2022-03

UTILIZING THE MESSAGING LAYER SECURITY PROTOCOL IN A LOSSY COMMUNICATIONS AERIAL SWARM

Dietz, Elyssa

Monterey, CA; Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

UTILIZING THE MESSAGING LAYER SECURITY PROTOCOL IN A LOSSY COMMUNICATIONS AERIAL SWARM

by

Elyssa Dietz

March 2022

Co-Advisors: Duane T. Davis
Britta Hale

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2022	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
UTILIZING THE MESSAGING LAYER SECURITY PROTOCOL IN A LOSSY COMMUNICATIONS AERIAL SWARM

5. FUNDING NUMBERS

6. AUTHOR(S) Elyssa Dietz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Recent advancements in unmanned aerial vehicle (UAV) capabilities have led to increasing research into swarming systems. Tactical employment of UAV swarms, however, will require secure communications. Unfortunately, efforts to date have not resulted in viable secure communications frameworks. Furthermore, the limited processing power and constrained networking environments that characterize these systems preclude the use of many existing secure group communications protocols. Recent research in secure group communications indicates that the Messaging Layer Security (MLS) protocol might provide an attractive option for these types of systems. This thesis documents the integration of MLS into the Advanced Robotic Systems Engineering Laboratory (ARSENL) UAV swarm system. The ARSENL implementation is intended as a proof-of-concept demonstration of the efficacy of MLS for secure swarm communications. Implementation test results are presented both for experiments conducted in a simulation environment and experiments with physical UAVs. These results indicate that MLS is suitable for a swarm, with the caveat that testing did not implement a delivery mechanism to ensure reliable packet delivery. For future work, mitigation of unreliable communications paths is required if a reliable MLS system is to be maintained.

14. SUBJECT TERMS
unmanned aerial vehicle, UAV, swarm, Messaging Layer Security, MLS, secure group communication, Advanced Robotic Systems Engineering Laboratory, ARSENL

15. NUMBER OF PAGES
89

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

UTILIZING THE MESSAGING LAYER SECURITY PROTOCOL IN A LOSSY COMMUNICATIONS AERIAL SWARM

Elyssa Dietz
Civilian, CyberCorps: Scholarship for Service
BS, University of California – Irvine, 2019
BA, University of California – Irvine, 2019

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2022

Approved by: Duane T. Davis
Co-Advisor

Britta Hale
Co-Advisor

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Recent advancements in unmanned aerial vehicle (UAV) capabilities have led to increasing research into swarming systems. Tactical employment of UAV swarms, however, will require secure communications. Unfortunately, efforts to date have not resulted in viable secure communications frameworks. Furthermore, the limited processing power and constrained networking environments that characterize these systems preclude the use of many existing secure group communications protocols. Recent research in secure group communications indicates that the Messaging Layer Security (MLS) protocol might provide an attractive option for these types of systems. This thesis documents the integration of MLS into the Advanced Robotic Systems Engineering Laboratory (ARSENL) UAV swarm system. The ARSENL implementation is intended as a proof-of-concept demonstration of the efficacy of MLS for secure swarm communications. Implementation test results are presented both for experiments conducted in a simulation environment and experiments with physical UAVs. These results indicate that MLS is suitable for a swarm, with the caveat that testing did not implement a delivery mechanism to ensure reliable packet delivery. For future work, mitigation of unreliable communications paths is required if a reliable MLS system is to be maintained.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

3DES Triple Data Encryption Standard

AES Authenticated Encryption Standard

API application programming interface

ARSENL Advanced Robotic Systems Engineering Laboratory

ART Asynchronous Ratcheting Tree

AS Authentication Service

CCM Counter with Cipher Block Chaining Message Authentication Code

CGKA Continuous Group Key Agreement

DH Diffie-Hellman

EAX Encrypt-then-Authenticate-then-Translate

FANET Flying Ad-Hoc Network

GCM Galois/Counter Mode

IETF Internet Engineering Task Force

KDF Key Derivation Function

KEM Key Encapsulation Mechanism

MAC message authentication code

MLS Message Layer Security

MQTT Message Queue Telemetry Transport

NPS Naval Postgraduate School

NSA National Security Agency

PCS post-compromise security

ROS Robot Operating System

SDN Software Defined Network

SITL software-in-the-loop

SIV Synthetic Initialization Vector

TCP Transmission Control Protocol

TLS Transport Layer Security

UAS unmanned air system

UAV unmanned aerial vehicle

UDP User Datagram Protocol

Acknowledgments

This material is based upon activities supported by the National Science Foundation under Agreement No 1565443. Any opinions, findings, and conclusions or recommendations expressed are those of the author and do not necessarily reflect the views of the National Science Foundation.

Thank you, Dr. Duane Davis and Dr. Britta Hale, for all your help and support with this thesis. I appreciate all of the work you both put in. You helped make this thesis and degree possible, and for that, I am very grateful.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Current advancements in unmanned aerial vehicles (UAVs) have led to research in swarming capabilities. Multi-UAV swarms currently have been suggested or utilized for a wide array of applications including, but not limited to:

- photography [1]
- film-making [1]
- wildfire monitoring [2]
- agriculture [1], [3]
- remote sensing and mapping [4]
- environmental monitoring [5]
- construction [6]
- drone delivery services [7]
- natural disaster management and recovery [8]
- military operations and defense [9]

Tactical utilization of UAV swarms will depend on secure communications. Unfortunately, individual swarming platforms have limited processing power, and swarm systems typically rely on bandwidth-limited and potentially unreliable communications frameworks. These limitations call the ability of these systems to meet security requirements into question.

Previous methods of securing communication for groups of devices are unlikely to be applicable to existing or envisioned swarm systems. Recent research in secure group communications, however, indicates that the Message Layer Security (MLS) protocol [10] can provide an attractive option with characteristics that seem particularly suited to these sorts of systems. This protocol provides a computationally efficient way to implement asynchronous secure group key management, but experimentation in realistic systems is required to assess the protocol's functionality in these computational- and communications-limited environments. This work undertakes implementing the MLS protocol on the Naval Postgraduate School (NPS) Advanced Robotic Systems Engineering Laboratory (ARSENL) UAV swarm for protection of specific information flows.

1.1 Problem Statement

The NPS ARSENL developed and utilizes an unmanned aerial vehicle (UAV) swarm system that has been successfully demonstrated with up to fifty UAVs [11]. Despite the significant potential this capability provides for military operations, the ARSENL system lacks communications security features necessary for eventual real world utilization. This thesis implements MLS on the ARSENL swarm system to assess its suitability for these sorts of systems more broadly.

MLS provides a number of capabilities that are particularly relevant to multi-UAV systems. MLS provides a mechanism for dynamically adding members to and removing members from the group while continuously providing secure communications among members of the group. Adding and removing group members are important capabilities since UAV swarm membership can be highly dynamic. As swarm size increases, the group security protocol must scale efficiently. It is also advantageous that the MLS protocol facilitates forced removal of UAVs that have been hijacked, compromised, or are malfunctioning. In these situations, the protocol provides the group with a means of updating the communication keys to exclude the compromised or malfunctioning UAV. This thesis aims to address the following questions:

1. Can the MLS protocol be adapted for use in the ARSENL UAV swarm?
2. How does MLS impact the performance of the ARSENL UAV swarm?
3. Can ARSENL UAVs join the group and communicate with other members of the swarm securely?
4. Are the group keys able to be updated periodically over the unreliable ARSENL swarm network?
5. In the event of compromise or other criteria, can a UAV be removed from the ARSENL group and no longer decrypt messages?

1.2 Scope

For this thesis, the use of MLS as a continuous group key protocol within the NPS ARSENL UAV swarm is researched. Community maintained C++ code from the MLS GitHub repository [12] is adapted for incorporation into the ARSENL swarm system code base. In particular, the MLS group operations for key update, member addition, and member re-

moval are implemented and tested. The research includes an analysis of the effect of MLS protocol utilization on ARSENL swarm performance. Metrics include packet transmission and receipt rates between individual UAVs, scalability, and timing.

1.3 Thesis Organization

The remainder of this paper is organized into four chapters. Chapter 2 provides background information necessary for understanding MLS and UAV swarms. This includes a discussion of multi-UAV swarms and common swarm communication architectures, the ARSENL swarm system, and potential secure communications approaches to include pairwise and group protocols. The chapter concludes with a discussion of MLS and how it works.

Chapter 3 describes the code development process. It begins with a summary of the ARSENL on-vehicle software's Robot Operating System (ROS) framework and the C++ application programming interface (API) that was utilized to implement MLS functionality. The chapter then discusses the implementation of the code, including a code overview and discussion of lessons learned from integrating MLS into the ARSENL swarm.

Chapter 4 discusses experimentation with the MLS implementation and analyzes its impact on the individual ARSENL swarm platforms as affected by swarm size and key update rate. This chapter includes a description of the testing process and describes the results.

Finally, Chapter 5 provides a conclusion covering the implications of this research and suggestions for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

This chapter provides background information relevant to this research and discusses previous research into UAV swarm communications and the MLS protocol. Information on multi-UAV systems, the NPS ARSENL UAV swarm, and the software-in-the-loop (SITL) simulation environment is included. Chapter 2 also discusses the motivation and research leading up to the introduction of the MLS protocol and provides an overview of the protocol itself.

2.1 Unmanned Aerial Vehicle Swarms

A UAV swarm can be defined as a coordinated unit of relatively simple UAVs that collectively produce a desired result or behavior [13]. In a UAV swarm, information is exchanged between vehicles or between vehicles and ground control stations. By its nature, a properly configured UAV swarm will exhibit a number of particular features:

- Survivability: As opposed to a single-UAV system, a UAV swarm will not experience mission failure if a single UAV fails, malfunctions, or is shot down [14], [15]. The swarm network must operate under the assumption that arbitrary UAVs can be removed from the group without impacting the rest of the swarm's abilities.
- Scalability: With a single UAV, operations have a limited physical area of coverage. Since UAVs can be added to a swarm arbitrarily, the potential range of operations for a mission [14], [16] can increase accordingly.
- Speed: UAV swarms can speed up task completion and mission accomplishment by processing things in parallel [14].
- Autonomy: Single UAV missions typically require a human operator. Direct control during multi-UAV missions is impractical. This necessitates on-board automation to ensure controlled flight and reliable execution of received directives [14].
- Cost: Multi-UAV systems can leverage economies of scale to operate more efficiently and execute missions more cheaply than single-UAV systems [13]–[15].

A swarm's communication architecture dictates how information is exchanged [14] and helps

facilitate the intelligent control and autonomous collaboration within UAV swarms. Swarm systems primarily employ one of two types of communication architectures: centralized or decentralized [14].

Use of a centralized architecture where one node communicates with the entire swarm is a natural extension of the typical single-UAV point-to-point communications paradigm [14]. This sort of centralized architecture can be implemented through a ground station, satellite, airborne platform, or other infrastructure node. Each UAV establishes a one-to-one communications link with the central node. Swarms of this nature work better on a smaller scale. The coverage area for a swarm utilizing a centralized architecture will be relatively small because of the requirement for continuous communications with the central node. Further, communication delays are a possibility since the infrastructure must support reliable transmission of information. One final issue associated with centralized architectures is that they are vulnerable to a single point of failure [14]. A successful attack against the central node or a malfunction within that node will result in failure of the entire swarm, breaking the survivability feature.

Although centralized communication schemes provided a bridge from single-UAV systems to multi-UAV and swarm systems, their disadvantages ultimately outweigh their advantages. Their schemes are not fault tolerant, tend to be inflexible, and are not scalable. As a result, the centralized architecture has been largely abandoned as a viable swarm communication architecture.

In a decentralized architecture, swarm members are organized as nodes in an ad-hoc, peer-to-peer network. The inherent mobility and highly dynamic nature of swarm operations makes an ad-hoc network the obvious choice for flexible and scalable communications [14]. Swarms must be very flexible since they perform tasks in unknown environments with threats and obstacles that require members to dynamically join or leave the group [14]. UAV swarms also exhibit high mobility among individual members meaning that the communications architecture must support frequent topology changes [14].

In many cases, multi-UAV systems can be categorized as Flying Ad-Hoc Networks (FANETs) [16]. Each UAV has its own on-board processing capabilities, maintains its own situational awareness, and makes its own decisions. The reliance on local processing eliminates the need for a centralized control system and is considered a defining character-

istic of swarming behavior [15].

Decentralized systems are more complicated than their centralized counterparts. They tend to exhibit higher power consumption, and they require more processing capability [15]. On the other hand, decentralized architectures allow for flexible and robust networks that eliminate single points of failure and leverage the capabilities of individual platforms. As a result, modern swarm systems rely almost exclusively on these sorts of architectures [16].

2.1.1 Related Work

Swarming is a popular emerging field among robotics researchers. Much of this research has been focused on reliable swarm communication architectures.

Researchers from the Army Engineering University of the Chinese People's Liberation Army and Chengdu University proposed a battlefield UAV swarm that utilizes a Software Defined Network (SDN) and Message Queue Telemetry Transport (MQTT) hybrid structure [9]. UAVs have limited battery capacity, and their on-board computational and communication is also frequently limited. MQTT, a publisher-subscriber middleware, is designed to function on devices with computation- and communication-constrained networks, low processing, and low memory [9]. SDN, on the other hand, is well suited to address the swarming requirement for a robust, self-organizing, and delay tolerant network. The researchers suggest that a combined SDN-MQTT network structure can be utilized by swarm systems to effectively in operational military environments. As of the publication of [9], the researchers have proposed the system but have not yet implemented the network structure in a real environment.

In 2015, Rosati et al. [2] proposed a FANET using optimized link-state routing and predictive optimized link-state routing for swarm communications. Optimized link-state routing is a network routing algorithm and predictive optimized link-state routing is an extension of optimized link-state routing that uses GPS data to improve routing. Messages within this system are transmitted over an 802.11n network. When using predictive optimized link-state routing, each UAV transmits its latitude, longitude, and altitude to each of its neighbors. The researchers conducted experiments with this network architecture with a system comprised of two fixed wing UAVs and a single ground station. The researchers found that predictive optimized link-state routing outperformed optimized link-state routing.

Their results indicated that predictive optimized link-state routing is faster to respond to topology changes and did so without interruptions.

Researchers from the University of North Dakota proposed a UAV swarm architecture relying on cellular mobile network infrastructure [13]. In this architecture, the UAVs communicate among themselves without the need for a ground control station. Telemetry data for each UAV is sent to other UAVs via cellular mobile infrastructure. This approach differs from a typical FANET in that the UAVs make the decisions about the network protocol and flight control, while the infrastructure is responsible for routing decisions. By relying on cellular networks, the UAVs can be dispersed across a larger area while also achieving more reliable data transfers.

Other research using cellular networks was proposed by Han et al. [17] in the form of a two-phase transmission protocol. The protocol leverages existing 4G and 5G cellular networks and also device-to-device communications capabilities. In the first phase of the protocol, a set of ground control stations transmits a control message to all vehicles simultaneously. In the second phase, all vehicles that have received a control message forward it to other vehicles in the swarm using device-to-device communications. This two-phase transmission protocol was shown to provide high reliability and low latency for communications within the swarm. The researchers modeled the reliability of communications as an approximated expression using Pearson distributions and gathered numerical results to prove the effectiveness of their protocol.

Of note, none of the approaches discussed here deal with communications security directly. Rather, they implicitly assume that it is handled by the infrastructure (layer 2 mostly–802.11, cell, etc.) or that it will be implemented on top of the communications architecture at the application layer. The research in this thesis aims to investigate an application-layer security architecture that can be adapted into swarming technology.

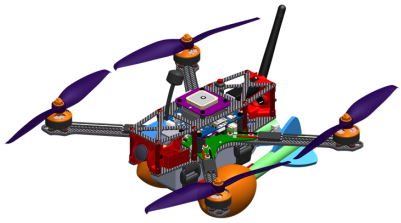
2.1.2 The NPS Swarm: ARSENL

The NPS Advanced Robotic Systems Engineering Laboratory (ARSENL) is a research group focusing on the development of swarm capabilities. The lab's work resulted in the successful demonstration of a swarm of 50 fixed-wing UAVs at Camp Roberts, California, in 2015 [11]. Subsequent ARSENL research has led to advancements in human-UAV

interaction, information exchange within the swarm, swarm behavior development, and heterogeneous and multi-domain swarming [18]–[21].

The ARSENL swarm system consists of UAVs and two ground stations. Two main UAVs are used: the ARSENL-developed Mosquito Hawk quadcopter and a modified Ritewing Zephyr II fixed wing UAV (Figure 2.1). Both UAVs use commercial off-the-shelf and 3D-printed parts [18]. A Pixhawk autopilot running ArduPlane autopilot firmware maintains safe flight and controls waypoint navigation [11]. On-board planning and coordination in support of swarming is conducted on a HardKernel ODroid companion computer [22]. The autonomy package is implemented on the companion computer as a set of ROS nodes that manage specific functions such as behavior planning, autopilot interaction, and communication with other swarm UAVs [11]. The ROS framework is described in more detail in Section 3.2.

