오픈 소스 기반 드론에 대한 취약점 방어 기술 개발

중간 보고서

지도 교수 : 손준영

분과 : D 하드웨어/보안

팀명: Joy security

팀원: 201924523 이경민

201624587 조수현

목 차

1. 요구조건 및 제약 사항 분석에 대한 수정사항

1.1. 과제 배경

최근 드론, 즉 UAV(Unmaned Aerial Vehicle) 기술의 발전에 따라 UAV 가 군사, 건설, 농업, 영상 촬영 등 다양한 분야에서 활용되고 있으며, 시장 또한 빠르게 성장하고 있다. 또한, 향후 물류 수송, 교통 관제, 통신 등 새로운 서비스/산업 분야로 그 영역을 확대하는 등 새로운 성장동력으로 주목받고 있다.

미래 항공 산업의 핵심 기술로 드론이 부각되면서 미국, EU, 중국, 일본 등 세계 각국에서 무인 항공기의 단계별 발전 로드맵을 발표하여, 드론 산업 육성과 시장 활성화에 노력을 기울이고 있다. 국내에서도 드론 산업 발전 기본 계획으로 네거티브 방식의 규제 최소화를 통한 드론 산업 실용화를 위해 법제도적 지원을 하고 있다.

하지만 이러한 급격한 UAV 시장의 성장은 UAV 의 취약점을 악용한 다양한 공격의 급증을 유발하고 있다. UAV 는 원거리에서 무선으로 원격 조정하거나 입력된 프로그램에 따라 비행하고, 센서를 통해 데이터를 전송한다. 이처럼 드론과 지상 제어 장치가 네트워크로 연결되어 있기 때문에, 드론이 탈취당하거나 서비스 장애가 발생할 수 있다.

드론 서비스에서 발생될 수 있는 사이버 보안 위협을 조금 더 자세히 살펴보면, 드론, 지상 제어 장치, 정보 제공 장치와 같은 자산 요소에서 발생될 수 있는 보안 위협이 존재한다.

먼저, 드론 환경에서는 GPS Spoofing 과 GPS Jamming, 제어신호 전파 방해 및 무력화, 센서 교란, 임베디드 시스템의 펌웨어 변조를 통한 시스템 권한 탈취 등이 발생될 수 있다. 다음으로, 지상 제어 장치에서는 기밀 정보 유출, 암호키 노출 및 취약한 암호 알고리즘으로 인한 제어권 탈취, 잘못된 설계 및 구성으로 인한 소프트웨어 오류 등이 발생될 수 있다. 또한, 정보 제공 장치 상에서는 세션 하이재킹, 재전송 공격, 중간자 공격과 같은 메시지 위변조 공격, 데이터 위변조 등이 발생될 수 있다.

또한, 드론 개발에 오픈 소스 코드를 이용하면서 발생할 수 있는 문제가 존재한다. 최근의 드론 개발자들은 드론 비행 제어 소스 코드의 크기가 커지고 기능들이 많아짐에 따라 오픈소스를 가져와 활용하는 것에 익숙해지고 있으며 별도의 보안 취약점에 대한 점검없이 활용하고 있다. 이러한 오픈소스는 공격자가 접근 가능하기 때문에 다양한 취약점에 노출될 수밖에 없다

.

1.2. 과제 목표

드론 사이버 공격의 실제 사례를 살펴보면, 2011 년 2 월 이란 핵시설을 정찰 중이던 미국 드론을 대상으로 GPS Spoofing 기술을 사용하여 드론을 탈취하는 사건이 발생했고, 2014 년 11 월에는 드론 시스템의 취약점을 활용하여 해킹, 촬영 영상 복원, 드론을 복제하는 사건이 발생하였다. 뿐만 아니라, 한국에서도 미확인 드론이 서울 상공에서 비행을 하는 등 소형 비행체를 이용한 크고 작은 사건들이 발생하고 있다.

이러한 문제들을 해결하기 위해 드론 시스템의 취약점을 파악하고, 취약점을 보완할 수 있는 방안을 탐색하는 것이 중요하다. 이는 드론 시스템의 기밀성, 무결성, 가용성의 보장으로 이어지므로, UAV 운영의 안전성과 신뢰성을 확보할 수 있다. 또한, 드론 기술이 더욱 다양한 산업 분야에서 안정적으로 활용될 수 있는 기반이 될 것이다.

이 과제의 목표는 대표적인 오픈소스 UAV 플랫폼인 PX4 Autopilot 과 MAVLink 프로토콜을 대상으로 알려진 취약점을 분석하고, 이러한 취약점에 대한 방어 기법을 개발하고 최적화하는 것을 목적으로 한다.

1.3. 요구조건 및 제약 사항 분석에 대한 수정사항

1.3.1. 기존 요구조건

Gazebo 시뮬레이터를 이용하여 GPS 스푸핑, 잘못된 센서 데이터 입력 등 다양한 악의적인 입력을 통해 시스템의 반응을 테스트하고, 데이터 전송 중 발생할 수 있는 위협 (중간자 공격, 재전송 공격 등)을 테스트한다. 또한, 보안 이벤트를 모니터링하고, 로그를 분석하여 보안 위협을 발견한다. 이후, 취약점의 심각도와 영향 범위 등을 분석하여, 보안 강화 방안 탐색의 기반이 될 수 있도록 한다.

Wireshark 를 통해 MAVLink 메세지를 분석하여 메세지의 무결성을 확인하고, MAVLink 는 기본적으로 암호화를 제공하지 않기 때문에 스니핑 등을 통해 암호화되지 않은 메세지에 대해 일어날 수 있는 공격 방안을 예측하고 예방할 수 있는 방법 또한 설계할 수 있도록 한다.

1.3.2. 제약 사항 및 수정 사항

초기 단계에서 계획한 내용으로는 프로토콜의 payload 를 암호화하여 패킷을 전송하는 것으로 기밀성의 수준을 향상한 뒤, 서명 및 인증 등의 기법을 활용하여 무결성을 강화하는 것이 목적이었으나 몇 가지 제약 사항에 의해 진행 방향을 수정하였다.

