drones

Article

Multi-Device Security Application for Unmanned Surface and Aerial Systems

Andre Leon *, Christopher Britt and Britta Hale *

Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943, USA; christopher.britt@nps.edu
* Correspondence: andre.leon@nps.edu (A.L.); britta.hale@nps.edu (B.H.)

Abstract: The use of autonomous and unmanned systems continues to increase, with uses spanning from package delivery to simple automation of tasks and from factory usage to defense industries and agricultural applications. With the proliferation of unmanned systems comes the question of how to secure the command-and-control communication links among such devices and their operators. In this work, we look at the use of the Messaging Layer Security (MLS) protocol, designed to support long-lived continuous sessions and group communication with a high degree of security. We build out MAUI—an MLS API for UxS Integration that provides an interface for the secure exchange of data between a ScanEagle unmanned aerial vehicle (UAV) and an unmanned surface vehicle (USV) in a multi-domain ad-hoc network configuration, and experiment on system limits such as the ciphersuite set-up time and message handling rates. The experiments in this work were conducted in virtual and physical environments between the UAV, USV, and a controller device (all of different platforms). Our results demonstrate the viability of capitalizing on MLS's capabilities to securely and efficiently transmit data for distributed communication among various unmanned system platforms.

Keywords: unmanned system security; UAS; messaging layer security (MLS); command and control; robotic operating system (ROS)

check for updates

Citation: Leon, A.; Britt, C.; Hale, B. Multi-Device Security Application for Unmanned Surface and Aerial Systems. *Drones* **2024**, *8*, 200. https://doi.org/10.3390/drones8050200

Academic Editors: Liuguo Yin and Shu Fu

Received: 21 April 2024
Revised: 11 May 2024
Accepted: 13 May 2024
Published: 15 May 2024

1. Introduction

Autonomous devices and unmanned systems (UxS) have become an increasing presence, ranging from shipment delivery [1] and airport protection [2]. In many cases, the use is not restricted to a single device and ground controller, but rather several devices interoperating, e.g., in swarming behavior or the transmission of relevant coordination data such as in self-driving vehicles. The *multi-device* setting common for UxS raises a natural question of how to secure communication links; under traditional communication protocols, links are secured on a one-to-one pairwise basis, implying that overhead scales linearly in the number of communication partners to a given entity. A new protocol, Messaging Layer Security (MLS) [3], proposed for standardization under the International Engineering Task Force (IETF) takes on the challenge of securing multiple devices as a "group" while reducing overhead that would normally occur with pairwise secured communication links. MLS was designed in the context of instant messaging but offers security properties that are attractive to many more application spaces in addition to efficiency for multi-device environments.

In this work, we look at the application of MLS to UxS, in particular to an unmanned surface vessel (USV) and unmanned aerial vehicle (UAV). We develop an integration application for MLS for secure and efficient command and control of distributed UxS platforms—the MLS API for UxS Integration (MAUI)—and assess ciphersuite efficiency within this application use environment.

1.1. Overview

The following phases represent a high-level overview of this work:

1.1.1. Phase 1

In the first phase, a virtual environment is used to test MLS between two host computers. This network configuration provided benchmark statistics for an MLS implementation over an 802.11 WiFi network. We used WireShark to capture and analyze packets. Once the implementation was verified, we compared the plaintext packet size to the ciphertext size, corresponding to the various MLS ciphersuites to calculate protocol overhead. The major differences in overhead were generated by the size of the cryptographic hash function, corresponding to authentication tags, in the selected ciphersuite.

1.1.2. Phase 2

In the second phase, we expanded the implementation from Phase 1 by integrating with the Robot Operating System (ROS). ROS was chosen because it is the primary operating system used in the UxS platforms chosen for this research. The test environment consisted of two Ubuntu Virtual Machines (VMs) installed with ROS to simulate the platforms. The VMs were tested with both 802.11n wireless network and military-grade mesh network radio configurations. We then ran our MLS ROS application on both VMs, one VM created the MLS group and the other joined the group.

For the initial metric of interest in this phase we considered MLS update frequency. We varied MLS update frequency using a range of 2, 10, 50, and 300 updates per 1000 messages, where "messages" refer to data transfer. After completing our initial test in this phase, we ran two benchmark tests to observe MLS protocol performance: initial group setup and message handling. We first tested and measured the time it took to set up and initialize an MLS group for different ciphersuite choices; initializing a group requires the creation of user credentials and the group itself. For MLS, this only happens once for any given group of devices. For message handling, we tested and measured how long it took to encrypt and send 1000 messages of 15 bytes each while updating the encryption key twice during the sending period. We cycled through all ciphersuite options with the same hash function and frequency parameters for baseline comparison.

1.1.3. Phase 3

In the third phase, we further expanded the implementation from Phase 2 by integrating our MLS ROS software onto the ScanEagle UAV VM and the Collaborative Autonomous Systems for Standoff Maritime Inspection and Response (CASSMIR) USV. The test environment consisted of (A) one Ubuntu VM serving as ground control for ScanEagle, (B) the ScanEagle UAV VM and (C) the CASSMIR USV. Control commands were sent while simultaneously sharing data with the CASSMIR USV using MLS over an ad-hoc Silvus military-grade wireless mesh radio network. The UAS VM was only utilized instead of the physical UAS device for reasons of flight permissions and ease of transport, given the size of the UAS. The VM was a high-quality simulation and was deemed equivalent to a physical ground test in lieu of flight permissions in the test locality.

1.2. Related Work

Messaging Layer Security Protocol

MLS is an Internet Engineering Task Force (IETF) end-to-end encryption standard built from the ground up to support communications for group sizes ranging into several thousands of devices [3]. While traditional protocols provide point-to-point secure channels, MLS provides a high degree of scalability by leveraging a tree-based key hierarchy to maintain a single, shared group key. This hierarchy supports key updates with $O(log(N))$ overhead.

MLS is also designed to provide Post-Compromise Security (PCS) and Forward Secrecy (FS). PCS is the protection of data after a compromise, also known as "self-healing" or "future secrecy", and is conditioned on the rotation of various keys as well as passive adversarial behavior [4,5]. On the other end, FS refers to the protection of data sent before a point of compromise [4].

Iterations of MLS have already been implemented in the commercial sector for communication services applications such as Cisco Webex and RingCentral [6], yet further applications of the protocol outside of instant messaging have yet to see much investigation. This includes the utilization of the MLS protocol for UxS. To date, one work has looked at the application of MLS within small UAS swarms [7]. However, that research focused on the viability of use (i.e., that small devices could handle the encryption) and tested within a single platform.

UxS Security

Related research on UxS secure links focuses on ciphertext algorithm performance [8,9] or on the improvement of network topography to allow for current cryptographic protocols to continue to function [10,11]. Secure communication protocols such as TLS and DTLS have been tested for efficiency in IoT [12,13], a somewhat related domain to unmanned systems. Those analyses conclude that TLS and DTLS, being not designed for IoT and the complexity of its infrastructure, were sub-optimal. Both protocols add latency to the network which should be considered as more devices join—a consideration that likewise applies to unmanned system uses. Prior work on using the MQTT protocol on unmanned systems [14] has shown an approximate overhead of 350 bytes of throughput with security—however, the security achieved by MQTT is considerably lower than what MLS provides.

While these works are relevant under the ever-increasing number of UxS and sensors that require interoperability, it continues to assume existent communication protocols designed for internet connections are the main available option for unmanned systems. Such works, therefore, do not cover the use or application of emergent protocols or communication standards that might address UxS needs in a holistic manner—link security, efficiency, and interoperability.

1.3. Contributions

This work contributes to the state of the art in unmanned system communication and command and control, demonstrating the viability of protocol use for improved security and efficient functionality. Specifically, we provide:

- test implementation and APO for integration of the Messaging Layer Security onto the Robotic Operating System.
- simulation benchmarking for overhead and setup time per ciphersuite.
- encryption key update interval frequency comparison testing.
- ciphersuite selection recommendations.
- the first cross-domain testing of MLS on unmanned systems, utilizing a ScanEagle UAS VM and CASSMIR USV physical device.

1.4. Outline

Section 2 describes the basics of the MLS architecture and protocol. Section 3 outlines the phased development and implementation approach for MAUI, used to integrate the MLS protocol within UxS platforms and methodology. It provides an overview of the use case design concept and the MLS and ROS architectures. Section 4 describes the experimentation and results. Finally, Section 5 summarizes the research.

2. UxS MLS Implementation

We start with a summary of MLS functionality. This research uses the MLS library developed and maintained by Cisco Following that, we introduce our API as an application that is device-agnostic and can be installed on any Unix-based system.

