UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Analisi e implementazione di sistemi di controllo sicuri per azionamenti elettrici

---

Analysis and implementation of safe control systems for electric drives

Relatore: Chiar.mo
Prof. Gianluca Ippoliti

Tesi di Laurea di:
Cristian Gonzato

Correlatori:
Prof. Giuseppe Orlando
Prof. Marco Baldi

Anno Accademico 2025/2026

Indice

1 Introduzione **3**
- 1.1 Contesto del progetto . . . 3
- 1.2 Obiettivi della tesi e perimetro di lavoro . . . 3

2 Architettura Hardware **4**
- 2.1 Componenti di sistema . . . 4
  - 2.1.1 Raspberry Pi 5 . . . 4
  - 2.1.2 Convertitore High-Precision AD/DA . . . 5
  - 2.1.3 Motore DC con Dinamo Tachimetrica . . . 6
- 2.2 Infrastruttura di rete . . . 7
  - 2.2.1 Cablaggio e configurazione . . . 7

3 Architettura Software **11**
- 3.1 Set-up SO a 64 bit . . . 11
- 3.2 Download della libreria LGPIO . . . 13
  - 3.2.1 Differenze con BCM2835 . . . 14
  - 3.2.2 Vantaggi . . . 15

4 Cifratura del sistema **16**
- 4.1 Fully Homomorphic Encryption (FHE) . . . 16
  - 4.1.1 Schema di funzionamento . . . 17
  - 4.1.2 Vantaggi e limiti . . . 19
- 4.2 Lo schema CKKS . . . 20
  - 4.2.1 Aritmetica approssimata . . . 20
  - 4.2.2 Tecnica del rescaling . . . 20
  - 4.2.3 Gestione del rumore . . . 21
- 4.3 Confronto con crittografia RSA . . . 22

5 Implementazione del Controllo Omomorfico **24**
- 5.1 Configurazione della libreria OpenFHE . . . 24
- 5.2 Caricamento contesto e chiavi . . . 26
  - 5.2.1 Master . . . 28
  - 5.2.2 Slave . . . 30
- 5.3 Acquisizione dell'errore, cifratura e attuazione . . . 31
- 5.4 Esecuzione del controllore PI sui dati cifrati . . . 35
- 5.5 Codice del sistema di comunicazione . . . 36
- 5.6 Codice completo del sistema di controllo sicuro . . . 39

5.6.1 Nodi del sistema 39
5.6.2 Sistema di comunicazione 52
5.6.3 Sezione di configurazione 54
5.7 Algoritmo RSA: confronto con FHE 55
5.7.1 Cifratura degli stati sullo Slave 55
5.7.2 Nessuna addizione omomorfica svolta sul Master 56
5.7.3 Il dato torna in chiaro sullo Slave 56

6 Conclusioni **58**
6.1 Risultati Sperimentali 58
6.1.1 Latenza del ciclo di controllo omomorfico 58
6.1.2 Ottimizzazione di rete e confronto in chiaro 60
6.2 Possibili sviluppi futuri 62

Bibliografia **64**

Ringraziamenti **65**

Capitolo 1

Introduzione

1.1 Contesto del progetto

Il progetto in esame riguarda l'analisi, l'implementazione e lo sviluppo di un sistema di comunicazione e di controllo gestito totalmente in modalità cifrata. Tramite la FHE, che esegue operazioni nel dominio cifrato, il sistema è in grado di ricevere input in chiaro dall'esterno e gestirli internamente cifrandoli. Il flusso di dati verrà reso in chiaro solo ai fini del DAC che provvederà all'attuazione del processo fisico sul motore. Le schede Raspberry rappresentano i nodi principali del sistema, grazie ai quali i dati vengono elaborati opportunamente e successivamente trasmessi in rete locale.

1.2 Obiettivi della tesi e perimetro di lavoro

Questo elaborato si prefigge l'obiettivo di studiare alcuni aspetti del sistema di controllo, a partire dall'hardware puro fino al software di alto livello.
Dopo un'attenta analisi del livello hardware, ci si sposterà sull'infrastruttura di rete e sul software interno agli stessi. Successivamente, si analizzerà l'architettura distribuita e come questa venga gestita dal sistema. Infine, si andrà nel dettaglio del protocollo di cifratura scelto, discutendone significato e implementazione software.

Capitolo 2

Architettura Hardware

In questo capitolo si porrà l'attenzione sull'architettura hardware del sistema, in particolare sugli elementi che ne permettono il funzionamento e il mantenimento dello stesso nel tempo.

2.1 Componenti di sistema

In dettaglio, i componenti che seguono sono costantemente connessi (si veda il Paragrafo 2.2) e collaborano attivamente per garantire una corretta gestione del segnale elettrico di controllo e, quindi, che l'attuazione del processo fisico avvenga conformemente ai limiti fisici del motore.

2.1.1 Raspberry Pi 5

I nodi principali del sistema sono rappresentati dalle schede Raspberry Pi 5, ultimo modello reso disponibile dalla casa madre.

Modello caratterizzato da numerose specifiche tecniche di rilevanza notevole ai fini della progettazione del sistema

- CPU Broadcom BCM2712 quad-core a 64 bit da 2.4 GHz.
- LPDDR4X-4267 SDRAM (4 GB).
- Wi-Fi dual-band 802.11ac.
- Slot per scheda microSD.
- 2 × Porte USB 3.0.
- Gigabit Ethernet.
- Alimentazione CC 5 V / 5 A tramite USB-C.
- Collettore standard a 40 pin.

In Fig. 2.1 è presente la scheda completa di componentistica

Figura 2.1: Raspberry Pi 5. *Fonte: elaborazione personale*

Rispetto alle precedenti Pi 3, l'evoluzione in termini di memoria RAM e CPU ha permesso di ottenere un'esperienza di utilizzo nettamente più fluida.

2.1.2 Convertitore High-Precision AD/DA

Affinché il processo fisico venga attuato sul motore, è necessario installare manualmente nel collettore della scheda Slave un convertitore AD/DA. In Fig. 2.2 si osserva come questa particolare scheda sia stata usata con una configurazione a jumper.

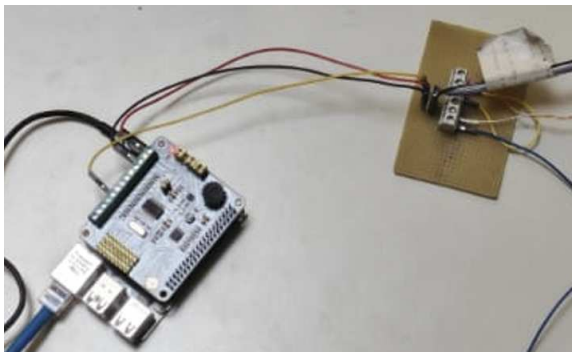

Figura 2.2: Convertitore AD/DA. *Fonte: elaborazione personale*

Poiché inizialmente la scheda era montata sulla precedente Pi 3, con l'aiuto di un cacciavite a stella, si è proceduto alla rimozione dei supporti che ancoravano fisicamente il convertitore alla scheda e successivamente all'estrazione con cautela dei pin dal collettore. Successivamente, previa visione sulla documentazione ufficiale della piedinatura della Pi 5 per evitare cortocircuiti[1], si sono ricollegati i pin al collettore e assicurati i supporti sulla Pi 5.

Mantenendo il connettore Type-C disconnesso, quindi la scheda sprovvista di alimentazione, dai colleghi precedenti è stata svolta la configurazione impiegando tre jumper.
In particolare, i jumper rosso e nero fanno riferimento rispettivamente all'alimentazione e alla messa a terra (GND), mentre i terminali del jumper giallo sono connessi

rispettivamente all'uscita digitale A01 della scheda e al pin di riferimento sull'amplificatore a Mosfet (anche quest'ultimo realizzato precedentemente).

Completata la configurazione, è possibile alimentare la scheda e assicurarsi che il led rosso superiore sia acceso in posizione fissa.
A questo punto, è di fondamentale importanza verificare sperimentalmente i valori di tensione previsti dalla documentazione della scheda e i valori necessari al sistema. Il convertitore, per costruzione, viene alimentato a 5 V. L'uscita digitale, invece, oscilla tra 0 V e nell'intorno di 3.3 V (valore di saturazione del motore).

Utilizzando un multimetro come strumento di misura, si appongono opportunamente i terminali positivo sull'ingresso e sull'uscita da controllare e negativo sulla GND.
Il valore di tensione che alimenta la scheda è così verificabile istantaneamente. L'uscita digitale, come si vedrà nel Capitolo 5, è responsabile del corretto inoltro del valore di tensione del segnale di controllo al motore, per cui è misurabile esclusivamente inizializzando il processo e verificando sia nel transitorio che a regime il dato a schermo.

2.1.3 Motore DC con Dinamo Tachimetrica

A valle del processo di controllo si trova il motore, in particolare il modello Selema Serie 30 controllato in CC (Fig. 2.3). Come si osserva in figura, è alimentato da un alimentatore da banco ed è incorporata internamente anche una Dinamo tachimetrica

Figura 2.3: Motore Selema *Fonte: elaborazione personale*

Nella tabella seguente si riportano i parametri del motore e della dinamo

Descrizione	**Simbolo**	**Valore**
Resistenza di armatura	R_a	2.44 Ω
Induttanza di armatura	L_a	2.6 mH
Costante di coppia	K_T	0.063 Nm/A
Costante di f.c.e.m.	K_e	0.063 Vs/rad
Inerzia del rotore	J	$13 \cdot 10^{-6}$ Kgm^2
Attrito viscoso	B	$10 \cdot 10^{-5}$ Nms
Guadagno dinamo tachimetrica	K_{dt}	0.034 V/rpm

Tabella 2.1: Parametri del motore Selema e della dinamo tachimetrica

2.2 Infrastruttura di rete

I singoli componenti hardware, per potersi definire "sistema", necessitano di una comunicazione costante. Di conseguenza, viene realizzata un'infrastruttura basata su trasmissione wired.
All'interno della stessa, impiegando il protocollo TCP, i dati fluiscono bidirezionalmente da e verso le schede e il motore.
Lo scambio dati risultante è di tipo Full - duplex.

2.2.1 Cablaggio e configurazione

Come si osserva in Fig. 2.4, la totalità dei nodi architetturali comunica attraverso cavi Ethernet CAT 5, i quali basano il loro funzionamento sul protocollo 802.3 (Ethernet). La connessione fisica, resa possibile da connettori RJ45, è effettuata sulle porte Gigabit Ethernet delle schede Raspberry.
Si tratta di cavi progettati per essere immuni al rumore esterno, grazie alla tecnica del twisting.
Inoltre, a differenza della comunicazione nell'etere attraverso il protocollo 802.11 (WI-FI), la trasmissione cablata garantisce, ai fini del nostro sistema di controllo, risultati nettamente superiori grazie all'abbattimento del rumore che altrimenti si avrebbe con una comunicazione di tipo wireless.

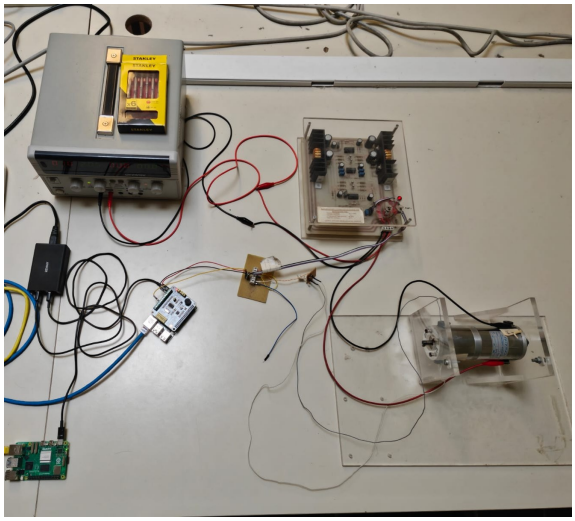

Figura 2.4: Infrastruttura di rete *Fonte: elaborazione personale*

I dati vengono trasmessi localmente dimensionando una rete locale. In particolare, con l'impiego dello switch Hp ProCurve in Fig. 2.5, i dati vengono instradati verso le schede Master e Slave collegando i rispettivi cavi e connettori agli ingressi dello stesso.
Gli utenti, così come fatto da noi, possono interfacciarsi al sistema collegando uno o più cavi Ethernet agli ingressi dello switch. Il numero di cavi è direttamente proporzionale al numero di utenti che sono intenzionati a comunicare.

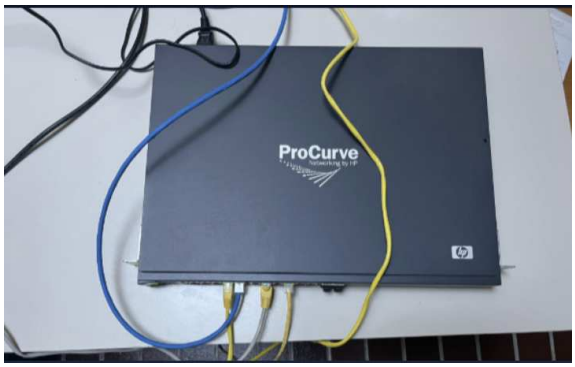

Figura 2.5: Hp ProCurve

Per ogni macchina, trattandosi di una rete locale, è necessario definire un indirizzo IP statico della stessa classe dell'indirizzo di rete (`10.10.1.X`). `X` può assumere qualsiasi valore intero non negativo, prestando attenzione a non utilizzare indirizzi già occupati, quindi non disponibili.

Sarà necessario recarsi nelle impostazioni del proprio Laptop provvisto di un qualsiasi sistema operativo disponibile in commercio e visualizzare il menù a sinistra, come in Fig. 2.6

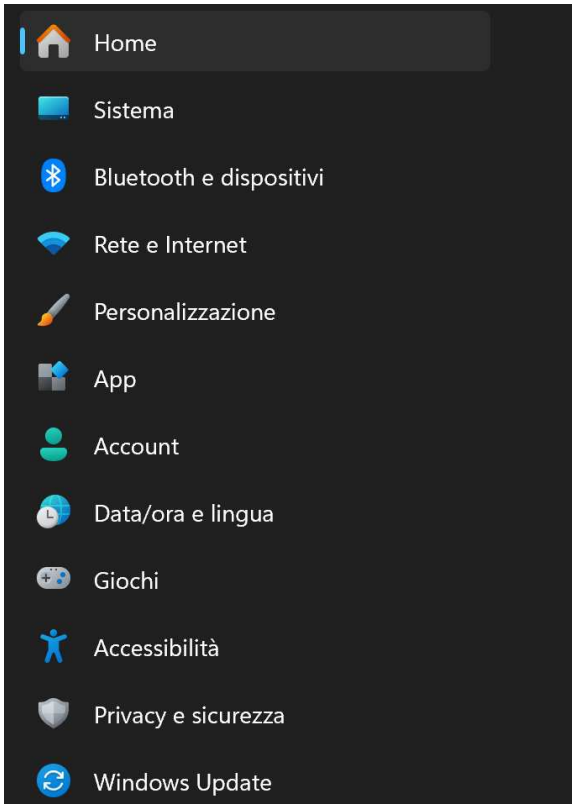

Figura 2.6: Menù impostazioni *Fonte: Sistema operativo Windows*

Successivamente, seguendo la voce Rete e Internet come in Fig. 2.7, espandere il menù della voce relativa alla connessione Ethernet.

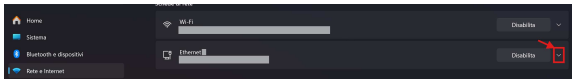

Figura 2.7: Voce "Rete e Internet" *Fonte: Sistema operativo Windows*

A questo punto, selezionare "Visualizza proprietà aggiuntive" e modificare l'indirizzo IP del Laptop nel dominio della rete locale configurata precedentemente, come in Fig. 2.8

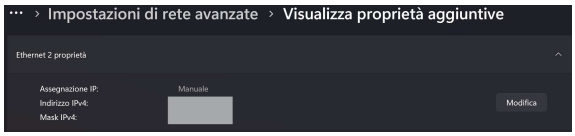

Figura 2.8: Settaggio indirizzo IP statico *Fonte: Sistema operativo Windows*

In prima configurazione, le schede sono state configurate staticamente, accedendovi con il modulo WI-FI installato sulle stesse.

