

오픈소스 드론에 대한 취약점 분석 및 방어 기술 개발



저자1 이경민

저자2 조수현

지도교수 손준영

목 차

1. 서론	1
1.1. 연구 배경	1
1.2. 기존 문제점	1
1.3. 연구 목표	2
2. 연구 배경	2
2.1. 배경 지식	2
2.1.1. PX4	2
2.1.2. QGC	3
2.1.3. MAVLink	3
2.1.4. ROS	3
2.2. 설계 상세	4
2.2.1. Signature 필드	4
2.2.2. ECDH	5
2.2.3. SHA256	6
2.3. 개발 환경	7
3. 연구 내용	8
3.1. 시스템 시나리오	8
3.2. 사용자 정의 메시지 송수신	9
3.2.1. ROS	9
3.2.2. PX4	14
3.3. 공유 비밀 키 생성	16
3.3.1. Monocypher	16
3.3.2. QGC	17
3.3.3. PX4	20
3.4. Signature 생성 및 검증	22
3.4.1. QGC	22
3.4.2. SHA256	26
3.4.3. PX4	27

3.5. 구성원별 역할	32
4. 연구 결과 분석 및 평가	33
4.1. 성능 측정	33
4.1.1. CPU & RAM 사용량	33
4.1.2. 메시지 처리 시간 성능 측정	35
4.2. 멘토 의견서 대응 방안	41
5. 결론 및 향후 연구 방향	41
6. 참고 문헌	42

1. 서론

1.1. 연구 배경

최근 드론, 즉 UAV(Unmanned Aerial Vehicle) 기술의 발전에 따라 UAV가 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 UAV의 취약점을 악용한 다양한 공격의 급증을 유발하고 있다. UAV는 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.

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

이러한 이유로 본 연구에서 오픈 소스 드론의 취약점을 보완하는 방법에 대해 분석하고 적용하고자 한다.

1.2. 기존 문제점

본 과제에서 사용할 드론은 PX4 Autopilot이다. PX4 드론은 GCS(Ground Control System)와 MAVLink 프로토콜을 이용해 통신한다. MAV는 Micro Air Vehicle의 약자로 이름에서 볼 수 있듯이 MAVLink는 소형 비행체의 통신을 위해 개발된 프로토콜이다.

MAVLink는 데이터의 무결성을 CRC를 통해 확인한다. CRC는 데이터를 전송할 때 전송된 데이터에 오류가 있는지를 확인하기 위한 checksum 값을 결정하는 방식이다. CRC를 이용하

면 데이터 전송 중 발생할 수 있는 오류를 검출할 수 있다. 하지만 CRC는 공격자에 의한 의도적인 데이터 변조를 검출하는 것은 불가능에 가까우며, 동일한 checksum 값을 갖는 악의적인 메시지를 생성할 수 있는 가능성 역시 존재한다. [1]

MAVLink 2.0 버전에서는 signature 기능을 도입하여 무결성을 보호하고자 하였다. 그러나 PX4에서는 해당 기능을 사용하지 않아 중간자 공격이나 메시지 변조 공격 등의 위협으로부터 취약하다.

1.3. 연구 목표

PX4와 QGC의 통신에서, 이전에 사용되지 않았던 MAVLink 2.0 버전의 signature 필드를 사용하여, 데이터의 무결성을 보강한다. 이때 ECDH 키 교환 알고리즘을 이용하여 shared secret key로 signature 필드를 생성함으로써, secret key를 직접 공유하지 않고도 PX4와 QGC 사이의 서명 검증이 가능해지도록 한다. 또한, 다른 QGC로부터의 전송된 조종 명령 관련 MAVLink 메시지에서 signature 필드가 활성화되지 않았거나 서명 검증에 실패한다면, 해당 명령을 무시하도록 하여 두 시스템 간 통신이 더욱 안정적으로 이루어지도록 한다. 이에 더해, 서명 기능을 추가한 후 오버헤드 측정을 통해 기존 시스템과의 성능을 비교하고, 이를 경량화 할 수 있는 방안을 모색한다.

2. 연구 배경

2.1. 배경 지식

2.1.1. PX4

PX4는 오픈 소스 자동 조종 시스템으로, 레이싱 드론, 운송용 드론, 자동차와 선박 등의 다양한 운송체에 적용하여 사용할 수 있다. 누구나 접근할 수 있는 오픈 소스이므로 연구 및 상업용 목적 등으로 활용될 수 있으며, 개발자의 입장에서 자유롭게 기능을 추가하거나 수정할 수 있다. PX4는 Modular Architecture를 가지고 있어 각 모듈에 대해 독립적으로 수정하거나 확장하기 용이하며, QGroundControl, Pixhawk 하드웨어, MAVLink 프로토콜, ROS 등과 통합하여 확장된 기능을 사용할 수 있다. [2]

2.1.2. QGC

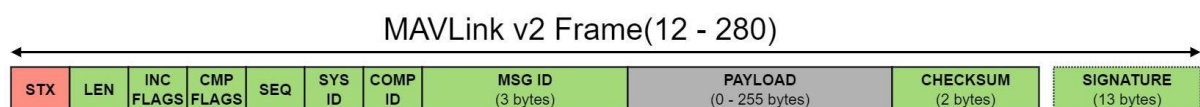
QGroundControl(QGC)는 Pixhawk를 기반으로 설계된 무인 비행장치에 대한 오픈 소스 지상관제소 프로그램이다. QGC는 PX4와 같은 자율 비행 시스템과 MAVLink 프로토콜을 통해 통신하며, 사용자가 드론의 비행 상태를 실시간으로 확인하고 제어 명령을 보낼 수 있는 직관적이고 사용자 친화적인 인터페이스를 제공한다. [3]

2.1.3. MAVLink

MAVLink(Micro Air Vehicle Link)는 드론 통신을 위한 매우 가벼운 메시징 프로토콜로, 직렬 통신 방식을 이용한다. MAVLink는 publish-subscribe 구조와 point-to-point 디자인 패턴을 따르고, 네트워크 상에서 최대 255개의 시스템까지 동시 적용이 가능하다.

MAVLink의 메시지는 XML 파일 내에서 정의된다. 각 XML 파일은 특정 MAVLink 시스템에서 지원하는 메시지 세트를 정의하며, 이를 "dialect"라고 한다. 대부분의 시스템에서 참조하는 메시지는 "common.xml" 파일 내에 정의되어 있다.

MAVLink 메시지는 버전 1과 버전 2로 나뉜다. 먼저, 버전 1은 각 패킷 당 8 bit의 오버헤드만을 가진다는 장점이 있는데, 버전 2와는 호환되지 않는다. 다음으로, 버전 2는 각 패킷 당 14 bit의 오버헤드를 가지지만, 훨씬 더 안전하고 새로운 필드의 확장이 가능하며, 버전 1과의 호환도 가능하다. 버전 2는 버전 1에 비해 flag 영역과 signature 영역이 추가되었는데, 패킷의 구조는 [그림 1]과 같다. [4]



[그림 1] MAVLink 버전 2의 패킷 구조

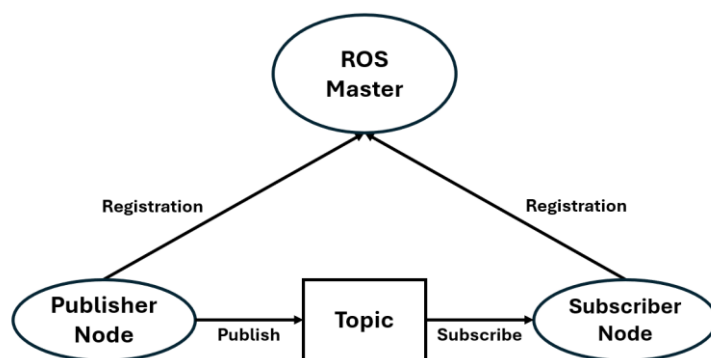
2.1.4. ROS

ROS(Robot Operating System)는 오픈 소스이자, 메타 운영체제로 스케줄링, 프로세스 관리 등을 가능하게 해준다. 또한, 프로세스 간 메시지 전달, 패키지 관리와 여러 컴퓨터 시스템에서 작동하는 코드를 실행하기 위한 도구 및 라이브러리를 제공한다.

ROS의 주된 특징 세 가지 중 첫 번째로는 코드 재사용성이 있는데, 필요한 부분만 개발하

고 다른 기능들은 다운로드 받아서 이용할 수 있다. 두 번째는, 하나의 서비스에 대해 목적에 따라 프로세서 중 최소 실행 단위인 "노드"로 나누는 통신 기반 프로그래밍으로, 원격 제어를 용이하게 할 수 있다는 것이다. 세 번째는, 하드웨어가 다르더라도 같은 소프트웨어 플랫폼을 이용하면 개발이 가능하듯, ROS가 하드웨어 기술을 통합할 수 있는 소프트웨어 플랫폼이 되어준다는 것이다.

ROS의 노드 간 통신 방식들 중에서, "토픽"을 이용하여 MAVLink 메시지를 전송할 수 있다. 토픽은 단방향 비동기 통신 방식을 이용하고, 연속적인 데이터 송수신이 가능하다. 토픽을 발표하는 Publisher 노드가 있고, 해당 토픽을 구독하는 Subscriber 노드가 있다. 이 두 노드는 ROS Master에 등록되어 관리되고, 관계도를 나타내면 다음과 같다.



[그림 2] 토픽을 이용한 ROS 노드 통신의 관계도

다음으로, MAVROS는 UAV에서 많이 이용되는 MAVLink 프로토콜을 ROS의 패키지로 만든 것으로, MAVLink를 사용하는 시스템들에 대하여 통신 드라이버를 제공한다. 또한, ROS의 토픽을 MAVLink 메시지로 변환하여 PX4로 전송하는 역할을 한다. [5][6]

2.2. 설계 상세

2.2.1. Signature 필드

1.2절에서 언급하였듯, MAVLink 2.0에서는 signature 필드를 도입하였지만 PX4에서는 이를 사용하고 있지 않다. 서명과 인증 기능은 무결성 보장 측면에서 중요한 역할을 하기 때문에 signature 필드의 활성화를 통해 서명과 인증 기능을 추가하여 데이터 무결성을 향상시킬 수

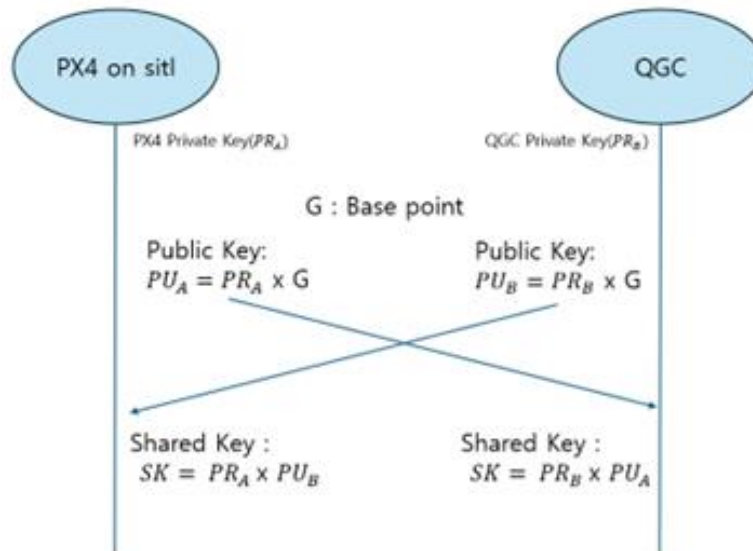
있도록 한다.

서명 기능을 사용하기 위해선 key가 필요하다. 따라서 어떤 방식으로 키를 관리해야 할지 결정해야 하고, 사전 공유 방식이나 중앙 집중식 키 관리 등 여러 방식을 선택할 수 있다. 본 연구에서는 MAVLink 통신에 키 교환 알고리즘을 적용하여 통신 당사자 간의 키를 관리할 수 있도록 하였고, 타원 곡선상의 Diffie-Hellman 알고리즘을 통하여 드론이라는 환경에 좀 더 적합한 방식으로 적용하였다.

또한, 서명을 어떤 알고리즘을 이용하여 생성할지에 대한 결정이 필요하다. 기존 설계 방안으로는 ECDSA, EDDSA, Poly1305, SHA256 등 여러 알고리즘을 적용한 뒤 각각의 성능을 측정하여 PX4에 가장 적합한 알고리즘을 찾는 것이었다. 하지만 기간 내에 모든 알고리즘을 적용하는 데 한계가 있어 SHA256 알고리즘만을 적용하여 서명을 생성하는 것으로 결정하였다.

2.2.2. ECDH

ECDH는 두 통신 참여자가 서로의 public key를 교환하여 공통된 공유 비밀 키를 생성하는 알고리즘이다. ECC(Elliptic Curve Cryptography)를 기반으로 하여, 타원 곡선이라는 수학적 구조를 사용하여 암호화 알고리즘을 구현한다.



[그림 3] PX4와 QGC 사이의 ECDH

PX4와 QGC에서의 ECDH 동작은 [그림 3]과 같다. 양측은 개인 키를 생성하고 개인 키를 통해 공개키를 생성한다. 이 후 서로 공개키를 교환하고 상대의 공개키와 자신의 개인 키를

이용하여 공유 비밀 키를 생성하게 된다. 따라서 양측은 서로의 개인키를 외부로 노출시키지 않고 안전하게 비밀키를 공유할 수 있다.

또한, 256비트 길이의 키만으로도 RSA의 3072비트 길이의 키와 같은 보안 수준을 제공할 수 있기 때문에 드론 같은 자원이 제한된 환경에서 유용하게 사용될 수 있다.

