

UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA



Corso di Laurea Magistrale in Informatica

**Securing MAVLINK protocol: a Post
Quantum cryptography-based approach**

ANNO ACCADEMICO 2022/2023

RELATORE

Prof. Arcangelo Castiglione

Università degli Studi di Salerno

CANDIDATO

Hermann Senatore

Matricola: **0522501273**

We're flying high

We're watching the world pass us by

Never want to come down

Never want to put my feet back down on the ground

(Depeche Mode)

Indice

1 Introduzione	5
1.1 Scopi e struttura della tesi	5
1.2 UAV, APR o Droni: nomenclatura	6
1.3 Cenni storici	6
1.4 Sviluppi recenti	7
2 Piattaforme software per UAV e GCS	8
2.1 ArduPilot	9
2.1.1 SITL - Software in the Loop	11
2.1.2 ArduPilot: architettura software	12
2.1.3 Compilare ArduPilot	12
2.1.4 ArduPilot: integrazione con il protocollo MAVLINK	14
2.2 QGroundControl	16
2.2.1 Panoramica sulle principali GCS disponibili	16
2.2.2 Architettura software di QGroundControl	17
2.2.3 Ottenere QGroundControl	18
3 Il protocollo MAVLINK	21
3.1 Principi di funzionamento	21
3.2 Architettura di un pacchetto MAVLINK	24
3.2.1 MAVLINK 1.0: struttura di un pacchetto	24
3.2.2 MAVLINK 2.0: struttura di un pacchetto	26
3.2.3 La firma digitale in MAVLINK 2.0	28
3.2.4 Algoritmo di generazione della firma digitale in MAVLINK 2.0	30
3.2.5 Scelta della versione del protocollo	31
4 Sicurezza del protocollo MAVLINK	33
4.1 Possibili attacchi contro MAVLINK	33
4.2 Possibili contromisure	34
4.2.1 MAVSec	35

INDICE

4.2.2	Securing Unmanned Aerial Vehicles by Encrypting MAVLINK Protocol	36
4.2.3	Sicurezza in ambienti resource-constrained: una breve introduzione al concetto di lightweight cryptography	37
4.2.4	MAVLINK e lightweight cryptography	39
5	Quantum Computing e Post-Quantum Cryptography	41
5.1	Cos’è il Quantum Computing	41
5.2	Il problema: RSA, DH e l’algoritmo di Shor	43
5.3	La soluzione: Post-Quantum Cryptography	45
5.4	L’algoritmo Kyber	47
5.4.1	KEM - Key Encapsulation Mechanism	47
5.4.2	Reticoli	50
5.4.3	Il problema LWE	52
5.4.4	Kyber : generazione delle chiavi	55
5.4.5	Kyber : la cifratura	56
5.4.6	Kyber : la decifratura	57
5.4.7	Kyber : discussione sulla sicurezza	58
6	Progettazione e descrizione della soluzione proposta	60
6.1	Scambio della chiave: perché Kyber ?	60
6.2	Cifratura simmetrica: perché AES ?	61
6.3	Definizione del protocollo	61
7	Implementazione della soluzione proposta	64
7.1	Step 1: installazione di un’implementazione di Kyber nell’ambiente di sviluppo	64
7.1.1	liboqs : descrizione ed installazione	65
7.2	Step 2: Introduzione di nuovi messaggi MAVLINK 2.0	68
7.2.1	KEYEXCHANGE	68
7.2.2	KEYEXCHANGEGCSACK	70
7.2.3	CAPSULE e CAPSULEACK	70

INDICE

7.3	Step 3: Integrazione di Kyber all'interno di ArduPilot	72
7.4	Step 4: Integrazione di Kyber all'interno di QGroundControl . .	73
7.5	Step 5: Alterazione delle funzioni di ArduPilot	74
7.5.1	ArduCopter/GCS_MAVLink.cpp	74
7.5.2	libraries/GCS_MAVLink/GCS_Common.cpp	75
7.6	Step 6: Alterazione delle funzioni di QGroundControl	82
7.6.1	src/comm/MAVLinkProtocol.cc	82
7.6.2	Invio di KEYEXCHANGEGCSACK	83
7.6.3	Invio di CAPSULEACK	84
7.6.4	Riassumendo	87
7.7	Step 7: Alterazione della libreria MAVLINK	88
7.7.1	AES: implementazione	88
8	Analisi delle performance	92
9	Conclusioni e sviluppi futuri	95
	Riferimenti bibliografici e risorse consultate	96

INDICE

Abstract

L'evoluzione tecnologica a cui si sta assistendo negli ultimi anni sta rivoluzionando pesantemente (tra le altre cose) il mondo dell'aviazione, merito anche (e soprattutto) dei cosiddetti **UAV** (unmanned aerial vehicle), che comunemente vengono definiti **droni**, impiegati sia in contesto "civile" che in contesto militare.

La potenziale delicatezza delle missioni che questi veicoli si trovano ad affrontare suggerisce dunque la necessità di definire dei requisiti di sicurezza che ne permettano un impiego più agevole. L'innovazione tecnologica porta però a nuove sfide anche nel campo della sicurezza.

In particolare, i recenti progressi nel campo del **quantum computing** aprono nuove sfide nel contesto della crittografia, rendendo quindi necessario sviluppare nuove tecniche resistenti ad attacchi veicolati mediante computer quantistici. In questo lavoro viene presentato un **proof of concept** di un'architettura basata sul protocollo MAVLINK che permetta una comunicazione sicura tra un drone e la sua **Ground Control Station** e, ad un livello più alto, la definizione della chiave di cifratura utilizzata mediante il **Key Encapsulation Mechanism** Kyber, selezionato dal **NIST** come lo standard per quanto riguarda gli algoritmi di incapsulamento **quantum resistant**.

1 Introduzione

1.1 Scopi e struttura della tesi

Il presente lavoro si concentra, come tra l’altro già accennato in precedenza nell’abstract, sull’analisi del proof of concept di un’architettura basata sul protocollo MAVLINK che permetta lo scambio di chiavi e la comunicazione cifrata tra un UAV e la sua centrale di comando a terra utilizzando il KEM quantum-resistant Kyber e l’algoritmo AES in modo tale da proteggere lo scambio di messaggi tra i due *endpoint* della comunicazione.

In particolare:

- Questo capitolo funge da introduzione al lavoro svolto durante l’attività di Tesi, ne illustra la struttura e ne chiarisce le motivazioni ed il contesto in cui è calato;
- il **Capitolo 2** fornisce una panoramica sulle principali piattaforme software considerate nel presente lavoro;
- il **Capitolo 3** fornisce una panoramica sul protocollo **MAVLINK**;
- il **Capitolo 4** discute alcune problematiche di sicurezza che affliggono il protocollo MAVLINK e discute alcuni approcci a questo problema presenti in letteratura;
- il **Capitolo 5** fornisce un’introduzione all’impatto dell’avvento dei **Computer Quantistici** nel campo della crittografia e al concetto di **Post-Quantum Cryptography** (PQC). Si procederà quindi ad una panoramica dell’algoritmo **Kyber**;
- il **Capitolo 6** illustra in dettaglio le **modifiche** effettuate alle piattaforme software considerate nel presente lavoro ed al protocollo MAVLINK stesso;
- il **Capitolo 7** fornisce le **conclusioni** al presente lavoro e illustra alcuni possibili **sviluppi futuri**.

1.2 UAV, APR o Droni: nomenclatura

Nell'ultimo decennio, l'importante evoluzione tecnologica nel contesto dell'aviazione ha permesso la progettazione e la concreta realizzazione di veicoli in grado di volare e compiere missioni anche **senza la presenza di un pilota umano** sempre più versatili, efficaci e precisi. Questa tipologia di veicoli viene definita, a seconda del contesto linguistico in cui ci si trova, **UAV** (*unmanned aerial vehicle*), **APR** (*aeromobile a pilotaggio remoto*) o, più comunemente, **Drone**. Tutti questi acronimi sono, quindi, equivalenti tra di loro.

La presenza di un essere umano continua tuttavia ad essere fondamentale in quanto il pilotaggio di questi dispositivi viene effettuato mentre una struttura di controllo "a terra", che prende il nome di **Ground Control Station** (da qui in poi *GCS*). Tipicamente, una GCS può essere rappresentata da un qualsiasi apparato in grado di comunicare in qualche modo con l'UAV, quindi anche un comune Personal Computer su cui viene posto in esecuzione un software specifico.

1.3 Cenni storici

Il concetto di aeromobile senza pilota in sé non è sorprendentemente prerogativa degli ultimi anni e delle conseguenze che l'avvento delle tecnologie informatiche si porta dietro. Risale infatti agli Anni '40 del XIX secolo il primo (rudimentale!) impiego di "dispositivi" volanti senza pilota in campo militare.

Per fronteggiare i moti rivoluzionari, peraltro diffusi anche in tutta Europa nel 1848, nella città di Venezia (che avevano portato alla creazione della cosiddetta Repubblica di San Marco), l'esercito austriaco lanciò dei **palloni** a cui era stato fissato dell'**esplosivo** dalla nave "Vulcano".

Questo primo esperimento portò a risultati "misti": alcuni di questi dispositivi riuscirono effettivamente a colpire la città, altri furono invece deviati dal vento.

Per tutto il XIX secolo, lo sviluppo di questo tipo di dispositivi rimase prerogativa militare, con gli americani e gli inglesi che si occuparono dello

sviluppo dei primi aeromobili comandati tramite radiofrequenza. Questi ultimi, nel 1917, testarono con successo un velivolo chiamato **Aerial Target**, mentre gli americani misero a punto il cosiddetto **Kettering Bug** [1].

Nessuno di questi due prototipi venne tuttavia impiegato nella **prima guerra mondiale**.

Come già accennato in precedenza, con il passare del tempo e con il sempre più spinto progresso tecnologico i droni e, più in generale i veicoli a guida remota, hanno trovato un fertile campo di applicazione anche nell'ambito **civile**. Ad esempio, nel 2022 è stato proposto uno studio in cui è stato utilizzato un insieme di veicoli senza pilota per monitorare il processo di realizzazione di un **ponte** [2]. Si è assistito, riassumendo, ad una "democratizzazione" dell'uso dei droni negli ambiti più disparati.

1.4 Sviluppi recenti

Proprio questa diffusione più capillare dell'uso di suddetti veicoli a guida remota ha portato ad una più grande attenzione nello sviluppo di tecnologie software specifiche a questo ambito.

A partire dagli anni 10 del XXI secolo sono stati concepiti diversi progetti software **open source** che forniscono piattaforme integrate compatibili con diversi tipi di apparati sia lato UAV che lato GCS.

Parimenti, è stato necessario definire uno *standard* per quanto riguarda l'ambito dei protocolli di comunicazione tra il drone e la sua *GCS*, che è rappresentato sicuramente dal protocollo **MAVLINK** e dalle sue successive evoluzioni.

Una panoramica più approfondita sul protocollo MAVLINK verrà in ogni caso affrontata nelle prossime sezioni in cui verranno illustrate le piattaforme software utilizzate nell'ambito del presente lavoro.

2 Piattaforme software per UAV e GCS

Il presente capitolo fornisce una panoramica delle principali tecnologie utilizzate in ambito **civile** ed **amatoriale** nel contesto dei velivoli a pilotaggio remoto. In particolare viene riservata particolare attenzione ai seguenti due progetti **open source**:

- **ArduPilot**: una piattaforma integrata scritta in **C++** utilizzabile su diverse tipologie di veicoli non necessariamente adatti al volo;
- **QGroundControl**: una **Ground Control Station** open source scritta in **C++**, che utilizza il framework **Qt** e che permette una più stretta interazione con la piattaforma ArduPilot.



Il logo di ArduPilot



Il logo di QGroundControl

Viene quindi svolta una panoramica sul protocollo di comunicazione **MAVLINK** che permette l'interazione tra queste due componenti.



Il logo del protocollo MAVLINK

2.1 ArduPilot

ArduPilot, come accennato, è una suite software universale che permette il controllo di diversi veicoli **non necessariamente volanti** [3] la cui nascita risale al 2007. L'idea di una suite software quale ArduPilot è stata formalizzata presso la piattaforma **DIYDrones.com**, community dedicata agli UAV fondata da **Chris Anderson** e che si definisce quindi "The Birthplace of ArduPilot"[4].

Nel 2009 venne prodotta la prima **board** che utilizzava questa piattaforma e nel novembre dello stesso anno il codice sorgente del progetto venne reso pubblico (ed è attualmente disponibile su GitHub all'url <https://github.com/ArduPilot/ardupilot>)[5].

Correntemente, il progetto permette l'utilizzo su diverse tipologie di dispositivi, mediante delle versioni del firmware leggermente diverse tra di loro. In particolare, ArduPilot supporta dispositivi del tipo:

- **Copter**: probabilmente quello più diffuso tra tutti e che tipicamente è composto da una board a cui sono associate diverse **eliche**. Il progetto di riferimento si chiama **ArduCopter**[6];
- **Plane**: un tipo di dispositivo ad **ala fissa**, tipicamente un **aereo radiocomandato**. Il progetto di riferimento si chiama **ArduPlane**[7];
- **Rover**: un tipo di dispositivo che non può spiccare il volo ma dotato di ruote e che di conseguenza viene utilizzato per missioni "a terra". Il progetto di riferimento si chiama semplicemente **Rover**[8];
- **Sub**: un tipo di dispositivo subacqueo, appartenente alla categoria degli **Autonomous Underwater Vehicles (AUV)**. Il progetto di riferimento si chiama **ArduSub**[9];



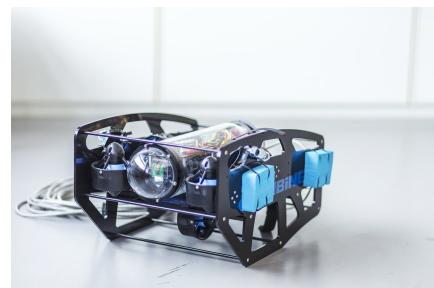
(a) Copter



(b) Plane



(c) Rover



(d) Sub

Il progetto fornisce anche del codice specifico per apparati "complementari" ai dispositivi qui sopra descritti. In particolare:

- **AntennaTracker**: che permette di governare apparati che si occupano di ricevere il segnale proveniente dai dispositivi controllati da remoto[10];
- **Blimp**: un particolare tipo di "pallone" aerostatico dotato di motore[11].



AntennaTracker



Blimp

2.1.1 SITL - Software in the Loop

Il progetto ArduPilot mette inoltre a disposizione anche un **emulatore** che permette il testing di tutte queste varianti della piattaforma **senza possedere l'hardware adatto** [12] e quindi procedere al **flashing** del firmware.

Tale emulatore prende il nome di **SITL** (*Software In The Loop*) e risulta quindi molto utile quando è necessario svolgere attività di testing preliminari per valutare il comportamento del firmware in use case differenti.

Tra l'altro, proprio l'utilizzo di SITL ha permesso la realizzazione di questo lavoro in totale autonomia da hardware specifico.

Maggiori informazioni su SITL e sul suo utilizzo saranno proposte in una successiva sottosezione dove verranno presentati ulteriori dettagli sull'architettura software del progetto ArduPilot.

2.1.2 ArduPilot: architettura software

Come menzionato in precedenza, il progetto ArduPilot utilizza il linguaggio di programmazione **C++**, che è uno di quelli più utilizzati per la programmazione di sistemi embedded in generale poiché **gira direttamente sull'hardware** senza bisogno di macchine virtuali di sorta (al contrario di Java o di linguaggi interpretati), permettendo di ottenere prestazioni migliori in un contesto in cui proprio le prestazioni rappresentano un requisito fondamentale.

Dal punto di vista della configurazione del progetto, la scelta degli sviluppatori di ArduPilot è ricaduta sul framework **waf**. Tale framework, scritto in Python, offre un ambiente di build modulare e per quanto possibile agnostico rispetto ai linguaggi di programmazione utilizzati nel progetto col quale lo si vuole usare.

In breve, la documentazione di waf [13] suggerisce che:

- Per utilizzare il framework è necessaria unicamente un'installazione di **Python**;
- Waf non definisce un nuovo "linguaggio" (come avviene nel meccanismo dei **Makefile**) ma si compone di moduli scritti in Python, che permette quindi una maggiore riusabilità delle componenti;
- I "target" sono definiti come oggetti python dichiarati in maniera separata dai comandi (definiti invece come funzioni in un file chiamato **wscript**).

2.1.3 Compilare ArduPilot

Il progetto GitHub della piattaforma ArduPilot contiene il codice sorgente specifico di tutte le tipologie di veicolo, organizzato in directory separate[14]. In particolare:

- ArduCopter/ contiene il codice sorgente specifico per i veicoli di tipo **Copter**;
- ArduPlane/ contiene il codice sorgente specifico per i veicoli di tipo **Plane**;
- ArduSub/ contiene il codice sorgente specifico per i veicoli di tipo **Sub**;
- Rover/ contiene il codice sorgente specifico per i veicoli di tipo **Rover**;
- AntennaTracker/ contiene il codice sorgente specifico per i dispositivi che comandano **antenne di invio/ricezione**;
- Blimp/ contiene il codice sorgente specifico per i veicoli di tipo **Blimp**, come descritti in precedenza;

Inoltre, mediante il comando `configure` ed il parametro `--board=` è possibile personalizzare ulteriormente la compilazione del firmware adattandolo ad un *tipo di board* specifico o istruendo il sistema a prepararsi ad essere utilizzato mediante **SITL**[15]. Quest'ultimo scenario è quello utilizzato nel presente lavoro e si ottiene eseguendo il comando:

```
./waf configure --board=sitl
```

e, per generare il firmware specifico per un determinato tipo di veicolo (in questo caso quello di elezione è **Copter**), è necessario eseguire il seguente comando:

```
./waf copter
```

Per avviare l'emulatore **SITL** configurato in precedenza è necessario recarsi nella directory del veicolo per cui si è scelto di compilare il firmware (specificato nel comando precedente) (in questo caso, **ArduCopter/**)

```
cd ArduCopter/
```

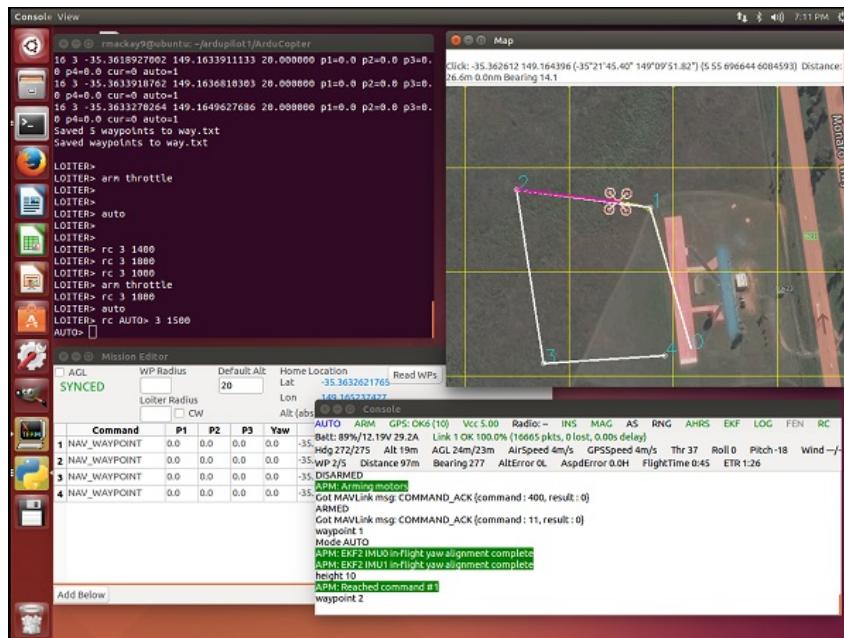
e finalmente eseguire l'emulatore mediante il comando

sim_vehicle.py

Nota: il file appena descritto è presente nel percorso `/Tools/autotest` ma non è necessario specificare il suo path assoluto perché durante l'installazione dei prerequisiti di ArduPilot, quest'ultimo viene aggiunto in maniera system-wide alla variabile d'ambiente PATH.

