Master's thesis

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

Emil Bragstad
Marius Arder

Improving Messaging Layer Security in a Military UAV Swarm

Master's thesis in Cyber Security and Data Communication
Supervisor: Tjerand Silde
Co-supervisor: Martin Strand
June 2025

NTNU
Norwegian University of
Science and Technology

Emil Bragstad
Marius Arder

Improving Messaging Layer Security in a Military UAV Swarm

Master's thesis in Cyber Security and Data Communication
Supervisor: Tjerand Silde
Co-supervisor: Martin Strand
June 2025

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

Unmanned Aerial Vehicles (UAVs), and especially UAV swarms, are playing an increasingly important role in modern military operations. In such dynamic environments, secure and reliable communication between entities is desirable. A major challenge in securing group communication is efficiently handling frequent and unpredictable changes in group membership, which is inherent to military UAV swarms. Nodes may join or leave continuously, requiring the swarm to update its group key dynamically and securely.

Prior research has identified the Messaging Layer Security (MLS) protocol as a promising solution. MLS is a key management protocol designed for efficient, asynchronous, and scalable group key establishment, supporting groups from a few to thousands of members.

A recent master project at Norwegian University of Science and Technology (NTNU) successfully implemented MLS in a drone swarm, demonstrating airborne key updates every 10 minutes. The project introduced a decentralized delivery service that ensured message ordering despite unstable connectivity and node mobility.

This project aims to advance the use of MLS in military UAV swarms, focusing on continuous key distribution for secure group communication. The research will optimize MLS for resource-constrained UAVs and evaluate its performance and security under realistic operational conditions. Through simulation and analysis, the project seeks to enhance the efficiency, resilience, and adaptability of the protocol to military swarm environments.

Approved: 2025-02-19 – Tjerand Silde, NTNU (Main supervisor)

Abstract

In military UAV swarms, secure group communication is increasingly important, yet providing strong security guarantees in dynamic, distributed networks remains a challenge. Messaging Layer Security (MLS), which was recently standardized, offers scalable end-to-end encryption with strong security guarantees for group messaging. In this thesis, we present Valkyrie MLS, a Rust-based implementation of the MLS protocol integrated into FFI's Valkyrie UAV swarm platform. Our system secures inter-drone communication by establishing a shared cryptographic group state for authenticated, confidential messaging.

We extend our system with a reliable Delivery Service that ensures total order of broadcast messages, and a proof-of-concept Authentication Service for credential verification. To assess performance and security, we conduct both simulated and physical drone tests. The results show that Valkyrie MLS strengthens messaging security in UAV swarms while maintaining acceptable performance, although certain challenges remain.

Sammendrag

I militære UAV svermer blir sikker gruppekommunikasjon stadig viktigere, men det er utfordrende å sikre sterke sikkerhetsgarantier i dynamiske, distribuerte nettverk. Messaging Layer Security (MLS), som nylig har blitt standardisert, tilbyr skalerbar ende-til-ende-kryptering med sterke sikkerhetsgarantier for kommunikasjon i gruppemeldinger. I denne oppgaven presenterer vi Valkyrie MLS, en Rust-basert implementasjon av MLS-protokollen integrert med FFI sin UAV-svermplattform Valkyrie. Systemet vårt sikrer kommunikasjon mellom droner ved å etablere en delt kryptografisk gruppetilstand for autentisert og konfidensiell meldingsutveksling.

Vi utvider systemet med pålitelig levering basert på totalordnet kringkasting, sammen med et konseptbevis på en enkel autentiseringstjeneste for verifisering av legitimasjon. For å evaluere ytelse og sikkerhet gjennomfører vi både simulerte og fysiske dronetester. Resultatene viser at Valkyrie MLS styrker meldingssikkerheten i UAV-svermer samtidig som ytelsen forblir akseptabel, selv om visse utfordringer gjenstår.

Acknowledgements

First and foremost, we would like to thank our supervisors, Tjerand Silde and Martin Strand, for their guidance and support. Special thanks to Martin for generously providing access to equipment and facilities at Kjeller.

We are also grateful to the team at FFI, particularly Aleksander Skjerlie Simonsen and Emil Marstrander, for their valuable input, technical discussions, and their willingness to share insights throughout the project, all of which have been greatly appreciated.

We would also like to thank our institute at NTNU, IIK, for providing an endless supply of coffee during the project period.

Contents

Chapter 1
Introduction

Swarm operations are becoming increasingly important in military contexts, creating a need for efficient and reliable methods to secure communication within these systems. Messaging Layer Security (MLS) has emerged as a promising protocol for secure group messaging in dynamic and resource-constrained environments. In this thesis, we explore how MLS can be applied in such settings and present Valkyrie MLS, a cryptographic middleware built on top of the open-source Rust-based OpenMLS library [RC25b].

1.1 Motivation

In this thesis, we aim to achieve efficient, secure, and scalable group communication in UAV swarms without compromising performance. Prior research has identified the MLS protocol [BBR+23] as a promising candidate, offering Forward Secrecy (FS), Post-Compromise Security (PCS), and efficient group key management. While initial efforts by Leon and Britt [LB22] and Marstrander [Mar23b] have demonstrated the feasibility of MLS in drone settings, several open questions remain, particularly related to resource usage, stability, and authenticated dynamic group membership during flight.

More broadly, MLS is a new group key agreement protocol for scalable and secure group communication. However, most practical applications so far have centered on secure messaging, where certain simplifying assumptions often hold, such as the presence of a centralized (though untrusted) delivery service, relatively stable connectivity, guaranteed message delivery, and Internet-like infrastructure.

Our work investigates the applicability of MLS in a different operational context: How can secure group key agreement be achieved efficiently in resource-constrained, dynamic, and decentralized environments? We aim to expand the practical applicability of MLS and inform future protocol extensions designed for constrained or

distributed systems.

1.2 Scope

This project explores the use of the Rust-based OpenMLS library to secure communication in distributed UAV swarms. It builds on the work of Marstrander [Mar23b], who previously implemented MLS using the C++-based Cisco library MLS++.

The Valkyrie system developed at Norwegian Defence Research Establishment (FFI) serves as the reference architecture for our UAV swarm implementation. We integrate OpenMLS under realistic constraints, including limited bandwidth, constrained computational resources, and decentralized coordination. We also compare its performance to MLS++, and we assess the feasibility of encrypting video streams.

This thesis focuses exclusively on using MLS as the protocol for group key establishment, with OpenMLS as the implementation and the Totem protocol [AMM+95] for the delivery service. Alternative protocols are not explored, although we acknowledge that they may present different trade-offs. We limit our attention to performance evaluation and the integration of authentication mechanisms.

1.3 Research Questions

This thesis is guided by research questions that explore key aspects of the problem we aim to investigate. These build on the questions defined in our earlier specialization project [AB24].

We address the following questions:

RQ1: Can OpenMLS be effectively used for secure communication in UAV swarms?

We evaluate the suitability of OpenMLS as a provider of MLS functionality and examine the challenges involved in adapting OpenMLS to a distributed, resource-constrained system that requires real-time communication. This includes both architectural and software considerations needed to integrate OpenMLS into the swarm.

RQ2: How can we extend an MLS-based swarm system to include an authentication service and a delivery service?

While MLS provides end-to-end encryption by default, it does not handle authentication or message delivery. To secure a real-world swarm, these capabilities must be

added. This question investigates how the Totem protocol can be used or adapted for reliable, ordered message delivery, and how drone authentication can be implemented through credential management, binding verification, and support for expiry and revocation.

RQ3: How does OpenMLS perform compared to MLS++ in a UAV swarm?

We compare OpenMLS with the MLS++ implementation from Marstrander [Mar23b], focusing on key performance metrics such as CPU and memory usage, as well as message size. The goal of this comparison is to determine whether OpenMLS offers performance improvements over MLS++, or if it introduces new limitations in the context of a UAV swarm.

RQ4: What is the impact of video streaming on MLS-based communication in UAV swarms?

Some of the Valkyrie drones can stream live video from their camera to the Ground Control Station (GCS) during flight. This question examines how adding video streaming affects the communication system when using MLS for encryption.

We use the same metrics as in RQ3, namely CPU and RAM usage, to measure the additional load introduced by video traffic. The goal is to evaluate whether MLS can still perform effectively under multimedia conditions.

1.4 Related Work

We have reviewed the state of the art and identified background material relevant to this thesis. In this subsection, we summarize the most important findings from the literature that are directly relevant to our project. Of the related works, the study by Marstrander [Mar23b] forms the primary foundation for our work and has significantly influenced the design and direction of this project.

Leon and Britt [LB22] evaluate the suitability of different key agreement protocols for secure multi-party communication among unmanned surface and aerial systems, emphasizing support for FS, PCS, and scalability. They identify MLS as a strong candidate for replacing conventional point-to-point encryption in such systems.

Dietz [Die22] extends this work by implementing MLS in a simulated UAV swarm using the Cisco-developed MLS++ library. Their evaluation focuses on packet loss tolerance and membership dynamics, highlighting the potential of MLS, while also revealing stability issues in environments with high packet loss and without ordered message delivery. These observations emphasize the importance of complementary

services, particularly a Delivery Service to support MLS operations in a distributed environment.

Building on this, Marstrander [Mar23b] integrates MLS++ into a live UAV swarm system and introduces the Totem protocol to ensure ordered message delivery. While this implementation demonstrates the feasibility of using MLS in airborne systems, it also reveals limitations in MLS++, particularly the lack of documentation.

In addition, Marstrander highlights several critical areas for future work. Notably, the absence of certificate verification allows unauthorized parties to potentially join the MLS group. This emphasizes the need for a proper Authentication Service. While conceptual solutions are proposed, implementation and validation were left as future work. Finally, the paper points out that synchronizing membership between MLS and Totem adds unnecessary complexity, suggesting that a tighter integration of the two protocols could improve both performance and maintainability.

More recently, Leon, Britt, and Hale [LBH24] introduced the MLS API for Unmanned Surface and Aerial Systems Integration (MAUI) framework. Their implementation demonstrates secure coordination between unmanned aerial and surface platforms, using MLS to protect both data and command and control (C2) traffic. They benchmarked performance using various cipher suites and key update strategies. A key challenge they observed was maintaining MLS session integrity, particularly due to the lack of concurrency controls for group operations. These findings highlight the difficulties of integrating MLS into dynamic, asynchronous environments with constrained infrastructure.

In a different line of work, Boyd *et al.* [BDdK+21] present a suite of Symmetric-Key Authenticated Key Exchange (SAKE) protocols that achieve full FS and robust synchronization, relying solely on symmetric cryptographic primitives. The lightweight nature and provable guarantees of these protocols make them interesting in the context of tactical or swarm communication settings, especially when public key infrastructure is unavailable or impractical. However, they depend on pre-shared keys between all parties, which is a notable limitation compared to MLS, where group keys are established on the fly. Integration with group-based protocols such as MLS remains unexplored, and the applicability of such symmetric approaches to decentralized swarm scenarios involving dynamic membership and group messaging remains an open question.

1.5 Our Contributions

In this thesis, we contribute practical implementation work that advances the integration of MLS in UAV swarms. Our key contributions can be summarized as:

Valkyrie MLS: We present Valkyrie MLS, a cryptographic middleware built on top of OpenMLS, an open-source implementation of MLS in Rust.

Delivery Service: We show how Corosync [Pro24], an open-source implementation of the Totem protocol, enables message ordering in an MLS-based system. This builds on Marstrander [Mar23b], who used a custom Totem variant to order handshake messages.

Authentication Service: We design and implement a proof-of-concept authentication service based on Ed25519 credentials, allowing drones to be authenticated before joining the swarm.

Dynamic Group Management: We extend Valkyrie MLS to automatically detect and add drones as they enter communication range. We also define the group logic specifying who is responsible for adding new members and under what conditions.

1.6 Thesis Outline

In Chapter 2, we present the relevant background for this thesis, covering UAV swarm operations, an overview of MLS and its underlying cryptographic primitives, key concepts from distributed systems, and the OpenMLS library.

In Chapter 3, we describe our development process and methodology, outlining the steps taken from initial research and design to implementation, testing, and evaluation.

In Chapter 4, we introduce Valkyrie MLS, our cryptographic middleware for use in the Valkyrie UAV swarm. We explain our design choices and detail the system architecture, including the implementation of the delivery and authentication services.

In Chapter 5, we present our testing methodology and results. The system was evaluated through unit tests, physical drone tests at Kjeller, and simulations under varying network conditions.

In Chapter 6, we answer our research questions, drawing on insights from development and testing. We also outline potential directions for future work.

In Chapter 7, we summarize the key findings of this project and reflect on its contributions. Finally, we suggest directions for further research in this field.

Chapter 2
Background

We begin our work by going through the essential background material in order to contextualize the design and implementation of our cryptographic middleware. Much of the background material we present builds on the specialization project preceding this thesis [AB24], which has been extended and refined to reflect new insights observed during the course of this work.

We start by introducing UAV swarms and swarm operations, which is the environment in which our cryptographic middleware is deployed. This provides context for the practical constraints and requirements that drive our design decisions. We then present Continuous Group Key Agreement (CGKA), an enabling cryptographic primitive for the MLS protocol. With this foundation in place, we introduce MLS itself, detailing its design goals, requirements, architecture, and core operations.

With this foundation in place, we turn to the challenges that arise when applying MLS in a distributed and unreliable environment, such as UAV swarms. MLS was originally designed for reliable networks, while swarm systems are inherently distributed and subject to message loss and partitioning. We use this section to highlight the gap between these assumptions and the challenges this gap presents.

Finally, we introduce OpenMLS, which we use as the cryptographic core of our system. This library provides the practical foundation for our middleware development and experimentation.

2.1 Military UAV Swarm Operations

Military UAV swarms depend on reliable coordination and secure communication to operate effectively in contested environments. In this section, we first introduce the concept of UAVs and their use in swarms. We then narrow our focus to the Flamingo drone and the Valkyrie swarm system developed by the FFI, which together form the operational and technical foundation for our work. As the reference platform

for this thesis, we examine the Valkyrie swarm's architecture and communication requirements in greater detail.

2.1.1 UAVs

An Unmanned Aerial Vehicle (UAV) is an aircraft that operates without an onboard human pilot, crew, or passengers. An operator may remotely control the drone, or its onboard systems may autonomously guide it using sensor data and control algorithms to navigate and make flight decisions. UAVs are commonly known as drones, and in this thesis, we use the terms UAV and drone interchangeably. However, the term drone more broadly refers to unmanned systems across various domains, including aerial, ground, surface, and underwater platforms. In military applications, the controlling entity is referred to as a Ground Control Station (GCS).

There is no universally adopted classification system for UAVs, but they are commonly categorized by size, operational range, flight altitude, airframe type, or their intended application. UAVs are used across civilian, industrial, and military domains, with applications ranging from aerial photography and environmental monitoring to parcel delivery, racing, and surveillance. In this thesis, we focus specifically on military use cases. Within this context, UAVs are commonly categorized into three operational roles [HBSS24]:

- **Surveillance drones**, which carry only sensors and cameras. These are used for Intelligence, Surveillance and Reconnaissance (ISR) operations.
- **Combat drones**, which are generally larger platforms capable of carrying and deploying warheads or weapons.
- **Loitering munitions**, or *suicide drones*, which are designed to strike targets directly, destroying themselves in the process.

Another common distinction is made between fixed-wing UAVs and rotor-based aircraft. Fixed-wing drones typically offer higher speeds and longer endurance, making them suitable for longer-range missions. In contrast, multi-rotor drones, such as quadcopters (which use four rotors), are favored for their maneuverability [HBSS24]. Smaller drones typically have shorter ranges and support lower-level tactical units, while larger drones tend to serve operational or strategic roles [HBSS24].

The choice of drone type depends on the operational context and the trade-offs involved. Larger drones typically support a higher maximum takeoff weight (MTOW), enabling the use of more powerful onboard computers, motors, and advanced radios. This allows for more complex onboard computation, longer operational ranges, and

greater autonomy from the Ground Control Station (GCS). Such systems have been widely used in conflicts like those in Afghanistan, Iraq, and Somalia. However, as highlighted in the war in Ukraine, these large UAVs have proven less effective in contested airspace where no actor maintains air superiority [Kun23]. In such environments, large drones become vulnerable and operationally fragile.

In contrast, smaller drones, despite limitations in flight time, communication range, and onboard processing, have demonstrated greater survivability and adaptability. As Kunertova notes [Kun23], these systems have shifted battlefield dynamics by improving the precision and pace of artillery strikes and enhancing situational awareness for ground forces. This underscores the strategic relevance of lightweight, mobile UAV platforms, even when constrained by hardware capabilities.

Independent of their classification, the development and use of UAVs is changing rapidly. New drone platforms and tactics are constantly emerging. This in turn has drastically changed the battlefield landscape and affects how we think about modern military strategy.

2.1.2 UAV swarms

A UAV swarm refers to a group of two or more UAVs that coordinate to accomplish shared objectives [AB24]. Compared to single-drone operations, swarms offer increased flexibility, scalability, and robustness, making them well-suited for complex and dynamic missions [Die22].

While the internal architecture of a swarm can vary widely depending on the use case and technological constraints, most systems fall into one of four general categories [Zie21; Sch14; HBSS24]:

- **Centralized:** Each UAV communicates directly with a central controller that collects data and issues commands to nodes in the swarm. This controller acts as the single point of coordination.
- **Hierarchical:** UAVs are organized in a layered structure, where lower-level drones report to and receive instructions from intermediate “squad leader” drones, which in turn may coordinate with higher-level controllers.
- **Consensus:** All UAVs participate in decentralized decision-making, often using peer-to-peer communication and voting mechanisms to agree on collective actions.
- **Emergent:** Coordination arises from simple, local interactions between UAVs, without any centralized planning. This approach mimics behaviors seen in biological swarms such as flocks of birds or schools of fish.

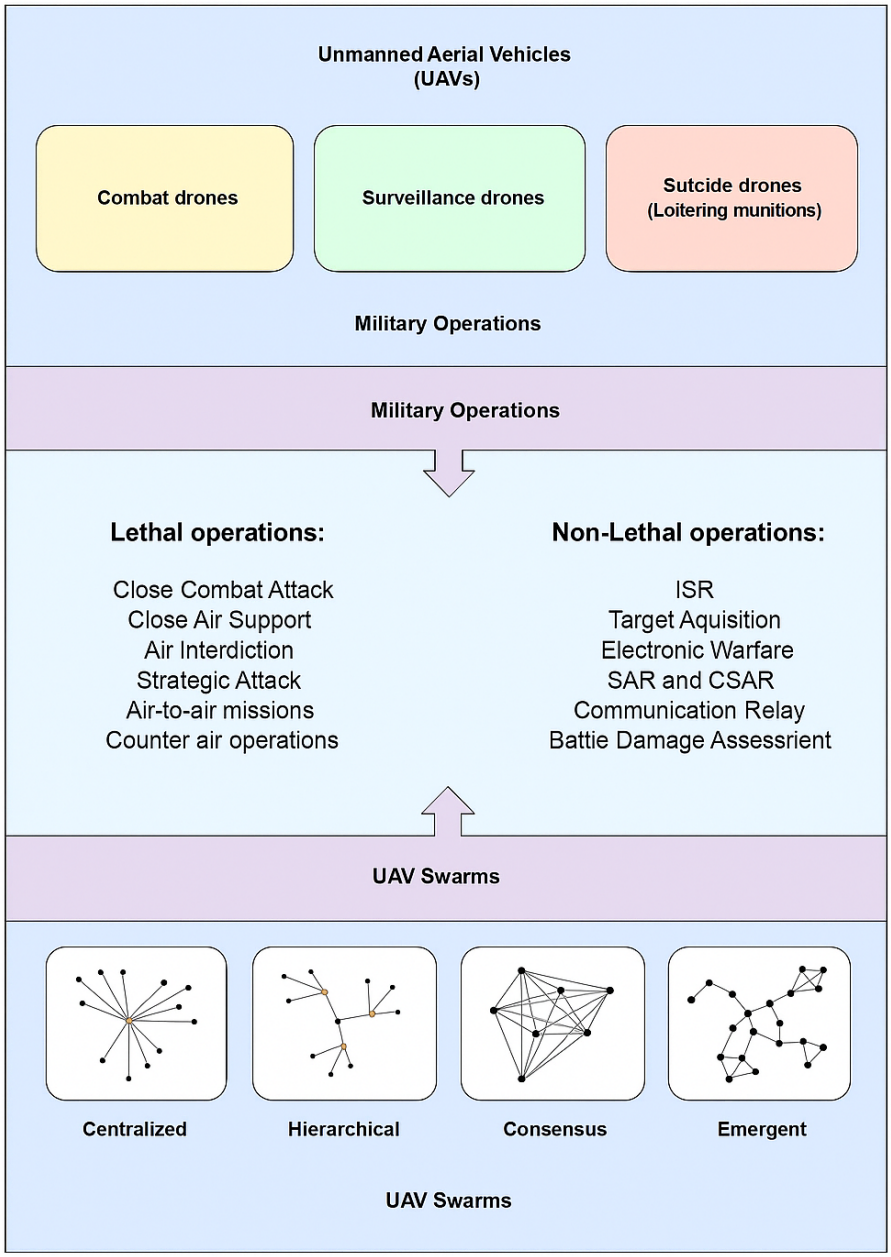


Figure 2.1: Illustrative operational scenarios for deploying UAVs or swarms. The examples are not exhaustive. The figure is self-made, inspired by Zieliński [Zie21].

The choice of architecture depends heavily on the capabilities and goals of the system. Factors such as available computing power, communication bandwidth, and autonomy levels all influence what is feasible. Each architecture has inherent trade-offs: centralized and hierarchical models are often simpler to implement and easier to control but are more vulnerable to single points of failure. In contrast, consensus and emergent architectures offer greater fault tolerance and adaptability but typically require more sophisticated autonomy and communication strategies.

Drone swarms can be applied to many of the same tasks as single UAVs, as summarized in Figure 2.1, but their advantage lies in their ability to exhibit swarm-specific properties. These include increased survivability, scalability, efficiency, autonomy, and reduced operational cost [Die22]. As noted by Halsør *et al.* [HBSS24], swarms are expected to operate reliably even under strong radio interference, given the complex environments they often encounter and the potential instability of data links.

While the simultaneous operation of multiple drones is increasingly common, these systems typically lack true swarm coordination, functioning instead as independent units. Hence, current research focuses on enabling practical and robust swarm capabilities. At present, most swarm systems remain in developmental or experimental stages [HBSS24].

2.1.3 The Flamingo UAV and Valkyrie Swarm System

In this thesis, we use the Flamingo drone [Num21], depicted in Figure 2.2, as the reference platform for developing and testing our cryptographic middleware. The Flamingo is a lightweight Class I quadcopter developed in-house at FFI for UAV autonomy research and experimentation. It is also used in the system developed by Marstrander [Mar23b], which forms the basis for the implementation presented in this thesis.

The total weight of Flamingo depends on its configuration, ranging from approximately 2.2 kg (unloaded) to 2.8 kg when equipped with a thermal camera and a 5.8 GHz mesh radio for swarm experiments [Num21]. The All Up Weight (AUW) refers to the total takeoff weight, including the drone's frame, battery, and any mounted payload or equipment. With an AUW of 2.5 kg, the drone can fly for about 45 minutes. In lighter configurations, this may extend to up to 70 minutes. For a typical ISR setup at 2.8 kg AUW, the operational flight time is around 35 minutes, accounting for a 20% battery reserve.

Flamingo is designed with modularity in mind, allowing core components to be swapped or reused across different platforms and applications [Num21]. While the platform continues to evolve beyond the configuration described in [Num21], we

Figure 2.2: The Flamingo UAV with a mesh-radio and a thermal camera. The image is retrieved from Nummedal [Num21].

require a consistent hardware baseline for system development and evaluation. The components most relevant to our work are the onboard computer and radio, as these define the limits for computation and communication. The current onboard computer is an Nvidia Jetson Xavier NX[1] [NVI24], which offers high-performance despite its compact and lightweight form. The Jetson Xavier NX features a six-core ARM CPU and runs Linux for Tegra. The drone is also equipped with a 5.8 GHz Rajant mesh radio, used to transmit application data and stream video to the GCS.

Valkyrie is the swarm system developed by FFI to support autonomy research, with Flamingo serving as its primary UAV platform [Num21]. The system is designed to be extensible, supporting integration with additional drone platforms such as Svale [MBB+24], a loitering munition or "suicide drone" [HBSS24]. Drones can be grouped into one or more swarms, and individual drones can dynamically leave one swarm and join another. These coordination actions are carried out by the drone operator through the Valkyrie GCS.

Communication within the Valkyrie swarm occurs over a shared mesh radio network, where signal quality and packet delivery vary significantly with distance and the presence of interference. Discussions with the Valkyrie development team highlight that packet loss is minimal at short distances, often approaching 0%. However, as the

[1]Due to platform iterations, some components differ from those listed in the original documentation [Num21]. For example, the platform now uses a Xavier NX instead of the originally specified Jetson TX2

Figure 2.3: Illustration of Valkyrie GCS graphical user interface (GUI) for operating a swarm. The illustration is retrieved from Halsør *et al.* [HBSS24].

signal-to-noise ratio degrades with increased range or physical obstructions, packet loss can gradually rise, sometimes reaching up to 50% before a sharp drop-off leads to complete link failure. For telemetry and command and control (C2) messages, a loss rate of approximately 30% is considered acceptable at longer ranges. This level of degradation still allows drones to coordinate effectively and maintain core functionalities such as collaborative tasking and anti-collision behavior.

The GUI in the Valkyrie GCS, as shown in Figure 2.3, presents a map-based overview showing the position of all drones in the swarm, heavily inspired by the ones seen in real-time strategy games. This interface allows the operator to select one or more active drones and issue high-level commands such as *screen this area*. The drones then coordinate among themselves to carry out the assigned task as effectively as possible. The architecture of Valkyrie is best described as emergent [HBSS24]. This means that communication within the swarm is not centrally controlled but rather arises from the distributed interactions of the drones. The operator can monitor individual drones for telemetry data such as battery status, signal strength (RSSI), and incoming sensor information. The drones and GCS have detection mechanisms for link failure. A drone will return if it has not received a *heartbeat* message from the GCS in the last ten seconds [Mar23b]. This safety feature ensures that the operator does not lose control of the drone. If a drone detects an object of interest, it can notify the operator, who may then send a command to the drone to start a live video stream to manually verify the observation. When a drone approaches low battery level, a replacement drone can automatically be dispatched from the base station so

that the swarm maintains continuous operation.

