Master's thesis

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

Noah Krogh Anderson

Implementing Distributed MLS for UAV Swarms

Master's thesis in Digital Infrastructure and Cyber Security (MSTCNNS)
Supervisor: Tjerand Silde
Co-supervisor: Martin Strand

June 2026

NTNU
Norwegian University of Science and Technology

Noah Krogh Anderson

Implementing Distributed MLS for UAV Swarms

Master's thesis in Digital Infrastructure and Cyber Security (MSTCNNS)
Supervisor: Tjerand Silde
Co-supervisor: Martin Strand
June 2026

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

NTNU
Norwegian University of
Science and Technology

Problem description

This project explores secure communication in military Unmanned Aerial Vehicle (UAV) swarms, a domain characterised by unstable networks, frequent membership changes, and the risk of node compromise. One promising solution is the Messaging Layer Security (MLS) protocol, which offers scalable group key agreement with strong security guarantees.

Prior work has found MLS suitable for unmanned systems, but the requirements imposed on the Delivery Service (DS) have proved difficult to meet in distributed settings. This project implements Distributed MLS (DiMLS), which eliminates the need for a complex distributed DS, and couples it with a lightweight reliability mechanism for handshake message delivery. Testing is conducted both on the Flamingo UAVs developed at Norwegian Defence Research Establishment (FFI) and in a containerised virtual environment.

The contributions of this project include a working implementation of DiMLS, empirical insights into how DiMLS performs relative to standard MLS in terms of resilience, efficiency, and security in the UAV swarm context, and a demonstration that the implementation integrates with the Flamingo platform to encrypt both telemetry and video streams. Finally, we propose optimisations to reduce the bandwidth and CPU overhead of DiMLS, making it more viable for operational deployment.

Approved: 2026-02-25 – Associate Professor Tjerand Silde, NTNU (Main supervisor)

Abstract

Securing communications in military Unmanned Aerial Vehicle (UAV) swarms requires adapting to challenges such as unstable network conditions, frequent membership changes, and the risk of node compromise. The Messaging Layer Security (MLS) protocol offers a promising foundation for group key agreement in this context, but prior work has shown that its requirement for a distributed Delivery Service (DS) with total handshake message ordering is difficult to satisfy in decentralised swarm environments.

This thesis implements and evaluates Distributed Messaging Layer Security (DiMLS), a draft protocol that eliminates the need for a complex distributed DS by introducing the concept of `SendGroups`, where each member owns and controls a single MLS group. The implementation is built on the `openmls` library and coupled with a lightweight retransmission mechanism to handle handshake message loss over the unreliable mesh radio network of the Flamingo UAV platform, developed at the Norwegian Defence Research Establishment (FFI).

Testing was conducted both physically on Flamingo UAVs and in a containerised virtual environment. Results demonstrate that DiMLS coupled with a lightweight retransmission mechanism maintains stable cryptographic groups under degraded network conditions, achieving near complete message decryption at packet loss rates of up to 50%. This represents a substantial improvement over previously evaluated Messaging Layer Security (MLS) based UAV swarm architectures, where MLS with Totem becomes non operational at loss rates above 10%. Physical testing further confirmed successful integration with the Valkyrie swarm software, with end-to-end encryption of both telemetry and video streams verified on the target hardware.

The performance testing showed that CPU consumption scales approximately linearly with group size, and bandwidth requirements for group operations grow quadratically with the swarm size, which is a trade-off of the `SendGroup` architecture. To mitigate some of these costs, we propose an initial path update during group formation to reduce the commit overhead from quadratic to approximately linear scaling. We also suggest running DiMLS as a key exchange module rather than a full message encryption layer to significantly reduce CPU consumption. With these optimisations, DiMLS represents a viable, standards based approach to securing communications in military UAV swarms.

Sammendrag

Sikring av kommunikasjon i militære UAV-svermer må tilpasses til utfordringer som ustabile nettverksforhold, hyppige medlemsendringer og risiko for kompromitterte noder. Messaging Layer Security (MLS) virker som en lovende protokoll for sikring av kommunikasjon i denne konteksten, men tidligere arbeid har vist at kravet om en distribuert Delivery Service (DS), som sikrer at meldinger blir levert i riktig rekkefølge, er vanskelig å oppfylle i desentraliserte sverm-miljøer.

Denne oppgaven implementerer og evaluerer Distributed Messaging Layer Security (DiMLS), et forslag til en protokoll som fjerner behovet for en kompleks distribuert DS ved å innføre konseptet `SendGroups`, der hver node i svermen eier og kontrollerer en MLS-gruppe hver. Implementasjonen er bygget ved hjelp av `openmls`-biblioteket og kombineres med en simpel retransmisjonsmekanisme for å håndtere tap av handshake-meldinger over det upålitelige mesh-radionettverket til Flamingo UAV-plattformen, som er utviklet ved Forsvarets Forskningsinstitutt (FFI).

Tester ble gjennomført både fysisk på Flamingo UAV-er og i et containerbasert virtuelt miljø. Resultatene viser at DiMLS med retransmisjonsmekanismen oppnår stabile kryptografiske grupper, med tilnærmet fullstendig meldingsdekryptering ved pakketap på opptil 50%, noe som er en betydelig forbedring sammenlignet med gjeldende løsninger som bruker av MLS i UAV-svermer. Fysisk testing bekreftet vellykket integrering med Valkyrie-sverm-programvaren, ved å ende-til-ende-kryptere både telemetri og videostrømming.

Ytelsestesting viser at CPU-forbruk skalerer omtrent lineært med antall medlemmer, og at båndbreddekravet for gruppeoperasjoner vokser kvadratisk med størrelsen på svermen, noe som er en konsekvens av `SendGroup`-konseptet. For å redusere noe av disse kostnadene foreslår vi en initiell path update under oppsettet av svermen for å redusere overhead fra kvadratisk til omtrent lineær skalering. Vi foreslår også å kjøre DiMLS som en nøkkelutvekslingsmodul, framfor et fullstendig meldingskrypteringslag, for å redusere CPU-forbruket. Med disse optimaliseringene representerer DiMLS en levedyktig, standardbasert tilnærming til sikring av kommunikasjon i militære UAV-svermer.

Acknowledgements

I would like to thank my supervisors Tjerand Silde and Martin Strand for their wise inputs and invaluable guidance during this project.

Additionally, I would like to thank my family. Special thanks to my son Linus, who got me up in the early mornings, and to my newborn daughter Hennie, whose due date was a constant source of motivation. Finally, I am deeply grateful to my partner Dina, whose support made it possible for me to complete this project.

Contents

List of Figures

List of Acronyms

API Application Programming Interface.

ARM Advanced RISC Machine.

ART Asynchronous Ratcheting Tree.

AS Authentication Service.

CGKA Continuous Group Key Agreement.

CLI Command Line Interface.

COTS Commercial of-the-shelf.

CPU Central Processing Unit.

DH Diffie-Hellman.

DiMLS Distributed Messaging Layer Security.

DS Delivery Service.

FFI Norwegian Defence Research Establishment.

FS Forward Secrecy.

GCS Ground Control Station.

HAL Hybrid Autonomy Layer.

IETF Internet Engineering Task Force.

KDF key derivation function.

KEM Key Encapsulation Mechanism.

MLS Messaging Layer Security.

MPSC Multiple Producer Single Consumer.

PCS Post-Compromise Security.

PKI Public Key Infrastructure.

PoC Proof of Concept.

PSK Pre Shared Key.

RAM Random Access Memory.

RFC Request For Comments.

SDG Sustainable Development Goals.

UAV Unmanned Aerial Vehicle.

UN United Nations.

Chapter 1
Introduction

Autonomous Unmanned Aerial Vehicle (UAV) swarms are an emerging capability in modern warfare, offering the potential to execute complex missions without risking human lives or requiring a dedicated operator for each unit. However realizing this potential depends on secure and reliable communication between swarm members. This is a non-trivial challenge in contested, resource constrained environments in which military UAVs operate. This thesis investigates Distributed Messaging Layer Security (DiMLS) as a solution to this challenge, exploring its implementation, performance, and viability in the context of a military UAV swarm.

In this chapter we first motivate the need for secure swarm communications and define the scope of the project. The research questions are then presented, followed by a survey of related work and a summary of the contributions made by this thesis. Finally we discuss the relevance of this project against the United Nations (UN) Sustainable Development Goals (SDG).

1.1 Motivation

Over the past few years, UAVs have proliferated across many sectors, enabling operators to quickly and cost-effectively carry out tasks such as infrastructure inspection, aerial photography, land surveys, and product delivery[1]. This is equally true in the military domain, where small and relatively inexpensive UAVs have changed the battlefield by providing cheap and effective ways to perform reconnaissance, target acquisition and precision target engagement [Kun23]. The increase in UAV use presents both dangers and opportunities. One such opportunity is the development of UAV swarms that can be assigned a mission and execute it autonomously, without risking human lives or requiring a dedicated operator for each UAV. This vision is increasingly reflected in Norwegian defence priorities. The Norwegian Defence Research Establishment (FFI) considers autonomy as one of their key areas of

[1] https://www.uasnorway.no/hva-brukes-droner-til-i-dag-og-hva-bringer-fremtiden/

research [FFI25] and The Norwegian Army has allocated 1.5 billion NOK over a 10-year period for UAV research and procurement [BV25]. The importance of UAV in the military domain is further underscored by a dedicated government strategy document for UAVs within the defence sector [For25], which mentions autonomy and swarm-technology explicitly.

However, realizing the full potential of autonomous UAVs depends critically on secure communications. A swarm operates in a highly dynamic, decentralized environment where nodes are constantly moving, and continuously forming and dissolving links, making conventional security protocols both impractical and computationally expensive. The consequences of a compromised swarm are severe[2]. An adversary who injects false commands could hijack individual drones, corrupt collective decision making, or collapse an entire mission through denial-of-service attacks. Because swarm behaviour emerges from inter-UAV communication, a single point of infiltration can propagate malicious influence across the whole swarm. Simultaneously, the resource constraints of UAV platforms, including limited battery life, processing power and bandwidth limit the cryptographic overhead that can be tolerated. Added to this is the contested nature of the radio spectrum, with threats of jamming, spoofing, and eavesdropping. Considering this it becomes clear that securing UAV swarm communications is both an operational necessity, and an interesting technical challenge that demands solutions capable of handling the unique challenges of swarm environments.

1.2 Scope

This project focuses on the implementation, testing and evaluation of Distributed Messaging Layer Security (DiMLS) [XLH25] in a UAV swarm setting. We build on the experiences and findings of previous efforts [Mar23; BA25] which explored MLS in the UAV setting. We extend the previous work by exploring DiMLS which reduces the need for a complex, distributed Delivery Service (DS), while preserving the security properties of regular MLS [BBR+23].

The work is exploratory in nature, with a primary goal of developing a Proof of Concept (PoC) of DiMLS and evaluating its performance across relevant parameters, including throughput, CPU consumption, and bandwidth requirements. Where applicable, results are compared directly against those of the preceding projects [Mar23; BA25]. We do not focus on the implementation of an Authentication Service (AS) or consider other protocols than MLS and DiMLS.

The Valkyrie system and Flamingo UAV [HBSS24], developed at Norwegian Defence Research Establishment (FFI), serve as the baseline platform for this project. The im-

[2] https://www.ffi.no/aktuelt/nyheter/hva-hvis-fienden-kaprer-dronen

plementation is designed with integration into this system as an explicit goal. Testing is conducted under its expected operational conditions, such as limited computational resources, degraded network connectivity, and decentralized coordination.

1.3 Research questions

This project aims to implement, experiment with, and evaluate DiMLS in a UAV swarm context. To achieve this, we first investigate how DiMLS can be implemented for use in a UAV swarm. We then evaluate its performance against previous efforts using MLS with the Totem single ring ordering protocol [BA25]. Finally, we examine how DiMLS scales with swarm size, to assess its viability as a communication protocol in this setting. With this as the background, we define the following research questions[3]:

RQ1 How can DiMLS be implemented to provide secure and resilient communication in UAV swarms?

RQ2 How does the use of DiMLS perform in terms of security, efficiency, and fault tolerance compared to MLS and Totem in UAV swarm scenarios?

RQ3 How does DiMLS scale as the network grows, regarding bandwidth- and CPU-usage?

RQ1 addresses the implementation of DiMLS in a UAV swarm context, considering the architectural design choices required to integrate it with the swarm system and the constraints imposed by limited computational resources and degraded network conditions. RQ2 evaluates the performance of DiMLS relative to previous efforts using MLS and Totem, focusing on parameters such as throughput, CPU consumption, and bandwidth usage, as well as its behaviour under degraded network conditions. RQ3 examines whether DiMLS remains a viable option as the number of nodes in the swarm increases. We do this by assessing how resource consumption grows with network size, identifying the bottlenecks and limitations of DiMLS.

[3]The research questions were refined during the course of the project to better reflect the findings and challenges encountered. Originally, RQ1 included scalability as an explicit concern. This was separated into its own research question (RQ3), reflecting the importance of scaling as an independent dimension of DiMLS viability in swarm contexts. Additionally, the project aimed to evaluate reliable transport protocols such as QUIC [IS21] for the delivery of handshake messages, but this was reconsidered when a lightweight retransmission mechanism proved sufficient to meet the reliability requirements. This made a broader protocol evaluation unnecessary, and we removed the corresponding research question.

1.4 Related work

Following the standardization of MLS [BBR+23], several efforts have been made to implement the protocol in UAV swarm settings. Leon and Britt [LB22] conducted a study evaluating security protocols for unmanned systems, finding MLS to be the most suitable option in terms of group security and efficiency, and implemented a PoC on two different unmanned platforms. Dietz, Davis and Hale [DDH23] implemented MLS on a UAV platform, finding the protocol promising but identifying the need for a reliable Delivery Service (DS). Marstrander [Mar23] addressed this by implementing and testing the Totem protocol [AMM+95] as a decentralized Delivery Service (DS), and Bragstad and Arder [BA25] subsequently improved on this work by evaluating alternative implementations of both MLS and Totem. Their findings show that MLS is a viable option for the swarm context, but the total ordering requirement and the implementation of the distributed DS remain a challenge. A recently proposed draft RFC [XLH25] introduces Distributed Messaging Layer Security (DiMLS), which aims to eliminate the need for a complex, distributed DS through a concept called `SendGroups`. We use this RFC draft as the blueprint for our implementation in this project, while building on the experiences and findings of Marstrander, and Bragstad and Arder.

1.5 Our contributions

This project presents an implementation and evaluation of DiMLS for secure communication in UAV swarms. We demonstrate that DiMLS, coupled with a lightweight retransmission mechanism, is significantly more resilient to packet loss compared to previous approaches based on MLS and Totem. We show this by recreating the network robustness testing done by Bragstad and Arder [BA25]. Their system, Valkyrie MLS[4], struggles to decrypt packages consistently as the packet loss approaches 10%, while our system decrypts almost 100% of the arriving packets, even as the packet loss rate approaches 50%.

Beyond resilience testing, we provide empirical insights into how DiMLS scales with swarm size, characterising its bandwidth and CPU consumption as the number of nodes grows. Our findings show that CPU consumption scales linearly with the packet rate, while the bandwidth requirements of DiMLS are quadratic for different operations, compared to logarithmic for regular MLS. Despite the increase in bandwidth, we argue that DiMLS is a step forward concerning secure swarm communications, compared to MLS with Totem. We propose how the bandwidth and CPU consumption can be optimized to make DiMLS a more viable choice, and discuss why the theoretical performance of MLS does not translate into practice, when combined with Totem.

[4] https://github.com/mkarder/valkyrie-mls

Together, these contributions advance the current state-of-the art regarding MLS usage in decentralized swarm environments, as well as providing insights into central properties of the DiMLS protocol.

1.6 Sustainability and ethics

This thesis concerns the communication security of autonomous military UAV swarms, and we believe that the ethics and sustainability of the topic should be discussed. We do this by first discussing how the project relates to the United Nations (UN) Sustainable Development Goals (SDG)s, and then by discussing the ethical considerations of autonomous UAV swarms in the context of this work.

1.6.1 Sustainability and the UN Sustainable Development Goals

Relating this topic to the UN Sustainable Development Goals (SDG)s may initially appear counterintuitive. SDG 16.1 calls for a significant reduction in all forms of violence and related death rates worldwide, and autonomous military UAV swarms may be viewed as enabling violence rather than reducing it. We do not claim this thesis directly advances the SDGs. However, two contributions relate meaningfully to specific targets. First, improved accountability in autonomous military systems can be connected to SDG 16.1, and secondly, secure communications in distributed, resource-constrained settings can be related to SDG 9.

SDG 16 relates to peace, justice and strong institutions, and our project connects specifically to sub-goal 16.6, which concerns the development of effective, accountable and transparent institutions. UAV swarms operating without robust cryptographic security create accountability gaps that can undermine lawful command authority. A compromised swarm cannot be reliably controlled, and by implementing DiMLS with verifiable group membership, this work ensures that each epoch secret is cryptographically bound to a specific authenticated membership list. This supports the development of autonomous military systems that are both technically secure and institutionally accountable, reducing the risk of such systems operating outside lawful command and control.

SDG 9 concerns industry, innovation and infrastructure, which calls for resilient infrastructure and the promotion of inclusive and sustainable innovation. The implementation of DiMLS is built on an Internet Engineering Task Force (IETF) draft and contributes to the broader field of secure distributed communication by demonstrating that the protocol is viable on resource-constrained embedded hardware. While the immediate context is military, the underlying technical contributions are not domain-specific. Secure group key agreement for dynamic, decentralised networks of autonomous nodes is a general problem with applications beyond defence,

including disaster response drones coordinating search and rescue operations, and autonomous systems performing infrastructure inspection in environments where reliable connectivity cannot be assumed. By advancing the implementation maturity of an open standard on constrained hardware, this work lowers the barrier for adoption in civilian contexts, contributing to a more resilient and secure communication infrastructure.

1.6.2 Ethics

The autonomy properties of the system must be addressed from an ethical standpoint. The UN has stated that fully autonomous lethal systems are both politically unacceptable and morally reprehensible [Uni25]. It is therefore important to clarify the nature of the autonomy involved in the Valkyrie system, which is the target deployment context of this work.

The Valkyrie swarm is not a fully autonomous lethal system. There is always a human operator in the loop, retaining decision authority over mission critical actions. The swarm does not independently select or engage targets, and the scope of autonomy is bounded to the swarm behaviour. Examples of this are the fine-grained navigation, cooperation between the UAVs and keeping the camera on an object.

A relevant ethical consideration in favour of UAV swarm technology is its potential to reduce the exposure of human soldiers to direct danger. By substituting unmanned systems for personnel in high-risk reconnaissance and surveillance operations, UAV swarms can reduce the number of soldiers required in hazardous environments.

The contribution of this thesis is confined to the communication security layer. DiMLS is a cryptographic protocol that governs how group members authenticate and exchange keys. It does not influence targeting, navigation, or mission planning. By strengthening the integrity and authenticity of command and control communications, this work supports human oversight rather than undermining it. A cryptographically secure swarm is harder to compromise, spoof, or manipulate, meaning the human operator retains more reliable control over the system.

In this light, the ethical concern is not that this work increases the autonomy of the system, but rather that it is applied in a military context. This is a legitimate concern, but the dual-use nature of the underlying technology means that the same protocol could be applied in civilian contexts where no such concerns arise. We consider the application defensible given that the system preserves human command authority, reduces the risk to human soldiers, and that the security properties implemented are oriented toward accountability and control rather than toward enabling offensive capability.

Chapter 2
Background

This chapter introduces the background knowledge necessary to contextualise and motivate the work presented in this thesis. We begin by describing UAV swarms and their operational characteristics, establishing the environment in which DiMLS is to be deployed. We then review the concept of Continuous Group Key Agreement (CGKA), before examining the key mechanisms of MLS and its decentralized variant, DiMLS. Finally, we describe the tools used in this project.

2.1 Unmanned Aerial Vehicles

An Unmanned Aerial Vehicle (UAV) is an aircraft that operates without a human pilot on board, controlled either remotely by an operator or autonomously via onboard computers. While the term drone is often used interchangeably with UAV, it is more general and encompasses unmanned vehicles across all domains, including ground, sea, and air[1].