A questo punto, il veicolo "virtuale" è pronto ad accettare connessioni da una GCS. In particolare, viene aperta la porta **5760** sull'interfaccia di **loopback** locale.

È anche possibile modificare il numero di porta utilizzato e l'interfaccia su cui mettersi in ascolto.



SITL in esecuzione

2.1.4 ArduPilot: integrazione con il protocollo MAVLINK

Il progetto ArduPilot usa il protocollo MAVLINK per comunicare con le Ground Control Stations. Il codice sorgente che implementa le primitive e le funzioni di supporto per la comunicazione non è però già disponibile all'atto della clonazione del progetto da GitHub. Al contrario, è stato configurato il tool `mavgen` come *submodule* all'interno della repository. Tale tool, richiamato

dal sistema di building `waf` all'atto della compilazione del codice per un determinato veicolo, permette di generare *dinamicamente* tutto il codice di basso livello necessario per le operazioni di comunicazione verso l'esterno[16]. Più nello specifico, viene generata una libreria *headers-only* in linguaggio **C**, che tuttavia risulta perfettamente interoperabile con il resto del software scritto invece, come già menzionato, in **C++**. Maggiori dettagli su questo tool saranno forniti durante la panoramica che verrà svolta riguardo il protocollo MAVLINK.

2.2 QGroundControl

2.2.1 Panoramica sulle principali GCS disponibili

Come menzionato in precedenza, un UAV per essere totalmente operativo ha bisogno di una componente "a terra" che lo governi: questa componente prende appunto il nome di **Ground Control Station** (GCS) e si occupa di gestire la connessione con l'UAV mediante il protocollo più adatto e di controlarlo mediante messaggi tipicamente aderenti alle specifiche **MAVLINK** (maggiori informazioni sono fornite nella sottosezione dedicata). È quindi pacifico affermare che il software in questione risulti essere nella maggior parte dei casi **molto complesso** dal punto di vista dei requisiti a cui deve rispondere e da quello dell'architettura software.

Attualmente, sono disponibili pubblicamente diverse Ground Control Station più o meno sofisticate. Tra queste si annoverano:

- **MAVProxy** [17]: una (non necessariamente) command-line-based GCS che possiede tutti gli strumenti necessari a comandare un UAV risultando al contempo utilizzabile su sistemi con a disposizione relativamente poche risorse. Di default, è quella che si avvia al momento dell'esecuzione di **SITL** a meno che non venga passato il parametro `--no-mavproxy` allo script `sim_vehicle.py` (come accennato in precedenza);
- **Mission Planner** [18], sviluppato da **Michael Orbone** e che permette di gestire diversi aspetti di una missione di un UAV: dalla configurazione dei waypoints (in termini di coordinate GPS) ai comandi da far eseguire all'UAV durante lo svolgimento della missione;
- **APM Planner 2**: sostanzialmente **un'evoluzione** dello strumento Mission Planner, creato dal team di sviluppo di **ArduPilot** e che permette di gestire sia dispositivi basati su ArduPilot stesso che basati sulla piattaforma **PX4**;
- **QGroundControl** [19]: probabilmente la GCS più funzionale tra quelle appena descritte. È stata sviluppata dal team che si occupa del pro-

tocollo **MAVLINK** e che permette di interfacciarsi con qualsiasi dispositivo che supporti i messaggi MAVLINK. Tale piattaforma è stata utilizzata per realizzare parte del lavoro di tesi qui presentato.

2.2.2 Architettura software di QGroundControl

Come accennato, QGroundControl è un software molto complesso e che fornisce una moltitudine di funzionalità, che vanno dalla pianificazione delle missioni al monitoraggio dei messaggi MAVLINK tramite un **inspector ad-hoc**, per citarne alcune.

Il progetto è stato sviluppato dall'organizzazione che si occupa di MAVLINK e, più in generale, del **MAV-SDK**: la **DroneCode Foundation**, che ha come missione quella di "creare uno standard nell'industria dei droni mediante progetti open-source"[20].

È scritto interamente in **C++**, eccetto per lo stack di comunicazione MAVLINK in uso, mediante l'ausilio del framework **Qt** ed il suo codice sorgente è disponibile su GitHub [21].

Al contrario delle altre GCS, QGroundControl **supporta anche la compilazione per la piattaforma Android**[22], il che di conseguenza permette l'utilizzo del software su dispositivi portatili e rendendo il progetto completamente **cross-platform**.

In ogni caso, il *core* della piattaforma è presente all'interno della directory **src/** nella root della repo GitHub. Qui sono presenti tutti i moduli che permettono, tra le altre cose, la gestione dei veicoli connessi alla GCS (classe **Vehicle** coadiuvata da un'istanza di **LinkManager**) e dei messaggi di cui è stato svolto il *parsing* dalla libreria MAVLINK (classe **MAVLinkProtocol**: presente nella sottodirectory **comm/**).

La comunicazione con l'UAV connesso avviene mediante **datagrammi UDP** gestiti da un **LinkManager** che contengono messaggi MAVLINK costruiti utilizzando le funzioni della libreria C di questo progetto.

Al contrario di quanto avviene nel contesto di ArduPilot, con QGroundControl suddetta libreria **non viene generata** al momento della compilazione

ma viene fatto uso dell'**implementazione di riferimento in C** [23] messa a disposizione pubblicamente da DroneCode su GitHub ed inclusa nella repo di QGroundControl utilizzando (ancora una volta) il meccanismo dei *submodules*.

Il "dialetto" utilizzato di default da QGroundControl per la libreria MAVLINK è (a partire dal 24 agosto 2023)[24] **all**, ma tale aspetto può essere variato in uno dei **file di configurazione del progetto**: `QGCExternal-Libs.pri`.

Per quanto riguarda questo lavoro, invece, viene ancora utilizzato il "dialetto" `ArduPilotMega` poiché tale cambiamento non era stato proposto ed approvato.¹

2.2.3 Ottenere QGroundControl

La natura open source e cross platform di tale progetto permette a chi è interessato di ottenerne una copia sostanzialmente in due modi:

1. Utilizzando una **release** periodica precompilata;
2. Compilando (ed eventualmente modificando!) il codice sorgente, approccio utilizzato tra l'altro nel presente lavoro.

Per quanto riguarda la (1.), pacchetti precompilati sono disponibili nella sezione "release" di GitHub, dove sono presenti di solito:

- Un file `.exe` che permette l'installazione su Microsoft Windows;
- Un'immagine disco `.dmg` montabile ed installabile su macOS;
- Un'*AppImage* compatibile la maggior parte delle distro GNU/Linux;
- Un file apk installabile su Android.

¹Cambiamenti così importanti sono particolarmente frequenti in progetti open source del genere, ndr

Per quanto riguarda la (2.), sono naturalmente necessari particolari accorgimenti, oltre ad una piattaforma che permetta di compilare agevolmente un progetto di grandi dimensioni.²

In particolare:

1. È necessario installare il framework **Qt**, mediante l'online installer oppure tramite tool di terze parti [25]. La versione di Qt necessaria attualmente è, secondo quanto indicato [26], la 5.15.2;
2. Tramite il gestore pacchetti `apt` è necessario installare i seguenti pacchetti:
 - `speech-dispatcher`;
 - `libudev-dev`;
 - `libsdl2-dev`;
 - `patchelf`;
 - `build-essential`;
 - `curl` (che probabilmente è già installato di default).

Successivamente è necessario preparare l'ambiente di build mediante i seguenti comandi nella *root* della repository:

```
mkdir build/ && cd build/  
qmake ../
```

L'invocazione di `qmake` permette a Qt di inizializzare i **Makefile** ed di caricare le **variabili d'ambiente** contenute nei **file di configurazione** (come ad esempio il dialetto di MAVLINK da utilizzare, già accennato in precedenza).

È ora finalmente possibile invocare `make` per iniziare la procedura di compilazione vera e propria:

```
make -j$(nproc --all)
```

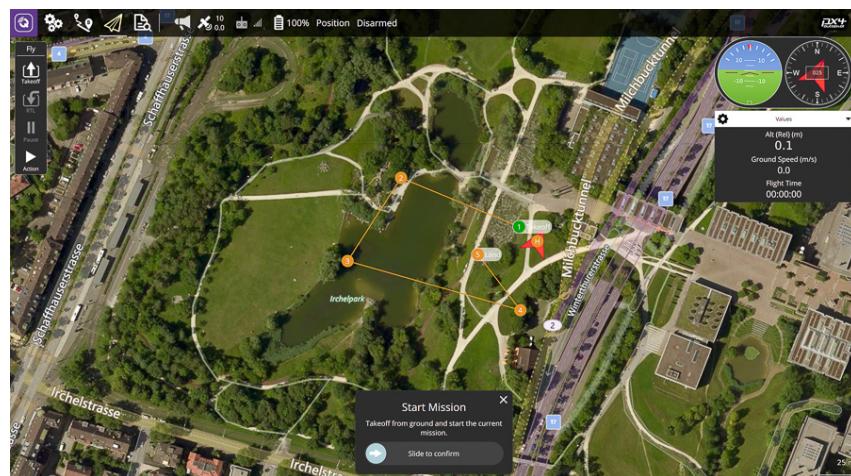
²Gli step riportati qui di seguito si riferiscono al sistema operativo **Ubuntu 22.04 LTS**, lo stesso utilizzato per il lavoro presentato.

2 PIATTAFORME SOFTWARE PER UAV E GCS

Il comando `nproc --all` permette di determinare il numero di core logici che la CPU della macchina su cui si sta svolgendo la compilazione ha a disposizione. Ciò permette una compilazione veloce quanto più possibile.

Una volta terminata la compilazione del software, l'eseguibile prodotto sarà posizionato all'interno della cartella `build/staging` e si chiama semplicemente `QGroundControl`. Sudetto file ha già assegnati i permessi di esecuzione ed è possibile eseguirlo mediante il seguente comando:

staging/QGroundControl



L'interfaccia di QGroundControl una volta aperto

Le impostazioni di QGroundControl

3 Il protocollo MAVLINK

Come accennato in fase di introduzione a questo lavoro, il **protocollo MAVLINK** è il vero e proprio "collante" tra le due parti della comunicazione, ovvero l'UAV e la GCS a terra. In questa sezione viene presentata la filosofia, le specifiche e l'architettura del protocollo MAVLINK e se ne illustra l'evoluzione con il passare del tempo.

3.1 Principi di funzionamento

Il protocollo MAVLINK (nome breve per Micro Aerial Vehicle **Link**) è stato proposto per la prima volta nel 2010 dallo sviluppatore **Lorenz Meier** [27] e si propone come un protocollo basato su **scambio di messaggi** che ha come caratteristica principale quello di essere **lightweight**: alla luce del contesto in cui si sta operando, quest'ultima caratteristica risulta essere particolarmente desiderabile.

Il principio di funzionamento del protocollo MAVLINK è tipico di altri protocolli di tipo **publish-subscribe**, in quanto le due parti della comunicazione (l'UAV e la GCS) pubblicano **messaggi MAVLINK** su un **topic** a cui sono entrambi iscritti.



Logo del progetto

La particolarità di questo protocollo risiede nella sua **duttilità** poiché (al contrario di quanto potrebbe suggerire il suo nome) può essere utilizzato per dispositivi diversi da quelli comunemente definiti come "droni". D'altronde, il progetto ArduPilot lo utilizza anche nei progetti che riguardano Rover e Sub.

Come menzionato in precedenza, l'implementazione di riferimento di questo protocollo è disponibile su **GitHub** e si configura come una libreria **header-only** scritta nel linguaggio di programmazione C e che può essere liberamente integrata nei progetti che intendono farne uso, secondo la licenza **LGPL**. [23]

3 IL PROTOCOLLO MAVLINK

Altra particolarità di questo protocollo è la sua **estensibilità**: è presente un **pool** di messaggi standard che permettono una corretta e completa a 360° comunicazione tra UAV e GCS ma è possibile aggiungerne liberamente degli altri per introdurre un comportamento personalizzato, ottenere informazioni non standard e **scambiare dati** di conseguenza.

Proprio questo ultimo aspetto è stato determinante per il lavoro presentato.

Data la sua estensibilità sono stati stabiliti, col passare del tempo, diverse varianti (o **dialetti**) di MAVLINK, alcuni di questi "supportati ufficialmente".

Ad esempio:

- **minimal**: il "minimo indispensabile", ovvero quei messaggi che vanno implementati su ogni veicolo pena il non funzionamento del protocollo;
- **common**: il sottoinsieme dei messaggi più **comuni** che *andrebbero* messi a disposizione su tutti i veicoli. Naturalmente, include **minimal**;
- **ardupilotmega**: i messaggi MAVLINK usati dal progetto **ArduPilot**, include **common**;
- **ASLUAV**: i messaggi MAVLINK utilizzati per veicoli ad ala fissa denominati **ASLUAV**;
- **uAvionix**: i messaggi MAVLINK utilizzati dal progetto **uAvionix**;
- **paparazzi**: i messaggi MAVLINK utilizzati dall'autopilot **paparazzi**;
- **all**: una collezione di tutti i messaggi definiti in precedenza.

Le definizioni di questi messaggi sono contenute in dei file XML presenti nella repo GitHub del progetto. In ciascuno di questi è presente la direttiva `<include></include>` che permette la definizione di **gerarchie di messaggi**.
[28]

La caratteristica che rende lightweight tale protocollo è il modo in cui i messaggi MAVLINK viaggiano sul link di comunicazione: essi vengono infatti **serializzati** in forma **binaria** [29], operazione che ne facilita di molto il

3 IL PROTOCOLLO MAVLINK

parsing, con un *gain* prestazionale non indifferente rispetto al parsing da altri formati **text-based** come JSON o XML, usati tipicamente in contesti in cui le risorse a disposizione non sono limitate, al contrario di quanto avviene nel caso degli UAV.

Con il passare del tempo sono state proposte **due revisioni maggiori** del protocollo MAVLINK:

- La versione **1.0**, standardizzata nel 2013 [30];
- La versione **2.0**, le cui migliorie sono descritte nella sezione successiva, rilasciata nella prima parte del 2017 [31].

Fino al 2013 era inoltre ampiamente utilizzata la versione **0.9** del protocollo, deprecata con l'adozione da parte dei vari progetti della versione 1.0.

3.2 Architettura di un pacchetto MAVLINK

Un pacchetto MAVLINK, al netto delle differenze tra le due revisioni, è composto sostanzialmente da due parti:

- Un **header** che contiene delle informazioni di controllo;
- Un **payload** che contiene i dati veri e propri scambiati nel pacchetto.

3.2.1 MAVLINK 1.0: struttura di un pacchetto

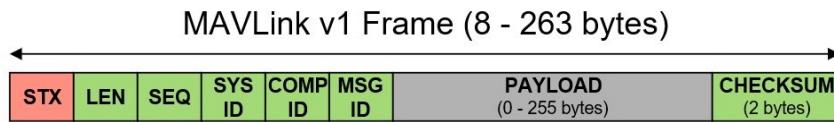
Un **frame** MAVLINK 1.0 è strutturato come segue [32]:

- Start of Text (**STX**, chiamato anche **MAGIC**): un byte che rappresenta l'inizio di un frame MAVLINK. Nella versione 1.0 del protocollo questo byte assume un valore fisso: 0xFE;
- Payload Length (**LEN**): un byte che rappresenta la **lunghezza in byte** del payload. Poiché questo campo è formato da un solo byte, la lunghezza del payload **non può superare i 255 bytes**;
- Packet Sequence Number (**SEQ**): un byte che rappresenta il numero di sequenza del pacchetto nel contesto di una comunicazione. Serve principalmente per la **detection** di **perdita** dei pacchetti e degli errori in generale;
- System ID (**SYSID**): un byte che rappresenta l'ID del **mittente** del messaggio. Non è possibile indicare 0 (l'indirizzo di broadcast) come mittente poiché viene interpretato come non valido dal parser;
- Component ID (**COMPID**): un byte che rappresenta la **componente** del veicolo che ha generato il messaggio (una videocamera o un qualche accessorio);
- Message ID (**MSGID**): un byte che rappresenta il **Message ID** del contenuto del payload. Serve alla libreria per identificare il messaggio e selezionare il parser più adatto per interpretare il contenuto e deserializzarlo;

3 IL PROTOCOLLO MAVLINK

- Payload (PAYLOAD): un vettore di byte di **al più** 255 bytes (si veda il ragionamento fatto in precedenza) che contiene i dati veri e propri scambiati tra l'UAV e la GCS;
- CRC (+ CRC Extra) (CHECKSUM): due bytes:
 - Il primo rappresenta il checksum del messaggio (nel suo calcolo non viene considerato STX);
 - Il secondo (denominato CRC_EXTRA) rappresenta il CRC della **struttura della definizione XML del messaggio**. Ciò permette alle due parti della comunicazione di assicurarsi di star considerando messaggi definiti allo stesso modo.

Di seguito è presente un riepilogo della struttura di un pacchetto MAVLINK nella sua revisione 1.0:



La struttura di un messaggio MAVLINK 1.0

Per quanto riguarda la dimensione dei pacchetti MAVLINK 1.0:

- La dimensione **minima** di un pacchetto MAVLINK 1.0 è di 8 bytes (quando **non** è presente un payload, tipicamente quando si ha a che fare con dei pacchetti di ACK);
- La dimensione **massima** di un pacchetto MAVLINK 1.0 è di 263 bytes (8 bytes di header + 255 bytes di payload, quando quest'ultimo è completamente "riempito").

3.2.2 MAVLINK 2.0: struttura di un pacchetto

Con il rilascio della versione 2.0 delle specifiche del protocollo, la struttura di un pacchetto è variata leggermente.

