Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica
Anno Accademico. 2021/2022
Sessione di Laurea Aprile 2022

Monitoraggio, validazione, testing e rilevazione di vulnerabilità informatiche sulle reti CAN

Relatore:
Prof. Guido Albertengo

Supervisore aziendale:
Ing. Davide Mazzucchi

Candidato:
Luca Fumarola

In collaborazione con:

ZIRAK
INFORMATION TECHNOLOGY

Ringraziamenti

Mi sento in dovere di dedicare questa pagina a tutti coloro che mi hanno supportato e sostenuto durante questo percorso formativo. Il raggiungimento di traguardi importanti nella vita è un fatto strettamente individuale, tuttavia, vorrei ringraziare la mia famiglia per avermi sempre sostenuto; alla base di ogni successo è necessario avere la piena consapevolezza delle proprie capacità.
Vorrei ringraziare chi ha creduto in me, chi mi ha incoraggiato ad intraprendere un percorso di cui non mi sentivo all'altezza, chi ha avuto fiducia in me prima di me stesso.
Ringrazio il mio relatore, il professore Guido Albertengo, per la sua infinita disponibilità, sempre pronto ad aiutarmi in ogni fase dello sviluppo dell'elaborato.
Ringrazio tutti i membri dell'azienda in cui ho svolto la tesi, per avermi accolto nel mondo del lavoro e per le competenze pratiche che mi hanno fatto sviluppare.

"Qualunque cosa la mente possa concepire e credere, la può raggiungere."

NAPOLEON HILL

Abstract

Lo sviluppo dell'industria automobilistica degli ultimi decenni ha portato a dei radicali cambiamenti ed innovazioni che hanno rivoluzionato il mondo automotive rispetto a come era stato concepito inizialmente. L' espansione del settore meccanico ed energetico avvenuta nei primi anni ottanta è stata soppiantata dall'espansione del settore elettronico, ancora oggi in forte crescita.
La diffusione dei componenti elettronici in misura sempre maggiore ha comportato la necessità di permettere la loro reciproca interazione. Gran parte delle funzionalità introdotte, richiedevano lo scambio di informazioni tra questi componenti, che non potevano operare in maniera completamente autonoma, isolati tra di loro.
Il protocollo CAN è un sistema di comunicazione, che permette l'interconnessione di più dispositivi elettronici, consentendo lo scambio di dati. Alla base di ogni servizio vi è una condivisione di informazioni tra stazioni diverse, ciò porta ad una vulnerabilità intrinseca del sistema. Il malfunzionamento di una di queste unità porterebbe ad un disservizio complessivo, che nel contesto automobilistico potrebbe comportare gravi conseguenze. Più problematico è il caso in cui il malfunzionamento non sia casuale, ma che si sia indotto un comportamento volutamente errato, con la finalità di danneggiare il sistema.
Prima di essere installati a bordo delle autovetture, ogni componente necessita di essere validato per non incorrere in problematiche future; è qui che nasce lo scopo della tesi: creare un software in grado di interagire attraverso una rete CAN con le unità di controllo elettroniche, in modo tale da verificarne il corretto funzionamento.
In secondo luogo, ma non meno importante, simulare un Host malevolo, il quale effettuando degli attacchi informatici permette di valutare il funzionamento della rete in condizioni critiche, identificando eventuali vulnerabilità.

The development of automotive industry in the last decades, has caused radical changes and innovations that have revolutionized automotive world, as it was originally conceived.
The expansion of mechanical and energy sectors in the early '80s has been replaced by electronic industry expansion, even now fast growing.
The increasingly diffusion of electronic components has led the necessity of allowing their communication inside the car. Most of the introduced features, require the exchange of information between these components, which can't work completely autonomously, isolated from each other.
CAN protocol is a communication system, that allow the interconnection of more electronic devices, allowing data exchange. At the base of every service there is an information sharing between different stations, this led to an inherent vulnerability of the system. The malfunction of one the units may cause an overall disservice, which may provoke serious consequences in the automotive context. More problematic is the case in which the malfunction is not random, but it is deliberately done to damage the system.
Previous than being installed on board of the cars, every component requires to be validated to avoid future problems; it is here that thesis purpose is born, creation of a software able to interact through CAN network with the electronic control units, to verify their functioning.
Secondly, but not least, capacity of malicious host simulation, which allow to evaluate network operation under cyber-attacks, showing eventual vulnerabilities.

Indice

1 Introduzione

1.1 Cenni Storici……....1
1.2 Obiettivi……...2
1.3 Zirak S.r.l…….....3
1.4 Organizzazione della tesi……..3

2 CAN Bus

2.1 Evoluzione della rete……...……..5
2.2 Livello fisico……...……..6
2.3 Livello di collegamento dati……..8
2.4 Cyclic Redundance…….....10
2.5 Fase d'arbitraggio…….....11
2.6 Error Detection…….....13
2.7 CAN FD…….....15
2.9 J1939…….....16

3 Hardware

3.1 Raspberry Pi…….....19
3.2 Micro-SD……..... 20
3.3 Raspberry Pi LCD 7'' Touch Screen……21
3.4 PiCAN FD Board…….....22

4 Attacchi informatici sulle reti CAN

4.1 Introduzione…….....25
4.2 Bus Flood Attack…….....27
4.3 Adaptive Spoofing…….....27
4.4 Bus Off Attack…….....28
4.5 Simple Frame Spoofing …….…….29
4.6 Double Receive Attack……....30

5 Toll, Packages e Librerie

5.1 Introduzione……...……..33
5.2 socketCAN…….....33
5.3 cansend…….....34
5.4 cangen…….....35
5.5 candump…….....36
5.6 cangw…….....38
5.7 canplayer…….....39
5.8 Linguaggio di programmazione…….39
5.9 Fase di Compilazione…….....41
5.10 GTK…….....43
5.11 LibNotify…….....44

6 Documentazione Software

6.1 Introduzione…….....46
6.2 WhiteBox…….....46
6.3 Ini File…….....47
6.3.1 Detect and act…….....48
6.3.2 Act detect act…….....51
6.3.3 Detect group…….....51
6.3.4 Act detect group…….....56
6.3.5 Traffic……..... 56
6.3.6 Command …….....57
6.4 BlackBox……..... 57
6.5 Fase di rilevazione…….....58
6.6 Rilevazione del protocollo……..58
6.7 Attacchi informatici……..... 61

7 Code Development

7.1 Introduzione…….....64
7.2 CAN Utils……..... 65
7.3 CAN Utilities……..... 66
7.4 Utilites…….....69
7.5 Parsing……..... 69
7.6 Test……..... 71
7.7 Suite……..... 75
7.8 GUI……..... 78
7.9 Malicious_host…….....82
7.10 Meccanismi di comunicazione……..83

8 Conclusioni e risultati ottenuti

8.1 WhiteBox…….....84
8.2 Conclusioni WhiteBox……....87
8.3 BlackBox……..... 88
8.4 Conclusioni BlackBox……..... 90

INTRODUZIONE

1.1 Cenni Storici

Nei primi anni 80, la diffusione dei dispositivi elettronici aveva portato ad una loro installazione in misura sempre maggiore nelle autovetture; basti pensare al sistema antibloccaggio (ABS), il controllo della trazione (TCS) e l'aria condizionata (ACS).
I sistemi informatici di allora erano caratterizzati da un approccio centralizzato, il quale prevedeva una sola unità centrale, detta *mainframe*, avente elevata potenza di calcolo e diverse stazioni periferiche interconnesse all'unità centrale tramite una connessione punto-punto.

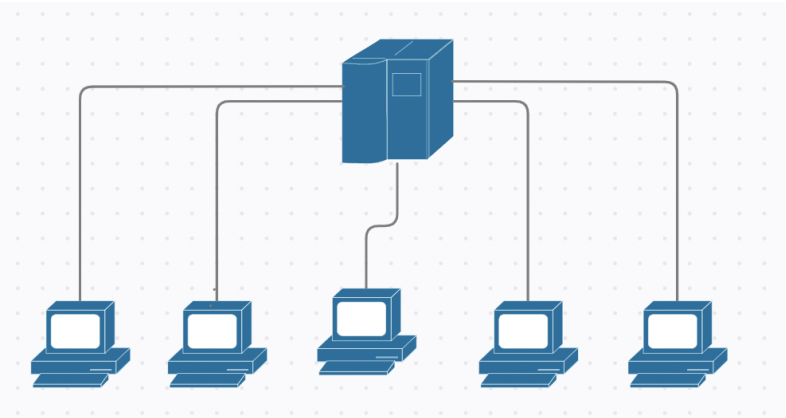

Figura 1.1: Architettura di una rete centralizzata

L'elaborazione dei dati avveniva principalmente nell'unità centrale, per cui le stazioni periferiche erano sprovviste di un'unità elaborativa.
La crescente necessità di aggiungere dispositivi elettronici all'interno delle autovetture causò, oltre ad un intuibile aumento dei costi di produzione, anche un aumento sostanziale della complessità di implementazione e ciò fu dovuto alla bassa scalabilità e flessibilità dei sistemi centralizzati.
Per ogni dispositivo connesso alla rete era necessario realizzare una connessione con l'unità centrale: all'aumentare del numero di stazioni connesse aumentavano il numero di collegamenti. Inoltre, l'enorme quantitativo dei cablaggi comportava un notevole ingombro fisico, cosa che nel contesto automobilistico non era accettabile.
Lo sviluppo dei microprocessori incentivò l'utilizzo di un approccio distribuito, in cui tutte le stazioni dispongono di un'unità elaborativa.
Nel 1986 lo sviluppo di CAN (*Controller Area Network*) da parte di Robert Bosch (su committenza della Mercedes) portò ad una definitiva decentralizzazione nelle reti automobilistiche.
CAN è un bus seriale, ovvero un canale di comunicazione che permette l'interconnessione di più dispositivi, chiamati ECU (*Electronic Control Unit*).

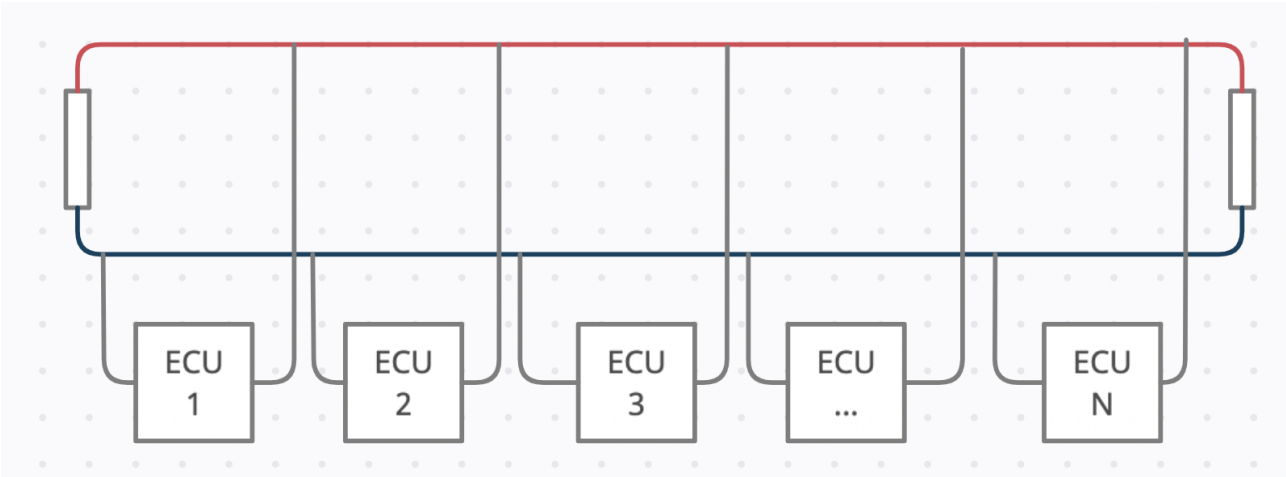

Figura 1.2: Architettura di una rete automobilistica basata su CAN Bus

Ogni stazione è connessa direttamente al bus composto da due fili, per cui il canale di comunicazione viene condiviso da tutte le periferiche, evitando quindi la necessità di creare dei collegamenti dedicati. Bosch era riuscito nella creazione di un sistema di comunicazione in grado di ridurre l'ingombro fisico, i costi, la complessità di implementazione garantendo una maggiore flessibilità, facilità di configurazione e resistenza ai disturbi elettromagnetici.
Il CAN bus divenne con il tempo il bus più utilizzato venendo adottato non solo nelle automobili, ma anche nelle macchine agricole, ascensori, aerei ed altri dispositivi.

1.2 Obiettivi

L'obiettivo della tesi è quello di sviluppare il software di un dispositivo in grado di interfacciarsi su una rete CAN, riuscendo ad interagire con le ECU della rete, capace quindi di inviare e ricevere dati. La finalità di questo software è quella di offrire uno strumento configurabile in grado di testare il comportamento delle stazioni in determinati contesti applicativi, rilevando possibili malfunzionamenti o vulnerabilità del sistema. Nello specifico, il *software* si suddivide in due moduli: *Whitebox* e *Blackbox*.
Il modulo di Whitebox permette di verificare il corretto funzionamento di una ECU avendo una conoscenza a priori della rete (frequenza di trasmissione, protocollo di rete, DBC della ECU), mentre il modulo di Blackbox non avendo nessuna informazione sulla rete a cui è connesso, dopo un periodo iniziale di ascolto in cui vengono analizzati i dati ricevuti effettua degli attacchi informatici per alterare il normale stato di funzionamento della rete.

1.3 Zirak S.r.l

Zirak S.r.l è una PMI italiana, fondata nel 2000. È formata da quattro dipartimenti distinti, operanti in settori complementari dell'Information Technology: Automotive, IOT & ML, Mobile Applications, Web Solutions. La ZIRAK si avvale di un laboratorio di Ricerca e Sviluppo in Portogallo, partecipa a diversi progetti di ricerca Europei e collabora con università ed istituti superiori internazionali. La sede principale dell'azienda si trova a Mondovì; il dipartimento ZIRAK Automotive Embedded ha una lunga e proficua collaborazione con le principali case automobilistiche internazionali, Tier1 e Tier2**.** Contribuisce a vari progetti in collaborazione con i principali Tier1 quali Marelli, MTA, Magna, BOSCH per clienti finali come BMW, PSA, Alfa Romeo, Porsche, Ferrari, FCA e nell'ambito di GENIVI Alliance, per studio e sviluppo di sistemi basati su Linux, Autosar e Android con focus specifico in: Cybersecurity, CarPlay ed Android Auto. Nel suo operare l'azienda si è contraddistinta per due forti obiettivi:
innovazione – sviluppare e proporre soluzioni appartenenti alle nuove frontiere tecnologiche.
soddisfazione del cliente – studiare con l'acquirente soluzioni ottimali e personalizzate.
La società si propone dunque come interlocutore unico per tematiche connesse al mondo dell'*Information and Communication Technology*, dall'analisi delle esigenze, alla progettazione e realizzazione di soluzioni personalizzate.

1.4 Organizzazione della tesi

In questa tesi verranno inizialmente analizzati nel dettaglio le caratteristiche del protocollo CAN, spiegando le motivazioni alla base dei vantaggi offerti dallo standard e le vulnerabilità intrinseche che lo rendono particolarmente fragile agli attacchi informatici. Successivamente saranno presentati i dispositivi *hardware* utilizzati, il linguaggio di programmazione e le principali librerie utilizzate, la logica risolutiva del programma seguita da una dettagliata analisi del codice, motivando le scelte implementative. Saranno infine discussi i risultati ottenuti, mettendo in luce i possibili sviluppi futuri del progetto. Sulla base di quanto fatto in precedenza dal tesista in Zirak S.r.l., Ian del Rio, i principi concettuali della tesi rimangono gli stessi, con la prerogativa di effettuare una radicale modifica sullo sviluppo del *software* tramite un *porting* del codice.

CAN Bus

2.1 Evoluzione della rete

CAN bus fu rilasciato da Robert Bosch nel 1986, nel 1991 la Mercedes-Benz W140 divenne il primo veicolo su cui fu installato.
Successivamente nel 1993 la International Organization for Standardization (ISO), rilasciò due standard distinti ISO 11898-1 che ricopre il data-link layer e ISO 11898-2 relativo al livello fisico. Nel 2012 Bosch rilasciò CAN Flexible Data-Rate, a differenza dello standard originale CAN F.D permette l'utilizzo di un bit rate più veloce, fino a 5 Mbps ed inoltre supporta trame con payload di lunghezza variabile. In riferimento al modello ISO/OSI lo standard CAN implementa i primi due strati della pila

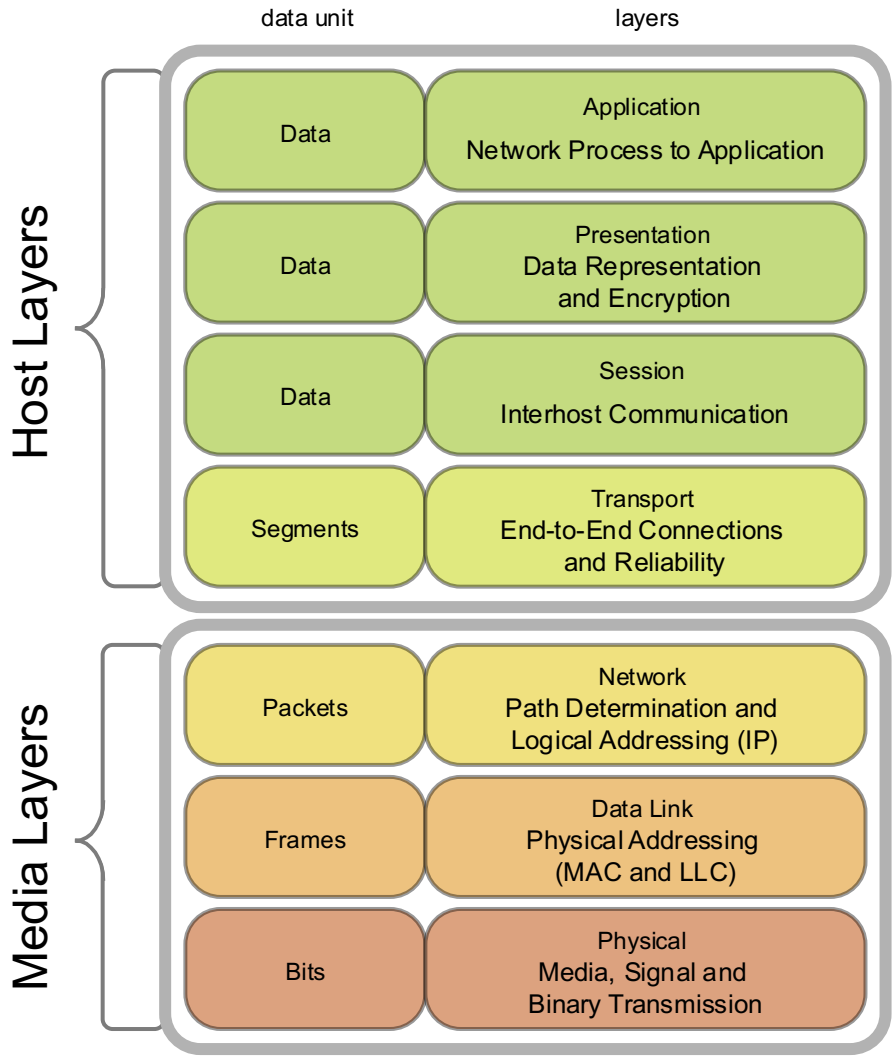

Figura 2.1: rappresentazione del modello ISO/OSI

Analizziamo nel dettaglio i livelli 1 e 2 della pila ISO/OSI del protocollo CAN.

2.2 Livello Fisico

Il primo livello del modello ISO/OSI è quello responsabile della conversione dei bit in segnali fisici, nel caso specifico di CAN questi bit vengono convertiti in delle tensioni elettriche, trasmessi successivamente sul mezzo fisico, ovvero il bus.

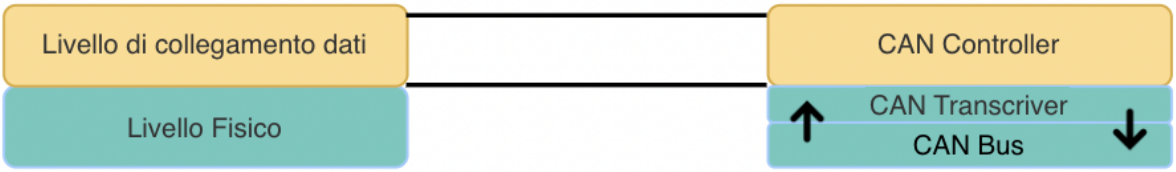

Figura 2.2: Rappresentazione modello ISO/OSI per un nodo di rete CAN

Il CAN Trascriver ed il CAN Controller sono i due componenti che formano un nodo di rete CAN, il Controller ha il compito di processare le informazioni da inviare sul bus o ricevute dal bus. Il CAN Trascriver è l'interfaccia presente tra il CAN Controller ed il CAN Bus, il Trascriver ha il compito di effettuare la traduzione bit-segnale elettrico.
In genere i due componenti sono elettricamente isolati, in questo modo se una sovratensione dovesse danneggiare il Trascriver, il Controller non subirebbe danni.

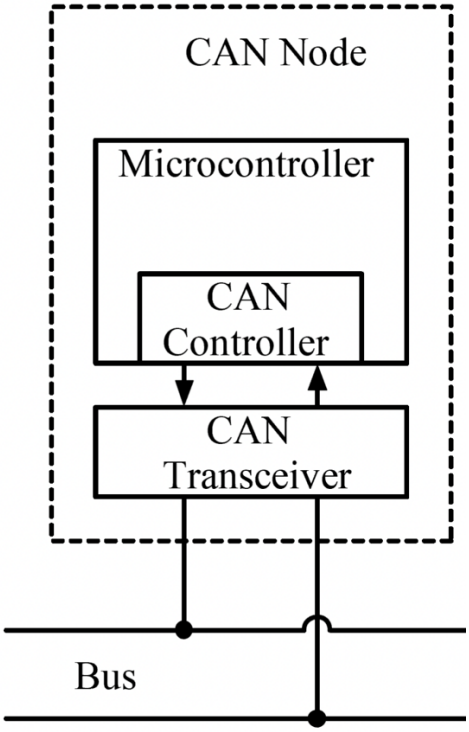

Figura 2.3: Rappresentazione nodo di rete CAN

I segnali elettrici trasmetti sul bus vengono rilevati dal CAN Transceiver che ne effettua la conversione in bits per poi passare la trama al livello superiore, ovvero al CAN Controller posto al livello data-link, il quale ne effettuerà un'elaborazione.

Come già evidenziato nell'introduzione, uno dei principali pregi di CAN è la resistenza ai disturbi elettromagnetici che permette il suo funzionamento in ambienti ostili, analizziamo le motivazioni.

