Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Emil Marstrander

Use of Messaging Layer Security in a Military UAV Swarm

Master's thesis in Information Security
Supervisor: Tjerand Silde
Co-supervisor: Martin Strand
December 2023

NTNU
Norwegian University of
Science and Technology

Emil Marstrander

Use of Messaging Layer Security in a Military UAV Swarm

Master's thesis in Information Security
Supervisor: Tjerand Silde
Co-supervisor: Martin Strand
December 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Norwegian University of
Science and Technology

Abstract

Research into Unmanned Aerial Vehicle (UAV) swarms has received significant interest over the last years. A military UAV swarm can perform high-risk missions with less risk to personnel but introduces a higher risk of an enemy compromising the drone. The swarm communication requires protection that can withstand compromise and continue providing secure communications. Previous work identifies Messaging Layer Security (MLS) as a promising solution that provides forward secrecy and post-compromise security for messages.

This thesis aims to show how we can implement MLS in a real-world UAV swarm and documents the development of a proof-of-concept application on FFI's Flamingo multi-UAV system. We implement the Totem protocol as a decentralized Delivery Service (DS) to provide a reliable communication environment for MLS. After extensive testing, we launch a swarm with two UAVs, demonstrating that MLS can be successfully employed in a UAV swarm. As far as we know, this is the first time MLS has ever taken flight.

We identify challenges to address before launching MLS in a larger UAV swarm. For example, Cisco's MLS++ library might be too resource-intensive, especially for application messages. This is a consequence of MLS not being developed for our specific use case but for messaging systems.

Sammendrag

Forskning på flyvende dronesvermer har fått betydelig interesse de siste årene. En militær dronesverm kan utføre risikofylte oppdrag med lavere risiko for personellet, men introduserer en økt risiko for at fienden kompromitterer dronen. Svermens kommunikasjon krever beskyttelse som kan tåle kompromittering og fortsette å sørge for sikker kommunikasjon. Tidligere arbeid identifiserer Messaging Layer Security (MLS) som en løsning som sørger for at meldingenes fremtidige sikkerhet, og sikkerhet etter en kompromittering, er ivaretatt.

Denne oppgaven har som mål å vise hvordan vi kan ta i bruk MLS i en fysisk dronesverm og dokumenterer utviklingen av et konseptbevis på Flamingo dronesvermen til FFI. Vi benytter Totem protokollen som en desentralisert leveringstjeneste for å gi MLS et pålitelig miljø for å kommunisere. Etter omfattende testing, fløy vi med en sverm bestående av to droner, noe som demonstrerer at MLS kan anvendes i en dronesverm. Så vidt vi vet er dette første gangen MLS har vært i luften noensinne.

Vi identifiserer utfordringer som må adresseres før MLS kan benyttes i en større dronesverm. For eksempel er Cisco sitt MLS++ bibliotek sannsynligvis for ressurskrevende, spesielt for applikasjonsmeldinger. Dette er en konsekvens av at MLS ikke er utviklet for denne bruken, men for meldingstjenester.

Contents

Chapter 1

Introduction

We increasingly employ UAVs in military operations, with new methods discovered and used as the technology matures. Using UAVs in a swarm has received significant interest over the last few years. A UAV swarm is a group of UAVs collaborating to solve one or more missions. The working theory is that this is more efficient than using a single UAV to solve the mission. In 2015, researchers launched a swarm containing 51 UAVs [1], showing that large UAV swarms are feasible. The Norwegian Defence Research Establishment (FFI) takes an interest in this field and develops its own UAV called Flamingo to research UAV swarms.

A military UAV swarm can perform high-risk missions with less risk to personnel. High-risk missions make it more likely and acceptable that the enemy compromises one or more UAVs. Therefore, they must be secured to ensure the completion of the mission, even when some UAVs are compromised. MLS [2] is a newly standardized security protocol identified as a possible solution. The Naval Postgraduate School (NPS) in the USA tested it on UAV swarms, concluding it is a viable solution [3, 4]. However, they identified several challenges that require addressing before being deployed. This project aims to solve some of the identified challenges and create a working prototype for the swarm of Flamingo UAVs at FFI.

1.1 Problem Statement

A UAV swarm can solve a range of different missions. FFI is currently exploring two different mission objectives. The first is to function as an Intelligence, Surveillance, and Reconnaissance (ISR) platform, aiming to locate enemy units in an area. The other is to function as a means to disable other UAVs by colliding with the target UAV at high speed. FFI uses different drones for these objectives, but they often work together, where one drone finds an enemy UAV, and another drone destroys it by colliding at high speed. There are several other possible objectives for a swarm, even more if we include the possibility of UAVs equipped with weapons or explosives.

In a UAV swarm, the drones communicate mission coordination among each other and report status and observations back to the swarm operator. The drones receive instructions from the swarm operator. This communication is mission-critical and requires protection. It is important to protect it from an enemy eavesdropping, but it is arguably more crucial to protect it from interference and manipulation by the enemy. Imagine a UAV equipped with explosives starting to act on commands given by an adversarial actor. This will have catastrophic results on the mission. On the other hand, the enemy intercepting observation reports is problematic but not necessarily devastating to the mission. The security protocol must, therefore, provide authenticity of messages so that it is possible to verify and be sure of the source. It should also provide integrity of the information to avoid misleading a swarm operator.

Today's technology can easily solve the requirements above. The challenge is what happens when a UAV is compromised. How do we protect the communication in such a scenario? All previous messages should be protected—this is called forward secrecy. All future messages should also be protected—this is called post-compromise security. These two properties are highly relevant for securing communications in a UAV swarm and ensure that only information on the drone at the moment of compromise will be lost. The other drones will be able to continue to communicate securely, with confidentiality, integrity, and authenticity. Previous communication will also be secure if the information is no longer present on the compromised drone.

The security must not come at the cost of significantly reduced operational capability of the UAVs. Therefore, the introduced security mechanisms should consume as little of the UAV's resources as possible and produce acceptable communication overhead. It should also be robust against attacks and protocol failure. An important factor is that the resource consumption and communication overhead do not significantly increase when adding new group members. Current security protocols, like Signal, create secure bilateral connections between all pairs of nodes [5]. When doing this, the number of direct links from a node grows proportionally with the number of nodes. The resource consumption and communication overhead also grow at the same rate, which is highly inefficient for large groups.

MLS is a promising solution to all the abovementioned challenges. Instead of creating pairwise secure connections to all members, it creates a group with shared secrets. This makes it more scalable to large groups. MLS guarantees forward secrecy and post-compromise security and has integrity and authentication properties. However, the protocol design favors a system providing centralized services, which is not beneficial for a UAV swarm. The entire swarm might be affected if the central node fails or becomes unavailable. This project investigates ways of implementing MLS without centralized services.

When Leon and Britt [3] and Dietz [4] implemented MLS on the unmanned systems at NPS, they did it without implementing the required services. The result was unstable group operations where nodes failed to be a part of the

group. This resulted in testing only being performed on the ground, with no actual swarm operations performed. This project will take this one step further by implementing a reliable communication scheme to ensure stable group operations, with the goal of using MLS during a swarm operation in flight.

1.2 Scope

We only consider using MLS in this project without analyzing alternatives. Such an analysis has been performed by Leon and Britt [3], and this project further investigates their conclusion. The focus is developing a functioning proof of concept that illustrates the feasibility of using MLS in UAV swarms. When developing the solution, we envision a scenario where the swarm can operate independently of the swarm operator on the ground. This means that the solution has to be independent of the Ground Control Station (GCS), which otherwise could have played the role of a centralized service.

The Authentication Service (AS) is essential to the MLS protocol operation. It is responsible for verifying the authenticity of the credentials provided by the clients. Without implementing this service, the MLS protocol will not be secure because nodes will have no way of verifying each other. Practical implementations of this service are beyond this project's scope, but we will describe theoretical considerations for implementing an AS.

Post-quantum security is also a requirement for communication in future military UAV swarms. The MLS standard does not currently contain this property but can provide post-quantum security by replacing the cryptographic functions with post-quantum secure versions. How this affects resource consumption on a UAV is unknown. Solutions and further elaborations on this challenge are beyond this project's scope.

1.3 Research Questions

This project aims to shed light on the broader question of how to achieve secure and efficient communication in a military UAV swarm. We do this by investigating how well the MLS protocol performs in this scenario. Therefore, we present the following main research question:

> *Can we implement MLS to achieve secure and efficient communication in a military UAV swarm?*

To answer this, we address the following questions:

1. How can we implement MLS services in a UAV swarm?
2. What parameters of MLS give sufficient security compared to the resource consumption on the UAVs?
3. How does MLS affect the performance of the UAV swarm?

Question 1 is about how we can fit the MLS services into the operational environment of a UAV swarm. The MLS standard assumes that both the AS and DS are available. Without them, the UAVs cannot authenticate each other, and we cannot guarantee that message delivery is in the same order for all nodes, disrupting MLS operation.

Question 2 is about investigating what MLS parameters are optimal to use when considering both security and resource consumption. Important parameters are how often we should update the key material and what cryptographic algorithms we should use. These parameters can influence both security and efficiency.

Question 3 is about how the MLS implementation affects the swarm operations. If the resource consumption is too high, we might need to adjust the communication rate, affecting the swarm's ability to communicate and make decisions.

1.4 Our Contributions

This project provides the following contributions not yet documented in the research literature:

- Combining MLS and Totem to provide order of handshake messages
- Implementing a secure communication protocol on the Flamingo UAV
- Using MLS on a launched UAV swarm demonstrating a successful implementation
- Discussing efficiency and security benefits and drawbacks of implementing MLS in the Flamingo UAV swarm

1.5 Thesis Organization

Chapter 2 reviews the most essential concepts for understanding this thesis. This includes UAV swarm operation, MLS, distributed systems, and the related work by students at NPS. Chapter 3 outlines different concepts for using MLS in a UAV swarm, both how authentication should work and how to agree on key material. Chapter 4 describes the software development process and presents the Flamingo MLS software. Chapter 5 presents the results from testing Flamingo MLS in a simulated environment, using containers and the same hardware architecture as the Flamingo UAVs. Chapter 6 presents how the software performed on the actual UAV swarm and how the program became ready for testing in flight. Chapter 7 discusses this project's most important lessons and results before Chapter 8 concludes the thesis and provides recommendations for future work.

1.6 Acknowledgments

I want to thank Aleksander Simonsen and the Flamingo team for their tremendous support during this project. You have helped me understand how a UAV swarm's operational requirements affect the secure communication protocol. We would never have achieved our goals without your help. I would also like to thank my supervisors, Martin Strand and Tjerand Silde, for their support and expertise.

Chapter 2

Background

This chapter provides the background information necessary for understanding this thesis. We start with an introduction to UAV swarms and their operation, which is important for understanding the context in which we are implementing MLS. Then, we review the essential aspects of MLS to provide a high-level overview of how it works. After this, we present general theory from the field of distributed systems and the primitive Total Order Broadcast. A drone swarm is a distributed system, and when implementing MLS, we face challenges already solved in the literature. Lastly, we review related works on implementing MLS in unmanned systems.

2.1 UAV Swarm Operation

An Unmanned Aerial Vehicle (UAV) swarm is a group of UAVs collaborating to solve a specific task. The idea is that the swarm can simplify several tasks and perform them more efficiently and robustly. Such tasks could be searching for missing people or providing network coverage. In a military setting, a drone swarm could, for instance, solve tasks within Intelligence, Surveillance, and Reconnaissance (ISR) or be equipped with weapons. A common task used in research projects is to search a specified area for particular objects.

Engebråten *et al.* [6] argue that the control should be decentralized to achieve the full benefit of a UAV swarm. Decentralized control means that the swarm should always be able to solve tasks without having a communication link to the Ground Control Station (GCS). It is still essential for the operator to have an overview of the swarm and be able to control it. The operator should not have to operate individual drones but instead give commands to the swarm. The position of the GCS is typically at the location of the launch. However, other concepts are also possible, such as switching to a GCS at another location during a mission.

Dietz [4] presents the following features for a properly configured UAV swarm:

Survivability The mission can continue even if a drone malfunctions or is shot down.

Scalability The range of operations for a mission can increase by adding more drones to the swarm.

Speed The mission can be accomplished in a shorter period because UAVs work in parallel.

Autonomy To function as a swarm typically requires on-board automation.

Cost Missions can be executed more cost-efficiently by leveraging economies of scale.

2.1.1 The Flamingo UAV Platform

Nummedal [7] presents the Flamingo UAV platform developed at the Norwegian Defence Research Establishment (FFI). The four-motor quadcopter UAV is continuously being developed to facilitate research activities in the autonomy research project. One of these activities is research on autonomy in UAV swarms. They conduct most of the research on aerial systems focusing on ISR operations. Using multiple UAVs provides better situational awareness than a single UAV. The Flamingo platform is developed for generic UAV operations and can operate in a swarm using the Valkyrie swarm system, also developed at FFI. Note that we present the most current specifications in this section, and some deviate from those presented by Nummedal. The research team working on the Flamingo UAV provided the most current specifications.

The standard ISR-swarm configuration of the Flamingo platform weighs 2.8 kg and has an operational flight time of 35 minutes. This configuration includes a thermal camera and a Rajant Breadcrumb DX2 mesh radio. This radio makes it possible to extend the range in-between drones (and the GCS) by using the drones as radio-relay. It uses frequencies of 2.4 GHz and 5 GHz. The maximum data rate at the physical layer is 300 Mbps, but the expected data rate is considerably lower, especially at longer distances. The network throughput is not expected to be a limiting factor when implementing a secure communication protocol because of the high data rate.

The companion computer for the latest generation of drones is the Jetson Xavier NX, which has a high performance compared to its size. The computer is capable of running autonomy- and sensor processing applications. The Nvidia website [8] states specifications of 8 GB RAM and a six-core Nvidia Carmel 64-bit ARMv8.2 CPU. The operating system is Linux for Tegra, a Linux-based distribution for the Tegra processor series. The processors and operating system are based on an ARM architecture and use the ARM instruction set. The sensor processing software uses most of the computational resources, and the secure communications protocol can only use the remain-

ing resources. It is, therefore, limited mainly by the amount used by other processors and not necessarily by the total processing capability.

All drones in the Flamingo swarm communicate with each other and the GCS. Telemetry data is frequently shared and contains information about drone internal operations, its location, and observations, to name a few. The drones distribute this information to all other drones and the GCS as multicast packets using the Battle Management Language (BML) format. The GCS controls the swarm by sending individual commands to the drones.

The drones and GCS have detection mechanisms for link failure. A drone will return if it has not received a heartbeat message from the GCS in the last ten seconds. This safety feature ensures that the operator does not lose control of the drone. The swarm still performs the autonomy independently from the GCS. A drone can also send a video feed from its thermal camera. The video is sent separately from the telemetry data and only to the GCS. Sending full-motion video from all drones simultaneously is very demanding on the radio network, so it is often turned off to reduce the bandwidth used. The drones then perform sensor reporting by sending pictures of the relevant objects.

2.2 Messaging Layer Security

Messaging Layer Security (MLS) is a protocol created to provide end-to-end security for a group of users. MLS is not intended as a complete messaging protocol but to be implemented as a part of an application. One of the driving forces for this specification is to provide a standardized way of communicating in groups so that different applications can interoperate at the cryptographic level. MLS underwent a standardization process over the last years and was specified in RFC9420 [2] in July 2023. This specification describes the protocol operation, and an informational document draft by the IETF [9] describes the architecture of MLS. This section uses these sources as references.

Forward secrecy and post-compromise security guarantee the security of messages sent before and after a compromise and are important security requirements guaranteed by MLS. We imagine an adversary currently compromises a user's key material. Forward secrecy will then ensure that all previous messages are protected and not accessible to the adversary. MLS achieves this by deleting the key material used to encrypt messages, which makes the key material and the messages inaccessible to the adversary.

Post-compromise security will ensure that all future messages are protected and inaccessible to the adversary. MLS achieves this by updating the key material so the adversary no longer possesses the current keys. To guarantee this when the adversary has compromised all key material is impossible. The adversary would then be able to impersonate the compromised user and get a hold of the new key material as well. Therefore, we assume that the group can perform one operation before the adversary is able to take action. Another way

to look at it is that forward secrecy protects messages from future compromise, while post-compromise security protects messages from a potential previous compromise.

Many messaging applications provide the same security guarantees as MLS. Cohn-Gordon *et al.* [5] describe popular messaging applications providing this for a pairwise connection, i.e., between two users. Some examples presented are WhatsApp, Facebook Messenger Secret Conversations, Apple iMessage, and Signal. These applications use Signal's Double Ratchet algorithm to ensure secure pairwise connections, providing forward secrecy and post-compromise security. To ensure the same properties for groups, the protocol must establish bilateral connections between all pairs of nodes, which is inefficient.

A common strategy to make it more efficient is distributing *sender keys* over the bilateral channels and using them to encrypt group messages. The solution ensures forward secrecy, but providing post-compromise security is challenging because updating key material for all the pairwise channels is resource-consuming. An adversary with access can often eavesdrop indefinitely since the keys are not updated. MLS provides a more efficient way of deriving shared symmetric encryption keys. The solution relies on tree structures, which makes the cost of the key derivation scale with the logarithm of the group size. Pairwise connections scale linearly with the group size. This means that for an increase in group size from 10 to 100, MLS would double its cost, while the cost of pairwise connections would be ten times larger.

2.2.1 MLS Architecture

The MLS architecture consists of an Authentication Service (AS), a Delivery Service (DS), and the clients. The AS is responsible for issuing and verifying credentials that attest to bindings between identities and signature key pairs. The service must be trusted and secure to allow clients to verify each other's messages and credentials. A compromised AS will make it possible to impersonate other clients and get illegitimate access to the group.

The DS is responsible for routing messages between the clients participating in a group. An essential requirement for the DS is that the order of handshake messages are the same for all clients to avoid different clients having inconsistent views of the MLS group. The DS is also responsible for delivering the initial public key material required for the group establishment. The information processed by the DS is already protected, so it is not required to be a trusted service. However, faults in the DS will affect the availability of the communications. Figure 2.1 shows the setup for a simple MLS system.

The services in the MLS architecture are abstract and only specify what tasks the services have to do. It is up to the application to implement the services in a way that satisfies the requirements. The standard solution is to implement them as centralized servers, where the clients send verification requests to the AS and route all messages through the DS. Distributed solutions

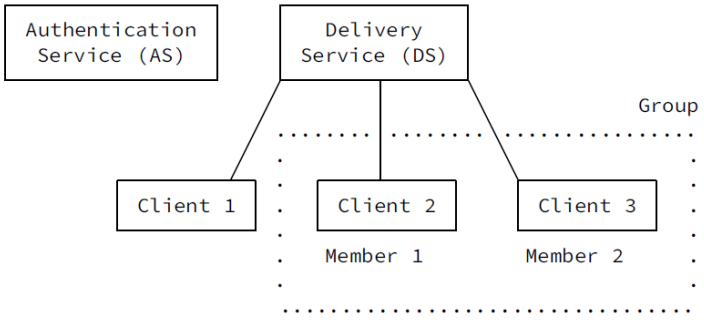


Figure 2.1: The architecture for a simple MLS system [9].

are also possible, where server and client components collaborate to fulfill the services.

2.2.2 Protocol Operation

MLS functions as a continuous authenticated group key exchange. The protocol ensures that the participants agree on common secret values and can verify each other's identities. MLS sessions are organized into *groups* and *epochs*. A group represents a logical collection of clients that share common secret values. The group evolves through epochs over time. The group members agree on values used to exchange messages in each epoch. The group state is consistent throughout an epoch and changes between epochs. The secret values always change between epochs, providing post-compromise security. Other parameters can also change, such as adding or removing group members. MLS ensures that only the members of the epoch have access to the secrets, and therefore, only the members can send and receive messages.

The clients send both application- and handshake messages during an epoch. An *application message* is an encrypted message carrying information specified by the application, for example, text messages in a messaging application. *Handshake messages*, on the other hand, are messages controlling state and epoch changes. *Commit* and *proposal* messages are the two types of handshake messages. A proposal message suggests a change in the current state of the protocol, for example, the addition or removal of clients. The state changes after sending a commit message referencing the proposals, and the entire group acts on the changes specified. The group then moves to a new epoch using the new state information.

MLS supports several different proposal messages. The following are the most important for this thesis:

Add Adding a new member to the group based on a key package for the client

Remove Removing a member from the group

Update Updating the key material of the client

When adding new clients to the MLS group, the *key package* of the client is required. The key package describes the client's capabilities and provides key material for adding the client to the group. The DS functions as a repository for key packages, but it is also possible that the system allows for requesting the key package from the clients directly. The clients use the key package when generating the add proposal. The committer sends a *welcome* message to the new client containing the key material necessary for communicating with the group. The first client initializes the group, creating a group with itself as the only member. It then gathers key packages for the joining clients and adds them to the group.

2.2.3 Ratchet Tree

At the heart of the MLS protocol is the *ratchet tree*. This tree consists of key pairs and secrets and generates secrets known to the entire group. The tree structure allows it to be efficiently updated to reflect changes in the group. The design makes encrypting messages to the entire group or subsets possible and efficient. When removing a member from the group, the committer can send new keys to all other members by doing $\log N$ operations, where N is the number of members in the group. In contrast, $N - 1$ operations would be required if new keys have to be sent directly to all other members.

