



UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Curriculum Sicurezza Informatica

TESI DI LAUREA

Securing CAN protocol with Post-Quantum Cryptography

RELATORE

Prof. Arcangelo Castiglione

Università degli studi di Salerno

CANDIDATO

Lorenzo Criscuolo

Matricola: 0522501268

È sapiente solo chi sa di non sapere, non chi s'illude di sapere e ignora così perfino la sua stessa ignoranza. (Socrate)

Sommario

Il mondo automobilistico è sempre più all'avanguardia grazie all'introduzione dell'elettronica all'interno delle auto. Infatti, con l'introduzione di *ECU*, è possibile garantire elevati standard di sicurezza, minimizzare i consumi di carburante e garantire un elevato comfort a guidatore e passeggeri. Tuttavia, per realizzare dei sistemi in grado di fornire queste caratteristiche, è necessario che le *ECU* siano in grado di comunicare tra loro in maniera affidabile e veloce. Per questa ragione sono stati realizzati dei protocolli che hanno lo scopo di permettere la comunicazione tra le *ECU* e che garantiscono diversi livelli di affidabilità e prestazioni, in maniera tale da adattarsi ad ogni contesto. Anche se i protocolli più diffusi svolgono i propri compiti garantendo diversi standard di affidabilità e prestazioni, lo stesso non vale per la sicurezza rispetto ad attacchi passivi e attivi, infatti, nessuno di questi garantisce un livello di sicurezza adeguato agli standard attuali. Prendendo ad esempio il protocollo *CAN*, questo è vulnerabile ad attacchi di ascolto passivo, attacchi di iniezione e modifica dei messaggi e attacchi di tipo *Denial of Service*, poiché non viene utilizzato né un meccanismo di cifratura né un meccanismo di autenticazione dei nodi connessi. Questo, tuttavia, è dovuto al fatto che altrimenti non sarebbe stato possibile garantire le prestazioni necessarie per gestire sistemi molto complessi come quelli **ADAS**. Dal momento che non vengono prese precauzioni a livello di protocollo, quello che si può pensare, sia per questioni di semplicità che per retrocompatibilità, è di aggiungere uno strato di cifratura direttamente sull'applicativo senza modificare il protocollo, in modo tale da far viaggiare solo messaggi cifrati invece che messaggi in chiaro. Il modo migliore per introdurre questo strato è tramite un cfrario ibrido, il quale permette di far accordare i nodi su una chiave di sessione comune e di contrastare le intercettazioni. Per realizzare il cfrario ibrido si possono usare degli algoritmi **Post-Quantum**, grazie ai quali si riesce a garantire protezione anche contro computer quantistici ma, ad ogni modo, è importante valutare attentamente le prestazioni di *CAN* con la modifica, in quanto un ritardo troppo alto è inaccettabile e non permette una corretta gestione dei sistemi dell'automobile.

Indice

Indice	ii
Elenco delle figure	v
Elenco delle tabelle	vii
Elenco dei codici	viii
1 Introduzione	1
1.1 Introduzione al mondo <i>Automotive</i>	1
1.2 Introduzione al problema	3
1.3 Obiettivi del lavoro svolto	5
1.4 Struttura della tesi	5
2 Protocolli Automotive	6
2.1 CAN	6
2.1.1 Caratteristiche del protocollo	6
2.1.2 Struttura dei messaggi	8
2.1.3 <i>Bit stuffing</i>	10
2.1.4 Tipologie di messaggi	11
2.1.5 Varianti del protocollo	14
2.2 FlexRay	16
2.2.1 Caratteristiche del protocollo	17
2.2.2 Ciclo di comunicazione	19

2.2.3	Sincronizzazione del Clock	22
2.2.4	<i>Wakeup</i> e <i>Startup</i>	23
2.2.5	Struttura di un messaggio	24
2.2.6	Topologie supportate	25
2.3	LIN	27
2.3.1	Caratteristiche del protocollo	27
2.3.2	Struttura dei messaggi	28
2.3.3	Tipologie di messaggi	30
2.3.4	<i>Wakeup & Sleep</i>	31
2.4	Confronto tra i tre protocolli	32
2.4.1	<i>LIN</i>	32
2.4.2	<i>CAN</i>	32
2.4.3	<i>FlexRay</i>	33
2.4.4	Sicurezza nei protocolli	33
3	Crittografia Post-Quantum	34
3.1	Problema dei computer quantistici	34
3.2	<i>KEM</i>	35
3.3	CRYSTALS-kyber	36
3.3.1	Sicurezza del cfrario	36
3.3.2	Problema di riferimento	38
3.4	Advanced Encryption Standard	39
3.4.1	Struttura di <i>AES</i>	40
3.4.2	Modalità di <i>AES</i>	43
3.4.3	Sicurezza del cfrario	46
4	Soluzione Proposta	48
4.1	Scelta del protocollo	48
4.2	Problematiche di sicurezza riscontrate	49
4.3	Soluzione proposta	49
4.3.1	Scelta degli algoritmi di cifratura	50
4.3.2	Funzionamento della soluzione	51
5	Implementazione	53
5.1	Sistema realizzato	53

5.1.1	Protocollo realizzato	55
5.1.2	Struttura del codice	56
5.2	Librerie utilizzate	57
5.2.1	CAN-utils	57
5.2.2	Kyber	58
5.2.3	OpenSSL	59
5.3	Difficoltà riscontrate	60
5.3.1	Introduzione di CAN-utils	60
5.3.2	Utilizzo delle <i>pipe</i>	61
5.3.3	Chiusura della pipe	63
5.4	Risultati ottenuti	65
5.4.1	Tempi per la generazione delle chiavi	65
5.4.2	Tempi per lo scambio di messaggi	65
6	Conclusioni	67
6.1	Considerazioni finali	67
6.1.1	Sicurezza garantita	67
6.1.2	Utilizzo di versioni ottimizzate dei cifrari	68
6.1.3	Ottimizzazione del protocollo	68
6.2	Sviluppi Futuri	69
6.2.1	Utilizzo di tecniche di cifratura <i>Lightweight</i>	69
6.2.2	Utilizzo in un contesto V2X	69
6.2.3	Aggiunta di un'autorità di certificazione	71
Ringraziamenti		73
Bibliografia		75

Elenco delle figure

2.1	Diagramma di flusso di un algoritmo CSMA/CD con ritrasmissione	7
2.2	Differenza di cablaggi con e senza <i>CAN</i>	8
2.3	Struttura dei messaggi <i>CAN standard</i> ed estesi	9
2.4	Differenza tra codifica NRZ e RZ , entrambe unipolari	11
2.5	Modalità operative con relative transizioni	13
2.6	Rilevazione e gestione degli errori in <i>CAN</i>	14
2.7	Differenze tra <i>CAN FD</i> e <i>CAN</i>	16
2.8	Struttura tipica di una <i>ECU</i> che utilizza <i>FlexRay</i>	18
2.9	Suddivisione del ciclo di comunicazione	20
2.10	Struttura di un frame <i>FlexRay</i>	24
2.11	Principali topologie di rete supportate da <i>FlexRay</i>	26
2.12	Rete <i>LIN</i> collegata ad una backbone <i>CAN</i>	28
2.13	Struttura dei frame <i>LIN</i>	29
2.14	Tipologie di messaggi in <i>LIN</i>	30
3.1	Esempio di sistema senza errori (sinistra) e lo stesso ma con l'aggiunta di errori (destra)	38
3.2	Alcuni esempi di reticolli	39
3.3	Esempio di problema <i>CVP</i>	39
3.4	Esempio di S-box e P-box	41
3.5	Schema dell'operazione <i>SubBytes</i>	42
3.6	Schema dell'operazione <i>ShiftRows</i>	42

3.7	Schema dell'operazione <i>MixColumns</i>	43
3.8	Schema dell'operazione <i>AddRoundKey</i>	43
3.9	Problema con l'utilizzo della modalità <i>ECB</i>	44
3.10	Modalità operative di <i>AES</i>	45
4.1	Schema di <i>CAN</i> senza e con l'aggiunta di un cifrario ibrido	50
4.2	Sequence Diagram del protocollo proposto	52
5.1	Codice sorgente del file <code>can.c</code>	54
5.2	Utilizzo delle funzioni di Kyber	59
5.3	Implementazione del cifrario AES-256-CTR	60
6.1	Schema del protocollo in un contesto V2X	70
6.2	Schema del protocollo in un contesto V2X con <i>CA</i>	70
6.3	Rete <i>CAN</i> con protezione del payload e nodo <i>CA</i>	71

Elenco delle tabelle

2.1 Assegnazione di default delle lunghezze	30
2.2 Confronto tra i tre protocolli	32
3.1 Numero di round predefinito in <i>AES</i>	41
3.2 Confronto tra le varie modalità operative	46
3.3 Sicurezza offerta da <i>AES</i> per computer classici e quantistici	47
5.1 Tempi medi rilevati	66

Elenco dei codici

5.1 Istruzioni per il Linker	54
5.2 Istruzioni per il Loader	54
5.3 Utilizzo di CAN-utils tramite linea di comando	61
5.4 Creazione di una pipe	62
5.5 Lettura dalla pipe	62
5.6 Operazioni necessarie per chiudere la <i>pipe</i>	64

CAPITOLO 1

Introduzione

In questo capitolo verrà introdotto il mondo dell'automotive con relative problematiche di sicurezza riscontrate e verrà illustrata la composizione del presente elaborato

1.1 Introduzione al mondo *Automotive*

Oggiorno il settore automobilistico è sempre più in crescita e sempre più innovativo, grazie al lavoro continuo di ingegneri e ricercatori con lo scopo di rendere i motori sempre più efficienti ed ecosostenibili e rendere la guida dei veicoli sempre più confortevole e sicura. Uno dei motivi per cui è stato possibile raggiungere questi traguardi è stata l'introduzione di *dispositivi elettronici* (anche definiti *sistemi embedded*) all'interno delle automobili [3], grazie ai quali è possibile realizzare moltissimi controlli e assistenze alla guida (alcuni in maniera del tutto automatica) ed è anche possibile garantire un comfort più elevato a guidatore e passeggeri.

I componenti elettronici che si occupano del controllo di tutti questi sistemi sono chiamati **centraline** o **ECU** (*Electronic Control Unit*), che saranno più o meno sofisticate in base allo scopo a cui devono assolvere e al sistema che devono gestire. Quest'ultime sono le parti più importanti di un singolo sistema e in media, in un'automobile moderna, se ne possono trovare dalle 40 alle 100 ECU installate all'interno [3].

Tutte queste ECU, in base al ruolo che ricoprono e al sistema che controllano, di solito vengono raggruppate in 4 sottosistemi:

1. Propulsione (**powertrain**), ovvero la gestione di motore, trasmissione, ruote, ecc. Le *ECU* in questo sottosistema di solito sono responsabili del controllo dell'iniezione di carburante, tracciamento e ottimizzazione di coppia motrice, flusso dell'olio, pressione interna dei cilindri, emissioni, ecc.
2. Telaio (**Chassis**), ovvero la gestione dell'impianto frenante, delle sospensioni e dello sterzo. In questo sottosistema le *ECU* sono responsabili della gestione di meccanismi come *ABS* (Anti-lock Braking System), frenata assistita, distribuzione della forza frenante (*EBD* o Electronic Brakeforce Distribution), ecc.
3. Abitacolo (**Body**), ovvero il controllo di componenti per il comfort dei passeggeri e del guidatore, come il climatizzatore, il cruise-control, i fari automatici, la regolazione elettronica dei sedili, ecc.
4. **Infotainment**, ovvero un'interfaccia che facilita l'interazione tra persone e automobile. Alcune delle funzionalità che di solito offre un sistema di infotainment sono la connessione WiFi/Bluetooth con smartphone per la riproduzione di contenuti o, con sistemi molto più avanzati, una diagnostica in tempo reale dell'automobile (anche da remoto) grazie all'intercomunicazione con gli altri sottosistemi. [31]

Per realizzare tutti questi sistemi, ovviamente, tutte le *ECU* non possono lavorare in maniera indipendente ma hanno bisogno di comunicare tra di loro per ottenere ed inviare informazioni utili. A questo proposito, sono stati ideati vari protocolli di comunicazione, che differiscono in prestazioni e tipologia di mezzo di comunicazione (*wired* o *wireless*) in base alle esigenze, per permettere l'interconnessione di più *ECU*. Alcuni dei protocolli *wired* più diffusi sono:

- **CAN** (Controller Area Network), il più diffuso nei sistemi **powertrain**, **chassis** e **body** per via della sua tolleranza agli errori, per il suo basso costo e per la sua semplicità di realizzazione.
- **FlexRay**, anch'esso diffuso nei sistemi **powertrain**, **chassis** e **body** come valida alternativa di *CAN* per via della sua affidabilità e della sua larghezza di banda più elevata. Grazie alla sua affidabilità e alle sue prestazioni, può essere utilizzato in sistemi avanzati come *brake-by-wire*, *Adaptive Cruise-control*, ecc.
- **LIN** (Local Interconnect Network), molto diffuso nei sistemi **body** dove le prestazioni e la resilienza di *CAN* non sono necessarie. Di solito è utilizzato in sistemi come

l’apertura e chiusura elettronica delle porte e finestrini, la regolazione automatica della temperatura, l’accensione automatica delle luci, ecc; [38]

- **MOST** (Media Oriented System Transport), protocollo molto diffuso nei sistemi **infotainment**. Permette il raggiungimento di una larghezza di banda di 50 Mb/s e, con una topologia ad anello, è in grado di collegare fino a 64 dispositivi MOST. Tuttavia, nonostante le sue prestazioni, il costo di realizzazione di una rete MOST è estremamente dispendioso ed è quindi utilizzato solo per realizzare collegamenti video e, in generale, per collegare telecamere installate nell’automobile. [31]

1.2 Introduzione al problema

Tutti questi protocolli focalizzano la propria attenzione sulla resilienza e sulle prestazioni, con il fine di garantire il corretto funzionamento di tutto l’equipaggiamento e di garantire l’incolumità (per quanto possibile) del guidatore e dei passeggeri. Inoltre, la semplicità di alcuni di questi protocolli permette anche di avere cablaggi più semplici e di utilizzare tecnologie meno costose per sistemi non **safety-critical**, abbattendo drasticamente i costi di produzione di un’automobile. Infatti, sicuramente non è necessario utilizzare un protocollo come *MOST*, il quale mette a disposizione larghezza di banda molto elevata e latenze molto basse, per gestire il climatizzatore ma ha molto più senso utilizzare un protocollo con meno prestazioni come *LIN*, il quale richiede costi molto più bassi.

Tuttavia, al fine di garantire queste caratteristiche (per ovvie ragioni), nel tempo sono stati trascurati quasi del tutto gli aspetti riguardo la **sicurezza** dei protocolli, lasciando spazio solo ed esclusivamente alla **safety**. Questo, purtroppo, non per negligenza da parte di chi li ha realizzati ma perchè nel periodo in cui sono nati la **security** non era un aspetto importante da tenere in considerazione nel mondo *Automotive*. Ad esempio, il protocollo *CAN* è nato negli anni ‘80 e inizialmente la rete *CAN* realizzata all’interno delle automobili doveva essere completamente isolata dal mondo esterno, per cui non era necessario prendere provvedimenti a riguardo. Però, per via dei cambiamenti nell’industria automobilistica che sono avvenuti nel tempo, si è cominciato a fornire l’accesso alla rete agli utenti finali (e quindi a molte delle *ECU* installate) per fini prevalentemente diagnostici, come ad esempio l’utilizzo di *OBD* (On-Board Diagnostic) per rilevare con molta facilità eventuali guasti all’automobile, per rilevare consumi anomali, ecc. [10]

Il problema di fondo di questi protocolli, quindi, è la mancanza di meccanismi di autenticazione e cifratura che impediscano ad un attaccante di inserirsi nella rete ed effettuare attacchi.

Sebbene in molti casi l'attaccante deve avere accesso fisico all'automobile per effettuare un attacco, in uno scenario V2X (Vehicle To Everything) dove l'automobile è connessa ad altre auto o addirittura ad Internet, un attaccante potrebbe essere in grado di lanciare attacchi comodamente da remoto senza nemmeno un accesso fisico all'automobile, sollevando quindi la necessità di correre ai ripari. Prendendo come riferimento il protocollo CAN, gli attacchi possono essere suddivisi in tre categorie:

- **Eavesdropping:** vista l'assenza di meccanismi di cifratura, l'attaccante si mette in ascolto passivo sulla rete per recuperare informazioni utili ad identificare i passeggeri (con conseguente violazione della privacy) analizzando eventuali comunicazioni con smartphone connessi o analizzando, ad esempio, il sensore collegato al pedale del freno. [10]. È possibile anche utilizzare questo tipo di attacchi per individuare gli identificativi dei messaggi utilizzati nello specifico modello e revisione dell'automobile per poi realizzare un attacco attivo.
- **Data Insertion:** vista l'assenza di meccanismi di autenticazione, l'attaccante può iniettare nella rete dei pacchetti CAN non autorizzati alterando il corretto funzionamento della strumentazione oppure generando comportamenti inattesi. È possibile mostrare dati del motore errati (RPM non reali), livello di carburante errato o una velocità alterata, è possibile causare l'accensione o lo spegnimento del motore, apertura e chiusura delle portiere e, con auto più moderne, è possibile persino attivare i freni senza la pressione del pedale o, peggio, prevenire l'attivazione dei freni alla pressione del pedale.
- **Denial of Service:** l'attaccante invia pacchetti malformati o incompleti per cercare di saturare la rete o solo alcune ECU. Forgiando pacchetti errati o alterando i pacchetti inviati da altri, è possibile persino sfruttare un meccanismo di CAN dove una ECU che invia molti pacchetti errati viene completamente disattivata per evitare che disturbino la rete. [10]

Sfortunatamente, si può notare come l'assenza di ogni tipo di requisito di sicurezza ha portato ad un protocollo che è diventato uno standard nel mondo *Automotive* ma che è attaccabile in maniera estremamente semplice. In letteratura sono stati effettuati vari tentativi per mettere in sicurezza il protocollo ma, essendo che non è stato pensato per essere sicuro, i metodi proposti hanno introdotto latenze che non sono accettabili per sistemi **safety-critical**. [10]

1.3 Obiettivi del lavoro svolto

L'obiettivo del lavoro svolto è stato quello di proporre un metodo alternativo alla messa in sicurezza del protocollo CAN valutandone le prestazioni, introducendo uno strato di sicurezza basato su primitive crittografiche avanzate appartenenti alla categoria **Post-Quantum Cryptography**, ovvero quelle primitive crittografiche che sono state realizzate con l'obiettivo di resistere all'attacco di un elaboratore quantistico [8]. Inoltre, in sinergia con l'introduzione delle primitive crittografiche citate, è stato realizzato un "protocollo" che permettesse l'identificazione delle ECU durante la trasmissione (essendo una rete broadcast non è stato previsto un metodo di identificazione) e che permettesse lo scambio di materiale crittografico all'avviamento del motore e a richiesta durante una sessione di guida.

Ovviamente, l'introduzione della cifratura e di "messaggi aggiuntivi" introduce ritardi che, anche se nell'ordine dei microsecondi, non sono trascurabili e devono essere valutati attentamente in una fase di validazione finale.

1.4 Struttura della tesi

Il presente lavoro è stato organizzato come segue:

1. Introduzione al problema e al mondo *Automotive*;
2. Una panoramica sui principali protocolli utilizzati nel mondo *Automotive* con successivo confronto tra questi per evidenziare pro e contro e comprendere quando utilizzarne uno piuttosto che un altro;
3. Una panoramica sulla parte di cifratura interessata dal lavoro svolto, ovvero la crittografia **Post-Quantum**, mettendo in evidenza le caratteristiche e come riescono a garantire una sicurezza contro elaboratori quantistici;
4. Un'esposizione delle idee alla base della soluzione proposta e della progettazione di questa, mostrando anche il modo in cui deve essere realizzata;
5. Un'esposizione del lavoro *pratico* svolto, illustrando dettagli implementativi, strumenti utilizzati e risultati ottenuti;
6. Un capitolo conclusivo dove verranno evidenziate alcune delle possibili migliorie che possono essere apportate al lavoro per migliorare i tempi ed alcuni degli sviluppi futuri del lavoro effettuato.

CAPITOLO 2

Protocolli Automotive

In questo capitolo verranno illustrati i dettagli sui principali protocolli utilizzati nel mondo *Automotive*, evidenziando punti di forza e debolezze. Successivamente, verrà effettuato un confronto tra questi in termini di prestazioni, applicabilità, costi, ecc.

2.1 CAN

2.1.1 Caratteristiche del protocollo

Il protocollo CAN (Controller Area Network) è un protocollo basato su scambio di messaggi che permette a più dispositivi di trasmettere informazioni in maniera affidabile e in logica basata su **priorità**. Tutti i messaggi (o "frame") inviati sono ricevuti da tutti i dispositivi connessi alla rete e, per questa ragione, la tipica topologia di rete usata in sinergia con CAN è **BUS**, ovvero tutti i dispositivi appartenenti alla rete vengono collegati su un singolo cavo detto anche *backbone*. Inoltre, è un protocollo **multi-master** [31], in quanto non ha bisogno di un nodo master che fa da arbitro per l'intera rete ma tutti i nodi connessi sono in grado di comunicare autogestendosi, utilizzando come schema di arbitraggio **CSMA/CD** (Carrier-Sense Multiple Access with Collision Detection) [25]. Con questo schema, un dispositivo che vuole trasmettere un messaggio si mette prima in ascolto sul cavo e, se non rileva trasmissioni in corso, comincia a trasmettere il messaggio ascoltando quello che sta inviando. Se nel frattempo che sta inviando il messaggio rileva un'interferenza, generata dall'arrivo di un altro messaggio già in fase di trasmissione, ferma immediatamente la trasmissione, trasmette una

sequenza di **jamming** con lo scopo di avvertire gli altri dispositivi che è avvenuta una collisione e attende un tempo casuale prima di ritrasmettere (ovviamente dopo aver stabilito che il canale è libero) [4]. Uno schema per comprendere meglio il funzionamento dell'algoritmo **CSMA/CD** è mostrato nella Figura 2.1.

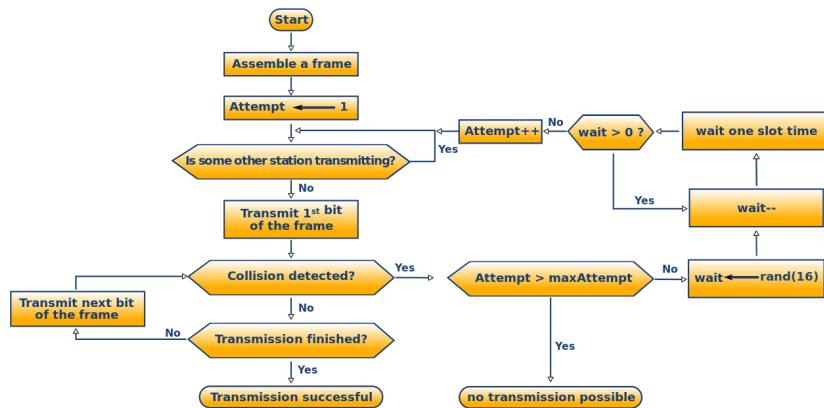


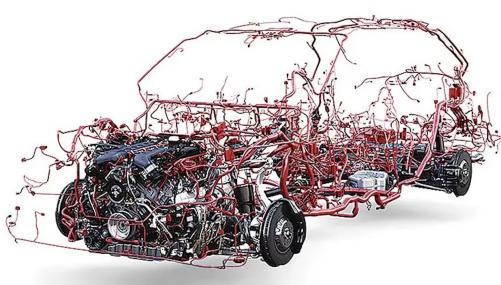
Figura 2.1: Diagramma di flusso di un algoritmo CSMA/CD con ritrasmessione

Ogni dispositivo connesso alla rete viene chiamato **nodo** e deve essere composto almeno da una *CPU* (processore per far funzionare tutto il dispositivo), un controller *CAN* (componente che sappia parlare il protocollo *CAN*) e un *ricetrasmettitore* (componente che "legge e scrive" sui cavi elettrici).

