor, and potentially alter the data, compromising their integrity. The possibility of data tampering during transmission raises concerns about the trustworthiness of the data received by the UAV–cloud system. **Confidentiality**: The attack has a high impact on confidentiality, as the attacker can gain unauthorized access to sensitive data by intercepting and potentially altering them during transmission. This unauthorized access can compromise the confidentiality of the data and potentially lead to severe consequences for the UAV–cloud system and its operations.

4.2.16. Modification Attack

Data loss: The impact on data loss is low, as the attacker primarily focuses on altering data rather than causing data loss. In this type of attack, the attacker's main objective is to tamper with the data transmitted between the UAV and the cloud, rather than to delete or corrupt the data. **Sensitivity**: Modification attacks have a high impact on sensitivity, as the attacker can compromise the integrity of sensitive information transmitted between the UAV and the cloud. The potential tampering of sensitive data may lead to severe consequences, depending on the specific data type and mission context.

Latency and delay: The impact on latency and delay is medium, as the attacker might introduce additional delay when intercepting and modifying data. The presence of the attacker in the communication process can cause performance degradation and slower response times, which may affect the overall efficiency of the UAV–cloud system. **Network availability**: The attack has a low impact on network availability since it does not directly affect the communication link between the UAV and the cloud. The primary goal of the attacker is to intercept and manipulate data, not to disrupt the communication link itself.

Authentication and integrity: Modification attacks have a high impact on authentication and integrity, as the attacker aims to intercept and alter the data, compromising their integrity. The possibility of data tampering during transmission raises concerns about the trustworthiness of the data received by the UAV–cloud system. **Confidentiality**: The attack has a medium impact on confidentiality, as the attacker can gain unauthorized access to sensitive data by intercepting and modifying them during transmission. Although the primary focus is on data manipulation, unauthorized access to sensitive information can still compromise the confidentiality of the data and potentially lead to negative consequences for the UAV–cloud system and its operations.

4.2.17. Replay Attack

Data loss: The impact on data loss is low, as the attacker primarily focuses on retransmitting previously captured data rather than causing data loss. In this type of attack, the attacker's main objective is to reuse old data to deceive the UAV–cloud system, rather than to delete or corrupt the data. **Sensitivity**: Replay attacks have a medium impact on sensitivity, as the attacker can compromise the integrity of the system by replaying old data, potentially leading to incorrect decisions or actions. The potential misuse of previously captured data may cause the UAV–cloud system to operate based on outdated or invalid information, affecting its performance and mission success.

Latency and delay: The impact on latency and delay is medium, as the attacker might introduce additional delay when intercepting and retransmitting data. The presence of the attacker in the communication process can cause performance degradation and slower response times, which may affect the overall efficiency of the UAV–cloud system. **Network availability**: The attack has a low impact on network availability since it does not directly affect the communication link between the UAV and the cloud. The primary goal of the attacker is to deceive the system by replaying old data, not to disrupt the communication link itself.

Authentication and integrity: Replay attacks have a high impact on authentication and integrity, as the attacker aims to compromise the integrity of the system by replaying old data. The possibility of data misuse during transmission raises concerns about the trustworthiness of the data received by the UAV–cloud system, potentially affecting its operation and decision-making processes. **Confidentiality**: The attack has a low impact on confidentiality, as it does not directly target the content or confidentiality of the data. Although the primary focus is on replaying previously captured data, maintaining data confidentiality remains crucial for the overall security of the UAV–cloud system and its operations.

4.2.18. Rushing Cyberattack

Data loss: The impact on data loss is medium, as the attacker aims to quickly forward data packets to gain an advantage in routing decisions, potentially causing some data loss. This type of attack can lead to the misdirection or loss of data packets, affecting the overall performance and mission success of the UAV–cloud system. **Sensitivity**: The attack has a low impact on sensitivity, as it does not directly target the content or confidentiality of the data. The attacker's primary objective is to manipulate routing decisions, which may indirectly affect the system's operation but not the sensitivity of the data themselves.

Latency and delay: The impact on latency and delay is medium, as the attacker might introduce additional delay when rushing the data-forwarding process. The attacker's interference in the routing process may cause performance degradation and slower response times, affecting the overall efficiency of the UAV–cloud system. **Network availability**: The attack has a medium impact on network availability, as it can impact the communication link between the UAV and the cloud, causing potential routing disruptions. The disruption of the communication link can affect the stability and resilience of the system, potentially impacting its availability and mission success.

Authentication and integrity: The impact on authentication and integrity is low, as the attack primarily focuses on forwarding data packets quickly rather than altering or forging them. While the attacker's intent is to influence routing decisions, the integrity of the data packets themselves is not directly targeted or compromised. **Confidentiality**: The attack has a low impact on confidentiality, as it does not directly target the content or confidentiality of the data. The attacker's focus on manipulating routing decisions rather than the data content means that the confidentiality of the data is not directly at risk, although maintaining data confidentiality remains important for overall system security.