- **시스템의 복잡성**

  PX4 Autopilot 과 MAVLink 는 상용화되어 있는 정교한 시스템을 가지고 있다. 따라서, 알고리즘을 구현하기 위해 코드를 수정하고 build 하였을 때 기존 시스템과 충돌하는 경우가 많았고, 기존 설계하였던 기밀성과 무결성을 동시에 제공하는 방법은 복잡한 구현을 요구하기 때문에 통신 level 이외에도 고려해야 할 점이 많아 무결성을 우선적으로 강화하는 방안으로 고안하였다.

- **연산 리소스 제한**

  패킷의 payload 를 암호화하여 전송하는 방법은 불필요한 데이터가 추가될 수 있음을 의미한다. 기존 적용시키고자 하였던 AES 알고리즘을 고려하였을 때 10bytes 내외의 짧은 데이터를 가진 패킷을 암호화하기 위해서는 padding 을 추가하게 되어 암호화된 데이터는 길이가 증가하며, 이는 데이터 전송량을 불필요하게 증가시킬 수 있다. 따라서, 경량화가 중요한 드론 시스템에서 이와 같은 부하를 최소화할 수 있는 방법을 고려하여 서명 기능을 우선적으로 구현하고자 하였다.

2. 설계 상세화 및 변경 내역

2.1. 연구 환경

- Ubuntu 22.04
- PX4: v1.14.3
- QGC: v4.3
- Gazebo
- MAVROS

2.2. 설계 상세화

PX4 드론과 GCS(Ground Control Station) 간의 통신에서 전달되는 MAVLink 메시지에 대한 보안에 초점을 맞추어 연구를 진행한다. MAVLink 는 드론 통신을 위한 경량화 메시지 프로토콜로, 두 가지의 버전으로 나뉜다. 그 중에서 버전 2 는 버전 1 에 비해 더 많은 메시지를 정의할 수 있도록 해주고, signature 필드를 추가적으로 제공한다. MAVLink 버전 2 의 프레임은 다음과 같이 구성된다.

[그림 1] MAVLink v2 데이터 구조

2.2.1. Signature 필드

기존의 PX4 Autopilot 에서는 MAVLink v2 의 signature 필드를 사용하지 않고 있다. 대신 checksum 필드를 이용하여 데이터의 전송 중의 오류를 검사하고 있는데, 이는 데이터의 무결성을 보장해주지 못한다. CRC 는 공격자의 의도적인 데이터 조작을 탐지하는 데 적합하지 않으며, 공격자가 데이터를 조작하면서 CRC 값을 동일하게 변경하는 것이 어렵지 않다. 해당 취약점으로 인하여 보안적인 측면에서 CRC 만을 이용하는 것은 적합하지 않다. 따라서 signature 필드를 활성화하여 서명과 인증 기능을 추가하여 PX4 와 GCS 간의 통신에서 데이터 무결성을 향상시키고자 한다.

Signature 필드를 생성하기 위해서는 키가 필요하다. 따라서 어떤 방식으로 키를 관리해야 할지 결정해야 하고, 초기에는 사전 공유 방식을 이용하려고 했으나 MAVLink 프로토콜의 특성상 이러한 방식은 키에 대한 보안을 보장하지 못 한다. 그러므로 PX4 와 GCS 양측 모두의 보안 수준을 위하여 키 교환 알고리즘을 적용하고, 일정한 주기마다 키가 갱신될 수 있도록 할 것이다. 여기서 키 교환 알고리즘의 최적화 및 경량화를 통해 드론의 성능을 어느 정도 보장할 수 있다. 드론에서의 성능은 비행 환경 변화에 적시에 대응하도록 하거나, 비행 안정성을 보여주는 지표이므로 중요한 요인이라고 할 수 있다. 따라서, 키 교환 알고리즘의 적용과 동시에 응답 시간과 같은 성능의 측정도 함께 진행하도록 한다.

2.2.2. 키 교환 알고리즘

signature 필드를 생성하기 위하여 필요한 키를 관리하기 위하여 공개키 암호 알고리즘과 키 교환 알고리즘에 대해 분석하였다.

알고리즘	설명	문제점	속도(순위)
Diffie-Hellman	두 명의 사용자가 공개적으로 키를 교환하고, 그 키를 사용하여 공유 비밀키를 생성할 수 있는 방법	-중간자 공격에 취약함 -매우 긴 키 요구됨	4
RSA	공개 키 암호화를 사용하여 키를 교환하는 방법으로, 보안성이 높음	-매우 긴 키 요구됨	3
ECC	타원 곡선을 기반의 공개키 알고리즘으로, RSA 보다 더 짧은 키 길이로 높은 보안성을 제공	-복잡한 구현	2
ECDH	타원 곡선 기반에서 Diffie-Hellman 알고리즘을 적용한 방식으로, 기존 알고리즘에 비해 더 짧은 키로 더 나은 수준의 보안성 제공	-중간자 공격에 취약함 -복잡한 구현	1

[표 1] PX4 에서 사용하기 위한 암호 알고리즘 정리

[표 1]의 4 가지 알고리즘들이 px4 의 signature 필드 생성을 위한 알고리즘의 후보였다. 최우선적인 고려사항은 적용 이후의 드론의 성능이 가장 적게 저하될 수 있는 알고리즘이었고, 해당 이유에서 오버헤드가 가장 적고 연산이 가장 빠른 ECDH(Elliptic Curve Diffie-Hellman) 알고리즘을 선택하기로 하였다. 이 후엔 ECDH 알고리즘을 사용하여 PX4 와 GCS 사이의 공유 비밀키를 효과적으로 생성하고 이 키를 사용하여 signature 필드를 생성하게 될 것이다.

2.2.3. ECDH

ECDH 알고리즘은 타원 곡선 암호학을 기반으로 하는 Diffie-Hellman 키 교환 알고리즘이다. ECDH 는 통신에 참여하는 당사자들 간에 안전한 방식으로 공유 비밀 키를 생성할 수 있도록 해준다. 효율성 면에서도 RSA 와 같은 알고리즘에 비해 월등한 성능을 가지는데, 2048 비트 키의 RSA 의 보안 수준을 ECC 에서는 224 비트 키만으로도 제공할 수 있다. 따라서 모바일이나 스마트카드, PDA 등 제한된 리소스를 가지는 환경에서 유용하게 사용될 수 있다.