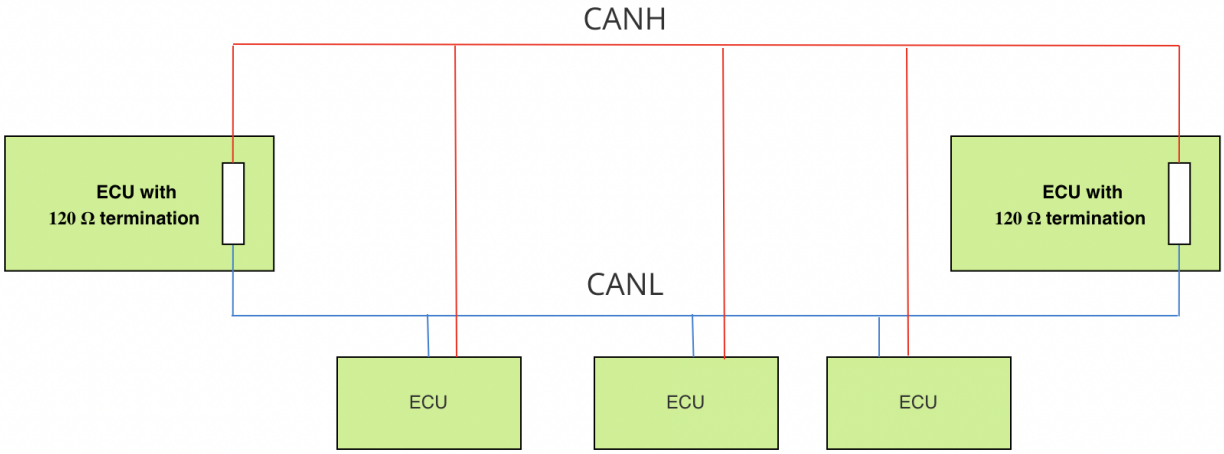

Figura 2.4: Rappresentazione CAN Bus a livello elettronico

Il CAN Bus è composto da due fili: CAN high (CANH) e CAN low (CANL) che formano un doppino intrecciato, un'impedenza da 120 ohm è posta alla terminazione di entrambi i cavi, questa tecnica in elettronica è largamente utilizzata per la costruzione dei bus in quanto riduce la distorsione delle tensioni elettriche.
La resistenza ai disturbi è anche ottenuta grazie alla tecnica del *differential signaling*: sia il CANH che il CANL sono ad una tensione di 2.5 Volt corrispondente allo stato logico 1.
Nel momento in cui deve essere trasmesso uno stato logico 0, CANH subisce un innalzamento di potenziale di 1 Volt (arrivando quindi a 3.5 Volt) in modo duale CANL subisce una diminuzione di 1 Volt.
I due fili del bus vengono monitorati costantemente dai trascriver delle ECU, nel caso in cui dovesse essere rilevato un innalzamento di 1 V sul CANH (oppure un abbassamento di 1 V sul CANL), se l'altro filo non dovesse presentare una modifica coerente, allora la variazione di potenziale verrà considerata rumore e quindi scartata.
La lunghezza massima del bus dipende dalla velocità trasmissiva usata. È richiesto che il fronte di salita di un bit possa viaggiare lungo tutto il bus e poi tornare indietro prima che il bit sia campionato.

Lunghezza del bus (metri)	Max bit rate (bit/sec)
40	1 Mbit/s
100	500 Kbit/s
200	250 Kbit/sec
500	125 Kbit/s
6K	10 Kbit/s

Figura 2.5: Tabella che mostra la lunghezza massima del bus CAN in base al bit/rate

Come intuibile, nel contesto automobilistico non è necessario disporre di bus di elevata lunghezza per cui è possibile utilizzare, senza particolari limitazioni, una frequenza di trasmissione di 1 Mbit, tuttavia bit/rate di 125 Kbit/s o 250 Kbit/s sono più comuni.
A causa delle limitazioni fisiche della rete è consigliato non interconnettere più di 110 ECU.

2.3 Livello di collegamento dati

Una particolarità del data link layer, a confronto con quello di Ethernet è relativo alla sua semplicità, nello specifico CAN non prevede la possibilità di instradare trame mediante comunicazione unicast, nessun campo della trama presenta né un indirizzo sorgente né un indirizzo di destinazione. Per cui a differenza di Ethernet, alle stazioni non è associato nessun indirizzo MAC ma internamente al frame è presente un campo ID, ovvero un identificatore del contenuto del messaggio, univoco in tutta la rete.
Ogni identificativo codifica un determinato evento (temperatura motore, livello di benzina del serbatoio, accensione della radio), associato ad esso vi è un campo data che ne rappresenta un'informazione.
Dato il CAN Frame 2.0A riportato qui sotto in figura, analizziamo nel dettaglio ogni suo singolo campo:

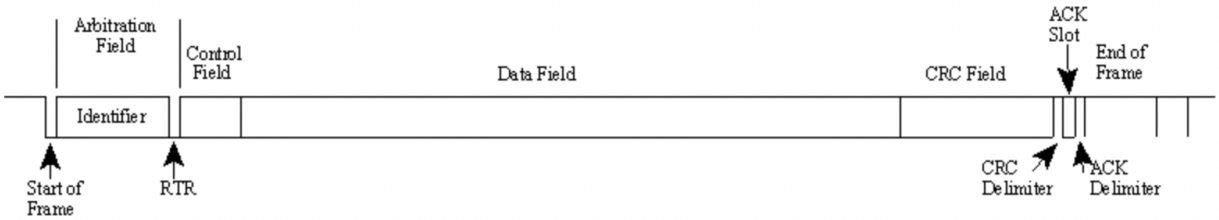

Figura 2.6: Formato CAN Frame 2.0A

- **Start of Frame:**
  Inizio del frame, campo costituito da un bit che deve sempre essere dominante ovvero un bit 0, questo perché un bit 1 non comporta delle variazioni di potenziale sul bus, sia il CANH che il CANL rimangono ad una tensione di 2.5 Volt.
- **Identifier:**
  Stringa di 11 bit che rappresenta l'identificatore univoco di un certo messaggio.
- **Remote Trasmission Request:**
  In alcuni casi, una ECU può fare una *request*, ovvero può richiedere informazioni su un messaggio avente uno specifico ID, in questo caso il flag viene settato ad un bit recessivo, nel caso di un frame normale invece il flag viene settato a 0.
  Nel caso di una request, il nodo interessato risponderà inviando un frame con il dato richiesto.
- **Identifier Extension:**
  Il protocollo CAN si suddivide in due versioni: CAN 2.0A e CAN 2.0B in cui la lunghezza del campo ID vale rispettivamente 11 e 29 bit.
  Questo flag identifica la versione utilizzata.
- **Reserved Bit:**
  Riservato per usi futuri, bit non utilizzato settato a 0.
- **Data Length Code:**
  Rappresenta il numero di byte che compongono il campo data, di massimo 8 byte.
- **Data:**
  Informazione contenuta nel messaggio.
- **Cyclic Redundance Check:**
  Campo CRC (sarà discusso nelle prossime sezioni) composto da 15 bit.
- **CRC Delimiter:**
  Bit recessivo che delimita la fine del campo CRC, serve per rilevare eventuali errori di trasmissione.

- **ACK:**
  Bit inizialmente trasmesso come 0, che viene poi sovrascritto nel momento in cui una ECU riceve correttamente l'intero frame. Nel caso in cui il bit dovesse rimanere a 0, la stazione dovrà ritrasmettere.
- **ACK Delimiter:**
  Analogo al CRC Delimiter.
- **End of Frame:**
  Una stringa di 7 bit recessivi che determina la fine della trama.

Ogni ECU ha un proprio DBC, ovvero un database contenente gli ID di tutti i messaggi che la stazione deve ricevere e che può trasmettere, il DBC contiene inoltre le informazioni necessarie per decodificare ognuno di questi messaggi convertendo i dati in valori fisici oppure in un formato leggibile all'uomo.
Il DBC è un file testuale contenuto nel CAN Controller, il formato del file è il seguente:

```
BO_ 500 IO_DEBUG: 4 IO
SG_ IO_DEBUG_test_unsigned : 0|8@1+ (1,0) [0|0] "" DBG
SG_ IO_DEBUG_test_signed : 8|8@1- (1,-128) [0|0] "" DBG
```

Il messaggio IO_DEBUG è codificato dall'identificativo 500: internamente al messaggio sono presenti due segnali distinti da 8 bit, il primo senza segno ed il secondo con segno.
A seguito della trasmissione del campo ID, ogni ECU in ascolto verifica se l'identificativo trasmesso è presente nel database, in questo modo un nodo ricevitore può così verificare il contenuto del messaggio e filtrare i soli frames a cui è interessato, ignorando gli altri.

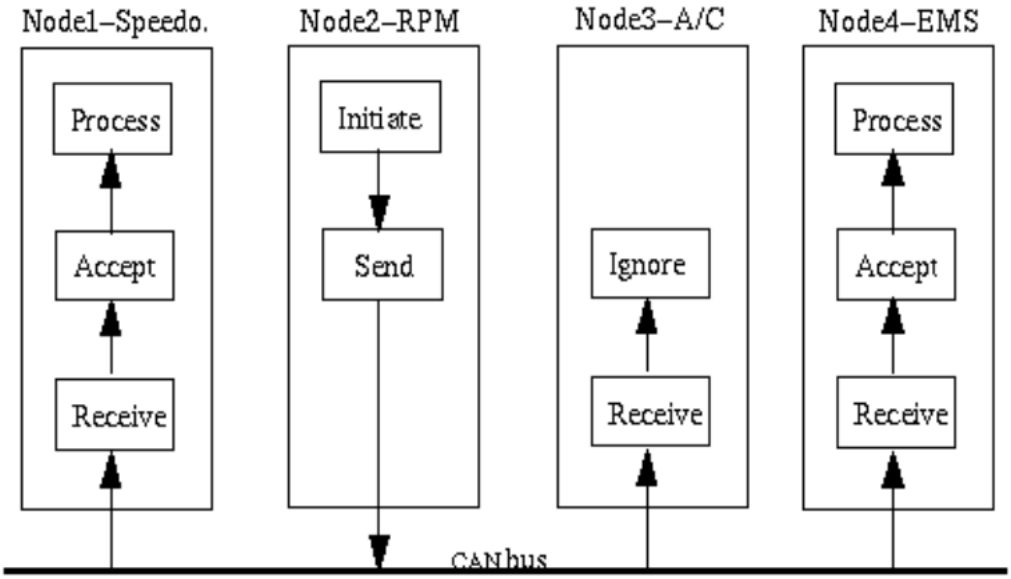

Figura 2.7: Filtraggio di una trama CAN

La figura 2.7 presenta un ipotetico contesto pratico in cui una ECU trasmette un frame contente informazioni relative al numero dei giri al minuto del motore, il tachimetro e l'Engine Management System accettano il pacchetto, l'unità di gestione dell'aria condizionata lo ignora.

L'instradamento delle trame è quindi di tipo *multicast*, ovvero la trama viene ricevuta da un gruppo di destinatari.

2.4 Cyclic Redundance Check (CRC)

Tra i campi del CAN Frame vi è il CRC, ma cos'è e come funziona?
Lo scopo del CRC è quello di offrire uno strumento in grado di rilevare eventuali errori di trasmissione, vediamo nel dettaglio come funziona:

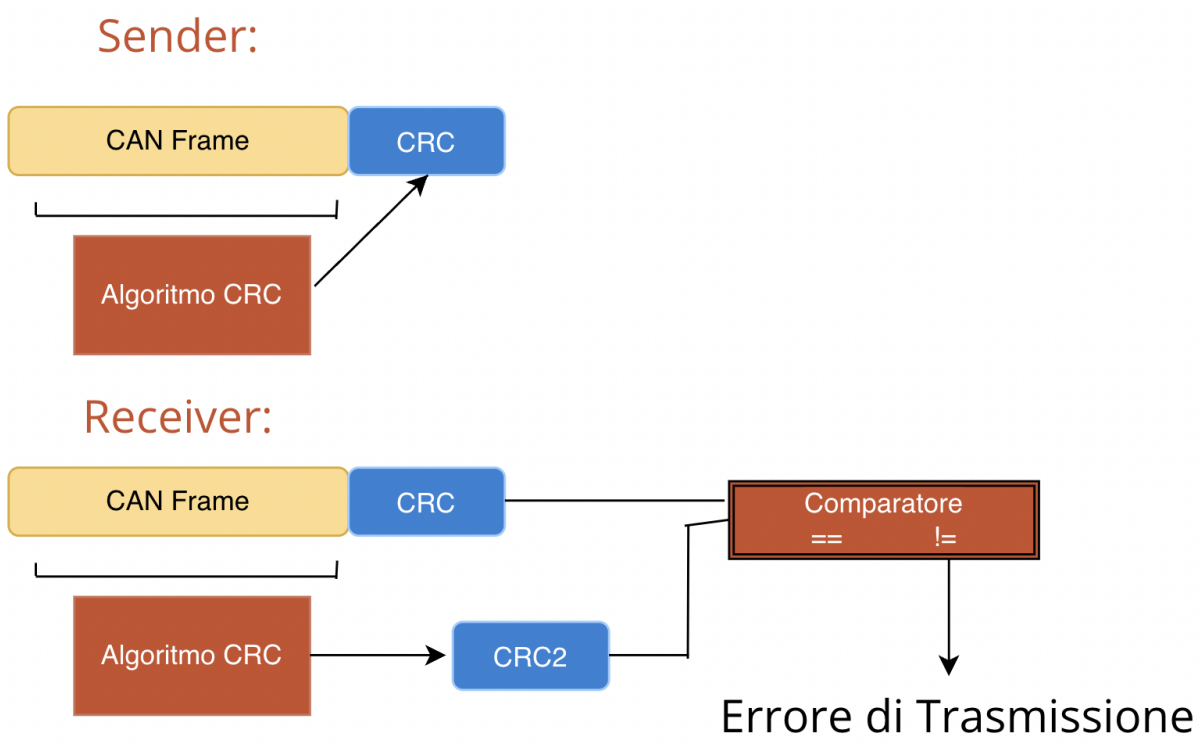

Figura 2.8: Rilevamento di errori di trasmissione tramite CRC

Il trasmettitore calcolerà un Hash (oppure un Digest) della trama, ovvero tramite un certo algoritmo verrà calcolato un riassunto del frame, l'algoritmo in questione potrebbe anche essere semplicemente uno XOR iterativo in cui vengono presi n byte alla volta e se ne calcola il risultato, che verrà poi scritto nell'apposito campo CRC. Il ricevitore nel mentre in cui riceve i bit del frame, effettua il calcolo del CRC, di seguito non appena l'intera trama sarà ricevuta, verrà confrontato il CRC appena calcolato con quello presente nel frame, se i due campi coincidono allora molto probabilmente non ci sono stati errori di trasmissione. Tutte le stazioni in ascolto calcolano il CRC e poi eseguono il confronto con il contenuto del campo, nel caso in cui nessuna stazioni rilevi l'uguaglianza allora il bit di ACK non verrà sovrascritto e rimarrà a 0, obbligando il trasmettitore a ritrasmettere la trama. Viceversa, se almeno una stazione rileva l'uguaglianza tra il CRC calcolato e quello presente nel frame, il bit di ACK verrà sovrascritto e la trama non verrà ritrasmessa.

2.5 Fase d'arbitraggio

Il bus per sua natura permette a più stazioni di leggere da questo, mentre una sola stazione alla volta può scriverci, questo vincolo impone la necessità di utilizzare un metodo per regolamentare l'accesso al mezzo fisico, ovvero il CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*). Cosa succede se due stazioni vogliono utilizzare contemporaneamente il bus per trasmettere un frame?
Il CAN Transceiver dopo aver ricevuto un frame dal CAN Controller, ascolta il canale per controllare se sia libero, a causa della natura del protocollo in cui un "1" logico corrisponde allo stato recessivo e lo "0" allo stato dominante, la rilevazione di un flusso contiguo di "1" da parte del Transceiver comporta la possibilità di utilizzare il bus. Nel momento in cui una ECU inizia la trasmissione di un frame, oltre a trasmettere il flusso di bits della trama, ascolta in contemporanea il canale per verificare che ciò che scrive sul bus è ciò che ascolta, qualora dovesse essere letto un valore diverso da quello scritto allora la ECU si renderà conto che un'altra stazione sta trasmettendo in contemporanea e che deve aspettare che quest'ultima termini la trasmissione per poter utilizzare successivamente il canale, una volta libero. Nel caso in cui si dovesse sviluppare un conflitto, ovvero nel momento in cui due o più stazioni trasmettitrici inviano un bit differente sul canale, il bit dominante andrà a sovrascrivere quello recessivo, per cui lo "0" logico sarà quello effettivamente scritto sul bus. Analizziamo delle casistiche di conflitto, per comprendere il comportamento della rete:

	Start bit	ID bits											The rest of the frame
		10	9	8	7	6	5	4	3	2	1	0	
Node 15	0	0	0	0	0	0	0	0	1	1	1	1	
Node 16	0	0	0	0	0	0	0	1	Stopped Transmitting				
CAN data	0	0	0	0	0	0	0	0	1	1	1	1	

Figura 2.9: Trasmissione contemporanea sul bus

Consideriamo il caso in cui sia il Transceiver del nodo 15 che quello del nodo 16, ricevono dal rispettivo Controller una trama da inviare sul bus. Dopo un ascolto preventivo del canale in cui risulta libero, entrambi i nodi iniziano la trasmissione contemporanea della trama, a seguito del campo SOF, vengono trasmessi gli 11 bit dell'identificativo, partendo dal MSB (Most significant bit) fino al LSB (Less significant bit). È importante notare che i due nodi non si rendono conto dell'utilizzo contemporaneo del bus fintanto che entrambi trasmettono bit uguali, nel momento in cui i due nodi trasmetto il bit con indice 4, si ha un conflitto. Il nodo numero 16 trasmette un bit recessivo mentre il nodo 15 trasmette un bit dominante, ogni conflitto viene risolto in accordo con il meccanismo del wired-and, per cui alla fine viene trasmesso sul bus uno 0 logico, il nodo 15 vincerà il conflitto e continuerà la trasmissione della trama. Nel mentre il nodo 16 si renderà conto di aver perso il conflitto perché invia sul bus un 1 logico ma ascoltando rileva uno 0, per cui interromperà la trasmissione della trama e si metterà in ascolto aspettando che il nodo 15 termini la trasmissione.

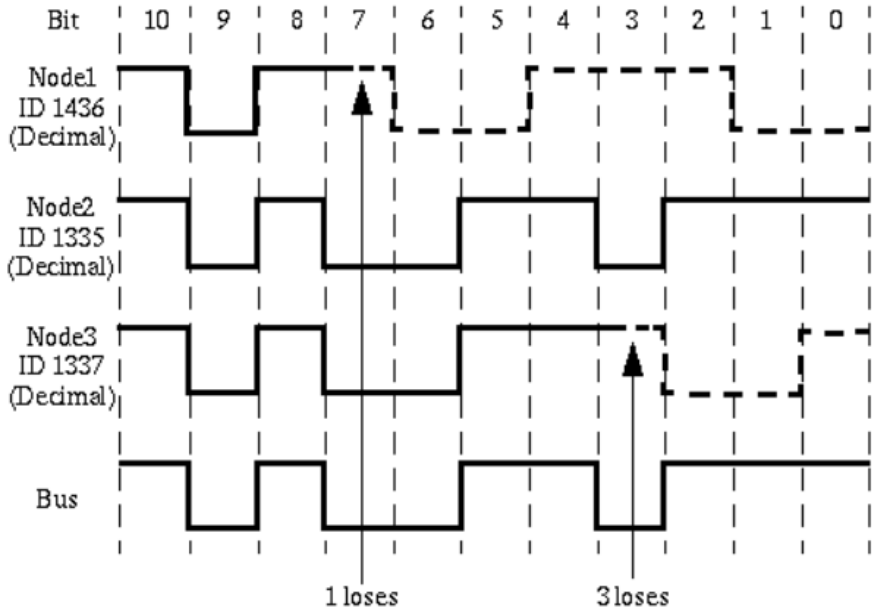

Figura 2.10: Trasmissione contemporanea sul Bus 2

Nell'esempio in figura tre nodi trasmettono un frame in contemporanea: il nodo 1 perde la competizione durante la trasmissione del bit 7, il nodo 3 invece durante la trasmissione del bit 3. I nodi perdenti diventano ricevitori e ritenteranno una trasmissione nel momento in cui il bus tornerà libero.
È importante notare che i bit della trama vengono trasmessi in ordine, ovvero a seguito del campo Start of Frame verrà trasmesso l'identificatore, per cui identificatori con un valore numerico più basso avranno priorità maggiore e vinceranno con maggiore probabilità un eventuale conflitto. Questa politica garantisce il determinismo dell'accesso al bus, e l'assenza di periodi di inattività del canale.
Come intuibile quindi, l'associazione messaggio-identificativo non è casuale, in genere vengono codificati da un identificativo a priorità maggiore tutti gli eventi che hanno un'urgenza maggiore rispetto ad altri.
Possiamo immaginare ad esempio, che l'evento riferito all'accensione/spegnimento della radio oppure dell'aria condizionata abbia una priorità bassa, rispetto ad esempio al messaggio che dovrebbe innescare la *eCall* (chiamata di emergenza).
Nel paragrafo 2.2 relativo al livello fisico, avevamo riportato la necessità di limitare il numero di ECU della rete ad un massimo di 110, il bus è unico ed è condiviso da tutte le stazioni, all'aumentare del numero di stazioni cresce la probabilità di avere un conflitto che potrebbe portare alcune stazioni trasmettitrici di messaggi con priorità molto bassa a non poter trasmettere la propria trama per un tempo molto lungo, per cui è necessario limitare il numero di stazioni anche per questo.
Questa assunzione risulterà particolarmente significativa nel momento in cui saranno presentati alcune tipologie di attacco informatico, particolarmente efficaci sulle reti CAN.

2.6 Error Detection

Un altro pregio di CAN è la sua elevata affidabilità dovuta al rilevamento degli errori, garantendo la correttezza delle informazioni e la loro coerenza sull'intero sistema, i meccanismi di rilevamento degli errori sono i seguenti:

- **Bit Stuffing:** Ogni qual volta che una sequenza di 5 bit dello stesso valore vengono inviati, viene inserito un bit aggiuntivo avente polarità opposta con la sequenza precedente. Se una stazione in ascolto dovesse rilevare una sequenza di più di 5 bit con stessa polarità, allora verrà generato un errore.
- **Bit monitoring:** I CAN Trascriver di ogni ECU ascoltano costantemente il bus anche quando inviano un frame, nel caso in cui dovesse essere rilevato un bit diverso da quello che è stato trasmesso (e non ci si trova né nella fase di arbitraggio né in quella di Acknowleadge), allora verrà generato un errore.
- **Acknowledgement check:** Nel momento in cui la stazione trasmettitrice invia il campo ACK, formato da un bit recessivo, si aspetta che il bit venga sovrascritto da un bit dominate. Se nessuna ECU dovesse sovrascrivere il valore, allora il trasmettitore genererà un errore.
- **CRC:** Se una qualsiasi stazione in ascolto riceve un checksum diverso da quello calcolato, verrà generato un errore.
- **Frame Error:** Il formato del frame è fisso e presenta alcuni campi di delimitazione, ogni ECU in ascolto confronta i bit ricevuti con quelli che si aspetta di ricevere e se rileva una incoerenza genererà un errore. Questo può avvenire, ad esempio nel caso in cui si dovesse riceve un bit dominante come delimitatore del CRC/ACK, o nel caso in cui sia violato il formato dell'End of Frame.