UAVs are widely used across many sectors, with applications ranging from search and rescue and aerial photography to package delivery and remote infrastructure inspection. This project is concerned specifically with military UAVs, which are employed for tasks such as intelligence gathering, targeting, and kinetic operations when equipped with munitions. Military UAVs can be categorised by size, range, airframe type, or application. According to Halsør et al. [HBSS24], they broadly fall into three categories: surveillance, combat, and loitering munitions. Surveillance drones carry sensors and cameras but no weapons. Combat drones are generally larger and carry some form of armament. Loitering munitions, sometimes referred to as suicide drones, are smaller platforms designed to seek and strike a target directly, destroying themselves in the process.

[1] https://www.japcc.org/chapters/c-uas-the-differences-between-unmanned-aircraft-drones-cruise-missiles-and-hypersonic-vehicles/

Although UAVs have featured in modern warfare for many years, their use has historically been dominated by large, specialised surveillance and combat platforms. This has changed considerably in recent years. According to Kunertova [Kun23], such platforms are increasingly vulnerable to electronic countermeasures and air defence systems, particularly in contested airspace where neither side holds air superiority. Smaller drones have consequently demonstrated greater survivability, and have reshaped battlefield dynamics by shortening artillery targeting cycles and providing real-time situational awareness [Kun23]. Their commercial availability and relatively low cost make them straightforward to acquire and easy to replace, further lowering the barrier to their widespread battlefield use.

2.1.1 UAV swarms

Zieliński [Zie21] defines a drone swarm as a "group of multiple unmanned aerial platforms and/or weapons systems deployed to achieve a common goal". In a military context, this common goal may range from intelligence gathering and reconnaissance of an area to direct target engagement of adversaries. By operating collectively, UAVs can cover larger areas, gather more information and complete more complex tasks than any individual unit could achieve alone. Swarms are also inherently fault-tolerant, as the loss of individual UAVs does not necessarily compromise the mission. Dietz [Die22] identifies several features that a well configured UAV swarm should exhibit:

Survivability The swarm remains operational even when individual UAVs are lost, shot down, or otherwise removed, as no single unit is critical to mission success.

Scalability The range and complexity of missions grows with the size of the swarm, allowing it to adapt to a wider variety of operational demands.

Speed Tasks can be decomposed and executed in parallel across multiple UAVs, reducing mission completion time.

Autonomy Individual UAVs must operate autonomously, making independent decisions without relying on continuous human oversight or centralised control.

Cost By leveraging economies of scale, swarms can execute missions more cheaply and efficiently than single-UAV systems.

Central to realizing these properties is how the swarm is commanded and controlled. The internal command and control structure determines how decisions are made and coordinated across the swarm, and varies depending on the use case and the level of autonomy in the swarm. Scharre [Sch14] proposes the following command and control architectures:

Centralized A single central entity, such as a Ground Control Station (GCS), acts as the sole point of coordination, with all UAVs communicating directly with it.

Hierarchical UAVs are arranged in a layered structure where some units hold higher authority than others, with lower ranked UAVs reporting to those above them.

Consensus A fully decentralized structure in which UAVs communicate on a peer-to-peer basis, using voting mechanisms to reach a shared decision.

Emergent Coordination arises from individual UAVs reacting to the behaviour of their neighbours, mimicking natural swarms such as flocks of birds, with no explicit coordination mechanism required.

The choice of architecture depends on the capabilities and goals of the system, and is directly related to the level of autonomy in the swarm [Zie21]. A less autonomous swarm relies on centralised coordination, while a highly autonomous swarm tends toward an emergent architecture, where individual UAVs react to inputs from neighbouring units, much like animals in a natural swarm.

While the command and control architecture defines how the swarm communicates and coordinates, it also has implications for information security in the swarm. A centralised structure requires only pairwise secure connections between the coordinating node and the remaining units, whereas a consensus-based architecture requires all nodes to be able to communicate securely with one another, posing a greater security challenge.

Additionally, there are some important properties that must be accounted for in an autonomous UAV swarm. First, the swarm is dynamic, meaning UAVs may join or leave at any time. This requires the secure communication protocol to support efficient addition and removal of members from the cryptographic group. Second, there is a risk of adversaries capturing or compromising individual UAVs, through depleted batteries, air defence or electronic jamming. This requires that past and future communications remain secure even if an active unit is compromised, properties known as Forward Secrecy (FS) and Post-Compromise Security (PCS). Third, network conditions in swarm environments are expected to be unstable and constrained. Not all UAVs will receive every message, and some may be temporarily unreachable, necessitating an asynchronous and fault-tolerant communication solution.

2.1.2 The Valkyrie swarm and the Flamingo UAV

This project is conducted in collaboration with Norwegian Defence Research Establishment (FFI), which has developed the Valkyrie autonomy system and the Flamingo

UAV. Together, these form the target platform for the implementation presented in this thesis, and their characteristics directly affect the design choices and constraints in our project. The following sections introduce the Flamingo UAV and the Valkyrie system, displayed in Figure 2.1, establishing the operational context in which DiMLS is developed and evaluated.

The autonomy software running on the Flamingo platform is named Valkyrie. Valkyrie implements the necessary components required for coordinating multiple autonomous UAVs. This includes protocols for UAV communication, a multi-UAV Ground Control Station (GCS) and the Hybrid Autonomy Layer (HAL) [Num21]. HAL is the autonomy decision module, also developed at FFI, which provides unmanned vehicles with the capability to perform advanced autonomous tasks. The Valkyrie system allows an operator to control several drones at the same time which reduces the need for human resources.

Flamingo [Num21] is a quad-copter developed for research in cooperative autonomous systems. Its weight varies by configuration, ranging from 2.2 to 2.8 kg, with flight times of 30 to 45 minutes. It is built primarily from Commercial of-the-shelf (COTS) components, meaning the platform is designed with modularity and ease of production in mind, making it inexpensive to manufacture. As the platform is continuously evolving, some specifications reported by Nummedal [Num21] may no longer reflect the current configuration.

The onboard computer is an Nvidia Jetson Xavier NX, a compact single-board computer designed for edge computing and embedded applications. It provides 8 GB of RAM and a six-core Nvidia Carmel 64-bit CPU with an ARM architecture [NVI26], running Linux for Tegra which is an Ubuntu-based distribution optimised for Jetson hardware. Marstrander [Mar23] mentions that the autonomy and sensor software uses approximately 75% of the CPU processing power, leaving a maximum of 25% for our cryptographic module before disrupting UAV operations.

For wireless communication, the Flamingo is equipped with a 5.8 GHz Rajant mesh radio, used to transmit application data and stream video to the GCS. Each UAV can act as a radio relay, extending the effective transmission range across the swarm. Communication between the UAVs relies on UDP multicast packets, while video streaming to the GCS uses UDP unicast. As noted by Bragstad and Arder [BA25], the system is designed to tolerate degraded link quality, with packet loss of approximately 30% considered acceptable for telemetry and command-and-control traffic, meaning the cryptographic protocol must be resilient against packet loss.

Figure 2.1: Flamingo UAV alongside the Ground Control Station (GCS)

2.2 Continuous Group Key Agreement Protocols

A key challenge in securing UAV swarm communications is how to efficiently establish and maintain a shared secret across all nodes. One cryptographic primitive designed to address exactly this is Continuous Group Key Agreement (CGKA). CGKA protocols enable members of a group to continuously agree on a shared secret as membership changes. They support addition and removal of group members regardless of participants being online or offline, and do not rely on a trusted group manager [ACJM20].

To better understand the principles of CGKA, we first describe Asynchronous Ratcheting Tree (ART), the first CGKA protocol to efficiently provide PCS. We then

describe TreeKEM, the CGKA protocol underlying MLS, before examining the MLS protocol itself in depth.

2.2.1 Asynchronous Ratcheting Trees

Asynchronous Ratcheting Tree (ART) was the first demonstration of how PCS could be achieved in realistic asynchronous group messaging systems [CCG+18]. Using a tree-based Diffie-Hellman key exchange, ART derives a shared group key without requiring the participants to be online simultaneously. This was a significant contribution, as it introduced a CGKA protocol that operates fully asynchronously, while efficiently providing PCS.

ART provides group key agreement through a tree-based structure in which each group member is represented as a leaf node of a binary tree. Every node in the tree holds a Diffie-Hellman (DH) key pair and to derive the root key, each member must know the secret keys of all nodes along its direct path to the root, as well as the public keys of the corresponding copath nodes. The copath is the path of siblings of each node along the direct path. Figure 2.2 shows the direct path as the purple nodes and the copath as the yellow nodes for leaf node `A`. The root key is then computed iteratively up the tree, with each internal node's key derived from the DH exchange between its two children. The tree structure enables efficient key distribution with $O(\log_2(N))$ complexity per update, where N is the number of group members. In comparison pairwise schemes require $O(N)$ operations. Considering the group in Figure 2.2, member `A` only has to encrypt and distribute three secrets, for the whole group to be able to derive the root key.

To initialize a group, each member first publishes a `prekey` to an untrusted server. The `prekeys` are ephemeral DH public keys that a client can fetch on demand, enabling asynchronous key agreement without requiring all parties to be online simultaneously. The group initiator generates a `setup_key` used just for the initial session establishment. By combining the `setup_key` with the fetched `prekeys` of each intended member, the initiator can derive secret leaf keys for all other members. The initiator uses these private keys to derive public keys for all the internal nodes of the tree. We describe the full group initialization in detailed steps below.

1. Alice generates a setup key pair, with public component `SUK` and private component `suk`.

2. Alice requests a long-term public identity key (`IK_i`) and an ephemeral prekey (`EK_i`) for each of Bob, Charlie, etc. from a key server.

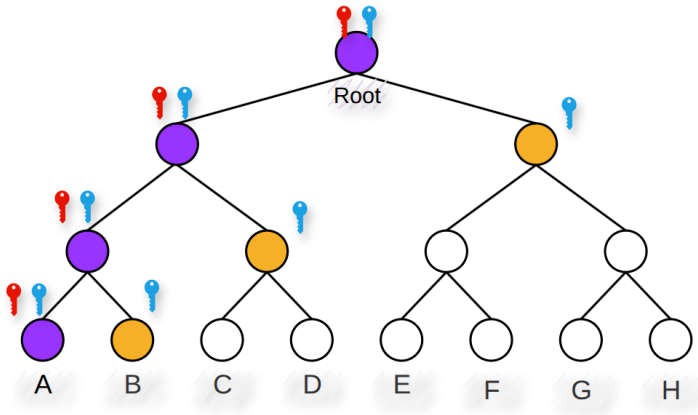


Figure 2.2: Illustrating the ART binary tree, with members A to H. Node A knows all private keys (marked in red) in the direct path from her node to the root node, and the public key (marked in blue) in both the direct path and the copath. The direct path is shown in purple and the copath in yellow.

3. For each peer, Alice performs one round of an authenticated key exchange protocol using her private long-term identity key `ik_a` and `suk` to derive a secret leaf key for that peer.

4. Alice generates a leaf key pair for herself and constructs the Diffie-Hellman tree bottom-up, using the derived leaf secrets to compute the private and public keys of all internal nodes. The secret derived at the root node serves as the initial group secret. At this point Alice holds all private keys in the tree, and is trusted to delete them. The update mechanism in ART removes this trust assumption by allowing each member to rotate their own leaf key to a secret only they know.

5. To initialise the group, Alice distributes the ephemeral prekeys `EK_i` and public identity keys `IK_i` she used, along with the public setup key `SUK`, the tree of public keys, and a signature over all shared material, signed with her private identity key `ik_a`.

6. Upon receiving these materials, Bob and Charlie verify the signature and extract the copath of public keys from the tree. Each member then derives their private leaf key using their private long-term key `ik_i`, Alice's long-term public key `IK_A`, the private component of their prekey `ek_i`, and the public setup key `SUK`. By iteratively performing Diffie-Hellman operations with the derived private keys in the direct path and the received public keys along the copath, up to the root, they recover the same initial group secret that Alice derived.

To achieve PCS in ART, a member can update their leaf key pair and recompute the private and public keys of all nodes along their direct path to the root. The new public keys are then broadcast to the group. Upon receiving this update, each member replaces the affected nodes in their copath with the new public keys, and recomputes the root secret by traversing their own path to the root. Since the updated leaf secret is known only to the updating member, any adversary who previously compromised the old key material can no longer derive the current group secret.

2.2.2 TreeKEM

TreeKEM is the group key agreement protocol underlying MLS. It builds on the same binary tree structure as ART, but replaces DH key exchanges with Key Encapsulation Mechanism (KEM) operations [BBR18]. Rather than deriving a shared secret through an exchange between two parties, a Key Encapsulation Mechanism (KEM) works by having the sender generate a random secret, encapsulate it under the recipient's public key, and transmit the ciphertext. The recipient then decapsulates it using their corresponding private key to recover the secret.

Unlike ART, TreeKEM uses a key derivation function (KDF) to derive node secrets from the leaf to the root, rather than relying on DH key exchanges. The KDF is often called a ratchet, and the tree constructed by TreeKEM is called a ratchet tree. Combining KDFs with KEM yields two important benefits over ART. First, TreeKEM is more efficient on the recipient side. Rather than performing a DH exchange at every internal node along their direct path, the recipient need only decapsulate the updated secret once and hash it up to the root. Second, because each node secret is derived from a generated random value from a leaf node, rather than computed as a function of both children, the parent secret is independent of the non-updated child. This makes TreeKEM more robust to concurrent updates, where multiple members update simultaneously, as conflicting updates do not invalidate each other's path computations [BBR18].

Having established the core mechanisms that enable MLS to provide efficient, asynchronous group key agreement with strong security guarantees, we now turn to the protocol itself.

2.3 Messaging Layer Security

Messaging Layer Security (MLS) is a cryptographic protocol designed to address the challenges of secure group messaging solutions, such as scalability and asynchronicity. Standardised by the IETF in Request For Comments (RFC) 9420 [BBR+23], MLS provides end-to-end encrypted group communication with strong security guarantees

including FS and PCS, while remaining efficient for large and dynamic groups. The core objective of MLS is to provide continuous authenticated key exchange for groups of arbitrary size [BBR+23]. Members agree on a common secret key while mutually verifying each other's identity, supporting groups ranging from two to several thousand participants. In this chapter we will give an overview of the core concepts, message types and the infrastructure services needed to operate MLS.

2.3.1 Groups and epochs

Two central organising concepts in MLS are groups and epochs. A group is the collection of members sharing a secret key at any given time. The members are represented by the `Leafnode` object, which stores their identity, credentials and capabilities [BRO+25].

An epoch describes the current state of the group. A sequence of epochs thus describes the evolution of the group over time. Within each epoch, authenticated members agree on a common epoch secret known only to the current group members, from which keys for message encryption can be derived. Members themselves decide when to advance the shared cryptographic state from one epoch to the next. This is achieved when the members perform group operations.

2.3.2 Group operations and message types

Group operations in MLS are operations that make changes to the group state, advancing the epoch counter. The most relevant operations are `Add`, `Remove`, `Update`, and `PreSharedKey`. The `Add` operation adds a member using a `KeyPackage`, `Remove` removes a member, and `Update` provides PCS by generating a new epoch secret and updating the key pairs along the direct path of the given `LeafNode`, resulting in a new root secret. The `PreSharedKey` proposal requests a Pre Shared Key (PSK) to be injected into the key schedule of the next epoch, providing a mechanism to introduce out-of-band entropy into the epoch secret.

To perform a group operation, a member must first create a `Proposal`, which describes the intended change to the group. Members can either send one or more proposals separately and follow them with a `Commit`, or embed the proposals directly in a `Commit` message [BRO+25]. A `Commit` transitions the group from one epoch to the next. To add a member, a `KeyPackage` must be obtained, containing the joining member's credentials, public key, and capabilities. After processing the `KeyPackage`, the initiator distributes a `Welcome` to the new member, providing the information necessary to join the group, and a `Commit` to the existing members, informing them of the addition.

MLS defines two wire formats that wrap the previously mentioned message types, `PublicMessage` and `PrivateMessage`. `PublicMessage` carries messages that are authenticated but not encrypted, and is typically used for handshake messages in scenarios where the Delivery Service (DS) must inspect or reorder them. `PrivateMessage` carries messages that are both encrypted and authenticated, and is used for application messages and for handshake messages where confidentiality from the DS is required. A handshake message refers to a `Proposal` or `Commit` carried in either wire format, while an application message refers to a message carrying encrypted and authenticated payload data.

Figure 2.3 shows a flow diagram illustrating the `Add` operation. First, the members publish `Keypackages` to the DS. Alice then adds Bob by requesting his `Keypackage`, which results in a `Welcome` which is distributed to Bob. For Charlie the process is the same, however the `Add` also results in a `Commit` which must be distributed to Bob so his group state is in the same epoch as Alice and Charlie.

2.3.3 The DS and AS

MLS requires two infrastructure services to function correctly and securely: The Delivery Service (DS) and the Authentication Service (AS). Both are described in the protocol specification [BBR+23], but are not implemented as part of MLS itself. The protocol description does not say how these services should be implemented, only what is required from them

The DS is responsible for routing messages to group members. While MLS permits application messages to arrive out of order, the DS must ensure that all `Proposal` messages arrive before the `Commit` that references them, and that all members agree on which `Commit` ends each epoch. If two members issue a `Commit` simultaneously without proper handling by the DS, members may apply different commits and diverge in their view of the group state. This is known as *group forking*. The DS is also responsible for storing `KeyPackages` on behalf of clients, which are used to form groups asynchronously.

The Authentication Service (AS) issues and verifies credentials that bind a member's identity to their signature key pair [BRO+25], ensuring that group members can authenticate one another. The protocol does not mandate a specific implementation. The AS can be realised as a traditional centralised Public Key Infrastructure (PKI), or through manual verification of fingerprints, as employed by Signal [Mar16].

2.3.4 Decomposition of MLS

Wallez et al. [WPB25] showed that MLS can be decomposed into three sub-protocols: TreeSync, TreeKEM and TreeDEM, with each being responsible for different parts of

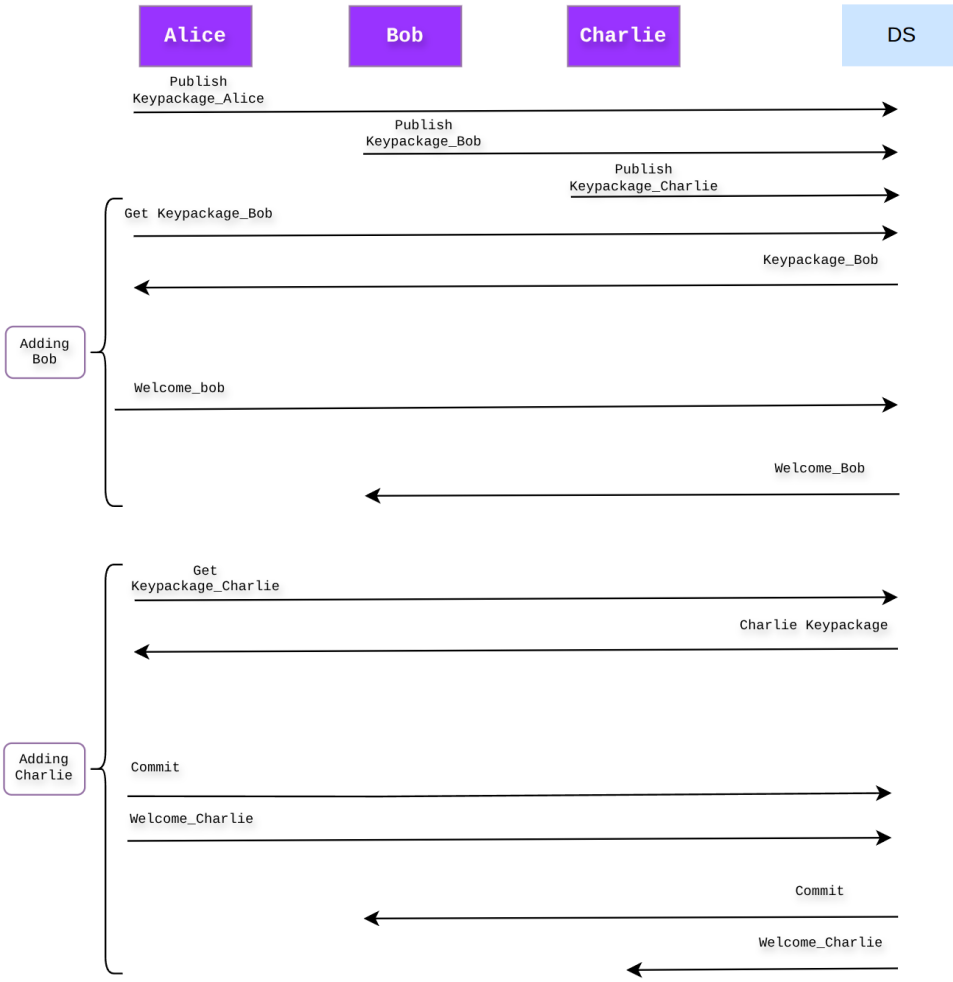


Figure 2.3: MLS flow diagram for adding members to a group.