2.2.3. SHA256

SHA256은 SHA(Secure Hash Algorithm) 알고리즘의 한 종류로, output의 길이가 256 bit 또는 64자의 문자열이다. SHA256은 "SHA-2" 버전에 포함되고, 현재 인터넷 뱅킹, 비트코인 등의 분야에서 이용되고 있다. 단방향 알고리즘이기 때문에 복호화가 불가능하고, 입력의 종류와 길이에 상관없이 항상 고정된 길이의 출력을 낸다는 특징이 있다. 또한, 같은 데이터를 입력하면 언제나 같은 해시값을 얻고, 만약 데이터의 작은 부분만 변경되더라도 해시값은 크게 달라질 수 있다. SHA256의 알고리즘은 공개되어 있고, 누구나 활용할 수 있다.

MAVLink 메시지 버전 2에 존재하는 signature 필드가 SHA256 알고리즘을 이용하고 있다. MAVLink 라이브러리의 "mavlink_sha256.h" 파일 내에 SHA256 연산에 이용되는 총 4개의 함수가 정의되어 있다. [7] [그림 4]는 SHA256에서 사용되는 함수에 대한 선언 부분만을 나타낸 것이다.

```
MAVLINK_HELPER void mavlink_sha256_init(mavlink_sha256_ctx *m)
{
    ...
}

static inline void mavlink_sha256_calc(mavlink_sha256_ctx *m, uint32_t *in)
{
    ...
}

MAVLINK_HELPER void mavlink_sha256_update(mavlink_sha256_ctx *m, const void *v, uint32_t len)
{
    ...
}

/*
 * get first 48 bits of final sha256 hash
 */
MAVLINK_HELPER void mavlink_sha256_final_48(mavlink_sha256_ctx *m, uint8_t result[6])
{
    ...
}
```

[그림 4] MAVLink의 SHA256 알고리즘 함수

각 함수에 대하여 간단하게 설명하자면, 먼저 mavlink_sha256_init 함수는 해시 계산에 필요한 기본 값을 설정하는 함수이고, mavlink_sha256_calc 함수는 해시 계산을 수행하는 함수이다. 다음으로, mavlink_sha256_update 함수는 계산할 데이터를 나눈 뒤

mavlink_sha256_calc 함수로 해시 계산을 넘겨주는 역할을 한다. 마지막으로 mavlink_sha256_final_48 함수는 해시 계산을 마무리하고, 생성된 값의 가장 앞 48 bit만을 결과로 반환한다.

결과로 가장 가장 앞의 48 bit만을 반환하는 이유는, 더 작은 크기의 해시값을 이용함으로써 패킷의 크기를 줄임과 동시에 빠른 비교가 가능해지기 때문이다. 이는 MAVLink의 경량화 통신에서 자원을 더 효율적으로 활용할 수 있게 해준다. 해시값의 길이가 짧아짐에 따라 충돌의 우려가 생길 수 있는데, MAVLink는 상대적으로 적은 양의 데이터를 전송하기 때문에 2^{48} 개의 값으로도 낮은 충돌 확률을 제공할 수 있다.

코드 작성 시에는 mavlink_sha256_init, mavlink_sha256_update, mavlink_sha256_final_48 함수를 순서대로 적용한다. mavlink_sha256_calc 함수는 mavlink_sha256_update 함수 내에서 자동으로 호출되기 때문에 따로 적용하지 않아도 된다.

2.3. 개발 환경

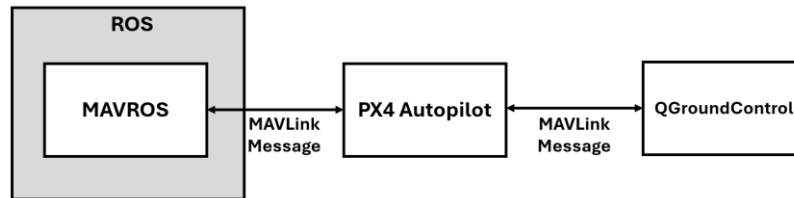
Ubuntu	20.04.06
PX4-Autopilot	main branch(1.14.3)
ROS	Noetic
QGC	4.4.1
Qt	6.6.3
Gazebo	11.0

[표 1] 개발 환경

3. 연구 내용

3.1. 시스템 시나리오

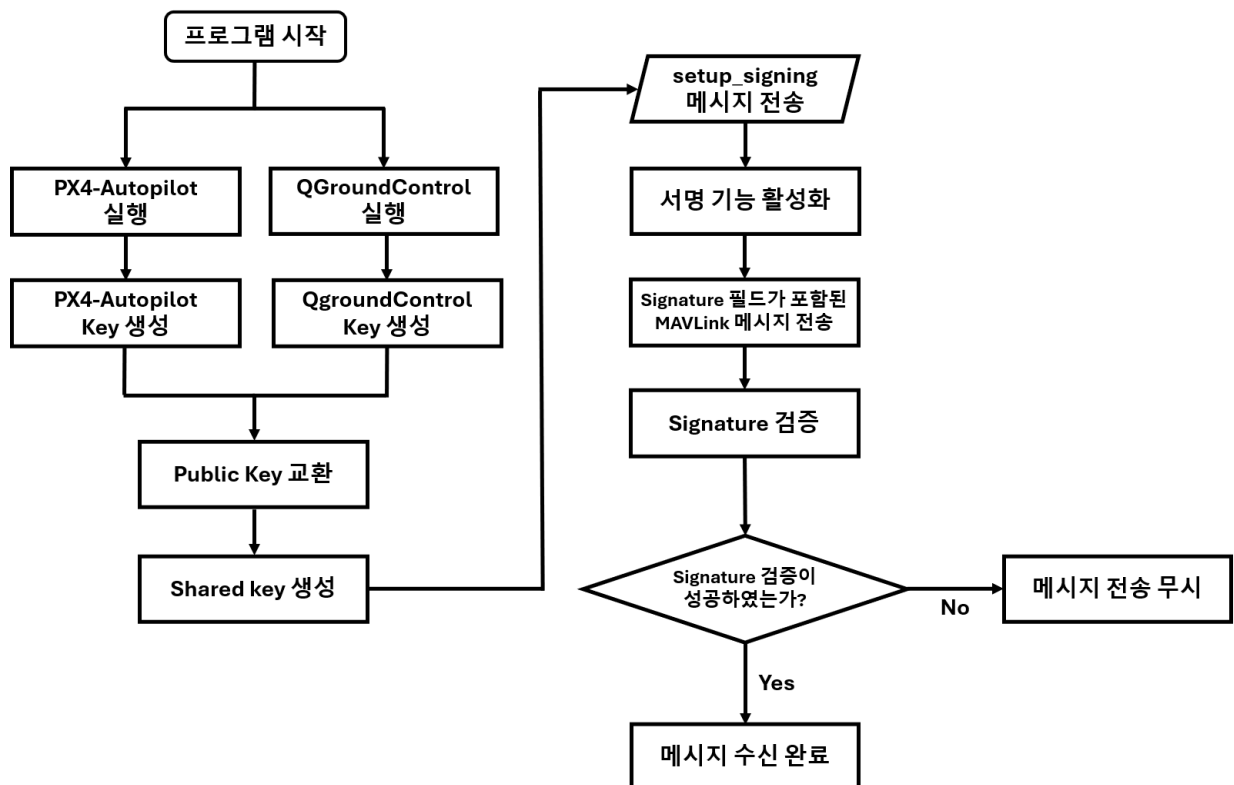
[그림 5]은 시스템의 전체 구성도를 나타낸 것이다.



[그림 5] 시스템의 전체 구성도

ROS의 패키지 중 하나인 MAVROS에서 ROS 토픽을 MAVLink 메시지로 변환하여, key 생성을 위한 메시지를 PX4로 전달한다. PX4와 QGC는 서로 MAVLink 메시지로 통신하며 필요한 정보를 주고받는다.

[그림 6]은 시스템의 전체 플로우 차트를 나타낸 것이다.



[그림 6] 시스템의 전체 플로우 차트

먼저, PX4와 ROS, MAVROS가 실행되면 PX4에서 secret key가 난수로 생성되고 동시에 secret key 기반으로 public key가 생성된다. QGC를 실행하면 PX4와의 연결을 통해 기체에 대한 객체가 생성되는 과정에서 PX4와 마찬가지로 secret key와 public key가 생성된다. 이후, 서로의 public key를 교환하게 되면 자신의 secret key와 상대방의 public key를 이용하여 공유 비밀 키를 생성하게 된다. 위 과정이 진행된 후, QGC에서 setup_signing 메시지를 전송하게 되면 서명 기능이 활성화되고, MAVLink 메시지에 signature 필드가 포함되어 전송된다. 메시지에 포함된 signature와 직접 계산한 signature를 비교하여 서명을 검증하는 HMAC을 적용하여 signature 검증에 성공한다면 메시지를 수신하게 되고, 검증에 실패한다면 메시지를 무시하게 된다.

3.2. 사용자 정의 메시지 송수신

QGC는 PX4와 연결되었을 때 생성자가 호출되어 public key를 생성하도록 구현되어 있다. PX4가 먼저 실행되고 QGC가 실행되기 이전에, QGC의 public key가 아직 생성되지 않은 상태이므로, PX4는 QGC의 public key 값 갱신에 대한 지속적인 모니터링이 필요하다. 이러한 모니터링 과정을 돕기 위해, MAVROS에서 PX4로 지속적으로 커스텀 메시지를 전송함으로써 PX4가 QGC의 public key를 확인하도록 하였다. 이와 동시에, PX4의 secret key 및 public key 생성도 이루어지게 하였다.

3.2.1. ROS

Ubuntu 20.04 환경에 맞는 "ROS Noetic"을 설치 및 환경을 구축한다.

```
# 1. ROS 패키지 저장소를 Ubuntu의 패키지 목록에 추가
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'

# 2. curl 설치 및 ROS 패키지의 GPG 키를 추가
$ sudo apt install curl
$ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key
add -

# 3. ROS Noetic 설치
$ sudo apt update
$ sudo apt install ros-noetic-desktop-full

# 4. 환경 변수 설정
$ echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

[표 2] ROS 개발 환경 구축

ROS 설치 및 환경 구축이 완료되었다면, 아래의 명령어를 통해 ROS Master를 실행할 수 있다.

```
$ roscore
```

[표 3] ROS Master 실행 명령어

실행 후, 아래의 화면을 확인할 수 있다.

```
soo@ubuntu:~$ roscore
... logging to /home/soo/.ros/log/f375c69a-83a2-11ef-97d0-3b6b9898f577/roslaunch-ubuntu-4991.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:46757/
ros_comm version 1.16.0

SUMMARY
=====
PARAMETERS
 * /roscpp: noetic
 * /rosversion: 1.16.0

NODES
auto-starting new master
process[master]: started with pid [5007]
ROS_MASTER_URI=http://ubuntu:11311/

setting /run_id to f375c69a-83a2-11ef-97d0-3b6b9898f577
process[rosout-1]: started with pid [5020]
started core service [/rosout]
```

[그림 7] roscore 실행 화면

다음으로, MAVROS에 대해 설치 및 환경을 구축한다.

```
# 1. 바이너리 설치
$ sudo apt install python3-catkin-tools python3-rosinstall-generator python3-osrf-
pycommon -y

# 2. 소스 설치
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws
$ catkin init
$ wstool init src

# 3. MAVLink 설치
$ rosinstall_generator --rosdistro kinetic mavlink | tee /tmp/mavros.rosinstall

# 4. MAVROS 설치
$ rosinstall_generator --upstream mavros | tee -a /tmp/mavros.rosinstall

# 5. 작업 공간과 의존성 만들기
$ cd catkin_ws
$ wstool merge -t src /tmp/mavros.rosinstall
$ wstool update -t src -j4
$ rosdep install --from-paths src --ignore-src -y

# 6. GeographicLib 데이터셋 설치
$ ./src/mavros/mavros/scripts/install_geographiclib_datasets.sh

# 7. PX4와의 UDP 통신 설정
catkin_ws/src/mavros/mavros/launch/px4.launch에서 "/dev/ttyACM0:57600" 부분을
"udp://:14540@127.0.0.1:14557"로 수정

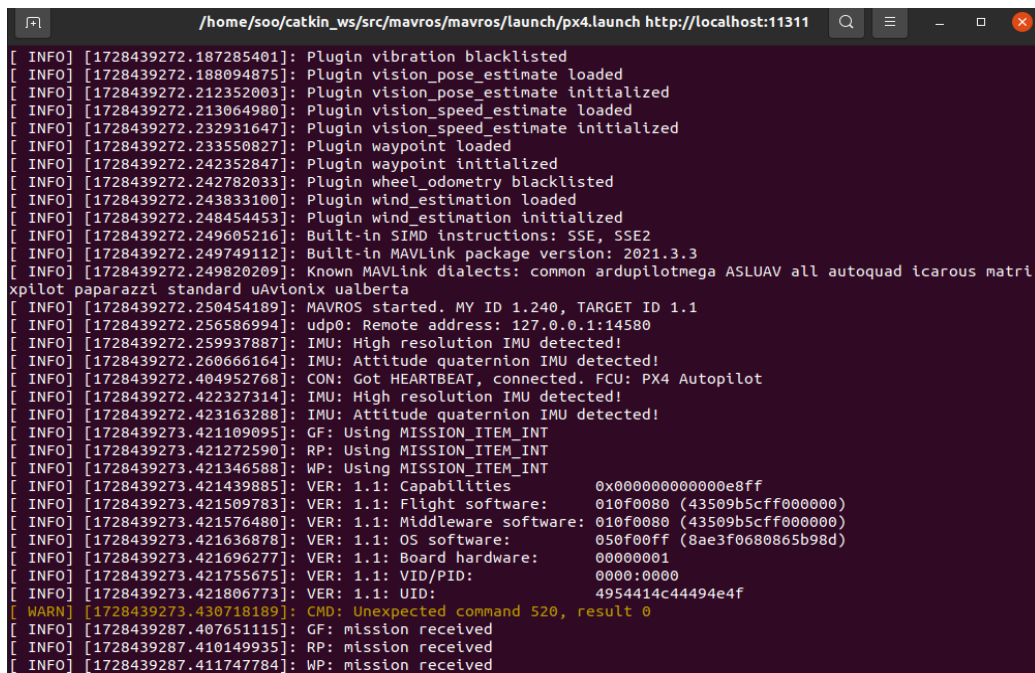
# 8. 소스 빌드
$ catkin build
```

[표 4] MAVROS 환경 구축

MAVROS 설치 및 환경 구축이 완료되었다면, catkin_ws 디렉토리에서 아래의 명령어를 통해 MAVROS를 실행할 수 있다.