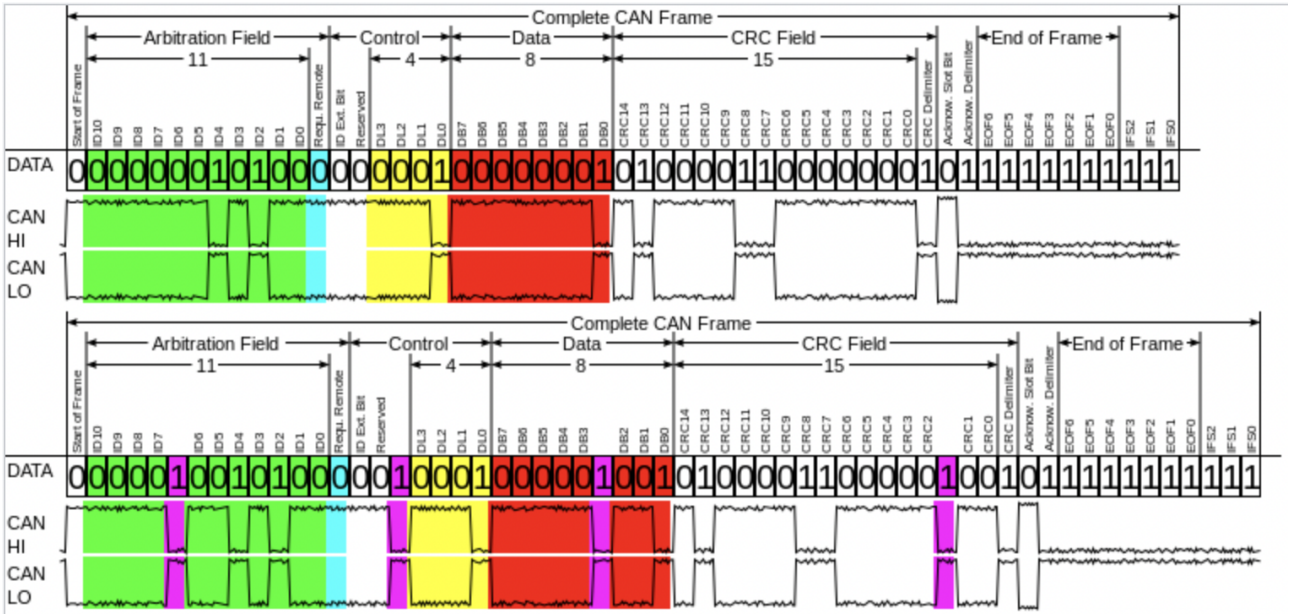

Figura 2.11: Bit stuffing

Nella figura 2.11 possiamo vedere colorati di porpora i bit di stuffing: nello specifico dopo 5 bit dominanti viene aggiungo un bit recessivo che però di fatto non viene considerato dalle stazioni riceventi ai fini della composizione del frame.

Cosa succede quando viene generato un errore?
Nel momento in cui il CAN Controller di una ECU rileva un errore, quest'ultimo invierà un *Error Frame* ovvero un frame composto da un *Error Flag* ed un *Error Delimiter*.
L'Error Flag è una sequenza di 6 bit aventi stessa polarità che viola volontariamente il principio del bit stuffing, mentre l'Error Delimiter è costituito da una sequenza di otto zeri.
Lo scopo del Error Frame è quello di rendere noto a tutte le ECU della rete della presenza di un errore di trasmissione, violando la regola del bit stuffing si innescherà la routine di gestione dell'errore anche negli altri CAN Controller.
Ogni CAN Controller ha due contatori interni, il TEC (Transmission Error Counter) ed il REC (Reception Error Counter), ogni qualvolta che il controller rileva un Error Frame questi contatori settati inizialmente a 0 vengono incrementati (+8 al TEC per gli errori di trasmissione, +1 al REC per errori di ricezione), ogni volta che viene ricevuto correttamente un frame il REC verrà decrementato di 1, ogni volta che viene trasmesso correttamente un frame il TEC verrà decrementato di 1.
In base al valore di questi contatori differenziamo tre possibili stati di errore:

1. **Active Error State:** Il nodo funziona correttamente, i valori di TEC e di REC sono entrambi minori di 127, nel momento in cui una ECU in questo stato dovesse rilevare un errore manderà un Error Frame composto da sei bit dominanti che andranno ad interrompere la trasmissione.

2. **Passive Error State:** Il nodo si trova in questo stato quando i valori di TEC e di REC sono compresi tra 127 e 255, se una ECU si trova in questo stato è molto probabile che ci siano delle interferenze elettromagnetiche, dei disturbi o un guasto interno alla ECU stessa.
   In questo caso, nel momento in cui viene generato un errore, sarà generato un error frame composto da sei bit recessivi, così facendo nel caso in cui una stazione stia trasmettendo, l'error frame non andrà a interrompe la trasmissione.

3. **Bus off State:** Il nodo in questo stato si stacca dal bus e non partecipa più alla comunicazione, lasciando agli altri nodi la possibilità di scambiarsi informazioni.
   Una ECU entra nello stato di Bus off quando uno dei due contatori supera il valore di 255.

La presenza di questi tre stadi di errore garantisce al sistema la proprietà di robustezza bizantina, ovvero un malfunzionamento in uno dei nodi della rete non porta ad un disservizio complessivo nell'intera rete, dopo un certo tempo un nodo guasto entrerà in stato di Bus Off in modo tale da isolare il malfunzionamento e ristabilire il corretto funzionamento della rete per consentire la comunicazione tra le altre stazioni. È importante notare però che un nodo danneggiato, prima di diventare inattivo creerà dei disservizi sulla rete, fintanto che il nodo si trova in Active Error State, il rilevamento di un errore comporterà l'invio di un Error Frame di sei bit dominati che impedirà alle altre stazioni di utilizzare il canale per un certo tempo.
Nel momento in cui viene rilevato un Error Frame, la stazione trasmettitrice è obbligata a ritrasmettere la trama andando eventualmente a competere con altre stazioni che devono anch'esse trasmettere un frame, tuttavia il nodo che aveva vinto in precedenza la fase di arbitraggio ha molta probabilità di vincere nuovamente, così facendo ci saranno stazioni che dovranno aspettare un certo tempo prima di poter utilizzare il canale.

Transmission rate [kbit/s]	Elapsed time for availability [µs]
125	184
250	92
500	46
1000	23

Figura 2.12: Tabella che lega il bit/rate al tempo massimo di attesa a seguito di un guasto

La tabella in figura mostra il tempo medio necessario al canale per tornare utilizzabile a seguito del rilevamento di un errore, per ogni bit/rate. Ovviamente all'aumentare della frequenza di trasmissione (e consequenzialmente del bit/rate) diminuisce il tempo medio di attesa.

2.7 CAN FD (CAN Flexible-Data Rate)

Con gli anni, lo sviluppo nel settore automobilistico aveva portato ad una richiesta sempre maggiore di banda, le tradizionali reti CAN risultavano inadatte non potendo superare il bit rate di 1 Mbit/sec. Nel 2011 Bosch sviluppò CAN FD portando a due miglioramenti sostanziali:

1. Il payload, ovvero il campo dati può contenere fino ad un massimo di 64 byte, rispetto agli 8 byte previsti del CAN tradizionale.
2. La frequenza di trasmissione può superare il limite massimo di 1 Mbit nel momento in cui si trasmette il campo dati.

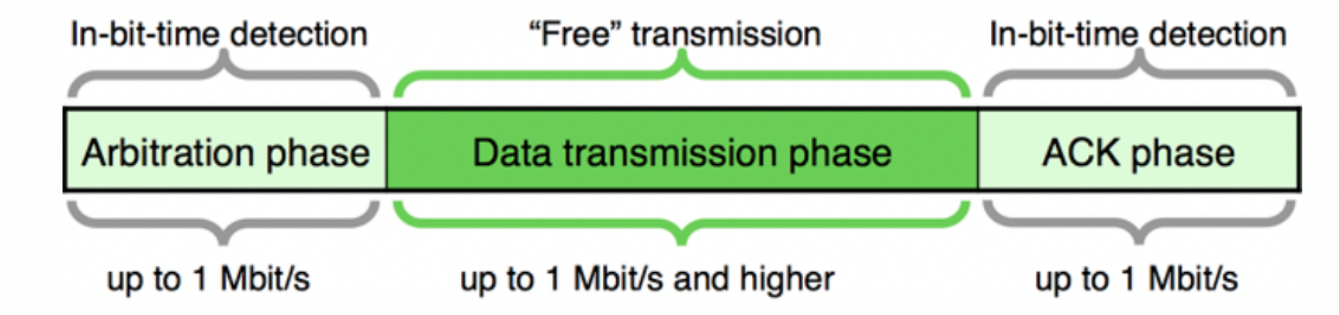

Figura 2.13: Trasmissione frame con CAN FD

Nel momento in cui deve essere trasmesso il payload non è necessario che i nodi siano sincronizzati tra loro, per cui a seguito della fase di arbitraggio è possibile aumentare il bit/rate, anche se prima dell'inizio di tale fase è necessario risincronizzare i nodi.
Supponendo un rapporto 1:8 per la trasmissione del payload, ovvero di 8 Mbit/s, la trasmissione complessiva della trama porterebbe ad un throughput sei volte maggiore rispetto al CAN tradizionale, tenendo conto che il formato della trama del CAN FD prevede un numero maggiore di bit per il campo dati, per il CRC e per l'identificatore.

2.8 SAE J1939

SAE J1939 è un protocollo che definisce i primi cinque strati della pila relativa al protocollo ISO/OSI, ovvero partendo dal livello fisico fino al livello sessione. J1939 si appoggia a CAN ISO 11898 per l'implementazione dei primi due livelli. Il formato dei pacchetti è di 29 bit, il bit rate utilizzato è tipicamente di 250 Kbit/s (per J1939/11 e J1939/15) mentre J1939/14 utilizza un bit rate di 500 Kbit/s. J1939 è tipicamente utilizzato sui veicoli commerciali pesanti: bulldozer, scavatrici, camion, autobus, macchine agricole quali ad esempio trattori e macchine mietitrici, veicoli militari e macchine per la costruzione di edifici tipo gru. Una particolarità di J1939 è che, pur utilizzando il protocollo CAN per il data link layer e per il livello fisico, presenta un campo source address ed eventualmente un campo destination address che permettono la trasmissione in unicast oltre che in broadcast. In CAN 2.0B, lo spazio di indirizzamento degli identificatori è interamente utilizzabile, questo implica che è possibile utilizzare un qualsiasi ID nel seguente intervallo: $[0, 2^{29} - 1]$, questo non vale per J1939 che impone alcuni vincoli sul formato degli identificativi. Il generico pacchetto SAE J1939 ha il seguente formato:

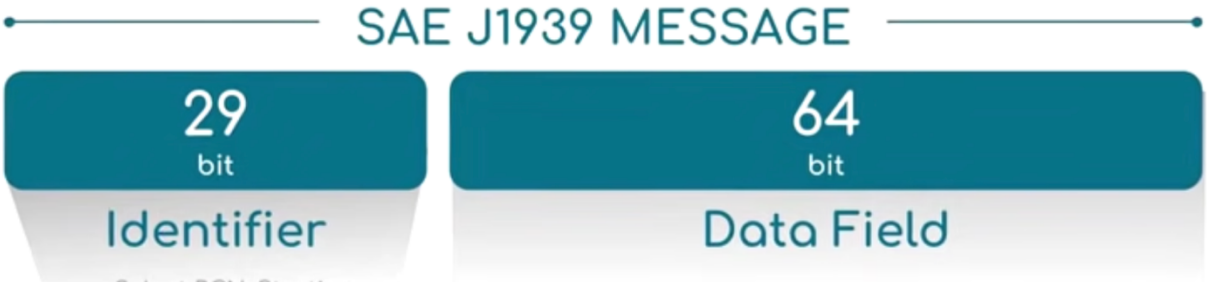

Figura 2.14: Formato pacchetto SAE J1939

Il campo di payload è codificato su 4 byte; il campo identifer è ulteriormente suddiviso in dei sotto campi, esaminiamo il formato di questo campo:

PRIORITY	RESERVED	DATA PAGE	PDU FORMAT	PDU SPECIFIC	SOURCE ADDRESS
3 bits	1 bit	1 bit	8 bits	8 bits	8 bits

Figura 2.15: formato del pacchetto

- I primi tre bit fanno riferimento alla priorità relativa alla fase di arbitraggio, la priorità massima è associata a pacchetti aventi priorità di $000_2 \rightarrow 0_{10}$.

- Il quarto bit è riservato per usi futuri ed è sempre settato a 0.

- 1 bit di Data Page, questo bit aumenta il numero dei possibili *Parameter group* che possono essere rappresentati dall'ID.

- Il campo PDU Format (PF) determina il tipo di comunicazione, ovvero se il messaggio deve essere trasmesso con un indirizzo di destinazione (comunicazione unicast) oppure in broadcast.

- Il valore del campo PDU Specific (PS) assume significati differenti in base al valore del campo precedente:

- Se il valore del PF è compreso tra 0 e 239, il messaggio è inviabile ad una stazione ed il campo PS specifica l'indirizzo di destinazione.

- Se il valore del PF è compreso tra 240 e 255, il messaggio deve essere inviato in broadcast ed il campo PS contiene una Group Extension.

- Il source address indica l'indirizzo sorgente, ovvero il mittente del messaggio.

Il Parameter Group Number (PGN) è utilizzato per riferirsi ai campi: Reserved bit, Data Page, PDU Format e PDU Specific per un totale di 18 bit. Il PGN rappresenta l'effettivo identificativo univoco che caratterizza uno specifico evento. I PGN che variano da 00FF00 fino a 00FFFF sono riservati e non posso quindi essere utilizzati, è possibile ricavare il PGN interno al campo identifier applicando la maschera 0x3FFFF00 oppure convertendo il valore in binario per poi selezionare successivamente i bit dal quarto fino al ventunesimo.

$Esempio$:

$id: 0CF00401, \ \ payload: FF\ FF\ FF\ 68\ 13\ FF\ FF\ FF$

$(0CF00401\ )_{Hex} = (0000\ 1100 \ \ \ 1111\ 0000 \ \ \ 0000\ 0100 \ \ \ 0000\ 0001)_{Bin}$

$(03FFFF00)_{Hex} = (0000\ 0011 \ \ \ 1111\ 1111 \ \ \ 1111\ 1111 \ \ \ 0000\ 0000)_{Bin}$

$Calcoliamo\ l'AND\ risultante, ovvero:\ id\ AND\ Mask$

$$\begin{array}{c|cccc} Mask & 0000\ 0011 & 1111\ 1111 & 1111\ 1111 & 0000\ 0000 \\ \hline Id & 0000\ 1100 & 1111\ 0000 & 0000\ 0100 & 0000\ 0001 \end{array}$$

$$PNG = (0000\ 0000 \ \ \ 1111\ 0000 \ \ \ 0000\ 0100)_{Bin} = (0C\ F0\ 04\ 01)_{Hex} = (61444)_{Dec}$$

A differenza del protocollo CAN, in cui la codificazione evento-identifier è arbitraria ed è a discrezione di ogni casa produttrice definire i file DBC di ogni ECU nel modo in cui preferiscono, per il protocollo SEA J1939 vi è una documentazione specifica, in cui ogni PGN è associato ad uno specifico identificativo, in riferimento all'esempio precedente consultando la J1939-17 documentation abbiamo:

PGN61444 - Electronic Engine Controller 1 - EEC1

Transmission Repetition Rate: Engine speed dependent
Data Length: 8 bytes
Data Page: 0
PDU Format: 240
PDU Specific: 4
Default Priority: 3
Parameter Group Number: 61444 (0x00F004)

Bit Start/Byte	Length	SPN ID	SPN Description
1.1	4 bits	899	Engine Torque Mode
2	1 byte	512	Driver's Demand Engine - % Torque
3	1 byte	513	Actual Engine - Percent Torque
4-5	2 bytes	190	Engine Speed
6	1 byte	1483	SA of Controlling Device for Engine Control
7.1	4 bits	1675	Engine Starter Mode
8	1 byte	2432	Engine Demand - Percent Torque

Figura 2.16: J1939-17 Documentation

HARDWARE

In questa sezione analizzeremo l'Hardware utilizzato per lo sviluppo del progetto.
Il software è designato per funzionare su Raspberry Pi, uno dei SBCs (*single-board computer*) più famosi in commercio, rilasciato dalla Raspberry Pi Foundation nel 2011.
Oltre alla suddetta scheda embedded, tra gli altri dispositivi usati citiamo: il Raspberry LCD (*Liquid Crystal Display*) per far visionare all'utente l'interfaccia grafica dell'applicazione, una scheda Micro SD necessaria in quanto il Raspberry Pi è sprovvisto di un Hard Disk o di un disco allo stato solido e la PICAN FD Duo Board per consentire l'invio e la ricezione delle trame CAN.

3.1 Raspberry Pi

Inizialmente ideata come strumento per l'apprendimento dell'informatica nelle scuole e nei paesi in via di sviluppo, la prerogativa di queste schede elettroniche era quella di offrire uno strumento economico con prestazioni non elevate per motivi, per l'appunto didattici o istruttivi.
Il primo modello vendette molto più del previsto suscitando l'interesse di un target di mercato differente da quello preventivato, grazie alla sua modularità e basso costo il Raspberry Pi è stato utilizzato in diversi contesti applicativi, suscitando l'interesse degli hobbisti elettronici. La presenza di porte USB ed HDMI rendeva il Raspberry uno strumento interfacciabile con le più comuni periferiche Hardware, limitando le conoscenze tecniche necessarie per il suo utilizzo.

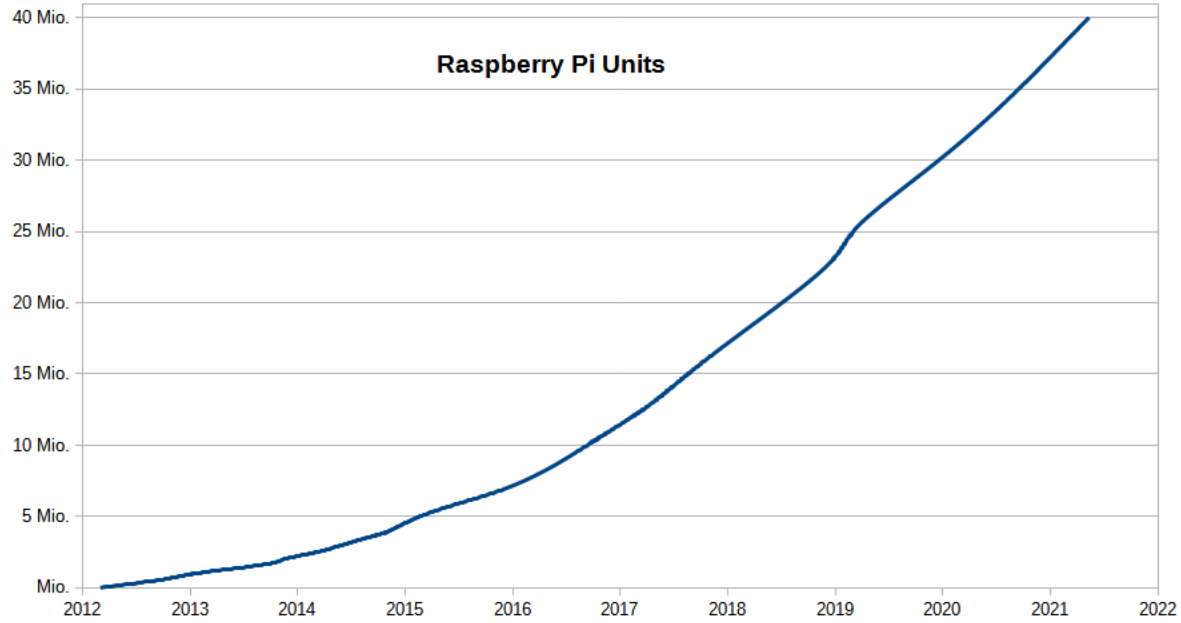

Figura 3xs.1: Raspberry Pi venduti negli anni

Con il tempo questi dispositivi embedded hanno letteralmente monopolizzato il mercato degli SBCs, avendo venduto più di 40 milioni di schede.
Nello specifico il modello che è stato utilizzato per questa tesi è il Raspberry Pi 3 B+, rilasciato nel 2018, rispetto al modello precedente fu aggiunto il PoE (*Power Over Ethernet*) ovvero la possibilità di alimentare il dispositivo direttamente da un cavo Ethernet (utilizzato sia per l'alimentazione che per traportare i dati), l'USB Boot ed il Network Boot, una Gigabit Ethernet tre volte più veloce oltre che un processore più veloce da 1.4 GHz.