Attraverso la tecnologia del tethering, le schede sono state connesse a un dispositivo mobile personale. Così facendo, siamo stati in grado di individuare l'indirizzo

di fabbrica delle schede e, accedendo all'interno del sistema, settarlo in modalità statica nel rispetto della rete locale.
Arrivati a questo punto, è possibile lanciare i seguenti comandi da terminale Linux per interfacciarsi alle schede, quindi accedere alla memoria interna.

- **Master** → `ssh pi@10.10.1.6`.
- **Slave** → `ssh raspy@10.10.1.20`.

La struttura delle istruzioni risulta essere: `ssh nome@indirizzo`.
I nominativi delle schede sono stati scelti in maniera del tutto arbitraria dalle impostazioni delle stesse.
Infine, tutta la parte hardware contribuisce al processo fisico grazie al collegamento con il partitore di tensione e l'Amplificatore a Mosfet, i quali permettono al segnale elettrico di arrivare correttamente al motore per il pilotaggio.
Entrambi i circuiti elettrici sono stati realizzati da colleghi che precedentemente hanno lavorato a questo sistema.

Capitolo 3

Architettura Software

Conclusa l'analisi della parte hardware, ci si concentrerà ora sul core di funzionamento, senza il quale tutto il sistema permarrebbe in uno stato poco utile ai nostri scopi.

3.1 Set-up SO a 64 bit

Dopo aver acquistato le schede Pi 5 e configurate così come osservato nel capitolo precedente, la prima operazione da svolgere è installare opportunamente il Sistema Operativo (SO) per abilitare i connettori, le porte di comunicazione e, in generale, tutte le funzionalità.

Fra tutte le specifiche tecniche discusse in precedenza, tornerà utile lo slot microSD. Munendosi di una microSD da 8 GB ed eventualmente di un adattatore in lettura per Laptop, sarà possibile procedere con l'installazione.

Si può procedere seguendo due possibili percorsi:

- Inserimento manuale dei file necessari nella scheda microSD.
- Installazione pulita grazie al software Raspberry Pi Imager.

L'installazione manuale si presta a diversi rischi, come mancanza di file e problemi generali di inconsistenza.
Motivo per il quale è preferibile optare per l'installazione tramite Imager: effettuando il download del software messo a disposizione da Raspberry, la procedura appare pulita e intuitiva.

La prima cosa da fare è inserire la microSD nel Laptop e avviare l'eseguibile `rpi-imager.exe`. All'avvio, l'interfaccia utente ci illustrerà, schermata per schermata, la procedura da svolgere, come si nota in Fig. 3.1.

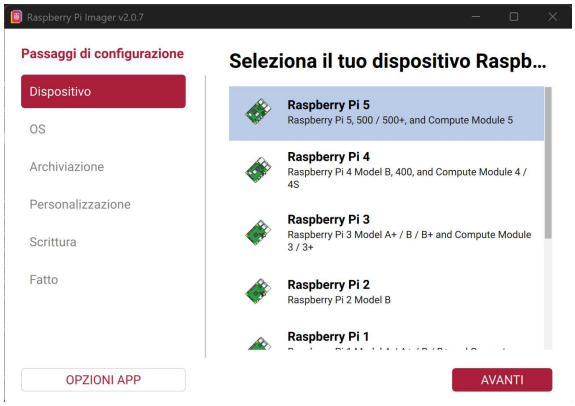

Figura 3.1: Interfaccia iniziale Imager *Fonte: elaborazione personale*

Come si nota dall'immagine di cui sopra, si seleziona il modello di scheda acquistato, nel caso in esame Raspberry Pi 5.
Procedendo oltre, si seleziona la tipologia di SO da installare. Per gli scopi del progetto, l'opzione più appropriata è il Pi OS a 64 bit, come in Fig. 3.2

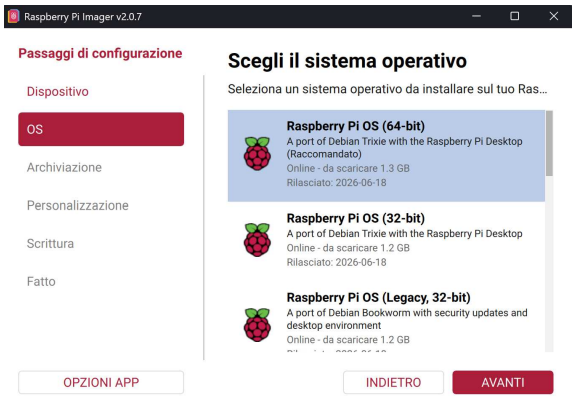

Figura 3.2: OS a 64 bit *Fonte: elaborazione personale*

Successivamente, si dovrà selezionare l'unità di archiviazione da formattare e sulla quale verranno installati tutti i file di configurazione: selezionare l'unità microSD precedentemente montata sul dispositivo.

Dopo aver personalizzato data e ora e indicato il fuso orario di riferimento, Imager procederà a scrivere il sistema operativo sulla scheda e, infine, svolgerà una verifica per la consistenza dei dati.

Ora è possibile rimuovere la microSD dal Laptop e spostarsi sulle schede Raspberry.

L'applicazione vera e propria dell'SO nelle scheda è resa possibile dallo slot per microSD presente a bordo.
Poiché le schede sono due, la procedura andrà ripetuta su di un'altra microSD da inserire nella seconda scheda.

Una volta inserita correttamente la microSD nello slot, si dovrà attendere che il led verde lampeggiante superiore si stabilzzi.
A questo punto, le schede sono pronte per la comunicazione e si potrà finalmente seguire la procedura per il settaggio dell'indirizzo statico discussa nel Paragrafo 2.2.1 del capitolo precedente.

È necessario in questa fase, poiché le schede Raspberry sono basate su kernel Linux, averlo installato sulla propria macchina.
Si può optare per il dual boot oppure, qualora non si voglia partizionare l'unità di storage disponibile, è possibile munirsi di una qualsiasi macchina virtuale (VM) come Oracle Virtual Box sulla quale installare Ubuntu.

Un'ulteriore alternativa, quella più semplice, è installare il terminale WSL basato su Linux, dal quale è possibile eseguire efficacemente tutti i comandi necessari. Dopo aver seguito la procedura d'installazione su macchina Windows messa a disposizione da Microsoft [2], il terminale si presenterà come in Fig. 3.3

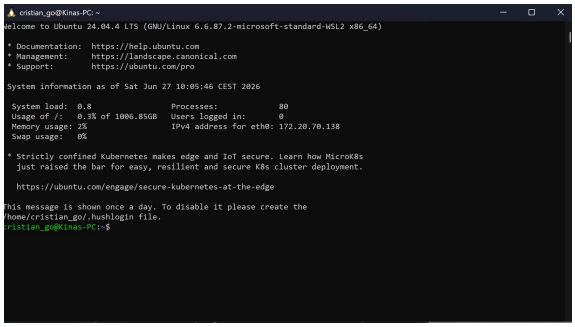

Figura 3.3: WSL amministratore *Fonte: elaborazione personale*

Da linea di comando, si accede al SO delle schede con le modalità discusse in precedenza e si procede con il download della libreria trattata nel Paragrafo 3.2.

3.2 Download della libreria LGPIO

Il collettore delle schede, per poter essere operativo, va abilitato. Nello specifico, è necessario agire via software effettuando il download e poi l'installazione della libreria ideata e scritta per questo scopo: LGPIO.

Libreria che rappresenta lo standard moderno per la gestione e il controllo dei pin GPIO, come in Fig. 3.4 [1].

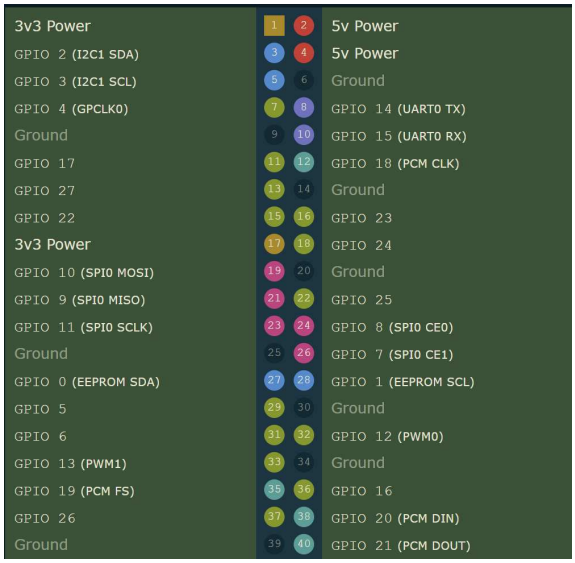

Figura 3.4: Piedinatura Pi 5 *Fonte: online*

La procedura d'installazione e applicazione della libreria sulle schede Raspberry è da effettuarsi seguendo scrupolosamente la documentazione ufficiale [3].

I comandi fondamentali per avviare l'installazione da terminale sono i seguenti

```
sudo apt update
sudo apt install python3-rpi-lgpio
```

I comandi vanno eseguiti in successione. In dettaglio, il primo comando si occupa di aggiornare l'elenco locale dei pacchetti disponibili, grazie al gestore dei pacchetti software di Raspberry Pi OS `apt`.
Non installa o aggiorna i programmi, ma scarica da internet una lista aggiornata per sapere quali software esistono e in quale versione. Il comando successivo, con `install`, dice al gestore di installare il pacchetto specifico per la libreria d'interesse

Inoltre, la libreria LGPIO include protocolli utili come I2C, comunicazione seriale e SPI.
Quest'ultimo è necessario ai fini della comunicazione progettuale, motivo per il quale è stato abilitato dalla console principale delle schede Raspberry, accessibile tramite il comando `sudo raspi-config` da terminale Linux.

3.2.1 Differenze con BCM2835

Si evidenziano in questo paragrafo le principali differenze rispetto alla libreria BCM2835, la quale non è più compatibile con le nuove Pi 5 a causa del cambiamento piuttosto notevole nell'architettura hardware delle schede.

In particolare, BCM2835 basa il suo funzionamento sull'accesso diretto in memoria, nello specifico ai registri del processore.

Così facendo, si riesce a bypassare il kernel, pilotando i pin GPIO alla massima velocità possibile.

Questo approccio non funziona più sulle moderne Pi 5, per due motivi fondamentali:

- Chip RP1 → la gestione dei pin è stata spostata su un chip proprietario che prende il nome di RP1. La libreria modificava indirizzi che ora non esistono più nell'alloggiamento principale [4]
- Sicurezza del Kernel → il SO delle Pi 5 rende più sicuro l'accesso in memoria, poiché lo blocca per motivi di stabilità.
  Ora il controllo hardware è affidato, attraverso il framework gpiod, al kernel Linux [5].

Nella tabella sottostante si riassumono le principali differenze e caratteristiche di incompatibilità.

Caratteristica	**bcm2835**	**lgpio**
Metodo di controllo	Accesso diretto alla memoria fisica (`/dev/mem`)	Interfaccia sicura tramite kernel Linux (chip RP1)
Compatibilità Hardware	Pi 1, 2, 3, 4 (vecchio SoC Broadcom)	Piena compatibilità con Pi 5
Velocità e Latenza	Estrema (nessun overhead)	Minore (richiede chiamate di sistema) ma più sicura
Privilegi richiesti	Esecuzione come root (`sudo`) strettamente necessaria	Esecuzione sicura in *user space*

Tabella 3.1: Confronto tra le librerie BCM2835 e LGPIO

3.2.2 Vantaggi

I principali vantaggi sono già stati citati pocanzi, tuttavia si noti l'importanza di non dover eseguire più, tranne quando strettamente necessario, i comandi in super user con `sudo`.
Questa caratteristica rende più leggera e meno prolissa l'esperienza d'utilizzo con la libreria

Capitolo 4

Cifratura del sistema

Dopo aver costruito il sistema lato hardware e averlo reso completamente funzionante lato software, si osservi che, allo stato attuale, quest'ultimo è esposto a notevoli rischi d'intromissione da parte di malintenzionati.

Si rende necessario, quindi, adottare una modalità di cifratura dei dati adatta all'ambito progettuale.
Nello specifico, si è deciso di utilizzare la cifratura completamente omomorfica.

Questa modalità di cifratura, definita anche Fully Homomorphic Encryption (FHE), è attualmente in fase sperimentativa, poiché ha l'obiettivo di sostituire completamente la precedente RSA.
Si osserverà, nei paragrafi successivi, come quest'ultima presenti delle differenze in termini operativi e di funzionamento rispetto a FHE

4.1 Fully Homomorphic Encryption (FHE)

Nel sistema in esame, tutta la comunicazione è finalizzata all'attuazione del processo fisico sul motore, quindi al pilotaggio dello stesso.
Caratteristica che rende necessario adottare una modalità di cifratura che sia la più sicura possibile.

Permanendo nel dominio in chiaro, la comunicazione è facilmente intercettabile sia a monte che a valle del sistema: in fase di trasmissione tra schede e in fase d'invio del segnale di controllo al motore.

La FHE si prefigge l'obiettivo di evitare questi comportamenti: la crittografia omomorfica attua un cambio di paradigma molto importante nell'ambito della sicurezza informatica.
Mentre i sistemi crittografici tradizionali, ad esempio RSA o AES, garantiscono la sicurezza dei dati esclusivamente durante la trasmissione (data-in-transit) o l'archiviazione (data-at-rest), la FHE permette di mantenere le informazioni riservate anche durante l'elaborazione delle stesse.
Comportamento che prende il nome di data-in-use.

L'elaboratore, oppure un eventuale hacker con accesso diretto in memoria, esegue il programma senza poter vedere i dati originali o i risultati intermedi.
L'utente invia i dati cifrati, il server esegue il calcolo e restituisce il risultato cifrato. Sarà l'utente la sola persona in grado di decifrare il contenuto.

Ponendo la trattazione sul piano puramente matematico, un sistema di crittografia si definisce omomorfico se, dati uno spazio dei messaggi, definito plaintext $\mathcal{M}$, dotato di un'operazione $\star$, e uno spazio dei testi cifrati, definito ciphertext $\mathcal{C}$, dotato di un'operazione $\odot$, è valida la relazione che segue:

$$Dec(Enc(m_1) \odot Enc(m_2)) = m_1 \star m_2 \quad \forall m_1, m_2 \in \mathcal{M} \tag{4.1}$$

dove $Enc(\cdot)$ e $Dec(\cdot)$ rappresentano rispettivamente le funzioni di cifratura e decifratura. Ciò significa che eseguire un'operazione algebrica $\odot$ nel dominio cifrato restituisce un ciphertext che, una volta decifrato, fornisce un risultato identico a quello che si otterrebbe eseguendo l'operazione corrispondente $\star$ direttamente nel dominio in chiaro [6].
Il termine "Fully" è stato introdotto da Craig Gentry nel 2009 e indica un sistema in grado di supportare un numero arbitrario e illimitato di operazioni di addizione e moltiplicazione.
Definito come un sistema "Turing-completo" per l'analisi e comprensione di qualsiasi circuito aritmetico.