MLS functionality.

TreeSync is responsible for synchronising the shared group state across all members, handling membership changes and ensuring a consistent view of the group [WPBB23]. The group is structured as a ratchet tree, where each occupied leaf node represents a member and each internal node represents a subgroup. The initiator creates the group by first establishing a one-member group containing only itself, generating a random secret that serves as the initial epoch secret. To add members, the initiator fetches a `KeyPackage` for each intended member and allocates one leaf node per member where the corresponding credentials and public keys are stored. When a `Proposal`, `Commit` or `Welcome` is received, TreeSync updates, changes or builds the

ratchet tree. The authenticated, synchronised tree is used by TreeKEM to derive an epoch secret.

TreeKEM uses the authenticated tree produced by TreeSync to derive the epoch secret [WPB25]. To create a new epoch secret, a member generates a fresh random secret at its own leaf node and hashes it up its direct path, storing the derived value for each internal node until the root is reached. For each internal node along this path, an asymmetric key pair is derived and the node secret is encrypted under the public key of the corresponding copath node. The resulting public keys and encrypted path secrets are broadcast to the group. Each member uses their private copath key to decrypt the relevant path secret, and then applies a key derivation function (KDF) iteratively to derive the remaining secrets along the path, ultimately allowing all members to independently compute the same epoch secret. TreeKEM thereby provides MLS with continuous group key agreement and PCS, as each epoch transition produces a fresh epoch secret. A compromised member can be healed by performing an update that replaces their leaf key with material unknown to the adversary. An important detail concerning the efficiency of the ratchet tree is that internal nodes whose subtree leaf nodes have not yet performed any updates to the group remain blank. This means they do not initially hold a key pair, with the consequence being a linear cost of updating keys in the worst case. In contrast, if all internal nodes have a key pair stored, the cost of updating keys is logarithmic.

After the epoch secret has been produced, TreeDEM uses it to derive the message encryption keys used to encrypt and decrypt application messages [WPB25]. After each message, the encryption keys are ratcheted forward using a KDF, ensuring that compromise of a current key does not expose past messages and thereby providing FS.

2.4 MLS in distributed systems

One of the requirements for the DS is to ensure global total ordering of handshake messages. In traditional scenarios, such as an instant messaging app, the DS can be implemented as a centralised server that sorts messages and makes sure that all `Commits` arrive at the members in the correct order.

In distributed environments however, the requirements for total ordering have proved to be a non-trivial problem. Tassopoulus [Tas25] compared and analysed five different delivery protocols for distributed systems and found the Totem single ring ordering protocol [AMM+95] to be most suitable. Efforts to implement MLS with Totem as the distributed DS show that Totem ensures total order on handshake messages, but scales moderately [Tas25] and blocks communications when a node is at the edge of radio coverage [BA25].

To mitigate the requirement for total ordering of handshake messages in distributed environments, Xue et al. [XLH25] proposed Distributed Messaging Layer Security (DiMLS), which adapts MLS to fully distributed scenarios. Note that the abbreviation DMLS can refer to two distinct protocols, distributed MLS and decentralised MLS [Koh25][2]. To distinguish between the two, we refer to distributed MLS as DiMLS in this thesis. At the time of writing, DiMLS remains a draft RFC and has not yet been standardised.

2.5 Distributed MLS

DiMLS is a composition protocol, meaning it builds on and remains compliant with standard MLS, while introducing new primitives and rules for group operations. The central innovation of DiMLS is the elimination of group forking through the introduction of `SendGroups`. In a DiMLS session, every member owns one `SendGroup` and is the sole entity permitted to perform group operations and send messages within it. All other members participate as passive recipients, meaning they cannot propose, commit, or send messages in another member's `SendGroup`. This design removes the need for a complex distributed DS, such as Totem, at the cost of additional overhead proportional to the number of members, since each participant maintains its own group. In DiMLS terminology, the set of all participants in a session is referred to as the `DGroup`, and individual participants are called `DMembers`.

2.5.1 DiMLS session operations

To initialise a DiMLS session, each `DMember` must distribute an initial `KeyPackage`, a `SendGroup` identifying scheme must be agreed upon, and the permitted cipher suites must be defined. Each `DMember` then initialises their own `SendGroup` using the assigned identifier, adds all other members using their distributed `KeyPackages`, and broadcasts the corresponding `Welcome` messages.

To send an encrypted message, the sender encrypts it using their own `SendGroup`. To decrypt a received message, the recipient looks up the MLS group identifier and applies the keys associated with the corresponding `SendGroup`.

To update the epoch secret in a `SendGroup`, the group owner issues a `Commit` that updates their own `LeafNode`. This provides PCS in scenarios where either the epoch secret or the group owner's `LeafNode` has been compromised.

[2]DeMLS is an adaptation of MLS that strengthens FS through the use of puncturable pseudo-random functions (PPRFs)[AMT23]. Unlike DiMLS, which eliminates group forks, DeMLS addresses group forks by securely storing past states, allowing members to resolve divergent group states. However, DeMLS still requires a DS capable of enforcing total message ordering to function correctly, making it unsuitable for the fully distributed UAV swarm setting where such a service has proved difficult to implement.

In the case of a full node compromise, where an adversary has obtained access to a UAV, the `DGroup` must perform the steps below to restore PCS. We use Bob as the compromised member:

1. Bob performs a `SelfUpdate` to rotate his `Leafnode` in his own `SendGroup`. This prevents the adversary from impersonating Bob, but does not prevent them from decrypting messages in the other `SendGroups`, where Bob's compromised `LeafNode` remains.
2. Upon receiving Bob's update, the other members extract a PSK from Bob's healed `SendGroup` and inject it into the key schedule of their own `SendGroup`. Since the adversary has been evicted from Bob's `SendGroup`, they cannot derive this PSK and are temporarily locked out of the other `SendGroups`.
3. Key injection provides PCS only for the current epoch. To permanently evict the adversary, each member must also perform a `LeafNodeUpdate`, replacing Bob's old `LeafNode` in their own `SendGroup` with the healed `LeafNode` from Bob's `SendGroup`. This ensures that the adversary can no longer decrypt any messages using the compromised key material.

For DiMLS to operate correctly, member removal must be propagated across the entire `DGroup`. When Alice removes Bob from her `SendGroup`, all other members, Charlie, David, and so on, must also remove Bob from their own `SendGroups`. The reason is that once Bob is removed from Alice's `SendGroup`, he can no longer incorporate key material from Alice's group into his own key schedule. Any subsequent DiMLS `Commit` by another member will inject a PSK derived from Alice's updated group state, which Bob cannot derive, causing his local state to diverge from the rest of the `DGroup` and rendering him unable to decrypt future messages.

2.5.2 DiMLS considerations

A key consideration when evaluating DiMLS is the increased overhead introduced by the `SendGroup` architecture. This overhead arises from two compounding factors.

First, DiMLS maintains N `SendGroups` rather than a single shared group, meaning every group operation must be performed N times, once per group. In standard MLS, a single `Commit` is sufficient to update the shared group secret, while in DiMLS, each member must commit to their own `SendGroup`, multiplying the cost by N.

Second, only the `SendGroup` owner performs path updates, leaving all internal tree nodes except those on the owner's direct path blank. This results in a flat ratchet tree, where new key material must be encrypted individually to each `LeafNode` rather than to internal nodes representing subtrees. A fully populated MLS tree provides

$O(\log_2 N)$ scaling per commit, while a flat tree requires $O(N)$ encryptions. Combined with the N-fold replication of group operations, the total cost of a PCS update across the DiMLS session grows quadratically with the number of members.

2.6 Tools

This section introduces the tools and technologies used in this project, providing the technical context necessary to understand the implementation and testing methodology.

The programming language of choice in this project is Rust. By combining high-level safety guarantees with low-level control over resource management, Rust ensures both computational efficiency and memory safety [KNC26]. These architectural advantages make the language a preferred choice for implementing cryptographic protocols in resource constrained environments such as UAV swarms. We use the rust-based `openmls` library which is a fully compliant implementation of the MLS RFC [BBR+23].

To build an asynchronous system, we employ the `tokio` library, which provides an asynchronous runtime and non-blocking alternatives to the standard library, enabling efficient and safe concurrent code without manual scheduling or synchronisation [KNC26]. Inter-task communication is handled using Multiple Producer Single Consumer (MPSC) channels, which allow data to be passed safely between threads or tasks [KNC26]. As the name implies, multiple producers can push data to a shared channel, but only a single consumer can read from it. The channel exposes two handles, a sender and a receiver, and we use this pattern throughout our system to implement event-driven communication, where one or more modules produce events that are consumed by another.

To accelerate development and testing, we use a containerised environment based on Docker[3], which allows us to simulate a swarm of nodes on a single host. Docker leverages the namespace feature of the Linux kernel to create isolated environments in which processes operate as if they have their own dedicated instance of the system [MSC26]. This enables us to simulate specific network scenarios and system states, facilitating both debugging and iterative development.

Since UAV swarm environments frequently operate under degraded network conditions, simulating packet loss is essential for realistic testing. We use `tc` (Traffic Control), a Linux kernel utility for configuring network traffic shaping, to inject packet loss at configurable rates in a deterministic and reproducible manner [ipr26]. For traffic generation, we use `mgen` (Multi-Generator), developed by the US Naval

[3] https://www.docker.com/

Research Laboratory, which allows us to test the system under varying traffic loads in terms of both message frequency and message size [Lab26]. CPU consumption is measured using `pidstat` [God26], which reports usage separately for userspace, kernel space, and the combined total, enabling us to distinguish between application-level and system-level overhead.

Performance measurements are collected on an Nvidia Jetson Nano, a single-board computer designed for embedded edge computing [Zen19]. The onboard computer in the Flamingo UAVs is the Jetson Xavier NX, which shares the same Advanced RISC Machine (ARM) architecture but offers considerably greater computational capacity. The Jetson Nano therefore provides a useful estimate of how the system will perform on the target hardware, but results should not be taken as definitive.

Chapter 3
Methodology

In this chapter we describe the methodology used to address the research questions defined in Chapter 1. The project is structured around five phases: research and information gathering, design and planning, implementation, testing, and evaluation. The relationship between these phases is visualised in Figure 3.1, which also illustrates the iterative nature of the approach, where findings in later phases may prompt a return to earlier ones. The phases are designed to build directly on each other, with the implementation phase addressing RQ1 and the testing and evaluation phases structured to collect and analyse the data needed to answer RQ2 and RQ3.

3.1 Research and information gathering

The purpose of this phase is to establish the theoretical and technical foundation necessary to design and implement DiMLS in a UAV swarm context. This includes identifying the key challenges in securing UAV swarm communications, understanding how previous efforts have addressed them, and analysing the DiMLS draft RFC [XLH25] as the primary protocol to be implemented.

The literature review will focus on prior work by Marstrander [Mar23] and Bragstad and Arder [BA25], which provide insights into the limitations of standard MLS with Totem in UAV swarm deployments. These works establish the baseline against which the performance of DiMLS will be evaluated in RQ2. The DiMLS draft RFC [XLH25] will be studied in detail to identify the protocol requirements that the implementation must satisfy.

Technical preparation will be conducted in parallel, including familiarisation with the `openmls` library[1], the Rust programming language and the establishment of a containerised development environment based on Docker. This environment will

[1] https://openmls.tech/

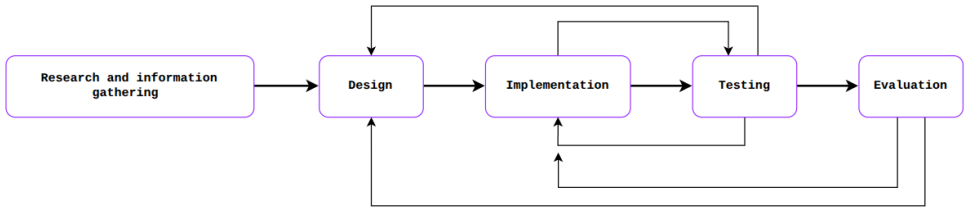


Figure 3.1: Visualization of the methodology in this project.

serve as the primary platform for both development and virtual performance testing throughout the project.

3.2 Design and planning

The intention of this phase is to define the system architecture and establish a clear set of requirements for a viable Proof of Concept (PoC) of DiMLS. The design must account for two categories of concern. First, the functional requirements given by the DiMLS draft RFC [XLH25], and second, the operational constraints of the UAV swarm environment, including packet loss, unreliable connectivity and the absence of centralised coordination.

The architecture will be designed as a cryptographic middleware layer, positioned between the network and the application, to allow integration with the existing Valkyrie system without requiring modifications to the swarm software. The DiMLS draft RFC will serve as the primary blueprint related to the functionality to implement. Because this is an unofficial draft rather than a ratified standard, certain aspects, such as the `DiMLS Commit` operation, require interpretation, and the design is expected to be refined iteratively as implementation progresses and as understanding of the protocol deepens.

3.3 Implementation

The purpose of this phase is to produce a working PoC of DiMLS that satisfies the requirements defined in the design phase, directly addressing RQ1. The implementation will be built on the `openmls` library and developed in Rust. It proceeds in three stages.

First, core MLS functionality will be implemented and verified, including `KeyPackage` generation, group creation, and member addition and removal. Second, the DiMLS specific logic will be implemented on top of this foundation. Third, features specific to the UAV swarm setting will be added, including automatic node discovery and a retransmission mechanism to recover from dropped handshake messages.

Correctness will be verified continuously during development. Unit tests will be used for core MLS features, while the swarm specific features will be tested in the containerised environment. The output of this phase will be a PoC of DiMLS, ready for functional and performance testing.

3.4 Testing

The testing phase is divided into two parts, each targeting different research questions. The first part addresses RQ1 through functional testing, verifying that the implementation behaves correctly on the target hardware and integrates with the Valkyrie system. The second part addresses RQ2 and RQ3 through performance and scalability testing.

Functional testing will be conducted physically on Flamingo UAVs at FFI's facilities. A test matrix is developed covering the core required functionality, including DiMLS initialisation, encryption and decryption of application messages, member addition and removal, PCS updates, and the retransmission mechanism. Each test has a defined expected outcome, and results provide evidence towards RQ1. Integration with the Valkyrie system is verified by routing live telemetry and video streams through the cryptographic middleware.

Performance testing will be conducted using the Jetson Nano and the containerised virtual environment. To address RQ2, network robustness will be evaluated by measuring the decryption success rate as packet loss increases from 0% to 50%, following the same methodology as Bragstad and Arder [BA25] to enable direct comparison. The hypothesis is that DiMLS, by eliminating the total ordering requirement, will maintain a significantly higher decryption rate than MLS with Totem under the same conditions.

To address RQ3, CPU consumption and protocol message bandwidth will be measured as the number of nodes in the swarm increases. CPU usage will be measured at varying packet rates and group sizes to determine how computational overhead scales. Bandwidth requirements will be measured by recording the size of handshake messages as a function of group size, and modelling the total bandwidth consumed by key group operations. The hypothesis is that both CPU consumption and bandwidth scale worse than standard MLS due to the `SendGroup` architecture, and the tests will be designed to measure this scaling.

3.5 Evaluation and analysis

The purpose of the evaluation phase is to analyse the data collected during testing and draw conclusions with respect to the research questions. For RQ1, the functional test

results will be reviewed against the test matrix to assess whether the implementation satisfies the defined requirements. For RQ2, the decryption success rate under packet loss will be compared against the results of Bragstad and Arder [BA25], using the same test parameters to assess the relative resilience of DiMLS. Security and efficiency will be assessed qualitatively against the same baseline. For RQ3, the measured scaling behaviour of CPU consumption and bandwidth will be analysed to determine whether DiMLS remains viable as the swarm grows, and to identify where the primary bottlenecks lie. Where the data reveals additional aspects worth investigating, the implementation phase may be revisited before conclusions are finalised.

Chapter 4
System implementation

This chapter presents the implementation of DiMLS in a UAV swarm context, directly addressing RQ1. We describe the system architecture, the implementation of core DiMLS features, and the reliability mechanisms developed to handle the constraints of the network environment of the swarm. We also mention the limitations of the `openmls` library with respect to implementing the full feature set specified in the DiMLS draft RFC [XLH25].

4.1 System structure

Our system is implemented as a cryptographic middleware positioned between the network and the application, as illustrated in Figure 4.1. A ciphertext received from the network is passed to the middleware, which decrypts it and forwards the resulting plaintext to the application, and vice versa. In the context of the Valkyrie swarm system, the middleware sits between the mesh radio and the Valkyrie application, as shown in Figure 4.2.

For the architecture, we drew inspiration from the modular approach of Marstrander [Mar23] and Bragstad and Arder [BA25], defining modules with clear, well-scoped responsibilities. This made the system easier to extend, debug, and integrate with the Valkyrie system, and allowed us to swap out individual components without affecting the rest of the codebase. The system is composed of four internal modules, described below:

Transport Module. Responsible for interfacing with the mesh radio and the Flamingo software. It receives input from the radio or the application and forwards it to the router. Separating this into its own module provides a clear and intuitive entry point for all traffic, and made it straightforward to add a debug interface for sending control messages to nodes during testing.

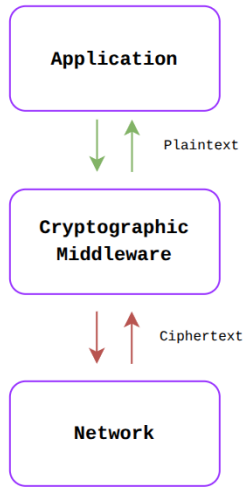


Figure 4.1: Data flow between the network, cryptographic middleware, and application.

Router Module. Responsible solely for routing events between modules. The router was introduced after an initial architecture without it produced excessive complexity and logic errors. Isolating routing into a dedicated module with a single, well-defined responsibility resolved these issues.

Discovery Module. Responsible for automatic node discovery when a node joins the network. Discovery is achieved by broadcasting periodic heartbeats containing the node's identifier, which neighbouring nodes use to detect new hosts. This module also triggers interval-based events such as `SelfUpdate` operations, as it is naturally structured as a recurring loop and provides a convenient location for time-based logic without requiring additional modules.

DiMLS Engine. Responsible for handling encrypted messages, plaintext messages, and protocol messages. It enforces the logic required by the DiMLS draft RFC [XLH25], and performs encryption, decryption, and MLS group operations such as member addition and removal.

The system progresses through three different phases at startup: `setup`, `initialisation`, and `ready`. These phases serve both a structural purpose, providing a clear sequence for how tasks are performed, and a performance purpose, by deferring group operations until sufficient information has been collected to minimise unnecessary protocol messages.

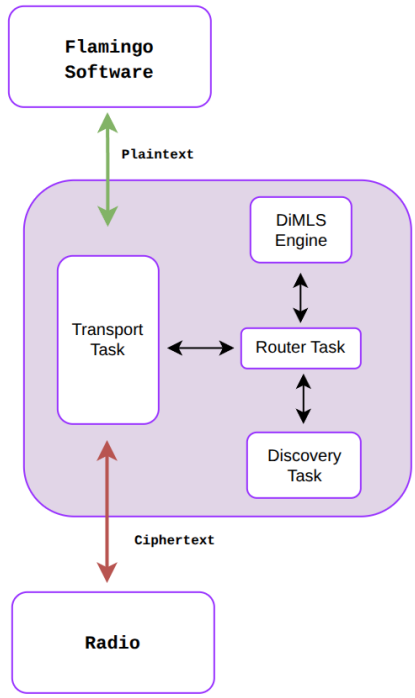


Figure 4.2: Internal architecture of the system.