ZephryII Fixed-Wing UAV. Source: [11].



Mosquito Hawk Quadrotor UAV. Source: [21].

Figure 2.1. ARSENL Multi-UAV System Platforms.

UAV-to-ground communication is required to parameterize, initiate, and monitor swarm behaviors and UAV-to-UAV communication is used to facilitate swarm behavior execution. In particular, UAVs exchange state information to facilitate situational awareness across the swarm [11]. Communications use an ARSENL-developed application layer protocol and User Datagram Protocol (UDP) messages broadcast over an 802.11n ad-hoc network [11]. Since secure swarm communications has not previously been an ARSENL research priority, encryption has not been used, and swarm communications currently utilize an unsecure network.

On-UAV mission execution is implemented as a finite state machine. Each UAV transitions through the following states over the course of a mission [11]:

1. Preflight: system health verifications, mission and software loading, installing the

battery and camera, etc.,
2. Flight ready: ready for final mission parameter load and automatic launch,
3. Ingress: safe movement of the UAV after takeoff to the flight operating zone,
4. Swarm ready: ready for sub-swarm assignment, waiting for initiation of a swarm behavior, or actively executing a swarm behavior,
5. Landing: sequential landing of UAVs, and
6. On deck and post-flight: retrieve and power-off UAVs followed by post-flight inspection, maintenance, and repair.

Swarm behaviors are executed by vehicles in the *swarm ready* state as directed by the ground station.

Software-in-the-loop (SITL) Simulation

The ARSENL SITL simulation environment provides a way to evaluate the effectiveness and performance of the UAV swarm without launching actual aircraft [18]. The SITL environment provides a realistic physically based simulation for use by UAVs using ArduPilot autopilot firmware [23]. Since the same ArduPilot firmware is used in the simulation as in the actual autopilot, the autonomy implementation, swarm interactions, and behavior performance can be thoroughly tested in simulation prior to attempting live experiments.

The ARSENL simulation system can be used to spawn multiple UAV instances and provides accurate information concerning the flight environment, individual UAV performance, interactions between UAVs, and interactions between ground stations and the swarm. The SITL simulation can use an internal network to allow multiple instances to be simulated on a single computer, or it can use an actual network to allow a simulation to be run across multiple computers [18]. The SITL simulation allows for accurate testing of swarm capabilities and aids in development of further advancements.

Secure ARSENL Communications

Secure communications within the ARSENL swarm are constrained by limited computational, network, and energy resources that potentially inhibit the use of complex security schemes. Thus, the ARSENL swarm system does not currently attempt to provide security for its communications. Operational swarms in military and many civilian environments,

however, have the potential to handle classified or sensitive data. Failure to provide viable communications security options will hamper the adoption of swarm systems for real world applications.

Since swarm communications often rely on broadcast messages and low bandwidth links, many common forms of asymmetric cryptography are not well suited. Using some typical forms of asymmetric cryptography would require restructuring the communication architecture in ways that may inhibit effective swarm operations.

The Authenticated Encryption Standard (AES) and Triple Data Encryption Standard (3DES) symmetric encryption algorithms are authorized for the encryption of classified data and were explored as options for secure swarm communications in [24]. Four AES modes were tested in a UAV swarm: Counter with Cipher Block Chaining Message Authentication Code (CCM), Galois/Counter Mode (GCM), Synthetic Initialization Vector (SIV), and Encrypt-then-Authenticate-then-Translate (EAX). The researchers also implemented the scheme ChaCha20-Poly1305, for the transmission of unclassified data. All of these encryption algorithms were implemented using Python libraries and encryption was done at the application layer. After testing these algorithms within the SITL simulation system discussed in Section 2.1.2 and the ODroid architecture discussed in Section 2.1.2, the researchers found that SIV and EAX modes were unable to support a large swarm within their architecture. GCM and CCM modes were assessed as potentially able to support a large swarm within their architecture, but it was noted that little processing power would be left to other processes. ChaCha20-Poly1305 was assessed as suitable for a large swarm, but only when transmitting unclassified data, as it is not approved by the National Security Agency (NSA) for classified information communications. Of note, [24] determined that none of the encryption algorithms they tested provided efficient enough cryptographic operations for classified information. When they tested the approved-for-classified-information algorithms on the Odroid computer with a 50 member swarm, they found that the processor was completely taken over by cryptographic operations. This left little to no processing for other tasks. The authors note that the encryption schemes could work if more powerful processors were available or if encryption was only used on a subset of the communications. Even in these cases, however, there will be a performance impact on the UAV swarm.

The goal of this research is to further investigate viable options for secure communications

within an unmanned air system (UAS). Towards that end, the rest of this chapter discusses candidate secure communication protocols in general and the research leading to the decision to test MLS in the ARSENL swarm in particular.

2.2 Pairwise Communication

This section provides a background on pairwise, or point-to-point communications, and discusses popular security protocols for these types of communications. Pairwise communication protocols work by exchanging a message between N members over $N-1$ channels with each channel encrypted separately. Pairwise communications can be synchronous, where parties must be online at the same time, or asynchronous, where the parties do not need to be online at the same time. In this section, we compare two common pairwise communication protocols based on differences in foundational design characteristics (synchronous sessions vs continuous key exchange): Transport Layer Security (TLS) [25] and Signal [26].

2.2.1 Transport Layer Security

As the Internet became popular, a secure communications protocol was needed. Transport Layer Security (TLS) [27] is a synchronous, secure-channel protocol used for communications over the Internet. TLS provides end-to-end encryption for short lived communications such as interaction with a secure web-page. Other applications of this protocol can include email, instant messaging, and voice over IP [28]. The first version of TLS was published in 1999 by an Internet Engineering Task Force (IETF) working group. The latest version is TLS 1.3 and was published in 2018 [27]. TLS provides for encryption, authentication, and integrity [25].

TLS works by initiating a handshake to establish communication between two parties. The handshake is used to specify the protocol version, agree on a cipher suite or set of cryptographic algorithms, optionally mutually authenticate, and establish session keys for message encryption [25]. Trust and authentication are established via certificates issued by a certificate authority. Data is signed with a message authentication code (MAC) to ensure integrity of data. Improvements to TLS in version 1.3 include improving speed and security by decreasing handshake round trips and removing insecure ciphersuite options.

For applications where users may not be online at the same time, TLS is not an option. Asynchronous protocols like Signal were proposed to mitigate this issue.

2.2.2 Signal

The Signal protocol [26] is used in popular messaging applications such as WhatsApp, Facebook Messenger, Skype, Google Allo, and Wire [29], [30]. The Signal protocol provides end-to-end encryption and has strong security guarantees such as post-compromise security (PCS) and forward secrecy [29].

The term PCS was introduced in [31] to refer to the notion that security will be recovered after a compromise occurs if the adversary is momentarily passive. PCS is an important property since it limits the scope of a compromise. That is, even a successful attacker that does not act as an active man-in-the-middle attacker will not be able to maintain its access to group communications. Forward secrecy is the notion that messages sent and received prior to a compromise remain secure [32]. PCS and forward secrecy are conceptually illustrated in Figures 2.2 and 2.3 respectively. Most modern secure messaging protocols require that both forward secrecy and PCS are provided. This guarantees that current data remains secure despite past or future compromises [32].

At the core of Signal is the double ratchet algorithm [29]. Double ratcheting allows for parties to heal after compromise and achieve PCS by updating keys through a process referred to as key ratcheting.

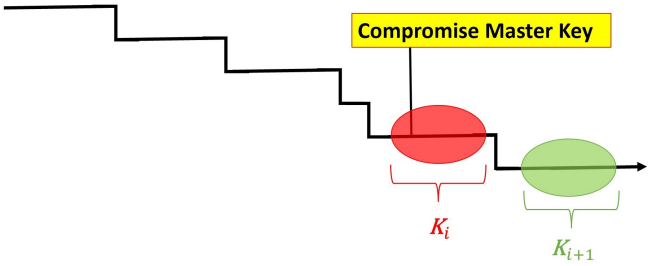


Figure 2.2. After a compromise, post compromise security ensures that keys going forward remain secure if an adversary is momentarily passive. Source: [33].

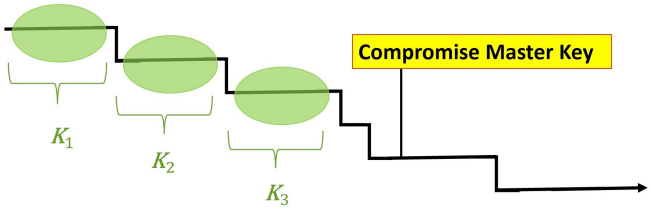


Figure 2.3. After a compromise, forward secrecy ensures that the keys derived prior to the compromise remain secure. Source: [33].

Double Ratchet

In the double ratchet protocol, two users exchange messages encrypted using a key derived from a key agreement protocol [34]. After deriving a shared key, users use the double ratchet to send and receive messages, ratcheting keys forward to provide PCS and forward secrecy.

The double ratchet algorithm works by deriving new keys following every message using a one-way function [34]. This ensures that earlier keys cannot be calculated from later keys. The algorithm also ensures that later keys cannot be calculated from earlier keys by adding Diffie-Hellman (DH) values to every messages.

An important core component of the double ratchet is a Key Derivation Function (KDF) [34]. A KDF is a cryptographic function that derives one or more keys from a secret value and input data [35]. The derived keys can be any specified length. If the first input into the KDF is random, output data from the KDF is indistinguishable from random (i.e., pseudorandom). Each user stores keys for three chains: a root chain, a sending chain, and a receiving chain. Two ratchets are leveraged: a symmetric-key ratchet and a Diffie-Hellman key exchange ratchet.

A symmetric-key ratchet process is used to derive message keys. A KDF chain takes the output from a KDF and uses it as part of the input to the next KDF computation. After decrypting a message, the message key can be deleted from memory. A symmetric-key ratchet uses these principles and produces a chain key and a message key with every round of the ratchet, as shown in Figure 2.4.

A DH ratchet is also used in the double ratchet algorithm so that both users provide a key

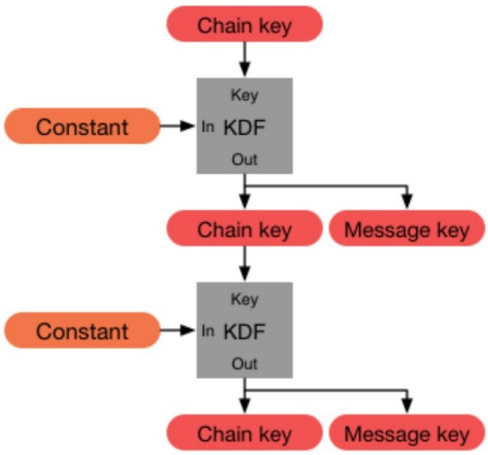


Figure 2.4. KDF Chain used in Symmetric-Ratchet step. Source: [34].

share to key derivations for security [34]. In a DH ratchet, the ratchet key is a DH key consisting of key shares generated by each party. A new public key share is sent with each message. After the message is received, the local copy of the ratchet key is replaced with the new key. This ping-pong behavior continues with each iteration of the ratchet. The DH output produced by each user is used to derive sending and receiving chain keys [34]. Each output is used as a KDF input in the root chain, which in turn produces outputs used in the sending and receiving chains.

The double ratchet algorithm is a combination of symmetric-key ratchet and DH ratchet [34]. The DH ratchet is first used when a public key is received to replace the chain keys, then the symmetric-key ratchet is applied to derive the message key when messages are sent and received. Figure 2.5 shows the double ratchet algorithm used in Signal.

For group messaging, Signal uses pairwise Signal channels within groups [36]. Pairwise communications likely have too high of communication and computation overhead for applications in UAVs or for very large groups in general. The double ratchet must be computed between all pairs of users, so the number of channels required scales quadratically with the group size [32], [37]. A description of a variant of the Signal protocol that achieves more efficient group communications is provided in 2.3.1.

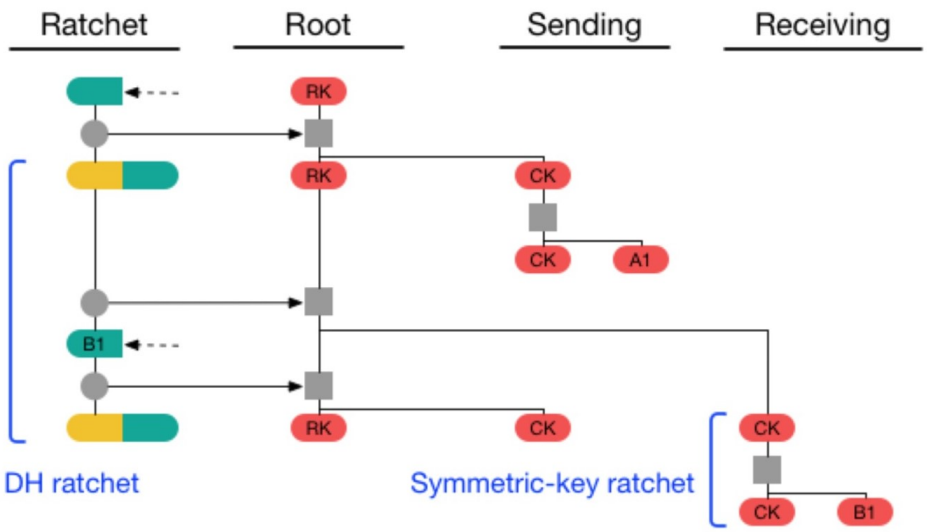


Figure 2.5. The Double Ratchet algorithm in Signal. Source: [34].

2.3 Group Protocols

In this section, secure group messaging protocols are described. With the popularity of group messaging applications, a need for a protocol for groups that is asynchronous, scales well, and is long lived was identified. Desirable security properties include PCS and forward secrecy as well as the ability to add and remove group members dynamically.

2.3.1 Sender Keys

In pairwise communication protocols, messages are sent over multiple point-to-point channels. For example, if Alice is member A of a five-member group, **G**, consisting of members A, B, C, D, and E and wants to send a message to the group, she needs to send it across four pairwise channels: between A and B, A and C, A and D, and A and E. If a member is in multiple groups, the member will send the same message over each of the separate channels. So in groups **G** and **G'**, A will send the message over the single channel it has with B regardless of the group in which it is communicating [37].

To mitigate the high overhead of this action in Signal, a variant called Sender Keys was

created. With Sender Keys, each member of the group sends an encrypted "broadcast" key to each group member over the pairwise channels [36]. Future messages are then encrypted using the broadcast keys. In the above example, A would encrypt a single broadcast key k and send it over her pairwise channels with B, C, D, and E. Messages from A would then be encrypted using her broadcast key k and broadcast to the entire group. Each member of the group must repeat this process. Overhead is thus reduced for future communications; however, some of the security properties of the double ratchet are lost. Sender Keys, for instance, do not provide efficient ways to update keys or remove group members [38]. Adding a ratchet to this algorithm can provide forward secrecy, but Sender Keys still do not efficiently provide for PCS for the group [10]. Achieving both guarantees essentially regresses the protocol back to the regular pairwise Signal protocol.

2.3.2 Tree-based Group Protocols

A Continuous Group Key Agreement (CGKA) protocol allows a group to continuously agree on a stream of fresh secret group keys [32]. This type of protocol provides the ability to add and remove members efficiently, as well as to conduct updates. Updates are used to introduce random values, otherwise known as update secrets [32], which allow users to refresh group key material and provide for stronger security properties such as PCS (much like the Signal DH ratchet values). Updates occur at regular intervals throughout the lifetime of the group, but the exact frequency is based on implementation choices [10].

CGKA protocols typically rely on a tree data structure where leaf nodes represent group members and the root node represents the group secret. The group protocols described here use binary trees. In these protocols, the path from a leaf node to the root node is referred to as a direct path, and the immediate sibling of the leaf node and other siblings of nodes on the direct path are referred to as the co-path. Figure 2.6 shows a diagram of a binary tree that includes the direct path and co-paths associated with member A.

Tree-based protocols fix many of the shortcomings associated with pairwise group communications. Asynchronous Ratcheting Tree (ART) and TreeKEM are examples of such CGKA protocols that have been considered for use in MLS [39].