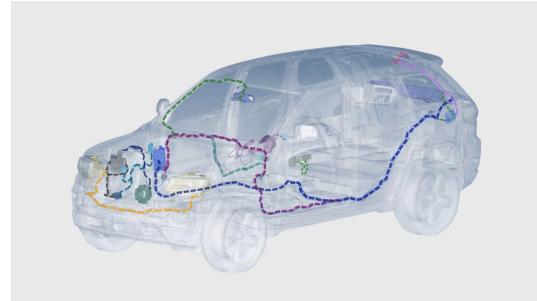
È il protocollo più utilizzato in ambito *Automotive* in quanto il suo utilizzo ha i seguenti vantaggi:

- È un protocollo che richiede un cablaggio semplicissimo e poco costoso (un semplice cavo con due fili), riducendo la latenza, il peso della rete (in termini di kilogrammi), il numero di cavi necessari e il numero di errori;
- È completamente centralizzato e, per questa ragione, basta un qualunque punto di aggancio alla rete per accedere a tutto il traffico in circolazione e comunicare con tutte le *ECU* connesse. Questo semplifica di molto il logging delle informazioni per fini diagnostici e la configurazione delle centraline;
- È molto robusto contro interferenze elettromagnetiche e disturbi elettrici, rendendolo ideale per applicazioni **safety-critical**;
- Grazie all'integrazione di una logica basata su **priorità**, messaggi con *ID* associati ad una priorità alta sono i primi ad accedere alla rete, senza causare interruzioni agli altri messaggi;

- Eliminando chilometri di cavi elettrici in eccesso, permette di ridurre il peso totale dell’automobile su cui è installata la rete migliorando anche i consumi di carburante;
- Dal momento che i chip e la strumentazione necessaria è molto semplice ed economica, il costo necessario alla creazione della rete e delle *ECU* è molto ridotto;
- Prevede meccanismi di correzione e rilevazione degli errori molto efficaci, permettendo alle informazioni di arrivare integre a destinazione;
- È facile aggiungere o rimuovere nodi dalla rete. [9]



(a) Cablaggio tipico di un’automobile che NON utilizza CAN



(b) Cablaggio tipico di un’automobile che utilizza CAN

Figura 2.2: Differenza di cablaggi con e senza CAN

Per via dei motivi sopracitati, oltre ad essere il protocollo più utilizzato in campo *Automotive* è anche utilizzato in moltissimi altri ambiti, tra cui:

- Aeronautica;
- Ascensori ed elevatori;
- Industrie e fabbriche;
- Navi;
- Elettrodomestici casalinghi come lavatrici, asciugatrici, ecc. [9]

2.1.2 Struttura dei messaggi

Ogni messaggio CAN ha una struttura ben definita ed è composto da **header**, **payload** e **trailer**. Inoltre, è possibile individuare due tipologie di messaggi che sono **standard** e **esteso**, la cui unica vera differenza sta nella presenza di un campo *ID* aggiuntivo nel messaggio (quindi cambia solo la lunghezza complessiva del messaggio).

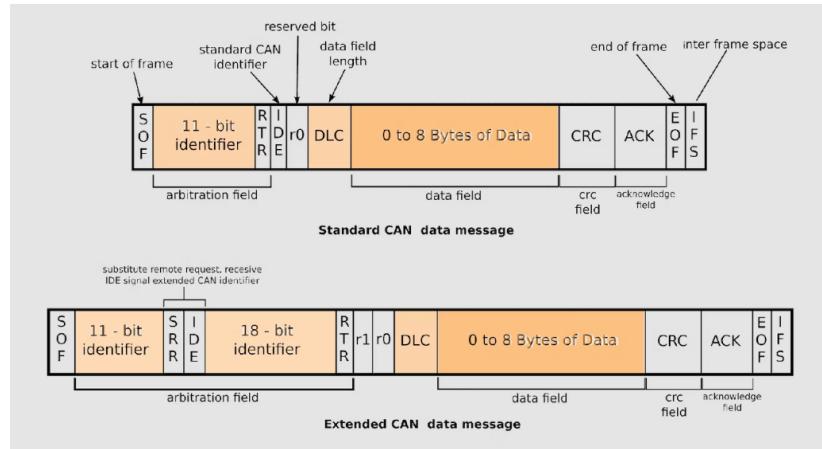


Figura 2.3: Struttura dei messaggi CAN standard ed estesi

I vari campi di un tipico messaggio CAN **standard** sono:

- **SOF** (Start of Frame): 1 bit che indica l'inizio del messaggio e sincronizza i nodi dopo un periodo di inattività;
- **ID**: 11 bit che definiscono la priorità del messaggio, dove un identificativo più basso corrisponde ad una priorità più alta;
- **RTR**: 1 bit che viene impostato a 0 (o *dominante*) quando quello che si sta inviando è una **richiesta di informazione** (non un'informazione) e 1 (o *recessivo*) in caso contrario. Quando il bit viene impostato a 0, non viene inviato nessun dato e l'**ID** determina l'informazione richiesta (quindi il nodo *target*);
- **IDE** (Identifier Extension): 1 bit che determina se il pacchetto è standard (0) o esteso (1);
- **r0**: 1 bit riservato che è privo di significato e lasciato per eventuali sviluppi futuri. Di solito è lasciato a 0, ma anche se impostato a 1 non fa differenza e viene accettato lo stesso;
- **DLC** (Data Length Code): 4 bit che indicano la lunghezza in **byte** del payload che contiene il messaggio;
- **Data**: 64 bit che indicano il dato che si sta inviando con il messaggio;
- **CRC** (Cyclic Redundancy Check): 16 bit che indicano il **checksum**, ovvero una *somma* associata al campo **data** che viene utilizzata, tramite particolari operazioni matematiche, per effettuare rilevamento degli errori;

- **ACK** (ACKnowledgment): 1 bit che determina se un messaggio è stato ricevuto con successo (0) oppure sono stati rilevati degli errori (1). Chi invia il messaggio imposta il bit a 1, mentre chi riceve con successo imposta il bit a 0;
- **EOF** (End of Frame): 7 bit che denotano la fine di un messaggio CAN.
- **IFS** (Inter Frame Space): 3 bit recessivi (impostati a 1) che separano il messaggio appena trasmesso da un nuovo messaggio. Dopo i primi 3 bit a 1, il primo bit dominante (impostato a 0) rilevato corrisponderà ad un bit **SOF**. [9] [15]

Oltre ai campi di un messaggio *standard*, i campi aggiuntivi in un messaggio **esteso** sono i seguenti:

- **SRR** (Substitute Remote Request): 1 bit impostato a 1 che serve a far prevalere sempre i messaggi *standard* durante l’arbitraggio;
- **ID**: 18 bit che si aggiungono al primo campo **ID**, componendo un identificativo di 29 bit totali;
- **R1**: 1 bit riservato come **R0**, quindi anche in questo caso il valore associato a questo campo non ha significato e viene accettato qualunque sia.

Un’altra piccola differenza è che il campo **IDE** è posizionato prima del bit **RTR** e, in mezzo a questi due campi è posizionato il secondo campo **ID**.

2.1.3 Bit stuffing

Per inviare i dati, a livello fisico, CAN utilizza una codifica chiamata **NRZ** (Non-Return-to-Zero), ovvero il bit 1 è codificato con un voltaggio *alto* mentre il bit 0 con un voltaggio *basso* e per codificare n bit consecutivi basta inviare n segnali *alti* o *bassi* in base al bit che si vuole inviare. Inoltre, la codifica può essere utilizzata secondo due tipologie:

- **Unipolare**: il segnale *alto* (quindi il bit 1) è identificato da una differenza di voltaggio **positiva** (detto anche *bias*), mentre il segnale *basso* è identificato dall’assenza di un *bias*;
- **Bipolare**: il bit 1 è codificato da un *bias* positivo mentre il bit 0 da un *bias* negativo.

Esiste anche una codifica che fa da controparte, chiamata **RZ** (Return-to-Zero), nella quale un bit è identificato da una variazione di voltaggio seguito dal ritorno al valore *normale* (nel caso dell’unipolare, il bit 0 è identificato semplicemente dall’assenza di *bias*). Una differenza tra le due codifiche (in versione unipolare) è mostrata in Figura 2.4.

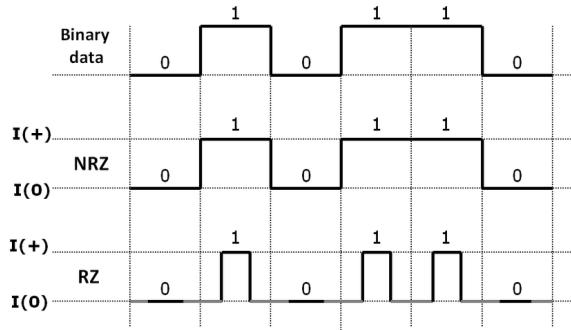


Figura 2.4: Differenza tra codifica **NRZ** e **RZ**, entrambe unipolari

Osservando le due codifiche, per inviare due bit 1 con **NRZ** basta inviare due impulsi consecutivi alti, mentre con **RZ** bisogna inviare un impulso *alto* seguito da un'assenza di *bias* e, di nuovo un impulso *alto* seguito da un'assenza di *bias*. [6]

Per via della natura della codifica **NRZ**, durante l'invio di molti bit consecutivi uguali c'è il rischio che i nodi perdano la sincronia e non siano più in grado di distinguere correttamente i bit inviati, questo poiché la polarità non varia per lunghi periodi di tempo. A questo proposito, l'invio dei messaggi con questa codifica viene affiancato da una tecnica chiamata **bit stuffing**, ovvero dopo l'invio di una certa soglia di bit uguali consecutivi viene inviato un bit di polarità inversa che non ha significato nel messaggio ma serve solo a mantenere i nodi sincronizzati (il bit viene detto *di stuffing*). Ad esempio, immaginando che la soglia di bit consecutivi sia 4, se un nodo deve inviare la sequenza 111111 (sei bit 1 consecutivi), invierà la sequenza 1111011 (dopo 4 bit invierà un bit *di stuffing*). Ovviamente, il nodo che riceve il messaggio è a conoscenza della tecnica, è in grado di rimuovere correttamente i bit *di stuffing* senza compromettere l'integrità del messaggio e nel momento in cui riceve un numero di bit consecutivi uguali al di sopra della soglia prestabilita rileva una violazione della codifica (nel caso di CAN dopo ogni 5 bit uguali consecutivi, **DEVE** essere inviato un bit *di stuffing*).

2.1.4 Tipologie di messaggi

Il protocollo *CAN* prevede quattro tipologie di messaggi (o *frame*), che si distinguono in base al compito che devono svolgere.

Data Frames

Questa tipologia di frame è l'unica che prevede l'utilizzo del campo **Data** e un **DLC** diverso da 0. È, quindi, la tipologia di messaggio che un nodo invia per condividere informazioni con uno o più nodi, associando all'identificativo il tipo di dato inviato e la priorità.

Remote Frames

Questa tipologia di frame permette ad un nodo di richiedere una specifica informazione, identificata dal campo **ID**. Anche se di solito le *ECU* inviano le informazioni sulla rete di propria iniziativa quando l'informazione è pronta, a volte un nodo destinazione può richiedere uno specifico dato da uno specifico nodo. Questa tipologia di messaggio ha un payload vuoto in quanto non vuole inviare informazioni ma, tuttavia, il campo **DLC** è diverso da zero ed indica la lunghezza del dato **richiesto** e non "inviato". Infine, il campo **RTR** in questo caso è posto a 1 (a differenza dei *data frame*) in modo da associare una priorità più bassa a questa tipologia di messaggio e fargli perdere l'arbitraggio in caso di contesa con un *data frame*. [15].

Error Frames

Questa tipologia di frame permette ai nodi, come espone in maniera molto chiara il nome, di segnalare eventuali errori rilevati a tutta la rete. La struttura di questi frame non segue quella descritta in precedenza ma è composta solo da 2 parti:

- **Error flags:** tra 6 e 12 bit tutti *dominanti* o tutti *recessivi*, ottenuti dalla sovrapposizione di *error frame* generati da nodi diversi;
- **Error Delimiter:** 8 bit tutti *recessivi* che delimitano il frame.

La caratteristica fondamentale che contraddistingue questa tipologia di frame è il fatto che il campo **Error flags** è composto da bit uguali e non presenta bit *di stuffing*, realizzando una violazione della regola del *bit stuffing*. Inoltre, ci sono due tipologie di questi frame:

- **Attivo:** tutti i bit del campo **Error flags** sono *dominanti* e questo tipo di frame è generato da un nodo in modalità **attiva**, ovvero un nodo che per primo ha rilevato un errore;
- **Passivo:** tutti i bit del campo **Error flags** sono *recessivi* e questo tipo di frame è generato da un nodo che è passato dallo stato **attivo** allo stato **passivo**.

Come già accennato, la tipologia di frame generato dipende dalla modalità in cui si trova il nodo, che può essere:

- **Attiva:** modalità di default di ogni nodo. In questa modalità il nodo è in grado di inviare dati e frame di errore **attivi**;

- **Passiva:** modalità in cui entrano i nodi che hanno generato un determinato numero di errori. In questa modalità i nodi sono in grado di inviare frame di errore **passivi** e sono in grado di inviare anche dati, tuttavia con l'obbligo di attendere 8 bit di **IFS** (invece dei classici 3 bit) prima di poter prendere il controllo della rete, in modo da dare la precedenza ai nodi **attivi**;
- **Bus off:** modalità in cui entrano i nodi particolarmente *problematici*. In questa modalità il nodo si disconnette completamente dalla rete e non è più in grado di inviare o ricevere messaggi.

Grazie a questo sistema, ai nodi *problematici* verrà assegnata un livello di privilegio sempre più basso fino a spegnerlo del tutto nel caso in cui arrechi troppo disturbo alla rete.

Le varie transizioni tra le possibili modalità sono illustrate nella Figura 2.5.

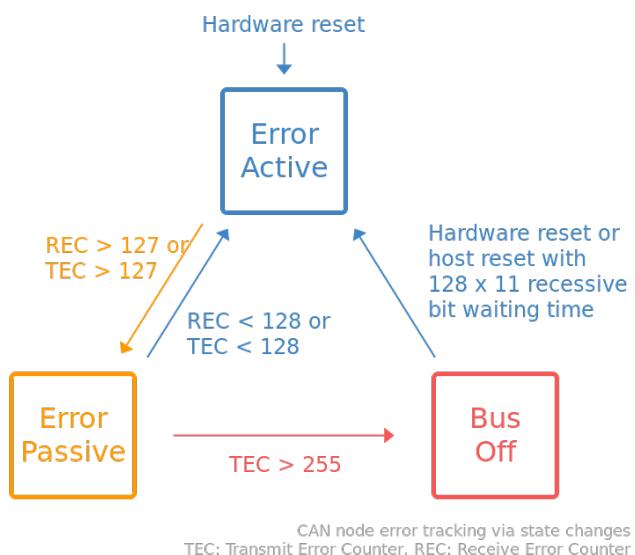
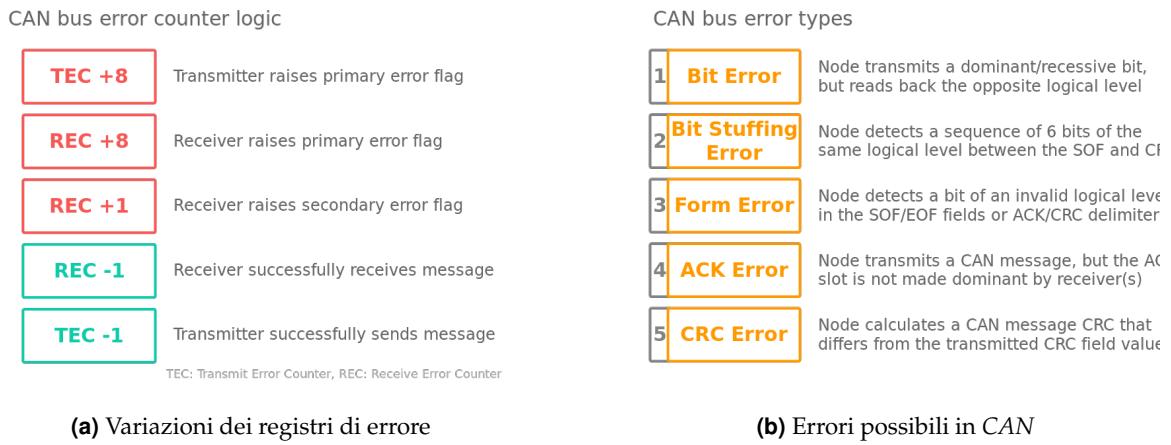


Figura 2.5: Modalità operative con relative transizioni

Per tenere traccia della modalità in cui si trova un nodo, vengono impiegati due registri:

- **TEC** (Transmit Error Counter);
- **REC** (Receive Error Counter).

L'utilizzo di questi due registri permette ad un nodo di capire come si sta comportando e, in base al valore raggiunto da questi, cambierà modalità e si comporterà come stabilito. I valori dei due registri vengono aggiornati come illustrato nella Figura 2.6a e, inoltre, nella Figura 2.6b sono illustrati i possibili errori che un nodo può rilevare. [28] [15]



(a) Variazioni dei registri di errore

(b) Errori possibili in CAN

Figura 2.6: Rilevazione e gestione degli errori in CAN

Overload Frames

Questa tipologia di frame permette ad un nodo saturo di avvisare gli altri nodi di fermare la trasmissione di frame di **dati e remoti**. La struttura è uguale ai frame di **errore** ma la differenza sta nel momento in cui viene inviato, ovvero durante la trasmissione dell'**IFS**. [15]

2.1.5 Varianti del protocollo

Esistono alcune varianti del protocollo CAN, con lo scopo di adattarlo ad ambienti e requisiti diversi. Sebbene se ne possano trovare diverse in letteratura, le principali varianti sono tre.

Low Speed CAN

Variante del protocollo con velocità di trasmissione massima di circa 125 Kbps, utilizzata in sistemi con elevati requisiti di affidabilità, che non richiedono una larghezza di banda elevata e che non richiedono aggiornamenti molto frequenti. Il cablaggio richiesto è molto più economico e di solito viene utilizzato in sistemi diagnostici, nei controlli e display del cruscotto, ecc. [9]

High Speed CAN

Variante del protocollo con velocità di trasmissione massima di circa 1 Mbps, utilizzata in sistemi che richiedono aggiornamenti molto più frequenti ed un'elevata precisione. Richiede un cablaggio più costoso rispetto alla variante **Low Speed** ma permette il corretto

funzionamento di sistemi *safety-critical* come **ABS**, airbag, controllo della stabilità, controlli del motore, ecc. [9]

CAN FD

Variante del protocollo (chiamata "CAN sotto steroidi" [9]) con velocità di trasmissione in grado di raggiungere circa 5 Mbps. Per raggiungere queste velocità si contraddistingue dalle due varianti precedenti sulla lunghezza del payload, dove prima poteva avere una lunghezza massima di 8 Byte mentre con *CAN FD* (Flexible Data-Rate) si possono raggiungere i 64 Byte (incremento dell'800%). Per realizzare questo incremento, ovviamente, l'header è leggermente diverso da quello standard ma è stato realizzato in modo da essere retrocompatibile con le trasmissioni CAN standard.

Le modifiche rispetto ad un frame standard sono:

- **RSS** (Remote Request Substitution) vs **RTR**: essendo che in *CAN FD* non esistono messaggi **remoti**, il bit **RTR** è sostituito con **RSS** ed è sempre 0;
- **FDF** (Flexible Data-Rate Format) vs **R0**: il bit riservato **R0** è sostituito con il bit **FDF** che è sempre 1 e stabilisce che il messaggio è di tipo *FD*. In questa tipologia di messaggio vengono introdotti anche tre bit aggiuntivi;
- **RES**: 1 bit riservato per sviluppi futuri per cui il suo valore non è influente. È aggiuntivo rispetto al frame standard;
- **BRS** (Bit Rate Switch): 1 bit che se impostato a 0 indica che la velocità di trasmissione sarà quella della fase di arbitraggio (velocità fino a 1 Mbps) mentre se impostato a 1 indica che verrà utilizzata una velocità superiore (fino a 5 Mbps). Anche questo è aggiuntivo rispetto al frame standard;
- **ESI** (Error Status Indication): 1 bit che indica la *modalità* in cui è entrato il nodo, quindi sarà 0 se il nodo è in modalità **attiva** mentre 0 se il nodo è in modalità **passiva**. Anche questo bit è aggiuntivo rispetto al frame standard;
- **DLC**: Come nei frame standard, anche qui indica la lunghezza del payload. Tuttavia, a differenza dello standard, la corrispondenza tra valore in bit e lunghezza è leggermente diversa per permettere l'invio di payload più lunghi di 8 byte (la mappatura è indicata in Figura 2.7c);

- **SBC** (Stuff Bit Count): 3 bit che rispettano la codifica di Gray, seguiti da un bit di parità. Sono stati aggiunti per migliorare l'affidabilità della comunicazione;
 - **CRC**: 17 bit o 21 bit per il controllo degli errori del payload di 16 Byte o 20-64 Byte, rispettivamente. A differenza dei messaggi standard, nei messaggi *CAN FD* sono impostati **SEMPRE** 4 bit di stuffing, mentre prima potevano essercene tra 0 e 3;
 - **FSB** (Fixed Stuff Bit): bit di stuffing impostati prima e dopo **SBC** e all'interno del campo **CRC**;
 - **ACK**: questo campo delimita la fine dell'invio a velocità superiori, ritornando alla velocità di default di *CAN*.

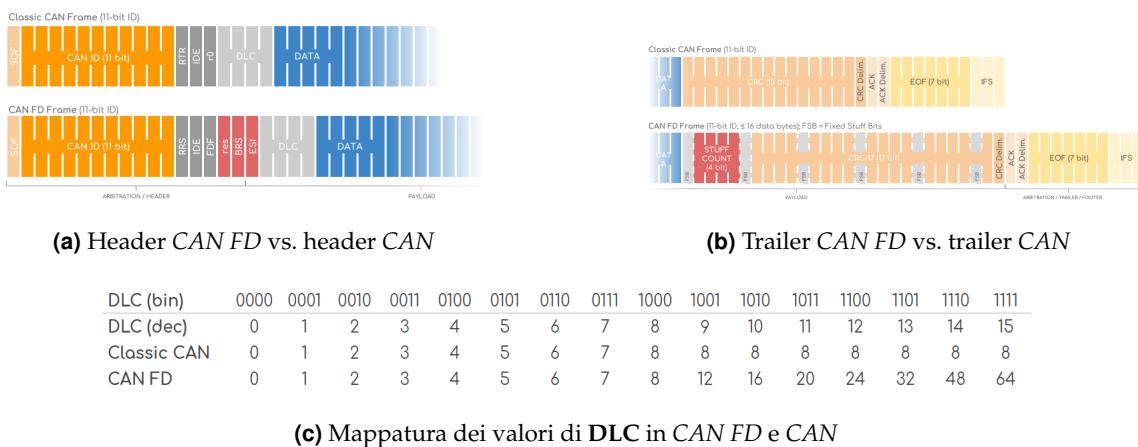


Figura 2.7: Differenze tra CAN FD e CAN

Un altro vantaggio molto importante che introduce questa variante è la possibilità di variare in maniera adattiva la velocità di trasmissione dei messaggi. Questo permette ai nodi di adattarla in base a ciò che si vuole inviare, lasciando la velocità invariata o accelerando in caso di dati più urgenti o messaggi più lunghi.

Rispetto alla variante standard, con CAN FD si possono realizzare sistemi più complessi che richiedono requisiti di banda più stringenti. Ad esempio, si possono realizzare sistemi **ADAS** (Advanced Driver Assistance System) come cruise control adattivo, frenata automatica d'emergenza, monitoraggio degli angoli ciechi, ecc. [26]

2.2 FlexRay

Con il procedere degli anni e con lo sviluppo di sistemi sempre più avanzati e complessi, è stato raggiunto un punto dove il protocollo CAN comincia a far pesare i propri limiti. Se si

pensa all'aumento del numero delle *ECU* all'interno delle automobili, allo sviluppo di sistemi **X-by-wire** e all'implementazione di sistemi di sicurezza sempre più all'avanguardia, anche utilizzando la variante *CAN FD* non si riescono a garantire i sempre più stringenti requisiti di banda e di affidabilità richiesti per il corretto funzionamento di tutti i sistemi. È proprio per questo motivo che un team specializzato ha realizzato **FlexRay**, un protocollo in grado di soddisfare elevati requisiti di banda, sicurezza, affidabilità e velocità di trasmissione. Non nasce con l'idea di sostituire *CAN* ma, piuttosto, con quella di affiancarsi e cooperare con esso, sebbene la tendenza che stanno seguendo i produttori è di sostituire *CAN* con *FlexRay*.