2.1. MLS Architecture

MLS [15] supports synchronous and asynchronous communication and authentication for groups of devices. It is designed to provide authenticated encryption through the use of symmetric keys (e.g., $Enc_K(data)$) while keys are distributed using a key encapsulation method ($KEM_{pk\_receiver}(K)$). MLS uses standard ciphersuites for encryption and key encapsulation but employs a novel key hierarchy for management which reduces the distribution overhead. This key management approach also supports *continuous key agreement* such that setup overhead is only incurred *once* for the lifetime of the devices. Thus, if devices $D_1, \ldots, D_n$ are in a group, they can be tasked separately and resume group communications on a joint tasking weeks or months later without re-incurring setup overhead. Due to space considerations, a detailed breakdown of key management in MLS can be found in the specification [15].

For practical deployment of MLS, there are two vital components of the implementation architecture: the Authentication Service and Delivery Service. Figure 1 depicts the protocol's architecture and overall functionality.

```
 ----------------       --------------
| Authentication |    | Delivery     |
| Service (AS)   |    | Service (DS) |
 ----------------       --------------
                    /        |         \            Group
                   / ************************************
                  /  *       |           \              *
       ----------    *   ----------        ----------   *
      | Client 0 |   *  | Member 1 |      | Member N |  *
       ----------    *   ----------        ----------   *
      ............   *  ............      ............  *
      User 0         *  User 0            User 1        *
                     *                                  *
                     ************************************
```

Figure 1. The Messaging Layer Security (MLS) Architecture [16].

Authentication Service Per the MLS Architecture RFC, the Authentication Service must provide two specific functionalities, namely "It authenticates the credentials used in a group… and it authenticates messages sent in groups by authenticating the message signature and the sending member's credentials" [16]. The Authentication Service is thus notably a *service functionality* rather than a *server*. The degree of the service functionality that is on the end-user device is determined by the implementation. Furthermore, the Authentication Service might be part of a federated service or something similar.

Delivery Service The Delivery Service stores and delivers initial public keys required by clients to establish a secret group key. The Delivery Service is in charge of broadcasting messages to all group members, which can have from two to thousands of clients sending and receiving information [16]. The client can be any device in the MLS architecture identified by a unique cryptographic signature. Like the Authentication Service, the Delivery Service is a service functionality versus a specific server. There are various instantiations of a Delivery Service. For example, message delivery might be managed by multiple end devices in the group to avoid a single point of failure in the architecture, and/or the group information can be saved on a separate server.

For our use case, the MLS application is installed on two UxS and a ground station. We abstract away part of the Authentication Service functionality, specifically the authentication of credentials for the group. We assume a pre-installed credential use case, for ease of implementation; however, such use of pre-installed certificates is potentially realistic for many UxS use cases, where all group devices may be "initialized" with required certificates before send-off. The function of a Delivery Service is minimalistic in our test case, conducted by the user creating the MLS group, in our case, the primary UxS.

2.2. MLS Application Programming Interface (API)

The MAUI API has three primary components. These are to **create users**, **create groups**, and **group maintenance**. The **create user** establishes a member's unique public and private keys and user credentials. Because MLS is a decentralized protocol, every user can create their own public and private keys. User credentials created and stored by an Authentication Service offer future capabilities similar to single sign-on for user control within an organization. **Create user** relies on two key variables, the first is the selected ciphersuite for the group, and the second is the username identifier. As the name implies, the **create group** creates a group or series of groups for a user. Lastly, **group maintenance** includes adding a user to the group, updating the keys for the group, and removing a user from the group. Each of the supporting group maintenance functions is followed by the **commit** sub-function. This **commit** sub-function provides an additional layer of security for agreement on adding, updating, or removing members. This group maintenance cycle is completed by sending a **welcome** message to the newly added group member and distributing a key **update** to existing members of the group. Furthermore, MLS allows for multiple simultaneous user additions followed by a single **commit**, a single **welcome** message, and a single **update** message to the group's current members. This functionality assists with the group key exchange performance capabilities of MLS. MLS version 12 supports the following ciphersuites:

(A) X25519_AES128GCM_SHA256_Ed25519
(B) P256_AES128GCM_SHA256_P256
(C) X25519_CHACHA20POLY1305_SHA256_Ed25519
(D) X448_AES256GCM_SHA512_Ed448
(E) P521_AES256GCM_SHA512_P521
(F) X448_CHACHA20POLY1305_SHA512_Ed448

Figure 2 is a step-by-step representation of the sequence in which each function executes for a successful encrypted message exchange using MAUI. These steps must be taken in sequence since the MAUI does not have contingencies to deal with out-of-sequence maintenance cycle processes. For example, messages being sent or received before **add**, **remove**, or **update** operations are concluded, and a **commit** by all parties is executed will break the users' connection to the group. In MLS, it is the responsibility of the Delivery Service to ensure in-sequence messages and future MLS applications can require Delivery Service design(s) that support concurrency mechanisms to ensure proper sequencing to overcome the lack of protocol contingencies.

2.3. MAUI and the Robot Operating System (ROS)

ROS is open-source software that provides a publisher and subscriber service and the necessary libraries to develop robotic applications. ROS applications are written in C++, Python, or Lisp. ROS runs on Unix-based systems with stable testing on Mac OS and Ubuntu [17].

The UxS platforms chosen for this use case utilize ROS; however, this implementation of MLS is agnostic to ROS internal functionality, leaving ROS functionality as an abstraction. The MLS MAUI application created for this use case interfaces with the existing subscribe and publish aspect of the ROS API as a ROS node for selected topic messages that will be encrypted and sent over a wireless ad-hoc network. For example, MAUI will subscribe to the GPS topic. As new GPS messages are posted to the topic, the MAUI application will retrieve them, encrypt them, and share them with the other MAUI users in the group. Figure 3 depicts MAUI's architecture and overall functionality.

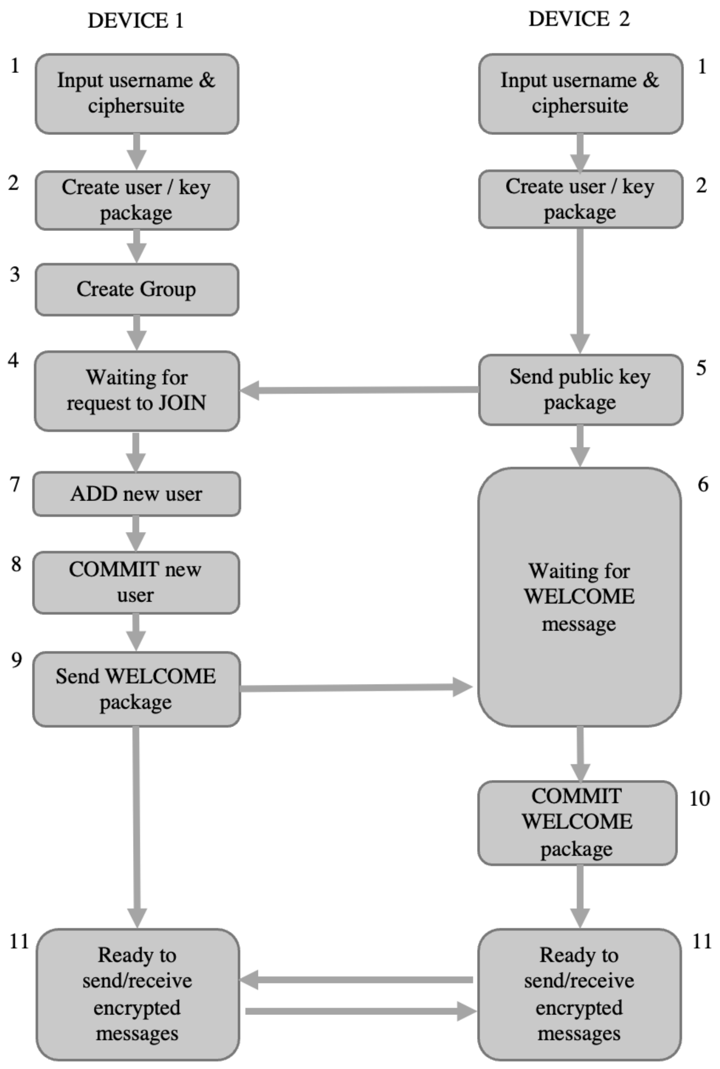

DEVICE 1
DEVICE 2
1 Input username & ciphersuite
2 Create user / key package
3 Create Group
4 Waiting for request to JOIN
7 ADD new user
8 COMMIT new user
9 Send WELCOME package
11 Ready to send/receive encrypted messages
1 Input username & ciphersuite
2 Create user / key package
5 Send public key package
6 Waiting for WELCOME message
10 COMMIT WELCOME package
11 Ready to send/receive encrypted messages

Figure 2. MLS Protocol Functionality Sequence.