Il dato, quindi, viene disaccoppiato dalla fase di computazione vera e propria.
L'entità che esegue i calcoli, nel caso in esame la scheda Master, non ha mai accesso né alla chiave privata né al testo in chiaro.
Questa differenza si evince dalla Fig. 4.1 di seguito

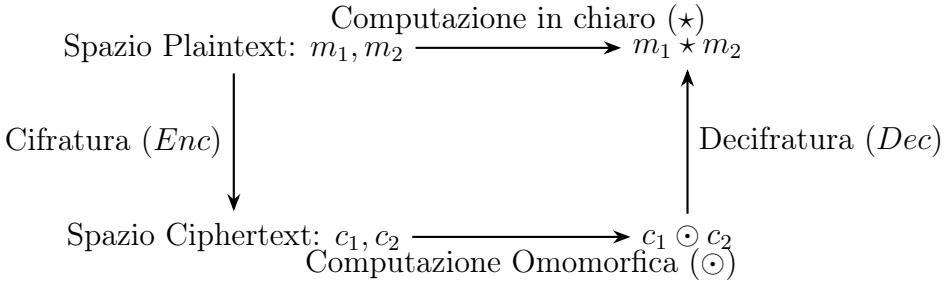

Figura 4.1: Diagramma commutativo del paradigma di Crittografia Omomorfica
Fonte: elaborazione personale

4.1.1 Schema di funzionamento

A differenza di RSA, la sicurezza degli schemi FHE non si basa sulla fattorizzazione dei numeri primi, bensì su problemi complessi dal punto di vista computazionale, in particolare il problema *Learning With Errors* (LWE) e la sua variante su anelli polinomiali, *Ring-Learning With Errors* (RLWE) [6], progettati per resistere anche

ad attacchi eseguiti da parte di computer quantistici.
Lavorando su un anello di polinomi $R_q = \mathbb{Z}_q[X]/(X^N + 1)$ con coefficienti modulo q.
Per chiarire questo concetto matematico a livello pratico, un anello di polinomi può essere immaginato come un insieme di polinomi creato seguendo in maniera scrupolosa due regole, necessarie per evitare che la dimensione dei dati esploda durante l'elaborazione da parte del calcolatore. La prima regola ($\mathbb{Z}_q$) impone che i coefficienti del polinomio non superino mai il valore q: se eccedono il valore, l'operazione di modulo li fa "ripartire da zero", come se fossero lancette di un orologio. La seconda regola ($/(X^N + 1)$) impone che il polinomio non superi mai il grado $N - 1$: ogni volta che una moltiplicazione genera un esponente X^N, quest'ultimo viene eliminato e sostituito algebricamente con il valore -1. In questo spazio matematico a dimensione fissa, la chiave segreta è un polinomio s con coefficienti "piccoli".
Per cifrare un messaggio $m \in R_q$, si genera casualmente un polinomio a e si introduce un polinomio di errore e, noto come **rumore crittografico** , anch'esso con coefficienti piccoli. Il testo cifrato c viene rappresentato dalla coppia di polinomi:

$$c = (a, b) = (a, a \cdot s + e + m) \pmod{q} \tag{4.2}$$

Affinché sia possibile decifrare il messaggio, è richiesta la chiave segreta s per isolarlo. Si calcola:

$$b - a \cdot s = (a \cdot s + e + m) - a \cdot s = m + e \pmod{q} \tag{4.3}$$

Essendo l'errore e sufficientemente piccolo rispetto a q, è possibile rimuoverlo e recuperare il messaggio originario m esattamente.
Si noti che la definizione matematica di queste coppie è necessaria per comprendere al meglio il nucleo dell'omomorfismo, il quale si occupa proprio di svolgere operazioni aritmetiche su queste coppie.
Siano $c_1 = (a_1, b_1)$ e $c_2 = (a_2, b_2)$ le cifrature di due messaggi m_1 e m_2 con i rispettivi errori e_1 e e_2. L'addizione omomorfica si effettua sommando componente per componente:

$$c_{add} = c_1 + c_2 = (a_1 + a_2, b_1 + b_2) \tag{4.4}$$

Si valuta a questo punto il risultato con la chiave segreta s e si ottiene:

$$b_{add} - a_{add} \cdot s = (b_1 + b_2) - (a_1 + a_2)s = (m_1 + m_2) + (e_1 + e_2) \tag{4.5}$$

Matematicamente, è stato dimostrato che il rumore derivante da un'addizione cresce in modo lineare ($e_{add} = e_1 + e_2$)[6]. Nella moltiplicazione omomorfica, tuttavia, ($c_{mult} = c_1 \otimes c_2$), i polinomi di rumore si moltiplicano tra loro, per cui la crescita dell'errore è esponenziale (o quadratica).
È importante considerare il rumore accumulato: se supera il modulo q, il dato viene corrotto irreversibilmente, dando vita al fenomeno che prende il nome di "over-flow" crittografico, e la decifratura fallisce.

È possibile superare questo limite grazie all'operazione di **Bootstrapping** introdotta da Gentry [6].
In particolare, si tratta di una tecnica che valuta il circuito di decifratura omomorficamente su se stesso per "ripulire" il ciphertext dal rumore accumulato, restituendo

un testo cifrato nuovo e pronto per ulteriori operazioni.
Lo schema di funzionamento risultante, come in Fig. 4.2, è esprimibile in termini meno complessi per comprendere al meglio il funzionamento

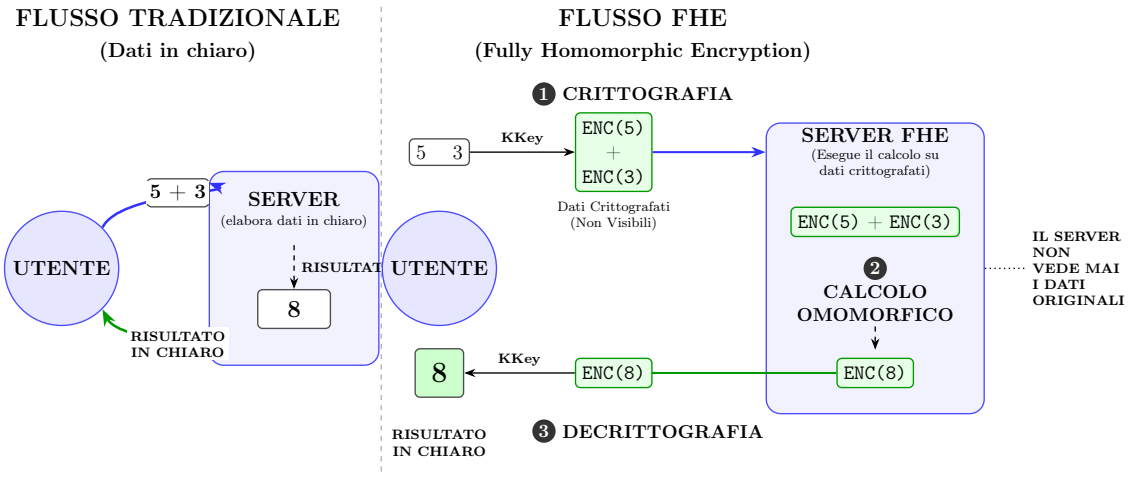

Figura 4.2: Schema di funzionamento FHE. *Fonte: elaborazione personale*

4.1.2 Vantaggi e limiti

Applicare la FHE significa ottenere vantaggi notevoli dal punto di vista teorico e nel controllo distribuito: garantisce la "Privacy by Design" e protegge i segnali di controllo da iniezioni o intercettazioni nemiche, mantenendo intatta la logica di calcolo del controllore.
Tuttavia, dal punto di vista pratico, la FHE presenta limitazioni particolarmente pesanti che incidono sulle prestazioni hardware:

- **Crescita esponenziale della complessità computazionale (Lentezza):** Le operazioni richiedono una mole enorme di cicli di CPU, in particolar modo la moltiplicazione polinomiale e il bootstrapping.
  Il prodotto tra due numeri scalari in chiaro richiede pochi nanosecondi. Prodotto traducibile, nel dominio FHE, in una moltiplicazione tra anelli polinomiali di grado elevato, portando i tempi di esecuzione nell'ordine dei millisecondi o secondi [6].

- **Espansione del testo cifrato:** Esattamente come citato nel paragrafo precedente, un numero, ad esempio 64 bit in virgola mobile, una volta cifrato si trasforma in una coppia di polinomi che può occupare diversi Megabyte in memoria.
  Occupazione in memoria che si traduce in un aumento drastico del consumo di memoria RAM e, soprattutto, in colli di bottiglia sulla banda passante della rete locale per la trasmissione dei dati tra Master e Slave [7].

- **Gestione complessa del rumore:** Poiché è necessario tenere conto dell'accumulo di rumore, si è costretti a prevedere a priori la "profondità moltiplicativa" del circuito oppure si è obbligati a pagare in termini computazionali per

l'operazione particolarmente onerosa del Bootstrapping, utile per mantenere l'integrità del dato.

4.2 Lo schema CKKS

La particolarità più utile per gli scopi del sistema in esame è rappresentata dalla possibilità di usufruire di diversi schemi di cifratura disponibili con FHE.

Attraverso questa caratteristica, si è osservato che lo schema, qualora venga sostituito, può essere adattato con un'unica codebase.
Per l'implementazione del controllore PI (Proporzionale-Integrale) sui dati cifrati, lo schema FHE generale descritto in precedenza non risulta sufficiente se limitato ai soli numeri interi.
I parametri di controllo, che assumono valori frazionari, e i segnali fisici del motore richiedono l'utilizzo di numeri reali, definiti floating-point [7].
A tale scopo, è stato adottato lo schema CKKS (Cheon-Kim-Kim-Song), ideato specificatamente per lavorare con numeri reali e complessi nel dominio cifrato.

4.2.1 Aritmetica approssimata

Gli schemi che lavorano esclusivamente con numeri interi sono definiti esatti poiché si basano sull'aritmetica modulare su interi.
A differenza di schemi esatti come BFV o BGV, lo schema CKKS supporta l'aritmetica approssimata.

L'idea sulla quale è nato lo schema CKKS è trattare il rumore crittografico come parte integrante dell'errore di approssimazione, non come un elemento dannoso da eliminare in fase di decifratura.

Per cifrare un numero reale z, è necessario prima moltiplicarlo per un fattore di scala elevato Δ e successivamente arrotondarlo all'intero più vicino: $m = \lfloor z \cdot \Delta \rceil$.
Così facendo, si garantisce che la precisione frazionaria venga preservata nei bit più significativi dell'intero che risulta da questa operazione.
Questo tipo di approssimazione è tollerata in ambito ingegneristico poiché i segnali fisici, come la velocità acquisita dalla dinamo, sono già affetti da incertezze di misura in maniera intrinseca.

4.2.2 Tecnica del rescaling

Introdurre il fattore di scala Δ comporta un problema notevole per quello che concerne la computazione omomorfica, in particolar modo nelle moltiplicazioni.
Moltiplicando due messaggi cifrati, ciascuno scalato per Δ, il risultato sarà scalato per Δ^2 ($z_1\Delta \cdot z_2\Delta = z_1 z_2 \Delta^2$).
Di conseguenza, se si eseguissero numerose moltiplicazioni in cascata, il fattore di scala crescerebbe esponenzialmente, causando un overflow del dato oltre il modulo q del ciphertext [6].

Si osservi che, come discusso nel Paragrafo 4.1.1, questo fenomeno non deve mai verificarsi. È quindi necessario prevenirlo.
Lo schema CKKS, per far fronte a questo fenomeno, introduce l'operazione di **rescaling**: dopo ogni moltiplicazione omomorfica, il testo cifrato viene diviso per il fattore Δ.
In questo modo, la scala viene riportata al valore originale e i bit meno significativi vengono scartati. Si noti che questi ultimi contengono il rumore accumulato.

Con questa operazione si ottiene l'equivalente, nel campo omomorfico, del troncamento nei calcoli tradizionali. Questo intero flusso, con l'aumento di scala e il successivo ripristino, è riassunto in Fig. 4.3.

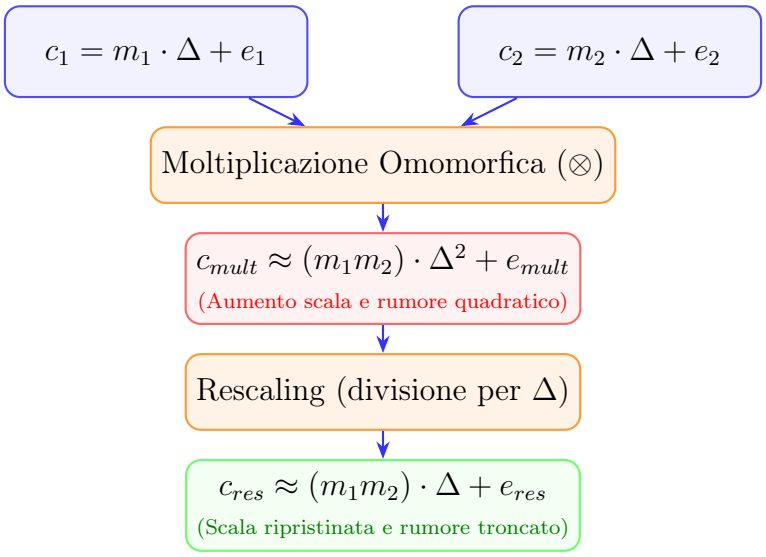

Figura 4.3: Flusso delle operazioni di moltiplicazione e rescaling nello schema CKKS
Fonte: elaborazione personale

4.2.3 Gestione del rumore

A differenza degli altri schemi FHE, nello schema CKKS il rumore è gestito in maniera differente.
Il rumore crittografico e e l'errore di approssimazione, il quale deriva dal rescaling, vengono fusi nei bit meno significativi del messaggio.
Affinché la decifratura restituisca un valore corretto per l'attuazione del processo sul motore, è fondamentale che i bit più significativi rimangano intatti.

Nel progetto in esame, la gestione del rumore è basata sulla calibrazione del modulo q, del fattore di scala Δ e, soprattutto, della profondità moltiplicativa del circuito.

Si osservi che, essendo l'algoritmo del PI a profondità limitata, è possibile dimensionare i parametri iniziali in modo che il rumore non vada mai a intaccare i bit del segnale di comando inviato al DAC.

In questo modo, si evita di dover ricorrere al bootstrapping, migliorando notevolmente le latenze del sistema [6].

4.3 Confronto con crittografia RSA

Precedentemente all'impiego della FHE da parte mia e delle mie colleghe, altri colleghi avevano basato il loro lavoro sperimentale sul protocollo RSA [7].

Per poter comprendere a pieno le innovazioni che ha portato FHE, si procederà ad analizzare RSA e a confrontarlo con la crittografia in esame.
Il protocollo RSA basa la sua sicurezza sulla fattorizzazione di numeri interi molto grandi (ottenuti dal prodotto di due numeri primi), quindi sulla complessità computazionale molto elevata di cui dispone.
Rappresenta uno standard per la sicurezza dei dati in transito. Tuttavia, presenta limitazioni per il controllo distribuito.
In prima analisi, RSA ha una proprietà omomorfica *parziale* e strettamente moltiplicativa: il prodotto di due testi cifrati corrisponde alla cifratura del prodotto dei testi in chiaro.
In nessun modo, tuttavia, viene supportata l'addizione omomorfica. Poiché la legge di controllo di un PI richiede la combinazione lineare di termini proporzionali e integrali, definita come $u(t) = K_p \cdot e(t) + K_i \cdot \int e(t)dt$, RSA non permette l'elaborazione del segnale di controllo nel dominio cifrato.

Inoltre, lo schema RSA si applica esclusivamente a numeri interi.
Questa particolare caratteristica rende complessa e inefficiente la gestione di valori frazionari.
Al contrario, la FHE, nello specifico la variante CKKS descritta nel Paragrafo 4.2, è concepita nativamente per l'aritmetica con numeri reali.

Il vantaggio vero e proprio di RSA rispetto a FHE è rappresentato dai tempi di cifratura e decifratura: RSA presenta tempi notevolmente inferiori e ciphertext molto leggeri rispetto ai polinomi di FHE, i quali sono particolarmente pesanti.

Ai fini del sistema in esame, la crittografia RSA rappresenta un problema importante: per elaborare i dati, il nodo Master della rete sarebbe costretto a decifrare l'errore, calcolare la legge di controllo in chiaro e ricifrare il risultato [7].
Questo tipo di operazione espone i parametri del PID a potenziali attacchi e intercettazioni.
La crittografia omomorfica, invece, elimina totalmente questo tipo di attacco: sacrifica l'efficienza della banda passante di cui dispone RSA con una computazione blindata sui nodi. Questa netta differenza architetturale e le relative implicazioni di sicurezza sul nodo Master sono schematizzate in Fig. 4.4.