```
$ export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$(pwd)
$ source devel/setup.bash
$ roslaunch mavros px4.launch
```

[표 5] MAVROS 실행 명령어



```
/home/soo/catkin_ws/src/mavros/mavros/launch/px4.launch http://localhost:11311
[ INFO] [1728439272.187285401]: Plugin vibration blacklisted
[ INFO] [1728439272.188094875]: Plugin vision_pose_estimate loaded
[ INFO] [1728439272.212352003]: Plugin vision_pose_estimate initialized
[ INFO] [1728439272.213064980]: Plugin vision_speed_estimate loaded
[ INFO] [1728439272.232931647]: Plugin vision_speed_estimate initialized
[ INFO] [1728439272.233550827]: Plugin waypoint loaded
[ INFO] [1728439272.242352847]: Plugin waypoint initialized
[ INFO] [1728439272.242782033]: Plugin wheel_odometry blacklisted
[ INFO] [1728439272.243833100]: Plugin wind_estimation loaded
[ INFO] [1728439272.248454453]: Plugin wind_estimation initialized
[ INFO] [1728439272.249605216]: Built-in SIMD instructions: SSE, SSE2
[ INFO] [1728439272.249749112]: Built-in MAVLink package version: 2021.3.3
[ INFO] [1728439272.249820209]: Known MAVLink dialects: common ardupilotmega ASLUAV all autoquad icarous matri
xpilot paparazzi standard uAvionix ualberta
[ INFO] [1728439272.250454189]: MAVROS started. MY ID 1.240, TARGET ID 1.1
[ INFO] [1728439272.256586994]: udp0: Remote address: 127.0.0.1:14580
[ INFO] [1728439272.259937887]: IMU: High resolution IMU detected!
[ INFO] [1728439272.260666164]: IMU: Attitude quaternion IMU detected!
[ INFO] [1728439272.404952768]: CON: Got HEARTBEAT, connected. FCU: PX4 Autopilot
[ INFO] [1728439272.422327314]: IMU: High resolution IMU detected!
[ INFO] [1728439272.423163288]: IMU: Attitude quaternion IMU detected!
[ INFO] [1728439273.421109095]: GF: Using MISSION_ITEM_INT
[ INFO] [1728439273.421272590]: RP: Using MISSION_ITEM_INT
[ INFO] [1728439273.421346588]: WP: Using MISSION_ITEM_INT
[ INFO] [1728439273.421439885]: VER: 1.1: Capabilities          0x000000000000e8ff
[ INFO] [1728439273.421509783]: VER: 1.1: Flight software:      010f0080 (43509b5cfff00000)
[ INFO] [1728439273.421576480]: VER: 1.1: Middleware software: 010f0080 (43509b5cfff00000)
[ INFO] [1728439273.421636878]: VER: 1.1: OS software:         050f00ff (8ae3f0680865b98d)
[ INFO] [1728439273.421696277]: VER: 1.1: Board hardware:      00000001
[ INFO] [1728439273.421755675]: VER: 1.1: VID/PID:            0000:0000
[ INFO] [1728439273.421806773]: VER: 1.1: UID:                4954414c44494e4f
[ WARN] [1728439273.430718189]: CMD: Unexpected command 520, result 0
[ INFO] [1728439287.407651115]: GF: mission received
[ INFO] [1728439287.410149935]: RP: mission received
[ INFO] [1728439287.411747784]: WP: mission received
```

[그림 8] MAVROS 실행 화면

다음으로, MAVROS에서의 플러그인 추가, 커스텀 메시지 정의 추가, uORB 메시지 파일 생성 등을 통해 PX4로 커스텀 메시지를 전송할 수 있게 된다.

먼저, common.xml 파일에 정의된 커스텀 메시지 key_exchange의 구조는 아래와 같다.

```
<message id="229" name="KEY_EXCHANGE">
  <description>Public key exchange</description>
  <field type="uint8_t[32]" name="public_key"> </field>
</message>
```

[그림 9] 커스텀 메시지 key_exchange의 구조

다음으로, "catkin_ws/src/mavros/mavros_extras/src/plugins" 디렉토리에서 새로 생성한 플러그인 파일 "key_exchange.cpp"에서 전송 시의 동작 및 주기 설정이 가능하다.

```
namespace mavros {
namespace extra_plugins {

class KeyExchangePlugin : public plugin::PluginBase {
public:
    KeyExchangePlugin() : PluginBase(),
        nh("~key_exchange")
    { };

    void initialize(UAS &uas_)
    {
        PluginBase::initialize(uas_);
        pub = nh.advertise<std_msgs::UInt8MultiArray>("key_exchange_pub", 10);
        timer = nh.createTimer(ros::Duration(10.0), &KeyExchangePlugin::send_key_exchange, this);
        ...
    }
};

}
```

[그림 10] key_exchange 플러그인 파일의 코드 일부

위의 코드에서, "Duration(10.0)" 부분을 통해 key_exchange 메시지가 10초마다 PX4로 전송되고, PX4는 key_exchange 메시지를 받아 주기적으로 QGC의 public key를 확인하게 된다. 아래의 [그림 11]은 ROS 메시지를 publish하고, MAVLink 메시지를 FCU(Flight Control Unit)로 전달하는 함수의 코드이다.

```
void send_key_exchange(const ros::TimerEvent&)
{
    mavlink::common::msg::KEY_EXCHANGE kc {};
    memcpy(kc.public_key.data(), public_key, KEY_SIZE);

    std_msgs::UInt8MultiArray ros_msg;
    ros_msg.data.clear();
    ros_msg.data.insert(ros_msg.data.end(), public_key, public_key + KEY_SIZE);

    // ROS 메시지 publish
    pub.publish(ros_msg);

    // FCU로 MAVLink 메시지 전송
    UAS_FCU(m_uas)->send_message_ignore_drop(kc);
}
```

[그림 11] key_exchange 플러그인 파일의 send_key_exchange 함수

해당 함수는 ROS 타이머 이벤트에 의해 주기적으로 실행되고, ROS 메시지에 대한 값을 초기화하고 할당하여 publish하고, MAVLink 메시지로 바꾸어 FCU로 전달한다.

3.2.2. PX4

3.2.1절에서 생성된 MAVROS의 key_exchange 메시지는 PX4로 전달되고, PX4는 메시지를 수신하면 자신의 secret key와 public key를 생성하고, shared secret key의 생성을 위해 QGC의 public key 값을 읽어 들인다.

secret key는 C++의 <random> 라이브러리를 통해 32바이트의 난수를 생성하게 되는데 C에서의 rand() 함수는 생성되는 난수열들의 상관관계가 높기 때문에 해당 라이브러리를 선택하게 되었다. <random> 라이브러리는 Mersenne Twister 알고리즘을 사용하는데 이 알고리즘은 생성해 내는 난수의 주기가 매우 크고, 난수 사이의 상관 관계가 매우 낮으며 빠른 속도를 낼 수 있다. 하지만, 난수의 특성(주기, 난수 범위)을 알고 있을 때 유한한 수의 난수만으로 현재 생성기의 상태를 알아낼 수 있고, 뒤의 난수를 예측해 낼 수 있는 암호학적으로 안전한 난수 생성기가 아니다. 따라서 현재 구현은 진행하지 않았지만, 생성된 난수에 hash 함수를 적용시키는 것으로 보안 수준을 높이는 것을 추후 연구 방향으로 계획하였다.

```
void key_generation(uint8_t *key) {
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_int_distribution<uint8_t> dis(0, 255);

    for(int i=0; i<32; i++) {
        key[i] = dis(gen);
    }
}
```

[그림 12] secret key 생성 함수

public key는 secret key를 기반으로 연산된다. Curve 25519 타원 곡선 상에서 연산을 진행하는데 이는 Monocypher 라이브러리를 사용하였다. PX4의 MAVLink 모듈에서 Monocypher를 사용할 수 있도록 설치를 진행하고 전체 디렉토리를 모듈의 디렉토리에 복사하여준다. 그런 다음 컴파일 시 이를 이용할 수 있도록 CMakeLists에 [그림 13]의 코드를 추가해 주었다. 이 과정이 완료되면 Monocypher의 crypto_x25519 관련 함수를 사용해 public key와 shared secret key를 생성할 수 있다.

```
set(MONOCYPHER_DIR ${CMAKE_CURRENT_SOURCE_DIR}/monocypher/src)
set(MONOCYPHER_SRC ${MONOCYPHER_DIR}/monocypher.c)
add_library(monocypher STATIC ${MONOCYPHER_SRC})
set_target_properties(monocypher PROPERTIES POSITION_INDEPENDENT_CODE ON)
target_include_directories(monocypher PUBLIC ${MONOCYPHER_DIR})
```

[그림 13] MAVLink 모듈의 CMakeLists.txt 수정

PX4에서는 수신한 MAVLink 메시지를 "mavlink_receiver.cpp" 파일 내에서 처리한다. 먼저, 아래의 handle_message 함수에서 MAVLink 메시지가 처리된다.

```
void
MavlinkReceiver::handle_message(mavlink_message_t *msg)
{
    switch (msg->msgid) {
        case MAVLINK_MSG_ID_COMMAND_LONG:
            handle_message_command_long(msg);
            break;

        case MAVLINK_MSG_ID_COMMAND_INT:
            handle_message_command_int(msg);
            break;

        case MAVLINK_MSG_ID_COMMAND_ACK:
            handle_message_command_ack(msg);
            break;

        ...

        case MAVLINK_MSG_ID_SET_MODE:
            handle_message_set_mode(msg);
            break;

        ...

        case MAVLINK_MSG_ID_HEARTBEAT:
            handle_message_heartbeat(msg);
            break;

        case MAVLINK_MSG_ID_KEY_EXCHANGE:
            handle_message_key_exchange(msg);
            break;

        ...
    }
}
```

[그림 14] handle_message 함수의 일부

handle_message 함수는 수신한 MAVLink 메시지의 ID 값을 통해, 각 메시지 종류에 대하여 별도의 함수를 호출한다. key_exchange 메시지의 경우, handle_message_key_exchange 함수를 호출한다.

```
void
MavlinkReceiver::handle_message_key_exchange(mavlink_message_t *msg)
{
    // Secret key, Public key, Shared key 생성
    ...

    mavlink_message_t mavlink_msg;
    mavlink_key_exchange_t key_exchange_msg;

    memcpy(key_exchange_msg.public_key, public_key, sizeof(key_exchange_msg.public_key));

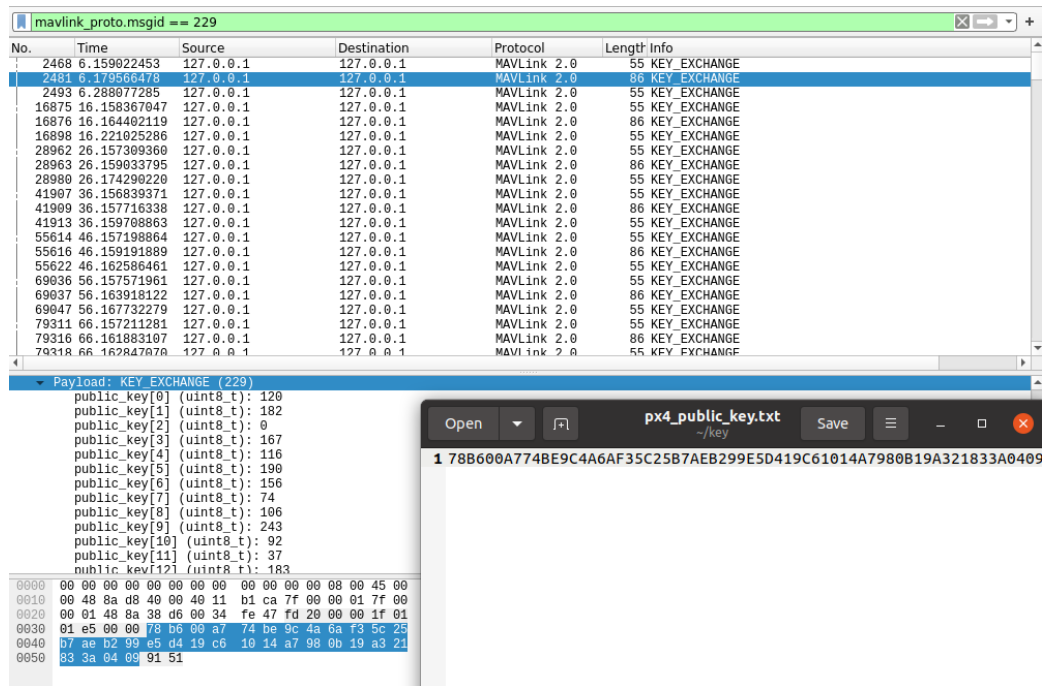
    mavlink_msg_key_exchange_encode(1, 1, &mavlink_msg, &key_exchange_msg);

    const uint8_t* msg_buffer = reinterpret_cast<const uint8_t*>(&key_exchange_msg);