Early versions of Valkyrie were typically configured with four drones and a single GCS [Num21]. The system has since evolved and is now being tested with larger swarm sizes. In prior FFI reports, swarms of up to 40 drones were used in a simulated environment [HBSS24], though such scales have not yet been demonstrated in live flight. Further research is required to determine optimal and practical swarm sizes in real-world scenarios, but current findings suggest that larger swarms may soon be operationally feasible.

2.1.4 Transport Protocols in Swarm Systems

For network transport, User Datagram Protocol (UDP) [EFS17] is the preferred protocol in swarm communication systems. Swarm communications typically operate over a shared wireless medium, where all nodes broadcast data. In such environments, minimizing network traffic is essential to reduce interference and ensure that the channel remains available for other nodes to transmit data. Compared to its counterpart, Transmission Control Protocol (TCP) [Edd22], UDP introduces significantly less overhead, making it well-suited for these constrained, low-latency scenarios.

The primary advantage of UDP lies in its connectionless and lightweight nature. UDP does not require session establishment, acknowledgments, or retransmission mechanisms [EFS17]. This makes it ideal for real-time coordination in UAV swarms, where rapid message propagation is more critical than guaranteed delivery. However, this comes with an inherent trade-off: UDP does not provide delivery guarantees, ordering, or congestion control. In practice, this means that if signal quality degrades or swarm members drift out of reliable radio coverage, packet loss becomes inevitable. Despite these challenges, the advantages of reduced protocol overhead and minimized channel contention generally outweigh the drawbacks.

Rather than focusing on ensuring that all messages are reliably delivered, swarm systems prioritize the swift dissemination of current information. In such systems, outdated data become irrelevant by the time it is received. Reliability mechanisms in TCP, such as acknowledgments and retransmissions, can introduce additional latency, consume valuable bandwidth, and increase congestion on the shared wireless channel. These characteristics are in direct conflict with the time-sensitive and bandwidth-limited nature of swarm communication. Consequently, UDP is generally preferred as the network transport protocol in swarm systems.

2.2 Continuous Group Key Agreement

Securing communication in dynamic and emergent swarms, like the Valkyrie swarm, requires protocols that can adapt to constantly changing group membership. Traditional secure messaging schemes struggle to scale in such environments, especially under frequent joins, leaves, and updates [RCB19]. To meet these challenges, the field has introduced the concept of CGKA [ACDT20].

CGKA refers to a class of cryptographic protocols designed to enable efficient, scalable, and secure key management in dynamic groups. These protocols ensure that group members can securely share and update encryption keys as membership changes. The MLS protocol is a prominent example of such a protocol.

2.2.1 Challenges with Traditional End-to-End Encrypted Group Communication

To understand the value of CGKA, it is important to first consider the limitations of earlier approaches, most notably pairwise End-to-End Encryption (E2EE) protocols such as the Signal Protocol [PM16; CCD+17]. These protocols have set the bar for what secure communication looks like in the modern era. Arguably, E2EE has become a non-negotiable feature for any messaging app that wants to be taken seriously. The Signal Protocol, with its double ratchet algorithm, is widely regarded as the gold standard for secure one-to-one communication and forms the backbone of many widely used applications. It ensures strong confidentiality of messages, even in the face of active surveillance.

Two of the main security properties that the Signal Protocol provides are FS and PCS, both of which are crucial for maintaining secure communication in dynamic group settings. FS ensures that even if an attacker gains access to a user's current keys, they cannot decrypt past messages. In contrast, PCS addresses the scenario where an adversary temporarily compromises a user's keys. PCS guarantees that future messages can still be kept secure once the compromise is resolved. These two security properties are essential for ensuring the robustness of group communication systems, particularly in the face of frequent membership changes and key compromises. Figure 2.4 illustrates the relationship between FS and PCS.

However, as secure communication has expanded from individual chats to group messaging, significant scalability challenges have emerged. Most traditional E2EE protocols were originally designed for one-to-one interactions and were later adapted to support group communication. This retrofit approach introduces inefficiencies, particularly when managing large groups with dynamic membership [BBR+23]. Key

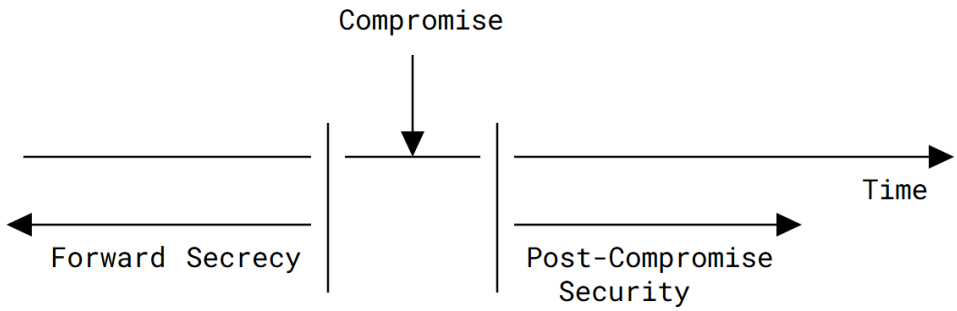


Figure 2.4: Forward Secrecy and Post-Compromise Security. The figure is adapted from Barnes *et al.* [BBR+23].

agreement protocols become more complex, and ensuring FS and PCS across all group members adds considerable overhead. Despite various messaging platforms using customized versions of the Signal Protocol [BBR+23], they commonly face the same limitations: the transition from one-to-one to one-to-many communication often results in trade-offs in performance, usability or security [CCG+18; AB24].

Pairwise Encryption: Client Fanout

To address these scalability challenges, several methods have been developed. The most basic approach is often referred to as client fanout, or pairwise encryption. The Signal Protocol [Mar14] originally used this approach. In a client fanout scheme, the sender treats a group message as N separate one-to-one messages (where N is the number of recipients). The sender encrypts the message individually for each recipient using their pairwise secure channel. This achieves the security of the underlying one-to-one protocol for each recipient, but at the cost of $O(N)$ encryption and transmission operations per message.

In fact, all group communication operations in this model, including group creation (establishing a new secure group), adding or removing members, updating a member's key material, and sending messages, require a linear number of cryptographic operations, i.e., $O(N)$ [RCB19]. While conceptually simple, and arguably the most secure, it does not scale to large groups as computation requirements increase linearly with the group size.

Sender Keys

To improve efficiency, many messaging systems, including WhatsApp [Wha24], use a *sender key* optimization. In this approach, each sender establishes a symmetric

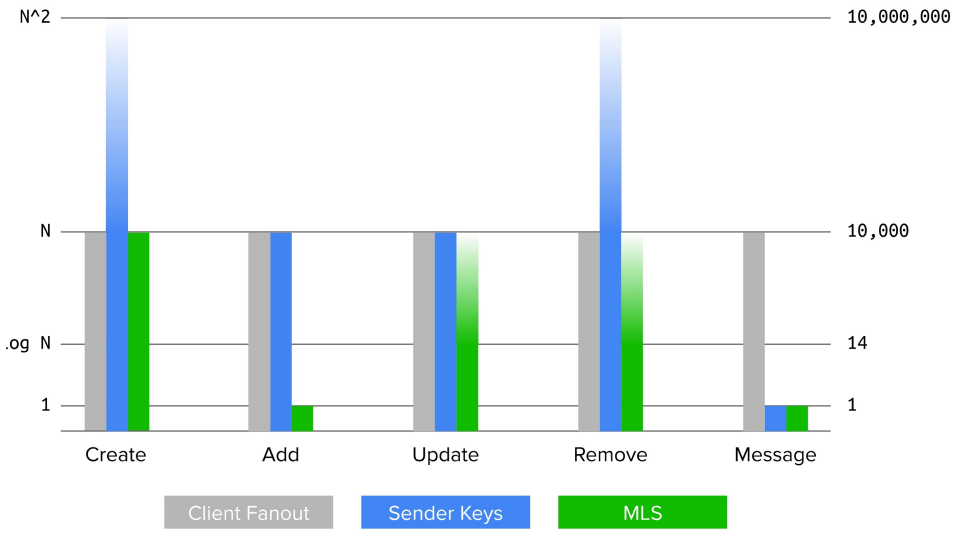


Figure 2.5: Comparing cost of operations in client fanout, sender keys and MLS. The figure is retrieved from a presentation at the Black Hat USA 2019 Conference by Robert, Cohn-Gordon, and Beurdouche [RCB19].

group key, known as a *sender key*, which is shared with all group members over secure pairwise channels. Once distributed, this key allows the sender to encrypt all outgoing messages with constant overhead, reducing the per-message cost to $O(1)$. However, the initial distribution still requires $O(N)$ work.

To preserve security, the sender key may be used with a symmetric hash ratchet, offering FS by deriving a new message key for each message [BCG23]. In long-lived groups, however, this model faces significant limitations. While the sender-key approach improves efficiency compared to pairwise client fanout [RCB19; BBR+23], it weakens long-term security guarantees. In particular, it struggles to achieve PCS, making it less suitable for applications demanding robust compromise recovery. If a sender's key is compromised, an attacker can passively decrypt all future messages from that sender until the key is replaced. Recovering from such a compromise involves generating and redistributing a new sender key to all members—an operation that require $O(N^2)$ operations.

In summary, traditional E2EE group messaging either treated the group as a collection of pairwise links (client fanout) or as a set of static sender keys. Both approaches have inherent inefficiencies, especially with regard to member sending messages or removal and compromise recovery, as illustrated in Figure 2.5. These limitations motivated the search for true group key agreement protocols that could provide FS

and PCS while scaling to large, dynamic groups.

2.2.2 Overcoming Scaling Issues with Continuous Group Key Agreement

To overcome the above issues of scaling E2EE for groups, researchers proposed a new class of protocols known as CGKA protocols [ACDT20]. The goal of a CGKA protocol is to allow a long-lived group of participants to continuously agree on fresh shared keys, even as members join, leave, or update their keys. Through this, it enables FS and PCS. In a CGKA, the group as a whole evolves a group secret over time across a sequence of epochs, so that each epoch's secret is known only to the current members and is independent of past keys.

In contrast to traditional group key exchange, which might assume a one-time setup or require all members to be online simultaneously, CGKA protocols aim to be *asynchronous* [ACDT20]. It makes no assumptions about if, when, or for how long members are online. Group members can perform key update operations locally or in isolation and then broadcast the necessary information for others to update the group key to obtain a shared cryptographic state. Achieving a scalable asynchronous protocol with strong security guarantees led designs of CGKA protocols to converge towards using cryptographic trees as a core data structure [BBR+23].

2.2.3 Asynchronous Ratcheting Trees

Asynchronous Ratcheting Trees (ARTs) [CCG+18] leverage the primitives presented by Diffie and Hellman [DH76], extending the classic Diffie-Hellman key exchange (DHKE) into a *tree-based group Diffie–Hellman* structure for efficiently deriving a shared group secret among N participants. In this construction, each group member is assigned to a leaf node of a binary tree, with each leaf holding a long-term DH key pair. Intermediary nodes represent virtual key agreements between their child nodes, and the value computed at each internal node is known by all its descendant leaves.

Since all leaf nodes are descendants of the root, the root node encapsulates a shared value known to the entire group. This root value is then used as a seed for a key derivation function (KDF), producing a chain of derived keys, similar to the ratcheting mechanism used in the Signal protocol [PM16].

To illustrate, if two members—Alice and Bob—have public DH keys g^a and g^b, their parent node holds the shared secret g^{ab}. The public component of the parent node would then be $g^{g^{ab}}$. To derive the root key, Alice recursively computes shared secrets from her own leaf up to the root. This requires her private key and the public keys

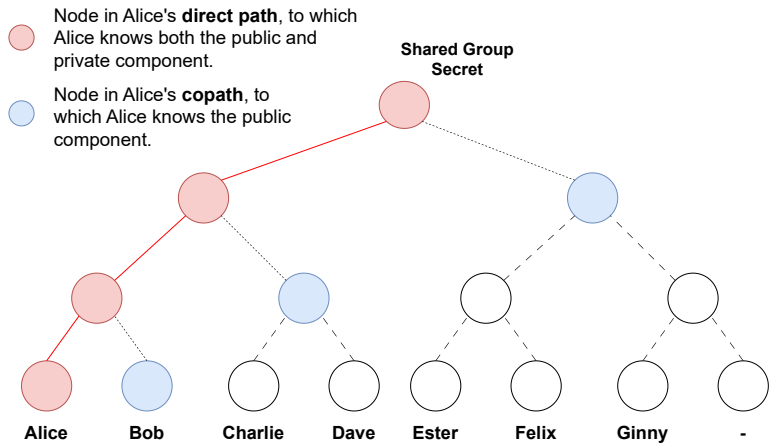


Figure 2.6: Example of a Diffie-Hellman (DH) tree in ART. Each node in the tree is implied to have a public and a private component. Nodes in Alice's direct path are marked in red, and nodes in blue lie in Alice's copath.

along the *copath* of the tree, as depicted in Figure 2.6. The copath refers to the list of sibling nodes encountered along the path from her leaf to the root[2]. To support asynchronous operation, ART assumes that *prekeys* (pre-published public keys) are available, to overcome the challenge of members being offline when their keys are needed[3].

ART was significant because it was the first design to realize strong security properties (in particular, PCS) in an asynchronous group scenario [CCG+18]. Its tree-based design supports scalable group key agreement: a tree of N members has depth $\log N$, so when a member updates their key, only the $\log N$ nodes along the path from their leaf to the root must be recomputed. PCS is achieved through regular updates of participant keys, injecting fresh entropy into the key derivation process.

2.2.4 Tree Key Encapsulation Mechanism

Following ART, the next evolution in CGKA was the proposal of TreeKEM (Tree Key Encapsulation Mechanism) [BBR18]. TreeKEM built upon the same Diffie–Hellman tree foundation as ART, but introduced important simplifications and improvements to better suit an Internet standard. The main selling point of TreeKEM over ART

[2]In practice, ART represents intermediary values using a mapping from group elements to integers [CCG+18], but the conceptual model remains unchanged.

[3]See the original paper [CCG+18] for a detailed discussion of the asynchronous key setup.

was its efficiency towards recipients and greater degree of support for concurrent operations [BBR18].

The core idea of TreeKEM is to treat each node update as a key encapsulation to the rest of the group, rather than a pure DH key exchange. A key encapsulation mechanism (KEM) allows a sender who knows a public key to generate a short random secret key and an encapsulation of the secret key by the encapsulation algorithm defined by the KEM. The receiver who knows the private key corresponding to the public key can recover the same random secret key from the encapsulation by the KEM's decapsulation algorithm [Gal12]. In TreeKEM, whenever a member updates their leaf, they generate a fresh random secret and derive new keys up the path to the root. But instead of requiring an interactive DH at each parent node, the updater uses a KEM to encrypt the new node secrets to the other group members [BBR18]. This concept provides the foundation for key agreement in MLS.

2.3 Messaging Layer Security

The Messaging Layer Security protocol is a recently standardized continuous group key agreement protocol that builds on the previously discussed TreeKEM [BBR18; WPBB23]. MLS is designed to provide strong security guarantees while maintaining efficiency in large and dynamic groups. The following sections are based mainly on the official specifications [BBR+23; BRO+25], which provide a detailed explanation of the design and operation of MLS.

2.3.1 Protocol Overview

The core functionality of MLS is continuous group authenticated key exchange (AKE). MLS supports dynamic groups, allowing membership to evolve over time while maintaining cryptographic consistency and security. It enables participants to verify each other's identity and agree on a shared secret, which in turn can be used to encrypt group messages. The shared secret changes when changes to the group are made, like the addition or removal of a member, so that only the current members know the shared secret. We refer to the period for which a group secret is valid as an *epoch* and the group secret as the *epoch secret*.

MLS assumes the presence of an Authentication Service (AS) and a Delivery Service (DS) as supporting services in the system architecture [BRO+25]. The AS is responsible for ensuring that only authenticated entities participate in the group. The DS functions as a message relay, storing and delivering encrypted messages between members.

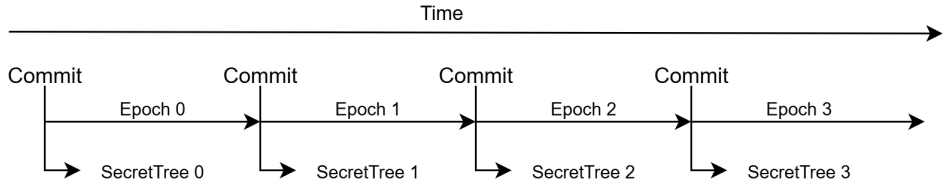


Figure 2.7: Epoch progression in MLS. Each epoch correlates to a unique SecretTree derived via a `Commit`, which updates the group state. The illustration is made by the authors.

2.3.2 Groups and Epochs

In MLS, a group represents a dynamic collection of members who wish to communicate securely. Each group maintains a shared cryptographic state that includes a ratchet tree, which is a binary tree structure used to efficiently manage the group's key material. The group state is consistent across all honest members, meaning every participant holds the same view of the current members, the keys in use, and the group's operational context.

A central concept in MLS is the epoch, which represents a specific version of the group's cryptographic state. Whenever the group changes, such as when a member joins, leaves, or updates their key material, it transitions to a new epoch, as shown in our illustration in Figure 2.7. This transition is triggered by a `Commit` message, which includes all the information needed to update the group state and derive new shared secrets. Epochs evolve in a linear fashion, with each representing a distinct group configuration and corresponding key schedule.

2.3.3 TreeKEM, TreeDEM, and TreeSync

Although TreeKEM forms the cryptographic core of the MLS protocol's key agreement, it is not a complete solution on its own. One limitation lies in handling group operations beyond basic updates and removals. While TreeKEM allows members to inject new entropy during updates, adding a new participant requires extending the tree, integrating the member's key, and securely distributing the resulting secrets. Ensuring that all members maintain a consistent view of group membership and tree state, especially during concurrent joins and leaves, is non-trivial. TreeKEM alone does not fully address inconsistencies, and its original design lacks support for PCS in the presence of active attackers or malicious insiders [BBR18].

Originally centered around TreeKEM [BBR18], the MLS protocol has since evolved into a modular structure composed of three coordinated sub-protocols: TreeSync,

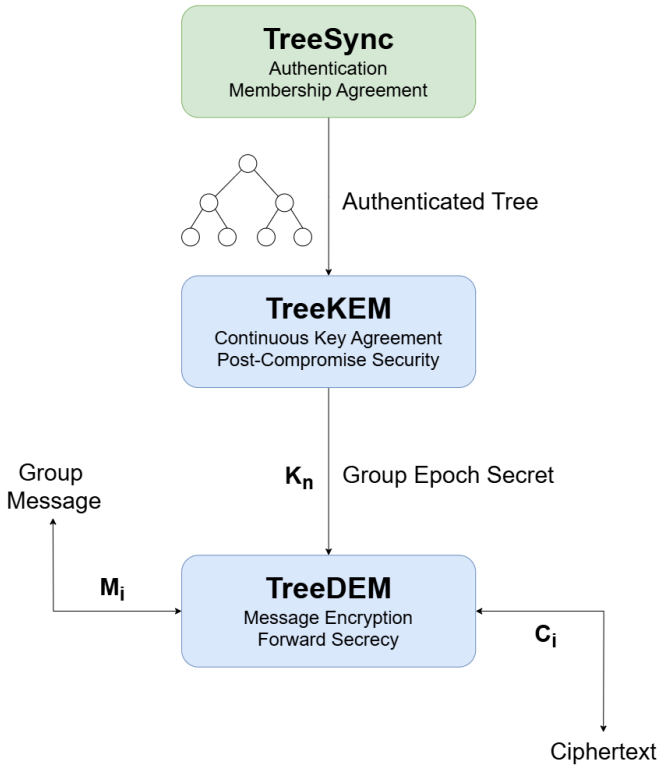


Figure 2.8: A modular decomposition of MLS into its sub-protocols: TreeSync, TreeKEM, and TreeDEM. Illustration is retrieved from [WPB25].

TreeKEM, and TreeDEM [BBR+23; WPBB23], as shown in Figure 2.8. This decomposition highlights the different responsibilities of each component and addresses the shortcomings of earlier designs.

TreeSync is responsible for maintaining a consistent group state across all members. It ensures that each participant has the same view of the group's structure and key history. The state is organized as a tree, where each occupied leaf node corresponds to a group member and each internal node represents a subgroup. Unlike earlier constructions such as ART, TreeSync authenticates both the initial group state and subsequent updates using signatures and Merkle-tree-style hashing [WPBB23]. It also verifies the integrity of the tree structure itself. The synchronized, authenticated tree generated by TreeSync is then passed to TreeKEM.

TreeKEM handles the group key agreement logic. Based on the authenticated tree

from TreeSync, it allows the group to derive a sequence of shared secrets, called epoch secrets (K_n), which are updated as members join or leave. This process provides PCS by refreshing keys with minimal overhead—typically requiring only a logarithmic number of public key encryptions and a single decryption per recipient [WPBB23]. Additionally, TreeKEM makes certain assumptions about the design of the overall system. Notably, TreeKEM relies on DS, which is tasked with receiving messages from individual group participants, and broadcasting them to all other group participants.

Finally, TreeDEM is a symmetric encryption scheme built on top of TreeKEM. It uses the epoch secret K_n to derive message encryption keys for each member. These keys are updated with each message using a ratchet mechanism to ensure FS [WPBB23].

The current version of TreeKEM and the accompanying sub-protocols in the MLS standard is the result of multiple revisions and extensions since its early designs. Together, these three sub-protocols enable groups in MLS to be dynamic, allow operations to happen asynchronously, and ensure that computations scale better than linearly, all while providing strong security guarantees [WPB25].

2.3.4 MLS Messages and Cryptographic Operations

MLS messages fall into three main categories: handshake messages, application messages, and auxiliary messages [BRO+25]. Handshake messages are `PublicMessage` or `PrivateMessage` objects carrying a `Proposal` or `Commit`, used to manage group state. Application messages are `PrivateMessage` objects that carry encrypted application data. Auxiliary messages, such as `Welcome`, `KeyPackage`, and `GroupInfo`, support group initialization and coordination.

All message types share a common framing structure. Content such as `Application Data`, `Proposals`, and `Commits` is first encapsulated in a `FramedContent` object, which is then signed to produce an `AuthenticatedContent`. This object ensures sender authentication and message integrity.

To protect the message during transmission, `AuthenticatedContent` is encoded as either a `PublicMessage` (signed only) or a `PrivateMessage` (signed and encrypted). `PrivateMessage` is preferred for both application and handshake messages, as it protects both payload and metadata. However, handshake messages *may* be sent as `PublicMessage` objects when visibility is needed by the DS [BRO+25].

All of these messages are ultimately wrapped as `MLSMessage` objects. Figure 2.9 illustrates the structure and flow, and Table 2.1 provides a summary of message types and their roles within the protocol.

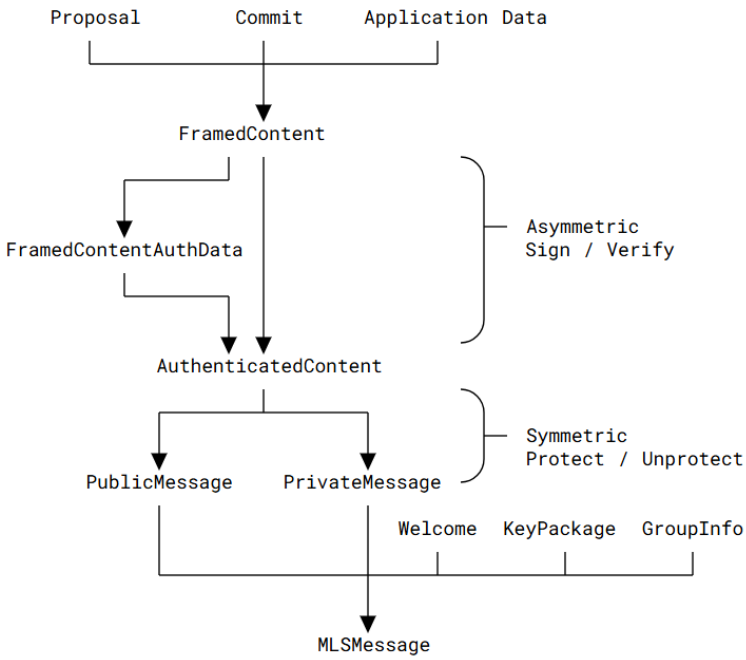


Figure 2.9: Overview of the MLS message structure and processing flow. The figure is retrieved from Barnes *et al.* [BBR+23].

Term	Description
`Proposal`	Suggested change to the group. `Proposals` are not applied until included in a `Commit`.
`Commit`	Applies one or more `Proposals` to update the group state and advance the epoch.
`Application Data`	Application messages exchanged by group members.
`Welcome`	Sent to new members to initialize their view of the group and provide the necessary secrets.
`KeyPackage`	Contains a member's public key and credentials, used when joining or updating in the group.
`GroupInfo`	Describes the current group state, including the group ID, epoch, and ratchet tree hash.

Table 2.1: The message types in MLS along with a description of their functionality.

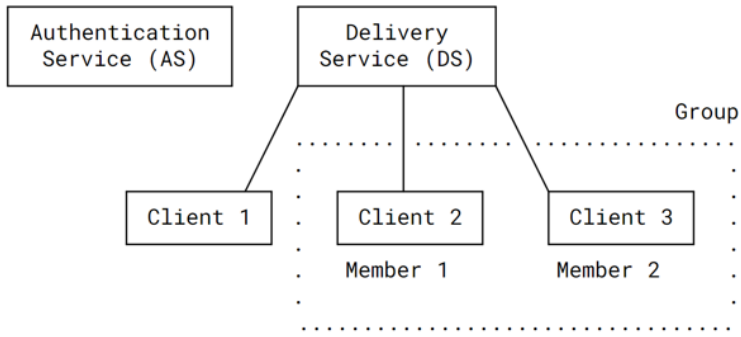


Figure 2.10: Overview of the abstract services in MLS. The figure is retrieved from Barnes *et al.* [BBR+23].