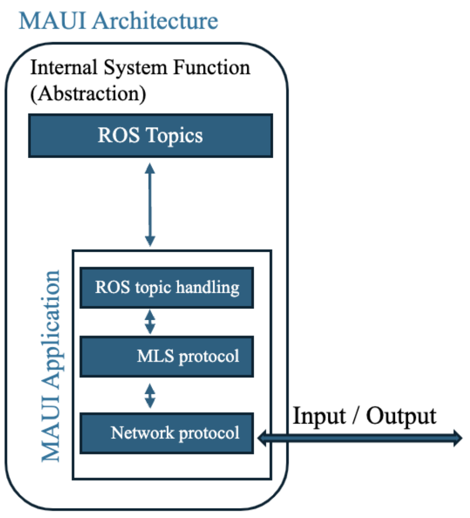
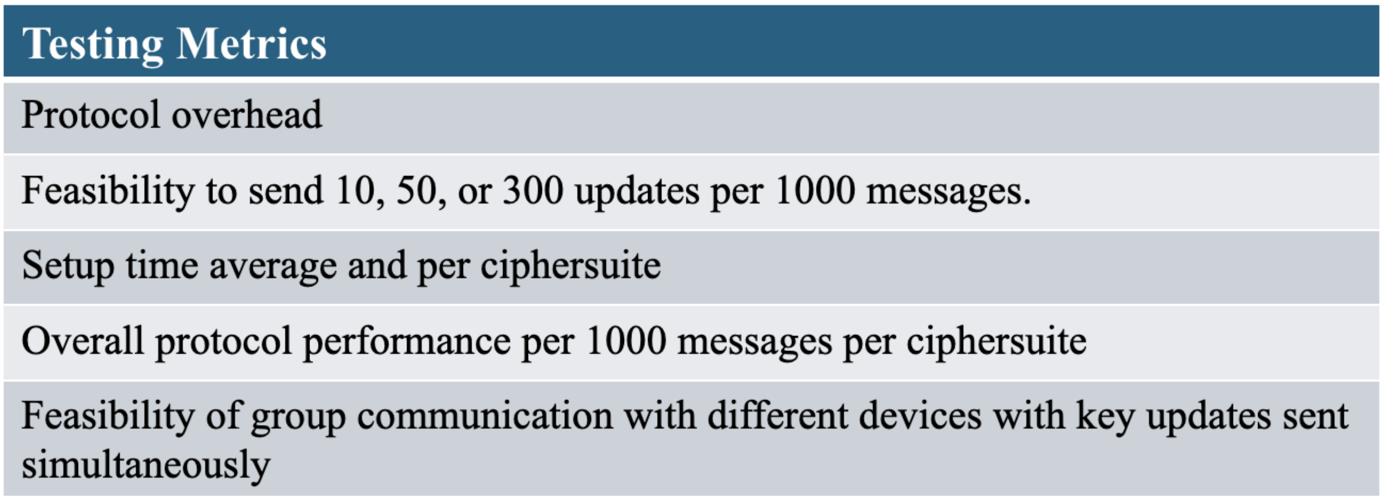


Figure 3. The MLS API for UxS Integration (MAUI) Architecture.

3. Methods and Materials

We develop an application that uses the MLS protocol as a viable means to provide secure command and control for UxS platforms and outlines the three-phase process utilized to create the MLS MAUI application that integrates the MLS and ROS libraries. We first test our application in a simulated environment and then integrate the application on two unique UxS platforms: a CASSMIR surface vessel and the ScanEagle developed by Insitu. The development of our application occurs in three sequential phases that build upon each other. These phases include Phase One, a basic chat application called MAUI. Phase Two is the addition of local ROS functionality to share static messages called MAUI ROS. Finally, Phase 3 ROS command and control applications update the MAUI ROS application to directly interface with the ROS Masters of the respective UxS platforms and add the ability to send C2 control messages to the UxS. We call this version MAUI ROS Live. The test objectives of these phases include the following metrics Figure 4:



Testing Metrics
Protocol overhead
Feasibility to send 10, 50, or 300 updates per 1000 messages.
Setup time average and per ciphersuite
Overall protocol performance per 1000 messages per ciphersuite
Feasibility of group communication with different devices with key updates sent simultaneously

Figure 4. The MAUI Metrics of interest.

Development Environment

The development environment where MLS is installed uses the Ubuntu 18.04 operating system or a later version, CMake 3.18.xx, ROS Noetic, C++ 17, and a compiled MLS library. The MLS library used for this implementation is draft C++ version 12 from the IETF MLS Working Group GitHub repository [18]. VSCode is the integrated development environment used due to the simplistic synchronization with the MAUI GitHub repository [19]. Leading up to MLS implementation on physical UxS platforms, we employed virtual instances for application development. These virtual instances are configured to the following parameters: one core CPU and 8 GB of RAM. This configuration sets a minimum baseline for UxS processing requirements for our test, which is below the performance baseline of our use case UxS. We made no modifications to the original MLS source code, functionality, or dependencies required to facilitate our MLS applications.

After compiling the MLS library, all library files and included directories are manually added to our MLS application folder hierarchy. Our application's CMakeList.txt file is updated to link the necessary MLS libraries to our program. Refer to our GitHub repository [19] for all three phases and a pre-compiled MLS library.

3.1. Experiment One Application—MAUI Chat

The development of MAUI chat in phase one allows us to understand and test the baseline requirements to implement the MLS functionality. The MAUI chat application was developed and run in Ubuntu virtual environments to simulate the UxS environment. We test the three primary components of the MLS protocol: create user, create group, and group maintenance. During this testing, we monitored the communication via WireShark to assess the packet contents to ensure the encryption of plaintext.

The following components are necessary for the initial application to create a secure chat session between two clients: MLS Functionality, Networking Functionality, and Message Exchange. The source code for MAUI chat is available in our GitHub repository named mls_chat [20]. To enable MLS functionality, we added the compiled MLS library to the development root folder containing the main.cpp file.

MLS Functionality

To access the MLS API library needed for our implementation, we added the following MLS header files throughout our code:

```
#include <mls/credential.h>
#include <mls/crypto.h>
#include <mls/session.h>
#include <mls/messages.h>
```

These header files provide access to required MLS classes and data types: **mls::Client, mls::Session**, **mls::PendingJoin**, and **bytes_ns::bytes**. All members of an MLS group are initialized by the client class that contains the required user credentials for that specific user. These stored credentials are then used to create or join a group for that user through the Session class. The Session class maintains a list of all groups initialized by a user and any groups joined by the user, managed by the MLS maintenance cycle functions (ADD, REMOVE, UPDATE, and COMMIT). For members to be added to the group, a user key package is created by the PendingJoin class. Once a user is added and committed to a group, that user receives the group's WELCOME message containing all the required artifacts to communicate with the group. This information is then saved to that user's session information. Once the groups are established, members can securely communicate over the chosen network.

To effectively encrypt and decrypt data across all our MLS applications, the functions **protect** and **unprotect** from the MLS library are used. These functions require data to be in type **bytes**; therefore, we need to convert other data types to and from **bytes**. To this end, we created a **convert** class that contains the necessary conversion functions for our application. These functions are **bytes_to_char**, **char_to_bytes**, **str_to_bytes**, and **bytes_to_str**.

Network Functionality

The MAUI chat application uses the Transmission Control Protocol (TCP) to establish server and client connections over an ad-hoc wireless network. The user creating the MLS group acts as the server for the TCP connection, in our case, the ScanEagle. Any other users trying to JOIN the group are denoted in our application as clients. To connect, every client needs to know the server's IP address. This IP address is entered at runtime. TCP was selected as the transport layer protocol because of the packet delivery guarantees inherent to the protocol. These guarantees allow us to implement MAUI chat connection in a more stable environment which mitigates packet loss that can adversely affect the sequencing of the MLS maintenance cycle [21]. For future work, a User Datagram Protocol (UDP) network can be implemented with the packet delivery guarantees at the application layer (using an appropriate MLS delivery service) to improve the group key management performance since UDP allows for the broadcast of messages [22].

Message Exchange

Following the initial group setup and the established TCP connection between server and client, the client then requests to join the group. If the client is added to the group, it receives the **welcome** message and joins the MLS group. The client is now part of the group, sends the first secure message to the server, and waits for a response. The message exchange is as follows.