4.2.19. Selfishness Attack

Data loss: The impact on data loss is medium, as the attacker aims to save its resources by not forwarding data packets, potentially causing some data loss. This type of attack can lead to the misdirection or loss of data packets, affecting the overall performance and mission success of the UAV–cloud system. **Sensitivity**: The attack has a low impact on sensitivity, as it does not directly target the content or confidentiality of the data. The attacker's primary objective is to conserve its resources, which may indirectly affect the system's operation but not the sensitivity of the data themselves.

Latency and delay: The impact on latency and delay is medium, as the attacker might introduce additional delay by not forwarding data packets. The selfish behavior of the attacker may cause performance degradation and slower response times, affecting the overall efficiency of the UAV–cloud system. **Network availability**: The attack has a medium impact on network availability, as it can impact the communication link between the UAV and the cloud, causing potential routing disruptions. The disruption of the communication link can affect the stability and resilience of the system, potentially impacting its availability and mission success.

Authentication and integrity: The impact on authentication and integrity is low, as the attack primarily focuses on saving resources rather than altering or forging data packets. While the attacker's intent is to conserve resources, the integrity of the data packets themselves is not directly targeted or compromised. **Confidentiality**: The attack has a low impact on confidentiality, as it does not directly target the content or confidentiality of the data. The attacker's focus on conserving resources rather than the data content means that the confidentiality of the data is not directly at risk, although maintaining data confidentiality remains important for overall system security.

4.2.20. SPOF Cyberattack

Data loss: The impact on data loss is high, as the attacker targets a single point of failure within the system, potentially causing a significant amount of data loss if successful. The disruption caused by this type of attack could lead to substantial data loss, jeopardizing the overall operation of the UAV–cloud system and its mission objectives. **Sensitivity**: The impact on sensitivity is medium, as the attack mainly affects data loss and not the content or confidentiality of the data. However, the loss of sensitive data could have severe consequences, particularly if it involves mission-critical information or proprietary data. The impact on sensitivity depends on the specific data affected and the context of the system's operation.

Latency and delay: The impact on latency and delay is high, as the attack can cause significant delays in data transmission if a critical component of the system is compromised. SPOF cyberattacks could lead to the disruption or complete failure of data transmission, significantly affecting the overall responsiveness and efficiency of the UAV–cloud system. **Network availability**: The impact on network availability is high, as the attack can render the system unavailable if a single point of failure is successfully targeted. The loss of a critical component could lead to network downtime, causing disruptions in communication and potentially rendering the entire system inoperable.

Authentication and integrity: The impact on authentication and integrity is medium, as the attacker might compromise the integrity of the system by targeting a single point of failure. If a critical component responsible for authentication or integrity checks is compromised, it could lead to unauthorized access or data corruption, undermining the trustworthiness of the UAV–cloud system. **Confidentiality**: The impact on confidentiality is medium, as the attacker can potentially gain unauthorized access to sensitive data by compromising a single point of failure. If a component responsible for data encryption or access control is targeted, it could expose sensitive information to unauthorized parties, placing the confidentiality of the data at risk.

4.2.21. Sybil Attack

Data loss: The impact on data loss is low, as the attacker primarily focuses on creating multiple fake identities rather than causing data loss. The main objective of a Sybil attack is to undermine the reputation system or disrupt consensus mechanisms, rather than directly affecting data transmission or storage. **Sensitivity**: The impact on sensitivity is medium, as the attacker can use the fake identities to gain unauthorized access to sensitive information or disrupt the UAV–cloud system. While the attack itself does not directly target the content of the data, the infiltration of fake identities could lead to unauthorized access or manipulation of sensitive information, depending on the system's vulnerabilities.

Latency and delay: The impact on latency and delay is low, as the attack does not directly introduce additional latency or delay in data transmission. However, the indirect effects of a Sybil attack, such as the disruption of consensus mechanisms or the skewing of reputation systems, could lead to a less efficient UAV–cloud system, potentially affecting the overall responsiveness. **Network availability**: The impact on network availability is medium, as the attack can impact the communication link between the UAV and the cloud, causing potential routing disruptions or affecting consensus mechanisms. If left undetected, a Sybil attack could undermine the UAV–cloud system's stability, leading to a less reliable network and degraded performance.

Authentication and integrity: The impact on authentication and integrity is high, as the attacker aims to compromise the system by creating fake identities and potentially forging data or altering routing decisions. The presence of multiple fake identities can undermine the trustworthiness of the system, leading to incorrect decisions or actions based on manipulated data or reputation scores. **Confidentiality**: The impact on confidentiality is medium, as the attacker can potentially gain unauthorized access to sensitive data by using fake identities to infiltrate the system. While the attack itself does not directly target data confidentiality, the attacker could exploit the fake identities to gain access to restricted information, placing the confidentiality of the data at risk.

4.2.22. SYN Flood Attack

Data loss: The impact on data loss is low, as the attacker primarily focuses on overwhelming the target with SYN requests rather than causing data loss. The main objective of a SYN flood attack is to consume resources and disrupt the availability of the target system, rather than directly affecting data transmission or storage. **Sensitivity**: The impact on sensitivity is low, as the attack does not directly target the content or confidentiality of the data. While the attack itself may not jeopardize sensitive information, organizations should remain vigilant and maintain comprehensive security measures to safeguard sensitive data from potential threats.