Setup Phase. On startup, the system reads configuration values from a file, including the list of nodes permitted to join the group, inactivity thresholds for automatic member removal, `SelfUpdate` intervals, and port numbers for the application and radio interfaces. It then initialises two core structures. First `DiMLSClient`, which creates the node's `SendGroup`, stores credentials and signature keys, and generates initial `KeyPackages`. Second `DiMLSSession`, which stores the active `SendGroups`, maintains message queues, and exposes methods for DiMLS session operations. Finally, the different modules are launched in an asynchronous runtime using the `tokio` library, with MPSC channels used for inter-module communication. Once all modules have initialised successfully, the system advances to the initialisation phase.

Initialisation Phase. Upon entering this phase, the node begins broadcasting periodic heartbeats and listening for those of other nodes. When a heartbeat is received, the node responds with a `KeyPackage`, while collecting `KeyPackages` from the others, without processing them. Once the timer expires, all collected `KeyPackages` are committed in a single operation, producing one `Welcome` message for all new members rather than a separate `Commit` and `Welcome` per node. Committing individually would produce $N - 1$ `Commit` messages and N `Welcome` messages. The

batching reduces this to a single `Welcome`, saving both bandwidth and compute. Although the batched `Welcome` is larger, the reduction in message count outweighs this cost.

Ready Phase. In this phase, the node has joined the `SendGroups` of all other active members it could reach in the `initialization` phase, and they have joined its own. The node is now fully operational, able to encrypt and decrypt application messages and perform all DiMLS group operations, including `Add`, `Remove`, and `SelfUpdate`. It continues to emit periodic heartbeats, monitors member inactivity and triggers removal when thresholds are exceeded. Additionally it detects missing or dropped handshake messages, recovering by requesting a retransmission of the lacking `Commit` or `Proposal`.

In summary, the system is composed of four modules with clearly defined responsibilities, progressing through three startup phases before becoming fully operational. It combines MLS operations provided by `openmls`, an asynchronous runtime provided by `tokio` and MLS message handling in the `dmls_engine` module. The following sections will describe the DiMLS implementation in detail.

4.2 DiMLS implementation

The implementation of DiMLS directly addresses RQ1 and represents one of the central contributions of this project. The DiMLS draft RFC [XLH25] serves as the primary blueprint, and since DiMLS is designed to be compliant with standard MLS, we use the `openmls` library, extending it with custom data structures and logic to realize the core concepts of the protocol. This section describes how this is implemented and discusses the obstacles encountered along the way.

4.2.1 Custom data structures

The core idea of DiMLS is that each member, referred to as a `DMember`, sends messages and performs group operations in its own `SendGroup`, while acting as a passive recipient in the `SendGroups` of other members. We define the `SendGroup` structure as shown in Code 4.1. It holds a reference to an MLS group, the group owner, and a timestamp recording the last time the owner performed a group operation. Although the draft RFC [XLH25] specifies that the node at leaf index 0 is the group owner, we store the owner explicitly in the structure for clarity, avoiding the need to resolve ownership by inspecting leaf indices at runtime. The intention of the timestamp is to monitor when the epoch secret was last updated, removing members that has been idle for too long.

Code 4.1: The structure of `SendGroup`.

```
pub struct SendGroup {
    pub group: MlsGroup,
    pub owner: DMember,
    pub timestamp: SystemTime,
}
```

Code 4.2: The Structure of `DiMLSClient`.

```
pub struct DiMLSClient {
    pub member: DMember,
    pub credentials: CredentialWithKey,
    pub signature: SignatureKeyPair,
    pub state: ClientState,
    pub initial_keypackages: HashMap<DMember, KeyPackage>,
    pub backend: OpenMlsBackend,
}
```

The `DiMLSClient` structure, shown in Code 4.2, stores the node's own identifier, the `DMember` field, represented as an unsigned 32-bit integer (`u32`), along with its credentials, signature key pair, and a `ClientState` enum tracking the current phase of the system (`setup`, `initialization`, or `ready`). It also holds a list of pre-generated `KeyPackages` distributed to newly discovered nodes, and a reference to `OpenMlsBackend`, which abstracts many of the underlying MLS operations, such as adding members, producing `Keypackages`, and handling of handshake messages.

The `DiMLSSession` structure, shown in Code 4.3, stores message queues and configuration variables, and implements the methods that provide the DiMLS session functionality. Its fields fall into three categories:

Configuration Variables. The `allowed_ciphersuites` and `export_key_length` fields are defined in accordance with the draft RFC [XLH25]. We use the default `openmls` cipher suite `MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519`, and an export key length of 128 bits. The exported key is extracted from one `SendGroup` and injected as entropy into the key schedule of another. 128 bits is sufficient for this purpose according to RFC 4086 [ECS05].

Lookup Tables. These fields store information in key-value hash maps. The `incoming_ keypackages` field accumulates `KeyPackages` collected during the initialisation phase. The `send_groups` field maps and tracks each `SendGroup` with

Code 4.3: The Structure of `DiMLSSession`.

```
pub struct DiMLSSession {
    // Configuration variables
    pub allowed_ciphersuite: Ciphersuite,
    pub export_key_length: usize,

    // Lookup tables
    pub incoming_keypackages: HashMap<DMember, KeyPackage>,
    pub send_groups: HashMap<DMember, SendGroup>,
    pub message_cache: EpochBuffer,
    pub time_since_last_keypackage: Option<Instant>,
    pub wrong_epoch_timer: HashMap<(GroupId, GroupEpoch), SystemTime>,
    pub handshake_message_tracker: HandshakeMessageTracker,

    // Outgoing queues
    pub handshake_message_queue: MessageQueue,
    pub retransmit_request_message_queue: MessageQueue,
    pub encrypted_message_queue: MessageQueue,
    pub key_export_queue: MessageQueue,
}
```

its owner. The `message_cache` buffers incoming handshake messages whose epoch is too high to process immediately, holding them until the missing `Commit` has been received and applied. To avoid premature retransmission requests caused by jitter or out-of-order delivery, the `wrong_epoch_timer` field defines a configurable waiting period, for example 500 ms, before a retransmission request is issued. Finally, the `handshake_message_tracker` caches all outgoing handshake messages, enabling retransmission when a peer requests a missing message.

Queues. The remaining fields are outgoing message queues, each of type `MessageQueue`, a custom MPSC structure carrying tuples of `(IpAddr, Vec<u8>)`, pairing a destination address with a serialized message. Separate queues are maintained for outgoing handshake messages, encrypted application messages, retransmission requests, and key export requests. Separating queues by message type allows each message to be wrapped in the appropriate enumeration variant before being passed to the router, which uses the type information to attach an identifier during serialisation. Receiving nodes use this identifier to dispatch the message to the correct handler.

`DiMLSSession` exposes a number of methods implementing the core DiMLS operations, including adding and removing members from `SendGroups`, handling

`SelfUpdates`, and key injection from other `SendGroups` to provide PCS. The logic behind these operations is described below.

Adding Members Members are added upon receiving a `KeyPackage`, handled by the `process_keypackage_in()` method on `DiMLSSession`. Before producing a `Commit` and `Welcome`, the method performs up to three checks:

1. **Verifying the intended recipient of the `KeyPackage`:** Since the Valkyrie swarm operates in a multicast environment and a `KeyPackage` may only be consumed once, each `KeyPackage` must be addressed to a specific node. We achieve this using the `application_id` extension on the `KeyPackage`, which the sender sets to the intended recipient's identifier. Each receiving node compares this field against its own identifier and discards the `KeyPackage` if there is no match.
2. **Checking for duplicate credentials:** Due to packet loss and out-of-order delivery, a `KeyPackage` may arrive for a node whose credentials are already present in the receiver's `SendGroup`. If the credentials are not present, a `Commit` and `Welcome` are produced. If they are already present, a third check is required.
3. **Handling a missed `Welcome`:** If `KeyPackages` continue to arrive from a node whose credentials are already in the group, this indicates that the sender did not receive our `Welcome`. To rule out delayed or out-of-order delivery, a timer is started upon receiving such a `KeyPackage`. If the timer expires and `KeyPackages` are still arriving, we conclude that the `Welcome` was lost. Before retransmitting, we check whether the epoch has advanced since the `Welcome` was stored. If it has not, the stored `Welcome` is retransmitted directly. If it has, the `Welcome` is no longer valid, and we must either retransmit the `Welcome` together with the missing `Commits`, or remove and re-add the member. To minimise bandwidth usage, we retransmit the `Welcome` and missing `Commit` if the epoch only has increased by two, and remove and re-add the member otherwise. In either case, at most three handshake messages need to be transmitted to recover.

Removing Members Member removal in DiMLS can be triggered in three ways:

1. **Inactivity threshold exceeded:** A member has not advanced the epoch secret in their `SendGroup` within a configurable threshold. This mechanism automatically removes members that have been offline for too long, as prolonged inactivity presents a threat to PCS.
2. **Cascading removal from another `SendGroup`:** As specified in the draft RFC [XLH25], when a member is removed from one `SendGroup`, all other

members must remove them from their own `SendGroups` upon processing the corresponding `Commit`. This is necessary because if Alice removes Bob from her `SendGroup` but Charlie does not, Charlie's next DiMLS `Commit`, which could inject key material from Bob's `SendGroup` as a PSK would produce an epoch secret that Alice can no longer derive, breaking the shared group state.

3. **Operator-issued removal command:** The GCS can issue an explicit command to remove a member from the swarm, for example if the operator suspects that a UAV has been compromised.

The removal of a member from one `SendGroup` triggers a cascading removal from all other `SendGroups` in the DiMLS session. When a node receives a `Commit` removing a member from another node's `SendGroup`, it removes the same member from its own `SendGroup`. Correspondingly, when a node detects that it has itself been removed from a `SendGroup`, it automatically deletes that group from its local state.

Performing PCS Updates Achieving PCS in our system requires two steps:

1. **`SelfUpdate` in own `SendGroup`:** The member rotates its leaf key and updates the internal nodes along its direct path, ensuring PCS for its own `SendGroup`.
2. **Key injection from other `SendGroups`:** A member issues a `Commit` incorporating a PSK extracted from another member's `SendGroup`, labelled with the source group identifier and epoch. By injecting this PSK into the key schedule, the epoch secret becomes dependent on key material that an adversary who has compromised the current state cannot derive, thereby providing PCS across the DiMLS session.

In our implementation, the `SelfUpdate` is performed automatically at a configurable interval. Key extraction and injection are implemented as a PoC only, requiring the GCS to explicitly issue commands to the nodes to perform the operation. PCS-updates in DiMLS is discussed in depth in Chapter 4.2.3.

4.2.2 The `dimls_engine` module

The `dmls_engine` module handles all DiMLS-relevant events, including processing incoming MLS messages, retransmitting handshake messages, and performing automatic `SelfUpdate` operations. The module is implemented as an infinite loop using the `tokio::select!` macro, which monitors several concurrent event sources and dispatches the handler for whichever event is ready first. Conceptually, this is analogous to a chef monitoring multiple timers for different dishes simultaneously. When the timer for one dish fires, the chef attends to it, then returns to waiting. In the

same way, `select!` waits for the first available event and executes the corresponding handler before returning to the loop. The events handled by the `dmls_engine` are as follows:

`mls_engine_event` Triggered when an encrypted MLS message is received, or when an event affecting group state occurs, such as an automatic `SelfUpdate` or a discovery event. The possible sub-events are shown in Figure 4.3 and described below:

- `MlsMessage` Handles all incoming MLS messages, including both application and handshake messages.
- `Discovery` Handles incoming discovery messages from nodes joining the network.
- `AutoUpdate` Triggers an automatic `SelfUpdate` in the node's own `SendGroup` at a configurable interval.
- `HandshakeRetransmitRequest` Handles incoming requests from peers for retransmission of missed handshake messages.
- `KeyExport` Triggers a key export from a specified `SendGroup` for use in external symmetric cryptographic operations.
- `FromInitToReady` A timer-triggered event that advances the system state from `initialisation` to `ready` once the initialisation phase is complete.

`plaintext_event` Triggered when plaintext is received from the application. The plaintext is encrypted using the node's own `SendGroup` and the resulting ciphertext is pushed to the `app_msg_queue`.

`app_msg_queue_event` Fired when an encrypted application message is pushed to the `app_msg_queue`. The message is wrapped in an `MlsOut::ToNode` variant and forwarded to the router for transmission.

`protocol_msg_queue_event` Fired when a MLS protocol message is produced and pushed to the protocol message queue. As with application messages, it is wrapped in an `MlsOut::ToNode` variant and forwarded to the router.

`retransmit_req_queue_event` Fired when the node produces a `HandshakeRequest` and pushes it to the `retransmit_req_queue`, as described in Section 4.3. The message is wrapped in an `MlsOut::HandshakeRequest` variant and forwarded to the router.

`key_export_req_queue_event` Triggered when a node wishes to export a symmetric key from its own `SendGroup` for use in operations such as encrypted video streaming. The node generates a `KeyExportMessage`, appends it to

the `key_export_req_queue`, wraps it in an `MlsOut::KeyExport` variant, and forwards it to the network.

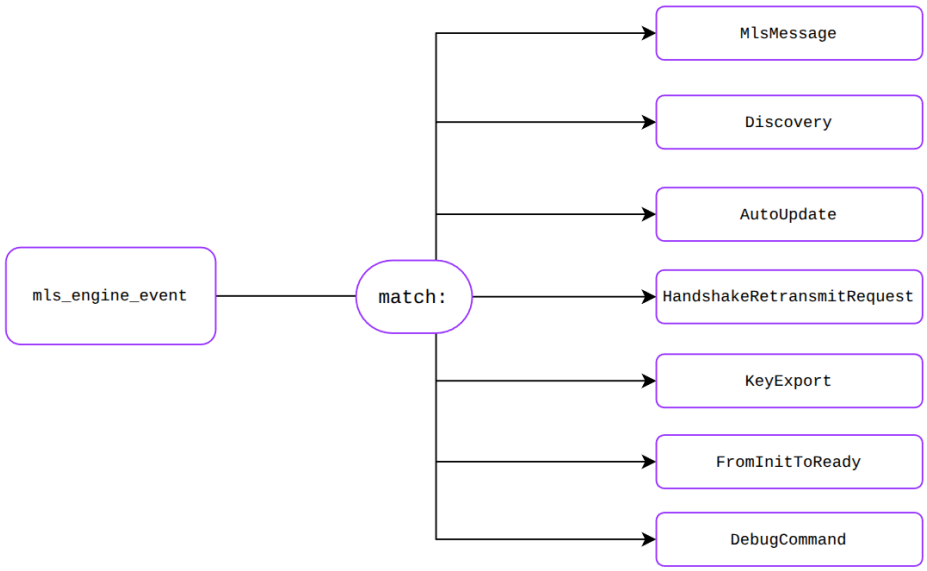


Figure 4.3: The different sub events an `mls_engine_event` can contain.

4.2.3 PCS updates

Achieving PCS in DiMLS is more complex than in standard MLS, as multiple `SendGroups` may need to be healed depending on what has been compromised. We distinguish between two scenarios.

Root key compromise. If only the root key of a `SendGroup` is compromised, the group owner performs a `SelfUpdate` to rotate the root key. This is sufficient to block an adversary from deriving future message secrets, and is the simplest form of PCS recovery.

Leaf node compromise. If a `Leafnode` belonging to a member other than the group owner is compromised, the DiMLS draft RFC [XLH25] specifies that this should be healed by performing a `LeafnodeUpdate`. This operation replaces the compromised `Leafnode` in the affected `SendGroup` with updated key material from the compromised member's own `SendGroup`. However, the current version of `openmls` does not expose the necessary methods to update another member's `Leafnode`, making this operation unavailable in our implementation. As a result, our system handles leaf node compromise in two ways. The first is temporary recovery via PSK injection.

This is done by the group owner extracting a PSK from the healed `SendGroup` of the previously compromised member and injecting it into the key schedule of its `SendGroup`. This introduces entropy that only members with valid `Leafnodes` in the source group can derive, making it infeasible for the adversary to derive message encryption keys for the current epoch. However, the compromised `Leafnode` remains in the group and can still derive the root key when the next `Commit` arrives, meaning this approach provides temporary PCS only. This functionality is implemented in our system with the `DiMLSSession::perform_pcs_update()` method, which takes the source `SendGroup` as an argument and proceeds as follows:

1. A `psk_id` is constructed, where the first eight bytes represents the source group and the last eight bytes represents the epoch from which the key is extracted.

2. A key is exported from the source group using `MlsGroup::export_secret()`, which takes the `psk_id`, a predefined label, and the desired key length as arguments. We set the label to "exporter-psk" and the key length is given by the `DiMLSSession.export_key_length`, which is 128 bits.

3. The exported PSK is proposed to be injected to the initiating node's `SendGroup`. The proposal is broadcast to all members, who use the `psk_id` to identify the source group and epoch, and extracts the corresponding key independently. The key is then stored in their keystore.

4. The initiating node commits the proposal, which injects the PSK into the key schedule of its `SendGroup`. The other members process this `Commit`, retrieve the PSK from their `keystore`, and injects it into the key schedule for the new epoch.

The other approach is permanent recovery via remove and re-add. To fully heal the group, the group owner removes the compromised `Leafnode` and re-adds them once they transmit a fresh `KeyPackage`. This is the only way to permanently evict the compromised `Leafnode`, but is costly in terms of bandwidth, requiring two `Commits`, one `Welcome` and one `Keypackage`, per group.

4.3 Reliability mechanisms

While DiMLS is designed to avoid negotiating a common state among members, handshake messages must still arrive and be processed in the correct sequence. In the unreliable mesh network of a UAV swarm, this requires reliability measures to ensure handshake delivery. Our solution is a simple retransmission mechanism built around three components:

1. A local cache of all handshake messages the UAV produces.
2. A timer that prevents unnecessary retransmission requests caused by delayed or out-of-order delivery.
3. A retransmission request message identifying the group and epoch for which a handshake message is missing.

Handshake message cache. The cache is implemented as a custom type `HandshakeMessageTracker`, shown in Code 4.4. It maps each `GroupEpoch` to a list of serialised handshake messages produced in that epoch, such as `Commit`s, `Welcome`s, and `Proposal`s. When a peer requests a missing handshake message, the relevant entry is retrieved from this cache and retransmitted.

Wrong-epoch timer. When a node receives a message with an epoch higher than its current local epoch, it records a timestamp in a `WrongEpochTimer`, which is a map from `(GroupId, GroupEpoch)` to a `SystemTime`, as shown in Code 4.4. If messages continue to arrive in the wrong epoch and the elapsed time exceeds a configurable threshold, the node concludes that a handshake message has been lost and issues a retransmission request. Through testing, we determined 200 ms to be a sufficient threshold. Long enough to absorb jitter and out-of-order delivery, while minimising the time spent in a stale epoch.

Retransmission request. If the threshold is exceeded, the node constructs a `HandshakeRequest`, which is a tuple of the `GroupId` and the node's current epoch, as shown in Code 4.4. This is wrapped in a `HandshakeRetransmit` enum and forwarded to the router, which prepends a type byte and passes it to the transport module for transmission. Upon receiving a `HandshakeRequest`, the receiving node checks whether the `group_id` corresponds to its own `SendGroup`. If it does, it retrieves the cached handshake messages for the requested epoch from its `HandshakeMessageTracker` and appends them to the outgoing handshake-message queue. A dedicated `HandshakeResponse` variant was considered but found to be unnecessary, as the cached messages could be handled through the existing handshake message pipeline without introducing a new event type.

4.4 Development and testing tools

This section describes the supporting code and tools developed during the project that contributed to both the development process and the final implementation. This includes the `OpenMlsBackend`, which abstracts the core MLS operations, the `commander` Command Line Interface (CLI) used to test DiMLS features, and a status dashboard used to verify system behaviour during testing.

Code 4.4: Data types used for handshake message reliability.

```
pub type HandshakeMessageTracker = HashMap<GroupEpoch, Vec<Vec<u8>>>;
pub type HandshakeRequest = (GroupId, GroupEpoch);
pub type WrongEpochTimer = HashMap<(GroupId, GroupEpoch), SystemTime>;
pub enum HandshakeRetransmit {
    HandshakeRequest(HandshakeRequest)
}
```

The `OpenMlsBackend` provides a set of methods encapsulating core MLS operations, including group creation, `KeyPackage` generation, and the processing of `Welcome` and `Commit` messages. These methods are accessible through the `DiMLSClient.backend` field. By abstracting these operations into a dedicated backend, the implementation code remains focused on DiMLS-specific logic rather than low-level MLS mechanics, improving readability and maintainability.