2.2.1 Caratteristiche del protocollo

Il protocollo *FlexRay* presenta diversi vantaggi:

- Permette il raggiungimento di una larghezza di banda di 10 Mbps, molto più alta rispetto a *CAN* e permette di variare la velocità di trasmissione in base alle necessità;
- Richiede un cablaggio semplice, composto da un cavo con due o quattro fili [5], ma è in grado di lavorare anche con cablaggi di tipo ottico;
- Supporta diverse topologie di rete oltre al **BUS**;
- Assicura una maggiore tolleranza ai guasti maggiore e determinismo;
- Permette l'invio di payload fino ad una lunghezza massima di 254 Byte, circa 30 volte più grande di *CAN*;
- Consente l'accensione e lo spegnimento dei nodi in maniera dinamica per permettere una riduzione dei consumi elettrici.

Costi

Sebbene sia migliore di *CAN* sotto molti aspetti, lo svantaggio più importante è il costo rischiato per una rete *FlexRay*, in quanto richiede cavi più costosi per garantire una banda più alta e dispositivi più sofisticati. Tuttavia, essendo che il protocollo è stato realizzato in maniera da poter cooperare con *CAN* e altri protocolli, è possibile abbattere i costi di produzione utilizzando *FlexRay* solo dove strettamente necessario, quindi per sistemi avanzati con stringenti requisiti di affidabilità e banda, e dove non è necessario si può utilizzare *CAN* o altri protocolli che richiedono cablaggi più economici. [35]

Affidabilità

Per una migliore tolleranza ai guasti, *FlexRay* utilizza i due fili come due canali di comunicazione separati, infatti, una *ECU* che opera con questo protocollo ha bisogno di un **controllore FlexRay** che permette l'utilizzo del protocollo stesso e due **ricetrasmettitori** che permettono di "scrivere" e "leggere" su entrambi i fili in maniera indipendente (come mostrato in Figura 2.8) e inviare uno stesso messaggio in maniera ridondata su entrambi i fili (oppure una sola volta su uno solo filo in base alla configurazione). I due fili possono essere usati anche come canali separati per inviare messaggi diversi, aumentando in questo modo la larghezza di banda disponibile. Inoltre, *FlexRay* utilizza anche un ulteriore circuito (opzionale), chiamato **Bus Guardian** [35], il cui scopo è quello di rilevare interferenze generate da messaggi non sincronizzati e proteggere il canale da questi. Utilizzando questo ulteriore dispositivo si riduce in maniera drastica la possibilità di ottenere collisioni. [35]

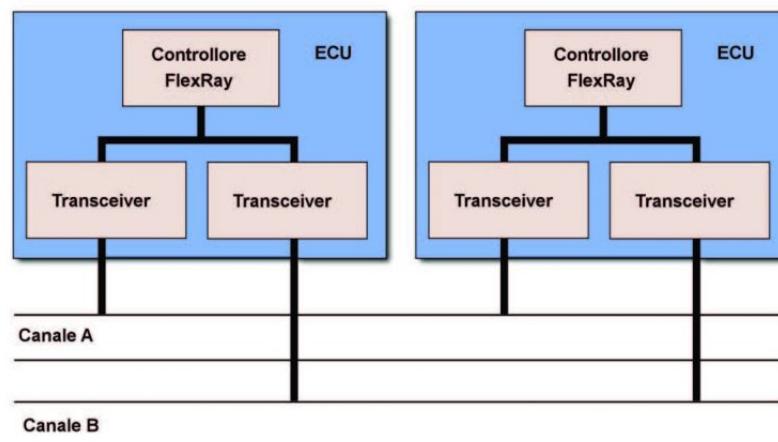


Figura 2.8: Struttura tipica di una *ECU* che utilizza *FlexRay*

Determinismo

FlexRay utilizza un meccanismo di arbitraggio chiamato **TDMA** (Time Division Multiple Access), ovvero ad ogni nodo della rete viene assegnato uno *slot* temporale in cui può trasmettere un messaggio. Affinchè non ci siano collisioni, è necessario che ogni nodo della rete abbia il proprio *clock* sincronizzato con gli altri, altrimenti un nodo potrebbe trasmettere in uno slot che non gli è stato assegnato facendo collidere il proprio messaggio con quello inviato dal nodo *legittimo*. Essendo un sistema distribuito, i nodi non hanno un *clock* fisico **assoluto** a cui far riferimento, ma vengono impiegati diversi **nodi di sincronizzazione**, il cui scopo è quello di inviare periodicamente **frame di sincronizzazione** che aiutino i nodi della

rete ad aggiustare il proprio clock in caso di *sfasature*. Così facendo, tutti i clock (compresi quelli dei nodi di sincronizzazione) rimangono allineati garantendo il corretto funzionamento dell’arbitraggio TDMA. [35]

2.2.2 Ciclo di comunicazione

In *FlexRay* l’accesso al canale condiviso è basato sull’assegnamento di uno slot temporale ad ogni nodo, all’interno del quale è possibile inviare un messaggio. Questo assegnamento viene reiterato in un ciclo che prende il nome di *ciclo di comunicazione* ed è l’elemento principale dello schema di accesso al canale condiviso. È definito in termini di una **gerarchia temporale** composta da 4 livelli, che sono i seguenti:

- **Communication Cycle:** è il livello più alto e definisce il ciclo in termini di 4 fasi (o segmenti), che sono **segmento statico**, **segmento dinamico**, **symbol window** e **network idle time**;
- **Arbitration Grid:** livello subito inferiore al precedente ed è lo scheletro di tutto lo schema di arbitraggio. Divide i vari segmenti in sezioni temporali che è possibile assegnare ai vari nodi della rete e, in particolare, nel *segmento statico* queste sezioni prendono il nome di **slot statici** mentre nel *segmento dinamico* prendono il nome di **minislot**. La differenza tra questi due sta nel meccanismo di arbitraggio utilizzato per assegnarli ai vari nodi.
- **Macrotick:** livello subito inferiore al precedente e definisce la durata di un ciclo e di ogni segmento e sezione, i quali hanno una durata corrispondente ad un numero intero di *Macrotick*. Un *Macrotick* si definisce come **un numero intero di Microticks**;
- **Microtick:** livello più basso e definisce l’elemento base di tutto il ciclo, il *Microtick*. Esso non è altro che un numero intero di *tick* (ticchettii, battiti) del clock di un oscillatore presente in un *controllore FlexRay* di un nodo (come in un normale processore all’interno di un computer).

Segmento statico

Segmento sempre presente in ogni *ciclo di comunicazione* ed è la fase iniziale del ciclo. Questa fase è suddivisa in *slot statici* dalla stessa lunghezza in **Macrotick** e il numero dipende dalla configurazione della rete. Questi slot vengono assegnati in maniera statica ai nodi della rete e, quindi, il meccanismo di arbitraggio presente in questa fase è il **TDMA**. Affinchè

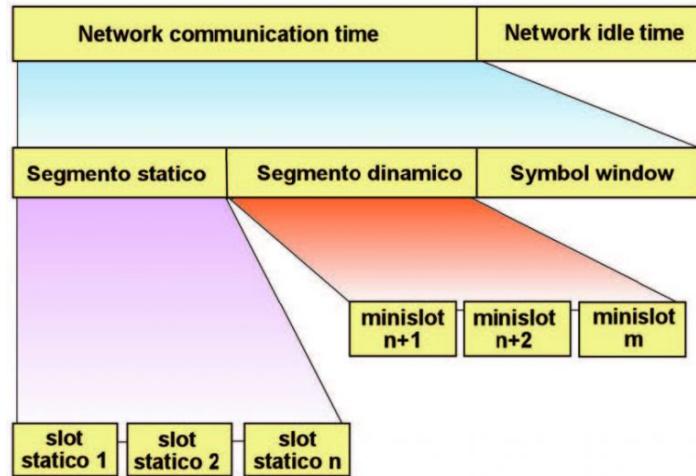


Figura 2.9: Suddivisione del ciclo di comunicazione

tutti i nodi siano in grado di rispettare i turni, ognuno mantiene un *contatore di slot* (uno per ogni canale) che viene incrementato ad ogni numero di **Macrotick** corrispondente alla lunghezza di uno *slot statico*. In questo modo, tutti sono a conoscenza dello slot corrente e a chi è assegnato. [11]

Questa fase garantisce un livello di determinismo massimo, dal momento che tutti sanno quando gli altri trasmetteranno dati, tuttavia un *segmento statico* troppo lungo rischia di rallentare inevitabilmente la rete e introdurre latenze eccessive per messaggi urgenti. Inoltre, se un nodo non invia nulla durante il proprio *slot statico*, questo è interamente sprecato e bisogna attendere la fase successiva o il successivo ciclo. [5]

Segmento dinamico

Segmento successivo e opzionale a quello *statico* che può essere omesso in fase di configurazione della rete. Questa fase è suddivisa in *minislot* tipicamente lunghi pochi **Macrotick** (anche uno solo e, in generale, più breve di uno *slot statico* [35]) e, anche qui, il numero di *minislot* dipende dalla configurazione della rete. In questa fase, i *minislot* vengono assegnati in base alla priorità delle informazioni da inviare, quindi un nodo che deve inviare un'informazione con priorità più elevata avrà assegnato un *minislot* più vicino all'inizio della fase mentre un nodo con informazioni meno prioritarie avrà assegnato un *minislot* più lontano. Anche in questa fase vengono impiegati dei contatori per stabilire in quale *minislot* ci si trova, solo che a differenza del *segmento statico* la durata di un singolo *minislot* può variare:

- Se il nodo a cui è assegnato il *minislot* rimane in *idle* e non trasmette nulla, la durata del *minislot* rimane effettivamente quella definita in fase di configurazione della rete;

- Se il nodo decide di trasmettere, la durata del *minislot* viene "espansa" facendolo durare più *minislot* e permettendo la trasmissione dell'intero dato senza interruzioni e in maniera *fault-tolerant*.

Questo schema di "espansione" permette di non perdere larghezza di banda preziosa nel caso in cui uno o più nodi decidano di non inviare nulla. Inoltre, è necessario perché la durata di un *minislot* è tale da non riuscire ad includere un'intera messaggio *FlexRay* e, per questa ragione, quando un nodo decide di trasmettere ha bisogno di un intervallo più lungo.

Come già anticipato, anche qui vengono utilizzati dei contatori per tenere traccia del *minislot* in cui ci si trova ma, essendo che chi trasmette acquisisce più *minislot*, viene fissato un limite di *minislot* "effettivi" oltre il quale passare alla fase successiva (o ciclo successivo) anche se non si è raggiunto il numero di *minislot* "definito". Questo significa che i nodi con messaggi meno prioritari potrebbero perdere il proprio turno e trasmettere nella fase successiva, ma non è un problema in quanto essendo meno prioritari sono meno suscettibili alle latenze rispetto a quelli con priorità più alta. [35] [11]

Lo schema di arbitraggio utilizzato in questa fase ricorda molto quello utilizzato da *CAN*.

Symbol Window

Segmento successivo al *dinamico* ed è una fase dove vengono scambiati messaggi di gestione della rete. Tipicamente le applicazioni di alto livello non interagiscono con questa fase [5] e i messaggi inviati possono essere di sincronizzazione del clock, allarmi vari o di wakeup o startup. Anche la lunghezza di questa fase è definita in fase di configurazione e in termini di **Macrotick**.

Network Idle Time

Segmento successivo alla *Symbol Window* ed è una fase in cui non avviene nessuno scambio di messaggi. Lo scopo di questa fase è quella di permettere ai nodi della rete di effettuare aggiustamenti al proprio clock interno per correggere eventuali *drift* e di eseguire eventuali task aggiuntivi relativi al ciclo di comunicazione. Anche la durata di questa fase è definita in fase di configurazione e tipicamente è molto breve, questo perché essendo un periodo di inattività tutta la banda assegnata a questa fase è sprecata. Il tutto risulta nell'introduzione di latenze aggiuntive che devono essere ottimizzate facendo durare questa fase il meno possibile. [11]

2.2.3 Sincronizzazione del Clock

In un sistema distribuito come una rete *FlexRay*, ogni nodo ha il proprio *clock* con cui scandisce il tempo. Tuttavia, a causa di fluttuazioni di temperatura, fluttuazioni di voltaggio, purità dei materiali usati nell'oscillatore, differenze dovute alla produzione, ecc. può succedere che i *clock* scandiscano il tempo in maniera diversa (più o meno veloce) e questo significa che anche se i *clock* partono sincronizzati, questi divergono dopo piccoli intervalli di tempo. In un sistema basato quasi interamente su turni e intervalli temporali, ovviamente, clock non affidabili portano all'impossibilità di comunicare senza errori e con le prestazioni desiderate. Per questa ragione, è necessario che un meccanismo di **sincronizzazione dei clock** che, a intervalli regolari, sincronizzi tutti i *clock* prima che questi divergano compromettendo la rete. Non è necessario che questi siano **perfettamente** sincronizzati, piuttosto basta che ogni coppia di nodi indichi il "tempo globale" con una differenza che non supera una determinata soglia. [11]

Affinchè i nodi rimangano sincronizzati, *FlexRay* prevede la presenza di **nodi di sincronizzazione** il cui scopo è quello di inviare dei messaggi chiamati *sync frame* ad ogni ciclo che vengono utilizzati da tutti i nodi per correggere il proprio *clock* da eventuali *drift* (deviazioni). Sebbene il numero di **Microtick** necessari per un **Macrotick** sia definito in fase di configurazione della rete, un nodo può modificare questo valore in maniera dinamica per correggere il *drift* del proprio *clock*, incrementandolo o diminuendolo in base a se il *clock* è troppo veloce o troppo lento [35] in modo tale da uniformare per tutti i nodi la durata di un **Macrotick**. In questo caso si parla di un problema di **frequenza**, ovvero la durata di un **Macrotick** non è la stessa, facendo slittare i vari *slot*. [11] [37]

Un altro aggiustamento che ogni nodo dovrebbe fare riguarda l'istante in cui una fase comincia poichè, anche se la durata dei **Macrotick** è uguale per ogni nodo, può succedere che l'inizio di un ciclo è sfasato e quindi il nodo crede di essere in una fase/*slot* errata. Anche in questo caso l'inizio e la fine di un ciclo sono impostati durante la configurazione della rete, ma ogni nodo è comunque in grado di impostare un *offset* in modo tale da eseguire uno *shift* avanti o indietro, in termini di **Microtick**, in modo da aggiustare la visione di ogni nodo riguardo il ciclo di comunicazione. In questo caso si parla di un problema di **offset**, ovvero l'inizio di un ciclo o di una fase è slittata di un determinato numero di **Microtick**. [11] [37]

Se un nodo non riceve abbastanza *frame di sincronizzazione*, il *clock* soffrirebbe di un *drift* inaccettabile per il ciclo di comunicazione e, per questa ragione, può diventare un problema per gli altri in quanto potrebbe trasmettere in uno slot non suo oppure trovarsi in un segmento

diverso da quello degli altri nodi. Per evitare che un nodo in questa situazione arrechi disturbi all'intera rete, *FlexRay* prevede che questo nodo segnali all'applicativo la sua entrata in una modalità *passiva*, dove rimane in grado di ricevere i messaggi dagli altri ma non è più in grado di trasmettere messaggi sulla rete. [35]

2.2.4 Wakeup e Startup

A differenza di *CAN*, *FlexRay* permette di mettere una o più *ECU* (ovviamente non necessarie) in uno stato di *sleeping* in modo tale da minimizzare i consumi elettrici della rete. Per gestire il meccanismo di accensione e spegnimento dinamico della rete o di un gruppo di *ECU* sono previste due funzioni: *Wakeup* e *Startup*.

Wakeup

Questa funzione permette ad una *ECU* l'invio di un messaggio speciale, chiamato **Wake Up Pattern** (*WUP*), che avverte tutte le *ECU* dormienti di cominciare la fase di avvio. Questo messaggio speciale di solito è inviato durante la *Symbol window* e deve essere inviato solo ed esclusivamente quando nessun'altro ha intenzione di trasmettere messaggi. [35]

Startup

Funzione che viene avviata subito dopo la ricezione di un messaggio *WUP* e il suo scopo è quello di allineare tutti i nodi all'arbitraggio **TDMA** e sincronizzare i clock. In base a quanti nodi vengono riattivati, questa funzione in due tipologie:

- **Coldstart**: si vogliono riattivare tutti i nodi della rete e quindi la rete non è operativa;
- **Integration**: si vogliono riattivare un sottoinsieme di nodi all'interno di una rete già operativa

Una volta terminato il *Wakeup*, un nodo si mette in ascolto sul bus per un determinato intervallo di tempo per capire se c'è del traffico. Se non viene rilevato traffico, allora ci si trova in una tipologia **Coldstart** e, per questo motivo, invia un **Collision Avoidance Symbol** (*CAS*) e successivamente un pattern di *coldstart* per un certo numero di cicli. Appena il nodo riceve risposta da un altro nodo, questo termina la fase di *startup* entrando nello stato *operativo* e generando un normale traffico. Se, invece, viene rilevato traffico sul bus allora ci si trova in una tipologia **Integration** e quindi sono presenti dei nodi attivi. In questo caso il nodo non genera dei pacchetti come nel **Coldstart** ma si mette nello stato *re-integrazione* e si sincronizza

con i messaggi che arrivano sul bus. Una volta terminata la fase di sincronizzazione, il nodo passa nello stato *operativo* partecipando attivamente alla rete.

2.2.5 Struttura di un messaggio

Un messaggio in *FlexRay* è composto da tre parti (**header**, **payload** e **trailer**) e si struttura come segue:

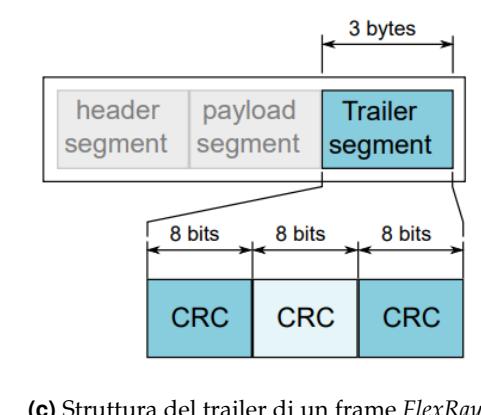
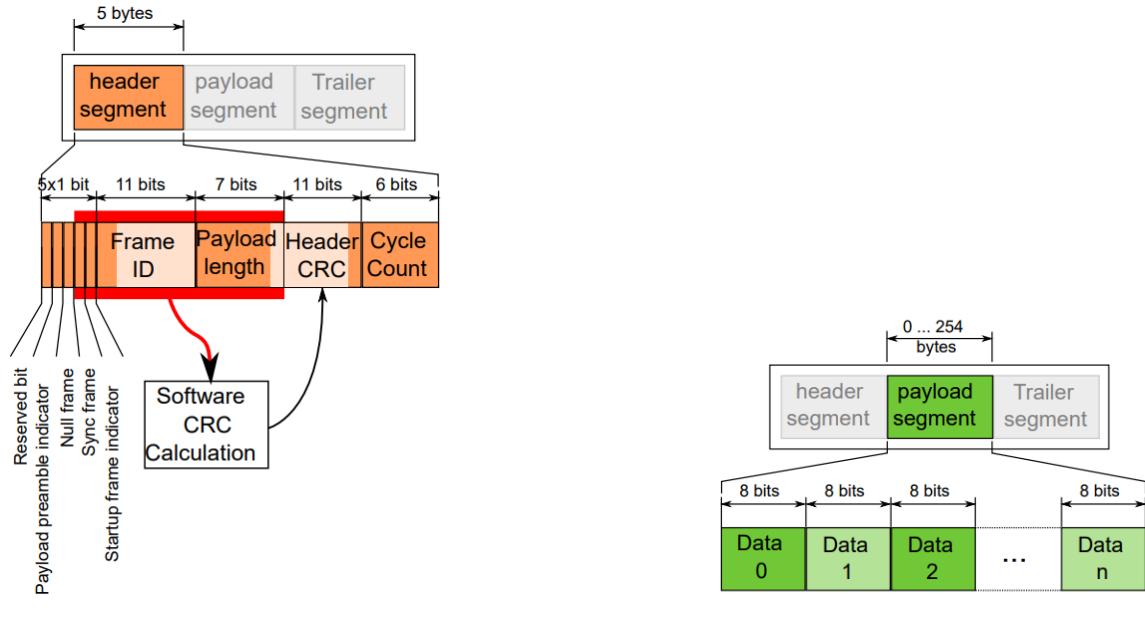


Figura 2.10: Struttura di un frame *FlexRay*

- **Riservato:** 1 bit che è lasciato per sviluppi futuri;
- **Payload Preamble Indicator:** 1 bit che specifica se il campo **payload** è utilizzato per scambio di dati (bit impostato a 0) oppure per un motivo diverso (bit impostato a 1);

- **Null frame:** 1 bit che indica se il payload contiene dati validi (bit impostato a 1) oppure se contiene dati non validi (bit impostato a 0). In quest'ultimo caso, tutti i bit del campo **payload** vengono impostati a 0;
- **Sync Frame:** 1 bit che indica se il frame è di sincronizzazione (bit impostato a 1) oppure no (bit impostato a 0). I frame di sincronizzazione possono essere inviati solo se ci si trova in un segmento statico;
- **Startup Frame Indicator:** 1 bit che indica se il frame è di tipo **startup** (bit impostato a 1) oppure no (bit impostato a 0). I frame di startup sono dei tipi particolari di frame che possono essere inviati solo da nodi *coldstart* per sfruttare il meccanismo dello *startup* dei nodi, inoltre, questo tipo di frame dovrebbe avere sia questo bit sia il bit **sync** a 1;
- **Frame ID:** 11 bit che definiscono lo slot temporale nel quale questo messaggio deve inviato. Ovviamente in un ciclo di comunicazione deve essere unico ed è un numero nel range 1-2047 (l'ID 0 non è valido);
- **Payload Length:** 7 bit che indicano quanti gruppi di 2 Byte sono presenti nel payload. Se ad esempio, questo campo viene impostato a 35, allora la lunghezza del payload sarà $35 \times 2 = 70$ Byte.
- **Header CRC:** 11 bit che indicano il codice **CRC** per individuare errori nell'header. Viene calcolato tenendo in considerazione i bit **Sync frame**, **Startup Frame Indicator**, **Frame ID** e **Payload Length**;
- **Cycle Count:** 6 bit che indicano in quale ciclo di comunicazione ci si trova al momento dell'invio del frame;
- **Payload:** segmento che ha una lunghezza massima di 254 Byte e che contiene i dati da inviare insieme al frame;
- **CRC:** 24 bit che rappresentato il codice **CRC** per individuare errori nel payload;

2.2.6 Topologie supportate

Il protocollo *FlexRay* per come è stato progettato supporta diverse topologie di rete, in modo tale da adattarsi al meglio ad ogni tipo di situazione e garantire sempre elevati requisiti di affidabilità. Le topologie maggiormente utilizzate sono le seguenti:

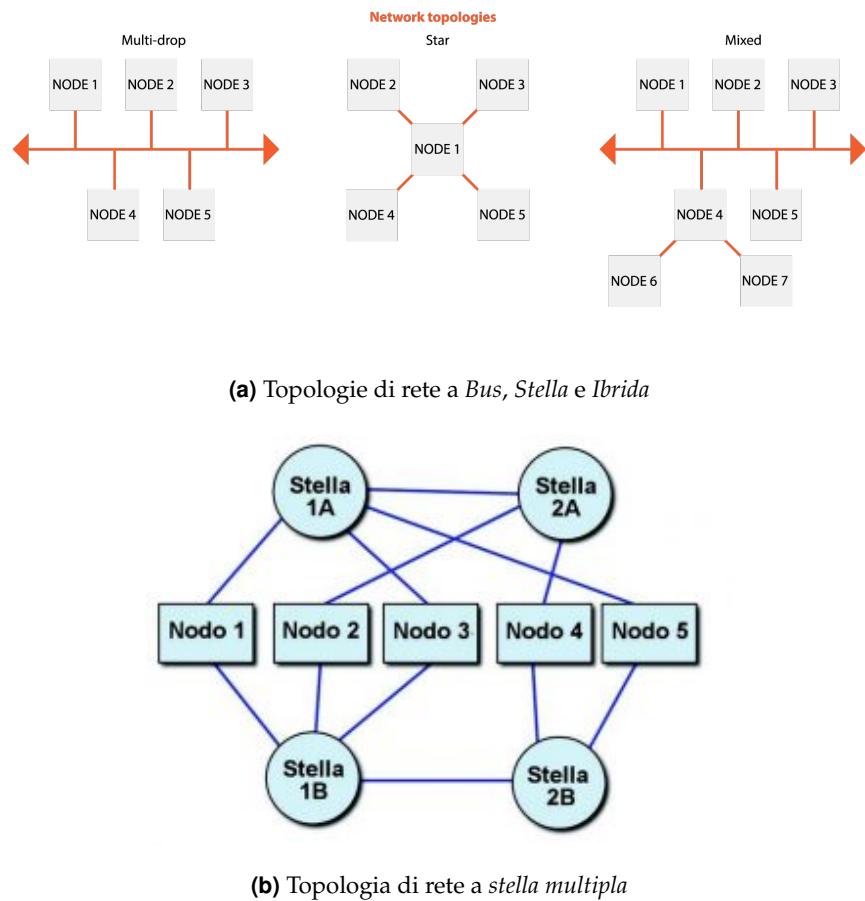


Figura 2.11: Principali topologie di rete supportate da *FlexRay*

- **Bus:** anche detta *Multi-drop Bus* è la topologia più utilizzata (Figura 2.11a). In questo caso, c'è un cavo principale che fa da tronco (*trunk*) e tutte le ECU sono collegate a questo cavo tramite un connettore e un cavo più breve. Il cavo principale è poi terminato alle estremità con dei resistori per evitare problemi di **riflessione** del segnale. Questa è una delle topologie più utilizzate in quanto è anche quella utilizzata da CAN, permettendo così una perfetta interoperabilità tra i due protocolli. Inoltre è anche la topologia che richiede un cablaggio più semplice rispetto alle altre topologie, rendendola la più economica. Tuttavia, essendo un unico cavo che collega tutti nodi, nel momento in cui questo si danneggia tutta la rete viene compromessa. Inoltre, essendo che questo cavo molto spesso è lungo, risente molto delle interferenze elettromagnetiche generate dal motore dell'automobile;
- **Stella:** in questa topologia (Figura 2.11a), tutte le ECU sono collegate con una ECU attiva che fa da *centro-stella* e ha un funzionamento che è simile a quello di un *hub* o uno *switch* casalingo. Il vantaggio di questa topologia sta nel fatto che non c'è un

unico cavo che collega tutti i nodi ma ci sono tanti piccoli segmenti che rendono i collegamenti indipendenti. Infatti, nel momento in cui un segmento si danneggia viene isolata solo una *ECU* lasciando la restante rete completamente funzionante. Inoltre, avendo collegamenti più corti, nel momento in cui si collegano più centri-stella tra di loro è possibile raggiungere distanze maggiori senza risentire delle interferenze elettromagnetiche;

- **Irida:** topologia che combina i vantaggi della topologia a *Stella* e della topologia *Bus* (Figura 2.11a). In questo caso c'è un *trunk* che collega sia singoli nodi che centri-stella, abbassando i costi dei cablaggi e sfruttando le performance e l'affidabilità delle stelle;
- **Stella multipla:** detta anche **Dual-channel single star** topologia che cerca di migliorare l'affidabilità della stella aggiungendo un'ulteriore stella che si collega a tutti i nodi o ad un sottoinsieme di nodi (Figura 2.11b). Questa topologia si può realizzare sfruttando il fatto che ogni cavo *FlexRay* è composto da due fili, quindi un filo sarà collegato ad una stella mentre l'altro ad un'altra stella. Ovviamente questo tipo di topologia rende il cablaggio più complesso però migliora l'affidabilità complessiva della rete. Inoltre, è possibile anche collegare in cascata più stelle tra di loro realizzando una topologia **Dual-channel cascaded star** aggiungendo ulteriori collegamenti ridondanti. [35] [11]

2.3 LIN

Il protocollo *LIN* (Local Interconnect Network) è un protocollo a bassa affidabilità e larghezza di banda ridotta rispetto agli altri protocolli. La sua utilità nasce dal fatto che, essendo a basso costo, può sostituire una rete *CAN* o *FlexRay* (più costose rispetto a *LIN*) per collegare componenti che non richiedono requisiti di affidabilità e banda particolari, come ad esempio i comandi dell'aria condizionata, i finestrini elettronici, la chiusura elettronica delle portiere, ecc.). Una prima differenza importante rispetto agli altri protocolli visti è che non è composto da nodi **paritari**, ma è composto da un nodo **master** e da un determinato numero di nodi **slave**. Per questa ragione non c'è bisogno di un **meccanismo di arbitraggio**, in quanto la rete è **arbitrata** dal nodo *master* che si occuperà di interrogare gli *slave* facendogli condividere informazioni e facendogli svolgere determinate operazioni

2.3.1 Caratteristiche del protocollo

Alcune delle caratteristiche di *LIN* sono le seguenti:

- Richiede un cablaggio a **basso costo** utilizzabile quando banda e affidabilità non sono richiesti. Consiste di un solo cavo con un singolo filo (gli altri ne usano almeno due);
- Utilizza un'infrastruttura di tipo **master-slave**, per cui non è richiesto un meccanismo di arbitraggio e una rete *LIN* è composta da 1 nodo *master* e fino a 16 nodi *slave*;
- Ha una larghezza di banda massima di circa 20 Kbps;
- Supporta un meccanismo di correzione degli errori, un meccanismo di **sleep & wakeup** (come *FlexRay*) e può essere configurato; [27]
- Può essere inviata una quantità variabile di dati (2, 4 e 8 Byte). [20]

Una particolarità di *LIN* è che può lavorare in sinergia con reti *CAN* e *FlexRay*, infatti, solitamente più reti *LIN* sono collegate tra loro utilizzando una *backbone* realizzata con uno di questi ultimi due protocolli o simili.

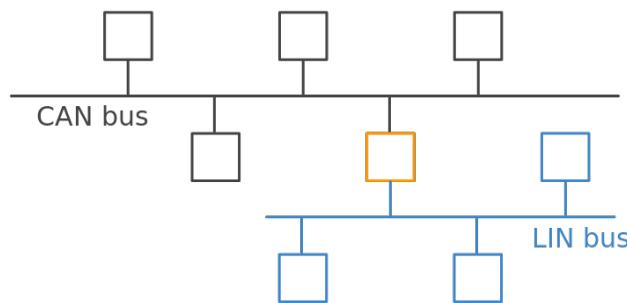


Figura 2.12: Rete *LIN* collegata ad una backbone *CAN*

Per questo motivo il nodo *master*, oltre a gestire la rete, fa anche da *gateway* per la rete di *backbone*, inviando e ricevendo messaggi su di essa per comunicare con le altre reti *LIN* collegate.

2.3.2 Struttura dei messaggi

Un tipico *frame LIN* consiste di *header* e *risposta*, entrambi con una struttura ben definita. Essendo che non viene previsto un meccanismo di arbitraggio e che è presente un nodo *master*, esso invia un *header* che viene intercettato da tutti i nodi *slave* e, quando l'interessato lo riceve, genera una *risposta* che viene inviata al *master*. Non possono essere generate collisioni in quanto il *master* può "interrogare" un solo *slave* alla volta, per cui ci sarà sempre una e sola trasmissione in atto sul canale.

L'*header* è composto come segue:



Figura 2.13: Struttura dei frame LIN

- **Break o SBF (Sync Break Field):** 14 bit o più (di solito ne vengono utilizzati almeno 18 bit) che agiscono come *SOF*, quindi avvertono i nodi *slave* che il nodo *master* sta per inviare un messaggio sul canale;
- **Sync:** 8 bit che hanno un valore preimpostato a 0x55 (corrispondente alla sequenza binaria 01010101) e che permettono ai nodi *slave* di sincronizzare il proprio clock e di capire a quale velocità il nodo *master* trasmetterà il messaggio;
- **Identifier:** 6 bit, seguiti da 2 bit di parità (utilizzati per accettare la correttezza dei bit precedenti), che identificano il nodo *slave* al quale il messaggio è diretto. [27]

Essendo che tutti i frame vengono inviati in broadcast, nel momento in cui uno slave riceve un header può intraprendere una delle seguenti azioni:

1. Ignorare la trasmissione in quanto non interessato;
2. Mettersi in ascolto di una trasmissione proveniente da un altro nodo;
3. Inviare un *frame* di risposta.

La *risposta* invece è composta come segue:

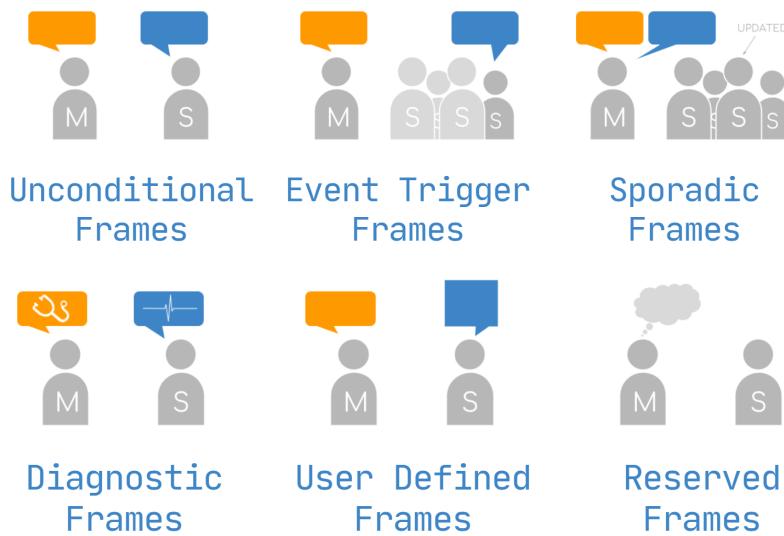
- **Data:** tra 16 e 64 bit che determinano il *payload* che un nodo *slave* vuole inviare. La lunghezza può variare tra 2, 4 e 8 Byte e, sebbene si possa personalizzare, di default la lunghezza del payload è associata ad un intervallo di identificatori (l'assegnazione di default è riportata in Tabella 2.1);
- **Checksum:** 8 bit che rappresentano la somma di controllo che serve ad assicurare l'integrità del messaggio. [27]

Intervallo ID	Lunghezza
0-31	2 Byte
32-47	4 Byte
48-63	8 Byte

Tabella 2.1: Assegnazione di default delle lunghezze

2.3.3 Tipologie di messaggi

Nel protocollo *LIN* sono previste 6 tipologie di messaggi, i quali non presentano differenze a livello di **struttura** ma differiscono solo per la **semantica**, in base al contenuto del payload, in base a quando vengono inviati o in base all'ID che hanno associato. Le tipologie sono le seguenti:

**Figura 2.14:** Tipologie di messaggi in *LIN*

1. **Unconditional Frames:** sono il metodo di comunicazione di default. Il nodo *master* invia un header per richiedere informazioni ad uno specifico nodo *slave*, il quale risponde all'header inviando un messaggio con le informazioni richieste; [27]
2. **Event Trigger Frames:** il nodo *master* interroga diversi nodi *slave* richiedendo delle informazioni. Appena un nodo *slave* ha le informazioni pronte, imposta come primo Byte del payload il proprio ID, seguito dalle informazioni richieste. Nel momento in cui il nodo *master* rileva una collisione (interrogando più nodi questi potrebbero rispondere contemporaneamente), ritorna allo schema di default **Unconditional**. Questo schema è

utile per aumentare la reattività della rete permettendo ai nodi di inviare le informazioni aggiornate appena pronte e senza aspettare l’interrogazione del *master*; [20]

3. **Sporadic Frames:** messaggi utilizzati nel caso particolare in cui il nodo *master* è già a conoscenza di informazioni aggiornate riguardo uno specifico *slave*. In questo caso, il nodo *master* invia un header e lui stesso si "comporta come uno *slave*" rispondendo al proprio header con il payload aggiornato di cui era a conoscenza. Questo tipo di messaggio serve principalmente ad aggiornare i nodi *slave* fornendo informazioni aggiornate e dinamiche;
4. **Diagnostic Frames:** messaggi utilizzati per scambiare informazioni **diagnostiche**. Questa tipologia di messaggio contiene sempre 8 Byte di dati e solitamente l’ID 60 corrisponde alla richiesta del *master* mentre l’ID 61 alla risposta di uno *slave*;
5. **User Defined Frames:** messaggio completamente personalizzabile in fase di configurazione della rete. L’ID che viene assegnato a questo messaggio è 62 e può contenere qualunque informazione l’utente desideri;
6. **Reserved Frames:** messaggio con ID 63 e che non dovrebbe essere mai utilizzato in quanto questo è riservato per sviluppi futuri. [27]

2.3.4 Wakeup & Sleep

Come nel protocollo *FlexRay*, anche in *LIN* è previsto un meccanismo di spegnimento e accensione dinamico dei nodi per risparmiare energia. Un nodo *master* può forzare lo spegnimento dei nodi inviando un *Diagnostic Frame* (quindi con ID 60) e con il primo Byte tutto a 0. In questo modo, i nodi che ricevono questo messaggio capiscono che devono mettersi in uno stato di *sleep* (a basso consumo), rimanendo però in ascolto sul canale. Un altro modo affinchè i nodi entrino in modalità *sleep* è semplicemente lasciando il canale inattivo per un lungo periodo di tempo (solitamente 4 secondi).

Affinchè i nodi si risveglino, è necessario inviare una particolare sequenza di *wakeup*, forzando una particolare sequenza di bit 1 e 0, per un determinato numero di volte e per determinati intervalli di tempo. Una volta che i nodi ricevono questa particolare sequenza, capiscono che devono tornare operativi e ricominciano a svolgere i propri compiti.

2.4 Confronto tra i tre protocolli

Tutti e tre i protocolli presentano debolezze e punti di forza differenti, che li rendono più o meno adatti in diversi contesti e, per questa ragione, di solito vengono usati tutti e tre all'interno di una stessa auto. Un riassunto delle principali differenze può essere consultato in Tabella 2.2 e di seguito verranno illustrate alcune osservazioni sui protocolli.

Protocollo	LIN	CAN	FlexRay
Velocità MAX	20 Kbps	1 Mbps	10 Mbps
Costo	Basso	Moderato	Alto
Fili necessari	1	2	2-4

Tabella 2.2: Confronto tra i tre protocolli

2.4.1 LIN

Protocollo che richiede costi molto ridotti e permette di collegare in maniera molto semplice sensori e attuatori all'interno dell'auto. Avendo anche delle prestazioni ridotte, è molto utile per collegare e far funzionare i sottosistemi appartenenti alla categoria **body** (strumentazione di bordo, chiusura elettronica di porte e finestrini, ecc.) dove l'impiego di protocolli come *CAN* e *FlexRay* sarebbe "inutile" poiché gran parte della banda non verrebbe sfruttata e non sono richiesti alti livelli di affidabilità. Inoltre non è richiesto hardware specifico, per cui si possono benissimo utilizzare componenti standard. [31]

2.4.2 CAN

Protocollo che richiede bassi costi e permette di realizzare una rete di nodi in maniera facile e versatile. Utilizzato spesso come rete di *backbone* per collegare più nodi **master LIN** ed è molto utilizzato nei sottosistemi **powertrain** e **chassis**, poiché grazie alla sua ampiezza di banda è in grado di garantire latenze minori e permettere la gestione di sistemi più complessi rispetto a *LIN*. Il principale punto di forza è la semplicità del cablaggio necessario e l'affidabilità garantita, mentre il principale punto debole è la larghezza di banda ridotta, per cui non si possono realizzare sistemi avanzati. [31]

2.4.3 FlexRay

Protocollo che richiede costi più elevati e permette di realizzare una rete ad alte prestazioni. Anch'esso viene utilizzato come rete di *backbone* (come CAN) e nei sottosistemi **powertrain** ma, tuttavia, grazie alla sua grande ampiezza di banda e alla sua architettura a doppio canale, è in grado di gestire sistemi ancora più avanzati rispetto a CAN come i sistemi **ADAS** e **X-by-wire**, dove sono necessarie prestazioni molto elevate e un'affidabilità molto alta. Sebbene realizzare una rete *FlexRay* sia molto più costoso di una rete CAN, per via dell'hardware specifico necessario, è in grado di abbassare lo stesso i costi di produzione in quanto una singola rete *FlexRay* è in grado di sostituire più reti CAN, realizzando quindi una singola rete ad alte prestazioni piuttosto che molteplici a basse prestazioni. [31]

2.4.4 Sicurezza nei protocolli

Essendo che il campo di applicazione di questi protocolli ha dei requisiti di prestazioni e costi molto stringenti, non sono stati previsti meccanismi di sicurezza che impedissero attacchi da parte di malintenzionati. Per cui tutti e tre i protocolli, non avendo meccanismi di sicurezza e di autenticazione, sono vulnerabili a numerosi attacchi come ad esempio alcuni dei seguenti:

- **Eavesdropping:** un attaccante si collega alla rete e si mette in ascolto passivo dei messaggi che vengono inviati. In questo modo può essere in grado, in base ai messaggi che intercetta, di stabilire le abitudini del guidatore, di intercettare messaggi particolari che permettono di accendere il motore, di disattivare il blocco delle porte, di attivare o disattivare i freni (anche in moto), ecc;
- **Data manipulation:** un attaccante collegato alla rete è in grado di inviare e manipolare i messaggi sulla rete, generando messaggi che provocano risposte come lo sblocco della porta, abbassamento dei finestrini, avvio del motore, ecc. oppure messaggi che mostrano informazioni farlocche come una velocità sbagliata, un numero di RPM sbagliato, livello di carburante più basso o più alto, ecc;
- **DoS:** un attaccante collegato alla rete genera una quantità enorme di traffico con lo scopo di saturare la rete o una ECU in particolare con lo scopo di creare un disservizio.

[10]

CAPITOLO 3

Crittografia Post-Quantum

Questo capitolo ha lo scopo di introdurre brevemente gli strumenti crittografici utilizzati, ponendo particolare enfasi anche sui concetti che sono alla base di questi strumenti.

3.1 Problema dei computer quantistici

Un *computer quantistico* è un elaboratore che, a differenza di un "classico" computer basato su *transistor*, sfrutta la *meccanica quantistica* per svolgere operazioni sui dati. In particolare, sfrutta le *proprietà quantistiche* della materia (come ad esempio la sovrapposizione degli stati) e per questa ragione non utilizza come *unità base di informazione* i **bit** per rappresentare lo stato della materia ma i **qubit**, simile ai *bit* ma con la caratteristica di poter essere anche in due stati contemporaneamente (non come i *bit* che possono avere o solo valore 0 o solo 1). Questo perchè lo stato della materia (dal punto di vista quantistico) non sempre è in un solo stato e, quindi, per rispecchiare questa caratteristica è stato introdotto il *qubit*. [22]

Utilizzando un'architettura completamente diversa, è stato teorizzato un modello computazionale che descriva in maniera astratta un computer quantistico: la **Macchina di Turing quantistica**. Questa ha la caratteristica di essere equivalente ad una **Macchina di Turing classica** dal punto di vista del potere computazionale, tuttavia dal punto di vista delle "**prestazioni**" sembra che la macchina *quantistica* sia molto più veloce di una macchina classica e, nella pratica, si traduce in elaboratori che sono molto più veloci dei normali computer diventando una vera e propria minaccia per la sicurezza dei sistemi. Infatti, esiste un algoritmo chiamato **Algoritmo di Shor** che, se eseguito su un computer quantistico abbastanza potente, permette

di risolvere dei problemi che sono "difficili" per un normale computer in tempo *polinomiale* (tempo molto breve), ovvero eseguire la fattorizzazione di un numero intero e il calcolo del logaritmo discreto, i quali sono i **problemi** su cui si basano rispettivamente il cifrario **RSA** e il cifrario **ElGamal** e sono attualmente gli schemi crittografici più utilizzati. Tuttavia, per utilizzare questo algoritmo in maniera efficace c'è bisogno di un computer quantistico in grado di utilizzare un numero di **qubit** abbastanza alto e per il momento ancora non esistono computer del genere, ma ciò non toglie che in futuro possa essere realizzato ed essere in grado di *rompere* tutti i cifrari conosciuti basati sui problemi citati in precedenza. [22]

Al fine di correre subito ai ripari, il **NIST** (National Institute of Standards and Technology) ha subito aperto un *concorso* aperto a chiunque con lo scopo di standardizzare degli algoritmi a chiave pubblica in grado di **resistere all'attacco di un computer quantistico**, quindi che non sono basati su un problema risolvibile dall'*algoritmo di Shor*. Questi algoritmi prendono il nome di **Crittografia Post-Quantistica** e sono tutti basati su problemi che non sono risolvibili in maniera facile nemmeno da un computer quantistico. [22]

3.2 KEM

Un **Key Encapsulation Mechanism** è un algoritmo a chiave pubblica che permette di inviare in maniera sicura un segreto condiviso ad un interlocutore. Solitamente un *KEM* tra due interlocutori **A** e **B** funziona nel seguente modo:

1. **A** prende la chiave pubblica di **B** e la usa per generare il segreto da inviare (che custodirà in maniera sicura) e una *capsula* che contiene il segreto in maniera "cifrata";
2. **A** invia la *capsula* a **B**;
3. **B** utilizza la sua chiave privata per aprire la *capsula* e leggere il segreto inviato da **A**.

Un *KEM* può essere anche definito come un insieme di tre algoritmi:

- *Gen* (*n*) : prende in input *n*, anche detto parametro di sicurezza (definisce quanto devono essere lunghe le chiavi generate), e restituisce una coppia di chiavi *pk* e *sk*, rispettivamente pubblica e privata;
- *Encaps* (*pk*) : prende in input la chiave pubblica *pk* e restituisce un segreto *ss* e il relativo testo cifrato *ct* (la *capsula*);
- *Decaps* (*ct*, *sk*) : prende in input la *capsula* *ct* e la chiave privata *sk* e restituisce il segreto *ss*.

L'utilizzo più diffuso di un *KEM* è quello di scambiare una chiave di sessione all'interno di un **cifrario ibrido**, ovvero uno schema di cifratura che utilizza sia una parte **asimmetrica** che una **simmetrica**, quindi il *segreto* che viene incapsulato è un valore totalmente casuale (generato ad ogni chiamata di `Encaps()`) che viene utilizzato come chiave. Tuttavia, è possibile modificare leggermente l'algoritmo `Encaps()` facendogli prendere in input anche un secondo valore `s` scelto arbitrariamente, in modo tale da permettere anche uno **scambio di messaggi** e non solo uno **scambio di chiavi**. Un esempio di *KEM* che può essere utilizzato in questo modo è **RSA**, che permette di scambiare sia chiavi che messaggi in modo da adattarsi a più utilizzi. [18]

3.3 CRYSTALS-kyber

Uno dei tanti algoritmi *Post-Quantum* disponibili è l'algoritmo **CRYSTALS-kyber**, un *KEM* che basa la sua sicurezza su un problema noto per essere difficile anche per un elaboratore quantistico. L'algoritmo si presenta in tre versioni, ognuna delle quali si differenzia per il *parametro di sicurezza* (lunghezza della chiave) e per il "livello" di sicurezza in grado di garantire. Facendo un parallelo con il noto algoritmo di cifratura **AES**, prendendo in considerazione solo il livello di sicurezza garantito, si può dire che **kyber-512** offre un livello di sicurezza comparabile con **AES-128**, **kyber-768** con **AES-192** e **kyber-1024** con **AES-256**. [1]

Sebbene offra un ottimo livello di sicurezza, le prestazioni ne risentono in maniera abbastanza evidente. Infatti, nell'ipotesi di uno scenario di cifratura di una chat, utilizzando un algoritmo **non-quantum-safe** (non resistente all'attacco di un computer quantistico) come **ECDH** (Elliptic-Curve Diffie-Hellman, un *KEM* basato sul problema del logaritmo discreto) si può notare che è 2 volte più veloce, consuma circa 2 volte meno energia e genera un overhead sui dati circa 70 volte inferiore rispetto a **kyber**. [19]

3.3.1 Sicurezza del cifrario

Un cifrario, per essere considerato sicuro, deve possedere la proprietà di **indistinguibilità**, ovvero preso un *testo cifrato* non deve essere possibile stabilire a quale *testo in chiaro* corrisponda. Persa questa proprietà è ovvio che uno schema non può essere considerato sicuro in quanto è sempre possibile risalire al corrispondente *testo in chiaro* senza conoscere la chiave di cifratura.