The main types of `Proposals` are `Add`, `Update` and `Remove` (not extensive). An `Add` proposes to include a new member using their `KeyPackage`, an `Update` is sent by a member when refreshing their key material to maintain PCS, and a `Remove` proposes to exclude a member and refresh shared secrets.

2.3.5 Abstract Services: Authentication Service and Delivery Service

While MLS provides the cryptographic foundation for secure group messaging, real-world deployment requires additional infrastructure services: the Authentication Service and the Delivery Service, as depicted in Figure 2.10. These abstract services enable identity management and message transport, which are essential for operationalizing MLS beyond pure cryptography. The AS establishes trust between clients by managing credentials, while the DS ensures reliable message delivery.

Authentication Service

The AS is responsible for issuing credentials that bind client identities to signature key pairs and enabling client verification [BRO+25]. MLS does not specify how this should be conducted, but states that whenever a new credential is introduced in the group, it must be validated by the AS. A member's credential is said to be validated when the AS verifies that the credential's presented identifiers are correctly associated with the presented signature key field in the member's leaf node in the authenticated tree, and that those identifiers match the reference identifiers for the member. The choice is left to the implementer to decide how this issuance and validation is conducted, but by ensuring this, authenticated participation in the

group and prevent identity spoofing or duplication [BRO+25].

Delivery Service

The DS acts as a message relay and storage service, enabling communication between clients, even when they are offline. As specified by Beurdouche *et al.* [BRO+25], it provides access to initial keying material, through `KeyPackages`, allowing clients to join groups. In addition and arguably more importantly, the DS is responsible for delivering MLS messages between clients, supporting both individual and group-wide communication.

The DS is responsible for delivering both client-targeted messages, such as `Welcome`, which are used to add new members to a group, and group-wide messages like `Commit`, which must be delivered to all current group members.

MLS tolerates out-of-order message delivery to some extent, but strict ordering is essential in two specific cases:

1. **`Proposals` before `Commits`:** All `Proposals` must arrive before the `Commit` that references them.
2. **Epoch progression:** The group must agree on the order of epoch transitions, each initiated by a `Commit`.

Concurrent `Commit` messages from different members can result in diverging group states, called *forks*, which effectively fork the group into separate epoch histories. This race condition must be mitigated by the DS, which should enforce a consistent ordering or prioritize one `Commit` over the other.

2.3.6 Group Operations

Figure 2.11 illustrates the step-by-step process of securely forming an MLS group using the AS and the DS. In Step 1, Alice, Bob, and Charlie create an account and obtain a credential from the AS, establishing a cryptographic identity. In Step 2, all participants publish their initial keying material, including their `KeyPackage`, to the infrastructure, allowing others to securely add them to an MLS group. In Step 3, Alice retrieves Bob's keying material, adds him to the group by generating a `Commit`, and produces a `Welcome` message that is stored on the DS for Bob to fetch asynchronously. In Step 4, Alice repeats the process to add Charlie, retrieving his keying material, sending a `Commit` to update the group, and issuing a corresponding `Welcome` message.

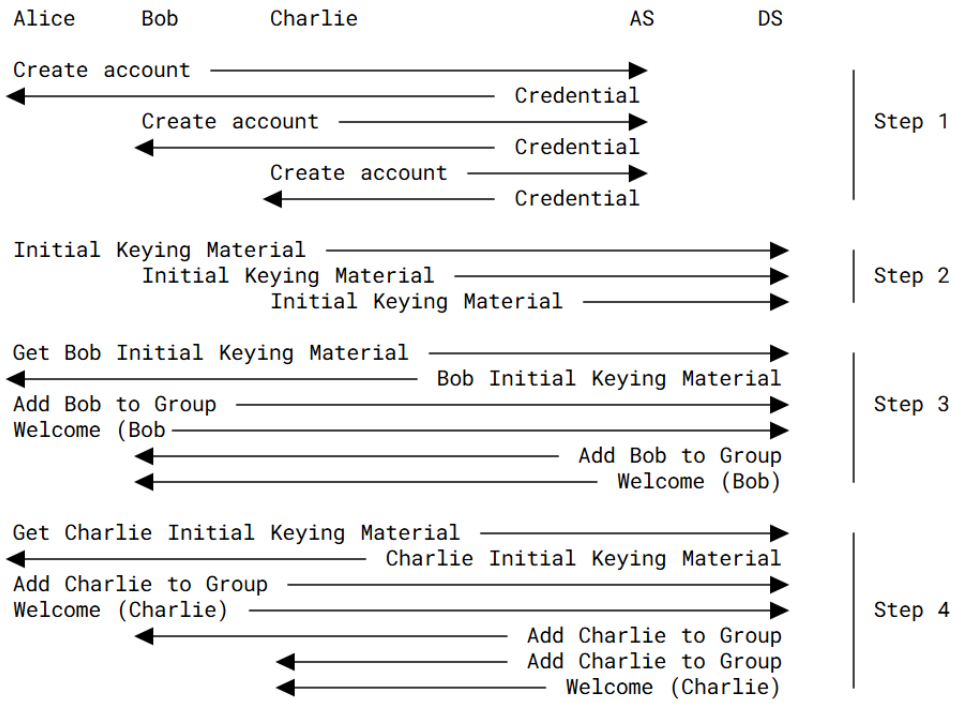


Figure 2.11: Example of group formation in MLS. The illustration is retrieved from Barnes *et al.* [BBR+23].

2.4 Distributed Systems

UAV swarms are by nature distributed systems: they consist of multiple autonomous nodes that must coordinate and communicate over unreliable wireless networks. To design secure and reliable communication middleware for such environments, it is essential to understand the foundational challenges of distributed systems. This section introduces core concepts and theoretical limitations that inform our system design, focusing particularly on the areas of agreement, consensus, and resilience under network partitions.

2.4.1 Challenges in Distributed Systems

UAV swarms function as distributed systems composed of multiple autonomous nodes that communicate over unreliable wireless links. This setup introduces several well-known challenges: unreliable communication, potential for node failure, and the difficulty of maintaining coordination in a dynamic topology.

The Eight Fallacies of Distributed Computing

The *Eight Fallacies of Distributed Computing* [DJLG94], first identified by Deutsch *et al.*, are a set of common but misleading assumptions often made when designing distributed systems. In the context of UAV swarms, where independent agents must communicate over shared, lossy wireless links in dynamic and potentially adversarial environments, these assumptions rarely hold. Ignoring them can result in fragile designs, especially for systems that depend on timely, secure, and synchronized communication.

Understanding these fallacies helps frame realistic expectations and is crucial when designing middleware for reliable and secure swarm operation:

1. **The network is reliable:** Communication is often disrupted by interference or contested mediums, making message loss a frequent reality.

2. **Latency is zero:** Delays vary significantly due to distance, congestion, or retransmissions, affecting synchronization and coordination.

3. **Bandwidth is infinite:** Low-throughput radio links limit the amount of data that can be transmitted, which has serious implications for protocol overhead and scalability.

4. **The network is secure:** Swarm communications use a shared, open medium. All communications can therefore be intercepted or tampered with by adversaries. Hence, encryption, authentication, and integrity protections are needed.

5. **Topology does not change:** Participants may move, go offline, or dynamically join or leave the swarm, requiring protocols that tolerate frequent reconfiguration.

6. **There is one administrator:** Centralized control may not be possible—especially in autonomous deployments—so systems must support decentralized coordination.

7. **Transport cost is zero:** Every message consumes both bandwidth and time on the shared medium, which are limited resources.

8. **The network is homogeneous:** While less critical in swarms, this fallacy assumes uniform hardware and capabilities across the network. In practice, variations in hardware, compute resources and software stacks do occur, so protocols should remain interoperable across heterogeneous systems.

These fallacies highlight the inherent challenges in designing robust, distributed protocols for UAV swarms. As we show in later sections, addressing these requires middleware that supports partition-tolerance, asynchronous messaging, and dynamic group membership without relying on idealized network conditions.

Consensus and Agreement Problems

Achieving consensus between nodes in a distributed system is a fundamental challenge. Several theoretical works illustrate the inherent difficulties in reaching consensus among distributed nodes. The variations of the problem differ in *strength*, meaning they differ in their system model. Different system models have different assumptions about failure models, communication synchronicity, channel reliability, and message authentication. Solutions to problems that have stricter requirements and stronger assumptions will typically solve weaker problems at the same time.

Consensus problems are often discussed in light of the following three properties:

- **Termination**: All non-faulty nodes eventually deciding on a value.
- **Agreement**: All nodes that decide on a value do so on the same value.
- **Validity**: Values that have been decided must have been proposed by some nodes, meaning no trivial or *fallback* values.

Any algorithm that exhibits these three properties can be said to solve the consensus problem.

Three notable works in this context are The Byzantine Agreement Problem [LSP82], The Consensus Problem [BDM93], and the Interactive Consistency Problem [TP88], which all highlight challenges with reaching consensus with varying assumptions. The Byzantine Agreement Problem [LSP82], also referred to as the Byzantine Generals Problem, highlights this challenge, where nodes must reach agreement despite the presence of faulty or malicious actors. Even in the absence of malicious actors, reaching agreement can be difficult due to issues such as message loss, reordering, or delays. The Consensus Problem [BDM93] focuses on the challenge of ensuring all non-faulty nodes agree on a single value. Without consensus, divergent interpretations may hinder the overall system in its decision making. A stricter requirement is posed by the Interactive Consistency Problem [TP88], where all correct nodes must agree on the values received from all other nodes. This is particularly important for tasks like shared key establishment, where consistency in each node's understanding of peer identities and key material is essential to secure operation.

Moreover, Fischer, Lynch and Paterson proved that there is no deterministic consensus algorithm in a fully asynchronous distributed system [FLP85]. However, their work does not state that consensus can *never* be reached: merely that under the certain assumptions, no algorithm can always reach consensus in bounded time given asynchronous communication between nodes. This asserts however, that consensus is a non-trivial task.

Network Partitions and the CAP Theorem

The CAP theorem, first proposed by Brewer and later formalized by Gilbert and Lynch [Bre00; GL12], states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees:

- **Consistency:** All nodes return the most recent and correct data in response to a request. The precise meaning of "correctness" is context-dependent and may relate to cryptographic state, configuration, or data values.
- **Availability:** Every request to a non-failing node receives a response, regardless of the current system state.
- **Partition Tolerance:** The system continues to operate correctly despite arbitrary message delays, losses, or node failures that create network partitions.

As partitions are an unavoidable scenario in real-world distributed systems, this theorem implies that distributed systems must choose between consistency and availability when partitions occur.

2.4.2 Totem: Reliable Group Communication and Ordering

Distributed systems that must maintain a consistent shared state across nodes, require the implementation of some kind of consensus algorithm. The choice of such an algorithm depends on the system model and underlying assumptions. MLS requires a shared state in which handshake messages, such as `Commits`, are interpreted in the same order for all participants. In swarm communications we assume asynchronous communication, message loss, variable delays, and the absence of a centralized ordering authority. We cannot guarantee consensus for such a system model [FLP85], but we can impose an ordering on the messages that are sent to achieve MLS' requirements of interpreting handshake messages in order. Marstrander [Mar23b] proposed using the Totem Single Ring Protocol [AMM+95] to achieve this ordering.

The Totem Single Ring Protocol address ordering by organizing nodes into a logical ring and circulating a token among them. Only the node holding the token may transmit, ensuring that messages are broadcast in a globally consistent order. This design enables total order broadcast, which is essential for maintaining consistent cryptographic state.

Totem supports multiple message ordering guarantees:

- **Agreed Order**: All correct nodes deliver messages in the same sequence.
- **Safe Order**: Messages are delivered only when all recipients can confirm delivery.
- **Causal Order**: Delivery respects causal relationships between messages.

In addition to message ordering, Totem manages dynamic group membership. When nodes join or leave, the protocol ensures that all remaining nodes agree on a new view before resuming communication. This is crucial in environments subject to failure or reconfiguration.

To handle message loss, Totem uses metadata embedded in the token to detect gaps and trigger retransmissions. It also adapts to network conditions by regulating token circulation speed.

Corosync Cluster Engine

Corosync [Pro24] is an open-source group communication engine built upon the Totem Single Ring Protocol to deliver total order broadcast and consistent membership views. While Totem provides the underlying guarantees for total order broadcast and membership agreement, Corosync packages these guarantees into a practical framework suited for real-world applications. Corosync extends Totem's foundation with an Application Programming Interface (API) for multicast communication, failure detection, and dynamic group management.

A central component of Corosync is the Closed Process Group (CPG) service, which enables applications to form logical groups and exchange messages with the ordering guarantees of Totem. Corosync ensures that all non-faulty nodes deliver messages in the same sequence, even during network disruptions or membership changes. This makes Corosync a practical and usable implementation of Totem.

2.5 OpenMLS

This section describes OpenMLS, an open-source Rust implementation of the Messaging Layer Security protocol, maintained by Phoenix R&D and Cryspen [RC25b]. OpenMLS provides a high-level API for managing MLS groups, as defined by the MLS specification [BBR+23]. It includes most of the core functionalities needed to build a secure messaging application based on MLS, including group setup, member management, and message handling. However, it does not include implementations of an Authentication Service or a Delivery Service, which must be provided externally. OpenMLS provides extensive documentation through both its official Rust API reference [RC25c] and the OpenMLS Book [RC25a].

OpenMLS supports the use of custom cryptographic providers, key stores, and random number generators. This is facilitated through the `CryptoProvider` trait, which defines an abstraction over the underlying cryptographic operations. Thus, OpenMLS allows different providers to be integrated as needed. This enables the use of various cryptographic algorithms, such as AES-128-GCM or ChaCha20-Poly1305, depending on the selected ciphersuite and the capabilities of the chosen provider. OpenMLS currently supports the following three ciphersuites:

- `MLS_128_HPKEX25519_AES128GCM_SHA256_Ed25519` (mandatory to implement)
- `MLS_128_DHKEMP256_AES128GCM_SHA256_P256`
- `MLS_128_HPKEX25519_CHACHA20POLY1305_SHA256_Ed25519`

2.5.1 Credentials and Key Packages

Members in OpenMLS are identified by `Credential` objects. These are structured as shown in Code 1, and include a `CredentialType` and a serialized payload. The actual payload is stored as a variable-length byte array (`VLBytes`), where interpretation depends on the specified `CredentialType`.

Currently, OpenMLS only supports the `BasicCredential`, which is a simple assertion of identity with no attached metadata. This provides minimal identity assurance, and systems requiring stronger authentication must implement or integrate additional credential mechanisms externally. `X509` credentials are reserved for future use, and currently have no associated functionality. `Other` is reserved for custom, application-defined credential formats.

MLS uses `KeyPackages` to support asynchronous group setup. A `KeyPackage` bundles together the cryptographic material and metadata necessary for a client to be added

Code 1 Credential structure in OpenMLS.

```rust
// A credential used to identify a group member
pub struct Credential {
    credential_type: CredentialType,
    serialized_credential_content: VLBytes,
}

// Supported credential types
pub enum CredentialType {
    // A basic MLS credential containing identity and signature key
    Basic = 1,
    // An X.509 certificate
    X509 = 2,
    // A custom, user-defined credential type
    Other(u16),
}
```

to a group. `KeyPackages` are designed to be pre-generated and published to the DS, enabling other clients to retrieve and use them when adding new members to an existing group. A `KeyPackage` is intended for one-time use and clients may generate multiple `KeyPackages` in advance. The corresponding private keys must be securely stored locally, as they are required when the client joins a group.

2.5.2 Groups

Groups in OpenMLS are managed using the `MlsGroup` object, which provides the primary high-level interface for group operations. Code 2 demonstrates how Alice initializes a group and adds Bob using his `KeyPackage`. Examples of group operations include adding or removing members and retrieving the current group membership.

When creating a group, a set of configuration parameters can be set. These parameters have to be agreed-upon by all clients joining the group. Among these parameters are:

`max_past_epochs`: Maximum number of past epochs for which application messages can be decrypted. The default is 0.

`out_of_order_tolerance`: Defines a window for which decryption secrets within the current epoch are kept. This is useful in case the DS cannot guarantee that all application messages have total order within an epoch. The use of this affects FS within an epoch. The default value is 5.

`maximum_forward_distance`: Defines how many incoming messages can be skipped. This is useful if the DS drops application messages. The default value is 1000.

After having set the desired parameters for the group and created the group, members can add new participants by generating `Welcome` messages based on their published `KeyPackages`. Members are added to the group `MlsGroup.add_members()` function using the corresponding `KeyPackages` from every new member as part of the input. The function returns the 3-tuple containing `(Commit, Welcome, GroupInfo)`. The resulting `Commit` message must be sent to all existing group members to apply the membership change. The `welcome` message must be sent to the newly added members. The `GroupInfo` object is an optional value used in groups where external joins are allowed.

Code 2 Create group and add member.

```
// Alice initializes a new MLS group
let mut alice_group = MlsGroup::new(
    cryptographic_provider,
    alice_signature_keys,
    group_configurations,
    alice_credential,
);

// Alice adds Bob to the group using his key package
let (commit, welcome, group_info) = alice_group.add_members(
    cryptographic_provider,
    &alice_signature_keys,
    &[bob_key_package],
);
```

To join a group from a `Welcome`, a `MlsGroup` can be instantiated from the `MlsMessageIn` message containing the `Welcome` through a two-step process. The reason for this two-phase process is to allow the recipient of a `Welcome` to inspect the message, for example, to determine the sender's identity, validate their credential, and so on.

Chapter 3
Methodology

The research conducted in this thesis is best described as *applied*, as it aims to address a specific and practical challenge: securing communication within a UAV swarm using the MLS protocol. Our methodology follows a software engineering approach, built around iterative development and continuous feedback. Throughout the project, system design evolved in response to implementation insights and testing outcomes, enabling a gradual refinement toward a functional and robust solution.

The work is also *experimental* in nature, involving systematic performance evaluation of the implemented system. Our benchmarking draws comparisons to prior work, particularly the evaluation of MLS++ conducted by Marstrander [Mar23b].

To clarify how each research question (RQ) shaped the methodology, we briefly outline their roles below:

RQ1 and RQ2 relate to the core development process. RQ1 explores how OpenMLS can be employed to enable secure communication in UAV swarms, while RQ2 considers how MLS can support authentication and message delivery in distributed swarm systems. These questions informed our architectural decisions, component design, and the integration strategy.

RQ3 concerns the system's performance. It prompted the development of a test framework to benchmark OpenMLS against MLS++ under comparable conditions. This included identifying appropriate metrics, aligning test environments, and defining evaluation criteria based on Marstrander's prior work.

RQ4 extends the evaluation by considering the integration of real-time video streaming into the MLS-secured system. To address both the architectural challenges and performance trade-offs, we tackled RQ4 with a combination of implementation work and targeted testing.

3.1 Project Overview

The development of Valkyrie MLS followed a five-phase model:

1. **Initial Development Phase**: Define functional requirements, establish the development environment and read up on relevant literature.
2. **Design Phase**: Outline the overall system architecture, with particular focus on integrating OpenMLS, the delivery service, and the authentication service.
3. **Implementation Phase**: Implement the outlined solution in code.
4. **Testing Phase**: Validate the implementation through functional and performance testing.
5. **Evaluation Phase**: Assess our solution and methodology, analyze performance results, and finalize documentation.

This model allowed us to begin with a clearly defined architectural vision, but the process was far from linear. In practice, the design, implementation, and testing phases (Phases 2–4) followed an iterative workflow. We continuously developed, tested, and refined the system in response to implementation challenges and insights from functional testing, gradually improving its robustness and efficiency.

3.2 Initial Development

The first phase of the project focused on establishing a strong technical foundation for the work that followed. This included acquiring the necessary programming skills, reviewing relevant literature, and preparing a suitable development environment.

We devoted a large part of the early effort to learning the Rust programming language. As neither of us had prior experience with Rust, we invested time in understanding its syntax and design patterns. Given that OpenMLS underpins the cryptographic operations of MLS, its documentation [RC25a; RC25c] offered valuable insight into what became a core component of our system.

In parallel, we studied Marstrander's thesis [Mar23b] in depth, along with other foundational work on MLS in unmanned systems, as discussed in Section 1.4. This helped us understand the architectural limitations and performance challenges associated with earlier MLS implementations in a swarm context.

To support local simulated experimentation and rapid iteration, we set up a containerized development environment using Docker, following a similar approach to

that described by Marstrander. The setup, depicted in Figure 3.1, included a Jetson Nano Developer Kit, a set of Ubuntu-based Docker containers, and one Ubuntu host machine. The Jetson Nano is a compact single-board computer sharing the same ARM architecture as the Jetson Xavier NX (used onboard the Flamingo drones), but offers fewer computational resources. The Docker containers, running Ubuntu 20.04, were connected via a Docker bridge network and orchestrated by the host machine. We physically connected the Nano to the host system via Ethernet, such that all components operated within a single local area network (LAN). This configuration enabled us to simulate a multi-node swarm environment and verify system functionality under realistic architectural constraints, without requiring immediate access to drone hardware.

3.3 Design

Many of the ideas presented by Marstrander [Mar23b] influenced the design of our solution. However, our aim was not to directly port the existing system from C++ to Rust. Instead, we evaluated the underlying design principles and architectural decisions described in the thesis, without relying heavily on the original implementation or its source code [Mar23a]. This approach allowed us to assess the core ideas independently of their original context.

We invested significant time in architectural planning early in the project. Although we later had to revisit and revise several of these initial decisions, this time was far from wasted. The early reflection laid a foundation for better decision-making and a deeper understanding of the system as the project progressed.

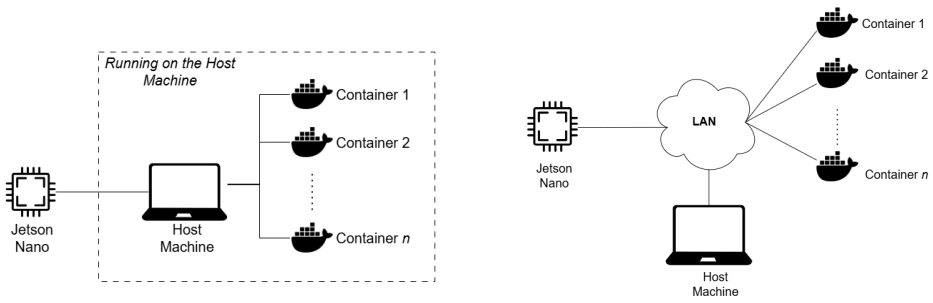


Figure 3.1: Local development environment used during testing and development. The left diagram illustrates the physical setup, including the Jetson Nano and host system. The right diagram shows the logical architecture, including containerized components and network configuration.

A general trend we observed was a gradual simplification of the architecture. The stages in the architectural evolution of our system reflect efforts to reduce unnecessary complexity and improve clarity in our system. The results of our final design are described in detail in Section 4.2.

3.4 Implementation

After agreeing on a working system architecture, we began by developing a baseline system and gradually extended it as system capabilities grew. Early work focused on end-to-end message passing, cryptographic operations using MLS, and integrating Corosync with our internal data flow. Later, we added support for authentication using the validation mechanism described in Section 4.4, and finally refined the automatic procedures outlined in Section 4.5.

To verify the correctness of these modules, we developed unit tests alongside the implementation. The core functionalities were tested through these tests, which helped ensure components behaved as specified and served as safeguards against regressions during later development. However, we were not able to cover all aspects of our system, particularly the asynchronous network exchanges.

3.5 Testing

We conducted physical testing over one week at FFI's facilities at Kjeller, where we had access to three Flamingo drones. This provided an opportunity for both functional and performance testing on the actual target hardware. Based on observations from the tests, particularly the functionality tests, we were able to patch issues and optimize key functionality, leading to improvements in the final result.

To organize our testing efforts, we developed a detailed test plan outlining the functional requirements and performance metrics of interest. We partially based this plan on the test procedures documented by Marstrander [Mar23b], which allowed us to generate results that could be compared meaningfully to prior work. However, several aspects of Marstrander's tests did not align well with our system architecture, making them difficult to replicate. For example, we omitted the heartbeat interval, incorporating its functionality into update messages instead. This made testing that parameter irrelevant to our implementation.

We categorized our testing into three main areas: functional testing, performance testing, and network testing. A more extensive overview of our test setup and testing methodology is provided for each of the different tests in Section 5.

3.6 Evaluation

The evaluation phase centered on analyzing the results from our performance and functional tests. These results formed the basis for answering RQ3 and partially RQ4, especially regarding system runtime behavior, resource usage, and the integration of video streaming within the MLS communication framework.

In parallel with designing, implementing, and testing, we documented key challenges encountered during development, particularly those for which we were unable to find satisfactory or scalable solutions. These observations provide valuable input to our later discussion and form the foundation for our proposed future work. They also contribute to answering RQ1 and RQ2, as they reflect decisions made throughout the project.

During this stage, we also finalized the codebase and accompanying documentation. Our goal was to make it easy for others to reproduce our system and tests or build upon our work with minimal overhead. We provided setup instructions, usage examples, and architectural notes to assist anyone looking to extend our solution.

Chapter 4
Valkyrie MLS

This chapter presents Valkyrie MLS, our implementation of MLS for FFI's Valkyrie swarm system. Below, we describe the system's high-level structure and the key design choices that shaped it. The source code of Valkyrie MLS can be found on Github[1].

4.1 System Overview

Valkyrie MLS functions as a cryptographic middleware positioned between the swarm application and the radio on each drone. Its primary goal is to enable secure communication within the Valkyrie swarm, which it achieves using MLS. We use the `MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519` ciphersuite, the default choice in OpenMLS. Our system, depicted in Figure 4.1, includes the following components:

`Router`: Acts as the central unit of our system, routing messages between the drone application process, the radio interface, the `CorosyncHandler`, and the `MlsEngine`.

`MlsEngine`: Maintains the state of the MLS group, validates fresh key material introduced to the group, and handles encryption and decryption of messages. It uses the OpenMLS API for core MLS operations while applying custom logic and policies suited to swarm communication.

`CorosyncHandler`: A custom wrapper that maps Corosync events to internal message-handling logic. It notifies the router of incoming messages from Corosync and informs Corosync of outbound MLS configuration messages.

`ValidationFunction`: Implements the validation mechanism for the AS. It takes a credential as input and returns a boolean indicating whether the credential and its associated key material are valid.