In particolare:

- Sono stati aggiunti due nuovi membri dell'header: **CMPFLAGS** e **INCFLAGS**;
- È stato aggiunto il supporto alla **firma digitale del pacchetto**;

Un frame MAVLINK 2.0 è quindi strutturato come segue [33]:

- Start of Text (**STX**, chiamato anche **MAGIC**): un byte che rappresenta l'inizio di un frame MAVLINK. Nella versione 2.0 del protocollo questo byte assume un valore fisso: 0xFD;
- Payload Length (**LEN**): un byte che rappresenta la **lunghezza in byte** del payload. Poiché questo campo è formato da un solo byte, la lunghezza del payload **non può superare i 255 bytes**;
- Incompatibility Flags (**INCFLAGS**): un byte che rappresenta quelle features del protocollo che **devono essere tassativamente supportate** da chi riceve il pacchetto poiché **ne alterano la struttura**. In caso contrario, le specifiche del protocollo indicano che il pacchetto deve essere **scartato**. L'unico valore attualmente supportato è 0x01, che rappresenta la possibilità di **firmare il messaggio**;
- Compatibility Flags (**CMPFLAGS**): un byte che indica la presenza di features "aggiuntive" che, anche se non supportate dal ricevente **non pregiudicano** la deserializzazione del pacchetto e che quindi lo scarto del pacchetto non è per forza necessario. Un esempio è una flag che denota un pacchetto "ad alta priorità".
- Packet Sequence Number (**SEQ**): un byte che rappresenta il numero di sequenza del pacchetto nel contesto di una comunicazione. Serve prin-

cipalmente per la **detection** di **perdita** dei pacchetti e degli errori in generale;

- System ID (**SYSID**): un byte che rappresenta l'ID del **mittente** del messaggio. Non è possibile indicare 0 (l'indirizzo di broadcast) come mittente poiché viene interpretato come non valido dal parser;
- Component ID (**COMPID**): un byte che rappresenta la **componente** del veicolo che ha generato il messaggio (una videocamera o un qualche accessorio);
- Message ID (**MSGID**): **tre bytes** che rappresentano il **Message ID** del contenuto del payload. Serve alla libreria per identificare il messaggio e selezionare il parser più adatto per interpretare il contenuto e deserializzarlo. Con l'aumento della dimensione di questo campo è possibile supportare un numero più grande di messaggi e di rendere la libreria ancora più flessibile rispetto a prima;
- Payload (**PAYLOAD**): un vettore di byte di **al più** 255 bytes (si veda il ragionamento fatto in precedenza) che contiene i dati veri e propri scambiati tra l'UAV e la GCS;
- CRC (+ CRC Extra) (**CHECKSUM**): due bytes:
 - Il primo rappresenta il checksum del messaggio (nel suo calcolo non viene considerato **STX**);
 - Il secondo (denominato **CRC_EXTRA**) rappresenta il CRC della **struttura della definizione XML del messaggio**. Ciò permette alle due parti della comunicazione di assicurarsi di star considerando messaggi definiti allo stesso modo.

Oltre all'header ed al payload, il protocollo MAVLINK 2.0 introduce anche la possibilità di aggiungere (*append*) una **firma digitale** ad un frame. Tale campo è lungo esattamente **13 bytes**.

3 IL PROTOCOLLO MAVLINK

Di seguito è presente un riepilogo della struttura di un pacchetto MAVLINK nella sua revisione 2.0:



La struttura di un messaggio MAVLINK 2.0

Alla luce di tutte le informazioni qui sopra riportate, riguardo la dimensione di un frame MAVLINK 2.0 è possibile menzionare che:

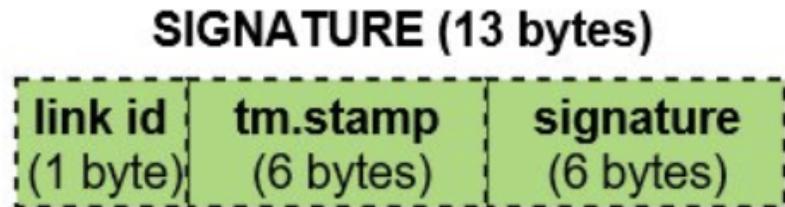
- La dimensione **minima** di un pacchetto MAVLINK 2.0 (senza payload e senza firma) è di **12 bytes**;
- La dimensione **massima** di un pacchetto MAVLINK 2.0 (con payload "riempito" e con apposta una firma digitale) è di **280 bytes** (**12** bytes di header + **255** bytes di payload + **13** bytes di firma).

3.2.3 La firma digitale in MAVLINK 2.0

Come menzionato, a partire dalla versione 2.0 del protocollo MAVLINK è stato aggiunto il supporto alla firma digitale dei frame. La struttura dell'header della firma del pacchetto risulta essere grande 13 bytes, così strutturati [34]:

- **Link ID (LINK_ID)**: un byte che rappresenta il **link** su cui è stato inviato un determinato pacchetto;
- **Timestamp (TM_STAMP)**: **sei bytes** che rappresentano il numero di unità di 10 **microsecondi** passati dal **1 Gennaio 2015**. Questo contatore deve aumentare in maniera *monotona* ad ogni messaggio inviato. Se ci si trova in un contesto in cui vengono inviati più di ~ 100.000 messaggi al secondo, questo timestamp può **superare** l'ora corrente;
- **Signature (SIGNATURE)**: **sei bytes** (48 bit) che contengono la firma digitale vera e propria.

Di seguito è presente una rappresentazione grafica dell'header aggiuntivo della firma digitale.



La struttura dell'header di firma

Un pacchetto MAVLINK firmato, come accennato in precedenza, è caratterizzato dal campo `INCFLAGS` impostato a `0x01`.

3.2.4 Algoritmo di generazione della firma digitale in MAVLINK 2.0

I sei bytes che compongono la firma digitale vera e propria sono ottenuti concatenando [35]:

- Una **chiave segreta** condivisa tra le due parti della comunicazione (**SECRET_KEY**);
- I bytes che compongono l'**header** del pacchetto (**HEADER**);
- I bytes che compongono il **payload** del pacchetto (**PAYLOAD**);
- I bytes che compongono il **CRC** dell'intero pacchetto (**CRC**);
- I bytes che compongono il **Link ID** dell'header della firma (**LINKID**);
- I bytes che compongono il **timestamp** dell'header della firma (**TM.STAMP**).

ed estraendo i primi **48 bit** dell'hash di questa sequenza ottenuto utilizzando l'algoritmo **SHA256**.

Ricapitolando:

```
SIGN = SHA256_48(SECRET_KEY + HEADER + PAYLOAD + CRC + LINKID +
                   TM.STAMP) ;
```

dove **SHA256_48** rappresenta il troncamento dell'hash prodotto ai primi **48 bit** e il simbolo **+** rappresenta la concatenazione tra bytes.

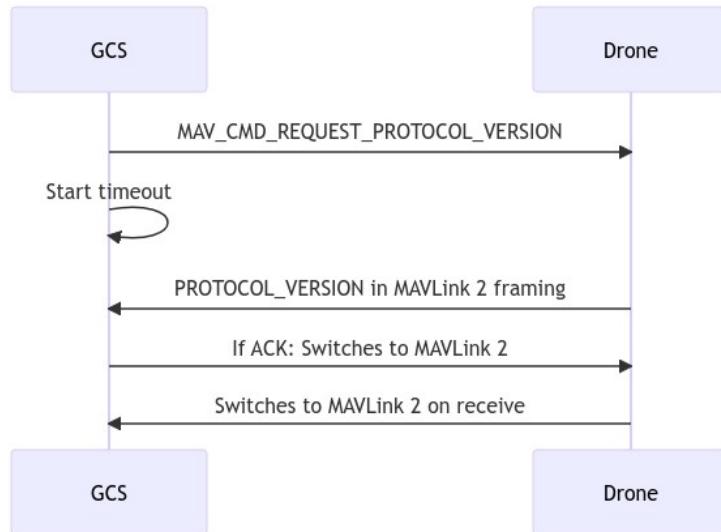
3.2.5 Scelta della versione del protocollo

Poiché l'uso del protocollo MAVLINK 1.0 è ancora ampiamente diffuso, è stato necessario implementare un meccanismo di "definizione" della versione del protocollo da utilizzare per una comunicazione: una sorta di **handshaking** [36] preliminare la cui struttura viene riportata di seguito:

- La **Ground Control Station** invia il messaggio `MAV_CMD_REQUEST_PROTOCOL_VERSION` al drone, richiedendo quindi la versione del protocollo in uso dal veicolo ed iniziando contemporaneamente un *countdown di timeout*;
- L'UAV a questo punto può rispondere in due modi:
 - Se è presente il supporto per il protocollo MAVLINK 2.0, risponderà con il messaggio di `ACK_PROTOCOL_VERSION` formattato come frame MAVLINK 2.0;
 - Se invece non vi è supporto per MAVLINK 2.0, risponderà con un **NACK** e, in caso di assenza di procedure apposite, **sarterà il messaggio precedente** e non risponderà affatto.
- – Se la GCS riceve il messaggio `PROTOCOL_VERSION` allora verrà effettuato lo *switch* alla versione 2.0 del protocollo.
 - Se viene ricevuto un NACK oppure si giunge al timeout, viene utilizzata la versione 1.0 del protocollo poiché evidentemente l'UAV non è riuscito ad interpretare in maniera corretta il frame precedente.

3 IL PROTOCOLLO MAVLINK

Di seguito è presente una rappresentazione grafica dell'handshaking appena descritto.



L'handshake effettuato per stabilire la versione del protocollo da utilizzare

4 Sicurezza del protocollo MAVLINK

4.1 Possibili attacchi contro MAVLINK

Come qualsiasi protocollo che fa utilizzo di un mezzo trasmittivo, MAVLINK non è di certo immune a determinati tipi di attacchi che vanno a colpire ognuno dei membri della cosiddetta triade **CIA**, ovvero:

- Confidentiality;
- Integrity;
- Availability

Inoltre, l'intrinseca natura di MAVLINK come protocollo **lightweight** implica un maggior focus sulle performance e sulla portabilità del protocollo piuttosto che sulla **sicurezza** dello stesso.

In uno studio [37] del 2019, **Kwon et al.** hanno svolto un'analisi delle possibili vulnerabilità che affliggono il protocollo MAVLINK. Tra queste si annoverano:

- **Man-in-the-middle**, che permette ad un attaccante di posizionarsi dal punto di vista logico "al centro" di una comunicazione tra un drone e la sua GCS. Questo è un attacco che viola la confidenzialità e (specialmente) l'integrità della comunicazione;
- **Eavesdropping**, che permette ad un attaccante di "origliare" (*eavesdrop*) la conversazione tra un drone e la sua GCS. Questo attacco viola la confidenzialità della comunicazione;
- **DoS - Denial of Service**, che permette ad un attaccante di operare in maniera tale da rendere uno o anche entrambi gli apparati coinvolti della comunicazione non in grado di continuare. Questo è un attacco all'availability. Ciò può avvenire tramite flooding (attacchi "rumorosi") o mediante lo sfruttamento di determinate vulnerabilità o errori di configurazione degli apparati stessi (attacchi "non rumorosi").

Gli autori dello studio menzionano due tipologie di attacchi portati a termine con successo, testati mediante **SITL** ed ArduPilot **Mission Control**:

- Attacco basato su **ICMP flooding**, che permette di saturare i due endpoint della comunicazione, modificando pesantemente la **varianza** dei tempi di *inter-ricezione* dei pacchetti, rendendo quindi la comunicazione **instabile**;
- Attacco basato su **packet injection**, che permette ad un attaccante di effettuare l'**hijacking** della comunicazione tra drone e GCS ed è basata su una vulnerabilità del **Waypoint Protocol**. In breve, quando un UAV riceve il comando **MISSION_COUNT(N)** da parte della GCS cancella tutte le informazioni relative alla missione che stava eseguendo e resta in attesa di indicazioni sulla nuova missione da eseguire. L'iniezione di un pacchetto di questo tipo permette ad un attaccante di "iniettare" una nuova missione da eseguire magari creata ad-hoc.

Inoltre, il fatto che il protocollo MAVLINK non preveda alcun tipo di cifratura dei pacchetti in transito [38] implica una intrinseca insicurezza dello stesso poiché tutte le informazioni sono scambiate in chiaro, rendendo quindi possibile gli attacchi di cui sopra.

4.2 Possibili contromisure

La versione 2.0 di MAVLINK, che introduce il sistema della firma digitale dei pacchetti mette **solo parzialmente** una pezza alle problematiche descritte poiché previene il **tampering** delle comunicazioni ma non risolve le problematiche relative alla **confidenzialità**.

Data la sempre maggiore attenzione riservata a questo tipo di dispositivi in particolare nell'ultimo decennio, il problema della messa in sicurezza dei protocolli come il MAVLINK ha acquisito sicuramente un'importanza maggiore. Il problema, inoltre, viene reso ancora più difficile dal fatto che si renda necessario fare un **tradeoff** tra il grado di sicurezza del protocollo e l'alto

throughput che lo stesso deve offrire pur trovandosi in contesti caratterizzati da risorse hardware tutt’altro che illimitate.

Negli ultimi anni sono stati, in effetti, compiuti alcuni passi in avanti in questo senso grazie a diversi lavori di ricerca.

4.2.1 MAVSec

In uno studio [39] pubblicato nel Maggio del 2019, **Allouch et al.**, oltre ad un’ulteriore overview delle potenziali vulnerabilità insite nel protocollo MAVLINK hanno proposto una soluzione che potesse risolvere la problematica della **confidenzialità** della comunicazione introducendo una sorta di *framework* denominato **MAVSec**, che prevede di modificare i progetti **ArduPilot**, **QGroundControl** e la libreria **MAVLINK** (nella sua implementazione di riferimento in C) al fine di supportare diversi algoritmi di cifratura **simmetrica** da applicare al **payload** dei messaggi MAVLINK prima della loro immissione nel mezzo trasmisivo.

Gli algoritmi considerati nello studio comprendono:

- AES in modalità **CBC**;
- AES in modalità **CTR**;
- RC4 (usato tra l’altro dal protocollo di sicurezza WEP);
- ChaCha20.

Gli autori dello studio hanno confrontato le performance di questi algoritmi sostanzialmente in tre macro-aree:

- **Consumo di memoria** da parte dell’UAV;
- **Numero di frame** inviati al secondo;
- **Consumo di CPU** da parte dell’UAV.

Dai risultati della ricerca emerge che l’algoritmo migliore in **tutti** questi campi risulti essere **ChaCha20**.

Tale algoritmo è stato proposto nel 2008 in un paper [40] pubblicato da **Daniel J. Bernstein** e si configura come un **cifrario a blocchi** variante di **Salsa20/20** che permette di migliorarne la resistenza ad attacchi basati su **crittoanalisi** riducendo in alcuni casi il tempo di esecuzione di ogni round. Il 20 all'interno del nome è dovuto al numero di round che vengono effettuati durante l'esecuzione dell'algoritmo. In generale, i membri della famiglia ChaCha consistono:

- ChaCha8, derivato dall'algoritmo **Salsa20/8** e che prevede **8 rounds**;
- ChaCha12, derivato dall'algoritmo **Salsa20/12** e che prevede **12 rounds**;
- Il già citato ChaCha20.

4.2.2 Securing Unmanned Aerial Vehicles by Encrypting MAVLINK Protocol

In un successivo studio [41] pubblicato alcuni anni più tardi, nel 2022, **Sabuwala et al.** proposero uno studio simile a quello menzionato in precedenza, prendendo però in considerazione algoritmi diversi. In particolare:

- ChaCha20, come nel caso dello studio precedente;
- **Encryption by Navid**, una tecnica di cifratura basata (in parte) su un **Cifrario di Cesare**;
- DMAV, uno schema di cifratura basata su una codifica denominata "DNA dinamica" proposto in uno studio del 2022.

I risultati di questo studio confermano, in un certo senso, quelli del precedente, determinando come l'algoritmo ChaCha20 sia il migliore dei tre confrontati negli ambiti di **pacchetti inviati al secondo**, **consumo di memoria nell'UAV**, **percentuale di CPU** in uso nell'UAV.

Si noti come in entrambi i casi sia stato utilizzato il progetto ArduPilot ed in particolare il già citato software **SITL** per condurre i test.

4.2.3 Sicurezza in ambienti resource-constrained: una breve introduzione al concetto di lightweight cryptography

Più o meno nello stesso periodo in cui si il panorama delle piattaforme hardware e software per UAV si è arricchito e l'utilizzo di questi dispositivi è andata "democratizzandosi", si è assistito, in generale, alla sempre più capillare di dispositivi "intelligenti" da utilizzare nei contesti più disparati. Nasce quindi il concetto di **Internet of Things** (comunemente definito con il suo acronimo **IoT**), che condivide un discreto numero di aspetti sia positivi che negativi con il contesto degli UAV.

Certamente, si ritrova in entrambi i casi la necessità dell'utilizzo di protocolli leggeri ed efficienti ma anche (e soprattutto) i limiti delle piattaforme hardware che permettono il funzionamento dei dispositivi (quindi tipicamente microcontrollori dotati di funzionalità di rete).

Vien da sé, quindi, che il discorso sulla generale insicurezza dei protocolli affrontato più volte in questo lavoro per gli UAV valga anche per i dispositivi IoT.

Nondimeno, la capillare diffusione di tali dispositivi in ambienti domestici ed aziendali pone un'ulteriore problematica di **privacy** oltre che di **safety** e di **security** come invece avveniva nel caso degli UAV.

Per far fronte a questo tipo di criticità e al (necessario) tradeoff tra performance e requisiti di sicurezza, ecco che col passare del tempo sorge una nuova "branca" di ricerca nell'ambito della crittografia dedicata allo sviluppo di algoritmi (e di relative piattaforme *ad-hoc* per la loro implementazione in hardware) pensati per funzionare in ambienti **resource-constrained**: la **lightweight cryptography**.

Come analizzato in una revisione della letteratura [42] compiuta nel 2018 da **Sadkhan e Salman**, diversi studi sono stati proposti nel campo della crittografia leggera e che hanno portato alla definizione di nuovi schemi di cifratura **simmetrica** ed **asimmetrica**.

Nell'Agosto del 2018, inoltre, il **NIST**³ ha annunciato l'inizio del processo

³National Institute of Standards and Technology, un importante ente statale americano

di standardizzazione [43] di algoritmi di crittografia leggera. Dopo **due rounds** di selezione, nel 2021 sono stati annunciati i 10 **finalisti** [44] di questo processo, ovvero:

- **ASCON**;
- **Elephant**;
- **GIFT-COFB**;
- **Grain128-AEAD**;
- **ISAP**;
- **Photon-Beetle**;
- **Romulus**;
- **Sparkle**;
- **TinyJambu**;
- **Xoodyak**.

Nel 2023, infine, l'algoritmo **ASCON** è stato dichiarato vincitore [45].

4.2.4 MAVLINK e lightweight cryptography

In un lavoro di tesi [46] proposto dallo studente **Angelo Passaro**, è stato proposto un approccio basato su crittografia leggera che riprendesse il framework e le tecniche presentate nel lavoro **MAVSec** menzionato in precedenza ed il cui codice è disponibile su GitHub [47].

Tale lavoro si differenzia dai precedenti poiché va a sfruttare diverse caratteristiche del protocollo MAVLINK e delle piattaforme software descritte nelle precedenti sezioni del lavoro qui presentato.