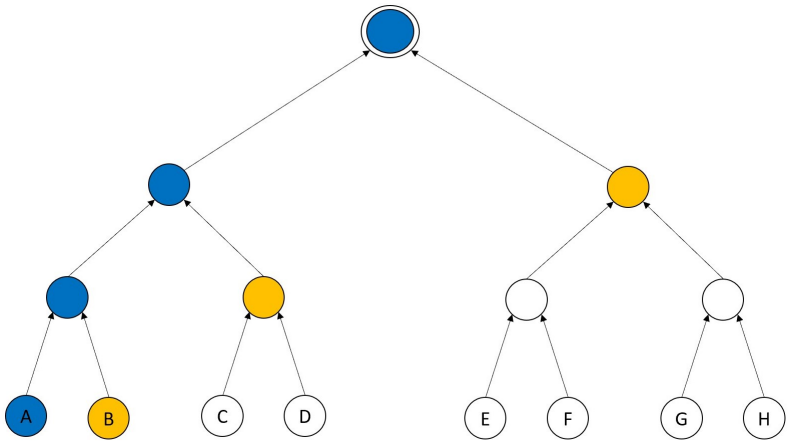


Figure 2.6. This tree represents a group with members A through H. The direct path for member A is highlighted in blue, and the co-path for member A is highlighted in orange. The root node with a double circle represents the group secret.

Asynchronous Ratcheting Tree

As noted previously, Sender Keys do not achieve PCS and scale inefficiently for large groups. PCS and scalability were the driving motivations behind the creation of Asynchronous Ratcheting Tree (ART) [36]. ART is an asynchronous group messaging protocol that uses tree-based Diffie-Hellman key exchange to derive a group shared secret. By using a tree structure, secure group messaging is more efficiently achieved. In ART, the group is modeled as a binary tree structure with each leaf representing a member or user in the group. Leaf nodes are labeled with an ElGamal secret key, x_i, and a public value, g^x [39]. An internal node with children i and j will contain a shared secret value equal to g^{x_i,x_j} and a public value of $g^{t(g^{x_i,x_j})}$ where t is mapping from group elements to integers [36], [39]. An example of an ART group is illustrated in Figure 2.7.

In an ART tree, leaf node keys are derived using the session keys of any one-round authenticated key exchange protocol [36]. ART also uses prekeys and setup keys [36]. A prekey is a DH ephemeral public key that is stored by an untrusted server and fetched by users when needed. A setup key is generated locally by the initiator of the group and is used to perform the initial key exchange with the prekeys. This approach allows for an initial key exchange while some group members may be offline.

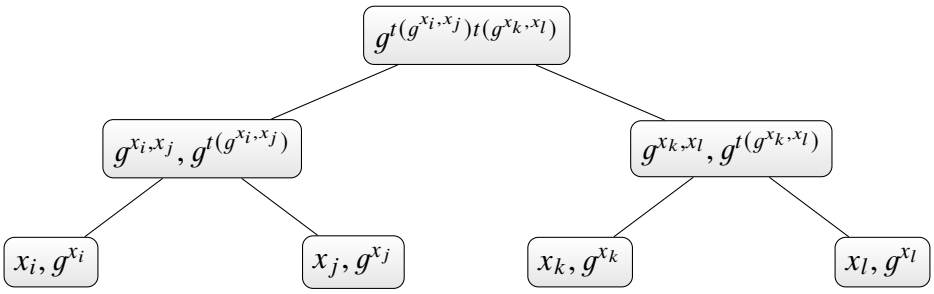


Figure 2.7. An ART group with four members. Each node contains a secret key, public key pair. The root node contains the group secret, and the leaf nodes represent group members. Adapted from [36].

Upon group creation, an initiator first creates the group. It then broadcasts the public prekeys and identities, the public setup keys, and the tree of public keys to every member of the group. The initiator also generates and broadcasts a signature over the broadcasted keys and identities to the group using its own identity key. As each member of the group receives these keys, they validate the signature, compute their leaf keys, extract the co-path of the public keys from the tree, and generate the shared symmetric key for the group at the root node. To generate the shared key, all public keys on the co-path are repeatedly exponentiated [36].

Updates are essential to providing PCS [36]. An update occurs when a user updates their secret leaf key from x to x' and from there establishes a new shared group key. The user will compute a new direct path to the root using their new key and will send the public values on their direct path to the group so the group members can update their views of the tree.

The TreeKEM Protocol

The TreeKEM protocol was inspired by ART [40] and is at the core of the current draft of MLS. The initial idea is described in [41] and utilizes a left-balanced binary ratchet tree structure to maintain the group state and keys [32]. Similar to ART, members or users of the group are represented by leaves, and the root contains the group shared secret. TreeKEM differs from ART in that each internal node contains a Key Encapsulation Mechanism (KEM) public and secret key pair. TreeKEM key pairs can be any pair of keys that support KEM, whereas ART specifically requires a DH key pair.

An internal TreeKEM node's secret key is known to every member below it in the sub-tree,

and its public key is known to every member of the group. Thus, each member of the group knows the root node secret key, the secret keys for all nodes in the path from the root node to its leaf node, and public keys for all nodes in the tree. After the initial group creation, there are three major operations associated with the TreeKEM protocol: adding members, updating leaf secrets, and removing members. These operations can occur asynchronously.

On creation of the group, a fresh KEM key pair is generated for each user at each leaf node. The secret key of any internal node is computed by application of the KDF to the secret key of the updating child node. In the case of creation, the updating node is the member initiating the group. The root node key is likewise calculated by applying the KDF to the secret key of the child of the root node that is on the direct path of the updating member. A TreeKEM group is illustrated in Figure 2.8.

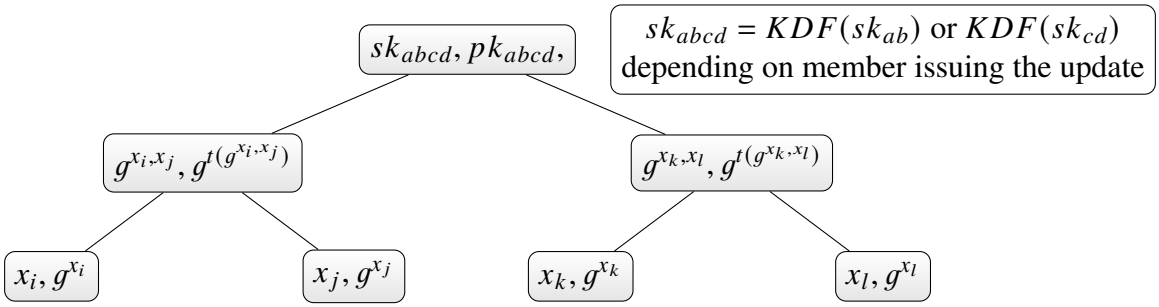


Figure 2.8. A TreeKEM group with four members. Each node represents a secret key and public key pair in the depicted order. The root node contains the group secret, and the leaf nodes represent group members. Subscripts indicate secret key access by identities. Adapted from: [33].

When members are added to the group, they use a public key corresponding to the secret key known only to the new group member. The public key is broadcast to the group and stored in a leaf node. A new group secret is then computed, encrypted, and sent to all other members of the group. All members decrypt the message and use the new secret to update their locally maintained views of the tree. As a result, each member knows the public keys of all other nodes in the tree and the secret keys of all nodes in its direct path.

To initiate an update, a member generates a new key pair and derives a new group secret. It then encrypts the secret key of the path node and sends it to the nodes on the co-path. An update operation initiated by member D of a hypothetical group is shown in Figure 2.9.

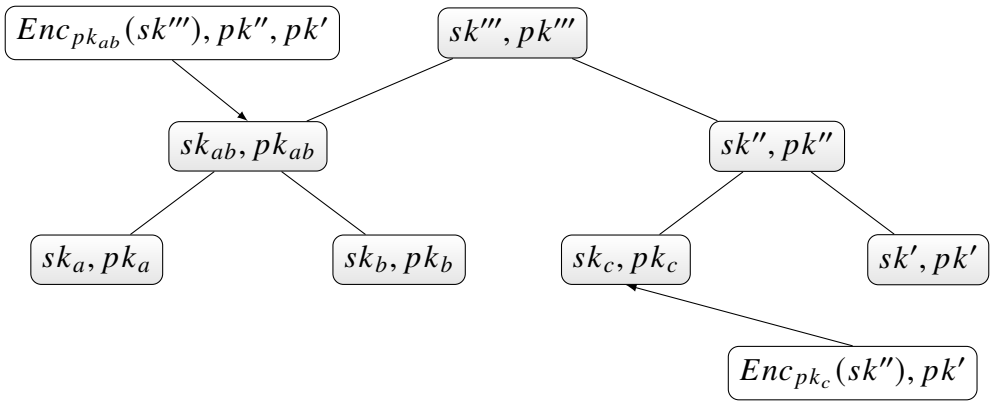


Figure 2.9. A TreeKEM group with four members. Subscripts indicate secret key access by identities. After member D initiates an update, it must send keys to other members of the group so that they can update their tree copy. Member D sends a message to each node on its co-path that is encrypted with the public key of that node. Adapted from: [33].

Removal of a member from the group necessitates an update of all secret information to which the removed member had access. When a member, A, is removed, for instance, another member of the group computes a new group key. The key is then encrypted for all members of the group except A using public keys for which A does not know the private keys. Upon receipt, members will decrypt the new group key and use it to update their locally maintained views of the tree.

One potential issue in TreeKEM arises when a member, A, adds or removes a member, B. Since A performed the add or remove operation, A has knowledge of the secret keys on B's direct path, which can be a problem if A is compromised. Even if the direct path for A is updated and healed, A had knowledge of the secret keys on B's direct path and can still compute the group secret [39]. TreeKEM with blanking, or TreeKEM$_B$, was proposed to mitigate this issue.

With TreeKEM, if a member adds or removes another member in the group, they will know a secret key for a node not in their direct path. With TreeKEM$_B$, any node for which a secret key would be known by a member of the group not in that node's direct path will be "blanked," or not assigned a key-pair value. This ensures that a group member only knows the secrets on their own direct path to the root [39]. When something needs to be encrypted

to a blanked node, it will instead be encrypted to the children of the blanked node, with this operation occurring recursively until a non-blanked node is reached.

2.3.3 Tainted TreeKEM

A variant of TreeKEM called Tainted TreeKEM has been proposed for MLS [39]. This protocol differs from TreeKEM mainly in the way members are added or removed. Instead of blanking nodes in the tree, Tainted TreeKEM proposes keeping track of the nodes that would be blanked under TreeKEM$_B$ and which secret keys have been created by which members of the group. Rather than blanking these nodes, they are marked as tainted. A tainted node is used as a regular node, meaning that all other members of the group treat this node normally for group operations. The tainted node will heal in a manner similar to a TreeKEM$_B$ blanked node when an update is performed. The efficiency of tainting versus blanking nodes depends on the sequence of operations that are performed in the group [39]. The security of this protocol is very similar to that of TreeKEM$_B$.

2.4 MLS

In 2017 the IETF established a working group devoted to designing a protocol for secure group messaging called Message Layer Security (MLS) [10] that would not be subject to the computational limitations of pairwise communications. The MLS IETF working group has members from Cisco, Google, Cloudflare, Facebook, Wickr, Wire, Twitter, and other stakeholders [32] who aim to establish a group messaging protocol that guarantees important security claims without compromising performance. Working MLS implementations of Draft 11 [42] are currently provided in C++ [12] and Rust [43].

Secure group messaging protocols like MLS use group keys maintained by tree structures to provide secure messaging. MLS aims to provide secure asynchronous group communication where members can send and receive confidential and authenticated messages [10]. In the early iterations of the protocol's development, MLS relied on ART [41], however the IETF removed ART in MLS Draft 2 [10]. As of Draft 7, the TreeKEM$_B$ protocol is at the core of MLS [10], [37].

Key Schedule

TreeKEM is not secure on its own since group keys become corrupt whenever one group member is corrupted. In order for the group to be secure, a mechanism is needed to chain keys across epochs. An epoch is defined as a period of time when a specific group key is in use [10], so each TreeKEM state is effectively confined to a specific epoch. MLS implements this through a key schedule that is updated with every update operation performed (i.e., the group transitions to a new epoch with each update). The MLS key schedule uses KDFs to chain the keys across epochs by combining new keying material with old keys [40]. The combination of TreeKEM and a key derivation schedule provides the security guarantees of PCS and forward secrecy [40].

Proposal and Commit

MLS has three major operations that can be performed on a group: update a member's secret keys, add a member, and remove a member [10]. When performing one of these actions, a member first proposes the action by sending a proposal message with the operation type (e.g. **Add**, **Update**, and **Remove**). Following this message, the member sends a **Commit** message to finalize the operation. The state of the tree is not affected until the **Commit** message is sent and processed. New shared secrets are contained within a **Commit** message and a member can process multiple proposals in one **Commit** [10].

Update

Updates occur to change a member's leaf secret and update the direct path of that member. Each update performed by a user results in an update secret, or high-entropy random variables that is used to refresh group keying material [32]. The MLS protocol ensures that each user receives the same update secret, that the update secret remains secure to anyone outside the group, and that past update secrets remain confidential. In combination, these characteristics assure PCS [32].

The period between updates is determined by the application and not the MLS protocol itself [10]. Any member can initiate an update at any time by creating an **Update** proposal message and a **Commit** message and broadcasting them to the group. An example update is illustrated in Figure 2.10.

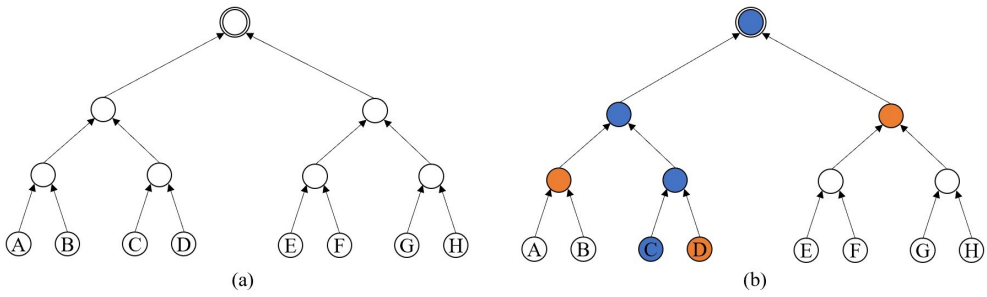


Figure 2.10. The state of a hypothetical group prior to an update is shown in (a). The state of the tree during an update is shown in (b). When member C wants to initiate an update, the member samples new public and secret key values for each node in the direct path, highlighted in blue. It then sends the secret keys to all nodes in the co-path, highlighted in orange, so that each member knows the secret values of the nodes in their path. Adapted from [40].

Initializing a Group and Adding Members

Each prospective member generates a key package prior to entering the group. The first member to establish a group initializes the group to contain only themselves. Subsequent additions are then performed by existing group members.

If a member, A, is adding a new member, D, to the group, A generates **Add** and **Commit** messages and broadcasts them to the group. A then generates a **Welcome** message containing the group state and sends it to D along with a **Commit** message. When D receives the **Welcome** and **Commit** messages, it establishes its view of the group so that it can communicate securely within the group. This process is illustrated in Figure 2.11.

Removing a group member

Removal of a member simply requires sending notification to the rest of the group and blanking all nodes in the removed member's direct path. For instance, for a member, D, to remove a member, C, from the group, D sends a **Remove** proposal and a **Commit** message to all members of the group except for node C. The remove process will blank the leaf node, C, and all nodes in C's direct path.

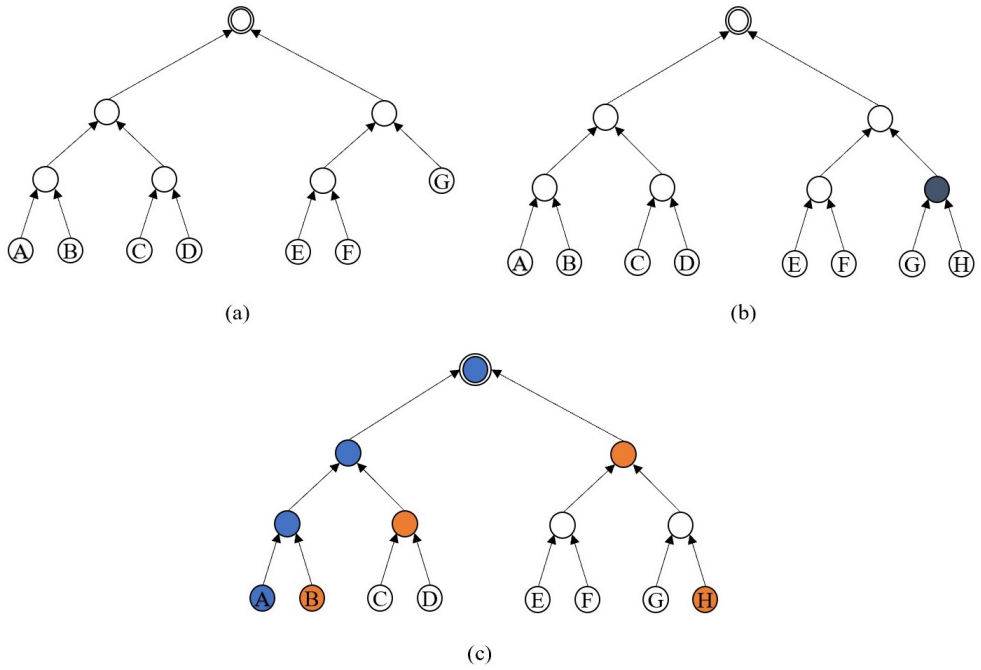


Figure 2.11. The state of a hypothetical group prior to the addition of a new member is shown in (a). When member A wants to add new member H, A first creates a blanked node and adds that to the tree in the place of G. A then shifts the position of G as a child of the new blanked node and adds H to the tree as a leaf node as well, as shown in (b) (blanked nodes are shown in gray). G retains the keys it had before. In (c), A performs an update and sends information to H regarding the tree status and also sends encrypted keys of the parent to H's direct parent (i.e., next unblanked node in its direct path) to H. Adapted from [40].