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

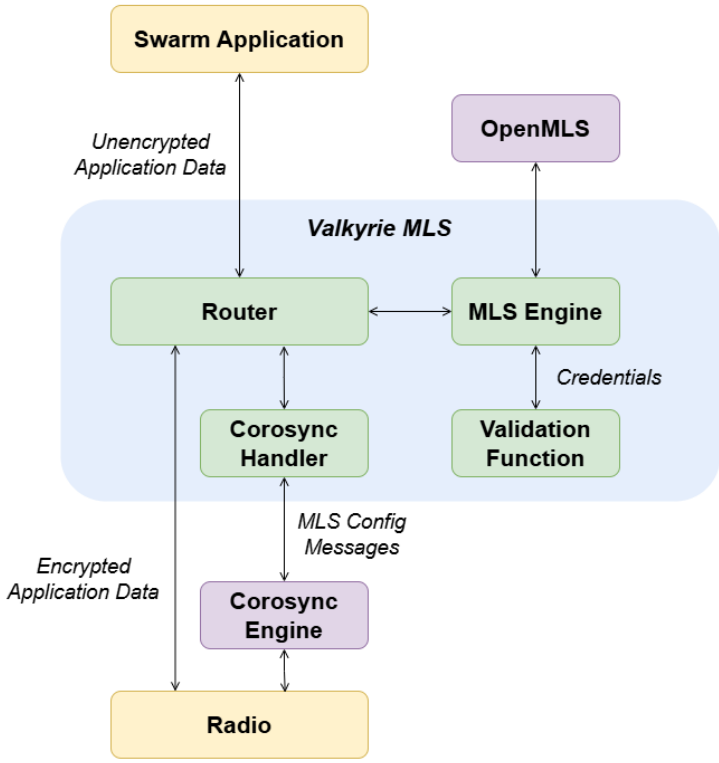


Figure 4.1: Detailed overview showing internal and external components of Valkyrie MLS. The yellow components highlight existing swarm components, the purple components highlight external processes our system depends on, and the green components highlight the custom components built through our development.

Our system relies on the presence of three external components:

Swarm Application: An independent process running on the drone, responsible for swarm-specific computations and control. It outputs coordination data to other drones and receives similar input in return.

Radio: The hardware component that provides the low-level network functionality required to communicate with other drones.

Corosync Engine: An instance of the Corosync application is expected to be running prior to launching our system. It operates as a standalone process alongside Valkyrie MLS. It should not be confused with the **CorosyncHandler**, which is a different component placed within Valkyrie MLS.

At the time of writing, OpenMLS is at version 0.6 [2], which we use as the basis of our system.

Upon system initialization, configuration parameters (e.g., node ID or credential type) are parsed and loaded from a file, then provided to other components through a shared configuration structure. The `MlsEngine` loads signature keys and its credential, and creates an MLS group consisting solely of itself. Once the `MlsEngine` is initialized, the `Router` is created. Upon initialization, the router spawns a thread for the `CorosyncHandler` and sets up the communication sockets used to send and receive messages. Once initialized, it enters an idle state, and handles all incoming and outgoing messages sequentially. The system can then add or join other nodes as it discovers them (see Section 4.5 for details).

When a drone needs to communicate information to the rest of the swarm, the application process sends data through an inter-process communication channel. Currently we use standard network sockets, but in theory any inter-process mechanism (e.g. Unix domain sockets or shared memory) could be used. The router receives the outbound data and forwards it to the `MlsEngine` for encryption. Once encrypted, the router sends the data to the radio, which transmits it over the network.

Incoming data is handled in a similar fashion. The router receives encrypted data on a designated socket, decrypts it, and then forwards the result to the application process.

The `MlsEngine` may fail to decrypt incoming data. If the failure is due to an internal validation error, such as an out-of-sync epoch state, a timer is started to signal the potential need for a self-healing mechanism. This helps detect drones that may have become desynchronized from the rest of the swarm and allows the system to recover from inconsistent group state.

If the error is caused by an unrelated issue, such as a message from an unknown group, the message is discarded. This suggests the message was not intended for the current node. We account for the possibility of arbitrary or irrelevant data arriving at any time, since the system operates in a broadcast environment.

When a MLS configuration message is received from the `CorosyncHandler`, it is passed to the router, which forwards it to the `MlsEngine` for further processing. The `MlsEngine` handles all group state changes (e.g., *Add*, *Remove*, *Update*) after validating the sender and associated key material through the `ValidationFunction`.

The router also runs a cyclic event loop that drives periodic MLS operations. This loop checks for pending `KeyPackages` to be added, removes nodes scheduled for removal, and issues periodic updates. Each of these actions invokes the corresponding

[2] https://github.com/openmls/openmls/releases/tag/openmls-v0.6.0

function in the `MlsEngine`. If any of them result in an outbound message (e.g., a `Commit`), the message is returned to the router and sent to the network via the `CorosyncHandler`. These automatic procedures are described in more detail in Section 4.5.

4.2 Design Choices

Great systems rarely come together by chance, but rather emerge from a series of thoughtful decisions. In this section, we outline the architectural choices behind Valkyrie MLS, with a focus on how they support robustness, adaptability, and long-term maintainability.

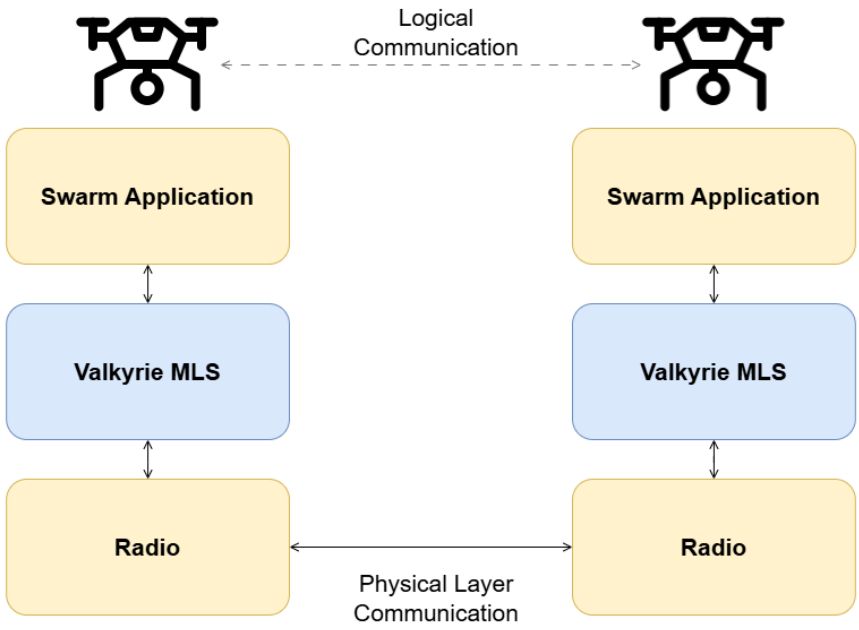


Figure 4.2: Overview of communication in the Valkyrie swarm, showing Valkyrie MLS as middleware between the application and radio.

Modular Component Design

We designed our system to be modular by clearly separating responsibilities across components and using well-defined interfaces. Since both the Valkyrie swarm and the Flamingo drone platform continue to evolve (see Section 2.1.3), our solution is built to adapt to changes in both the swarm application and the underlying network radio hardware.

Furthermore, we keep the internal coupling of components loose. This allows us to replace relevant parts of the AS (`ValidationFunction`), DS (`CorosyncHandler`), or even the MLS API (OpenMLS) with minimal effort.

Figure 4.2 shows the system overview. The swarm application runs as a separate process and sends opaque data, which we treat as a byte stream without interpreting its structure. This stream is transmitted over the radio, with our Valkyrie MLS acting as a middleware layer that encrypts outgoing and decrypts incoming data.

Sequential Handling of Data

To maintain a consistent MLS group state, all incoming messages are processed sequentially rather than concurrently. In a distributed swarm, handling multiple events in parallel, such as an application message and a `Commit`, can lead to race conditions or epoch forks. By enforcing sequential processing, we mitigate these concurrency hazards.

Our system is built around an event-driven loop that uses the `select!` macro from the Tokio-rs crate [Tok25]. The `select!` macro works like an elevator: idle until a request arrives, serves it fully, then returns idle, ensuring one event is handled at a time. This allows the system to listen for multiple asynchronous inputs at once, while ensuring that only one branch executes when an input becomes ready. The `select!` call is placed inside an infinite loop, so the system repeatedly waits for an event, processes it to completion, and then resumes waiting for the next one.

The `select!` macro monitors several input sources:

- Plain application messages from the swarm process
- Encrypted messages received over the network
- Protocol messages (e.g. `Commits`) from other MLS group members

When an event arrives, the appropriate `select!` branch executes its handler. Depending on the event type, the system either applies changes to the group state, such

as processing a `Commit`, or forwards application data, for instance by delivering a decrypted message to the application layer.

Because message arrival is nondeterministic and external, we rely on asynchronous futures to manage event handling efficiently. Each branch in `select!` awaits its corresponding future, and once one resolves, its handler runs to completion before the loop resumes.

Supporting Additional Input Streams

Our modular design makes it straightforward to integrate new input streams. For example, Marstrander encountered challenges integrating a drone video stream, as it originated from a separate process [Mar23b]. In a tightly coupled system, such integration can be difficult.

Our system avoids this problem. Because each input source communicates of a dedicated socket, we can add new streams by simple adding a corresponding handler at startup, without requiring any significant architectural change.[3] This flexibility reinforces our goal of building a system that is easy to extend and maintain.

Two Data Flows

We handle application messages and MLS handshake messages differently because their delivery requirements differ [BBR+23]. Application messages can arrive out of order, while MLS messages must arrive in the correct sequence.[4]

To handle this, we separated the two data streams. This allows us to process non-critical application messages in any order, while enforcing strict ordering for critical MLS configuration using Corosync. The separation is implemented using two sockets for inter-process communication between the radio and the MLS component, as shown in Figure 4.1.

As each message arrives on a dedicated socket, we identify its source. The router then forwards it to the correct handler inside the `MlsEngine`. The router remains agnostic to OpenMLS internals, using only enough logic to direct messages appropriately (application data vs handshake events).

[3] We currently use network sockets for inter-process communication. The Valkyrie team has indicated plans to transition to shared memory. To support this, we will need a notification mechanism that alerts `select!` when new data becomes available in the shared buffer.

[4] Although strict ordering isn't required, decryption depends on each member's secret state. Each message derives a new symmetric key using the KDF specified in the chosen ciphersuite. The *max forward distance* and *out-of-order-tolerance* parameters control how tolerant the system is to skipped messages.

Immediate Commit on Group Changes

OpenMLS supports two approaches for applying group changes: *immediate operations* and *propose-then-commit*. In the immediate operations approach, the initiating node applies the change to its local state and directly issues a `Commit` to the other group members. Alternatively, in the propose-then-commit approach, nodes can suggest changes by sending `Proposals`. These proposals are later bundled into a `Commit`, issued by any group member, that collectively applies the proposed changes. If a node receives a `Commit` referencing missing proposals, OpenMLS returns an error.

The propose-then-commit approach offers flexibility, allowing proposals to arrive in any order before the associated `Commit`. However, it also raises the question of who should send the `Commit`. One possible solution is to designate a group leader who periodically broadcast a `Commit` bundling all proposals collected during the last epoch. This, however, introduces centralization, requires leader election, and entails mechanisms for failures if the leader becomes unavailable. Moreover, it increases message overhead, since each change requires both a `Proposal` and a subsequent `Commit`. While batching proposals into a single `Commit` reduces the number of messages, the approach still introduces additional coordination overhead and implementation complexity.

To minimize system complexity and avoid the need for leader election or additional message overhead, we adopt immediate operations in our design. This approach avoids the coordination burden associated with proposal handling. For example, if *drone 1* receives *drone 2*'s `KeyPackage`, it can immediately add *drone 2* and issue a `Commit`. A drawback is that loss of this `Commit` causes nodes to diverge: the initiator and recipients move to a new epoch, while others remain in the old one. This issue, however, is not exclusive to immediate operations, as loss of a `Proposal` under the alternative approach also leads to failure once the corresponding `Commit` is processed.

4.3 Implementation of the Delivery Service

As described in Section 2, the DS in MLS is responsible for (1) routing messages between clients and (2) storing and providing `KeyPackages`. In traditional MLS deployments, this functionality is centralized on a trusted server. Since Valkyrie MLS operates in a peer-to-peer setting, specifically within a distributed UAV swarm, we decompose and re-implement these DS responsibilities in a decentralized fashion.

Key Package Distribution

We have no central server to use for storing and retrieving `KeyPackages`. `KeyPackages` are broadcast as soon as they are created, and each node processes any received `KeyPackages` immediately upon arrival. This mechanism ensures that group additions can proceed without requiring a central persistent storage backend, as long as swarm members receive the necessary packages in a timely manner. The decision is based upon the assumptions that `KeyPackages` are intended for one time use.

Message Routing and Total Order

Maintaining a consistent group state across all clients requires that MLS configuration messages are processed in the exact same order to all participants. This property is critical to prevent forks where cryptographic states diverges. As stated in our scope (Section 1.2), we opt to use the Totem protocol to achieve this in our system. The motivation for using this over other consensus mechanisms is discussed in [Mar23b]. Although a detailed comparison is beyond the scope of this work, alternatives are briefly discussed in Chapter 6.

To implement this, we distinguish between two types of messages:

- **MLS Configuration Messages**: These consist of handshake messages (`Proposals` and `Commits`), `Welcome` messages, and `KeyPackages`.
- **Application Messages**: These are regular messages passing between the drone application and the radio, such as telemetry data or control commands. They do not affect the MLS state and are more tolerant of delay or loss.

We send these MLS configuration messages using Corosync's Closed Process Group (CPG) module to enforce strict guaranteed delivery with total agreed ordering using the Totem protocol. Application messages are delivered over a separate, unordered multicast channel, as they do not affect the cryptographic state and are more tolerant of latency and loss. Furthermore, we tweak the parameters out-of-order-tolerance and maximum-forward-distance to better handle application messages that are delivered out of order.

Use of Corosync Closed Process Groups

In our system, we integrated Corosync, whose CPG module provides reliable multicast and totally ordered message delivery, which is a necessary requirement for MLS configuration messages. This choice improves robustness and reduces overhead compared

to previous work on Valkyrie [Mar23b], which involved manually implementing the Totem protocol. That effort quickly revealed the complexity and fragility of building distributed consensus from scratch. While Corosync is not fully transparent and limits low-level visibility (for example, token ownership during runtime), it integrated smoothly with our message formats and state logic without introducing additional complexity. This tradeoff allowed us to offload the challenges of implementing the Totem protocol from scratch.

Consistency Model

Designing the DS involves tradeoffs between availability and consistency during network partitions. From the CAP theorem [Bre00], we have two choices:

- **Strong Consistency (CP)**: Guarantees a consistent message order for all clients but may delay or reject messages during a partition. To achieve this, the epoch state should ideally not evolve during a partition or under stable network conditions where delivery guarantees are uncertain.
- **Eventual Consistency (AP)**: Prioritizes availability by allowing messages to be processed even if they arrive out of order or inconsistently during network partitions.

In Valkyrie MLS, we adopt a mixed solution by splitting the data stream in two. Handshake messages are transmitted through Corosync using a CP-model to ensure the consistency of the cryptographic group state. In contrast, application data is sent over the regular multicast channel that tolerates eventual consistency.

4.4 Implementation of the Authentication Service

As outlined in Section 2.3, the MLS protocol requires an AS to operate securely, but leaves the implementation to be decided. According to the architecture document [BRO+25], an AS must support the following operations:

1. Issuing new credentials with a defined lifetime
2. Validating credentials against a claimed identity
3. Determining whether two credentials represent the same logical client
4. Optionally revoking credentials that are no longer authorized

Our implementation addresses requirement (1) and (2). Requirement (4) is not implemented, but we discuss theoretical solutions below and further in Section 6. Requirement (3) is less relevant in our context, as UAVs do not operate across multiple clients or devices. Nevertheless, it is still partially addressed. Nodes in the system are identified by a global ID, and therefore all credentials using the same ID represent the same client.

Design Considerations

There are multiple approaches to implementing an AS. The MLS architecture document [BRO+25] outlines several options:

- Traditional Public Key Infrastructure (PKI), where a trusted Certificate Authority (CA) issues and signs credentials.
- Fingerprint-based key verification, as used in systems such as the Signal Protocol [CCD+17].
- Transparency logs (Key Transparency), based on the work on CONIKS [MBB+15].

Of these, traditional PKI is the most natural fit: it provides a clear, well-understood framework for verifying credentials, aligns with a model where certificates and keys are preloaded, and supports hierarchical trust via CAs. While more decentralized solutions may be attractive in some settings, we believe a PKI offers the right balance between security, scalability, and operational simplicity in a UAV swarm context.

The MLS specification includes X.509 certificates as a standard credential type, which is part of the PKI standard. In a PKI, a CA issues certificates for public keys and signs them with its private key. A client signs its messages using its own private key and provides the certificate to prove its claimed identity. This proof includes the certificate itself and a certificate chain leading to a trusted CA. In Internet PKI, this typically means a root CA, whose root certificate is preinstalled on all clients.

Marstrander notes that this requires connectivity to a centralized CA for issuing and verifying credentials [Mar23b], which is disadvantageous for autonomous swarm operation. To solve this, Marstrander proposes preinstalling root certificates on the drones. The swarm operator (as root CA) signs each drone's certificate before flight. This yields shorter certificate chains (each drone has only a single client certificate signed by the trusted CA). Issuance and revocation both require connectivity to a CA. In our case, we assume issuance is done pre-flight.

In the case of a compromise, it is necessary to revoke the old certificate and establish a secure channel to generate new keys and a new certificate, which is difficult in-flight. A solution could be that the GCS maintains a revocation list and distributes signed updates to all swarm members. Members could then request the most recent version of this list from each other.

There is also an important perspective on how we conceptualize the swarm system [Mar23b]. Marstrander distinguishes between two types of operational deployments: the standalone swarm and the joint swarm system. These models influence the design and implementation of the authentication service, particularly in how control is managed prior to deployment.

In a standalone swarm, the system consists of a fixed number of UAVs controlled by a single, dedicated GCS. This central unit is responsible for issuing certificates, and its keys are preinstalled on all drones. If a compromise occurs, the GCS can distribute a signed revocation list, indicating which keys or identifiers are no longer valid. After each mission, the swarm is reset and drones are rekeyed, simplifying key lifecycle management and limiting long-term exposure.

In contrast, a joint swarm system assumes a larger, more dynamic environment, where multiple swarms (potentially operated by different GCSs) coordinate within the same operational area. For instance, two separate military units might deploy swarm teams concurrently and require secure communication between them. This setting demands greater interoperability and mutual authentication across domains.

Supporting such operations requires a shared PKI infrastructure for managing certificates, identities, and revocations. One possible solution is to pre-install all relevant CA certificates on each drone. However, this increases the attack surface: a single compromised CA could undermine the security of the entire system. This risk highlights the need for updating key material and certificates during flight, not just pre-deployment.

While both models are relevant for future swarm operations, current real-world deployments more closely align with the standalone model. Its reduced architectural complexity makes it more practical today. For this reason, we implement a custom authentication service inspired by PKI, tailored to the simpler standalone swarm model.

Implementing an Authentication Service

The AS is split into two conceptual components:

Issuance: We assume that UAVs can be preconfigured with credentials before deployment. This is a realistic assumption in tactical environments, and it allows the GCS operator to act as a CA, generating and signing credentials and key material before flight.

Validation: Each UAV runs a lightweight credential validator locally. Upon receiving a credential from a peer, it checks whether the credential was signed by a trusted authority and is still valid.

Credential Type and Structure

In MLS, each group member presents a credential that provides one's identity and binds it to the member's signing key [BBR+23]. The identities and signing key are verified by the Authentication Service. It is up to the application to decide which identifiers to use at the application level [BBR+23]. For our use case, we want credentials contain information about the subject (its ID), the public key that it holds, the validity of the credential, a signature over it from a trusted issuer and the identity of the issuer who signed it.

In terms of credentials, there are several ways to go about to structuring them. In OpenMLS, we have two specified credential types, in addition to an application specific credential, as described earlier in Section 2.5.1 and in Code 1.

Using the `BasicCredential` leaves us with no authentication security, as this is exposes only an identity to represent a client and does not contain any key material or any other information. This leaves us with the choice of implementing our own credential structure based on either X.509 or a custom scheme defined by ourselves.

While X.509 is widely supported and aligns with the long-term direction of the MLS standard, it presents notable drawbacks for our use case. The typical certificate structure includes fields that are irrelevant in a swarm context, resulting in unnecessarily large payloads. Prior work has reported certificate sizes of up to 2600 bytes [Mar23b], which could lead to network congestion given that credentials are transmitted with every `KeyPackage`, `Add`, and `Update` message. Moreover, although X.509 is not yet supported in OpenMLS, it is expected to be in the future. This creates a risk that any custom implementation we add now may become obsolete.

As an alternative, we opt to design a lightweight, custom credential format that minimizes size while preserving the necessary authentication guarantees. This

approach allows us to include only the fields relevant for our system, leading to a more efficient wire format. Although this shifts the responsibility of ensuring authenticity and integrity to our own implementation, we consider this a worthwhile trade-off for the swarm setting. Our custom credential, `Ed25519Credential`, is tailored specifically to the constraints and requirements of a UAV swarm.

Code 3 `Ed25519Credential` Structure

```
pub struct Ed25519Credential {
    pub identity: u32,
    pub credential_key_bytes: [u8; 32],
    pub not_after: u64,
    pub signature_bytes: [u8; 64],
    pub issuer: u32,
}
```

The `Ed25519Credential` is built on the notion of the Ed25519 signature scheme [JL17] and is illustrated in Code 3. The choice of using Ed25519 signatures was made due to its widely adopted nature, its small key size combined with the fact that it is used as the signature type of our chosen ciphersuite in OpenMLS. The credential contains the minimal required fields needed for our authentication service to validate a signature key pair and should be interpreted as follows:

- `identity`: A 32-bit identifier for the UAV. This fixed-size type provides a predictable layout and matches Corosync's node ID format (which also uses `u32` for IDs).
- `credential_key_bytes`: The UAV's Ed25519 public key.
- `not_after`: UNIX timestamp indicating when the credential expires.
- `signature_bytes`: A 64-byte Ed25519 signature over the credential payload.
- `issuer`: The ID of the credential issuer (e.g., a swarm administrator or trusted CA).

The signature is produced using the Ed25519-dalek library[5], which is also used by OpenMLS. The payload is hashed with `SHA-512` and signed with the issuer's expanded secret key. The signed message m is the concatenation of the credential's three core fields:

$$m = \texttt{identity} \,\|\, \texttt{credential\_key\_bytes} \,\|\, \texttt{not\_after}$$

[5] https://docs.rs/ed25519-dalek/latest/ed25519_dalek/

Trust Model and Credential Validation

Our system follows a simple yet effective trust model: a UAV only accepts credentials signed by a known and trusted issuer. Prior to deployment, a set of trusted issuer public keys are stored on a UAV. Validation of a received credential proceeds as follows:

1. Verify that the `issuer` is among the trusted preloaded signature keys.
2. Reconstruct the signed payload from `identity`, `credential_key_bytes`, and `not_after`.
3. Verify the signature using the trusted public key.
4. Ensure the credential is not expired.

Credentials that pass all checks are accepted. If the issuer is unknown or any checks fail, we mark the credential invalid and discard it.

Credential Issuance

Credential issuance is performed by the swarm operator or a trusted authority as follows:

1. Generate an Ed25519 key pair.
2. Construct the credential payload:
   $m = \texttt{identity} \| \texttt{credential\_key\_bytes} \| \texttt{not\_after}$.
3. Sign m with the issuer's Ed25519 private key.
4. Generate a signed credential object using the signature and other relevant fields. Store the signed credential on the target UAV.

There are multiple ways to generate Ed25519 key pairs, but in our case, we use an external toolkit. Specifically, we choose `OpenSSL`, a widely used open-source cryptographic library and command-line tool [The23]. `OpenSSL` provides safe and convenient support for key and certificate creation, among many other things.

To generate an Ed25519 key pair, we use the following commands:

```
openssl genpkey -algorithm Ed25519 -outform DER -out <privkey>
openssl pkey -in <privkey> -inform DER -pubout -outform DER -out
    <pubkey>
```

The first command generates a new Ed25519 private key in DER format, making it easy to read from file and minimizing its storage size. The second command derives and exports the corresponding public key, also in DER format.

4.5 Automatic Procedures

The OpenMLS API offers a convenient interface for executing group operations via the `MlsGroup` object, which encapsulates the full group state. While the `Add`, `Remove`, and `Update` operations are straightforward in isolation, automating them within a dynamic, distributed swarm is non-trivial. The GCS cannot oversee all membership in real time, so questions such as *when to update*, *who should add whom*, and *when to remove a member* require automated decision-making and clearly defined policies.

Before delving in to automating the different procedures, we begin by reiterating two types of group abstractions for our system:

Totem (CPG) group: the network membership group, defined and managed by Corosync. We refer to Totem as the general ordering and membership protocol and Corosync as the implementation of it.

MLS group: the cryptographic group in MLS. The cryptographic state of the group changes as nodes perform `Add`, `Update` and `Remove` operations on the group.

Operations in Totem is already automated and assumed [AMM+95; Pro24]. Furthermore, Totem, and consequently also Corosync, is assumed to be self-healing, meaning that if network subgroups are formed due to temporary loss of connectivity, they will automatically merge once the nodes are able to communicate with each other again. Our overall goal is to automate these procedures in our cryptographic group as well.

4.5.1 Automatic Updates

Updates are the primary mechanism for refreshing cryptographic key material and play a central role in maintaining PCS. This operation can be automated in several ways, but two natural strategies emerge: triggering updates at fixed time intervals or after a certain number of messages have been sent.

While message-based updates may offer a fine-grained approach, the number of messages exchanged in the swarm can vary significantly depending on the application—and may increase substantially if video streaming is incorporated. Since we lack precise assumptions about message frequency and must accommodate potentially

high-throughput scenarios (e.g., video streaming), we choose a time-based update strategy. This approach provides a predictable and application-agnostic mechanism for key renewal.

In our system, the update interval also serves as a baseline for other operations. For instance, `Commits` that contain an `Add` or `Remove` are also processed during this scheduled update. Marstrander's earlier work [Mar23b] triggered operations on the Totem token: `Commits` were only sent when holding a token, providing an implicit global lock. This prevented concurrent commits by different nodes. Without such coordination, simultaneous `Commits` could fork the cryptographic group state.