The ratchet tree consists of nodes. A node without descendants (no nodes beneath) is called a leaf. The leaves represent the group members, and the number of leaves decides the tree size. The tree is binary, meaning every node (except leaf nodes) has precisely two descendants. Every node is either blank (containing no value) or contains an asymmetric key pair with some associated data. The leaves also contain a credential provided by the represented client. Figure 2.2 shows an example tree.

Every group member knows the asymmetric public keys associated with every node. However, private keys are only known by some. A node's private key is known to the leaf nodes in the subtree of that node. This means a member knows all private keys in a line from its own leaf node's parent to the root node. However, there is an exception to this. A node will not initially know all these values when it joins the group. It is then known as an *unmerged leaf*. Figure 2.3 shows an example of public and private trees. All nodes know the private keys of the root node, but only nodes A and B know the private keys of node AB.

The clients mainly use the asymmetric keys from the ratchet tree for encrypting key material when moving between epochs. The ratchet tree structure allows new keys to be efficiently encrypted so that only the rightful receivers can decrypt them. The encryption uses the public keys of the nodes in the ratchet tree. By encrypting a message with the public key of the node, only the descendants of that node can decrypt it. For instance, if we encrypt a

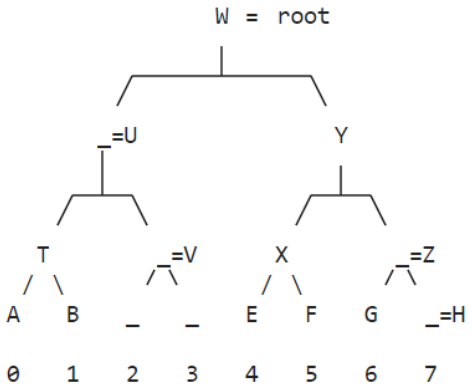


Figure 2.2: Example of a ratchet tree. Nodes represented by underscore are blank. The leaf nodes A, B, E, F, and G represent the group members. [2]

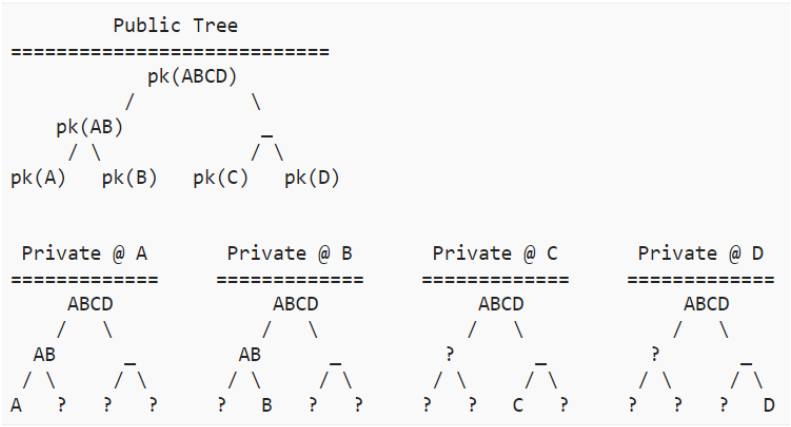


Figure 2.3: Example of ratchet tree with public and private trees [2].

message with the public key of node AB in Figure 2.3, only A and B can decrypt it. However, all members can decrypt if we encrypt with the public key of $ABCD$. In this way, we can encrypt to subsets of the group. If the node has an unmerged leaf as a descendant, we will have to encrypt separately to that member using the leaf's public key.

We update the tree for every commit. The changes depend on the proposal messages sent during the epoch. If we add members, they replace blank leaf nodes. If no leaf nodes are blank, the tree expands to the right. If we remove members, the leaf nodes representing those members become blank. If possible, the tree shrinks. If a member sends an update proposal, it updates its leaf node with a new key pair and blanks all nodes above the leaf node. This process provides post-compromise security. When sending a commit message, the client updates its leaf with new key pairs and creates new key pairs for the blank nodes it is a descendant of. It sends the private keys of these nodes to the

descendants using keys from the previous ratchet tree.

2.2.4 Key Schedule

The key schedule of MLS generates keys for multiple purposes, including encryption of message metadata, application messages, handshake messages, and more. We derive these keys from the *epoch secret*, derived from the ratchet tree for the current and previous epoch.

We use the *encryption secret* to create a *secret tree*, which provides key material for encrypting MLS messages. This secret tree has the same structure as the ratchet tree, and the same leaf nodes are related to the same members. In the tree, all leaves contain a secret derived from the encryption secret, which we use to create two *sender ratchets*, one for application messages and one for handshake messages. These ratchets consist of hash chains, where we derive a key and nonce for every hash value. The secret tree dedicates a key and nonce to the encryption or decryption of a single message. When sending messages, a client uses a specific key and nonce from the ratchet. The receivers of the message derive and use the same values for decrypting. MLS uses a symmetric authenticated encryption scheme for this operation.

An essential part of the key schedule is the deletion of old and used keys. Deletion of keys is a necessity to provide forward secrecy. When a message is encrypted or decrypted, the client deletes all keys involved in deriving the encryption key and nonce. This includes the encryption secret, parts of the secret tree, and the proceeding hash values.

2.2.5 Security and Reliability

MLS achieves confidentiality of messages through symmetric encryption using keys from the secret tree. Only members have access to the secrets necessary to derive these keys. Knowing the initial secret shows that the client was a member of the previous epoch. Knowledge of the commit secret shows a successful calculation of the private secrets of the ratchet tree. A joining member receives these values through a welcome message. MLS supports multiple symmetric authenticated encryption cipher suites. AES-128-GCM, AES-256-GCM, and CHACHA20-POLY1305 are currently specified.

MLS achieves two forms of authentication. The first is that group members can verify that the message originated from a group member. The authenticated encryption algorithm guarantees this, as described in the previous paragraph. The second form of authentication is that group members can verify that the message originated from a specific group member. Every message has a digital signature created using the signature key of the sending client. Other members can verify this signature with the AS and trust that the claimed identity sent the message. It is, therefore, not possible to impersonate members. MLS supports multiple cipher suites for signatures and hash algorithms

that can be chosen based on the required security level. Digital Signature Algorithm (DSA) using different elliptic curves is currently specified, together with hash algorithms SHA-256, SHA-384, and SHA-512.

The deletion schedule achieves forward secrecy in MLS. After sending a message, the client deletes the symmetric encryption key and every key used to derive it. If every member does this correctly, forward secrecy is achieved for every message. If a member is compromised, no previously used keys are present; therefore, the adversary cannot decrypt any previous messages.

MLS achieves post-compromise security through update, remove, and commit messages. These messages replace or remove the compromised key material to ensure the security of messages after a compromise. All clients in an MLS group regularly update key material by creating update proposals or generating a commit. The application decides how often to perform this, specifying the window of opportunity for compromised keys. Updating key material will not create a secure state if the client and not only the keys are compromised. Then, removing the compromised client from the group achieves post-compromise security.

The confidentiality of some message metadata fields is not protected. This will allow anyone eavesdropping to learn about the group state and membership. This information can, for instance, allow for Denial of Service (DoS) attacks that target specific clients based on the information available in the message metadata. To avoid this type of attack, the standard recommends carrying MLS messages over a secure transport such as Transport Layer Security (TLS).

The order of messages is essential for MLS operation. The DS is responsible for ensuring that everyone receives the messages in the correct order. Misordering application messages is not a problem since MLS marks every application message with a generation number and sender ID specifying the key to use. On the other hand, some handshake messages require that all members agree on the order. A client can receive proposal messages in any order as long as it receives them before the commit message. However, commit messages received in different order will make the members commit different states, creating inconsistencies in the group. The clients derive different keys, dividing the group into subgroups unable to communicate with each other.

2.3 Distributed Systems

Singhal and Shivaratri [10] give a general overview of agreement protocols in distributed systems and present two main classes of systems: synchronous and asynchronous systems. In synchronous systems, the nodes run in a lock-step manner, inherently synchronized against each other. In asynchronous systems, every node can send and receive messages and perform computations at any time. Agreement problems are not solvable in an asynchronous system if there are nodes that fail.

In a distributed system, the different nodes must agree on specific values. They agree based on communication between them. The communication link or the nodes themselves might be faulty, creating challenges. This is called an agreement problem. The literature generally classifies these into three types of problems.

The Byzantine Agreement Problem Also called the Byzantine Generals Problem. An arbitrary node broadcasts a value. The goal is for all non-faulty nodes to agree on this value.

The Consensus Problem Every node broadcasts its initial value. The goal is for all non-faulty nodes to agree on a common value.

The Interactive Consistency Problem Every node broadcasts its initial value. The goal is for all non-faulty nodes to agree on common values.

These problems are closely related because we can use a solution to The Byzantine Agreement Problem to derive solutions to the other problems.

2.3.1 Total Order Broadcast

Défago *et al.* [11] introduce a primitive called *Total Order Broadcast*. The goal of this primitive is to ensure the delivery of messages in the same order to all participants. The article overviews different ordering mechanism classes, surveys sixty published Total Order Broadcast algorithms, and presents the following general ordering mechanisms.

Fixed Sequencer A single node has the role of sequencer. When sending a message, the sequence number is coordinated with the sequencer node to ensure correct ordering.

Moving Sequencer A set of nodes has the role of sequencer, and the work is distributed among these nodes.

Privilege-based A node can only send when given a token. This token circulates among the senders.

Communication History Messages are given timestamps to determine the order.

Destination Agreement The destination nodes receive the messages and must agree on the order.

Total Order Broadcast algorithms have to make assumptions about their environment. An important assumption is how reliable the underlying channel is. Some algorithms rely on a communication layer that takes care of message loss. Another assumption is how and if nodes in the system can fail. The article presents three main categories of failure: crash failures, omission failures, and

Byzantine failures. *Crash failure* is when a process crashes and ceases to function forever. *Omission failure* is when a process omits to perform some action but continues to contribute after the omission. Finally, a *Byzantine failure* can change message content, duplicate messages, or even maliciously try to break down the system. The article defines a *correct process* as never doing any of the faulty behaviors mentioned.

An algorithm can also assume that the system is *partitionable*. When links between nodes in the network break down, the group splits into several isolated subgroups. The algorithm is partitionable if all members can continue communicating within their subgroup. The alternative is that only a primary partition can continue sending messages.

The algorithms can ensure several different properties. *Agreement* is one such property and ensures that all correct processes receive all broadcasted messages. *Total order* is another property that ensures all correct processes have the same order of received messages. Both these properties can be *uniform* or *non-uniform*. Uniform applies to all processes, correct or not. This is the strongest guarantee and is more robust against process crashes. The non-uniform properties only apply to correct processes.

The total order property only considers the order at the destinations and does not ensure the correct order based on which node sent the message first. *First In, First Out (FIFO) order* guarantees that the order at the destinations equals the order in which the clients sent the messages. *Causal order* is similar to FIFO order but is a weaker guarantee. Instead of using absolute time to make the ordering decision, causal order uses the relationship between events to decide if the sending of a message "precedes" the sending of another message. If we establish this precedence, the order of messages reflects this. If not, the message delivery is in an arbitrary order.

Most algorithms include fault tolerance mechanisms. Two types of failures are process failure and communication failure. *Group Membership Service* is a mechanism that can detect process failure by managing the membership of processes in the group. The membership view is updated and communicated to the remaining processes if a process crashes. For communication failure, the detection mechanism can use positive and negative acknowledgment. When using positive acknowledgment, the nodes acknowledge the reception of messages. The nodes only report missing messages when using negative acknowledgment.

2.4 Related Work

Implementing MLS in unmanned systems is a research topic of interest at the Naval Postgraduate School (NPS) in the USA, which has produced two master's theses on the subject. Leon and Britt [3] did a qualitative study of security requirements for a communication protocol in unmanned systems. They then identified MLS as the most suited protocol and implemented it on two unmanned systems. Dietz [4] compared different secure communication

protocols and argues that MLS was well suited for use in a UAV swarm. They then implemented the protocol on the Advanced Robotic Systems Engineering Laboratory (ARSENL) Unmanned Aerial System (UAS) and performed ground testing on a swarm with up to 12 UAVs.

2.4.1 Security Requirements for Unmanned Systems and Implementation of MLS

Leon and Britt [3] performed a qualitative study to identify several security properties advantageous for a secure communication link between unmanned systems. They interviewed military and civilian field experts, both cyber security and unmanned systems experts. The study identified the following requirements:

- Security of data before and after a compromise (forward secrecy and post-compromise security)
- Ability to detect a compromise occurrence in real-time
- Asynchronicity—not all parties need to be online at the same time
- Scalable overhead when the number of participants increases
- Software-defined protocol to avoid sensitive cryptographic material
- Interoperability—it is possible to install the protocol on any device

They then evaluated the following secure messaging protocols based on these requirements: Open Pretty Good Privacy (OpenPGP), Internet Protocol Security (IPSec), Transport Layer Security (TLS), MLS, Signal, Wickr, Cisco Webex, and Joint Range Extension Applications Protocol (JREAP). Of these, only MLS and Cisco Webex satisfy all the abovementioned requirements. Wickr and Signal satisfy all requirements except for scalable overhead when participants increase. Cisco Webex uses MLS for key exchange and management, so they conclude that MLS is the best-suited security protocol.

Leon and Britt [3] implemented MLS on the CASSMIR Unmanned Surface Vehicle (USV) and the ScanEagle UAV using Cisco's implementation of MLS draft 12. They sent telemetry data between the CASSMIR and the ScanEagle, demonstrating that MLS can provide secure communication between different unmanned systems. They also sent a turn-rate message between the GCS and the ScanEagle.

The implementation lacked an AS, meaning there was no authentication of group members and messages. The ScanEagle performed the DS functionality, but no concurrency measures existed. It also used Transport Control Protocol (TCP) as the transport layer protocol, which is disadvantageous because of the lack of broadcast and multicast features, among other things.

During the testing, they experienced problems with key updates not being processed in sequence, causing the MLS session to break. They argue that the cause of this problem was messages being sent and received at the same time as add and commit messages were processed. Their solution was to reduce the sending rate of messages and found one hundred messages per second optimal.

A DS, either centralized or decentralized, was identified as a solution to this problem.

They determined the cost of different parameters with the following result.

Encryption overhead Depending on the cipher suite used, the message size increased between 171 and 277 bytes. It seemed like the hash generated accounted for the difference.

Setup times The group initialization times ranged from 4 ms to 6 ms for five out of six cipher suites. The cipher suite P521-AES-256-GCM-SHA-512-P521 performed significantly worse with approximately 18 ms setup time.

Message Handling Times Measuring the time used for sending 1000 messages with five update messages showed that most cipher suites performed relatively equally. X25519-AES-128-GCM-SHA-256-Ed25519 and P256-AES-128-GCM-SHA-256-P256 had worse performance than the rest.

Leon and Britt [3] conclude that MLS is suited as a secure communication protocol following the requirements identified. The use case testing shows that MLS is a viable solution for secure data exchange and control of different unmanned systems. The developed application works as a proof of concept and is not ready for real-world usage. The authors identify the following future work for the MLS application:

- More extensive and realistic testing of the MLS application
- Development of an AS, either centralized or decentralized
- Development of a DS, either centralized or decentralized

2.4.2 Implementation of MLS on the ARSENL UAV Swarm System

Dietz [4] implemented MLS on the Advanced Robotic Systems Engineering Laboratory (ARSENL) UAV swarm system and tested it using Mosquito Hawk quadcopter UAVs. The software used Cisco's implementation of MLS draft 11. The work does not include the development of an AS or a DS. They evaluated the solution in a simulated environment and on grounded UAVs. The thesis provides an overview of different group and pairwise communication protocols. Based on this, they argue that MLS provides a lightweight way of scaling secure group communication and is well suited for use in a UAV swarm.

In the solution, they perform a handshake between the joining node and a group member prior to adding the member to the group. This handshake is necessary to ensure the member notices the key package. This process took at most 58 ms. The group's initial creation by the first node took less than 5 ms.

In the simulation testing, they used between five and ten UAVs. They varied between issuing an update message every 50, 150, and 250 messages and sending no updates. At least one UAV failed or dropped out of the group communication for all tests. The identified reasons for this are unprocessed update

messages or missed commit messages. Update every 250 messages successfully sent and received the most messages and had the lowest UAV failure rate (of course, no update messages performed better). More than 250 messages between updates are likely even better, but there is a security concern with longer update intervals.

They experienced the same issues while testing on UAVs on the ground. Only when using three UAVs in the swarm did they not experience node failure. They tested with up to twelve UAVs. The failing node rate shows the need for a DS or some mechanism to deal with packet loss and out-of-order messages.

The thesis concludes that MLS can perform well in a swarm, but we must develop mitigations for the unreliable communication scheme. They also identified the following for future work:

- Additional testing to determine the best update interval
- A solution that enables the recovery from dropped MLS packets
- Better error handling so that recovery from failure is possible
- Mechanism for reliable update operations
- AS implementation
- Protocol to decide when to classify a UAV as compromised

Chapter 3

Concepts for Using MLS in a UAV Swarm

This chapter presents conceptual solutions to the identified issues when implementing MLS in a UAV swarm. We have not found previous work on this use case, so we have no sources for solving the presented challenges. Therefore, we developed the conceptual solutions during the project. This applies primarily to Sections 3.1 and 3.3. Section 3.2 uses sources from the distributed systems literature to solve the challenges.

For some of the presented problem areas, there is not only an issue of making a reliable solution but also security concerns. For instance, if a drone loses connection to the group, it might be because of a compromise. If this is the case, we should keep it from joining the group. This chapter focuses on making functional solutions without considering these security concerns, which we address in Section 7.3.

3.1 Authentication Service

The MLS specification requires an Authentication Service to be present in order to guarantee the security requirements but does not specify how to implement it. The main tasks are to issue credentials and enable clients to verify the credentials. The credential attests to the binding between identities and signature key pairs. The architecture informational document by the Internet Engineering Task Force (IETF) [9] gives some examples of how to do this:

- Using Public Key Infrastructure (PKI). Issuing would be done by a Certificate Authority (CA), and the clients would verify certificates.
- Users verify each other's key fingerprints. Issuing would be the generation of a key pair. Verification would be an application functionality that enables clients to verify keys.
- End user Key Transparency. Issuance would be to insert a key into the Key Transparency log under the user's identity. Verification would be to

verify the key's inclusion and monitor the Key Transparency log.

The most basic way of implementing the AS is to use a PKI, which is widely used today and, therefore, simple to implement. The MLS specification also includes X.509 certificates as a standard credential type, a PKI standard. One downside of a PKI is that it usually requires a connection to a centralized infrastructure for issuing and verifying credentials. This is disadvantageous in a UAV swarm since we want it to operate autonomously. We present proposed solutions to this problem in the rest of this section.

In a PKI, the CA is responsible for issuing certificates for public keys the clients generate. The CA signs these certificates with their private key. The client signs its messages using its private key and provides the certificate to prove its claimed identity. This proof includes its certificate and a certificate chain to a trustworthy CA. In Internet PKI, this is the root CAs, and all clients have preinstalled their certificates. In a swarm environment, we can use a similar method. Even though the signing CA is not a root CA, its certificate can be preinstalled on the nodes to make the CA trusted. The advantage of this is shorter certificate chains that only contain the client's certificate.

The operations requiring a connection to the CA are issuing certificates and checking the revocation list. Certificate issuance can be handled pre-flight and is not of a concern after that in most cases. In the case of compromised keys, there is a need to establish a secure channel to generate new keys and a certificate, which is difficult in flight.

Checking against the revocation lists is challenging because it requires a connection to the CA. We can implement this in a few ways:

- The GCS has a revocation list, and swarm members ask the GCS for this.
- There is an agreement in the swarm on the revocation list. A node asks the swarm if the certificate is valid. Since the member presenting the certificate is also a swarm member, we must ensure it cannot manipulate the process.
- The GCS is responsible for the revocation list and distributes a signed version to all swarm members. Members can ask each other for the most current version of this list.

The maintenance of a revocation list has many similarities to the maintenance of a list of compromised nodes. A compromised node would likely also be on the revocation list because that is a node we do not trust. However, a drone can lose its private key without the drone itself being compromised. For instance, if the drone is physically recovered after the compromise and reinstalled with new keys. In such a situation, we revoke the certificate, but the node would not be on the list of compromised nodes. Another solution could be to create a new ID for the drone when reinstalling. The revocation list and list of compromised nodes would remain equal since we keep the old ID on the list of compromised nodes. The lists are usually equal; therefore, we

should combine these functionalities to ensure efficient and secure operation.

The job of the certificate is to provide an identity to a public key. This is the node's public signature key in this setting, and we can, for instance, represent the identity as a name, email address, or domain name. How we represent the identity is not critical for the certificate's security. However, all nodes and the CA must interpret the identity similarly. For instance, if we represent the identity as "drone1", "drone2", and so on, the CA must ensure that certificates are issued to correct identities and that two nodes do not claim the same identity. Manual transfer of public keys, identities, and certificates between drones and CA is one way to ensure the identities are correct. It might also be possible to achieve in a closed wired network, but we must ensure no adversaries are present.

We can also use the identity to identify different roles in a UAV swarm. We could, for instance, have a leader drone with the identity "leader". However, in such a setup, it is vital to ensure that arbitrary drones cannot claim the identity of "leader". It is also possible to do this with GCSs. For instance, "gcs1", "gcs2", and so on. For both these setups, we can have logic on the drones that specific commands are only acted on when they come from the leader or GCS.

How we implement the PKI depends on how we set up the swarm system. We consider two main system setups in the following subsections. The first is a system where each swarm functions as a standalone system, and the second is a system where we interconnect multiple swarms in a joint system. We propose AS implementations for both systems.