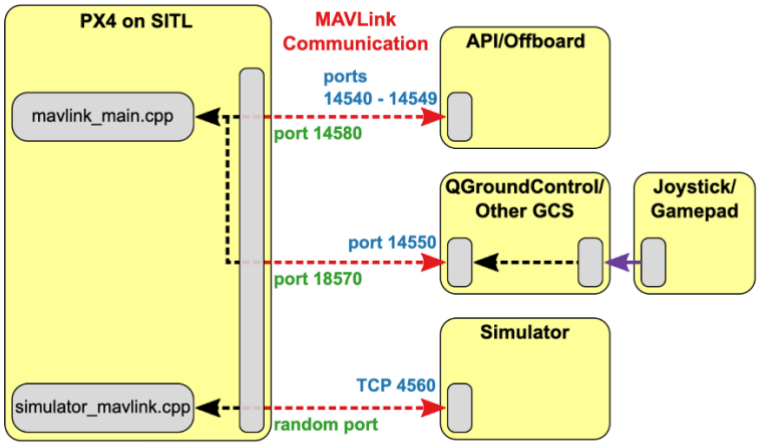
ECDH 는 ECC(Elliptic Curve Cryptography)를 기반으로 하여, 타원 곡선이라는 수학적 구조를 사용하여 암호화 알고리즘을 구현한다. ECDH 알고리즘을 직접 구현하기란 매우 복잡한 일이지만, 여러 암호화 라이브러리에서 ECDH 알고리즘인 Curve25519 함수를 제공한다.

Curve25519 는 ECDH 키 교환 알고리즘에서 매우 흔하게 사용되는 타원 곡선 방정식으로, 전통적인 방정식이 아닌 Montgomery 형태의 타원 곡선 방정식을 사용한다.

$$y^2 = x^3 + 486662x^2 + x$$

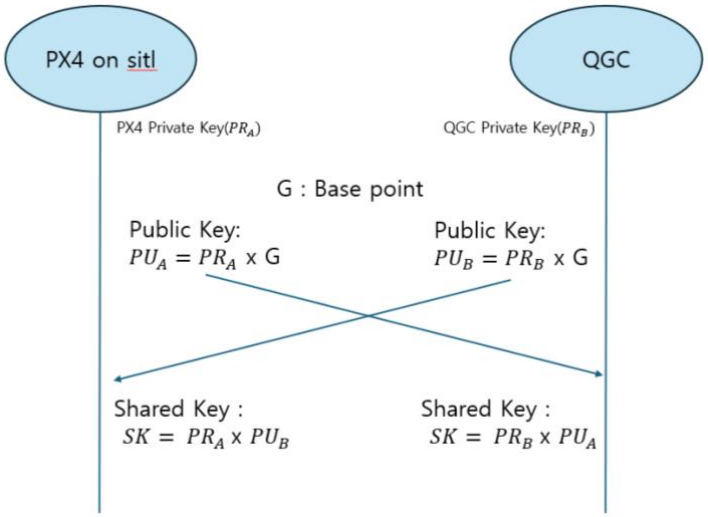
Montgomery 곡선이란, $By^2 = x^3 + Ax^2 + x$ 의 형태를 기본으로 가지는 곡선으로, 타원 곡선 상의 Scalar 곱셈과 같은 연산을 최적화하는 데 중점을 둔 수학적 표현이다. Curve25519 는 소수 $2^{255} - 19$로 정의된 소수체 위에서 동작하고, Montgomery Ladder 라는 알고리즘을 사용하여 스칼라 곱셈을 일정한 시간 내에 안전하게 수행할 수 있다.

2.2.4. ECDH on MAVLink



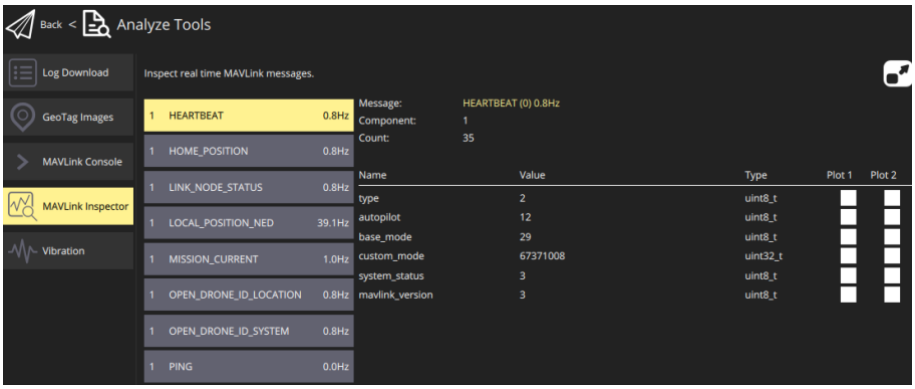
[그림 2] SITL 환경에서의 MAVLink 통신 구조

[그림 2]는 SITL(Software In The Loop) 환경에서 PX4 와 Offboard, GCS, Simulator 가 MAVLink 프로토콜로 통신하게 되는 구조를 보여준다. PX4 가 SITL 모드로 실행되면 MAVLink 인스턴스가 초기화되며 주기적으로 MAVLink 메시지를 전송하기 시작한다. QGC 가 PX4 에서 보낸 Heartbeat 메시지를 수신하면, 해당 메시지를 기반으로 PX4 에 연결되었음을 인식하고, 연결 상태를 표시한다. 연결이 성공하면 QGC 는 PX4 로부터 실시간으로 상태 데이터 등을 수신할 수 있다. 연결이 성공한 이후에도 PX4 와 QGC 는 지속적으로 Heartbeat 메시지를 주고받으며, 연결 상태를 유지한다.