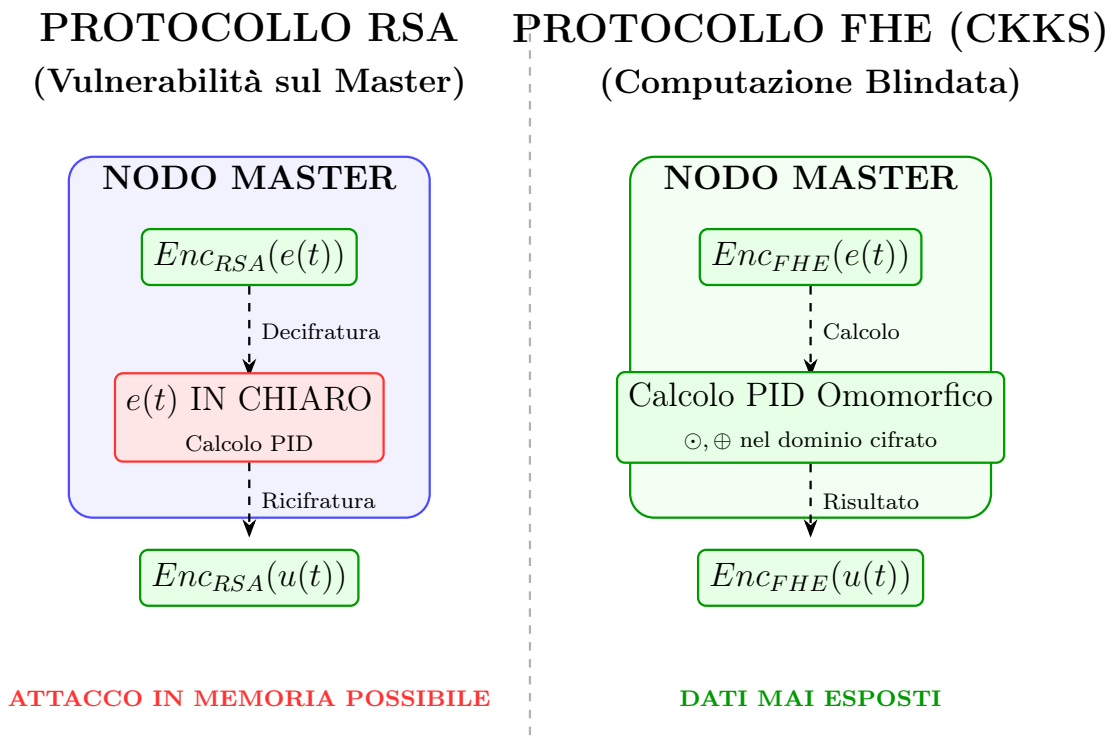

Figura 4.4: Confronto architetturale e di sicurezza tra RSA e FHE sul nodo Master
Fonte: elaborazione personale

Capitolo 5

Implementazione del Controllo Omomorfico

Si rende ora necessario effettuare una transizione: da un piano puramente teorico a uno piuttosto pratico.

Finora, si è trattato il controllo omomorfico dal punto di vista semantico e, soprattutto, matematico.
Affinché sia possibile applicare i concetti discussi nel capitolo precedente, il controllo è da applicarsi fisicamente all'interno del sistema in esame.

Si osserverà come il sistema, anche con l'implementazione della FHE, fornisca a valle del circuito il segnale di controllo necessario per l'attuazione del processo.

5.1 Configurazione della libreria OpenFHE

Per rendere effettiva l'applicazione di FHE nel sistema, si agisce tramite codice scritto attraverso il linguaggio di programmazione `C++` (estensione .cpp).

Il codice, per quanto concerne la crittografia FHE, necessita di un'apposita libreria, chiamata OpenFHE.
La procedura d'installazione è da effettuarsi attenendosi scrupolosamente alla documentazione ufficiale [8].

L'installazione va eseguita per entrambe le schede Raspberry e, poiché durante il processo richiede alcuni file online, necessitano di connessione a internet: si utilizzi, quindi, il modulo WI-FI presente a bordo per garantire una corretta installazione della libreria.

Una volta terminata questa fase, va installato CMake per la compilazione in `C++` e creato un file di configurazione rinominato `CMakeLists.txt`, necessario alla libreria a runtime.

```
cmake_minimum_required(VERSION 3.16.3)
project(demo C CXX)
set(CMAKE_CXX_STANDARD 17)

find_package(OpenFHE CONFIG REQUIRED)

set(CMAKE_CXX_FLAGS ${OpenFHE_CXX_FLAGS})

include_directories(${OpenFHE_INCLUDE})
include_directories(${OpenFHE_INCLUDE}/third-party/include)
include_directories(${OpenFHE_INCLUDE}/core)
include_directories(${OpenFHE_INCLUDE}/pke)
include_directories(/usr/local/include)
link_directories(/usr/local/lib)

link_directories(${OpenFHE_LIBDIR})
link_libraries(${OpenFHE_SHARED_LIBRARIES})

add_executable(Prova slave_sinusoide.cpp DEV_Config.c ADS1256.c DAC8532.c
                openfhe_bridge.cpp socket_utils.cpp)

set_source_files_properties(DEV_Config.c PROPERTIES LANGUAGE C)
set_source_files_properties(ADS1256.c PROPERTIES LANGUAGE C)
set_source_files_properties(DAC8532.c PROPERTIES LANGUAGE C)

target_link_libraries(Prova lgpio)
```

Di seguito si riportano le spiegazioni dettagliate, riga per riga, delle istruzioni fondamentali contenute nel file di cui sopra:

- `find_package(OpenFHE CONFIG REQUIRED)` — CMake cerca il file di configurazione installato dalla libreria (`OpenFHEConfig.cmake`). Qualora non venga trovato, la compilazione si interrompe.
  È quindi necessario aver già seguito la procedura d'installazione e compilazione di OpenFHE sul sistema. Si noti che, nell'eventualità in cui la libreria non sia stata compilata e installata, è possibile farlo con il comando `sudo make install` dalla sua cartella di build [8].

- `set(CMAKE_CXX_FLAGS ${OpenFHE_CXX_FLAGS})` — Importa i flag di compilazione specifici di OpenFHE, tra cui `-fopenmp` e le opzioni di ottimizzazione necessarie per i calcoli CKKS polinomiali.

- Le `include_directories` aggiungono i percorsi dei vari header di OpenFHE. In particolare `/pke` contiene `openfhe.h`, l'header principale, e `/binfhe` è invece utilizzato per schemi binari.
  Schemi che, per gli scopi del progetto in esame, non sono stati usati. Tuttavia, vengono comunque inclusi per coerenza con l'installazione standard.

- `link_libraries(${OpenFHE_SHARED_LIBRARIES})` — Collega le librerie condivise (`.so`) di OpenFHE al file eseguibile. Queste includono `OPENFHEpke`, `OPENFHEcore` e `OPENFHEbinfhe`.

- `add_executable(Prova ...)` — definisce l'eseguibile finale (`Prova`) che aggrega i file C++ dello Slave con i driver hardware in C (`DEV_Config.c`, `ADS1256.c`, `DAC8532.c`). Inoltre, include anche i file relativi a bridge e socket, i quali verranno approfonditi nel Paragrafo 5.5.
  Il nome dell'eseguibile può essere scelto in modo arbitrario, è sufficiente sostituirlo all'interno di add_ executable().

- `set_source_files_properties(...LANGUAGE C)` — Necessario perché CMake, come comportamento di default, tratta tutti i file come C++. I driver Waveshare, che riguardano quindi DAC e ADC, sono C puro e devono essere compilati attraverso il compilatore C (`gcc`), non C++ (`g++`).

- `target_link_libraries(Prova lgpio)` — Collega la libreria LGPIO, già approfondita nel Paragrafo 3.2.

5.2 Caricamento contesto e chiavi

Predisposto l'ambiente di lavoro, è ora fondamentale generare contesto e chiavi per permettere ai nodi Master e Slave di leggerli all'avvio.

Inizialmente, è necessario produrre un file, chiamato nel caso in esame `keygen.cpp`, necessario per definire i parametri dello schema CKKS.
Inoltre, genera la Chiave Privata per lo Slave, la Chiave Pubblica per entrambi i nodi e la Chiave di Moltiplicazione.

```
#include "openfhe.h"

#include "ciphertext-ser.h"
#include "cryptocontext-ser.h"
#include "key/key-ser.h"
#include "scheme/ckksrns/ckksrns-ser.h"

using namespace lbcrypto;
using namespace std;

int main() {

    CCParams<CryptoContextCKKSRNS> parametri;

    parametri.SetSecurityLevel(SecurityLevel::HEStd_NotSet);
    parametri.SetRingDim(4096);
    parametri.SetMultiplicativeDepth(2);
    parametri.SetScalingModSize(30);
    parametri.SetBatchSize(8);
```

```
20
21  auto level = parametri.GetSecurityLevel();
22  std::cout << "Security level: " << level << std::endl;
23
24      //Generiamo il contesto
25      CryptoContext<DCRTPoly> cc = GenCryptoContext(parametri);
26
27      //Abilitiamo la chiave publica, il key switch e le addizioni e
        ↪  moltiplicazioni FH
28      cc->Enable(PKE);
29      cc->Enable(KEYSWITCH);
30      cc->Enable(LEVELEDSHE);
31
32      KeyPair<DCRTPoly> keyPair = cc->KeyGen();
33      cc -> EvalMultKeyGen(keyPair.secretKey);
34      cc->EvalAtIndexKeyGen(keyPair.secretKey, {1, 2, -1, -2});
35      //Salvataggio su file
36      string dati = "dati";
37
38      //Check salvataggio su file
39      if (!Serial::SerializeToFile(dati + "/cryptocontext.txt", cc,
        ↪  SerType::BINARY)) {
40          cerr << "Errore nel salvataggio del cryptocontext" << endl;
41          return 1;
42      }
43
44      if (!Serial::SerializeToFile(dati + "/key-public.txt",
        ↪  keyPair.publicKey, SerType::BINARY)) {
45          cerr << "Errore nel salvataggio della chiave pubblica" << endl;
46          return 1;
47      }
48
49      if (!Serial::SerializeToFile(dati + "/key-private.txt",
        ↪  keyPair.secretKey, SerType::BINARY)) {
50          cerr << "Errore nel salvataggio della chiave privata" << endl;
51          return 1;
52      }
53
54
55      ofstream multKeyFile(dati + + "/key-eval-mult.txt", ios::out |
        ↪  ios::binary);
56      if (multKeyFile.is_open()) {
57          if (!cc->SerializeEvalMultKey(multKeyFile, SerType::BINARY)) {
58              cerr << "Errore nel salvataggio della chiave di
                ↪  moltiplicazione" << endl;
59              return 1;
60          }
61          multKeyFile.close();
62      }
```

```
63
64
65  std::ofstream rotFile("dati/key-eval-rot.txt");
66
67  if (!cc->SerializeEvalAutomorphismKey(rotFile, SerType::BINARY)) {
68
69      std::cerr << "Errore nel salvataggio delle rotation keys\n";
70
71      return 1;
72
73  }
74
75  rotFile.close();
76
77
78      cout<<"File salvati nella cartella"<< dati <<endl;
79      return 0;
80
81  }
```

Solo dopo aver generato correttamente il contesto, a cui vengono passati i parametri di riferimento scelti per i nostri scopi, e le chiavi, è possibile passare all'applicazione sui nodi Master e Slave.
Le chiavi sono strettamente legate ai parametri matematici dello schema, nel progetto in esame CKKS

Inoltre, il file di cui sopra salva tutto in dei file che i due nodi andranno a leggere una sola volta all'avvio.

5.2.1 Master

Il codice che segue è necessario per caricare il contesto e le chiavi di cui sopra nel nodo Master.

`master_openfhe.cpp`

```
1   bool load_context_and_keys(CryptoContext<DCRTPoly>& cc,
2                              PublicKey<DCRTPoly>& publicKey) {
3
4       if (!Serial::DeserializeFromFile("dati/cryptocontext.txt", cc,
5                                        SerType::BINARY)) {
6           std::cerr << "[MASTER] Errore caricamento cryptocontext.txt\n";
7           return false;
8       }
9
10      if (!Serial::DeserializeFromFile("dati/key-public.txt", publicKey,
11                                       SerType::BINARY)) {
12          std::cerr << "[MASTER] Errore caricamento key-public.txt\n";
```

```
13          return false;
14      }
15
16      std::ifstream mkf("dati/key-eval-mult.txt", std::ios::in |
        ↪  std::ios::binary);
17      if (!mkf.is_open() ||
18          !cc->DeserializeEvalMultKey(mkf, SerType::BINARY)) {
19          std::cerr << "[MASTER] Errore caricamento key-eval-mult.txt\n";
20          return false;
21      }
22
23      std::ifstream rotFile("dati/key-eval-rot.txt",
24                            std::ios::in | std::ios::binary);
25      if (!rotFile.is_open() ||
26          !cc->DeserializeEvalAutomorphismKey(rotFile, SerType::BINARY)) {
27          std::cerr << "[MASTER] Errore caricamento rotation keys\n";
28          return false;
29      }
30
31      return true;
32  }
```

- `CryptoContext<DCRTPoly>` — rappresenta l'oggetto fondamentale di OpenFHE.
  Contiene tutti i parametri dello schema CKKS e tutte le operazioni omomorfiche (cifratura, decifratura, moltiplicazione, addizione). `DCRTPoly` (Double-CRT Polynomial) è un tipo di ritorno specifico per quanto concerne la rappresentazione matematica interna dei polinomi.

- `SerType::BINARY` — Poiché il cyphertext non è trasmissibile nello stato attuale, è necessario applicare una serializzazione, nello specifico in formato binario. Formato più compatto e veloce rispetto a JSON, ottimizzato quindi per la latenza.

- `DeserializeEvalMultKey` — le chiavi di valutazione per la moltiplicazione non sono parte del `CryptoContext` base: dipendono dalla chiave segreta e dall'algoritmo di relinearizzazione (*relinearization keys*). Senza questa chiave, l'operazione `cc->EvalMult(...)` fallisce.
  Operazione che verrà approfondita nel Paragrafo 5.4 relativo all'esecuzione del controllo PI cifrato.

- `DeserializeEvalAutomorphismKey` — chiavi per le operazioni di rotazione o shift sui vettori CKKS.

- `key-private.txt` — la chiave privata non è presente in questa fase poiché il Master non ha il compito di decifrare.

  Riceve il ciphertext, esegue le operazioni omomorfiche e trasmette direttamente il nuovo ciphertext contenente le operazioni svolte.

La chiave privata è esclusiva dello Slave. Questo tipo di approccio garantisce il disaccoppiamento del dato.

5.2.2 Slave

Il codice che segue è necessario per caricare il contesto e le chiavi nel nodo Slave.

`slave_openfhe.cpp`

```
    bool load_context_and_keys(CryptoContext<DCRTPoly>& cc,
                           PublicKey<DCRTPoly>& publicKey,
                           PrivateKey<DCRTPoly>& privateKey) {

    if (!Serial::DeserializeFromFile("dati/cryptocontext.txt", cc,
                                     SerType::JSON)) {
        std::cerr << "[SLAVE] Errore caricamento cryptocontext.txt\n";
        return false;
    }
    if (!Serial::DeserializeFromFile("dati/key-public.txt", publicKey,
                                     SerType::JSON)) {
        std::cerr << "[SLAVE] Errore caricamento key-public.txt\n";
        return false;
    }
    if (!Serial::DeserializeFromFile("dati/key-private.txt", privateKey,
                                     SerType::JSON)) {
        std::cerr << "[SLAVE] Errore caricamento key-private.txt\n";
        return false;
    }
    return true;
}
```

Si noti con attenzione che i file di testo contenenti rispettivamente il cryptocontext, la public e la private key vengono associati a tre variabili passate per riferimento come parametri della funzione booleana.
Questa scelta è stata fatta per renderli utilizzabili durante lo sviluppo del codice, come si osserverà nei paragrafi successivi.

Di seguito è illustrato il significato del codice di cui sopra:

- **Caricamento di `key-public.txt` e `key-private.txt`** — a differenza del Master, lo Slave carica sia la chiave pubblica sia la chiave privata.
  La prima sarà utile per cifrare l'errore letto dal sensore, mentre la seconda si renderà indispensabile per decifrare il comando di controllo $u(t)$ elaborato e ricevuto dal Master.

- `SerType::JSON` — lo Slave utilizza la serializzazione in formato JSON anziché in formato binario. Si è osservato che, nonostante questa scelta risulti meno efficiente in termini di dimensioni del payload, JSON garantisce elevata leggibilità del dato. Si rivela, quindi, fondamentale in fase di sviluppo e debug.