3.1.1 Each Swarm as a Standalone System

In this system, a small set of GCS (typically one or two) can communicate with a dedicated swarm. The swarm operates independently of all other swarms and cannot communicate with other GCSs. This setup is the simplest case for a UAV swarm; therefore, creating an independent PKI is a possible solution. Implementing a solution where a GCS functions as a CA is straightforward in this system. The GCS is less likely to be compromised than the drones, so CA compromise is not the most significant concern. The signature verification process is simple and requires little network resources if we store the CA certificate on every drone. We can then trust the CA certificate.

Even though the signature verification process is straightforward, handling revocation and revocation lists is still challenging. We use revocation lists to revoke certificates with a compromised private key. It is tough to detect the occurrence of such a compromise. How can we ensure the rest of the drone is not compromised? In most cases, we would have to assume that the drone is compromised if its private key is compromised. Therefore, this issue is closely related to the detection of compromised drones. The simplest solution would be to assume that the private key is lost if a drone is compromised and vice

versa. Of course, there are exceptions, but it would be beneficial to make this assumption to ease the complexity of the implementation.

Even if we somehow detect that a drone has lost its private key in flight but is not compromised, it is still challenging to restore trust. A new certificate cannot be issued over the air because another entity can impersonate the drone using its private key. Therefore, the drone must return to the GCS for a new certificate. In these scenarios, there is little benefit in distinguishing between compromised nodes and compromised private keys.

For efficiency, it is unnecessary to maintain both a revocation list and a list of compromised nodes, especially since they mainly present the same nodes. Which list is maintained is not very important. If the system maintains the list for many operations, it introduces unnecessary complexity when checked. To avoid this, we can reset the system after missions where compromises occur. We can do this by re-keying the CA, invalidating all previously issued certificates. We must then issue new certificates and distribute the CA certificate to all drones.

To provide more robust post-compromise security, we can take this one step further and re-key the drones. This will ensure that the system is secure, even if the compromise of a private key is not detected. Since compromise detection is difficult, performing regular re-keying would be beneficial.

A solution for a standalone swarm system can be to avoid implementing a revocation list altogether. During a mission, we maintain a list of compromised drones. This can be manually updated by the swarm operator or by swarm members. Revocation of certificates during a mission is not necessary. After a mission where a drone is lost or compromised, we perform a re-keying process as described in the previous paragraphs.

3.1.2 Multiple Swarms in a Joint System

For a more dynamic setup, we have a pool of drones and a pool of GCSs. These should be able to work together, be organized in numerous ways, and still validate each other. We can have multiple swarm operations performed simultaneously, and if they are operating in the same area, they can communicate securely.

We need a shared PKI infrastructure to handle the certificates, identity, and revocation of such a large and complex operation. A solution could be to use the same strategy as in Section 3.1.1 and preinstall all CA certificates on the drones. However, this is a security risk since a more extensive system is more likely to have a compromised CA. A compromised CA will make the entire system vulnerable. If another UAV swarm team is physically compromised, their GCS and CA are compromised, affecting the operations of other UAV swarm teams.

Another solution is to add a central CA to the system, which functions as a root CA. This CA could issue all certificates or allow the GCS to issue local

certificates. The most user-friendly way is having the controller issue local certificates. Then, we can re-key the drones without connecting to central infrastructure. Revocation of the CA at the GCSs can be performed with this solution, which means that a compromised swarm team does not compromise the entire system. The root CA and its private key must be sufficiently secured because the entire structure falls apart if compromised. We must store the root CA certificate on the drones and GCS so the nodes trust the root CA.

In this scenario, updating and distributing the revocation list to UAVs in flight is necessary. This is because compromises could happen often if multiple units perform large-scale swarm operations. We can imagine a scenario where a drone was compromised many hours or even days ago, but the revocation list did not update until after the UAVs launched.

3.2 Ensuring Agreement for Handshake Messages

A reliable DS is an essential component of the MLS architecture. One of its tasks is ensuring all nodes receive messages in the same order, which is essential for handshake messages. If two drones handle commit messages in different order, they will create different MLS states and cannot communicate. This scenario can happen when two nodes issue commit messages at approximately the same time.

It is only necessary to receive proposal messages before the corresponding commit message. The order does not matter. If they are not received when the commit arrives, the message handling will produce an error because the node does not recognize the proposals referenced by the commit message. All drones must, therefore, agree on the order of handshake messages. Once this is in place, the MLS standard ensures all nodes agree on the new state.

Dropped handshake messages are also challenging. The node can no longer communicate with the group if a commit or proposal message is lost. Therefore, implementing mechanisms for ensuring that all nodes receive the same handshake messages is essential.

The straightforward solution is to implement the DS as a central server that receives all messages and sends them to their recipient. This solution is disadvantageous in a UAV swarm because it generates substantial traffic when sending messages to and from the central server. In addition, if the server goes down, all communication is lost. If this server is, for instance, implemented on a single drone, all communication would be lost if that drone is taken out.

There are ways of using a central DS more efficiently. The order of application messages is not vital for MLS operation, so we do not have to send them through the DS. This removes the asynchronous part of MLS because nodes must be online to receive application messages. The benefit is decreased network load since handshake messages are infrequent compared to application messages. A solution is to use the GCS as a centralized DS. This requires the swarm to be in communication range of the GCS when exchanging handshake

messages.

We can choose the DS based on the available nodes to avoid being dependent on a single DS. This is similar to the moving sequencer ordering mechanism, where we distribute the sequencing among the selected nodes. If the drone acting as DS becomes unavailable, another drone will take the responsibility, and the swarm can continue communicating. The challenge is to figure out a mechanism for selecting the drone to act as DS. This requires agreement between the nodes.

Instead of reinventing the wheel, we can use the Total Order Broadcast mechanisms from Défago *et al.* [11] presented in Section 2.3.1. The privilege-based ordering mechanism presents the most desirable solution to this challenge. It is the only mechanism where sending is synchronized between the nodes. This prevents cases where two nodes send commit messages simultaneously, removing unnecessary computational load. The sender also decides the order of messages, which ensures the delivery of a proposal message before the corresponding commit message. Even though all mechanisms could provide a working solution, privilege-based ordering reduces the complexity by synchronizing the communication between the nodes.

The swarm communication uses User Datagram Protocol (UDP), an unreliable transport protocol. The privilege-based order algorithms must be adapted to this and tolerate communication faults. Based on these requirements, we identify the following algorithms from the work of Défago *et al.* [11]:

- Train
- Totem
- Token-Passing Multicast (TPM)
- RTCast

The Train algorithm [12] uses a token that functions as a train and moves between the participants in the group. The members send messages by adding them to the train, and messages are received by reading them from the train. This process functions well to control handshake message flow. However, in a broadcast network where everyone can communicate at all times, using this train to communicate messages is inefficient. In MLS, we would have to wait for the train to pass all members before they updated their key material and entered the new epoch. Updates should ideally be done simultaneously by all participants. For this reason, the Train algorithm is not suited for our use case.

The Totem algorithm [13] is designed for partitionable systems, which is positive for a UAV swarm since we can imagine scenarios in future research where the swarm splits into different groups. Totem uses a membership protocol to maintain a ring that the token traverses. This protocol detects process failure, partitions, and token loss and ensures all nodes receive every message. Totem is a complex algorithm, but pseudocode is provided in the paper, making it manageable to implement.

The TPM algorithm [14] operates similarly to Totem but does not support

group partition. It also provides uniform agreement and total order, while Totem provides both uniform and non-uniform agreement and total order. According to the Totem authors, TPM requires two and a half rounds to ensure safe message delivery, while Totem only requires two.

The RTCast algorithm [15] is designed for real-time systems and focuses on high performance. It guarantees atomic ordered delivery of messages within one round, compared to two rounds for Totem. However, while technically correct, RTCast guarantees this in one round because if a node fails to receive messages, it takes itself out of the group. To simplify the protocol, RTCast assumes synchronized clocks between nodes. Although we can guarantee synchronized clocks in a UAV swarm, it is not desirable to implement measures to ensure this. High performance is not essential when implementing MLS in a UAV swarm. We require a reliable protocol focusing on maintaining the group. RTCast's focus on performance, with a low threshold for members leaving, is undesirable for this operation.

Based on this analysis, Totem is the protocol we will implement in this project. Train and RTCast have undesirable properties for supporting MLS operations in a UAV swarm. Totem and TPM are similar in operations, but Totem has more desirable properties. The main drawback of Totem is that it is a complex protocol, which will be challenging to implement. TPM seems to be more straightforward. However, the Totem authors provide pseudocode in the paper, which should make the implementation process more streamlined[1]. We present Totem in more depth in Section 4.2.

3.3 Reestablishing Synchronized MLS Groups

During MLS operation, there is always a possibility that the group state between one or many nodes becomes unsynchronized. The likely cause is proposal or commit messages that were not delivered correctly. The messages were not delivered at all, delivered in incorrect order, or corrupted in transmission. Totem, the algorithm proposed in the previous section, should prevent these faults, but errors could still occur because of high packet loss or loss of network connectivity.

Regardless of the cause, there should be mechanisms for reinitializing the connection to the group. The first step in such a mechanism is to detect that a node has lost synchronization with the group. A node can detect this when trying to decrypt messages. The message encryption uses an authenticated encryption scheme, meaning we use both encryption and authentication. When checking the Message Authentication Code (MAC), we will know whether we have the correct key material. Incorrect key material means we have lost synchronization with the message sender. An update that has not yet reached

[1] In retrospect, the Totem pseudocode seemed better at first glance than what it actually was.

the receiver could also cause this behavior. We, therefore, have to wait an appropriate amount of time or messages before concluding that synchronization with the node is lost.

After determining that we are unsynchronized with another node, we need to determine if there are other nodes to which we are unsynchronized. We follow the same procedure as the first one, ultimately creating an overview of our synchronized and non-synchronized nodes. We use this information to determine if we need to initiate a resynchronization or wait for other nodes to do so.

There are multiple ways of performing the resynchronization. We have identified three main possibilities:

- Restore the old connection to the group by resending the necessary information
- Asking a member to add the node to the group
- Using MLS functionality such as *reinitialization proposal* and *external proposal*

We can restore the old connection by resending the missing commit and proposal messages. The unsynchronized node can then calculate the correct group state. This requires storing old messages or states, and the most obvious place to store them is on the drones. Storing old states breaks the principle of forward secrecy because an adversary can use the state information to decrypt old messages. Storage of encrypted messages is acceptable because only group members can decrypt them.

The unsynchronized node can ask the group to add it again. It can do this by sending a message saying it would like to join the group, including its key package. It can send the message to all members but should only send it to one to avoid multiple members adding it. After joining the group, the old version of the node should be immediately removed using a remove proposal to keep the MLS state current.

MLS provides additional features that can be useful in this setting. We can use the reinitialization proposal to reinitialize the group and the external proposal to let the node add itself to the group. Although these methods are possible, we have implemented the most straightforward solution in this project. This is the solution where the unsynchronized node asks to be added to the group again.

For all resynchronization methods, the nodes must know when to initiate it. The simplest case is if there are few unsynchronized nodes. These unsynchronized nodes can then ask to be added by a synchronized member of the MLS group. With many unsynchronized nodes, it becomes more difficult because we cannot be sure of the group composition. There could be multiple subgroups that are only synchronized internally. If we try to join the group of another node, we risk joining only a subgroup.

Imagine a scenario where we have three subgroups. The subgroups have

synchronization internally but not with each other. For these subgroups to merge into a single synchronized group, we require an agreement between the nodes of who is joining whom. If the nodes join other groups randomly, we risk ending up in the same scenario, only with a different subgroup composition. A node cannot detect this situation because it can only see two groups: those with synchronization and those without. This makes it even more challenging to have the nodes agree on a solution.

An ideal solution would make the nodes agree on what subgroup to join, and then all nodes would join the same group. However, since this is challenging to achieve, we will only implement a simple solution in this project. A node will initialize the resynchronization procedure if it detects synchronization with half of the nodes or less. This threshold ensures that we perform resynchronization procedures for all situations with an unsynchronized group. There will always be a subgroup containing half the nodes or less.

The disadvantage of the solution is that it works poorly when multiple subgroups only have internal synchronization. All the subgroups probably contain less than half the nodes. In that scenario, all nodes will try to join the MLS group of the other subgroups. This creates a chaotic situation but should eventually converge on a stable, synchronized group by chance. The advantage of the solution is that we never end up in a deadlock situation where no nodes initiate any action. At least one node will meet the criteria if the group is not synchronized. Therefore, this solution will usually work, giving the group a fighting chance to become resynchronized.

A possible scenario during swarm operations is getting two separate groups out of each other's communication range. This could be intentional because of the mission requirements or unintended. The scenario is quite similar to having unsynchronized nodes. The difference is that these groups cannot reestablish the MLS group since they are out of communication range. These groups can continue to communicate internally when out of range because both Totem and MLS support partitioned groups.

The challenge is what happens when the subgroups are within communication range of each other again. Then, the subgroups should merge so all members can communicate. The same procedures as for unsynchronized nodes will provide a solution. The primary difference is that the groups need to know the total number of nodes to decide who should initiate the merger. The smaller group joins the larger one. Totem provides the number of nodes since it uses a group membership protocol. It will register all nodes within the communication range.

It is also helpful to have a mechanism for the initial setup of the MLS group when the UAVs are getting ready for flight. This will allow for a faster launch time than manual setup. Dietz [4] performed the setup by having the UAV with ID 1 establish the group, and all other UAVs join by asking the UAV with ID one number above them. As Dietz says, this only worked in a simulation environment because, in realistic operations, we cannot be sure

that the drone with ID 1 participates, and we cannot rely on every node being a part of the swarm. We need a more dynamic and flexible solution than this. Ideally, the UAVs should be turned on and automatically set up the group. Totem provides a good foundation for implementing a mechanism because it creates a group of all nodes within the communication range. A new node can use the list of group members to ask to be added to the MLS group.

Chapter 4

Development of Flamingo MLS

We name the software Flamingo MLS and use this term to refer to the program we develop in this project. The source code is available on GitHub[1]. In the following chapters, we use different fonts to give specific meanings to expressions. Sans Serif specifies program- or protocol-specific names, and `Typewriter` specifies command line output.

4.1 Software Development and Structure

Flamingo MLS uses MLS for secure communication in a group of distributed nodes. We use the Totem protocol to ensure the agreed order of handshake messages to keep the nodes in MLS synchronization. The nodes regularly initiate the MLS update procedure. Flamingo MLS provides automatic group setup and restores the MLS group when failure occurs. The software consists of three main components:

- Cisco's MLS++ library implementing MLS
- The implementation of Totem
- The main program integrating the components and adding communication with the UAV system

This project develops a software implementation of the Totem protocol and integrates it with the MLS library in the main program. Figure 4.1 outlines the message flow. When MLS receives application messages from the drone system, it encrypts them and sends them directly to their destination through the drone's radio interface. When receiving application messages from the radio, MLS decrypts them and sends them directly to the drone system. MLS generates the handshake messages without communication with the drone system. These messages are not sent directly to the radio but through the Totem module. This is to ensure that all nodes agree on the order of the handshake messages and can apply them correctly to avoid corruption of the MLS session.

[1] `https://github.com/emilmarstrander/Flamingo-MLS`

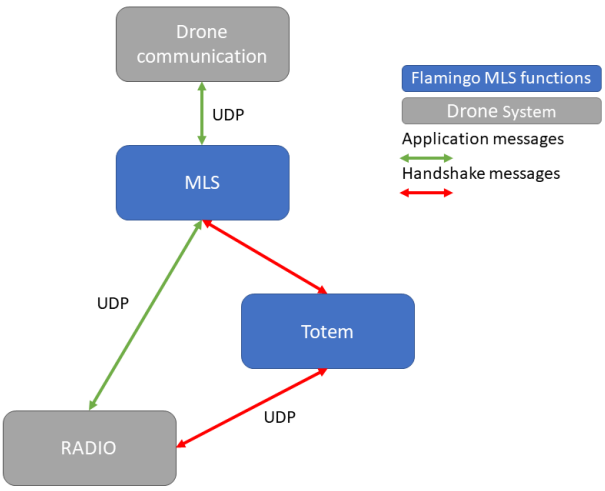


Figure 4.1: The message flow in Flamingo MLS

Figure 4.2 shows a technical overview of the different modules of Flamingo MLS and how they interact. The modules use MLS through the MLS session object. All modules, except for Proposal creation, are realized as threads to allow for parallel operation. Application and handshake traffic is distinguished based on UDP port numbers. The Handshake message handler, Proposal creation, and Totem communicate through memory objects. We briefly describe the modules below and provide more details in the following sections.

Handshake message handler Responsible for setting up the environment, initializing the other modules as threads, and handling handshake messages for the MLS protocol.

Totem Runs the Totem protocol, which transmits and receives MLS handshake messages. When the Totem token is received, this thread also runs the Proposal creation module.

Proposal creation This module is responsible for creating proposal messages for MLS and transmitting them to Totem.

Local receive Receives application messages from the drone system, encrypts them using the MLS session object, and transmits them to the radio.

Local transmit Receives application messages from the radio, decrypts them using the MLS session object, and transmits them to the drone system.

Software development is a central part of this project. We use modular programming, developing different modules independently and then integrating them. This method makes it possible to change modules if necessary. For

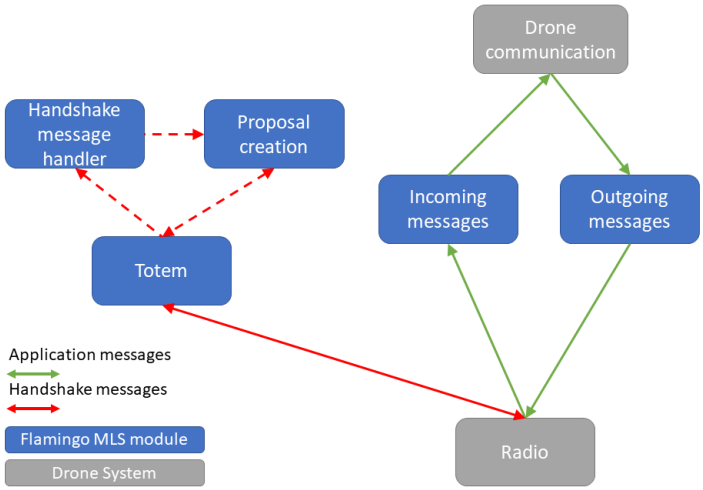


Figure 4.2: Technical overview of Flamingo MLS. All solid lines use UDP, while dotted lines represent communication through memory objects.

instance, if we discover a more suited Total Order Broadcast algorithm, we can easily replace Totem. One challenge with this methodology is to create clean and functional interfaces between the different modules, allowing them to communicate orderly and efficient. These interfaces are essential in order to swap modules easily.

We started the development process by learning how the MLS library worked and experimenting with different features. Then, we created a prototype for the main program, which simulated communication with the drone and used the MLS library to implement MLS functions. Next, we developed the Totem protocol in software and tested it independently. After this, we integrated the Totem implementation into the main program.

The development of MLS and Totem started with clean and ordered interfaces between the modules, but these clean interfaces degraded over time. After building the program's foundation, it was more complex to implement new features using the existing interfaces. For instance, MLS and Totem only communicated through the functions transferring messages. When MLS needed information about the Totem state in other situations, we needed to make an additional interface. Ultimately, this resulted in several unorderly structured interfaces that are difficult to understand. A more comprehensive planning of the software at the beginning of the project could have prevented this.

4.2 Implementation of Totem

The Totem Single-Ring Ordering and Membership Protocol ensures the total order of broadcast messages. It ensures the order by allowing nodes to send

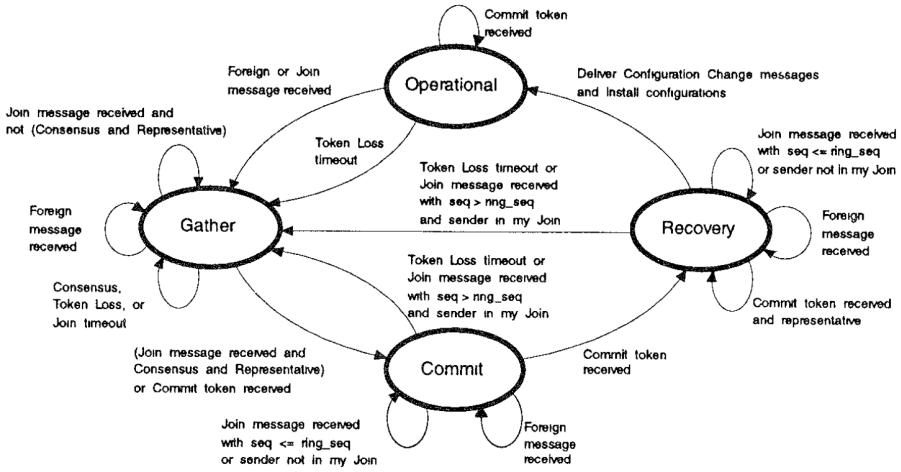


Figure 4.3: Overview of Totem states and operation [13].

only when given a token that circulates among the members. The sender marks the messages with a sequence number retrieved from the token, which ensures a unique sequence number for all messages. The protocol handles system reconfiguration with failures, restarts, and when network partitions and merging occur. In this section, we will go through the most essential concepts of Totem before describing how we implemented it in software.