Il modello che di solito si utilizza per dimostrare la sicurezza di un cifrario è un gioco dove un avversario ha due messaggi in chiaro e un testo cifrato corrispondente ad uno dei due messaggi. L'obiettivo dell'avversario, ovviamente, è quello di indovinare il testo in chiaro corrispondente e nel farlo può avere a disposizione più o meno potere computazionale per cercare di indovinarlo, come ad esempio la possibilità di ottenere dei testi cifrati a partire da testi in chiaro a piacere (ovviamente che non fanno parte dei messaggi in chiaro della sfida). Se l'avversario, grazie al potere a disposizione, è in grado di stabilire l'origine del testo cifrato con una buona probabilità, allora il cifrario **non è sicuro contro quell'avversario**, mentre se l'avversario non riesce ad estrapolare informazioni e l'unica cosa che riesce a fare è tentare a caso, allora **il cifrario è sicuro contro quell'avversario**. In base al potere fornito all'avversario si possono identificare tre livelli di sicurezza:

1. **IND-CPA** (*Chosen-Plaintext Attack*): l'attaccante ha accesso ad un **oracolo di cifratura** che gli permette di ottenere il corrispondente *testo cifrato* a partire da *testi in chiaro* a piacere che non siano quelli della sfida. Ovviamente l'oracolo utilizza la stessa chiave utilizzata per cifrare il *testo cifrato* della sfida e l'avversario non la conosce perché l'oracolo lavora come una **scatola chiusa**;
2. **IND-CCA1** (*Chosen-Ciphertext Attack*): oltre ad avere accesso all'**oracolo di cifratura**, l'attaccante ha anche accesso ad un **oracolo di decifratura** che gli permette di decifrare (sempre con la chiave utilizzata per la sfida) *testi cifrati* arbitrari. Tuttavia, nel momento in cui riceve il **cifrato di sfida**, non può più utilizzare l'**oracolo di decifratura** e deve scegliere uno dei due testi in chiaro di sfida;
3. **IND-CCA2** (*adaptive Chosen-Ciphertext Attack*): nella versione *adattiva*, l'unica differenza rispetto alla precedente è che adesso l'avversario può utilizzare l'**oracolo di decifratura** anche dopo aver ricevuto il **cifrato di sfida**.

È chiaro che l'avversario **IND-CCA2** è il più potente mentre l'avversario **IND-CPA** è il meno potente e, ovviamente, se un cifrario è sicuro nei confronti di un avversario con un certo potere sarà sicuro anche per quelli con potere minore. [12]

Per quanto riguarda il cifrario **CRYSTALS-kyber**, è stato dimostrato essere **IND-CCA2**-sicuro, per cui è in grado di resistere anche agli attacchi più sofisticati e agli attaccanti più potenti. [1]

3.3.2 Problema di riferimento

Il problema su cui si basa il cifrario **CRYSTALS-kyber** è chiamato **Learning With Errors (LWE)**, appartenente alla famiglia dei problemi **lattice-based** (basati sui reticolati ordinati). Sebbene sia un problema algebrico abbastanza complesso, l'idea di base è molto semplice: dato un sistema di equazioni lineari che descrivono il valore del segreto da condividere, questo si può nascondere aggiungendo degli errori al sistema. In questo modo, un sistema risolvibile in maniera facile diventa risolvibile in maniera difficile persino per un computer quantistico, anche con degli **errori piccoli**.

Tipicamente per rendere più efficiente la risoluzione del sistema vengono introdotti dei **numeri interi** come *errori*, in quanto le operazioni necessarie per la risoluzione sono implementabili in maniera molto efficiente. Un esempio di sistema a cui vengono aggiunti errori può essere visualizzato in Figura 3.1 [30]

$$\begin{pmatrix} 3 & 3 \\ 2 & 5 \\ 4 & 1 \end{pmatrix} \mathbf{s} = \begin{pmatrix} 6 \\ 7 \\ 5 \end{pmatrix}$$

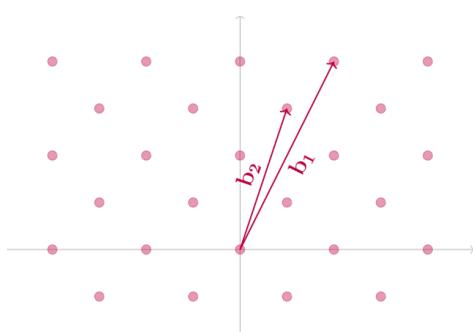
$$\begin{pmatrix} 3 & 3 \\ 2 & 5 \\ 4 & 1 \end{pmatrix} \mathbf{s} + \begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 6 \\ 6 \end{pmatrix}$$

Figura 3.1: Esempio di sistema senza errori (sinistra) e lo stesso ma con l'aggiunta di errori (destra)

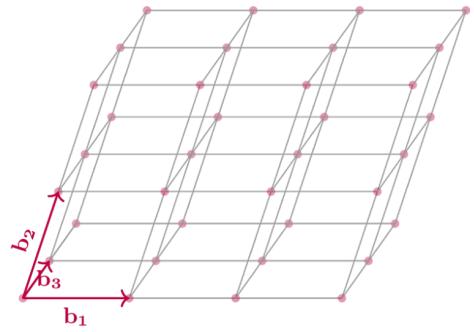
Il fatto che sia complesso anche per un computer quantistico deriva dal fatto che questo problema è possibile ridurlo ad un altro problema già noto essere difficile per un computer quantistico. Il problema in questione è il **CVP** (Closest Vector Problem) che è definito su una struttura geometrica chiamata reticolo, un insieme di punti che può essere descritto come una **griglia**. In pratica, definiti due o più vettori di base, il reticolo risultante è un insieme di "punti" ottenuti facendo combinazioni lineari **interne** (che utilizzano solo numeri interi) di questi vettori. Alcuni esempi di reticolati possono essere visti in Figura 3.2

A questo punto, il problema **CVP** consiste nel trovare il vettore del reticolo più vicino ad un vettore target definito all'esterno del reticolo. Come si può vedere in Figura 3.3, i punti rossi sono i vettori del reticolo, il vettore blu \mathbf{t} è il vettore target definito **all'esterno** del reticolo e il punto \mathbf{x} è il vettore **del reticolo** più vicino al vettore \mathbf{t} . [29]

Questo problema è molto semplice quando le dimensioni del reticolo sono due, tuttavia, se si prende in considerazione un reticolo con n dimensioni il problema diventa molto difficile e nemmeno la potenza di un computer quantistico consente di risolvere il problema in maniera efficiente [36].



(a) Esempio di reticolo a due dimensioni



(b) Esempio di reticolo a tre dimensioni

Figura 3.2: Alcuni esempi di reticoli

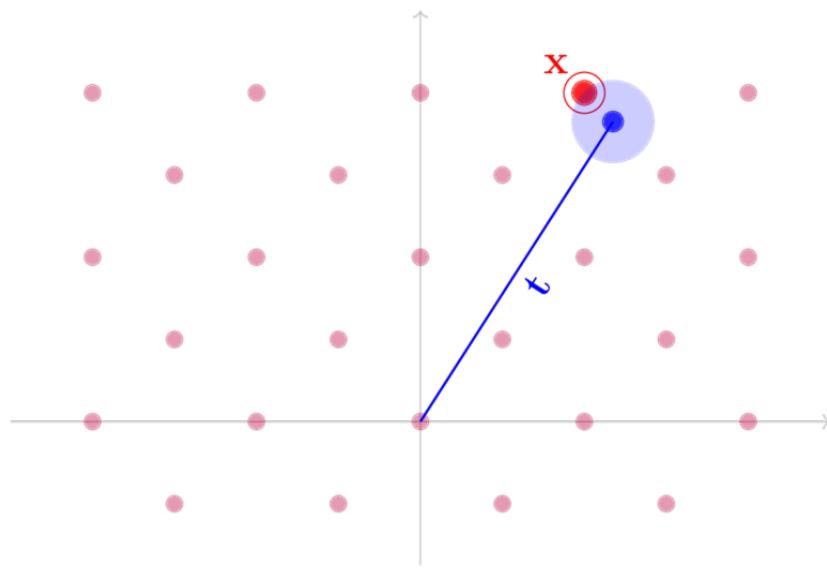


Figura 3.3: Esempio di problema CVP

3.4 Advanced Encryption Standard

Con il termine **AES** si intende la specifica per la protezione dei dati digitali stabilita dal **NIST** nel 2001 a seguito della volontà di voler sostituire **DES** (Data Encryption Standard), lo standard utilizzato fino a quel momento. Per definire l'algoritmo **AES** il **NIST** ha avviato un concorso aperto a chiunque dove si poteva proporre un algoritmo che doveva rispettare i requisiti imposti dall'ente. Al termine di questo concorso l'algoritmo vincitore è stato il **Rijndael**, un cifrario a blocchi che si presenta in tre versioni differenziate principalmente dalla lunghezza della chiave (e quindi dal livello di sicurezza che vogliono garantire).

Sostanzialmente un **cifrario a blocchi** è un **cifrario a chiave simmetrica** che non opera su un byte alla volta ma organizza il dato da cifrare in gruppi della stessa lunghezza e opera su un intero gruppo alla volta. Nel caso di *AES*, l'input viene suddiviso in gruppi di 128 bit e

laddove la lunghezza dell'input non dovesse essere un multiplo di 128 viene aggiunto un *padding* per riempire l'ultimo blocco.

Il vantaggio di utilizzare un **cifrario a blocchi** piuttosto che uno **stream cipher** (cifrario che cifra un bit alla volta invece che a gruppi), è la possibilità di implementarlo realizzando una struttura chiamata **rete a sostituzione e permutazione** (*SP-network*) [13] con la quale si possono facilmente garantire due caratteristiche *chiave* di ogni cifrario:

- **Confusione:** un cifrario dovrebbe fare in modo che ogni bit del testo cifrato deve dipendere da più bit della **chiave di cifratura**, in modo da oscurare quanto più possibile la relazione tra le due;
- **Diffusione:** un cifrario dovrebbe fare in modo che nel momento in cui cambia un singolo bit del testo in chiaro devono cambiare almeno la metà dei bit del testo cifrato, in modo tale da nascondere quanto più possibile le relazioni statistiche tra testo in chiaro e testo cifrato. [16]

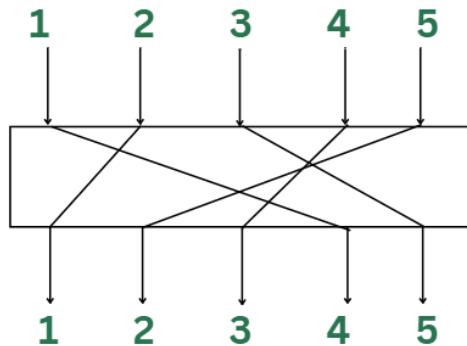
Il modo in cui si possono garantire queste proprietà è per mezzo dell'introduzione di due particolari strutture all'interno dell'algoritmo:

- **S-box** (Substitution box): è una tabella che definisce il modo in cui ogni sequenza di una determinata lunghezza deve essere sostituita. Una tipica *S-box* prende un byte, lo divide in due gruppi di 4 bit (chiamati solitamente **nibble**) e utilizza le due metà per identificare riga e colonna di una tabella. Il byte contenuto nella cella sarà quello che sostituirà il byte iniziale. Grazie a questa è possibile introdurre la **confusione**, in quanto quello che vuole realizzare una *S-box* è una relazione **non-lineare** tra input e output; [23]
- **P-box** (Permutation box): è un tabella che definisce il modo in cui devono essere permutati i bit di un una sequenza fornita in input. Grazie a questa è possibile introdurre la **diffusione**. [21]

Un esempio di **P-box** può essere visto in Figura 3.4a mentre un esempio di **S-box** in Figura 3.4b

3.4.1 Struttura di AES

L'algoritmo *Rijndael* è stato progettato seguendo lo schema di una **rete a sostituzione e permutazione** con dimensione del blocco fissa a 128 bit e lunghezza della chiave che può



(a) Esempio di una P-box

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

The column is determined by the least significant nibble, and the row by the most significant nibble. For example, the value $9a_{16}$ is converted into $b8_{16}$.

(b) S-box utilizzata dall'algoritmo Rijndael

Figura 3.4: Esempio di S-box e P-box

essere lunga 128 bit, 192 bit o 256 bit. La dimensione della chiave influisce sul **numero di round** (cicli) che vengono eseguiti sul testo in chiaro per generare il testo cifrato finale (la corrispondenza è illustrata in Tabella 3.1).

Lunghezza chiave	Numero di round
128	10
192	12
256	14

Tabella 3.1: Numero di round predefinito in AES

Il primo passo che esegue l'algoritmo è un espansione della chiave in modo da ottenere le chiavi da utilizzare per ogni round, mentre ogni round è composto principalmente da 4 operazioni: SubBytes, ShiftRows, MixColumns e AddRoundKey.

SubBytes

Lo scopo di questa operazione è quella di realizzare una sostituzione **non lineare** dei byte, per mezzo delle **S-box**, in modo da introdurre **confusione** ed evitare banali attacchi algebrici. Una visualizzazione dell'operazione può essere vista in Figura 3.5. [13]

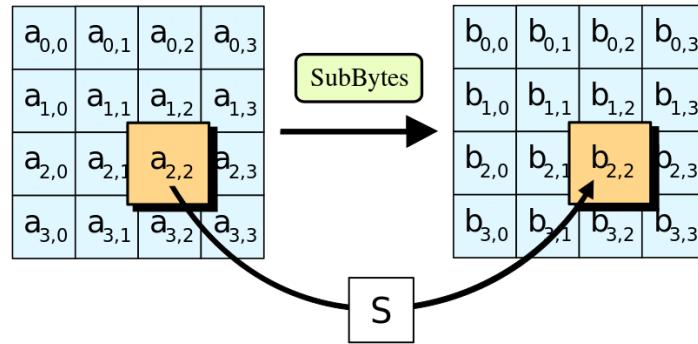


Figura 3.5: Schema dell’operazione *SubBytes*

ShiftRows

Lo scopo di questa operazione è quella di effettuare lo **shift** di ogni riga di un determinato offset, in modo da non cifrare le colonne in maniera indipendente degenerando in 4 cifrari a blocchi indipendenti (si introducerebbe una problematica simile a quella affrontata nella sottosezione 3.4.2 con la modalità *ECB*). In generale l’offset è 0 per la prima riga (rimane invariata) e aumenta di 1 ad ogni riga successiva, in questo modo ogni colonna conterrà byte appartenenti ad ogni altra colonna. Una visualizzazione dell’operazione può essere vista in Figura 3.6. [13]

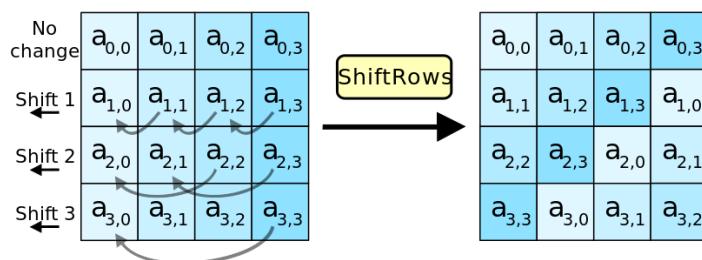


Figura 3.6: Schema dell’operazione *ShiftRows*

MixColumns

Lo scopo di quest’operazione è quella di combinare le varie colonne in maniera **lineare** e **reversibile**, facendo in modo che la combinazione dei 4 byte della colonna influiscano **sempre** su ogni byte della colonna di output. Insieme all’operazione di **ShiftRows**, quest’operazione introduce la **diffusione**. Una visualizzazione dell’operazione può essere vista in Figura 3.7. [13]

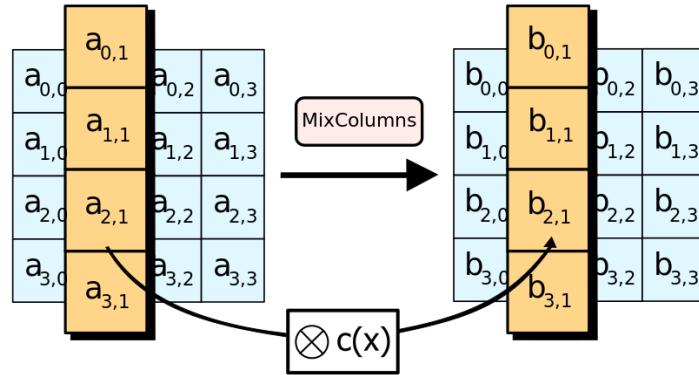


Figura 3.7: Schema dell’operazione *MixColumns*

AddRoundKey

Lo scopo di questa operazione è quella di combinare il blocco con la chiave del round, una chiave avente la stessa lunghezza del blocco e ottenuta a partire dalla **chiave di cifratura** utilizzata. In pratica la chiave di round viene organizzata nello stesso modo di un blocco e su ogni cella viene effettuato lo *XOR* tra cella del blocco e cella della chiave di round. Una visualizzazione dell’operazione può essere vista in Figura 3.8. [13]

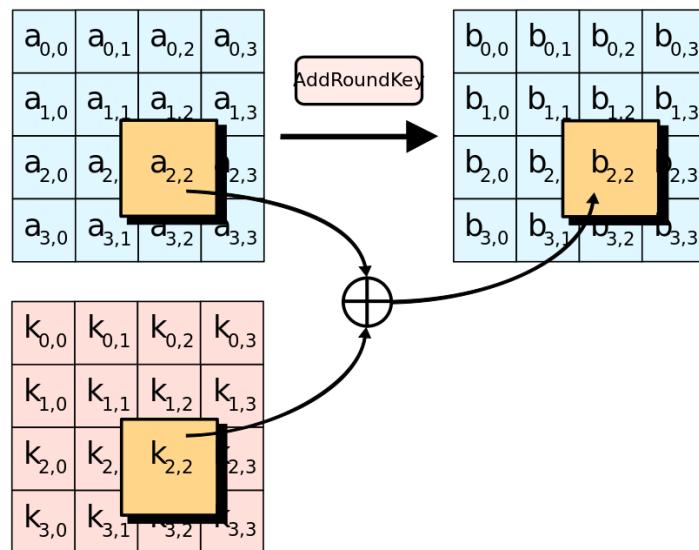


Figura 3.8: Schema dell’operazione *AddRoundKey*

3.4.2 Modalità di AES

Essendo un cifrario a blocchi, opera su un dato preso in input blocco per blocco. Sebbene l’elaborazione di un singolo blocco è uguale per ognuno dal momento che l’algoritmo applicato è sempre uguale, lo stesso non vale per un gruppo consecutivo di blocchi in quanto

questi possono essere elaborati in maniera diversa fornendo risultati completamente diversi. Si parla quindi di **modalità operative**, ovvero del modo in cui il cifrario elabora un input composto da più blocchi e le più diffuse sono le seguenti:

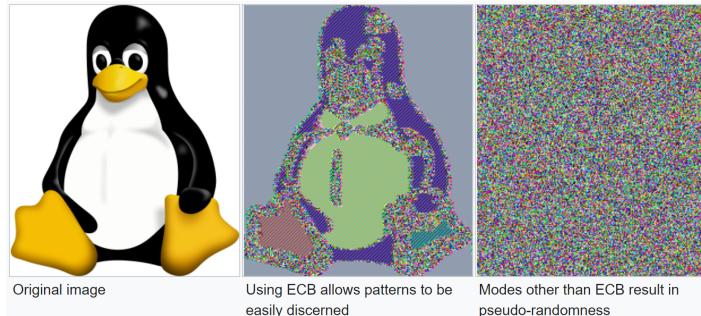


Figura 3.9: Problema con l'utilizzo della modalità *ECB*

- **ECB** (Electronic Codebook): modalità più semplice dove ogni blocco viene elaborato in maniera indipendente. Non più utilizzato in quanto un *testo in chiaro* viene codificato sempre con lo stesso *testo cifrato*, non vengono nascosti bene i pattern e non viene introdotta la **diffusione**. Uno schema che descrive il funzionamento si trova in Figura 3.10a mentre un esempio che mostra il problema evidenziato si trova in Figura 3.9;
- **CBC** (Cipher Block Chaining): modalità in cui ogni blocco contribuisce all'elaborazione del blocco successivo. In pratica, nel caso della cifratura, il testo cifrato ottenuto dal blocco precedente viene messo in *XOR* con il blocco attuale e il risultato viene dato in pasto alla cifratura mentre, nel caso della decifratura, il testo cifrato precedente viene messo in *XOR* con l'output della decifratura per ottenere il testo in chiaro. Essendo che per il primo blocco non esiste un blocco precedente, viene usato un blocco predefinito e noto chiamato **vettore di inizializzazione** che non fa parte dell'input ma viene aggiunto. Uno schema che descrive il funzionamento si trova in Figura 3.10b;
- **CFB** (Cipher Feedback): modalità molto simile alla *CBC*, dove nel caso della cifratura il testo in chiaro viene messo in *XOR* con l'output dello *XOR* tra cifratura del blocco precedente e testo in chiaro precedente per ottenere il testo cifrato, mentre nel caso della decifratura il testo cifrato viene messo in *XOR* con l'output della **cifratura** del blocco precedente. Anche in questo caso è necessario un **vettore di inizializzazione** e uno schema che ne descrive il funzionamento si trova in Figura 3.10c;
- **OFB** (Output Feedback): modalità dove nel caso della cifratura il testo in chiaro viene messo in *XOR* con l'output della cifratura del blocco precedente per ottenere il

testo cifrato, mentre nel caso della decifratura l’output della **cifratura** del blocco precedente per ottenere il testo in chiaro. Anche in

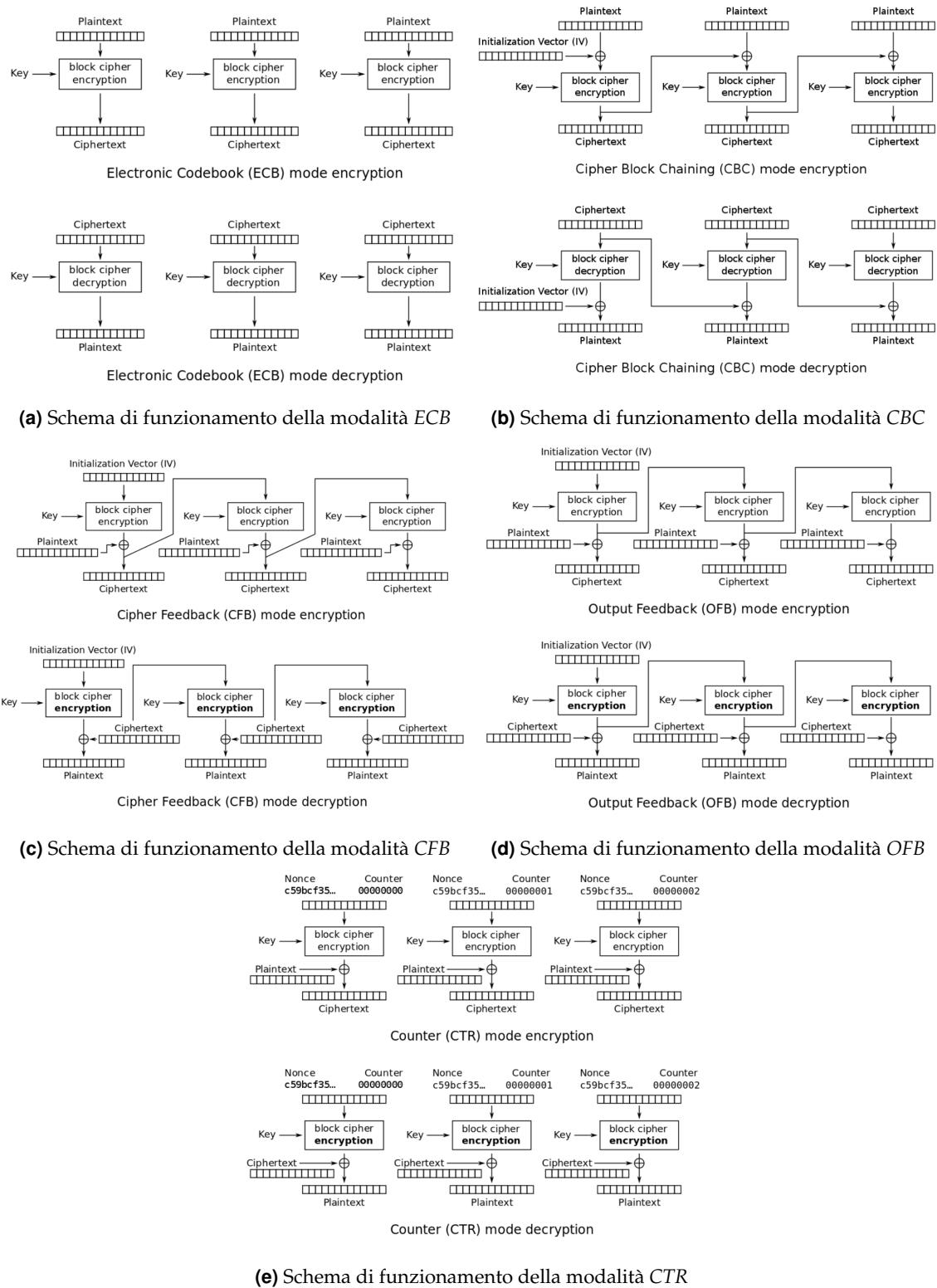


Figura 3.10: Modalità operative di AES

questo caso è necessario un **vettore di inizializzazione** e uno schema che ne descrive il funzionamento si trova in Figura 3.10d.