The client generates a plaintext message of type **string** which is then converted to type **bytes** via our **convert** class. As indicated above, type **bytes** is then encrypted by the

MLS library **protect** function. The ciphertext output of the **protect** function is in type **bytes** and then it is converted to type **char*** to be transmitted over the wireless ad-hoc network. The receiver converts the encrypted message and converts it from type **char*** to type **bytes**, then calls the MLS library **unprotect** function to decrypt the message. Once the message is decrypted, it is converted from type **bytes** to type **string** in order for the message to be displayed. This bidirectional exchange of secure messages repeats until the server or client sends a message with the pound sign "#" to close the TCP connection.

The client MAUI chat functionality has two sections: sending and receiving. The sending section has the following sequence: input a message (data type string), convert the message (data type string to bytes), encrypt the message with MLS, convert the encrypted message (data type bytes to char*), send the encrypted message over TCP (data type char*). The receive section commences after a message is sent. This section has the following sequence: receive the encrypted message, convert the message (data type char* to bytes), decrypt the MLS message, convert the message (data type bytes to string), and finally display string. Between these two sections, the while loop checks if the message contains "#" sign. If the "#" sign is present, then the while loop stops and the program exits, else the while loop continues and prompts the user to input another message.

The server MAUI chat functionality has two sections: receiving and sending. The receive section has the following sequence: receive the encrypted message, convert the message (data type char* to bytes), decrypt the MLS message, convert the message (data type bytes to string), and finally display string. The receive section commences after a message is received. The sending section has the following sequence: input a message (data type string), convert the message (data type string to bytes), encrypt the message with MLS, convert the encrypted message (data type bytes to char*), send the encrypted message over TCP (data type char*). Between these two sections, the while loop checks if the message contains "#" sign. If the "#" sign is present, then the while loop stops and the program exits, else the while loop continues and waits for another message.

3.2. Experiment Two Application—MAUI ROS

The second MLS application, named MAUI ROS, is built upon all components of the MAUI chat application from phase one. MAUI ROS replaces the manual generation of plaintext messages found in MAUI chat with ROS plaintext topic messages that our application subscribes to. The plaintext ROS topic messages are published by the default publisher node named **talker** [23]. By using this publisher setup, it allowed us to control the speed at which messages are published, therefore, creating a controlled message delivery environment. During this phase, we install ROS Noetic [24] on two virtual environments. As in the MAUI chat server and client setup, MAUI ROS is installed on both virtual environments. One virtual environment simulates the user creating the MLS group, and the other, the user joining the MLS group. The default ROS installation creates a development folder named **catkin_ws/src**.

The two MAUI ROS applications send the data to the other node via TCP using MLS as the security protocol. Unlike MAUI chat, MAUI ROS allows for simultaneous bidirectional message exchange via threading implementation. The application has two threads. The first thread subscribes to a predefined ROS topic, encrypts the received topic message, and transmits it over the TCP connection. The second thread remains in a constant loop listening for incoming TCP messages over the established connection. When a message is received, it decrypts the received message. The received messages are ROS topic messages. Thus all ROS data are sent via MLS in our test environment. This data exchange simulates two different UxS from different countries or services sharing information over an ad-hoc wireless network.

3.3. Experiment Three Application—MAUI ROS Live

The third MLS application, named MAUI ROS Live, is built upon all components of the MAUI ROS application from phase two. Phase three has two primary design objectives.

The first is to exchange telemetry data between ScanEagle and CASSMIR. The second is to send control messages from a third party acting as ground control for ScanEagle. We installed MAUI ROS within the ScanEagle, CASSMIR, and ground control virtual environments during this phase. At runtime, MAUI ROS prompts the user to select which profile to run. The profiles ScanEagle, CASSMIR, or ground control. Each profile comes with specific functions enabled or disabled for overall use case functionality. The ScanEagle virtual environment creates the MLS group, and the CASSMIR and ground control virtual environments represent the second and third members of the MLS group. Figure 5, shows the architecture setup for the MAUI ROS Live application.

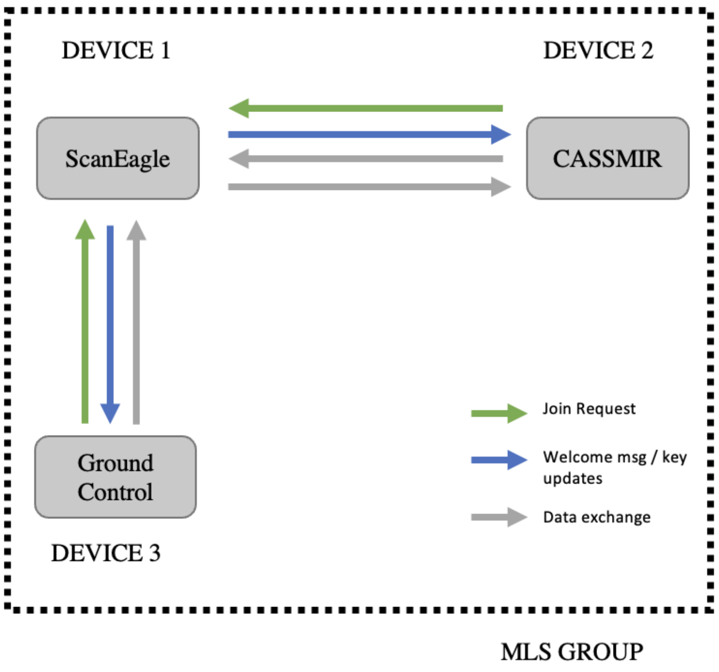


Figure 5. Depicts architecture used to test the MLS MAUI ROS Live Application.

For the first objective, MAUI ROS Live replaces the generic plaintext topic messages with actual ScanEagle and CASSMIR ROS topic data. The UxS unique topic data types of interest are the odometer data from ScanEagle and the GPS data from CASSMIR. Unlike MAUI ROS, where the topic messages are of data type **string**, the odometer and GPS data are more complex data types containing multiple parameters. To handle these complex data types, we created the **odom_to_string** and **gps_to_str** functions that convert each parameter within the unique data type to a string, and these newly concatenated strings are converted into a single string. This conversion allows MAUI ROS Live to utilize all prior functions to encrypt, decrypt, and send and receive the ScanEagle and CASSMIR data over the IP network. Since the odometer and GPS data have been concatenated, the received message must be parsed to display the data in their original form. To address this, we created the **parse_string** function which was also added to the **convert** class.

For the second objective, MAUI ROS Live adds the ability for the ScanEagle to receive commands from a third member acting as ground control. This functionality is accomplished by creating a third thread that allows for another TCP connection between ScanEagle and the ground control client as shown in Figure 6. The ground station prompts a user to pass a control command to adjust the ScanEagle turn rate. The turn rate must be a number between -30 and 30. The negative numbers represent a left turn, and the positive numbers represent a right turn. These control commands are sent utilizing all prior functions over the IP network. When the ScanEagle receives the control command, it also utilizes all prior functions. Once the message is converted into a data type **string** it is then converted to a **float** data type and published to the **turn_rate** topic to control the ScanEagle within the simulated environment.

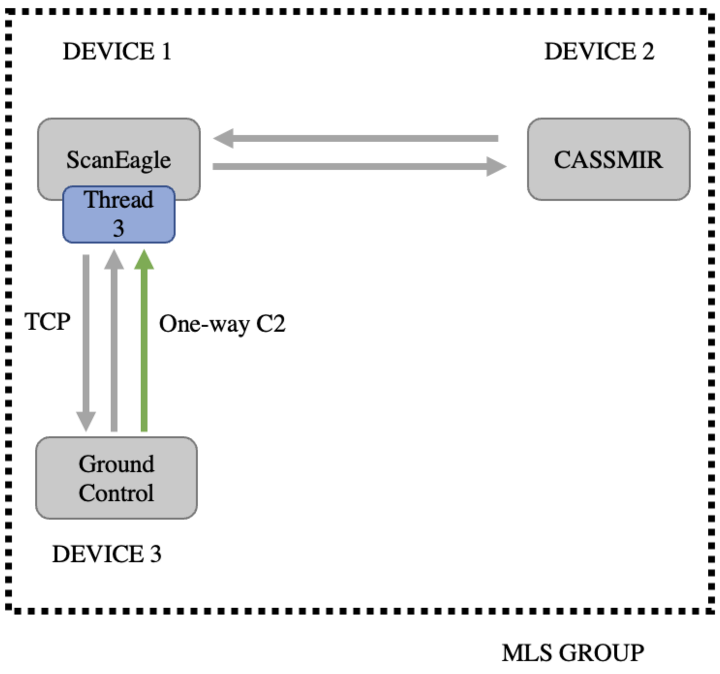


Figure 6. MAUI ROS Live Application third thread design.