To test and verify functionality, we developed a CLI tool named `commander`, which acts as a simulated GCS in our test environment. Each node listens on a dedicated debug port and responds to commands issued by the tool. The supported operations are `remove`, `add`, `self-update`, `pcs-update`, `export-key`, and `faulty-commit`. The `faulty-commit` command instructs a node to commit to its `SendGroup` without broadcasting the resulting `Commit` message, simulating a dropped handshake message and triggering the retransmission mechanism when the other nodes detect the epoch difference.

As we increased the node count during development, monitoring system state through debug logs became convoluted. To address this, we built a lightweight server running on the host machine, to which each node broadcasts a status packet once per second. Each packet contains the node's identifier, its list of `SendGroups`, and the current epoch for each group. The server aggregates and displays this information in a web dashboard, shown in Figure 4.4. Combined with the `commander` CLI, this dashboard allowed us to trigger group operations and immediately verify their effect across all `SendGroups`, significantly reducing the time needed to diagnose issues as the system grew in complexity.

Together, these tools formed an essential part of the development infrastructure, enabling rapid iteration and systematic verification throughout the project.

MLS Node Dashboard

node4

Send Group Owner	node5	node3	node2	node4
Epoch	3	3	3	3
Members	node5, node4, node2, node3	node3, node2, node5, node4	node2, node5, node4, node3	node4, node3, node2, node5

node5

Send Group Owner	node2	node4	node5	node3
Epoch	3	3	3	3
Members	node2, node5, node4, node3	node4, node3, node2, node5	node5, node4, node2, node3	node3, node2, node5, node4

node2

Send Group Owner	node3	node2	node4	node5
Epoch	3	3	3	3
Members	node3, node2, node5, node4	node2, node5, node4, node3	node4, node3, node2, node5	node5, node4, node2, node3

node3

Send Group Owner	node5	node2	node3	node4
Epoch	3	3	3	3
Members	node5, node4, node2, node3	node2, node5, node4, node3	node3, node2, node5, node4	node4, node3, node2, node5

Figure 4.4: Status dashboard used during development of the system.

Chapter 5
Testing and results

This chapter describes the tests conducted on our system and presents the corresponding results. Testing was carried out both physically on the Flamingo UAVs at FFI's facilities and virtually using a containerized network of simulated nodes, using Docker. This is visualised in Figure 5.1. Performance measurements were also collected on the Jetson Nano to evaluate the system under hardware constraints representative of the target platform. The following sections cover the setup, methodology, and results of the tests.

5.1 Physical testing

Physical testing was performed on Flamingo UAVs at FFI's facilities in Kjeller. The primary objectives were to verify that the system runs correctly on the target hardware, operates over the Rajant mesh radios, and can be integrated with the Valkyrie software to protect telemetry and video streams. These tests contribute to answering RQ1 by confirming that the implementation behaves as expected on an actual UAV swarm.

5.1.1 Test setup

Three Flamingo UAVs connected via Rajant mesh radios with IP interfaces were used for testing, along with an external mesh radio node for the test laptop. The UAVs were powered by a bench power supply and remained earthbound through all tests. A GCS was also made available, enabling integration with the Valkyrie software and testing with real telemetry data rather than simulated traffic. The physical setup is shown in Figure 5.2.

5.1.2 Functionality testing

Prior to the visit to FFI, a suite of functionality tests was developed and carried out during the testing session.

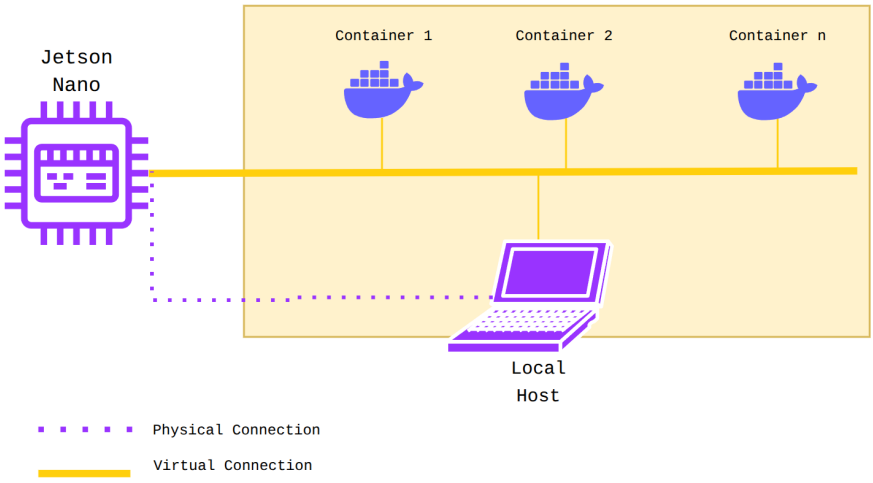


Figure 5.1: Environment used for the development and testing of our system.

Test 1: Compilation and deployment. The software was cross-compiled for the ARM architecture and copied to the UAVs. The configuration files were updated with the correct IP addresses and `NODE_ID`s corresponding to the Rajant mesh radio subnet (`192.168.12.0/24`). The system time on the UAVs had to be set manually, as it defaulted to 1 January 1970 on boot, causing `KeyPackage` verification to fail. Additionally, the UAVs exposed two Wi-Fi interfaces alongside the Rajant radio interface, all sharing the same IP range. To ensure traffic was routed exclusively over the mesh radio, the Wi-Fi interfaces were disabled. Once these steps were completed, our system ran successfully.

Test 2: DiMLS initialisation. The purpose of this test was to confirm that each node created its `SendGroup` and added the other UAVs, as specified by the DiMLS draft RFC [XLH25]. The test was conducted with both the default `Basic` credential and the `Ed25519Credential` implementation from Bragstad and Arder [BA25], and passed successfully for both.

Test 3: Encryption and decryption. Once the groups were established, the `mgen` traffic generator was used to produce data packets for the system to process. Application logs confirmed that the system successfully encrypted outgoing data and decrypted incoming packets for both credential types.

Figure 5.2: Test setup with three Flamingo UAVs and a GCS.

Tests 4 and 5: `SelfUpdate` and PCS updates. Automatic `SelfUpdate` operations were verified at a 10-second interval. PCS updates, as described in Section 4.2.3, were triggered manually using the `commander` CLI. Both operations were confirmed successful through the debug output logs.

Tests 6 and 7: Member addition and removal. Manual addition and removal of nodes were performed using the `commander` CLI and verified through debug output. Automatic removal was also tested by terminating the process on one UAV and

confirming that the corresponding member was removed from the remaining two `SendGroups` after the inactivity threshold was exceeded.

Test 8: Reliability mechanism. Prior to dedicated testing, the reliability mechanism was observed to function correctly during other tests. Even with the UAVs in close proximity, packet loss was present over the mesh radio. When group operations were performed in rapid succession, some protocol messages were lost, but the output logs confirmed that the missing messages were detected, requested, and retransmitted, with all groups converging to a consistent state. This was further validated explicitly using the `commander` CLI, by instructing a UAV to commit to its `SendGroup` without broadcasting the `Commit`. The remaining UAVs detected that incoming application messages were one epoch ahead of their local state, issued a retransmission request for the missing `Commit`, and successfully recovered, confirming the correctness of the mechanism.

Overall, the functionality tests passed with few issues. This was expected given the extensive testing conducted in the virtual environment prior to the visit. The physical tests nonetheless provided valuable insight into the natural packet loss characteristics of the mesh network and practical experience with the target hardware.

5.1.3 Integration with Valkyrie

Network traffic in the Valkyrie system can be categorised as telemetry, command and control, and video streaming. Integration with Valkyrie was straightforward for telemetry, but presented challenges for command and control traffic and video streaming. We describe the different processes below.

Telemetry encryption. Routing telemetry through our system on the Flamingo UAV was achieved by redirecting the outgoing multicast address in the Flamingo configuration file to the localhost address and port our system ran on. The same change was applied on the GCS. With these two configuration changes in place, telemetry transmitted by the UAVs was passed to our system, encrypted, sent over the network, received by the GCS, decrypted, and displayed to the operator in Valkyrie, confirming end-to-end encrypted telemetry.

Command traffic. Routing command traffic from the GCS to the UAVs proved more difficult. This traffic is unicast, and the Valkyrie software performs automatic peer discovery at runtime, meaning the destination addresses are not statically configurable. As a result, we had no mechanism to intercept and route this traffic through our system without modifying the Valkyrie source code, which was outside the scope of this project. Consequently, we were not able to encrypt the command traffic in the testing period.

Video streaming. Version mismatches between the Valkyrie software on the UAVs and the GCS caused the GCS application to crash intermittently and prevented the video feed from initialising. To work around this, a debug tool used internally by the Valkyrie development team was used, which allowed a UAV to be instructed to stream video to a specified IP address and port. By directing the stream to the localhost and the port of our system, the video data was encrypted and forwarded over multicast, then decrypted on the GCS and displayed in the debug application. While this was a workaround, it served as a PoC that encrypted video streaming with DiMLS is feasible.

The video stream suffered from a low frame rate of approximately one frame per second, which would be unusable in an operational context. However, as noted by Marstrander [Mar23], Rajant mesh radios buffer multicast traffic, and we believe this buffering, rather than the performance of our system, was the primary cause of the degraded frame rate. Encrypted video streaming over unicast would likely perform significantly better, but this was not tested.

5.2 Virtual testing

This section presents the performance and scalability testing of our system, examining how it responds to variations in parameters such as packet loss, packet frequency, packet size, and group size. The results provide insight into the suitability of DiMLS for operational UAV swarm use, covering protocol resilience, resource consumption, and behaviour under degraded network conditions. These tests are designed to answer RQ2 and RQ3.

Where applicable, tests follow the same methodology as Bragstad and Arder [BA25] to enable direct comparison of results. In addition, we introduce tests specific to our implementation, notably measuring how handshake message sizes and bandwidth requirements grow as the number of nodes increases, which is of particular relevance given the additional overhead introduced by DiMLS relative to standard MLS.

The test infrastructure is shown in Figure 5.1. Performance measurements are collected on the Jetson Nano, while a network of swarm nodes is simulated using Docker containers. Time constraints prevented us from conducting performance tests on the Flamingo UAVs and this limits the direct comparability of our results with those of Bragstad and Arder, as our tests were run on a Jetson Nano rather than the Jetson Xavier NX. While the Jetson Nano provides a useful estimate of system performance, the hardware difference must be taken into account when comparing results.

Traffic is generated using `mgen`, and packet loss is injected at configurable rates using

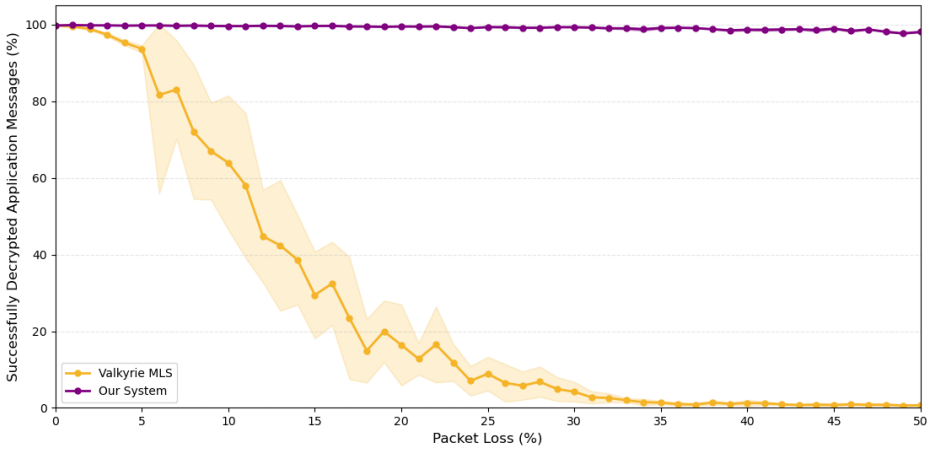


Figure 5.3: Rate of successfully decrypted application messages with increasing packet loss, comparing DiMLS with MLS and Corosync [BA25].

`tc`. CPU utilisation is measured using `pidstat` at a sampling rate of one sample per second. All tests are run with the application compiled in release mode, achieved by using the `--release` flag.

5.2.1 DiMLS performance under packet loss

This test measures how our system performs as the rate of packet loss increases, and the results directly compare against those of Bragstad and Arder [BA25]. We replicate their setup using three Docker containers, with `tc` injecting random packet loss at an increasing rate. The `mgen` tool generates traffic at 10 packets per second with a packet size of 4 kB. Nodes perform a path update in their `SendGroups` every 5 seconds, compared to every 10 seconds in Bragstad and Arder's setup. Packet loss is incremented by one percentage point per run, with each run lasting ten minutes.

Figure 5.3 plots our decryption rate alongside those of Bragstad and Arder [BA25], whose system uses standard MLS with Totem [AMM+95] as the distributed DS. The results demonstrate that our implementation of DiMLS coupled with the retransmission mechanism is significantly more resilient to packet loss than standard MLS. While MLS with Totem is unable to recover a synchronized state as the packet loss rate increases, DiMLS decryption rate is largely unaffected by the constrained network conditions. This increase in robustness against packet loss makes DiMLS a more viable choice than Valkyrie MLS as the UAV swarm will operate in contested environments with high packet loss.

Figure 5.4 shows the decryption success rate measured in 10-second intervals. Even

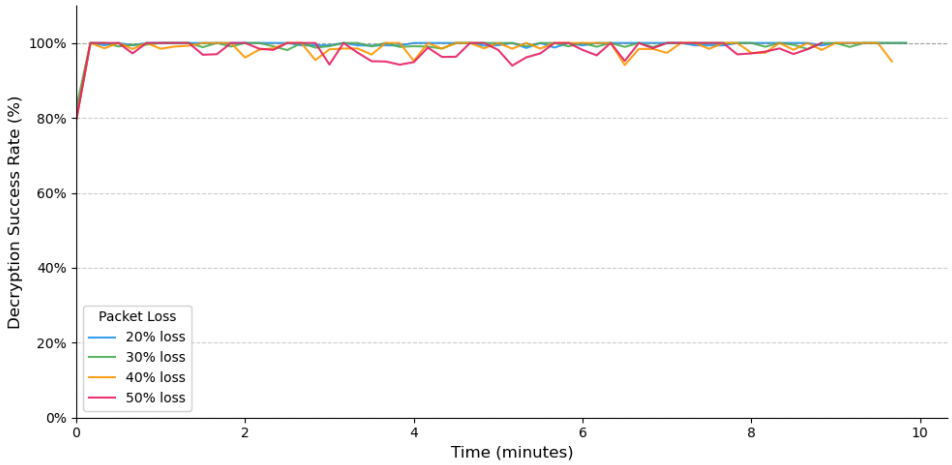


Figure 5.4: Decryption success rate over time at varying packet loss rates.

with 50% packet loss, the system maintains a relatively stable state, with only occasional drops occurring when handshake messages are lost. In these cases, the system recovers by requesting retransmission of the missing handshake messages, allowing the groups to quickly resynchronize. This demonstrates that our system can recover from lost handshake messages in a simple and efficient manner.

In contrast, in MLS, a missed or simultaneous `Commit` may result in group forking, requiring all nodes to coordinate in order to restore a synchronized group state. Furthermore, the Valkyrie MLS implementation requires the Totem group membership to converge before communication is permitted. As a result, under high packet loss conditions, Totem may repeatedly include and exclude members, causing communication to be blocked until convergence is achieved. In the worst case, a single node operating at the edge of radio coverage could prevent the entire swarm from communicating by preventing the Totem group from converging. This issue is mitigated in DiMLS, where each node independently maintains the state of its own group. Consequently, the protocol essentially eliminates the chance of group forking and does not depend on an external membership service converging before data transmission can continue, furthermore strengthening the argument for DiMLS.

Given the strong resilience observed with three nodes, we extended the test to larger swarm sizes to evaluate how performance scales. Figure 5.5 shows results for groups of 3, 5, 8, and 20 nodes. The decryption rate decreases as group size increases, though the system remains reasonably robust for smaller groups. For a group of 20 nodes, however, a sharp decline in decryption rate is observed even at low packet loss rates.

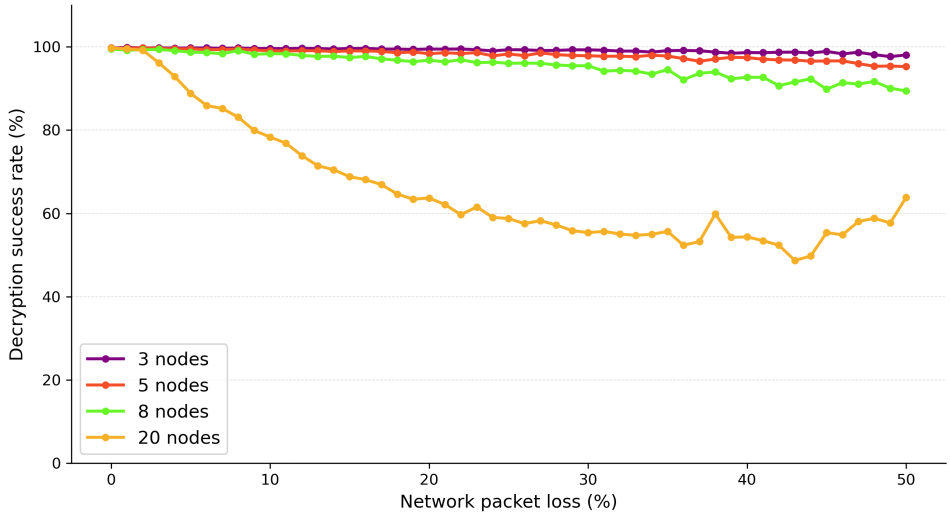


Figure 5.5: Decryption success rate for different group sizes under increasing packet loss.

Investigation revealed that this degradation is caused by the `openmls` library blocking packet processing while resynchronising the forward ratchet after packet loss. Since our system is single-threaded, multiple simultaneous ratchet resynchronisations cause a noticeable processing hang. During this hang, incoming packets are dropped because the processing pipeline is temporarily blocked and not because the group state is inconsistent. This likely inflates the number of packets recorded as not decrypted correctly, and should be taken into account when interpreting the results for larger group sizes.

5.2.2 DiMLS CPU performance testing

In a UAV swarm context, CPU resources are scarce, making it important to minimise the computational overhead of cryptographic operations so that the UAV can dedicate its processing capacity to operational tasks. This test aims to quantify the CPU consumption of our system and assess how it scales as the number of nodes in the network increases.

Ideally, these tests would be performed directly on the Flamingo UAVs, but time constraints prevented this. Instead, measurements were collected on the Jetson Nano, which shares a similar architecture but has significantly weaker hardware than the Jetson Xavier NX found in the Flamingo UAVs. This is a methodological limitation as the results do not reflect exact CPU usage on the target hardware. However, they provide a useful estimate of system performance and help identify which factors have

Parameter	Default Value
Packet Size	4 kB
Packet Frequency	10 packets/s
Self-Update Interval	10 seconds
Max Forward Distance	1000
Out-of-Order Tolerance	5
Max Past Epochs	1

Table 5.1: Default parameters used for CPU performance testing.

the greatest influence on computational efficiency. For this reason, the results are only indicative of performance, rather than definitive.

The default parameters used across all CPU tests are listed in Table 5.1. These values were chosen to match those used by Bragstad and Arder [BA25], enabling comparison with their results.

Test 1: Packet volume. This test measures CPU usage across different packet rates and packet sizes while keeping the total byte throughput constant by varying packet frequency and size inversely. The results, shown in Table 5.2, demonstrate that CPU usage increases with packet rate despite the corresponding decrease in packet size. This observation is consistent with the findings of Marstrander [Mar23] and Bragstad and Arder [BA25].

Since each MLS packet must be individually authenticated and decrypted, the cryptographic overhead dominates the processing cost, while the total data volume has comparatively little impact on CPU usage. This is further illustrated in Figure 5.8, where a tenfold increase in packet size results in only a modest increase in CPU consumption.