- **CTR** (Counter): modalità dove i blocchi vengono elaborati in maniera indipendente (simile a *ECB*). Per prima cosa viene generato un **nonce** (un numero casuale) al quale viene sommato un contatore che parte da 0 e viene incrementato man mano che si processano i blocchi. Il valore ottenuto viene dato in pasto alla cifratura e il risultato viene messo in *XOR* con il blocco per ottenere il testo cifrato. Anche nel caso della decifratura il risultato della somma viene dato in input alla **cifratura**, ma questo viene messo in *XOR* con il testo cifrato per ottenere il testo in chiaro. Uno schema che ne descrive il funzionamento si trova in Figura 3.10e. [14]

Un confronto che si può fare tra le varie modalità riguarda la possibilità di parallelizzare cifratura e decifratura e la possibilità di effettuare l'accesso casuale in lettura ad un blocco dell'input. Una tabella riassuntiva di tale confronto può essere consultata in Tabella 3.2 [14]

Modalità	ECB	CBC	CFB	OFB	CTR
Cifratura parallelizzabile?	si	no	no	no	si
Decifratura parallelizzabile?	si	si	si	no	si
Accesso casuale	si	si	si	no	si

Tabella 3.2: Confronto tra le varie modalità operative

3.4.3 Sicurezza del cifrario

Un algoritmo di cifratura è considerato sicuro se il miglior algoritmo utilizzabile per "romperlo" è l'algoritmo di **forza bruta**, ovvero vengono provate tutte le possibili chiavi in sequenza fino ad indovinare quella corretta. AES, fino ad oggi, è ancora considerato sicuro in quanto non è stato ancora trovato nessun attacco realizzabile nei suoi confronti anche se sono stati **teorizzati** dei possibili attacchi. Uno degli attacchi più famosi in letteratura è chiamato **XSL Attack** (eXtended Sparse Linearization), il quale cerca di sfruttare una debolezza di AES dovuta alla scarsa complessità delle componenti **non lineari** ma è stato dimostrato che questo attacco non è praticabile, quindi rimane sicuro anche rispetto a questo attacco. Un altro attacco molto famoso è l'attacco **biclique**, una variante del **Meet-In-The-Middle** (un attacco che sfrutta combinazioni di testi in chiaro e cifrati per ricavare la chiave di cifratura) che è stato dimostrato essere 4 volte più veloce di un attacco a forza bruta. Tuttavia, analizzando il

guadagno ottenuto in termini di **parametro di sicurezza** quello che si ottiene è che invece di ottenere una sicurezza a 128 bit (per AES-128) si ottiene una sicurezza di 126 bit (2 bit in meno anche per le altre varianti di AES), che è ancora un ottimo livello di sicurezza in quanto per concludere con successo un attacco del genere sono ancora richiesti **miliardi di anni**. Quindi anche nei confronti di quest'ultimo attacco è considerato ancora sicuro. [13]

Inoltre, è stato recentemente dimostrato che AES è anche **Quantum-safe** ed è quindi persino in grado di resistere ad un attacco di un computer quantistico. Ovviamente, essendo un computer quantistico più potente, il livello di sicurezza offerto dalle varie versioni dell'algoritmo sarà minore rispetto a quelle offerte per i computer classici. Un confronto, in termini di livello di sicurezza offerto, tra computer classici e quantistici può essere visto in Tabella 3.3. [34]

Algoritmo	Lunghezza chiave	Sicurezza classica	Sicurezza quantum
AES-128	128 bit	128 bit	64 bit
AES-256	256 bit	256 bit	128 bit

Tabella 3.3: Sicurezza offerta da AES per computer classici e quantistici

CAPITOLO 4

Soluzione Proposta

In questo capitolo vengono discusse le motivazioni dietro la scelta del protocollo *CAN*, nonché delle problematiche riscontrate nel protocollo e di come può essere possibile risolverle

4.1 Scelta del protocollo

Al fine di procedere con il lavoro di messa in sicurezza è stato necessario scegliere uno dei protocolli *Automotive* disponibili in letteratura. La prima scelta è stata *FlexRay* in quanto è il protocollo più promettente e con hardware più prestante ma, sfortunatamente, dopo un'attenta ricerca in rete non è stata trovata nessuna implementazione software del protocollo. Erano disponibili solo all'acquisto delle *board di testing* (con costi anche abbastanza sostanziosi) che permettevano di simulare il protocollo tramite strumentazione adeguata. Non avendo avuto fortuna con *FlexRay*, la seconda scelta è ricaduta su *LIN*. Purtroppo anche qui non è stato possibile trovare implementazioni **funzionanti** e **complete** del protocollo, ma solo tentativi abbandonati di implementarlo. Per queste ragioni, infine, si è scelto il protocollo *CAN* che è supportato in maniera nativa dal kernel **Linux** tramite dei moduli appositi (oltre ad avere anche diverse implementazioni software complete e funzionanti), permettendo persino la creazione di uno o più nodi virtuali.

4.2 Problematiche di sicurezza riscontrate

Nonostante CAN sia uno dei protocolli più utilizzati nell’ambito *Automotive*, questo presenta gravi vulnerabilità di sicurezza. Soprattutto, i problemi principali sono due:

- Assenza di qualunque meccanismo di **protezione del payload** dei messaggi, lasciandolo vulnerabile a modifiche maliziose e all’intercettazione;
- Assenza di un qualunque meccanismo di **autenticazione** dei nodi, permettendo a chiunque si connetta alla rete di inviare e leggere dati, permettendo anche l’invio di **messaggi falsi** e l’invio di un grande numero di messaggi *spazzatura* con lo scopo di disturbare la rete o renderla **non disponibile**.

Questa situazione è originata dal fatto che i creatori del protocollo hanno posto l’attenzione soprattutto sulle **prestazioni**, tralasciando gli aspetti di sicurezza. Il motivo dietro questa scelta è che, essendo utilizzato in un ambiente *safety-critical*, introducendo dei ritardi nella comunicazione si generano dei rischi all’incolumità delle persone all’interno delle auto in cui è utilizzato CAN, nonchè di quelle nelle vicinanze, sfociando così in una situazione **inaccettabile**. Se si pensa ad un sistema **Steer-by-wire**, nel quale una *ECU* controlla la rotazione delle ruote in base al movimento dello sterzo, un ritardo tra movimento dello sterzo e rotazione *effettiva* delle ruote potrebbe tramutarsi in un impatto con un oggetto (edificio, veicolo, ecc.) o persona.

4.3 Soluzione proposta

Per cercare di risolvere il problema della mancanza di **protezione del payload** all’interno del protocollo CAN, quello che si è voluto proporre è l’**aggiunta di uno strato di cifratura** che protegga lo scambio di messaggi sulla rete. Una soluzione abbastanza banale può essere quella di decidere, in fase di configurazione della rete, una chiave di cifratura e utilizzare **sempre** quella per inviare messaggi cifrati, tuttavia, una soluzione del genere è inaccettabile per molti motivi, tra cui:

- Per poter utilizzare questa chiave è necessario che essa venga salvata, per cui è necessario utilizzare hardware apposito in maniera tale da salvarla **in modo sicuro**;
- Considerando che la vita media di un veicolo può superare i 10 anni, usare la stessa chiave così a lungo diventa problematico, in quanto, durante un periodo di tempo così

lungo la rottura della chiave può essere plausibile e questo rende completamente vano l'utilizzo della cifratura, anche per messaggi già inviati in precedenza;

Quindi è necessario trovare un modo che permetta di **non dover salvare la chiave** e che consenta un **aggiornamento della chiave da utilizzare**, in modo tale da evitare modifiche alle ECU (rendendo la soluzione **retrocompatibile** con le automobili già in commercio) e far sì che in caso di violazione della chiave sia compromessa solo una sessione e non tutte.

Il modo migliore per garantire queste due caratteristiche è l'utilizzo di un **cifrario ibrido**, ovvero un cifrario che ha una componente asimmetrica per far accordare i nodi su una stessa chiave di sessione e che ha una componente simmetrica con la quale viene utilizzata la chiave di sessione come chiave di cifratura per i messaggi che devono essere scambiati. Uno schema riassuntivo di ciò che vuole introdurre la soluzione si può consultare nella Figura 4.1.

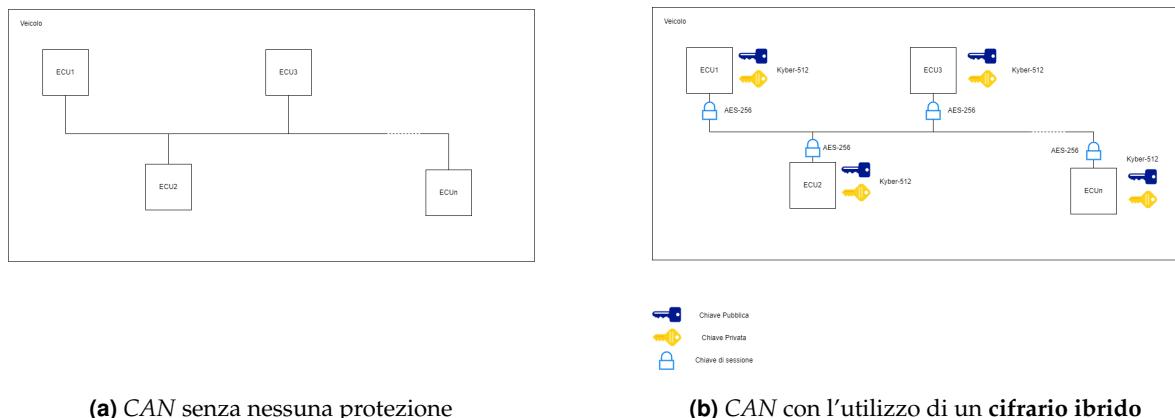


Figura 4.1: Schema di CAN senza e con l'aggiunta di un **cifrario ibrido**

4.3.1 Scelta degli algoritmi di cifratura

Per poter realizzare il **cifrario ibrido** citato in precedenza, è necessario scegliere due algoritmi di cifratura: uno *simmetrico* e uno *asimmetrico*. In particolare, si è deciso di impiegare cifrari appartenenti alla categoria **Post-Quantum** con il fine di garantire la massima sicurezza contro la maggior parte degli attacchi noti in letteratura e garantire anche la sicurezza contro un attacco da parte di un computer quantistico.

In seguito a questa decisione, quindi, si è deciso di impiegare come algoritmo *simmetrico* il cifrario **AES-256** mentre come algoritmo *asimmetrico* il KEM **CRYSTALS-kyber-512**.

4.3.2 Funzionamento della soluzione

Una delle caratteristiche che si vogliono garantire con l'introduzione del **cifrario ibrido**, come citato in precedenza, è la **retrocompatibilità** con le *ECU* che non adottano la soluzione, in modo tale da estendere la soluzione proposta a quante più auto possibile.

Il modo migliore per garantire questa caratteristica è quella di **non modificare il protocollo**, senza alterarne né la struttura né i messaggi scambiati e senza applicare modifiche all'hardware necessario. Quindi, l'aggiunta del cifrario non può avvenire a livello di protocollo (implicherebbe una modifica a *CAN*) ma deve avvenire a livello di **applicazione**. L'idea è quella di far sì che l'applicativo sull'*ECU* effettui la cifratura del payload **prima** che questo venga inviato come messaggio tramite *CAN*. In questo modo, il payload che viaggia *in chiaro* è in realtà un payload cfrato che verrà decifrato dalla *ECU* interessata, senza il rischio che un attaccante possa fare **eavesdropping** del messaggio.

Per poter realizzare un sistema del genere, è necessario creare un **protocollo** che si occupi di:

- Generare una coppia di chiavi;
- Scambiare una chiave di sessione;
- Cifrare un messaggio;
- Decifrare un messaggio.

Tutto questo, ovviamente, senza aggiungere nuove tipologie di messaggi e senza alterare gli ID *CAN* disponibili, in modo da non causare incompatibilità con *CAN*. Quindi, un possibile protocollo che riesca ad occuparsi di tutti i compiti citati può essere la seguente sequenza di passi:

1. Generazione di una coppia di chiavi *asimmetriche*;
2. Scambio della chiave pubblica;
3. Generazione di una chiave di sessione;
4. Scambio e salvataggio della chiave di sessione;
5. Cifratura e decifratura del traffico.

Un sequence diagram che chiarisce il funzionamento può essere visto in Figura 4.2.

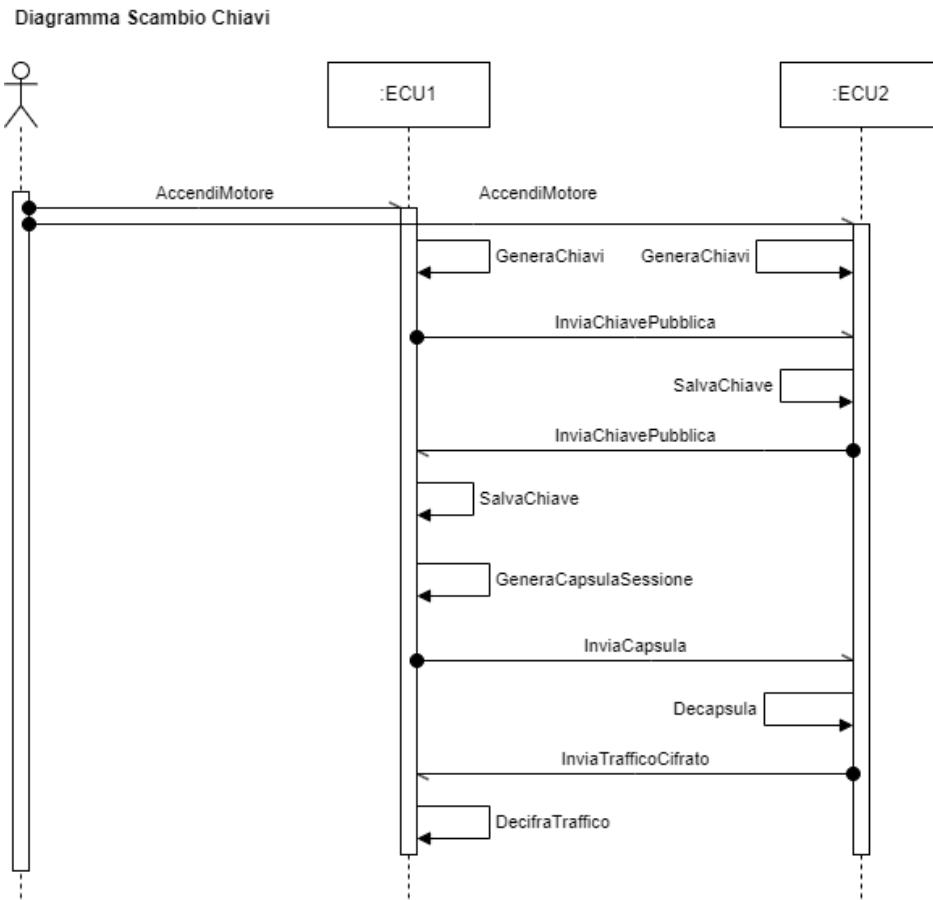


Figura 4.2: Sequence Diagram del protocollo proposto

Riuscendo ad implementare il protocollo proposto senza dover necessariamente alterare caratteristiche di CAN, permetterebbe di riuscire a **proteggere il payload** e raggiungere l’obiettivo fissato per il lavoro.

CAPITOLO 5

Implementazione

In questo capitolo verrà illustrato come è stato possibile implementare la soluzione proposta e verranno illustrati anche i risultati che sono stati ottenuti

5.1 Sistema realizzato

Al fine di valutare le prestazioni e la fattibilità della soluzione proposta, è stato realizzato un applicativo molto elementare che realizza le operazioni crittografiche e utilizza CAN per inviare dati e messaggi. Il codice contenente l'applicativo, le dipendenze e un tester può essere consultato nella repository al seguente indirizzo: <https://github.com/lorycris99/tesi-magistrale>.

Per l'implementazione è stato scelto di utilizzare il linguaggio **C** sia perchè le implementazioni di **CAN** e delle primitive crittografiche utilizzate sono disponibili principalmente in questo linguaggio e sia per via della sua efficienza, in modo da avere una stima delle prestazioni più "realistica" possibile (solitamente anche gli applicativi reali sono scritti in **C**). Inoltre, per effettuare la compilazione dell'applicativo è necessario tener conto di alcuni accorgimenti:

- Il sistema operativo su cui è stato realizzato e testato è **Ubuntu**, per cui su sistemi diversi come **Windows** non sarà possibile utilizzarlo;
- Dal momento che l'applicativo necessita di dipendenze che non sono presenti nei percorsi di default del **Linker**, è necessario istruirlo su dove trovare le varie librerie.

Osservando il Codice 5.1 si può vedere il comando da inviare per compilare correttamente, dove con `-L/path/to/lib` si indica la cartella dove si trovano le librerie che verranno indicate e con `-l:lib.so` si indica il nome della libreria da includere;

- Oltre ad istruire il *Linker*, è necessario istruire anche il **Loader** poichè anche se la compilazione va a buon fine, il *Loader* non troverà le librerie specificate al *Linker*. Il comando da eseguire è mostrato nel Codice 5.2 e il **path** da specificare (ovviamente) deve essere lo stesso di quello specificato al **Linker**.

```
1  gcc test.c ../kyber/ref/randombytes.c -L/path/to/lib -l:
2    libpqcrystals_kyber512_ref.so -l:libpqcrystals_aes256ctr_ref.so -l:
3    libpqcrystals_fips202_ref.so
```

Codice 5.1: Istruzioni per il Linker

```
1  export LD_LIBRARY_PATH=/path/to/lib
```