[그림 3] PX4 와 QGC 사이에서의 키 교환 흐름

PX4 와 QGC 사이에서의 키 교환의 대략적인 흐름은 [그림 3]과 같다. 여기서의 논점은 MAVLink 프로토콜에서 어떤 방식으로 ECDH 알고리즘을 사용하느냐이다. 현재 계획 중인 설계로는 두 가지 방향이 있다. 첫 번째는 MAVLink 에 기존에 존재하지 않는 새로운 메시지를 만들어 timestamp 등과 같은 데이터를 포함해 공개키를 전송하는 것이고, 두 번째는 Heartbeat 메시지 구조를 수정하여 공개키 데이터를 포함하게 하여서 서로에게 전송하는 방법이다. [그림 4]를 살펴보면 HEARTBEAT 라는 단어 옆 쪽에 0.8Hz 라고 쓰여 있는 것을 확인할 수 있다. 해당 숫자는 Heartbeat 메시지의 주기로 본 메시지가 1 초에 0.8 번 전송된다는 것을 의미한다. Heartbeat 메시지에 공개키 데이터를 포함하게 하는 방법은 주기적인 공유 비밀 키의 갱신을 가능하게 하여 보안 수준이 향상 될 수 있지만, 추후 성능 측정을 통해 많은 오버헤드를 발생시킨다면 첫 번째 방법 혹은 더 긴 주기를 가지는 메시지를 이용하는 방법 등 다른 방법을 탐색해보아야 할 것 이다.



[그림 4] Heartbeat 메시지 정보

2.2.5. 서명 및 인증

키 교환이 정상적으로 이루어지게 되면, 해당 공유 키를 사용하여 signature 필드를 생성하게 된다. Signature 필드는 모든 메시지에 탑재되어 전송되게 되고, 수신 측에서는 공유 키를 이용하여 signature 필드를 검증하여서 데이터의 무결성을 검증한다.

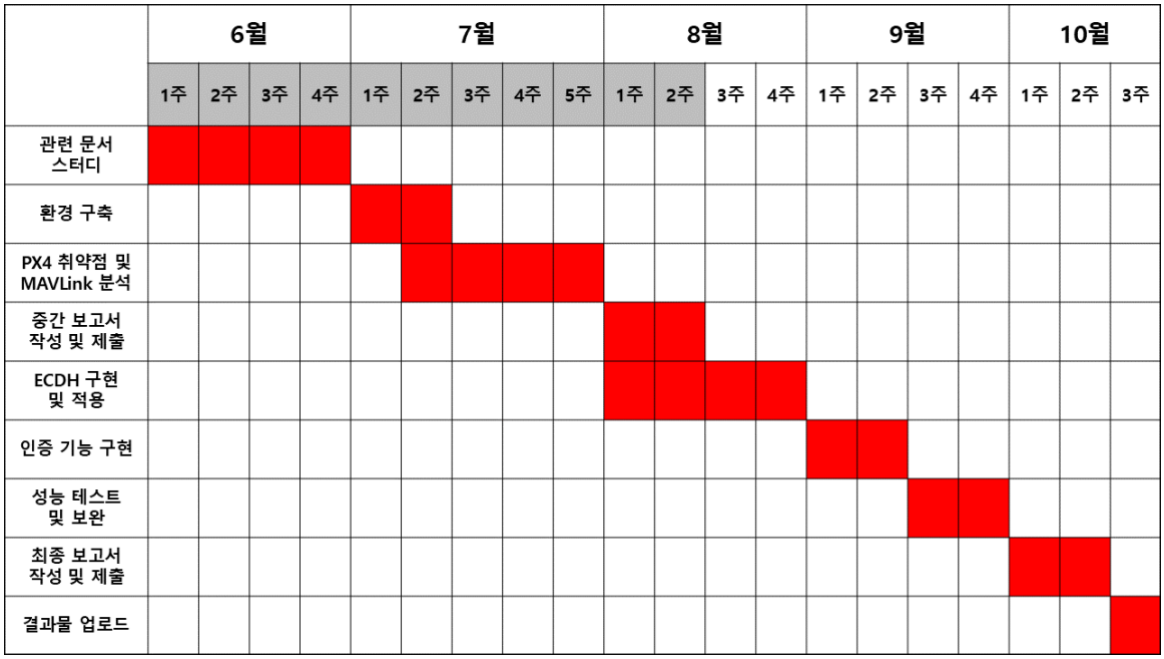
2.3. 변경 내역

이전의 설계에서는 기존의 방어 기법에 최적화 및 경량화를 적용하는 것으로 범위를 넓게 설정했으나, 이후 MAVLink 메시지의 signature 와, 키 교환 알고리즘 최적화 및 경량화에 대한 연구로 범위를 좁혔다. 이에 따라 다음의 연구를 진행할 수 있었다.

먼저, MAVLink 에 대한 전반적인 이해를 위해 MAVLink 공식 문서와, "common.xml" 파일에 정의되어 있는 메시지 및 명령어에 대한 학습을 진행하였다. 이후 직접 커스텀 메시지를 생성하여 전송해 보거나, 기존에 정의되어 있는 미션을 등록하여 시뮬레이션에서 자동으로 미션을 수행하도록 하는 연구도 진행해 보았다. 또한, Wireshark 에서 MAVLink 메시지에 대한 패킷 분석으로 전체 데이터를 확인할 수 있었다.

다음으로, 여러 키 교환 알고리즘에 대하여 작동 원리, 연산 속도 등을 조사하고, 드론에 최적화된 알고리즘에 대해 고려해 보았다. 이후 드론 경량화를 위하여 라이브러리를 이용하지 않고, 직접 키 교환 알고리즘을 작성해 보았다. 또한, 작성한 코드의 적용을 위하여 PX4 의 파일 구조와 소스 코드를 분석하며 전체 구조와 MAVLink 메시지 흐름을 파악하는 연구도 진행하였다.