Marstrander [Mar23] reports that the Flamingo UAVs transmit between 2 and 50 packets per second, with packet sizes ranging from 1 kB to 50 kB. Our results show that workloads with high packet frequencies impose a processing burden, while the packet size is not the determining factor for CPU consumption, meaning low frequency packet rates, combined with large packet sizes, are ideal for MLS.

Test 2: Group size. This test examines how CPU usage scales with the number of nodes in the network. We begin with a baseline measurement using three nodes and the default parameters from Table 5.1, replicating the setup of Bragstad and Arder [BA25]. As shown in Figure 5.6, CPU usage remains stable, with an average of 11.9%.

Packets per second	Packet size (B)	CPU usage (%)
10	8000	16.4
20	4000	24.9
50	1600	39.6
80	1000	50.3
100	800	54.4

Table 5.2: CPU usage for varying packet frequency and size at constant throughput.

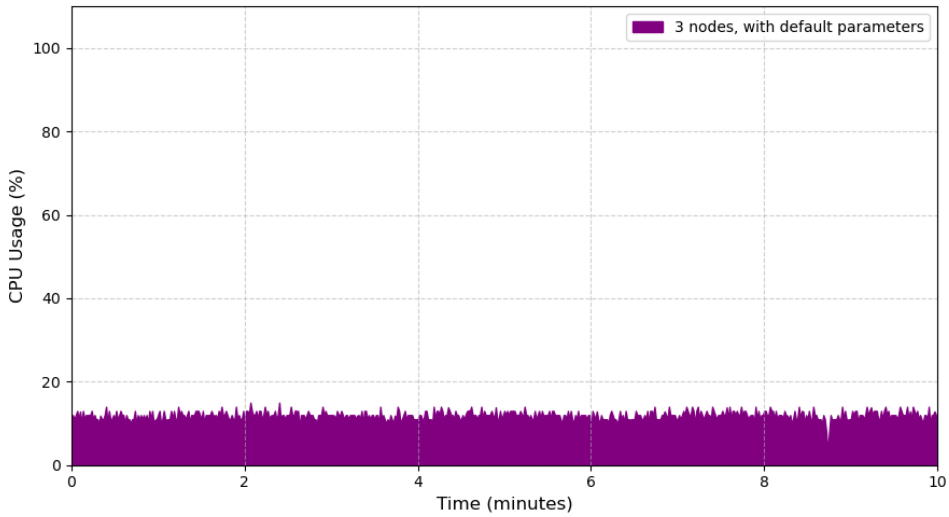


Figure 5.6: Baseline CPU usage with three nodes on the Jetson Nano.

Bragstad and Arder reported an average CPU usage of 4.9% for a comparable baseline scenario. However, their measurements were performed on a Jetson Xavier NX, which provides significantly greater computational performance than the Jetson Nano used in our experiments. The higher CPU utilization observed in our results is therefore expected and primarily reflects the differences in hardware capability rather than differences in protocol behaviour.

Next, we want to observe how CPU consumption increases as the number of members in the swarm increases. We do this by deploying one node every ten minutes, reaching a total of ten members, with nine Docker containers and the Jetson Nano. All nodes transmit 10 packets per second with a size of 4 kB. Figure 5.7 breaks down the CPU consumption into network-level (kernel) usage and application-level (userland) usage. The network-level component reflects the cost of simulated traffic and cannot be reduced through application optimisation, while the userland component represents the overhead of our code. The results show that the majority of CPU usage stems

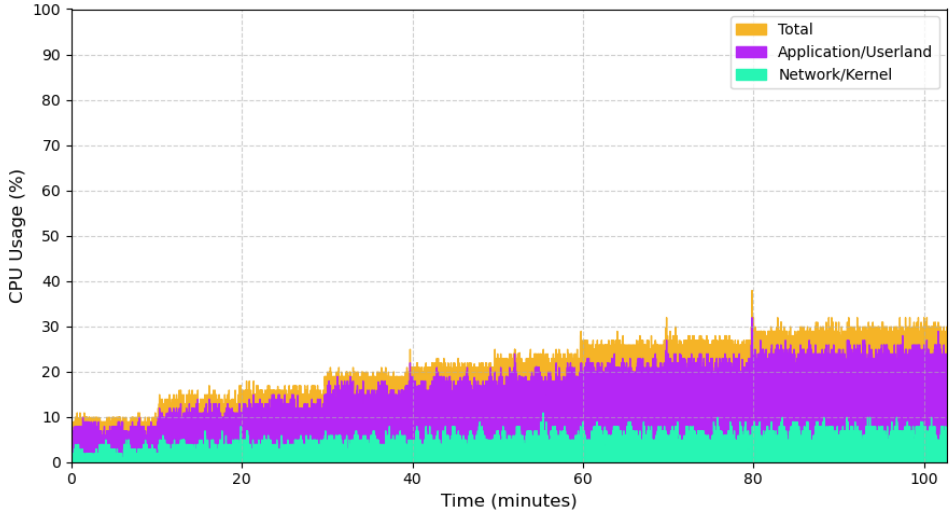


Figure 5.7: CPU consumption as node count gradually increases from 1 to 10.

from the userland component, meaning the userland component is the primary consumer of CPU resources.

To push our system, we did a final test where we deployed nodes at one-minute intervals until either the host or the Jetson Nano became CPU-saturated, testing at packet sizes of 400 B and 4 kB at a packet rate of 10 packets per second. Figure 5.8 shows CPU consumption for network sizes between 1 and 40 nodes. Both configurations grow approximately linearly, with only a small difference between the two packet sizes, consistent with the findings of Test 1. Pronounced spikes are visible whenever a new node joins, attributable to two compounding factors. First, each new member triggers a burst of `Welcome` and `Commit` messages across the group. Second, since nodes perform a `SelfUpdate` every 10 seconds and are deployed at one-minute intervals, the `SelfUpdate` cycles of all nodes periodically synchronise, producing a significant spike in CPU consumption that compounds with the join related overhead.

To identify which parts of the system consume the most CPU, we produced a flamegraph of the running application, shown in Figure 5.9. In a flamegraph, the x-axis represents the proportion of time spent in each function, while the y-axis shows the call hierarchy. Wider bars therefore indicate functions with higher CPU usage.

Analysis of the flamegraph shows that `openmls` accounts for the vast majority of the CPU consumption, while the logic introduced by our implementation contributes comparatively little overhead. This is expected, as the system must perform a large

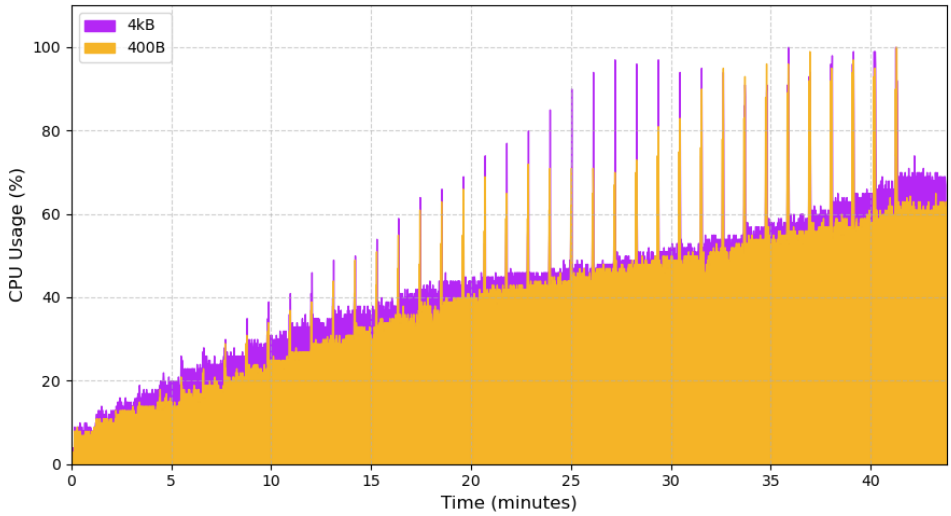


Figure 5.8: CPU consumption for 1 to 40 nodes at packet sizes of 400 B and 4 kB.

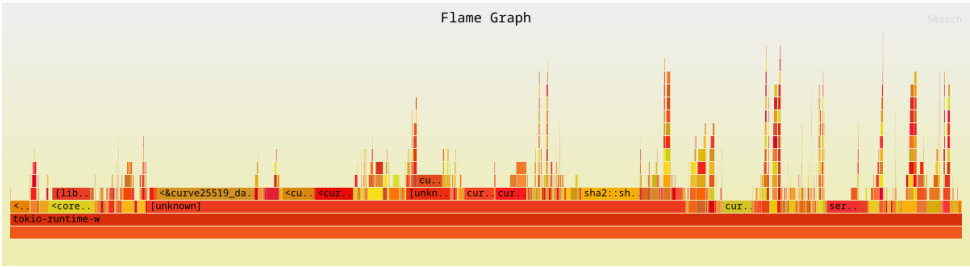


Figure 5.9: Flamegraph of our system showing the call stack and how much relative CPU time each function uses.

number of authentication and decryption operations, and the cost of these operations increases proportionally with the number of nodes. The results therefore indicate that the additional logic imposed by DiMLS has only a minor impact on the overall CPU usage.

5.2.3 Protocol message bandwidth usage

Bandwidth is a constrained resource in UAV swarm environments. This section measures the bandwidth consumed by DiMLS protocol messages and examines how this scales with group size, addressing both RQ2 and RQ3. First, we measure the size of `Welcome` and `Commit` messages as group size increases, and use these measurements to model the bandwidth requirements for different group operations.

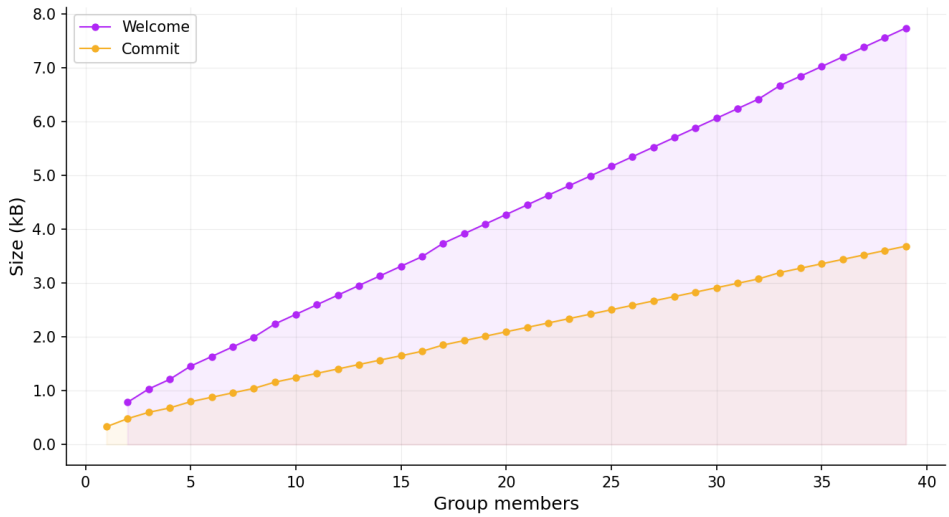


Figure 5.10: Size of `Welcome` and `Commit` messages as a function of group size.

Then we examine and propose optimisations that reduce the `Commit` message size, which makes DiMLS a more viable option in the UAV swarm scenario.

Handshake message size. Message sizes are measured using the virtualized environment described in Chapter 5.1, deploying one node at a time and recording the size of the resulting `Commit` and `Welcome` messages. As shown in Figure 5.10, both message types grow linearly with group size. The `Welcome` is larger than the `Commit`, because it contains the full ratchet tree, which the joining node needs to derive the epoch secret. The `Welcome` would look identical in comparison with regular MLS, while the `Commit` would grow linearly in the worst case, and logarithmically in the best case. Additionally DiMLS maintains N `SendGroups`, meaning the total bandwidth consumed by handshake messages grows quadratically.

To exemplify this we consider a 30-node network, where each node produces `Welcome` messages of approximately 6 kB. In the case of DiMLS this results in a total bandwidth of 180 kB across all groups. Similarly, `Commit` messages of approximately 3 kB each result in a total of 90 kB. In the case of regular MLS the corresponding bandwidth requirements would simply be 6 kB for the `Welcome` and 3 kB for the `Commit`.

Bandwidth modelling. The previous measurements allow us to model the bandwidth requirements for three key group operations. Below, we derive expressions for the bandwidth requirement regarding initialization of the swarm, adding a member to the swarm and removing a member from the swarm.

Group initialisation: For a swarm of size N, each node must transmit $N-1$ `KeyPackages`, producing $N(N-1)$ `KeyPackages` in total. Each node then commits all collected `KeyPackages` in a single operation, producing N `Welcome` messages. The total bandwidth, BW_{init}, required can be expressed as

$$BW_{\mathsf{init}} = N(N-1) \cdot \mathsf{keypackage}_{\mathsf{size}} + N \cdot \mathsf{welcome}_{\mathsf{size}}$$

Member addition: When a new node joins a swarm of N existing members, the joining node must transmit one `KeyPackage` to each existing member and each existing member must transmit one `KeyPackage` to the joining node, resulting in $2N$ `KeyPackages`. Every node, including the new member, produces one `Welcome`, and each existing node produces one `Commit`. The total bandwidth for the add operations, BW_{add} is given as

$$BW_{\mathsf{add}} = (N+1) \cdot \mathsf{welcome}_{\mathsf{size}} + N \cdot \mathsf{commit}_{\mathsf{size}} + 2N \cdot \mathsf{keypackage}_{\mathsf{size}}$$

Member removal: When one node is removed from a swarm of N members, each remaining node must produce one `Commit`, resulting in $N-1$ `Commits`. The total bandwidth, BW_{remove} is given as

$$BW_{\mathsf{remove}} = (N-1) \cdot \mathsf{commit}_{\mathsf{size}}$$

Figure 5.11 shows the total bandwidth requirements for each operation across increasing group sizes, illustrating the trade-off DiMLS makes. By providing resilience and decentralisation, the protocol overhead is significantly higher compared to standard MLS.

To quantify this trade-off, consider a node joining a swarm of 35 members. With DiMLS, assuming `KeyPackages` of 300 B and message sizes from Figure 5.10 the required bandwidth is

$$BW_{\mathsf{add}} = 36 \cdot 7\,\mathsf{kB} + 35 \cdot 3.4\,\mathsf{kB} + 70 \cdot 0.3\,\mathsf{kB} = 392\,\mathsf{kB}$$

Standard MLS requires only 1 `KeyPackage`, 1 `Commit`, and 1 `Welcome` for the same operation, totalling approximately 10.7 kB, roughly 37 times less overhead than DiMLS.

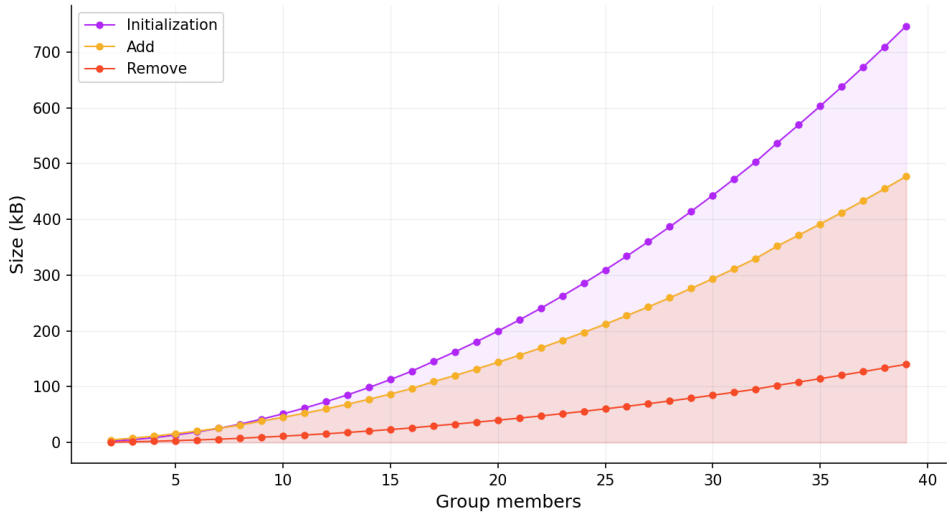


Figure 5.11: Total bandwidth requirements for initialisation, add, and remove operations in DiMLS as a function of group size.

Optimising commit size. While `Welcome` message sizes are fixed, `Commit` sizes can be reduced. In standard DiMLS operation, only the group owner performs path updates in their `SendGroup`, resulting in a flat tree structure, meaning each new group secret must be encrypted to every individual node. This eliminates one of the key efficiency advantages of MLS, which is the $O(\log N)$ complexity for providing PCS.

To illustrate this, we compare `Commit` sizes under two conditions. Standard DiMLS operation, and a modified initialisation procedure that permits an initial path update when a member is added to a `SendGroup`. While this violates the DiMLS specification [XLH25], by allowing group operations by a non `SendGroup` owner, it populates the internal nodes of the tree, enabling logarithmic scaling of subsequent `Commits`. The results of this test are shown in Figure 5.12, illustrating that an initial path update per member, causes `Commit` sizes to grow logarithmically rather than linearly. For large groups, this difference is substantial. Under logarithmic scaling, doubling the group size from 500 to 1000 members increases the `Commit` size by only one additional encryption, from 9 to 10, whereas linear scaling increases the number of encryptions from 499 to 999.

Since each node broadcasts a `Commit`, a swarm of N nodes generates N `Commits` per update cycle. Figure 5.13 shows the accumulated `Commit` size across the network as it grows. Total `Commit` bandwidth grows quadratically under standard DiMLS, while the optimised variant scales with $O(N \log N)$ complexity.

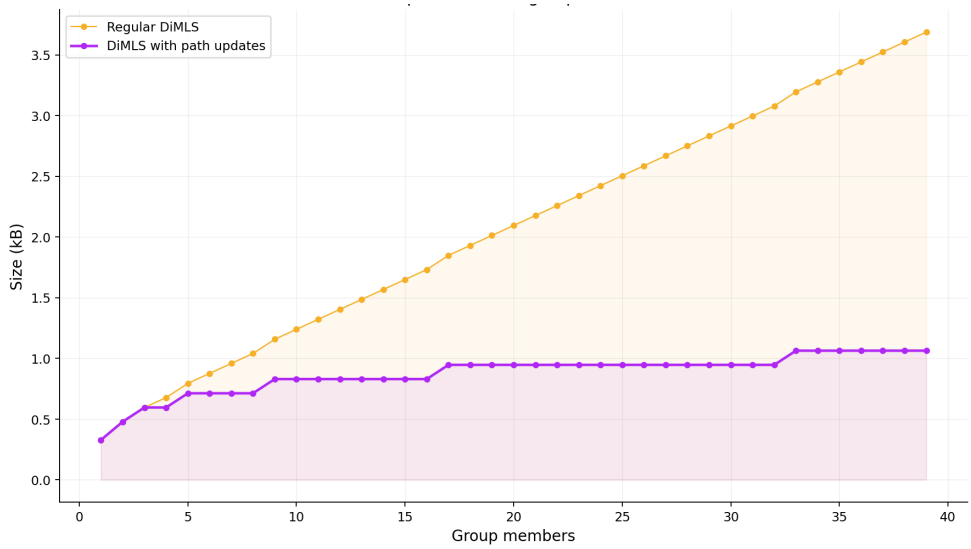


Figure 5.12: `Commit` size in standard DiMLS versus DiMLS with an initial path update upon joining.

The optimised initialisation does introduce additional upfront overhead, requiring each member to transmit $N-1$ extra `Commits`, for a total of $N(N-1)$ additional commits. However, if the operational context permits the swarm to form groups on the ground under stable network conditions, this one-time cost yields significantly smaller `Commits` during operation, freeing bandwidth for mission critical data such as video-streams and telemetry.