Since we use Corosync as an abstraction over Totem, we no longer receive explicit notifications about token ownership. To reduce the risk of concurrent `Commits`, we staggered the update timers of nodes. This naive approach is fragile, particularly as node count increases or update intervals shrink, and should be revisited in future work to include more robust coordination mechanisms.

4.5.2 Automatic Removal

Removal is itself a topic for a master thesis. What should cause a removal from the cryptographic group? We want to remove any node that loses connection or is removed from the swarm. We only discuss the challenge of in-flight removals, and choose to focus on the removing nodes based on loss of connection in our implementation. One way of going about this, is using *heartbeat* messages, used to detect link failures. Assuming heartbeat messages are already part of the swarm application, we could detect a lost link by monitoring application messages. Each drone could track the time since last message from every other drone (similar to [Mar23b]). In this work, four events were presented as reasons to remove a member:

1. **No recent `Update`:** A member has not sent an `Update` in a long time.
2. **Left Totem group:** A member is no longer part of the Totem group.
3. **No application traffic:** A member is not sending application messages.
4. **Duplicate member:** A node with the same identity already exists in the MLS group.

In our proposed system, we aimed to omit some of these events to simplify the automatic removal functionality. The first two events are sufficient reasons to remove a member. If a member does not update within a given time period, this breaks the principle of per consistent state. Additionally, if a member is no longer in the Totem

group, it should also be evicted from the MLS group, as these group abstractions should remain synchronized. If a member is not sending application messages, this is a swarm application issue and not something that inherently compromises the security goals of MLS. If we already receive `Updates` from a drone, we know that we can receive data from it, meaning the communication link is active. Therefore, the swarm application itself should decide how to handle the lack of application messages, as this may indicate that there is something else wrong with the drone.

Duplicate members should not be allowed in the group. This is already enforced when the credential is validated. OpenMLS will reject or error out if it detects duplicate signature keys on add. This kind of invalid group state should therefore never occur.

We propose the following: if a node is evicted from Totem or fails to send timely updates, it should be marked for eviction from the MLS group. Since Corosync already notifies us of Totem evictions, we can propagate the corresponding node identifiers to the `MlsEngine` to schedule their removal. By using a consistent identifier for each drone across both the Totem and MLS groups, the `MlsEngine` can reliably schedule the eviction of nodes from the MLS group during the next update cycle.

To detect stale members, each drone locally tracks the latest updates received from other group members. If a member has not sent an update within, for example, three times the expected update interval, it is considered inactive. This check is performed during the drone's own update cycle.

To avoid removing the same node twice, we extend the system's functionality to handle concurrent removals more gracefully. For example, in a group of three drones *node 1, node 2, node 3*, if *node 3* is evicted from the Totem group, both *node 1* and *node 2* will mark it for removal from the MLS group. If both attempt to perform the removal, one will fail. To prevent this, each drone checks incoming `Commits`. If a `Commit` contains a removal targeting a node already marked as pending, the drone clears that entry from its pending removals before applying the `Commit`. It does not matter which node ultimately issues the `Commit`, as long as duplicate removal attempts are avoided. By checking all incoming `Commits` for overlapping removals and cleaning up any matching pending entries, we prevent errors and ensure a consistent group state.

Another aspect is marking nodes for removal in the event of a compromise. Deciding if a node is compromised is outside this thesis's scope. However, the GCS can mark nodes for removal via a manual command (our API supports this). For security, we should authenticate GCS commands (e.g. with an extra credential) to prevent unauthorized removals.

4.5.3 Automatic Add and Discovery

Leveraging evictions in Totem, combined with the absence of updates, provides a solid baseline for automating removals in our system. However, this logic does not extend as easily to automating add and discovery. Totem only informs us of the ID of a node that has joined the group, but to add that node to the MLS group, we also need its current `KeyPackage`, including its credential. Since there is no central service to query `KeyPackages` by ID, we must rely on an alternative method: either querying the node directly or having it broadcast its `KeyPackage`, a classic *push-or-pull* scenario. While both are valid approaches, broadcasting is preferable in our case, as the `KeyPackage` must be sent regardless, and the underlying radio network already supports broadcast communication. Therefore, each node should periodically broadcast its `KeyPackage` to enable discovery by other drones. These broadcasts are sent via Corosync, meaning the message will be handled as a DS message upon receipt, and also confirms that the sending node is already part of the Totem group.

Add

A valid `KeyPackage` should result in a `Welcome`. That is, if *node 1* receives a `KeyPackage` from *node 2*, it should first validate the credential and then generate a `Welcome` message for *node 2*. To prevent concurrent `Add` operations, each node first stores received `KeyPackages` in a local list. The first node to enter its update cycle will process all pending `KeyPackages` and issue corresponding add proposals. The remaining nodes, upon receiving the resulting `Commit`, check whether it contains an add proposal for a `KeyPackage` already present in their *pending_key_packages* list and remove it accordingly, similar to how pending removals are handled. Additionally, before storing a received `KeyPackage`, nodes must ensure that its credential is not already part of the current group.

Join

When a node receives a `Welcome` message, it must decide whether to accept the invitation. To answer the question of *who should join whom*, we define the following policy.

Each node should join the group if it includes the GCS, provided the GCS is present in the network. This requirement was expressed by FFI's Valkyrie team during our discussions. The rationale is straightforward: The GCS should maintain communication and control over all available drones. Therefore, upon receiving a

`Welcome` message, a node checks whether the group includes the GCS's credential identifier. If it does, the node accepts the invitation.

This policy encourages convergence toward a stable group centered around the GCS, assuming it is reachable. To reduce unnecessary broadcast of `KeyPackage` messages, nodes already in a group containing the GCS are instructed not to broadcast their `KeyPackage`. This design assumes the presence of only one GCS in the network. Supporting multiple GCS units would require additional parameters, such as one to determine which GCS to prioritize.

In cases where the GCS is not present, for instance due to lack of radio coverage, we still require drones to communicate securely using MLS. In such situations, the swarm may partition into one or more subgroups, where each subgroup is a subset of drones sharing the same cryptographic state, i.e., they belong to the same group and epoch. To maintain coherence, all nodes should attempt to join the largest subgroup currently available.

Finally, we must account for scenarios where multiple groups are of equal size. In such cases, we need a tiebreaking mechanism. The simplest strategy is to have nodes join the group that includes the member with the lowest identifier, assuming that node IDs are numeric and globally comparable.

4.5.4 Challenges with the Policies Above

Although the ideas above are sensible, a few complications arise in practice. The first problem is that even if the GCS is present, multiple subgroups may coexist temporarily. A node might first join a group without a GCS before eventually migrating to the GCS group. While this still leads to eventual convergence, we aim to reduce such intermediate reconfigurations. To do so, nodes inspect the Totem membership list before initiating `Add` operations. If the GCS is detected, additions to other subgroups are suppressed. Optionally, this check can also be performed upon receiving a `Welcome`, though it is redundant if all nodes follow the addition policy correctly.

Expanding on this, we could outline a theoretical attack for this, where someone poses as a GCS. To avoid this, we use the same assumption as for the AS, where the ID of the GCS is defined in the configuration loaded on startup. To ensure this is the correct GCS, we need some sort of additional authentication mechanism for the Totem membership. Corosync supports both encryption and authentication[6]. This could also prevent forms of Denial-of-Service (DoS) attacks, where an attacker

[6]Corosync defines optional configurations within its directive, including configuration for secrecy and authentication: https://manpages.ubuntu.com/manpages/focal/man5/corosync.conf.5.html

would try to overwhelm the Totem service with never-ending messages to deny its availability. Unfortunately, we did not have time to incorporate this in our system.

The second problem is multi-faceted:

- Nodes cannot determine how many subgroups exist at a given time.
- Nodes cannot confirm whether a peer has accepted their `Welcome`, which can lead to incorrect assumptions about group size.
- Our current implementation adds nodes in batches during update cycles, which may further complicate the problem of incorrect assumptions a node can have about its group size.

Each drone has limited visibility into the system: it knows (1) its own MLS group size, (2) the size of any `Welcome` message's group it receives, and (3) the total number of nodes participating in the Totem group (i.e., the underlying Corosync cluster). Given this limited perspective and the absence of a centralized GCS, it is challenging to guarantee stable convergence of membership.

One way of going about this is to apply policies also when adding new members. We can infer that we are the largest MLS group in the network if our MLS group size $> \frac{Totemgroup}{2}$. If we are in such a group, then we can add other nodes.

A problem arises when no majority group exists, as this prevents any node from adding others. In such scenarios, additional information is required to identify the largest group. To convey this information, nodes would need to include extra data when sending `KeyPackages`. Although this is technically feasible, for example by adding a custom header with bits indicating the current group size, it introduces unnecessary complexity. Instead, we assume that nodes join the network gradually rather than forming multiple subgroups at once, and therefore choose not to complicate the design further.

To ensure convergence toward a stable group, we expand our policy for adding members by introducing a simple rule. Specifically, we add an exception to the rule of only adding nodes when belonging to the majority group: if a node's current MLS group contains the lowest ID in the network, it may also add other nodes to its group. The lowest ID present in the network is obtained from the Totem membership list. While this approach may not be optimal, it ensures convergence toward a stable group, either the majority group or the one containing the lowest ID.

We summarize the rules for broadcast, add, and join behavior based on each node's role:

- **We are the GCS or part of a group that includes the GCS:**
  - Does *not* broadcast its `KeyPackage`.
  - *Adds* other nodes.
  - Ignores incoming `Welcome` messages.
- **We are in a majority group (over half of the Totem group):**
  - Broadcasts its `KeyPackage`.
  - May add nodes if no GCS is present.
  - Accepts `Welcome` messages only from groups with the GCS.
- **All other nodes:**
  - Broadcasts its `KeyPackage`.
  - May add nodes if no GCS is present and they are in the same group as the lowest-ID node.
  - Accept `Welcome` messages.

Together, these rules makes the swarm converge to a stable group: priority goes to joining the GCS group first, then any majority group, and finally (if needed) the group containing the lowest-ID node. We assume only one GCS exists in the swarm.

4.6 Handling Crypto-State Forks

One issue we encounter in our system is crypto-state forks. A crypto-state fork happens when two nodes or more nodes apply a different `Commit` at the same time. This may happen if *node 1* applies a change before receiving the Totem send token, and before *node 1* is able to broadcast its `Commit` to the rest, another `Commit` is sent to the MLS group. This will then split the current MLS group, leaving *node 1* in another epoch than the rest. The same will happen if a `Commit` is lost or not received by some nodes. Testing revealed various scenarios and edge cases, but usually only one node is affected.

A key indicator of a fork is a `ValidationError` caused by epoch divergence on a drone. By default, we set `max_past_epochs` to 1, allowing drones to decrypt messages from up to one epoch in the past. This means that drones ahead by one epoch can still decrypt messages from a lagging peer. However, a drone that falls behind cannot

decrypt messages from nodes that are already one epoch ahead, as those messages rely on updated key material that the lagging drone has not yet received.

We propose a simple recovery method for those cases where a node lags one epoch behind the group. When a node detects that they are one epoch behind, it triggers a reset timer and removes itself from the group. It can then rejoin the group through the automatic discovery mechanism. If the node regains the ability to decrypt messages after starting the timer, for example if the `Commit` message is simply delayed), the node aborts the timer.

4.7 System Summary

In this chapter, we have described how Valkyrie MLS achieves authentication, message ordering and dynamic group management:

Authentication: The system includes a working prototype of an authentication service. It uses Ed25519 credentials and custom X.509 certificates, ensuring that only authorized drones can join the cryptographic group.

Message Ordering: Valkyrie MLS uses Corosync, which implements the Totem protocol, to ensure that the MLS configuration messages arrive in the correct order.

Dynamic Group Management: The system automatically detects new drones as they come within communication range, determines which node should add each new member, and then handles the group update process without manual intervention. It also includes a recovery mechanism for drones that fall behind in cryptographic epoch state.

Chapter 5
Testing and Results

Our testing combines unit tests, simulation-based evaluations, and real-world drone testing to evaluate the Valkyrie MLS system. We begin briefly outlining our unit testing, which serve as a baseline test suite to validate internal logic during development. We then move on to our physical testing with drones, which involves both functionality testing and performance testing. In our functionality testing we aim to verify the correctness and stability of our system under various network conditions, while performance testing aims to mainly measure CPU and RAM usage across varying protocol parameters. Lastly, we tackle various tests conducted in our simulated environment. These involve network testing, where we assess system behavior under artificial packet loss, as well as looking at message and credential sizes to measure expected overhead.

5.1 Unit Testing

We developed and maintained unit tests alongside the core components to validate the internal business logic and ensure correctness in isolation. Moreover, we created a comprehensive test suite in our code base to evaluate the core business logic of Valkyrie MLS throughout our development phase. This allowed us to verify and validate that secure group operations like **Add**, **Update**, and **Remove** worked as expected among multiple participants, and also included functional tests for the authentication mechanisms. The tests validate both protocol correctness and group coordination logic. The tests cover message sending, group operations, credential validation, and group joining behavior under various conditions needed for automatic procedures.

Figure 5.1: Physical testbed with three Flamingo drones from the testing conducted at FFI's location at Kjeller.

5.2 Physical Testing

We conducted functional and parameter testing at the FFI's facilities at Kjeller. There, we tested our system using actual Flamingo drones, focusing on confirming full-system functionality and measuring runtime resource usage (CPU and RAM) across different protocol configurations. These results also form the basis for performance evaluation. Although in-flight testing was not feasible due to practical constraints, we manually moved the drones to simulate dynamic network topologies and varying link qualities. We use the same test setup and environment for both functionality and performance tests during our physical testing, but different methodologies.

5.2.1 Test Environment

We set up a physical testbed consisting of three Flamingo drones as shown in Figure 5.1. Each drone is equipped with an NVIDIA Jetson Xavier NX with a six-core ARM processor running Linux for Tegra, and uses a Rajant radio for wireless communication. The drones communicate over the `eth0` interface within the IP network 192.168.12.0/24, using multicast on the network layer. Each node ID corresponds to the last 8 bits of its IP address, providing a straightforward mapping between identity and address. To port the Valkyrie MLS software to the drones, we

compile the software natively on each drone. Cross-compiling is also possible, but we have to account for the ARM architecture when doing so. As our host machine uses an x86 architecture, we find it easiest to rather compile it on the drones, allowing for altering files locally if needed.

To communicate, control, and oversee operations, we use an Ubuntu host machine acting as the GCS. It connects to the same IP network as the drones via a Rajant radio module. We control group membership remotely from the host through a command-line interface. We developed a custom API for Valkyrie MLS to facilitate sending commands such as `broadcast_key_package()`, `add_pending_key_packages()`, and `remove_member(LeafIndex)`, enabling manual management of the group's cryptographic state. This proved especially useful, as most of our tests required establishing a stable cryptographic group before beginning measurements. Additionally, we use `ssh` to monitor each drone's local view and apply configuration changes when necessary.

To simulate application-layer traffic, we use MGEN (Multi-Generator), a data generation tool developed by the U.S. Naval Research Laboratory. MGEN sends data to a local socket on each drone, which is read by the Valkyrie MLS process. This mirrors how an actual application would transmit data to Valkyrie MLS. The Valkyrie MLS process then encrypts the data and multicasts it to the group. Upon reception, the other nodes decrypt and process the message. By using MGEN, we can control both the number and size of the application messages, enabling consistent and repeatable testing.

5.2.2 Functionality Testing

We conducted functionality testing to verify that the system behaves correctly under various conditions. These tests focus on the drones' ability to form stable MLS groups, authenticate credentials, and dynamically manage group membership in both stable and unstable network environments.

The testing is divided into four parts. We begin by validating the manual functionality of the `Add` and `Remove` operations, along with automatic `Update` behavior. These tests use `BasicCredentials` to isolate core group operations. We then evaluate the system's authentication mechanisms by testing the `Validation Function` using `Ed25519Credentials`. Once both manual group control and credential validation are confirmed, we proceed to test the automatic procedures. First, we evaluate their behavior under stable network conditions, and finally, under unstable conditions that simulate dynamic link quality and partitioning.

Functionality Test 1: Manual ADD and REMOVE with Automatic UPDATE

This test aims to validate the manual control operations provided by the `MlsEngine`, ensuring that three UAVs can be manually configured to form a stable MLS group.

To isolate manual behavior, all automatic procedures are disabled. Before starting the test, we power on all three UAVs and confirm they are connected to the radio network. Each drone is configured with `CredentialType::Basic`, an `update_interval` of 10 seconds, and a correct node ID. We also verify that an instance of Corosync is running on each drone.

Once initialized, we confirm that each drone emits periodic `Update` messages and begins sending application data using MGEN at a rate of 1 kB per second. At this point, we do not expect messages to be decrypted correctly, since no MLS group has been formed.

Next, we instruct *drone 2* and *drone 3* to broadcast their Key Packages. We then command *drone 1* to add its pending Key Packages, thereby forming the group. We verify group formation by checking *drone 1*'s MLS group state and ensuring epoch consistency across nodes. We also confirm that application data and Updates are now correctly decrypted and processed by all participants. Finally, we remove *drone 3* from the group by referencing its index, and subsequently add it back again to test removal and re-addition.

This test confirms that manual operations function as intended. Drones are able to add and remove each other successfully, and epoch updates are correctly synchronized across the group. The automatic update mechanism performs as expected. However, when all nodes are initialized simultaneously, overlapping update timers can lead to crypto-state forks. To avoid this, we stagger drone initialization during testing. This behavior highlights the fragility of relying on desynchronized update timers.

Functionality Test 2: ED25519 Credential Validation

This test evaluates the `Validation Function` for `Ed25519Credential`. We aim to verify that `Commit` messages are accepted only when the credentials are valid and correctly issued.

The setup mirrors that of Functionality Test 1, but with the credential type set to `Ed25519Credential`. Prior to testing, we generate and issue credentials on the host machine, which acts as a CA, as described in Section 4.4.

To test credential rejection, we intentionally issue one drone an invalid creden-

tial—both by corrupting a valid credential and by issuing one from an unauthorized CA.

The test confirms that drones correctly accept and process only valid credentials. Group formation proceeds as expected when all credentials are valid, mirroring the results from Test 1. Invalid credentials are rejected, and no `Add` operation is performed. Although credential revocation is not yet supported, these results confirm that the implemented authentication service performs successful credential issuance and validation.

Functionality Test 3: Automatic ADD and REMOVE under Stable Network Conditions

This test validates that drones are able to automatically broadcast and discover each other, in accordance with our framework for automated procedures. The goal is to confirm that discovery leads to successful `Add` operations and the formation of a stable MLS group under stable network conditions.

We use the same setup as in Functionality Test 1, but enable all automatic procedures. The test is performed with both `CredentialType::Basic` and `CredentialType::Ed25519`.

Once the drones are initialized, they are expected to automatically form a group without manual intervention. We use MGEN to send application-layer traffic, allowing us to verify that the group has formed correctly and that messages can be decrypted by all members. To validate automatic `Remove` functionality, we terminate the Valkyrie MLS process on individual drones one at a time, and later restart it to test rejoining behavior.

Nodes that are powered off or disconnected are automatically removed from the group, and epoch advancement proceeds correctly among the remaining members. Upon re-initialization, the removed node is initially unable to participate but is automatically re-added after broadcasting its Key Package. This confirms that the automatic group maintenance logic functions correctly under stable network conditions.

Functionality Test 4: Automatic ADD and REMOVE under Unstable Network Conditions

This test assesses the system's ability to maintain secure group communication and recover from disruptions when drones experience unstable or intermittent network connectivity.

We follow the same procedure as in Functionality Test 3, but instead of shutting down drones, we physically move one drone in and out of coverage to simulate varying link conditions. Depending on its placement in the local terrain, the drone experiences:

- **Stable coverage:** Close proximity and no terrain obstruction.
- **Unstable coverage:** Medium distance and partial terrain interference.
- **No coverage:** Greater distance with significant terrain obstruction.

We observe how the system responds as the drone transitions between these zones, focusing on message decryption, epoch synchronization, and membership dynamics.

When moving *drone 3* rapidly in and out of coverage, the system correctly detects its absence and removes it from the group. This is triggered by a Totem eviction. Upon returning, the drone broadcasts its Key Package and is successfully re-added. During its absence, the remaining drones continue to decrypt each other's messages and maintain epoch consistency.

In some cases, *drone 3* re-enters coverage still believing it is part of the old group. Due to missed Update messages, its epoch state lags behind. This inconsistency is detected as a cryptographic fork, prompting the drone to self-remove and rejoin the group, after which synchronization is restored.

However, when the drone is moved more slowly or resides for extended periods in the unstable coverage zone, we observe group-wide desynchronization. This occurs even when two drones are in close proximity. The issue stems from Totem failing to maintain an operational state, which delays or blocks MLS configuration messages. In such conditions, encrypted application data cannot be processed, and automatic updates push nodes into diverging epoch states. Communication resumes only when the unstable node is either fully removed or re-enters stable coverage.

These observations highlight the dependency of MLS group stability on Totem's operational state and the sensitivity of handshake and update mechanisms to fluctuating network quality.

Findings from Functionality Testing

During a week of on-site testing and development at Kjeller, we achieved significant functional improvements to our system. Most notably, we successfully enabled the automatic addition and removal of drones in the swarm. When we physically removed a drone from coverage, the system reliably detected its absence via Totem eviction and removed it from the group. Upon re-entering coverage, the drone broadcast its

`KeyPackage`, and the swarm rediscovered it and automatically reintegrated it into the group.

Our testing also revealed key limitations. The use of unsynchronized Update timers led to cryptographic state divergence in some cases, making it difficult for groups to maintain a stable shared state. Additionally, we observed that unstable link conditions severely impact group coordination. Specifically, the stability of the MLS group is closely tied to Totem's ability to remain in its operational phase. When Totem becomes unstable, communication and group synchronization across the swarm degrade, regardless of the proximity between nodes.

5.2.3 Performance Testing

We now turn to performance testing of Valkyrie MLS, focusing on its resource usage under varying protocol configurations. The primary objective is to evaluate how the system impacts overall CPU and memory usage. To do this, we vary individual parameters in isolation and observe their effect on system performance.

The goal is to quantify CPU consumption during encrypted communication and compare our results to prior measurements reported by Marstrander [Mar23b]. This comparison provides insight into whether our OpenMLS-based system is lightweight enough for real-world use in drone swarms and how it stacks up against the previous MLS++-based implementation. Ideally, the added cryptographic overhead should be negligible.

Methodology

Each test is run under controlled conditions for 10 minutes, during which we vary one parameter at a time from Table 5.1, keeping all others fixed at their default values. We assume a steady-state MLS group at the start of each run, with three synchronized nodes communicating securely via a shared group key. Automatic add and remove operations are enabled, although self-healing (re-join) functionality is not included in this version[1].

Our choice of parameters and value ranges reflects operational realities and theoretical design trade-offs. For example:

- **Update Interval** determines how frequently the cryptographic state changes via `Commit` messages. We test a range from no updates to longer intervals (up

[1]We later developed the self-healing mechanism based on observations from these tests, and we verified it in Functionality Test 4.

to 100 seconds) to assess the impact on performance and group stability.

- **Max Past Epochs** affects memory usage and group resilience. Retaining past epochs enables nodes to recover if handshake messages are missed. We test configurations ranging from zero epochs (no state retention), to one epoch (a realistic baseline), and up to ten epochs to explore this trade-off.
- **Out-of-Order Tolerance** allows a node to decrypt messages received out of sequence. We vary this parameter from 5 to 500 to analyze the memory and processing overhead of storing extra keys, and its effect on CPU usage.
- **Maximum Forward Distance** defines how many messages a node can skip before decryption fails. This is relevant during traffic spikes or packet loss. We evaluate values from 0 (no tolerance) up to 100,000 to test the system's robustness under bursty or congested conditions.
- **Log Level** can affect CPU load due to runtime I/O. While logging is not required for swarm operation, it is useful for debugging and monitoring. We measure performance across verbosity levels ranging from `no logging` to `error`.
- **Application Message Size and Frequency** simulate different swarm workloads. MGEN limits messages to a maximum of 8 kB. We test message sizes of 1, 4, and 8 kB, and frequencies of 2, 10, 50, and 300 messages/s to represent a spectrum from light control traffic to high-rate data streams (e.g., compressed video). This helps assess whether fewer large packets are more efficient than many small ones.

During each run, we log CPU and memory usage of both the Valkyrie MLS and Corosync processes using `pidstat`, sampling once per second. This provides a high-resolution view of runtime overhead. Results are written to CSV files for post-processing and analysis. In all tests, we also verify that MLS group functionality is retained.

Table 5.1 summarizes all tested parameters, their value ranges, and default settings.

Results

The baseline performance test using the default configuration values for all parameters from Table 5.1 show that Valkyrie MLS introduces minimal overhead under typical conditions. The combined CPU usage of Valkyrie MLS and Corosync averages 4.9% on one core, as depicted in Figure 5.2, corresponding to less than 1% of total CPU capacity on the six-core system. `valkyrie-mls` accounts for most of the load. RAM usage remains stable at around 47,000 kB for the duration of the tests, well below the system's 8 GB limit.

Parameter	Default Value	Explanation
Update Interval [s]	[No updates, 3 **10**, 100]	Time between periodic `Update` messages sent by each node.
Max Past Epochs	[0, **1**, 10]	Number of past epochs for which decryption keys are retained.
Out-of-Order Tolerance	[**5**, 50, 500]	Number of out-of-order messages accepted per epoch. Impacts forward secrecy.
Maximum Forward Distance	[0, **1000**, 100 000]	Max number of future messages that can be skipped before not being able to decrypt.
Log Level	[no logging, debug, **info**, warn, error]	Verbosity of system logs.
Application Message Size [kB]	[1, **4**, 8]	Size of each application-layer message generated and encrypted by Valkyrie MLS.
Application Message Frequency [messages/s]	[2, **10**, 50, 300]	Frequency of messages sent per second using MGEN.

Table 5.1: Default parameter values for testing. Bold values indicate defaults used when varying other parameters.