4. Results

We apply three distinct phases of implementation. In the first phase, simulation was conducted in a virtual environment between two host computers to provide benchmark statistics and the assurances of an error-free MLS implementation over an 802.11 WiFi network. The second phase builds upon the first phase by replacing a single host computer with the ScanEagle platform virtual environment. The 802.11 WiFi network was replaced by using military-grade mesh network radios. This testing phase was used to assess performance in a semi-realistic communications environment and provide assurances of secure and efficient data exchange between the two devices. The final phase builds upon the prior two phases to provide command and control functionality between the ScanEagle and CASSMIR platforms. Live platform testing was also used in this phase to meet the overall research objective of providing command and control between two disparate UxS operating in the sea and air domain.

4.1. Experiment One—MAUI Chat

There were a total of two areas of interest identified to test and measure in phase one:

1. Encryption and Decryption of messages
2. MLS protocol ciphersuite overhead

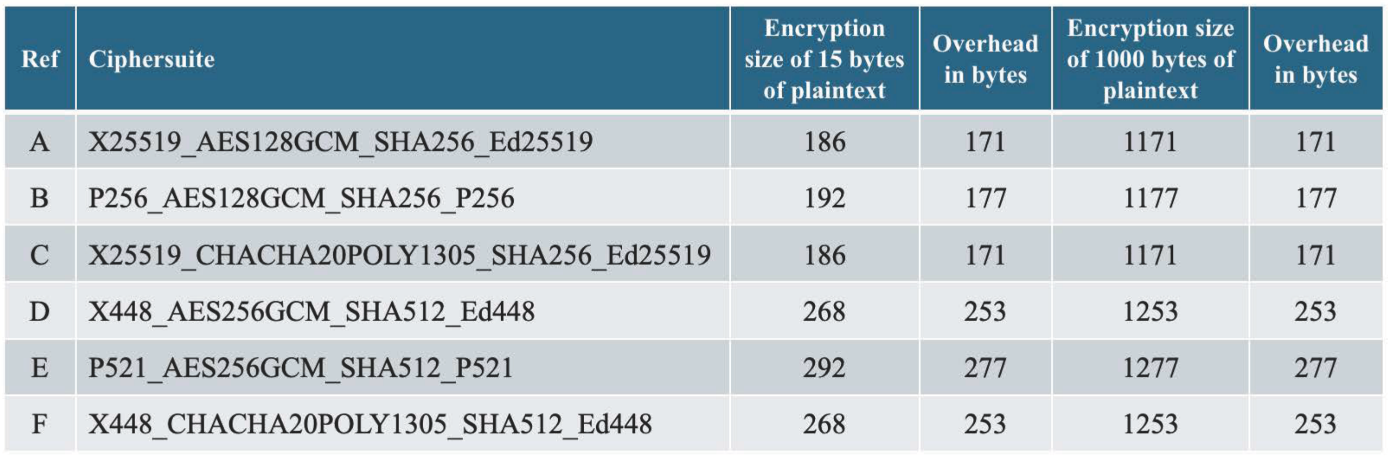
4.1.1. Simulation Environment

The test environment for phase one consists of two logically separated Ubuntu VMs connected to the same 802.11n wireless network. We install our MAUI chat application from Section 3.1 in each Ubuntu instance. Once the applications were installed, we ran our MAUI chat application on both VMs, one VM created the MLS group and the other joined the group. The message exchange configuration of MAUI chat allowed us to capture the variables of interest for this initial phase of testing.

4.1.2. Testing and Results

During this phase of testing, we ensured that MLS was implemented according to the specifications outlined in Section 2. We used WireShark to capture and analyze packets to confirm the encryption of plaintext. Once the implementation was verified, we compared the plaintext packet size to the ciphertext size, which corresponds to the chosen MLS ciphersuite to calculate protocol overhead.

We first sent an initial plaintext message of 15 bytes followed by the same message encrypted with each of the six ciphersuites available in MLS version 12. We also performed the same test with a plaintext message of 1000 bytes. Testing revealed that MLS encryption generated an overhead ranging from 171 to 277 bytes depending on the ciphersuite chosen, regardless of the plaintext size (15 B or 1000 B) as shown in Figure 7. The major difference in overhead seems to be generated by the size of the cryptographic hash function used in the selected ciphersuite (SHA256 or SHA512).



Ref	Ciphersuite	Encryption size of 15 bytes of plaintext	Overhead in bytes	Encryption size of 1000 bytes of plaintext	Overhead in bytes
A	X25519_AES128GCM_SHA256_Ed25519	186	171	1171	171
B	P256_AES128GCM_SHA256_P256	192	177	1177	177
C	X25519_CHACHA20POLY1305_SHA256_Ed25519	186	171	1171	171
D	X448_AES256GCM_SHA512_Ed448	268	253	1253	253
E	P521_AES256GCM_SHA512_P521	292	277	1277	277
F	X448_CHACHA20POLY1305_SHA512_Ed448	268	253	1253	253

Figure 7. Depiction of the different overhead cost per ciphersuite.

4.1.3. Findings

The available capacity of the network must be carefully considered when employing encryption techniques for the variety of messages used for UxS communications. As such, this first experiment primarily focuses on understanding the average added overhead of a selected MLS ciphersuite for a common plaintext message. During this phase, we do not employ the full scope of the MLS protocol maintenance messages (such as update, add, remove, etc.) to better identify baseline metrics. Although SHA512 is double the cryptographic strength against a brute force attack as opposed to SHA256, the additional overhead is not ideal for our use case that has the potential to send an aggregate of thousands of messages per second. This is also highly dependent upon the UxS group size and the information the group is sharing.

4.2. Experiment Two—MAUI ROS

There are a total of three areas of interest identified for test and performance measurement in phase two:

1. MLS update intervals
2. Initialization benchmarks per MLS ciphersuite
3. MLS message handling metrics

4.2.1. Simulation Environment

Before actual unmanned systems integration, the test environment for phase two consists of two virtual Ubuntu VMs installed with ROS software (version 1.15.1, Open Robotics Consortium, Berkeley, CA, USA). We installed the MAUI ROS application in each VM. Once the applications are installed, we start the ROS master on each VM, and the default ROS talker node is modified to publish 1000 messages per second. We then ran our MLS ROS application on both VMs, where one VM creates the MLS group and the other joins the group. We tested the VMs connected to an 802.11n wireless network and mesh network radios separately to confirm functionality. It is important to note, that for our overall performance testing, we utilized the 802.11n wireless to gather the required metrics for our experiments. Mesh radios were employed in the testbed; however, as the radios add a second layer of proprietary encryption, we exclude them from the metric collection to ensure a clear comparison.

4.2.2. Testing and Results

For the initial metric of interest in this experiment, we consider the MLS update frequency. We vary the MLS update frequency using a range of 2, 10, 50, and 300 updates per 1000 messages. At runtime, the user creating the group has the option to run a benchmark test, selecting the total messages to send and how many MLS key updates to perform. We selected 1000 messages to send with two key updates during the message session. Without performing key updates, MLS-protected messages could be sent between two VMs at a rate of 1000 messages per second. When performing key updates MLS protected messages could be sent between two VMs at a rate of 100 messages per second, for the reasons as follows: The trial's first test showed that key updates were not being processed in sequence, causing key schedule inconsistency in the MLS session. It was identified that this error was due to group members receiving new messages while an **add** and **commit** of the new key update was being performed (This issue derives from the absence of Delivery Service design inclusion into our application. This is further discussed in the limitations, Section 4.4). To mitigate this issue we manually reduced the message delivery rate of the default ROS talker.cpp file in our application. We found the optimal message transmission rate for all ciphersuites to be 100 messages per second.

After completing the initial test in this phase, we ran two benchmark tests to observe MLS protocol performance under MAUI ROS: initial group setup and message handling. We first tested and measured the time it took to set up and initialize an MLS group depending on the ciphersuite chosen; initializing a group requires the creation of user credentials and the group itself. For message handling, we tested and measured how long it took to encrypt and send 1000 messages of 15 bytes each while updating the encryption key two times during the sending period.

We cycled through all ciphersuite options with the same parameters for baseline comparison. The results of the group initialization times per ciphersuite are shown in Figure 8. These tests reveal that the average setup time for five out of the six best-performing ciphersuites was 5.1 ms; with `P521_AES256GCM_SHA512_P521` being the outlier performing 3.5 times slower than the average. Furthermore, the message handling results are shown Figure 9 indicated that following group initialization, `X25519_AES128GCM_SHA256_Ed25519` and `P256_AES128GCM_SHA256_P256` ciphersuites had the worst performance, although their initial group set up times were faster than average. These combined results indicate points for consideration when selecting a particular ciphersuite suitable for the use of MLS within UxS applications.