Amir *et al.* [13] published the protocol in 1995. The technology has advanced significantly since the paper's publication, meaning arguments regarding high transfer rates and efficient mechanisms might not be relevant in today's high-speed wireless networks. However, this project will only use the Totem protocol on MLS handshake messages. These are sent infrequently to avoid excessive computational and network load. The main requirement is, therefore, reliability, not speed. Network equipment is more reliable today than in 1995, so we argue that a protocol that provides reliable operations in 1995 would perform even better today.

Totem delivers messages in an *agreed order*, which guarantees delivery to the application in a consistent total order. A message is only delivered if all previous messages have been delivered. Totem does not guarantee the delivery of all messages but the agreed order of all delivered messages. Since the environment allows for node failures, it is not certain that a node is available to receive the message.

Figure 4.3 shows an overview of Totem operation and the different states of the protocol. The *operational state* is where messages are sent and delivered in agreed order. Totem enters the other states when it detects a failure. In the operational state, the nodes transmit the token in a ring, and members can send messages when they have received the token. When the node has reached the maximum number of sent messages or the input buffer is empty, the node

forwards the token to the next member on the ring.

The operational state provides a retransmission feature. When a node receives the token, it checks whether it has received all messages up to the current sequence number. If not, it adds the missing sequence numbers to the *retransmission field* in the token. When the next node receives the token, it checks the retransmission field and retransmits the messages. For this to work, all nodes must buffer received messages until we are confident that every node has received every message. Totem detects this using the *all-received-up-to field* in the token, which specifies the highest sequence number received by all nodes. If this value equals the sequence number for more than one round, every node has received all messages, and the nodes can empty their buffer.

The operational state continues to provide agreed delivery of messages until a failure occurs. Totem detects failures by token loss using timeouts. The node retransmits the token when the *token retransmission timeout* occurs and continues to retransmit until the *token loss timeout* occurs. Then, the protocol enters the *gather state* to recover from the token loss.

The purpose of the gather state is to detect all other nodes that should be a part of the ring. The state ends when there is a consensus between the nodes. Information is shared using *join messages* containing information about detected and failed nodes. The nodes keep track of this information and update when it receives a join message. They then broadcast a join message with this newly updated state. When the node has received a join message equal to its own state from every node that should be a part of the ring, it is ready to move to the next state.

In the gather state, there are timeouts that ensure this process eventually converges. The *join timeout* occurs when the node does not receive a join message for the specified time and triggers the sending of a new join message. The *consensus timeout* occurs when the nodes do not reach consensus within the specified time. The node records what nodes it disagrees with, and the gather state restarts. Totem also uses the gather state to allow new nodes to join the group. We detect other nodes by receiving messages from unknown nodes, so-called *foreign messages*.

When the nodes reach consensus, Totem moves to the *commit state*. The node with the lowest ID initiates the transition. This node sends a *commit token* through the new ring. Every member updates the token with information about the old ring ID, received messages, and the need for retransmission. The purpose of the commit state is to ensure all members are committed to the new ring and to gather the node information. The token is transmitted for two rounds before the nodes enter the *recovery state*. The nodes detect token loss in the same way as for the operational state.

The recovery state aims to retransmit missing messages from the previous ring. The node initiates the state when receiving the commit token for the second time. The commit token is converted into the regular token with an additional field and keeps track of what messages need retransmission. New

ring members should not need to receive messages from the old ring. When all messages have been retransmitted and received, the protocol moves to the operational state and continues to provide agreed delivery of messages.

4.2.1 Implementation

We implement the Totem protocol using the C++ programming language. Even though the protocol is well documented in the paper of Amir *et al.* [13], some parts had to be interpreted during the development. We describe the relevant parts later in the section.

We did not implement all Totem features because we considered them unnecessary for this application. The *configuration change message* is a feature to communicate configuration changes to the application. Communication between Totem and the rest of Flamingo MLS instead uses memory. The advantage is that it is easier to program and more efficient since both the application and Totem will be part of the same process. The authors describe flow control mechanisms intended to optimize the protocol for high message throughput. Since the high throughput of MLS handshake messages is of no concern, we chose not to implement the feature. Still, Flamingo MLS contains some simple flow control mechanisms, such as a maximum number of messages each node can send before forwarding the token.

In Flamingo MLS, we implement the different messages and tokens as classes. When sending a message over the network interface, we represent it as a text string generated by the `serialize()` method of the message- or token class. The function serializes the messages by converting the value to text and separating the values with a "|" symbol. It uses "," and "!" when the values consist of multiple sub-values that need separation. The position in the string decides the value's interpretation, which changes for every message type. The first value is always the message type, which determines how to interpret the rest of the message. The following string is an example of a commit token: `5|8|4|0|0|0|1,0,0,0,0!2,0,0,0,0|0`.

The serialization solution is inefficient, but we use it to simplify the development in this project. The generated packets are larger than necessary because of the separator, and every character is treated as an 8-bit character. Most of the values do not need an 8-bit representation. The message type only has seven different values, which means that three bits would be enough to represent it. We could also remove the separator and instead use the exact location of every value at the bit level. The overhead for a regular message is approximately twenty bytes, and the regular token is approximately thirty bytes. An MLS handshake message can be larger than one thousand bytes. The overhead generated by this inefficient solution is therefore not significant and not a priority to solve in this project. However, it is still beneficial to produce less overhead, and we identify improving this serialization as future work.

The Totem class contains the protocol implementation and all methods and values necessary for protocol operation. A critical method in this class is the receive() method, which takes the message in string format as input. When receiving a message, it decides the type and converts it to the appropriate object. It then invokes functions specific to the state and message type.

The Totem protocol transmits the token as soon as the messages are sent. This benefits high throughput systems because the channel is idle for as short a time as possible. However, it is not necessarily beneficial for handshake messages in MLS, where the goal is reliable operation. Therefore, we add a delay before sending the token to the next member, which slows down the speed of the operation, making faults less likely and reducing resource consumption. The functionality creates a new thread that sleeps for the delay duration before transmitting the token.

We implement the Totem timeouts by recording the time the timer starts and regularly checking for timeouts based on the recorded time. We found this to be the most reliable solution. A disadvantage is that the timeout trigger could be more precise. However, it varied by only a couple of tens of milliseconds, which did not significantly affect protocol operation. Earlier in the implementation, we used threads, locks, and conditional variables to achieve the functionality. This created many faults and was challenging to debug.

The duration of the timeouts is configurable and depends on several parameters. Token retransmission and token loss timeouts are closely related. The retransmission timeout should give the protocol a second chance to succeed in the operational state by retransmitting the token. When this does not work, the token loss timeout should initiate the gather state to restore the proper protocol operation. Therefore, the token retransmission timeout should be smaller than the token loss timeout. The difference decides how many times we retransmit the message. Based on experience in this project, two retransmissions are suitable before initiating token loss procedures. Both timeouts should be larger than the expected turnaround for the entire ring. If not, timeout will regularly occur and hamper protocol operation. The timeouts' value depends on the number of members, latency for the channel, and the delay added when sending regular tokens.

Join and consensus timeouts are also closely related. As for the retransmission timeout, the join timeout gives the protocol a second chance to achieve consensus in the gather state by having every node send another join message. If this does not work, the consensus timeout is triggered, adding the nodes that did not achieve consensus to the failure set. The join timeout should, therefore, be smaller than the consensus timeout. Based on experience, two join timeouts before consensus timeout is a reasonable value. We can set these values lower than the token loss and token retransmission timeouts because they do not depend on the ring. The factors determining the values are the latency of the channel and how often we expect packets to be lost. The values should be set larger than the highest expected latency. If there is a high

packet loss, more join timeouts could be allowed before the consensus timeout is triggered.

When implementing the Totem protocol based on the descriptions of Amir *et al.* [13], it was not always clear how the protocol should behave. Therefore, we made some assumptions and adjustments based on what was reasonable during development. We document most changes here, sorted by what operation or state they affect.

Shift to gather Resetting the consensus array. The array contains outdated information from the old gather state, which is not valid anymore.

Shift to commit Added resetting of my_token_aru to 0. Since it is a new ring with a new ID, resetting the token's all-received-up-to value is logical. old_ring_aru and high_delivered are set to the same value in memb_list in the commit token. The all-received-up-to value and highest sequence number delivered are the same in all cases analyzed in this project. Therefore, we treat them the same in the implementation.

Shift to recovery Setting the sequence number of the commit token to 0. Since it is a new ring with a new ID, resetting the token's sequence number is logical.

Shift to operational We do not deliver configuration change messages to the application.

Operational state We update my_aru when messages are delivered. Therefore, my_aru is both a measure of received and delivered messages. The specification says the retransmission request list should be updated whenever a message is received. Instead, we do this when receiving the token because this is when we use the list.

Gather state When a token loss timeout occurs, Totem specifies to perform consensus timeout and shift to gather functionality. However, shift to gather might also be performed in the consensus timeout function. Therefore, we make sure to run the shift to gather procedure only once. When only a single node is part of the network, Totem does not provide a way to move to the next state. However, delivering messages with only one member is necessary in Flamingo MLS. To solve this, we added a conditional statement to the consensus timeout procedure, which checked if only one member was part of the group. In that case, Totem continues to the commit state. We also added a feature that ensures the node does not put itself in the fail set. If that happens, there is no mechanism for reversing it, and the node will no longer be operational.

Commit state We removed functionality for join timeouts because they should not occur in the commit state.

```
1: m ← received from handshake message socket
2: if m.type = regular message then
3:     if m.seq = next sequence number then
4:         Deliver m
5:         Deliver buffered messages if correct order
6:     else
7:         Buffer m
8: else if m.type = regular token then
9:     Invoke Proposal creation module
10:    Retransmit lost messages
11:    Transmit messages added to input buffer
12:    Transmit token to next member after delay.
13:    Reset token retransmission and token loss timeouts
14: else if m.type = join message then
15:    Shift to gather state
16:    Send join message
17: else if m.ring_id = incorrect and m.sender_id ≠ id in member list then
18:    Shift to gather state
19:    Send join message
```

Figure 4.4: Pseudocode for the operational state in the Totem module

Recovery state Totem uses a regular token in the recovery state, with an added value of retrans_flg. We implement this as an own type of token, *the recovery token*. The token converts to a regular token when shifting to the operational state. This simplifies the program development.

Figure 4.4 shows the pseudocode for how the Totem module functions when Totem is in the operational state. The code runs in a continuous loop.

4.3 Implementation of MLS

Since MLS only recently became an official standard, no complete implementations exist. However, multiple implementations are available, developed alongside the MLS drafts. The MLS Working Group GitHub repository[2] maintains a list of current implementations and their versions. When this master's project started in January 2023, only the following four implementations were available:

- MLS++ [16]
- OpenMLS [17]
- MLS-TS [18]
- GO-MLS [19]

[2] `https://github.com/mlswg/`

Both MLS-TS and GO-MLS were outdated and did not have a full implementation of MLS. They are now marked as outdated by the MLS Working Group. The only usable implementations were MLS++, developed by Cisco in C++, and OpenMLS, developed by Phoenix R&D and Cryspen in Rust. They were both quite similar in implemented features. For the use cases in this project, the only relevant feature missing in OpenMLS is that it only supports basic credentials. It cannot use certificates, so testing the certificate concepts described in Chapter 3.1 will be challenging. MLS++ supports X.509 certificates but does not have a way of verifying the certificate chain. We perform some practical testing of AS concepts in this project but do not prioritize it because of a lack of implementation support. It could, however, be useful in later projects.

Besides credential support, the implementations have little technical differences except for the programming language. The individuals involved in the development had a greater familiarity with C++ compared to Rust, making MLS++ the more straightforward choice. It is also the library used by Leon and Britt [3] and Dietz [4] at the NPS. The project started with using this library, and considerable familiarity was established before venturing into other possibilities. Therefore, we chose MLS++ as the library for this project. One downside with MLS++ is its lack of documentation, while OpenMLS had documentation available on its website. This has, in retrospect, been identified as a significant shortfall and made the development more complicated than necessary. Therefore, OpenMLS should be considered for future work.

The final version of Flamingo MLS contains a version of MLS++ downloaded from Cisco's GitHub repository[3] on 13th August 2023. The repository continuously updates, but we did not implement newer versions to ensure the version's reliability before the scheduled testing. MLS++ uses the Session class as the basis for all MLS operations. This class interacts with the rest of the library to provide the required services. We use the following methods of the Session class in this project.

protect(plaintext) Encrypts application messages and returns the encrypted message.

unprotect(ciphertext) Decrypts application messages and returns the decrypted message.

add(key_package_data) Creates an add proposal message for the node in the provided key package data and returns the encrypted add proposal.

remove(index) Creates a remove proposal message for the member specified by the given index and returns the encrypted remove proposal.

update() Creates an update proposal message and returns the encrypted update proposal.

[3] `https://github.com/cisco/MLSpp`

commit() Creates a commit message for the currently handled proposals and returns the encrypted commit.

handle(handshake_data) Handles the commit or proposal message provided as handshake data. A message does not take effect before the program runs this method on the message. It returns a boolean value based on whether it handled the message correctly.

The Client object initiates the session. The client requires a cipher suite, a private signature key, and a credential. It can then create a new session or generate a key package to join an existing session. Flamingo MLS uses the cipher suite P256-AES-128-GCM-SHA-256-P256. This is one of the cipher suites Leon and Britt [3] found to perform worst for time used when handling messages. We chose this particular suite because it is the only one using the P256 curve, which was more accessible to manipulate in the OpenSSL library. This made it easier to create and manipulate X.509 certificates.

A challenge with MLS++ is that there is no simple way of retrieving information about the MLS messages. It is, for instance, not trivial to distinguish between application and handshake messages. Since we must insert the message into the correct method of the Session object, it is essential to know what message type is received. In order to achieve this, we use different UDP ports for application and handshake messages. The handshake messages themselves must also be distinguished. It is irrelevant for proposal and commit messages since the handle() method can handle both. The challenge is for key packages and welcome messages. We need to insert them using special methods and, therefore, be able to distinguish them. The solution in Flamingo MLS, inspired by Dietz [4], is to add a number at the start of each handshake message, symbolizing what kind of message it is.

A security consideration when using MLS++ is that it is possible to decrypt messages from old epochs as long as they have not been received before. According to the MLS standard, it is up to the application to define a policy for how long to keep unused nonce and key pairs from older epochs. We have yet to find an obvious way of specifying this limit in MLS++. Therefore, MLS++ keeps the unused values for the duration of the session. Keeping old keys is useful when messages are delivered late but affects the forward secrecy of the application. If a node is compromised, it could still have keys and nonces for older messages not yet received, which means those old messages are not forward secret.

Flamingo MLS uses credentials with X.509 certificates. We do this by retrieving the public key from the SignaturePrivateKey object. Since no AS is present in the system, Flamingo MLS signs the certificate using a randomly generated signature key pair. It also adds a corresponding AS certificate to the certificate chain (der_chain). Every node will produce different AS keys and certificates, and cannot verify each other's chains. However, the program still works since MLS++ does not verify the last certificate in the chain. This

Table 4.1: Size of MLS messages

Message type	Size	Comment
Update	750 B	
Commit	400 B	
Add	1100 B	
Welcome	1700 B	2 members, no certificate
Welcome	19 000 B	10 members, no certificate
Key package	800 B	no certificate
X.509 certificate	2600 B	

means that we cannot be sure of the identity of the nodes, so anyone can create a fake certificate chain and join the MLS group. It is possible to make some changes to the source code of MLS++ to improve security. A simple solution could be to add a check of the last certificate in the chain and compare that to the list of root certificates of the host machine.

Table 4.1 gives an overview of the expected sizes for different MLS messages in MLS++. We obtained these values by running MLS++ and recording the message sizes. The values are only valid for the parameters used and only intend to give a rough estimate of the sizes. We used two members if nothing is specified, and the cipher suite was the same as specified earlier. The size of the handshake messages is generally between 400 and 1100 bytes, except for welcome messages, which could become significantly larger. Certificates add substantial size to key packages, add proposals and welcome messages when used. The handshake messages are large and could generate notable overhead if sent frequently.

An overall experience when implementing MLS is that using both proposals and commits is unsuitable for a high-speed environment. During testing, updates occurred approximately every 100 ms, and there were many situations where having to process both caused handling errors. These can easily be suppressed and do not cause a problem for the protocol operation. However, it is unnecessary to use computational and network resources to handle messages that are not valid. When updating at this speed, a node is unlikely to collect proposals from other members before committing. Therefore, the benefit of proposal messages diminishes. It is more beneficial to commit immediately after creating the proposal message, which is how Flamingo MLS does it.

The MLS++ library has two main flaws discovered during development. We reported both as *issues* at the MLS++ GitHub repository but without response. The first is that it produces the following error message when committing update proposals: `Invalid proposal list`. This indicates that there is an error in the creation of the update proposal. This was not an issue in the MLS++ build from December 2022. The workaround in Flamingo MLS is to avoid creating an update proposal and instead create a commit message that references no proposals, an *empty commit*. This will update the committer's

1: Receive message from drone
2: Protect message
3: Broadcast protected message over the radio

Figure 4.5: Pseudocode for the Local receive module

1: Receive message from radio
2: Unprotect message
3: Transmit unprotected message to drone

Figure 4.6: Pseudocode for the Local transmit module

contribution to the group and provide post-compromise security with regard to the committer.

Another problem occurs when reducing the MLS group to a single member. It produces the following error: `Malformed UpdatePath`. This indicates that something goes wrong when trying to update the ratchet tree. It is unknown whether this is a flaw or simply an unrealistic scenario since reducing an MLS group to a single member in a messaging system is usually pointless. Why have a secure group with no one to communicate with? The solution in Flamingo MLS is to create a new session containing a single member.

We made some changes to the MLS++ source code to make the program more straightforward to develop and avoid errors. One change was creating a way to initialize an empty Session object as a global variable. We added a default constructor to the class in `session.h` and `session.cpp`.

By default, MLS++ keeps the state information for all previous epochs indefinitely. This can consume much unnecessary memory[4]. To improve this, we restrict the number of epochs that the Session can store. We insert this restriction in the handle() function by deleting excess elements from the history vector in `session.cpp`. We create the size_of_history variable of the Session class in `session.h` to decide the new max size.

We now move on to the four different modules that run MLS in Flamingo MLS: two handle application messages and two handle handshake messages. Figures 4.5 and 4.6 show the pseudocode for the Local transmit and Local receive modules. These modules run as threads listening to dedicated UDP ports and handle application messages. They receive a message, protect or unprotect it, and then pass it to the destination.

The Proposal creation and Handshake message handler modules create and process handshake messages. The Proposal creation module is the only module that creates and sends proposal messages and commits. Flamingo MLS invokes the module after receiving the Totem token, which synchronizes the sending of handshake messages with the token. Early in the development, the working solution was to send handshake messages as soon as possible and then

[4]We demonstrate this in Section 5.3.

```
1: if members to remove then
2:     Create and handle remove proposals for all members to remove
3:     Broadcast remove proposals
4:     Create commit of the proposals
5:     Broadcast commit message
6: else if received key package then
7:     Create and handle add proposals for all received key packages
8:     Broadcast add proposals
9:     Create commit of the proposals
10:    Transmit welcome message to the joining members
11:    Broadcast commit message
12: else if time since last update > update interval then
13:    Create and handle update proposal        ▷ Not supported by MLS++
14:    Broadcast update proposal                ▷ Not supported by MLS++
15:    Create commit of the proposal
16:    Broadcast commit message
```

Figure 4.7: Pseudocode for the Proposal creation module

let Totem deal with ordering and delivery of the messages. This method induced a number of errors. When node A generated handshake messages, it was unaware that node B had already generated the same type of messages. Node B had not sent them because it was still waiting for the token. This resulted in two different commit messages created for the same epoch. If everything works as intended, this is unproblematic and will only generate error messages. However, we experienced several faults because of this, so we changed to have more synchronization between the Totem token and the broadcasting of handshake messages. The node does not transmit the token before both proposal and commit messages are broadcast. This leaves less room for error since only one node has active proposals at a time.

The Proposal creation module checks whether it is necessary to create remove, add, or update proposals before generating the appropriate messages. It first checks if it should remove any members. Flamingo MLS implements four reasons to remove members:

- Member has not updated in a long time
- Member is no longer in the Totem group
- Member is not sending application messages
- There is a duplicate of the node in the MLS group

We can configure what reasons to use in the configuration file. The Proposal creation module checks for stored key packages and adds the nodes to the group. Lastly, it checks the time since sending the last update message to decide if a new update is necessary. Figure 4.7 shows the pseudocode for the Proposal creation module.

```
1: m ← received from Totem
2: if m.type = key package then
3:     Add key package to object containing received key packages
4: else if m.type = commit message then
5:     Handle commit
6: else if m.type = proposal message then
7:     Handle proposal
```

Figure 4.8: Pseudocode for the Handshake message handler module

The Handshake message handler receives and handles handshake messages based on the type. For add, update, remove, and commit messages, the module handles them using the handle() method of Session. When key packages are received, they are stored until processed by the Proposal creation module. Figure 4.8 shows the pseudocode for the module.