    mavlink_msg_key_exchange_send(MAVLINK_COMM_0, reinterpret_cast<const uint8_t*>(msg_buffer));
}
```

[그림 15] handle_message_key_exchange 함수의 메시지 전달 부분

Key 생성 과정을 거치고, 3.1.1절에서 정의한 커스텀 메시지인 key_exchange 메시지 구조체에 public key 필드를 포함한 MAVLink 메시지를 QGC로 전달한다.



[그림 16] PX4에서 QGC로 전달되는 key_exchange 메시지 패킷의 Wireshark 화면

3.3. 공유 비밀 키 생성

이전의 내용이 모두 진행되면 PX4와 QGC는 각자의 비밀키와 서로의 공유키를 알고 있는 상태가 된다. 두 개의 키를 이용하여 공유 비밀 키 값을 연산하게 되는데, 이는 Curve25519 라는 타원 곡선 상에서 계산된다. 공유 비밀 키가 생성이 되면 이후의 PX4와 QGC 간의 통신에서 이를 기반으로 보다 안전한 통신을 할 수 있다.

3.3.1. Monocypher

본 연구에서 ECDH 알고리즘을 사용하기 위해 선택한 라이브러리는 Monocypher이다. 해당 라이브러리를 선택하게 된 이유는 타 라이브러리에 비하여 경량화 되어 있다는 점이였다.[8] OpenSSL의 경우 많은 암호화와 관련된 알고리즘을 사용할 수 있다. 하지만 리소스가 제한된 임베디드 시스템이나 드론 같은 경우에는 다양한 기능이라는 장점이 단점이 될 수 있다. 또한 많은 알고리즘과 프로토콜을 지원하기 때문에, 많은 의존성을 가진다. 따라서 빌드가 복잡해질 수 있고, 특히 속도가 느려질 수 있다. 이런 이유로 실시간 성능이 중요한 드론

에 적용하기 어렵다.

Monocypher는 Raspberry pi 같은 작은 프로세서나 더 작은 프로세서에서 가장 빠른 속도를 낼 수 있다. 이에 더해, 전체 코드가 3000줄보다 작으며 종속성이 없는 경량화 된 라이브러리이지만 연구에서 적용시키고자 했던 ECDH, EDDSA, Poly1305 등의 알고리즘을 포함하고 있기 때문에 해당 라이브러리를 선택하게 되었다.

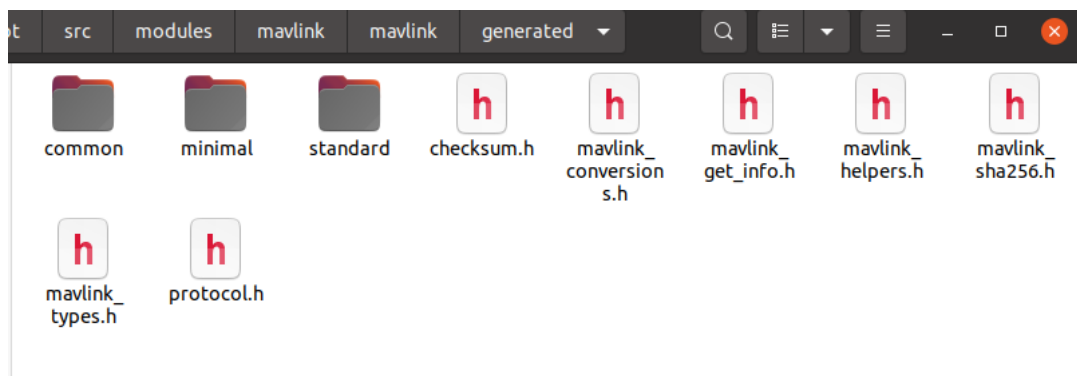
3.3.2. QGC

QGC가 기존 사용하고 있는 MAVLink에 대한 라이브러리에는 새로 생성된 key exchange 메시지에 대한 정보가 포함되어 있지 않다. 그러므로 PX4의 MAVLink 모듈 디렉토리에서 새로 작성된 프로토콜에 대한 라이브러리를 생성하여 QGC에 포함해 주어야 한다.

```
$ python3 -m pymavlink.tools.mavgen --lang=C --wire-protocol=2.0 --  
output=generated/ message_definitions/v1.0/common.xml
```

[표 6] key_exchange 메시지가 포함된 MAVLink 라이브러리 생성

위의 명령어를 입력하게 되면 output 옵션으로 지정한 generated 라는 디렉토리가 생성되고 내부에 common.xml에 선언되어 있는 메시지에 대한 헤더파일들이 새롭게 생성되어 있다.



[그림 17] 생성된 헤더파일 목록

생성된 헤더파일들을 "/qgroundcontrol/libs/mavlink/include/mavlink/v2.0" 경로에 복사하여 넣어주어 QGC에서 사용할 수 있도록 한다. 이 과정을 진행한 뒤 key_exchange에 관련된 함수나 구조체를 사용하려 하면 해당 멤버가 선언되어 있지 않다는 오류가 발생하게 되는데, 이는 QGC의 MAVLink 모듈을 컴파일 할 때 libs 디렉토리의 mavlink 라이브러리를 사용

하는 것이 아닌 매 컴파일 시 github의 mavlink를 fetch하여 진행하는 것으로 확인했다.

```
include(FetchContent)
FetchContent_Declare(mavlink
  GIT_REPOSITORY https://github.com/mavlink/c_library_v2.git
  GIT_TAG 052b8579f8aeb941f34cc9896af22cf1f38939b9
)
FetchContent_MakeAvailable(mavlink)

target_include_directories(MAVLink
  PUBLIC
    ${mavlink_SOURCE_DIR}
    ${mavlink_SOURCE_DIR}/all
    ${mavlink_SOURCE_DIR}/common
)
```

[그림 18] QGC의 MAVLink 모듈 CMakeLists.txt 일부

따라서 해당 부분을 주석 처리한 뒤 로컬에서 관리하는 MAVLink를 사용할 수 있도록 파일을 수정하였다. 추가적으로 [그림 13]와 같이 QGC에서도 Monocypher를 사용하기 위하여 Vehicle 모듈의 CMakeLists를 수정해 준다.

```
set(MONOCYPHER_DIR ${CMAKE_CURRENT_SOURCE_DIR}/monocypher/src)
set(MONOCYPHER_SRC ${MONOCYPHER_DIR}/monocypher.c)
add_library(monocypher STATIC ${MONOCYPHER_SRC})
set_target_properties(monocypher PROPERTIES POSITION_INDEPENDENT_CODE ON)
target_include_directories(monocypher PUBLIC ${MONOCYPHER_DIR})
target_link_libraries(Vehicle monocypher)
```

[그림 19] Vehicle 모듈 CMakeLists.txt 추가 코드

다음은 QGC의 Vehicle 객체에서 전송받은 메시지를 다루는 함수를 수정해야 한다. Vehicle.cc 파일의 _mavlinkMessageReceived() 함수에서 전송받은 메시지를 처리하게 되는데 message의 id에 따라 switch-case문에서 각각의 메시지 핸들링 함수를 호출하고 있다. [그림 20]과 같이 key_exchange 메시지에 대한 case를 추가하여 해당 메시지를 처리하는 함수를 호출할 수 있도록 구현한다.

```
switch (message.msgid) {  
  
case MAVLINK_MSG_ID_KEY_EXCHANGE:  
    _handleKeyExchange(link, message);  
    break;
```

[그림 20] key_exchange 메시지에 대한 case 추가

정상적인 메시지 수신을 하게 되는지 확인하기 위해 log를 출력하는 함수를 추가하여 확인해 보았다. 하지만 _mavlinkMessageReceived() 함수 내부에서는 key_exchange 메시지의 id인 255를 확인할 수 없었고, QGC의 communication flow에 따라 역으로 함수를 조사했지만, LinkManager 객체에서 PX4와 연결이 시작된 지점부터 QGC 내부에서 key_exchange에 대한 로그를 출력할 수 없었다. 구현 단계 중에서 많은 시간을 할애했으나 해결 방법을 찾지 못하였다. 이후 기존 메시지(heartbeat, ping 등)를 활용하는 방안을 고안하였으나 기존의 메시지는 각각의 역할이 정해져 있으므로 키 교환에 사용하기에 적합하지 않다고 판단하여 적용을 보류하였고, 후의 과정이 많이 남아있었기에 public key를 텍스트 파일로 저장하여 관리하는 방안으로 남은 과정을 진행하였다.

```

void Vehicle::writePublicKey(uint8_t *key) {
    QFile file("/home/soo/key/qgc_public_key.txt");

    if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {
        qDebug() << "Failed to open the file for writing";
        return;
    }

    QTextStream out(&file);
    for(int i=0; i<KEY_SIZE; i++) {
        out << QString::number(static_cast<unsigned char>(key[i]), 16).rightJustified(2, '0').toUpper();
    }

    file.close();
    return;
}

void Vehicle::readPublicKey() {
    QFile file("/home/soo/key/px4_public_key.txt");

    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qDebug() << "Failed to open the file for writing";
        return;
    }

    QTextStream openFile(&file);
    QString keyData = openFile.readAll();
    uint8_t temp[32];

    bool ok;

    for(int i=0; i<KEY_SIZE; i++) {
        QString byteString = keyData.mid(i * 2, 2);
        temp[i] = static_cast<uint8_t>(byteString.toUInt(&ok, 16));
        if (!ok) {
            qWarning() << "Error converting hex to byte:" << keyData.mid(i, 2);
            return;
        }
        // else {
        //     qDebug() << byteString << " " << temp[i];
        // }
    }
    memcpy(px4_public_key, temp, KEY_SIZE);
    crypto_x25519(shared_key, secret_key, px4_public_key);

    file.close();
}

```

[그림 21] public key read/write 함수

Vehicle 객체는 기체와 연결이 이루어진 후에 생성되게 되는데 이때 호출되는 생성자 내부에서 [그림 12]의 코드를 추가해 연결된 기체와 통신에 사용될 secret key와 public key를 생성하게 한다. write 함수에서 자신의 public key를 저장하도록 하고, read 함수에서는 상대의 public key를 읽어온 뒤 이 키와 자신의 secret key를 기반으로 crypto_x25519() 함수를 사용하여 공유 비밀 키를 생성한다.

3.3.3. PX4

3.2.2절에서 언급된 handle_message_key_exchange 함수에서 secret key, public key, 그리고 shared secret key를 생성한다. 함수는 [그림 22]와 같이 구현하였다.

```

void
MavlinkReceiver::handle_message_key_exchange(mavlink_message_t *msg)
{
    static bool flag = false;

    if (!flag) {
        flag = true;
        // 1. secret key 생성
        key_generation(secret_key);

        // 2. public key 생성
        crypto_x25519_public_key(public_key, secret_key);

        FILE *file_px4 = fopen("/home/soo/key/px4_public_key.txt", "w");
        if(file_px4 == nullptr) { return; }

        for(int i=0; i<32; i++) {
            fprintf(file_px4, "%02X", public_key[i]);
        }

        fclose(file_px4);
    }
    // 3. shared key 생성
    uint8_t qgc_public_key[32];
    FILE *file_qgc = fopen("/home/soo/key/qgc_public_key.txt", "r");

    if(file_qgc == nullptr) { return; }

    for (int i = 0; i < 32; i++) {
        unsigned int value;
        int result = fscanf(file_qgc, "%02X", &value);
        if (result != 1) { return; }
        qgc_public_key[i] = static_cast<uint8_t>(value);
    }
    fclose(file_qgc);

    crypto_x25519(shared_key, secret_key, qgc_public_key);

    ...
}

```

[그림 22] handle_message_key_exchange 함수의 key 생성 부분

먼저, key_exchange 메시지는 주기적으로 수신되므로, 해당 메시지가 수신될 때마다 secret key와 public key를 생성할 경우 계속해서 Key 값이 변하게 된다. 따라서 static 변수인 “flag”를 이용하여 key_exchange 메시지를 최초로 수신했을 때만 secret key와 public key를 생성하도록 하였다. secret key는 3.2.2절에 언급하였던 [그림 12]의 코드를 이용하여 생성되고, public key는 crypto_x25519_public_key 함수로 생성된다. 생성된 public key 값은 로컬 파일에 쓰는 방식으로 전달하도록 하였다.

다음으로, QGC의 public key 값을 계속해서 확인하면서 shared secret key를 생성한다. QGC와 연결이 성공적으로 이루어지면 signature 검증에 이용할 shared secret key 값을 가지게 된다.

3.4. Signature 생성 및 검증

3.4.1. QGC

처음 시도했던 방법은 LinkInterface 객체의 initMavlinkSigning() 함수를 호출하여 PX4와 QGC 사이 통신에 서명을 활성화시키는 것이었다. 생성된 shared key를 이용할 수 있도록 하기 위해 기존 함수를 오버로딩하여 인자를 추가해 주었고, 전달받은 key 인자를 사용할 수 있도록 함수를 수정하였다.