MLS Security Analysis

The security of MLS has been analyzed in several papers, and a summary of those findings is provided here.

In [32], the authors analyzed the security of the TreeKEM protocol, the core of MLS. They formally defined a CGKA protocol and outlined the security notions for this protocol. They showed that TreeKEM provided PCS, but only weak forward secrecy. Forward secrecy is violated because update secrets are kept since they are needed to process future updates. If an adversary corrupts a user other than the update initiator, they can gain access to older

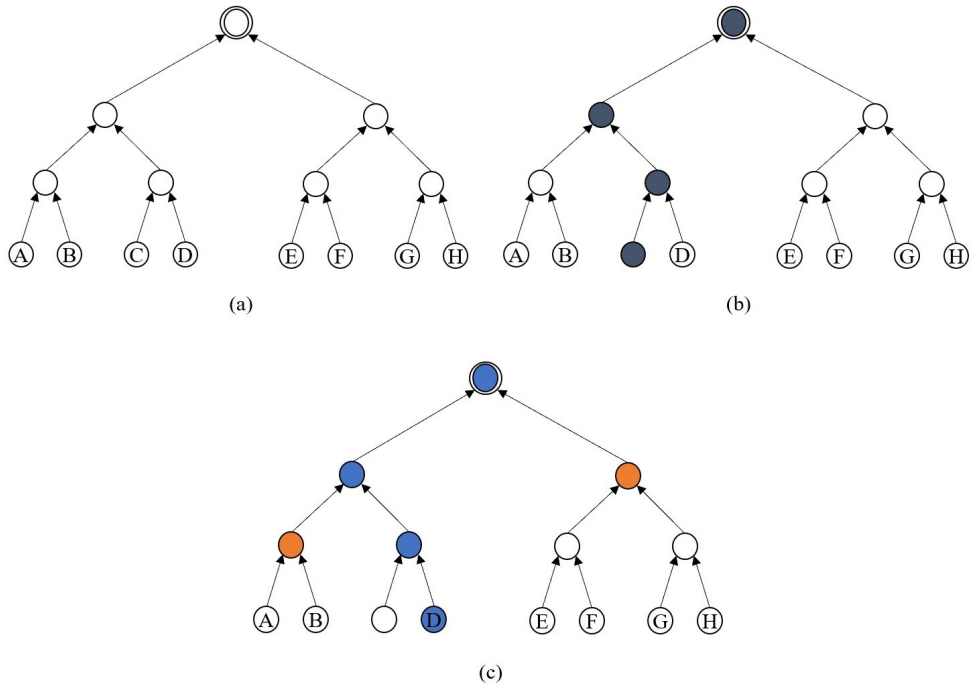


Figure 2.12. The state of a hypothetical group prior to removal of a member is shown in (a). When member D wants to remove member C, it first blanks C's leaf node and all nodes in C's direct path. The tree at this stage is shown in (b) with blanked nodes shown in gray. In (c), D finalizes the update by sending the required messages and updates all keys in its direct path as shown in blue. D then sends all nodes in its co-path (shown in orange), the secret keys of their direct parents. Adapted from [40].

update secrets. A solution to mitigate this attack is to use updatable public-key encryption. The authors assert that, by using this mitigation, TreeKEM can achieve optimal forward secrecy.

The authors of [38] analyzed the security of Draft 7 of MLS. They found several attacks against the protocol and proposed mitigations for the associated vulnerabilities. Since MLS uses TreeKEM with blanking, MLS is susceptible to the double join attack, but this vulnerability can be avoided by adding authentication via signatures for the nodes in the tree. They proposed an approach that extends the TreeKEM$_B$ protocol described in Section 2.3.2, called signed trees for TreeKEM$_B$, or TreeKEM$_{B+S}$. A second attack that was identified was

a cross-group forwarding attack. If users *A* and *B* are in two groups **G** and **G'** together and *A* sends a message to group **G**, *B* must know that it was intended for group **G** and not **G'**. If there is no indication of which group the message is intended for, *B* can take the message with the signature of *A* and forward it to group **G'**. Every member of **G'** will believe the message came from *A*. The authors suggest mitigating this vulnerability by adding a transcript hash to all signatures.

The first work to look at the healing behavior when a member belongs to multiple groups was [37]. Pairwise groups are able to fully achieve PCS and heal when members are part of multiple groups since groups do not exist independently (i.e., overlapping groups share associated pairwise channels). If a member is compromised in one group, an update will heal the pairwise channels between individual group members, regardless of which groups those members are in and including those that are shared with the other group. In ART, TreeKEM and MLS, however, groups are independent so if a member heals in one group, they can remain compromised in another group due to the limited scope of the update operation. The authors outlined a solution to heal MLS in cross group scenarios. This type of integration decision has been left to the Authentication Service (AS) in the MLS specification, and is therefore not enforced in the core protocol.

2.5 Protocol Comparison

Table 2.1 compares the efficiencies of group messaging protocols for sending and receiving messages. Not surprisingly, many tree-based group protocol operations have logarithmic complexity while their pairwise counterparts have linear or quadratic complexity. Sender keys requires linear complexity group creation and has no mechanism to remove members or update keys. An appealing feature of TreeKEM is the reduction of processing time for message receivers to $O(1)$ public key operations for updates, adds, and removes [38].

Since the UAV swarm is such a computational- and communications-limited entity, a proposal for a group key messaging scheme must take these limitations into consideration. UAV swarms must be able to dynamically add and remove members from the swarm, and MLS provides a lightweight way to scale secure group communications with logarithmic add, remove, and update operations.

Table 2.1. Secure Group Communication Protocol Efficiencies. Adapted from [38].

Protocol	Create		Add			Remove		Update	
	Send	Receive	Send	Receive	New	Send	Receive	Send	Receive
Sender Keys	N^2	N	1	1	N	–	–	–	–
ART	N	$log(N)$	$log(N)$	$log(N)$	$log(N)$	–	–	$log(N)$	$log(N)$
TreeKEM	N	$log(N)$	$log(N)$	1	1	$log(N)$	1	$log(N)$	1
*MLS	N	1	1	1	1	$^{**}log(N)...N$	1	$^{**}log(N)...N$	1

*MLS as of Draft 7 uses the TreeKEM$_B$ protocol
$^{**}log(N)...N$ indicates that the computational cost ranges from $log(N)$ to N, depending on the number of blanked nodes. In the worst case (i.e., all nodes in the group are blanked), this cost is N. In the best case (i.e., no nodes are blanked), this cost is $log(N)$.

Having secure properties like PCS and forward secrecy is also very important in UAV swarm applications. UAVs can be easily lost or compromised. The protocol needs to ensure that data that was previously sent to or from a compromised UAVs remains secure and that the rest of the swarm can heal communications even if a swarm member is lost and must be evicted from the group. The MLS protocol is well suited for this application because the protocol provides relatively efficient ways to remove a member and update the keys such that the ejected member can no longer communicate with the group [10]. These key updates will help to keep the data secure going forward. The goal of this thesis was to successfully implement MLS in the swarm and evaluate the effect utilizing MLS has on the UAV swarm. MLS has not been implemented in UAV swarm before, and assessing the application of the MLS protocol to UAV swarms is an important advancement in secure group communications for UASs.

2.6 Chapter Summary

This chapter provided background information on UAV swarms and secure group communications. An overview of the ARSENL swarm was given and other UAV swarm communication schemes were discussed. Secure group communication protocols such as Sender Keys, ART, and TreeKEM were covered. This chapter also discussed the motivation of MLS and provided an overview of the protocol itself. Chapter 3 will describe the process of integrating MLS C++ code into the ARSENL swarm.

CHAPTER 3:
Integrating MLS into the ARSENL System

This chapter outlines the process of integrating MLS into the ARSENL codebase using the C++ API developed by Cisco. An overview of ROS, its utilization in the ARSENL system, and its use in the MLS integration is also provided.

3.1 MLS C++ Code

The development of the MLS protocol is an ongoing effort by the IETF working group. All research conducted in this thesis uses Draft 11 of the MLS protocol. Documentation can be download from the IETF tracker website [10]. The IETF provides information about available MLS Draft 11 implementations at the their Github repository [42]. For this research the Cisco-developed C++ API was used. It can be downloaded from the Cisco Github repository [12]. The Cisco C++ API implementation will be referred to as MLS++ for the remainder of this thesis.

The MLS++ API was installed alongside the ARSENL codebase. The MLS++ header files were imported to the relevant ARSENL code, and the static libraries were automatically linked at compile time by the ROS *catkin_make* build tool. The MLS++ libraries required some reconfiguration to facilitate their use on the Linux platforms used by the ARSENL system. For the time being, this process will need to be repeated for any newer MLS++ versions. In particular, the MLS++ dependencies on Ubuntu 20.04 and CMake version 3.12 were downgraded to Ubuntu 18.04 and CMake version 3.10, and links to the MLS++ library files and include directories were manually added to the ARSENL ROS directory hierarchy. No modifications to the original MLS++ source code, functionality, or dependencies were required to facilitate utilization with the ARSENL system. A detailed description of the process used to incorporate the MLS++ library into the ARSENL on-vehicle software is provided in [44].

Three important MLS++ classes were needed for this implementation: **Client, Session,** and **PendingJoin**. Every MLS group member needs to be initialized with a **Client** class instance which is then used to create or join an MLS group. It contains information about

the ciphersuite and public and private keys for the group member.

After joining a group, a member's state in the group is stored in an instance of the **Session** class. The **Session** class contains all of the code that manages a member's participation in the group as updates, adds, removes, and **Commit**s are executed. State information maintained by the **Session** instance includes various keys and information about other group members.

When a member wants to be added to the group, they initialize a **PendingJoin** class instance. This object generates key packages and executes required operations like processing the **Welcome** message.

3.2 Robot Operating System (ROS)

This section introduces the ROS middleware framework and discusses its use in the integration of the MLS protocol into the ARSENL UAS.

3.2.1 ROS Overview

ROS [45] is an open source framework supporting the development and implementation of robust robotic applications. It consists of a set of software libraries and tools that can easily be integrated into other frameworks [46]. ROS is designed primarily for Unix-based platforms and provides Python, C++, and Lisp APIs. The ARSENL UAS is implemented primarily in Python; however, the C++ API [47] was used for this research to facilitate integration of the MLS++ functionality.

ROS Processes

A ROS Master process manages all ROS functionality at runtime [48]. The ROS Master maintains system-wide state information. Every ROS process registers with the ROS Master process and provides required information at startup. The ROS Master process then manages all system-wide state maintenance, event and process scheduling, and inter-process communication and interaction.

ROS functionality for a specific application is achieved through the development of one or more ROS nodes. A node is a single process running within the framework that typically performs a single function [49]. One node might, for instance, implement a driver for a

particular sensor while another node might use the provided sensor data for path planning. Nodes register with the ROS Master at startup, and each node assigns itself a unique name that is provided to the ROS Master upon registration.

While the ROS Master provides runtime management of the ROS system, the internal functionality of each node is isolated from the ROS Master and from other nodes. This arrangement facilitates rapid development by allowing compartmentalization. Specific functionality for each node can be developed in isolation without affecting any functionality of other nodes. Similarly, new sensors or capabilities can be easily incorporated by adding new nodes to the system. This arrangement also facilitates the use of different programming languages. For this research, the MLS node was written in C++, but the rest of the ARSENL system is written in Python [11].

ROS Communications

ROS implements a publisher-subscriber model that is used by nodes to communicate with one another. Nodes send information to other nodes by publishing messages to topics to which other nodes can subscribe. A sensor managing node, for instance, will publish the sensor data to a topic to which nodes requiring the data subscribe. ROS Messages are typed data structures that can be used to encode atomic data elements (e.g., integers, floating point numbers, or strings) or composite data elements (i.e., structs) [50].

ROS topics are specified using text strings and act as named buses to which nodes publish and subscribe [51]. Nodes inform the ROS Master of their intent to publish to a topic or their desire to subscribe to a topic. The ROS Master then receives messages from publishers and forwards them to the topic's subscribers. Arbitrary nodes can publish or subscribe to a topic, and other nodes do not know the origin or destination of the topic's messages. In this sense, communications are anonymous. Any node can send messages by publishing to a particular topic or receive messages by subscribing to a topic. Thus, this model provides for many-to-many messaging in that multiple nodes can publish to a topic, and multiple nodes can subscribe to a topic. An example of ROS nodes and topics in use is depicted in Figure 3.1.

Each topic is assigned a message type when it is initialized. When publishing or subscribing to the topic, messages must match the designated message type for that topic. ROS utilizes

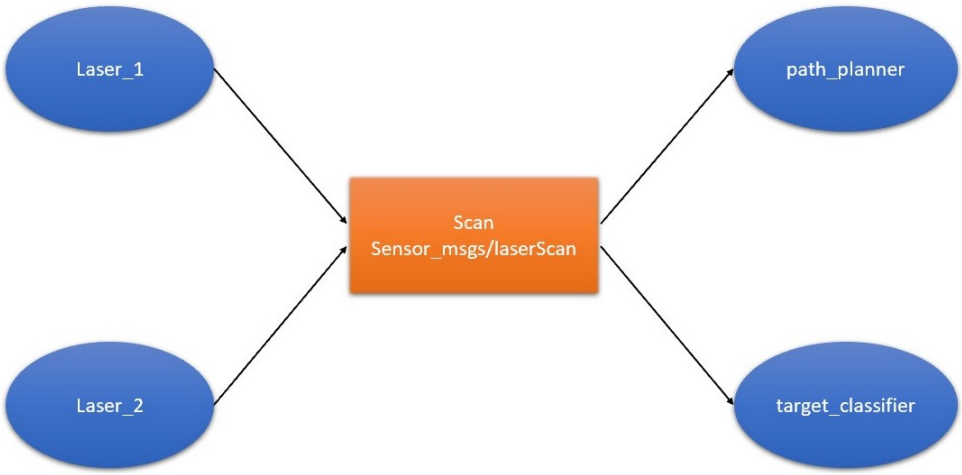


Figure 3.1. This figure shows ROS nodes represented as blue ovals, and topics represented as orange squares. A line pointing to a node indicates that node is subscribed to a topic. A line pointing from a node indicates the node publishes to a topic. In this figure, *Laser_1* and *Laser_2* nodes control separate physical LIDAR sensors. They publish messages containing sensor readings to the topic *Scan* in the form of $sensor\_msgs/laserScan$. Two nodes, *path_planner* and *target_classifier*, are subscribed to this topic. They use the data for two different purposes. Subscribing nodes are unaware of which nodes initially published the data and which other nodes are subscribed to the topic.

the underlying network implementation to send messages using either Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). Only UDP is supported by the *roscpp* library with which MLS functionality was implemented [47].

ROS also provides a service-oriented communications capability that functions like a remote procedure call [52]. ROS service messages contain both request and reply sections. The request is sent by the calling node to the node providing the service, and the providing node responds with the reply. Although ROS services are used within the ARSENL system, they were not required for the MLS implementation.

3.2.2 MLS as a Node

ARSENL on-vehicle functionality is implemented as a set of ROS nodes: the *network_bridge* is responsible for air-to-air and air-to-ground communications, the *autopilot_bridge* interacts with the PixHawk autopilot to achieve swarm behaviors, the *safety* node provides for safety of flight, the *task_scheduler* initiates preplanned actions as mission milestones are achieved, and the *swarm_manager* controls execution of swarm behaviors [11]. Interactions between nodes rely primarily on publishing to and subscribing from ROS topics. MLS functionality is provided by an *mls* node that was added to the *arsenl_communications* ROS package within the ARSENL on-vehicle codebase [53]. There are four ROS topics associated with MLS (Figure 3.2) to which the node publishes or from which the node subscribes:

1. Subscribe to *mls/init_mls*: The *task_scheduler* publishes to this topic when the UAV transitions to the *flight ready* state to initiate a join operation. The callback will either create the MLS group if it does not yet exist (message contents of 0) or request to join an established group (message contents indicate the UAV to contact).
2. Subscribe to *mls/recv_join_msg*: The *network_bridge* node publishes to this topic when it receives an ARSENL message from another UAVs indicating that the UAV wants to join the group. The callback for this subscription executes the join operation for the requesting UAV.
3. Subscribe to *mls/send_mls_msg*: This topic is used by the *network_bridge* node to send an encrypted ARSENL message. The callback for this topic encrypts the message and sends it over the MLS socket to the other group members.
4. Publish to *mls/recv_mls_msg*: This topic is used to send ARSENL messages decrypted with the MLS group key to the ARSENL code. The *network_bridge* node subscribes to this topic and processes the decrypted messages as required.