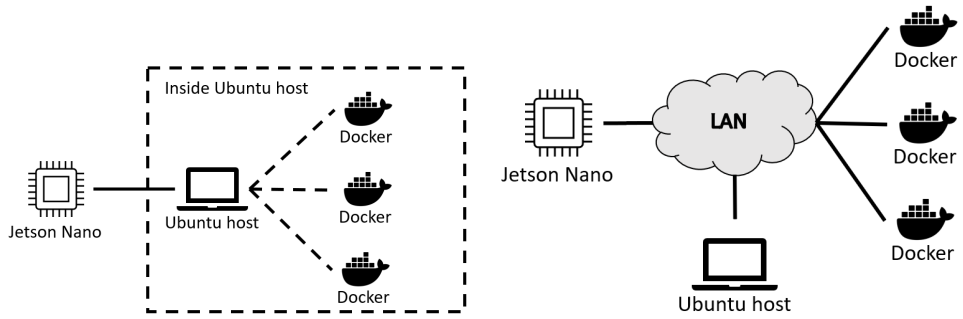
Chapter 5

Testing in a Simulated Environment

One of the most critical parts of software development is to evaluate how it performs. Many ideas work well in theory but fall short in practical use. In this chapter, we evaluate how Flamingo MLS performs in a simulated environment.

A simulated environment allows for controlled measurements of the performance of Flamingo MLS. We can easily monitor and adjust network and program parameters to support measurements of essential features. The intention is to evaluate the performance of the software with different parameters and under different circumstances. Resource consumption is the main focus. With limitations in size and weight, the drones are a resource-constrained environment. The drones need the resources to perform their mission, and therefore, Flamingo MLS should consume as little as possible. The most relevant resources for this project are CPU time, RAM usage, and network usage. We do not test network usage because the onboard radio is the bottleneck for network traffic and is not available for the simulated testing. However, since the overhead for MLS and Totem is relatively small, it will not significantly increase network bandwidth.

One Jetson Nano Developer Kit was available for the duration of testing. This is functionally the same hardware that runs on the Flamingo drones, with reduced performance because of a slower CPU and less available RAM. We make the resource consumption measurements on the Jetson Nano to get measurements that are as realistic as possible. We simulate the rest of the nodes using *Docker containers* [20]. A Docker container runs an operating system instance in a closed environment on a physical host. The host can run multiple containers which can use all the available resources at the host. Ideally, the Docker containers should run Linux for Tegra, the operating system used on the drones, using an ARM processing architecture. This was the original setup, but the Flamingo MLS application could not run properly on these Docker images for unknown reasons. The program terminated because of runtime errors, like `bad_optional_access` and `Unexpected index`. We used



(a) Physical **(b)** Logical

Figure 5.1: Test setup

images running Ubuntu instead. This should not impact the test results since we are performing the measurements on the Jetson Nano.

Figure 5.1a shows the physical setup for the tests. The Jetson Nano is connected to the Ubuntu machine using an ethernet cable. The docker containers run on the physical Ubuntu machine. We use the following equipment:

- Jetson Nano Developer Kit
- HP EliteBook 840 G6 running Ubuntu 23.04

Figure 5.1b shows the logical network topology. Using the *macvlan network driver* in Docker, the containers are placed on the same Local Area Network (LAN), meaning they are on the same IP subnet and can communicate directly without going through a router. This is the same logical setup as when the drones communicate over the radio channel.

We use the Top utility for measurements of resource consumption. Top is a tool for monitoring processes running on the machine and can provide processor and memory usage. We use the following command for the measurements:

```
top -c -p $(pgrep -d',' -f flamingo_mls) -n 2 -b -S -d
    ↪ #INTERVAL#
```

This command searches for all instances of the `flamingo_mls` command and produces two measurements on that process. `INTERVAL` determines the time difference between the measurements. The second measurement is the most valid and shows the average resource consumption in the specified interval. This is usually five seconds, but sometimes we need a larger interval to get a more accurate average, for example, when sending update messages every twenty seconds. Then, the measurement is performed over twenty seconds to include the update message. We use `-S` to activate Cumulative Timer Mode, which means that Top lists each process with the CPU time of itself and its dead children.

Figure 5.2 shows an example output from Top. The measurement would

```
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3.5 us,  1.1 sy,  0.0 ni, 95.3 id,  0.1 wa,  0.0 hi,  0.1 si,  0.0 st
MiB Mem :   1828.3 total,   1271.1 free,    227.0 used,    330.3 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   1460.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 3510 emil      20   0  315596   9488   7684 S  21.0   0.5   0:27.22 ./flamingo_mls
```

Figure 5.2: Output from the Top command

be 21.0 % CPU and 0.5 % RAM. The CPU percentage is for one core, so the maximum for the Jetson Nano with four cores is 400 %. Information on the Top command can be found in the manual [21].

We use the Multi-Generator (MGEN) Network Test Tool to simulate application traffic. MGEN is a tool developed by the United States Naval Research Laboratory [22] to measure and analyze the characteristics of a network. It is used by configuring a sender and a receiver. In the tests, MGEN sends UDP packets, and we use message interval and size as test parameters. The receiver generates a log file of the received packets.

The sender uses the following command. `FLOW` is an identification number used to identify the sender. `FREQ` and `SIZE` are the frequency and size of the packets. `127.0.0.1/4100` is the destination address, which is the address configured to receive application messages in Flamingo MLS.

```
./mgen event "ON #FLOW# UDP SRC 5001 DST 127.0.0.1/4100
    ↪ PERIODIC [#FREQ# #SIZE#]" output log_send.drc txlog
```

The receiver uses the following command. `4000` is the port used to receive application traffic from Flamingo MLS.

```
./mgen event "listen udp 4000" output log_recv.drc
```

Tests 1 through 4 aim to measure resource consumption on Jetson Nano during different scenarios. The goal is to see how different parameters affect resource consumption. We perform most tests multiple times and present the average measurement. Test 5 investigates how resource consumption is distributed throughout the application. Test 6 performs testing under varying network parameters. Tests 1 through 4 evaluate the following Flamingo MLS parameters.

UPDATE_INTERVAL How often each node generates MLS update messages.

TOKEN_SEND_INTERVAL How long a node keeps the Totem token before sending it to the next node.

HEARTBEAT_INTERVAL How often each node sends application heartbeat messages to prevent other nodes from removing it. Note that we only send heartbeat messages if no other application messages have been sent in the interval.

LOG_LEVEL Determines how many program operations are written to the log files.

MAX_EPOCH_STATE_HISTORY_SIZE Determines how many previous MLS epoch states are stored. We refer to this as max epoch states in this chapter.

In addition, two configurations significantly affect performance: the number of nodes in the network and how much application traffic they each send.

Some of these parameters and configurations depend on each other to have an effect. For instance, there is no point in having an update interval lower than the token send interval because the node only sends updates after receiving the token. Another example is that the heartbeat interval is only relevant if it is lower than the interval between application messages sent.

The parameters also depend on other program parameters, but they are not in focus in these tests. For instance, if the token send interval is too high, it will trigger the token retransmission timeout without the token being lost. To avoid this situation, we set relevant parameters to the necessary values. This does not affect the resource consumption measured but enables the program to function as it should during testing. We provide the entire configuration file for the test along with the source code at the GitHub repository presented in Chapter 4.

We choose the frequency and size of messages to fit the traffic characteristics of the UAV swarm. The developers of the Flamingo UAV stated that the frequency can be between two and fifty packets per second, and the size can be between 1 kB and 50 kB. Unfortunately, MGEN has a maximum UDP packet size of 8192 B. The simulated traffic, therefore, cannot test all scenarios but gives a good indication of how Flamingo MLS works under different traffic parameters.

5.1 Test 1: Parameter Testing

The parameters of Flamingo MLS impact resource consumption, and in Test 1, we measure how much effect they have. The setup consists of two nodes, and application packets are sent once per second with a size of fifty bytes. We turn off application packets when testing the heartbeat interval. We evaluate one parameter at a time and set the other variables to the standard values in Table 5.1.

Table 5.2 shows the results when varying the update interval parameter. There is a significant difference of almost 5 pp between 400 ms and 2 s and an almost 2 pp difference between 2 s and 20 s. However, there is no significant difference between 20 s and 40 s, which means we do not gain performance benefits above 20 s. Since two nodes are present, an update will happen every 10 s in this case. If more nodes are present, updates will be more frequent when using the same parameter value, and we must adjust the value accordingly.

Table 5.1: Standard parameter values

Parameter	Value
UPDATE_INTERVAL	40 s
TOKEN_SEND_INTERVAL	50 ms
HEARTBEAT_INTERVAL	20 s
LOG_LEVEL	2
MAX_EPOCH_STATE	10

Table 5.2: Update interval results

Update interval	CPU usage
400 ms	24.8 %
2 s	19.0 %
20 s	17.3 %
40 s	17.4 %

Table 5.3: Token send interval results

Token send interval	CPU usage
50 ms	17.4 %
500 ms	15.5 %
5000 ms	No result

Table 5.3 shows the results when varying the token send interval parameter. There is a reduction of almost 2 pp when going from 50 ms to 500 ms. This is a slight reduction, and as long as the update interval is sufficiently large, a token send interval of 50 ms is unnecessarily short. A larger interval will make changes to the group happen more seldom. For instance, if a member would like to add or remove a node, it would have to wait to receive the token to make the changes take effect. However, making these changes more often than every few seconds is rarely necessary. The token send interval could be even larger than 500 ms, but because of a bug in the software, the test with 5000 ms did not produce valuable results. The nodes did not manage to establish a group with this value. There were issues with forming a group at 500 ms, but we could eventually establish the group. Despite this, we still manage to produce reliable measurements.

Table 5.4 shows the results when varying the heartbeat interval parameter. There is a reduction of 6.5 pp when increasing the heartbeat interval from 200 ms to 2 s. The reduction was below 1 pp when increasing it further to 20 s. The heartbeat interval determines how quickly we detect corrupt sessions and can deploy measures. It is, therefore, important to have a low heartbeat interval. It should be closer to 2 s than 20 s, but not much lower than 2 s. The drones continuously send application messages during operations, so they should rarely need the heartbeat. The parameter must exceed the interval between application messages to avoid extra messages and resource consumption. The minimum message frequency of the drones is two messages per second, so a heartbeat interval between 1 s and 2 s is fitting. However, because of the resource consumption measured, it should be closer to 2 s unless we need a rapid detection of corrupted sessions.

Table 5.5 shows the results when varying the log level parameter. There

Table 5.4: Heartbeat interval results

Heartbeat interval	CPU usage
200 ms	22.8 %
2 s	16.3 %
20 s	15.5 %

Table 5.5: Log level results

Log level	CPU usage
0	16.9 %
2	17.4 %
5	38.1 %

is a minimal difference between log levels 0 and 2, but a more than 20 pp increase when using log level 5. This is because log level 5 writes to the log files for every line of code in some functions, which is very resource-intensive. This is only necessary when debugging tricky bugs. If not, the logging will have a significant effect on the performance. Log level 0 only reports error messages, while log level 2 logs program operations such as epoch changes, addition and removal of members, and Totem operations. Most faults do not result in error messages but rather unexpected program operations. Hence, it is necessary to have program operations logged with log level 2. Since the difference between 0 and 2 is very small, we can use it for all experimental activity without significantly affecting performance.

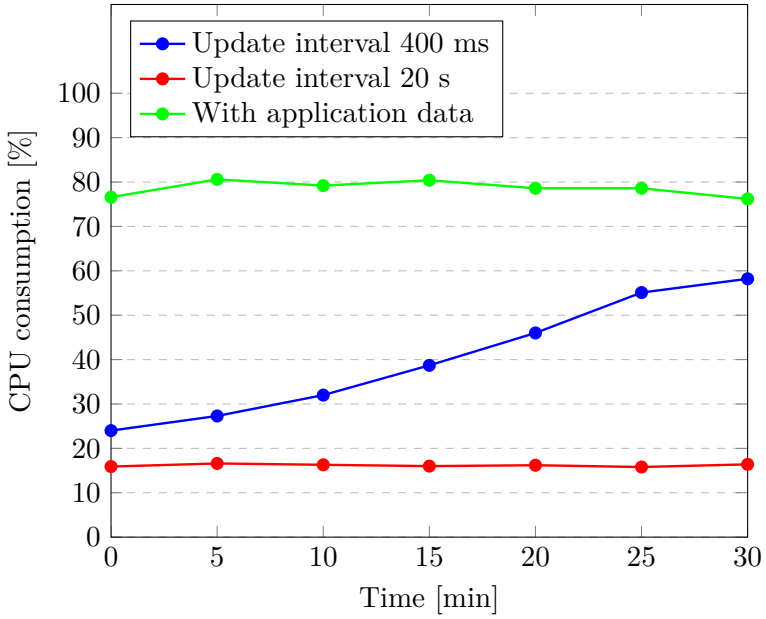
5.2 Test 2: Time-based Resource Consumption

It is not sufficient to measure resource consumption only when the program starts. The consumption can change over time. Test 2 measures how much this changes over thirty minutes. We perform the test multiple times with different update intervals, with and without application traffic. We use the same base parameters as in Test 1, except for the heartbeat interval, which we set to 2 s. When using application data, we send 100 packets per second containing 2 kB each and set the update interval to 20 s.

Figure 5.3 shows the results for Test 2. We found no significant increase when using an update interval of 20 s and with application data. However, with an update interval of 400 ms there was an increase of 34.2 pp over thirty minutes. An explanation could be that the program needs more CPU power in higher epochs. To explore this further, we performed a test where the nodes started at epoch 5000, see Figure 5.4. In the blue plot, we use a third node to create a session with epoch 5000, before nodes 1 and 2 joined. Node 3 then left the session. The goal is to see if this affected the resource consumption over time. The result shows no difference from the results in Figure 5.3, so the epoch does not seem to influence consumption. In the red plot, node 2 has an update interval of 400 ms, while node 1 has an update interval of 20 s. We perform the measurements at node 1. The values change between nodes 1 and 2 in the green plot. The results show that creating update packets more frequently is more resource-intensive than just receiving them.

We also measured RAM usage in this test. All measurements were at 0.2 %, except when the update interval was 400 ms. Then, it measured 0.7 % after

Figure 5.3: Results showing resource consumption over time



thirty minutes. Even though this is a significant increase, it is unlikely to affect drone operations because the RAM consumption in total is minimal.

5.3 Test 3: Stored Epoch States

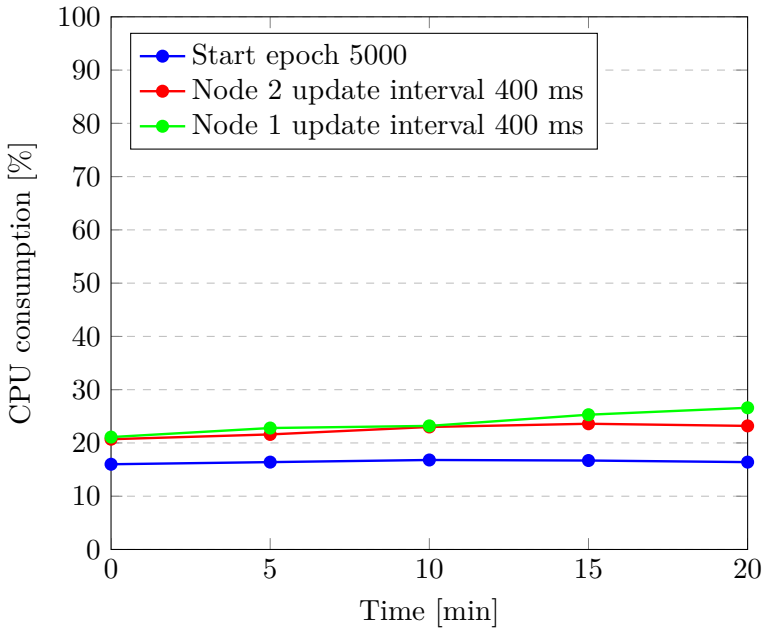
We discovered an issue with resource consumption during the development of Flamingo MLS. The source of the problem was too many old epoch states stored. The MLS++ library has no limitations, so we altered the source code to limit this. Test 3 demonstrates the need for this feature and helps determine what value to set. The only parameter changing is the maximum number of epoch states stored. We perform measurements after 1000 epochs and use an update interval of 100 ms to make the test faster to execute. The token send interval is 50 ms, the heartbeat interval is 2000 ms, and application messages are disabled.

Table 5.6 shows the results when varying maximum epoch states between 10 and 1000. When more epoch states are stored, there is a slight decrease

Table 5.6: Results for maximum epoch states stored

Max epoch states	CPU usage	RAM usage
10	49.2 %	0.3 %
100	47.9 %	0.7 %
1000	47.4 %	2.8 %

Figure 5.4: Further investigation of resource consumption over time



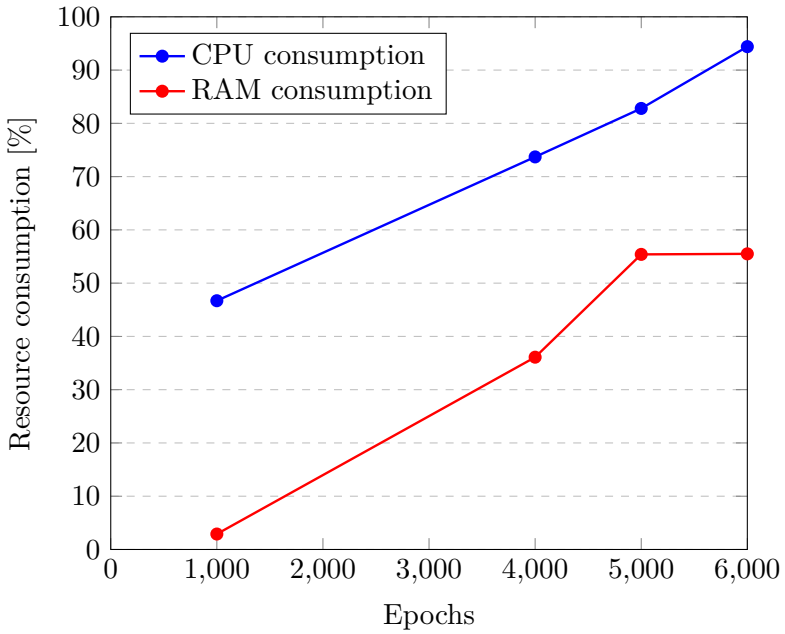
in CPU consumption. This was unexpected but could be explained by fewer computations needed to delete the stored states. The RAM usage increased significantly when storing more epoch states. However, the increase is not proportional to the increase in states stored. With ten times as many states to store, we would think closer to 7 % RAM would be used, but we measured only 2.8 %. One hundred stored states might be ideal because it provides low CPU and RAM consumption.

Figure 5.5 shows what happens when there are no limitations to the number of stored epochs (10000 epochs in practice). The figure shows how CPU and RAM consumption increase as the program progresses through the epochs. It then consumes significant portions of the system resources. At epoch 6642, it consumed so many resources that the program crashed. This shows why limiting the number of epoch states stored when using MLS++ is essential.

5.4 Test 4: Group Size-based Resource Consumption

It is useful to see how Flamingo MLS works in a controlled environment with only two nodes, but the real magic happens in larger groups. Test 4 investigates this by measuring how group size and message volume affect resource consumption. First, we sent no application messages and set the update interval so that updates happened every two seconds for the entire group, see blue graph in Figure 5.6. For three members, this means using an update interval of 6 s. Then, we set the update interval to five minutes so as not to influence

Figure 5.5: Results when storing all epoch states



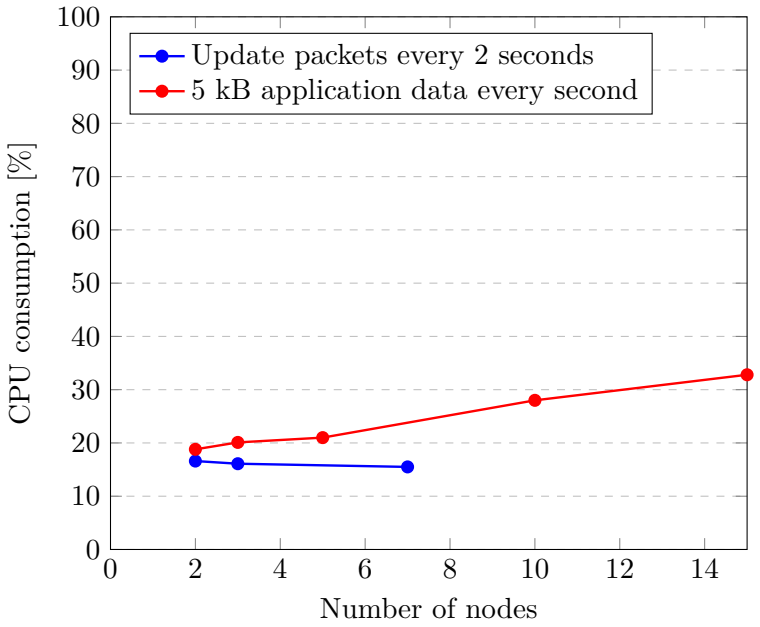
the rest of the test and sent application messages at a rate of once per second, with 5 kB per message. Figure 5.6 shows this in red.

CPU consumption marginally decreases when the number of nodes increases, with an update message sent every two seconds. An explanation is that the update interval per node increases when group size increase. Each node has to create update messages more infrequently, and as we saw in Test 2, it is more demanding to create rather than handle update messages[1]. When sending application data, CPU consumption increases with the group size. Every node receives more data in larger groups since all send the same amount. One would expect this trend to linearly increase CPU consumption when the number of nodes increases. Based on the graph, it is not perfectly linear, but it is close. The discrepancy is likely a result of uncertainty in the measurements.

The rest of Test 4 was supposed to investigate how different application packet sizes and frequencies affected resource consumption. However, when performing this, we discovered a significant packet loss. This happened both when sending to and from the Jetson Nano. Table 5.7 shows the only results achieved without significant packet loss. High CPU consumption is likely the reason for the packet loss. Strangely, this results in packet loss instead of increased latency, which we should investigate further. One reason might be full input buffers because of the high processing time for each packet. Regardless of this, the test shows an important point: it is more efficient to send fewer packets with larger sizes. Both the lines in Table 5.7 use the same data rate,

[1] We show this in Table 5.9 as well.