Sono stati considerati diversi algoritmi che appartengono al pool della crittografia leggera sia dal punto di vista della cifratura **simmetrica** che dal punto di vista degli algoritmi di **scambio delle chiavi** e sono stati condotti diversi test per stabilire quale fosse quello più conveniente rispetto ai criteri definiti in precedenza dagli altri studi.

In particolare, gli algoritmi di cifratura **simmetrica** integrati nel lavoro qui considerato consistono in:

- Trivium;
- Rabbit;
- ChaCha20 (utilizzato principalmente negli altri studi);
- Simon & Speck;

Per quanto riguarda invece il concetto di scambio delle chiavi, nel lavoro considerato in questa sezione vengono discussi i risultati di uno studio [48] presentato nel 2017 da **Alvarez et al.** che propone un confronto tra i seguenti *meccanismi* di scambio delle chiavi:

- RSA;
- DH: scambio delle chiavi **Diffie-Hellman**;
- ECDH: meccanismo di scambio delle chiavi ispirato a Diffie Hellman ma che si basa sul concetto di **curva ellittica**;

- Curve25519: altro meccanismo di scambio delle chiavi basato su **curve ellittiche** che mira a migliorare le prestazioni rispetto ad ECDH;
- FourQ: ulteriore meccanismo basato su curve ellittiche proposto da **Microsoft** nel 2015.

I risultati raggiunti nel paper menzionato in precedenza suggeriscono che gli algoritmi basati su **curve ellittiche** offrano le migliori prestazioni ed un minore consumo di risorse (memoria e percentuale di CPU utilizzata) ed in particolare **FourQ** risulta prevalere sugli altri algoritmi del pool considerato.

Proprio per questa ragione, l'algoritmo che sta alla base del meccanismo di scambio delle chiavi implementato nel lavoro di Angelo Passaro consiste proprio in **FourQ**.

Dal punto di vista tecnologico, invece, sono state utilizzate le stesse piattaforme considerate negli altri studi menzionati in precedenza, ovvero:

- **ArduPilot** per l'utilizzo di SITL;
- **QGroundControl** per controllare il simulatore;
- **MAVLINK** (nella sua implementazione di riferimento in C) per integrare i nuovi messaggi per lo scambio delle chiavi e gestire cifratura e decifratura.

Il lavoro proposto in questo elaborato ricalca il framework stabilito da **MAVSec** prima e dal lavoro di **Angelo Passaro** poi dal punto di vista delle modifiche introdotte nello stack tecnologico, adattandolo poi al contesto della **Post-Quantum Cryptography**.

Una descrizione tecnica approfondita dell'implementazione effettuata verrà svolta nel **Capitolo 6**.

5 Quantum Computing e Post-Quantum Cryptography

In questo capitolo verrà svolta una breve introduzione al concetto di **Quantum Computing** e si discuteranno le implicazioni della sua diffusione, tra gli altri, nel campo della crittografia.

5.1 Cos'è il Quantum Computing

Sebbene il concetto di Quantum Computing stia acquisendo sempre più importanza negli ultimi decenni, l'idea che sta alla sua base risale ai **primi anni '80** del XX secolo, quando in uno studio [49] del fisico statunitense **Paul Benioff** (venuto a mancare di recente, nel marzo del 2022) venne proposto un primo modello di **macchina di Turing** operante sotto le leggi della **mecanica quantistica**. Da quel momento molti altri studi vennero basati su questa idea: tra questi annoveriamo quello proposto da **Richard Feynman** a proposito della simulazione di fenomeni fisici usando computer quantistici.

La vera differenza che intercorre tra un computer classico ed una macchina quantistica consiste nel modo in cui sono rappresentate le informazioni: se nel caso "classico" a noi familiare si ragiona in termini di **bit** (ovvero la più piccola unità con cui possono essere codificate delle informazioni), nel caso quantistico si passa al concetto di **qubit**, termine coniato nel 1995 [50] dal fisico Benjamin Schumacher che rappresenta l'unità di informazione quantistica.

Se nel contesto della computazione "classica" gli **stati** possibili che un bit può assumere sono solamente due: lo stato "0" e lo stato "1", nel contesto quantistico il confine tra gli stati che un qubit può assumere è in un certo senso **più labile**, in accordo con il fenomeno quantistico della **sovraposizione tra stati**.

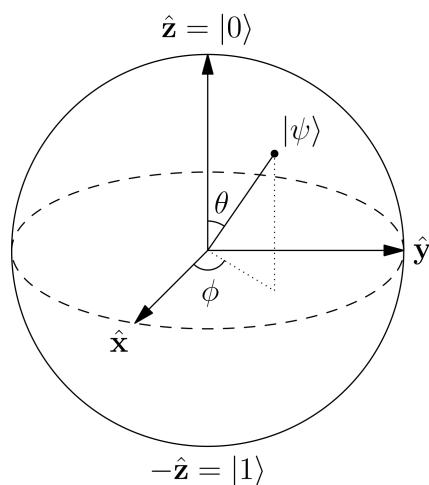
Un qubit può quindi trovarsi rispettivamente:

- Nello stato 0 (con il 100% di probabilità);
- Nello stato 1 (con il 100% di probabilità);

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

- In una sovrapposizione di due stati: con il p di probabilità nello stato 0 e con probabilità $1 - p$ nello stato 1.

Al di là dei (comunque importanti e certamente necessari!) formalismi matematici, l'utilizzo di un sistema basato su **qubit** al posto dei classici **bit** permette alle macchine di ottenere performance nettamente maggiori a quelle dei computer tradizionali, tant'è che col passare del tempo questa tipologia di macchine sono state usate per provare a risolvere in tempo polinomiale problemi per cui non erano noti algoritmi "classici" efficienti.



La sfera di Bloch: una rappresentazione geometrica dei possibili stati in cui un **qubit** può trovarsi

5.2 Il problema: RSA, DH e l'algoritmo di Shor

Uno degli esempi lampanti del potere computazionale notevolmente maggiore dei computer quantistici rispetto a quelli classici consiste nella pubblicazione di uno studio [51] da parte di **Peter W. Shor** nel 1994 che propone un **algoritmo** (conosciuto appunto come **Algoritmo di Shor**) per la fattorizzazione di interi.

Dal punto di vista teorico, tale algoritmo si configura come una **riduzione** dal problema della fattorizzazione di interi ad un problema di ricerca dell'**ordine** di un gruppo (inteso come struttura algebrica), anch'esso considerato come problema *difficile* nel contesto della computazione classica.

Si noti comunque che il processo di riduzione in sé avviene in tempo polinomiale anche in un computer tradizionale.

Il *core* dell'algoritmo proposto da Shor si occupa quindi di risolvere efficientemente (leggasi, *in tempo polinomiale*) il problema verso cui è stata effettuata la riduzione, seppur con un tasso di errore minore o uguale ad $\frac{1}{3}$ nel caso pessimo. La soluzione di tale problema, per com'è strutturato il processo della riduzione di sicurezza, implica la **soluzione anche del problema di partenza**.

L'analisi della complessità dell'algoritmo proposto da Shor suggerisce che il suo tempo di esecuzione, comprensivo di tutte le operazioni descritte in precedenza sia proporzionale a:

$$\mathcal{O}((\log N)^2(\log \log N))$$

a patto di utilizzare il metodo di **Harvey e Van Der Hoeven** [52] per la moltiplicazione efficiente di interi in tempo $\mathcal{O}(n \log n)$ e che quindi il problema della fattorizzazione di interi appartenga, dal punto di vista della teoria della computazione, alla classe **BQP**, ovvero **la classe dei problemi risolvibili in tempo polinomiale mediante computer quantistici**.

L'approccio proposto da Shor, secondo quanto menzionato nel sommario dello studio, è applicabile non solo al problema della fattorizzazione di interi ma anche al problema del **logaritmo discreto**.

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

Le implicazioni dello studio di Shor sono **enormi** nel campo della crittografia poiché il problema della fattorizzazione di interi e del logaritmo discreto sono alla base degli schemi più utilizzati nel campo della crittografia asimmetrica (o a chiave pubblica): rispettivamente di **RSA** e del meccanismo di Key Exchange **Diffie Hellman**.

La risoluzione in tempo polinomiale mediante computer quantistici di tali problemi renderebbe del tutto insicuri tali schemi, ponendo un considerevole rischio correlato al funzionamento dei protocolli crittografici basati su problemi difficili, come appunto RSA e Diffie Hellman.

Tuttavia, vale la pena sottolineare come il tasso di errore di tale algoritmo (dovuto al concetto di sovrapposizione degli stati) sia ancora troppo alto, pregiudicando quindi l'esecuzione dell'algoritmo anche sulle piattaforme più potenti allo stato attuale.

5.3 La soluzione: Post-Quantum Cryptography

Il problema di situazioni del genere è che una volta note le modalità, ciò che manca è quindi la **potenza di calcolo**, che con l'avanzamento tecnologico sarà con ogni probabilità raggiunta col passare del tempo, ponendo quindi un rischio concreto per la sicurezza dei sistemi informatici (e non solo).

Proprio per reagire a questo tipo specifico di minaccia, negli anni ha preso piede un nuovo concetto nel campo della crittografia, che permette di superare la minaccia posta dall'avvento del Quantum Computing: quello di **Post-Quantum Cryptography**.

Il grosso della ricerca in questo campo si è concentrato nel campo della crittografia a chiave pubblica e quindi nel contesto dello scambio di chiavi sicuro poiché è quello che fa affidamento su problemi difficili come quelli descritti in precedenza (si pensi appunto ad RSA ed a Diffie Hellman).

Gli schemi crittografici che sono stati messi a punto nel contesto di questo nuovo campo di studi vengono divisi in diverse categorie, a seconda dell'approccio che adottano [53]. Tra questi:

- Approccio basato su **Reticoli**: l'approccio su cui si basa lo schema Kyber e che fa riferimento ad una struttura algebrica detta **Reticolo** e che include i sistemi crittografici basati sul problema **LWE** (acronimo per **Learning With Errors**) o su loro varianti;
- Approccio basato su **crittografia multivariata**: tale approccio si basa sulla difficoltà della risoluzione di **sistemi di equazioni multivariate**;
- Approccio basato su **hash**, che si basa sull'assunzione della sicurezza delle **funzioni hash**. Esempi di schemi che sfruttano questo approccio consistono nello schema di firma digitale di **Merkle** o comunque altri schemi che usano il **Merkle Tree**;
- Approccio basato su **codici**: ad esempio lo schema di cifratura di **McEliece**, che si basa sui **Codici di Goppa**.

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

Proprio l'attenzione riservata col passare del tempo a questo campo di ricerca, anche e soprattutto in seguito allo studio di Shor ed alla definizione del "suo" algoritmo, ha spinto il **NIST** a cercare di stabilire uno **standard** per algoritmi di cifratura a chiave pubblica resistenti ad attacchi con computer quantistici, un po' come successo (e menzionato in una precedente sezione del presente lavoro) con la crittografia leggera.

A dicembre 2016 [54] venne rilasciata una **Request for Nomination** sul sito web del NIST, prima milestone nel processo di standardizzazione, con deadline fissata al 30 novembre 2017.

Dopo quattro round di sottomissioni per tale processo, nel 2022 gli algoritmi selezionati dall'istituto sono i seguenti:

Per la crittografia a chiave pubblica:

- CRYSTALS-Kyber.

Riguardo invece gli schemi di firma digitale:

- CRYSTALS-DILITHIUM;
- FALCON;
- SPHINCS+

Si noti come nel caso della crittografia a chiave pubblica sia stato selezionato **un unico** algoritmo e che quindi è divenuto lo **standard** per questo tipo di schemi.

Anche nel contesto di questo lavoro, lo schema **Kyber** è stato quello scelto per eseguire operazioni di scambio chiavi.

Di seguito viene svolta una panoramica sulle basi e sulla struttura di tale schema crittografico.

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

5.4 L'algoritmo Kyber

Come affermato in precedenza, CRYSTALS-KYBER è l'unico schema di cifratura selezionato dal NIST nel processo di standardizzazione.

Fa parte della suite **CRYSTALS** (acronimo di **Cryptographic Suite for Algebraic Lattices**) [55] e si configura come "meccanismo di incapsulamento delle chiavi CCA-sicuro basato sulla difficoltà del problema LWE" [56].

Viene proposto alla comunità scientifica nel 2017 in uno studio [57] di **Joppe et al.** e successivamente valutato dal NIST per il processo di standardizzazione.

Prima di descrivere il funzionamento di tale schema è necessario fare una panoramica sulle sue basi matematiche e scientifiche, in particolare il concetto di KEM, il concetto di **Reticolo** ed il problema **LWE** (acronimo di **Learning with Errors**).

5.4.1 KEM - Key Encapsulation Mechanism

L'acronimo KEM è un abbreviazione per **Key Encapsulation Mechanism** ed è un particolare sistema crittografico *ibrido* che, in sostanza, permette ai due "interlocutori" di una conversazione di utilizzare **un algoritmo di cifratura asimmetrica** per lo scambio di una chiave di cifratura da utilizzare nel contesto di un cifrario **simmetrico**. Naturalmente, il concetto di KEM non è solamente prerogativa della crittografia post quantistica ma è stato studiato ed implementato anche nel contesto della crittografia classica. Un esempio lampante ne è il KEM **basato sullo schema RSA** o su **curve ellittiche**, ampiamente utilizzati al giorno d'oggi.

Formalmente, un generico schema di incapsulamento KEM K è una tupla di algoritmi **probabilistici di tempo polinomiale**:

$$K = (\text{Gen}(), \text{Encaps}(), \text{Decaps}())$$

dove:

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

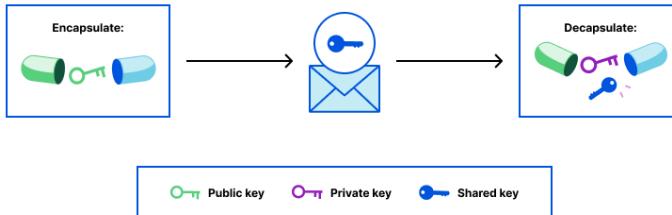
- **Gen(sec_par)** è un algoritmo che si occupa di generare (e restituire) una **coppia di chiavi** (**pk**, **sk**) di una lunghezza determinata da **parametro di sicurezza sec_par** dell'algoritmo. In particolare:
 - **pk** rappresenta la **chiave pubblica**;
 - **sk** rappresenta la **chiave segreta**.
- **Encaps(pk)** è l'algoritmo di **incapsulamento** che prende in input la chiave pubblica generata da **Gen** e restituisce:
 - **k**: una chiave segreta da usare con un cifrario simmetrico. Non deve essere inviata direttamente;
 - **s**: un **testo cifrato**, ovvero la "capsula" che viene inviata all'altro membro della conversazione e che contiene la chiave **k** appena descritta.
- **Decaps(s, sk)** è l'algoritmo di **decapsulamento** che prende in input:
 - **s**: la capsula generata da **Encaps**;
 - **sk**: la chiave segreta generata da **Gen**.E restituisce in output:
 - **k**: la chiave che era stata incapsulata da **Encaps** se non si sono verificati errori.

Questo tipo di costruzione crittografica si pone come **ibrido** tra uno schema di cifratura a chiave pubblica ed a chiave privata.

Si noti come, a differenza dei normali schemi di cifratura, non sia possibile scegliere un messaggio da incapsulare. Al contrario, **Encaps()** si occupa di scegliere la chiave simmetrica **k** e di incapsularla autonomamente.

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

Di seguito è presente un diagramma che riassume il funzionamento di un generico KEM K , appena descritto formalmente.



Funzionamento di un Key Encapsulation Mechanism⁴

L'architettura di questo tipo di schemi di cifratura rimane sostanzialmente la stessa anche nel caso degli algoritmi post-quantum e l'algoritmo **Kyber** non fa eccezione. Il concetto che cambia è invece l'**assunzione** che sta alla base del funzionamento dello schema: nel caso classico i principali KEM utilizzati si basano sul problema **RSA** (e quindi sulla fattorizzazione di interi) e su **curve ellittiche** come menzionato in precedenza, mentre algoritmi come **Kyber** si basano sul problema **LWE**, che può essere ricondotto ad un altro problema definito su un **Reticolo**, entrambi oggetti di discussione nelle prossime sezioni.

⁴Fonte dell'immagine

5.4.2 Reticoli

Nelle sezioni precedenti sono stati presentati i vari approcci alla crittografia post-quantistica: uno di questi consiste in quello basato su **Reticoli**.

Il nome "reticolo" (in inglese *lattice*) usato senza contestualizzazione risulta essere fuorviante poiché vi sono collegati due concetti distinti, ovvero:

1. **Reticolo** inteso come **struttura algebrica** o, equivalentemente, come **insieme parzialmente ordinato** tale che per ogni coppia di elementi (x, y) è possibile individuare l'**estremo superiore** *Sup* e l'**estremo inferiore** *Inf*. Tale tipologia di insiemi può essere rappresentato mediante un **diagramma di Hasse**;
2. **Reticolo** inteso come particolare **sottogruppo** dello spazio vettoriale \mathbb{R}^n .

Il **secondo** concetto è quello utilizzato nel campo della crittografia.

Nel lavoro **Introduction to Post-Quantum Cryptography** [53] è presente una definizione formale di reticolo nel capitolo dedicato a tale concetto, curato da **Daniele Micciancio** ed **Oded Regev**, che viene qui proposta.

Intuitivamente:

Definizione 1. Un reticolo è un insieme di punti in uno spazio n -dimensionale con una struttura periodica.

Formalmente:

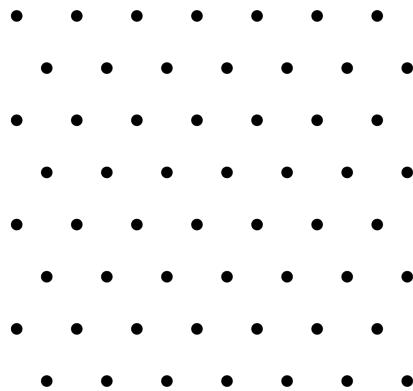
Definizione 2. Siano $b_1, b_2, \dots, b_n \in \mathbb{R}^n$ n vettori linearmente indipendenti (dunque una **base** di \mathbb{R}^n). L'insieme definito come:

$$\mathcal{L}(b_1, b_2, \dots, b_n) = \left\{ \sum_{i=1}^n x_i b_i : x_i \in \mathbb{Z} \right\}$$

prende il nome di **reticolo**. I vettori b_1, b_2, \dots, b_n prendono il nome di **base** del reticolo.

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

L'impiego di questo particolare concetto nel campo della crittografia deriva principalmente dalla definizione di alcuni problemi che si suppone generalmente essere **difficili**, come il problema **SVP** (acronimo per **Shortest Vector Problem**, che consiste nel trovare il vettore più piccolo all'interno di un reticolo) e, mediante opportune trasformazioni, del problema **LWE**, che sta alla base dell'algoritmo **Kyber** e che verrà ora descritto.