Latency and delay: The impact on latency and delay is high, as the attack can cause significant delays in data transmission by overwhelming the target with SYN requests. The system's ability to process legitimate requests may be significantly hindered, leading to poor performance and unresponsiveness. **Network availability**: The impact on network availability is high, as the attack can render the target system unavailable due to resource exhaustion. The overwhelming flood of SYN requests consumes available resources, potentially causing the system to become unresponsive or even crash.

Authentication and integrity: The impact on authentication and integrity is low, as the attack primarily focuses on overwhelming the target system rather than altering or forging data packets. Although SYN flood attacks do not directly compromise data authentication or integrity, organizations should ensure that robust authentication and data integrity mechanisms are in place to protect against potential threats. **Confidentiality**: The impact on confidentiality is low, as the attack does not directly target the content or confidentiality of the data. While the attack itself does not compromise data confidentiality, organizations should remain proactive in ensuring data encryption and implementing access control measures to protect sensitive information.

4.2.23. Wormhole Attack

Data loss: The impact on data loss is medium, as the attacker aims to create a wormhole tunnel to forward data packets. Data loss can occur if the tunnel is disrupted or data packets are dropped. The primary objective of a wormhole attack is to intercept and reroute data, potentially causing data loss in the process. **Sensitivity**: The impact on sensitivity is high, as the attacker can compromise the integrity and confidentiality of the data by intercepting and forwarding data packets through the wormhole tunnel. The attacker has the ability to access and potentially alter sensitive data, posing a significant risk to the affected system.

Latency and delay: The impact on latency and delay is medium, as the attacker might introduce additional delay when intercepting and forwarding data packets through the wormhole tunnel. This redirection of data packets could cause delays in data transmission, impacting the performance of the system. **Network availability**: The impact on network availability is medium, as the attack can impact the communication link between the UAV and the cloud, causing potential routing disruptions. The rerouting of data packets through the wormhole tunnel can lead to confusion in the network, affecting its availability and stability.

Authentication and integrity: The impact on authentication and integrity is high, as the attacker aims to compromise the integrity of the system by intercepting and forwarding data packets through the wormhole tunnel. The unauthorized access to data packets allows the attacker to potentially alter or forge data, posing a significant risk to the system's data integrity. **Confidentiality**: The impact on confidentiality is high, as the attacker can gain unauthorized access to sensitive data by intercepting and forwarding data packets through the wormhole tunnel. The attacker's ability to access sensitive information can compromise the confidentiality of the data, highlighting the importance of employing robust security measures to protect against wormhole attacks.

4.2.24. Brute Force Attack

Data loss: The impact on data loss is low, as the attacker primarily focuses on gaining unauthorized access to the system rather than causing data loss. The main goal of a brute force attack is to break into a system, which may not result in data loss unless the attacker chooses to delete or modify data. **Sensitivity**: The impact on sensitivity is high, as the attacker can compromise the integrity and confidentiality of the data if successful in gaining unauthorized access. Once the attacker has access to the system, sensitive data are at risk, emphasizing the need for robust security measures to protect sensitive information.

Latency and delay: The impact on latency and delay is low, as the attack does not directly introduce additional latency or delay in data transmission. Brute force attacks target authentication mechanisms rather than affecting data transmission, so their impact on latency and delay is minimal. **Network availability**: The impact on network availability is low, as the attack does not directly affect the communication link between the UAV and the cloud. While a brute force attack might not directly impact network availability, it is essential to monitor for any unusual network behavior that may signal an ongoing attack.

Authentication and integrity: The impact on authentication and integrity is high, as the attacker aims to compromise the system by gaining unauthorized access. Successful brute force attacks can result in unauthorized access, placing the system's authentication and data integrity at risk. **Confidentiality**: The impact on confidentiality is high, as the attacker can potentially gain unauthorized access to sensitive data by successfully guessing passwords or encryption keys. Gaining access to sensitive data may allow the attacker to view, modify, or steal confidential information, highlighting the importance of strong encryption and access control measures to protect data confidentiality.

4.2.25. Leaks of Data Due to Human Mistake

Data loss: The impact on data loss is low, as the attacker primarily focuses on exploiting human mistakes rather than causing data loss. The main objective of this attack is to

take advantage of human errors, which might not result in data loss unless the exploited mistakes lead to data exposure or deletion. **Sensitivity**: The impact on sensitivity is high, as the attacker can compromise the integrity and confidentiality of the data by exploiting human mistakes. Once the attacker takes advantage of human errors, sensitive data may be at risk, emphasizing the need for employee training and strong security policies to protect sensitive information.

Latency and delay: The impact on latency and delay is low, as the attack does not directly introduce additional latency or delay in data transmission. Human error exploitation targets weaknesses in human behavior rather than affecting data transmission, so its impact on latency and delay is minimal. **Network availability**: The impact on network availability is low, as the attack does not directly affect the communication link between the UAV and the cloud. While human error exploitation might not directly impact network availability, it is essential to monitor for any unusual network behavior that may signal an ongoing attack.