UAVs must successfully join the MLS group before they can actively participate in swarm behaviors. These topics and their uses throughout the MLS code are described in the rest of this chapter.

3.3 Integration of the MLS Protocol

This section will discuss how the MLS protocol was integrated into the ARSENL swarm. All code development was done in the C++ language and utilized MLS++ and ROS.

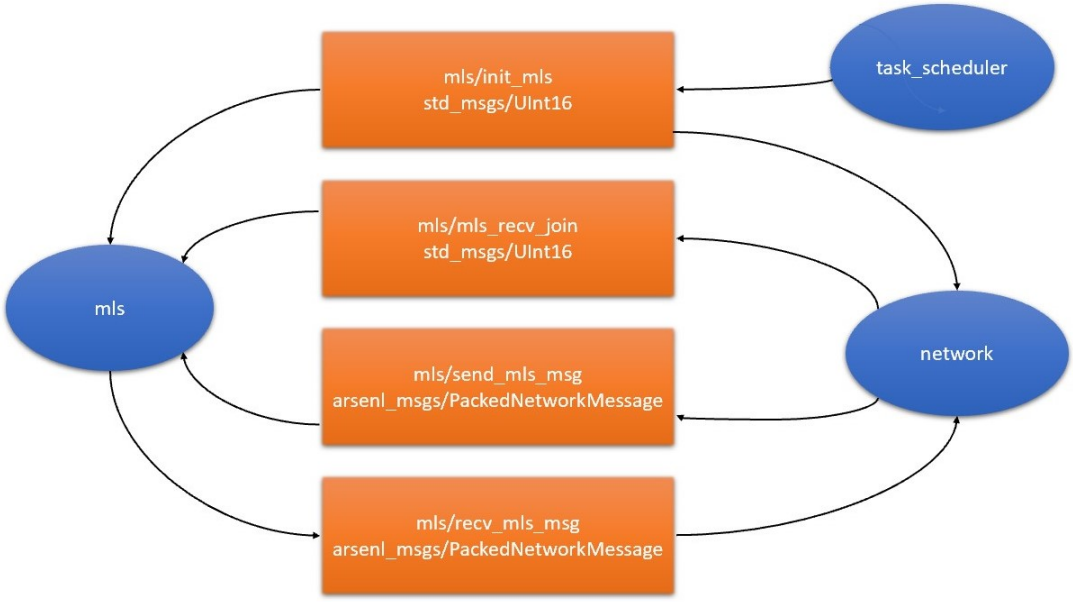


Figure 3.2. Four topics are associated with the *mls* node (nodes are represented in blue and the topics in orange). The *mls* node subscribes to the *mls/init_mls*, *mls/mls_recv_join*, and *mls/send_mls_msg* topics and publishes to the *mls/recv_mls_msg* topic (message types are indicated in the second line of the topics). The ARSENL *task_scheduler* node publishes to the *mls/init_mls topic* to initiate group creation or joining. The *network* node publishes to the *mls/mls_recv_join* topic when another UAV wants to join the group and to the *mls/send_mls_msg* when an ARSENL message is to be encrypted and sent. The *network* node subscribes to the *mls/init_mls* topic to initiate a join request to another UAV and to the *mls/recv_mls_msg* topic to receive decrypted ARSENL messages.

3.3.1 Code Development

The code was developed over the course of several months in iterative stages. Testing of swarm performance was not possible until the full implementation of sending and receiving encrypted messages was completed. The SITL simulation environment described in Section 2.1.2 was used extensively in the development and testing the *mls* node prior to experimentation with ARSENL UAVs.

One class was created for this implementation to handle all MLS group operations. The functionality contained in this class is described over the remainder of this section. Node

functionality was developed in stages to sequentially account for all major aspects of the MLS protocol:

1. Network Connection: Code developed in this stage creates a broadcast socket to send MLS related messages to other UAVs on the network and receive messages sent by other UAVs.
2. Group Creation: In this stage, code was developed to create or join a MLS group. The group is created if no group is currently established (i.e., no UAV has transitioned to the *flight ready* state). A join is initiated if the MLS group has already been established.
3. Sending and Receiving messages: Coding during this development stage provided encrypt-and-send and receive-and-decrypt functionality. ARSENL messages published to the *mls/send_mls_msg* topic by the *network_bridge* node are encrypted then broadcast over the MLS socket. Upon receipt, a message is decrypted by the receiving UAV and published to the *mls/recv_mls_msg* topic for handling by the ARSENL nodes.
4. Update: Coding in this stage provided the MLS update capability. Upon initiation of the update, each UAV updates the keys in its direct path. All other UAVs in the group are also made aware of these changes so that messages can be encrypted and decrypted with the new keys.
5. UAV Removal: In this final development stage, the member removal operation was implemented. Once removed, a UAV is no longer able to decrypt encrypted messages sent to them.

For much of the development, debugging could not be done with a debugger because of the highly parallelized nature of the ARSENL system. Successful development, therefore, was heavily reliant on analysis of ROS log files and messages printed to the vehicle-specific terminal windows while the simulation system was running. All debugging was done by hand, and log files were analyzed for potential errors. Wireshark was used to analyze and debug data sent across the network.

3.3.2 Code Overview

The **MLS** class manages all group operations related to MLS and is the primary class with which the *mls* node is implemented. Important information about each UAV is provided in

the form of ROS parameters set by the ARSENL code: UAV ID, network device name, and port. The UAV ID is a unique integral value associated with the vehicle that serves as its identification in the MLS group. The network device name is used to obtain the IP address for the broadcast socket. The port is used to calculate the UDP port to be used by the MLS socket for inter-UAV communications. A brief overview of the code is provided here.

The first UAV to reach the *flight ready* state initializes the MLS group. Subsequent UAVs reaching this state request to join the already established group. Both establish and join events are triggered by the *task scheduler* node by publication of a ROS message to the *mls/init_mls* topic. An integer 0 in the message data field indicates that the UAV is the first reach the *flight ready* state. Any other value indicates that the UAV is not the first. In this case the message data field value indicates the UAV that will execute the join operation. The new UAV will wait to be added in this case.

When a UAV needs to be added to an existing MLS group, the *network_bridge* node on the new UAV (UAV_{new}) sends an unencrypted ARSENL message to an established member of the group ($UAV_{established}$) that is to perform the join operation. The *network_bridge* node on $UAV_{established}$ publishes a ROS message to the *mls/recv_join_mls_msg* topic indicating that UAV_{new} wishes to join the group. The **MLS** class instance of the *mls* node on $UAV_{established}$ then initiates the process to add UAV_{new} to the group. Authentication of UAV_{new} prior to executing the join is beyond the scope of the MLS specifications and of this integration as it is part of the AS. This failure to ensure that only authenticated UAVs induces a gap in security, but resolution is left to future research. More discussion on authentication is provided in Section 5.2.5.

Addition of a member requires a **KeyPackage** for the new member. In the example, when $UAV_{established}$ receives this **KeyPackage** from UAV_{new}, it adds the member and broadcasts the **Add** proposal and **Commit** messages to all other UAVs. Every group UAV processes the messages and adds the new member to their view of the group. $UAV_{established}$ then sends the **Welcome** message to UAV_{new}, and UAV_{new} processes the **Welcome** message to finalize its addition to the group.

All members of the group can encrypt and send messages at any time to other group members and can decrypt received messages. When the *network_bridge* node needs to send an ARSENL message, it publishes an unencyrpted copy to the *mls/send_mls_msg*

topic. The message-handling callback in the **MLS** class encrypts the message and uses the network socket to broadcast it to the group. Received encrypted messages are processed by the *mls* node's socket object. The **MLS** object to which it belongs decrypts the message and publishes the plaintext to the ROS *mls/recv_mls_msg* topic. The *network_bridge* node processes the plaintext message for use by the ARSENL nodes.

To initiate an update, the *mls* node on the initiating UAV uses the socket to send an **Update** proposal and a **Commit** messages to all other UAVs. Receiving UAVs process those messages so that each UAV has the same view of the group tree.

The process for removal of an UAV from the is the same as for an update: the UAV initiating the removal sends proposal and commitment messages that are processed by other group members. After removal, the removed UAV can no longer decrypt messages received over the socket.

Network Creation

The first stage of development focused on code for creating the MLS network. Creating the network socket itself was the biggest hurdle in this code development stage. Wireshark was used heavily in error checking as well as was manual checking of log files.

A UDP broadcast socket is used for this implementation, and messages are sent to the broadcast address. Since all UAVs are listening on the network, they all potentially receive every message that is sent. Since much of the ARSENL code was written to rely on UDP communications, it made sense to continue with this pattern. There was no checking that messages were received, however, and UDP provides no such guarantees. A future enhancement to the code is warranted to monitor the distributed MLS service to ensure rebroadcast of dropped packets. This service is left to future work as discussed in Section 5.2.

A dedicated thread is implemented to read from the MLS socket using a blocking read to account for the asynchronous communications. This allows both MLS maintenance and ARSENL messages to be read and processed as they arrive. The alternative would be an inefficient polling loop. Writing to the socket occurs primarily within ROS topic callback functions. The callback for *mls/send_mls_msg*, for instance, will write the encrypted message to the socket. Callback functions are invoked from within a specific ROS message-

handling thread in response to messages received from subscribed topics.

MLS-specific data, such as encrypted messages, **KeyPackage**s, **Commit** messages, and **Welcome** messages are stored in a data type called **Bytes**. This data type is defined and managed in the MLS++ API, and its contents are represented as a vector of unsigned chars (i.e., bytes). Because data is written to and read from a socket as an array of unsigned chars, serialization of the **Bytes** data is required to send data and deserialization is required to convert read data back to a **Bytes** object. Many different ideas were tested, such as type casting the objects to various types to achieve serialization and deserialization functionality. The final functional serialization version loops through the **Bytes** data object and adds each char to an array. The reverse of this process is used to deserialize the data.

Special attention was paid to making sure the data flowed across the socket in a manner that enabled it to be reassembled in a meaningful way. When the data is serialized, information relating to the message type is lost. The MLS++ API differentiates between message types through subtyping (i.e., different message types inherit from the **Bytes** class). The subtype is lost, however, when the object is converted to a char array. To account for this, the first char of the serialized array is used as a packet type indicator. Each message type is represented as a unique integer that is checked during deserialization to determine what type of **Byte** object to use and how to process it.

After integrating the MLS++ code, Wireshark and the SITL simulation were not showing the expected results. Specifically, it was noted that the socket was implemented in a way that allowed a UAV to join a group, but not to send and receive messages after that. The first issue that was investigated was an "address already in use" binding error. This was resolved by setting the socket option *SO_REUSEPORT*, but this did not fix the larger problem. Further testing indicated that the reads and writes were relying on the same **sockaddr_in** structure (a **sockaddr_in** is used by C++ to represent an IP-port pair). When using the same **sockaddr_in** instance for reads and writes, the socket appeared to only read messages that were sent from its own IP address. To fix the issue the socket needed to be configured to read from any source address. A second **sockaddr_in** for which the *INADDR_ANY* option was set allowed the read socket to bind to the same port as the write socket and enabled it to read from any IP address. The socket used for reading messages is not bound to a specific IP address, and all messages are sent and received correctly as a result.

One socket option that was considered but ultimately not implemented, was to use multiple sockets for sending different types of information. One socket would be reserved for MLS maintenance messages like the **KeyPackage**, **Welcome** message, proposal messages (**Add**, **Update** and **Remove**), and **Commit**s. A second socket would be used for encrypted messages between UAV. Since the packet type is easily determined from the first char in the serialized message, this option was deemed unnecessary, and a single socket was used for all types of *mls* node communication.

Group Creation and Member Addition

The initial approach to creating and adding members to the MLS group relied on UAV ID numbering starting with "1" for the first vehicle to start and increasing sequentially from there. When this is true, it is fairly straightforward for the first UAV to initialize the group on startup and for each subsequent UAV to prompt the UAV with the ID number immediately below theirs to add them to the group. For example, UAV 1 would initialize the group, UAV 2 would request that UAV 1 add them to the group, UAV 3 would request that UAV 2 add them to the group, and so on. This made code development easier for testing purposes, and this solution works in the simulation environment. However, this will not work in live flight since UAVs IDs are permanently assigned to specific platforms as opposed to being determined dynamically at startup. Thus, ID assignments for live-fly events are effectively arbitrary, and hard-coding the vehicle ID for the add request is unrealistic.

To solve this issue, the first UAV to transition to the *flight ready* state, regardless of ID, initiates the group. As subsequent UAVs transition to the *flight ready* state, they identify another *flight ready* UAV and request a join from that UAV. Extra ROS topics were added to the ARSENL system to support this approach, and the *mls* node uses these to dynamically add members. This approach is suitable for the SITL simulation system and also for live flight testing. Algorithm 1 shows the process for creating a group and joining a group.

The join process is depicted in Figure 3.3. When a UAV requests to join a group, it sends its **KeyPackage** to a UAV already in the group (i.e., the selected *flight ready* UAV). That member adds the new UAV to the group and generates a **Welcome** message. The **Welcome** message contains information regarding the current state of the group and any public and private keys the new UAV needs. Since the *mls* node utilizes a broadcast socket, all messages are sent to all other aircraft on the network. A separate topic is therefore required to specify

Algorithm 1 Group Creation and Member Addition

$UAV \leftarrow client$
$create\_socket$
upon event $flight\ ready$ **do**
  if UAV is first to $flight\ ready$ **then**
    $start\_group(client)$
  else
    $UAV_{join} \leftarrow$ any $flight\ ready$ UAV
    $join\_group(UAV_{join})$
  end if
while $running$ **do**
  send/receive messages
end while

the intended recipient of the join operation messages.

The *task_scheduler* node publishes the ID of the UAV to which the request will be sent to the *mls/send_join_msg* ROS topic which is subscribed to by both the *mls* and *network_bridge* nodes. The *mls* node callback uses the message contents to prepare the **KeyPackage**. The *network_bridge* node sends an unencrypted ARSENL message to notify the intended recipient. The *network_bridge* node on the receiving UAV publishes the contents to the *mls/recv_join_msg* triggering the *mls* node callback function. In this function, a handshake is sent indicating that the UAV is ready to receive a **KeyPackage** and add the UAV. Upon receipt of handshake message, the joining UAV sends the **KeyPackage** and waits for a **Welcome** message. Without the handshake, timing between aircraft sending and receiving messages was not ensured, and it was possible for the UAV adding the new UAV to miss the **KeyPackage**. The UAV adding the new UAV adds the **KeyPackage**, **Commit**s the change, and sends the resulting **Welcome** package to the added UAV.