Figure 5.6: Results of resource consumption when group size increases

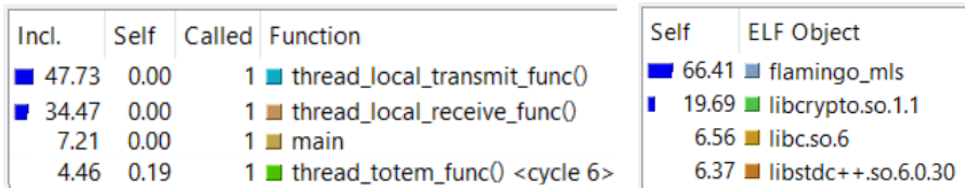


but there is almost a 40 pp difference in CPU usage. One explanation for this is that some processes are performed once per packet, regardless of size. Signature creation is one example of this. Having such processes performed more infrequently is beneficial for resource consumption.

After discovering packet loss during the testing, we changed the goal to find how much application data could be processed by Flamingo MLS before experiencing packet loss. The test used two nodes, and only one node was sending. We set the max size of 8192 B at the sender, and the frequency was changed to see how much traffic could pass through. Since sending large packets is beneficial for maximum throughput, this test should show the maximal rate that can be achieved with Flamingo MLS. The result was that the Jetson Nano could receive at a frequency of 90 Hz, a data rate of 5.9 Mbps. The CPU consumption was then at 100.9 %. It could send at a frequency of 120 Hz, a data rate of 7.9 Mbps. It then had a CPU consumption of 94.9 %.

Table 5.7: Results when sending application data

Frequency	Size	CPU usage	RAM usage
100 Hz	800 B	102.5 %	0.2 %
10 Hz	8000 B	63.2 %	0.2 %



(a) Based on threads **(b)** Based on ELF objects

Figure 5.7: CPU distribution in Flamingo MLS

5.5 Test 5: CPU Consumption Analysis

After seeing that Flamingo MLS consumes significantly more resources than expected, we seek to find an explanation. This is especially true when sending and receiving large amounts of application messages. To understand why this happens, we need to investigate how the CPU consumption is distributed in the program. We use Valgrind with the Callgrind [23] tool to track how many CPU cycles each function uses and create profiling data for Flamingo MLS. The command for doing this is:

```
valgrind --tool=callgrind ./flamingo_mls
```

Valgrind creates an output file that other programs can analyze, and we use QCachegrind [23] to analyze the data. The screenshots in this section come from QCachegrind.

We use two nodes: the Valgrind command runs one node, and the other runs normally. We use Ubuntu, and not the Jetson Nano, since the program runs similarly on an x86 and an ARM processor. The update interval is 40 s, and application data is sent five times per second, with 1000 bytes per message.

We create Figure 5.7a by searching for "thread" and "main" among the recorded function calls in QCachegrind, combining the searches into a single figure. It shows the distribution between the different program threads. The local transmit function, responsible for decrypting messages, uses 47.73 % of the total CPU cycles. The local receive function, responsible for encrypting messages, uses 34.47 % of the total. Based on the numbers, Flamingo MLS uses more CPU cycles to handle incoming messages than outgoing ones. In comparison, the main thread and the Totem receive thread use very few CPU cycles. This is as expected since the update interval is high.

Figure 5.7b shows functions grouped by ELF object. The Flamingo MLS object comprises everything this project created, including the MLS++ library. The remaining objects are libraries referenced in the program. `libcrypto.so.1.1` is the OpenSSL library MLS++ uses for all cryptographic operations. Since we find only 19.69 % of operations in this object, cryptographic computations are not the most resource-consuming part of the program.

Figure 5.8 shows a call graph generated by QCachegrind for the local receive and transmit threads. protect_data() and unprotect_data() are called 177

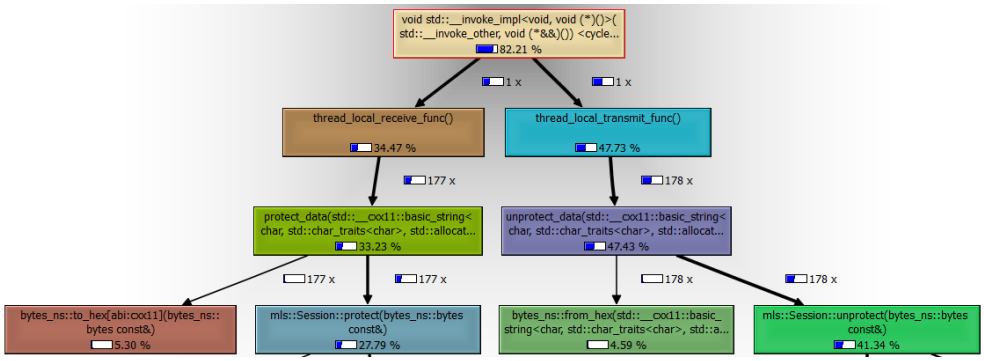


Figure 5.8: Callgraph for the transmit and receive functions

and 178 times, respectively. This means the number of messages sent and received are almost the same, and the results for encrypting and decrypting messages are comparable. The function calls on the last line are all MLS++ functions. Since they add up to 79.02 %, Flamingo MLS uses only about 3 %, while MLS++ and the libraries it calls use the remaining resources. The conversion to and from hexadecimal format uses almost 10 %, which is much for such a simple operation. It should be possible to improve this. mls::Session::protect() uses 27.79 %, while mls::Session::unprotect() uses 41.34 %. These are the functions responsible for protecting and unprotecting messages.

When protecting messages, MLS does the following cryptographic operations:

AEAD encryption Authenticated Encryption with Associated Data (AEAD) is symmetric encryption and authentication of the message.

Key derivation The derivation of the next key from the secret tree, used for AEAD encryption.

Signature creation Creating an asymmetric signature for the message.

The hpke::AEADCipher::seal() function performs the AEAD encryption. This function uses 0.22 % of the CPU cycles. Figure 5.9 shows the call graph for the key derivation process. The hash ratchet uses a Hash-based Message Authentication Code (HMAC) to derive new secrets. The function is both called from the protect and unprotect functions. The only part doing cryptographic operations is the hpke::Digest::hmac() function in the bottom left corner. Since the protecting and unprotecting functions have an equal share, 0.38 % is used by each.

Figure 5.10 shows the call graph for the signing process. Flamingo MLS calls the mls::SignaturePrivateKey::sign() function in two situations: signing the GroupInfo-object and signing a message. In addition, only two of the three functions shown as called at the bottom are doing cryptographic operations. These two functions add up to 6.06 %. The middle function is doing TLS marshal

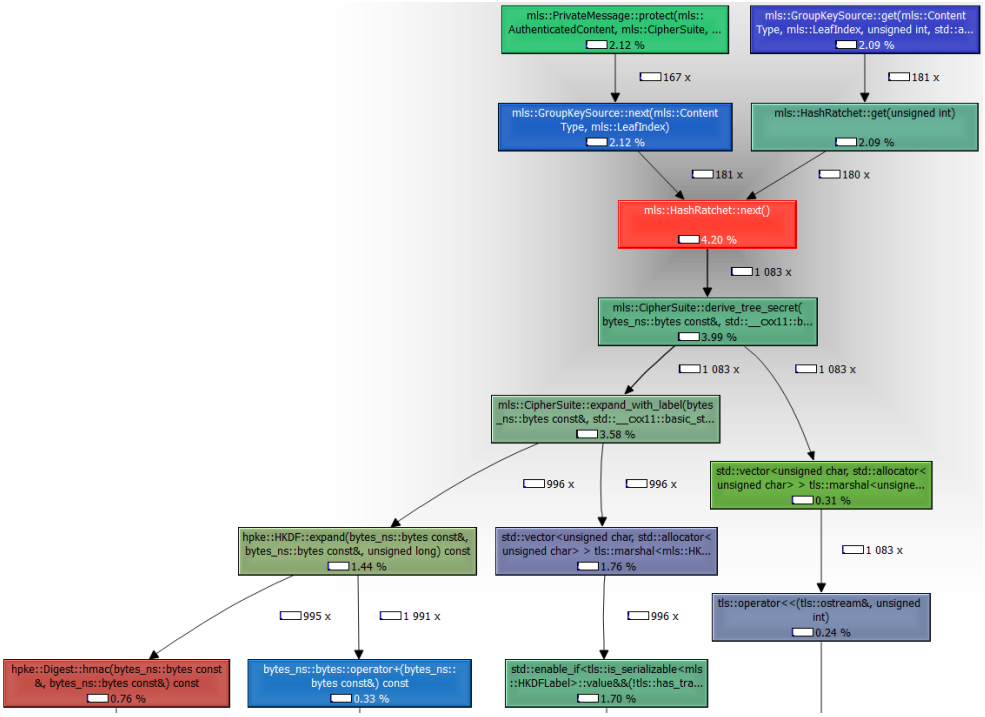


Figure 5.9: Callgraph for the key derivation process

operations. The path in the upper right corner is for protecting application messages, while the left is for GroupInfo-objects. Signing application messages are 87.4 % of all signatures created with this function. Therefore, the cryptographic operations used to perform the application message signing equals 5.3 % of the total cycles used.

By summing up the numbers for AEAD encryption, key derivation, and signature creation, the protecting function uses 5.9 % of the total CPU cycles for cryptographic operations. The signature process uses 5.3 % and is the most resource-intensive part. This amounts to 21.2 % of the computational load used by the protect function. Table 5.8 summarizes this.

When unprotecting messages, MLS does the following cryptographic operations:

AEAD decryption Symmetric decryption and verification of the message's authenticity.

Key derivation The derivation of the correct key from the secret tree, used for AEAD decryption.

Signature verification Verifying the asymmetric signature of the message.

The hpke::AEADChipher::open() function uses 0.21 % of the CPU cycles. The key derivation usage is the same as for the protect function. Figure 5.11

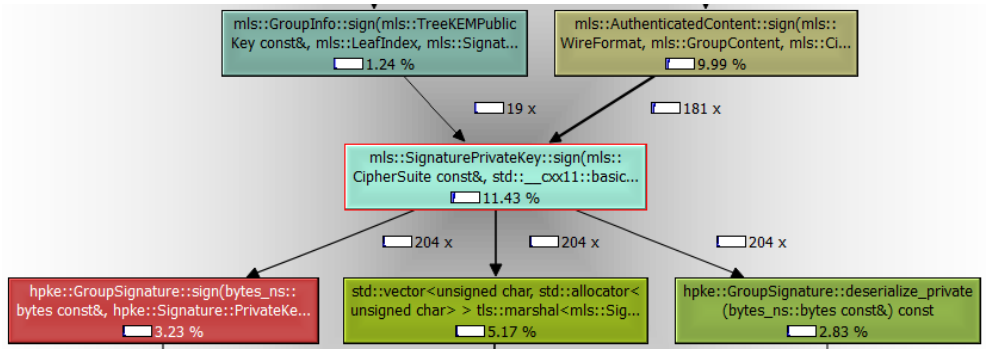


Figure 5.10: Callgraph for the signing process

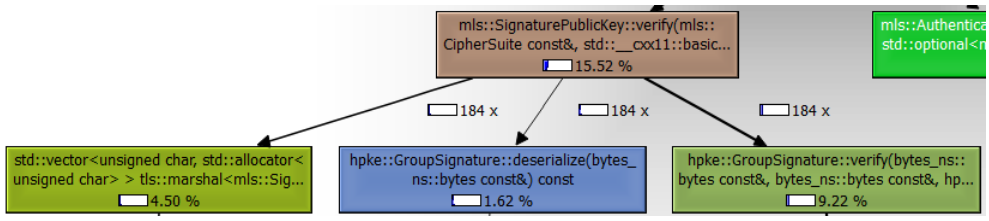


Figure 5.11: Call graph for signature verification

shows the call graph for the signature verification function. The two functions at the bottom row to the right are the only functions doing cryptographic operations. These add up to 10.84 %. In total, cryptographic operations equal 11.43 % for unprotecting messages. This is 27.6 % of the unprotect function total.

Table 5.8 summarizes resource consumption for cryptographic operations in the unprotect and protect functions. In total, Flamingo MLS uses 17.33 % for cryptographic operations out of 69.13 % in total for the two functions, which amounts to 25.1 %. The cryptographic functions used in this configuration are AES-128-GCM for AEAD, SHA-256 for hashing, and the P256 curve for signature and Hybrid Public Key Encryption (HPKE) operation. After this analysis, we are left with the question: what are all the remaining CPU cycles used for?

To answer this, we have to look at the functions that are not doing the cryptographic operations. Many of these functions are called something related to "TLS marshal". Marshaling is the process of changing the memory representation of an object into a format more suitable for transmission or storage [24]. This is an important part of preparing the messages before and after protecting and unprotecting.

Figure 5.12 shows the CPU usage of the most resource-consuming functions having "marshal" in the function name. All functions are part of different call stacks, except those with 7.45 and 6.61 as usage. This means that we can add the contributions from these functions to get the total usage of marshal

Table 5.8: Summary of cryptographic operations for the protect and unprotect functions. "CPU total" is the cycles used compared to the total program cycles used. "% of function" is CPU consumption as a percentage of the protect or unprotect function.

Operation	CPU total		% of function	
Protect:				
Protect function total	27.79 %			
AEAD encryption	0.22 %	5.90 %	0.79 %	21.23 %
Key derivation	0.38 %		1.37 %	
Signature creation	5.30 %		19.07 %	
Unprotect:				
Unprotect function total	41.34 %			
AEAD decryption	0.21 %	11.43 %	0.51 %	27.65 %
Key derivation	0.38 %		0.92 %	
Signature verification	10.84 %		26.22 %	

and unmarshal functions. The functions with 7.45 and 6.61 are part of the same call stack as mls::unmarshal_ciphertext_content() at line 3. The marshal functions add up to a total of 46.2 %. This shows that Flamingo MLS uses a significant amount of the program resources for marshaling processes.

It is also interesting to know how different operations of MLS++ compare to each other regarding resource consumption. This is calculated by dividing the function's resource consumption by the number of times it was called. By doing this, we can make a relative comparison between the functions, see Table 5.9.

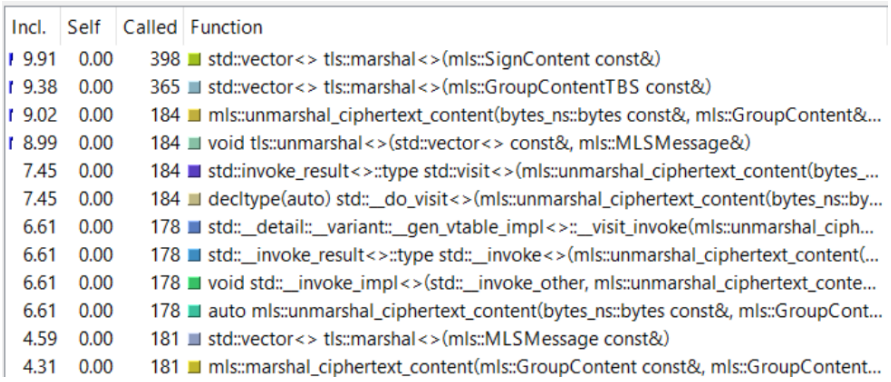


Figure 5.12: Marshal functions

Table 5.9: Comparison of CPU consumption for MLS++ functions

Function name	CPU usage
protect	0.16 %
unprotect	0.23 %
commit	0.98 %
handle	0.58 %
add	0.56 %

Table 5.10: Default parameter values for network testing

Parameter	Value
UPDATE_INTERVAL	400 ms
TOKEN_SEND_INTERVAL	50 ms
TOKEN_RETRANSMISSION_TIMEOUT	1000 ms
TOKEN_LOSS_TIMEOUT	2500 ms
JOIN_TIMEOUT	200 ms
CONSENSUS_TIMEOUT	500 ms

5.6 Test 6: Network Testing

It is essential to know how Flamingo MLS functions under different network scenarios. The drone communication is often high speed, with low latency, packet loss, and jitter. However, in some scenarios, these parameters can change, for instance, when the drones are at the edge of the radio coverage. Test 6 looks into how Flamingo MLS functions when these parameters are varied. Other parameters, such as bandwidth restrictions and duplicate packets, are also relevant but were not prioritized in this test.

We configure the network parameters using the Linux Traffic Control (TC) tool. This tool worked out of the box on the Ubuntu machine but not on the Jetson Nano or the Docker containers. The test setup was, therefore, altered to use multiple instances of Flamingo MLS running on the same machine and connecting them to the Jetson Nano. The TC tool only affects outgoing traffic, affecting the traffic from the instances on the Ubuntu machine, but not from the Jetson Nano. We used the following commands to create the network conditions:

```
tc qdisc add dev eth0 root netem delay #DELAY# #JITTER#
tc qdisc add dev eth0 root netem loss #LOSS#
```

Table 5.10 shows the default parameters set in this test. We tested static and dynamic group operations. In static group operations, no members left or tried to join, while for dynamic operations, one node left the group and joined again. We tested with the node rejoining immediately after leaving and rejoining after the group found out the member had left and removed it.

We used the same general method when testing for latency, jitter, and

Table 5.11: Results when varying network parameters

Parameter	Scenario	With Jetson node	Without Jetson node
Latency	Static	525 ms	
Latency	Dynamic	450 ms	
Jitter	Static	50 ms	1000 ms
Jitter	Dynamic	50 ms	900 ms
Packet loss	Static	3 %	6 %
Packet loss	Dynamic	4 %	8 %

packet loss in static group operations. The value started low and increased until it affected the program's operation. Such an effect could be that a node lost synchronization with the group. At this value, we used lower increments to find the threshold. The goal was to find a value that could sustain operations for five minutes. Members could leave as long as the group managed to recover. We did not always reach the five minutes but still recorded the value as an appropriate threshold.

We used the same procedure for dynamic group operations but made a group member leave the group and then rejoin. There is no leave function in Flamingo MLS, so this was done by terminating the program and then starting it again when the node should join. When a node stops forwarding the token, the other nodes remove it from the group after waiting for the token loss timeout period. It is, therefore, important to test what happens when the node joins before and after this timeout has expired. We test both of these scenarios with the values in the results.

Table 5.11 presents the results. Testing with the Jetson Nano node sometimes gave anomalous results. The results are, therefore, presented with and without this node for some of the tests.

The results for latency were as expected. When having 525 ms latency in the static test, one round of the token ring takes 2350 ms.[2] This is just below the token loss timeout value and avoids Totem reinitializing the group. Any higher value would initiate the token loss procedure, making it impossible to maintain a Totem group. The same principle applies to the dynamic test. The value is just below the consensus timeout value of 500 ms. A node must receive join messages from the remaining nodes before this time to reach a consensus. With a fixed latency close to this value, it will never receive a join message before reaching this timeout.

We did not expect that the response to jitter would be so different with and without the Jetson Nano node. Jitter has two effects on Totem: it increases latency by a random amount and, by doing this, also changes the order of messages. Ordering messages should not be a problem because that is Totem's specialty. We would expect results similar to those of the latency test for the latency contribution. Therefore, the results without the Jetson Nano are more

[2] 525 ms times four nodes and 50 ms token delay times five nodes

accurate. Why it failed with the Jetson Nano node is unclear. The network setup could be one explanation. We only applied jitter to outgoing traffic at the Ubuntu nodes, which could cause irregular behavior. Another reason could be that we created a unique network setup for this test. Flamingo MLS expects that either all nodes use different addresses on a LAN or all are on the same host. A combination of these is not part of the program; it is only created for this test. This is not well tested and might have affected the results somehow.

Flamingo MLS can sustain higher levels of jitter than latency. This is due to the random nature of jitter. A constantly high latency will affect Totem operations, while a sporadic high latency does not affect in the same way. If the latency becomes too high for one packet, it can be considered a lost packet, which can be dealt with as long as it does not happen too often.

We did not expect that the response to packet loss would be so different with and without the Jetson Nano node. The same potential reasons as for jitter might apply. It is not easy to say which is more realistic based on the values in this case. However, based on these and jitter testing results, there is likely something wrong with the Jetson Nano setup. Therefore, the results without the Jetson Nano node are likely more accurate in this case as well.

Totem can be adjusted to be more robust against network conditions by adjusting Totem's timeout parameters. We give some examples we tested in this paragraph. When increasing the token loss timeout value to 5000 ms, the program could sustain static operations with a 1000 ms latency compared to 525 ms before. When increasing the consensus timeout value to 1000 ms, the program could sustain a 900 ms latency compared to 450 ms before. For packet loss, we can adjust the retransmission and join timeouts to have more retransmissions when packets get lost. With a token retransmission timeout reduced to 300 ms, the program could sustain static operations with a 5 % packet loss with the Jetson Nano node, compared to 3 %. With a join timeout reduced to 50 ms, the program could sustain dynamic operations with a 5 % packet loss compared to 4 % before.