3. 갱신된 과제 추진 계획



4. 구성원별 진척도

공통 : PX4 취약점 및 시스템 코드 분석

이경민 : MAVLink 메시지 송수신 코드 분석

Signature 필드 분석 및 구현

ECDH 알고리즘 분석 및 구현

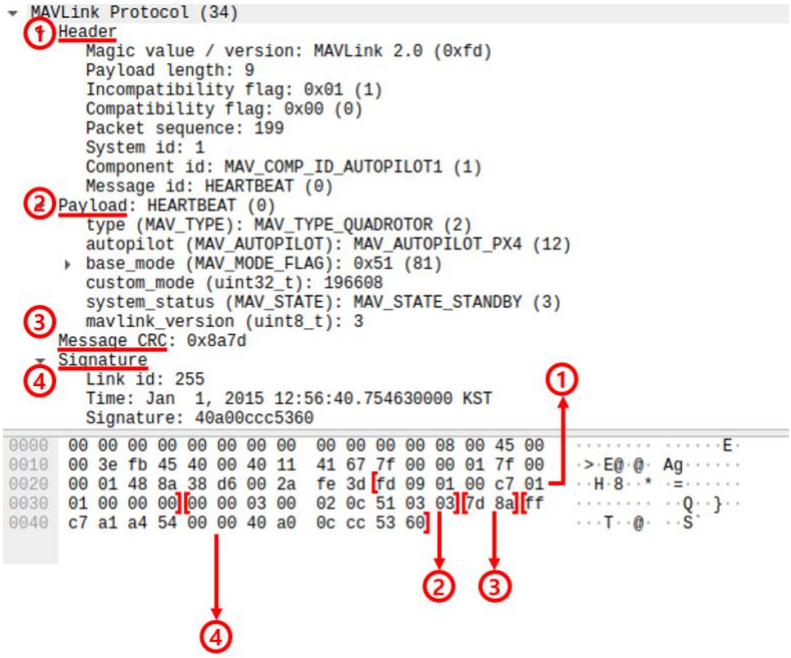
조수현 : MAVLink custom message 구현 및 테스트

MAVLink 메시지 패킷 구조 분석

5. 보고 시점까지의 과제 수행 내용 및 중간 결과

5.1. Wireshark로 MAVLink 메시지 패킷 확인

Wireshark로 MAVLink 메시지의 패킷 정보를 확인해 보았을 때, 프로토콜은 크게 Header, Payload, Message CRC, signature로 이루어져 있음을 알 수 있다.



[그림 5] Wireshark로 MAVLink 메시지 패킷 확인

5.1.1. Header

Header 부분에는 프로토콜의 버전, payload의 길이, 플래그, 패킷의 sequence, 그리고 system, component, message의 id 정보를 담고 있다.

5.1.2. Payload

Payload는 MAVLink 메시지의 데이터 부분으로, "Message id" 값에 따라 전체 구조가 달라진다. mission이나 command의 경우, 각 Parameter가 어떤 값을 나타내는지를 "command의 id" 값이 결정하기도 한다. 크기는 0 byte에서 255 byte로 다양하다.

5.1.3. Message CRC

데이터를 전송할 때 전송된 데이터에 오류가 있는지를 확인하기 위한 체크값으로, 총 2 byte의 크기를 가진다.

5.1.4. Signature

데이터 무결성을 보장하여, 위조 방지의 역할을 한다. MAVLink 버전 1에는 존재하지 않고, 버전 2에서만 존재한다.

5.2. MAVROS를 이용한 MAVLink 메시지 전송

드론의 조종과 관련된 MAVLink 메시지 전송을 통해 드론과 GCS 간의 통신, 그리고 MAVLink에 대한 이해를 높였다. 드론 조종 메시지 전송을 위하여 MISSION 프로토콜, COMMAND_LONG 프로토콜의 두 가지 프로토콜을 이용하였다.

5.2.1. MISSION 프로토콜

5.2.1.1. 개요

MISSION 프로토콜은 미션 업로드, 다운로드, 제거 등을 수행하는 작업을 포함하는 프로토콜로, 대부분의 MAVLink가 해당 프로토콜을 이용한다.

5.2.1.2. 명령어 실행

명령어의 종류에 따라, Parameter에 들어갈 값이 달라진다. "common.xml" 파일에 정의되어 있는 명령어들, 즉 사전에 MAVLink에서 정의한 명령어들을 직접 실행해 보았다. 특정 waypoint로 드론을 이동하도록 하는 명령어인 "MAV_CMD_NAV_WAYPOINT" (command 16번) 명령어를 예시로 들면, "common.xml" 파일에서 7개의 Parameter는 각각 Hold, Accept Radius, Pass Radius, Yaw, Latitude, Longitude, Altitude로 정의되어 있다. 해당 Parameter에 유의하여 명령어를 작성하면 다음과 같다.

첫 번째로, 미션을 등록한다.

```
rosservice call /mavros/mission/push "waypoints: [

{frame: 3, command: 16, is_current: true, autocontinue: true,

param1: 0.0, param2: 0.0, param3: 0.0, param4: 0.0,

x_lat: 47.4, y_long: 8.54, z_alt: 50.0}]"
```

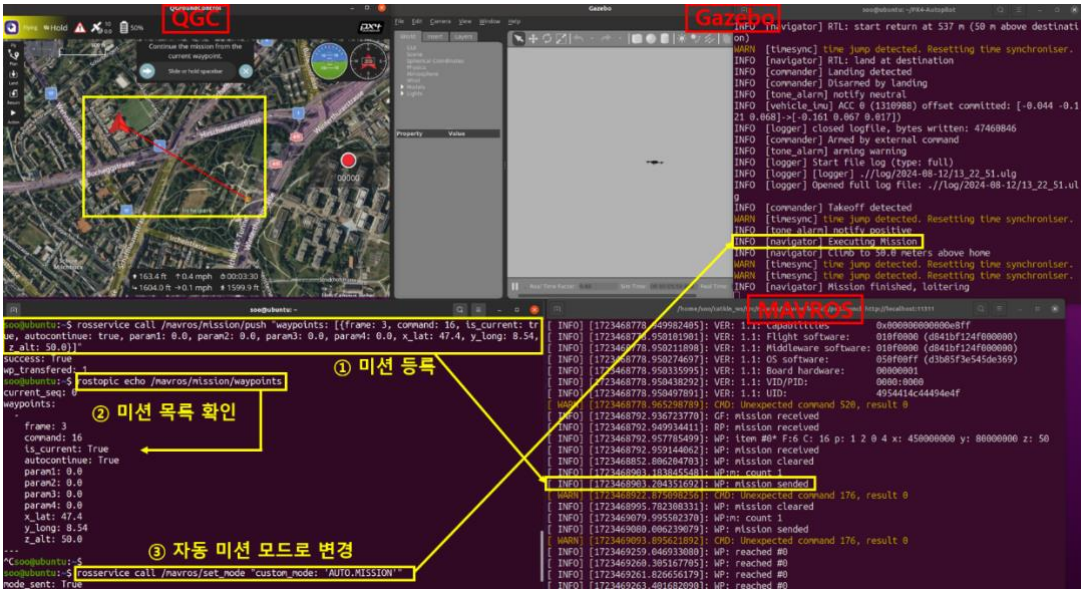
두 번째로, 미션의 목록을 확인한다.

```
rostopic echo /mavros/mission/waypoints
```