Since communication buffers at the socket, there was a potential issue with a UAV using a **Welcome** message that was not intended for their UAV. To mitigate this, an ID was added to the handshake to indicate the aircraft for which the handshake was intended. This indicator lets the UAV know when it can proceed with joining the group by processing the next **Welcome** packet they receive. Another potential option to solve this problem would be to maintain a queue of **KeyPackage**s and wait until the *mls/recv_join_msg* callback is invoked before processing one. The difficulty with this solution would be making sure the correct

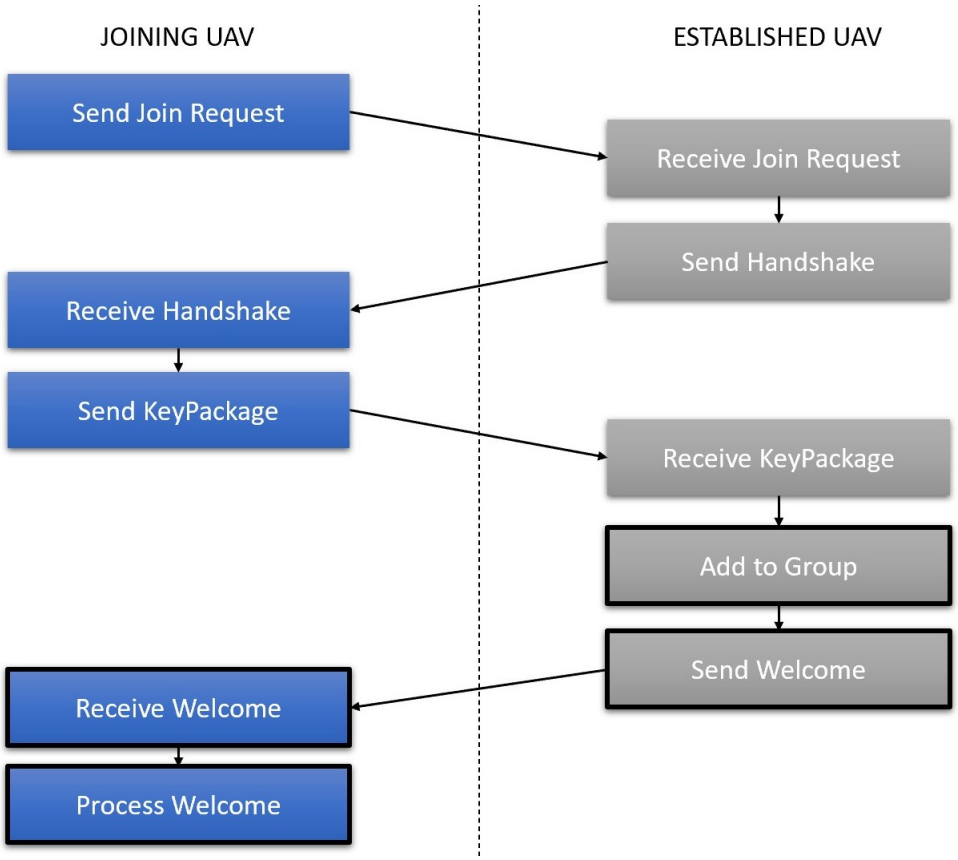


Figure 3.3. When a UAV wants to join a group, the ARSENL system sends a request to another UAV in the group. After the established UAV receives the request, it sends a handshake message indicating that it is ready to receive the joining UAV's **KeyPackage**. When the joining UAV receives the handshake, it verifies the handshake by checking the ID. If it matches its own ID, the joining UAV sends its **KeyPackage**. When the established UAV receives the **KeyPackage**, it adds the new UAV to the group, and sends the **Welcome** message to the joining UAV. The joining UAV processes the **Welcome** and can subsequently participate in group communications. The steps completed with functions from the MLS++ API are outlined in black, while others that are developed as part of this thesis work are not outlined.

key package was used from the queue. This would require that the UAV adding the new member conduct an exhaustive search of identity keys to verify the **KeyPackage**. There would also be a potential problem with space based on the number of **KeyPackage**s that would need to be stored given that there is no limit to the number of other aircraft a UAV

might be asked to add to the group.

The join and add processes are described in Algorithms 2 and 3.

Algorithm 2 Joining UAV Process

```
upon event mls/init_mls do
    joined ← false
    do
        upon event handshake do
            if handshake.id == own_id then
                send key_package
                upon event welcome do
                    process welcome
                    done ← true
            end if
    until joined
```

Algorithm 3 Adding UAV Process

```
upon event mls/recv_join_msg do
    send handshake, id
    upon event receive key_package do
        update tree and commit
        send welcome
```

Testing for correct join functionality in the SITL environment was difficult because of how the SITL simulation and ROS implementation interact with the MLS++ API. Processes outside the *mls* node continue running even if the *mls* node fails and there is little information provided regarding why and where in the code the failure occurred. Failure diagnosis required manual code inspection and extensive analysis of runtime print statements. One such error was a segmentation fault resulting from accessing invalid pointers. This error went unreported and led to downstream errors. Through manual inspection of the code, it was determined that the cause was accessing a temporary **PendingJoin** object that was returned from one of the functions in the MLS++ API.

The **MLS** class ongoing access to the **Client**, **Session** and **PendingJoin** objects in the form of member variables. Storing these objects properly in the **MLS** class was tricky since API developers made the design decision to not provide copy constructors or no-parameter constructors for MLS++ classes. To get around this, C++ vectors are used to store the objects.

When an object is returned from a function in the MLS++ API, it is added to the vector and accessed using the zero index. For example, $UAV\_join[0].key\_package()$ would be used to access the **KeyPackage** for the UAV requesting to be added to the group. Only one object is ever added to the vector during the lifetime of the UAV, so the index never changes. This solution worked well and did not appear to affect access times to class functions. It proved to be a long-term fix, and no further issues during the integration appeared because of it.

Commits

Commits are periodically initiated by individual UAVs using a 2-step proposal-commit process. After sending an **Add**, **Remove**, or **Update** proposal to other UAVs, the UAV initiating the action **Commit**s the proposal. The MLS++ code returns a **Commit** message that is then forwarded to the rest of the group. Group members process the received proposal and **Commit** to bring their local views up to date. Ensuring that every UAV has the same view of the group requires **Commit**s to be processed in the same order by all group members.

For the **Commit** operation, occasional errors were noted as UAVs process **Commit**s at different rates. For instance, if a **Commit** is processed and a message is received from another UAV that has not yet processed the **Commit**, an error will occur. This is not an issue with MLS implementations that assume that packet ordering is deconflicted (e.g., by the distribution service per the MLS protocol specification [10]). The distributed, peer-to-peer nature of the UAV swarm, however, makes deconfliction difficult. Resolution of this issue has been identified as an area for future work as described in Section 5.2

Another known issue is that UAV messages are broadcast, and there is currently no filtering in the *mls* node for self-sent messages. This will cause an error as well since **Commit**s cannot be processed twice. This issue is difficult to address in the SITL environment because UAVs share a single IP address. Future work can mitigate this problem on physical vehicles by filtering and ignoring self-sent messages based on the source IP address.

Message Sending and Receipt

As implemented, the *mls* node only encrypts some ARSENL network traffic. Information exchanged between UAVs and the ground station that is required to ensure safety of flight and maintain operator oversight is still sent as plaintext. The information that is sent between

UAVs, however, is encrypted using *mls*. In particular, state/telemetry messages transmitted by each UAV at a rate of eight hertz and autopilot status messages transmitted at one hertz are encrypted. Since these messages comprise the bulk of the ARSENL network traffic, performance observations are considered representative of the overall system.

The *mls* node subscribes to the *mls/send_mls_msg* topic. When the ARSENL *network_bridge* node has a message that needs to be sent to other UAVs, the *mls* node callback function for this topic is invoked. The *mls* node encrypts the message and broadcasts the encrypted data over the socket. When a UAV receives an encrypted message, the *mls* node decrypts the message and publishes the plaintext message to the *mls/recv_mls_msg* topic. From there, the *network_bridge* node forwards the plaintext message contents for use by other ARSENL nodes as required. Analysis of log output verified that this process was working correctly (i.e., received and decrypted messages matched the original messages that were encrypted and sent).

Messages are sent frequently and this group operation consumes the most network bandwidth. Message sending and receiving functionality and performance, therefore, were of particular interest during the testing.

Update

The update operation is conducted periodically to ensure that security of the group is maintained. When a UAV wants to initiate an update, it sends an **Update** proposal message to the group, and follows with a **Commit** message. Each UAV that receives the messages processes them to update their view of the keys in the tree.

Update functionality was tested in the SITL environment with various time periods between updates to determine how swarm operations were impacted as update frequency changed. When updates were initiated after every message was encrypted, the induced overhead effectively stifled inter-UAV communications. Since SITL environment performance typically exceeds on-vehicle performance, it is assumed that this degradation would carry over to the actual UAVs. Communication did not return to normal until updates were only initiated once for every 150 messages sent, which is approximately once every fifteen seconds per UAV. Additional testing was conducted determine the best interval between update operations. This testing is discussed in Chapter 4.

Member Removal

The last MLS operation implemented was removal of a member. This operation is required when a member voluntarily leaves the group or when other group members determine that the member has failed, is compromised, or is malfunctioning. The removal process is similar to the update operation in that it is effected by **Remove** proposal and **Commit** messages from one member to the rest of the group. Testing of this function was only conducted with the same UAV (i.e., UAV 2) to verify that the function worked correctly. Multiple test runs in the SITL simulation environment verified that UAV 1 could reliably remove UAV 2 and that once removed, UAV 2 could no longer decrypt messages sent within the group. An obvious potential area of future work is to develop and test approaches to group-wide consensus for determining when to remove a UAV from the group.

3.4 Chapter Summary

This chapter outlined the process that was utilized to integrate MLS++ functionality into the ARSENL swarm to include discussion of difficulties and problems encountered during the implementation process. An overview of ROS was also provided to facilitate better understanding of the relationship between the existing ARSENL swarm system and the MLS implementation. Chapter 4 provides the testing results and discusses observed MLS impact on swarm communications.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Results

This chapter discusses testing results of the *mls* node implementation. Testing was conducted in two stages. The first tests were conducted in the SITL simulation environment and provided baseline metrics and assurances of correct functionality. The second phase consisted of ground tests with various numbers of ARSENL UAVs and was used to assess performance in the actual communications environment. Live vehicle testing was also used to evaluate various update intervals in the lossy communication environment and absence of delivery service implementation.

4.1 Simulation System Testing

There were six variables of interest that were evaluated during this testing phase:

1. time required to join the swarm once in flight ready state,
2. update frequency,
3. bytes of plaintext encrypted and bytes of ciphertext sent per UAV,
4. bytes of ciphertext received and decrypted per UAV,
5. number of messages encrypted and sent, and
6. number of messages received and decrypted.

Statistics were captured and logged every three seconds.

Testing was conducted with various update intervals associated with the number of messages sent by each UAV, meaning an update message was issued by UAV for every x number of messages that the UAV sent. Data was collected for update intervals of x equals 50 messages, 150 messages, and 250 messages. Data was also collected with no updates being processed by the UAVs.

The UAV responsible for starting the group was able to create a group comprised of only itself using the MLS++ API in under five milliseconds. After the group was created other UAVs could be successfully added to the group. The longest a UAV took to join the group after reaching the *flight ready* state was 58 milliseconds. It was of interest to test the time it

takes a UAV to join the group since a non-MLS handshake for sharing the MLS **KeyPackage** which incurred wait time to the join process was introduced. (It should be noted that this method of sharing the **KeyPackage** is one architectural design choice and does not rule out other methods that could be used). The low times recorded indicate very efficient create and join operations for the swarm.

4.1.1 Update Frequency Testing

For the update messages, four different update intervals were tested with swarms of size five and 10 UAVs. Intervals were calculated per device. For example, if the interval between updates is set to 50, then after a single UAV sends 50 messages, that UAV will issue an update. To verify that MLS was implemented correctly and that the ARSENL swarm was properly handling the group protocol, log files were analyzed to ensure the correct number of messages were encrypted and that the correct amount of data was received and decrypted.

No Updates

To get a baseline for how the *mls* node was performing, the SITL simulation was run with no UAVs issuing updates. Each UAV sends nine ARSENL messages to the group per second (eight state messages and one autopilot status message). In a swarm of five UAVs, therefore, 45 messages should be received and decrypted per second per UAV. The SITL test verified these statistics, with each UAV decrypting an average of 44 messages per second per UAV. In a swarm of 10 UAVs, 90 messages should be received and decrypted per UAV per second. The tests showed that a swarm of 10 UAVs decrypted an average of 88 messages per second per UAV. These results were considered reasonable for the SITL environment in which communications are subject to packet loss associated with UDP. These tests indicated that the *mls* node was working properly and that UAVs were able to successfully send encrypted messages to other UAVs and that those messages could be properly decrypted by receiving UAVs using the group key.

On average an ARSENL UAV encrypts approximately 495 bytes of plaintext data and sends approximately 1,749 bytes of ciphertext per second. Based on these numbers a swarm of sizes 5 and 10 should receive 8,745 and 17,490 bytes of ciphertext per second respectively and decrypt it to 2,475 and 4,950 bytes of plaintext. The ciphertext is larger than the plaintext since the chosen ciphersuite included both authentication and additional authenticated data,

and the MLS++ API included information about the group id, epoch, and sender within the ciphertext. These numbers closely aligned with the observation in both tests with no updates. Each UAV in a swarm of size five received between 8,651 and 8,752 bytes of ciphertext that decrypted to between 2,454 and 2,472 bytes of plaintext per second. For a swarm of size 10 between 16,835 and 17,513 bytes of ciphertext per second were received equating to between 4,756 and 4,948 bytes of plaintext. These numbers do indicate a small amount of packet loss associated with UDP even with the largely reliable SITL communications.

50 Message Update Interval

To test the update functionality, a 50 message interval was first used. After sending 50 messages, each UAV issued an update and went through the update process described in Section 3.3.2. Updates occurred approximately every five seconds.

When testing this update interval, a number of errors were frequently observed. In the worst case, one UAV failed after communicating with the swarm for a period of time, and all of the log data was lost as a result. Another common error was associated with a UAV using a key that from an incorrect epoch because of a missed update. When this occurred the UAV would attempt to encrypt and decrypt messages using an old group key. On UAVs receiving messages from the affected UAV, an error message presented indicating that the received message was encrypted using the wrong key. When the affected UAV attempts to decrypt messages received from UAVs that successfully performed the update, the MLS++ code presents a similar error indicating that it has no state for the epoch associated with the encrypted data, since it did not receive the update(s).

For a UAV group with five members, an average of 42 messages were received per second per UAV. The test was stopped after observing that two UAVs had stopped communicating with the swarm. If the testing period had continued, this average would have approached 27, since only three UAVs out of the five were communicating in the group. This average was expected to be close to 45 if all five UAVs were processing updates correctly. The size 10 UAV swarm performed much worse, as expected, with an average of 41 messages received per second. The average number of messages sent per second per UAV was much lower than the expected value of 90 for a 10-UAV swarm. This indicates that more UAVs were effectively removing themselves from the communications group due to missed **Commit** messages associated with the update function.

Due to lost log data for one member of the size five swarm, all results are based on the remaining four members of the group. The five UAV swarm received between 8,289 and 8,376 bytes of ciphertext per second per UAV on average which decrypted to between 2,342 and 2,366 bytes of plaintext. A UAV in the 10-member swarm received at most an average of 10,140 bytes of ciphertext per second and as little as 6,882 bytes per second. The resulting data from the MLS++ API functions showed that each UAVs was decrypting, on average, between 1,972 and 2,864 bytes of plaintext. This data indicates that ciphertext for approximately six UAVs was received on the high side, and approximately four UAVs on the low side. These numbers are significantly lower than what should have been received for a swarm of size 10, and are approximately the amount of data that was expected for a swarm that was decrypting data for five UAVs.

The ROS log entries output to the terminal as the simulation system ran, also showed that the issues were associated with the update functions.

All of this data indicates an update interval of 50 messages is much too low for the swarm to handle and function properly. Since the average messages decrypted for a size 10 swarm was drastically worse than expected, this interval between updates was demonstrated to be unsuitable even for this relatively small group and would certainly not scale well for larger ARSENL swarms.

Missed **Commit** messages are a big problem since there is no easy recovery. If reliable delivery MLS maintenance packets can be guaranteed, this update interval should be retested to see if it can be successfully utilized in the ARSENL swarm.

150 Message Update Interval

An interval of 150 messages sent between updates was tested with five- and 10-member swarms. For this interval, each update occurs approximately every 15 seconds. The five-UAV swarm decrypted an average of 42 messages per second per UAV. This is relatively close to the ideal average of 45 messages per UAV per second. For the 10-UAV swarm, an average of 75 messages were decrypted per second per UAV that did not fail. During this test, eight UAVs communicated properly for the entire duration of the test, but two UAVs did drop out from communications. One process failed halfway through the five minute test. When a UAV failed, a zero was appended to the remainder of the data for decrypted

messages to account for the failure. It is difficult to diagnose why the UAV process failed, but output from ROS messages to the terminal indicated that the issue was related to processing a **Commit** message. The other UAV failed to process an update so it was using a key from a past epoch.

Each UAV in the swarm of size five received between 6,446 and 8,267 bytes per second per UAV. This data decrypted to between 1,825 and 2,336 bytes of plaintext per second per UAV. For a swarm of size 10, between 5,892 and 15,134 bytes of ciphertext per second were received, equating to between 1,645 and 4,276 bytes of plaintext. The very large range for this test is accounted for by the two UAVs that dropped out of the communications. Since they did not realize that their key was out of date, these two UAVs were still able to decrypt their self-sent messages for much of the testing period but were unable to decrypt messages sent by other UAVs.

250 Message Update Interval

The last interval that was tested was a 250 message update interval, where each update occurs approximately every 25 seconds. For this test one UAV failed and two stopped communicating with the swarm after close to 10 minutes of running the simulation. The size five UAV swarm decrypted an average of 44 messages per second per UAV. Since each UAV decrypts its own self-sent messages and each UAV encrypts nine messages per second, a size five swarm is expected to decrypt 45 messages per second. This observed data is only one message shy of the expected number of decrypted messages and is also the same number of decrypted packets seen in the test with no updates. These results indicate that for small swarms, this update interval performs similarly to not updating. For the 10-UAV swarm, an average of 87 messages per second per UAV were decrypted. This is only three messages less than the expected number. Overall, this update interval performed reasonably well, but not as well as the no-update test. Regardless, most UAVs were able to communicate with the swarm for a majority of the testing duration.