La rappresentazione di un reticolo su un piano euclideo

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

5.4.3 Il problema LWE

Strettamente legato al problema SVP appena menzionato è il problema LWE (acronimo per **Learning with Errors**) poiché mediante un processo di **riduzione** [58] è possibile affermare che se (una variante del) problema SVP (ed in particolare GapSVP) è difficile, allora anche il problema LWE è difficile.

L'idea che sta informalmente alla base di questo problema consiste nel rappresentare determinate informazioni segrete aggiungendo un **errore** ad esse.

Il problema LWE, nonostante si riduca ad un problema definito su reticolli, non è esso stesso un problema definito su reticolli ma un problema **algebrico**.

In particolare, esso sfrutta alcune proprietà dei **sistemi di equazioni lineari**.

Infatti, un generico sistema della forma

$$Ax = b$$

(espresso in forma matriciale per ragioni di compattezza) può essere risolto facilmente mediante uno dei metodi "classici" come quello di **sostituzione** o quello di **eliminazione di Gauss-Jordan**.

Tuttavia, se ad un sistema di questa forma viene aggiunto del "rumore", ovvero un certo vettore e della forma:

$$e = \begin{bmatrix} e_1 \\ e_2 \\ \dots \\ e_n \end{bmatrix}$$

facendo quindi diventare il sistema della forma

$$Ax + \mathbf{e} = b$$

la situazione cambia drasticamente poiché **non è più possibile** utilizzare i metodi sopracitati per la risoluzione.

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

Il problema LWE è definito usando esattamente tale osservazione, anche se di solito per questo tipo di operazioni si tende a restringersi all'insieme degli interi $\text{mod } q$, con $q \in \mathbb{N}$.

Segue ora una definizione formale [59] del problema LWE.

Definizione 3. Sia $q \in \mathbb{N}$ e sia $\mathbb{Z}_q = \{0, 1, \dots, q - 1\}$. Dati m vettori a_1, a_2, \dots, a_m di k componenti ciascuno (formalmente, definiti sull'insieme \mathbb{Z}_q^k), si definisca $s = \{s_1, s_2, \dots, s_k\} \in \mathbb{Z}_q^k$ vettore **segreto** ed $e = \{e_1, e_2, \dots, e_m\} \in \mathbb{Z}_q^m$ vettore degli **errori**.

Si consideri un generico sistema di equazioni della forma:

$$\left\{ \begin{array}{l} \langle a_1, s \rangle + e_1 = b_1 \text{ (mod } q) \\ \langle a_2, s \rangle + e_2 = b_2 \text{ (mod } q) \\ \vdots \\ \langle a_m, s \rangle + e_m = b_m \text{ (mod } q) \end{array} \right.$$

che scritto in forma compatta (derivata da quella matriciale diventa):

$$As + e = b \text{ (mod } q)$$

Il problema LWE consiste, data una coppia (A, b) ed un vettore e , di trovare il "segreto" s .

Oltre alla variante del problema di **ricerca** appena mostrato esiste anche un problema **decisionale**, del tutto equivalente dal punto di vista della difficoltà, a questo.

L'algoritmo Kyber precedentemente menzionato si basa su una particolare variante del problema appena descritto, ovvero **ModuleLWE** [60], a sua volta derivato dal problema **RingLWE** [61].

La principale differenza tra il problema originale e i suoi "derivati" consiste nel fatto che l'insieme di riferimento non è più \mathbb{Z}_q ma lo spazio dei polinomi a coefficienti in \mathbb{Z}_q , con la relazione $X^n = -1$, formalmente:

$$R_q = \mathbb{Z}_q[X]/(X^n + 1)$$

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

Definiti i concetti matematici alla base della crittografia basata su **reticolis**, è possibile finalmente svolgere una panoramica sui principi di funzionamento dell'algoritmo **Kyber**, che nasce come uno schema di cifratura a chiave pubblica ma viene utilizzato comunemente come **KEM** e di conseguenza **standardizzato** come tale.

La descrizione del funzionamento di Kyber [62] ricalca sostanzialmente quella proposta per LWE, adattata per funzionare con l'insieme R_q .

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

5.4.4 Kyber: generazione delle chiavi

La **chiave privata** in Kyber è composta da un **vettore di k polinomi** con coefficienti relativamente piccoli, chiamato s ;

La **chiave pubblica**, invece, è composta da una coppia (A, t) in cui:

- A è una matrice di dimensione $k \times k$ di polinomi i cui coefficienti sono campionati in maniera casuale da \mathbb{Z}_q ;
- t è un ulteriore vettore di k polinomi.

Il calcolo di t dipende da A , da s ma anche da un ulteriore vettore di k polinomi con coefficienti relativamente piccoli, chiamato **vettore degli errori**.

Formalmente,

$$t = As + e$$

utilizzando le operazioni di addizione e di prodotto riga per colonna.

Come accennato in fase di definizione del problema, risulta essere difficile (a causa del vettore e) recuperare s a partire da (A, t) poiché equivarrebbe a risolvere il problema **ModuleLWE**.

Tutti i coefficienti dei polinomi sono campionati in **maniera uniforme** dall'insieme \mathbb{Z}_q .

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

5.4.5 Kyber: la cifratura

Come un qualsiasi schema di cifratura asimmetrica (dato che formalmente **Kyber** ricade in questa categoria), le operazioni di **cifratura** avvengono con la **chiave pubblica** (ovvero (A, t)).

La cifratura in **Kyber** è composta da **diversi passi preliminari**:

1. Dato un generico messaggio m , per prima cosa se ne estrae la **codifica binaria**: sia m' tale stringa;
2. Le cifre binarie di m' vengono utilizzate come **coefficienti di un polinomio**: sia m_b tale polinomio;
3. m_b viene **scalato** di un fattore pari a $\lfloor \frac{q}{2} \rfloor$ (ovvero il numero intero più vicino a $\frac{q}{2}$) poiché è necessario che i coefficienti dei polinomi devono essere "grandi".

Per l'operazione di cifratura sono necessarie tre **strutture di supporto**, generate in maniera casuale ad ogni nuova cifratura:

- Un vettore r di k componenti detto **vettore di randomizzazione**;
- Un vettore e_1 , anch'esso di k componenti detto **vettore errore**;
- Un polinomio e_2 detto **polinomio errore**.

A questo punto il messaggio m_b è pronto per essere cifrato.

Il **ciphertext** in **Kyber** consiste nella coppia $c = (u, v)$, con u vettore di k componenti e v polinomio, ottenuti nel seguente modo:

$$u = A^T r + e_1$$

$$v = t^T r + e_2 + m$$

dove A^T rappresenta l'operazione di **trasposizione** della matrice A e t^T rappresenta l'operazione di trasposizione del vettore t .

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

5.4.6 Kyber: la decifratura

Una volta ottenuto c , il primo step della decifratura avviene mediante la seguente operazione:

$$m_n = v - s^T u \quad (1)$$

dove s^T rappresenta la **trasposizione** della **chiave privata** s .

Espandendo la (1) si ottiene:

$$m_n = e^T r + e_2 + m + s^T e_1 \quad (2)$$

Il polinomio m_n ottenuto viene detto "rumoroso" poiché di per sé **non contiene** m_b . Tuttavia, siccome i coefficienti di tutti i polinomi (fatta eccezione per m_b , scalato di un fattore $\lfloor \frac{q}{2} \rfloor$) sono relativamente "**piccoli**" è possibile recuperare i coefficienti di quest'ultimo iterando tra i coefficienti di m_n ed operando come segue.

Sia a un generico coefficiente di m_n :

- Se a si avvicina di più a $\lfloor \frac{q}{2} \rfloor$ che a q o a 0, vuol dire che il coefficiente a lui corrispondente in m_b valeva 1, dunque è necessario arrotondarlo a $\lfloor \frac{q}{2} \rfloor$;
- Al contrario, se a si avvicina di più a q o a 0 che a $\lfloor \frac{q}{2} \rfloor$, vuol dire che il coefficiente a lui corrispondente in m_b valeva 0, dunque è necessario arrotondarlo a 0;

A questo punto, m_b è stato recuperato ed è necessario **invertire** l'upscaleing avvenuto in fase di cifratura mediante una divisione per $\lfloor \frac{q}{2} \rfloor$.

Dopo tale operazione è dunque possibile fare riferimento ai coefficienti del risultato per risalire alle cifre di m' e, di conseguenza, di m , completando il processo di **decifratura**.

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

5.4.7 Kyber: discussione sulla sicurezza

Gli algoritmi di crittografia post quantistica si contraddistinguono per una certamente maggiore dimensione delle chiavi e naturalmente l'algoritmo **Kyber** non fa eccezione. Nelle specifiche di tale algoritmo vengono riportate diverse "modalità" di funzionamento, in termini di lunghezza delle chiavi e della *capsula*. La seguente tabella riporta tali valori ed il livello di sicurezza di ciascuna modalità quando confrontato con l'algoritmo **AES**. Tutte le lunghezze sono espresse in **bytes**.

Modalità	sk ⁵	pk ⁶	Lunghezza <i>capsula</i>	Livello di sicurezza
Kyber512	1632	800	768	AES128
Kyber768	2400	1184	1088	AES192
Kyber1024	3168	1568	1568	AES256

Dal punto di vista della struttura interna, l'unica differenza che intercorre tra le tre versioni dell'algoritmo consiste nella variazione del parametro k , introdotto nelle precedenti sezioni e che rappresenta il numero delle componenti dei vettori. In particolare:

- Nel caso di **Kyber512**, $k = 2$;
- Nel caso di **Kyber768**, $k = 3$;
- Nel caso di **Kyber1024**, $k = 4$;

Questa caratteristica rende l'implementazione di **Kyber** estremamente semplice ed efficace poiché tutte e tre le versioni di tale algoritmo possono utilizzare virtualmente lo stesso codice, aumentando notevolmente la **modularità** dello schema.

⁵Lunghezza della **chiave segreta**

⁶Lunghezza della **chiave pubblica**

5 QUANTUM COMPUTING E POST-QUANTUM CRYPTOGRAPHY

Gli altri parametri, invece, risultano **invariati** tra tutte le modalità:

- Il **grado massimo** (n) dei polinomi utilizzati è fissato a 256;
- Il **modulo** (q) utilizzato per definire l'insieme di riferimento è fissato a 3329.

Una discussione sulle **implementazioni software** per questo algoritmo viene proposta nella sezione successiva di tale documento, che svolge una panoramica sul sistema proposto in questo lavoro.

La scelta della versione di Kyber da utilizzare in questo lavoro è ricaduta su Kyber512 per fra fronte al tradeoff necessario tra prestazioni e livello di sicurezza.

6 PROGETTAZIONE E DESCRIZIONE DELLA SOLUZIONE PROPOSTA

6 Progettazione e descrizione della soluzione proposta

L'idea principale che sta alla base di questo lavoro consiste nel definire un meccanismo di **scambio delle chiavi** utilizzando uno schema **quantum-resistant** allo scopo di stabilire una comunicazione sicura tra un **UAV** e la sua **Ground Control Station** utilizzando uno **schema di cifratura simmetrico** per intervenire sul **payload** di messaggi MAVLINK già serializzati e quindi sotto forma di **stream** di byte, come descritto nelle sezioni precedenti, anch'esso **quantum-resistant**.

Per realizzare quanto detto, le modifiche proposte da questo lavoro si concentrano principalmente su tre **componenti**:

- **ArduPilot**, che gioca la parte dell'UAV;
- **QGroundControl**, usato come GCS;
- La libreria **MAVLINK**, ovvero la componente di più basso livello di questo stack tecnologico e che viene utilizzata da entrambi i progetti

6.1 Scambio della chiave: perché Kyber?

Come già ripetuto nel capitolo dedicato a esso dedicato, l'algoritmo **Kyber** è stato scelto poiché risulta essere il vincitore del processo di standardizzazione di schemi di cifratura **quantum-resistant**, rendendolo quindi una scelta quasi obbligata.

Poiché standardizzato come un **KEM**, **Kyber** viene utilizzato come tale. Di conseguenza, il meccanismo di scambio della chiave deve sottostare alle specifiche di tale meccanismo di cifratura e decifratura, utilizzando quindi operazioni di **incapsulamento** e **decapsulamento** di uno **shared secret**, ovvero la chiave da utilizzare con il **cifrario simmetrico**.

È stato, di conseguenza, stabilito un vero e proprio **protocollo** in cui sono eseguite le operazioni appena descritte.

6 PROGETTAZIONE E DESCRIZIONE DELLA SOLUZIONE PROPOSTA

6.2 Cifratura simmetrica: perché AES?

Lo scopo principale di questo lavoro, come ripetuto più volte nel presente documento, consiste nell'utilizzare tecnologie **quantum-resistant**: la scelta della componente di cifratura simmetrica per la comunicazione non può escludere certamente da questo requisito. Di conseguenza, si è scelto di utilizzare il cifrario AES poiché è opinione comune che tale cifrario resista ad attacchi di crittoanalisi condotti da algoritmi quantistici. Il miglior "attacco quantistico" a questo cifrario conosciuto fino ad ora consiste in quello veicolato tramite l'uso del cosiddetto **Algoritmo di Grover** [63], un algoritmo quantistico originariamente pensato per effettuare operazioni efficienti di ricerca in database **non ordinati** con una complessità temporale di $\mathcal{O}(\sqrt{N})$ e spaziale di $\mathcal{O}(\log N)$. Quando applicato in un contesto di **ricerca esaustiva**, tale algoritmo permette quindi di **dimezzare** il tempo necessario all'individuazione di un valore e, nel caso di attacchi *bruteforce*, de recupero della chiave di cifratura.[64]

Quanto appena detto conduce quindi ad un duplice risultato:

- Nel caso di AES, il livello di sicurezza è quindi **dimezzato**: ad esempio, per la variante AES-128 il livello di sicurezza in un contesto quantistico si riduce a quello offerto da una (ipotetica!) variante con chiavi a 64 bit di tale schema in un contesto classico;
- La variante AES-256 passa ad un livello di sicurezza pari a quello di AES-128, considerato oggi ancora **sufficiente**.

Per questa ragione, lo schema AES viene considerato, allo stato attuale, **quantum-resistant**.

6.3 Definizione del protocollo

Fatta questa premessa, è ora necessario capire *cosa* fare per effettuare correttamente uno scambio di chiavi allo scopo di stabilire una comunicazione cifrata successivamente.

6 PROGETTAZIONE E DESCRIZIONE DELLA SOLUZIONE PROPOSTA

È necessario quindi stabilire *quale* delle due "parti" impegnate nella comunicazione, con riferimento al meccanismo di funzionamento di un KEM descritto in precedenza, deve svolgere le operazioni di `Gen()`, `Encaps()` e `Decaps()`.

In fase di progettazione del nuovo sistema sono state effettuate le seguenti scelte, tenendo anche conto della possibile⁵ differenza di capacità computazionale tra l'UAV e la GCS (che, si ricorda, è generalmente ospitata su hardware consumer, senza particolari limitazioni di risorse):

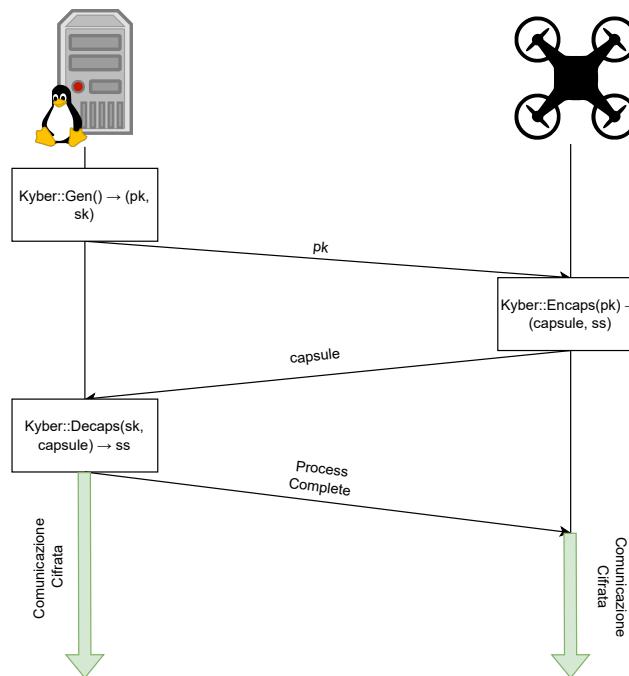
- L'operazione di `Gen()` viene effettuata dalla componente **GCS**, che **invia la chiave pubblica** all'UAV;
- Una volta ricevuta la chiave pubblica, la componente **UAV** effettua l'operazione di `Encaps()` (che implica la generazione di una **chiave segreta**). Successivamente, lo stesso UAV **invia la capsula** alla GCS;
- Alla ricezione della capsula, la **GCS** effettua l'operazione di `Decaps()`, **estrae la chiave segreta**, la **imposta** ed infine **invia conferma** all'UAV;
- A questo punto, **le due parti** posseggono la stessa chiave segreta: è finalmente possibile iniziare la **comunicazione cifrata** utilizzando la componente **MAVLINK**.

Nello schema presente qui di seguito:

- `pk` ed `sk` rappresentano rispettivamente la **chiave pubblica** e la **chiave privata** restituite dalla procedura `Gen()` dell'algoritmo **Kyber**;
- `capsule` rappresenta la **capsula** generata dalla procedura `Encaps()` dell'algoritmo **Kyber**;
- `ss` rappresenta lo **shared secret**, utilizzata come **chiave segreta** per la comunicazione cifrata mediante cifrario **AES**.

⁵In realtà, molto probabile!

6 PROGETTAZIONE E DESCRIZIONE DELLA SOLUZIONE PROPOSTA



Una rappresentazione grafica del protocollo descritto

Si noti che quello mostrato fino ad ora è una descrizione ad alto livello che permette di farsi un'idea di ciò che accade durante la fase di **scambio della chiave**. Una descrizione a più basso livello dei messaggi effettivamente scambiati tra le due parti della comunicazione viene svolta man mano che vengono presentati i vari step che hanno permesso la creazione del sistema proposto nella sezione successiva del presente documento.

7 Implementazione della soluzione proposta

Dopo una descrizione sullo stato dell’arte del contesto applicativo, delle fondamenta scientifiche e matematiche dell’algoritmo **Kyber** ed a seguito di una descrizione logica del sistema proposto è ora finalmente possibile svolgere una panoramica ad un livello più basso delle modifiche ai progetti precedentemente introdotti.

1. Installazione di un’implementazione di **Kyber** nell’ambiente di sviluppo;
2. Introduzione di nuovi messaggi MAVLINK 2.0;
3. Integrazione di **Kyber** all’interno di ArduPilot;
4. Integrazione di **Kyber** all’interno di QGroundControl;
5. Alterazione delle funzioni di **ArduPilot**;
6. Alterazione delle funzioni di **QGroundControl**;
7. AES ed alterazione della libreria MAVLINK.

Le sezioni successive approfondiscono ciascuno di questi.