```
bool LinkInterface::initMavlinkSigning()
{
    if (!isSecureConnection()) {
        auto appSettings = qgcApp()->toolbox()->settingsManager()->appSettings();
        QByteArray signingKeyBytes = appSettings->mavlink2SigningKey()->rawValue().toByteArray();
        if (MAVLinkSigning::initSigning(static_cast<mavlink_channel_t>(_mavlinkChannel), signingKeyBytes, MAVLink
            if (signingKeyBytes.isEmpty()) {
                qDebug(LinkInterfaceLog) << "Signing disabled on channel" << _mavlinkChannel;
            } else {
                qDebug(LinkInterfaceLog) << "Signing enabled on channel" << _mavlinkChannel;
            }
        } else {
            qWarning() << Q_FUNC_INFO << "Failed To enable Signing on channel" << _mavlinkChannel;
            // FIXME: What should we do here?
            return false;
        }
    }

    return true;
}

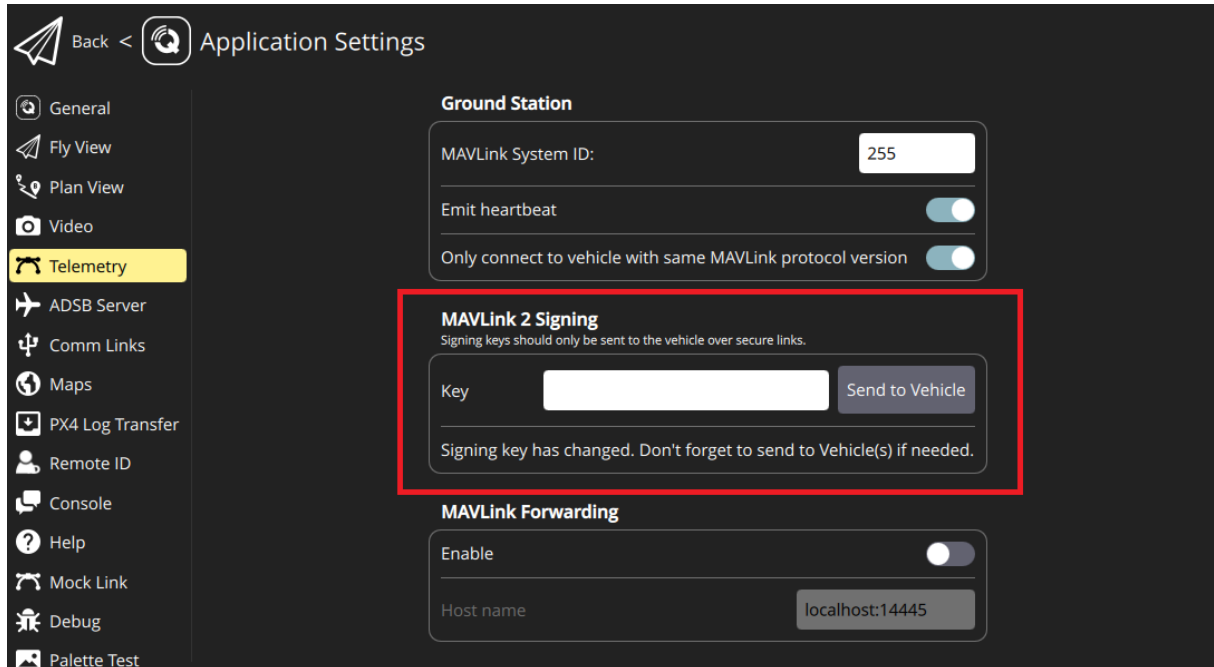
bool LinkInterface::initMavlinkSigning(uint8_t* key)
{
    if (!isSecureConnection()) {
        auto appSettings = qgcApp()->toolbox()->settingsManager()->appSettings();
        QByteArray signingKeyBytes(reinterpret_cast<const char*>(key), sizeof(key));
        if (MAVLinkSigning::initSigning(static_cast<mavlink_channel_t>(_mavlinkChannel), signingKeyBytes, MAVLink
            if (signingKeyBytes.isEmpty()) {
                qDebug(LinkInterfaceLog) << "Signing disabled on channel" << _mavlinkChannel;
            } else {
                qDebug(LinkInterfaceLog) << "Signing enabled on channel" << _mavlinkChannel;
            }
        } else {
            qWarning() << Q_FUNC_INFO << "Failed To enable Signing on channel" << _mavlinkChannel;
            // FIXME: What should we do here?
            return false;
        }
    }

    return true;
}
```

[그림 23] initMavlinkSigning 함수 오버로딩

이후 진행한 테스트에서 가장 처음 조건문의 조건인 isSecureConnection() 함수가 true만을 반환하는 것을 확인하였다. 따라서 조건문의 내부로 진입이 불가하였기 때문에 대책으로 initSigning 함수를 호출하는 방법을 찾아보게 되었다. QGC 어플리케이션의 Application settings에서 Telemetry 메뉴를 보게 되면 MAVLink 2 Signing 레이아웃을 확인할 수 있다.

이 레이아웃의 역할을 알아보기 위하여 UI 소스코드를 확인해 본 결과 “Send to Vehicle” 버튼을 클릭하게 되면 key label의 값을 setup_signing 메시지의 secret_key로 사용하여 연결되어 있는 기체로 setup_signing 메시지를 전송하게 된다.



[그림 24] QGC Application Settings의 Telemetry 메뉴

```

RowLayout {
    spacing: ScreenTools.defaultFontPixelWidth

    LabelledFactTextField {
        Layout.fillWidth: true
        textFieldPreferredWidth: ScreenTools.defaultFontPixelWidth * 32
        label: qsTr("Key")
        fact: mavlink2SigningGroup._mavlink2SigningKey
    }

    QGCBUTTON {
        text: qsTr("Send to Vehicle")
        enabled: _activeVehicle

        onClicked: {
            sendToVehiclePrompt.visible = false
            _activeVehicle.sendSetupSigning()
        }
    }
}

```

[그림 25] MAVLink 2 Signing layout 코드 일부

onClicked 이벤트의 코드를 보면 연결된 기체의 sendSetupSigning 함수를 호출하는 것을 확인할 수 있다. 해당 함수는 기체와 QGC 사이의 channel에서 서명이 포함될 수 있도록 서

명을 설정하는 메시지를 기체로 보내는 함수이다. 메시지 전송 만으로 서명이 생성될 수 있는지 테스트를 진행해 보았으나, 단순히 메시지 전송 이외의 결과는 확인할 수 없었다. 따라서 서명을 생성한 뒤 메시지를 전송하고 PX4에서 메시지를 전송 받았을 때 PX4에서 서명을 생성할 수 있도록 직접 구현해야 했다.

```
void createSetupSigning(mavlink_channel_t channel, mavlink_system_t target_system,
                      mavlink_setup_signing_t &setup_signing)
{
    (void) memset(&setup_signing, 0, sizeof(setup_signing));
    setup_signing.target_system = target_system.sysid;
    setup_signing.target_component = target_system.compid;

    const mavlink_signing_t* const signing = _getChannelSigning(channel);
    if (signing) {
        setup_signing.initial_timestamp = signing->timestamp;
    }
}
```

[그림 26] setup_signing 메시지에 필요한 parameter 할당

[그림 26]의 함수를 sendSetupSigning 함수에서 호출할 수 있도록 수정하였고, 이후에 initSigning 함수를 호출하도록 구현하였다. 추가적으로, 서명에 생성에 필요한 키를 ECDH 알고리즘으로 생성된 shared secret key로 사용할 수 있도록 하였다. 테스트를 진행할 때 key label에 어떤 값을 입력한 상태로 "Send to Vehicle" 버튼을 클릭하게 되면 입력 값을 key로 사용하게 되고 연결된 기체로 입력 값을 전송하게 되기 때문에 key label은 비워 둔 상태로 진행하였다.

```

bool initSigning(mavlink_channel_t channel, QByteArrayView key, mavlink_accept_unsigned_t callback)
{
    if (!key.isEmpty() && !callback) {
        qWarning() << Q_FUNC_INFO << "callback must be specified";
        return false;
    }

    mavlink_status_t* const status = mavlink_get_channel_status(channel);
    if (key.isEmpty()) {
        status->signing = nullptr;
        status->signing_streams = nullptr;
    } else {
        static mavlink_signing_t s_signing[MAVLINK_COMM_NUM_BUFFERS];
        static mavlink_signing_streams_t s_signing_streams;

        mavlink_signing_t* const signing = &s_signing[channel];
        signing->link_id = channel;
        signing->flags |= MAVLINK_SIGNING_FLAG_SIGN_OUTGOING;
        signing->accept_unsigned_callback = callback;

        // _setSigningKey(signing, key);
        memcpy(signing->secret_key, key.data(), key.size());

        _setSigningTimestamp(signing);

        status->signing = signing;
        status->signing_streams = &s_signing_streams;
    }

    return true;
}

```

[그림 27] initSigning 함수

```

void Vehicle::sendSetupSigning()
{
    SharedLinkInterfacePtr sharedLink = vehicleLinkManager()->primaryLink().lock();
    if (!sharedLink) {
        qDebug(VehicleLog) << Q_FUNC_INFO << "Primary Link Gone!";
        return;
    }

    const mavlink_channel_t channel = static_cast<mavlink_channel_t>(sharedLink->mavlinkChannel());

    mavlink_setup_signing_t setup_signing;
    memcpy(setup_signing.secret_key, shared_key, KEY_SIZE);

    QByteArray signingKeyBytes(reinterpret_cast<const char*>(shared_key), sizeof(shared_key));
    if (MAVLinkSigning::initSigning(channel, signingKeyBytes, MAVLinkSigning::insecureConnectionAcceptUnsigned)
        if (signingKeyBytes.isEmpty()) {
            qDebug(LinkInterfaceLog) << "Signing disabled on channel" << channel;
        } else {
            qDebug(LinkInterfaceLog) << "Signing enabled on channel" << channel;
        }
    } else {
        qWarning() << Q_FUNC_INFO << "Failed To enable Signing on channel" << channel;
    }

    mavlink_system_t target_system;
    target_system.sysid = id();
    target_system.compid = defaultComponentId();

    MAVLinkSigning::createSetupSigning(channel, target_system, setup_signing);

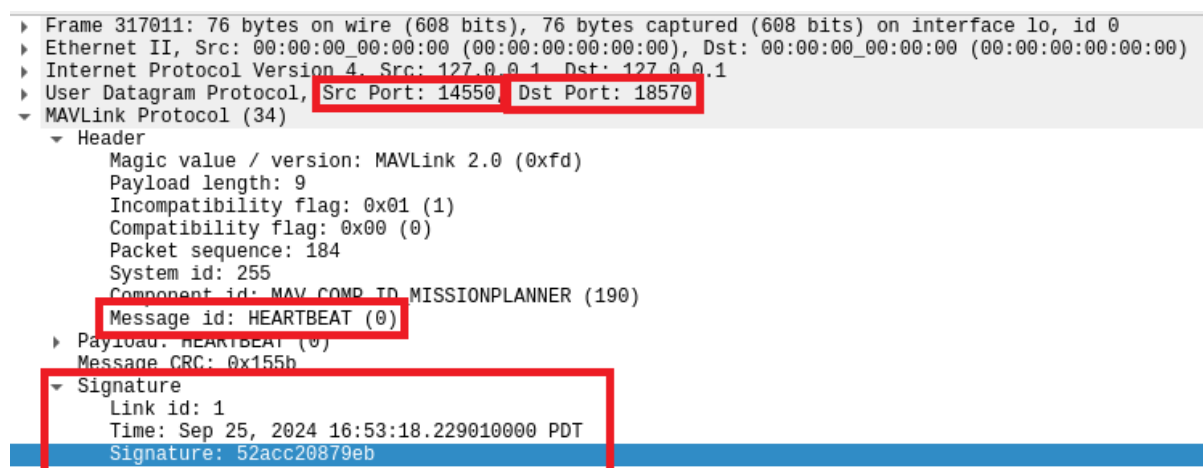
    mavlink_message_t msg;
    (void) mavlink_msg_setup_signing_encode_chan(_mavlink->getSystemId(),
                                                _mavlink->getComponentId(), channel, &msg, &setup_signing);

    // Since we don't get an ack back that the message was received send twice to try to make sure it makes it
    for (uint8_t i = 0; i < 2; ++i) {
        sendMessageOnLinkThreadSafe(sharedLink.get(), msg);
    }
}

```

[그림 28] sendSetupSigning 함수

위 과정을 모두 진행한 뒤 "Send to Vehicle" 버튼을 클릭하여 setup_signing 메시지를 전송하게 되면, QGC에서 PX4로 전송하는 메시지에 signature가 포함되게 된다. Wireshark를 이용하여 확인한 결과는 [그림 29]과 같다. 해당 패킷의 정보를 간략히 설명하면 다음과 같다. "Src Port : 14550"은 QGC의 port이고 "Dst Port : 18570"은 PX4의 port이며, heartbeat 메시지에 signature가 포함되어 있는 모습이다.



[그림 29] signature가 포함된 메시지 정보

3.4.2. SHA256

수신한 MAVLink 메시지의 signature에 대한 검증을 위해, 자체적으로 signature를 연산해야 한다. SHA256 알고리즘을 MAVLink 메시지의 signature 생성에 활용할 때, 아래의 공식을 따른다.

$$signature = sha256_48(secret_key + header + payload + CRC + link - ID + timestamp)$$

또한, SHA256에 이용되는 함수들은 MAVLink 라이브러리의 "mavlink_sha256.h" 파일에 정의되어 있다. 해당 파일의 함수를 이용하여, 데이터에 SHA256 해시를 적용할 수 있다. 순서대로 mavlink_sha256_init, mavlink_sha256_update, mavlink_sha256_final_48 함수를 적용하고, 최종적으로 앞의 48비트, 즉 6바이트가 계산된 Signature가 된다.

```
// 전체 데이터를 한 번에 해시
mavlink_sha256_ctx sha_ctx;
mavlink_sha256_init(&sha_ctx);

// SHA256 연산
mavlink_sha256_update(&sha_ctx, combined_data, combined_length);

// SHA256 마무리 및 결과 저장
mavlink_sha256_final_48(&sha_ctx, calculated_signature);
```

[그림 30] 데이터에 SHA256 함수를 순차적으로 적용하는 코드

3.4.3. PX4

QGC로부터 setup_signing 메시지를 수신한 시점부터 서명 검증을 시작하도록 구현했다.

```
void
MavlinkReceiver::handle_message_setup_signing(mavlink_message_t *msg)
{
    signature_verification = true;
    PX4_INFO("Signature verification ON");

    mavlink_setup_signing_t setup_signing_msg;
    mavlink_msg_setup_signing_decode(msg, &setup_signing_msg);

    memcpy(setup_signing_msg.secret_key, shared_key, 32);
    _mavlink.setup_signing(shared_key, msg);
}
```

[그림 31] handle_message_setup_signing 함수

전역 변수인 "signature_verification"의 값이 true가 되면, handle_message 함수에서 이후 수신하는 메시지들에 대해 서명 검증을 시작한다.

```
void
MavlinkReceiver::handle_message(mavlink_message_t *msg)
{
    ...

    if(signature_verification && !verify_signature(msg)) {
        return;
    }

    case MAVLINK_MSG_ID_COMMAND_LONG:
        handle_message_command_long(msg);
        break;
    ...
}
```

[그림 32] handle_message 함수의 일부

"command_long" 메시지를 예시로 들면, 메시지에 대해 signature_verification 값이 true가 되었지만, 서명을 검증하는 verify_signature 함수가 false를 반환, 즉 서명 검증에 실패한 경우가 있다. 이 경우 return을 처리함으로써 handle_message_command_long 함수를 호출하지 않도록 하고, 이는 해당 메시지를 무시한 것과 같은 동작을 하게 된다.

서명을 검증하는 함수인 verify_signature 함수는 bool 타입의 반환 값을 가지고, 코드를 살

펴보면 다음과 같다.