With a five-UAV swarm, each UAV received between 8,676 and 8,754 bytes of ciphertext per second per UAV. This decrypted to a plaintext range of 2,451 to 2,473 bytes per second per UAV. These are relatively small ranges that are close to the expected values. As each UAV joined the swarm, the number of bytes of ciphertext decrypted per UAV increased proportionately. This test received almost perfect communications for all of the UAVs in the

swarm. With a 10-UAV swarm, between 16,108 and 16,615 bytes of ciphertext per second were received, decrypting to between 4,551 and 4,694 bytes of plaintext. This range was relatively small and accounted for a majority of the data that was expected to be received. As UAVs were added to the swarm, the bytes of data received and decrypted by each UAV scaled proportionally.

This update interval performed well for the most part, and was the best interval that was tested. There was still some packet loss, which is expected with UDP, and the UAVs that dropped from the group communication failed due to unprocessed **Commit**s.

Update Interval Testing Summary

Every update interval test experienced at least one UAV that failed or dropped out of group communications. Empirical evidence (i.e., log entries and ROS messages to the terminal) implies that these failures were caused by unprocessed update packets and other errors associated with missed **Commit** messages. Testing for the update frequency was not exhaustive and should be more thoroughly investigated in future research. A distribution service, as discussed in Section 5.2.2 that guarantees reliable MLS maintenance messages will likely be required before sufficient testing can occur to determine the best update interval.

Results from the SITL environment update interval tests are depicted in Figure 4.1. Most intervals for a swarm of size five were close to the expected value of 45 decrypted messages per second per UAV. Missed updates were evidently not an issue. Not surprisingly, the swarms that had no updates and the swarms with the 250-message update interval outperformed the other intervals tested. For the 10-UAV swarm, the update interval significantly impacted the average number of messages decrypted per second. With an update interval of 50 messages, the 10 UAVs received and decrypted fewer than half of the messages that were sent. The average number of messages decrypted per second got closer to the expected value as the update interval increased, indicating that the swarm performs better with a longer period between updates. The improvement as the interval increased from 50 to 150 messages and from 150 messages to 250 messages implies that even longer intervals might yield results equivalent to those of the no update test. Even so, the occasional unprocessed updates would be problematic in an operational system as those UAVs would not be able to communicate with the group beyond the point of failure. In addition, the decrease in

performance between the five-UAV and 10-UAV tests implies that an interval sufficient for a 10-UAV system is unlikely to scale as size increases. Security concerns with longer update intervals will also need to be addressed. Since the 250 message interval worked reasonably well, only this interval and no update processing was tested on physical aircraft in ground tests.

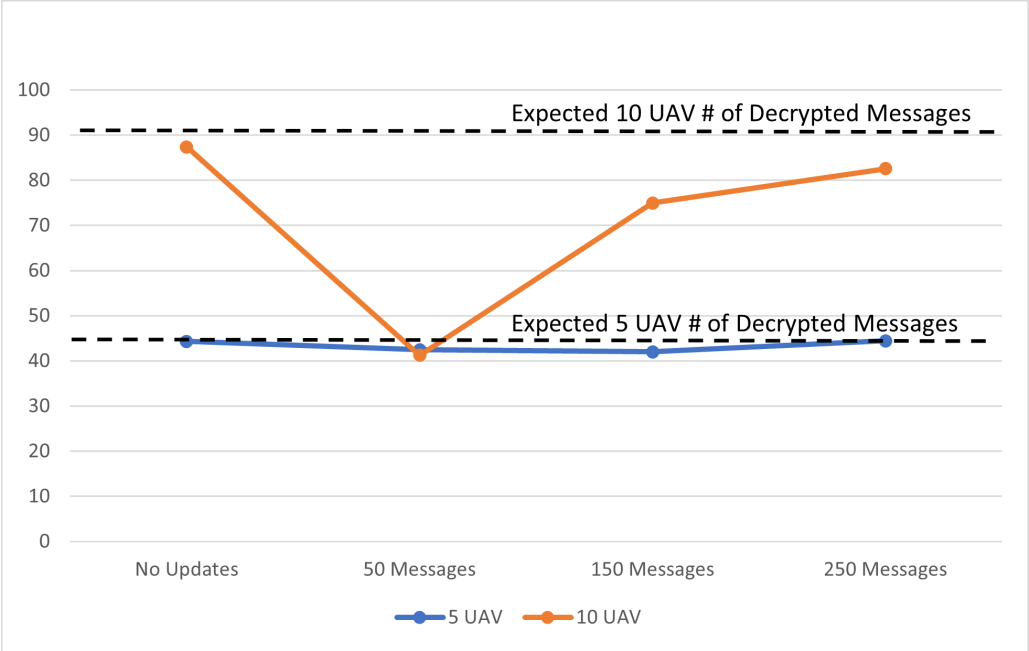


Figure 4.1. This graph depicts the average number of decrypted messages per second per UAV with groups of five and 10 UAVs. The y-axis depicts the average number of decrypted messages per second, and the x-axis depicts the update intervals. The solid blue line depicts results for swarms with five UAVs, and the orange line depicts results for swarms with 10 UAVs. The dashed black lines indicate the expected number of messages for a five- and 10-UAV swarm.

4.2 Ground Testing

Following testing in the SITL environment, ground tests were conducted with physical UAVs to test the functionality of the *mls* node in the actual ARSENL vehicles. All testing was conducting on the Mosquito Hawk quadcoptor described in Section 2.1.2. The Mosquito Hawk utilizes a HardKernel Odroid-C0 companion computer [22]. The C0 is a single-board

computer with an Advanced Reduced Instruction Set Computer Machine (ARM) version 7 quadcore chipset that operates at 1.5 gigahertz. It has one gigabyte of random access memory, and a 32 gigabyte embedded MultiMediaCard (eMMC). Inter-UAV communications takes place over an 802.11n network in the 2.4 gigahertz band.

Update issues identified in the simulation environment were taken into account when testing with the physical UAVs. A main focus with the ground testing was to determine if the *mls* node worked properly on physical aircraft (i.e., that the UAVs could join a group, encrypt messages to and decrypt messages from other members). Data was collected from a log file to which summary data was written every three seconds. Collected data included bytes of data encrypted and decrypted, number of messages encrypted and decrypted, and the time in milliseconds required to join the swarm.

Three errors were noticed during the ground tests:

1. failure to join the MLS group,
2. failure to process **Commit** messages (join or update operation), and
3. failure to write to the log file.

These failures are all related in some way to UDP unreliability, which in turn affects how MLS functions within the swarm. Reliable receipt of MLS maintenance messages is an important requirement that future research will need to address. It is important to note that with the data that was collected, it is impossible to differentiate between failures 1 and 3. Since no bytes were logged to a file for either of these failures, either the UAV did not join the MLS group, or the UAV did join the group but the file was corrupted when the *mls* node failed. Without a terminal on which to monitor the on-aircraft ROS logging in realtime, it is difficult to determine the cause of a file to which no data was written.

No Updates

Ground tests were first conducted with UAV swarms that were not issuing MLS updates. Four different size swarms were tested: three members, five members, seven members, and 10 members. All failure types described above were observed during the ground tests with no updates. When no updates are being processed, UAVs still need to process **Commit** messages associated with joins. Sometimes a UAV misses one of these **Commit**s and cannot

communicate with the rest of the group using the group key as a result.

For a UAV swarm of size three, all UAVs joined the group, and decrypted messages were received from every other UAV in the swarm. Each UAV decrypted an average of 24 messages per second, which is three messages less than the expected number of 27. This test demonstrated that MLS can be used in the swarm communications and that the MLS protocol functioned well in a very small group despite the lossy communications as long as no updates were being issued.

In the swarm with five members, two of the members did drop out of communications with the rest of the group, one process failed, and the other did not process the **Commit** messages associated with UAVs that joined the group after them. They did decrypt messages for a period after initially joining the swarm but were unable to decrypt group messages sent using the new keys after the missed **Commit**. The UAV that failed decrypted messages for a majority of the testing period, and it was close to the end of the test that it stopped encrypting and decrypting messages. The rest of the swarm decrypted an average of 36 messages per second per UAV. This is nine messages less than expected for this size swarm, but was in line with the number of UAVs that remained in the group for a majority of the testing period.

During the test with a seven-UAV swarm, one UAV dropped out of communications, and two UAVs did not have data written to their log files. It is unknown why the UAVs did not write to the log file. It is possible that the UAVs did not successfully join the group, or it is possible that the UAVs failed and the log file was not saved properly. The expected number of messages a swarm of size seven should decrypt per second per UAV is 63 messages. For this test, the UAVs decrypted an average of 43 messages per second per UAV, indicating one of the two UAVs that failed did participate in the group communications to some extent.

For the 10 member swarm, two of the UAVs did not record a log file (failure 1 or 3), and two others dropped out of group communications because of a missed **Commit** (failure 2). For the rest of the swarm, each UAV decrypted approximately 60 messages per second, when the expected number was 90. It is possible, and likely given the 60 average messages per UAV, that one or both of the UAVs that did not record a log file were communicating with the rest of the group prior to their failure.

An interesting behavior observed with this test was that it appeared as if the UAVs were communicating with two different group keys. In this test one UAV was decrypting an average of 13 messages per second while the majority of the UAVs were decrypting an average of 62 messages per second per UAV. This indicates that for a portion of the test, two UAV were communicating using one group key while the rest of the swarm was using a different group key. This is an interesting point of failure in the swarm that is also due to missed **Commit** messages. For this failure to occur, multiple UAVs would have needed to miss the same **Commit** message. When this occurred, the affected UAVs were able to communicate with one another but not with the rest of the swarm.

No Update Summary

Figure 4.2 shows the results of the ground tests conducted with no updates. The blue line in the graph shows that as the swarm size grows, the average number of messages decrypted per second per UAV grows approximately linearly. This is expected behavior from the swarm and is provides for a promising outlook for the future use of MLS for secure swarm communications.

250 Message Update Interval

After testing swarms with no updates, a 250-message interval between updates was introduced, which equates to approximately 25 seconds between updates. This interval performed the best in the simulation tests so it was also tested on physical UAVs. Swarm sizes of three, four, five, seven, and 12 were tested for this interval, but unreliable MLS maintenance message receipt made testing with updates difficult. Communications issues among the UAVs led to multiple ground tests that yielded no data for any UAVs in the group. For the tests of swarm size four and seven, for instance, not a single UAV produced a log file. An empty log file can occur when a process simply fails before any data is written. It can also happen when a UAV fails to successfully join the group. Not joining the group can happen for a couple reasons. First, a UAV can get stuck during the handshake if a packet to or from the adding UAV is lost. In the ARSENL implementation, it is also possible for a UAV to ask a UAV that was never added to the swarm to add them. This cascading effect of UAVs failing to join the group was not accounted for in this implementation but should be addressed through the follow-on research proposed in Section 5.2.2. As with a number of

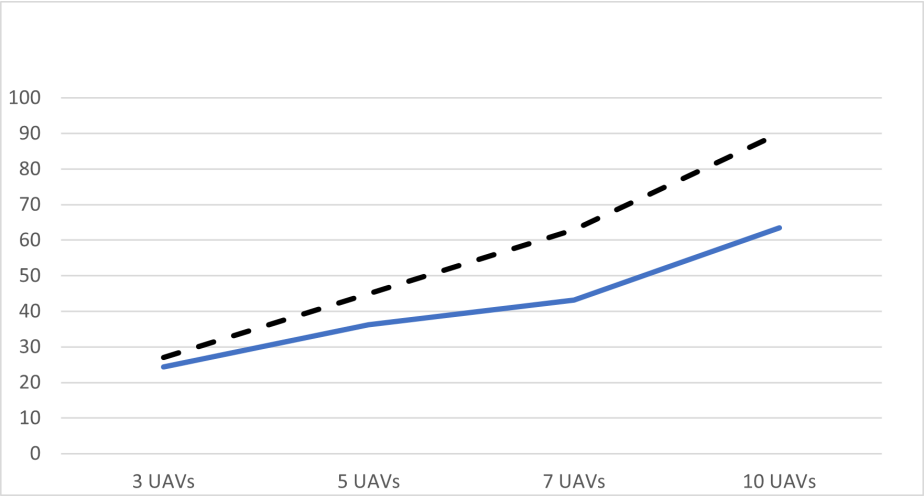


Figure 4.2. This graph shows the average number of messages decrypted per second per UAV during a ground test with no key updates. The y-axis depicts the average number of decrypted messages per second, and the x-axis depicts the different UAV swarm sizes tested. The blue line shows that as the swarm size increased, the average number of messages decrypted per second per UAV also increased. The dashed black line depicts the expected values for the different swarm sizes.

failure modes encountered on the actual UAVs, debugging the was difficult since there was no terminal with which to view the ROS logs as they were generated.

During the ground test with a three member UAV swarm, one of the UAVs dropped out of the group communications. The rest of the group decrypted an average of 19 messages per second per UAV. This indicates that the other two aircraft had synchronized on the same group key to use for the communications.

For the testing with a five-UAV swarm, the UAVs decrypted an average of 11 messages per second per UAV. During this test one of the UAVs did not produce a log file. Every other UAV dropped out of the group communications because of a missed **Commit** message early in the test.

For the test with a size 12 UAV swarm, only two of the 12 UAVs wrote to a log file. Each of these UAVs decrypted an average of 10 messages per second indicating that none of the UAVs were able to communicate successfully with the rest of the group.

On-UAV Update Testing Summary

Figure 4.3 summarizes the results of the 250-message update interval tests. The tests where no UAVs produced log files are omitted. This graph demonstrates the necessity for a distribution service as described in Section 5.2.2. As swarm sizes increased and more update operations were processed, UAVs were more likely to miss processing a **Commit** message, and lose communication with the rest of the group. This graph shows that as the swarm size increased, fewer messages were decrypted on average, indicating that there were few UAVs that were synchronized with a common group key. These results clearly indicate that a reliable delivery service is required for MLS to work with a lossy-communications swarm system such as ARSENL's.

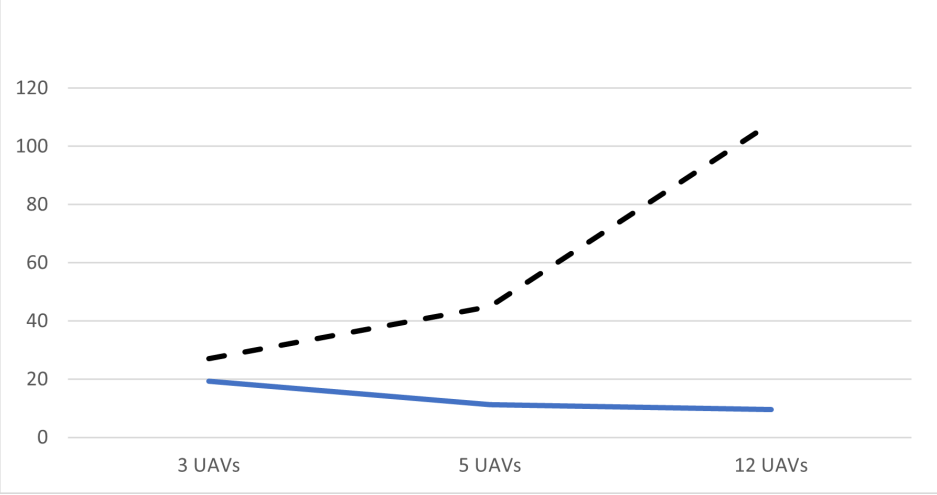


Figure 4.3. The blue line in this graph shows the average number of decrypted messages per second per UAV with a 250 message update interval. The y-axis depicts the average number of decrypted messages per second, and the x-axis depicts the different swarm sizes tested. The dashed black line depicts the expected values for the different swarm sizes. The average number of message per second trend decreases quickly as swarm size increases. This result demonstrates that MLS needs other support before it will function as intended swarms relying on lossy-communications networks.

4.3 MLS Implementation Limitations

Testing indicated that MLS has good potential for secure group communications in a UAV swarm; however, the *mls* node implementation described here is not ready for in-flight

use in its current state. Test results do, however, provide evidence that with further testing and support for reliable packet transmission, MLS can be successfully utilized for swarm communications.

4.3.1 Key Updates

Virtually all shortcomings identified with the ARSENL MLS implementation are associated with updates. Since packet loss is inevitable in UDP-based communications architectures, UAVs can easily get out of synchronization with the rest of the swarm. This loss of key synchronization contributed to some limitations in the testing and verification of the *mls* node in the ARSENL swarm. More information is provided and potential mitigations for this problem are proposed in Section 5.2.2.