7.1 Step 1: installazione di un’implementazione di Kyber nell’ambiente di sviluppo

Sul sito web del progetto CRYSTALS [65] viene fatta una panoramica delle soluzioni software esistenti che permettono l’utilizzo dello schema di incapsulamento **Kyber**.

Su GitHub è presente una **implementazione di riferimento** [66] di tale schema nel linguaggio C ottimizzata per processori Intel x86 che sfrutta il set di istruzioni AVX2 (che aggiunge il supporto ai vettori a 256 bit, permettendo calcoli in **virgola mobile** ad alte prestazioni [67]).

Esistono tuttavia implementazioni di **Kyber** integrate in progetti più strutturati come **PQClean** o **OpenQuantumSafe**. Quest’ultimo consiste in "un

framework per la prototipazione e lo sviluppo di crittografia quantum-resistant" [68].

7.1.1 liboqs: descrizione ed installazione

Il progetto di punta di questa organizzazione è però **liboqs**: una libreria open source che fornisce accesso ad implementazioni di diversi algoritmi post quantistici mediante un'API semplificata. Oltre a **Kyber**, questa suite include anche altri algoritmi [69] **KEM** candidati nel processo di standardizzazione del NIST (Classic McEliece, BIKE, HQC, FrodoKEM ed NTRU-Prime) e degli schemi di **firma digitale** (CRYSTALS-Dilithium, Falcon e SPHINCS+). Il codice sorgente di tale libreria è presente su GitHub [70] insieme ad una guida per la compilazione e l'installazione del progetto, qui brevemente riportata:

- Innanzitutto è necessario clonare la repository del progetto in una directory locale mediante il comando

```
git clone -b https://github.com/open-quantum-safe/liboqs
```

e dunque spostarsi nella directory che contiene i sorgenti:

```
cd liboqs/
```

- Poiché **liboqs** usa il sistema di building **cmake** utilizzato insieme a **ninja**, è necessario installare alcune dipendenze. Per i sistemi GNU/Linux basati su **Debian** (come Ubuntu) il comando da eseguire è il seguente:

```
sudo apt install astyle cmake gcc ninja-build libssl-dev  
python3-pytest python3-pytest-xdist unzip xsllproc doxygen  
graphviz python3-yaml valgrind
```

- È tutto pronto per iniziare il processo di building:

```
mkdir build && cd build  
cmake -GNinja ..  
ninja
```

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

- Una volta terminato il processo di building, la **libreria statica liboqs.a** sarà disponibile nella directory `/build/lib/`. Se, invece, si desidera generare uno **shared object** è possibile passare il parametro `-DBUILD_SHARED_LIBS=ON` durante l'invocazione di `cmake`. Il risultato della compilazione, in questo caso, sarà il file `liboqs.so` nello stesso percorso menzionato in precedenza, oltre che la creazione di una directory `include/`, che contiene gli **headers** relativi alla libreria.
- Infine, è possibile generare un **pacchetto .deb** per installare la libreria in maniera *system-wide* mediante un job di `ninja`:

```
ninja package
```

e successivamente installando il pacchetto appena generato:

```
sudo dpkg -i <nome_pacchetto>.deb
```

A questo punto, `liboqs` può essere utilizzata all'interno di un generico programma C/C++ utilizzando la direttiva:

```
#include <oqs/oqs.h>
```

ed aggiungendo i flag `-loqs -lcrypto` in fase di building del programma per comunicare al **linker** il requisito di tale libreria (che possiede `libcrypto` come dipendenza).

Quest'ultimo aspetto viene qui sottolineato perché necessario negli step successivi descritti in questo capitolo.

Tale libreria, in fase di compilazione, è configurabile in modo da permettere l'abilitazione solo di algoritmi specifici mediante il *flag* `OQS_ALGS_ENABLED`.

In particolare, per ogni schema KEM abilitato nella libreria `liboqs` offre tre funzioni [71] che ricalcano sostanzialmente le procedure `Gen()`, `Encaps()` e `Decaps()` di un generico schema KEM. Per `Kyber512`, ad esempio:

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

- `OQS_KEM_kyber_512_keypair()` permette di generare la coppia di chiavi (`pk`, `sk`): corrisponde al generico `Gen()`;
- `OQS_KEM_kyber_512_encaps()` permette di generare uno shared secret ed "incapsularlo": corrisponde (naturalmente!) al generico `Encaps()`;
- `OQS_KEM_kyber_512_decaps()` permette di decapsulare lo shared secret presente nel cfrato: corrisponde al generico `Decaps()`;

7.2 Step 2: Introduzione di nuovi messaggi MAVLINK 2.0

Nella sezione dedicata al funzionamento della libreria MAVLINK è stato sottolineato come fosse possibile **estenderlo** andando a modificare i file XML che descrivono i messaggi previsti per un certo "dialetto" del protocollo.

Per modellare il funzionamento dello schema di scambio delle chiavi è contestualmente necessario prevedere dei messaggi ad hoc che permettano lo scambio delle informazioni illustrate nella sezione precedente.

Poiché al momento dell'implementazione di questo lavoro⁶ il progetto **QGroundControl** utilizzava il dialetto **ardupilotmega** (prima di effettuare la migrazione al dialetto **all** a fine Agosto 2023) i nuovi messaggi sono stati aggiunti a tale dialetto e, di conseguenza, inclusi nel file **ardupilotmega.xml**.

In particolare, sono stati aggiunti **quattro** messaggi:

- KEYEXCHANGE;
- KEYEXCHANGEGCSACK;
- CAPSULE;
- CAPSULEACK.

7.2.1 KEYEXCHANGE

Questo messaggio **viene inviato dall'UAV alla GCS** e corrisponde ad una *richiesta* da parte del drone di una **porzione** della chiave pubblica generata dalla GCS.

La ragione dietro all'adozione di questo approccio è duplice:

- La prima è da ricercarsi in un dettaglio implementativo presente nella piattaforma **ArduPilot**. In questo software è presente un meccanismo che permette di determinare a tempo di compilazione quale tipo di messaggio deve essere inviato e con quale frequenza. Un meccanismo del genere

⁶Inizio Agosto 2023

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

non è invece presente nativamente all'interno del software **QGroundControl**, dunque si è scelto di "delegare" tale responsabilità all'UAV;

- La seconda deriva da un limite intrinseco del protocollo MAVLINK 2.0, che, come già accennato in una sezione precedente, fissa la dimensione del payload di un messaggio a 255 bytes. La dimensione della **chiave pubblica** per Kyber512 è invece fissata ad 800 bytes: è stato quindi necessario **frammentarla** in diversi *chunks* durante la fase di scambio delle chiavi.

La struttura della entry corrispondente al messaggio KEYEXCHANGE all'interno del file `ardupilotmega.xml` è la seguente:

```
<message id="10000" name="KEYEXCHANGE">
    <description>Key Exchange Mechanism (ArduPilot version).</description>
    <field type="uint8_t" name="seq">Sequence Number.</field>
    <field type="uint16_t" name="chunk_start">Read from.</field>
    <field type="uint16_t" name="chunk_end">Read to.</field>
    <field type="uint8_t[16]" name="iv">IV</field>
</message>
```

In particolare:

- Il campo **seq** consiste in un **byte** che rappresenta il **numero di sequenza** del messaggio all'interno del processo di scambio della chiave;
- I campi **chunk_start** e **chunk_end** rappresentano rispettivamente gli **indici** di inizio e della fine del chunk della chiave pubblica che l'UAV desidera ricevere. Entrambi questi campi sono **interi a 16 bit**;
- Il campo **IV** rappresenta il **vettore di inizializzazione** utilizzato per la comunicazione cifrata instaurata alla fine del processo di scambio delle chiavi. Consiste in un vettore di 128 bit, come richiesto dalle specifiche del cifrario AES.

7.2.2 KEYEXCHANGEGCSACK

Questo messaggio rappresenta la "risposta" della **Ground Control Station** ad un messaggio KEYCHANGE da parte del drone. Contiene la porzione di chiave pubblica richiesta da quest'ultimo. La sua struttura è la seguente:

```
<message id="10420" name="KEYEXCHANGEGCSACK">
    <description>Key Exchange Mechanism (GCS version).</description>
    <field type="uint8_t" name="seq_ack">Sequence Number ACK.</field>
    <field type="uint16_t" name="chunk_start">Read from.</field>
    <field type="uint16_t" name="chunk_end">Read to.</field>
    <field type="uint8_t" name="more_data">More data to read.</field>
    <field type="uint8_t[200]" name="data">Data</field>
</message>
```

In particolare:

- Il campo `seq_ack` rappresenta il **prossimo** sequence number che la GCS si aspetta. Funge, come suggerisce il nome, da meccanismo di **Acknowledgement**;
- I campi `chunk_start` e `chunk_end` rappresentano rispettivamente gli **indici** di inizio e della fine del chunk della chiave pubblica che l'UAV desidera ricevere. Entrambi questi campi sono **interi a 16 bit**. Riflettono quelli inviati nel messaggio precedente;
- Il campo `more_data` rappresenta un flag che permette, quando è impostato a 0, di avvisare l'UAV che lo scambio della chiave pubblica è terminato.

7.2.3 CAPSULE e CAPSULEACK

In maniera speculare a quanto avviene durante lo scambio della chiave pubblica, il messaggio CAPSULE rappresenta l'invio, da parte dell'UAV di una porzione della **capsula**, generata dopo aver ricevuto completamente la chiave pubblica ed aver invocato `Encaps()`.

La struttura di CAPSULE è la seguente:

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

```
<message id="10421" name="CAPSULE">
    <description>KEM Capsule.</description>
    <field type="uint8_t" name="seq">Sequence Number.</field>
    <field type="uint16_t" name="chunk_start">Read from.</field>
    <field type="uint16_t" name="chunk_end">Read to.</field>
    <field type="uint8_t" name="more_data">More data to read.</field>
    <field type="uint8_t[192]" name="data">Data</field>
</message>
```

In particolare:

- Il campo **seq** consiste in un **byte** che rappresenta il **numero di sequenza** del messaggio all'interno del processo di scambio della capsula;
- I campi **chunk_start** e **chunk_end** rappresentano rispettivamente gli **indici** di inizio e della fine del chunk della capsula che l'UAV desidera inviare alla GCS. Entrambi questi campi sono **interi a 16 bit**;
- Il campo **more_data** rappresenta un flag che permette, quando è impostato a 0, di avvisare l'UAV che lo scambio della capsula è terminato e la GCS può di conseguenza invocare **Decaps()**.

Il messaggio CAPSULEACK testimonia semplicemente l'avvenuta ricezione della GCS di un messaggio CAPSULE ed in particolare contiene **il prossimo numero di sequenza atteso**.

Ecco la struttura di un messaggio CAPSULEACK:

```
<message id="10422" name="CAPSULEACK">
    <description>KEM Capsule ACK.</description>
    <field type="uint8_t" name="ack">Next Sequence Number.</field>
</message>
```

7.3 Step 3: Integrazione di Kyber all'interno di ArduPilot

Come accennato in fase di presentazione di tale piattaforma software, **ArduPilot** utilizza il sistema di building **waf**. Di conseguenza, l'introduzione di nuove librerie all'interno di un progetto che fa uso di questo sistema di building deve naturalmente avvenire nei suoi file di configurazione. Per rendere quanto più semplice possibile il processo di integrazione e di linking di **liboqs** (come già descritto nello **Step 1**), tale libreria è stata installata in maniera system-wide sulla macchina su cui è stato svolto il lavoro allo scopo di minimizzare le modifiche a cui sottoporre il progetto. Di conseguenza, l'unico accorgimento da adottare consiste nell'aggiungere i flag **-loqs -lcrypto** in fase di linking. Tale operazione, di norma, si effettua andando a modificare la variabile d'ambiente **LDFLAGS**.

Nel caso di **waf**, invece, è necessario andare a modificare il file **wscript** (uno script Python), presente nella root della repository del progetto **ArduPilot**.

In particolare, è necessario considerare la funzione **build(bld)**, il cui parametro espone l'oggetto **env**, che modella **una copia** delle **variabili d'ambiente** da consultare in fase di compilazione e di linking, tra cui **CXXFLAGS** e (convenientemente!) **LDFLAGS**.

Ciascuna di queste variabili d'ambiente è resa come una **lista di valori**, che può essere modificata ed **estesa**. L'aggiunta dei due flag ad **LDFLAGS** si effettua quindi come segue:

```
def build(bld):
    bld.env.LDFLAGS += ['-loqs', '-lcrypto']
```

A questo punto è possibile ricompilare **ArduCopter** mediante il comando:

```
./waf copter -v
```

utilizzando lo switch **-v** per abilitare la modalità *verbose* e di conseguenza accertarsi che suddetti flag siano presenti nella stringa del linker mostrata durante l'ultima fase del processo di building.

7.4 Step 4: Integrazione di Kyber all'interno di QGroundControl

Al contrario di **ArduPilot**, **QGroundControl** utilizza il framework **Qt** per la gestione del progetto e delle procedure di building. Per andare ad integrare **liboqs** è quindi necessario andare ad intervenire sulla configurazione di Qt stesso.

Il file su cui intervenire, nello specifico, si chiama **QGCExternalLibs.pri** ed è situato nella *root* della repository del progetto. L'inclusione di librerie installate in maniera system-wide, come nel caso di ArduPilot descritta nello Step 4, si effettua semplicemente alterando i flags che il sistema di building passa al linker. Nel caso di Qt, è possibile alterare la variabile **LIBS** alla fine del file prima menzionato ed aggiungendo i flags **-loqs -lcrypto** in questo modo:

```
LIBS += -loqs -lcrypto
```

Dopo di che, è necessario eseguire una **build pulita del progetto**, andando ad eseguire i seguenti comandi:

```
cd build/  
make clean
```

e successivamente rieseguendo i comandi descritti nella sezione relativa alla compilazione di QGroundControl nel presente documento.

7.5 Step 5: Alterazione delle funzioni di ArduPilot

Una volta descritto il funzionamento ad alto livello del processo di scambio delle chiavi, è ora possibile scendere ancora di più nel dettaglio e descrivere quali sono stati i cambiamenti alla struttura del codice sorgente del progetto **ArduPilot**.

In particolare, il grosso dei cambiamenti si è concentrato sostanzialmente in due file:

- Il file `GCS_Common.cpp` presente nella directory `libraries/GCS_MAVLink`, partendo dalla *root* della repository del progetto. Questo file contiene la logica di gestione dei messaggi MAVLINK **comune per ogni dispositivo**;
- Il file `GCS_MAVLink.cpp` presente invece nella directory `ArduCopter/`, che invece gestisce lo scheduling dei messaggi MAVLINK specifici per il progetto **ArduCopter**.

Segue una panoramica esaustiva sulle modifiche applicate ad entrambi i file.

7.5.1 ArduCopter/GCS_MAVLink.cpp

In questo file è presente la logica che gestisce lo scheduling dei messaggi MAVLINK per il dispositivo (sia virtuale che fisico) **Copter**, ossia quello utilizzato in questo lavoro.

Per prima cosa, è stato necessario andare ad aggiungere le costanti `MSG_KEYEXCHANGE` e `MSG_CAPSULE` (esposte dalla libreria MAVLINK) al vettore `STREAM_EXTRA3_msgs[]` per informare la piattaforma che si intende includere tali messaggi nel processo di invio periodico.

```
static const ap_message STREAM_EXTRA3_msgs[] = {  
    // messaggi precedenti  
    MSG_KEYEXCHANGE,  
    MSG_CAPSULE,
```

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

```
// eventuali messaggi successivi  
};
```

L'ordine con cui vengono inseriti i messaggi in questo vettore è ininfluenzante.

È necessario che tali costanti siano aggiunte anche nel file `/libraries/GCS_MAVLink/ap_messages.h`, nell'`enum` presente al suo interno, immediatamente prima della costante `MSG_LAST` che, come suggerito da un commento in sudetto file, deve essere sempre l'ultimo.

```
enum ap_message : uint8_t {  
    // messaggi precedenti  
    MSG_KEYEXCHANGE,  
    MSG_KEYEXCHANGEGCSACK,  
    MSG_LAST // MSG_LAST must be the last entry in this enum  
};
```

7.5.2 `libraries/GCS_MAVLink/GCS_Common.cpp`

È invece in questo file che si concentra il grosso delle modifiche effettuate.

Prima di tutto, è necessario includere `oqs.h` come accennato in precedenza per poter accedere alle API di `liboqs`:

```
#include <oqs/oqs.h>
```

Successivamente, è necessario intervenire sulla funzione

```
bool GCS_MAVLINK::try_send_message(const enum ap_message id)
```

che, per ogni messaggio dello scheduler, richiama una funzione (definita ad hoc in questo stesso file) che si occuperà di inviare un'istanza di quel messaggio. `try_send_message()` è implementata mediante uno `switch` che permette di discriminare il messaggio sulla base del suo tipo. È quindi sufficiente **aggiungere** un `case` in corrispondenza dei due messaggi che l'autopilot desidera inviare (il messaggio di Key Exchange e la capsula), come mostrato di seguito.

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

```
bool GCS_MAVLINK::try_send_message(const enum ap_message id)
{
    switch(id) {
        // messaggi precedenti
        case MSG_KEYEXCHANGE:
            CHECK_PAYLOAD_SIZE(KEYEXCHANGE);
            send_keyexchange();
            break;

        case MSG_CAPSULE:
            CHECK_PAYLOAD_SIZE(CAPSULE);
            send_capsule();
            break;

        // messaggi successivi
    }
}
```

Successivamente, vanno definite le funzioni `send_keyexchange()` e `send_capsule()`:

```
void GCS_MAVLINK::send_keyexchange() const {
    if (key_exchange_complete) {
        return;
    }

    if (is_iv_set == false) {
        RAND_bytes(iv, 16);
        is_iv_set = 1;
    }

    mavlink_msg_keyexchange_send(
```

```
    chan,  
    keyexchange_seq,  
    request_remote_key_start,  
    request_remote_key_end,  
    iv  
);  
}
```

Questa funzione si occupa di richiedere una porzione della chiave pubblica (i cui estremi, in termini di numero di bytes, sono definiti dalle variabili `request_remote_key_start` e `request_remote_key_end`, definite esternamente ed inizializzate rispettivamente a 0 ed a 200) della GCS, al contempo inviando un `vettore di inizializzazione` utilizzato successivamente per la comunicazione cifrata e generato mediante la funzione `RAND_bytes()` presente nella libreria `rand.h` di OpenSSL che, di conseguenza, va inclusa.

```
void GCS_MAVLINK::send_capsule() const {  
  
    if (!should_send_capsule) {  
        return;  
    }  
  
    if (!is_capsule_generated) {  
        OQS_init();  
        if (OQS_KEM_kyber_512_encaps(capsule, key, remote_key)  
            != OQS_SUCCESS) {  
            printf("Failed to encaps, quitting.\n");  
            exit(-1);  
        } else {  
            printf("Generated secret key\n");  
            printf("Generated capsule\n");  
        }  
        OQS_destroy();  
    }  
}
```

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

```
    is_capsule_generated = true;

} else {

    uint8_t chunk[192];

    int index = 0;

    for (int i = read_capsule_start;
          i < read_capsule_end;
          i++) {
        chunk[index++] = capsule[i];
    }

    short more_data = (read_capsule_end ==
                      OQS_KEM_kyber_512_length_ciphertext) ? 0 : 1;

    mavlink_msg_capsule_send(
        chan,
        capsule_message_sequence_number,
        read_capsule_start,
        read_capsule_end,
        more_data,
        chunk
    );
}
```

In sostanza, questa funzione si comporta in maniera simile a `send_keyexchange()` e si occupa di inviare alla GCS, una volta ottenuta completamente la sua chiave remota, una porzione della **capsula** generata da `Encaps()` delimitata dagli indici `read_capsule_start` e `read_capsule_end`. Gli argomenti di `Encaps()` sono vettori di bytes (leggasi, `uint8_t`) definiti esternamente ed allocati con lo spazio necessario secondo le specifiche di Kyber.⁷

⁷liboqs espone delle costanti che rappresentano le dimensioni in bytes della coppia di chiavi, della capsula e dello *shared secret*

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

Quando il protocollo di scambio della chiave pubblica e della capsula è terminato, questi due messaggi **non vengono più inviati**.