Authentication and integrity: The impact on authentication and integrity is high, as the attacker aims to compromise the system by exploiting human mistakes and potentially gaining unauthorized access. Successful exploitation of human errors can result in unauthorized access, placing the system's authentication and data integrity at risk. **Confidentiality**: The impact on confidentiality is high, as the attacker can potentially gain unauthorized access to sensitive data by exploiting human mistakes. Gaining access to sensitive data may allow the attacker to view, modify, or steal confidential information, highlighting the importance of strong encryption and access control measures to protect data confidentiality.

4.2.26. Data Loss Due to System Crash

Data loss: The impact of data loss is high, as a system crash can result in the loss of critical control data, telemetry, or payload data. System crashes can lead to data loss, which may have serious consequences, especially if the lost data are vital to the UAV's operation or mission. **Sensitivity**: The impact on sensitivity is medium, as data loss is unintentional and not directly targeted. Although system crashes are not intentional, the loss of sensitive data during a crash could have severe consequences, depending on the specific data type and mission context.

Latency and delay: The impact on latency and delay is high, as a system crash can cause significant delays in data transmission, directly affecting the real-time control and responsiveness of the UAV. A system crash may interrupt data transmission, leading to latency and delay issues, which may impact the UAV's performance and mission success. **Network availability**: The impact on network availability is high, as a system crash can render the system unavailable, leading to a potential loss of control and system unavailability. System crashes may cause a loss of network connectivity, affecting the UAV's ability to communicate with the cloud or other systems.

Authentication and integrity: The impact on authentication and integrity is medium, as a system crash might compromise the integrity of the system, but the crash itself is not an intentional attack. While a system crash is not a targeted attack, it can still result in data corruption or other integrity issues due to the sudden loss of system functionality. **Confidentiality**: The impact on confidentiality is medium, as a system crash can potentially expose sensitive data if proper encryption and security measures are not in place to protect it. System crashes might lead to the exposure of sensitive data if security measures such as encryption or access controls are insufficient or fail during the crash.

5. Complexity Analysis

Complexity analysis is a theoretical evaluation, and actual performance may vary depending on factors such as hardware, programming languages, and compilers. It is often referred to as time and space complexity analysis, and it is used to evaluate the efficiency of an algorithm or a piece of code. It provides an estimate of the amount of computational resources, such as time (execution speed) and memory (space), that an algorithm will

consume as the input size increases. Complexity analysis helps developers choose the best algorithms for their specific use cases and helps them identify potential bottlenecks or areas for optimization in their code.

In this paper, we expose several complexity analyses of security solutions based on our implementation of defense pattern code in *node.js*, and readers can download our implementation in *github* [44]. Based on Table 2, we find the time and space complexity of each solution, and the analysis can be used by the community as a consideration to implement protection and data security modules in UAV–cloud-based application. The high performance of a prevention system can lead to a higher chance of security detection; therefore, the analysis of complexity can be an appropriate analysis for practical reasons.

Table 2. Complexity analyses of security solutions.

Defense For	Time Complexity	Space Complexity
Black hole	O(mn + n log n)	O(m + n)
Collision network	O(1)	O(1)
Data tampering	O(1)	O(n)
Deauthentication	O(n)	O(n)
DDoS and Slowloris	O(1)	O(1)
Flooding	O(n)	O(m)
GPS spoofing	O(n)	O(1)
Telemetry spoofing	O(n)	O(n)
Gray hole	O(n^2)	O(n)
Impersonate	O(n)	O(1)
Insider	O(n^2)	O(n)
Jamming	O(n)	O(1)
Eavesdropping	O(n)	O(n)
Poor link and latency	O(n log n)	O(n)
Man-in-the-middle	O(n)	O(1)
Modification	O(n)	O(n)
Replay	O(n)	O(n)
Rushing attack	O(n)	O(1)
SPOF	O(1)	O(n)
Sybil	O(n)	(1)
SYN flood	O(n)	O(n)
Wormhole	O(n^2)	O(n^2)
Brute force	O(n)	O(n)
Leaks of data by human	O(n)	O(n)
Data loss due to system	O(n)	O(n)
Selfishness	O(n)	O(n)

Based on the time and space complexity table provided, the following recommendations can be made with respect to trade-offs between performance and cost for different types of attacks:

1. For defense patterns that have a constant time and space complexity (O(1)), such as collision network, DDoS and Slowloris, GPS spoofing, jamming, man-in-the-middle, rushing attack, and SPOF, the focus should be on creating cost-effective and lightweight defenses that do not compromise system performance.
2. For defense solutions that have a linear time and space complexity (O(n)), such as data tampering, flooding, impersonate, eavesdropping, modification, replay, brute force, leaks of data by human, data loss due to system, and selfishness, the focus should be on creating defenses that strike a balance between effectiveness and system performance. In particular, security developers should aim to optimize the time and space complexity of their defenses to ensure that they are efficient and cost-effective while still providing adequate protection against these types of attacks.