The testing conducted here did indicate that a longer periods between updates work better with the swarm architecture. Longer intervals between updates do leave the swarm open to some period of vulnerability if a UAV is compromised. Utilization of a distribution service may make it possible to conduct updates more frequently. Regardless, update interval decisions will often come down to a tradeoff between what limited communications bandwidth will allow (particularly as swarm size increases) and what security the mission requires.

4.3.2 Requirement for Further Testing

Use of the MLS++ API for Draft 11 is limited to environments that can guarantee little to no packet loss, in that it is an implementation of the MLS protocol, and the protocol architecture requires that the local system provides for retransmission in case of packet loss. The simulation environment provided a good testing venue for this implementation, but showed that MLS is very sensitive to lost maintenance packets.

Extensive development and testing needs to be conducted to demonstrate a fully functional MLS implementation for UAV swarms. We did not test UAV swarms larger than 12 members; nor did we conduct tests on other UAVs besides the Mosquito Hawk quadcopter. Testing on additional platforms and with larger swarms is required fully characterize the requirements and performance in various configurations. In particular, more thorough testing should be conducted after mitigations for UDP shortcomings are provided as described in Section 5.2.2.

4.4 Chapter Summary

This chapter covered the testing results for MLS protocol integration into the ARSENL swarm. Testing in the SITL environment provide evidence that the use of MLS for secure UAV swarm communications is possible, but shortcomings were identified in this particular implementation that make it unsuitable for real swarm systems at present. Ground tests indicate that the swarm platforms can support encryption and decryption requirements, but unreliable delivery of MLS maintenance messages made ongoing maintenance of even small MLS groups impossible.

CHAPTER 5:
Conclusion and Future Research

5.1 Conclusion

This work was intended as a proof of concept for using MLS to provide secure communications within a UAV swarm. Achieving this capability is seen as a significant step towards the application of UAV swarm technology to Department of Defense problems. The results of this thesis indicate that MLS can perform well in a swarm, but mitigation for the unreliable communication schemes on which these systems rely is needed for MLS maintenance messages.

5.2 Future Research

This thesis only covered the implementation of MLS in the ARSENL UAV swarm using the Cisco-developed MLS++ API. There are many viable additional areas for future research, both in building on the implementation covered here and in the utilization of MLS with other swarm systems. This section discusses some of these future work options.

5.2.1 Additional Testing

More thorough and comprehensive testing is needed to fully characterize MLS' effect on swarm performance. Testing in support of this research consisted of SITL environment and ground tests on a relatively small number of UAVs over a short period of time with a subset of the ARSENL messages. A more thorough test protocol should include tests over a full flight duration with various numbers of UAVs (up to the maximum that can be expected in a typical swarm mission) and the full ARSENL message set. Longer test periods with larger numbers of vehicles will foster better documentation of the tendency for UAVs to get out of synchronization due to unprocessed or missed commits so that solutions can be proposed and appropriate update intervals can be identified. More robust testing will also further validate MLS' suitability for aerial swarms and asses its scalability as swarm size increases.

5.2.2 Recovery from Dropped MLS Packets

UDP communications architectures on which swarm systems commonly rely are implicitly susceptible to dropped packets. Swarm systems such as ARSENL's are designed with this in mind, so when encrypted ARSENL operational messages are dropped, it does not affect the functionality of the swarm. MLS, on the other hand, was not developed to be lossy-communications tolerant, so missed MLS maintenance messages are not permitted. Delivery failure for MLS messages associated with add, update, or remove operations will lead to UAVs not being able to participate in group communications. That is, if a UAV has not processed all of the commit messages, it will not have the correct view of the group and will not be able to communicate with other group members. There is currently no mechanism in place for easily recovering from missed commits within the code developed for the ARSENL swarm. Some further architecture development (i.e., beyond MLS and the current ARSENL architecture) is needed to provide resiliency in the face of dropped MLS packets. Such a distribution service is required if successful MLS approaches are to be found for these systems. Potential solutions include timeout mechanisms, message receipt verification systems, and protocols to detect and retransmit messages that were not processed.

5.2.3 Error Handling

In the event an error occurs within the MLS++ code, an error message is passed to the invoking code. At present, these error messages are ignored by the MLS code developed for this research. While this was considered acceptable for a "proof of concept" example, a more robust implementation would include a protocol for handling these messages and recovering from the associated errors. In the event the errors indicate that a UAV cannot continue communicating with the rest of the swarm, for example, one possibility would be to have the UAV remove itself from the swarm and land. Alternatively, notification might be provided to the swarm operators so that they can initiate a recovery process that will allow the UAV to rejoin the group.

This sort of error handling is highly dependent on specific use cases. In this thesis, code was developed to use the MLS++ API, but there are other repositories with MLS code in other languages. The development approach discussed in this thesis can be adapted for use with other MLS libraries which may rely on different error protocols and messages.

Error handling protocols that are not a part of the MLS protocol itself, therefore, should be adaptable to approaches beyond those of the MLS++ library.

5.2.4 Reliable Update Operations

As discussed previously, adequate testing to identify the best update interval was not conducted. Future work can look at better evaluating the performance using various intervals with different swarm configurations. In addition, research might include a study of global rather than local interval calculations. For instance, the interval might be based on a time-since-last-update or a system-wide message count rather than the per-vehicle message count discussed in Section 3.3.2.

A potential issue with how updates are implemented in this thesis is that there is a potential race condition if two UAVs issue an update at the same time. This can lead to one update (or possibly both) not being processed correctly and can prevent one or more UAVs from continuing to communicate in the group. One possible mitigation for competing updates is to impose an ordering mechanism so that UAVs are forced to issue updates sequentially. This would require the UAVs to synchronize their updates.

Update operations are also susceptible to failures caused by missed MLS messages. Solutions and research proposed in Section 5.2.2 would, therefore, be relevant here as well.

5.2.5 Authentication Service

The research of this thesis focused on the technical implementation of MLS communications within a swarm system, but no effort was made to authenticate group members. For any real-world implementation, however, the swarm must have a mechanism in place to authenticate UAVs before allowing them into the MLS group. Authentication will mitigate the possibility of an adversary launching a malicious UAV into the swarm or otherwise infiltrating the group. Authentication can be implemented a number of ways, with one possibility being the presentation of a certificate by the joining UAVs that can be validated by the adding UAV before initiating the add process. For added security, an additional factor such as the presentation of an operator-provided token might be incorporated into the authentication process. The authentication method will need to be evaluated based on the requirements and threat models associated with different types of missions.

5.2.6 Protocol to Remove a Compromised UAV

Finally, further research might look into identifying compromised or malfunctioning UAVs so that they can be removed from the swarm. While MLS provides a means of removing a UAV, the question of how to determine when a UAV should be removed is outside of the scope of the protocol itself. A robust removal protocol might be developed to apply criteria for identifying a UAV that has been compromised or should be ejected from the group for other reasons. The ejection process would need to incorporate some form of "consensus" to prevent a single (possibly malicious) UAV from ejecting correctly performing group members. In the event that the swarm collectively recognizes that one of the UAVs is to be ejected, any member of the swarm can initiate a removal operation.

List of References

[1] B. Canis, "Unmanned aircraft systems (UAS): Commercial outlook for a new industry." Available: https://www.semanticscholar.org/paper/Unmanned-Aircraft-Systems-(UAS)%3A-Commercial-Outlook-Canis/84a2a559474a816f6d41ddad77fff8102750cbb2

[2] S. Rosati, K. Krużelecki, G. Heitz, D. Floreano, and B. Rimoldi, "Dynamic routing for flying ad hoc networks," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 3, pp. 1690–1700.

[3] C. Zhang and J. M. Kovacs, "The application of small unmanned aerial systems for precision agriculture: a review," *Precision Agriculture*, vol. 13, no. 6, pp. 693–712. Available: https://doi.org/10.1007/s11119-012-9274-5

[4] I. Colomina and P. Molina, "Unmanned aerial systems for photogrammetry and remote sensing: A review," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 92, pp. 79–97, 2014. Available: https://www.sciencedirect.com/science/article/pii/S0924271614000501

[5] S. Manfreda, M. F. McCabe, P. E. Miller, R. Lucas, V. Pajuelo Madrigal, G. Mallinis, E. Ben Dor, D. Helman, L. Estes, G. Ciraolo *et al.*, "On the use of unmanned aerial systems for environmental monitoring," *Remote sensing*, vol. 10, no. 4, p. 641, 2018.

[6] J. Irizarry and D. B. Costa, "Exploratory study of potential applications of unmanned aerial systems for construction management tasks," *Journal of Management in Engineering*, vol. 32, no. 3. Available: http://ascelibrary.org/doi/10.1061/%28ASCE%29ME.1943-5479.0000422

[7] P. Cohn, A. Green, M. Langstaff, and M. Roller, "Commercial drones are here: The future of unmanned aerial systems." Available: https://www.mckinsey.com/industries/travel-logistics-and-infrastructure/our-insights/commercial-drones-are-here-the-future-of-unmanned-aerial-systems

[8] M. Erdelj, M. Król, and E. Natalizio, "Wireless sensor networks and multi-UAV systems for natural disaster management," *Computer Networks*, vol. 124, pp. 72–86. Available: https://linkinghub.elsevier.com/retrieve/pii/S1389128617302220

[9] F. Xiong, A. Li, H. Wang, and L. Tang, "An SDN-MQTT based communication system for battlefield UAV swarms," *IEEE Communications Magazine*, vol. 57, no. 8, pp. 41–47.

[10] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert, "The Messaging Layer Security (MLS) Protocol," Dec. 2020, work in Progress. Available: https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-11

[11] T. H. Chung, M. R. Clement, M. A. Day, K. D. Jones, D. Davis, and M. Jones, "Live-fly, large-scale field experimentation for large numbers of fixed-wing UAVs," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1255–1262.

[12] *The Messaging Layer Security (MLS) Protocol*, Accessed Aug. 1, 2021 [Online]. Available: https://github.com/mlswg/mls-protocol

[13] M. Campion, P. Ranganathan, and S. Faruque, "A review and future directions of UAV swarm communication architectures," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*.

[14] X. Chen, J. Tang, and S. Lao, "Review of unmanned aerial vehicle swarm communication architectures and routing protocols," *Applied Sciences*, vol. 10, no. 10, p. 3661. Available: https://www.mdpi.com/2076-3417/10/10/3661

[15] M. Hati, "Swarm robotics: A technological advancement for human-swarm interaction in recent era from swarm-intelligence concept," *International Journal of Science and Research (IJSR)*, vol. 5, no. 5, pp. 1165–1168. Available: https://www.ijsr.net/archive/v5i5/NOV163497.pdf

[16] I. Bekmezci, O. K. Sahingoz, and S. Temel, "Flying ad-hoc networks (FANETs): A survey," *Ad Hoc Networks*, vol. 11, no. 3, pp. 1254–1270. Available: https://www.sciencedirect.com/science/article/pii/S1570870512002193

[17] Y. Han, L. Liu, L. Duan, and R. Zhang, "Towards reliable UAV swarm communication in d2d-enhanced cellular networks," *IEEE Transactions on Wireless Communications*, vol. 20, no. 3, pp. 1567–1581.

[18] M. A. Day, M. R. Clement, J. D. Russo, D. Davis, and T. H. Chung, "Multi-UAV software systems and simulation architecture," in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 426–435.

[19] D. T. Davis, T. H. Chung, M. R. Clement, and M. A. Day, "Consensus-based data sharing for large-scale aerial swarm coordination in lossy communications environments," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 3801–3808.

[20] N. Carter, P. Pommer, D. T. Davis, and C. E. Irvine, "Increasing log availability in unmanned vehicle systems," in *National Cyber Summit*. Springer, 2021, pp. 93–109.

[21] K. B. Giles, D. T. Davis, K. D. Jones, and M. J. Jones, "Expanding domains for multi-vehicle unmanned systems," in *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 1400–1409, ISSN: 2575-7296.

[22] "Hardkernel Odroid-C0," Accessed Mar. 7, 2022 [Online]. Available: https://www.hardkernel.com/shop/odroid-c0/

[23] A. I. Hentati, L. Krichen, M. Fourati, and L. C. Fourati, "Simulation tools, environments and frameworks for UAV systems performance analysis," in *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*. IEEE, 2018, pp. 1495–1500.

[24] R. B. Thompson and P. Thulasiraman, "Confidential and authenticated communications in a large fixed-wing UAV swarm," in *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pp. 375–382.

[25] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," num Pages: 160. Available: https://datatracker.ietf.org/doc/rfc8446

[26] "Signal technical information," Accessed Mar. 3, 2022 [Online]. Available: https://signal.org/docs/

[27] "Transport layer security (TLS)," Accessed Feb. 6, 2022 [Online]. Available: https://datatracker.ietf.org/group/tls/about/

[28] "What is transport layer security?" Accessed Feb. 6, 2022 [Online]. Available: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/

[29] J. Alwen, S. Coretti, and Y. Dodis, "The double ratchet: Security notions, proofs, and modularization for the signal protocol," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Springer International Publishing, pp. 129–158.

[30] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," *Journal of Cryptology*, vol. 33, no. 4, pp. 1914–1983, 2020.

[31] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pp. 164–178, ISSN: 2374-8303.

[32] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis, "Security analysis and improvements for the IETF MLS standard for group messaging." Available: http://eprint.iacr.org/2019/1189

[33] B. Hale, “Group messaging security,” Naval Postgraduate School, CS4615 Cryptographic Protocol Design and Attacks, November 2021.

[34] T. Perrin and M. Marlinspike, “The double ratchet algorithm,” Accessed Jan. 20, 2022 [Online]. Available: https://signal.org/docs/specifications/doubleratchet/

[35] C. W. Chuah, E. Dawson, and L. Simpson, “Key derivation function: The SCKDF scheme,” pp. 125–138, series Title: IFIP Advances in Information and Communication Technology. Available: http://link.springer.com/10.1007/978-3-642-39218-4_10

[36] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner, “On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees” (CCS ’18). New York, NY, USA: Association for Computing Machinery, 2018, p. 1802–1819. Available: https://doi.org/10.1145/3243734.3243747

[37] C. Cremers, B. Hale, and K. Kohbrok, “The complexities of healing in secure group messaging: Why cross-group effects matter,” Cryptology ePrint Archive, Report 2019/477, 2019, https://ia.cr/2019/477.

[38] K. Bhargavan, B. Beurdouche, and P. Naldurg, “Formal models and verified protocols for group messaging: Attacks and Proofs for IETF MLS,” Inria Paris, Research Report, Dec. 2019. Available: https://hal.inria.fr/hal-02425229

[39] J. Alwen, M. Capretto, M. Cueto, C. Kamath, K. Klein, I. Markov, G. Pascual-Perez, K. Pietrzak, M. Walter, and M. Yeo, “Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement,” Cryptology ePrint Archive, Report 2019/1489, 2019, https://ia.cr/2019/1489.

[40] C. Brzuska, E. Cornelissen, and K. Kohbrok, “Cryptographic security of the MLS RFC, draft 11,” *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 137, 2021.

[41] K. Bhargavan, R. Barnes, and E. Rescorla, “TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS),” May 2018. Available: https://hal.inria.fr/hal-02425247

[42] *mlswg/mls-implementations*, Accessed Mar. 3, 2022 [Online]. Available: https://github.com/mlswg/mls-implementations

[43] *OpenMLS*, Accessed Mar. 3, 2022 [Online]. Available: https://github.com/openmls/openmls

[44] “MLS++ incorporation to ARSENL codebase,” Available by request: dtdavi1@nps.edu. Available: https://gitlab.nps.edu/arsenl/mlspp/-/blob/nps_dev/ARSENL_INTEGRATION.md

[45] “Documentation - ROS wiki,” Accessed Feb. 8, 2022 [Online]. Available: http://wiki.ros.org/

[46] “ROS/introduction - ROS wiki,” Accessed Jan. 16, 2022 [Online]. Available: http://wiki.ros.org/ROS/Introduction

[47] “roscpp - ROS wiki,” Accessed Jan. 16, 2022 [Online]. Available: http://wiki.ros.org/roscpp

[48] “Master - ROS wiki,” Accessed Feb. 8, 2022 [Online]. Available: http://wiki.ros.org/Master

[49] “Nodes - ROS wiki,” Accessed Jan. 16, 2022 [Online]. Available: http://wiki.ros.org/Nodes

[50] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, “Ros: an open-source robot operating system,” in *ICRA 2009*, p. 6.

[51] “Topics - ROS wiki,” Accessed Jan. 16, 2022 [Online]. Available: http://wiki.ros.org/Topics

[52] “Services - ROS wiki,” Accessed Feb. 8, 2022 [Online]. Available: http://wiki.ros.org/Services

[53] D. Davis and E. Dietz, “MLS node source code,” Available by request: dtdavi1@nps.edu. Available: https://gitlab.nps.edu/arsenl/swarm-autonomy/-/tree/mls

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California