Oltre alla definizione di funzioni che modellano l'invio di messaggi MAVLINK, è stato necessario codificare delle funzioni che modellassero la **ricezione dei messaggi di gement**, ovvero KEYEXCHANGEGCSACK e CAPSULEACK. Entrambe queste funzioni si occupano di informare ArduPilot che la GCS ha ricevuto la porzione di informazione precedente e che è di conseguenza possibile avanzare nel processo di scambio(in termini di numeri di sequenza e di indici da richiedere ed inviare).

Tali funzioni vanno richiamate nel corpo della funzione handle_common_message, come segue:

```
/*
  handle messages which don't require vehicle specific data
*/
void GCS_MAVLINK::handle_common_message(const mavlink_message_t &msg)
{
    case MAVLINK_MSG_ID_KEYEXCHANGEGCSACK: {
        handle_keyexchangepcsack(msg);
        break;
    }

    case MAVLINK_MSG_ID_CAPSULEACK: {
        handle_capsuleack(msg);
        break;
    }
}
```

Ecco il corpo della funzione handle_keyexchangepcsack():

```
void GCS_MAVLINK::handle_keyexchangepcsack
(const mavlink_message_t &msg) const {
```

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

```
    mavlink_keyexchangegecsack_t decoded_message;
    mavlink_msg_keyexchangegecsack_decode(&msg, &decoded_message);

    if (decoded_message.seq_ack != keyexchange_seq + 1) {
        // Out-of-sequence
        exit(-1);
    }

    int chunk_start = decoded_message.chunk_start;
    int chunk_end = decoded_message.chunk_end;

    uint8_t *data = decoded_message.data;

    int index = 0;
    for (int i = chunk_start; i < chunk_end; i++) {
        remote_key[i] = data[index++];
    }

    if (decoded_message.more_data == 0) {
        key_exchange_complete = true;
        should_send_capsule = true;
    } else {
        keyexchange_seq = decoded_message.seq_ack;
        request_remote_key_start = request_remote_key_end;
        request_remote_key_end += 200;
    }
}
```

Tale meccanismo è stato ripreso anche per il messaggio CAPSULEACK:

```
void GCS_MAVLINK::handle_capsuleack
(const mavlink_message_t &msg) const {
```

```

mavlink_capsuleack_t decoded_message;
mavlink_msg_capsuleack_decode(&msg, &decoded_message);

if (decoded_message.ack !=

capsule_message_sequence_number + 1) {

    // Out-of-sequence
    exit(1);

} else {

    if (read_capsule_end ==

OQS_KEM_kyber_512_length_ciphertext) {

        should_send_capsule = false;

        printf("Successfully sent the capsule.\n");

        set_session_key(key, iv);

        // we are done.

    } else {

        read_capsule_start = read_capsule_end;

        read_capsule_end += 192;

        capsule_message_sequence_number = decoded_message.ack;

    }

}
}

```

In quest'ultimo caso, quando la capsula è stata scambiata, viene impostata anche la **chiave di sessione**, generata in precedenza mediante chiamata ad `Encaps()`: da questo momento in poi è possibile iniziare la **comunicazione cifrata** tra l'UAV e la GCS. La funzione `set_session_key()` viene definita nella libreria MAVLINK, che sarà oggetto di discussione successivamente.

7.6 Step 6: Alterazione delle funzioni di QGroundControl

L'altro progetto che è stato necessario modificare in questo lavoro consiste, come detto, in **QGroundControl**, la cui responsabilità è tripla:

- **Generare** la coppia (pk, sk) ed inviare pk su richiesta dell'UAV;
- **Ricevere** la capsula creata mediante pk dall'UAV;
- **Decapsulare** la capsula ed estrarre la **chiave di sessione**.

Le modifiche effettuate a questo progetto si concentrano principalmente nel file `MAVLinkProtocol.cc` e, di conseguenza, nel suo *header file* `MAVLinkProtocol.h`, che si occupa di gestire i messaggi MAVLINK in entrata ed in uscita dalla GCS e di interagire con il **LinkManager**, ovvero il gestore le connessioni. Entrambi questi file si trovano nella directory `src/comm/`, a partire dalla *root* della repository del progetto.

7.6.1 src/comm/MAVLinkProtocol.cc

In suddetto file è presente una funzione chiamata

```
void MAVLinkProtocol::receiveBytes(LinkInterface* link,
                                     QByteArray b)
{ ... }
```

che si occupa di effettuare il parsing, un carattere alla volta, di un messaggio MAVLINK deserializzato ed offre (a parsing completato) la possibilità di eseguire del codice in base all'**ID** del messaggio mediante un costrutto **if-else**.

Data la prolissità del corpo della funzione, se ne riporta qui il funzionamento generale e gli **snippet** riguardanti gli aspetti più importanti.

In particolare, **alla ricezione dei messaggi KEYEXCHANGE e CAPSULE**, la GCS emette rispettivamente i messaggi KEYEXCHANGEGCSACK ed CAPSULEACK mediante l'aggiunta di un **case** dove previsto.

7.6.2 Invio di KEYEXCHANGEGCSACK

Nel caso della ricezione di un messaggio di tipo KEYEXCHANGE, la GCS preleva la porzione di chiave pubblica richiesta dall'UAV e, nel caso del primo messaggio, chiama la procedura Gen() di Kyber, come segue:

```
if (are_keys_generated == 0) {  
    OQS_KEM_kyber_512_keypair(  
        ground_control_station_public_key,  
        ground_control_station_secret_key  
    );  
    are_keys_generated = 1;  
}
```

successivamente, viene costruito il buffer che ospita la porzione di chiave richiesta:

```
uint16_t start_from = keyExchange.chunk_start;  
uint16_t end_to = keyExchange.chunk_end;  
  
uint8_t data[200];  
int index = 0;  
for (int i = start_from; i < end_to; i++) {  
    data[index++] = ground_control_station_public_key[i];  
}
```

Infine, viene allocato e popolato un messaggio KEYEXCHANGEGCSACK:

```
mavlink_message_t keyExchangeAck;  
mavlink_msg_keyexchangegcsack_pack_chan(  
    getSystemId(),  
    getComponentId(),  
    link->mavlinkChannel(),  
    &keyExchangeAck,
```

```
    incoming_sequence_number + 1,  
    start_from,  
    end_to,  
    more_data,  
    data  
);
```

Gli indici `start_from` ed `end_to` coincidono con quelli del messaggio `KEYEXCHANGE` che ha triggerato l'invio del presente messaggio. Il flag `more_data` segnala che il messaggio che sta per essere inviato rappresenta, quando posto a 0, **l'ultima porzione di chiave** da comunicare.

Il messaggio viene poi serializzato:

```
uint8_t serialization_buffer[MAVLINK_MAX_PACKET_LEN] ;  
uint8_t length = mavlink_msg_to_send_buffer(  
    serialization_buffer,  
    &keyExchangeAck  
);
```

e passato al `LinkManager`, a cui viene delegato l'invio effettivo:

```
link->writeBytesThreadSafe(  
    (const char *) serialization_buffer,  
    length  
);
```

7.6.3 Invio di CAPSULEACK

Il funzionamento del meccanismo di invio di questo messaggio rispecchia a grandi linee quello appena visto per `KEYEXCHANGEGCSACK`.

In particolare, la GCS recupera gli indici dal payload del messaggio `CAPSULE` appena ricevuto:

```
mavlink_capsule_t capsule;  
mavlink_msg_capsule_decode(&_message, &capsule);
```

```
uint8_t* data = capsule.data;
uint16_t startIndex = capsule.chunk_start;
uint16_t endIndex = capsule.chunk_end;
```

Successivamente, imposta la porzione di capsula ricevuta:

```
uint8_t data_index = 0;

for (uint16_t index = startIndex; index < endIndex; index++) {
    received_capsule[index] = data[data_index++];
}
```

E, quando il flag `more_data` del messaggio CAPSULE acquisisce finalmente il valore 0, invoca la procedura `Decaps()`, impostando contestualmente lo `shared_secret` appena estratto ed il vettore di inizializzazione ricevuto in precedenza:

```
if (capsule.more_data == 0) {

    uint8_t shared_secret[OQS_KEM_kyber_512_length_shared_secret];
    if (OQS_KEM_kyber_512_decaps(shared_secret, received_capsule,
        ground_control_station_secret_key) == OQS_SUCCESS) {
        set_session_key(shared_secret, iv);
    } else {
        exit(-1);
    }
}
```

Infine, viene inviato il messaggio CAPSULEACK, in maniera del tutto analoga a quanto visto per KEYEXCHANGEGCSACK:

```
mavlink_message_t capsule_ack;
mavlink_msg_capsuleack_pack_chan(
    getSystemId(),
    getComponentId(),
```

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

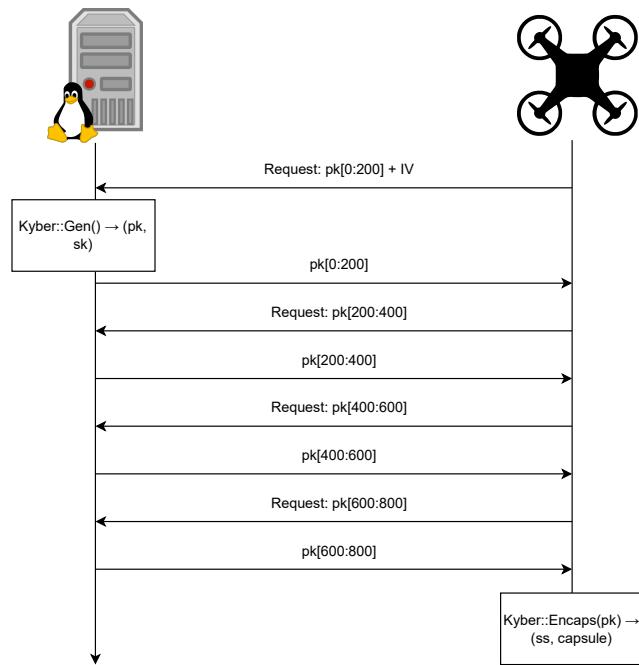
```
link->mavlinkChannel() ,  
&capsule_ack,  
capsule.seq + 1  
);  
  
uint8_t serialization_buffer[MAVLINK_MAX_PACKET_LEN] ;  
uint8_t length = mavlink_msg_to_send_buffer(  
    serialization_buffer,  
&capsule_ack  
);  
link->writeBytesThreadSafe(  
    (const char *) serialization_buffer,  
    length  
);
```

A questo punto, la GCS è finalmente pronta per iniziare la comunicazione cifrata con l'UAV.

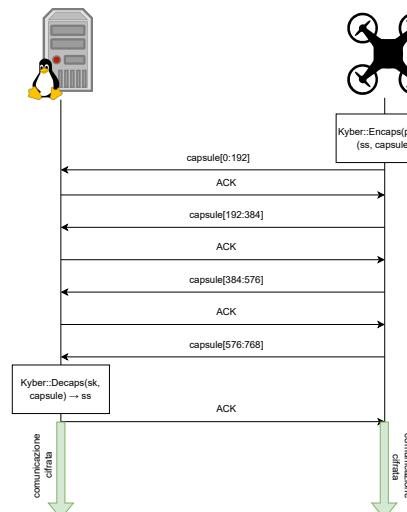
7.6.4 Riassumendo

Ora che è stato descritto nel dettaglio il funzionamento del meccanismo di scambio della chiave e della capsula, è possibile fornire una versione aggiornata e più dettagliata del grafico riassuntivo mostrato nel capitolo dedicato alla progettazione del presente lavoro.

Per quanto riguarda lo scambio della chiave:



Per quanto riguarda il protocollo di scambio della capsula, invece:



7.7 Step 7: Alterazione della libreria MAVLINK

L'ultima parte delle modifiche effettuate per questo lavoro riguarda proprio la libreria C MAVLINK, comune ad entrambi i progetti. In particolare, a questo livello è stata inserita la logica per gestire la comunicazione cifrata tra le due parti e, nello specifico, sono state richiamate le funzioni di una libreria che implementa il cifrario **AES**, la cui scelta è giustificata nel capitolo precedente.

7.7.1 AES: implementazione

Si descrive ora il processo di integrazione di un'implementazione del cifrario **AES** all'interno della libreria MAVLINK.

L'implementazione scelta per tale processo si chiama **tiny-AES-c**, proposta dall'utente **kokke**, ed il suo codice è disponibile liberamente su GitHub. [72]

Tale implementazione è stata scelta per due ragioni pratiche:

- Non è necessario alcun tipo di **compilazione** preventiva;
- L'utilizzo di memoria derivato dalla semplice inclusione di tale libreria in un progetto **non supera** i 200 bytes.

La libreria, recita il suo README, offre le varianti di AES con chiavi da **128**, **192** e **256** bit nelle modalità operative **ECB**, **CBC** e **CTR**.

La libreria si compone sostanzialmente di due file:

- **aes.h**: un header file dove sono dichiarate le funzioni e definiti i parametri dell'algoritmo, ovvero la **modalità operativa** da utilizzare e la **lunghezza della chiave** desiderata;
- **aes.c**: un file sorgente che contiene l'implementazione concreta delle funzioni dichiarate nell'header file.

La configurazione scelta per l'utilizzo consiste in:

- **Lunghezza della chiave di 256 bit**, alla luce del ragionamento svolto nella precedente sezione di questo capitolo;

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

- **Modalità operativa CTR**, poiché la modalità ECB viene generalmente reputata insicura e la modalità CBC richiederebbe il **padding** del plain text ad un multiplo di 16 bytes, operazione non sempre possibile a causa delle limitazioni alla grandezza del payload di un messaggio MAVLINK.

Le modalità di integrazione della libreria varia a seconda del progetto:

- Nel caso di QGroundControl, è sufficiente aggiungere le seguenti righe alla fine di `QGCExternalLibs.pri`:

```
SOURCES += libs/mavlink/include/mavlink/v2.0/aes.c  
HEADERS += libs/mavlink/include/mavlink/v2.0/aes.h
```

Successivamente, è necessario posizionare i file `aes.c` ed `aes.h` nella directory `libs/mavlink/include/mavlink/v2.0` e ricompilare il progetto.

- Nel caso di ArduPilot, l'aggiunta di altri file da aggiungere al progetto di building è più complicata. In particolare, non è possibile informare `waf` che si intende includere un altro file `.c` o `.cpp` nella compilazione. Dunque, nel caso di uso di funzioni implementate in tale file (e richiamate mediante l'header file a lui associato), il linker restituirà lo **status code 1** e genererà un messaggio di **undefined reference**. La soluzione⁸ consiste nel rendere la libreria *header-only* e spostare semplicemente l'implementazione concreta delle funzioni nell'header file. A questo punto, sia la compilazione che il linking vanno a buon fine.

Una volta chiarito questo aspetto è possibile procedere nell'*overview*.

Il *core* delle modifiche alla libreria MAVLINK risiede nel file `mavlink_helpers.h`, dove sono presenti le funzioni che regolano a basso livello la **serializzazione** e la **deserializzazione** dei messaggi MAVLINK.

Queste ultime sono state **modificate** in modo tale da includere chiamate alle funzionalità di **cifratura** e **decifratura** offerte da `aes.h`. In particolare:

⁸il *workaround*, più che altro

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

1. MAVLINK_HELPER `uint16_t` `mavlink_finalize_message_buffer()`, che calcola il checksum del messaggio da inviare. Utilizzata da QGroundControl;
2. MAVLINK_HELPER `void` `_mav_finalize_message_chan_send()`, che effettua le medesime operazioni ma viene utilizzata da ArduPilot poiché tale progetto definisce la macro `MAVLINK_USE_CONVENIENCE_FUNCTIONS`;
3. MAVLINK_HELPER `uint8_t` `mavlink_frame_char_buffer()`, che si occupa di effettuare il parsing di un messaggio in arrivo e ne verifica il checksum. Viene usata da entrambi i progetti.

Nel caso delle funzioni **1** e **2**, è necessario effettuare l'operazione di **cifratura** mediante l'invocazione della funzione `void AES_CTR_xcrypt_buffer()`. Questa operazione, per mantenere intatto il funzionamento del meccanismo di controllo integrità di MAVLINK, viene effettuata **prima del calcolo del checksum**.

Nella funzione **3**, è invece necessario effettuare l'operazione di **decifratura**. Specularmente, tale operazione viene effettuata **dopo la verifica del checksum**. Per decifrare, è stata invocata la **stessa funzione** delle funzioni **1** e **2** poiché nel contesto di AES in CTR-mode, **cifratura e decifratura condividono la stessa procedura**.

Sono state inoltre previste tre funzioni *utility*:

1. `void set_session_key()`, richiamata sia da ArduPilot che da QGroundControl e che permette di impostare lo **shared secret** estratto dalla capsula ed il **vettore di inizializzazione**;
2. `uint8_t *get_session_key()`, che restituisce un **puntatore** allo **shared secret** una volta impostato;
3. `uint8_t *get_iv()`, che restituisce un **puntatore** al **vettore di inizializzazione** una volta impostato;

7 IMPLEMENTAZIONE DELLA SOLUZIONE PROPOSTA

Le operazioni di **cifratura** e **decifratura** non sono effettuate nel caso di invio e ricezione dei messaggi di HEARTBEAT o di scambio della chiave o della capsula (e relativi messaggi di acknowledgement). Questo controllo viene effettuato facendo riferimento all'**ID** del messaggio.

8 Analisi delle performance

A valle della fase di implementazione di questo lavoro, è stato effettuato un breve confronto in termini di **utilizzo di CPU** e di **utilizzo di memoria** tra la versione *vanilla* di ArduPilot e quella sviluppata e descritta nelle sezioni precedenti.

In particolare, il test è stato effettuato mediante il tool `htop`, tipico dei sistemi GNU/Linux e che permette, per ogni processo, di identificare il suo PID, il suo utilizzo di memoria ed il suo impatto sul carico di CPU del sistema.