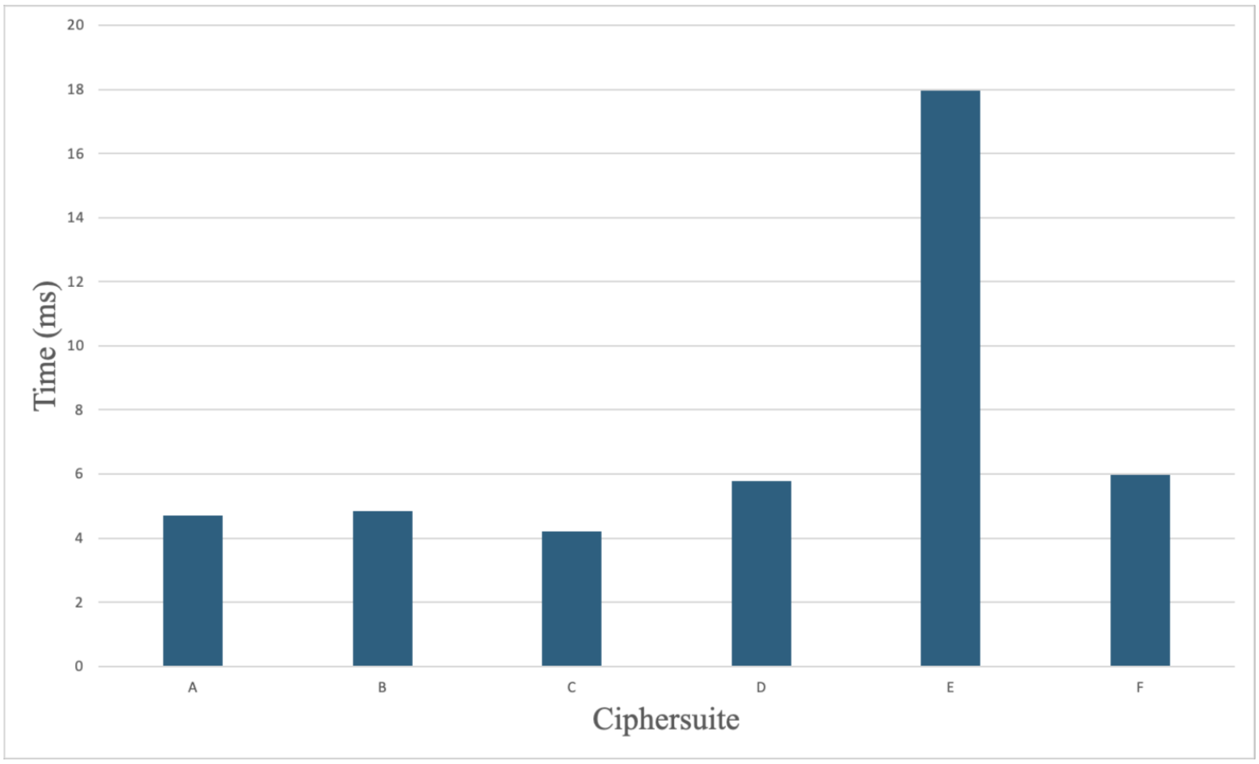


Figure 8. Ciphersuite setup time comparison in milliseconds, MAUI ROS.

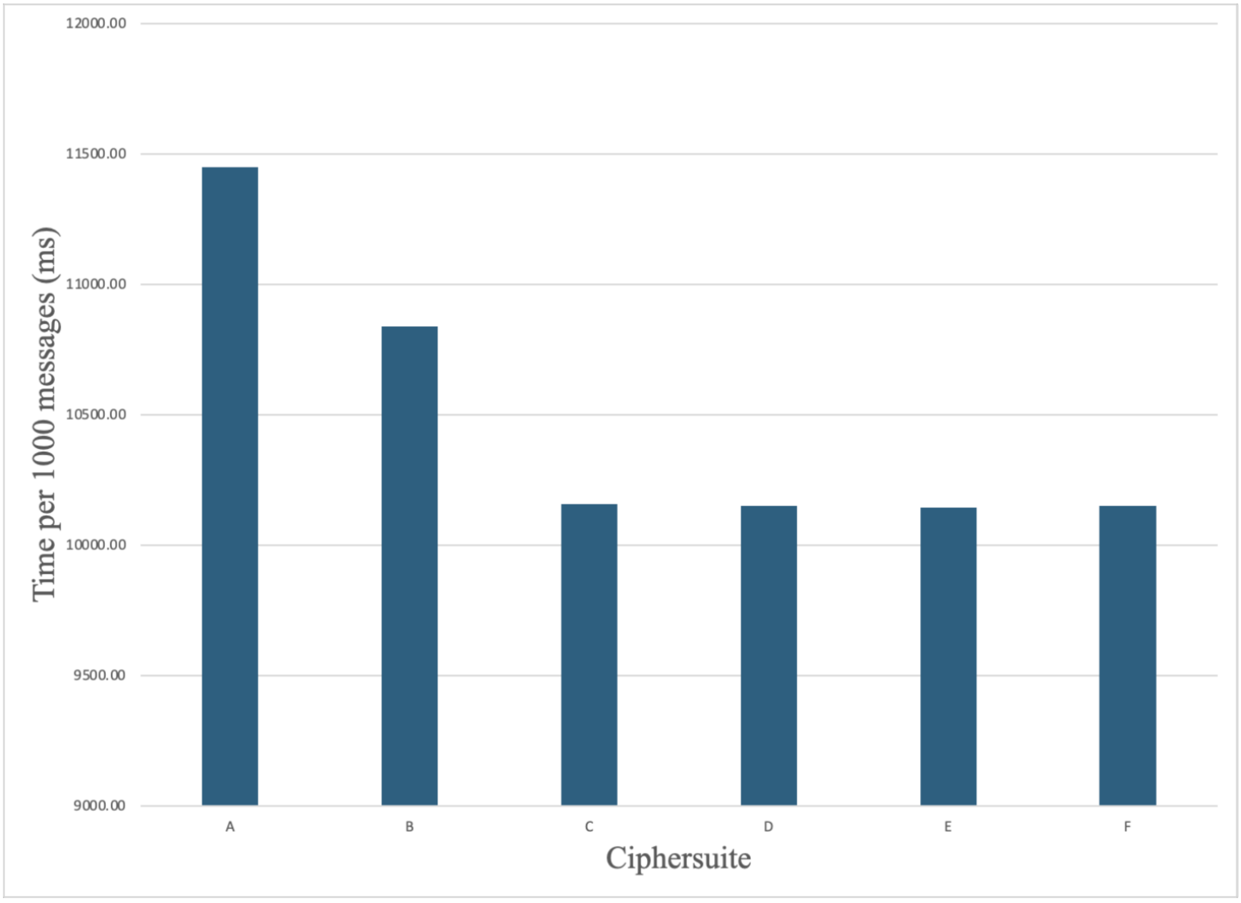


Figure 9. Message handling time comparison in milliseconds per 1000 messages, MAUI ROS.

4.2.3. Findings

This second experiment primarily considers UxS performance due to varying form factors associated with UxS types ranging from handheld devices to full-scale unmanned aircraft and ships. From this performance experiment, ciphersuites `X448_AES256GCM_SHA512_Ed448`, `X448_CHACHA20POLY1305_SHA512_Ed448`, and `X25519_CHACHA20POLY1305_SHA256_Ed25519` appear to provide a good performance balance between MLS setup time and message handling. However, if the UxS chosen has significant processing power limitations or if concerns with network capacity exist, as explained in our first experiment, we recommend `X25519_CHACHA20POLY1305_SHA256_ Ed25519`. This ciphersuite provides a slightly lower overhead per message by reducing the cryptographic hash functions from SHA512 to SHA256.

4.3. Experiment Three—MAUI ROS Live

The section discusses the testing conducted for the proof-of-concept for this thesis use case. There were a total of three areas of interest identified:

1. Data exchange between ScanEagle and CASSMIR using MLS
2. Command and control of ScanEagle using MLS
3. MLS Key Update

4.3.1. Simulation Environment

The test environment for phase three consists of (A) one virtual Ubuntu VM serving as ground control for ScanEagle, (B) the ScanEagle UAV VM, and (C) the physical CASSMIR USV (shown in Figure 10). The MAUI ROS Live application is installed in each environment. Once the applications were installed and compiled, we initialized MAUI ROS Live on all three environments in sequence, starting with the ScanEagle, then CASSMIR, and finally the ground station VM. The ScanEagle VM creates the MLS group, and all others join the group. The tested environments were connected to mesh network radios separately for physical layer transport.