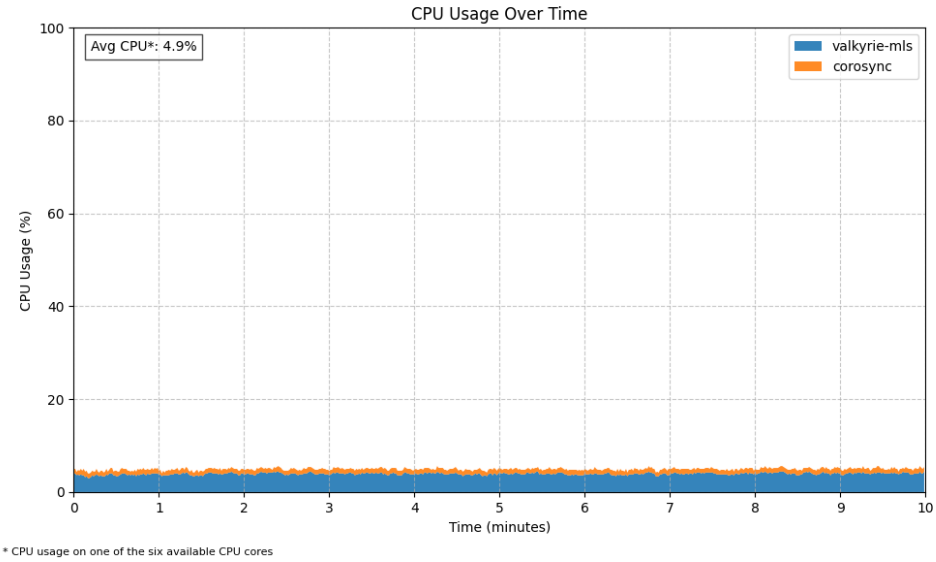


Figure 5.2: CPU usage of Corosync and Valkyrie MLS (Baseline).

Update Interval [s]	**Avg CPU [%] (One core)**	**RAM [%]**	**Observations**
None	4.7	~ 0.6	No anomalies observed. Lowest CPU usage overall.
3	5.9	~ 0.6	Periodic message decryption failures (1–3 messages after `Commit`); pattern varies across runs.
10	5.0	~ 0.6	Occasional decryption delays observed.
100	4.5	~ 0.6	Stable behavior; lowest CPU usage among tested intervals.

Table 5.2: Performance results with varying update intervals.

We vary the update interval to assess its effect on system performance and stability. Table 5.2 summarizes the results. The average CPU load fluctuates between 4.5% and 5.9% on a single core, a difference of just 1.4 percentage points. Even at its peak, the load accounts for less than 1% of the drone's total computational capacity. RAM usage remains stable at approximately 47,000 kB across all configurations, constituting to about 0.6% of the total available system memory. With high-frequency updates (e.g., 3 seconds), we occasionally observe 1–3 messages failing to decrypt immediately after a `Commit`. These dropouts follow consistent patterns within a test but differ between runs, indicating that timing interactions between periodic Updates and MGEN's deterministic message scheduling likely cause the behavior.

Table 5.3 summarizes the results for three group protocol parameters `max_past_epochs`, `out_of_order_tolerance` and `maximum_forward_distance`. These parameters influence state retention in OpenMLS and synchronization behavior in Valkyrie MLS. Across all tests, we observed minimal variation in CPU and RAM usage, but some configurations affected the robustness on the system—placing nodes in different epochs. Increasing `max_past_epochs` showed minor increases in CPU and RAM usage, but significantly improved system robustness. Differences in RAM usage were negligible, ranging from 48,912 kB at the lowest measurement to 50,803 kB at the peak. Storing one past epoch allowed the system to recover from dropped handshake messages, whereas storing none led to desynchronization as soon as one `Commit` was lost. The difference between storing one and ten epochs was negligible. Varying `out_of_order_tolerance` and `maximum_forward_distance` had no significant effect on performance under normal traffic conditions. However, setting `maximum_forward_distance` to zero caused frequent decryption failures, highlighting its importance for tolerating burst loss or reordering.

Log verbosity had some impact on CPU usage (Table 5.4). Debug logging increased

Table 5.3: Performance impact of parameters affecting state retention and tolerance.

Parameter (Value)	**Avg CPU [%]**	**RAM [%]**
Max Past Epochs = 0	5.2	~ 0.6
Max Past Epochs = 1	5.5	~ 0.6
Max Past Epochs = 10	5.4	~ 0.6
Out of Order Tolerance = 0	4.7	~ 0.6
Out of Order Tolerance = 5	4.9	~ 0.6
Out of Order Tolerance = 50	4.8	~ 0.6
Out of Order Tolerance = 500	4.9	~ 0.6
Max Forward Dist. = 0	4.6	~ 0.6
Max Forward Dist. = 1000	4.9	~ 0.6
Max Forward Dist. = 100,000	5.0	~ 0.6

Log Level	**Avg CPU [%]**	**RAM [kB]**
`debug`	6.4	~ 0.6
`info`	4.3	~ 0.6
`error`	4.2	~ 0.6
`warn`	4.1	~ 0.6
none	4.0	~ 0.6

Table 5.4: CPU and RAM usage at different log levels.

Message Size [kB]	**Avg CPU [%]**
1	4.7
4	4.9
8	5.2

Table 5.5: Impact of message size on CPU usage.

Frequency [msg/s]	**Avg CPU [%]**
2	2.0
10	4.9
50	15.9
300	31.0

Table 5.6: Impact of message frequency on CPU usage.

CPU usage by more than 2%, while disabling logs entirely led to the lowest resource consumption. Although the recorded peak usage of 6.4% still corresponds to only 1.1% of the total available computing power, memory usage remained stable across all levels. These results suggest that limited logging provides a better balance between observability and efficiency, though more verbose logging remains feasible without noticeably affecting system performance.

We test the effect of varying message size and frequency on CPU usage, as shown in Table 5.5 and 5.6. Message size has minimal impact, with CPU usage rising slightly

Message Size [kB]	Frequency [msg/s]	Avg CPU [%]
8	25	9.9
4	50	15.3
1	200	40.2

Table 5.7: CPU usage for equivalent message loads using different size-frequency combinations.

from 4.7% to 5.2% as size increases from 1 kB to 8 kB. In contrast, message frequency significantly affects CPU load. Increasing the rate from 2 to 300 messages per second raises CPU usage from 2.0% to 31.0%. At 300 msg/s, the system initially fails due to poor error handling. After fixing this, we manage to run the test, but the MLS group becomes unstable: the maximum forward distance (set to 1000) is exceeded after three update cycles, and the group fails to recover.

To investigate how message size and frequency interact under a fixed message load, we test different combinations that yield comparable data rates. Results (Table 5.7) show that using fewer, larger messages consistently yields lower CPU usage. For example, sending 25 messages of 8 kB results in 9.9% average CPU load, while sending 200 messages of 1 kB raises usage to 40.2%. This suggests that minimizing message frequency is an effective strategy for reducing system overhead under high data loads. We also note that we found a practical limit of 200 application messages per second when using a maximum forward distance of 1000. Above this threshold, we experience that group functionality deteriorates, and never recovers. With the message frequency set to 300 messages per second, we observed a significant number of application messages failing to decrypt due to incorrect generation numbers. This occurred because, within a single epoch, the ratcheting mechanism became misaligned as a result of extensive message loss. To address this issue, we substantially increased the max_forward_distance parameter, allowing the system to tolerate a large number of dropped messages within an epoch without losing ratchet synchronization. Figure 5.5 illustrates the CPU usage under this configuration. This setup may be particularly suitable for high-throughput scenarios such as video streaming, where frequent packet transmission is expected. We discuss this in greater detail in Section 6.

Findings from Performance Testing

Our CPU measurements closely matched, and even surpassed those of Marstrander [Mar23b], indicating that our cryptographic layer introduces minimal extra load on the Flamingo platform. Figure 5.2 indicates that under baseline conditions, CPU usage was about 4.9% on one of a total of six cores. In the stress test at 300 msg/s (Figure 5.4), CPU usage on one core reached about 31% on the drones, but where we

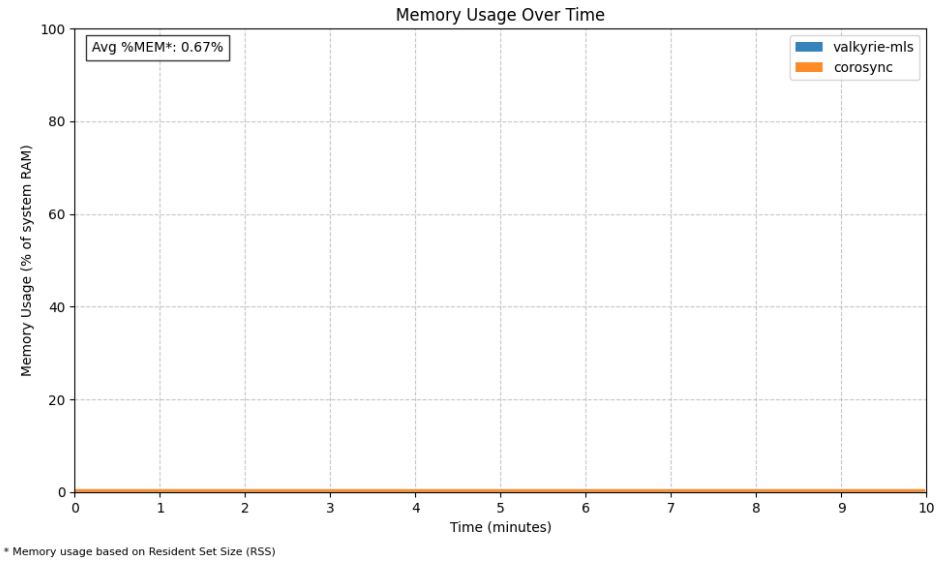


Figure 5.3: Memory usage under default parameters.

experienced sporadic forks. These forks are aligned the spikes in CPU consumption in the figure. Increasing maximum forward distance to 100,000, while using the same message frequency, lowered the overall consumption with 6.1 pp, compared to using the default of 1000 for this parameter.

Throughout all tests, memory usage remained stable and consistently below 1%. Figure 5.3 shows memory usage with default test parameters. The memory stays constant at 0.67% of total 8 GB available system memory.

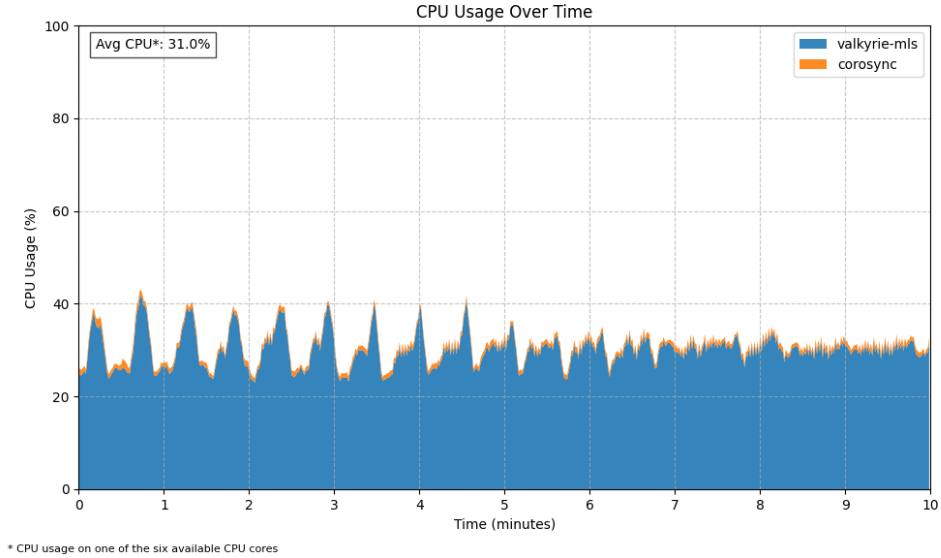


Figure 5.4: CPU usage when sending 300 messages per second.

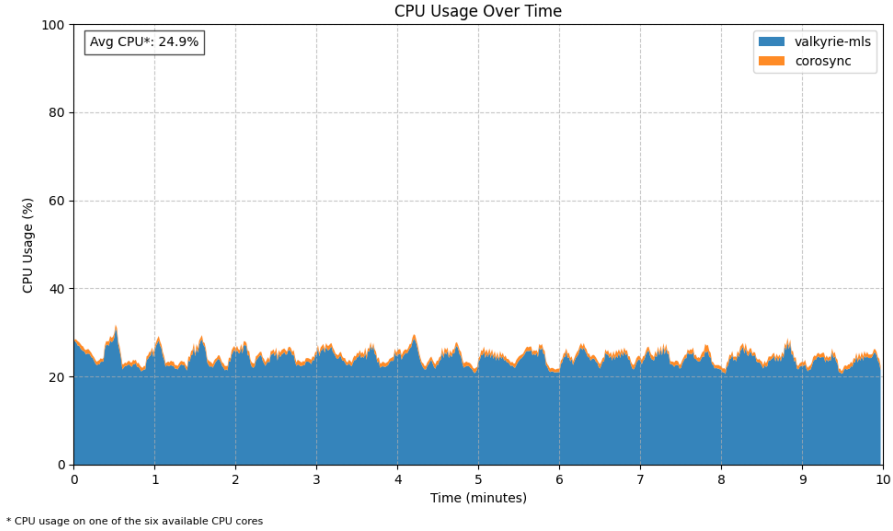


Figure 5.5: CPU usage for 300 messages per second with `max_forward_distance` of 100,000.

5.3 Simulated Testing

In addition to physical tests, we wish to conduct experiments in a controlled, simulated environment. This allows us to precisely evaluate the system's behavior under varying network conditions and to measure properties that do not require physical hardware.

The simulated tests serve two main purposes: (1) to assess system robustness under adverse network conditions, and (2) to record credential and protocol message sizes. The latter helps verify our assumptions about the network overhead introduced by our system and by different credential formats—particularly between standard X.509 certificates and our lightweight Ed25519-based credential.

5.3.1 Network Robustness Testing

For the network testing, we use the containerized swarm environment described in Section 3.2, with three Docker containers. Each container represents a drone instance and is connected via Docker's default `bridge` network. This setup allows us to control network conditions precisely using Linux traffic control tools (`tc`), simulating varying levels of packet loss across nodes.

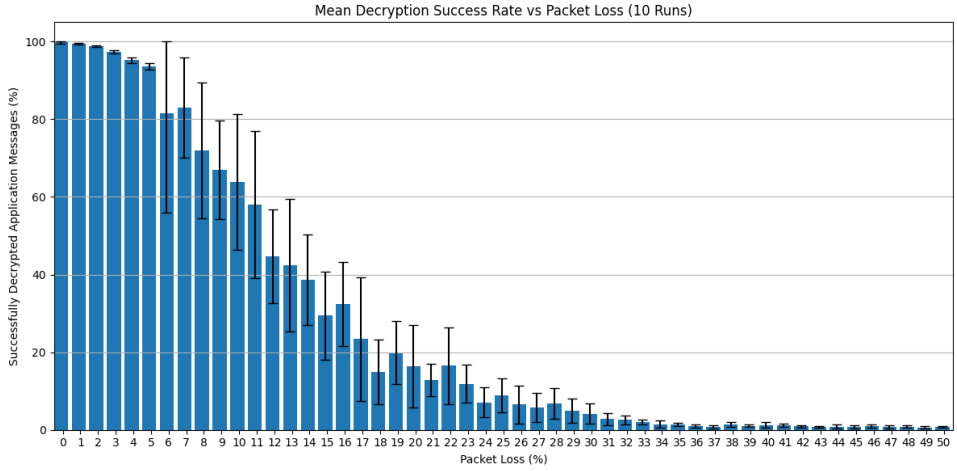


Figure 5.6: Decryption success rate vs. packet loss. The blue bar indicate the average decryption rate, while the black highlight the standard deviation.

To evaluate the resilience of Valkyrie MLS under lossy network conditions, we measure the amount of successfully decrypted application messages. We use MGEN for sending application data, and send this at a rate of 10 packets per second with a message size of 4 kB. We simulate varying levels of packet loss using Linux traffic control (`tc`) on

the Docker `bridge` network. We apply gradually increasing amount of random packet loss on the link interface, going from 0% loss to 50% loss. In the `Router` component, we measure the percentage of successfully decrypted application messages over ten 10-minute runs. When loss is introduced, we expect MLS handshake messages to be dropped, placing the nodes in different epochs. Nodes then have to re-establish the MLS group in order to decrypt application messages again. As we increase the random loss, the probability of a handshake message being loss increases, in turn affecting the node's ability to decrypt application messages.

As shown in Figure 5.6, the decryption success rate declines sharply with increasing packet loss. At 12% packet loss, fewer than half of the received messages are successfully decrypted. For losses beyond 17%, only 20% of application messages are correctly processed, leaving swarm communication is effectively non-functional.

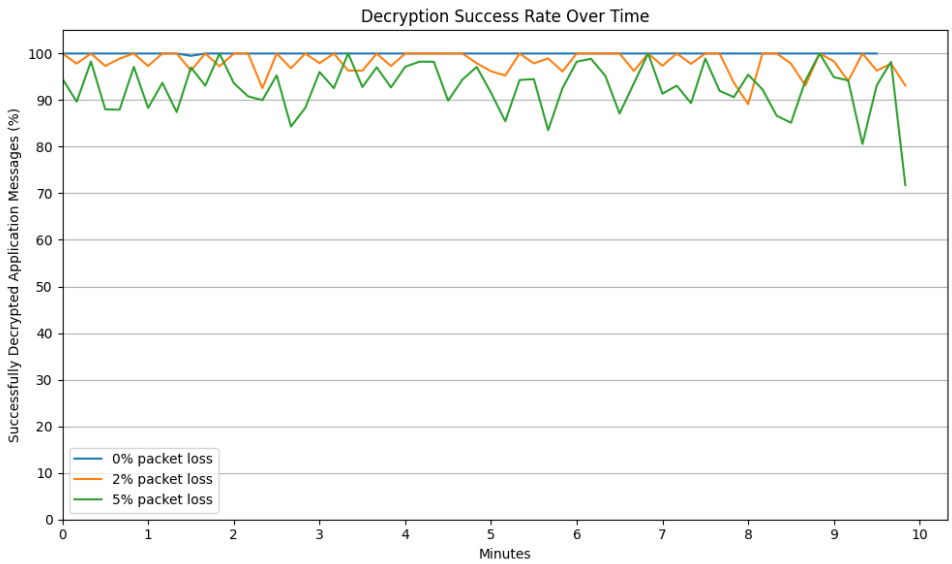


Figure 5.7: Decryption success rate over time.

Figure 5.7 further illustrates the system's sensitivity. At 0% packet loss, decryption remains consistently successful. At 2–5% loss, the success rate varies more, ranging from 90% to 95%, with occasional drops due to missed handshake messages. These results highlight the importance of a reliable transport layer and suggest that additional resilience mechanisms, such as retransmission, may be necessary in unstable environments.

Message	Basic Cred.	Ed25519 Cred.	Marstrander
`Update`*	339 B	464 B	750 B
`Add` + `Commit`**	687 B	937 B	400 B (`Commit`) + 1100 B (`Add`)
`Welcome` (2 Members)	827 B	1077 B	1700 B
`Welcome` (10 Members)	3663 B	4913 B	19,000 B
`KeyPackage`	282 B	407 B	800 B

Table 5.8: Comparison of MLS message sizes against prior recorded sizes.

* Messages from the group initiator are typically larger. These values reflect typical (mode) sizes.

** Marstrander sent Add proposals followed by a separate `Commit`. Our system embeds the Add proposal within the `Commit` message, significantly reducing total size.

5.3.2 Message Size Comparison

To evaluate the added network overhead of MLS protocol messages we generate relevant messages locally on the host machine. Protocol message sizes are measured by setting up group structures using customized Rust binaries—similar to our unit tests—and recording the serialized messages sent from the `Router` to Corosync.

Table 5.8 compares message sizes between our implementation and the results reported by Marstrander. Across all message types, the recorded sizes from this thesis are consistently reduces message sizes. `Welcome` and `Commit` messages, in particular, show significant improvements. The main efficiency gain comes from combining the `Add` and `Commit` messages into one, which reduces overhead and simplifies processing—unlike Marstrander's approach that sent them separately.

5.3.3 Credential Size Comparison

For credential sizes, we generate the certificates and credentials directly on the host and record their serialized sizes as stored locally. For our X.509 certificates, we used a self-signed root certificate as the CA, using the same key type as the subject: e.g., if the subject has a 2048 bit RSA key signed, the CA also uses a 2048 bit RSA key. Credential size is an important consideration in for our system, as credentials are sent in every `KeyPackage`, `Add`, and `Update` message. Table 5.9 compares several formats:

Previous recordings from Marstrander reported X.509 credential sizes of up to 2600 bytes [Mar23b]. In contrast, our custom `Ed25519Credential` drastically reduces overhead, requiring only 128 bytes. Even when using standard Ed25519-based X.509 certificates, the size drops to just 245 bytes.

Certificate Type	**Key Type**	**Format**	**Size**	
			Not Compressed	*Compressed*
X.509	RSA 2048	PEM	977 B	782 B
X.509	RSA 2048	DER	679 B	696 B
X.509	Ed25519	PEM	351 B	300 B
X.509	Ed25519	DER	217 B	227 B
`Ed25519Credential`	Ed25519	Custom	128 B	148 B

Table 5.9: Credential format size comparison.

We tried reducing the size of the credentials using `gzip`, a tool for file compression. Interestingly, this only reduced the size of the PEM-formatted certificates. The DER-formatted certificates remained smaller than even the compressed PEM certificates.

Chapter 6
Discussion

In this chapter, we address our four research questions, drawing on the measurements presented in Chapter 5, and insights gained during development of Valkyrie MLS. Each section focuses on one question, presenting our key findings and reflecting on the implications for secure communication in UAV swarms.

Before diving into the discussion, we briefly revisit the research questions that have guided this thesis:

1. Can OpenMLS be effectively used for secure communication in UAV swarms?
2. How can we extend an MLS-based swarm system to include authentication and message delivery services?
3. How does OpenMLS perform compared to MLS++ in a UAV swarm?
4. What is the impact of video streaming on MLS-based communication in UAV swarms?

In the sections that follow (Sections 6.1–6.4), we explore each of these questions in depth, discussing what we observed, the challenges we encountered, and what the results suggest for future deployments of secure communication in UAV swarms.

6.1 Evaluating OpenMLS for Secure Communication

While OpenMLS offers a strong cryptographic foundation via the MLS protocol, it alone is insufficient for secure end-to-end communication in UAV swarms. Achieving this requires domain-specific integration and system-level support tailored to swarm communication. We also consider whether alternative protocols may be better suited for this task.

OpenMLS Is a Suitable Choice for Implementing MLS

OpenMLS provides nearly all the functionality needed to implement the MLS protocol effectively. The library includes all core cryptographic operations needed for end-to-end encrypted group messaging. It implements the fundamental MLS sub-protocols TreeSync, TreeKEM, and TreeDEM, which together cover group membership management, continuous key agreement, and secure message broadcasting. These mechanisms are well-suited to the dynamic and asynchronous nature of UAV swarms: MLS was explicitly designed for asynchronous group messaging with strong security properties, meaning OpenMLS can seamlessly re-establish keys and synchronize group state as drones join, depart, or lose connectivity.

The reliability of these security properties also depends on implementation correctness and system-level constraints. OpenMLS benefits from Rust's focus on memory safety and performance without a garbage collector, making it a strong fit for our resource-constrained drone environment. Rust's compile-time checks for ownership, borrowing, and lifetimes help prevent memory issues like buffer overflows and use-after-free bugs, which are hard to detect and can compromise security. Although our limited experience with Rust made the initial development phase challenging, the language promoted a disciplined approach to handling sensitive cryptographic data. Compared to MLS++, written in C++, Rust provides stronger built-in guarantees against memory-related errors, giving us increased confidence in the security and reliability of our implementation.

OpenMLS stands out as a practical and developer-friendly choice for implementing MLS in resource-constrained systems. Its API is well-structured and intuitive, and integrating it into our system required no modifications to the core library. We used the mandatory ciphersuite `MLS_128_HPKEX25519_AES128GCM_SHA256_ED25519`, which offers low setup time and message overhead, although it exhibits lower message handling performance [LBH24]. If needed, switching to a different ciphersuite is straightforward.

OpenMLS is supported by active maintainers, detailed documentation, and an engaged open-source community. The official OpenMLS Book [RC25a] provides concrete examples and clear guidance, and the project is maintained by organizations central to MLS standardization, including Phoenix R&D and Cryspen. The community surrounding OpenMLS also provides support through its forum[1], a forum which we benefited of personally through this development. OpenMLS's ease of use, thorough documentation, and strong community support makes it a strong candidate for future use in this line of research.

[1] https://openmls.zulipchat.com/

OpenMLS Alone Cannot Provide End-to-End Secure Swarm Communication

While OpenMLS is a suitable choice for implementing the MLS protocol, it cannot by itself provide end-to-end secure communication in UAV swarms. In particular, OpenMLS lacks built-in support for two key architectural components mandated by the MLS specification: the AS and DS. This provides flexibility in terms of development and customized use of the library in messaging applications, but requires developers that use OpenMLS to carefully design the complementary abstracted services.

OpenMLS does not yet support standard credential formats such as X.509 out of the box. The library only defines the minimal `BasicCredential`. This type provides a bare assertion of identity using an ID and a signature key, without any context for validation, which in practice offers minimal assurance. This requires current users of the library to create their own validation mechanisms. To provide stronger authentication, we extended OpenMLS with our custom `Ed25519Credential` and associated validation function. Although this met the needs of our system, it runs counter to best practices in secure communication, which caution against ad hoc cryptographic schemes. Custom credential mechanisms risk introducing subtle vulnerabilities and limit interoperability, making this a significant limitation in OpenMLS for real-world swarm applications.

To Achieve Secure Communication in UAV Swarms With MLS, More Domain-Specific Integration Is Needed

UAV swarms pose unique operational, networking, and security challenges that differ significantly from the environments for which OpenMLS and the broader MLS protocol were originally designed. While OpenMLS serves as a general-purpose cryptographic library optimized for asynchronous group messaging over relatively stable Internet infrastructure, the UAV swarm operates under very different conditions. This mismatch highlights the need for deeper domain-specific integration when applying OpenMLS in such systems.

Our current implementation remains general-purpose and lacks tight integration with swarm-specific systems. Due to time constraints, development and testing were limited to a one-week period at FFI's facilities. Valkyrie MLS demonstrates proof-of-concept functionality and is yet to be embedded in production flight software or adapted to specific swarm use cases. As such, the system does not yet reflect the full complexity of real-world missions or the operational expertise held by experienced swarm operators.