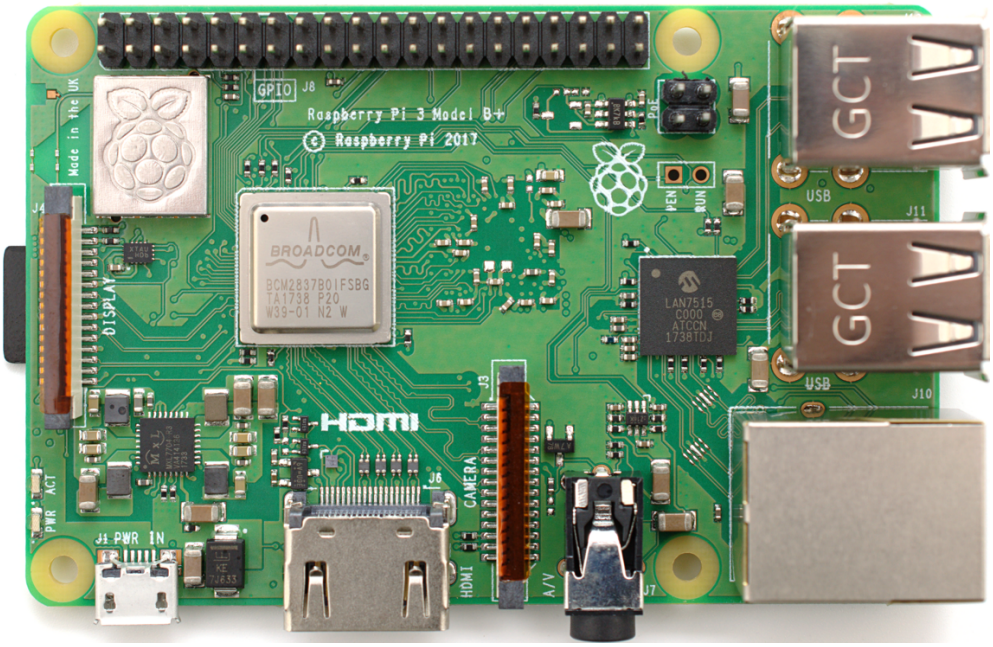

Figura 3.2: Raspberry Pi 3 B+

Riportiamo nel dettaglio le caratteristiche tecniche del Raspberry Pi 3+:

- Broadcom BCM2837B0, SoC Cortex-A53 (ARMv8) da 1,4 GHz e 64 bit.
- Header con GPIO (*General Purpose Input/Output*) a 40 pin
- SDRAM LPDDR2 da 1 GB
- Bluethooth 4.2/BLE
- Supporto Power over Ethernet
- Wireless LAN dual-band da 2.4 GHz e 5 GHz 802.11.b/g/n/ac
- Uscita video HDMI full-size
- 4 Porte USB 2.0
- Alimentazione tramite cavo micro USB 5 V/2.5 A
- Porta DSI per il collegamento del display Touch Screen
- Porta CSI per il collegamento della Raspberry Pi Camera
- Uscita audio stereo a 4 poli
- Porta micro-SD
- Rete PXE potenziata

3.2 Micro-SD

A differenze di Arduino che è un microcontrollore, Raspberry Pi è un vero e proprio computer in cui l'Hardware non è accessibile direttamente, ma il suo accesso viene regolamentato dal sistema operativo. Essendo sprovvisto di un Hard Disk, il Raspberry ha necessità di una unità di storage esterna su cui venga caricato il file immagine (estensione .iso) del sistema operativo.
Raspberry Pi supporta diversi sistemi operativi basati tutti su Linux ad eccezione di Windows IoT Core. La scelta è ricaduta su Raspbian, un sistema operativo ottimizzato per Raspberry, principalmente perché rispetto ad altre distribuzioni la gestione e l'utilizzo dei pin di GPIO è molto più semplice. Una scheda Micro-SD da 64 GB è stata utilizzata per memorizzare la iso Raspbian GNU/Linux 10 (buster).

3.3 Raspberry Pi LCD 7’’ Touch Screen

È necessario offrire uno strumento di interazione intuitivo, mediante il quale l’utente sia in grado di immettere informazioni in modo semplice e pratico e allo stesso tempo offrire delle informazioni visive al termine dell’esecuzione del programma.

Figura 3.3: Raspberry Pi Touch Screen

Il Raspberry Pi LCD Touch Screen rappresenta lo strumento utilizzato per soddisfare questi propositi, in particolare ogni operazione di input è realizzata mediante l’interazione tra l’utente ed il display touch capacitivo (pressione dei bottoni, spunta dei checkbox ecc..) mentre l’output viene proiettato direttamente su schermo. In questo modo si evita l’uso di periferiche di I/O quali mouse, tastiera, e monitor connettibile tramite HDMI, riducendo l’ingombro fisico e garantendo conseguenzialmente un dispositivo compatto e portatile per l’utilizzo.

Figura 3.4: Connessione tra Raspberry Pi e display LCD

Il collegamento tra la scheda ed il display LCD viene realizzato mediante GPIO, nello specifico si collega la piattina del monitor al connettore J4 del Raspberry, i pin di 5V e GND del Raspberry forniscono l'alimentazione al display.
Riportiamo le caratteristiche tecniche del Raspberry Pi LCD Touch Screen:

- Monitor Touch capacitivo
- Angolo di visione di 70°
- Dimensioni dello schermo: 110x194x20 mm
- Misura dello schermo visibile: 86x155 mm
- Alimentazione tramite micro USB da minimo 500 mA
- Colore RGB a 24 bit
- Risoluzione: 800x480 pixel a 60 fps (*frames per second*)
- Tocco capacitivo a 10 dita
- Assenza di interferenze elettroniche

3.4 PiCAN FD Board

In maniera analoga ad Ethernet, anche sulle reti CAN è necessario utilizzare delle schede di rete per ottenere connettività con la rete stessa. Le NIC (*Network Interface Card*) sono delle interfacce digitali che svolgono tutte le funzionalità di elaborazione e di trasmissione dei dati sulla rete.
Le schede di rete CAN compongono il CAN Controller ed il CAN Trascriver, nello specifico il modello utilizzato è la PiCAN FD Board.

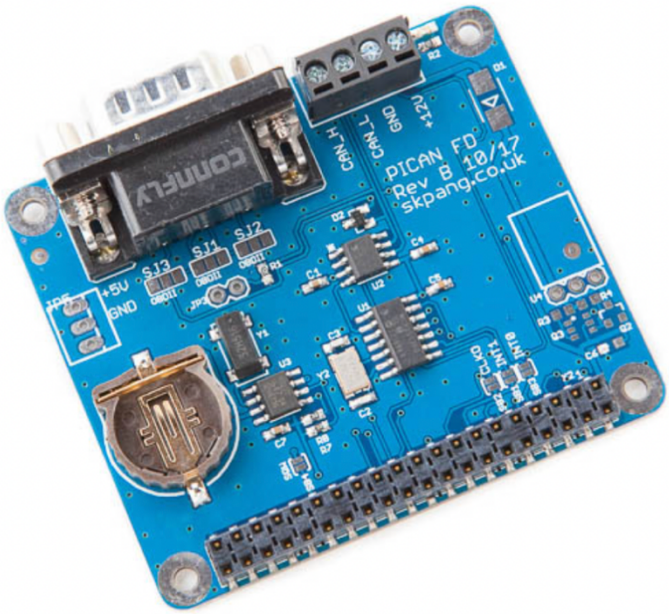

Figura 3.5: PiCAN FD Board

La suddetta scheda permette la connettività sia sulle reti CAN che sulle reti CAN FD, l'interconnessione tra Raspberry Pi e la PICAN FD Board viene fatta tramite cavo DB-9 o tramite un morsetto a 4 vie. La PiCAN FD Board è dotata di un Real-Time Clock, il CAN FD Controller è implementato da un chip MCP2517FD, mentre il Trascriver da un chip MCP2562FD.

Riportiamo le specifiche tecniche della PiCAN FD:

- Fase d'arbitraggio ad un Bit Rate massimo di 1 Mbit/sec
- Trasmissione del Payload ad un Bit Rate Massimo di 8 Mbit/sec
- Compatibilità del CAN Controller sia con CAN2.0B che con CAN FD
- Conformità con lo standard ISO11898-1:2015
- Interfaccia SPI ad alta velocità
- Compatibilità con i cavi OBDII e DB-9.
- Impedenza da 120Ω
- LCD seriale
- indicatore LED
- Quattro fori di fissaggio fixing, conformi con lo standard Pi Hat
- driver SocketCAN
- Interrupt RX al GPIO25
- RTC con backup della batteria

Attacchi Informatici sulle reti CAN

4.1 Introduzione

In seguito all'illustrazione delle caratteristiche e dei principi di funzionamento del protocollo CAN, tenendo conto anche della breve digressione sull'hardware utilizzato, focalizziamoci sulle vulnerabilità che ne scaturiscono e su eventuali attacchi informatici che potrebbero avvenire.
Occorre sottolineare che alla base dei principali attacchi a cui si farà riferimento, vi è l'assunzione forte per cui un host malevolo riesca a connettersi alla rete; nella maggior parte dei casi il CAN Bus è interno all'autovettura, per cui non essendo facilmente raggiungibile l'interconnessione di nuove stazioni risulta particolarmente complicata.
Tuttavia, la diffusione di nuovi tipi di connessione come, ad esempio, il *Vehicle to Cloud* (V2C), in cui la macchina è collegata con un sistema cloud da cui ottiene informazioni esterne, potrebbero rappresenta una strada alternativa per iniettare delle trame malevoli senza avere fisicamente accesso al CAN Bus.

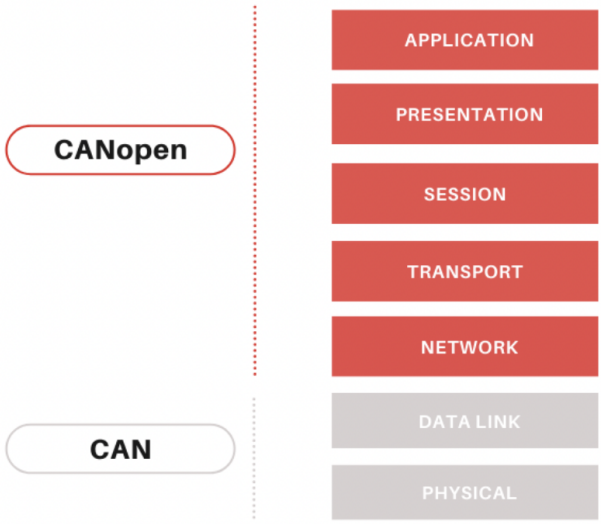

Figura 4.1: CAN Open nel Modello ISO-OSI

Il livello fisico ed il livello data link sono definiti dal CAN, mentre l'Application-layer è definibile liberamente, sebbene negli anni si sia imposto come standard il protocollo CANOpen.
CANOpen offre supporto per la gestione e la configurazione di rete, nello specifico è in grado di modificare il comportamento della rete in base al contesto di utilizzo. Ovvero, detto in altri termini, con gli anni si sono affermati dei nuovi protocolli a livello applicazione capaci di interagire in remoto con le reti CAN. Questi strumenti potrebbero rappresentare un'opzione aggiuntiva per accedere alla rete ed effettuare degli attacchi informatici.
CAN fu concepito per essere utilizzato in contesti applicativi in cui la sicurezza informatica non rappresentava un requisito importante, proprio per questo il protocollo è caratterizzato da diverse vulnerabilità:

- Assenza di un campo indirizzo sorgente:

Alle ECU non è associato un indirizzo, questo porta ad una impossibilità di autenticazione della stazione trasmettitrice, il ricevitore non ha modo di verificare se un certo messaggio proviene dalla stazione attesa. Ciò porta ad una maggiore semplicità da parte di un host malevolo ad effettuare un attacco.

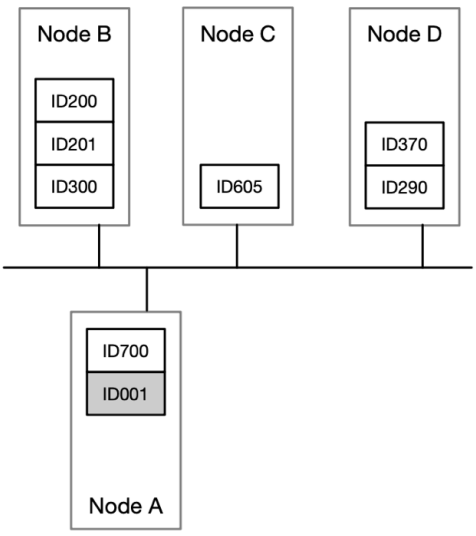

Figura 4.2: Trasmissione contemporanea

L'assenza di un indirizzo sorgente impossibilita quindi l'identificazione del mittente, per cui non è possibile garantire il principio di *fairness*, ovvero non è possibile ripartire in maniera equa le risorse di rete tra tutte le ECU, ma saranno avvantaggiate le stazioni trasmettitrici di identificativi a priorità alta.

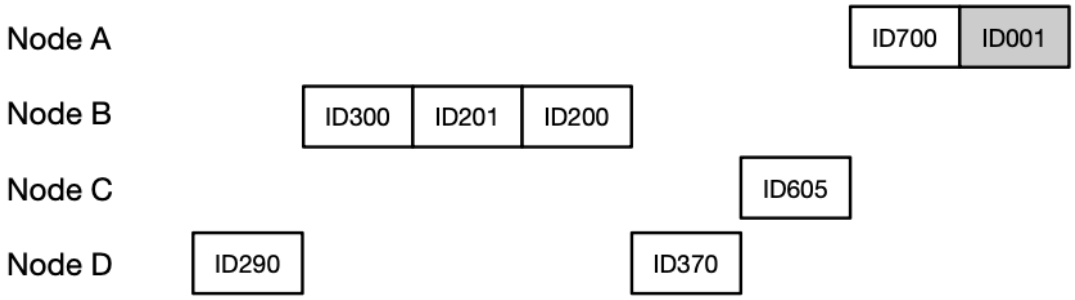

Figura 4.3: Ordine di invio delle trame

Questo potrebbe portare alcuni frame ad essere tramessi dopo un ritardo sostanziale, impedendo al ricevitore di elaborare il frame nei tempi previsti.

- Impossibilità di instradamento unicast:

  Per sua natura il CAN Bus permette unicamente un instradamento di tipo broadcast che porta a due principali problemi: in primo luogo un host malevolo è in grado di monitorare tutto il traffico di rete, ottenendo così ad esempio gli identificativi utilizzati dalle altre ECU, la periodicità di ogni messaggio ed altre informazioni che facilitano un potenziale attacco. Pur non avendo accesso al DBC delle singole ECU è possibile risalire alla codificazione di un particolare identificativo in base alla sua priorità e periodicità (il cambio della stazione radio è un evento a bassa priorità e con bassa periodicità, al contrario il numero di giri al minuto del motore è un evento ad alta priorità e periodicità).
  In secondo luogo, questo tipo di instradamento può portare con maggiore facilità ad una congestione di rete che rende inutilizzabile il bus a tutte le stazioni per un certo tempo.
  Risulta quindi più semplice effettuare un attacco DoS (*Denial of Service*) che porta ad una completa saturazione delle risorse della rete.

CAN Attacks:

4.2 Bus Flood Attack:

Lo scopo di questo attacco è quello di trasmettere delle trame il più velocemente possibile per esaurire la maggior parte della banda del bus, portando così le altre ECU a ricevere le trame attese dopo un ritardo sostanziale creando un malfunzionamento complessivo della rete.
La trasmissione di messaggi aventi un identificativo basso e consequenzialmente priorità elevata rende questo attacco più performante, qualora dovesse esserci un conflitto in cui più stazioni cercano di trasmettere una trama contemporaneamente vincerebbe l'arbitraggio la stazione trasmettitrice dell'identificativo più basso. La trasmissione di un frame con ID = 0 bloccherebbe tutte le altre stazioni, essendo l'identificativo con priorità massima, tuttavia potrebbe essere presente un security gateway che consentirebbe la trasmissione solo di determinati identificativi.
Per ovviare a questo problema è necessario modificare l'attacco originale; la stazione attaccante deve rilevare un identificativo valido, per cui dopo un periodo di ascolto iniziale, l'host memorizza il frame con l'ID avente priorità massima e successivamente invierà in sequenza trame con il suddetto identificativo.

4.3 Adaptive Spoofing:

Con il termine *Spoofing* si fa riferimento ad un particolare tipo di attacco in cui un ricevitore elabora un messaggio proveniente da una stazione attaccante, pensando che sia originato da una stazione legittima. Il limite principale dello spoofing è che il ricevitore processa entrambi i frame, dunque potrebbe non riscontrare errori, ma tale svantaggio può essere superato dall'adaptive spoofing.
Solitamente, quando una ECU riceve un frame, quest'ultimo non viene processato direttamente, ma viene memorizzato internamente in un buffer; dopo un certo tempo il buffer viene svuotato e tutte le trame contenute vengono inviate all'livello utente. In questo modo si ammortizza il costo dell'interrupt, evitando di eseguire la routine associata per ogni singolo frame.
Questa ottimizzazione porta ad una debolezza intrinseca, la quale viene sfruttata dall'attacco di spoofing adattivo; quando una stazione riceve una trama già contenuta nel buffer, quest'ultima andrà a sovrascrivere la precedente.

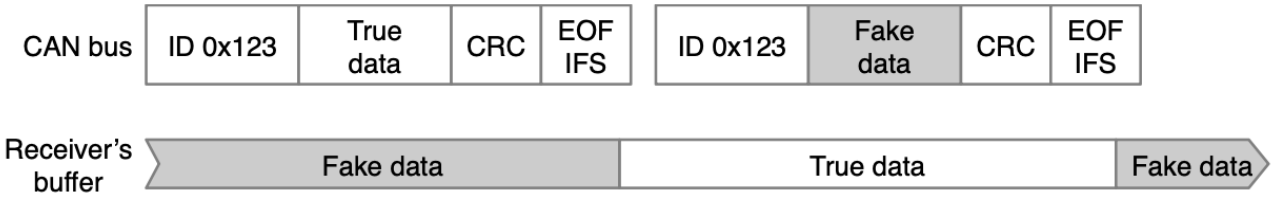

Figura 4.4: Adaptive Spoofing Attack

Supponiamo che un nodo della rete invii una trama con id 0x123 ed un certo payload, successivamente l'host malevolo invierà anch'esso una trama con id 0x123 ma con payload contraffatto, se il frame dovesse arrivare prima dello svuotamento del buffer del ricevitore, quest'ultimo sovrascriverà il frame originale. Pertanto, la stazione ricevente elaborerà unicamente il messaggio contraffatto; lo spoofing adattivo risulta più performante del normale spoofing, tuttavia il

messaggio legittimo rimane nel buffer per un piccolo lasso di tempo vincolando l'host malevolo a reagire in tempi estremamente brevi, nel dettaglio per reti CAN con frequenza di trasmissione di 500 Kbit il tempo di risposta non deve essere maggiore di 8µs.

4.4 Bus Off Attack:

L'attacco di tipo Bus off sfrutta una debolezza intrinseca della gestione degli errori del protocollo CAN, ricapitoliamo in sintesi quanto visto in precedenza.

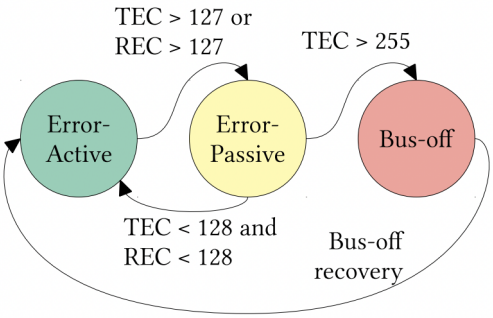

Figura 4.5: Macchina a stati della gestione degli errori per un singolo nodo

Nel momento in cui un nodo rileva un errore autonomamente oppure riceve un Error Frame da un'altra stazione i rispettivi contatori di TEC e REC del nodo in questione vengono incrementati (il TEC di 8 mentre il REC di 1), il TEC nel caso in cui il nodo stia trasmettendo, il REC nel caso in cui sia in ascolto sul canale. Se uno dei due contatori dovesse superare il valore di 255 la stazione entrerebbe in stato di Bus off non potendo più inviare o riceve informazioni, il più delle volte è necessario l'intervento di un operatore per ripristinare una ECU in questo stato. Il Bus Off Attack designa una specifica stazione da attaccare, quando essa trasmette si andrà a sovrascrivere un bit recessivo del payload in modo tale da far rilevare alla stazione attaccata una incoerenza tra ciò che ha scritto sul bus e ciò che ha letto, innescando una routine di errore che farà incrementare il contatore TEC di 8.

Questa operazione verrà fatta ogni volta che la ECU in questione trasmette, tuttavia è importante evidenziare due aspetti:

- Potrebbe capitare che un certo identificativo sia utilizzato da ECU differenti, non essendo presente un indirizzo sorgente non c'è modo di differenziare la stazione attaccata dalle altre.

- Ogni volta che la ECU attaccata riceve correttamente un frame, entrambi i contatori di TEC e REC vengono decrementati di 1, se la stazione in media dovesse inviare un frame dopo otto o più frames ricevuti correttamente non sarebbe possibile farla andare in Bus Off. Ciò comporta la necessità di effettuare questo tipo di attacco bersagliando stazioni con elevata periodicità di trasmissione.

4.5 Simple Frame Spoofing:

Il protocollo CAN impedisce alle stazioni di trasmettere durante la stessa fase di arbitraggio trame aventi lo stesso ID, in quanto le ECU trasmettitrici non si renderebbero conto della trasmissione contemporanea. Entrambe le stazioni presupponendo quindi di aver vinto la fase di arbitraggio, trasmetteranno il campo dati fintanto che una delle due ECU non rileverà una incoerenza tra il bit trasmesso sul bus e quello ricevuto durante l'invio del payload, ciò innescherà una routine di errore: verrà inviato un error frame che porterà tutte le stazioni a risincronizzarsi ed eventualmente a ritrasmettere.
Tuttavia, se le due stazioni trasmettitrici hanno vinto la fase di arbitraggio in precedenza è molto probabile che la rivincano durante la seconda trasmissione, per cui i due nodi trasmetteranno nuovamente in contemporanea innescando un comportamento ciclico chiamato *Arbitration Doom Loop.*

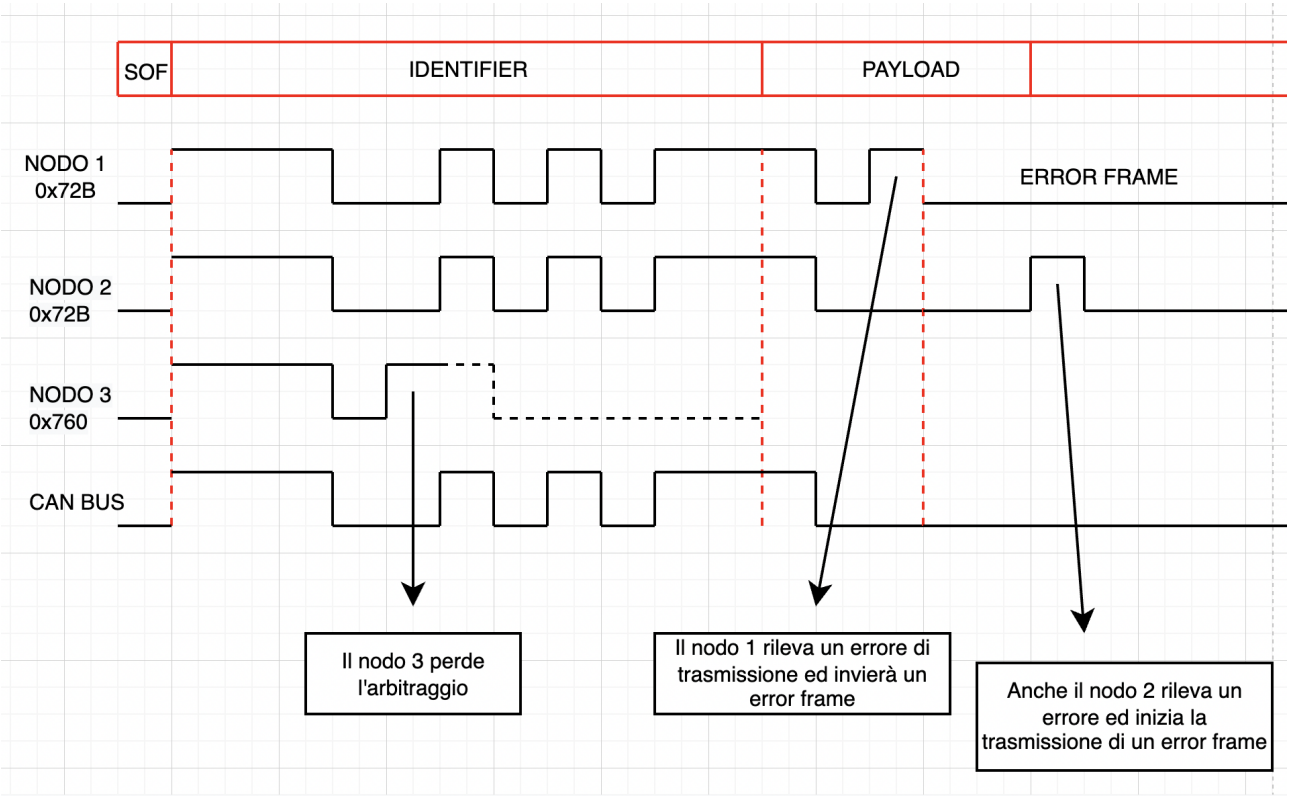

Figura 4.6: Arbitration Doom Loop

Nell'esempio in figura abbiamo tre nodi, i quali cercano di utilizzare il bus in contemporanea per inviare una trama sulla rete, il nodo 1 ed il nodo 2 trasmettono una trama con lo stesso identificativo, ovvero 0x72B mentre il nodo 3 trasmette un frame con identificativo 0x760. Il nodo 3 rileva una collisione durante la trasmissione del 5° bit, si rende conto di aver perso l'arbitraggio e si mette in ascolto a differenza del nodo 1 e del nodo 2, i quali non rilevando nessuna differenza tra i bit scritti e quelli letti presuppongono di aver vinto l'arbitraggio e di poter quindi continuare con l'invio della trama, finché il nodo 1 rileva un errore di trasmissione ed interrompe l'invio del payload per inviare un Error Frame che aumenterà il TEC del nodo 1 e 2 di otto unità. Il nodo 1,2 e 3 proveranno a ritrasmettere la trama creando l'Arbitration Doom Loop. Il loop porta a due principali conseguenze:

- il contatore del TEC della ECU coinvolte (nell'esempio il nodo 1 e 2) aumenterà ad ogni trasmissione portando ad un certo punto le due stazioni in stato di bus off.