- **Mancanza delle chiavi di valutazione** — lo Slave non carica in memoria i file `key-eval-mult.txt` e `key-eval-rot.txt`. Dal momento che non deve eseguire moltiplicazioni omomorfiche o rotazioni, queste chiavi sono superflue per i suoi scopi.
  Infatti, l'onere computazionale per le operazioni di alto livello è delegato al nodo Master.

5.3 Acquisizione dell'errore, cifratura e attuazione

In questo paragrafo si analizzerà il core del loop di controllo all'interno dello Slave.

Il primo elemento fondamentale da analizzare è sicuramente l'ADC e la successiva conversione in RPM per attuare il processo sul motore.

`slave_openfhe.cpp`

```
    ADS1256_GetAll(ADC);

double raw_voltage = (ADC[3] * 3.3) / 0x7FFFFF;
if (raw_voltage < 0.0) raw_voltage = 0.0;
if (raw_voltage > 3.3) raw_voltage = 3.3;

filtered = FILTER_ALPHA * raw_voltage + (1.0 - FILTER_ALPHA) * filtered;

const double measured_voltage = filtered;
const double measured_rpm = (measured_voltage / MAX_VOLTAGE_IN) *
↪  MAX_RPM_OUT;
```

- `ADS1256_GetAll(ADC)` — Funzione che si occupa di leggere tutti gli 8 canali dell'ADC ADS1256 a 24 bit.
  Il canale 3 , nel codice `ADC[3]`, è collegato all'uscita della dinamo tachimetrica tramite il partitore di tensione, come si è osservato precedentemente in Fig. 2.2 del Capitolo 2.

- **Conversione in Volt** — la formula `(ADC[3] * 3.3) / 0x7FFFFF` è fondamentale per convertire il valore grezzo a 23 bit in volt.
  In binario si conta a partire da 0, il quale rappresenta il bit meno significativo (LSB)
  Il valore esadecimale `0x7FFFFF`, che in decimale corrisponde a $8388607 = 2^{23} - 1$, rappresenta il fondo scala positivo del convertitore a 24 bit con segno, poiché il bit più significativo (MSB) funge da segno.

- `[0, 3.3] V` — Con la tecnica del clamping, si forza la lettura all'interno di questo range. È necessario per evitare che rumori o offset eventuali portino la lettura a valori fuori dal range fisico di funzionamento, oltre il quale il motore rischia di bruciarsi.

- **Filtro passa-basso digitale** — Poiché il segnale dell'ADC è soggetto a rumore, per evitare di incorrere in letture falsate, si è ricorsi all'implementazione di un filtro PB digitale. Agisce tramite l'equazione del primo ordine $\texttt{filtered} = \alpha \cdot \texttt{raw} + (1 - \alpha) \cdot \texttt{filtered}$ con $\alpha = \texttt{FILTER_ALPHA} = 0.15$. In particolare, un valore di α piccolo produce un filtro più aggressivo: la risposta risulta più lenta, il segnale più pulito; il valore assegnato al parametro, pari a 0.15, è stato calibrato sperimentalmente come compromesso tra reattività e stabilità.

- **Conversione in RPM** — Avviene in maniera lineare: considerando $\texttt{MAX_VOLTAGE_IN} = 5.0$ V e $\texttt{MAX_RPM_OUT} = 6000$ RPM, la relazione che produce gli RPM misurati è data dal rapporto dei volt misurati filtrati e il voltaggio massimo in ingresso, entrambi moltiplicati per gli RPM massimi in uscita.

Si osservi ora come è stato calcolato l'errore in base al segnale di riferimento. In merito a quest'ultimo, poiché sperimentalmente è stato utilizzato, in ultima analisi, il riferimento sinusoidale, il codice che segue fa riferimento allo stesso.
Si tenga in considerazione che le modalità utilizzate valgono allo stesso modo per il riferimento a rampa.

```
    double sinusoidal_reference_raw =
    AMPLITUDE_RPM * sin(2 * M_PI * FREQUENCY_REF * current_time_s) + 200;
double sinusoidal_reference = sinusoidal_reference_raw + OFFSET_RPM;

if (sinusoidal_reference < 200)    sinusoidal_reference = 200;
if (sinusoidal_reference > 2200.0) sinusoidal_reference = 2200.0;

const double error = sinusoidal_reference - measured_rpm;
```

- **Segnale di riferimento** — segnale sinusoidale impostato con frequenza di 0.2 Hz, caratterizzato da un'ampiezza di 1000 RPM e un offset di 1000 RPM. Il valore di riferimento viene limitato necessariamente nell'intervallo `[200, 2200]` RPM al fine di proteggere il motore da saturazione e problemi fisici.

- **Calcolo dell'errore** $e(k)$ — Rappresenta l'errore di inseguimento. Matematicamente, si definisce come $e(k) = r(k) - y(k)$, dove $r(k)$ rappresenta il riferimento sinusoidale e $y(k)$ gli RPM misurati.
  Prima di procedere con la fase di cifratura, viene calcolato totalmente in chiaro. Questa operazione è l'unica che avviene nel dominio in chiaro ed è eseguita dal nodo Slave: l'elaborazione successiva della legge di controllo del PID avviene sul nodo Master nel dominio cifrato.

A questo punto, l'errore è, come anticipato, nel dominio in chiaro. Va, pertanto, cifrato.
Il codice seguente mostra in maniera esplicativa il processo di cifratura.

```
std::vector<double> error_vec = {error};

```

```
const auto e_pt = cc->MakeCKKSPackedPlaintext(error_vec);
const auto e_enc = cc->Encrypt(publicKey, e_pt);
```

Come si osserva dal codice, il processo è piuttosto semplice lato programmatore: il lavoro viene svolto principalmente dalle funzioni messe a disposizione da OpenFHE.
Andando nel dettaglio, la prima cosa da fare è rendere vettoriale l'errore precedentemente calcolato. Questo passaggio è necessario poiché, nella riga successiva, verrà convertito in plaintext CKKS.

- `MakeCKKSPackedPlaintext` — converte il vettore di tipo `double` in un plaintext CKKS. Lo schema CKKS utilizza un tipo particolare di codifica che prende il nome di *slot encoding*: un singolo ciphertext può contenere fino a $N/2$ valori reali in parallelo. N rappresenta la dimensione dell'anello polinomiale.

- `cc->Encrypt(publicKey, e_pt)` — In questo momento l'errore, sotto forma di plaintext, viene cifrato utilizzando la chiave pubblica.
  Il ciphertext che risulta da questa operazione è una coppia di polinomi $(a, b) \in R_q^2$, come descritto nel Capitolo 4. In base ai parametri N e q definiti e discussi precedentemente, la dimensione può variare: tipicamente, siamo nell'ordine di centinaia di KB.

È ora possibile analizzare la parte relativa all'invio dell'errore appena trattato, alla ricezione del segnale di controllo cifrato e alla decifratura dello stesso.

```
    if (!send_ciphertext(sock_master, e_enc)) { break; }

    Ciphertext<DCRTPoly> u_enc;
    if (!recv_ciphertext(sock_master, u_enc)) { break; }

    Plaintext u_pt;
    cc->Decrypt(privateKey, u_enc, &u_pt);
    u_pt->SetLength(1);
    double u_raw = u_pt->GetCKKSPackedValue()[0].real();
```

- `send_ciphertext / recv_ciphertext` — Si tratta di funzioni che serializzano il ciphertext in un buffer binario, lo precedono con 4 byte di lunghezza, ovvero assegnano un *header*, e lo trasmettono via TCP, il quale verrà dettagliato nel Paragrafo 5.5.

- `cc->Decrypt(privateKey, u_enc, &u_pt)` — Utilizzando un'espressione lambda che tiene conto del cryptocontext, la funzione in questione decifra l'ingresso con la chiave privata per isolare $m + e$, ovvero il messaggio vero e proprio a cui si somma l'errore.
  Poiché l'errore crittografico e è piccolo rispetto a q, si ottiene una buona approssimazione del valore in floating point.

- `SetLength(1)` — Per la proprietà dello *slot encoding*, il ciphertext CKKS decodificato ha sempre $N/2$ slot. Poiché ne viene usato solo uno, si forza la lunghezza per leggere solo il primo slot, ovvero quello con il comando di controllo.

- `.real()` — Lo schema CKKS lavora utilizzando l'insieme dei numeri complessi $\mathbb{C}$. Poiché la parte immaginaria dei numeri che si ottengono rappresenta l'errore di approssimazione, con `.real()` lo si scarta.

Si procede ora utilizzando l'ingresso `u_raw` appena ricavato, si applica un clamping per limitare la variazione massima che ha questo segnale rispetto ai vari campioni.
Successivamente, attraverso il DAC, si applica il processo al motore. Di seguito il dettaglio.

```
1  double du = std::clamp(u_raw, -MAX_DU_PER_SAMPLE, MAX_DU_PER_SAMPLE);
2  double u_sat = u_prev + du;
3  u_sat = std::clamp(u_sat, config::U_MIN_VOLT, U_SAFE_MAX);
4
5  if (u_sat > 0.0 && u_sat < MOTOR_MIN_EFFECTIVE_U)
6      u_sat = MOTOR_MIN_EFFECTIVE_U; // 0.20 V
7
8  u_value = u_sat;
9  u_prev = u_value;
10
11 DAC8532_Out_Voltage(channel_A, static_cast<float>(u_value));
```

- `MAX_DU_PER_SAMPLE` — Si occupa di limitare la variazione massima del segnale di controllo per campione a 0.12 V.
  Si noti che questa misura di sicurezza è fondamentale per proteggere il motore da transitori molto bruschi, poiché l'errore di approssimazione introdotto dallo schema CKKS potrebbe generare scatti problematici tra campioni consecutivi.

- **Forma velocity, oppure incrementale, del controllore** — Si tratta di una tipologia di logica definita dall'espressione `u_sat = u_prev + du`: lo Slave aggiunge, come si osserva, l'incremento Δu elaborato e ricevuto dal Master. Si osservi che Δu viene usato solo ed esclusivamente previa operazione di clamping, sfruttando proprio il segnale ricevuto dal Master `u_raw`. Di conseguenza, il segnale di controllo viene mantenuto nella variabile `u_prev`.

- `MOTOR_MIN_EFFECTIVE_U` — Qualora il segnale calcolato `u_sat` sia positivo ma inferiore a 0.20 V, viene forzato a questo valore, poiché rappresenta la soglia minima di funzionamento del motore. Con questa operazione, si riesce a coprire la zona morta, definita *deadband*, dello stesso.

- `DAC8532_Out_Voltage(channel_A, ...)` — Solo arrivati a questo punto è possibile, attraverso la funzione in questione, scrivere la tensione di controllo appena calcolata sul canale A del DAC.
  Il segnale digitale, quindi, viene convertito in analogico e raggiunge il motore passando attraverso il partitore di tensione e l'amplificatore a MOSFET.

5.4 Esecuzione del controllore PI sui dati cifrati

Questo paragrafo ha l'obiettivo di analizzare il nucleo vero e proprio dell'omomorfismo.
Come accennato nel paragrafo precedente, il Master basa il suo funzionamento su una forma velocity, o incrementale.
In particolare, quindi, opera un controllore PI in questa forma nel dominio cifrato.

`master_openfhe.cpp`

```
1  double k1 = config::KP + (config::KI * config::TS); // KP + KI·Ts
2  double k2 = config::KP;
3
4  auto zero_pt = cc->MakeCKKSPackedPlaintext(std::vector<double>{0.0});
5  auto e_prev_enc = cc->Encrypt(publicKey, zero_pt);
6  int cycle_count = 0;
7
8  while (true) {
9      Ciphertext<DCRTPoly> e_enc;
10     if (!recv_ciphertext(sock_slave, e_enc)) break;
11
12     Ciphertext<DCRTPoly> u_enc;
13
14     if (cycle_count <= 1) {
15         u_enc = cc->Encrypt(publicKey, zero_pt);
16     } else {
17         auto term1 = cc->EvalMult(e_enc,      k1); // k1 * e_k
18         auto term2 = cc->EvalMult(e_prev_enc, k2); // k2 * e_{k-1}
19         u_enc = cc->EvalSub(term1, term2);          // ΔU = k1·e_k -
           ↪  k2·e_{k-1}
20     }
21
22     e_prev_enc = e_enc; // aggiornamento memoria
23
24     if (!send_ciphertext(sock_slave, u_enc)) break;
25 }
```

Il motivo dietro l'utilizzo della forma velocity è dovuto alla profondità moltiplicativa e al relativo "consumo" di rumore.

Nel dettaglio, nella forma standard il controllore PI richiede di sommare, per ogni passo, il termine integrale $K_I \cdot \sum e_k \cdot T_s$. Per farlo, l'operazione necessita di uno stato interno $\sum e$ cifrato, il quale sia aggiornabile nel tempo.
Se si optasse per questa opzione, si avrebbero moltiplicazioni omomorfiche in cascata e la profondità moltiplicativa del circuito con il relativo consumo di rumore crescerebbero notevolmente.
Introducendo, quindi, la forma incrementale si evita questo problema.

La legge di controllo diventa:

$$\Delta U_k = (K_P + K_I T_s) \cdot e_k - K_P \cdot e_{k-1} \tag{5.1}$$

con $k_1 = K_P + K_I T_s$ e $k_2 = K_P$. In questo modo, ogni ciclo è indipendente e a profondità moltiplicativa costante pari a 1.
Si noti che la variazione del segnale di controllo è direttamente proporzionale all'errore misurato allo stato attuale e a quello precedente.
I termini Proporzionale e Integrale agiscono opportunamente e garantiscono un segnale di controllo ottimale.

- `cc->EvalMult(e_enc, k1)` — Si tratta di una funzione messa a disposizione da OpenFHE che si occupa di moltiplicare un ciphertext per uno scalare in chiaro. È molto più economica della moltiplicazione ciphertext-ciphertext perché non richiede di relinearizzare il risultato. Inoltre, introduce meno rumore.

  Si osservi che la funzione appena introdotta è necessaria per il calcolo del primo termine (`term1`). Il calcolo del secondo termine (`term2`) utilizza la stessa logica: l'unica differenza è sull'utilizzo dell'errore allo stato precedente.
  Entrambi i termini sono necessari per l'operazione che segue nel punto successivo.

- `cc->EvalSub(term1, term2)` — Esegue una sottrazione omomorfica tra due ciphertext, nello specifico i due termini visti nel punto precedente. Inoltre si osservi che, come discusso nel Paragrafo 4.1.1, il rumore cresce linearmente nell'addizione e sottrazione, non quadraticamente.

- `e_prev_enc = e_enc` — Assegnazione fondamentale poiché il Master mantiene in memoria il ciphertext dell'errore precedente.
  Questo è l'unico stato interno del Master: non conosce il valore reale dell'errore, mantiene solo la sua versione cifrata.

- **Parametri di controllo** — `KP`, `KI` e `TS` sono definiti nel file `config.hpp` e vengono distribuiti su entrambi i nodi.

5.5 Codice del sistema di comunicazione

Finora, si è trattato tutto l'aspetto relativo al funzionamento interno dei due nodi e sono stati fatti solo accenni al sistema di comunicazione. Ora si procederà ad analizzarlo nel dettaglio.

La comunicazione è realizzata con il protocollo TCP (Transmission Control Protocol) attraverso l'apertura di un socket per la trasmissione affidabile e poi di un "bridge" che serializza il ciphertext.

socket_utils.cpp

```cpp
bool send_all(int sock, const void* data, std::size_t len) {
    const char* ptr = static_cast<const char*>(data);
    std::size_t total = 0;
    while (total < len) {
        const ssize_t sent = ::send(sock, ptr + total, len - total, 0);
        if (sent <= 0) return false;
        total += static_cast<std::size_t>(sent);
    }
    return true;
}

bool recv_all(int sock, void* data, std::size_t len) {
    char* ptr = static_cast<char*>(data);
    std::size_t total = 0;
    while (total < len) {
        const ssize_t recvd = ::recv(sock, ptr + total, len - total, 0);
        if (recvd <= 0) return false;
        total += static_cast<std::size_t>(recvd);
    }
    return true;
}
```

Le funzioni `send()` e `recv()` dello standard POSIX non garantiscono il trasferimento di tutti i byte in una singola chiamata di sistema.
Questo avviene in quanto il kernel può frammentare in modo arbitrario i dati in più segmenti TCP. Poiché nel progetto in esame si lavora con oggetti OpenFHE serializzati, la cui dimensione è dell'ordine centinaia di KB, fare solo una chiamata a `recv()` restituisce solo una porzione del buffer totale. Ciò determina la corruzione del processo di deserializzazione.