Codice 5.2: Istruzioni per il Loader

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main(int argc, char* argv[]) {
6
7      if(argc != 2) {
8          printf("USAGE:\n");
9          printf("%s start/stop\n\n", argv[0]);
10         return -1;
11     }
12
13     if (!strcmp(argv[1], "start")) {
14         printf("Starting vcan0 interface\n");
15         //Create virtual CAN interface
16         system("sudo ip link add dev vcan0 type vcan");
17         //Bring the virtual CAN interface up
18         system("sudo ip link set up vcan0");
19     } else if(!strcmp(argv[1], "stop")){
20         printf("Stopping vcan0 interface\n");
21         //Delete virtual CAN interface
22         system("sudo ip link del vcan0");
23     } else {
24         printf("%s: command not found\n\n", argv[1]);
25     }
26 }
```

Figura 5.1: Codice sorgente del file can.c

Una volta istruiti **Linker** e **Loader** su dove trovare le librerie, quello che bisogna fare è creare un nodo CAN con cui poter comunicare. Per essere in grado di farlo, bisogna innanzitutto abilitare il **modulo del kernel** interessato lanciando il comando `sudo modprobe`

vcan, successivamente basta compilare ed eseguire il file `can.c` per avviare il nodo CAN virtuale. Il codice del file è illustrato nella Figura 5.1.

5.1.1 Protocollo realizzato

Poichè quello che si sta cercando di implementare non è presente **nativamente** in CAN, c'è bisogno di una strategia per fare in modo da inserire all'interno del protocollo un metodo per lo scambio di chiavi e per richiedere eventualmente una nuova chiave. Essendo che CAN lavora principalmente con gli **ID** dei messaggi, la strategia più conveniente è stata quella di utilizzare degli **ID inutilizzati dal protocollo** e assegnarli a dei messaggi *personalizzati* che svolgono le operazioni aggiuntive che si vogliono implementare. Sfortunatamente, non esistono degli ID inutilizzati **universalmente** in quanto ogni casa automobilistica gestisce gli **ID** a proprio piacimento per cui bisognerebbe adattare gli **ID** caso per caso.

Ai fini delle simulazioni realizzate, si sono presi degli **ID bassi** (solitamente sono i meno utilizzati) e si è immaginato che non corrispondano a messaggi già assegnati. Gli **ID** utilizzati sono stati i seguenti:

- 111: messaggio che indica la presenza di una parte della **chiave pubblica** nel payload;
- 112: messaggio per richiedere ad un nodo di reinviare la **chiave pubblica**;
- 122: messaggio che indica la presenza di una parte della **capsula** nel payload;
- 123: messaggio per richiedere ad un nodo di reinviare la **capsula**;
- A: messaggio che serve ad innescare la generazione di traffico casuale, utilizzato per valutare la correttezza dell'applicativo e delle operazioni di crittografia;
- B: messaggio generato casualmente e cifrato tramite **AES-256**, in risposta all'**ID A**;
- 0: messaggio che fa terminare l'applicativo;

A questo punto, il prossimo passo è stato quello di implementare il protocollo ma, tuttavia, sono sorte delle problematiche da risolvere: la lunghezza della **chiave pubblica** e della **capsula**. Un messaggio CAN è in grado di inviare fino a 8 Byte di dati mentre una chiave pubblica di **kyber-512** è lunga 800 byte e una *capsula* è lunga 768 byte, per cui la soluzione più semplice è sembrata quella di dividere entrambi in gruppi da 8 byte (avendo entrambi una lunghezza multipla di 8) e non inviare un solo messaggio ma inviarne 100 per la **chiave pubblica** e 96 per la **capsula**. A tal proposito, il protocollo ideato è il seguente:

1. Invio di un messaggio contenente **ID** che specifica l'intenzione e nel payload un identificatore del nodo che lo invia;
2. Se necessario, invio di uno o più messaggi contenenti lo stesso **ID** e nel payload il dato che si vuole inviare.

L'invio di un identificatore come primo messaggio aiuta a distinguere chi è che ha inviato un dato, in quanto nel momento in cui viene inviata una **chiave pubblica** non è possibile capire chi l'abbia inviata (essendo una rete *broadcast*) e quindi a chi associarla per inviargli, successivamente, una capsula contenente la chiave di sessione.

5.1.2 Struttura del codice

L'applicativo utilizzato per testare la soluzione è stato realizzato **ex novo** ed è stato organizzato in funzioni, ognuna delle quali assolve ad un compito specifico. Quelle principali sono:

- `encrypt()`: si occupa di realizzare la cifratura di una stringa presa in input utilizzando **AES-256** in modalità *CTR*;
- `decrypt()`: si occupa di realizzare la decifratura di una stringa presa in input utilizzando **AES-256** in modalità *CTR*;
- `sendkey()`: si occupa di inviare la propria chiave pubblica, ottenuta con il cifrario **CRYSTALS-kyber-512**, utilizzando la rete *CAN*;
- `receivekey()`: permette di ricevere una chiave pubblica in arrivo dalla rete *CAN* e la salva in un'array di chiavi "note", chiamato `publicKeys` e nella posizione corrispondente all'**ID** di chi ha inviato il messaggio (nel protocollo è sempre il primo messaggio);
- `sendCipherText()`: si occupa di inviare una capsula contenente un segreto da condividere sulla rete *CAN*. La capsula viene creata nel main in seguito alla generazione di un dato casuale e, prima di inviare la capsula, viene salvato il dato in un array chiamato `sharedKeys()` e poi viene richiamata la funzione in questione passandogli la capsula;
- `receiveCipherTextAndSharedSecret()`: si occupa di ricevere una capsula dalla rete *CAN*, la decapsula sfruttando la propria chiave privata e successivamente la salva in un array di chiavi di sessione chiamato `sharedKeys()` nella posizione corrispondente all'**ID** di chi ha inviato il messaggio;

- `cangen()`: si occupa di generare traffico casuale cifrato oppure no (in base ad un parametro). Prima che venga generato traffico, viene controllato se in corrispondenza dell'ID che ha richiesto il traffico è presente una chiave di sessione, se questa non è presente viene inviata una **richiesta di inviare un testo cifrato** (ID 123) mentre se è presente, viene inviato prima il proprio ID (per rispettare il protocollo), poi viene inviato il numero di messaggi che saranno mandati (per aiutare l'interlocutore a sincronizzarsi) e successivamente vengono inviati i messaggi cifrati con la chiave di sessione;
- `receiveCangen()`: si occupa di ricevere il traffico generato sulla rete CAN e decifrarlo se necessario;
- `intToHex()`: funzione ausiliaria che si occupa di convertire un numero **intero** in un numero **esadecimale**;
- `hexToInt()`: funzione ausiliaria che si occupa di realizzare l'inverso di `intToHex()`
- `main()`: funzione principale che realizza l'applicativo e si occupa di generare la coppia di chiavi pubblica e privata e di richiamare le funzioni corrette alla ricezione di un determinato ID CAN. Si occupa anche di effettuare controlli (come quello precedente alla chiamata di `cangen()`) e operazioni ausiliarie.

5.2 Librerie utilizzate

Per realizzare tutte le funzionalità dell'applicativo con relativa cifratura e comunicazione tramite protocollo CAN, è stato necessario utilizzare delle librerie esterne (sempre in linguaggio C) che implementassero le operazioni richieste. Le librerie utilizzate sono:

- CAN-utils;
- Kyber;
- OpenSSL.

5.2.1 CAN-utils

Il cuore dell'intero progetto, è la libreria (come si intuisce dal nome) che realizza le primitive di comunicazione tramite protocollo CAN. Inoltre, questa lavora in sinergia con il sottosistema **CAN per Linux** (chiamato **SocketCAN**) offrendo moltissimi strumenti per interfacciarsi con tale modulo. Affinchè si possa utilizzare correttamente, bisogna creare

un’interfaccia CAN virtuale (utilizzando il file `can.c` come visto in Figura 5.1) e successivamente utilizzare le funzioni specificando il nome dell’interfaccia (all’interno del progetto è stato utilizzato il nome `vcan0`). Nonostante sia completo di molte funzioni, all’interno del progetto è stata utilizzata solo una piccola parte di queste:

- `cangen`: funzione che permette di generare del traffico in maniera totalmente casuale o casuale ma in base a dei parametri (ID fissato, lunghezza del payload fissata, ecc.);
- `candump`: funzione che permette di leggere il traffico inviato ad un’interfaccia CAN virtuale, visualizzando in maniera semplice l’ID, la lunghezza del payload e il payload;
- `cansend`: funzione che permette di inviare un singolo messaggio CAN specificando tutti i parametri compreso il payload.

5.2.2 Kyber

Per aggiungere al progetto il cifrario **Kyber-512**, è stata utilizzata la libreria ufficiale realizzata dal gruppo **pq-crystals**, autori del cifrario in questione e di altri (sempre **Post-Quantum**). Questa offre tutte le funzionalità del KEM **kyber** rendendo disponibile anche una versione dell’implementazione ottimizzata per i processori realizzati da **Intel** (quindi con architettura chiamata **x86**) ma, tuttavia, è stata utilizzata l’implementazione di riferimento e non quella ottimizzata, poichè una *ECU* potrebbe utilizzare un processore con architettura differente e quindi i risultati potrebbero non essere realistici. Per poterla utilizzare all’interno del progetto, è stato necessario creare delle **librerie condivise** (seguendo la guida fornita con l’implementazione) che, spiegato in maniera molto semplice, sono dei file compilati in maniera tale da poter essere utilizzati in altri progetti in maniera del tutto indipendente.

Tra le funzioni offerte, quelle utilizzate sono:

- `pqcryptals_kyber512_ref_keypair()`: funzione che permette di creare una copia di chiavi, una pubblica e una privata, in modo da poter utilizzare il cifrario (Figura 5.2a);
- `pqcryptals_kyber512_ref_enc()`: funzione che genera un dato casuale chiamato *shared secret* e lo **incapsula** all’interno di una capsula chiamata *ciphertext* utilizzando una **chiave pubblica** fornita in input (Figura 5.2b);
- `pqcryptals_kyber512_ref_dec()`: funzione che **decapsula** lo *shared secret* dalla capsula utilizzando la **chiave privata** correlata alla chiave pubblica utilizzata per creare la capsula (Figura 5.2c).

<pre> 515 printf("GENERO CHIAVIN\n"); 516 test = pqcrystals_kyber512_ref_keypair(pk, sk); 517 518 for (int i = 0; i < N_ECU; i++) { 519 for (int j = 0; j < pqcrystals_kyber512_BYTIES; j++) { 520 sharedKeys[i][j] = 0; 521 } 522 } </pre>	<pre> 602 //ID = 123 603 case 291: 604 printf("REQUEST CIPHER\n"); 605 pqcrystals_kyber512_ref_enc(ct, ss, publicKeys[id]); 606 for (int i = 0; i < pqcrystals_kyber512_BYTIES; i++) { 607 sharedKeys[id][i] = ss[i]; 608 } 609 sendCipherText(ct, fp); 610 break; </pre>
(a) Utilizzo della funzione keypair	(b) Utilizzo della funzione enc
<pre> 255 printf("DECAPS\n"); 256 pqcrystals_kyber512_ref_dec(sharedKeys[id], cipherText, sk); 257 258 printf("SHARED KEY:"); 259 for (int i = 0; i < pqcrystals_kyber512_BYTIES; i++) { 260 printf(" %u", sharedKeys[id][i]); 261 } </pre>	
(c) Utilizzo della funzione dec	

Figura 5.2: Utilizzo delle funzioni di **Kyber**

Sebbene le funzionalità offerte siano complete e funzionanti, questa libreria ha una mancanza che non permette di sfruttare le piene potenzialità di **kyber**, ovvero si limita ad implementare solo la funzionalità *KEM* e quindi facendolo lavorare solo con dati casuali e non scelti. È una limitazione importante poichè **kyber**, analogamente a **RSA**, può essere usato anche come **cifrario a chiave pubblica** (cifrando messaggi arbitrari **scelti**) e non solo come *KEM* per lo scambio di **chiavi di sessione** (quindi con messaggi casuali generati **ad-hoc**). Tuttavia, all'interno del progetto questa limitazione non ha avuto nessun impatto sull'applicativo realizzato, ma ha semplicemente reso la fase di testing della correttezza dei messaggi leggermente più complicata.

5.2.3 OpenSSL

Per includere nel progetto il cifrario **AES** è stata utilizzata una libreria della *suite OpenSSL*, una raccolta di strumenti crittografici completa, efficiente e inclusa nativamente all'interno del kernel **Linux**. Molto potente e utilizzabile nativamente su sistemi operativi basati su **Linux**, questa raccolta offre moltissimi cfrari oltre **AES** e, in merito a quest'ultimo, fornisce anche l'implementazione di tutte le sue modalità operative. Infatti, nel progetto è stata utilizzato **AES-256** in modalità **CTR**, in quanto nella libreria di **kyber** viene fatto riferimento a questa e, per evitare eventuali problemi di compatibilità, è stata usata tale modalità.

Per poter essere in grado di sfruttare la libreria basta semplicemente includere le librerie di OpenSSL (`evp.h`, `conf.h` e `err.h`) e, successivamente, realizzare una o più funzioni che richiamino le operazioni di cifratura e decifrazione. All'interno del progetto sono state realizzate due fuzioni:

- `encrypt()`: funzione che si occupa di preparare **AES** per la cifratura leggendo i

parametri forniti (chiave di cifratura, nonce e lunghezza del testo da cifrare) e di effettuare la cifratura del **testo in chiaro**, come mostrato in Figura 5.3a.

- `decrypt()`: funzione che si occupa di preparare AES per la decifratura leggendo i parametri forniti (come per la cifratura) e di effettuare la decifratura del **testo cifrato**, come mostrato in Figura 5.3b.

<pre> 786 int encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *key, 787 unsigned char *iv, unsigned char *ciphertext) 788 { 789 EVP_CIPHER_CTX *ctx; 790 791 int len; 792 793 int ciphertext_len; 794 795 /* Create and initialise the context */ 796 if(!(ctx = EVP_CIPHER_CTX_new())) 797 handleErrors(); 798 799 /* 800 * Initialise the encryption operation. IMPORTANT - ensure you use a key 801 * and IV size appropriate for your cipher 802 * In this example we are using 256 bit AES (i.e. a 256 bit key). The 803 * IV size for "most" modes is the same as the block size. For AES this 804 * is 128 bits 805 */ 806 if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_256_ctr(), NULL, key, iv)) { 807 handleErrors(); 808 } 809 810 /* Provide the message to be encrypted, and obtain the encrypted output. 811 * EVP_EncryptUpdate can be called multiple times if necessary 812 */ 813 if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len)) { 814 handleErrors(); 815 } 816 ciphertext_len = len; 817 818 /* 819 * Finalise the encryption. Further ciphertext bytes may be written at 820 * this stage. 821 */ 822 if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len)) { 823 handleErrors(); 824 } 825 ciphertext_len += len; 826 827 /* Clean up */ 828 EVP_CIPHER_CTX_free(ctx); 829 830 return ciphertext_len; 831 }</pre>	<pre> 834 int decrypt(unsigned char *ciphertext, int ciphertext_len, unsigned char *key, 835 unsigned char *iv, unsigned char *plaintext) 836 { 837 EVP_CIPHER_CTX *ctx; 838 839 int len; 840 841 int plaintext_len; 842 843 /* Create and initialise the context */ 844 if(!(ctx = EVP_CIPHER_CTX_new())) { 845 handleErrors(); 846 } 847 848 /* 849 * Initialise the decryption operation. IMPORTANT - ensure you use a key 850 * and IV size appropriate for your cipher 851 * In this example we are using 256 bit AES (i.e. a 256 bit key). The 852 * IV size for "most" modes is the same as the block size. For AES this 853 * is 128 bits 854 */ 855 if(1 != EVP_DecryptInit_ex(ctx, EVP_aes_256_ctr(), NULL, key, iv)) { 856 handleErrors(); 857 } 858 859 /* Provide the message to be decrypted, and obtain the plaintext output. 860 * EVP_DecryptUpdate can be called multiple times if necessary 861 */ 862 if(1 != EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext_len)) { 863 handleErrors(); 864 } 865 plaintext_len = len; 866 867 /* 868 * Finalise the decryption. Further plaintext bytes may be written at 869 * this stage. 870 */ 871 if(1 != EVP_DecryptFinal_ex(ctx, plaintext + len, &len)) { 872 handleErrors(); 873 } 874 plaintext_len += len; 875 876 /* Clean up */ 877 EVP_CIPHER_CTX_free(ctx); 878 879 return plaintext_len; 880 }</pre>
---	---

(a) Funzione per l'operazione di cifratura

(b) Funzione per l'operazione di decifratura

Figura 5.3: Implementazione del cifrario AES-256-CTR

5.3 Difficoltà riscontrate

Durante la realizzazione dell'applicazione sono state riscontrate diverse difficoltà, principalmente riguardo l'utilizzo delle librerie.

5.3.1 Introduzione di CAN-utils

La difficoltà più complicata da gestire è stata l'introduzione della libreria **CAN-utils**, in quanto questa è stata rilasciata come strumento *indipendente* e non è stata prevista la possibilità di utilizzarla come **libreria**. Per questo motivo è stato necessario adottare una strategia per utilizzare gli eseguibili di **CAN-utils** (ottenuti effettuando la compilazione dei

codici sorgenti) all'interno dell'applicativo realizzato. La strada seguita, quindi, è stata quella di richiamare gli eseguibili tramite linea di comando, utilizzando la funzione `system()` offerta dalla libreria `stdlib.h` inclusa in **C**, la quale non fa altro che creare un nuovo processo ed eseguire il comando `sh -c STRINGA_INPUT`. In questo modo, se si costruisce una **stringa** che corrisponde al comando completo da eseguire (compreso di argomenti e parametri) e si esegue la funzione con questa stringa, quello che si ottiene è che viene eseguita l'operazione, ovviamente a patto che la stringa non contenga errori. Un esempio, estratto dal codice, può essere visto nel Codice 5.3.

```
1 // Invio ID e numero di messaggi
2 char* idToSend = ".../can-utils-2023.03/cansend vcan0 00B#";
3 int sendLen = strlen(idToSend);
4 char* preCmd = malloc( (sendLen + 20) * sizeof(char));
5 strcpy(preCmd, idToSend);
6 strcat(preCmd, ID_ECU);
7 system(preCmd);
```

Codice 5.3: Utilizzo di CAN-utils tramite linea di comando

5.3.2 Utilizzo delle *pipe*

Purtroppo il semplice utilizzo della funzione `system()` non è bastata per utilizzare la libreria. Questo perchè la funzione si limita semplicemente all'esecuzione del codice che gli viene fornito, senza restituire un'eventuale **output** generato dall'esecuzione del comando. Ad esempio se si esegue `candump` per leggere i messaggi che viaggiano sulla rete con `system()`, quello che succede è che oltre a non mostrare nessun output, il codice si blocca in quanto `candump` è **bloccante** (poichè rimane in ascolto perpetuo di messaggi). Per risolvere questo problema, bisogna utilizzare un altro strumento: le **pipe**.

Una **pipe** è uno strumento di **comunicazione tra processi** che consiste nella creazione di un canale unidirezionale tra un processo e un altro, ovvero solo uno di due processi può **scrivere** sulla *pipe* e solo l'altro può **leggere** dalla stessa. Alcuni vantaggi dell'utilizzo di una *pipe* sono:

- Semplicità: sono un modo semplice e diretto per far comunicare due processi;
- Efficienza: permettono di trasferire dati velocemente e con il minimo *overhead*;
- Affidabilità: sono in grado di effettuare la rilevazione di errori durante la trasmissione e di assicurare la corretta consegna dei dati;

- Flessibilità: permettono di realizzare tantissimi tipi di protocollo di comunicazione, sia unidirezionali che bidirezionali (aprendo due pipe in versi opposti). [2]

Tuttavia, l'utilizzo di una *pipe* presenta anche alcuni svantaggi:

- Capacità limitata: non hanno una capienza adeguata per eventuali enormi quantità di dati trasferite in una sola volta, limitandosi solo con piccole quantità di dati;
- Unidirezionale: come già accennato, permettono una comunicazione solo in un verso e, se si desidera realizzare una comunicazione bidirezionale, bisogna necessariamente creare una seconda *pipe* nel verso opposto alla prima;
- Sincronizzazione: nel caso di una comunicazione bidirezionale, è necessario che entrambi i processi siano sincronizzati per assicurare che la trasmissione avvenga nell'ordine corretto;
- Scalabilità limitata: sono limitate ad una comunicazione tra pochi processi su uno stesso computer. [2]

Utilizzando una *pipe*, è quindi possibile ottenere l'output di un processo in esecuzione e, sfruttando questa caratteristica, è possibile intercettare i messaggi letti con `candump`. Come si può vedere nel Codice 5.4, utilizzando la funzione `popen()` si crea una *pipe* con un nuovo processo che eseguirà il comando fornito in input come stringa (stessa logica di `system()`) e si può specificare se il processo che ha lanciato `popen()` leggerà dalla *pipe* (con il parametro "r") o scriverà sulla *pipe* (con il parametro "w").

```
1 // Mi metto in ascolto sul bus
2 FILE* fp = popen("../can-utils-2023.03/candump vcan0", "r");
```

Codice 5.4: Creazione di una pipe

Per poter leggere dalla pipe si può utilizzare una qualsiasi funzione che legge da uno **stream**, nel caso del progetto è stata utilizzata la funzione `fgets` come mostrato nel Codice 5.5.

```
1 fgets(buffer, 50, fp);
2 j = 9;
3 k = 0;
4 // Ottengo l'ID che definisce la tipologia di messaggio
5 while (!isspace(buffer[j])) {
6     msgCode[k] = buffer[j];
7     k++;
}
```

```

8         j++;
9
10        }
11        j = 0;
12        // Estraggo l'ID del sender
13        for(int i = 0; i < DATA_LENGTH; i+=2) {
14            tempID[i] = buffer[20 + j];
15            tempID[i+1] = buffer[20 + j +1];
16            j+=3;
17        }
18        tempID[DATA_LENGTH] = '\0';

```

Codice 5.5: Lettura dalla pipe

Ovviamente, quello che si legge dalla pipe è una stringa, quindi è necessario effettuare un *post-processing* di questa per estrarre le informazioni contenute. Un esempio di *post-processing* si trova sempre nel Codice 5.5.

5.3.3 Chiusura della pipe

Al termine di ogni programma, è buona norma liberare tutto lo spazio allocato, chiudere tutti i file aperti, chiudere tutte le pipe create, ecc. al fine di evitare possibili problemi di **memory leak** o inconsistenze. Quindi, al fine di seguire queste *best-practices*, come ultime operazioni da eseguire prima del termine dell'applicativo si cerca di chiudere la *pipe* creata ma, tuttavia, quello che succede è che il programma si ferma e non prosegue (quindi non termina). Questo problema è dovuto al fatto che quando si chiude una pipe, la funzione `pclose()` (necessaria per chiudere una *pipe*) **attende che il processo collegato alla pipe termini** [7], evento che non accadrà mai poichè (come accennato in precedenza) `candump` si mette in ascolto **perpetuo**. Quindi, per chiudere la *pipe* è necessario **terminare forzatamente** il processo creato da `popen` ma, anche qui, sorge un ulteriore problema, ovvero non si conosce nessuna informazione riguardo il processo creato. Per poter chiudere un processo è necessario sapere il suo ID (chiamato **PID**) e inviare un segnale di interruzione, ma le uniche informazioni note (anche grazie alla documentazione di `popen`) sono che il processo è **figlio** di quello che chiama la funzione `popen` e che il nome del processo è esattamente il comando lanciato per eseguirlo, quindi è necessario adottare una strategia per ottenere il **PID** del processo figlio. Dopo un'attenta ricerca e vari tentativi, la strategia vincente è stata la seguente:

1. Ottenimento del **PID** del processo padre tramite la funzione `getpid()`;

2. Utilizzo del comando `ps -eaf` tramite la costruzione di una stringa assemblata con il **PID** del padre e il nome del processo;
3. Creazione di una *pipe* sul comando appena costruito in modo da ottenere l'output di quest'ultimo;
4. Lettura dalla *pipe* ed estrazione del **PID** del processo figlio;
5. Invio del segnale di terminazione al processo figlio;
6. Chiusura di entrambe le *pipe*.

Il codice necessario per implementare questa strategia si può trovare nel Codice 5.6, con l'aggiunta di ulteriori commenti per rendere più chiaro ogni punto.

```
1 // Termino il processo associato alla pipe
2 pid_t pid = getpid();
3 char num[8];
4 sprintf(num, "%d", pid);
5
6 // Comando da lanciare per ottenere il PID del processo figlio
7 char* cmd = "ps -eaf | grep \"/..can-utils-2023.03/candump vcan0\" | grep ";
8 int tempLen = strlen(cmd) + 8;
9 char* temp = malloc(tempLen * sizeof(char));
10 strcpy(temp, cmd);
11 // Aggiungo il PID del processo padre al termine della stringa
12 strcat(temp, num);
13
14 // Creo una nuova pipe sul comando per ottenere l'output di questo
15 FILE* targetProcess = popen(temp, "r");
16
17 // Leggo dalla nuova pipe
18 fgets(buffer, 50 ,targetProcess);
19
20 // Estraggo il PID dalla stringa ottenuta
21 int i = 0;
22 while(!isspace(buffer[11 + i])) {
23     num[i] = buffer[11 + i];
24     i++;
25 }
26
27 int targetPid;
28 sscanf(num, "%d", &targetPid);
```