```
switch (msg->msgid) {
    case MAVLINK_MSG_ID_PING: {
        mavlink_ping_t ping_msg;
        mavlink_msg_ping_decode(msg, &ping_msg);
        message_data = reinterpret_cast<uint8_t*>(&ping_msg);
        message_length = MAVLINK_MSG_ID_PING_LEN;
        break;
    }
    case MAVLINK_MSG_ID_SET_MODE: {
        mavlink_set_mode_t set_mode_msg;
        mavlink_msg_set_mode_decode(msg, &set_mode_msg);
        message_data = reinterpret_cast<uint8_t*>(&set_mode_msg);
        message_length = MAVLINK_MSG_ID_SET_MODE_LEN;
        break;
    }
    ...

    default:
        PX4_WARN("This message is not supported for signature verification");
        return result;
}
```

[그림 33] verify_signature 함수의 메시지 구분

먼저, MAVLink 메시지 종류에 따라 서로 다른 payload 값을 가지므로, 각 메시지에 대하여 payload와 payload 길이를 따로 지정한다. heartbeat, ping, command_int, command_long, set_mode, setup_signing 등의 메시지들에 대하여 서명 검증을 수행하도록 하였다.

다음으로, 3.4.2절의 signature 계산 공식에 따라 각 데이터를 순서대로 "combined_data" 배열에 합친다. 이때, little endian으로 값이 저장되는 것을 감안하여 1 byte보다 큰 사이즈를 가지는 필드, msgid(3 byte)와, CRC(2 byte)에 대해서는 데이터를 반전한 뒤에 연산해야 한다.

```

if (message_data != nullptr) {
    uint8_t combined_data[512];
    size_t combined_length = 0;

    // shared_key
    memcpy(&combined_data[combined_length], shared_key, sizeof(shared_key));
    combined_length += sizeof(shared_key);

    // header
    combined_data[combined_length++] = msg->magic;
    combined_data[combined_length++] = msg->len;
    combined_data[combined_length++] = msg->incompat_flags;
    combined_data[combined_length++] = msg->compat_flags;
    combined_data[combined_length++] = msg->seq;
    combined_data[combined_length++] = msg->sysid;
    combined_data[combined_length++] = msg->compid;

    uint32_t msgid = msg->msgid;
    combined_data[combined_length++] = static_cast<uint8_t>(msgid & 0xFF);
    combined_data[combined_length++] = static_cast<uint8_t>((msgid >> 8) & 0xFF);
    combined_data[combined_length++] = static_cast<uint8_t>((msgid >> 16) & 0xFF);

    // payload
    memcpy(&combined_data[combined_length], message_data, message_length);
    combined_length += message_length;

    // CRC
    uint16_t crc = msg->checksum;
    combined_data[combined_length++] = static_cast<uint8_t>(crc & 0xFF);
    combined_data[combined_length++] = static_cast<uint8_t>((crc >> 8) & 0xFF);

    // link-ID
    combined_data[combined_length++] = msg->signature[0];

    // timestamp
    memcpy(&combined_data[combined_length], &msg->signature[1], 6);
    combined_length += 6;
}

```

[그림 34] SHA256 적용을 위하여 데이터를 하나의 배열에 합치는 코드

위와 같이 데이터를 합친 후 SHA256을 적용하고, 계산된 signature와 수신한 MAVLink의 signature를 비교하여 같은 값을 가지는지 확인한다.

```

// 서명 검증
if (memcmp(&msg->signature[7], calculated_signature, 6) == 0) {
    PX4_INFO(COLOR_GREEN "*** Signature Verification Succeeded ***" COLOR_RESET);
    result = true;
} else {
    PX4_INFO(COLOR_RED "*** Signature Verification Failed ***" COLOR_RESET);
}

```

[그림 35] verify_signature 함수의 서명 검증 부분

만약 서명 검증에 성공한다면 함수의 반환 값인 "result" 값이 true가 되고, [그림 35]의 handle_message 함수를 종료시키는 조건문을 만족하지 않게 되어 수신 받은 메시지의 핸들링 함수를 호출하는 switch-case문으로 넘어갈 수 있게 된다.

서명 검증을 성공할 수 있는지 테스트해보기 위해 시뮬레이션을 여러 번 진행해 보았을 때, 서명을 생성하기 위해 setup_signing 메시지를 전송한 후 QGC와 PX4의 연결이 종료되는 문제가 발생하였다. wireshark를 통해 패킷 전송 내역을 확인한 결과, [그림 36]에 표시된 setup_signing 메시지 이후 전송되는 패킷이 QGC 내부에서 거부되었다.

33360	15.281667748	127.0.0.1	127.0.0.1	MAVLink 2.0	95 LOCAL_POSITION_NED
33361	15.281675097	127.0.0.1	127.0.0.1	MAVLink 2.0	118 POSITION_TARGET_LOCAL_NED
33362	15.281680469	127.0.0.1	127.0.0.1	MAVLink 2.0	104 SERVO_OUTPUT_RAW
33366	15.283721540	127.0.0.1	127.0.0.1	MAVLink 2.0	95 LOCAL_POSITION_NED
33368	15.283733132	127.0.0.1	127.0.0.1	MAVLink 2.0	299 ODOMETRY
33374	15.287010777	127.0.0.1	127.0.0.1	MAVLink 2.0	95 ATTITUDE
33377	15.292155008	127.0.0.1	127.0.0.1	MAVLink 2.0	77 SETUP_SIGNING
33378	15.292174131	127.0.0.1	127.0.0.1	MAVLink 2.0	77 SETUP_SIGNING
33380	15.298140183	127.0.0.1	127.0.0.1	MAVLink 2.0	129 HIGHRES_IMU
33388	15.298181139	127.0.0.1	127.0.0.1	MAVLink 2.0	99 ATTITUDE_QUATERNION
33390	15.298226122	127.0.0.1	127.0.0.1	MAVLink 2.0	95 GLOBAL_POSITION_INT
33398	15.301513262	127.0.0.1	127.0.0.1	MAVLink 2.0	95 ATTITUDE
33399	15.301658521	127.0.0.1	127.0.0.1	MAVLink 2.0	99 ATTITUDE_QUATERNION
33400	15.301711175	127.0.0.1	127.0.0.1	MAVLink 2.0	103 ATTITUDE_TARGET
33401	15.301739694	127.0.0.1	127.0.0.1	MAVLink 2.0	95 GLOBAL_POSITION_INT
33402	15.301757233	127.0.0.1	127.0.0.1	MAVLink 2.0	95 LOCAL_POSITION_NED

[그림 36] QGC와 PX4 통신 데이터 확인

PX4와 QGC의 여러 함수에서 로그를 출력해 보며 테스트해본 결과, 서명이 활성화 된 stream에서 PX4의 signing stream 데이터 중 system의 id와 component의 id가 QGC의 id와 동일하지 않아 QGC의 서명 검증 단계에서 패킷을 받아들이지 않는 문제로 확인되었다. 문제를 해결하기 위해 PX4에서 서명을 설정하는 함수를 새로 구현하여 연결된 QGC의 id를 지정할 수 있도록 하였다.

```
void Mavlink::setup_signing(uint8_t *key, mavlink_message_t* msg) {
    memcpy(_mavlink_signing.secret_key, key, 32);
    set_timestamp();

    global_mavlink_signing_streams.stream[0].link_id = _mavlink_signing.link_id;
    global_mavlink_signing_streams.stream[0].sysid = msg->sysid;
    global_mavlink_signing_streams.stream[0].compid = msg->compid;

    _mavlink_status.signing = &_mavlink_signing;
    _mavlink_status.signing_streams = &global_mavlink_signing_streams;
}
```

[그림 37] signing 설정 함수

이후, PX4와 QGC를 실행하고 QGC의 Application Settings - Telemetry의 "Send to Vehicle" 버튼을 클릭하게 되면, setup_signing 메시지가 전달되면서 서명 활성화 및 검증이 진행된다. 서명 검증에 성공하거나 실패하면, 아래와 같은 결과가 출력된다.

```

INFO [mavlink] *** Signature Verification Succeeded ***
INFO [mavlink] | Message: PING
INFO [mavlink] | CRC: 46E7
INFO [mavlink] | Received Signature: B8A41BA73CB5
INFO [mavlink] | Calculated Signature: B8A41BA73CB5
INFO [mavlink] -----
INFO [mavlink] *** Signature Verification Succeeded ***
INFO [mavlink] | Message: HEARTBEAT
INFO [mavlink] | CRC: C9D3
INFO [mavlink] | Received Signature: 831E2CEB90BB
INFO [mavlink] | Calculated Signature: 831E2CEB90BB
INFO [mavlink] -----
INFO [mavlink] *** Signature Verification Succeeded ***
INFO [mavlink] | Message: COMMAND_LONG
INFO [mavlink] | CRC: 0C05
INFO [mavlink] | Received Signature: 1B8E0CD26298
INFO [mavlink] | Calculated Signature: 1B8E0CD26298
INFO [mavlink] -----
INFO [mavlink] *** Signature Verification Succeeded ***
INFO [mavlink] | Message: COMMAND_LONG
INFO [mavlink] | CRC: D4D3
INFO [mavlink] | Received Signature: 17B0BC4673B1
INFO [mavlink] | Calculated Signature: 17B0BC4673B1
INFO [mavlink] -----
INFO [commander] Armed by external command
INFO [tone_alarm] arming warning
INFO [mavlink] *** Signature Verification Succeeded ***
INFO [mavlink] | Message: HEARTBEAT
INFO [mavlink] | CRC: 2C19
INFO [mavlink] | Received Signature: A75A6C36E28C
INFO [mavlink] | Calculated Signature: A75A6C36E28C
INFO [mavlink] -----
INFO [commander] Takeoff detected
INFO [mavlink] *** Signature Verification Succeeded ***
INFO [mavlink] | Message: SET_MODE
INFO [mavlink] | CRC: BC07
INFO [mavlink] | Received Signature: 06469D567FAC
INFO [mavlink] | Calculated Signature: 06469D567FAC
INFO [mavlink] -----

```

[그림 38] 서명 검증 성공 출력

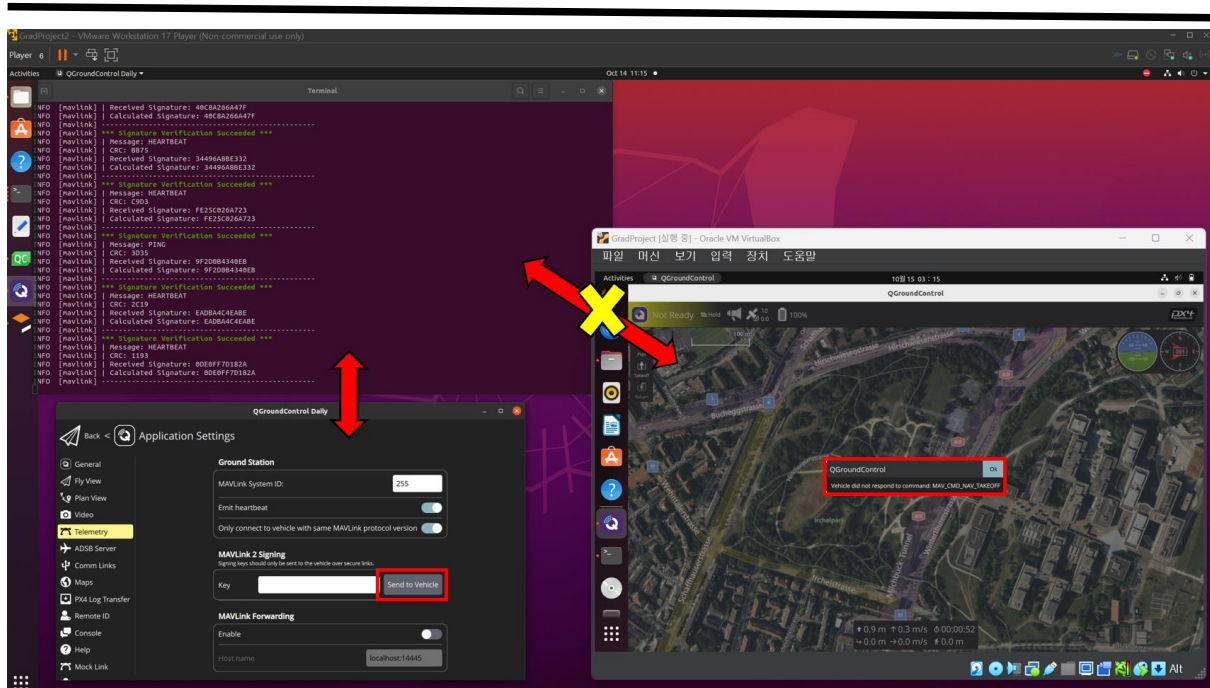
```