- il bus sarà inutilizzabile dalle altre stazioni finché quelle coinvolte nel loop non andranno in Bus Off.

Calcoliamo il tempo medio di attesa nel caso in cui avvenga l'Arbitration Doom Loop facendo le seguenti assunzioni:

1. La rete ha una frequenza di trasmissione di 128 Kbit.
2. Le trame sono di lunghezza massima, ovvero di 128 bit (standard CAN2.0 B)
3. Si considera la minima spaziatura tra frame di 3 bit.
4. L'Error frame è composto da 6 bit dominanti e da un Error Delimiter di 7 bit recessivi per un totale di 13 bit.
5. Supponiamo per semplicità che il conflitto venga rilevato al primo bit del payload.
6. I contatori di TEC sono settati entrambi a 0 inizialmente.

$$Numero\ di\ trasmissioni:\ {}^{255}/_{8} = 31.875\ \approx\ 32\ trasmissioni$$

$$Numero\ di\ bit\ per\ singola\ trasmissione:\ 128 + 3 + 13 + 1 = 145\ bit$$

$$Numero\ di\ bit\ totali: 32 * 145 = 4640\ bit$$

$$Tempo\ totale:\ \frac{4640\ bit}{125\ Kbit/sec} =\ 0.03712\,\text{sec} = 37.12\ ms$$

Si evince quindi che le stazioni coinvolte andranno in Bus Off dopo circa 37 millisecondi, un tempo non trascurabile. L'idea del Simple Frame Spoofing è quella di applicare l'Arbitration Doom Loop, l'host malevolo trasmetterà di proposito un frame con stesso ID della stazione che ha vinto l'arbitraggio.

4.6 Double Receive Attack:

Uno dei principali pregi del protocollo CAN è l'alta affidabilità che garantisce. Recenti studi hanno stimato che su una rete con bit rate di 500 Kbit/sec, supponendo un utilizzo di otto ore al giorno per ogni giorno dell'anno si riscontrerà un errore di trasmissione di un bit ogni 0.7 secondi, un errore non rilevato capita una volta ogni mille anni; da ciò si deduce chiaramente che la probabilità che avvenga un errore di trasmissione è estremamente bassa, risulta ancora più improbabile se consideriamo la possibilità che un errore di trasmissione avvenga in un campo specifico della trama. Il formato della trama CAN prevede un campo denominato EOF (End of Frame), una stringa di 7 bit recessivi che delimita la fine della trama, la probabilità che avvenga un errore di trasmissione durante l'invio dell'ultimo bit del EOF è pari a quella dell'errore di trasmissione di un bit generico moltiplicata per il numero di bit della trama, calcoliamo un valore plausibile:

$$Supponiamo\ che: \quad Freq_{trasmissione} = 500\frac{Kbit}{sec} \quad , \quad lunghezza\ trama = 128\ bit$$

$$Freq_{bit_{error}} = \frac{1}{0.7\ sec} = 1.4286\ Hz$$

$$Numero\ di\ bit\ trasmessi\ senza\ errori: \ Num_{bits} = \frac{500\ Kbit/sec}{1.4286\ Hz} \approx 350\ Kbit = 350.000\ bit$$

$$bit\ error\ prob = \frac{1}{350000} = 0.000002857$$

$$EOF_0\ error\ prob = \frac{1}{350000 * 128} = 2{,}23 * 10^{-8}$$

Per cui è molto improbabile che l'ultimo bit del campo EOF possa subire un errore di trasmissione, tuttavia, un attaccante potrebbe volutamente sovrascrivere il suddetto bit (essendo un bit recessivo e quindi sovrascrivibile) per alterare in modo radicale il funzionamento della rete, creandole un forte disservizio, analizziamo le motivazioni con l'ausilio di un grafico.

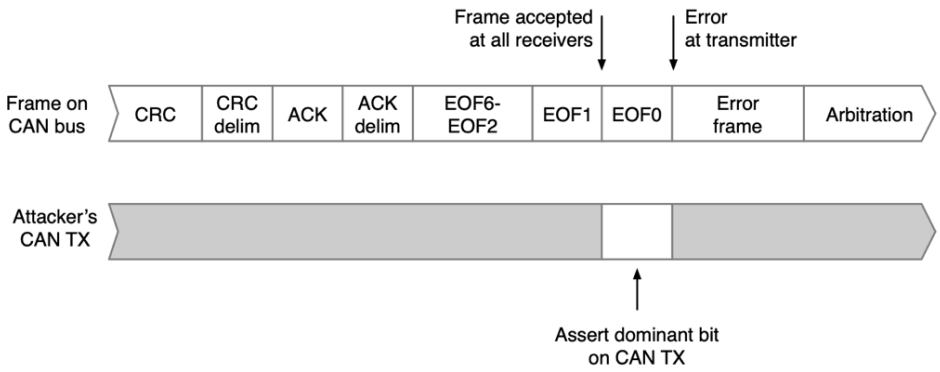

Figura 4.7: Double Receive Attack

Le conseguenze che derivano da questo tipo di attacco sono molto simili a quelle del Simple Frame Spoofing, nello specifico quando la stazione trasmettitrice riscontra una incoerenza tra il bit scritto sul bus e quello letto dallo stesso, innescherà una routine di errore, verrà inviato un Error Frame che incrementerà il suo contatore TEC di otto unità e dovrà ritrasmettere la trama in quanto l'errore è stato riscontrato nell'ultimo bit della trama, è probabile che la stazione coinvolta rivinca la fase di arbitraggio portando ad un comportamento ciclico; l'host malevolo sovrascriverà l'ultimo bit del EOF ogni volta causando nella stazione attaccata in stato di Bus Off.

Toll, Packages e librerie

5.1 Introduzione

In questo capitolo analizzeremo e motiveremo le scelte implementative dei componenti software alla base del progetto, in particolare ci soffermeremo sul linguaggio di programmazione utilizzato, sulle librerie esterne, l'ambiente di esecuzione, sui packages ed i toll di supporto.

5.2 SocketCAN

SocketCAN è un command line tool, formato da un insieme di driver di rete CAN, compatibile con il kernel linux, rispetto ai driver CAN tradizionali è basato sulle Berkeley socket API, ovvero le librerie che lo compongono fanno riferimento allo stack di rete di Linux. Per comunicare su una rete CAN è necessario in primo luogo inizializzare un socket, successivamente bisogna associare il socket all'interfaccia di rete, a seguire il socket può essere utilizzato per interagire con la rete come se fosse un UDP Socket tramite le system call *read* e *write.* SocketCAN consente agli applicativi posti a livello utente di interfacciarsi sulla rete CAN, riuscendo così sia a leggere che a scrivere dati sul CAN Bus.

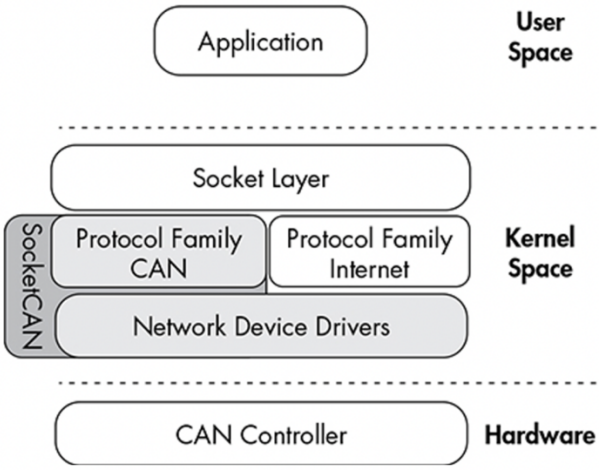

Figura 5.1: CAN Communication Layer

L'applicazione, situata al livello utente, attraverso l'utilizzo di opportune system call interagisce con il socket che si trova a livello kernel il quale realizza il collegamento con la scheda di rete CAN del dispositivo, in particolare con il CAN Controller della PiCAN FD.

SocketCAN è un tool utilizzabile sui sistemi Linux da linea di comando, è possibile installare tutti i pacchetti necessari tramite i comandi:

```
user:~$ sudo apt update

user:~$ sudo apt install can-utils
```

SocketCAN permette anche la definizione e l'utilizzo di interfacce virtuali per motivi di testing, riportiamo i comandi necessari per definire una interfaccia di rete CAN sia virtuale che fisica. Per definire un'interfaccia virtuale e poi attivarla:

```
user:~$ sudo ip link add dev vcan0 type vcan
user:~$ sudo ip link set up vcan0
user:~$ ip link show vcan0
3: vcan0: <NOARP> mtu 72 qdisc noqueue state DOWN mode DEFAULT group default qlen 1000
    link/can
```

Per definire un'interfaccia fisica invece è necessario specificare il bitrate

```
user:~$ sudo ip link add dev can0 type can bitrate 125000
user:~$ sudo ip link set up can0
user:~$ ip link show can0
3: vcan0: <NOARP> mtu 72 qdisc noqueue state DOWN mode DEFAULT group default qlen 1000
    link/can
```

È possibile verificare che l'interfaccia sia stata correttamente attivata attravero il comando ip link, il cui output dovrebbe essere di questo tipo:

```
user:~$ ip link

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
link/ether 00:1c:42:fa:ed:d6 brd ff:ff:ff:ff:ff:ff

3: vcan0: <NOARP,UP,LOWER_UP> mtu 72 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000 link/can

4: vcan1: <NOARP> mtu 72 qdisc noop state DOWN mode DEFAULT group default qlen 1000
link/can
```

Analizziamo nel seguito le principali funzioni di SocketCAN, soffermandoci nello specifico su quelle utilizzate dal programma. È importante sottolineare che socketCAN operando a livello applicazione non ha direttamente accesso al livello fisico, per cui nessuna API permette di scrivere o leggere sul bus un singolo bit, alla base di ogni interazione vi è un trasferimento di trame.

5.3 cansend

Come intuibile dal nome, il comando cansend invia sulla rete un frame. È necessario specificare l'interfaccia dalla quale mandare la trama e l'id ed il payload del frame separati da un cancelletto.

```
user:~$ cansend can0 123#DE
```

In questo caso l'interfaccia can0 invierà una trama con id 123 e payload DE, è necessario rispettare la lunghezza dei due campi per consentire l'invio. Ogni carattere viene considerato come esadecimale, per cui il campo l'ID, essendo formato da 11 bit, deve essere codificato da tre cifre esadecimali che corrispondono ad un nibble ognuna (ovvero 4 bit). Il payload invece deve essere formato da un numero intero di byte per cui è necessario inserire un numero pari di cifre esadecimali, questo perché due nibble formano un byte. Riportiamo due casistiche di errore:

```
user:~$ cansend can0 F203#DE

Wrong CAN-frame format
```

In questo caso l'id è troppo lungo, ovvero di 16 bit.

```
user:~$ cansend can0 F23#DEF67

Wrong CAN-frame format
```

In questo caso invece il payload è formato da 5 nibble, ovvero da 2.5 byte.

5.4 cangen

Rispetto al semplice cansend, il comando cangen prende il nome di *frame generator* ovvero permette la generazione ed il successivo invio di un numero indefinito di frame, in cui i campi di id e payload possono variare casualmente, essere statici oppure variare in base ad uno specifico criterio scelto dall'utente tramite l'utilizzo di alcuni flag. Questo comando non termina mai autonomamente la propria esecuzione (tranne nel caso in cui si setti un flag specifico), la generazione dei frames continuerà finché il processo associato non riceverà un segnale di SIGKILL o SIGTERM. Tra i flag più importanti abbiamo:

- -g: gap in millisecondi, di default il valore è di 200 ms
- -f: per generare frame CAN FD, di default vengono generati frame CAN 2.0B
- -v: i frame inviati vengono stampati su console
- -b: per generare frame CAN FD con cambio di bitrate
- -g: per generare frame in formato esteso
- -R: invia un RTR (Remote Trasmission Request) frame
- -m: un insieme dei flag -f -b -R -e
- -x: per disabilitare l'interfaccia di loopback
- -i: per ignorare il valore di ritorno della system call write() nel caso in cui sia l'Errno di ENOBUFS, che si manifesta nel caso in cui non ci sia abbastanza spazio nel buffer per l'operazione richiesta
- -n <count>: il processo termina la sua esecuzione dopo la generazione di count frames.
- -p <timeout>: è possibile specificare dopo quanto tempo il buffer si deve svuotare per inviare i frames sul bus
- -L <modalità>: Permette di specificare il numero di byte che devono comporre il campo payload.
- -D <modalità>: flag per la generazione del payload, permette di specificare la modalità di generazione, nello specifico è possibile assegnare un payload fisso per ogni frame oppure generarlo casualmente o in modo incrementale.

- -I <modalità>: flag per la generazione dell'ID, le modalità di generazione sono le stesse del payload.

Vediamo alcuni esempi applicativi:

```
user:~$ cangen vcan0 -g 100 -v -I 550 -I i -n 4
vcan0  550#C3.F4.7C.2B.9E.CE.E3.5B
vcan0  551#3E.D7.76.22.A3.2E.6F.19
vcan0  552#B4.0D.A1
vcan0  553#9E.F5.F9.33.64.33.1E.29
```

Con un gap di 100 ms tra i frames, vengono generati quattro frames partendo dall'id 550 in modo incrementale, il payload non essendo specificato il flag di default viene generato in maniera casuale.

```
user:~$ cangen vcan0 -e -v -L 3 -D r -I 30912 -n 4 -f
  vcan0  00030912##0.0B.1B.95
  vcan0  00030912##0.63.A5.48
  vcan0  00030912##0.E9.94.EF
  vcan0  00030912##0.6A.57.57
```

Vengono generati quattro frames relativi allo standard CAN FD, con payload casuale di 3 bytes, ID fisso 30912 visibili sulla console in formato esteso. È possibile lanciare un semplice cangen senza argomenti:

```
user:~$ cangen vcan0
```

In questo caso non sarebbe visibile nessun output vista l'assenza del flag v, il processo associato genererà un numero indefinito di frames relativi allo standard CAN 2.0 B con Payload di lunghezza variabile casuale ed ID di lunghezza fissa di 11 bit e contenuto casuale. È necessario terminare manualmente il processo inviando un comando di SIGTERM.

5.5 candump

In maniera duale al cangen, il comando di candump permette di mettersi in ascolto sulla rete, non appena l'interfaccia riceve un frame, esso verrà stampata su console. Anche in questo caso il processo terminerà la propria esecuzione a seguito della ricezione di un segnale di SIGTERM. Tra i flag più importanti abbiamo:

- -t <tipo>: permette di visualizzare il timestamp associato alla ricezione di ogni frame, è possibile definire tre tipi differenti: absolute, delta e zero.
- -H: verranno visualizzati i timestamp reali dell'hardware anziché quelli di sistema.
- -b <can>: modalità bridge, manda i frame ricevuti ad un'altra interfaccia CAN.
- -i: per visualizzare l'output binario.
- -n <count>: il processo associato al candump terminerà dopo la ricezione di count numero frames.
- -d: flag che permette di vedere i frames scartati.
- -e: permette la visualizzazione dei frame di errore in formato human-readable.
- -r <size>: flag per fissare la dimensione del buffer in ricezione alla dimensione size.
- -T <msecs>: il processo termina dopo che sono passati msecs millisecondi dalla ricezione dell'ultimo frame.

- -l: questo flag ridireziona la stampa dei frames ricevuti da stdout a file, verrà generato un file con estensione .log con nome *candump-timestamp.log* dove il timestamp ha il formato: *aaaa-mm-millisecondi*.
- -L: permette di vedere il formato del file di log anche su stdout.

Vediamo alcuni esempi applicativi:

```
user:~$ candump vcan0
  vcan0  230   [7]  3C A3 AD 67 6F EC 26
  vcan0  33A   [4]  12 EC EA 7A
  vcan0  28A   [8]  1E B8 C1 37 52 0D 80 03
  vcan0  07C   [8]  18 C5 3A 0F D4 6F 2B 7A
```

Notiamo che nella stampa di ogni trama viene riportata l'interfaccia da cui è stata ricevuta, tra parentesi quadre il numero di byte the compongono il payload ed infine il campo dati diviso per byte in formato esadecimale

```
user:~$ candump vcan0 -t a -T 800 -i
 (1643791096.808280)  vcan0  668   [2]  00000011 10110101
 (1643791097.308421)  vcan0  59A   [2]  10010010 00001100
 (1643791097.813585)  vcan0  59F   [2]  10111001 10010101
```

In questo caso è stato specificato il flag di timestamp in formato assoluto, che è il primo termine che appare nell'output per ogni riga, da cui possiamo ricavare una periodicità di trasmissione di circa 500 ms. A seguire il flag i induce la visualizzazione in formato binario del payload mentre il flag T porta il processo a termine dopo 800 ms dalla ricezione dell'ultimo frame.

```
user:~$ candump vcan0 -l -n 5
Disabled standard output while logging.

Enabling Logfile 'candump-2022-02-02_094451.log'
```

In questo è stato specificato il flag l, il quale andrà a ridirezionare l'output da console su file. Per cui verrà creato un file con nome "candump-2022-02-02_094451.log" in cui sono presenti i primi cinque frames ricevuti. Il file in questione ha il suddetto contenuto:

```
user:~$ cat candump-2022-02-02_094533.log
(1643791533.417260) vcan0 406#A805
(1643791533.919225) vcan0 5A9#59B6
(1643791534.420602) vcan0 19B#DDE1
(1643791534.925270) vcan0 044#CA9E
(1643791535.429420) vcan0 73E#DAA2
```

5.6 cangw

Il comando cangw, come intuibile dal nome, permette di eseguire un gateway tra due interfacce di rette, ovvero permette l'inoltro del traffico proveniente da una sorgente ad una certa destinazione.
Tramite la stesura di specifiche regole è possibile fare in modo che: tutto il traffico generato da una specifica stazione sia inviato automaticamente ad un'altra interfaccia di rete (che può essere sia virtuale che fisica). In maniera simile a quanto avviene per Ethernet, possiamo pensare che il servizio offerto da questo comando sia quello di inserire manualmente una route.
Tra i flag più importanti citiamo i seguenti:

- -A: Per aggiungere una nuova regola.
- -D: Per eliminare una regola preesistente.
- -F: Per eliminare tutte le regole precedentemente dichiarate.
- -L: Per stampare su console tutte le regole presenti.

Nel momento in cui si aggiunge una nuova regola, è necessario specificare questi flag aggiuntivi:

- -s <source_device>: In cui si indica il nome dell'interfaccia di rete sorgente.
- -d <destination_device>: In cui si indica il nome dell'interfaccia di rete destinazione.

Tra i flag opzionali abbiamo i seguenti:

- -l <num_hops>: Limita il numero di hops ad un valore specifico.
- -f <filtro>: Permette di applicare un filtro.
- -e: Flag di echo, da utilizzare con le interfacce virtuali.
- -m <modalità>: Forza una specifica modalità sul formato di stampa dei frame.
- -t: Non aggiorna il campo timestamp, per cui il destinatario elaborerà i frames con il timestamp originali del mittente.

Creiamo un gateway tra un'interfaccia fisica ed una virtuale in modo tale che tutto il traffico inviato da una sia ricevuto dall'altra. In questo caso avremo bisogno di lanciare due cangw in cui invertiamo sorgente e destinatario.

```
user:~$ cangw -A -s can0 -d vcan0 -e
user:~$ cangw -A -s vcan0 -d can0 -e
```

Verifichiamo che le regole siano state aggiunte correttamente:

```
user:~$ cangw -L

cangw -A -s vcan2 -d vcan0 -e # 0 handled 0 dropped 0 deleted
cangw -A -s vcan0 -d vcan2 -e # 0 handled 0 dropped 0 deleted
```

Lanciamo un candump su una delle due interfacce ed un cangen sull'altra:

user:~$ candump can0 vcan0 3A5 [8] 53 C2 98 64 11 EA C7 64 vcan0 22D [8] 22 85 53 4F 6E E1 DA 42 vcan0 2FF [4] FC C4 D5 10 vcan0 399 [8] 06 87 48 76 09 4E 23 13	**user:~$** cangen vcan0 -n 4

5.7 canplayer

Il comando canplayer è strettamente legato con i file di log generati dal candump con il flag l; questo comando prende come argomento un file di log ed invia tutti i frame contenuti nel file di log con gli stessi timestamp, è l'unico comando in cui non si deve specificare l'interfaccia di rete essendo già presente nel file di log. Tra i flag più importanti abbiamo:

- -I <infile>: per indicare il file in input.
- -t: per ignorare i timestamp e mandare i frames istantaneamente.
- -g <ms>: gap in millisecondi che deve passare dopo la terminazione del file.
- -l <num>: indica il numero di volte che deve essere processato il file di input, con 'i' si indica un loop infinito in cui il file di input viene processato finché il processo non riceve un SIGTERM.
- -x: per disabilitare il local loopback dei frames inviati.
- -v: per stampare su console i frames inviati.