3. For defense patterns that have a quadratic time and space complexity ($O(n^2)$), such as gray hole and wormhole, the focus should be on creating effective defenses that can prevent or mitigate these types of attacks. However, because these attacks require significant resources to execute, defenses with higher time and space complexity may be warranted in order to effectively protect against them.
4. For attacks that have a combination of linear and logarithmic time complexity (O(n log n)), such as poor link and latency, the focus should be on creating efficient defenses that can minimize the impact of these attacks on system performance. In particular, security developers should aim to optimize the time complexity of their defenses to ensure that they are efficient and cost-effective while still providing adequate protection against these types of attacks.

Overall, the recommendations for trade-offs between performance and cost will depend on the specific needs of the organization and the network being secured. Security developers should carefully consider the time and space complexity of different types of attacks and their corresponding defenses, and make informed decisions about the optimal trade-offs between security effectiveness, system performance, and cost.

6. Limitations and Threats to Validity

In order to ensure a thorough and comprehensive analysis, it is crucial to consider the following limitation details:

1. Architectural diversity: The study specifically examines centralized architectures for UAV systems, which may limit its applicability to other architectural paradigms, such as decentralized, distributed, or hybrid architectures. Comprehensive research exploring attack and defense patterns across diverse architectural designs would provide a more complete understanding of UAV system security.
2. Expanding the attack landscape: The research identifies 26 attack variations, but this may not encompass all possible threats in the UAV system security landscape. To obtain a more exhaustive understanding, future studies should investigate a broader range of attack patterns and techniques, including those arising from new and emerging technologies.
3. Interdisciplinary approach: This study presents countermeasures and defense strategies from a software analyst's perspective, which may not cover all aspects of UAV system security. To develop a more comprehensive understanding, future research should adopt an interdisciplinary approach, incorporating expertise from fields such as hardware engineering, network security, cryptography, and human–computer interaction.
4. Platform and language adaptability: The provided *node.js* code template may not be suitable for all UAV systems, as they may use different programming languages or platforms. To enhance the practical applicability of the research, future studies should develop code templates or guidelines for a variety of programming languages and platforms commonly used in UAV system development.
5. Robust evaluation methodologies: Assessing the effectiveness of proposed defense strategies based on time and space complexity provides valuable insights. However, incorporating additional evaluation methodologies such as real-world testing, simulations, case studies, and quantitative metrics could lead to more rigorous and practical assessments of defense strategies' effectiveness across various scenarios.
6. Customized security solutions: The research findings and recommendations may not be universally applicable to all UAV systems, given their unique requirements and constraints. Future research should consider the specific needs and challenges of different UAV applications, including commercial, military, humanitarian, and environmental use cases, and design security measures tailored to each context.
7. Human factors and usability: The study does not directly address the impact of human factors on UAV system security, such as user error, social engineering attacks, or insider threats. A comprehensive understanding of security challenges in UAV

systems should also explore the human aspect, including human-related risks and vulnerabilities, as well as the usability of security measures and interfaces.
8. Evolving cybersecurity landscape: The dynamic nature of the cybersecurity landscape requires continuous research and adaptation of security measures. Future work should monitor and analyze emerging threats, attack patterns, and technological advancements, ensuring that defense strategies remain current and effective in addressing the ever-evolving challenges in UAV system security.
9. Legal and ethical considerations: The research does not discuss the legal and ethical implications of UAV system security, such as data privacy, surveillance concerns, and regulatory compliance. A more comprehensive approach to UAV system security should consider the legal and ethical dimensions, as well as potential conflicts between security measures and privacy rights.
10. Resilience and recovery: The study focuses on attack and defense patterns but does not address the resilience of UAV systems in the face of successful attacks, nor their ability to recover from security breaches. Future research should investigate strategies to enhance system resilience and recovery capabilities, ensuring that UAV systems can effectively respond to and recover from security incidents.

Threats to validity are factors that may impact the accuracy or generalizability of the study's results or conclusions. In this context, the limitations mentioned earlier could potentially affect the external and internal validity of the research:

1. External validity: This refers to the extent to which the research findings can be generalized to other contexts or settings. The limitations related to architectural diversity, platform and language adaptability, and customized security solutions, among others, may impact the external validity of the study, as they could limit the applicability of the findings to different architectures, programming languages, platforms, or UAV applications.
2. Internal validity: This refers to the extent to which the research design allows for accurate conclusions about the relationships between variables. The limitations related to the interdisciplinary approach, robust evaluation methodologies, human factors and usability, and resilience and recovery, among others, may impact the internal validity of the study, as they could affect the completeness or accuracy of the conclusions drawn from the research.

7. Conclusions

In the UAV–cloud problem, drones are often used to collect data and perform various tasks that require communication with a cloud-based server. However, the communication between the drone and the cloud server can be vulnerable to various types of security threats, including those listed in the time and space complexity table. For example, GPS spoofing could be used to mislead the drone's location, while DDoS attacks could be used to overwhelm the cloud server with requests and cause a denial of service. To address the security threats associated with the UAV–cloud problem, security developers must carefully consider the trade-offs between performance and cost when designing their defenses. On the one hand, it is important to create defenses that are as effective as possible in preventing attacks. However, these defenses must also be lightweight and cost-effective to ensure that they do not compromise the drone's flight performance or add unnecessary costs to the overall system.