For the CASSMIR to join and communicate within the MLS group consisting of the ScanEagle and ground control, we added a second network adapter in the CASSMIR. This adapter was configured with a similar host IP address and network mask as the ScanEagle and the ground control station. Adding a dual network adapter allowed the CASSMIR to join the ad-hoc network without modifying its baseline internal network architecture.

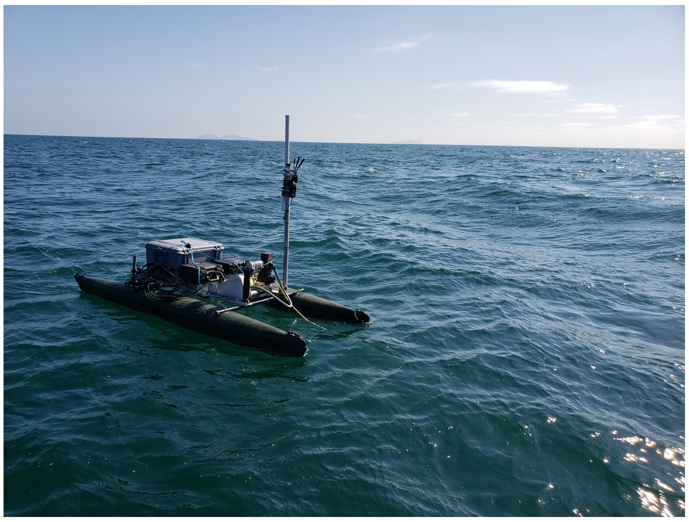


Figure 10. Physical CASSMIR USV platform.

4.3.2. Testing and Results

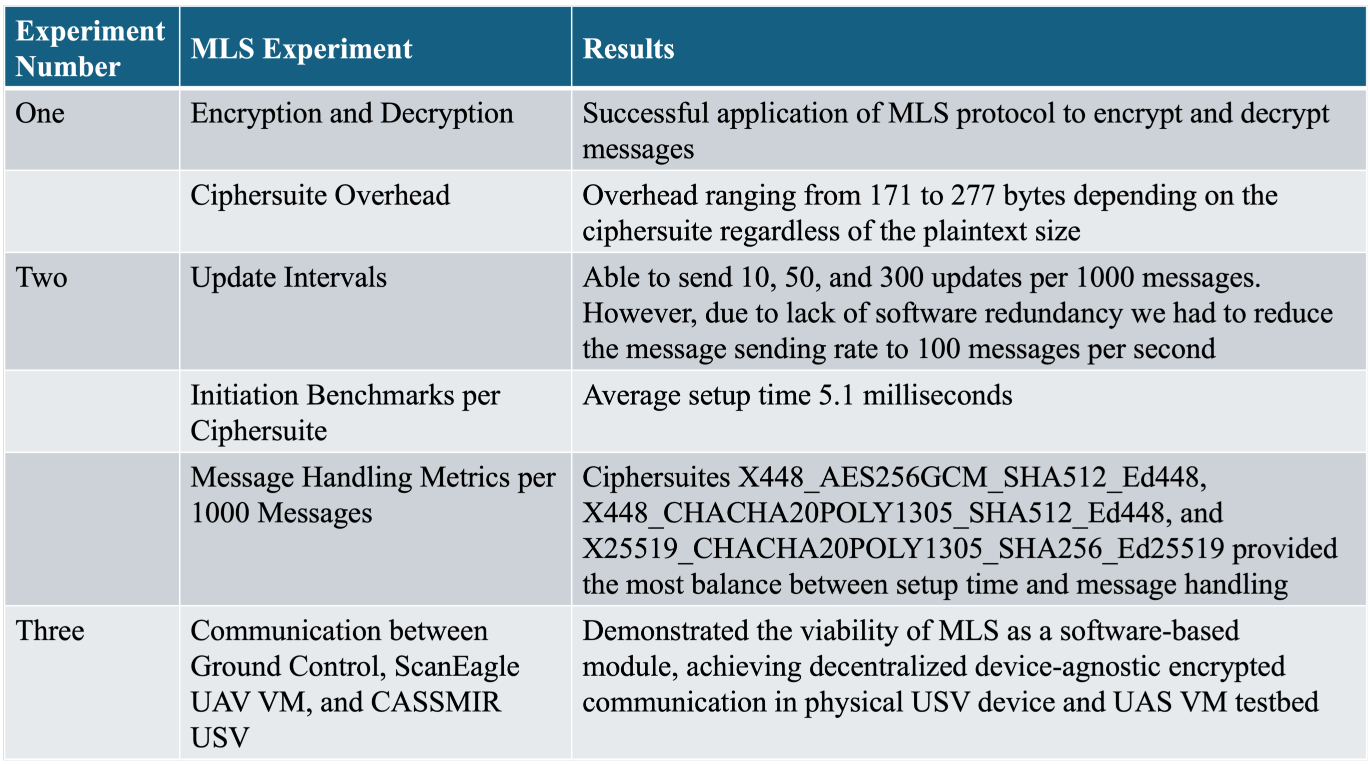
During this phase, the first metric of interest was to integrate the CASSMIR, the ScanEagle, and the ground control and share UxS data. We successfully used MLS to securely exchange data between the ScanEagle and CASSMIR. The ScanEagle UAV shared its odometer data with the CASSMIR USV which displayed the UAV positional data on the CASSMIR terminal. The CASSMIR USV shared its GPS data, also displaying the USV positional data on the ScanEagle terminal.

For our second metric of interest, the command and control testing of the ScanEagle was also a success. The ScanEagle received an MLS encrypted turn-rate message from the ground control VM, decrypted the message, and then published this command to the appropriate ROS topic. The ground control commands were sent while simultaneously sharing data with the CASSMIR USV using MLS over an ad-hoc wireless mesh radio network.

For our final metric of interest, we repeated our second metric test while running the testing mode of our program to calculate setup times and the number of MLS updates sent. We attempted to conduct a total of five MLS key updates within a single transmission of 1000 messages during this test. However, our implementation's lack of concurrency made it difficult for ground control to join the MLS group while updates were taking place. To solve this issue, we artificially reduced the speed at which the application processed messages on the ScanEagle specifically, as the group manager, putting the odometer data sending process to sleep on the ScanEagle before processing data for encryption and sending, resulting in a ScanEagle send rate of two messages per second, and receive rate of eight messages per second. Thus, as the group manager, the ScanEagle processed a total of 10 messages per second. This new setting allowed enough time to add ground control to the MLS group before the key updates started. We were then able to successfully process five group key updates while transmitting 1000 messages while ScanEagle and CASSMIR shared data, and the ScanEagle received commands from the ground station. However, it is important to note that if we had implemented a Delivery Service or concurrency controls (data processing sequence scheme) there would not have been a need to put the process to sleep, and therefore, we have been able to fully utilize the available computational capabilities and network. Nonetheless, with this testbed workaround, it is notable that MLS ran successfully on the multi-UxS testbed.

4.3.3. Findings

Our first two experiments provide an understanding of how the MLS protocol (sans Delivery Service) performs with respect to network capacity and the common UxS operating system (shown in Figure 11). With these additional data, our final experiment is the real-world application of the MLS protocol that demonstrates the viability of obtaining the security gains of the protocol for UxS use cases. The final test environment also shows the viability of MLS as a software-based module, that can be implemented for achieving decentralized device-agnostic encrypted communication. Additionally, our experiments show the protocol capability of MLS to provide real-time on-the-fly flexible encryption key updates as needed translates into the UxS environment.



Experiment Number	MLS Experiment	Results
One	Encryption and Decryption	Successful application of MLS protocol to encrypt and decrypt messages
	Ciphersuite Overhead	Overhead ranging from 171 to 277 bytes depending on the ciphersuite regardless of the plaintext size
Two	Update Intervals	Able to send 10, 50, and 300 updates per 1000 messages. However, due to lack of software redundancy we had to reduce the message sending rate to 100 messages per second
	Initiation Benchmarks per Ciphersuite	Average setup time 5.1 milliseconds
	Message Handling Metrics per 1000 Messages	Ciphersuites X448_AES256GCM_SHA512_Ed448, X448_CHACHA20POLY1305_SHA512_Ed448, and X25519_CHACHA20POLY1305_SHA256_Ed25519 provided the most balance between setup time and message handling
Three	Communication between Ground Control, ScanEagle UAV VM, and CASSMIR USV	Demonstrated the viability of MLS as a software-based module, achieving decentralized device-agnostic encrypted communication in physical USV device and UAS VM testbed

Figure 11. Summary of analysis.