Si può risolvere questo problema con le funzioni `send_all` e `recv_all` che implementano una logica *loop-until-done*: continuano a iterare finché non hanno trasmesso o ricevuto l'esatta lunghezza specificata dal parametro `len`, ovvero la corretta quantità di byte.

Inoltre, per semplificare la sintassi, si introducono i template `send_pod<T>` e `recv_pod<T>`. Questi ultimi wrappano le funzioni `send_all` e `recv_all` per i tipi primitivi, come i `double`, eliminando la necessità di effettuare il cast manuale dei puntatori a `char*`.

Si osservi di seguito come i ciphertext vengano serializzati in `openfhe_bridge.cpp`.
`openfhe_bridge.cpp`

```
    template <typename T>
bool send_serialized_object(int sock, const T& obj) {
    std::stringstream ss;
    Serial::Serialize(obj, ss, SerType::BINARY);
    const std::string data = ss.str();

    const uint32_t len_net = htonl(static_cast<uint32_t>(data.size()));
    if (!send_all(sock, &len_net, sizeof(len_net))) return false;
    return send_all(sock, data.data(), data.size());
}

template <typename T>
bool recv_serialized_object(int sock, T& obj) {
    uint32_t len_net = 0;
    if (!recv_all(sock, &len_net, sizeof(len_net))) return false;

    const uint32_t len = ntohl(len_net);
    std::vector<char> buffer(len);
    if (!recv_all(sock, buffer.data(), buffer.size())) return false;

    std::stringstream ss;
    ss.write(buffer.data(), static_cast<std::streamsize>(buffer.size()));
    Serial::Deserialize(obj, ss, SerType::BINARY);
    return true;
}
```

Si è adottato il protocollo di framing TLV (*Type-Length-Value*) semplificato a *Length-Value*: vediamone il funzionamento per punti

- **Serializzazione del ciphertext** in un `stringstream` usando `SerType::BINARY`.
- **Calcolo della lunghezza in byte** e conversione in *network byte order* con `htonl()` (*host-to-network long*) — Questa scelta è stata fatta per garantire portabilità tra architetture *little-endian* (x86) e *big-endian*.
- **Invio dei 4 byte di lunghezza** seguiti dal payload, ovvero dal carico di lavoro.
- **In ricezione, il procedimento è inverso**: si leggono prima i 4 byte di lunghezza, si alloca un buffer della dimensione esatta, si riceve il payload e si deserializza.
  Si osservi che utilizzare `std::vector<char>` allocato dinamicamente è necessario perché la dimensione del ciphertext varia in funzione dei parametri CKKS e non è nota a compile time.

5.6 Codice completo del sistema di controllo sicuro

5.6.1 Nodi del sistema

master_openfhe.cpp

```cpp
#include "config.hpp"

#include "openfhe_bridge.hpp"

#include "socket_utils.hpp"


#include "openfhe.h"


// Serializzazione OpenFHE

#include "cryptocontext-ser.h"

#include "ciphertext-ser.h"

#include "pke/cryptocontext-ser.h"

#include "key/key-ser.h"

#include "scheme/ckksrns/ckksrns-ser.h"


#include <arpa/inet.h>

#include <netinet/in.h>

#include <sys/socket.h>

#include <unistd.h>


#include <chrono>

#include <fstream>

#include <iostream>

#include <string>

#include <vector>


using namespace lbcrypto;
```

```
50
51
52
53  namespace {
54
55
56
57  int setup_server(int port) {
58
59      int server_fd = ::socket(AF_INET, SOCK_STREAM, 0);
60
61
62
63      if (server_fd < 0) {
64
65          std::perror("master: socket");
66
67          return -1;
68
69      }
70
71
72
73      int opt = 1;
74
75
76
77      if (::setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) <
        ↪  0) {
78
79          std::perror("master: setsockopt");
80
81          ::close(server_fd);
82
83          return -1;
84
85      }
86
87
88
89      sockaddr_in addr{};
90
91      addr.sin_family = AF_INET;
92
93      addr.sin_addr.s_addr = INADDR_ANY;
94
95      addr.sin_port = htons(port);
96
97
98
99      if (::bind(server_fd, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) < 0)
        ↪  {
100
101         std::perror("master: bind");
102
103         ::close(server_fd);
```

```
104
105         return -1;
106
107     }
108
109
110
111     if (::listen(server_fd, 1) < 0) {
112
113         std::perror("master: listen");
114
115         ::close(server_fd);
116
117         return -1;
118
119     }
120
121
122
123     std::cout << "[MASTER] In ascolto sulla porta "
124
125               << port << "..." << std::endl;
126
127
128
129     const int client_fd = ::accept(server_fd, nullptr, nullptr);
130
131
132
133     if (client_fd < 0) {
134
135         std::perror("master: accept");
136
137         ::close(server_fd);
138
139         return -1;
140
141     }
142
143
144
145     std::cout << "[MASTER] Slave OpenFHE connesso." << std::endl;
146
147
148
149     ::close(server_fd);
150
151
152
153     return client_fd;
154
155 }
156
157
158
159 bool load_context_and_keys(CryptoContext<DCRTPoly>& cc,
```

```
                        PublicKey<DCRTPoly>& publicKey) {

    if (!Serial::DeserializeFromFile("dati/cryptocontext.txt", cc,
    ↪  SerType::BINARY)) {

        std::cerr << "[MASTER] Errore caricamento cryptocontext.txt" <<
        ↪  std::endl;

        return false;

    }



    if (!Serial::DeserializeFromFile("dati/key-public.txt", publicKey,
    ↪  SerType::BINARY)) {

        std::cerr << "[MASTER] Errore caricamento key-public.txt" << std::endl;

        return false;

    }



    std::ifstream mkf("dati/key-eval-mult.txt", std::ios::in |
    ↪  std::ios::binary);

    if (!mkf.is_open() || !cc->DeserializeEvalMultKey(mkf, SerType::BINARY)) {

        std::cerr << "[MASTER] Errore caricamento key-eval-mult.txt" <<
        ↪  std::endl;

        return false;

    }



    std::ifstream rotFile("dati/key-eval-rot.txt", std::ios::in |
    ↪  std::ios::binary);

    if (!rotFile.is_open() || !cc->DeserializeEvalAutomorphismKey(rotFile,
    ↪  SerType::BINARY)) {

        std::cerr << "[MASTER] Errore caricamento rotation keys" << std::endl;

        return false;

    }



    return true;
```

```
209 }
210
211
212
213 } // namespace
214
215
216
217 int main() {
218
219     CryptoContext<DCRTPoly> cc;
220
221     PublicKey<DCRTPoly> publicKey;
222
223
224
225     if (!load_context_and_keys(cc, publicKey)) {
226
227         return 1;
228
229     }
230
231
232
233     const int sock_slave = setup_server(config::PORT_TO_MASTER);
234
235
236
237     if (sock_slave < 0) {
238
239         return 1;
240
241     }
242
243
244
245     auto zero_pt = cc->MakeCKKSPackedPlaintext(std::vector<double>{0.0});
246
247     // Rimuoviamo somma_e e inseriamo e_prev_enc (memoria dell'errore al ciclo
    ↪  k-1)
248
249     auto e_prev_enc = cc->Encrypt(publicKey, zero_pt);
250
251     int cycle_count = 0;
252
253
254
255     // Calcoliamo le costanti del PI in forma Velocity:
256
257     // DeltaU = (Kp + Ki*Ts) * e_k - Kp * e_{k-1}
258
259     double k1 = config::KP + (config::KI * config::TS);
260
261     double k2 = config::KP; // Lo teniamo positivo e usiamo EvalSub dopo
262
263
```

```
264
265     while (true) {
266
267         Ciphertext<DCRTPoly> e_enc;
268
269
270
271         if (!recv_ciphertext(sock_slave, e_enc)) {
272
273             std::cerr << "[MASTER] Errore ricezione e_enc" << std::endl;
274
275             break;
276
277         }
278
279
280
281         auto t_start = std::chrono::high_resolution_clock::now();
282
283         ++cycle_count;
284
285         Ciphertext<DCRTPoly> u_enc; // Ora conterrà il DELTA U, non la U totale!
286
287
288
289         if (cycle_count <= 1) {
290
291             u_enc = cc->Encrypt(publicKey, zero_pt);
292
293         } else {
294
295             // Calcolo DeltaU omomorfico in profondità 1 costante!
296
297             // term1 = k1 * e_k
298
299             auto term1 = cc->EvalMult(e_enc, k1);
300
301             // term2 = Kp * e_{k-1}
302
303             auto term2 = cc->EvalMult(e_prev_enc, k2);
304
305
306
307             // DeltaU = term1 - term2
308
309             u_enc = cc->EvalSub(term1, term2);
310
311         }
312
313
314
315         // Aggiorniamo la memoria: l'errore attuale diventa il precedente
316
317         e_prev_enc = e_enc;
318
319
```

```
320
321         if (!send_ciphertext(sock_slave, u_enc)) {
322
323             std::cerr << "[MASTER] Errore invio U_enc" << std::endl;
324
325             break;
326
327         }
328
329
330
331         auto t_end = std::chrono::high_resolution_clock::now();
332
333         const auto duration_us =
334
335             std::chrono::duration_cast<std::chrono::microseconds>(
336
337                 t_end - t_start
338
339             ).count();
340
341
342
343         std::cout << "[MASTER] Ciclo=" << cycle_count
344
345                   << " | KP=" << config::KP
346
347                   << " | KI=" << config::KI
348
349                   << " | tempo=" << duration_us << " us ("
350
351                   << duration_us / 1000.0 << " ms)"
352
353                   << std::endl;
354
355     }
356
357
358
359     ::close(sock_slave);
360
361
362
363     return 0;
364
365 }
```

slave_openfhe.cpp

```
1 // ============================================================================
2 // slave_openfhe.cpp
3 // ----------------------------------------------------------------------------
4 // RUOLO NEL SISTEMA
5 // - Gira sulla Raspberry Pi5.
6 // - È il "nuovo slave" del sistema distribuito a 3 nodi.
```

```
// - Ha la chiave privata e la chiave pubblica.
// - Fa da ponte intelligente tra communicator e master.
//
// COSA FA
// 1) Riceve dal communicator un valore double in chiaro (misura del processo).
// 2) Calcola error, sum_error e delta_error in chiaro.
// 3) Cifra questi stati con OpenFHE.
// 4) Invia i ciphertext al master.
// 5) Riceve U_enc dal master.
// 6) Decripta U_enc.
// 7) Applica saturazione e rimanda il double finale al communicator.
//
// MODIFICA ATTUALE
// - Il riferimento non è più costante.
// - È stata aggiunta una rampa da 0 RPM a 2000 RPM in 5 secondi.
// ============================================================================
#include "openfhe.h"
#include "pke/cryptocontext-ser.h"
#include "scheme/ckksrns/ckksrns-ser.h"
#include "ciphertext-ser.h"
#include "key/key-ser.h"
#include <chrono>
#include <vector>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>

#include "config.hpp"
#include "socket_utils.hpp"
#include "openfhe_bridge.hpp"

#include <algorithm>
#include <cmath>
#include <iostream>
#include <string>
#include <stdio.h>

using namespace lbcrypto;

namespace {

struct ControlState {
    double previous_error = 0.0;
    double sum_error = 0.0;
};


//-------------------------------------------
// PARAMETRI DI RIFERIMENTO A SINUSOIDE
//-------------------------------------------

//double FREQUENCY_REF = 0.2;    // 0.5 Hz = una oscillazione completa ogni 2
↪  secondi
//double AMPLITUDE_RPM = 1000; // Metà della variazione totale (2000 / 2)
//double OFFSET_RPM = 1000.0;    // Punto centrale dell'oscillazione
```

```
62
63
64
65
66
67
68
69
70
71
72
73
74  // ------------------------------------------------------------------------------
75  // Parametri del riferimento a rampa
76  // ------------------------------------------------------------------------------
77  /*constexpr double RAMP_DURATION_S = 5.0;
78  constexpr double RAMP_START_RPM  = 0.0;
79  constexpr double RAMP_END_RPM    = 2000.0;
80
81  double compute_ramp_reference_rpm(double time_s) {
82     if (time_s <= 0.0) {
83          return RAMP_START_RPM;
84      }
85      if (time_s >= RAMP_DURATION_S) {
86          return RAMP_END_RPM;
87      }
88
89      return RAMP_START_RPM +
90             (RAMP_END_RPM - RAMP_START_RPM) * (time_s / RAMP_DURATION_S);
91  }
92
93  */
94  int setup_server_for_communicator(int port) {
95
96      int server_fd = ::socket(AF_INET, SOCK_STREAM, 0);
97      if (server_fd < 0) {
98          std::perror("slave_openfhe: socket server");
99          return -1;
100     }
101
102     int opt = 1;
103     if (::setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) <
        ↪  0) {
104         std::perror("slave_openfhe: setsockopt");
105         ::close(server_fd);
106         return -1;
107     }
108
109     sockaddr_in addr{};
110     addr.sin_family = AF_INET;
111     addr.sin_addr.s_addr = INADDR_ANY;
112     addr.sin_port = htons(port);
113
114     if (::bind(server_fd, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) < 0)
        ↪  {
115         std::perror("slave_openfhe: bind communicator");
```

```
116         ::close(server_fd);
117         return -1;
118     }
119 
120     if (::listen(server_fd, 1) < 0) {
121         std::perror("slave_openfhe: listen communicator");
122         ::close(server_fd);
123         return -1;
124     }
125 
126     std::cout << "[SLAVE] In attesa del communicator sulla porta "
127               << port << "..." << std::endl;
128 
129     const int client_fd = ::accept(server_fd, nullptr, nullptr);
130     if (client_fd < 0) {
131         std::perror("slave_openfhe: accept communicator");
132         ::close(server_fd);
133         return -1;
134     }
135 
136     std::cout << "[SLAVE] Communicator connesso." << std::endl;
137     ::close(server_fd);
138     return client_fd;
139 }
140 
141 int connect_to_master(const char* ip, int port) {
142     int sock = ::socket(AF_INET, SOCK_STREAM, 0);
143     if (sock < 0) {
144         std::perror("slave_openfhe: socket master");
145         return -1;
146     }
147 
148     sockaddr_in addr{};
149     addr.sin_family = AF_INET;
150     addr.sin_port = htons(port);
151 
152     if (::inet_pton(AF_INET, ip, &addr.sin_addr) <= 0) {
153         std::cerr << "[SLAVE] IP master non valido: " << ip << std::endl;
154         ::close(sock);
155         return -1;
156     }
157 
158     std::cout << "[SLAVE] Connessione al master " << ip << ":" << port << "..."
    ↪  << std::endl;
159     while (::connect(sock, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) <
    ↪  0) {
160         std::perror("[SLAVE] connect master fallita, riprovo");
161         ::sleep(2);
162     }
163 
164     std::cout << "[SLAVE] Connesso al master." << std::endl;
165     return sock;
166 }
167 
168 bool load_context_and_keys(CryptoContext<DCRTPoly>& cc,
169                            PublicKey<DCRTPoly>& publicKey,
```

```
                                PrivateKey<DCRTPoly>& privateKey) {
    if (!Serial::DeserializeFromFile("dati/cryptocontext.txt", cc,
    ↪  SerType::JSON)) {
        std::cerr << "[SLAVE] Errore caricamento cryptocontext.txt" <<
        ↪  std::endl;
        return false;
    }
    if (!Serial::DeserializeFromFile("dati/key-public.txt", publicKey,
    ↪  SerType::JSON)) {
        std::cerr << "[SLAVE] Errore caricamento key-public.txt" << std::endl;
        return false;
    }
    if (!Serial::DeserializeFromFile("dati/key-private.txt", privateKey,
    ↪  SerType::JSON)) {
        std::cerr << "[SLAVE] Errore caricamento key-private.txt" << std::endl;
        return false;
    }
    return true;
}

double convert_voltage_to_rpm(double voltage) {
    const double clamped = std::clamp(voltage, 0.0, config::MAX_VOLTAGE_IN);
    return (clamped / config::MAX_VOLTAGE_IN) * config::MAX_RPM_OUT;
}

double saturate_control(double u) {
    return std::clamp(u, config::U_MIN_VOLT, config::U_MAX_VOLT);
}

} // namespace

int main() {
    CryptoContext<DCRTPoly> cc;
    PublicKey<DCRTPoly> publicKey;
    PrivateKey<DCRTPoly> privateKey;

    if (!load_context_and_keys(cc, publicKey, privateKey)) {
        return 1;
    }

    const int sock_com =
    ↪  setup_server_for_communicator(config::PORT_FROM_COMMUNICATOR);
    if (sock_com < 0) {
        return 1;
    }

    const int sock_master = connect_to_master(config::MASTER_IP,
    ↪  config::PORT_TO_MASTER);
    if (sock_master < 0) {
        ::close(sock_com);
        return 1;
    }

    ControlState state;
    double current_time_s = 0.0;