Bridging this gap requires deeper domain knowledge and closer alignment with practical swarm development. Although we reviewed literature and consulted with the Valkyrie MLS development team, our perspective remains academic. This limited understanding influenced our design decisions. While our system reflects assumptions common in general distributed systems, it may not fully align with tactical requirements. However, this generality could also be a strength by potentially enabling the architecture to serve as a flexible foundation for a broader class of resource-constrained, infrastructure-less systems beyond UAV swarms.

Alternative Protocols Should Be Considered for Secure Swarm Communication

While MLS is a robust and scalable group messaging protocol, its assumptions do not perfectly match UAV swarm environments. MLS was designed for settings with relatively stable and reliable networks (e.g. TCP/IP), not wireless links without delivery guarantees or message ordering. We experienced that lost MLS messages caused forks in the cryptographic group state between members and that the coordination needed to restore a synchronized state reduced the *goodput* of application messages. Thus, the mechanisms that provide strong and efficient security guarantees in MLS become liabilities when swarm members experience desynchronization in their cryptographic state.

Moreover, the swarm architecture assumes a trusted GCS that issues credentials to each UAV prior to deployment. During this process, drones are also manually loaded with a set of trusted keys, effectively bootstrapping trust through an out-of-band mechanism. As a result, the basic requirement for authentic identities relies on using pre-distributed material. If the system already relies on static, out-of-band trust, are the dynamic group management features of MLS truly necessary, or do they instead introduce unnecessary complexity for real-world drone swarms?

In this context, we raise the questions of alternatives to MLS. One particularly promising alternative is the use of a Symmetric-Key Authenticated Key Exchange (SAKE)-based Pre-Shared Key (PSK) scheme, as explored in a study by Boyd *et al.* [BDdK+21]. Their approach uses a pairwise long-term Pre-Shared Key (PSK) that is cryptographically evolved after each session, providing full FS and robustness to synchronization issues, while avoiding the computational overhead of public-key operations. Though limited to two-party settings, this method may still offer a lightweight and secure alternative in systems where the set of communicating nodes is stable and known in advance.

Group size is another important factor when evaluating the suitability of MLS. While MLS excels in large groups, typical UAV swarms are comparatively small. In

scenarios described by FFI, swarm sizes usually range from 2 to 40 drones [HBSS24], whereas MLS is designed to handle groups from two up to thousands of members. As such, whether its scalability provides any real advantage at the scale of tens of nodes is debatable. In a small, fixed-size swarm, a simpler full-mesh or static keying approach might offer equivalent security with lower complexity.

Given these constraints, adopting MLS may introduce unnecessary complexity. Its reliance on reliable, ordered transport and dynamic key updates does not align well with lossy radio links and pre-established trust. In this context, the benefits of MLS are limited and come at the cost of extra protocol overhead. Consequently, while MLS is a powerful general-purpose solution, it can be overly complicated for secure communication in performance-critical, small-scale UAV swarms.

6.2 Extending MLS for Authentication and Message Delivery

As mentioned earlier, MLS provides the core cryptographic mechanisms for secure group communication, but it is not a complete messaging system on its own. It depends on the existence of additional supporting services to manage authentication, credential distribution, and reliable message delivery. To bridge this gap and support secure, coordinated communication in a UAV swarm, our system employs Totem (Corosync) and a custom-designed authentication scheme based on Ed25519 signatures. Together, these components ensure that messages are delivered in a consistent order and that all members of the swarm are authenticated. We begin by discussing the implementation of these components, before we highlight current limitations and open challenges in our system design.

6.2.1 Delivery Service Implementation

We argue that our implementation satisfies the necessary functionality expected of a Delivery Service. As defined in [BRO+25], the DS is a conceptual component responsible for two primary tasks: (1) providing a directory for initial keying material and (2) routing messages between clients within the group.

In our implementation, nodes periodically broadcast `KeyPackages`, which decentralizes the directory of initial keying material. Furthermore, we ensure the routing of messages between clients by using peer-to-peer multicast, and the Totem protocol provides total ordered delivery guarantees.

Periodically Broadcasting `KeyPackages` Decentralizes the Directory of Initial Keying Material

To make initial keying material readily available, clients periodically broadcast their `KeyPackages`. This eliminates the need for a centralized delivery service and allows any MLS group member to obtain the necessary information to add an external client to its group. Upon receiving a `KeyPackage`, the recipient immediately processes it. The credential is first validated, and if it is valid and the sender is not already a group member, the `KeyPackage` is stored for use in the next update cycle when creating a `Welcome` message. Invalid or duplicate `KeyPackage` are discarded immediately, and no backlog is kept. Once a `KeyPackage` is used to create a `Welcome`, it is discarded. This strategy adheres to the one-time-use principle for `KeyPackages`, eliminates additional coordination overhead, and enforces immediate processing requirements.

A possible future improvement is to allow local storage of `KeyPackages`. The current design demands instant admission decisions, which we see restrictive in lossy or highly dynamic networks. An alternative would be to store received `KeyPackages` locally, so they can be processed later. For instance, after operator approval or when connectivity improves. This would allow for more flexibility regarding handling of `KeyPackages` and avoid coordinated activity that could lead to forks during periods with poor network conditions.

Another option is on-demand querying triggered by discovery signals (e.g., periodic heartbeat messages). Such querying would save bandwidth by avoiding unsolicited `KeyPackage` broadcasts. However, it introduces an extra round-trip. Valkyrie MLS therefore favors broadcasting for its simplicity and lack of coordinating dependencies, but any stored `KeyPackages` must still respect the one-time-use rule and be deleted once consumed.

Routing Using Peer-to-Peer Multicast and Total Ordered Delivery Through the Totem Protocol

To accommodate for message routing, messages are broadcast and use a multicast address. Each drone sends its messages directly to the swarm over a shared communication channel, which eliminates the need for traditional point-to-point routing. While some drones may not have direct line-of-sight connections, the underlying Rajant radio provides functionality that abstracts these limitations and maintains swarm-wide message propagation [Raj19].

As discussed in Section 2.3, MLS imposes strict ordering requirements on messages in two key situations: (1) proposals must be received before their corresponding `Commit`, and (2) all members must agree on the order of epoch transitions, each

initiated by a `Commit`. We simplify the first requirement by committing immediately on each proposal. This eliminates the need to track or sequence multiple outstanding proposals. However, the second requirement, ensuring global agreement on the order of epoch transitions, remains essential for maintaining a consistent group state.

To account for this, we impose ordering through the use of the Totem protocol provided through Corosync. Totem functions as a lightweight consensus layer that guarantees total, agreed-upon ordering of handshake messages. This ensures that all drones process epoch-changing messages in the same sequence. Our approach was inspired by prior work [Mar23b], which implemented a similar mechanism from scratch. In contrast, we leverage Corosync to reduce implementation complexity while achieving the same guarantees of deterministic message ordering.

Based on our functionality tests, Totem provides the necessary guarantees for message ordering. However, we identified a limitation in fully distributed setups: the lack of a coordinating entity introduces problems with consensus, which is not guaranteed attainable under system model [FLP85]. One possible improvement would be to adopt a hybrid architecture, where a temporary central node is elected to act as a coordinating entity. Other nodes could broadcast their proposals, while the central node could handle tasks such as message sequencing and decide on when and what proposals to commit on, particularly during poor connectivity. To increase resilience, a backup node could also be elected to take over if the primary node fails. The election mechanism could draw inspiration from existing routing protocols like Open Shortest Path First (OSPF) [Moy98], which use metrics and deterministic rules to select a designated router and a backup. Applying similar ideas could help manage group coordination more effectively during edge cases.

6.2.2 Authentication Service Implementation

The MLS architecture [BRO+25] defines core functionalities the Authentication Service must provide: (1) issuance of credentials that bind client identities to signature key pairs, (2) enabling clients to verify another's credential against a reference identifier, and (3) allowing members to check if two credentials belong to the same client. Valkyrie MLS fulfills these requirements in turn. First, it uses a model inspired by Public Key Infrastructure (PKI) to issue credentials binding each UAV's identity to its MLS signing key pre-flight. Second, each drone locally verifies any received credential by checking that the issuer is known, the credential information is well-formed and valid, and the issuer's signature is correct. Third, every credential includes a unique reference ID so a drone can compare two credentials' IDs to determine if they refer to the same UAV. Each of these mechanisms is described in detail in Section 4.4. Together, they enable the AS to function as intended, assuming pre-distributed credentials and trusted keys.

6.2.3 Challenges Regarding the Delivery Service and Authentication Service

Although we assess our implementation to be functional with its extended services, there are still issues that remain. We now describe five challenges we have identified from our project work and sketch possible solutions to each of those challenges. The challenges we have identified, which we discuss below, are the occurrence of epoch forks, epoch skews, disruption of operation, membership inconsistencies and change of trust in-flight.

Challenge 1: Epoch Forks

The occurrence of forks or epoch divergences between nodes is an issue for Valkyrie MLS. We define a fork as two or more nodes have inconsistent view of the current cryptographic group epoch state. As the drone swam is a distributed system, it is prone to inconsistencies. This is an issue we frequently encountered during our testing and we have identified two major causes of these forks:

1. **Concurrent commits without the Totem send token:** Two or more nodes commit at nearly the same time.
2. **Packet loss over the unreliable network:** `Commits` are lost in transit between nodes.

Concurrent commits occur when two or more drones generate and broadcast `Commit` messages at nearly the same time, resulting in inconsistent epoch histories across the swarm. For example, if Drone A and Drone B both create a `Commit` simultaneously; `CommitA` and `CommitB` respectively, each drone may apply its own `Commit` first and then the other's, resulting in different epoch orders: A sees `CommitA` → `CommitB`, while B sees `CommitB` → `CommitA`. This desynchronization leads to incompatible encryption keys, making it impossible for the drones to decrypt each other's messages.

In theory, this should not be a problem, as Totem imposes an ordering on the sending of these `Commit` messages. The lack of coordination between MLS and Totem in our system, however, has not made it possible to notify nodes of when they hold the Totem send token. In our current implementation, the Corosync process does not stop the `MlsEngine` from commiting at any given time, which results in the possibility of epoch forks happening. Hence, Drone A can end up committing when they are not holding the send token and has to wait to send `CommitA`, even though it has applied this change locally. If Drone B then receives this token in the meantime and commits, we would end up in the scenario where the group receives `CommitB` first,

and then `CommitA`. Drone A also receives this commit, but has already transitioned into its new epoch.

Our current solution is to stagger drone startups to reduce simultaneous commits. As a temporary workaround, we stagger the startup of drones by a few seconds to offset their commit cycles. This reduces the likelihood of simultaneous commits. While effective in practice during testing, this approach is fragile and non-scalable. It relies on manual intervention and does not guarantee consistent operation under real-world conditions. Addressing this limitation is essential for achieving a robust and fault-tolerant system.

Several solutions, including leader election and token-gated commits, could mitigate the problem. To eliminate epoch forks caused by concurrent commits, we have identified the following solutions:

Leader election: Elect a single drone as the group leader responsible for issuing all commits. Other members send proposals to the leader, who batches them and commits them at fixed intervals. This feature is already available, as a member can reference others' proposals when committing. This reduces commit frequency and improves coordination.

Token-gated commits: Restrict commit generation to the drone currently holding the Totem token, ensuring that only one node can commit at a time. This builds on existing Corosync infrastructure and naturally prevents concurrent commits.

Delay-and-decide: Upon generating a commit, briefly pause to observe whether any concurrent commits are received. If so, apply a deterministic tie-breaking rule to select one and discard the others. This approach tolerates limited concurrency while maintaining consistency.

Accept-then-rollback: Immediately accept the first commit received but keep a backup of the previous group state. If another valid commit for the same epoch arrives shortly after, apply a tie-breaking rule to decide whether to continue or roll back to the alternative. This method balances responsiveness with correctness, but requires careful handling of state retention to preserve FS.

Among these, we consider leader election and token-gated commits the most promising, as they offer scalable, systematic ways to coordinate group state transitions and prevent the race conditions causing epoch forks.

The second identified cause of epoch forks in is the loss of `Commit` messages in transit. When a `Commit` is dropped, one or more drones miss a critical update to the group state, resulting in inconsistent epoch states. This leads to cryptographic

divergence, where nodes derive incompatible keys and can no longer decrypt each other's messages, or we get a situation where one node is an epoch ahead, and can decrypt messages from the other node, while the other node is an epoch behind, and cannot decrypt messages from the first drone since it is in front.

This issue stems from our use of UDP as the transport layer for all MLS messages. Unlike TCP, UDP offers no delivery guarantees, dropped packets are not acknowledged or retransmitted. While MLS was originally designed with reliable transport like TCP in mind, our swarm environment relies on lightweight communication over potentially lossy wireless links, making message loss a real and recurring threat. This challenge has also been noted by prior work, including [Mar23b; BBR+23; Die22].

Corosync's total ordered delivery reduces the impact of lost `Commits`. As a mitigation, we send MLS configuration messages over Corosync, which ensures all participants receive messages in the same order and, in theory, retransmits lost ones. However, since Corosync still runs over an unreliable medium, packet loss can still occur. We have not yet formally evaluated the reliability of Corosync compared to raw UDP, so its impact on reducing forks remains an open question.

We propose using Quick UDP Internet Connections (QUIC) [IT21] as the transport layer protocol. QUIC is a modern transport protocol that builds on UDP but adds important features such as acknowledgments, retransmissions, and stream multiplexing. It provides stronger delivery guarantees than UDP while maintaining lower overhead than TCP. Given the intermittent and lossy nature of swarm networks, QUIC is a promising alternative for transmitting MLS messages reliably. This aligns with observations from prior work such as [Die22], which highlights UDP's shortcomings for MLS use in unreliable environments.

Another mitigation strategy involves adopting schemes like *Fork-Resillient* CGKA [AMT23] for protocol-layer recover. The paper defines and construct a new type of CGKA that supports processing packets in any causally respecting order. FR-CGKA allows nodes to store past group states and recover from missed `Commits`. If a node misses a `Commit`, it can retroactively apply it after retrieving the missing message, enabling re-synchronization with the group. This adds robustness by allowing eventual consistency for nodes that have become out of sync with the rest of the group. Work within the MLS community has begun looking into such solutions, but are still early in its development [Koh25].

If a packet is lost in transit over the physical medium, there is no clear fix other than resending that package. At initial glance, we believe these approaches could work to mitigate the consequence of lost commit messages generating epoch forks, as they each have some sort of recovery mechanism in place. QUIC on the transport layer, while FR-CGKA in the MLS protocol layer.

Challenge 2: Two Communication Channels Cause Epoch Skews

We observed that encrypted application messages occasionally arrived either too early or too late relative to the recipient's current epoch. In MLS, application messages must be both encrypted and decrypted within their designated epoch; otherwise, decryption fails. Our architecture uses two separate communication channels: Corosync for transmitting MLS handshake messages and a UDP multicast channel for encrypted application messages. This separation led to situations where a drone would generate a `Commit`, begin encrypting and sending application messages using the new epoch, but the corresponding `Commit` had not yet propagated to other drones. This was mainly due to the previously mentioned issue regarding applying `Commits` locally, without holding the Totem send token. This results in epoch mismatches and decryption failures for other members of the group.

Increasing max-past-epochs allows decryption of messages that arrive late. The `max_past_epochs` parameter in OpenMLS is, by default, set to 0, meaning messages from any previous epoch are immediately rejected. During parameter testing, we increased this value to 1 and observed a notable improvement in message delivery. Messages that arrived one epoch late could now be successfully decrypted, significantly reducing message loss in the presence of moderate network delays. However, this adjustment had no impact on messages that arriving *too early*. Hence, there is still the need for additional mechanisms to handle forward epoch skew.

One approach is to encrypt messages using past epochs to create a forward buffer. One approach involves introducing a forward buffer by encrypting application messages using an older, stored epoch. This would allow receiving drones to maintain a buffer of both past and future epochs, increasing the likelihood of successful decryption despite network delays. As stated above, OpenMLS supports retention of past epochs via the max-past-epochs parameter. For instance, by setting max-past-epochs to 10 and encrypting messages using epoch 5, drones that have advanced further can still decrypt these messages, effectively creating a forward window. While this strategy weakens security guarantees, it significantly improves robustness. We see this as an acceptable tradeoff given the swarm's current operational challenges.

Another approach is to unify all communication over a single channel to eliminate ordering inconsistencies. One proposed mitigation is to send all messages, both MLS handshake and encrypted application messages, over a single communication channel, such as Corosync. While this could eliminate ordering inconsistencies between channels, we suspect that Corosync may not sustain the high message throughput required by our system and could become a bottleneck. This remains untested, however, and empirical evaluation should be done to assess its feasibility. An alternative approach is to use raw UDP for all messages, eliminating the consensus layer entirely. Although this would likely increase the frequency of forks, it may be

possible to maintain coordination by for instance introducing a lightweight distributed locking mechanism, as used in distributed database systems.

The most straightforward approach however, would be to employ the token-gated commit mechanism discussed for epoch forks. Using Corosync, we have not been able to obtain the Totem send token, but prior work, which implemented the Totem protocol from scratch, used this approach [Mar23b].

Challenge 3: Unreliable Network Conditions Cause Disruption of Operations

When a drone operates at the edge of radio coverage, it frequently drops in and out of the Totem group, causing instability that halts the entire system's progress. Because Corosync requires all members to be synchronized to circulate the Totem send token, a single unstable node blocks token circulation for the whole group. This stops MLS handshakes being sent, meaning updates are only applied locally and the system fails completely rather than continuing with the stable nodes. Corosync exposes no direct control over token management, offering only limited parameter tuning to mitigate this behavior.

Currently, we do not implement any solution to this problem. Token-gated commits would help in not evolving epochs locally, but it would not fix the problem of handshake messages being halted. However, we observe that simply shutting down the unstable drone or moving it completely out of range results in the system stabilizing. This suggests that a clear removal is better than a partially connected node. We therefore propose using principles from mobile communication networks to improve group stability, specifically through hysteresis and time-to-trigger (TTT) mechanisms. These techniques are commonly used to prevent unnecessary state changes in response to short-term signal fluctuations [ZWG+12].

Hysteresis introduces a buffer zone between thresholds, ensuring that a state change (e.g., joining or leaving a group) only occurs when the signal strength crosses a threshold by a certain margin. This prevents rapid toggling between states when the signal hovers near the boundary. For example, in wireless handover, a device only switches to a new base station if its signal is significantly stronger, avoiding frequent switching between similar signals. Time to Trigger (TTT) adds a temporal filter by requiring that a condition be met for a minimum duration before triggering a state change. This avoids reacting to brief or noisy changes in signal strength.

By combining these two mechanisms, drones would only be added or removed from the Totem group if their signal strength remains consistently above or below a threshold for a set period. This helps eliminate "flapping" behavior where a weakly connected drone repeatedly joins and leaves the group. As a result, group membership becomes

more stable, the Totem token can circulate reliably, and we reduce the impact one unstable drone has on the entire swarm. We also note that this requires additional control over Totem operations, and advocates for customizing Corosync or implement Totem from scratch.

Challenge 4: Synchronizing Membership Between MLS and Totem

In our architecture, a single drone maintains multiple notions of group state across different system layers: the network group (Totem), the cryptographic group (managed by OpenMLS), and the application layer group (defined by the swarm application). As the system is agnostic to the swarm-level application logic, the application layer group is abstracted away from the scope of this discussion. We therefore focus on the interaction between the network group and the cryptographic group.

These layers operate independently and are not inherently synchronized. As a result, each layer maintains its own view of group membership, which can lead to inconsistencies. For instance, a drone might be removed from the MLS group but still remain a member of the Totem group.

Such desynchronization introduces both security risks and inefficiencies. A node that remains in the network group despite being excluded from the cryptographic group will continue to receive encrypted messages it cannot decrypt, consuming radio bandwidth unnecessarily. Furthermore, even without access to the message content, the node may still extract metadata, observe communication patterns, and perform traffic analysis. This opens the door for passive surveillance or even denial-of-service attacks.

Conversely, if a node is removed from the network group but still considered part of the MLS group, the remaining drones may falsely believe the node is still reachable. We describe this scenario as false situational awareness. This can result in degraded coordination or decision-making due to inaccurate assumptions about the communication topology.

Valkyrie MLS implements partial synchronization. In the current implementation, our system supports one-way synchronization: when a node is removed from the Totem network group, it is automatically scheduled for removal from the MLS group. While this provides a basic mechanism for reducing exposure to unreachable nodes, it does not address the reverse case. Nodes removed from OpenMLS may still remain active in the Corosync network group, thereby continuing to receive traffic and introducing the risks discussed above.

To resolve this, we propose a dedicated membership coordination module to ensure bidirectional synchronization between the Totem (Corosync in our case) and MLS

group. This module would continuously monitor membership events in both layers and enforce consistency by removing nodes from the MLS group when Corosync detects a failure, and conversely removing nodes from the Corosync group when they are excluded from the MLS group. By aligning membership views across both layers, this approach improves overall system robustness, reduces unnecessary bandwidth consumption, and mitigates security vulnerabilities arising from membership inconsistencies.

Challenge 5: Change Trust in Flight

Our current implementation of the AS assumes that all credentials and trusted keys are preloaded before flight. It does not support adding new trusted keys, issuing new credentials, or revoking access for compromised nodes during operation. This static design requires any trust changes, such as accepting drones with credentials from a different GCS, to be handled through out-of-band communication. While this is acceptable for small or controlled deployments, it becomes a significant operational bottleneck and a security risk for larger, more dynamic, and flexible swarm operations.

We propose adding an authenticated Authentication Service API to enable secure, over-the-air updates of trust relationships within the swarm. The API would support three key operations: Add-Trusted-Key, Remove-Trusted-Key and Revoke-Credential, each carrying the relevant public key or credential identifier. To ensure security, all API messages must be encrypted to preserve confidentiality and signed by an already trusted entity to guarantee integrity and authenticity. Access control is enforced by requiring that only nodes possessing a trusted key may issue API commands, with every command individually signed. This design offers practical operational benefits: swarms can dynamically merge by having respective Ground Control Station adding the key of the other GCS once, enabling all drones to authenticate peers signed by any participating GCS. Credential revocation follows the same mechanism, where trusted entities could maintain and distribute a revocation list. This however, raises a concern about an ever-growing trust store, as the set of trusted keys grows larger. To counteract this, we could remove trusted keys if e.g., two merged swarms disband. In-flight issuance of new credentials would still need out-of-band communication, as this information should not be broadcast.

6.3 Performance Comparison: OpenMLS vs. MLS++

OpenMLS performs as well as or better than MLS++ in our tests, but differences in setup and methods make direct comparison difficult. Our results show similar or better performance. However, architectural differences between the systems make it

difficult to draw meaningful, isolated comparisons between OpenMLS and MLS++. Furthermore, methodological mismatches limit direct comparison.

Our Results Show Similar or Better Performance

Our parameter tests suggest that OpenMLS does not introduce any significant bottlenecks in terms of system performance. Given a baseline of handling a total of 120 kB/s (three nodes each sending 40 kB of data every second), we observe an average CPU load of 4.9% on a single core. This corresponds to less than 1% of the total available computational resources on a six-core Flamingo UAV, and leaves substantial headroom for other processes, such as the swarm application.

Even under stress-test conditions, beyond the expected message rates for a single entity, we only observe an average load of approximately 5% of total computational capacity (31% on a single core), with short peaks reaching about 40% on that core. We argue that this demonstrates that the system can handle high message rates. We therefore conclude that the computational overhead introduced by Valkyrie MLS remains low and is negligible for the expected message load.

We found it hard to define exact test that replicates prior recorded results, as Marstrander [Mar23b] conducted their best-documented tests primarily on Jetson Nano hardware. Of the tests recorded on the actual drone platform, the CPU consumption was 10.5% of one core when using similar amounts of application data.

A clearer comparison emerges when examining message and credential size overhead. As shown in Sections 5.3.2 and 5.3.3, our recorded sizes show significant reductions compared to those reported for MLS++ [Mar23b]. One factor is our use of smaller Ed25519 signature keys, which reduce credential sizes—both for our custom scheme and X.509 certificates—and thus shrink messages that include them. Prior work appears to use keys based on the P256 curve instead. Although the key and signature sizes of both schemes are generally similar, we observe notable differences in practice. While we cannot pinpoint the exact cause, our OpenMLS-generated messages and credentials based on Ed25519 consistently exhibit substantially smaller sizes.

The main takeaway from our performance testing is that computational capacity is not a limiting factor for typical usage. However, stress tests reveal an upper bound to what the system can handle before functionality issues begin to appear. Comparing Figure 5.4 and Figure 5.5, we note that performance degrades when failures occur, despite the message load being constant across both tests. The only difference in the latter is an increased `max_forward_distance`, which permits decryption of messages further ahead in the epoch sequence, and appears to help mitigate crypto-state forks.

Architectural Differences Between the Systems Make It Difficult to Draw Meaningful, Isolated Comparisons Between OpenMLS and MLS++

There are distinct architectural differences between the our system and the system used in prior testing [Mar23b]. In our implementation, the Token mechanism is managed using the Corosync library. In contrast, Marstrander implemented Token functionality manually, which likely introduces different performance characteristics and failure-handling behavior. Another key difference is our avoidance of heartbeat signals. While Marstrander's system uses periodic heartbeats to monitor group membership and trigger updates. These differences reflect distinct system design and architecture, and have significant implications for runtime behavior and system performance.

Therefore, conclusions on the differences between OpenMLS and MLS++ are difficult to draw. Given the architectural and methodological differences outlined above, it is difficult to isolate the effect of the cryptographic library alone when comparing our results to those of Marstrander. Performance outcomes are clearly influenced by factors beyond the choice of library or programming language. As a result, any conclusions drawn about the relative performance or efficiency of OpenMLS versus MLS++, or Valkyrie MLS versus Flamingo-MLS, must be interpreted with caution.

Methodological Differences Limit Comparison with Marstrander

Hardware differences limit comparability. Marstrander conducted extensive testing in [Mar23b, Section 5 – *Testing in a Simulated Environment*], where the results presented in that section reflect performance on the Jetson Nano. In [Mar23b, Section 6 – *Flamingo MLS Takes Off*], most of the simulated results are discerned due to the realization that the final hardware platform, Jetson Xavier NX, used on the Flamingo drone is significantly more powerful than the Jetson Nano. To ensure our results are relevant to the operational context of the Valkyrie swarm, we instead chose to measure CPU and memory usage directly on the drones used in deployment.