4.4. Experiment Limitations

Due to time constraints and the complexities of coding and implementing robust network controls and message-handling schemes, we opted to use application threading to manage MLS group users for development simplicity. Consequently, we forfeited a more complex design that can dynamically set up, update, and track multiple TCP connections.

This design decision also affected message handling. Per the error discovered in phase two testing, our application does not have the necessary exception handling and concurrency controls to manage out-of-sequence MLS maintenance messages. Therefore, our application is susceptible to the message receive rate, which can be quite fast in UxS settings. The use of a Delivery Service in combination with concurrency controls can overcome this implementation limitation by blocking the MLS group members from sending new messages until all group members have processed the new group key.

5. Conclusions

This research outlined our MAUI implementation of the MLS version 12 (using the Cisco library) for UxS within virtual and physical environments. MLS is of particular interest due to its security properties and adaptability in supporting long-lived sessions. The latter, in particular, makes it resistant to link dropping that can occur in UxS use cases and alleviates the need for regular session reestablishment.

The testing of our MLS MAUI ROS and MLS MAUI ROS Live applications in the virtual environment provides evidence that the use of MLS can support encryption and decryption requirements to exchange data between disparate UxS systems operating in multiple domains. Tests of our MLS MAUI ROS Live application on a combination of virtual and physical devices demonstrate that MLS can serve as a multi-device security protocol that optimizes interoperability, agnostic to IP network and platform type in a multi-domain ad-hoc network configuration. Our use case testing results show that the MLS protocol is a viable solution for our multi-domain proof of concept for secure data exchange and command and control of distributed and dissimilar UxS agnostic to IP network architecture.

Achieving these results is a significant step towards improving UxS security hand-in-hand with interoperability.

Author Contributions: Conceptualization, A.L. and C.B. and B.H.; methodology, A.L. and C.B. and B.H.; software, A.L. and C.B.; validation, A.L. and C.B.; investigation, A.L. and C.B.; resources, B.H.; writing—original draft preparation, A.L. and C.B. and B.H.; writing—review and editing, A.L. and B.H.; visualization, A.L. and C.B. and B.H.; supervision, B.H.; project administration, B.H.; funding acquisition, B.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by CRUSER grant number 4019.00.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Acknowledgments: We would like to thank Jonathan Lussier, Anthony Jones, Annie Lin, Joeseph Lukefahr, and Aurelio Monarrez for their input and testing access support at various points in this project.

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

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
C2	Command and Control
CASSMIR	Collaborative Autonomous Systems for Standoff Maritime Inspection and Response
FS	Forward Secrecy dichroism
IETF	Internet Engineering Task Force
MAUI	MLS API for UxS Integration
MDPI	Multidisciplinary Digital Publishing Institute
MLS	Messaging Layer Security
PCS	Post Compromise Security
ROS	Robotic Operating System
TCP	Transmission Control Protocol
UAS	Unmanned Aerial System
UDP	User Datagram Protocol
USV	Unmanned Surface Vehicle
UxS	Unmanned System
VM	Virtual Machine

References

1. Streitfeld, D. Look, Up in the Sky! It's a Can of Soup! *The New York Times*, 4 November 2023.
2. Lee, C.H.; Thiessen, C.; Van Bossuyt, D.L.; Hale, B. A Systems Analysis of Energy Usage and Effectiveness of a Counter-Unmanned Aerial System Using a Cyber-Attack Approach. *Drones* **2022**, *6*, 198. [CrossRef]
3. Barnes, R.; Beurdouche, B.; Robert, R.; Millican, J.; Omara, E.; Cohn-Gordon, K. *The Messaging Layer Security (MLS) Protocol—Draft 20*. 2023. Available online: https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/20/ (accessed on 6 May 2024)
4. Cremers, C.; Hale, B.; Kohbrok, K. The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual, 11–13 August 2021; USENIX Association: Berkeley, CA, USA, 2021; pp. 1847–1864.
5. Dowling, B.; Hale, B. Secure Messaging Authentication against Active Man-in-the-Middle Attacks. In Proceedings of the 2021 IEEE European Symposium on Security and Privacy (EuroS&P), Vienna, Austria, 6–10 September 2021.
6. CISCO. *Zero-Trust Security for Webex White Paper*; CISCO: San Jose, CA, USA, 2021.
7. Dietz, E.; Davis, D.; Hale, B. Utilizing the Messaging Layer Security Protocol in a Lossy Communications Aerial Swarm. In Proceedings of the 56th Hawaii International Conference on System Sciences, HICSS 2023, Maui, HI, USA, 3–6 January 2023; Bui, T.X., Ed.; ScholarSpace: Merced, CA, USA, 2023; pp. 6591–6600.
8. Ozmen, M.; Yavuz, A. Dronecrypt—An Efficient Cryptographic Framework for Small Aerial Drones. In Proceedings of the MILCOM 2018—2018 IEEE Military Communications Conference (MILCOM), Los Angeles, CA, USA, 29–31 October 2018.
9. Thompson, R.B.; Thulasiraman, P. Confidential and Authenticated Communications in a Large Fixed-Wing UAV Swarm. In Proceedings of the 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), Cambridge, MA, USA, 31 October–2 November 2016.

10. Xiong, F.; Li, A.; Wang, H.; Tang, L. An SDN-MQTT Based Communication System for Battlefield UAV Swarms. *IEEE Commun. Mag.* **2019**, *57*, 41–47. [CrossRef]
11. Chen, X.; Tang, J.; Lao, S. Review of Unmanned Aerial Vehicle Swarm Communication Architectures and Routing Protocols. *Appl. Sci.* **2020**, *10*, 3661. [CrossRef]
12. Tiburski, R.T.; Amaral, L.A.; de Matos, E.; de Azevedo, D.F.G.; Hessel, F. Evaluating the use of TLS and DTLS protocols in IoT middleware systems applied to E-health. In Proceedings of the 2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 8–11 January 2017; pp. 480–485. [CrossRef]
13. Kothmayr, T.; Schmitt, C.; Hu, W.; Brünig, M.; Carle, G. A DTLS based end-to-end security architecture for the Internet of Things with two-way authentication. In Proceedings of the 37th Annual IEEE Conference on Local Computer Networks—Workshops, Clearwater, FL, USA, 22–25 October 2012; pp. 956–963. [CrossRef]
14. Mukhandi, M.; Portugal, D.; Pereira, S.; Couceiro, M.S. A novel solution for securing robot communications based on the MQTT protocol and ROS. In Proceedings of the 2019 IEEE/SICE International Symposium on System Integration (SII), Paris, France, 14–16 January 2019; pp. 608–613.
15. Barnes, R.; Beurdouche, B.; Robert, R.; Millican, J.; Omara, E.; Cohn-Gordon, K. The Messaging Layer Security (MLS) Protocol—Draft 13. 2022. Available online: https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/13/ (accessed on 7 May 2024).
16. Beurdouche, B.; Rescorla, E.; Omara, E.; Inguva, S.; Kwon, A.; Duric, A. *The Messaging Layer Security (MLS) Architecture*. 2023. Available online: https://datatracker.ietf.org/doc/html/draft-ietf-mls-architecture-10 (accessed on 10 May 2024).
17. ROS.org: Robotic Operating System. ROS Wiki. Available online: https://wiki.ros.org/ (accessed on 27 February 2023).
18. Barnes, R.; Nandakumar, A.S.; Roques, O.; Jennings, C.; Idicula, J. mlspp. 2022. Available online: https://github.com/cisco/mlspp (accessed on 10 May 2024).
19. Leon, A.; Britt, C. Brosito Repository, 2022. Available online: https://github.com/brosito/ (accessed on 6 May 2024).
20. Leon, A.; Britt, C. mls_chat. 2022. Available online: https://github.com/brosito/mls_chat (accessed on 6 May 2024).
21. USC. *Transmission Control Protocol*; USC. 1981. Available online: https://www.rfc-editor.org/rfc/rfc793 (accessed on 6 May 2024).
22. Postel, J. *User Datagram Protocol*; 1980 . Available online: https://www.rfc-editor.org/rfc/rfc768 (accessed on 4 May 2024).
23. ROS Tutorials Writing Publisher Subscriber (c++). ROS Wiki. Available online: https://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29 (accessed on 8 May 2022).
24. Ubuntu Install of ROS Noetic. ROS Wiki. Available online: https://wiki.ros.org/noetic/Installation/Ubuntu (accessed on 8 May 2022).