Tale tool è installabile su virtualmente ogni distribuzione GNU/Linux, quindi (naturalmente) anche su Ubuntu: il sistema operativo su cui è stato stabilito l'ambiente operativo per questo lavoro che, riassumendo, possiede queste specifiche:

- Processore **Intel Core i5-8265U**, con processo produttivo a 14nm ed una frequenza massima di 3,90GHz quando la tecnologia **TurboBoost** è attivata;
- **8GB** di RAM DDR4;
- Ubuntu **22.04 LTS** Jammy Jellyfish;
- Kernel **6.2.0-33-generic**, compilato in data 7 Settembre 2023;
- GNU `make` 4.3;
- GNU `gcc` 11.4

Il test è stato condotto mettendo in esecuzione il tool `htop` una volta fatto partire **SITL**, che si mette in attesa di connessioni in ingresso. Il processo di riferimento è unico e si chiama `arducopter` ed è isolabile mediante la funzionalità di ricerca del tool (invocabile tramite `F3` e digitando la stringa `arducopter`).

Il processo di scambio della chiave e della capsula inizia subito dopo l'avvenuta connessione tra UAV e GCS, quindi è immediatamente possibile valutare l'impatto dell'introduzione delle modifiche in tale lavoro

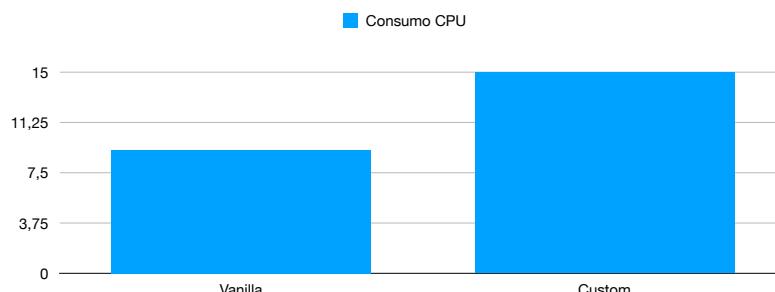
Per quanto riguarda il **carico di CPU**:

- Il progetto *vanilla* (quindi ArduPilot senza modifiche usato con l'AppImage di QGroundControl) vede un utilizzo di CPU pari al **9,2%**, costante durante lo scambio dei messaggi MAVLINK;
- Il progetto **modificato** (con le versioni *custom* di ArduPilot e QGroundControl) vede invece un consumo medio di CPU del **12%**, con un incremento al **15%** nel momento in cui viene effettuato `Encaps()`.

Per quanto riguarda invece il **consumo di memoria**:

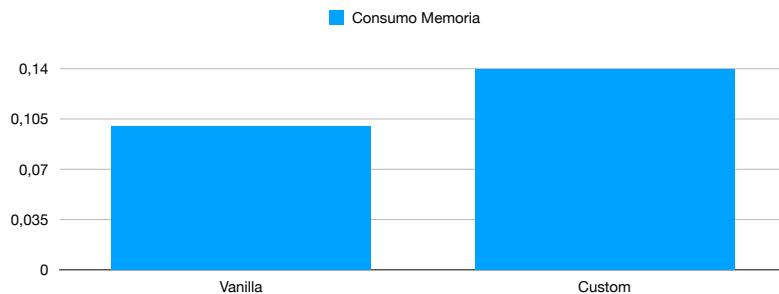
- Il progetto *vanilla* (quindi ArduPilot senza modifiche usato con l'AppImage di QGroundControl) vede un utilizzo di memoria pari al **0,10%**, pari a circa **7,8MB** sul totale della memoria disponibile nell'ambiente di lavoro;
- Il progetto **modificato** (con le versioni *custom* di ArduPilot e QGroundControl) vede invece un consumo medio di memoria dello **0,14%** pari a circa **10,9MB** sul totale della memoria disponibile nell'ambiente di lavoro.

Di seguito è presente un grafico che visualizza graficamente **l'utilizzo della CPU** da parte di ArduPilot nei due casi appena considerati.



8 ANALISI DELLE PERFORMANCE

Il grafico seguente, invece, rappresenta la percentuale di **occupazione di memoria** da parte di ArduPilot nei due casi considerati.



Nella prossima sezione si tireranno, infine, le somme sul lavoro svolto alla luce dei risultati ottenuti ed appena presentati.

9 Conclusioni e sviluppi futuri

L'introduzione dell'algoritmo **Kyber** e, conseguentemente, del cifrario **AES** risultano dunque **rilevanti** in termini di consumo di memoria e di carico di CPU ma **non** in maniera tale da **prgiudicare** l'operatività del protocollo MAVLINK.

Viene inoltre dimostrata la **duttilità** e l'**estendibilità** dei progetti **ArduPilot** e **QGroundControl**, proprio grazie alla loro natura **open-source** che quindi permette a tutti di contribuire al loro sviluppo e sfruttare al meglio gli sforzi di una community attiva e sempre aperta al confronto, come dimostrato dalla sempre vivace interazione tra i membri del server Discord **DroneCode**, che ospita sia discussioni tecniche che di carattere generale.

Per concludere questa trattazione, i principali possibili **sviluppi futuri** di questo lavoro di Tesi si articolano nello specifico in **due filoni**:

- **Generalizzare** la soluzione proposta per altri tipi di piattaforma oltre ArduPilot (una su tutte la piattaforma **PX4** anch'essa **open source**);
- **Completare** la migrazione ad uno stack crittografico post-quantum introducendo lo schema di firma **DILITHIUM**: lo standard del NIST per gli algoritmi di firma post-quantistici. Tale soluzione, tuttavia, risulta complicata a causa della dimensione massima di un pacchetto MAVLINK, discussa nelle sezioni precedenti.

Riferimenti bibliografici e risorse consultate

- [1] Imperial Wars Museum. *A brief history of drones*. <https://www.iwm.org.uk/history/a-brief-history-of-drones>. URL consultato il 5 settembre 2023.
- [2] Antonio Bono et al. “Path Planning and Control of a UAV Fleet in Bridge Management Systems”. In: *Remote Sensing* 14.8 (2022). ISSN: 2072-4292. DOI: 10.3390/rs14081858. URL: <https://www.mdpi.com/2072-4292/14/8/1858>.
- [3] *Sito web di ArduPilot*. <https://ardupilot.org>. URL consultato il 6 settembre 2023.
- [4] *Sito web di DIYDrones*. <https://diydrones.com>. URL consultato il 6 settembre 2023.
- [5] *Documentazione di ArduPilot - sezione "History"*. <https://ardupilot.org/dev/docs/common-history-of-ardupilot.html>. URL consultato il 6 settembre 2023.
- [6] *Documentazione di ArduCopter*. <https://ardupilot.org/copter/index.html>. URL consultato il 6 settembre 2023.
- [7] *Documentazione di ArduPlane*. <https://ardupilot.org/plane/index.html>. URL consultato il 6 settembre 2023.
- [8] *Documentazione di Rover*. <https://ardupilot.org/rover/index.html>. URL consultato il 6 settembre 2023.
- [9] *Documentazione di ArduSub*. <http://www.ardusub.com>. URL consultato il 6 settembre 2023.
- [10] *Documentazione di AntennaTracker*. Link. URL consultato il 6 settembre 2023.
- [11] *Documentazione di Blimp*. <https://ardupilot.org/blimp/index.html>. URL consultato il 6 settembre 2023.

RIFERIMENTI BIBLIOGRAFICI E RISORSE CONSULTATE

- [12] *Introduzione a SITL.* <https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html>. URL consultato il 7 settembre 2023.
- [13] *Introduzione a WAF.* https://waf.io/book/#_the_waf_framework. URL consultato il 7 settembre 2023.
- [14] *Repository principale di ArduPilot.* <https://github.com/ArduPilot/ardupilot/tree/master>. URL consultato il 7 settembre 2023.
- [15] *Repository principale di ArduPilot - istruzioni per la compilazione.* <https://github.com/ArduPilot/ardupilot/blob/master/BUILD.md>. URL consultato il 7 settembre 2023.
- [16] *Informazioni su mavgen.* https://mavlink.io/en/getting_started/generate_libraries.html. URL consultato l'8 settembre 2023.
- [17] *Documentazione di MAVProxy - pagina iniziale.* <https://ardupilot.org/mavproxy/index.html>. URL consultato l'8 settembre 2023.
- [18] *Documentazione di Mission Planner - pagina iniziale.* <https://ardupilot.org/planner/index.html>. URL consultato l'8 settembre 2023.
- [19] *Sito web di QGroundControl.* <http://qgroundcontrol.com>. URL consultato l'8 settembre 2023.
- [20] *Home page della fondazione DroneCode.* <https://dronecode.org>. URL consultato l'8 settembre 2023.
- [21] *Codice sorgente di QGroundControl.* <https://github.com/mavlink/qgroundcontrol>. URL consultato l'8 settembre 2023.
- [22] *Codice sorgente di QGroundControl - App Android.* <https://github.com/mavlink/qgroundcontrol/tree/master/android>. URL consultato l'8 settembre 2023.
- [23] *Implementazione di riferimento in linguaggio C della libreria MAVLink.* https://github.com/mavlink/c_library_v2. URL consultato l'8 settembre 2023.

- [24] *QGCExternalLibs.pri: switch to all mavlink dialect by default.* Link al commit relativo al cambiamento. URL consultato l'8 settembre 2023.
- [25] *aqtinstall.* <https://github.com/miurahr/aqtinstall>. URL consultato l'8 settembre 2023.
- [26] *QGroundContol: Getting Started with source & builds.* https://dev.qgroundcontrol.com/master/en/getting_started/index.html. URL consultato l'8 settembre 2023.
- [27] *Initial commit.* Link al primo commit del progetto MAVLink. URL consultato il 13 settembre 2023.
- [28] *Definizione dei messaggi MAVLink per le varie piattaforme.* https://github.com/mavlink/mavlink/tree/master/message_definitions/v1.0. URL consultato il 13 settembre 2023.
- [29] *Overview - MAVLink Developer Guide.* <https://mavlink.io/en/about/overview.html>. URL consultato il 13 settembre 2023.
- [30] *MAVLink Versions - MAVLink Developer Guide.* https://mavlink.io/en/guide/mavlink_version.html. URL consultato il 13 settembre 2023.
- [31] *MAVLink 2 - MAVLink Developer Guide.* https://mavlink.io/en/guide/mavlink_2.html. URL consultato il 13 settembre 2023.
- [32] *Struttura di un pacchetto MAVLink 1.0.* https://mavlink.io/en/guide/serialization.html#v1_packet_format. URL consultato il 13 settembre 2023.
- [33] *Struttura di un pacchetto MAVLink 2.0.* https://mavlink.io/en/guide/serialization.html#v2_packet_format. URL consultato il 13 settembre 2023.
- [34] *Struttura dell'header della firma in MAVLink 2.0.* https://mavlink.io/en/guide/message_signing.html#frame-format. URL consultato il 13 settembre 2023.

RIFERIMENTI BIBLIOGRAFICI E RISORSE CONSULTATE

- [35] *Algoritmo per la creazione della firma in MAVLink 2.0.* https://mavlink.io/en/guide/message_signing.html#signature. URL consultato il 13 settembre 2023.
- [36] *Handshake per la definizione del protocollo.* https://mavlink.io/en/guide/mavlink_version.html#version_handshaking. URL consultato il 13 settembre 2023.
- [37] Young-Min Kwon et al. “Empirical Analysis of MAVLink Protocol Vulnerability for Attacking Unmanned Aerial Vehicles”. In: *IEEE Access* 6 (2018), pp. 43203–43212. DOI: 10.1109/ACCESS.2018.2863237.
- [38] *MAVLink FAQ - How secure is MAVLink?* <https://mavlink.io/en/about/faq.html>. URL consultato il 14 settembre 2023.
- [39] Azza Allouch et al. “MAVSec: Securing the MAVLink Protocol for ArduPilot/PX4 Unmanned Aerial Systems”. In: *CoRR* abs/1905.00265 (2019). arXiv: 1905.00265. URL: <http://arxiv.org/abs/1905.00265>.
- [40] Daniel Bernstein. “ChaCha, a variant of Salsa20”. In: <https://cr.yp.to/papers.html#chacha> (Jan. 2008).
- [41] Noshin Sabuwala and Rohin D Daruwala. “Securing Unmanned Aerial Vehicles by Encrypting MAVLink Protocol”. In: *2022 IEEE Bombay Section Signature Conference (IBSSC)*. 2022, pp. 1–6. DOI: 10.1109/IBSSC56953.2022.10037546.
- [42] Sattar B. Sadkhan and Akbal O. Salman. “A survey on lightweight-cryptography status and future challenges”. In: *2018 International Conference on Advance of Sustainable Engineering and its Application (ICASEA)*. 2018, pp. 105–108. DOI: 10.1109/ICASEA.2018.8370965.
- [43] *Announcing Request for Nominations for Lightweight Cryptographic Algorithms.* <https://csrc.nist.gov/News/2018/requesting-nominations-for-lightweight-crypto-algs>. URL consultato il 15 settembre 2023.

- [44] *Lightweight Cryptography Standardization: Finalists Announced.* <https://csrc.nist.gov/News/2021/lightweight-crypto-finalists-announced>. URL consultato il 15 settembre 2023.
- [45] *Lightweight Cryptography Standardization Process: NIST Selects Ascon.* <https://csrc.nist.gov/news/2023/lightweight-cryptography-nist-selects-ascon>. URL consultato il 15 settembre 2023.
- [46] *A custom MAVLink with lightweight crypto.* <https://github.com/angelopassaro/SEC-UAV/tree/master>. URL consultato il 15 settembre 2023.
- [47] *Codice sorgente di MAVSec.* <https://github.com/aniskoubaa/mavsec>. URL consultato il 15 settembre 2023.
- [48] Rafael Álvarez, Juan Santonja, and Antonio Zamora. “Algorithms for Lightweight Key Exchange”. In: *Ubiquitous Computing and Ambient Intelligence*. Ed. by Carmelo R. García et al. Cham: Springer International Publishing, 2016, pp. 536–543. ISBN: 978-3-319-48799-1.
- [49] Paul Benioff. “The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines”. In: *Journal of Statistical Physics* 22.5 (May 1980), pp. 563–591. ISSN: 1572-9613. DOI: 10.1007/BF01011339. URL: <https://doi.org/10.1007/BF01011339>.
- [50] Benjamin Schumacher. “Quantum coding”. In: *Phys. Rev. A* 51 (4 Apr. 1995), pp. 2738–2747. DOI: 10.1103/PhysRevA.51.2738. URL: <https://link.aps.org/doi/10.1103/PhysRevA.51.2738>.
- [51] P.W. Shor. “Algorithms for quantum computation: discrete logarithms and factoring”. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

- [52] David Harvey and Joris van Der Hoeven. “Integer multiplication in time $O(n \log n)$ ”. In: *Annals of Mathematics* (Mar. 2021). DOI: 10.4007/annals.2021.193.2.4. URL: <https://hal.science/hal-02070778>.
 - [53] Daniel J. Bernstein. “Introduction to post-quantum cryptography”. In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–14. ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_1. URL: https://doi.org/10.1007/978-3-540-88702-7_1.
 - [54] *Announcing Request for Nominations for Public-Key Post-Quantum Cryptographic Algorithms*. <https://csrc.nist.gov/news/2016/public-key-post-quantum-cryptographic-algorithms>. URL consultato il 19 settembre 2023.
 - [55] *CRYSTALS - Cryptographic Suite for Algebraic Lattices*. <https://pq-crystals.org/index.shtml>. URL consultato il 19 settembre 2023.
 - [56] *Kyber*. <https://pq-crystals.org/kyber/index.shtml>. URL consultato il 19 settembre 2023.
 - [57] Joppe Bos et al. *CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM*. Cryptology ePrint Archive, Paper 2017/634. <https://eprint.iacr.org/2017/634>. 2017. DOI: 10.1109/EuroSP.2018.00032. URL: <https://eprint.iacr.org/2017/634>.
 - [58] Oded Regev. “On Lattices, Learning with Errors, Random Linear Codes, and Cryptography”. In: *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*. STOC ’05. Baltimore, MD, USA: Association for Computing Machinery, 2005, pp. 84–93. ISBN: 1581139608. DOI: 10.1145/1060590.1060603. URL: <https://doi.org/10.1145/1060590.1060603>.
 - [59] *La matematica dietro la PQC: Learning With Errors*. <https://www.telsy.com/it/la-matematica-dietro-la-pqc-learning-with-errors-lwe/>. URL consultato il 21 settembre 2023.
-

- [60] Adeline Langlois and Damien Stehle. *Worst-Case to Average-Case Reductions for Module Lattices*. Cryptology ePrint Archive, Paper 2012/090. <https://eprint.iacr.org/2012/090>. 2012. URL: <https://eprint.iacr.org/2012/090>.
 - [61] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. *On Ideal Lattices and Learning with Errors Over Rings*. Cryptology ePrint Archive, Paper 2012/230. <https://eprint.iacr.org/2012/230>. 2012. URL: <https://eprint.iacr.org/2012/230>.
 - [62] *Kyber - How does it work? / Approachable Cryptography*. <https://cryptopedia.dev/posts/kyber/>. URL consultato il 22 settembre 2023.
 - [63] Lov K. Grover. *A fast quantum mechanical algorithm for database search*. 1996. arXiv: quant-ph/9605043 [quant-ph].
 - [64] Scott Fluhrer. *Reassessing Grover's Algorithm*. Cryptology ePrint Archive, Paper 2017/811. <https://eprint.iacr.org/2017/811>. 2017. URL: <https://eprint.iacr.org/2017/811>.
 - [65] *Kyber - Software*. <https://pq-crystals.org/kyber/software.shtml>. URL consultato il 26 settembre 2023.
 - [66] *pq-crystals/kyber*. <https://github.com/pq-crystals/kyber>. URL consultato il 26 settembre 2023.
 - [67] *Intel® Advanced Vector Extensions 2 (Intel® AVX2) - 010 - ID:655258 | 12th Generation Intel® Core™ Processors*. Informazioni su AVX2. URL consultato il 26 settembre 2023.
 - [68] *Home / Open Quantum Safe*. <https://openquantumsafe.org>. URL consultato il 26 settembre 2023.
 - [69] *Algorithms / Open Quantum Safe*. <https://openquantumsafe.org/liboqs/algorithms/>. URL consultato il 26 settembre 2023.
 - [70] *open-quantum-safe/liboqs: C library for prototyping and experimenting with quantum-resistant cryptography*. <https://github.com/open-quantum-safe/liboqs>. URL consultato il 26 settembre 2023.
-

RIFERIMENTI BIBLIOGRAFICI E RISORSE CONSULTATE

- [71] *kem.h / Open Quantum Safe*. <https://openquantumsafe.org/liboqs/api/kem>. URL consultato il 28 settembre 2023.
- [72] *kokke/tiny-AES-c: Small portable AES128/192/256 in C*. <https://github.com/kokke/tiny-AES-c>. URL consultato il 4 ottobre 2023.