세 번째로, 비행 모드를 "자동 미션 모드"로 설정하여, 미션을 수행하도록 한다.

```
rosservice call /mavros/set_mode "custom_mode: 'AUTO.MISSION'"
```

위의 세 명령어를 순서대로 실행하면, 다음의 결과를 확인할 수 있다.



[그림 6] 미션 등록 및 수행 결과

QGC와 Gazebo에서는 미션 수행이 반영되어 지정한 waypoint로 비행하는 것을 확인했고, MAVROS에서는 미션을 등록하는 과정을 매개하고 있음을 확인하였다.

또한, 기존에 등록되어 있던 미션들을 모두 삭제할 수도 있다.

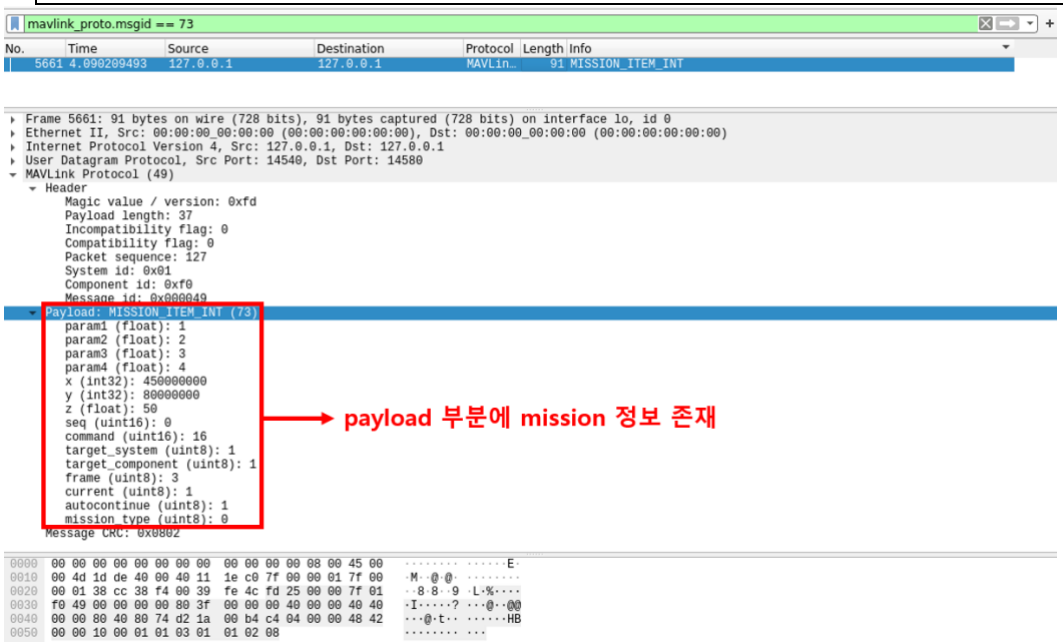
```
rosservice call /mavros/mission/clear "{}"
```

5.2.1.3. Wireshark에서 메시지 패킷 확인

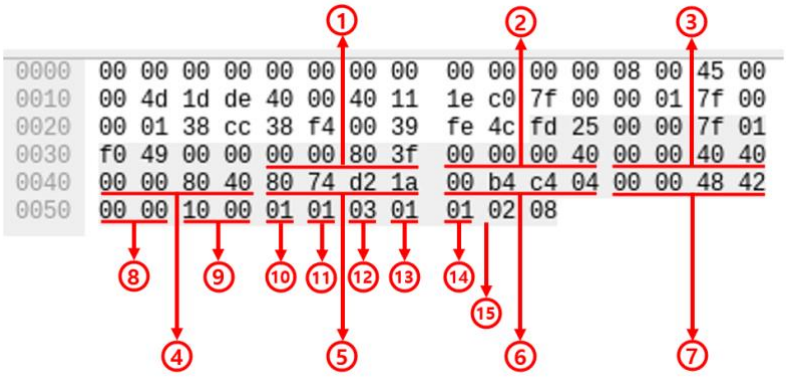
Mission 프로토콜의 경우, 메시지 "MISSION_ITEM_INT" (message 73번)에서 메시지 패킷 정보를 확인할 수 있다. 따라서, 따라서, Wireshark에서 필터를 "mavlink_proto.msgid == 73"으로 지정해야 한다.

아래의 command 16번에 대한 미션을 등록하도록 하는 명령어를 입력하고, Wireshark에서 패킷을 확인해보면 다음과 같다.

```
rosservice call /mavros/mission/push

"waypoints: [{

frame: 3,

command: 16,

is_current: true,

autocontinue: true,

param1: 1.0, param2: 2.0, param3: 3.0, param4: 4.0,

x_lat: 45.0, y_long: 8.0, z_alt: 50.0}]"
```



[그림 7] MAVLink의 Payload 영역에서 미션 정보 확인



[그림 8] MAVLink의 Payload 영역에서 미션 정보 분석

각 데이터는 "Little endian"으로 저장되고, 각 번호에 대한 설명은 다음과 같다.