For example, when creating defenses against DDoS attacks, security developers could use techniques such as rate limiting or traffic filtering to prevent overwhelming the cloud server. These defenses can be designed with a lower time and space complexity, which will help to keep system performance optimized while minimizing the costs of the overall system. Similarly, when creating defenses against GPS spoofing attacks, security developers could use techniques such as cryptographic verification or multisensor fusion to improve the accuracy of the drone's location data. These defenses can be designed with a higher time and space complexity, which will help to ensure that the drone's location is accurately

determined and prevent the drone from being misled by spoofed GPS data. Overall, the trade-offs between performance and cost in the UAV–cloud problem will depend on the specific needs of the drone system being designed. Security developers must carefully consider the time and space complexity of different types of attacks and their corresponding defenses, and make informed decisions about the optimal trade-offs between security effectiveness, system performance, and cost to ensure that the UAV–cloud system is both secure and efficient.

This research, although providing insights into the security challenges in UAV–cloud systems, has certain limitations. One such limitation is that we focused primarily on the time and space complexity of various attacks and defenses, without thoroughly exploring the trade-offs between performance and cost in implementing these defenses. A comprehensive study considering the balance between security effectiveness, system performance, and cost will provide a more detailed understanding of the UAV–cloud problem. In future work, we plan to delve deeper into the trade-offs between performance and cost when designing defenses for UAV–cloud systems. By examining various techniques, such as rate limiting and traffic filtering for DDoS attacks, and cryptographic verification or multisensor fusion for GPS spoofing attacks, we aim to develop defenses with optimized time and space complexity. This will ensure that the drone's flight performance is not compromised and that the overall system remains cost-effective. Additionally, we will consider the specific needs of different drone systems when designing security measures, thus tailoring the optimal trade-offs between security effectiveness, system performance, and cost for each use case. This comprehensive approach will contribute to the development of more secure and efficient UAV–cloud systems in the future.

Author Contributions: Conceptualization, G.A.; Software, G.A.; Investigation, G.A.; Resources, G.A.; Writing—original draft, G.A.; Writing—review and editing, A.L.; Supervision, A.L.; Project Administration, A.L.; Funding Acquisition, A.L. All authors have read and agreed to the published version of the manuscript.

Funding: The works of the authors were funded by the National Science and Technology Council, Taiwan (109-2221-E-194-022-MY3 & 111-2218-E-194-004) and Atma Jaya Catholic University of Indonesia.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Mohsan, S.A.H.; Othman, N.Q.H.; Li, Y.; Alsharif, M.H.; Khan, M.A. Unmanned aerial vehicles (UAVs): Practical aspects, applications, open challenges, security issues, and future trends. *Intell. Serv. Robot.* **2023**, *16*, 109–137. [CrossRef]
2. Mekdad, Y.; Aris, A.; Babun, L.; Fergougui, A.E.; Conti, M.; Lazzeretti, R.; Uluagac, S. A survey on security and privacy issues of UAVs. *Comput. Netw.* **2023**, *224*, 109626. [CrossRef]
3. Hadi, H.J.; Cao, Y.; Nisa, K.U.; Jamil, A.M.; Ni, Q. A comprehensive survey on security, privacy issues and emerging defence technologies for UAVs. *J. Netw. Comput. Appl.* **2023**, *213*, 103607. [CrossRef]
4. Ahmed, F.; Jenihhin, M. A Survey on UAV Computing Platforms: A Hardware Reliability Perspective. *Sensors* **2022**, *22*, 6286. [CrossRef] [PubMed]
5. Bansal, G.; Chamola, V.; Sikdar, B.; Yu, F.R. UAV SECaaS: Game-Theoretic Formulation for Security as a Service in UAV Swarms. *IEEE Syst. J.* **2022**, *16*, 6209–6218. [CrossRef]
6. Xia, Z.; Du, J.; Han, Z. Distributed Artificial Intelligence Enabled Aerial-Ground Networks: Architecture, Technologies and Challenges. *IEEE Access* **2022**, *10*, 105447–105457. [CrossRef]
7. Qu, Y.; Dai, H.; Zhuang, Y.; Chen, J.; Dong, C.; Wu, F.; Guo, S. Decentralized Federated Learning for UAV Networks: Architecture, Challenges, and Opportunities. *IEEE Netw.* **2021**, *35*, 156–162. [CrossRef]
8. Haider, S. Ensuring Aircraft Safety In Single Point Failures, Automation and Human Factors. In Proceedings of the 2020 Annual Reliability and Maintainability Symposium (RAMS), Palm Springs, CA, USA, 27–30 January 2020; IEEE: Piscataway, NJ, USA, 2020.