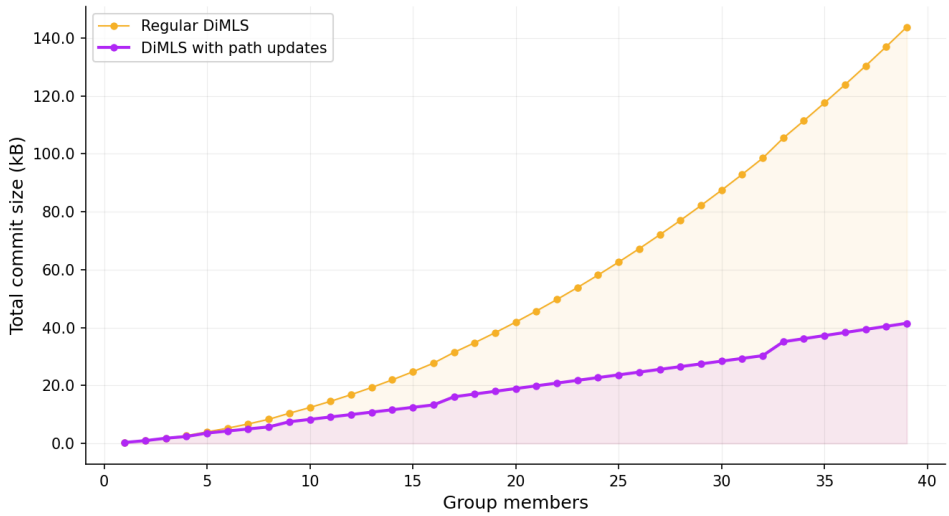


Figure 5.13: Total accumulated `Commit` size across the network: standard DiMLS versus DiMLS with an initial path update.

Chapter 6
Discussion

After implementing DiMLS and conducting extensive testing, simulation, and measurements, we will discuss the results in the context of the research questions. We begin by addressing each research question, to evaluate DiMLS in the UAV swarm context. Then we address the limitations of our project, in terms of methodology and knowledge. Finally we discuss whether a less complex protocol such as Sender Keys would be sufficient and why the additional complexity of DiMLS is justified despite the two approaches delivering broadly similar bandwidth characteristics.

6.1 Use of DiMLS in UAV swarms

While MLS is theoretically more efficient than DiMLS, fulfilling the total ordering requirement of the DS in a fully distributed setting has proved difficult in practice [BA25; DDH23]. The packet loss conditions typical of UAV swarm deployments lead to forked groups and loss of synchronisation that is costly to recover from. DiMLS eliminates this problem by design, at the cost of increased protocol overhead.

The most significant finding of our project is the resilience DiMLS shows against packet loss, as shown by Figure 5.4. MLS with Totem becomes non-operational at loss rates above 10% [BA25], while DiMLS maintains near complete message decryption at loss rates of up to 50%. This robustness comes at the cost of increased overhead, as the `SendGroup` architecture requires each membership operation to be replicated across all groups in the session. For large swarms this trade-off becomes difficult to justify, but for swarms of the scale of the Valkyrie system, with current node counts of up to 40 UAVs [HBSS24], the overhead remains within acceptable bounds.

During testing we identified two practical bottlenecks that limit scalability. The first is the mentioned bandwidth overhead introduced by the `SendGroup` architecture. The second is CPU consumption, which scales approximately linearly with message rate. This is not a consequence unique to DiMLS, but rather of the high packet rates

characteristic of UAV swarm operations, where message authentication and encryption impose a significant computational burden. Both bottlenecks are addressable through the optimisations discussed in the following sections, where we return to, and address the research questions.

> *RQ1: How can DiMLS be implemented to provide secure and resilient communication in UAV swarms?*

In this project we have demonstrated that DiMLS can be implemented using the `openmls` library, coupled with a lightweight retransmission mechanism to handle packet loss. The results show that this system provides stable and secure communications in lossy environments, at the cost of increased overhead.

The library chosen to implement the DiMLS logic is `openmls` and we consider it a suitable choice for several reasons. First, the library is fully compliant with the MLS RFC [BBR+23]. Second, `openmls` provides extensive documentation[1] and a well-defined Application Programming Interface (API), which reduced implementation complexity and made it straightforward to map DiMLS session operations onto the underlying MLS group primitives. Third, the library is written in Rust, a programming language that enforces memory safety at compile time, eliminating vulnerabilities such as buffer overflows, use-after-free errors, and data races that are common in C and C++ implementations. This is particularly relevant in a security-sensitive protocol implementation where memory corruption vulnerabilities could undermine the cryptographic guarantees of the protocol. Finally, Rust's zero cost abstractions mean these safety guarantees come without a runtime performance penalty, which is an important consideration given the constrained computational resources available on UAV platforms.

The modular middleware architecture makes integration with the existing Valkyrie system straightforward, as demonstrated during physical testing where end-to-end encryption of both telemetry and video streams was successfully achieved. Routing command traffic through the middleware would require unicast support, necessitating modifications to the Valkyrie source code to redirect unicast packets to our system, with an embedded destination address that our system can extract. While achievable, this was out of the scope of this project and not something we prioritized.

The primary strength of DiMLS is its resilience against packet loss. Each UAV independently controls the state of its own `SendGroup`, meaning dropped messages do not cause global group state divergence. Combined with the retransmission mechanism, this allows the swarm to recover from lost handshake messages without

[1] https://book.openmls.tech/

stalling global communication. Testing on both physical Flamingo UAVs and in the virtual environment demonstrates unambiguously that the system maintains stable cryptographic groups even under heavy packet loss.

One limitation observed during testing with larger swarms is that the retransmission mechanism is inefficient under conditions of widespread packet loss. If a UAV issues a `Commit` and the packet is lost for all other members simultaneously, every member will independently detect the missing message and issue a retransmission request, potentially saturating the available bandwidth. We propose two complementary mitigations. The first is a randomised back-off period before issuing a retransmission request, making it likely that at least one request is served before the others are sent, suppressing redundant requests without requiring coordination between nodes. The second is a cool-down period on the retransmitting node, where upon receiving a request the node serves it once and ignores subsequent requests for the duration of the window. Together these mechanisms address the problem from both sides, reducing both the number of requests generated and the number of retransmissions served.

> *RQ2: How does the use of DiMLS perform in terms of security, efficiency, and fault tolerance compared to MLS and Totem in UAV swarm scenarios?*

Our experiences and testing show that DiMLS outperforms MLS and Totem with respect to fault tolerance, and provides equivalent security guarantees. While DiMLS requires more overhead than MLS, we argue that it supports larger group sizes in the swarm context. The following paragraphs discuss each of these claims in depth as we compare the fault tolerance, security and efficiency of DiMLS with MLS and Totem.

Fault Tolerance Our testing shows that DiMLS is significantly more robust than MLS with Totem under constrained network conditions. In small groups, DiMLS is almost unaffected by packet loss and recovers quickly when handshake messages are dropped, while MLS with Totem becomes non-operational at loss rates above 10%. This is shown in Figure 5.3 and Figure 5.4. Two properties of DiMLS contribute to this resilience. First, the `SendGroup` concept eliminates the global total ordering requirement for protocol messages, meaning packet loss does not lead to unsynchronized group states. The members continue operating independently and the affected `SendGroups` are simply in an older state, and not forked as in MLS with Totem. Second, the retransmission mechanism detects dropped messages and requests only the missing data, which enables the `SendGroups` to regain correct state.

This solution stands in contrast to Totem, which requires global convergence before communication can resume. In scenarios where a UAV is at the edge of radio coverage,

Totem struggles to converge and blocks all communication until the group stabilises. The retransmission mechanism used in DiMLS is non-blocking, meaning a single unreachable node does not affect the rest of the swarm. While the mechanism is simple, it provides the fault tolerance DiMLS requires in a lossy network.

A further resilience property of DiMLS is its resistance to insider denial-of-service attacks. In regular MLS, a compromised member has commit authority over the shared group state, meaning an adversary could deliberately desynchronize the group and deny service to all members. In DiMLS, commit authority is scoped to the owner's `SendGroup`. A compromised UAV can only disrupt its own outgoing communications, leaving all other `SendGroup`s intact and operational.

Security The current version of the `openmls` API does not support the `LeafNodeUpdate` operation, which prevents a fully compliant DiMLS implementation. The operation would allow a member to overwrite another member's `Leafnode`, giving any insider the ability to silently replace other group members' keys. This is a significant reason why support for this operation is unlikely to be added to `openmls` in the future.

Despite this, we argue that the operation is in most cases unnecessary in the UAV swarm setting. If a UAV is compromised, it has most likely crashed or fallen into adversarial hands, which would trigger its removal from the swarm. For an adversary to obtain leaf-node key material, they would need memory access to the UAV, either physically or remotely via an exploit. The former is addressed by the remove operation, if the UAV has some tamper detection mechanism that is triggered, while the latter implies the UAV is already compromised through other channels, for which only a remove operation can heal the group.

There are, however, scenarios where `LeafNodeUpdate` would matter. First, if an adversary captures a UAV, clones it without the tamper detection mechanism being triggered, and releases it, they could eavesdrop on swarm communication indefinitely, since without `LeafNodeUpdate` the `SendGroups` cannot heal from this compromise. Second, if a UAV has been offline for a period and its key material should be refreshed upon return, a `SelfUpdate` only provides PCS in that UAV's own `SendGroup`. An adversary who tampered with the UAV while offline could still listen in on the remaining `SendGroups`.

A temporary mitigation is however available, as the other UAVs can perform a PCS update as described in Section 4.2.3, injecting entropy the adversary cannot obtain, thereby providing PCS within the epoch. This is not a permanent fix however, since the leaf-node itself is not updated. This means the adversary regains access to the group after the next path update. Permanent PCS requires fully removing and re-adding the compromised leaf-node from the `SendGroups`, which is significantly

more bandwidth-intensive than a `LeafNodeUpdate` would have been. This presents a disadvantage to regular MLS, where PCS is provided simply by the compromised node performing a path update.

However, considering the threat model of a UAV swarm, realistic compromise scenarios lead to removal regardless, and the two edge cases described can be handled through existing operations. The absence of `LeafNodeUpdate` therefore affects the security of the DiMLS implementation in UAV swarms in only a minimal way, but it does make our implementation less efficient.

Another security consideration of DiMLS is the handshake cache and how the storage of old handshake messages affects Forward Secrecy (FS). If we consider regular MLS, then the DS is responsible for storing and distributing handshake messages in the correct order. The DS is untrusted by design, meaning a compromised DS will not disclose any private keys, message contents or the group secret, despite being able to retrieve all handshake messages. We consider the handshake cache in DiMLS as the local DS for its `SendGroup`. This means that an adversary can query the handshake cache for all traffic messages, but not be able to obtain any information about message contents or group secret, without any private keys. If a private key is leaked, the adversary could decrypt the leaf-node secrets encrypted to that key and derive the epoch root secret. This is however bounded to the epochs where that key was valid, and this is equally true in standard MLS if private keys were to be leaked. Therefore, buffering handshake messages for retransmission introduces no additional risks to FS beyond that of regular MLS.

Efficiency When discussing the efficiency of DiMLS, one could consider the number of cryptographic operations required to provide FS and PCS, the CPU consumption, or the bandwidth and overhead added by the protocol. In this project we have measured CPU usage and bandwidth requirements, finding similar CPU consumption between DiMLS and MLS, but significantly higher bandwidth usage for DiMLS.

As for CPU consumption, we conclude that the message rate is the determining factor. This is supported both by the results in Table 5.2 and by the increase in CPU usage as swarm size grows, as seen in Figures 5.7 and 5.8. The latter observation follows from the fact that a larger swarm produces a higher aggregate message rate, as each additional node contributes to the total volume of messages to be encrypted and decrypted. This corresponds well with the findings of Bragstad and Arder [BA25]. As shown by the flamegraph in Figure 5.9, `openmls` accounts for the majority of CPU consumption, while the overhead imposed by our DiMLS implementation is negligible in comparison. Furthermore, a fair comparison must account for the fact that Totem implementations, such as Corosync[2] used by Bragstad and Arder, introduce their

[2]https://corosync.github.io/corosync/

own CPU overhead, which DiMLS is not reliant upon. We therefore argue that the CPU efficiency of DiMLS is comparable to, and arguably competitive with, that of MLS with Totem. Concerning Random Access Memory (RAM), no measurements were conducted in this project, as Bragstad and Arder [BA25] found consistently low RAM consumption in their MLS implementation. It should be noted, however, that this finding does not straightforwardly transfer to DiMLS, as the RAM consumption of DiMLS scales quadratically with group size. This is because each node in the swarm must store a full copy of the ratchet tree for every `SendGroup` it participates in, resulting in $O(N^2)$ memory usage, for N members of the swarm.

Figure 5.11 shows the bandwidth requirements for the swarm, for different group operations in DiMLS. We found that all operations scale quadratically with the swarm size for DiMLS, while MLS scales linearly in the worst case and logarithmically in the best case. This means that DiMLS requires orders of magnitude more bandwidth resources than MLS which is a clear disadvantage of the protocol. However, when comparing the bandwidth utilization of DiMLS to MLS with Totem, we must also consider how well each protocol holds up under the network conditions typical of UAV swarm deployments, rather than evaluating theoretical scaling alone. While DiMLS scales worse than MLS in terms of bandwidth and maximum group size, the theoretical advantage of MLS with Totem does not translate to practice. Bragstad and Arder [BA25] show that Totem struggles to converge when a UAV operates at the edge of radio coverage, leading to a global lock in communications, and that packet loss leads to forking, which is costly to recover. In contrast, DiMLS tolerates these conditions without stalling and eliminates the risk of forking. The result is that despite its worse theoretical scaling, DiMLS supports larger group sizes and more consistent communication throughput in practice, in the constrained and dynamic network environments where UAV swarms operate.

In summary, DiMLS outperforms MLS with Totem in fault tolerance and delivers comparable security and efficiency for UAV swarm scenarios. The `SendGroup` concept eliminates the global total ordering requirement that makes MLS with Totem brittle under packet loss, and the retransmission mechanism provides recovery without the convergence problems that limit Totem in dynamic network conditions. The security properties are equivalent to standard MLS in relevant threat scenarios, with the absence of `LeafNodeUpdate` having minimal impact given that realistic compromise scenarios lead to node removal regardless. While DiMLS scales worse than MLS in theory, this disadvantage is offset in practice by its ability to maintain stable operation under the constrained network conditions typical of UAV deployments, where the theoretical efficiency of MLS with Totem cannot be realised.

RQ3: How does DiMLS scale as the network grows, regarding bandwidth- and CPU-usage?

We have demonstrated that DiMLS is viable for swarms of small and medium sizes. However, as the swarm grows, both bandwidth requirements and CPU consumption increase substantially. Our results show that bandwidth for group operations scales quadratically in DiMLS, compared to logarithmic scaling in regular MLS. The root cause is that every group operation must be performed in all `SendGroups`, and in the case of adding a UAV, each must distribute a `Welcome` message containing the full ratchet tree. This cost grows with both the number of groups and the group size simultaneously.

To mitigate the commit overhead, we suggest that a DiMLS deployment should include an initial setup phase in which each UAV, upon joining, performs a path update to populate the intermediate nodes of the ratchet tree. Without this step, all intermediate nodes along non-owner paths remain blank, and the tree provides no logarithmic scaling benefit. With it, individual `SendGroup` commits scale logarithmically, reducing the aggregate commit overhead across the session from $O(N^2)$ to $O(N \log N)$, approximately linear for small node counts. Welcome messages remain quadratic, as the full tree must still be transmitted to new joiners, but in practice membership changes are infrequent relative to application traffic, making this a more acceptable cost. While linear scaling is worse than regular MLS, this must be considered in the context of UAV swarms. MLS is designed for groups of potentially thousands of members, whereas Valkyrie is tested (virtually) with up to 40 nodes [HBSS24], which is a scale where these costs remain manageable.

To potentially reduce the overhead of `Welcome` messages, we propose adopting the MLS light client extension [KBB+25]. The extension was designed to reduce bandwidth requirements for asymmetric group settings where most members are receive-only, such as large webinars. In standard MLS, a `Welcome` message contains the full ratchet tree, causing its size to grow linearly with group size. The light client extension reduces this to $O(\log(N))$ by transmitting only the subset of the tree necessary for the joining member to derive the epoch secret, rather than the full tree. As a consequence, light clients cannot perform group operations, but they remain full group members and can derive new epoch secrets and decrypt application messages normally. This constraint aligns naturally with the DiMLS model, where only the `SendGroup` owner is permitted to perform group operations regardless. Adding non-owner UAVs as light clients rather than full clients would therefore reduce `Welcome` message sizes without violating any DiMLS properties. At the time of writing, this extension remains a draft and has not yet been standardised, but its properties align well with the DiMLS model.

Concerning CPU scaling, which grows linearly with message rate, we propose that DiMLS is to be employed as a key exchange module rather than as a full message protection layer. Under this configuration, DiMLS is used to derive short-lived symmetric keys, which are then used to encrypt high-rate telemetry and video streams directly. This avoids the most CPU-intensive operations in MLS, particularly signature verification on every incoming packet and deriving keys from the ratchet. Rochford et al. [RBMT26] demonstrated a 20x performance improvement when using MLS in a key-exchange configuration compared to full MLS message protection. The trade-off is that individual application messages no longer carry a signature, meaning a receiving UAV cannot attribute or authenticate a given packet to a specific sender. However, given the constrained compute resources available on UAV platforms and the high packet rates characteristic of telemetry and video streaming, the cost of signature verification on every packet is difficult to justify. Confidentiality and group membership integrity are both preserved under this configuration, and these are the properties most critical to the operational context. While packet authenticity is desirable, it is a reasonable trade off for the massive increase in CPU performance.

To summarize, DiMLS scales worse than regular MLS, and is resource demanding in its original specification, in terms of both bandwidth and compute. To improve the scalability, we recommend a small deviation from the DiMLS draft. By allowing non-owners to perform an initial path update during the setup phase, the bandwidth for commits is reduced from quadratic to approximately linear scaling across the session. Additionally, by running DiMLS as a key exchange module rather than as a full message protection layer, we can significantly reduce the CPU load, enabling support for larger swarms. With these adjustments, DiMLS becomes a viable security solution for swarms of current sizes.

6.2 Limitations

This section discusses the main limitations of our project, including limited testing on the target hardware, limited domain knowledge of UAV swarms, and the absence of the `LeafnodeUpdate` feature.

First, we conducted only limited testing on the Flamingo UAVs themselves. As a result, the performance measurements presented in this work should be considered indicative rather than definitive. Furthermore, our network robustness testing was performed using generated data in a simulated environment and focused solely on packet loss. In an operational deployment, the data rate would naturally fluctuate, and network conditions would also include jitter and delay in addition to packet loss. Since we did not evaluate the system under such conditions, we cannot draw conclusions about its performance in the presence of jitter and delay.

Second, the implementation was carried out without direct collaboration with the Valkyrie team at FFI. This represents a limitation, as our understanding of the UAV swarm domain is limited and the implementation was therefore not tailored to the specific needs of the swarm. Consequently, the system was designed based on general assumptions about decentralized and embedded systems. While this provides flexibility, the system may lack features or functionality required in a tactical context with specialized operational requirements.

Finally, we were unable to implement the `LeafnodeUpdate` feature due to limitations in the `openmls` library. Although it would have been possible to fork the library and add the necessary functionality, we chose not to prioritize this work because of time constraints and the complexity involved. Supporting `LeafnodeUpdate` would require modifications to the `TreeKEM` protocol implementation and changes to how `openmls` validates the ratchet tree. The absence of this feature means that our implementation is not fully compliant with the DiMLS draft [XLH25].

6.3 Comparing DiMLS with Sender Keys

The quadratic scaling of bandwidth consumption in DiMLS raises the question of whether simpler protocols might be more suitable for UAV swarm communications. A relevant protocol of comparison is Sender Keys, where each group member distributes a symmetric key to all other members and uses it to encrypt outgoing messages. On the surface, DiMLS operates similarly, where each member owns a `SendGroup` with a single root key, and the other members are receive-only. The setup cost is $O(N^2)$ in both cases [BCG23], and without the initial path update described earlier, the standard DiMLS configuration leaves internal tree nodes blank, resulting in $O(N^2)$ PCS update costs that match Sender Keys. From a pure scaling perspective, DiMLS can therefore be viewed as a more complex version of Sender Keys where the efficiency gains of the binary tree structure are not realised. This raises the question of whether the added complexity of DiMLS is justified. To provide an argument for DiMLS over Sender Keys, we have to consider other properties than bandwidth efficiency that DiMLS provides.