- 1~4: param 1~4 (float)
- 5~7: 각각 x_lat (int32_t), y_long(int32_t), z_alt(float)
  - x_lat(45.0)에 10^7을 곱한 값을 16진수로 변환 → Little endian으로 "80 74 d2 1a"
  - y_long(8.0)에 10^7을 곱한 값을 16진수로 변환 → Little endian으로 "00 b4 c4 04"
  - 이렇게 10^7 만큼 곱해주는 이유는, float로 입력된 값을 패킷에서는 int로 변환해서 전송하기 위함이다.
- 8: sequence number (uint16_t)
- 9: command id (uint16_t)
- 10: target_system (uint8_t)
- 11: target_component (uint8_t)
- 12: frame (uint8_t)
- 13: is_current (uint8_t)
- 14: autocontinue (uint8_t)
- 15: mission type (uint8_t) (생략됨)

5.2.2. COMMAND 프로토콜

5.2.2.1. 개요

MAVLink의 command 전달을 보장하는 프로토콜로, 전송을 위해 COMMAND_INT 또는 COMMAND_LONG으로 인코딩될 수 있다. 모든 Parameter를 float type으로 입력하고자 COMMAND_LONG 프로토콜을 이용하였다.

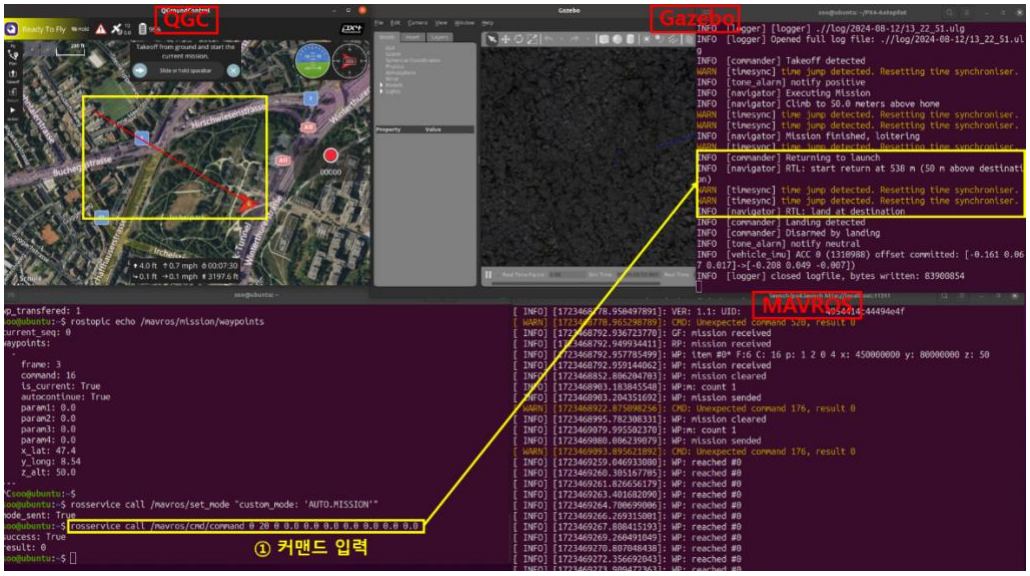
5.2.2.2. 명령어 실행

catkin_ws/src/mavros/mavros_msgs/srv/CommandLong.srv 파일의 구조체를 확인해 보았을 때, uint16 command, uint8 confirmation, float32 param1~7의 Parameter가 존재했다. 이에 맞게 시작 지점으로 돌아오게 하는 명령어인 "RETURN_TO_LAUNCH" (command 20번)를 수행하도록 하는 명령어를 작성하면 다음과 같다.

```
rosservice call /mavros/cmd/command 0 20 0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

위 명령어의 숫자 부분은 순서대로 broadcast, command id, confirmation, param1, param2, ..., param7 이다.

위의 명령어를 실행하면, 다음의 결과를 확인할 수 있다.



[그림 9] 커맨드 수행 결과

QGC와 Gazebo에서 커맨드 수행이 반영되어, 시작 지점으로 돌아와 착륙하는 것을 확인하였다.

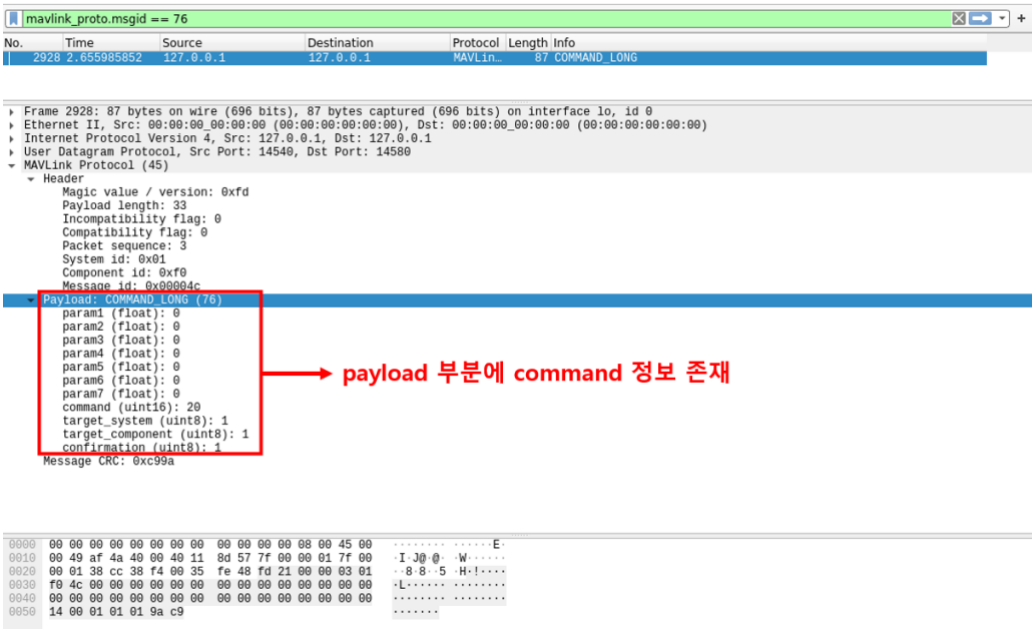
5.2.2.3. Wireshark에서 메시지 패킷 확인

COMMAND_LONG 프로토콜의 경우, 메시지 "COMMAND_LONG" (message 76번)에서 메시지 패킷 정보를 확인할 수 있다. 따라서, Wireshark에서 필터를 "mavlink_proto.msgid == 76"으로 지정해야 한다.

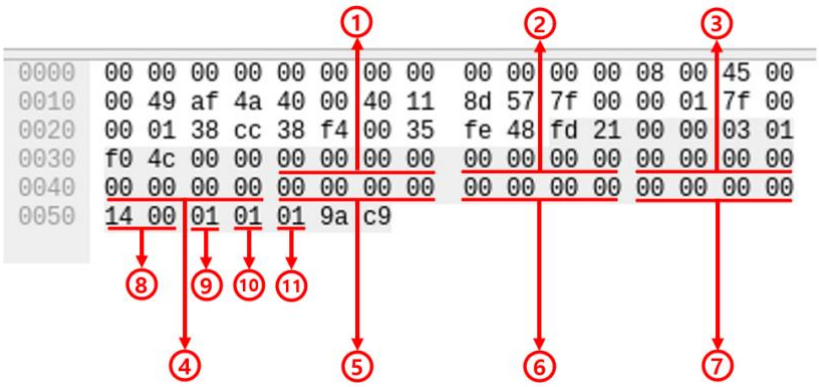
아래의 command 20번을 수행하도록 하는 명령어를 입력하고, Wireshark에서 패킷을

확인해보면 다음과 같다.