9. Tlili, F.; Fourati, L.C.; Ayed, S.; Ouni, B. Investigation on vulnerabilities, threats and attacks prohibiting UAVs charging and depleting UAVs batteries: Assessments & countermeasures. *Ad Hoc Netw.* **2022**, *129*, 102805.
10. Pandey, G.K.; Gurjar, D.S.; Nguyen, H.H.; Yadav, S. Security Threats and Mitigation Techniques in UAV Communications: A Comprehensive Survey. *IEEE Access* **2022**, *10*, 112858–112897. [CrossRef]
11. Mansfield, K.; Eveleigh, T.; Holzer, T.H.; Sarkani, S. Unmanned aerial vehicle smart device ground control station cyber security threat model. In Proceedings of the 2013 IEEE International Conference on Technologies for Homeland Security (HST), Waltham, MA, USA, 12–14 November 2013; IEEE:Piscataway, NJ, USA, 2014.
12. Bekmezci, İ.; Sahingoz, O.K.; Temel, Ş. Flying Ad-Hoc Networks (FANETs): A survey. *Ad Hoc Netw.* **2013**, *11*, 1254–1270. [CrossRef]
13. Altawy, R.; Youssef, A.M. Security, Privacy, and Safety Aspects of Civilian Drones: A Survey. *ACM Trans. Cyber-Phys. Syst.* **2016**, *1*, 1–25. [CrossRef]
14. Lin, C.; He, D.; Kumar, N.; Choo, K.K.R.; Vinel, A.; Huang, X. Security and Privacy for the Internet of Drones: Challenges and Solutions. *IEEE Commun. Mag.* **2018**, *56*, 64–69. [CrossRef]
15. McEnroe, P.; Wang, S.; Liyanage, M. A Survey on the Convergence of Edge Computing and AI for UAVs: Opportunities and Challenges. *IEEE Internet Things J.* **2022**, *9*, 15435–15459. [CrossRef]
16. Yang, W.; Wang, S.; Yin, X.; Wang, X.; Hu, J. A Review on Security Issues and Solutions of the Internet of Drones. *IEEE Open J. Comput. Soc.* **2022**, *3*, 96–110. [CrossRef]
17. Fernandez, E.B. A pattern for a secure cloud-based IoT architecture. In Proceedings of the 27th Conference on Pattern Languages of Programs, Online, 12–16 October 2020; pp. 1–9.
18. Iba, T.; Isaku, T. A pattern for a UAV-aided wireless sensor network. In Proceedings of the PLoP '16: The 23rd Conference on Pattern Languages of Programs, Monticello, IL, USA, 24–26 October 2016; ACM: New York, NY, USA, 2016; Volume 11, pp. 1–63.
19. Papa, R.; Fernandez, E.B.; Cardel, M. A pattern for a UAV-aided wireless sensor network. In Proceedings of the PLoP '19: The 26th Conference on Pattern Languages of Programs, Ottawa, ON, Canada, 7–10 October 2019; ACM: New York, NY, USA, 2021; Volume 5, pp. 1–9.
20. Fu, Y.; Li, G.; Mohammed, A.; Yan, Z.; Cao, J.; Li, H. A Study and Enhancement to the Security of MANET AODV Protocol Against Black Hole Attacks. In Proceedings of the 2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), Leicester, UK, 19–23 August 2019; IEEE: Piscataway, NJ, USA, 2019.
21. Cho, J.W.; Kim, J.H. Traffic Generation Scheduling for Performance Improvement in WLAN Based Drone Networks. In Proceedings of the 2022 13th International Conference on Information and Communication Technology Convergence (ICTC), Jeju Island, Republic of Korea, 19–21 October 2022; IEEE: Piscataway, NJ, USA, 2022.
22. Yapp, J.; Seker, R.; Babiceanu, R. Providing accountability and liability protection for UAV operations beyond visual line of sight. In Proceedings of the 2018 IEEE Aerospace Conference, Big Sky, MT, USA, 3–10 March 2018; IEEE: Piscataway, NJ, USA, 2018.
23. Kadripathi, K.N.; Ragav, L.Y.; Shubha, K.N.; Chowdary, P.H. De-Authentication Attacks on Rogue UAVs. In Proceedings of the 2020 3rd International Conference on Intelligent Sustainable Systems (ICISS), Thoothukudi, India, 3–5 December 2020; IEEE: Piscataway, NJ, USA, 2021.
24. Mairaj, A.; Majumder, S.; Javaid, A.Y. (Eds.) Game Theoretic Strategies for an Unmanned Aerial Vehicle Network Host Under DDoS Attack. In Proceedings of the 2019 International Conference on Unmanned Aircraft Systems (ICUAS), Atlanta, GA, USA, 11–14 June 2019; IEEE: Piscataway, NJ, USA, 2019.
25. Padam, R.; Malhotra, J. Secure Techniques for the UAV Networks: A Review. In Proceedings of the 2018 International Conference on Computing, Power and Communication Technologies (GUCON), Greater Noida, India, 28–29 September 2018; IEEE: Piscataway, NJ, USA, 2019.
26. Gaspar, J.; Ferreira, R.; Sebastião, P.; Souto, N. Capture of UAVs Through GPS Spoofing. In Proceedings of the 2018 Global Wireless Summit (GWS), Chiang Rai, Thailand, 25–28 November 2018; IEEE: Piscataway, NJ, USA, 2019.
27. Agyapong, R.A.; Nabil, M.; Nuhu, A.R.; Rasul, M.I.; Homaifar, A. Efficient Detection of GPS Spoofing Attacks on Unmanned Aerial Vehicles Using Deep Learning. In Proceedings of the 2021 IEEE Symposium Series on Computational Intelligence (SSCI), Orlando, FL, USA, 5–7 December 2021; IEEE: Piscataway, NJ, USA, 2022.
28. Kou, G.; Wei, G.; Qin, Y. Research on Key Agreement Protocol for Static UAV networks. In Proceedings of the 2022 IEEE 5th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), Chongqing, China, 16–18 December 2022; IEEE: Piscataway, NJ, USA, 2023.
29. Benfriha, S.; Labraoui, N. Insiders Detection in the Uncertain IoD using Fuzzy Logic. In Proceedings of the 2022 International Arab Conference on Information Technology (ACIT), Abu Dhabi, United Arab Emirates, 22–24 November 2022; IEEE: Piscataway, NJ, USA, 2023.
30. Alrefaei, F.; Alzahrani, A.; Song, H.; Alrefaei, S. A Survey on the Jamming and Spoofing attacks on the Unmanned Aerial Vehicle Networks. In Proceedings of the 2022 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS), Toronto, ON, Canada, 1–4 June 2022; IEEE: Piscataway, NJ, USA, 2022.
31. Wu, H.; Li, M.; Gao, Q.; Wei, Z.; Zhang, N.; Tao, X. Eavesdropping and Anti-Eavesdropping Game in UAV Wiretap System: A Differential Game Approach. *IEEE Trans. Wirel. Commun.* **2022**, *21*, 9906 –9920. [CrossRef]