```

```
220     FILE* csv = fopen("tempi.csv", "w");
221     if (!csv) {
222         perror("Errore apertura CSV");
223         ::close(sock_master);
224         ::close(sock_com);
225         return 1;
226     }
227 
228     fprintf(csv, "time_s,ref_rpm,v_meas,RPM,err,int,u\n");
229 
230     while (true) {
231 auto t_start = std::chrono::high_resolution_clock::now();
232 // ------------------------------------------------------------------
233         // 1) Ricezione misura in chiaro dal communicator.
234         // Il communicator manda il valore filtrato in volt.
235         // ------------------------------------------------------------------
236         double measured_voltage = 0.0;
237         if (!recv_pod(sock_com, measured_voltage)) {
238             std::cerr << "[SLAVE] Errore ricezione misura dal communicator" <<
                ↪  std::endl;
239             break;
240         }
241 
242         // ------------------------------------------------------------------
243         // 2) Conversione misura e riferimento a rampa.
244         // ------------------------------------------------------------------
245         const double measured_rpm = convert_voltage_to_rpm(measured_voltage);
246 
247         current_time_s += config::TS;
248         const double reference_rpm = compute_ramp_reference_rpm(current_time_s);
249         // Formula: Ampiezza * sin(2 * pi * f * t) + Offset
250         //double sinusoidal_reference_raw = AMPLITUDE_RPM * sin(2 * M_PI *
            ↪  FREQUENCY_REF * current_time_s);
251         //double sinusoidal_reference = sinusoidal_reference_raw + OFFSET_RPM;
252 
253 //      if (sinusoidal_reference < 0.5)    sinusoidal_reference = 0.5;
254 //      if (sinusoidal_reference > 2000.0) sinusoidal_reference = 2000.0;
255         const double error = reference_rpm - measured_rpm;
256         //const double error = sinusoidal_reference - measured_rpm;
257 
258         std::vector<double>error_vet = {error};
259         // ------------------------------------------------------------------
260         // 3) Cifratura degli stati PID.
261         // ------------------------------------------------------------------
262 
263          const auto e_pt = cc->MakeCKKSPackedPlaintext(error_vet);
264 
265          const auto e_enc = cc->Encrypt(publicKey, e_pt);
266 
267 
268 if (!send_ciphertext(sock_master, e_enc)) {
269 
270     std::cerr << "[SLAVE] Errore invio ciphertext unico al master" << std::endl;
271 
272     break;
273 
```

```
}
        // ------------------------------------------------------------------
        // 4) Ricezione del comando cifrato dal master e decrittazione locale.
        // ------------------------------------------------------------------
        Ciphertext<DCRTPoly> u_enc;
        if (!recv_ciphertext(sock_master, u_enc)) {
            std::cerr << "[SLAVE] Errore ricezione U_enc dal master" <<
            ↪  std::endl;
            break;
        }

        Plaintext u_pt;
        cc->Decrypt(privateKey, u_enc, &u_pt);
        u_pt->SetLength(1);

        double u_value = u_pt->GetCKKSPackedValue()[0].real();
        u_value = saturate_control(u_value);

        // ------------------------------------------------------------------
        // 5) Invio del comando finale in chiaro al communicator.
        // ------------------------------------------------------------------
        if (!send_pod(sock_com, u_value)) {
            std::cerr << "[SLAVE] Errore invio u_value al communicator" <<
            ↪  std::endl;
            break;
        }


auto t_end = std::chrono::high_resolution_clock::now();



auto duration_us =
    std::chrono::duration_cast<std::chrono::microseconds>(t_end -
    ↪  t_start).count();
std::cout << "[SLAVE] Tempo elaborazione ciclo: "
          << duration_us << " us ("
          << duration_us / 1000.0 << " ms)"
          << std::endl;


        std::cout << "[SLAVE] t=" << current_time_s
                  << " | ref=" << reference_rpm
                  << " | V_meas=" << measured_voltage
                  << " | RPM=" << measured_rpm
                  << " | err=" << error
                  << " | u=" << u_value << std::endl;

        fprintf(csv, "%f,%f,%f,%f,%f,%f\n",
                current_time_s,
                reference_rpm,
                measured_voltage,
                measured_rpm,
                error,
                u_value);
        fflush(csv);
```

```
    }
}
```

5.6.2 Sistema di comunicazione

socket_utils.cpp

```
#include "socket_utils.hpp"

#include <cerrno>
#include <cstring>
#include <string>

#include <sys/socket.h>
#include <unistd.h>

bool send_all(int sock, const void* data, std::size_t len) {
    const char* ptr = static_cast<const char*>(data);
    std::size_t total = 0;

    while (total < len) {
        const ssize_t sent = ::send(sock, ptr + total, len - total, 0);
        if (sent <= 0) {
            return false;
        }
        total += static_cast<std::size_t>(sent);
    }
    return true;
}

bool recv_all(int sock, void* data, std::size_t len) {
    char* ptr = static_cast<char*>(data);
    std::size_t total = 0;

    while (total < len) {
        const ssize_t recvd = ::recv(sock, ptr + total, len - total, 0);
        if (recvd <= 0) {
            return false;
        }
        total += static_cast<std::size_t>(recvd);
    }
    return true;
}

std::string last_socket_error() {
```

```
    return std::strerror(errno);
}
```

openfhe_bridge.cpp

```
#include "openfhe_bridge.hpp"
#include "socket_utils.hpp"

#include <arpa/inet.h>
#include <sstream>
#include <string>
#include <vector>

using namespace lbcrypto;

namespace {

template <typename T>
bool send_serialized_object(int sock, const T& obj) {
    std::stringstream ss;

    // BINARY è preferibile a JSON per latenza e volume di traffico.
    Serial::Serialize(obj, ss, SerType::BINARY);
    const std::string data = ss.str();

    const uint32_t len_net = htonl(static_cast<uint32_t>(data.size()));
    if (!send_all(sock, &len_net, sizeof(len_net))) {
        return false;
    }
    return send_all(sock, data.data(), data.size());
}

template <typename T>
bool recv_serialized_object(int sock, T& obj) {
    uint32_t len_net = 0;
    if (!recv_all(sock, &len_net, sizeof(len_net))) {
        return false;
    }

    const uint32_t len = ntohl(len_net);
    std::vector<char> buffer(len);
    if (!recv_all(sock, buffer.data(), buffer.size())) {
        return false;
    }

    std::stringstream ss;
    ss.write(buffer.data(), static_cast<std::streamsize>(buffer.size()));
    Serial::Deserialize(obj, ss, SerType::BINARY);
    return true;
}

} // namespace

bool send_ciphertext(int sock, const Ciphertext<DCRTPoly>& obj) {
    return send_serialized_object(sock, obj);
```

```
}

bool recv_ciphertext(int sock, Ciphertext<DCRTPoly>& obj) {
    return recv_serialized_object(sock, obj);
}

bool send_plaintext(int sock, const Plaintext& obj) {
    return send_serialized_object(sock, obj);
}

bool recv_plaintext(int sock, Plaintext& obj) {
    return recv_serialized_object(sock, obj);
}
```

5.6.3 Sezione di configurazione

config.hpp

```
#pragma once

// ============================================================================
// config.hpp
// ----------------------------------------------------------------------------
// Unico punto in cui personalizzare IP, porte, periodo di campionamento,
// guadagni PID e limiti del sistema.
//
// Architettura finale:
//   Pi3 (communicator)  <----TCP---->  Pi5 (slave_openfhe)
//   Pi5 (slave_openfhe) <----TCP---->  Master (master_openfhe)
//
// NOTA:
// - Il master resta server, come nei file vecchi del progetto.
// - Lo slave OpenFHE fa da server verso il communicator e da client verso il
↪  master.
// - Il communicator resta vicino all'hardware e NON usa OpenFHE.
// ============================================================================

namespace config {

// --------------------------- Rete -------------------------------------------
//inline constexpr const char* SLAVE_LISTEN_IP = "0.0.0.0";  // Pi5 in ascolto
↪  su tutte le interfacce
//inline constexpr int PORT_FROM_COMMUNICATOR = 5000;         // Pi3 -> Pi5

inline constexpr const char* MASTER_IP = "10.10.1.6";      // Cambiare con l'IP
↪  reale del Master
inline constexpr int PORT_TO_MASTER = 4343;                 // Coerente con i
↪  file vecchi

// Sul Pi3, questo deve essere l'IP della Pi5.
// ------------------------- Campionamento ----------------------------------
inline constexpr double TS = 0.07; // 80 ms, coerente col loop che avevi
↪  descritto

// ---------------------------- PID -------------------------------------------
```

```
// Questi guadagni sono il punto di partenza. Se vuoi mantenere pari pari i
// valori dei file vecchi, cambiali qui e ricompila.
inline constexpr double KP = 0.015;
inline constexpr double KI = 0.015; //0.0140
inline constexpr double KD = 0.0;

// ------------------------ Riferimento e limiti -------------------------------
inline constexpr double REFERENCE_RPM = 2000.0; // Riferimento costante di
↪  esempio
inline constexpr double U_MIN_VOLT = 0.0;
inline constexpr double U_MAX_VOLT = 3.3;

// ---------------------- Conversioni / scaling ------------------------------
inline constexpr double MAX_VOLTAGE_IN = 5.0;
inline constexpr double MAX_RPM_OUT = 6000.0;

} // namespace config
```

5.7 Algoritmo RSA: confronto con FHE

Questo paragrafo ha il solo obiettivo di evidenziare le differenze operative che ci sono tra la crittografia asimmetrica RSA, discussa nel Capitolo 5, e la FHE.
In particolare, si osserverà l'assenza di addizione omomorfica e l'esposizione del dato in chiaro sul nodo Master.
Si tenga in considerazione che le porzioni di codice che seguono provengono dal lavoro svolto dal collega Francesco Iurilli, il quale ha precedentemente analizzato nel dettaglio e utilizzato questo tipo di crittografia nel sistema.

5.7.1 Cifratura degli stati sullo Slave

```
//slave_1024.c
double error = (sinusoidal_reference - simulated_rpm_to_master) *
↪  1000000.0;
sum_error += error * SYNC_INTERVAL_S;

char e_hex[HEXBUF], S_hex[HEXBUF];

rsaossl2_encrypt_hex(rsaossl2_encode_signed(error),    e_hex,
↪  sizeof(e_hex));
rsaossl2_encrypt_hex(rsaossl2_encode_signed(sum_error), S_hex,
↪  sizeof(S_hex));
```

Si osservi che, a differenza della cifratura CKKS, in cui il vettore di `double` viene cifrato direttamente tramite `MakeCKKSPackedPlaintext`, RSA lavora solo ed esclusivamente su numeri interi.
Nello specifico, `rsaossl2_encode_signed` converte il `double` in un `uint64_t` usando l'operazione del complemento a due.
Solo a questo punto avviene la cifratura $c = m^e \bmod n$.

Possiamo definirlo un passaggio intermedio: passaggio che non esiste nello schema CKKS, poiché come proprietà intrinseca gestisce numeri reali.
Si faccia riferimento al punto trattato nel Paragrafo 4.2, il quale parla proprio dell'inefficienza di RSA con valori frazionari.

5.7.2 Nessuna addizione omomorfica svolta sul Master

```
// master_1024.c
rsaossl2_multiply_hex(e_hex, Kp_hex, P_hex, sizeof(P_hex));
rsaossl2_multiply_hex(S_hex, Ki_hex, I_hex, sizeof(I_hex));

snprintf(buffer, sizeof(buffer), "%s %s", P_hex, I_hex);
send(new_socket, buffer, strlen(buffer), 0);
```

L'unica operazione omomorfica che RSA concede è la moltiplicazione.
Il Master calcola $P_{enc} = Enc(e \cdot K_P)$ e $I_{enc} = Enc(\Sigma e \cdot K_I)$ separatamente, basandosi sulla proprietà $Enc(a) \cdot Enc(b) \equiv Enc(a \cdot b) \pmod{n}$.
Tuttavia, non può sommarli: vengono spediti distinti allo Slave (`P_hex` e `I_hex`). Quindi, l'addizione $u = P + I$ richiesta dalla legge di controllo non è eseguibile nel dominio cifrato RSA.

In FHE, invece, l'operazione `cc->EvalSub(term1, term2)`esegue l'intera combinazione lineare del PI interamente sul Master, senza mai spezzare il risultato in componenti da ricomporre, come si è osservato nel Paragrafo 5.4.

5.7.3 Il dato torna in chiaro sullo Slave

```
// slave_1024.c
uint64_t P_i, I_i;
rsaossl2_decrypt_hex(P_hex, &P_i);
rsaossl2_decrypt_hex(I_hex, &I_i);

double P = rsaossl2_decode_signed(P_i) / 100000000000.0;
double I = rsaossl2_decode_signed(I_i) / 1000000000000.0;

double u = P + I;   // addizione eseguita in chiaro
if (u < 0.0) u = 0.0;
if (u > 3.3) u = 3.3;
```

In questo blocco di codice si vuole fare riferimento al confronto osservato in Fig. 4.4. Parlando dell'addizione `u = P + I`, quest'ultima avviene dopo la decifratura, quindi nel dominio in chiaro.

Ciò significa che, nonostante il Master non veda mai e o Σe, lo Slave è costretto a esporre i termini Proporzionale e Integrale separatamente per poterli sommare.

Per costruzione, sappiamo che in FHE questa informazione non emerge in chiaro

in nessun nodo della rete: la combinazione lineare avviene già cifrata sul Master e lo Slave decifra direttamente il segnale di controllo $u(t)$ necessario al DAC.

Capitolo 6

Conclusioni

6.1 Risultati Sperimentali

Al fine di analizzare in modo quantitativo le performance del sistema di controllo cifrato, fisicamente implementato, durante la fase di esecuzione è stato misurato il tempo necessario per completare l'intero ciclo di controllo omomorfico, considerato come il percorso di comunicazione dallo Slave al Master, e successivamente ritorno allo Slave.
In particolare, sono stati presi in considerazione i seguenti passaggi: la raccolta e la cifratura dell'errore, la trasmissione attraverso la rete, l'elaborazione omomorfica della legge di controllo PI in forma incrementale sul nodo Master, approfondita nel Paragrafo 5.4, la ritrasmissione del ciphertext e la decifratura conclusiva sullo Slave.

Il campione raccolto per la prima valutazione a regime è costituito da 665 cicli di controllo consecutivi, per una durata complessiva di acquisizione di circa 46.6 secondi, con periodo di campionamento impostato a $T_s = 70$ ms, definito unitamente agli altri parametri nel file `config.hpp`. In questa fase preliminare di test, l'infrastruttura di rete locale era gestita da un apparato switch Netgear di vecchia generazione.

6.1.1 Latenza del ciclo di controllo omomorfico

L'immagine mostrata in Figura 6.1 illustra l'andamento del tempo di esecuzione registrato per ogni ciclo di controllo, mettendolo a confronto con il periodo di campionamento T_s. Invece, la distribuzione statistica è presentata in Figura 6.2, mentre la Tabella 6.1 riassume i principali parametri estratti dai log sperimentali, quali `tempi.csv` per il controllo FHE e `timing_Master.csv` e `timing_Slave.csv` per il ciclo in chiaro.