INFO [mavlink] *** Signature Verification Failed ***
INFO [mavlink] | Message: COMMAND_LONG
INFO [mavlink] | CRC: 1C83
INFO [mavlink] | Received Signature: F92865DF5156
INFO [mavlink] | Calculated Signature: B13E36ADD3E7
INFO [mavlink] -----
INFO [mavlink] *** Signature Verification Failed ***
INFO [mavlink] | Message: HEARTBEAT
INFO [mavlink] | CRC: A2CF
INFO [mavlink] | Received Signature: 8C0962DA3DDD
INFO [mavlink] | Calculated Signature: 606E59F73F65
INFO [mavlink] -----
INFO [mavlink] *** Signature Verification Failed ***
INFO [mavlink] | Message: HEARTBEAT
INFO [mavlink] | CRC: AD8C
INFO [mavlink] | Received Signature: B0A8072E30AE
INFO [mavlink] | Calculated Signature: F418E9685C95
INFO [mavlink] -----
INFO [mavlink] *** Signature Verification Failed ***
INFO [mavlink] | Message: COMMAND_LONG
INFO [mavlink] | CRC: 0400
INFO [mavlink] | Received Signature: 2F83BED83173
INFO [mavlink] | Calculated Signature: 340C1B484CDB
INFO [mavlink] -----
INFO [mavlink] *** Signature Verification Failed ***
INFO [mavlink] | Message: HEARTBEAT
INFO [mavlink] | CRC: DF2A
INFO [mavlink] | Received Signature: 652D239A9093
INFO [mavlink] | Calculated Signature: 3592687C1848
INFO [mavlink] -----

```

[그림 39] 서명 검증 실패 출력

또한, 포트 설정을 통하여 서로 다른 QGC에서 같은 드론으로 연결을 할 수 있는데, 한쪽 QGC에서 서명 활성화를 실행하면, 서명 활성화가 되지 않은 다른 QGC에서 같은 드론에 조종 명령을 전송했을 때 해당 명령을 무시하도록 구현하였다. [그림 40]은 take off에 대한 명령이 서명 검증에 실패하여 명령이 무시된 모습이다.



[그림 40] 다른 QGC로부터의 명령이 무시된 장면

3.5. 구성원별 역할

이름	역할
이경민	<ul style="list-style-type: none"> 시나리오 설계 ECDH 알고리즘 적용 signature 필드 활성화 및 서명 적용 비행 로그 분석 추후 연구 방향 설계
조수현	<ul style="list-style-type: none"> 환경 구축 custom message를 통한 통신 메시지 설계 서명 검증 과정 구현 서명 검증에서의 오버헤드 측정

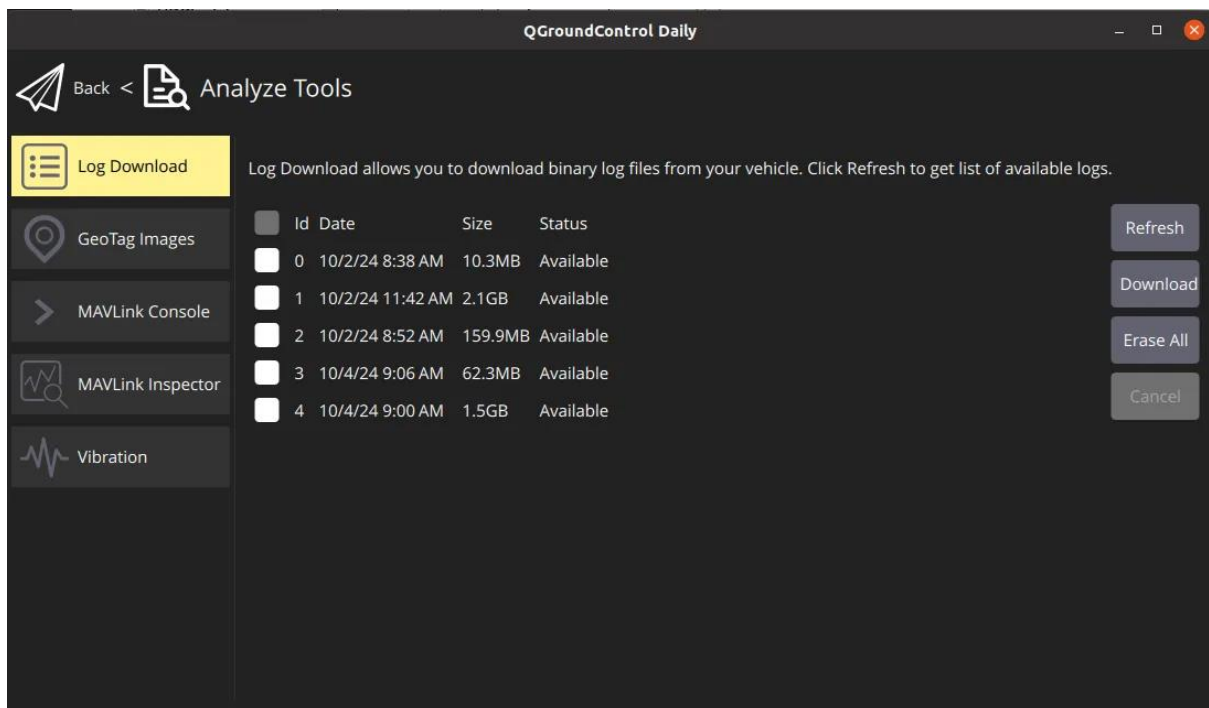
[표 7] 구성원별 역할

4. 연구 결과 분석 및 평가

4.1. 성능 측정

4.1.1. CPU & RAM 사용량

PX4와 QGC를 제공하는 Dronecode 프로젝트에서는 비행 로그를 분석할 수 있는 Flight review라는 tool을 제공하고 있다. QGC에서 비행 로그를 다운로드 받아 tool에 업로드하면 비행 모드에 따라(Loiter, Mission, Position, Take off 등) 비행 중 생긴 문제 혹은 비행 당시의 속도, 위치, 모터의 속도, GPS noise 등 많은 데이터들을 시각화하여 나타내준다.



[그림 41] 비행 Log Download

여러 데이터들 중 본 연구에서 유의미한 결과를 보여 줄 수 있는 CPU와 RAM 사용량에 대한 분석에 집중하였다. 비행 로그를 생성하기 위하여 QGC와 PX4를 실행시킨 뒤 대략 10분 간 여러 모드의 비행을 진행하였다. 본 연구에서 구현한 서명 기능이 활성화된 PX4와 QGC에서와 서명 기능이 포함되지 않은 PX4와 QGC에서의 비행을 각각 진행하여 2개의 로그를 생성하였다. 이후 두 개의 로그를 업로드 하여 CPU와 RAM 사용량에 대한 그래프를 확인한 결과는 아래와 같다.



[그림 42] 서명 기능이 포함되지 않은 Application에서의 로그



[그림 43] 서명이 포함된 Application에서의 로그

[그림 42]는 서명 기능이 활성화되지 않은 상태의 새로운 QGC 어플리케이션을 다운로드 받아 비행을 진행한 후의 로그이다. RAM 사용량은 평균 약 73%이며, CPU는 평균 약 19%의 사용량을 보여준다.

[그림 43]은 서명 기능이 활성화된 이후의 비행에 대한 로그이다. RAM 사용량은 평균 약 74%이며, CPU는 평균 약 19%의 사용량을 보여준다. 두 개의 지표에서 거의 동일한 사용량을 확인할 수 있었다.

4.1.2. 메시지 처리 시간 성능 측정

서명 기능 활성화 전과 후에 대한 메시지 처리 시간을 측정하여, 서명 기능이 추가되었을 때의 오버헤드를 확인해 보았다. 메시지에 따라 패킷의 길이가 다르고, 메시지 수신 시 처리하는 동작도 다르므로, 각 메시지에 따라 따로 측정해야 한다. 먼저 "heartbeat" 메시지에 대해서 측정하였는데, heartbeat 메시지는 MAVLink 네트워크에서 시스템의 존재를 broadcast 하고, 1초의 주기로 전송된다.

메시지 처리 시간은 PX4가 메시지를 수신한 시점부터 서명 검증 완료까지의 시간을 측정하였다. 먼저, "start_time"은 mavlink_receiver.cpp 파일 내의 handle_message 함수 시작 부분으로 지정하였다. 다음으로, "end_time"은 signature 필드를 검증하는 verify_signature 함수와, 수신된 heartbeat 메시지를 처리하는 handle_message_heartbeat 함수의 호출이 끝난 이후로 지정하였다. 이후 "start_time"과 "end_time"의 시간차를 "ns" 단위로 계산하도록 하였다.

```
void
MavlinkReceiver::handle_message(mavlink_message_t *msg)
{
    ...

    uint64_t start_time = get_time_ns();

    ...

    if(signature_verification && !verify_signature(msg)) {
        return;
    }

    case MAVLINK_MSG_ID_HEARTBEAT: {
        handle_message_heartbeat(msg);

        if (msg->sysid == 255) {
            if (signature_verification) { // 서명 기능 활성화 후
                uint64_t end_time = get_time_ns();
                uint64_t time_taken = end_time - start_time;

                // 연산 및 출력
                ...
            }
        } else { // 서명 기능 활성화 전
            uint64_t end_time = get_time_ns();
            uint64_t time_taken = end_time - start_time;

            // 연산 및 출력
            ...
        }
    }
    break;
}
```

[그림 44] 성능 측정을 위한 코드 요약

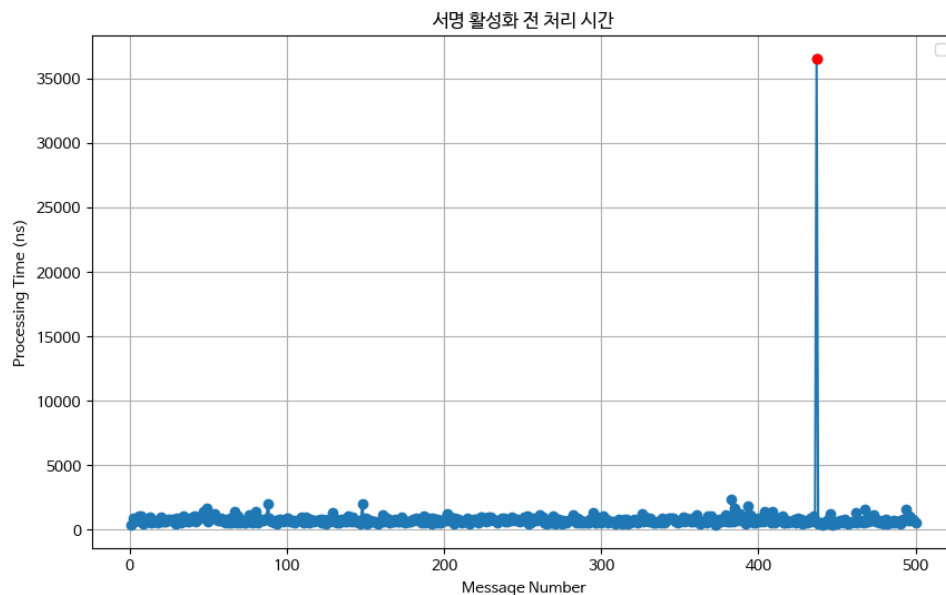
메시지 처리 시간은 서명 기능 활성화 전후 총 500번의 메시지 전송에 대하여 측정해 보았다. 연산 및 출력으로 인한 오버헤드를 줄이기 위하여 100번, 200번, 300번, 400번, 500번째

의 메시지 전송에서만 최소, 최대, 평균 처리 시간과, pps(packet per second), BPS(Byte per second)를 출력하도록 하였다. 그리고 모든 메시지 전송에 대하여 메시지 처리 1회에 걸린 시간을 각각 출력하도록 하였다.

```
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 772 ns (0.77 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 1342 ns (1.34 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 521 ns (0.52 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 561 ns (0.56 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 982 ns (0.98 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 771 ns (0.77 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 551 ns (0.55 us)
INFO [mavlink] --- setup_signing 적용 전 성능 측정 (300회 전송) ---
INFO [mavlink] | 최소 메시지 처리 시간: 411 ns (0.41 us)
INFO [mavlink] | 최대 메시지 처리 시간: 2014 ns (2.01 us)
INFO [mavlink] | 평균 메시지 처리 시간: 765 ns (0.77 us)
INFO [mavlink] | pps: 1.01 packets/sec
INFO [mavlink] | BPS: 72.47 bytes/sec
INFO [mavlink] -----
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 1062 ns (1.06 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 942 ns (0.94 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 781 ns (0.78 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 541 ns (0.54 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 651 ns (0.65 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 681 ns (0.68 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 662 ns (0.66 us)
```

[그림 45] heartbeat 메시지 처리 시간 성능 측정 출력의 일부

먼저, 서명 기능 활성화 전의 메시지 처리 시간 성능 측정 결과는 아래와 같다.



[그림 46] 서명 활성화 전 각 heartbeat 메시지에 대한 처리 시간 그래프

메시지 처리 수	min (μs)	max (μs)	avg (μs)	pps (packets/s)	BPS (byte/s)
100	0.41	2.01	0.81	1.02	73.41
200	0.41	2.01	0.77	1.01	72.70
300	0.41	2.01	0.77	1.01	72.47
400	0.41	2.34	0.77	1.00	72.35
500	0.41	36.54	0.83	1.00	72.28

[표 8] 서명 기능 활성화 전의 heartbeat 메시지 처리 성능 측정 결과

[표 8]의 측정 결과의 특징에 대하여 총 세 가지를 설명하자면, 첫 번째는 400번의 메시지 처리까지 안정적인 처리 속도를 보이고 있다는 것이다. 안정적으로 메시지가 처리된다면 약 0.8μs의 평균 처리 시간을 가진다는 것을 알 수 있다.

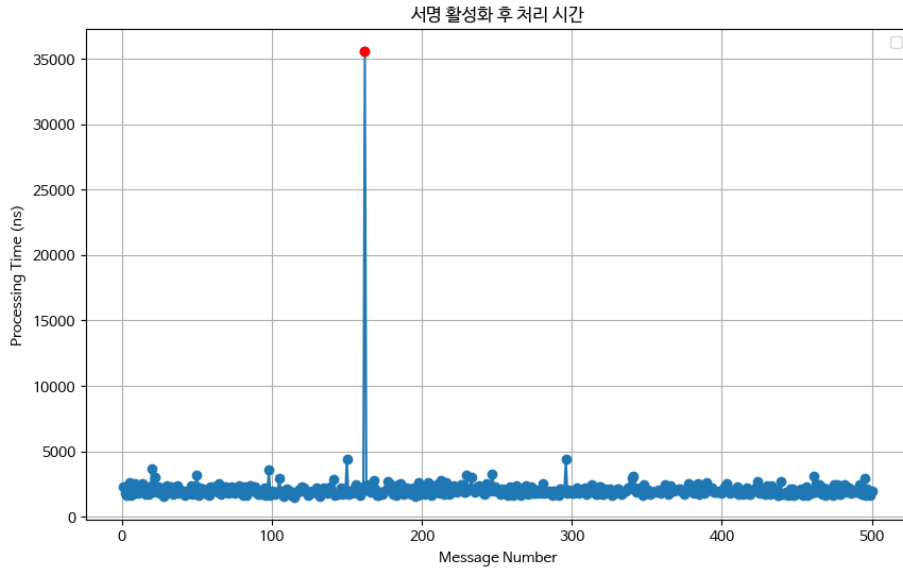
두 번째는, pps와 BPS가 각각 "1.00"과, "72.00"으로 수렴하고 있는 것을 볼 수 있는데, 이는 처음 PX4와 QGC가 연결되면서 heartbeat 메시지의 전송이 여러 번 이루어졌기 때문이다. heartbeat 메시지는 1초 당 1번의 메시지가 전달되므로 pps가 "1.00" 값으로 수렴되고, heartbeat 메시지 패킷의 길이가 72 byte이기 때문에 BPS가 "72.00" 값으로 수렴되는 것이다.

세 번째는, 400번과 500번 메시지의 사이에서 max 값이 36.54 μs로 오른 것을 확인할 수 있다. 이는 시간 동기화 문제 등으로 인하여 처리 시간이 급증한 것으로 보인다. 단 한 번의 급증 말고는 모두 평균에서 크게 벗어나지 않는 속도로 메시지를 처리하였다.