Prendiamo in esame il file di log generato dall'esempio precedete e consideriamo questo caso applicativo:

```
user:~$ canplayer -l 2 -I candump-2022-02-02_094533.log -g 2000 -t -v
vcan0 (vcan0)     406   [2]  A8 05
vcan0 (vcan0)     5A9   [2]  59 B6
vcan0 (vcan0)     19B   [2]  DD E1
vcan0 (vcan0)     044   [2]  CA 9E
vcan0 (vcan0)     73E   [2]  DA A2
vcan0 (vcan0)     406   [2]  A8 05
vcan0 (vcan0)     5A9   [2]  59 B6
vcan0 (vcan0)     19B   [2]  DD E1
vcan0 (vcan0)     044   [2]  CA 9E
vcan0 (vcan0)     73E   [2]  DA A2
```

In questo caso il file candump-2022-02-02_094533.log verrà inviato due volte consecutive con uno spaziamento tra le due iterate di 2000 ms, i timestamp dei frames saranno ignorati per cui le trame verranno inviate istantaneamente, grazie al flag v i frames inviati sono anche stampati sul terminale.

5.8 Linguaggio di programmazione

Il linguaggio di programmazione scelto per la realizzazione del progetto è C++, dopo un'attenta analisi da parte dell'azienda è stato ritenuto opportuno effettuare un porting del codice da Python a C++. Il C++ è un linguaggio più complesso rispetto al Python, in quanto ha una sintassi decisamente più difficile da rispettare, è fortemente tipizzato, ovvero il programmatore è tenuto a specificare il tipo di ogni elemento sintattico definito nel programma, inoltre l'assenza del garbage collector comporta la necessità di gestire manualmente la memoria inoltre non è portabile, ovvero su sistemi operativi differenti è necessario ricompilare il codice ed in alcuni casi potrebbe essere necessario modificare il codice sorgente per renderlo eseguibile su una determinata piattaforma. Il vantaggio principale del C++ è che rispetto a Python è un linguaggio compilato anziché interpretato.
Con il termine linguaggio compilato, si fa riferimento ad un linguaggio di programmazione in cui è presente un compilatore, ovvero un traduttore in grado di convertire il codice sorgente del programma

in codice macchina. Nei linguaggi interpretati è presente un interprete che esegue direttamente il codice sorgente a tempo di esecuzione, in generale i linguaggi interpretati risultano quindi più lenti e meno adatti per essere eseguiti su dispositivi embedded ovvero con Hardware poco performante.
Il Raspberry Pi, pur rientrando nella categoria dei dispositivi embedded, risulta essere abbastanza potente per eseguire programmi scritti in Python, garantendo un'interazione fluida tra l'interfaccia grafica del programma e l'utente, offrendo nel complesso una buona User Experience.
Nel contesto applicativo del nostro programma è fondamentale ridurre al minimo i tempi di reazione dell'applicazione, questo perché alla base dell'interazioni con le altre stazioni della rete vi è la necessità di inviare dei frames alla stazione ricevente rispettando dei timing molto stringenti, o innescare una certa azione non appena si riceve un determinato frame. È fondamentale massimizzare la reattività del programma per consentire il testing delle ECU o per rendere più efficienti gli attacchi informatici, per questo motivo l'utilizzo di un linguaggio interpretato risulta poco adatto per soddisfare i requisiti del progetto. Per rendere più comprensibile le differenze tra i due linguaggi a livello di tempi di esecuzione, complessità implementative e modalità di esecuzione riportiamo il seguente esempio:
Consideriamo la realizzazione di un programma per il calcolo dei numeri primi, il programma deve stampare su console quanti numeri primi sono presenti nei primi 250000 numeri partendo da 2.

```
import time

def isPrime(n):
    for i in range(2,n//2+1):
        if (not (n%i)):
            return 0;
    return 1;

numPrimes = 0;
start = time.time()
for i in range(2,250001):
    numPrimes = numPrimes + isPrime(i);

end = time.time()
print("numPrimes: " + str(numPrimes) + " Time: " + str(end - start))
```

```
IDLE Shell 3.9.7
Python 3.9.7 (v3.9.7:1016ef3790, Aug 30 2021, 16:25:35)
[Clang 12.0.5 (clang-1205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==================== RESTART: /Users/luca/Desktop/Test.py ====================
numPrimes: 22044 Time: 68.60974383354187
>>>
Ln: 7  Col: 4
```

Figura 5.2: Programma scritto in Python

Importiamo la libreria time per calcolare il tempo impiegato dal programma per eseguire il calcolo, come visibile in figura il programma impiegherà 68.61 secondi per rilevare i 22044 numeri premi.
Non è necessario compilare il codice, una volta scritto e salvato è direttamente runnabile.

```cpp
#include <iostream>
#include <chrono>
bool isPrime(int n){
    for ( int i = 2 ; i <= n/2 ; i++ )
        if ( !(n%i) )
            return 0;
    return 1;
}
int main() {
    int numPrimes = 0;
    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
    for ( int i = 2 ; i < 250001 ; i++ )
        numPrimes += isPrime(i);

    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "numPrimes: " << numPrimes << " Time: "
                << std::chrono::duration_cast<std::chrono::milliseconds>(end - begin).count() << " [ms]" << std::endl;
    return 0;
}
```

```
Scrivania — -zsh — 87×14
luca@MacBook-Pro-di-Luca Desktop % time g++ test.cpp
g++ test.cpp  0,18s user 0,03s system 95% cpu 0,224 total
luca@MacBook-Pro-di-Luca Desktop % ./a.out
numPrimes: 22044 Time: 2778 [ms]
luca@MacBook-Pro-di-Luca Desktop %
```

Figura 5.3: Programma scritto in C++

Mettendo a confronto i due codici, risulta evidente di come la sintassi del Python a differenza di quella del C++ risulti più leggibile e meno verbosa. In questo caso dobbiamo prima compilare il programma lanciando da terminale "time g++ test.cpp" dove il comando time ci restituisce il tempo impiegato per generare l'eseguibile, ovvero 0.224 secondi. Successivamente possiamo lanciare l'eseguibile che, come prevedibile, rileverà 22044 numeri primi come in precedenza, impiegando 2778 millisecondi ovvero 2.78 secondi per l'esecuzione del programma a cui vanno sommati i 0.224 secondi per la compilazione per un totale di circa 3 secondi complessivi.
3 secondi nel caso del C++ contro i 68.61 secondi del Python, risulta evidente quanta differenza ci sia a livello di prestazioni tra i due linguaggi.

5.9 Fase di compilazione

All'aumentare delle dimensioni del programma, l'aggiunta di flag, librerie esterne e file differenti comporterebbe una maggiore difficoltà in fase di compilazione al programmatore. Ad ogni modifica è necessario ricompilare, ciò comporta la necessità di riscrivere una lunga sequenza di istruzioni che allunga in modo significativo i tempi di sviluppo.

Per automatizzare e velocizzare la fase di compilazione è stato utilizzato il make, ovvero un tool disponibile sui sistemi operativi UNIX, che semplifica notevolmente la compilazione di programmi scritti in C/C++.
Il makefile è un file testuale, contenente tutte le direttive ed istruzioni relative alla modalità di compilazione di un programma. È necessario lanciare il comando di make da terminale nella directory contenente il makefile per ottenere l'eseguibile risultante.
Analizziamo il makefile del progetto:

```
PROG = App
COMPILER = g++
STD = c++17
FILES.CPP = GUI.cpp Test.cpp Suite.cpp Blackbox.cpp can_utilities.cpp
Parsing_log.cpp Parsing_ini.cpp utilities.cpp
CFLAGS = `pkg-config --cflags gtk+-3.0`
LIBS = `pkg-config --libs gtk+-3.0 libnotify`
WARNING_FLAGS = -w
END_FLAGS = -lstdc++fs

${PROG}: ${PROG}.cpp

	${COMPILER} -std=${STD} ${WARNING_FLAGS} ${CFLAGS} ${PROG}.cpp
${FILES.CPP} -g -o ${PROG} ${LIBS} ${END_FLAGS}

clean:
	rm ${PROG}
```

Il programma è conforme allo standard di C++ 17, include i flag e le librerie esterne GTK per l'interfaccia grafica e libnotify per la gestione di un sistema di notifiche, l'ultimo flag serve per consentire l'utilizzo della libreria filesystem.
All'aumentare della lunghezza del programma aumenterà consequenzialmente il tempo di compilazione, i dispositivi embedded sono generalmente molto limitati ed in alcuni casi non possono effettuare la normale compilazione, oppure richiederebbero troppo tempo per la generazione dell'eseguibile. I microcontrollori, ad esempio, non possiedono nessun sistema operativo per tanto non è possibile eseguire nessun compilatore o linker su questi dispostivi. Per permettere l'esecuzione di programmi compilati si ricorre alla tecnica del cross-compiling; il codice sorgente viene compilato da un cross-compilatore che genererà un file binario compatibile con una architettura differente rispetto a quella su cui si trova il cross-compilatore, in questo modo dispositivi più potenti possono compilare dei programmi per i dispositivi embedded che devono semplicemente eseguire il file binario. I Raspberry Pi pur essendo dei dispositivi embedded sono risultati abbastanza preformarti per compilare i programmi richiesti in tempi relativamente brevi, basti pensare che il primo modello rilasciato disponeva di un unico core da 700 MHz, in grado di compilare 160 righe di codice sorgente in 1 secondo; il Raspberry Pi 4, ovvero l'ultimo modello, dispone di 4/8 core da 1.5 GHz ed è in grado di compilare circa 2000 righe di codice sorgente al secondo.
Alcune tecniche di scrittura del codice permettono di abbattere notevolmente i tempi di compilazione, nello specifico è sconsigliata la scrittura di programmi monolitici, oltre che ad un evidente carenza di leggibilità e flessibilità bisogna anche considerare l'impossibilità di parallelizzare il lavoro. Ogni file sorgente viene compilato da un core differente, per cui una architettura modulare permette di parallelizzare la fase di compilazione abbattendo notevolmente i tempi richiesti.
In conclusione, avendo adottato i dovuti accorgimenti, il cross compiling non è risultato necessario per la compilazione del programma su Raspberry Pi.

5.10 GTK

GTK+ (GIMP Toolkit) è un toolkit per la creazione di GUI (Graphical User Interface) ovvero di interfacce grafiche, pur essendoci diverse librerie come opzioni per l'implementazione della parte grafica la scelta è ricaduta su GTK+.
Nel 2019 Simon Long, ingegnere della Raspberry Pi responsabile dello sviluppo di applicazioni Desktop, rilasciò il libro "An introduction to C & GUI Programming" in cui introduce la libreria GTK per la realizzazione di interfacce grafiche orientate su dispositivi Raspberry. La presenza di una documentazione molto dettagliata e specifica ha fatto ricadere la scelta proprio su GTK, pur essendo una libreria scritta in linguaggio C quindi non orientata agli oggetti. Il blocco fondamentale alla base di ogni componente grafico in GTK è il *widget*, ogni elemento visibile sullo schermo che sia un pulsante, una finestra o un menù a tendina è un widget, la libreria GTK gestisce la creazione di ogni widget, il rilascio delle risorse allocate da ogni componente grafico ed eventualmente l'esecuzione di callback associate all'interazione dell'utente con l'interfaccia grafica. Tutto ciò che il programmatore deve fare è specificare quale widget creare, la posizione in cui deve apparire e ciò che il componente deve fare, tutto il resto viene gestito da GTK. GTK è una libreria grafica di alto livello, consente le creazioni di interfacce grafiche complesse scrivendo un numero di righe di codice contenute. È possibile ridurre ulteriormente il codice scritto utilizzando Glade un'applicazione di alto livello che permette la creazione di GUI.

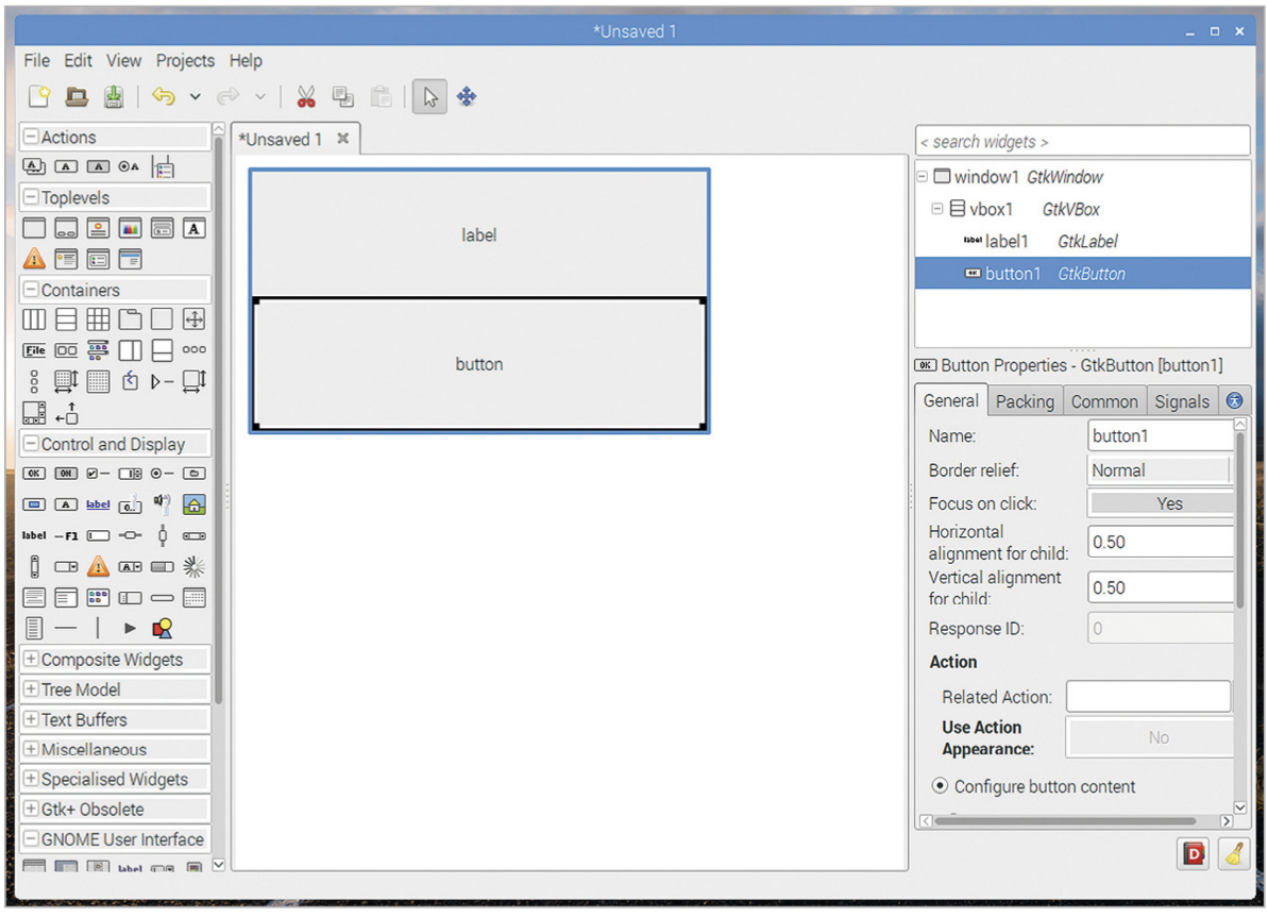

Figura 5.4: Glade

Con Glade non è necessario scrivere codice, basta semplicemente trascinare i widget per comporre la GUI, gli elementi inseriti genereranno un file XML in cui è salvata l'interfaccia creata, essa è leggibile attraverso il GtkBuilder, una particolare struttura di Gtk che effettua il parsing del file XML in modo

indipendente dal linguaggio di programmazione. Glade rappresenta quindi uno strumento di alto livello che semplifica drasticamente la complessità di creazione di una interfaccia grafica, riducendo in modo significativo il codice scritto. Tra le applicazioni più famose che utilizzano GTK citiamo GNOME, GIMP, Mozilla Firefox, Mozilla Thunderbird e Wireshark.

5.11 Libnotify

Libnotify è una libreria esterna, essa offre un sistema di notificazione ovvero permette la generazione di popup che notificano l'utente in modo asincrono, dell'avvenimento di un particolare evento.

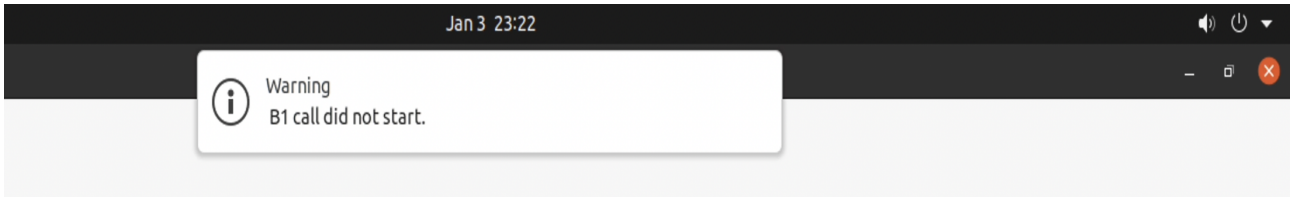

Figura 5.5: Esempio di notifica

Libnotify è compatibile sia con GTK che QT, con sistemi operativi UNIX; il suo utilizzo comporta la necessità di avere un Notification server.

Documentazione Software

6.1 Introduzione

Dopo aver introdotto i principali concetti teorici alla base del protocollo CAN, necessari per comprende il contesto di sviluppo dell'applicazione, ed aver descritto i principali componenti hardware, le librerie ed i tool esterni utilizzati, a seguire verrà analizzata la documentazione software. Sarà esaminato la struttura a livello di architettura del codice, l'implementazione e le funzionalità che il programma assolve. Il codice sorgente segue il paradigma orientato agli oggetti, ogni scelta implementativa rispetta il principio fondamentale della modularità, questo per rendere più agevoli eventuali sviluppi futuri dell'applicazione, abbattere i tempi di compilazione, migliorare la leggibilità del codice e per semplificare il rilevamento di un eventuale errore.

6.2 WhiteBox

Il modulo di WhiteBox offre come funzionalità principale quella di validare un dispositivo di rete CAN, ovvero permette di verificare il corretto funzionamento di una ECU da un punto di vista retistico. È possibile verificare se l'invio di una certa sequenza di frames da parte del tester va ad innescare una specifica azione nel dispositivo testato. L'azione potrebbe essere un evento fisico rilevabile dall'utilizzatore, ad esempio l'accensione di una spia sul quadrante della macchina, l'avvio di una chiamata di emergenza (eCall) oppure l'invio di un certo gruppo di frames con dei timing specifici, rilevabili dal tester. A sua volta il software deve notificare l'utilizzatore, fornendo informazioni riguardo il comportamento del dispositivo da validare.
Possiamo generalizzare quanto detto in precedenza nel seguente modo: il modulo di WhiteBox permette l'esecuzione di uno o più test, eseguibili in parallelo o in sequenza; ogni test appartiene ad una categoria specifica che delinea un comportamento differente. È tuttavia possibile convogliare l'algoritmo di ogni categoria nel seguente: dopo una eventuale trasmissione iniziale di uno o più frames, il tester si metterà in ascolto per rilevare uno specifico frame oppure un gruppo di essi, la ricezione di queste trame deve avvenire entro lo scadere di un time-out, qualora fosse specificato. Se il rilevamento va a buon fine il test sarà considerato come superato, viceversa se il time-out scade il test è considerato come fallito, in entrambi i casi la terminazione del test porta il software ad innescare delle azioni differenti in base all'esito del test. Se il time-out non è attivo, il tester si metterà in ascolto per un periodo di tempo indeterminato fino alla ricezione della trama o gruppo di trame di riferimento.
Alla base del funzionamento di questo modulo, deve essere noto il DBC del componente da testare, ovvero è necessario sapere quali sono gli identificativi che codificano certi eventi e per ognuno di essi è necessario sapere come decifrare il payload, quest'ultimo contiene uno o più segnali che rappresentano informazione da ottenere. Questi segnali sono definiti da: un nome, una lunghezza in bit, un offset, una endianness che specifica l'ordine dei byte (big endian o little endian) da questi parametri è possibile ricavare il valore associato al segnale.
Per semplificare, velocizzare e rendere più elastico il testing di un componente o dell'intero sistema la Zirak ha provveduto alla creazione di un tool di supporto, prendendo in input un file DBC, il tool permette la creazione dei test attraverso un'interfaccia grafica che genera successivamente un file di configurazione chiamato *ini file*, ogni ini file contiene una ed una sola suite caratterizzata da un nome, una descrizione ed uno o più test. Il file di configurazione, una volta generato, deve essere trasferito all'Raspberry Pi all'interno della cartella *ini* del progetto.

6.3 Ini File

Esaminiamo la struttura dei file di configurazione vedendo nello specifico tutte le sezioni che le compongono:

- **[SUITE]**

  Identifica l'inizio della suite, essa raggruppa al suo interno un insieme di test, questa sezione contiene due campi differenti:

  - **suite_name:** Il nome della Suite.
  - **suite_description:** Una descrizione sintetica relativa all'obiettivo della suite. Dopo una introduzione iniziale sulle caratteristiche principali della ECU da testare, ci si focalizzerà su quali funzionalità specifiche saranno validate.

- **[TEST_IDENTIFIER]**

  Questa sezione delimita l'inizio di un test, ogni test è identificato da un numero intero che differisce per ognuno di essi. Ogni test appartiene ad una specifica categoria, in base alla categoria di appartenenza variano i campi che formano il test; i campi comuni a tutte le categorie sono i seguenti:

  - **test_name:** Il nome del test.
  - **test_description:** Breve descrizione della singola funzionalità, di cui se ne verifica il corretto funzionamento.
  - **loop:** Questo campo è relativo al numero di iterazioni associate al test, esso è formato dalla seguente tupla:

    $$[numero\ di\ ripetizioni, condizione\ di\ terminazione]$$

    Dove:

  - **numero di ripetizioni:** Numero intero senza segno che rappresenta il numero di iterazioni da eseguire per il test, se il valore è uguale a zero allora il campo loop viene ignorato.
  - **condizione di terminazione:** Stringa testuale che può assumere tre possibili valori: "S", "F", "n_a". Nel caso in cui il numero di ripetizioni è uguale a zero, la condizione di terminazione sarà uguale ad "n_a" (not applicabile), in caso contrario sarà necessario eseguire il loop un numero indefinito di volte fino a quando il test non terminerà con esito positivo (valore "S" che sta per Success) o negativo (valore "F" che sta per failure).
  - **test_type:** Questo campo indica la categoria di appartenenza del test, essa può essere:
    - **detect_and_act:** Il test si mette in attesa per rilevare la ricezione di un singolo frame specifico, successivamente eseguirà una o più azioni non appena il rilevamento va a buon fine. In caso contrario, allo scadere del time-out verranno innescati una o più azioni differenti.