```
rosservice call /mavros/cmd/command 0 20 1 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```



[그림 10] MAVLink의 Payload 영역에서 커맨드 정보 확인



[그림 11] MAVLink의 Payload 영역에서 커맨드 정보 분석

각 데이터는 "Little endian"으로 저장되고, 각 번호에 대한 설명은 다음과 같다.

- Payload의 데이터를 "Little endian"으로 저장한다.
- 1~7: param 1~7 (float)
- 8: command id (uint16_t)
- 9: target_system (uint8_t)
- 10: target_component (uint8_t)
- 11: confirmation (uint8_t)

5.3. Monocypher

```c
void crypto_x25519_public_key(u8       public_key[32],
                              const u8 secret_key[32])
{
    static const u8 base_point[32] = {9};
    crypto_x25519(public_key, secret_key, base_point);
}
```

```c
void crypto_x25519(u8       raw_shared_secret[32],
                   const u8 your_secret_key  [32],
                   const u8 their_public_key [32])
{
    // restrict the possible scalar values
    u8 e[32];
    crypto_eddsa_trim_scalar(e, your_secret_key);
    scalarmult(raw_shared_secret, e, their_public_key, 255);
    WIPE_BUFFER(e);
}
```

[그림 12] 키 생성 코드

```c
void crypto_eddsa_trim_scalar(u8 out[32], const u8 in[32])
{
    COPY(out, in, 32);
    out[ 0] &= 248;
    out[31] &= 127;
    out[31] |= 64;
}

static void scalarmult(u8 q[32], const u8 scalar[32], const u8 p[32], int nb_bits) {
    fe x1;
    fe_frombytes(x1, p);

    fe x2, z2, x3, z3, t0, t1;

    fe_1(x2);
    fe_0(x2);
    fe_copy(x3, x1);
    fe_1(z3);
    int swap = 0;
    for(int pos = nb_bits-1; pos >= 0; --pos) {
        int b = scalar_bit(scalar, pos);
        swap ^= b;
        fe_cswap(x2, x3, swap);
        fe_cswap(z2, z3, swap);
        swap = b;

        fe_sub(t0, x3, z3);
        fe_sub(t1, x2, z2);
        fe_add(x2, x2, z2);
        fe_add(z2, x3, z3);
        fe_mul(z3, t0, x2);
        fe_mul(z2, z2, t1);
        fe_sq (t0, t1    );
        fe_sq (t1, x2    );
        fe_add(x3, z3, z2);
        fe_sub(z2, z3, z2);
        fe_mul(x2, t1, t0);
        fe_sub(t1, t1, t0);
        fe_sq (z2, z2    );
        fe_mul_small(z3, t1, 121666);
        fe_sq (x3, x3    );
        fe_add(t0, t0, z3);
        fe_mul(z3, x1, z2);
        fe_mul(z2, t1, t0);
    }

    fe_cswap(x2, x3, swap);
    fe_cswap(z2, z3, swap);

    fe_invert(z2, z2);
    fe_mul(x2, x2, z2);
    fe_tobytes(q, x2);
}
```

[그림 13] scalar 곱연산 및 키 조정 함수

Monocypher 는 여러 암호 알고리즘을 제공하는 라이브러리이다. 해당 라이브러리는 Curve25519 를 사용할 수 있는 함수를 제공한다. Monocypher 는 외부 종속성이 없으며, 경량화 되어 있어 리소스가 제한된 다양한 환경에서 유용하게 사용할 수 있다.

[그림 12]와 [그림 13]의 코드를 이용하여 PX4 와 QGC 는 각각 키를 생성하게 된다. 공개키를 생성하기 위해서는 비밀키가 필요한데 아직 적절한 난수 생성 모델을 구현하지 못 하였다. 추후 난수 생성 모델을 구현하여 비밀키를 생성하게 되면, 비밀키를 이용하여 공개키를 생성할 수 있고, 2.2.4 절에 언급하였던 방식으로 공개키를 전송하게 되면 해당 키를 이용하여 공유 비밀키를 생성할 수 있게 된다.

```
if ((_mode != MAVLINK_MODE_ONBOARD) && broadcast_enabled() &&
    (!get_client_source_initialized() || !is_gcs_connected())) {

    if (!_broadcast_address_found) {
        find_broadcast_address();
    }

    if (_broadcast_address_found && _buf_fill > 0) {

        int bret = sendto(_socket_fd, _buf, _buf_fill, 0, (struct sockaddr *)&_bcast_addr, sizeof(_bcast_addr));

        if (bret <= 0) {
            if (!_broadcast_failed_warned) {
                PX4_ERR("sending broadcast failed, errno: %d: %s", errno, strerror(errno));
                _broadcast_failed_warned = true;
            }

        } else {
            _broadcast_failed_warned = false;
        }
    }
}
```

[그림 14] mavlink_main.cpp 내부 코드

공유 비밀키를 생성한 이후에는 이 키를 이용하여 signature 필드를 생성하고 메시지에 포함시킨다. 이 메시지를 [그림 14]의 코드를 이용하여 PX4 가 QGC 로 전송할 수 있다.