32. Bose, T.; Suresh, A.; Pandey, O.J.; Cenkeramaddi, L.R.; Hegde, R.M. Improving Quality-of-Service in Cluster-Based UAV-Assisted Edge Networks. *IEEE Trans. Netw. Serv. Manag.* **2022**, *19*, 1903–1919. [CrossRef]
33. Hassija, V.; Chamola, V.; Agrawal, A.; Goyal, A.; Luong, N.C.; Niyato, D.; Yu, F.R.; Guizani, M. Fast, Reliable, and Secure Drone Communication: A Comprehensive Survey. *IEEE Commun. Surv. Tutorials* **2021**, *23*, 2802–2832. [CrossRef]
34. Chaari, L.; Chahbani, S.; Rezgui, J. MAV-DTLS toward Security Enhancement of the UAV-GCS Communication. In Proceedings of the 2020 IEEE 92nd Vehicular Technology Conference (VTC2020-Fall), Victoria, BC, Canada, 18 November–16 December 2020; IEEE: Piscataway, NJ, USA, 2021.
35. Sánchez, H.S.; Rotondo, D.; Vida, M.L.; Quevedo, J. Frequency-based detection of replay attacks: Application to a quadrotor UAV. In Proceedings of the 2019 8th International Conference on Systems and Control (ICSC), Marrakesh, Morocco, 23–25 October 2019; IEEE: Piscataway, NJ, USA, 2020.
36. Liao, C.; Xu, K.; Zhu, H.; Xia, X.; Su, Q.; Sha, N. Secure transmission in satellite-UAV integrated system against eavesdropping and jamming: A two-level stackelberg game model. *China Commun.* **2022**, *19*, 53–66. [CrossRef]
37. Sakic, E.; Kellerer, W. Decoupling of Distributed Consensus, Failure Detection and Agreement in SDN Control Plane. In Proceedings of the 2020 IFIP Networking Conference (Networking), Paris, France, 22–26 June 2020; IEEE: Piscataway, NJ, USA, 2022.
38. Bhandarkar, A.B.; Jayaweera, S.K.; Lane, S.A. Adversarial Sybil attacks against Deep RL based drone trajectory planning. In Proceedings of the MILCOM 2022—2022 IEEE Military Communications Conference (MILCOM), Rockville, MD, USA, 28 November–2 December 2022; IEEE: Piscataway, NJ, USA, 2023.
39. Tsao, K.Y.; Girdler, T.; More, V.G.V.S. A survey of cyber security threats and solutions for UAV communications and flying ad-hoc networks. *Ad Hoc Netw.* **2022**, *133*, 102894. [CrossRef]
40. Al-Turjman, F.; Abujubbeh, M.; Malekloo, A.; Mostarda, L. UAVs assessment in software-defined IoT networks: An overview. *Comput. Commun.* **2019**, *150*, 519–536. [CrossRef]
41. Alsuhli, G.; Fahim, A.; Gadallah, Y. A survey on the role of UAVs in the communication process: A technological perspective. *Comput. Commun.* **2022**, *194*, 86–123. [CrossRef]
42. Nnamani, C.O.; Khandaker, M.R.; Sellathurai, M. Secure data collection via UAV-carried IRS. *ICT Express* **2022**. [CrossRef]
43. Li, L.; Zhang, X.; Yue, W.; Liu, Z. Cooperative search for dynamic targets by multiple UAVs with communication data losses. *ISA Trans.* **2021**, *114*, 230–231. [CrossRef] [PubMed]
44. Security Pattern Code. Available online: https://github.com/techmentalist/securitypattern (accessed on 24 April 2023).