```
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 812 ns (0.81 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 511 ns (0.51 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 621 ns (0.62 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 621 ns (0.62 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 592 ns (0.59 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 621 ns (0.62 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 902 ns (0.90 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 621 ns (0.62 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 1072 ns (1.07 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 731 ns (0.73 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 36539 ns (36.54 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 511 ns (0.51 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 591 ns (0.59 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 521 ns (0.52 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 420 ns (0.42 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 531 ns (0.53 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 631 ns (0.63 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 812 ns (0.81 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 451 ns (0.45 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 1222 ns (1.22 us)
INFO [mavlink] 메시지 처리 1회에 걸린 시간: 410 ns (0.41 us)
```

[그림 47] 서명 기능 활성화 전의 heartbeat 메시지 처리 시간 급증 케이스

다음으로, 서명 기능 활성화 후의 메시지 처리 시간 성능 측정 결과는 아래와 같다.



[그림 48] 서명 활성화 후 각 heartbeat 메시지에 대한 처리 시간 그래프

메시지 처리 수	min (μs)	max (μs)	avg (μs)	pps (packets/s)	BPS (byte/s)
100	1.58	3.72	2.07	1.00	72.29
200	1.51	35.59	2.21	1.00	72.15
300	1.51	35.59	2.17	1.00	72.10
400	1.51	35.59	2.13	1.00	72.07
500	1.51	35.59	2.11	1.00	72.06

[표 9] 서명 기능 활성화 후의 heartbeat 메시지 처리 성능 측정 결과

[표 9]의 측정 결과의 특징에 대하여 총 두 가지를 설명하자면, 첫 번째는 100번까지의 메시지 처리에서 2.07 μs의 평균 처리 시간을 보여준 이후로, 100번과 200번 사이에서 max 값이 급증하며 평균 처리 시간도 함께 상승한다. 이후 다시 처리 시간의 급증 없이 안정적인 처리 속도를 보여, 500번 메시지에서 평균 처리 시간이 2.11 μs가 된다. 이를 미루어 보아, 메시지가 안정적으로 처리된다면 평균 처리 시간을 약 2.10 μs로 예상할 수 있다.

두 번째는, 서명 기능 활성화 전과 마찬가지로 pps와 BPS의 값이 "1.00"과 "72.00"으로 수렴

하고 있다. 여기서 BPS의 초기값이 "72.00"이 아닌 이유는 서명 기능 활성화로 인해 PX4와 QGC 사이의 재연결이 일어났기 때문에 heartbeat 메시지가 더 전달된 것으로 보았다.

heartbeat 메시지 처리 성능 측정 결과를 종합하자면, 서명 기능 활성화 전의 평균 처리 시간은 약 0.8 μ s로, 서명 기능 활성화 후의 평균 처리 시간은 약 2.1 μ s로 볼 수 있다. 이는 약 2.6배만큼 처리 시간이 증가한 것이고, 차이를 계산하면 서명 검증에 걸리는 시간은 약 1.3 μ s 라고 할 수 있다.

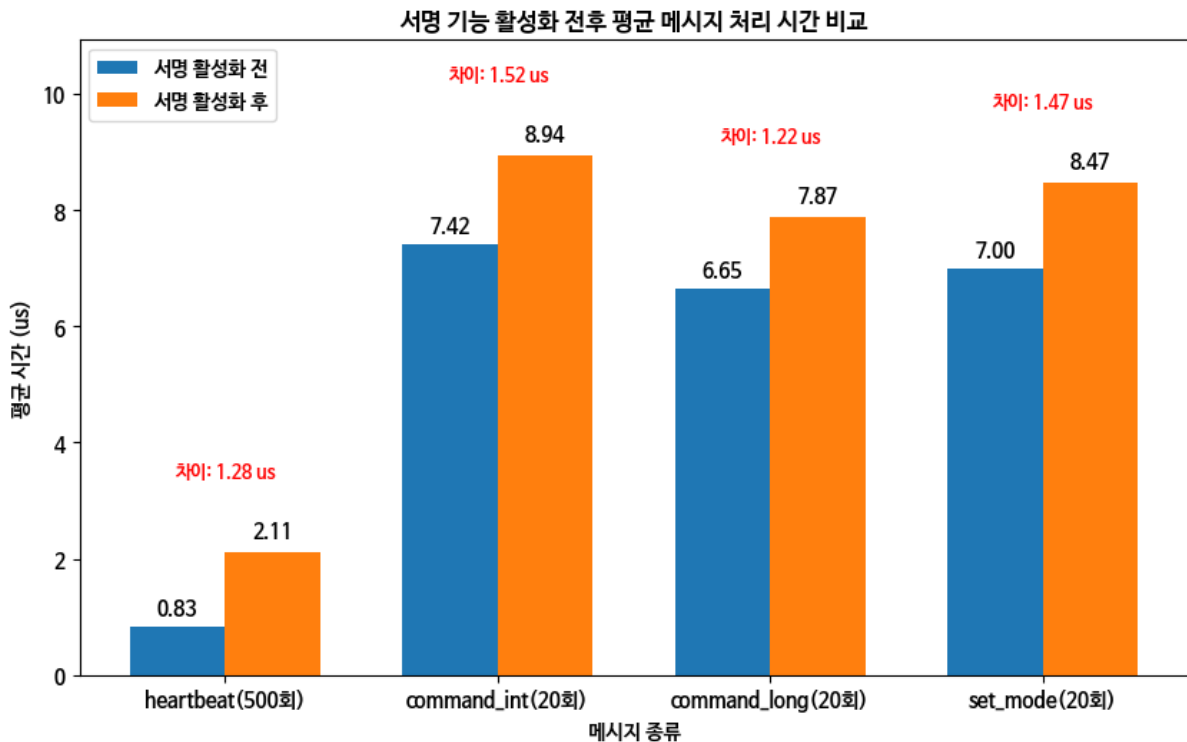
다음은 heartbeat 메시지와 동일한 성능 측정 방법을 적용하여, 드론의 조종과 관련된 command_int, command_long, set_mode 메시지에 대해서도 처리 시간을 측정해 보았다. command_int 메시지는 지도 상에서 좌표를 지정하여 이동할 때 전송되는 "MAV_CMD_DO_REPOSITION" 커맨드에 대하여, command_long 메시지는 takeoff 명령을 내릴 때 전송되는 "MAV_CMD_NAV_TAKEOFF" 커맨드에 대하여, 그리고 set_mode 메시지는 land 명령을 주었을 때를 기준으로 측정하였다. 각 메시지에 대한 처리 시간은 20회씩 측정되었으며, 측정 결과에 대한 표는 아래와 같다.

메시지 종류	서명 기능 활성화 여부	min (μ s)	max (μ s)	avg (μ s)
command_int (reposition)	비활성화	5.90	10.50	7.42
	활성화	7.10	11.29	8.94
command_long (takeoff)	비활성화	5.40	9.30	6.65
	활성화	6.20	10.70	7.87
set_mode (land)	비활성화	5.60	9.90	7.00
	활성화	6.60	12.19	8.47

[표 10] 드론 조종 관련 명령어들의 메시지 처리 성능 측정 결과(각 20회씩 측정)

[표 10]의 각 메시지에 대하여 서명 기능 활성화 전후의 평균값 차이를 살펴보면, 우선 command_int는 1.52 μ s의 차이를, command_long은 1.22 μ s의 차이를, set_mode는 1.47 μ s의 차이를 나타낸다. 이는 조종 관련 메시지에서도 약 1.3 μ s의 차이를 보였던 heartbeat 메시지와 서명 검증에 걸리는 시간이 유사하다는 것을 보여준다. 또한, 서명 기능을 활성화했을 때 약 1.2배, 1.18배, 1.21배만큼 처리 시간이 증가하였는데, 2.6배만큼 처리 시간이 증가했던 heartbeat 메시지보다 낮은 증가폭을 보였음을 알 수 있다.

메시지 처리 시간 성능 측정 결과를 종합하여, heartbeat, command_int, command_long, set_mode에 대한 서명 기능 활성화 전후의 평균값을 그래프로 정리하면 아래와 같다.



[그림 49] 서명 기능 활성화 전후 평균 메시지 처리 시간 비교

여러 메시지들에 대하여 서명 검증에 약 1.2~1.5 μ s 정도의 오버헤드가 나타났다. 서명 기능을 활성화함으로써 MAVLink 메시지의 무결성을 검증하고, 공격자의 드론 제어를 방어하여 보안성을 높일 수 있다는 점에서, 충분히 수용할 만한 오버헤드로 보인다.

4.2. 멘토 의견서 대응 방안

● 의견 1 : ECDH 키 교환 후 무결성 보장하는 방안

- SHA256 알고리즘을 적용하여 HMAC을 통해 서명을 생성 및 검증함으로써 무결성을 보완하였다.

● 의견 2 : ECDSA 적용

- 기존 목표에서의 고려 사항이었으나, 시간적인 문제로 인하여 적용하지 못하였다. 향후 연구를 통하여 ECDSA 뿐만 아니라, EDDSA, Poly1305 등 서명 인증에 사용되는 여러 알고리즘을 적용하여 가장 적합한 알고리즘을 찾아낼 수 있도록 할 것이다.

5. 결론 및 향후 연구 방향

PX4와 QGC에서 ECDH 알고리즘을 적용하여 공통된 비밀 키를 갖게 하였고, 서명 생성에 해당 키를 활용하여 SHA256 알고리즘을 적용시켜 서명 생성 및 검증 과정을 구현하였다. 이후, 서명 검증에서의 오버헤드를 측정하여 서명 기능을 적용하는 것이 시스템 성능에 미치는 영향을 평가하였다.

특히, PX4와 QGC에서 서명 기능을 활성화할 때 하나의 기체에서는 가장 처음 연결되는 GCS만 서명 스트림에 추가하도록 구현하였기 때문에, 공격자가 무선 네트워크에 있는 PX4와 QGC를 cracking 하여 연결에 성공하였다 하더라도, 서명 스트림이 이미 설정되어 있는 상태이거나, 공격자가 key를 모르는 상태라면 드론을 제어할 수 없게 된다.

향후 연구가 진행된다면, MAVLink 프로토콜을 이용하여 public key 전송에 실패하였던 부분에 대해 보완하고, 기밀성까지 제공할 수 있는 방향으로 시스템을 구현할 것이다. 이를 통해 통신의 무결성과 기밀성을 동시에 확보할 수 있는 방안을 모색할 것이다. 추가적으로, 서명 생성 및 검증을 하는 과정에서의 오버헤드를 최소화하기 위하여 ECDSA, EDDSA, Poly1305, blake2b 등 여러 알고리즘을 적용한 뒤 가장 적합한 알고리즘을 찾아내고, 혹은 알고리즘을 모두 적용한 뒤 각각의 Power 사용량, CPU 부하, RAM 사용량 등을 확인하여 어떤 알고리즘을 사용하여 서명을 생성할지 선택할 수 있는 환경을 구현해 볼 수 있을 것이다.

6. 참고 문헌

- [1] Tae-Wan Kim, Se-Yoon Lee, Seo-Woo Jung, Han-saem Wi, and Ok-yeon Yi, "A Research on the Security of Drone Control Data Using Quantum Entropy-Based Random Number Generator," Journal of the Korea Institute of Information Security & Cryptology, vol. 31, no. 2, pp. 133-144, Apr. 2021. (in Korean)
- [2] PX4-Autopilot github [Online]. Available: <https://github.com/px4/px4-autopilot/>
- [3] QGroundControl github [Online]. Available: <https://github.com/mavlink/qgroundcontrol>
- [4] MAVLink Developer Guide [Online]. Available: <https://mavlink.io/en/>
- [5] ROS Wiki [Online]. Available: <https://wiki.ros.org/>
- [6] Hyeong-Min Kim, Dae-Woo Lee, "Integrated Simulation Environment for Heterogeneous Unmanned Vehicle using ROS and Pixhawk," J. Korean Soc. Aviat. Aeronaut., Vol. 27, No. 3, pp. 1-14, Sep. 2019. (in Korean)
- [7] mavlink_sha_256.h. Mavlink C Library v2. github [Online]. Available: https://github.com/mavlink/c_library_v2/blob/master/mavlink_sha256.h
- [8] Eva Kupcova, Patrik Zelenak, Matus Pleva, Milos Drutarovsky, "Optimization of Ristretto255 Group Implementation for Cortex-M4 based Cryptographic Applications", IEEE, 2024