```
29  
30 // Termino il processo figlio  
31 kill(targetPid, SIGKILL);  
32  
33 free(temp);  
34  
35 // Chiudo sia la pipe di supporto che la pipe principale  
36 pclose(targetProcess);  
37 pclose(fp);  
38 printf("CHIUSA LA PIPE\n");
```

Codice 5.6: Operazioni necessarie per chiudere la *pipe*

5.4 Risultati ottenuti

Una volta completato l'applicativo e risolti tutti i problemi riscontrati, il passo successivo è stato quello di misurare l'**impatto prestazionale** apportato dall'aggiunta della cifratura. Sono stati raccolti due tempi, il tempo necessario per generare le chiavi di cifratura e il tempo necessario per lo scambio dei messaggi, entrambi in **microsecondi**.

5.4.1 Tempi per la generazione delle chiavi

Il tempo medio necessario per la generazione delle chiavi e per lo scambio di una chiave di sessione tra due nodi è di circa 50829 µs. A livello prestazionale, assumento che avvenga all'avvio della rete, l'impatto prestazionale non è elevato ma è comunque **molto alto**, in quanto è necessario scambiare una grande quantità di messaggi per inviare capsula e chiave pubblica (quasi 200 per ogni nodo). Per cui potrebbero essere introdotti dei ritardi che, su reti contenenti molte *ECU*, potrebbero essere percettibili dall'uomo e allungare i tempi di accensione di tutti i sistemi.

5.4.2 Tempi per lo scambio di messaggi

Per quanto riguarda il tempo necessario per lo scambio di messaggi, un confronto può essere visto in Tabella 5.1. I tempi sono stati raccolti a partire dall'inizio della cifratura e fino alla decifratura dell'ultimo byte (in pratica quando il contenuto del messaggio è disponibile all'applicativo), quindi considerando anche eventuali ritardi in trasmissione. A differenza del precedente, questo impatto è **molto elevato**, in quanto si introduce un ritardo per trasmissione

di quasi 100 ms , inaccettabile per sistemi *safety-critical* e per la gestione di un motore di un'auto in corsa.

Tempo medio senza cifratura	Tempo medio con cifratura
93.12 μs	96433.16 μs

Tabella 5.1: Tempi medi rilevati

Se si considera un'auto in corsa ad una velocità di 180 km/h , un ritardo di trasmissione di circa 100 ms corrispondono a circa 5 m percorsi senza nessuna risposta dai comandi e senza considerare un eventuale ritardo aggiuntivo necessario per processare il messaggio. Prendendo per esempio un sistema **brake-by-wire**, 5 m di frenata in più possono fare una grande differenza ed evitare incidenti (anche mortali).

CAPITOLO 6

Conclusioni

In questo capitolo verranno aggiunti ulteriori dettagli sui risultati ottenuti e verranno introdotte eventuali problematiche riscontrate. Infine verranno illustrati alcuni degli sviluppi futuri che è possibile realizzare partendo dal lavoro svolto.

6.1 Considerazioni finali

Prendendo in considerazione il confronto effettuato sui tempi raccolti, si possono discutere con molta più chiarezza i lati positivi e negativi di questa soluzione, approfondendo anche eventuali modifiche che potrebbero migliorare i risultati ottenuti.

6.1.1 Sicurezza garantita

A livello di guadagno in termini di sicurezza, sicuramente la soluzione è sicura contro:

- **Eavesdropping**, ovvero l'ascolto passivo della rete. Infatti, grazie alla cifratura simmetrica e all'aggiornamento della chiave di sessione (cambiando anche chiavi di cifratura asimmetrica) non si è più in grado di distinguere i vari messaggi in transito sulla rete;
- **Data manipulation**, poiché alterando anche solo un bit di un messaggio cifrato, il messaggio cambia completamente, inoltre, la probabilità che la modifica porti ad un messaggio con un **payload** e un **CRC** corretto (quindi che ne conferma l'integrità) è talmente bassa da rendere la fattibilità di questo tipo di attacco quasi impossibile. Inoltre, si può migliorare ulteriormente la resistenza contro questo attacco utilizzando

la modalità **AES-GCM**, che è una modifica della modalità *CTR* che la rende **autenticata** e quindi qualunque modifica al messaggio fa produrre un errore in decifratura. [17]

Tuttavia, anche con questa soluzione il protocollo rimane vulnerabile ad alcuni attacchi:

- **Data injection:** un malintenzionato che si collega alla rete, può generarsi delle chiavi asimmetriche ed effettuare lo scambio di una sessione, con la conseguenza che è in grado di inviare messaggi arbitrari **validi**, arrecando disturbi e mostrando informazioni false. Purtroppo, per risolvere questo problema o si può utilizzare un'**autorità di certificazione** che garantisca la validità delle chiavi, aggiungendo ulteriore **overhead** a delle prestazioni già pessime;
- **Denial of Service:** un malintenzionato è ancora in grado di collegarsi alla rete e generare messaggi arbitrari arrecando disturbo alla rete e saturandola con grandi quantità di dati *spazzatura*.

6.1.2 Utilizzo di versioni ottimizzate dei cifrari

Il problema principale della soluzione proposta riguarda i tempi necessari per effettuare le varie operazioni crittografiche. Questo è dovuto principalmente al fatto che sono state utilizzate versioni **generiche** degli algoritmi e quindi non ottimizzate allo scopo, per cui utilizzando versioni **ottimizzate** di questi, è possibile ridurre drasticamente i tempi. Infatti, è possibile realizzare una versione di **AES-128** che aggiunge un overhead nell'ordine dei **nanosecondi** [10] ed esiste una versione di **kyber** ottimizzata per determinate architetture di processori. Per cui, utilizzando queste piuttosto che le generiche, sicuramente si possono ottenere risultati **molto più soddisfacenti**.

6.1.3 Ottimizzazione del protocollo

Un altro problema è dovuto al fatto che il protocollo realizzato genera l'invio di molti messaggi, appesantendo molto la fase di scambio delle chiavi di sessione. A questo punto, una modifica che potrebbe essere fatta è quella di utilizzare *CAN FD*, il quale permette di inviare fino a 64 Byte alla volta e riduce drasticamente il numero di messaggi scambiati ottenendo un guadagno prestazionale non indifferente. Un'altra possibile strada da seguire, per continuare ad utilizzare la versione base di *CAN*, consiste nel modificare il protocollo in modo da tentare di *ottimizzare* il numero di messaggi da scambiare e velocizzare questa fase (provando anche ad effettuare una **compressione** delle chiavi pubbliche per ridurne la grandezza).

6.2 Sviluppi Futuri

Oltre alle varie considerazioni che è possibile fare in seguito al confronto, un ulteriore passo che si può fare è quello di pensare a come espandere e migliorare la soluzione proposta. In questo modo non si limita la soluzione ad un solo contesto, trovando persino eventuali soluzioni ad ulteriori problemi.

6.2.1 Utilizzo di tecniche di cifratura *Lightweight*

Dal momento che le *ECU* utilizzate nelle automobili solitamente non hanno una grande potenza computazionale, l'utilizzo di un cifrario come **AES** potrebbe non essere appropriato in quanto richiede delle prestazioni adeguate per non pesare troppo sul processore. Per questa ragione, una possibile modifica che si può realizzare è la sostituzione di **AES** con un cifrario *lightweight*. Questa categoria è composta da cifrari realizzati appositamente per dispositivi con capacità computazionali molto ristrette [33] (come ad esempio i dispositivi *IoT*) che fossero in grado di garantire **cifratura autenticata** e, eventualmente, anche funzionalità di **hashing**. Ovviamente, come per gli altri standard, il **NIST** si è interessato di proporre il problema e scegliere una serie di algoritmi che rispettassero i vincoli descritti dal problema. L'utilizzo di questi cifrari potrebbe migliorare le prestazioni della soluzione garantendo lo stesso un adeguato livello di sicurezza, basti pensare che in letteratura già esistono dei cifrari *ibridi Post-Quantum* che utilizzano cifrari *lightweight*.

6.2.2 Utilizzo in un contesto V2X

Il protocollo realizzato, per come è stata realizzato, si presta molto bene ad un adattamento in un altro contesto: **Vehicle-to-Everything** (o *V2X*).

Questa non è altro che una tipologia di comunicazione che coinvolge un'auto e qualunque altra entità "intelligente", che può essere uno smartphone, una casa, un cloud (ad esempio per ricevere aggiornamenti di sistema), altri veicoli o persino biciclette e sedie a rotelle. Le motivazioni dietro questa tipologia di comunicazione includono:

- Aumento della sicurezza stradale;
- Migliore gestione dei flussi di traffico;
- Ottimizzazione dei consumi energetici;
- Sorveglianza di massa. [24]

Per cui, in questo caso, quello che si può fare è prevedere un cifrario asimmetrico **comune** (anche lo stesso **kyber**) e, nel momento in cui si vuole comunicare con un'altra entità, cominciare con lo scambio di chiave pubblica e di sessione, come riassunto nello schema in Figura 6.1.

Vehicle to Everything

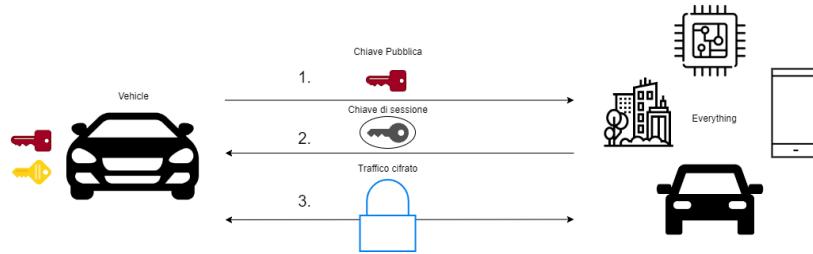


Figura 6.1: Schema del protocollo in un contesto V2X

Tuttavia, in un contesto del genere, risulterebbe molto semplice per un attaccante ottenere informazioni sensibili o inviare informazioni false e errate semplicemente effettuando lo scambio e fingendosi qualcun'altro. Per tentare di ovviare a questo problema, si potrebbe prevedere un'autorità di certificazione che autentichi e certifichi ogni chiave pubblica che viene scambiata, come mostrato in Figura 6.2.

Vehicle to Everything
(con CA)

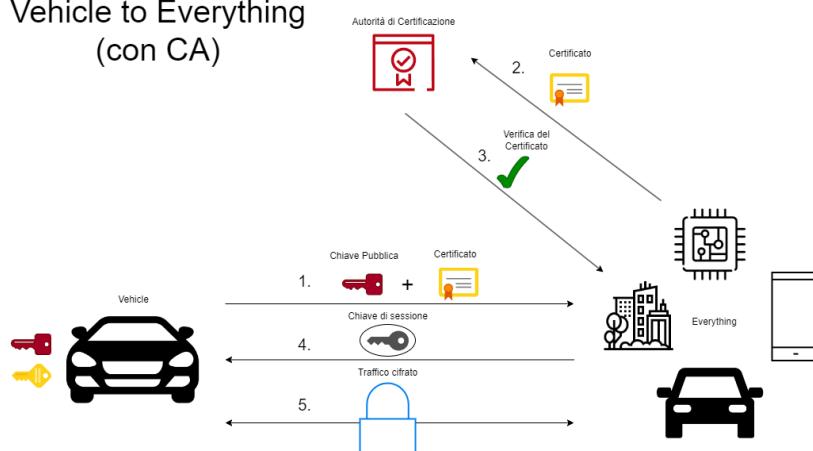


Figura 6.2: Schema del protocollo in un contesto V2X con CA

In questo modo, rilasciando dei certificati per ogni dispositivo nessuno può impersonare qualcun'altro e un attacco da parte di un malintenzionato diventa più difficile. Ma per effettuare tutte queste operazioni, purtroppo, è necessaria molta potenza di calcolo e molto

tempo, quindi bisogna ottimizzare il protocollo per stringere i tempi e utilizzare dei processori che permettano una verifica dei protocollli più veloce.

6.2.3 Aggiunta di un'autorità di certificazione

La soluzione si è focalizzata principalmente sul problema della protezione del payload, tuttavia, questa può essere adattata anche al problema dell'**autenticazione** facendo alcuni accorgimenti.

Il problema principale di CAN è che chiunque si collega alla rete è in grado di ricevere e trasmettere come il resto dei nodi, per cui è necessario un *filtro* con cui stabilire chi può essere in grado di comunicare sulla rete. Riguardo la ricezione, per come è stato realizzato CAN, non si può evitare ad un malintenzionato di mettersi in ascolto ma si possono proteggere i messaggi con la cifratura in modo tale da non esporre informazioni sensibili, mentre riguardo l'invio si può fare in modo che tutti i nodi **legittimi** ignorino ogni messaggio proveniente da un nodo non autorizzato. Per poter realizzare un sistema del genere bisognerebbe fare due modifiche:

- Evitare di generare una coppia di chiavi ad ogni avvio, ma generarla solo la prima volta ed utilizzare quella. Questo non riduce la sicurezza del sistema realizzato ma bisogna fare attenzione alla segretezza delle **chiavi private**, poichè in caso di compromissione bisogna rigenerarle **immediatamente**.
- Prevedere la presenza di un nodo CA che aiuti la rete a distinguere le chiavi pubbliche **autorizzate e non**, come mostrato in Figura 6.3

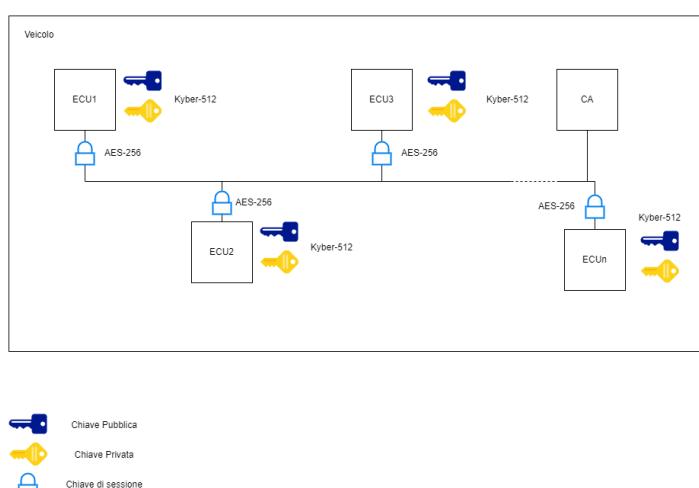


Figura 6.3: Rete CAN con protezione del payload e nodo CA

In questo modo, si potrebbe modificare il protocollo in modo tale che, alla ricezione di una chiave pubblica, si attenda un riscontro positivo o negativo da parte del nodo CA e, in base a questo, salvare o scartare la chiave ricevuta (oltre al traffico che proviene da quel nodo).

Ovviamente, oltre alla modifica al protocollo, bisognerebbe anche trovare un modo per garantire la provenienza dei riscontri da parte della CA, poiché un nodo malizioso potrebbe generarsi da solo un riscontro positivo e inviarlo spacciandosi per il nodo CA. In questo caso, un modo per garantire la provenienza del riscontro può essere quello di prevedere la presenza di una **firma**, ottenuta utilizzando uno schema di **firma digitale Post-Quantum** come **Dilithium**, anch'esso costruito su un problema basato sul **problema dei reticolari**. Tuttavia, oltre a dover necessariamente generare più messaggi per inviare sia riscontro che firma, si rischia di introdurre un onere computazionale troppo grande per l'hardware limitato di una ECU e anche un ulteriore ritardo che rischia di compromettere la funzionalità di CAN.

Ringraziamenti

Tutto il lavoro realizzato per la scrittura di questa trattazione è stato frutto della continua-zione di uno dei percorsi più complessi della mia vita ma che probabilmente non dimenticherò mai, in quanto mi ha dato la possibilità di sfidare costantemente i miei limiti e mi ha dato l'opportunità di incontrare delle persone meravigliose che mi hanno accompagnato durante tutto il mio percorso e che desidero ringraziare.

Innanzitutto ci tengo a ringraziare il mio relatore Arcangelo Castiglione per avermi dato l'opportunità di lavorare ad un progetto molto stimolante e per avermi seguito attivamente durante tutto il lavoro svolto, consigliandomi sempre la via più corretta di procedere.

Ringrazio Vincenzo, per avermi accompagnato durante tutto il mio percorso, sia nei momenti più belli che nei momenti più difficili, per avermi sempre sostenuto e incoraggiato quando serviva e per essere stato un compagno di studio e lavoro insostituibile. Senza i giri intorno al tavolo della tua camera, molto probabilmente non avrei concluso la mia carriera accademica in maniera così brillante e senza le "serate confessionale" probabilmente non avrei mai condiviso aspetti della mia vita molto problematici e non avrei mai trovato una soluzione ad essi.

Ringrazio Orazio, per avermi accompagnato durante tutto il mio percorso, per essere stato un compagno di studio e di lavoro impeccabile e per essere stato una guida e un fratello maggiore, che cerca di consigliare sempre il modo migliore di fare tutto e che è sempre a disposizione quando c'è bisogno, sia per un aiuto che per uno sfogo. Senza di te non sarei mai diventato quello che sono oggi, non mi sarei mai appassionato così al mondo dell'elettronica e senza di te non avrei mai preso il mio Acer Nitro 5. Grazie per essere stato sempre al mio fianco, per avermi insegnato a ragionare in maniera molto più logica e per avermi insegnato le basi della contrattazione.

Ringrazio Hermann, per avermi accompagnato durante tutto il mio percorso, per essere

stato un compagno essenziale di studio e di banco e per essere stato un "compagno di mille avventure". Ancora ricordo di quella volta che compilammo *Gentoo* a casa mia o di quella volta in cui "distrussi" *Windows* e non potevo creare la pennetta USB per reinstallarlo. Mi hai sempre fatto compagnia quando ne avevo bisogno e ti sei messo sempre a disposizione per aiutarmi e senza di te, molto probabilmente, ancora non saprei utilizzare i sistemi basati su Linux. Grazie per esserci sempre stato per me e per esserti sempre confrontato con me sui progetti a cui abbiamo lavorato insieme (soprattutto su quest'ultimo).

Ringrazio Felice, Mattia M., Mattia C., Emanuele, Luigi, Francesco, Simone, Vincenzopio e tutti i miei amici (anche quelli già citati) per avermi accompagnato in questo lungo percorso, per avermi sostenuto e per avermi voluto bene come un se fossi un vostro fratello. Con voi ho passato i momenti più belli e indimenticabili di tutta la mia vita e porterò sempre una parte di voi nel mio bagaglio di vita.

Ringrazio mia sorella Teresa e Stefano, per essermi stati vicini dal primo giorno di università, per avermi compreso quando ne avevo bisogno, per avermi sempre sostenuto e aiutato a concludere il mio percorso nella maniera migliore possibile e per avermi sempre fornito un "rifugio sicuro" dove rintanarmi per qualche giorno quando avevo bisogno di "staccare un po' la spina". Grazie per avermi sempre guidato e per avermi sempre ascoltato.

Ringrazio Raffaele per essere stato la mia bussola morale, per avermi consigliato sempre il miglior modo di procedere e per avermi sempre ascoltato quando ne avevo bisogno. Le chiacchierate fatte durante le passeggiate con Koba sono sempre state le migliori e le più stimolanti, oltre ad essere le migliori valvole di sfogo contro tutto lo stress accumulato all'università o durante la vita di tutti i giorni.

Ringrazio mia Zia Giusy per aver sempre creduto in me, per avermi sostenuto in ogni modo in cui hai potuto e per avermi voluto bene come un figlio. Senza di te non avrei scelto questo percorso e non avrei ottenuto questi risultati. Grazie per esserci sempre stata.

Ringrazio tutta la mia famiglia, per tutto il sostegno che mi avete dato e per avermi dato l'opportunità di intraprendere questo percorso che mi ha aiutato ad oltrepassare i miei limiti e a dimostrare quanto valgo.

Infine, ci tengo a ringraziare i miei nonni Teresa e Francesco che, anche se non sono più qui da molto tempo, sono sempre stati al mio fianco per permettermi di superare ogni ostacolo. Sono sicuro che sareste fieri del percorso che ho intrapreso e questo mi ha dato la forza di continuare.

Grazie a tutti voi, il vostro contributo è stato unico e insostituibile.

Bibliografia

- [1] (2019), «Kyber», <https://pq-crystals.org/kyber/>. (Citato alle pagine 36 e 37)
- [2] (2020), «IPC technique PIPES», <https://www.geeksforgeeks.org/ipc-technique-pipes/>. (Citato a pagina 62)
- [3] (2023), «Automotive electronics», https://en.wikipedia.org/wiki/Automotive_electronics. (Citato a pagina 1)
- [4] (2023), «Carrier-sense multiple access with collision detection», https://en.wikipedia.org/wiki/Carrier-sense_multiple_access_with_collision_detection. (Citato a pagina 7)
- [5] (2023), «FlexRay Automotive Communication Bus Overview», <https://www.ni.com/en/shop/seamlessly-connect-to-third-party-devices-and-supervisory-systems/flexray-automotive-communication-bus-overview.html>. (Citato alle pagine 17, 20 e 21)
- [6] (2023), «Non-return-to-zero», <https://en.wikipedia.org/wiki/Non-return-to-zero>. (Citato a pagina 11)
- [7] (2023), «popen, pclose - pipe stream to or from a process», <https://man7.org/linux/man-pages/man3/popen.3.html>. (Citato a pagina 63)
- [8] (2023), «Post-quantum cryptography», https://en.wikipedia.org/wiki/Post-quantum_cryptography. (Citato a pagina 5)

- [9] (2023), «What is the CAN Bus», <https://dewesoft.com/blog/what-is-can-bus>. (Citato alle pagine 8, 10, 14 e 15)
- [10] BOZDAL, M., SAMIE, M. e JENNIONS, I. (2018), «A Survey on CAN Bus Protocol: Attacks, Challenges, and Potential Solutions», *2018 International Conference on Computing, Electronics and Communications Engineering (iCCECE)*, URL https://dspace.lib.cranfield.ac.uk/bitstream/handle/1826/15153/Survey-CAN_bus_protocol-2019.pdf. (Citato alle pagine 3, 4, 33 e 68)
- [11] CONSORTIUM, F. (2005), *FlexRay Communications System Protocol Specification Version 2.1*, FlexRay Consortium, URL https://www.softwareresearch.net/fileadmin/src/docs/teaching/SS08/PS_VS/FlexRayCommunicationSystem.pdf. (Citato alle pagine 20, 21, 22 e 27)
- [12] CONTRIBUTORS, W. (2022), «Ciphertext indistinguishability», URL https://en.wikipedia.org/wiki/Ciphertext_indistinguishability. (Citato a pagina 37)
- [13] CONTRIBUTORS, W. (2023), «Advanced Encryption Standard», https://en.wikipedia.org/wiki/Advanced_Encryption_Standard. (Citato alle pagine 40, 41, 42, 43 e 47)
- [14] CONTRIBUTORS, W. (2023), «Block cipher mode of operation», https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation. (Citato a pagina 46)
- [15] CONTRIBUTORS, W. (2023), «CAN bus», https://en.wikipedia.org/wiki/CAN_bus. (Citato alle pagine 10, 12, 13 e 14)
- [16] CONTRIBUTORS, W. (2023), «Confusion and diffusion», https://en.wikipedia.org/wiki/Confusion_and_diffusion. (Citato a pagina 40)
- [17] CONTRIBUTORS, W. (2023), «Galois/Counter Mode», https://en.wikipedia.org/wiki/Galois/Counter_Mode. (Citato a pagina 68)
- [18] CONTRIBUTORS, W. (2023), «Key encapsulation mechanism», https://en.wikipedia.org/wiki/Key_encapsulation_mechanism. (Citato a pagina 36)
- [19] CONTRIBUTORS, W. (2023), «Kyber», Url=<https://en.wikipedia.org/wiki/Kyber>. (Citato a pagina 36)

- [20] CONTRIBUTORS, W. (2023), «Local Interconnect Network», https://en.wikipedia.org/wiki/Local_Interconnect_Network. (Citato alle pagine 28 e 31)
- [21] CONTRIBUTORS, W. (2023), «Permutation box», https://en.wikipedia.org/wiki/Permutation_box. (Citato a pagina 40)
- [22] CONTRIBUTORS, W. (2023), «Quantum computing», https://en.wikipedia.org/wiki/Quantum_computing. (Citato alle pagine 34 e 35)
- [23] CONTRIBUTORS, W. (2023), «S-box», <https://en.wikipedia.org/wiki/S-box>. (Citato a pagina 40)
- [24] CONTRIBUTORS, W. (2023), «Vehicle-to-everything», <https://en.wikipedia.org/wiki/Vehicle-to-everything>. (Citato a pagina 69)
- [25] COOK, J. e FREUDENBERG, J. (2008), *Controller Area Network (CAN) EECS 461*, University of Michigan, URL https://www.eecs.umich.edu/courses/eecs461/doc/CAN_notes.pdf. (Citato a pagina 6)
- [26] ELECTRONICS, C. (2021), «CAN FD Explained - A Simple Intro», <https://www.csselectronics.com/pages/can-fd-flexible-data-rate-intro>. (Citato a pagina 16)
- [27] ELECTRONICS, C. (2021), «LIN Bus Explained - A Simple Intro», <https://www.csselectronics.com/pages/lin-bus-protocol-intro-basics>. (Citato alle pagine 28, 29, 30 e 31)
- [28] ELECTRONICS, C. (2022), «CAN Bus Errors Explained - a Simple Intro», <https://www.csselectronics.com/pages/can-bus-errors-intro-tutorial>. (Citato a pagina 13)
- [29] GIUSEPPE, C. D. (2023), «La matematica dietro la PQC: i reticoli», URL <https://www.telsy.com/it/la-matematica-dietro-la-pqc-i-reticoli/>. (Citato a pagina 38)
- [30] GIUSEPPE, C. D. (2023), «La matematica dietro la PQC: Learning With Errors», <https://www.telsy.com/it/la-matematica-dietro-la-pqc-learning-with-errors-lwe/>. (Citato a pagina 38)

- [31] HUANG, J., ZHAO, M., ZHOU, Y. e XING, C.-C. (2019), «In-Vehicle Networking: Protocols, Challenges, and Solutions», *IEEE Network*, vol. 33, p. 92–98. (Citato alle pagine 2, 3, 6, 32 e 33)
- [32] KUMARI, S., SINGH, M., SINGH, R. e TEWARI, H. (2022), «A post-quantum lattice based lightweight authentication and code-based hybrid encryption scheme for IoT devices», *Computer Networks*, vol. 217, p. 109327, URL <https://www.sciencedirect.com/science/article/pii/S138912862200367X>.
- [33] LAWRENCE BASSHAM, J. K., DONGHOON CHANG (2013), «Lightweight Cryptography», <https://csrc.nist.gov/projects/lightweight-cryptography>. (Citato a pagina 69)
- [34] RAO, S., MAHTO, D., YADAV, D. e KHAN, D. (2017), «The AES-256 Cryptosystem Resists Quantum Attacks», *International Journal of Advanced Research in Computer Science*, vol. 8, p. 404–408, URL https://www.researchgate.net/publication/316284124_The_AES-256_Cryptosystem_Resists_Quantum_Attacks. (Citato a pagina 47)
- [35] REDAZIONE (2019), «Il protocollo FlexRay», <https://it.emcelettronica.com/il-protocollo-flexray>. (Citato alle pagine 17, 18, 19, 20, 21, 22, 23 e 27)
- [36] ROBECCHI, R. (2022), «La crittografia post-quantistica è arrivata: capiamo come funziona», https://edge9.hwupgrade.it/articoli/innovazione/6330/la-crittografia-post-quantistica-e-arrivata-capiamo-come-funziona_index.html. (Citato a pagina 38)
- [37] SEMICONDUCTORS, N. (2021), «MPC5744P FlexRay Interface in Pictures», <https://www.nxp.com/docs/en/application-note/AN12233.pdf>. (Citato a pagina 22)
- [38] VINODH KUMAR, B. e RAMESH, J. (2014), «Automotive in vehicle network protocols», URL <https://ieeexplore.ieee.org/document/6921836>. (Citato a pagina 3)

Siti Web consultati

- Wikipedia – www.wikipedia.org

- Cranfield University – <https://dspace.lib.cranfield.ac.uk/>
- IEEE Xplore – <https://ieeexplore.ieee.org/>
- Dewesoft – <https://dewesoft.com/>
- University of Michigan – <https://eecs.engin.umich.edu/>
- CSS Electronics – <https://www.csselectronics.com/>
- Elettronica Open Source – <https://it.emcelettronica.com/>
- NXP Semiconductors – <https://www.nxp.com/>
- National Instruments – <https://www.ni.com>
- University of Salzburg – <https://www.softwareresearch.net/>
- Telsy – <https://www.telsy.com/>
- PQ-Crystals – <https://pq-crystals.org/>
- Hardware Upgrade – <https://edge9.hwupgrade.it/>
- GeeksforGeeks – <https://www.geeksforgeeks.org>
- Linux Man pages online – <https://man7.org/linux/man-pages/index.html>
- NIST - Computer Security Resource Center – <https://csrc.nist.gov/>