- **act_detect_act:** A seguito di una trasmissione inziale il cui scopo è quello di triggerare un certo comportamento nella stazione da validare, il test si metterà in ascolto ed opererà in modo analogo alla categoria precedentemente descritta.

- **detect_group:** Il test si metterà in attesa per rilevare un gruppo di frames, il rilevamento può imporre dei vincoli stringenti relativi ai timing di ricezione delle trame, oppure all'ordine di ricezione.

- **act_detect_group:** Dopo una trasmissione iniziale il cui scopo è quello di triggerare un certo comportamento nella stazione da validare, il test si metterà in ascolto sul CAN bus ed opererà in modo analogo alla categoria detect_group.

Approfondiamo nei paragrafi seguiti i campi di appartenenza di ognuna delle suddette categorie:

6.3.1 Detect and act

- **msg_id:** Campo ID, contenetene l'identificativo del frame da rilevare.

- **check:** Espressione booleana separata da operatori logici AND (&) e OR (|), gli operatori sono formati dalla seguente tupla:

$$[starting\ bit, lenght, endianess, value, type, signal\ name\ ]$$

Dove:

- **starting bit:** Indica il bit di partenza dalla quale inizia il segnale, questo valore non può essere più grande della dimensione del payload, ovvero non può essere maggiore di 63 nelle reti CAN, o maggiore di 511 nelle CAN FD.

- **length:** Numero intero positivo che indica la lunghezza in bit del segnale.

- **endianess:** Indica quale ordine dei byte utilizzare, il valore 1 indica la Big Endian, il valore 0 indica la Little Endian.

- **value:** Valore di riferimento da confrontare con quello relativo al segnale, ottenibile considerando la sequenza di lunghezza length partendo dallo starting bit.

- **type:** Questo campo indica il tipo di confronto che occorre fare tra il valore di riferimento contenuto nel campo *value* e quello relativo al segnale. Il *type* può assumere i seguenti valori:

  - **greater_than:** Strettamente maggiore.
  - **equal_to:** Uguale.
  - **lesser_than:** Strettamente minore.

- **signal name:** Nome del segnale da ricercare nel file DBC della ECU di riferimento.

Consideriamo alcuni esempi esplicativi:

$$Dato\ il\ frame\ 2E8\#9F72\ , verifichiamo\ il\ check:\ \ [6,4,1,5, greater\ than, \text{engineOilTemp}]$$

$$endianess = 1\ (Big\ Endian) \rightarrow\ Il\ MSB\ \text{è}\ all'incide\ 0, mentre\ il\ LSB\ \text{è}\ all'indice\ 15$$

$$il\ check\ sar\text{à}\ verificato\ se:\ Value_{signal} > Value_{rif} \rightarrow Value_{engineOilTemp} > 5$$

$$Payload = (9F72)_{hex} = (1001\ 1111\ 0111\ 0010)_{bin}$$

$$Calcoliamo\ V_{engineOilTemp}, consideriamo\ la\ sequenza\ che\ va\ dal\ bit\ con\ indice\ 6\ al\ bit\ con\ indice\ 9:$$

$Indice$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bit	1	0	0	1	1	1	1	1	0	1	1	1	0	0	1	0

$$Value_{engineOilTemp} = (1101)_{bin} = (13)_{dec}$$

$$Value_{engineOilTemp} > Value_{ref} \rightarrow\ 13 > 5\quad TRUE\ \checkmark$$

$$Per\ cui\ il\ check\ risulta\ soddisfatto$$

$$Dato\ il\ frame\ 10C\#4064\ , verifichiamo\ il\ check:$$

$$[12,3,1,7, equal\ to, eCall]\ \&\ [8,2,1,3, lesser\ than, warningState]$$

$$Payload = (4064)_{hex} = (0100\ 0000\ 0110\ 0100)_{bin}$$

$Indice$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bit	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	0

$$Value_{eCall} = (010)_{bin} = (2)_{dec}$$

$$Value_{warningState} = (01)_{bin} = (1)_{dec}$$

$$Value_{eCall} > Value_{ref} \rightarrow\ 2 = 7\quad FALSE\ X$$

$$Value_{warningState} < Value_{ref2} \rightarrow\ 1 < 3\quad TRUE\ \checkmark$$

$$Calcoliamo\ l'AND\ risultante: FALSE\ \&\ TRUE = FALSE\quad X$$

$$Per\ cui\ il\ check\ risulta\ non\ soddisfatto$$

- **period:** Intervallo di tempo che deve passare tra due frames con lo stesso msg_id, per considerare la ricezione valida. Se il campo period è uguale a zero, non è presente nessun vincolo sulle tempistiche di ricezione.

- **period_comparison_type:** Campo formato dalla seguente tupla:

$$[number\ of\ messages, comparison\ type, tolerance]$$

Dove:

- **number of messages:** Rappresenta il numero di messaggi con il msg_id che occorre rilevare in ricezione, per ognuno di essi è necessario verificare il campo check.

- **comparison type:** Può assumere i seguenti valori:

  - **greater_than:** strettamente maggiore
  - **lesser_than:** strettamente minore
  - **equal_to:** uguale
  - **n_a:** da non considerare, nel caso in cui il period è uguale a zero.

- **tolerance:** È un numero compreso tra 0 e 100, rappresenta il valore percentuale della tolleranza con cui si valuta il periodo. In base al valore del campo comparison type, è necessario impostare la disequazione corrispondente.

Supponiamo di avere:

$period = T_k$
$period\ comparison\ type = [\ n_k, type, tol\ ]$

La ricezione va a buon fine se:

1.) $Vengono\ rilevati\ n_k\ frames\ con\ id = msg\ id\ e\ payload\ che\ soddisfa\ il\ campo\ check.$

2.) $Indichiamo\ con\ timestamp_k\ \ il\ timestamp\ di\ arrivo\ del\ k - esimo\ frame\ ricevuto$

$$\forall j \in [0, k-2] :$$

$$Se\ comparison\ type == \text{ greater than}$$
$$Deve\ essere\ verificato\ che:\ \ timestamp_{k+1} - timestamp_k > T_n - T_n * {}^{tol}/_{100}$$

$$Se\ comparison\ type == \text{ lesser than}$$
$$Deve\ essere\ verificato\ che:\ \ timestamp_{k+1} - timestamp_k < T_n + T_n * {}^{tol}/_{100}$$

$$Se\ comparison\ type == \text{ equal than}$$
$$Deve\ essere\ verificato\ che:$$
$$T_n - T_n * {}^{tol}/_{100} < timestamp_{k+1} - timestamp_k < T_n + T_n * {}^{tol}/_{100}$$

- **timeout:** Se uguale a zero, il timeout non viene considerato, per cui il test rimane in ascolto per un tempo indeterminato, la sua esecuzione terminerà alla ricezione dei frame desiderati oppure alla terminazione del programma stesso.

Un qualsiasi valore diverso da zero indica un tempo in millisecondi entro in quale la ricezione deve essere rilevata, altrimenti il test fallisce.

- **command_delay_n:** Indica il tempo in millisecondi che è necessario aspettare a seguito di una rilevazione avvenuta con successo.

- **command_n:** Azione che deve essere triggerata quando il test termina ritornando un valore *TRUE*, è possibile innescare più azioni contemporaneamente. Ognuna di esse si rifà ad un template standard che analizzeremo nel seguito.

- **command_parameter_n:** Parametri da passare al rispettivo *command_n*.

- **else_command_delay_n**: Indica il tempo in millisecondi che è necessario aspettare allo scadere del timeout, ovvero quando il test fallisce.

- **else_command_n:** In maniera duale al *command_n,* le azioni da triggerare quando il test termina con *FALSE*.

- **else_command_parameter_n:** Parametri da passare all'*else_commad_n*.

6.3.2 Act detect act

Questa categoria, rispetto alla precedente è caratterizzata dagli stessi campi, ad eccezione di uno aggiuntivo:

- **traffic_data_for_initial_send:** Questo campo è settato uguale ad una stringa testuale della forma [TRAFFIC_N], essa fa riferimento ad una sezione di traffico, contenente le trame da inviare prima di mettersi in ascolto. Analizzeremo la struttura delle sezioni di traffico successivamente.

6.3.3 Detect Group

Rispetto al *detect_and_act*, i test appartenenti a questa categoria sono caratterizzati da un gruppo di frames differenti da dover rilevare. Per cui saranno presenti i seguenti campi:

- **msg_id_n:** Campo ID, contenetene l'identificativo del n-esimo frame da rilevare.

- **check_n:** Espressione booleana da dover verificare per l'n-esimo frame, la sintassi di questo campo è identica a quella del campo *check* delle categorie precedenti.

- **time_n:** Intervallo di tempo che deve essere rilevato tra la prima occorrenza del messaggio n-esimo e la prima occorrenza del messaggio n-1-esimo.

- **period_n:** Intervallo di tempo che deve essere rilevato tra i tempi di arrivo di due frames aventi lo stesso identificativo.

- **period_comparison_type_n:** Il formato di questo campo è pressoché identico a quello delle categorie del *period_comparison_type* delle categorie precedenti, con l'unica differenza che il campo *tolerance* si applica sia al *period_n* che al *time_n*.

- **match_type:** Questo campo specifica il tipico di rilevamento che occorre fare, esso può assumere i seguenti valori:

    - **or:** La fase di detection risulterà superata con successo se almeno uno degli n messaggi viene rilevato correttamente. In questo caso i campi *time_n*, *period_n* e *period_comparison_type_n* vengono ignorati.

    - **forbidden:** Match type duale rispetto all'or, se anche solo uno degli n messaggi viene rilevato correttamente il test fallisce. Il test terminerà con esito positivo nel caso in cui dopo lo scadere del time-out nessuno dei messaggi del gruppo verrà rilevato.

    - **lame:** In questo caso il test terminerà ritornando *TRUE*, se tutti i messaggi verranno rilevati entro lo scadere del time-out. I timestamp di arrivo dei frames non saranno considerati.

    - **simple:** A differenza del *lame*, con un match type di tipo *simple* è necessario che, oltre al rilevamento di tutti i messaggi, questi ultimi siano ricevuti con dei timestamp coerenti. Tali per cui la disequazione corrispondente che lega il periodo, la differenza di tempo tra i due timestamp, la tolerance ed il period_comparison_type sia verificata.

    - **strict:** Match type più stringente tra tutti, in questo caso verrà considerato oltre al vincilo di tempo imposto dal campo *period*, anche quello imposto dal campo *time*.

Consideriamo ora alcuni contesti applicativi per comprendere meglio il funzionamento del match_type.

Consideriamo il seguente *detect group*:

```
msg_id_1 = 0x2EF
check_1 = [9, 3, 1, 4, greater_than, EngineOilTemp]
time_1 = 0
period_1 = 100
period_comparison_type_1 = 5, greater_than, 3

msg_id_2 = 0x5AE
check_2 = [32, 1, 1, 1, equal_to, FuelWaterPresentSts]
time_2 = 0
period_2 = 80
period_comparison_type_2 = 2, lesser_than, 3

match_type = simple
```

Supponiamo inoltre la ricezione del seguente gruppo di frame:

Timestamp	Frames
(1638982895.615231)	2EF#FF70
(1638982895.665628)	564#0444F6740211BC08
(1638982895.716634)	2EF#FF70
(1638982895.767399)	082#0C973878E0D48F07
(1638982895.817865)	48E#6F207F
(1638982895.868565)	2EF#FF70
(1638982895.919340)	116#427991637CF9E575
(1638982895.969804)	45C#9A84F46D
(1638982896.020614)	2EF#FF70
(1638982896.071424)	1E7#F1035744737AE846
(1638982896.121922)	314#77263F79
(1638982896.172747)	0BF#7648A3
(1638982896.223591)	2EF#FF70
(1638982896.274067)	511#F3BF2847F8B6394D
(1638982896.324663)	309#6C312214
(1638982896.375371)	779#2FE7784E04CD8D0B
(1638982896.415874)	5AE#FE5555FF80
(1638982896.476629)	5AE#FE5555FF80
(1638982896.527030)	298#B3C1B6
(1638982896.578374)	5B2#F0F390

Filtriamo i frame del detect group, prendendo i frames con id 0x2EF e 0x5AE.

Timestamp	Frames
(1638982895.615231)	2EF#FF70
(1638982895.716634)	2EF#FF70
(1638982895.868565)	2EF#FF70
(1638982896.020614)	2EF#FF70
(1638982896.223591)	2EF#FF70
(1638982896.415874)	5AE#FE5555FF80
(1638982896.476629)	5AE#FE5555FF80

Verifichiamo se il corrispettivo campo check è verificato:

$$il\ check\ sarà\ verificato\ se:\ Value_{signal} > Value_{rif} \rightarrow Value_{engineOilTemp} > 4$$

$$Payload = (FF70)_{hex} = (1111\ 1111\ 0111\ 0000)_{bin}$$

$Calcoliamo\ V_{engineOilTemp}, consideriamo\ la\ sequenza\ che\ va\ dal\ bit\ con\ indice\ 6\ al\ bit\ con\ indice\ 9$:

$$\begin{array}{c|cccc|cccc|cccc|cccc} Indice & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline Bit & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{array}$$

$$Value_{engineOilTemp} = (111)_{bin} = (7)_{dec}$$

$$Value_{engineOilTemp} > Value_{ref} \rightarrow 7 > 4 \quad TRUE\ \checkmark$$

$$Per\ cui\ il\ check\ risulta\ soddisfatto$$

$$il\ check\ sarà\ verificato\ se:\ Value_{signal} > Value_{rif} \rightarrow Value_{FuelWaterPresentSts} = 1$$

$$Payload = (\text{FE5555FF80})_{hex} = (1111\ 1110\ 0101\ 0101\ 0101\ 0101\ 1111\ 1111\ 1000\ 0000)_{bin}$$

$Calcoliamo\ V_{engineOilTemp}, consideriamo\ la\ sequenza\ che\ va\ dal\ bit\ con\ indice\ 6\ al\ bit\ con\ indice\ 9$:

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Bit	1	1	1	1	1	1	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0

$$Value_{FuelWaterPresentSts} = (1)_{bin} = (1)_{dec}$$

$$Value_{FuelWaterPresentSts} = Value_{ref} \rightarrow 1 = 1 \quad TRUE\ \checkmark$$

$$Per\ cui\ il\ check\ risulta\ soddisfatto$$

Consideriamo ora i timestamp dei messaggi ricevuti dividendoli in due gruppi:

```
2EF:
1638982895.615231,
1638982895.716634,
1638982895.868565,
1638982896.020614,
1638982896.223591
```

```
5AE:
1638982896.415874,
1638982896.476629
```

$$Calcoliamo\ timestamp_{k+1} - timestamp_k:$$

$$2EF: 1638982895.716634 - 1638982895.615231 = 0.10140299797 \approx 101\ ms$$
$$2EF: 1638982895.868565 - 1638982895.716634 = 0.15193104744 \approx 152\ ms$$
$$2EF: 1638982896.020614 - 1638982895.868565 = 0.15204882621 \approx 152\ ms$$
$$2EF: 1638982896.223591 - 1638982896.020614 = 0.20297718048 \approx 203\ ms$$
$$5AE: 1638982896.476629 - 1638982896.415874 = 0.06075501441 \approx 61\ ms$$

Sapendo che:

```
period_1 = 100
period_comparison_type_1 = 5, greater_than, 3
```

$$Dobbiamo\ verificare\ la\ disequazione:$$

$$\forall k \in [0,4]\ \ timestamp_{k+1} - timestamp_k > T_1 - T_1 * tol/100$$

$$timestamp_{k+1} - timestamp_k > 100 - 100 * 0.03 \rightarrow \ timestamp_{k+1} - timestamp_k > 97ms$$

$$101\ ms > 97\ ms\ \checkmark$$
$$152\ ms > 97\ ms\ \checkmark$$
$$152\ ms > 97\ ms\ \checkmark$$
$$203\ ms > 97\ ms\ \checkmark$$

```
period_2 = 80
period_comparison_type_2 = 2, lesser_than, 3
```

$$Dobbiamo\ verificare\ la\ disequazione:$$

$$timestamp_2 - timestamp_1 < T_2 + T_2 * tol/100$$

$$timestamp_2 - timestamp_1 < 80 + 80 * 0.03 \rightarrow \ timestamp_2 - timestamp_1 < 82.4\ ms$$

$$61\ ms < 82.4\ ms \ \checkmark$$

Per cui il match type simple risulta verificato.
Consideriamo ora lo stesso detect group ma con match differente, ovvero:

```
msg_id_1 = 0x2EF
check_1 = [9, 3, 1, 4, greater_than, EngineOilTemp]
time_1 = 0
period_1 = 100
period_comparison_type_1 = 5, greater_than, 3

msg_id_2 = 0x5AE
check_2 = [32, 1, 1, 1, equal_to, FuelWaterPresentSts]
time_2 = 800
period_2 = 80
period_comparison_type_2 = 2, lesser_than, 3

match_type = strict
```

Supponiamo la ricezione dello stesso gruppo di frames dell'esempio precedente.
Per cui già sappiamo che i campi check sono soddisfatti per tutti i frames di entrambi i messaggi, consideriamo quindi i loro corrispettivi timestamps.

```
2EF:
1638982895.615231,
1638982895.716634,
1638982895.868565,
1638982896.020614,
1638982896.223591
```

```
5AE:
1638982896.415874,
1638982896.476629
```

Abbiamo già verificato il superamento del vincolo imposto dal campo period, per cui soffermiamoci su quello imposto dal campo time.

```
time_2 = 800
```

$$Dobbiamo\ verificare\ la\ disequazione:$$

$$t_{n+1} - t_{n+1} * \frac{tol}{100} < timestamp_{n+1,1} - timestamp_{n,1} < t_{n+1} + t_{n+1} * \frac{tol}{100}$$

$$800 - 800 * 0.03 < timestamp_{n+1,1} - timestamp_{n,1} < 800 + 800 * 0.03$$

$$776\,ms < timestamp_{n+1,1} - timestamp_{n,1} < 824\,ms$$

$$776\,ms < 1638982896.415874\text{ sec} - 1638982895.615231\text{ sec} < 824\,ms$$

$$1638982896.415874\text{ sec} - 1638982895.615231\text{ sec} = 0.80064296722\text{ sec} \approx 800.64\,ms$$

$$776\,ms < 800.64\text{ ms} < 824\,ms \;\checkmark$$

Per cui il match type strict risulta verificato correttamente.
A seguire gli altri campi presenti nel detect group sono analoghi a quelli visti per le altre categorie, ovvero:

- **timeout**
- **command_delay_n**
- **command_n**
- **command_parameter_n**
- **else_command_delay_n**
- **else_command_n**
- **else_command_parameter_n**

6.3.4 Act detect group

Questa categoria, rispetto alla precedente presenta le stesse caratteristiche, differisce unicamente per la presenta di un campo aggiuntivo:

- **traffic_data_for_initial_send**

6.3.5 [TRAFFIC_N]

Questa sezione presenta un gruppo di frames da inviare, l'invio può precedere la fase di detection nel caso di test appartenenti a categorie *act_detect_act* o *act_detect_group* o alla terminazione del test, in quanto come vedremo dopo un comando chiamato "*CAN_Inject_start*" può richiedere l'invio di traffico. Una sezione di traffico è formata dai seguenti campi:

- **msg_id_n:** ID del n-esimo frame da inviare.
- **payload_n:** Payload del n-esimo frame da inviare.
- **time_n:** Tempo che il frame n-1-esimo deve aspettare prima di essere inviato.
- **count_n:** Numero di volte che il frame n-esimo deve essere inviato.
- **period_n:** Tempo in millisecondi che deve passare tra due frame n-esimi.

6.3.6 Command

In questa sezione analizzeremo tutti i possibili comandi i quali, rappresentano tutte le potenziali azioni innescabili a seguito della terminazione di un test.

- **CAN_record_start:** Il test si mette in ascolto e memorizza tutti i frame ricevuti in un file di log, ovvero in riferimento a quanto visto in precedenza, viene eseguita la funzione di candump passando il flag l. I parametri che riceve il comando sono il nome del file su cui si vuole scrivere ed il numero massimo di millisecondi per cui il candump deve rimanere attivo.

- **CAN_record_stop:** Termina la memorizzazione dei frame ricevuti da un comando di CAN_record_start. Questo comando ha come parametro il nome del file su cui il CAN_record_start sta scrivendo. Può essere eseguito solo se il timeout del CAN_record_start non è ancora terminato.

- **CAN_rise_alert:** Notifica pop-up per l'utente, (generata tramite libnotify), il parametro è il messaggio che deve essere visionato dall'utente.

- **CAN_inject_start:** Inizia la trasmissione di un gruppo di frame, il parametro in questo caso è il numero di una sezione di traffico.

- **CAN_inject_stop:** Per terminare preventivamente l'invio dei frame dovuti ad un CAN_inject_start.

- **CAN_inject_from_trace:** Permette l'invio di tutte le trame contenute in un file di log, ottenibili ovvero attraverso un comando di CAN_record_start. Il parametro è il nome del file di log. Questo comando è realizzato mediante la funzione canplayer di socketCAN.

- **Start_test:** Per avviare un certo test della suite, eccetto il test stesso. Il parametro è l'identificativo del test all'interno della suite.

6.4 Blackbox

Il requisito fondamentale alla base del funzionamento di WhiteBox è quello di conoscere nel dettaglio il dispositivo da testare. Ciò implica che oltre al file DBC della ECU in questione sia necessario conoscere la modalità di funzionamento della stazione, ovvero il protocollo di rete che utilizza (CAN, CAN FD) e la relativa frequenza di trasmissione. È compito dell'utilizzatore definire correttamente l'interfaccia di rete, specificando sia il protocollo che il bitrate per consentire lo scambio di informazioni. A differenza di Ethernet, il protocollo CAN non prevede una fase ai autonegoziazione, ovvero una fase preliminare in cui le stazioni interconnesse non appena rilevano la presenza di un nuovo host collegato, inviano una sequenza di segnali sul canale per negoziare quale frequenza di trasmissione e quale modalità di funzionamento (Half duplex o Full duplex) utilizzare, le schede di rete partono dalla modalità migliore da poter utilizzare e poi vanno successivamente a scendere, ad esempio nel caso di Ethernet si parte da 1 Gbit/s in modalità Full Duplex, poi 1 Gbit/s in Half Duplex, poi 100 Mbit/s Full duplex e così via. Nel protocollo CAN, quindi non sono gli host a rilevare

direttamente quali sono i parametri per comunicare correttamente, ma è compito dell'utente definire opportunamente l'interfaccia di rete, una definizione errata di questi parametri porta gli host comunicanti a rilevare degli errori di trasmissione. Nello specifico la stazione ricevitrice incrementerà il suo contatore di REC e genererà successivamente un Error Frame non recepibile all'altra stazione, il quale a sua volta incrementerà il suo TEC ed invierà anch'essa un Error Frame. La procedura innesca quindi un loop che alla lunga porterà entrambe le stazioni in stato di Bus Off necessitando quindi di un reset Hardware.
Il modulo di BlackBox simula un host malevolo collegato alla rete CAN, il cui obiettivo è quello di ottenere informazioni riguardo le ECU e la rete stessa per poi effettuare degli attacchi informatici volti ad alterare il normale funzionamento della rete, con il fine di creare un disservizio che potrebbe anche mettere a rischio l'incolumità del guidatore.