The first argument is standardisation. MLS is a published standard [BBR+23] that has undergone extensive formal analysis and peer review. Sender Keys, as used for example in Signal's group messaging, is not standardised and lacks the same depth of cryptographic scrutiny [BCG23]. For a security-critical application such as UAV swarm communications, building on a well-analysed standard reduces the risk of subtle protocol-level vulnerabilities.

The second argument is the MLS `GroupContext`, which cryptographically binds the current epoch secret to a specific membership list. This means that for any given

epoch key, there is a verifiable record of exactly which members were authorised to hold it at the time it was derived. In PCS scenarios, this is useful because a key rotation can be tied to a membership state. After a `Remove` operation, the new epoch secret is bound to a list of members that does not include the removed member, providing a verifiable guarantee of eviction. Sender Keys provides no equivalent binding between the distributed key and the membership list, meaning there is no cryptographic basis for determining which members a given key is associated with or verifying that a removed member no longer has access.

The third argument is the PCS update cost with the path update optimisation described earlier. Without the initial path update, both DiMLS and Sender Keys require $O(N^2)$ work for a full PCS update across the group. However, with the path update applied during the setup phase, individual `SendGroup` commits scale logarithmically, reducing the aggregate PCS update cost to $O(N \log N)$. Sender Keys has no equivalent optimization as providing PCS requires redistributing the new key to all N members individually, with no tree structure to exploit. DiMLS with the path update optimisation provides a concrete efficiency advantage over Sender Keys despite the initial similarity between the two approaches.

In summary, DiMLS can be viewed as a more capable version of Sender Keys that trades implementation complexity for stronger security guarantees and potentially a more efficient PCS update path. Whether this trade-off is justified depends on the threat model and operational requirements, but for a UAV swarm operating in a contested environment where PCS recovery is an important aspect, DiMLS provides stronger PCS guarantees and verifiable membership binding, and is the preferred alternative of the two.

Chapter 7
Conclusion

This project has implemented and evaluated Distributed Messaging Layer Security (DiMLS) for securing communications in a UAV swarm. We have demonstrated that `openmls` can be used to implement the core functionality required by the DiMLS draft [XLH25], and that the resulting system integrates with the Valkyrie swarm software with relative ease. The one exception is the `LeafNodeUpdate` operation, which is incompatible with the `openmls` API due to fundamental constraints in the MLS trust model. We have argued that this limitation has minimal practical impact within the UAV swarm threat model, where realistic compromise scenarios result in the affected UAV being removed from the swarm regardless.

The primary finding of this work is that DiMLS, combined with a simple retransmit mechanism, provides substantially better resilience than MLS with Totem under the constrained network conditions typical of UAV deployments. Our results show a packet decryption rate approaching 100% at packet loss rates of up to 50%, compared to decryption rates falling below 50% at just 10% packet loss in previous efforts using MLS with Totem [BA25]. This improvement comes from two properties of DiMLS. First, the elimination of the total ordering requirement through the `SendGroup` concept, and second, the non-blocking retransmission mechanism that allows the swarm to recover from dropped handshake messages without stalling global communication. We have demonstrated stable operations with groups of up to 40 nodes in the virtualized environment.

The main limitation of DiMLS is its bandwidth scaling, which grows quadratically with group size due to the fan-out of membership operations across N `SendGroup`s. To address this, we recommend an initial path update during the setup phase to reduce the commit overhead, and propose running DiMLS as a key exchange module rather than a full message encryption layer. The latter significantly reduces CPU consumption by offloading high rate telemetry and video encryption to a symmetric cipher, consistent with findings by Rochford et al. [RBMT26] who demonstrated a 20x performance improvement under this configuration. The MLS light client extension

should also be explored and integrated with DiMLS. With these optimizations, DiMLS represents a viable and standards-based approach to securing communications in UAV swarms, offering stronger resilience than previous efforts, and formal security guarantees compared to simpler alternatives such as Sender Keys.

7.1 Future work

This project delivers a PoC implementation of DiMLS, and several directions for future work have been identified before it could be a system ready for production. To achieve full compliance with the DiMLS draft [XLH25], `openmls` should be forked and extended to support the `LeafNodeUpdate` operation. As discussed, this operation is not compatible with the current `openmls` API, but is necessary for complete PCS healing in scenarios where a node's state is compromised without being removed from the group.

In addition, the current implementation does not perform any credential validation, meaning anyone can construct a valid `KeyPackage` and join the swarm undetected. Replacing `Basic` credentials with a certificate-backed scheme, such as the `Ed25519Credential` proposed by Bragstad and Arder [BA25], would close this gap and should be investigated further.

For future proofing, the suitability of post-quantum cryptographic algorithms within DiMLS should also be explored. The IETF is actively working on post-quantum variants of the cryptographic primitives used in MLS[1], and evaluating their performance and feasibility in the resource-constrained UAV swarm context is a relevant direction for future work.

To address the quadratic bandwidth scaling, future work should explore combining DiMLS with the MLS Light Client draft [KBB+25]. Light clients are prohibited from performing group operations and do not require the full ratchet tree, which reduces the size of `Welcome` messages and lowers the processing requirements for receive-only nodes. This fits naturally with the `SendGroup` concept, where non-owner members are already receive-only. The initial path update operation during the setup phase should also be implemented, as the reduction in commit overhead from $O(N^2)$ to $O(N \log N)$ has only been demonstrated, but not thoroughly tested or integrated into the final DiMLS system.

The retransmit mechanism performs well in small groups but requires refinement as group size increases. Specifically, the randomised back-off and sender cool-down mitigations described in Section 6.1 should be implemented and evaluated to determine their effectiveness in suppressing retransmission storms under realistic packet loss

[1] https://messaginglayersecurity.rocks/mls-pq-ciphersuites/draft-ietf-mls-pq-ciphersuites.html

conditions. Finally, running DiMLS in the key exchange configuration rather than as a full per-message encryption layer should be tested empirically to quantify the performance gains in the UAV swarm context, building on the theoretical arguments presented in this work.

References

[ACJM20] J. Alwen, S. Coretti, et al., "Continuous Group Key Agreement with Active Security", en, in *Theory of Cryptography*, R. Pass and K. Pietrzak, Eds., vol. 12551, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 261–290. last visited: May 29, 2026. [Online]. Available: https://link.springer.com/10.1007/978-3-030-64378-2_10.

[AMM+95] Y. Amir, L. E. Moser, et al., "The Totem single-ring ordering and membership protocol", en, *ACM Transactions on Computer Systems*, vol. 13, no. 4, pp. 311–342, Nov. 1995. last visited: Oct. 24, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/210223.210224.

[AMT23] J. Alwen, M. Mularczyk, and Y. Tselekounis, *Fork-Resilient Continuous Group Key Agreement*, Publication info: A major revision of an IACR publication in CRYPTO 2023, 2023. last visited: Sep. 17, 2025. [Online]. Available: https://eprint.iacr.org/2023/394.

[BA25] E. Bragstad and M. Arder, "Improving Messaging Layer Security in a Military UAV Swarm", en, Jun. 2025. last visited: May 29, 2026. [Online]. Available: https://nva.sikt.no/registration/019bbb4f0a2d-3b350d3e-76c0-4581-8af5-ea6ae58f650f.

[BBR+23] R. Barnes, B. Beurdouche, et al., "The Messaging Layer Security (MLS) Protocol", Internet Engineering Task Force, Request for Comments RFC 9420, Jul. 2023, Num Pages: 132. last visited: Oct. 7, 2025. [Online]. Available: https://datatracker.ietf.org/doc/rfc9420.

[BBR18] K. Bhargavan, R. Barnes, and E. Rescorla, "TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups", en, Mar. 2018. [Online]. Available: https://inria.hal.science/hal-02425247/file/treekem+%281%29.pdf.

[BCG23] D. Balbás, D. Collins, and P. Gajland, *WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs*, Publication info: A major revision of an IACR publication in ASIACRYPT 2023, 2023. last visited: May 2, 2026. [Online]. Available: https://eprint.iacr.org/2023/1385.

[BRO+25] B. Beurdouche, E. Rescorla, et al., "The Messaging Layer Security (MLS) Architecture", Internet Engineering Task Force, Request for Comments RFC 9750, Apr. 2025, Num Pages: 41. last visited: May 12, 2026. [Online]. Available: https://datatracker.ietf.org/doc/rfc9750.

[BV25] S. Bugge and H. Vågenes (foto), *Norge oppretter landdrone-program: Skal bruke 1,5 milliarder*, no, Section: Nyheter, Oct. 2025. last visited: May 7, 2026. [Online]. Available: https://www.vg.no/nyheter/i/5EPRK6/norge-oppretter-landdrone-program.

[CCG+18] K. Cohn-Gordon, C. Cremers, et al., *On Ends-to-Ends Encryption*, en, 2018. last visited: May 29, 2026. [Online]. Available: https://dl.acm.org/doi/epdf/10.1145/3243734.3243747.

[DDH23] E. Dietz, D. Davis, and B. Hale, "Utilizing the Messaging Layer Security Protocol in a Lossy Communications Aerial Swarm", en, 2023. last visited: Sep. 19, 2025. [Online]. Available: http://hdl.handle.net/10125/103431.

[Die22] E. Dietz, *Utilizing the Messaging Layer Security Protocol in a Lossy Communications Aerial Swarm*, Mar. 2022. last visited: Mar. 5, 2026. [Online]. Available: https://apps.dtic.mil/sti/citations/trecms/AD1173274.

[ECS05] D. E. Eastlake 3rd, S. Crocker, and J. I. Schiller, "Randomness Requirements for Security", Internet Engineering Task Force, Request for Comments RFC 4086, Jun. 2005, Num Pages: 48. last visited: Apr. 13, 2026. [Online]. Available: https://datatracker.ietf.org/doc/rfc4086.

[FFI25] FFI, *Droner - forskning ved Forsvarets forskningsinstitutt*, no, Oct. 2025. last visited: May 7, 2026. [Online]. Available: https://www.ffi.no/forskning/tema/droner.

[For25] Forsvarsdepartementet, *Dronestrategi for forsvarssektoren*, nb-NO, Plan, Publisher: regjeringen.no, Dec. 2025. last visited: May 7, 2026. [Online]. Available: https://www.regjeringen.no/no/dokumenter/dronestrategi-for-forsvarssektoren2/id3141461/.

[God26] S. Godard, *Pidstat(1): Report statistics for tasks - Linux man page*, May 2026. last visited: May 19, 2026. [Online]. Available: https://linux.die.net/man/1/pidstat.

[HBSS24] M. Halsør, D. H. Bentsen, et al., "Using drone swarms with manoeuvre units", Tech. Rep., 2024. last visited: Mar. 5, 2026. [Online]. Available: https://www.ffi.no/en/publications-archive/using-drone-swarms-with-manoeuvre-units.

[ipr26] iproute2, *Tc(8) - Linux manual page*, May 2026. last visited: May 19, 2026. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html.

[IS21] J. Iyengar and I. Swett, "QUIC Loss Detection and Congestion Control", Internet Engineering Task Force, Request for Comments RFC 9002, May 2021, Num Pages: 42. last visited: Oct. 24, 2025. [Online]. Available: https://datatracker.ietf.org/doc/rfc9002.

[KBB+25] F. Kiefer, K. Bhargavan, et al., "Light MLS", Internet Engineering Task Force, Internet Draft draft-kiefer-mls-light-02, Mar. 2025, Num Pages: 19. last visited: May 22, 2026. [Online]. Available: https://datatracker.ietf.org/doc/draft-kiefer-mls-light-02.

[KNC26] S. Klabnik, C. Nichols, and K. Crycho, *The Rust Programming Language - The Rust Programming Language*, May 2026. last visited: Oct. 7, 2025. [Online]. Available: https://doc.rust-lang.org/book/title-page.html.

[Koh25] K. Kohbrok, "Decentralized Messaging Layer Security", Internet Engineering Task Force, Internet Draft draft-kohbrok-mls-dmls-03, Oct. 2025, Num Pages: 6. last visited: Nov. 6, 2025. [Online]. Available: https://datatracker.ietf.org/doc/draft-kohbrok-mls-dmls-03.

[Kun23] D. Kunertova, "The war in Ukraine shows the game-changing effect of drones depends on the game", *Bulletin of the Atomic Scientists*, vol. 79, no. 2, pp. 95–102, Mar. 2023. last visited: Mar. 5, 2026. [Online]. Available: https://doi.org/10.1080/00963402.2023.2178180.

[Lab26] U. N. R. Laboratory, *Multi-Generator Network*, May 2026. last visited: May 19, 2026. [Online]. Available: https://www.nrl.navy.mil/Our-Work/Areas-of-Research/Information-Technology/NCS/MGEN/.

[LB22] A. Leon and C. J. Britt, "UXS Authentication and Key Exchange Requirements for Multidomain Operation and Joint Interoperability", en, 2022. [Online]. Available: https://apps.dtic.mil/sti/trecms/pdf/AD1185016.pdf.

[Mar16] M. Marlinspike, *Safety number updates*, en, Nov. 2016. last visited: May 13, 2026. [Online]. Available: https://signal.org/blog/safety-number-updates/.

[Mar23] E. Marstrander, "Use of Messaging Layer Security in a Military UAV Swarm", eng, Accepted: 2024-02-20T18:19:43Z, M.S. thesis, NTNU, 2023. last visited: Sep. 29, 2025. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3118795.

[MSC26] A. Madhavapeddy, D. J. Scott, and J. Cormack, "A Decade of Docker Containers", *Commun. ACM*, vol. 69, no. 3, pp. 56–65, Feb. 2026. last visited: May 19, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3761803.

[Num21] O. R. Nummedal, *Flamingo - a UAV for autonomy research*, en, 2021. [Online]. Available: https://www.ffi.no/publikasjoner/arkiv/flamingo-a-uav-for-autonomy-research.

[NVI26] NVIDIA, *The World's Smallest AI Supercomputer*, en-us, Mar. 2026. last visited: Mar. 6, 2026. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/.

[RBMT26] P. Rochford, W. J. Buchanan, et al., "Securely Scaling Autonomy: The Role of Cryptography in Future Unmanned Aircraft Systems (UASs)", en, *Cryptography*, vol. 10, no. 2, p. 20, Apr. 2026, Publisher: Multidisciplinary Digital Publishing Institute. last visited: Apr. 20, 2026. [Online]. Available: https://www.mdpi.com/2410-387X/10/2/20.

[Sch14] P. Scharre, "Robotics on the Battlefield Part II", en, Oct. 2014. [Online]. Available: https://www.files.ethz.ch/isn/184587/CNAS_TheComingSwarm_Scharre.pdf.

[Tas25] S. Tassopoulos, “Delivery service comparison for secure group protocols”, en, Mar. 2025. [Online]. Available: https://calhoun.nps.edu/server/api/core/bitstreams/3cd333fd-86d8-450f-831c-e77aa23e2e26/content.

[Uni25] United Nations, *Lethal Autonomous Weapon Systems | United Nations Office for Disarmament Affairs*, Shorthand: UN25, Sep. 2025. last visited: Jun. 9, 2026. [Online]. Available: https://disarmament.unoda.org/index.php/en/our-work/emerging-challenges/lethal-autonomous-weapon-systems.

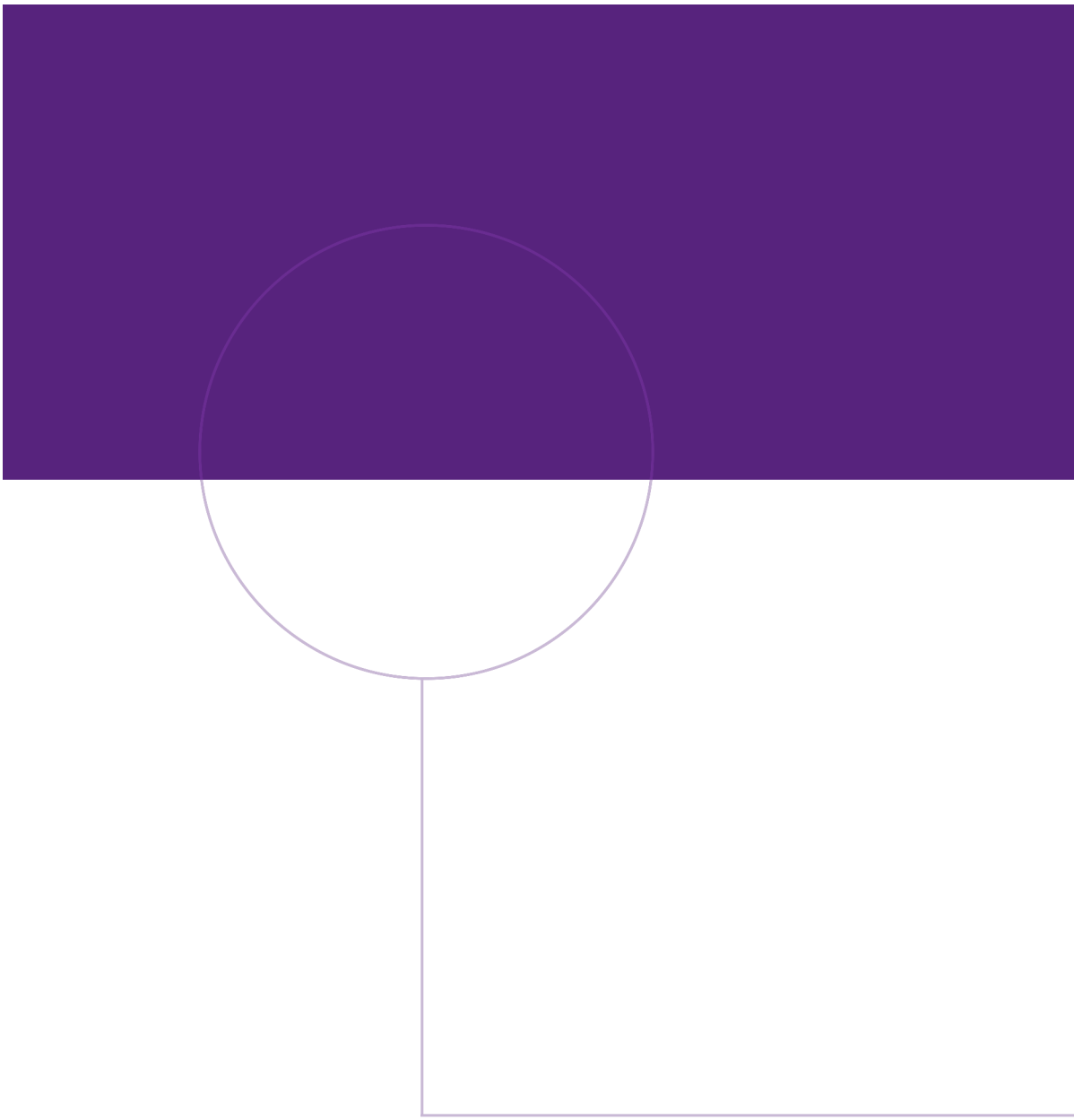
[WPB25] T. Wallez, J. Protzenko, and K. Bhargavan, “TreeKEM: A Modular Machine-Checked Symbolic Security Analysis of Group Key Agreement in Messaging Layer Security”, in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 4375–4390. [Online]. Available: https://ieeexplore.ieee.org/document/11023415.

[WPBB23] T. Wallez, J. Protzenko, et al., “TreeSync: Authenticated Group Management for Messaging Layer Security”, in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 1217–1233. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/wallez.

[XLH25] M. Xue, J. W. Lukefahr, and B. Hale, “Distributed MLS”, Internet Engineering Task Force, Internet Draft draft-xue-distributed-mls-02, Oct. 2025, Num Pages: 10. last visited: Nov. 5, 2025. [Online]. Available: https://datatracker.ietf.org/doc/draft-xue-distributed-mls.

[Zen19] M. Zensius, “Jetson Nano Developer Kit”, en, Dec. 2019. [Online]. Available: https://developer.download.nvidia.com/embedded/L4T/r32-3-1_Release_v1.0/Jetson_Nano_Developer_Kit_User_Guide.pdf.

[Zie21] T. Zieliński, “Factors Determining a Drone Swarm Employment in Military Operations”, en, *Safety & Defense*, vol. 7, no. 1, pp. 59–71, May 2021. last visited: Mar. 4, 2026. [Online]. Available: https://sd-magazine.eu/index.php/sd/article/view/112.



NTNU

Norwegian University of
Science and Technology