During the testing, we documented the program faults that happened. These often occurred when Flamingo MLS did many operations quickly, for example, with a network parameter at the limit and group operations happening. The faults are related to memory operations and are not easy to trace. We developed Flamingo MLS as a proof-of-concept application, so we expect errors, and it is not a big deal at this point. However, for a production-ready application, these issues must be fixed. We experienced the following error messages:

```
terminate called after throwing an instance of 'std::bad\_alloc'
Segmentation fault (core dumped)
free(): invalid pointer
double free or corruption (out)
```

Table 5.12: Optimal values for Flamingo MLS

Parameter	Value
UPDATE_INTERVAL	100 s
TOKEN_SEND_INTERVAL	500 ms
HEARTBEAT_INTERVAL	2000 ms
LOG_LEVEL	2
MAX_EPOCH_STATE_HISTORY_SIZE	100

5.7 Discussion of Results

The results show that the program parameters can significantly affect resource consumption. However, reducing consumption does have consequences in the form of decreased performance or security. Table 5.12 shows values that balance resource consumption against performance and security for the program parameters. These are the optimal values derived from the results. The update interval and token send interval should vary based on the group size, which is not possible in the version of Flamingo MLS used during simulated testing[3]. We base the values in Table 5.12 on a group size of ten nodes. In that scenario, every node would have to process an update every ten seconds and update its key material every one hundred seconds. This has a low resource consumption and leaves a window of one hundred seconds for using compromised key material. This will be sufficient for most drone operations. The token would complete one round every five seconds, which means updates to the group would take effect no later than after five seconds. This provides a low resource consumption, and a compromised node would be removed from the group five seconds after it is detected, which is an acceptable time frame.

It is clear from the results that sending and receiving application messages will be the most resource-intensive part of Flamingo MLS in practical use. We can adjust the program parameters for small resource consumption with minor security and performance drawbacks. Application messages, however, are an essential part of drone operations, and reducing the rate and size would significantly impact the operational performance. The minimum requirement of two packets of 1 kB per second is feasible. Figure 5.6 shows that 15 nodes sending 5 kB packets every second had a CPU consumption of about 33 %. This is similar to the performance for the minimum requirement and is an acceptably low resource consumption. The maximum requirement of 50 packets of 50 kB per second is not feasible. The maximal receiving rate achieved in Test 4 was 8192 bytes 90 times per second, with a CPU consumption of over 100 % of one core. This was from only one other node, using a quarter of the processor capacity. The rate would have to be drastically reduced when having multiple other nodes. The achievable data rate is closer to the minimum

[3]The final version of Flamingo MLS changes the *token send interval* into a *token round interval* that specifies the time interval for the token to make one round in the group.

requirement than the maximum since we need to leave enough resources for other applications.

Reception of messages is the limiting factor in this setup since the number of messages increases proportionally with the increase in group size. Unfortunately, the results show that receiving a message is more resource-consuming than sending a message, as shown in Test 4. Test 5 confirms this since the receiving thread used more CPU cycles than the sending thread.

The results show that MLS++ uses more resources than expected. Test 5 shows that MLS++ only uses about one-fifth of all operations for doing cryptographic operations. In addition, there are some cases where MLS++ consumes almost all resources available. One example is keeping all previous epoch states, which shuts down the program at around epoch 6000. Another example is the increase in resource consumption when using a low update interval in Test 2. This shows that MLS++ is not made for a high-performance environment, which is logical since MLS is supposed to be a group messaging protocol, not a high-performance communication protocol. A messaging application sends messages more infrequently, and it is, therefore, possible to rely on a less effective message-handling process. MLS++ is also not designed for the resource-constrained environment of a UAV.

Chapter 6

Flamingo MLS Takes Off

Only with UAVs in the air can we fully evaluate how MLS performs in a UAV swarm. To achieve this, we performed testing and troubleshooting of Flamingo MLS running on the Flamingo UAVs at FFI. As a result, the UAVs could take off and perform a flight using MLS to protect the communication. We believe this is the first time MLS has ever been airborne.

We experienced interesting behavior when performing this experiment. The testing revealed numerous challenges, and we implemented solutions to many. The experimentation took one week and was mostly in FFIs location at Kjeller. The week started with multiple minutes before the MLS group was reestablished, and at the end of the week, we were down to below half a minute. This chapter outlines the experiences from the process and describes measures taken to enable Flamingo MLS to take flight. We were limited to three Flamingo drones this week.

6.1 Ground Testing with Stable Network

The first challenge was to get Flamingo MLS working correctly on the ground with a stable network link. We used two Flamingo drones and one Ground Control Station (GCS). The Flamingo MLS program was pre-compiled using Docker and could be transferred to the drones and run. The GCS is running Ubuntu, and we needed to compile the program on the machine due to library incompatibilities. The instances could communicate with each other and establish an MLS group. They operated for one hour without any issues.

Valkyrie, the program controlling the drones, had to be configured to use Flamingo MLS as its communication link. We switched off the multicast functionality, enabled unicast, and set the unicast address to the local machine address using the same transport port number as Flamingo MLS for transmission and reception. This forces the traffic through Flamingo MLS, which then multicasts it to the network. This solution allowed telemetry data and commands to be shared between the drones and the GCS. The setup was verified by monitoring the telemetry data at the GCS and sending commands to

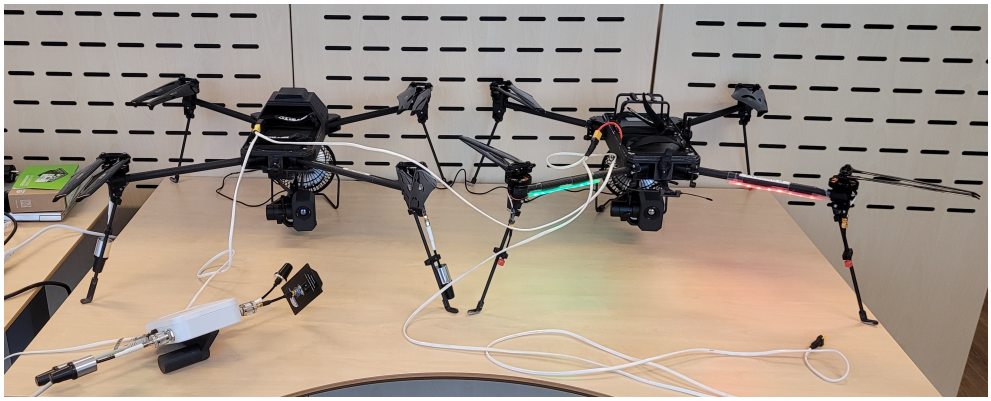


Figure 6.1: Some of the Flamingo drones used during testing

the drones. We did not verify the communication between the drones, but it should function the same way as between a drone and the GCS.

Video streams and other sensor streams use a different communication setup and are not easy to integrate with the current version of Flamingo MLS. Each drone streams to the GCS, transmitted using different port numbers from the telemetry data. The ports are hardcoded and used to determine from which drone the stream originated. Flamingo MLS currently only handles one data stream using one port number. We can solve this in the future by enabling multiple data streams. Every drone would need multiple streams, one for telemetry and one for each video or sensor stream, while the GCS would need to be able to receive all streams from the drones.

However, video streaming is an optional part of the operational procedure of the swarm. The drones usually operate with the video stream turned off to save radio bandwidth. If the drone makes an observation, pictures are sent through the same channel as the telemetry data. Therefore, we prioritized using MLS on telemetry data, but use on sensor streams is still relevant for future work.

The telemetry packets from the drones had a size of approximately 700 B. When still images were included, it reached about 1000 B per packet. These packets were sent ten times per second. After adding another Flamingo to the swarm, we measured the total data between 16 kB and 18 kB. Theoretically, it should be approximately 21 kB. The difference could be because of inaccurate sending frequency in Valkyrie or Flamingo MLS. The reason is not very important because the goal is to show how much data was sent and handled by Flamingo MLS. The measured values translate to between 128 kbps and 144 kbps.

With this setup, the CPU consumption was 10.5 % of one core. Since the Flamingo drone uses a six-core processor, the usage is 1.8 % of the total computational capacity. Flamingo MLS cannot use more than 25 % of the total CPU consumption before disrupting drone operations since the drones'

Table 6.1: Comparing CPU consumption on Jetson Nano and Jetson Xavier NX

Update interval	Heartbeat interval	Nano	Xavier NX
400 ms	2 s	24.0 %	8.9 %
4 s	20 s	16.6 %	2.7 %

autonomy and sensor processing software use approximately 75 %.

The CPU on the computer used with the current generation of Flamingo drones is not the same as during the testing in Chapter 5. We used Jetson Nano for the simulated testing, while the drones used Jetson Xavier NX. According to Nvidia, it provides up to 25 times the performance of the Jetson Nano[1]. It has a CPU with a higher clock rate and six cores, compared to four on the Jetson Nano. Therefore, we expect the Flamingo drones' performance to be better than what we observed during testing in Chapter 5. To verify this, we performed a test with the same parameters as the simulated testing, see Table 6.1. We used no application traffic and transmitted the token every 50 ms.

In this experiment, we see that Jetson Xavier NX is two to six times as efficient as Jetson Nano per core. This means that the limitations in performance measured in Chapter 5 do not necessarily apply to the Flamingo drones.

A ping test between the drones showed latency averaged at 6 ms for a round trip. When the radio link has poor quality, the latency could increase to 150 ms for sporadic packets. This affects Totem operations, which assume a low latency. The join timeout is 100 ms, meaning it expects to receive a join message within 100 ms of sending its join message. If the latency is 150 ms, no join messages will meet this limit, and Totem will send many unnecessary join messages. However, since higher latency only occurs when the radio link quality is poor, we can keep the 100 ms join timeout and accept sending more join messages when the link quality is poor.

6.2 Unreliable Network

An unreliable network connection can happen when a drone travels far away from the other drones, resulting in a weak radio link. It is crucial to investigate how Flamingo MLS responds to this scenario. To simulate this behavior while keeping the drones on the ground, we carried the drone to a place where we observed an effect on the network connection. We observed high packet loss, latency, and jitter. The location created significant attenuation so that the signal-to-noise ratio got sufficiently low. We used the same location for every

[1]The Nvidia website states that Jetson Xavier NX has more than ten times the performance of Jetson TX2 [8]. It also states that Jetson TX2 has up to 2.5 times the performance of Jetson Nano [25]. This is a general statement of the entire machine, not just the CPU and RAM.

Figure 6.2: Ground testing arrangement with three Flamingo drones and one GCS at the laptop to the right

test to recreate the same network conditions for each test. Even though we cannot recreate the exact network conditions, it is good enough to see how Flamingo MLS behaves during poor network conditions.

In the test setup, there was a problem with nodes not receiving handshake messages, even when the network connection was stable[2] and after multiple retransmissions. The reason is that the Rajant radio handles multicast and unicast packets differently. The nodes received Totem token transmissions, which use unicast packets. MLS handshake and application messages use multicast packets. One node with a weak radio link seemed to trigger the behavior, resulting in high packet loss for multicast between two nodes with a good radio link.

When the fault occurred, it created a situation where some nodes advanced to the next epoch without the other nodes. This situation creates false situational awareness for the operator of the GCS because it only allows for one-way communication. The node with the newest epoch can decrypt messages from older epochs but not vice versa. For instance, the GCS could receive telemetry messages but not send commands to the drones.

To mitigate this problem, we altered Flamingo MLS to use unicast for all messages. Instead of sending multicast messages, we sent unicast messages to the address of every individual node. This is a less efficient solution because we send the same packet multiple times, but since it is far more reliable, it is

[2]Determined using the ping command

worth the cost. We implemented this solution in the later stages of testing, so observations described later in this section might be affected by the multicast error.

One key insight discovered during testing is to make as few changes to the MLS group as possible when there is a node with poor network connectivity. Flamingo MLS can remove members in various situations, depending on the configuration, and when using all the features, we will quickly remove a member from the MLS group. When developing Flamingo MLS, this was considered an advantage: remove the node as quickly as possible so that the rest of the group can function correctly. However, we discovered arguments for the opposite case during testing. If we remove the member, it will also have to rejoin the group when the network connectivity improves. The feature for rejoining an MLS group is the most delicate feature of Flamingo MLS since it received the least attention. This can further disrupt the MLS group.

When removing a member from the MLS group, it will quickly try to rejoin the group if it is still receiving messages and is part of the Totem group. This happens despite the unstable network link since some messages will come through. Then, we are in a situation where the nodes try to reestablish the MLS group when the unstable network provides high latency, packet loss, and jitter. This is a receipt for unstable MLS groups due to a high probability of losing proposals or commits. We experienced that the situation could lead to members leaving the original stable group, either to join the node with an unstable connection or to create its own group. We could then end up in a situation with three different groups, which is difficult to merge under the best of circumstances. Flamingo MLS is not yet able to solve this efficiently.

Until the nodes resolve the situation, almost all communications break down, even though only one node has a poor network connection. This must not happen since a communication breakdown compromises the entire mission. A node having a poor network connection should only affect the communications to that node. Therefore, we focused on ensuring this does not happen during the testing. Avoiding making changes to the group during poor network connections is an essential factor in avoiding this.

What happens to application messages in this scenario? If the MLS group immediately removes the node, it cannot communicate with anyone. Poor network stability does not mean that we cannot send any packets. Therefore, many of the application messages will still come through. This is even more important when a node has a poor connection since it is far from the rest of the group and needs to communicate with them or the GCS to return. We need to configure the system to enable the passage of as many application messages as possible by minimizing alterations to the group. Ideally, the group makes no changes, and no advances in epoch occur. The ideal situation is if MLS does not detect the poor connection at all. This happened on some of the tests, and the only time the GCS did not receive the telemetry data was when the network connection was too poor to send anything through. We could not

distinguish this from normal operations by just monitoring MLS.

We made configuration and programming changes to Flamingo MLS to achieve this behavior. We turned off removing nodes that are no longer part of the Totem group. This feature removes nodes quickly when they experience a weak network connection, which is undesirable in this scenario. We also set a 60-second threshold for removing nodes from which we have not received messages. If not a single message gets through in this period, it is safe to assume it is not just an unstable network link that is the problem. It is more likely that the node is so far away that all radio communications have shut down or the node is lost altogether. It is then safe to remove it as it will not sporadically interfere with the MLS handshake process. The reasoning behind this strategy is that it is better to have one member too many rather than one member too few, at least for system performance. This benefits the system's availability, but we must balance this with confidentiality and integrity, which is more vulnerable the longer a compromised node stays in the group. However, the probability of exploitation within 60 seconds is minimal.

Increasing the update interval to 600 seconds decreases the likelihood of sending update messages during unstable network conditions. However, more is needed to ensure its prevention. To decrease the chance, we also introduced a procedure to check for unreliable network connections to avoid sending update messages at the wrong time. This procedure ensures that all nodes have sent a message within the last five seconds and that MLS and Totem agree on the group composition. The measures ensure that the group nodes are active and have a sufficiently good link to maintain the Totem group. In addition, the procedure checks if the MLS group state is corrupt. It determines this by checking the MLS handling error count, which counts the number of messages received that cannot be decrypted. It also counts if the node receives messages from an old epoch. When the group state is corrupt, it is not beneficial to continue advancing the group state. It will only make the recovery more complex.

During the testing, we focused on achieving the best possible performance, which often comes at the cost of security. Infrequent updates of key material will reduce the post-compromise security of the application by leaving a larger period for exploiting compromised key material. Features that prohibit the removal of members also reduce post-compromise security since the nodes might not be sending due to a compromise. The longer the node stays in the group, the longer the communication will be exposed to adversarial exploitation. This does not affect forward secrecy since the ratcheting procedure in the secret tree of MLS prevents the decryption of old messages.

As the use of MLS in UAV swarms is still in the early stages of development, we focus on making as few interruptions to normal communications flow as possible. However, it is critical to consider the consequences of performance optimization and find a balance between performance and security. An example is setting a cut-off for members not updating their key material. This is

essential to keep the system post-compromise secure, but we must balance this with prematurely removing members from the group, which we experienced multiple times.

Another challenge with the update interval being too high is that it is one of the only multicast messages Totem sends. The messages are one of the mechanisms for detecting nodes not part of the Totem group. Reception of a foreign message triggers a shift to the gather state, which starts the joining process for new members. During testing, the lack of these messages sometimes prevented members from joining the Totem group and the MLS session. A solution to this challenge is regularly introducing a "dummy" multicast packet, which functions as a foreign message for the node and allows it to join the group. This feature is implemented in Flamingo MLS and sends the packet every time it receives the Totem token.

6.3 Ready for Take-off

After a week of troubleshooting and testing at FFI, Flamingo MLS was ready for take-off. Having the Flamingo UAV swarm use Flamingo MLS while in the air is a substantial milestone in this project. The final version of the program is very stable during reliable network connections and has proven to handle unreliable network connections as well. Testing on the ground produced enough confidence in its operation to advance to drones in flight. An error in Flamingo MLS in flight could make the swarm operators lose control of the drone, making them resort to their fail-safe mechanisms. The confidence in its operation is therefore essential.

We performed the in-flight testing using two Flamingo UAVs at Rena on 16th October 2023. We believe this is the first time MLS has ever been used on airborne drones. The flight was successful, with the drones sending reports to the GCS and the GCS sending commands to the drones. The drones were airborne for approximately ten minutes. Figure 6.3 shows the view of the GCS. We can see the path flown and the observations made. The swarm operator tasked the drones to different locations during the flight.

The communication was flawless for the first five minutes of the flight. After this, the swarm operator experienced having to send commands to the drones multiple times before being accepted. The packets were lost somewhere between the GCS software and the drone software. Packet loss on the radio interface is possible but unlikely because of the short distances. The drones were less than 550 meters from the GCS at the maximum, well within the expected radio coverage. It is also possible that a fault in Flamingo MLS caused the packets to get lost. The log files from the flight do not show any signs of this. MLS on the drone was able to decrypt all messages it received.

Figure 6.3: View on GCS during testing in flight

Chapter 7

Discussion

After many hours of research, development, and testing, we have produced numerous results and experiences. We start this chapter by discussing the results and connecting them to the research questions. Then, we justify how the implementation of Totem does not affect the security of MLS. Lastly, we share thoughts on the neighboring challenge of addressing compromised UAVs in the swarm.

7.1 Use of MLS in a UAV Swarm

Our experience is that there are many challenges when implementing MLS on UAVs. We experienced the downsides of using MLS since it was developed for a different use case in chat and messaging services. These services have lower demand for throughput and latency, and expect more available resources than a UAV swarm can provide. When sending messages infrequently, it does not matter how resource-demanding it is to protect the message. However, when sending messages multiple times a second, the resource demand becomes a significant issue. This likely affects the implementations of MLS as well and is presumably why we see high resource consumption in Cisco's MLS++ library. The developers did not design it for a resource-constrained environment on a UAV.

The application messages in MLS++ consumed more resources than expected and are currently the limiting factor for how well MLS++ performs in a UAV swarm. We assumed that producing and processing handshake messages would be most resource-consuming, but this was wrong. While significantly affecting resource consumption, we can easily adjust handshake messages using longer intervals between update messages. However, we cannot do that for application messages without significantly reducing the UAVs' ability to communicate. The bottleneck is the number of messages we send and not the number of members as initially hypothesized. With the MLS++ implementation, we lose much of the gain in efficiency by using most of the resources on marshaling instead of cryptographic operations.

We now recount and answer the research question based on the results obtained in this project.

Q1: How can we implement MLS services in a UAV swarm?

We demonstrated how to use a decentralized Delivery Service (DS) to ensure agreement of MLS handshake messages in a UAV swarm. The Totem protocol achieves this, and the results indicate it creates a reliable environment for MLS to agree on the order of handshake messages. Delivery of key packages is also a part of the DS, which we implemented by asking the node for its package. We did not implement the Authentication Service (AS) but examined how to use the service conceptually. We recommend an AS at the GCS for the most straightforward configuration. This creates a centralized service in the network but is unproblematic since the only in-flight communication is to check the revocation list. The service going down is disadvantageous but of little consequence.

Q2: What parameters of MLS give sufficient security compared to the resource consumption on the UAVs?

In Chapters 5 and 6, we investigated how different parameters affect the performance of MLS and discussed how the choice of parameters affected the system's security. Precise security measurements are impossible, so we can only do an educated estimate. We identify the following parameters as the most essential for the security of the swarm:

- Cryptographic functions
- Update interval
- Removal of MLS members

Leon and Britt [3] show the performance of the available cryptographic functions, and we must compare this to the cryptographic protection. We have not performed the comparison in this project. The simulated testing concluded that the optimal update interval is every 100 seconds, which has low resource consumption and allows for only a short period to exploit the system. Removing members is important for keeping the system post-compromise secure, but removing members too quickly disrupts the system's availability. Removal is most critical for a compromised member, but we have not developed a way of detecting this. Removing members who are not updating their key material is also important. This prevents the system from being exploited by compromised key material, keeping it post-compromise secure. However, we have yet to investigate the optimal threshold for when the removal should occur.

Q3: How does MLS affect the performance of the UAV swarm?

When working correctly, MLS did not affect the performance of the UAV swarm during testing. However, we used only a few drones and did not protect

the video, making resource consumption less realistic. The simulated testing results indicate that with more drones and the protection of more data, resource consumption will increase significantly. With limited available computational resources on the drones, this can limit swarm operations. We identified inefficiencies in Cisco's MLS++ library as the cause and recommended testing other libraries. OpenMLS is an obvious choice for further research. We require more research and testing to determine if MLS will affect the performance of a UAV swarm. More testing on the Jetson Xavier NX should also be performed since most of the testing in this project was performed on the weaker Jetson Nano.

> *Main research question: Can we implement MLS to achieve secure and efficient communication in a military UAV swarm?*

This project shows that reliably implementing MLS is possible, but further research is needed to decide whether it can provide efficient communication in a large UAV swarm. The implementation lacks the AS, an essential security service. We must show we can implement the AS before declaring our solution secure, but based on the conceptual work in this thesis, it should be achievable.