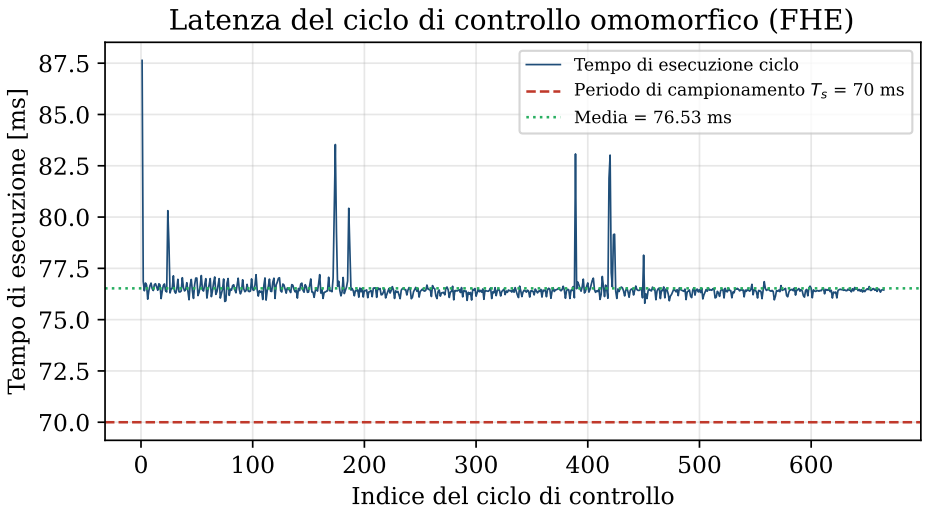

Figura 6.1: Tempo di esecuzione del ciclo di controllo omomorfico in funzione dell'indice di ciclo, confrontato con il periodo di campionamento $T_s = 70$ ms. *Fonte: elaborazione personale*

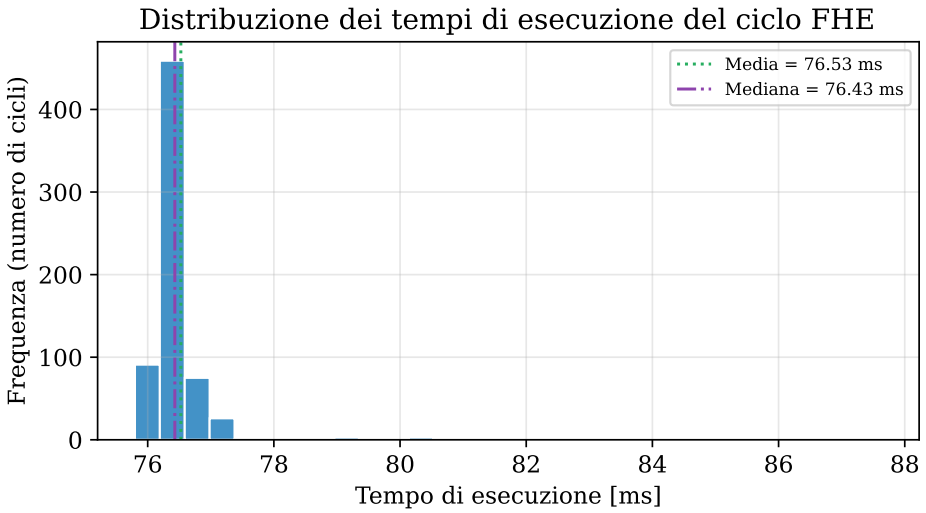

Figura 6.2: Distribuzione dei tempi di esecuzione del ciclo di controllo omomorfico su 665 campioni. *Fonte: elaborazione personale*

Tabella 6.1: Statistiche riassuntive dei tempi di esecuzione del ciclo di controllo omomorfico (Slave–Master–Slave) con lo switch Netgear preliminare. *Fonte: elaborazione personale*

Grandezza	**Valore**
Numero di cicli campionati	665
Durata totale acquisizione	46.55 s
Periodo di campionamento nominale T_s	70 ms
Tempo di esecuzione medio	76.526 ms
Deviazione standard	0.777 ms
Tempo di esecuzione mediano	76.430 ms
Tempo di esecuzione minimo	75.798 ms
Tempo di esecuzione massimo	87.636 ms
Cicli con overhead rispetto a T_s	665 / 665 (100%)
Overhead medio rispetto a T_s	+6.53 ms (+9.3%)

Dai dati presentati nella tabella, emerge che il tempo medio necessario per completare un ciclo si attesta intorno ai $76,5$ ms.
Inoltre, la deviazione standard, contenuta e pari a circa $0,78$ ms, indica un comportamento costante e ripetibile del sistema.

I rari picchi visibili nel grafico, che arrivano fino a un massimo di $87,6$ ms, sono principalmente attribuibili al primo ciclo di avvio e allocazione del sistema, noto come "cold start". Altri picchi possono essere ricondotti a latenze occasionali della rete o a problemi di scheduling del sistema operativo, risultando comunque eventi isolati.

Un aspetto significativo è emerso in prima analisi: il tempo medio di esecuzione ha superato il periodo di campionamento nominale di $T_s = 70$ ms, presentando un overhead medio di circa 6,5 ms ($+9,3\%$).
Questa evidenza consente di misurare il costo computazionale e di rete associato alla FHE, come discusso nella sezione teorica del Capitolo 4. La cifratura CKKS e, in particolare, la trasmissione bidirezionale di ciphertext di grandi dimensioni, saturavano le capacità dello switch Netgear utilizzato nella fase preliminare.
Nonostante ciò, il sistema si è dimostrato utilizzabile per il controllo del motore in esame. Tuttavia, lo scostamento rappresentava un vincolo per un controllo di tipo "hard real-time".

6.1.2 Ottimizzazione di rete e confronto in chiaro

Affinché si comprenda meglio lo sforzo che il sistema esegue per lavorare nel dominio cifrato, questo paragrafo si prefigge l'obiettivo di confrontare i risultati in FHE con il processo in chiaro, che utilizza gli stessi algoritmi di controllo ma invia dati senza OpenFHE.
Successivamente, si noterà il bisogno di apportare ottimizzazioni all'architettura hardware.

Nel sistema in chiaro, il payload di rete ha un peso trascurabile. Si utilizzano variabili di tipo `double`, ciascuna da 8 byte, per cui irrisorie dal punto di vista del loop di controllo. Infatti, il ciclo si completa in soli 20.77 ms. Si noti che questo risultato rimane costante, quindi indipendente dagli apparati di rete impiegati.

La situazione cresce di complessità quando si ricorre alla crittografia: il dato cifrato si trasforma in una coppia di polinomi, occupando vari KB.
Di conseguenza, inviare un payload con questo peso genera un collo di bottiglia a causa del vecchio switch Netgear, portando a un tempo di ciclo complessivo di 76.53 ms.

Si è quindi considerata la sostituzione dello switch Netgear con lo switch professionale HP ProCurve per risolvere questo problema.
Dopo averlo installato, si è osservato un netto miglioramento nell'instradamento dei pacchetti TCP: le latenze sul canale sono state eliminate e i tempi di comunicazione si sono ridotti notevolmente.

Tramite il nuovo switch, il tempo di loop complessivo del sistema FHE si è abbassato notevolmente a **40.00 ms**.

Nella Tabella 6.2 e nel grafico in Fig. 6.3 viene riassunto l'impatto dell'infrastruttura hardware e della FHE sul tempo di loop.

Parametro	**In Chiaro**	**FHE (Vecchio Switch Netgear)**	**FHE (Nuovo Switch HP)**
Carico di Rete	Minimo (8 Bytes)	Elevato (Kilobytes)	Elevato ma Ottimizzato
Tempo Cifratura/Decifratura	0.00 ms	Integrato nel loop	Integrato nel loop
Durata Totale del Ciclo	20.77 ms	76.53 ms	**40.00 ms**
Rispetto del Vincolo $T_s = 70$ ms	Sì	No (Overhead +9.3%)	**Sì**

Tabella 6.2: Confronto prestazionale: si noti come l'implementazione dello switch HP garantisca che il sistema rispetti il vincolo real-time.

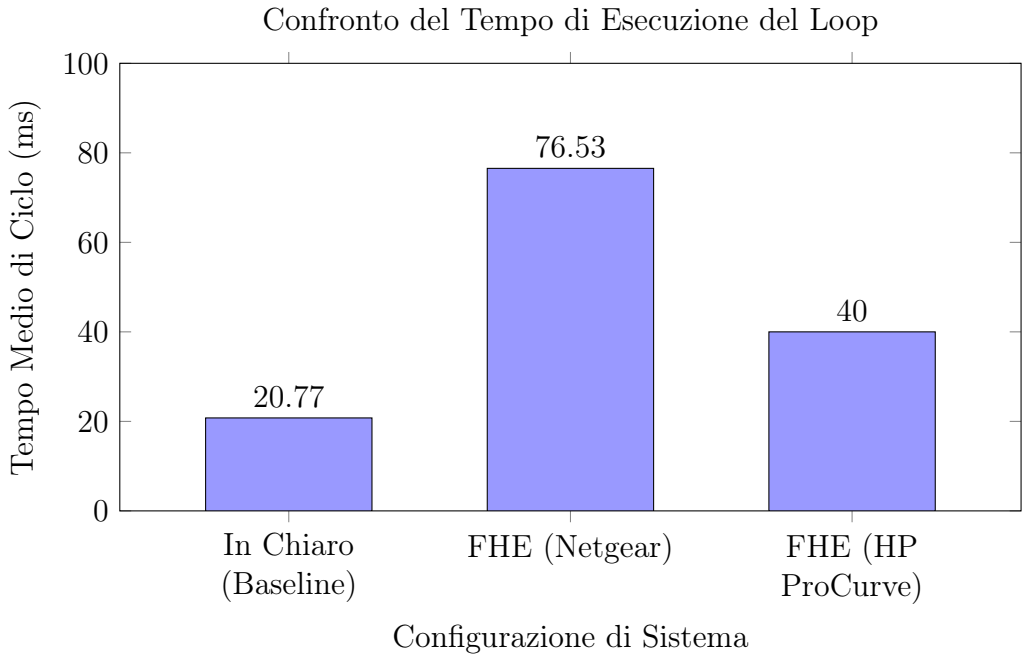

Figura 6.3: Evoluzione del Loop di Controllo: dal dominio in chiaro, all'implementazione FHE limitata dallo switch Netgear, fino all'ottimizzazione con lo switch HP.

Raggiungere i 40 ms si traduce in un abbassamento del tempo di esecuzione al di sotto della soglia dei 70 ms.
In questo modo, il sistema FHE riesce a rientrare nel periodo di campionamento T_s, garantendo l'esecuzione del controllo omomorfico rispettando i vincoli di tempo.

Analizzando queste tempistiche e confrontandole con le vulnerabilità dell'algoritmo RSA esposte nel Paragrafo 5.7, si può giungere alla conclusione.

In questa fase il sistema FHE raggiunge l'obiettivo della tesi: risolto il problema legato alle prestazioni con lo switch HP, la computazione è avvenuta totalmente su dati cifrati.
Il nodo Master ha eseguito le combinazioni lineari dell'algoritmo PI lavorando solo ed esclusivamente su dati cifrati, proteggendo i dati in uso ed eliminando il rischio di intercettazione.

Per un'analisi più approfondita dei controllori utilizzati all'interno del sistema, quali PID e MPC, si rimanda agli elaborati delle mie colleghe Rania Zouaghi e Lucrezia Bartoletti.

6.2 Possibili sviluppi futuri

Il lavoro svolto e i risultati ottenuti dimostrano che il controllo completamente omomorfico è realmente realizzabile.
Si ponga l'attenzione sul tempo di loop ottenuto in ultima analisi, pari a 40 ms, il

quale si dimostra ampiamente al di sotto dei 70 ms imposti dal tempo di campionamento.
Tenendo in considerazione questa caratteristica, si potrà pensare ad alcuni approcci per sviluppare ulteriormente il sistema, dal punto di vista dell'algoritmo e da quello architetturale:

- **Controllo Multi-Asse tramite SIMD:** Poiché si è sottolineata la caratteristica della codifica a slot dello schema CKKS, ricordando che lo stesso permette di cifrare fino a $N/2$ valori all'interno di un singolo ciphertext, operando in modalità SIMD, acronimo di Single Instruction, Multiple Data.
  Attualmente, come citato in precedenza, il sistema sfrutta un solo slot per l'errore.

  Un possibile sviluppo futuro potrebbe essere quello di effettuare letture su più azionamenti, o sensori aggiuntivi, in un unico array cifrato.
  Questo garantirebbe, ad esempio, che il nodo Master calcoli simultaneamente le leggi di controllo per un ARM robotico multi-asse, senza costi aggiuntivi in termini di latenza.

- **Ottimizzazione per processori ARM:** L'implementazione attuale esegue la crittografia sulla CPU Broadcom delle Raspberry Pi 5.
  Si potrebbe ottimizzare la libreria OpenFHE anche per architetture ARM, in particolare sfruttando le sue estensioni vettoriali, come ARM NEON/SVE, specifiche dell'architettura AArch64.
  Tramite questa ottimizzazione si potrebbero abbattere ulteriormente i tempi di cifratura, e di conseguenza di valutazione dei polinomi, senza hardware aggiuntivo.

Bibliografia

[1] *Piedinatura Pi 5*. Disponibile all'indirizzo: https://pinout.xyz/

[2] Microsoft, *Installazione WSL*. Disponibile all'indirizzo: https://learn.microsoft.com/en-us/windows/wsl/install

[3] *Documentazione ufficiale RPI-LGPIO*. Disponibile all'indirizzo: https://rpi-lgpio.readthedocs.io/en/latest/install.html

[4] *Documentazione chip RP1*. Disponibile all'indirizzo: https://www.raspberrypi.com/documentation/computers/io-controllers.html

[5] *Paper ufficiale Pi*. Disponibile all'indirizzo: https://pip-assets.raspberrypi.com/categories/685-whitepapers-app-notes/documents/RP-006553-WP/A-history-of-GPIO-usage-on-Raspberry-Pi-devices-and-current-best-practices

[6] *FHE di Craig Gentry*. Disponibile all'indirizzo: https://github.com/user-attachments/files/19464528/FHE.org.Keynote.Craig.pdf

[7] Bartoletti L., Gonzato C., Zouaghi R., *Analisi e implementazione di sistemi di controllo crittografati per azionamenti elettrici*. Disponibile all'indirizzo: https://canva.link/icak6loy2ogk2ai

[8] *Documentazione ufficiale libreria OpenFHE*. Disponibile all'indirizzo: https://openfhe-development.readthedocs.io/en/latest/sphinx_rsts/intro/installation/linux.html

Ringraziamenti

In primo luogo, desidero ringraziare il mio relatore, il Professor Gianluca Ippoliti che ha saputo indirizzarmi brillantemente e senza il quale non avrei realizzato questo lavoro sperimentale. Lo ringrazio non solo per la fiducia accordatami nell'affidarmi questo progetto, ma soprattutto per la pazienza mostrata e per i preziosi consigli tecnici che si sono rivelati fondamentali.
Ringrazio anche i correlatori Prof. Giuseppe Orlando e Prof. Marco Baldi.

Inoltre, ringrazio la mia famiglia e tutte le persone care che mi sono state vicine e che mi hanno supportato in questi anni di duro e faticoso lavoro.
In particolar modo ci tengo a ringraziare mia mamma Katjuscia, mio fratello Daniel, le mie zie Roberta e Ilenia e i miei nonni Rodolfo e Gabriella che hanno accompagnato il mio percorso accademico costantemente. Nei momenti in cui la mia determinazione vacillava, erano sempre presenti per aiutarmi. Senza di loro, non sarei mai riuscito in quest'impresa.

Infine, vorrei ringraziare le mie colleghe Lucrezia Bartoletti e Rania Zouaghi per tutti i mesi in cui siamo letteralmente impazziti in laboratorio per far funzionare tutto. Senza la loro preziosa collaborazione, questo progetto non sarebbe stato lo stesso.