Ambiguity in test descriptions complicates interpretation. Furthermore, the tests performed by Marstrander on the actual drone hardware are described in a way that makes it difficult to discern precisely what was measured and under what conditions. This lack of clarity limits the extent to which we can meaningfully compare our results to theirs.

Limitations in testing due to MGEN's load constraints. During the testing phase, we used MGEN to generate synthetic application-layer traffic. While effective for simulating basic communication patterns, the tool introduced certain limitations. In its current configuration, MGEN supports message sizes only up to 8192 bytes (8kB),

whereas real-world drone communications may involve packets as large as 50kB. As a result, we were unable to evaluate the system's performance with larger payloads. That said, within the tested range, we observed no issues related to message size, and we found no evidence suggesting that larger packets would pose significant problems for the delivery layer in practice.

6.4 Assessing the Impact of Video Streaming

Video streaming may be feasible, but current evidence is inconclusive. Our tests show it significantly impacts swarm performance, though one tested configuration showed potential. We suggest that a more lightweight security mechanism might suffice for video streaming, but further investigation is needed.

Preliminary Impact of Video Streaming on Swarm Performance

High-frequency message load test as a proxy for video traffic. During our physical testing at Kjeller, we conducted extensive performance evaluations of the system. In one of the scenarios, we generated application-layer data at a very high frequency, up to 300 outgoing messages per second from each drone. This meant that each drone processed approximately 300 outgoing and 600 incoming messages per second. While we did not test actual encrypted video streaming, this setup approximates the message frequency typically associated with live video transmission, allowing us to assess how the system might behave under similar load conditions.

The results at this message rate revealed several critical performance bottlenecks. CPU usage exceeded 30% on a single core on the six-core Jetson Xavier NX, but more concerning were the functional failures. The high message rate caused loss of commit messages, leading to epoch forks that the system was unable to recover from during the tests. Additionally, our implementation encountered runtime panics due to internal errors triggered under high load. These issues were later patched, as the system should not fail under high traffic alone. The root cause was traced to Corosync being unable to reliably deliver messages under such load, resulting in critical communication failures.

Another issue observed was related to `max_forward_distance`. Our system uses a default value of 1000. This value was quickly exceeded when one node fell out for a short period. Once this limit was reached, the receiving drones were unable to decrypt further messages, and the system did not recover until a new commit was issued to reset the generation counter. This behavior poses a risk to system stability under high-throughput conditions.

One promising configuration we discovered during testing involved significantly increasing the `maximum_forward_distance` parameter from the standard value of 1000, to much higher values such as 100,000 or even 100,000,000. This adjustment enabled drones to decrypt messages that were much further ahead within the the epoch, making the system more resilient to packet loss.

A More Lightweight Security Mechanism Could Be Sufficient For Video

Video streams in swarm operations typically have lower security requirements than command and control traffic. Since live video provides only short-lived situational awareness and quickly loses tactical value, leakage or tampering is rarely safety-critical. As a result, core MLS guarantees like FS and PCS may be less relevant for such ephemeral data. Enforcing strict properties, such as sub-minute key updates, may offer limited benefit while increasing the risk of cryptographic state divergence in dynamic, lossy networks. In such contexts, lighter or more relaxed security measures could be more appropriate.

This prompts a broader question: how critical are FS and PCS in real-world threat models? Although valuable, these properties introduce complexity and fragility. If key compromise is unlikely and the operational impact is minimal, less frequent updates, such as every few minutes, may strike a better balance between security and stability.

Instead of encrypting video directly with MLS, one alternative is to export an epoch key from the group and use it as a static symmetric key for the streaming session over a separate channel. To preserve some FS, the key could be evolved using a KDF, separating it from those used for control traffic. While this is compatible with the MLS architecture, it introduces coordination challenges, such as selecting and managing exported keys. Supporting this approach would require system extensions to enable flexible key handling for non-critical data like video.

The Current Evidence Remains Limited

We did not prioritize testing video streaming with MLS during the physical testing at Kjeller, as we focused instead on functional and performance evaluation of the system. As a result, we did not test with actual live video transfers and have limited empirical data on the specific impact of video streaming. However, we defend the relevance of our earlier analysis, as video traffic is characterized by high packet frequency, an aspect we did test for. Nonetheless, dedicated testing of encrypted video streaming remains left out as an exercise to the reader. Further experiments are necessary to confirm system robustness and ensure compatibility with real-world swarm scenarios.

Chapter 7
Conclusion

In this thesis, we have designed and implemented Valkyrie MLS, a cryptographic middleware based on OpenMLS [RC25b], and evaluated its suitability and efficiency in a military UAV swarm. We demonstrated that OpenMLS provides the core functionality required to enable secure group communication, but it depends on auxiliary services and well-defined policies to support autonomy and resilience in decentralized, resource-constrained environments.

We have shown how Corosync can be used to provide Totem-based delivery ordering as a DS in our MLS-based system. Additionally, we demonstrated the feasibility of implementing an AS using `Ed25519Credentials`. Assuming pre-shared signature keys and credentials, we can support mutual authentication among swarm nodes without requiring centralized infrastructure. Furthermore, both the `Ed25519Credentials` and X.509 certificates derived from the same key scheme were shown to reduce message overhead for MLS traffic, which is beneficial in environments where bandwidth and latency are critical concerns. To enable autonomous operation of MLS in UAV swarms, we proposed a set of policies and procedures to handle membership changes, key updates, and group recovery in the absence of central coordination. This allows MLS to function in a fully distributed setting, where nodes must operate without persistent communication with the ground control station or centralized services.

Our testing indicates that OpenMLS shows potential for secure group communication in UAV swarms, but there are significant limitations that must be addressed. One of the main challenges is the lack of reliable and ordered message delivery, particularly for handshake messages. In lossy radio environments, the failure to receive such messages leads to group state divergence and forks, which can stall the exchange of application messages. Reestablishing a shared state under these conditions is difficult and time-consuming, especially when nodes operate at the edge of network coverage. Our system does not handle this loss well, as observed in our simulated network tests, and there is a need to overcome the issue of message loss in order to meet requirements of swarm communication. We also observed that Corosync provides

consensus necessary functionality through ordering, but the added coordination needed to achieve this halts operation when nodes operate at the edge of network coverage.

Performance testing on the drone platform demonstrates that Valkyrie MLS, and by extension OpenMLS, achieves promising results in terms of resource usage. By our observed CPU and RAM measurements, we do not regard resource usage as a constraint for OpenMLS. While our results suggest that OpenMLS performs as well as or better than MLS++ under our test conditions, differences in system architecture and methodology complicate direct comparisons. We found that the drone platform favors larger and less frequent handling of application messages, compared to smaller and more frequent messages, and that storing past epochs and allowing for greater forward distance for messaging support a more robust functionality without affecting the performance significantly.

Future Work

Our primary goal was to improve and evaluate MLS-based secure communication in drone swarms, and through this work we have identified multiple directions for further research. First, improving the reliability of handshake message delivery is essential. This could involve introducing transport-layer protocols for ensuring delivery guarantees, like QUIC over UDP, to improve reliability while maintaining a low message overhead. Another approach is to assume message loss is inevitable, and define policies that allow the system to tolerate and recover gracefully when these losses happen to handshake messages. Future work is already initiated in the MLS community [XLH25; Koh25], but is still in an early phase and may not take into consideration adapting the protocol to swarm-specific use cases. Another area of interest is the stabilization of Totem membership when nodes are operating at the edge of network coverage. Frequent join and leave events currently cause disruptions to other system operations.

The interaction between the Totem and MLS group abstractions should also be more tightly coupled, such that membership changes in one are reflected in the other. This would help ensure consistency and reduce operational complexity. Furthermore, the AS could benefit from enhanced capabilities, particularly in supporting dynamic revocation and trust updates. Given the assumptions placed on the AS, future work should also explore alternative group key management schemes, such as symmetric pre-shared key protocols like SAKE-PSK [BDdK+21], which may offer more practical solutions.

Technical Glossary

Technical Glossary

Corosync	Open source implementation of the Totem Single Ring Ordering and Membership protocol.
Flamingo	The ISR drone used in the Valkyrie system.
MGEN	Application-layer traffic generator developed by the U.S. Naval Research Laboratory, used to simulate network traffic patterns in testing and experimentation.
MLS++	C++ implementation of the MLS protocol.
OpenMLS	Rust implementation of the MLS protocol.
pidstat	A Linux performance monitoring tool that reports statistics per process or thread, including CPU usage, memory, and I/O activity.
Totem	Reliable group communication protocol used by Corosync to ensure ordered message delivery in distributed systems.
Traffic Control (tc)	Linux command-line utility used to configure traffic shaping, scheduling, and network emulation features on network interfaces.
Valkyrie	UAV Swarm System developed at FFI.

Valkyrie MLS	Cryptographic middleware developed as the main contribution of this master's thesis. It runs on each drone in the Valkyrie swarm and provides secure group communication by encrypting messages using the MLS protocol. Valkyrie-MLS is implemented in Rust and built on top of the OpenMLS library. The source code is available at https://github.com/mkarder/valkyrie-mls.

List of Acronyms

List of Acronyms

AGKE Authenticated Group Key Exchange.

AKE authenticated key exchange.

API Application Programming Interface.

ART Asynchronous Ratcheting Tree.

AS Authentication Service.

AUW All Up Weight.

C2 command and control.

CA Certificate Authority.

CGKA Continuous Group Key Agreement.

CPG Closed Process Group.

DH Diffie-Hellman.

DHKE Diffie-Hellman key exchange.

DoS Denial-of-Service.

DS Delivery Service.

E2EE End-to-End Encryption.

FFI Norwegian Defence Research Establishment.

FS Forward Secrecy.

GCS Ground Control Station.

GUI graphical user interface.

HPKE Hybrid Public Key Encryption.

IETF Internet Engineering Task Force.

ISR Intelligence, Surveillance and Reconnaissance.

KDF key derivation function.

KEM key encapsulation mechanism.

KT Key Transparency.

MAUI MLS API for Unmanned Surface and Aerial Systems Integration.

MLS Messaging Layer Security.

MTOW maximum takeoff weight.

NTNU Norwegian University of Science and Technology.

OSPF Open Shortest Path First.

PCS Post-Compromise Security.

PFS Perfect Forward Secrecy.

PKI Public Key Infrastructure.

pp percentage points.

PSK Pre-Shared Key.

QUIC Quick UDP Internet Connections.

RFC Request for Comments (IETF specification documents).

ROS Robot Operating System.

SAKE Symmetric-Key Authenticated Key Exchange.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TTT Time to Trigger.

UAS Unmanned Aerial System.

UAV Unmanned Aerial Vehicle.

UDP User Datagram Protocol.

UxS Unmanned Systems.

References

[AB24] M. Arder and E. Bragstad, "Improving Messaging Layer Security in a Military UAV Swarm", Department of Information Security and Communication NTNU – Norwegian University of Science and Technology, Project report in TTM4502, Dec. 2024.

[ACDT20] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis, "Security Analysis and Improvements for the IETF MLS Standard for Group Messaging", in *Advances in Cryptology – CRYPTO 2020*, D. Micciancio and T. Ristenpart, Eds., Cham: Springer International Publishing, 2020, pp. 248–277.

[AMM+95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "The Totem Single-Ring Ordering and Membership Protocol", *ACM Trans. Comput. Syst.*, vol. 13, no. 4, pp. 311–342, Nov. 1995. [Online]. Available: https://doi.org/10.1145/210223.210224 (last visited: May 1, 2025).

[AMT23] J. Alwen, M. Mularczyk, and Y. Tselekounis, "Fork-Resilient Continuous Group Key Agreement", in *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part IV*, Santa Barbara, CA, USA: Springer-Verlag, 2023, pp. 396–429. [Online]. Available: https://doi.org/10.1007/978-3-031-38551-3_13 (last visited: Jun. 1, 2025).

[BBR+23] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon, *The Messaging Layer Security (MLS) Protocol*, RFC 9420, Jul. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9420 (last visited: Jun. 15, 2025).

[BBR18] K. Bhargavan, R. Barnes, and E. Rescorla, "TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS)", Inria Paris, Research Report, May 2018. [Online]. Available: https://inria.hal.science/hal-02425247 (last visited: Jun. 1, 2025).

[BCG23] D. Balbás, D. Collins, and P. Gajland, "WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs", in *Advances in Cryptology – ASIACRYPT 2023: 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4–8, 2023, Proceedings, Part V*, Guangzhou, China: Springer-Verlag, 2023, pp. 307–

341. [Online]. Available: https://doi.org/10.1007/978-981-99-8733-7_10 (last visited: May 15, 2025).

[BDdK+21] C. Boyd, G. T. Davies, B. de Kock, K. Gellert, T. Jager, and L. Millerjord, "Symmetric Key Exchange with Full Forward Security and Robust Synchronization", in *Advances in Cryptology – ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part IV*, Singapore, Singapore: Springer-Verlag, 2021, pp. 681–710. [Online]. Available: https://doi.org/10.1007/978-3-030-92068-5_23 (last visited: Jun. 1, 2025).

[BDM93] M. Barborak, A. Dahbura, and M. Malek, "The Consensus Problem in Fault-Tolerant Computing", *ACM Comput. Surv.*, vol. 25, no. 2, pp. 171–220, Jun. 1993. [Online]. Available: https://doi.org/10.1145/152610.152612 (last visited: Jun. 1, 2025).

[Bre00] E. A. Brewer, "Towards Robust Distributed Systems", in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '00, Portland, Oregon, USA, 2000, pp. 343–477.

[BRO+25] B. Beurdouche, E. Rescorla, E. Omara, S. Inguva, and A. Duric, *The Messaging Layer Security (MLS) Architecture*, RFC 9750, Apr. 2025. [Online]. Available: https://www.rfc-editor.org/info/rfc9750 (last visited: Jun. 15, 2025).

[CCD+17] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A Formal Security Analysis of the Signal Messaging Protocol", in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 451–466.

[CCG+18] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner, "On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees", in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 1802–1819. [Online]. Available: https://doi.org/10.1145/3243734.3243747 (last visited: May 15, 2025).

[DH76] W. Diffie and M. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[Die22] E. Dietz, "Utilizing the Messaging Layer Security Protocol in a Lossy Communications Aerial Swarm", M.S. thesis, Naval Postgraduate School, 2022.

[DJLG94] P. Deutsch, B. Joy, D. Lyon, and J. Gosling, *The Eight Fallacies of Distributed Computing*, Sun Microsystems, 1994. [Online]. Available: https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing (last visited: May 21, 2025).

[Edd22] W. Eddy, *Transmission Control Protocol (TCP)*, RFC 9293, Aug. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9293 (last visited: Jun. 1, 2025).

[EFS17] L. Eggert, G. Fairhurst, and G. Shepherd, *UDP Usage Guidelines*, RFC 8085, Mar. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8085 (last visited: Jun. 1, 2025).

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[Gal12] S. D. Galbraith, *Mathematics of Public Key Cryptography*, 1st ed. Cambridge University Press, 2012.

[GL12] S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem", *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[HBSS24] M. Halsør, D. H. Bentsen, A. S. Simonsen, and H. Stien, "Using Drone Swarms with Manoeuvre Units", Norwegian Defence Research Establishment (FFI), Norway, Technical Report, Oct. 2024, p. 27. [Online]. Available: https://hdl.handle.net/20.500.12242/3333 (last visited: Jun. 1, 2025).

[IT21] J. Iyengar and M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport*, RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000 (last visited: Jun. 1, 2025).

[JL17] S. Josefsson and I. Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, RFC 8032, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8032 (last visited: Jun. 1, 2025).

[Koh25] K. Kohbrok, *Decentralized Messaging Layer Security*, Internet-Draft, work in progress, Mar. 2025. [Online]. Available: https://www.ietf.org/archive/id/draft-kohbrok-mls-decentralized-mls-00.html (last visited: Jun. 1, 2025).

[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, 2023. [Online]. Available: https://doi.org/10.1080/00963402.2023.2178180 (last visited: May 1, 2025).

[LB22] A. Leon and C. J. Britt, "UXS Authentication and Key Exchange Requirements for Multidomain Operation and Joint Interoperability", M.S. thesis, Naval Postgraduate School, 2022.

[LBH24] A. Leon, C. Britt, and B. Hale, "Multi-Device Security Application for Unmanned Surface and Aerial Systems", *Drones*, vol. 8, no. 5, 2024. [Online]. Available: https://www.mdpi.com/2504-446X/8/5/200 (last visited: Jun. 1, 2025).

[LSP82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982. [Online]. Available: https://doi.org/10.1145/357172.357176 (last visited: Jun. 1, 2025).

[Mar14] M. Marlinspike, *Private Group Messaging*, Signal Foundation Blog, 2014. [Online]. Available: https://signal.org/blog/private-groups/ (last visited: May 13, 2025).

[Mar23a] E. Marstrander, *Flamingo-MLS*, 2023. [Online]. Available: https://github.com/emilmarstrander/Flamingo-MLS (last visited: May 14, 2025).

[Mar23b] E. Marstrander, “Use of Messaging Layer Security in a Military UAV Swarm”, M.S. thesis, Norwegian University of Science and Technology, 2023. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3118795 (last visited: Jun. 14, 2025).

[MBB+15] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, “CONIKS: Bringing Key Transparency to End Users”, in *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C.: USENIX Association, Aug. 2015, pp. 383–398. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara (last visited: Jun. 1, 2025).

[MBB+24] E. A. Mentzoni, M. Baksaas, *et al.*, “LandX22 – Experimentation with Uncrewed Systems in a Future Combat Unit”, Norwegian Defence Research Establishment (FFI), Norway, Tech. Rep. ISBN 978-82-464-3540-4, Dec. 2024, p. 26. [Online]. Available: https://nva.sikt.no/registration/0196a492d45b-2215ca74-87b9-4156-bec7-e0b123f3fa62 (last visited: May 15, 2025).

[Moy98] J. Moy, *OSPF Version 2*, RFC 2328, Apr. 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2328 (last visited: Jun. 1, 2025).

[Num21] O. R. Nummedal, “Flamingo - A UAV for Autonomy Research”, Norwegian Defence Research Establishment (FFI), Norway, Tech. Rep., Feb. 2021, p. 26. [Online]. Available: https://hdl.handle.net/20.500.12242/2839 (last visited: Jun. 15, 2025).

[NVI24] NVIDIA Corporation, *Jetson Xavier NX Series*, 2024. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/ (last visited: May 8, 2025).

[PM16] T. Perrin and M. Marlinspike, *Signal: The Double Ratchet Algorithm*, 2016. [Online]. Available: https://signal.org/docs/specifications/doubleratchet/ (last visited: May 12, 2025).

[Pro24] T. C. Project, *Corosync: The Corosync Cluster Engine*, 2024. [Online]. Available: https://github.com/corosync/corosync (last visited: May 14, 2025).

[Raj19] Rajant Corporation, *Rajant DX2 Specification Sheet*, 2019. [Online]. Available: https://rajant.com/wp-content/uploads/2019/12/Rajant_SpecSheet_DX2.pdf (last visited: Jun. 13, 2025).

[RC25a] P. R&D and Cryspen, *OpenMLS Book*, Project documentation, maintained by Phoenix R&D and Cryspen, 2025. [Online]. Available: https://book.openmls.tech/ (last visited: May 14, 2025).

[RC25b] P. R&D and Cryspen, *OpenMLS Docs*, 2025. [Online]. Available: https://openmls.tech/ (last visited: May 14, 2025).

[RC25c] P. R&D and Cryspen, *OpenMLS Rust Documentation*, 2025. [Online]. Available: https://docs.rs/openmls/latest/openmls/ (last visited: May 14, 2025).

[RCB19] R. Robert, K. Cohn-Gordon, and B. Beurdouche, *Messaging Layer Security: Towards a New Era of Secure Messaging*, Presentation at Black Hat USA 2019, Las Vegas, NV, 2019. [Online]. Available: https://www.blackhat.com/us-19/briefings/schedule/#messaging-layer-security-towards-a-new-era-of-secure-group-messaging-16230 (last visited: May 12, 2025).

[Sch14] P. Scharre, "Robotics on the Battlefield Part II: The Coming Swarm", Center for a New American Security, Tech. Rep., Oct. 2014. [Online]. Available: https://www.cnas.org/publications/reports/robotics-on-the-battlefield-part-ii-the-coming-swarm (last visited: May 1, 2025).

[The23] The OpenSSL Project, *OpenSSL 1.1.1 Documentation*, OpenSSL Software Foundation, 2023. [Online]. Available: https://docs.openssl.org/1.1.1/ (last visited: Jun. 1, 2025).

[Tok25] Tokio-rs, *tokio::select! Macro*, 2025. [Online]. Available: https://docs.rs/tokio/latest/tokio/macro.select.html (last visited: May 14, 2025).

[TP88] P. Thambidurai and Y.-k. Park, "Interactive Consistency with Multiple Failure Modes", in *Proceedings [1988] Seventh Symposium on Reliable Distributed Systems*, 1988, pp. 93–100.

[Wha24] WhatsApp, *WhatsApp Encryption Overview*, Aug. 2024. [Online]. Available: https://faq.whatsapp.com/820124435853543 (last visited: May 12, 2025).

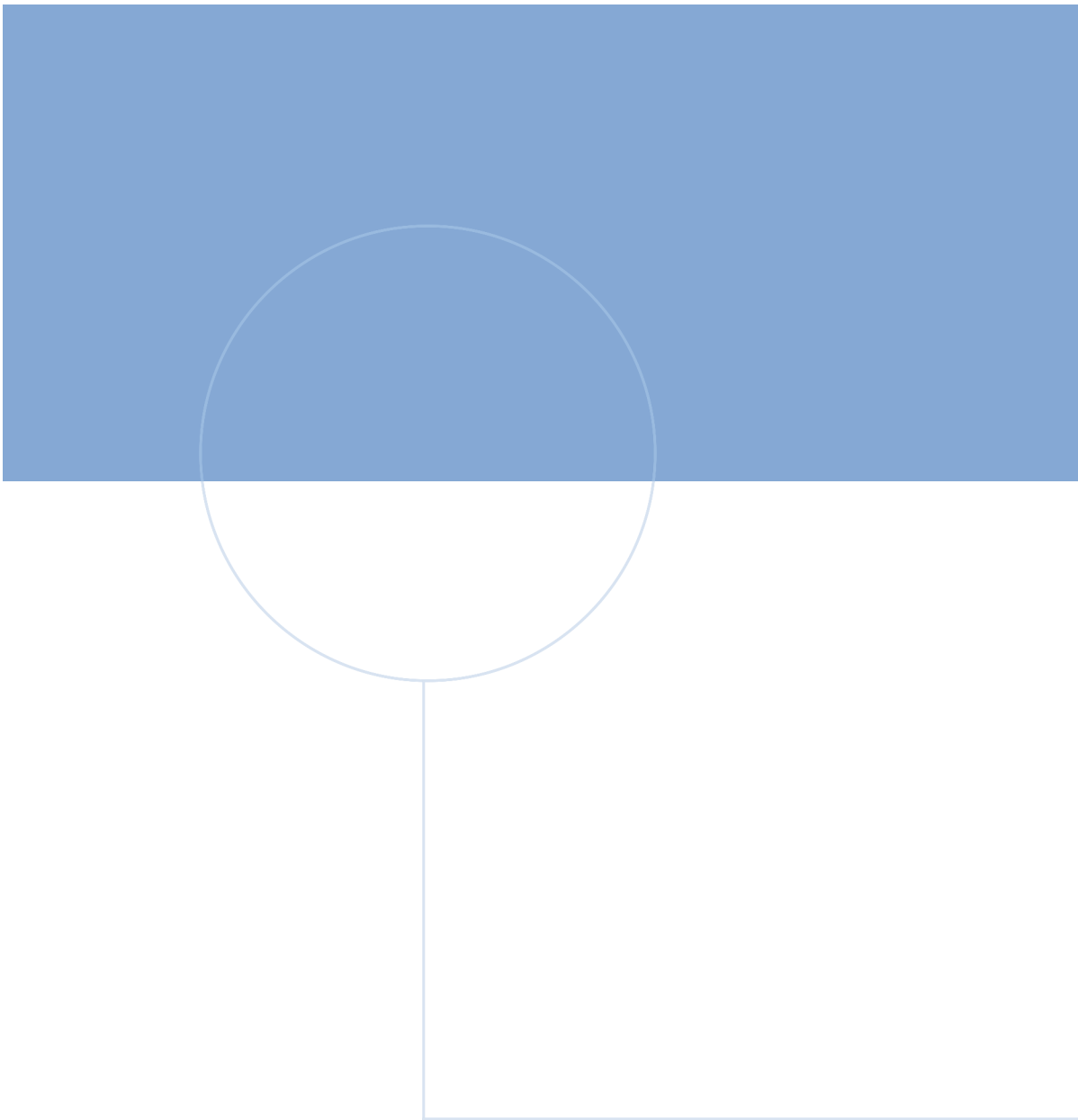
[WPB25] T. Wallez, J. Protzenko, and K. Bhargavan, *TreeKEM: A Modular Machine-Checked Symbolic Security Analysis of Group Key Agreement in Messaging Layer Security*, Cryptology ePrint Archive, Paper 2025/410, 2025. [Online]. Available: https://eprint.iacr.org/2025/410 (last visited: Jun. 1, 2025).

[WPBB23] T. Wallez, J. Protzenko, B. Beurdouche, and K. Bhargavan, "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 (last visited: Jun. 1, 2025).

[XLH25] M. Xue, J. Lukefahr, and B. Hale, *Distributed Messaging Layer Security (MLS)*, Internet-Draft, work in progress, Mar. 2025. [Online]. Available: https://www.ietf.org/archive/id/draft-xue-distributed-mls-00.html (last visited: May 15, 2025).

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

[ZWG+12] Y. Zhang, M. Wu, S. Ge, L. Luan, and A. Zhang, "Optimization of Time-to-Trigger Parameter on Handover Performance in LTE High-Speed Railway Networks", in *The 15th International Symposium on Wireless Personal Multimedia Communications*, 2012, pp. 251–255.



NTNU

Norwegian University of
Science and Technology