Since the current version of MLS is not adapted for the distributed nature of a UAV swarm, we should consider developing a version of MLS better suited for this use case. Both MLS and Totem create and maintain separate groups. Instead of trying to keep both groups synchronized, we can implement the mechanisms of Totem in a distributed MLS. Totem is up to date most of the time, while the MLS group is not agile enough to keep up, at least in Flamingo MLS. A tighter integration between MLS and Totem can decrease the conceptual complexity of the system and make it more reliable and straightforward to implement.

We have investigated how to use MLS to enable secure communication in a UAV swarm. However, we can generalize most of the results to other unmanned systems. Some challenges discovered for a UAV swarm will probably be less challenging with other unmanned systems. The Flamingo UAV is a small drone with constrained computational resources. Larger systems can afford more computational resources, so the computational load needed for Flamingo MLS might not be an issue anymore. There are also more stationary systems with more predictable connections, which would require a less dynamic protocol than Totem.

7.2 Composed Security of MLS and Totem

When combining two different protocols, it is essential to consider how it affects the security. MLS by itself is considered secure, but the specification [2] recommends carrying the MLS messages over a secure transport channel, such

as TLS. The argument is to protect the metadata from being read by an attacker and avoid this information being used to, for instance, perform a selective denial of service attack on specific messages. The developers did not design Totem with the security perspective in mind, and we cannot consider it a secure transport channel. Therefore, we cannot consider all metadata secure in Flamingo MLS. This does not affect the confidentiality and integrity of MLS message content, but it can affect the availability of messages. An attack on the availability in Totem will also affect the availability of MLS messages.

In Totem, there are simple methods for impeding the availability. Totem does not authenticate its messages, which means there is no way of knowing a message's legitimacy and who sent it. To give an example of an attack on the availability in Totem, we assume an attacker with access to the network infrastructure. In the case of a UAV swarm, an attacker can communicate with the Rajant radios as a part of the IP network. With access to the network, the attacker can send malicious Totem messages that affect the operation of Totem.

A simple example is to send join messages continuously. When a Totem node receives a join message, it enters the gather state to reconfigure the group, adding the new member. The nodes will have to agree on the new group, and as long as the attacker does not agree with the rest, they will never reach a consensus. They will then continue until reaching the consensus timeout, and then the ID of the attacker will be marked as a failed node. The rest will then be able to continue to form a group. However, the attacker can continuously alter its ID or spoof the ID of the other nodes. This will make the gather state continue forever, effectively stopping all communication that rely on Totem. We cannot add new members to the MLS group, remove members, or update key material. Existing members can still send application messages but lose most of the security guaranteed by MLS.

We can protect the Totem protocol from tampering by authenticating the messages. An attacker cannot continuously send join messages because the other nodes will detect that the messages are not authenticated and discard them. We can use MLS functionality to perform the authentication, thereby integrating the protocols more tightly, as described earlier in this chapter. Since Totem members are not necessarily a part of the MLS group, the functionality cannot depend on group membership. Every node has an MLS key package, which is independent of the MLS group and contains the node's signature key. We can use the key to sign and authenticate Totem messages. This solution does require an AS implementation, which Flamingo MLS currently does not have.

7.3 Addressing Compromised UAVs

So far, we focused on achieving security from outside attackers, but what happens when the attack comes from inside the group? This can happen when an

adversary gets physical control over a UAV or by manipulating the radio interface. The UAV swarm is then vulnerable to tampering by the compromised UAV. It could disrupt the swarm behavior or leak the information gathered. This scenario might be realistic when operating close to an enemy and can impact the swarm operation.

The problem statement is closely related to implementing secure communication in a UAV swarm. Secure communication can help prevent compromised UAVs and recover from a compromised state. Finding solutions to this problem is beyond this project's scope, but we will present some thoughts on the matter. We will discuss why MLS alone cannot recover from this type of compromise, how to detect a compromised drone, who should decide to kick out a node, and in what scenarios the problem is most relevant.

Since MLS is post-compromise secure, we would think that it is simple to recover from such a compromise. However, it is not that simple in practice. When we say that MLS has post-compromise security, we emphasize two main features: key material is updated regularly, and members can be removed from the group. An update of key material will only help recover when the adversary compromises the keys and nothing more. The group will enter a secure state when the node updates the compromised keys. This does not help when the entire node is compromised because the adversary can access the new key material. We must remove the member to recover from this scenario. This does not happen automatically; someone or something must make the decision.

It is difficult to detect that a UAV is compromised. It could behave exactly like the other UAVs and only diverge in critical moments. Alternatively, it could send minor misinformation to the swarm that, over time, makes the mission fail. If the goal of the compromise is to affect the operation of the swarm, then some changes in normal behavior have to occur. These could be minor and difficult to detect but exist nonetheless. It might be possible to analyze and compare the behavior to normal UAV behavior.

Another solution is to detect the moment of compromise. This is effective for physical compromises since the drone will deviate from normal behavior. The drone might have to land, which is easy to detect by using the drone's sensors or because of loss of network connection. These scenarios can sometimes be legitimate, so we should be careful with classifying a node as compromised automatically. Loss of network connectivity can quickly happen if the UAVs have too large of a distance or fly behind an obstacle.

Who can classify a drone as compromised and remove it from the group? It is problematic if all nodes can make this decision. We can imagine a scenario where a compromised drone misuses this feature to disrupt the group by removing legitimate members. One solution is to have a leadership role that possesses the power to remove members. This could be a trusted UAV or the GCS, but it introduces a single point of failure in the system. Another solution is to use a mechanism for having all the nodes agree on the decision. For instance, we remove the drone if the majority agree. All nodes must agree on

the result for MLS operations. If not, one node might remove the compromised node while another lets it rejoin. Therefore, one node cannot just decide independently. There needs to be a consensus on the matter.

Defending against malicious members is complex, and solutions are advanced and expensive. It might cost more than the operational effect on the other end can justify. The battery capacity severely limits the operational time for the Flamingo UAV swarm. Even though there is a high risk of losing UAVs during the operations, it is unlikely that the enemy has the capacity and resources to capture, reprogram, and relaunch the UAV to disturb the operation. However, we can imagine an adversary with a longer planning horizon than a single flight. An adversary can exploit the compromised UAV to affect multiple swarm operations over an extended period.

There are operations with unmanned systems that have longer operational time than the Flamingo UAVs. Larger UAVs have considerably longer flight times, and the same goes for Unmanned Surface Vehicles (USVs), where the operation can last for days, weeks, or months. The longer the operation continues, the higher the risk of compromise, and the adversary can influence the operation for a longer time.

Chapter 8

Conclusion

In this project, we showed that MLS can be used during UAV swarm operations by successfully demonstrating its use on two Flamingo UAVs in flight. This achievement marks a milestone in this line of research, affirming that we can implement MLS to provide the necessary levels of reliability and performance. This was made possible through the integration of MLS with Totem, a protocol designed to ensure the reliable consensus of nodes on the sequencing of messages. This solution provided minimal overhead and was reliable against latency, jitter, and packet loss during testing.

The solution relies on Cisco's MLS++ library for the MLS implementation. Testing revealed a surge in CPU consumption when the rate of application messages increased. Benchmark testing on the smaller Jetson Nano board showed a max data rate of 5.9 Mbps for the reception of messages, using the entire capacity of a CPU core for this operation. This is more resource consumption than expected and can affect the feasibility of larger swarms and more data. Investigations into CPU usage showed that cryptographic operations were only a small part of the contribution, with less than 20 % of cycles used in a test configuration. Marshaling operations used nearly half the cycles. This is probably because MLS++ is designed for use in messaging services and not the resource-restrained environment of UAV swarms.

In this project, we only presented conceptual solutions for an Authentication Service (AS), so the solution is vulnerable to tampering by illegitimate nodes. Therefore, we did not achieve secure communications from all threats. The performance is an issue for application messages, and even though it worked with the three drones available during testing, it will not be efficient for larger swarms. To answer the main research question, we showed how to implement secure communication in a UAV swarm and explained what remains before fully achieving it. We revealed essential drawbacks to how MLS++ implements MLS that we can further use to develop efficient solutions.

8.1 Future Work

This project focused on developing a proof-of-concept program to demonstrate that MLS can be successfully deployed in a UAV swarm. Even though this was a success, more work is needed to develop this into a complete solution. The current solution is not secure because the certificates are not verified. That means anyone can create an MLS key package and join the group. Therefore, the need for an AS is essential to maintain security. We described conceptual solutions to this challenge in Section 3.1, relying on PKI and having the GCS function as a CA. We must implement this concept and test it to see how it works.

The performance of MLS++ is not optimal for use on UAVs with constrained resources. We need to investigate ways of implementing MLS with better performance. We should test other implementations to evaluate the performance. OpenMLS is a prime candidate. If no implementations provide the desired results, we can alter existing implementations or develop one designed for high-throughput and resource-constrained environments. We also recommend further testing on the Jetson Xavier NX to provide a realistic picture of the resource consumption.

Totem performed well in the testing, but the protocol description could have been more precise on the desired behavior, and therefore, the implementation could be more optimal. We should perform further testing and overall protocol design, and also investigate other protocols. Attention should be given to combining MLS and Totem (or a similar protocol). Both protocols maintain a group, and we use much effort to synchronize them. This generates unnecessary conceptual complexity. A tighter integration can also address challenges with joining the MLS group, removing members, and merging groups.

Acronyms

AEAD Authenticated Encryption with Associated Data. 58, 59, 60

AES Advanced Encryption Standard. 14

ARSENL Advanced Robotic Systems Engineering Laboratory. 18, 19

AS Authentication Service. 3, 4, 10, 14, 18, 19, 20, 22, 23, 40, 41, 76, 77, 78, 81, 82

BML Battle Management Language. 9

CA Certificate Authority. 21, 22, 23, 24, 25, 82

CPU Central Processing Unit. 8, 47, 48, 49, 52, 54, 55, 56, 57, 58, 59, 60, 61, 62, 65, 66, 68, 69, 81

DoS Denial of Service. 15

DS Delivery Service. iii, 4, 10, 12, 15, 18, 19, 20, 25, 26, 76

DSA Digital Signature Algorithm. 15

ELF Executable and Linkable Format. 57

FFI Norwegian Defence Research Establishment. iii, v, 1, 8, 67, 73

FIFO First In, First Out. 17

GCS Ground Control Station. 3, 7, 8, 9, 18, 22, 23, 24, 25, 67, 68, 70, 71, 73, 76, 79, 82

HMAC Hash-based Message Authentication Code. 58

HPKE Hybrid Public Key Encryption. 60

IETF Internet Engineering Task Force. 21

IPSec Internet Protocol Security. 18

ISR Intelligence, Surveillance, and Reconnaissance. 1, 7, 8

JREAP Joint Range Extension Applications Protocol. 18

LAN Local Area Network. 48, 64

MAC Message Authentication Code. 27

MGEN Multi-Generator. 49

MLS Messaging Layer Security. iii, v, 1, 2, 3, 4, 7, 9, 10, 11, 12, 14, 15, 17, 18, 19, 20, 21, 22, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 36, 37, 39, 40, 41, 42, 43, 44, 47, 49, 50, 58, 59, 66, 67, 68, 70, 71, 72, 73, 75, 76, 77, 78, 79, 80, 81, 82

NPS Naval Postgraduate School. 1, 2, 4, 17, 40

OpenPGP Open Pretty Good Privacy. 18

PKI Public Key Infrastructure. 21, 22, 23, 82

RAM Random-Access Memory. 8, 47, 49, 52, 53, 54, 69

SHA Secure Hash Algorithm. 15, 60

TC Linux Traffic Control. 62

TCP Transport Control Protocol. 18

TLS Transport Layer Security. 15, 18, 58, 78

TPM Token-Passing Multicast. 26, 27

UAS Unmanned Aerial System. 18

UAV Unmanned Aerial Vehicle. iii, 1, 2, 3, 4, 5, 7, 8, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 29, 30, 31, 50, 66, 67, 72, 73, 75, 76, 77, 78, 79, 80, 81, 82

UDP User Datagram Protocol. 26, 32, 33, 41, 43, 49, 50

USV Unmanned Surface Vehicle. 18, 80

Bibliography

[1] T. H. Chung, M. R. Clement, M. A. Day, K. D. Jones, D. Davis and M. Jones, 'Live-fly, large-scale field experimentation for large numbers of fixed-wing UAVs,' in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1255–1262. doi: 10.1109/ICRA.2016.7487257.

[2] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara and K. Cohn-Gordon, *The Messaging Layer Security (MLS) Protocol*, RFC 9420, Jul. 2023. doi: 10.17487/RFC9420. [Online]. Available: https://www.rfc-editor.org/info/rfc9420.

[3] A. Leon and C. J. Britt, 'UXS authentication and key exchange requirements for multidomain operation and joint interoperability,' M.S. thesis, Naval Postgraduate School, Jun. 2022. [Online]. Available: http://hdl.handle.net/10945/70738.

[4] E. Dietz, 'Utilizing the messaging layer security protocol in a lossy communications aerial swarm,' M.S. thesis, Naval Postgraduate School, Mar. 2022. [Online]. Available: http://hdl.handle.net/10945/69631.

[5] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican and K. Milner, 'On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees,' in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 1802–1819, isbn: 9781450356930. doi: 10.1145/3243734.3243747. [Online]. Available: https://doi.org/10.1145/3243734.3243747.

[6] S. Engebråten, K. Glette and O. Yakimenko, 'Field-testing of high-level decentralized controllers for a multi-function drone swarm,' in *2018 IEEE 14th International Conference on Control and Automation (ICCA)*, 2018, pp. 379–386. doi: 10.1109/ICCA.2018.8444354.

[7] O. R. Nummedal, 'Flamingo - a UAV for autonomy research,' *Norwegian Defence Research Establishment*, 2021. [Online]. Available: http://hdl.handle.net/20.500.12242/2839.

[8] NVIDIA Corporation, *Jetson Xavier NX Series*, [Online; accessed 14-October-2023], 2023. [Online]. Available: https://www.nvidia.com/en-sg/autonomous-machines/embedded-systems/jetson-xavier-nx/.

[9] B. Beurdouche, E. Rescorla, E. Omara, S. Inguva and A. Duric, 'The Messaging Layer Security (MLS) Architecture,' Internet Engineering Task Force, Internet-Draft draft-ietf-mls-architecture-11, Jul. 2023, Work in Progress, 47 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-mls-architecture/11/.

[10] M. Singhal and N. G. Shivaratri, *Advanced concepts in operating systems*. McGraw-Hill, Inc., 1994.

[11] X. Défago, A. Schiper and P. Urbán, 'Total order broadcast and multicast algorithms: Taxonomy and survey,' *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, Dec. 2004, issn: 0360-0300. doi: 10.1145/1041680.1041682. [Online]. Available: https://doi.org/10.1145/1041680.1041682.

[12] F. CRISTIAN, 'Asynchronous atomic broadcast,' *IBM Tech. Discl. Bull.*, vol. 33, no. 9, pp. 115–116, 1991. [Online]. Available: https://cir.nii.ac.jp/crid/1571417125087635712.

[13] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal and P. Ciarfella, 'The Totem single-ring ordering and membership protocol,' *ACM Trans. Comput. Syst.*, vol. 13, no. 4, pp. 311–342, Nov. 1995, issn: 0734-2071. doi: 10.1145/210223.210224. [Online]. Available: https://doi.org/10.1145/210223.210224.

[14] B. Rajagopalan and P. McKinley, 'A token-based protocol for reliable, ordered multicast communication,' in *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, 1989, pp. 84–93. doi: 10.1109/RELDIS.1989.72752.

[15] T. Abdelzaher, A. Shaikh, F. Jahanian and K. Shin, 'RTCAST: Lightweight multicast for real-time process groups,' in *Proceedings Real-Time Technology and Applications*, 1996, pp. 250–259. doi: 10.1109/RTTAS.1996.509542.

[16] Cisco. 'MLS++.' (2023), [Online]. Available: https://github.com/cisco/mlspp (visited on 12/12/2023).

[17] Phoenix R&D and Cryspen. 'OpenMLS.' (2023), [Online]. Available: https://github.com/openmls/openmls (visited on 12/12/2023).

[18] matrix-org. 'MLS-TS.' (2023), [Online]. Available: https://gitlab.matrix.org/matrix-org/mls-ts (visited on 12/12/2023).

[19] Cisco. 'go-mls.' (2023), [Online]. Available: https://github.com/cisco/go-mls (visited on 12/12/2023).

[20] Docker Inc., *Docker*, [Online; accessed 10-September-2023], 2023. [Online]. Available: `https://www.docker.com/`.

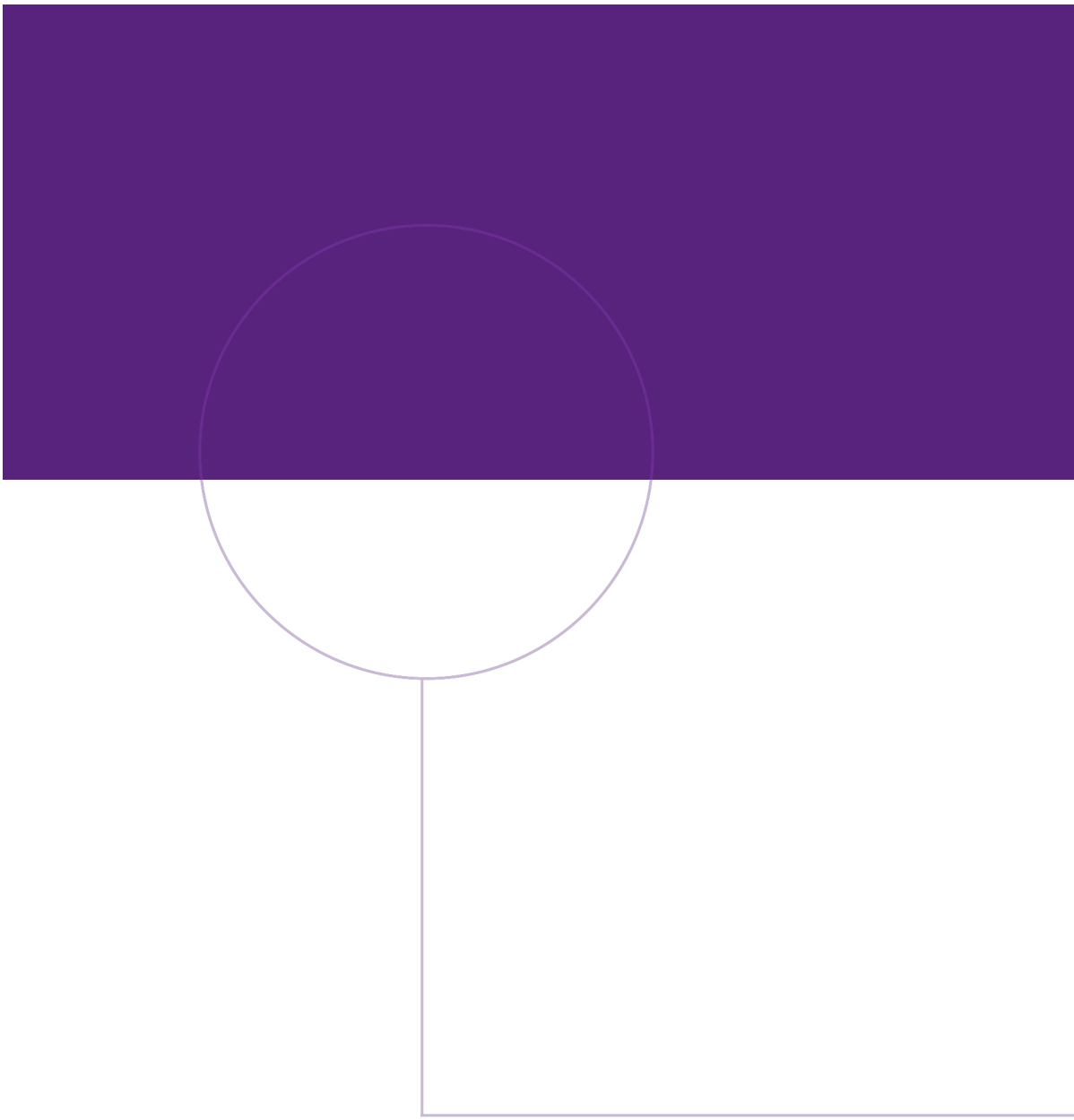
[21] man7.org. 'Top manual page.' (2023), [Online]. Available: `https://man7.org/linux/man-pages/man1/top.1.html` (visited on 01/09/2023).

[22] United States Naval Research Laboratory, *Multi-Generator (MGEN) Network Test Tool*, [Online; accessed 10-September-2023], 2023. [Online]. Available: `https://www.nrl.navy.mil/Our-Work/Areas-of-Research/Information-Technology/NCS/MGEN/`.

[23] J. Weidendorfer. 'KCachegrind.' (2013), [Online]. Available: `https://kcachegrind.github.io/html/Home.html` (visited on 09/09/2023).

[24] Wikipedia contributors, *Marshalling (computer science) — Wikipedia, The Free Encyclopedia*, [Online; accessed 9-September-2023], 2023. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Marshalling_(computer_science)&oldid=1146228628`.

[25] NVIDIA Corporation, *Jetson TX2 NX Module*, [Online; accessed 14-October-2023], 2023. [Online]. Available: `https://developer.nvidia.com/embedded/jetson-tx2-nx`.



NTNU

Norwegian University of
Science and Technology