6.5 Fase di calibrazione

La prima azione svolta da questa entità è quella di identificare i parametri di rete, alla base di ogni possibile attacco vi è la necessità di poter sia inviare che ricevere frame correttamente. In primo luogo, è necessario trovare il bitrate corretto, lanciare un candump ad un bitrate differente rispetto a quello delle altre stazioni connesse porta la scheda di rete a rilevare in ricezione frames con il formato 07C#000F000000000000. La strategia risolutiva è quella di effettuare un candump per monitorare e memorizzare in un file di log un numero ridotto di frame (10-20) per ogni bitrate, analizzando il contenuto del file è possibile dedurre se il bitrate è corretto oppure no, verificando che ci siano almeno tre frames con id diverso tra loro (questo perché i frames generati in caso di errore potrebbero essere di due diversi tipi). Si parte da una frequenza di 125 Kbit che viene raddoppiata a 250 Kbit al secondo tentativo, successivamente ogni volta in cui la ricezione non va a buon fine, la frequenza viene aumentata di 250 Kbit fino al raggiungimento di 1 Mbit/s che rappresenta il massimo bitrate per la trasmissione durante la fase di arbitraggio (sia per CAN che per CAN FD).

6.6 Rilevazione del Protocollo

È possibile distinguere trame appartenenti al protocollo CAN rispetto a trame appartenenti al protocollo CAN FD direttamente dal formato di stampa del candump, sebbene la distinzione dei due protocolli sia facilmente rilevabile dalla differente lunghezza dei loro campi.
Possiamo verificare quanto detto, facendo il seguente esperimento:
Lanciamo in un primo terminale il seguente comando:

```
user:~$ candump vcan0 -L
```

In un secondo terminale lanciamo il comando:

```
user:~$ cangen vcan0 -n 5 && cangen vcan0 -n 5 -f
```

Ovvero verranno eseguiti sequenzialmente due cangen, il primo invierà cinque frames relativi al protocollo CAN e terminerà la sua esecuzione. A seguire il secondo cangen invierà anch'esso cinque frames ma relativi al protocollo CAN FD.
Il candump rileverà i frame di entrambi i comandi stampandoli nell'altra shell.

```
parallels@ubuntu-linux-20-04-desktop: ~/Desktop/luca_rpi_development
parallels@ubuntu-linux-20-04-desktop:~/Desktop/luca_rpi_development$ candump vcan0 -L
(1644601506.915329) vcan0 753#59219F38E7ECD3
(1644601507.120823) vcan0 52E#C1CA506D
(1644601507.326145) vcan0 1BE#55AFC3539C237149
(1644601507.526718) vcan0 6C1#
(1644601507.727426) vcan0 17E#D332131BF83ECC49
(1644601507.934862) vcan0 313##03977825AE6BD19553977825AE6BD19553977825A
(1644601508.135039) vcan0 34D##01D089D0B27C43F711D089D0B27C43F71
(1644601508.335264) vcan0 4B5##0DC44153605084E1DDC44153605084E1DDC44153605084E1DDC44153605084E1D
(1644601508.536151) vcan0 11A##0401E50
(1644601508.736493) vcan0 31E##0
```

Figura 6.1: Output candump

Possiamo notare che i frame relativi al protocollo CAN, ovvero i primi cinque sono stampati con il seguente formato:

$$timestamp\ \ interface\ \ id\#payload$$

Mentre i frame appartenenti al protocollo CAN FD, ovvero gli ultimi cinque, sono stampati con il formato:

$$timestamp\ interface\ id\#\#payload$$

Possiamo notare la presenza di due cancelletti separatori tra id e payload anziché solo uno. Dopo aver identificato correttamente il bitrate, la distinzione tra CAN e CAN FD risulta quindi estremamente banale. Prendendo in esame una qualsiasi riga del file di log è semplicemente necessario contare le occorrenze del carattere '#', qualora ci dovesse essere una unica occorrenza del suddetto carattere, il protocollo utilizzato sarebbe il CAN, in caso contrario sarebbe il CAN FD.
Risulta invece più complesso distinguere i frame del protocollo J1939 rispetto a quelli del CAN FD, oltre a non essere presente nessuna distinzione nel formato di stampa, il J1939 presenta un formato pacchetto simile a quello del CAN FD, nello specifico in entrambi i casi l'ID è codificato su 29 bit. Come già visto nelle sezioni precedenti, il SAE J1939 è un protocollo di livello cinque che si appoggia su CAN per il livello fisico ed il data link layer, nello specifico J1939 utilizza CAN 2.0B con gli identificativi estesi a 29 bit. Il protocollo, a differenza di CAN, impone dei limiti rispetto allo spazio di indirizzamento del campo, alcuni ID non possono essere utilizzati.

Il campo identifier è formato da 29 bit, per cui lo spazio di indirizzamento permette di distinguere $2^{29} = 536870912$ indirizzi differenti, tuttavia dobbiamo ricordare due principali vincoli che portano alla diminuzione dello spazio di indirizzamento:

- Il quarto bit dell'ID è il Reserved Bit, un bit riservato per usi futuri, il quale è sempre posto uguale a zero.

- Il Parameter Group Number è una parte dell'identificativo formata da 18 bit, il protocollo SEA J1939 impossibilità l'utilizzo di alcuni PGN, nello specifico tutti i valori compresi tra 0x00FF00 e 0x00FFFF sono riservati.

Per cui una potenziale strategia per rilevare l'utilizzo del protocollo SAE J1939 potrebbe essere quella di verificare che tutte le trame ricevute abbiano il Reserved Bit uguale a zero e che il PGN di un frame non sia compreso nell'intervallo dei valori riservati.
L'idea è quindi quella di analizzare un numero elevato di frames e verificare se i vincoli descritti precedentemente sono soddisfatti oppure no. È importante sottolineare che, questo metodo non porta all'assoluta certezza che il protocollo rilevato sia effettivamente quello, per quanto il numero di frames analizzati sia elevato potrebbe capitare che casualmente, tutti i vincoli siano soddisfatti ma che il protocollo di rete sia il CAN anziché il J1939.
Notiamo che il secondo vincolo risulta essere poco stringente, consideriamo il seguente contesto applicativo: supponiamo che ogni identificativo abbia la stessa possibilità di essere generato:

$$Numero\ di\ indirizzi\ riservati:\ 0x00FFFF - 0x00FF00 = 0x0000FF = 255_{10}$$

$$Numero\ di\ \ indirizzi\ totatli: 2^{29} = 536870912$$

$$Percentuale\ di\ indirizzi\ riservati\ sul\ totale:\ \frac{255}{536870912}\% = 0.00004749745\%$$

Il primo vincolo risulta invece abbastanza stringente, supponendo una distribuzione uniforme dei bit trasmessi, ovvero presupponendo che un qualsiasi bit abbia il 50% di possibilità di essere uno zero oppure un uno, è piuttosto improbabile che su un numero elevato di frames analizzati nessuno presenti il quarto bit uguale a 1 nel caso in cui il protocollo sia CAN.
Basti pensare che su venti frames analizzati la probabilità che nessuno di questi presenti il Reserved bit uguale a 1 è: $0.5^{20} = 0.00009536743\ \%$. Possiamo quindi reputare il suddetto metodo estremamente affidabile per il rilevamento del J1939.

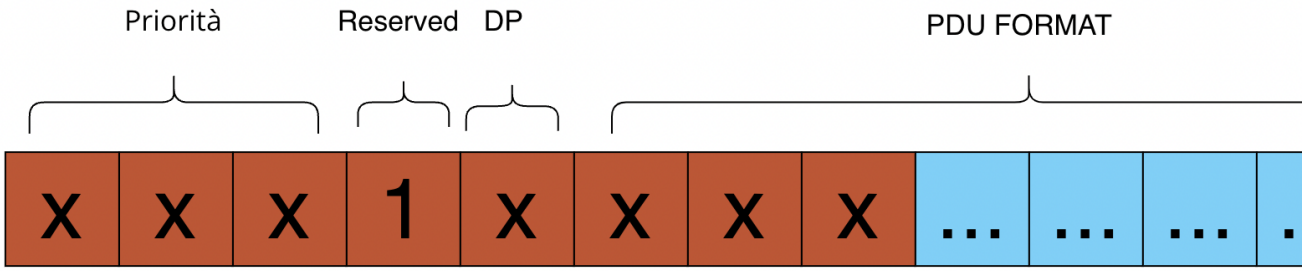

Figura 6.2: Primo byte del campo ID di PDU J1939

Possiamo quindi focalizzare l'analisi sul valore del primo nibble, nello specifico sappiamo che quest'ultimo non può avere il LSB uguale a zero. I tre MSB possono assumere qualsiasi valore, per cui ci sono $2^3 = 8$ valori non accettabili ovvero:

$$0000 = \ 0x0$$
$$1110 = 0xE$$
$$0010 = 0x2$$

$$0100 = 0x4$$
$$1000 = 0x8$$
$$0110 = 0x6$$
$$1010 = 0xA$$
$$1100 = 0xC$$

Se il primo nibble del primo byte di un qualsiasi frames è uguale ad uno di questi valori, il protocollo è sicuramente il CAN, altrimenti se nessuno dei frames analizzati presenta uno di questi valori, allora si presuppone che il protocollo utilizzato sia il J1939.

6.7 Attacchi Informatici

Dopo aver rilevato correttamente sia il protocollo di rete che il bitrate è finalmente possibile effettuare degli attacchi informatici per alterare il normale stato di funzionamento della rete. La rilevazione di protocolli differenti porta a modifiche estremamente riduttive, nello specifico nel caso in cui sia stato rilevato l'utilizzo del protocollo CAN non è necessario specificare nessun flag, nel caso in cui sia stato rilevato il CAN FD è necessario specificare il flag f per l'utilizzo del cangen, mentre per il J1939 il flag e per generare frames in formato esteso.
Rispetto agli attacchi informatici citati a livello teorico nel capitolo quattro, l'unico implementabile effettivamente è il Bus Flood Attack; la maggior parte degli attacchi necessità di modificare alcuni bit specifici della trama, la scrittura dei singoli bit è un compito che viene svolto dal Trascriver, componente realizzato in Hardware dalla scheda di rete e quindi non modificabile.
Anche il livello di collegamento dati è implementato dalla PiCAN FD, quindi anch'esso risulta non modificabile pregiudicando la possibilità di effettuare un potente attacco, ovvero quello di inviare un elevato numero di trame ad un bitrate volutamente differente rispetto a quello delle altre ECU, in modo tale da far incrementare i contatori di REC delle stazioni ascoltatrici fino al raggiungimento del loro stato di Bus Off. Il problema è che sia la gestione degli stati di errore che l'incremento o decremento dei contatori di TEC e REC avvenendo nel Data Link Layer risultano non modificabili. Questo comporterebbe un incremento di otto unità del TEC della stazione attaccante ad ogni trasmissione, contro l'incremento di una unità del REC delle stazioni ascoltatrici. Così facendo, la stazione attaccante andrà in stato di Bus off prima delle stazioni attaccata.
Una stazione va in stato di Bus Off nel momento in cui uno dei due contatori supera il valore di soglia di 255, nel caso dell'host malevolo questa assunzione implica il raggiungimento dello stato di Bus Off dopo $^{255}/_{8} \approx 32$ trasmissioni, nel mentre le stazioni attaccate avranno subito un incremento del contatore di REC di sole 32 unità nel caso peggiore, ricordiamo che nel momento in cui una stazione riceve correttamente un frame, il contatore di REC subisce un decremento di uno. Tralasciando una eventuale modifica della scheda di rete, un attacco di questo tipo potrebbe essere attuabile nel caso in cui sia possibile effettuare un reset Hardware della stazione attaccante da remoto, in questo modo al riavvio i due contatori verranno resettati a zero permettendo all'host di trasmettere nuovamente.
Gli attacchi implementati sono i seguenti:

- **Bus Flood Attack:** Dopo un periodo di ascolto preventivo della rete in cui l'host malevolo elabora tutti i frames ricevuti, memorizza la trama avente l'identificativo con priorità massima, ovvero numericamente minore. La stazione attaccante elabora i primi 1000 frames, ricava l'ID più piccolo e successivamente invia con una periodicità di 15 millisecondi trame con il suddetto id. L'attacco è realizzato mediante l'uso di un *candump* e di un successivo *cangen*.

- **Reply Attack:** L'attacco di tipo reply ha lo scopo di indurre un funzionamento inaspettato del veicolo, ovvero quello di provocare dei cambiamenti fisici nella vettura che possono portare a conseguenze anche gravi. L'host malevolo si mette in ascolto per dieci secondi, salvando tutte le trame ricevute in un file di log e successivamente invia tutte le trame ricevute. Così facendo, se l'utente ha effettuato delle azioni durante il periodo di recording, come ad esempio cambio di una stazione radio, abbassamento del finestrino o apertura/chiusura del bagagliaio queste azioni porteranno alla generazione di specifiche trame che se rinviate porterebbero ad innescare nuovamente le suddette azioni. Questo tipo di attacco risulterà sempre attuabile in quanto il protocollo CAN non permette l'autenticazione dei messaggi, in ricezione non è possibile comprendere se il messaggio è stato generato dalla stazione prevista oppure da una malevola. Il Reply Attack è realizzato mediante l'uso di un *candump* con il corrispettivo flag l, per la generazione del file di log e di un successivo *canplayer* che prende come parametro il nome del file di log appena generato.

- **Spoofing Attack:** Attacco simile al Reply Attack, il cui scopo è lo stesso, ovvero quello di innescare dei comportamenti indesiderati all'interno dell'auto. Rispetto al Reply Attack i payload non vengono inviati così come sono ricevuti ma questi ultimi vengono generati casualmente. Dopo un periodo iniziale di ascolto in cui vengono memorizzati tutti i frames ricevuti, per ognuno di essi si invia una trama con stesso id e payload generato randomicamente. Lo Spoofing Attack è realizzato mediante l'uso di un *candump* con il flag l per la generazione del file di log, il file viene elaborato per ottenere l'insieme di tutti gli id presenti, successivamente per ognuno di essi viene lanciato un cangen con il flag I per specificare iterativamente l'n-esimo id ed il flag n per genera un unico frame per ogni identificativo.

Code Development

7.1 Introduzione

Il programma è caratterizzato da un sistema concorrente basato su più processi, formati a loro volta su più thread. Il programma è scritto in C++ 17, in conformità con lo standard ISO/IEC 14882. Per lo sviluppo del codice si è seguito un approccio orientato agli oggetti; i principi fondamentali che hanno motivato tutte le scelte implementative sono la modularità del codice e la leggibilità dello stesso, per questo la definizione di ogni modulo è svolta nei rispettivi file di header (aventi estensione .h) mentre l'implementazione è fatta nel relativo file .cpp.
Il programma al suo interno presenta le seguenti cartelle:

- **img:** directory contenente immagini differenti (tra cui frecce, pulsanti, icone, logo dell'azienda) utilizzate per l'implementazione dell'interfaccia grafica.

- **ini:** Cartella contenente uno o più file di configurazione contenenti la suite con i relativi test, utilizzati per il modulo di WhiteBox.

- **log:** In questa cartella viene generato un file di log dopo ogni simulazione, per offrire uno strumento utile in fase di debugging del modulo di WhiteBox. Viene generato un file testuale in cui si riporta in che ordine vengono avviati i test ed in che ordine terminano, permettendo di capire la causa di una terminazione inattesa e per verificare se la modalità di esecuzione selezionata dall'utente (in parallelo o sequenziale) sia effettivamente rispettata.

- **records:** In questa cartella vengono salvati i file di record generati tramite il comando di candump.

- **screens:** Cartella in cui sono presenti degli screenshot per ogni finestra differente dell'interfaccia grafica.

- **debug test:** Per ogni test della suite di riferimento, è presente un file di log lanciabile tramite il comando di canplayer che permette di verificare la terminazione con successo di un test, è presenta un file di log anche per verificare la terminazione in caso di fallimento.

- **socketCAN:** Directory contenente il codice sorgente di ogni funzione di socketCAN utilizzata dal programma, per ogni funzione (cangen, candump, canplayer, cansend ecc.) è presente il rispettivo file contenente il codice scritto in linguaggio C, un makefile per compilare ogni file .c e i relativi eseguibili ottenuti dalla compilazione.

7.2 CAN Utils

La funzione *std::system* definita nella libreria *cstdlib* permette di eseguire un comando da shell dal processo chiamante. Presupponendo un utilizzo in ambiente UNIX, l'utente non deve occuparsi della creazione di un nuovo processo tramite *fork,* della funzione *execvp* per sostituire il processo corrente con uno nuovo e della funzione *wait* per attendere la terminazione del processo.
Si è tuttavia ritenuto più funzionale evitare l'uso della *std::system* ed utilizzare direttamente il codice sorgente delle funzioni di socketCAN, pur introducendo una complessità maggiore a livello di scrittura del codice questo approccio presenta due vantaggi sostanziali:

1.) La *std::system* è per definizione bloccante, il processo padre è bloccato fin tanto che il processo figlio non termina la propria esecuzione. Utilizzando le primitive *fork*, *execvp* e *wait* è possibile utilizzare le funzioni di socketCAN anche in modalità non bloccante, offrendo quindi maggior elasticità.

2.) Avendo accesso al codice sorgente è possibile effettuare delle modifiche, personalizzando i CAN Utils qualora necessario. Nello specifico per le esigenze dell'applicazioni si è apportata una piccola modifica nel file *candump.c*.

```
parallels@ubuntu-linux-20-04-desktop: ~/Desktop/luca_rpi_development
parallels@ubuntu-linux-20-04-desktop:~/Desktop$ cd luca_rpi_development/
parallels@ubuntu-linux-20-04-desktop:~/Desktop/luca_rpi_development$ candump vcan0 -l
Disabled standard output while logging.

Enabling Logfile 'candump-2022-02-13_204621.log'
```

Figura 7.1: candump originale

Nel momento in cui il file di log viene creato, il nome del file è della forma: $candump\ aaaa - mm - gg - hh - timestamp$ dove il campo timestamp rappresenta l'orario convertito in millisecondi.

```
parallels@ubuntu-linux-20-04-desktop: ~/Desktop/luca_rpi_development/socketCAN
parallels@ubuntu-linux-20-04-desktop:~/Desktop/luca_rpi_development/socketCAN$ ./candump vcan0 -l
Disabled standard output while logging.
Enabling Logfile '1752442-2022-02-13_204707.log'
```

Figura 7.2: candump modificata

Nella figura 7.2 al posto di utilizzare il comando bash associato al candump, lanciamo l'eseguibile ottenuto dalla compilazione del file candump.c, notiamo che in questo caso il nome del file generato presenta il formato: $code - aaaa - mm - gg - hh - timestamp$ dove il codice rappresenta il PID del processo che esegue il candump. Questa modifica era necessaria per permette di distinguere i file di log creati da test differenti, qualora questi siano eseguiti parallelamente, in questo modo ogni test salvandosi il PID del processo figlio, può identificare il proprio file di log.

I file sorgenti scritti in linguaggio C sono pubblici e disponibili su un repository online citato nella documentazione. Le suddette funzioni sono state scritte nel periodo che va dal 2002-2009 da Volkswagen Group Electronic Research e tutti i diritti sono riservati.

7.3 CAN_Utilities

Esaminiamo ora il primo modulo effettivamente presente nel progetto, tutti i file presenti nella directory socketCAN fanno riferimento a processi esterni che devono essere opportunamente richiamati da un layer interno, visibile dal processo principale. Nel mentre in cui il processo principale esegue il programma, quando necessario, un nuovo processo viene creato per eseguire una specifica funzione CAN. È necessario fornire un meccanismo di comunicazione tra i processi, questo compito viene realizzato dal modulo di CAN_Utilities. CAN_Utilities rappresenta il layer di basso livello, il gateway che permette l'accesso agli strumenti offerti da socketCAN.
Il principio di funzionamento di ogni funzione di socketCAN è pressoché lo stesso, ad eccezione di alcuni casi particolari in cui specifico la modalità di esecuzione sincrona o asincrona, tuttavia, il template generico è il seguente:

```
void cangen(const std::vector<std::string> &args){

    int status;
    char** argv = new char*[args.size()+1];
    for ( int i = 0 ; i < args.size() ; i++ )
        argv[i] = const_cast<char*> (args.at(i).c_str() );

    argv[args.size()] = NULL;
    int PID = fork();
    if ( PID == 0 )
        execvp("./socketCAN/cangen",argv);

    wait(&status);
    delete[] argv;
}
```

Figura 7.3: Implementazione cangen

Un nuovo processo viene creato con la *fork()*, successivamente il processo figlio, ovvero il processo avente PID uguale a zero, esegue il rispettivo eseguibile di una specifica funzione di socketCAN (in questo caso la cangen), nel mentre il processo padre attende la terminazione del figlio essendo in questo caso la modalità di esecuzione sincrona. Tutti i flag che verrebbero specificati da Shell nel momento in cui lanciamo un comando, in questo caso devono essere passati come argomenti da linea di comando, il primo argomento è il nome dell'eseguibile mentre i successivi sono i flag con i rispettivi valori, eliminando i caratteri di spaziatura. Le funzioni ricevono come input un riferimento ad un vector di stringhe, da cui si ottiene un vettore di array di caratteri che viene passato alla execvp. Ad esempio, supponendo di voler eseguire un "cangen can0 -g 150 -n 5 -v" in modalità sincrona, si dovrebbero scrivere:

```
std::vector<std::string> args;
args.push_back("./cangen");
args.push_back("-g");
args.push_back("-150");
args.push_back("-n");
args.push_back("-5");
args.push_back("-v");

cangen(args);
```

Figura 7.4: Invocazione cangen

Nel caos in cui sia possibile definire una modalità di esecuzione sincrona o asincrona il template cambia leggermente:

```
int socketCAN_function(std::vector<std::string> args, int flag){

    int status;
    char** argv = new char*[args.size()+1];

    for ( int i = 0 ; i < args.size() ; i++ )
        argv[i] = const_cast<char*> (args.at(i).c_str() );

    argv[args.size()] = NULL;
    int PID = fork();

    if ( PID == 0 )
        execvp("./socketCAN/socketCAN_function",argv);

    if ( flag == SYNC )
        wait(&status);

    delete[] argv;

return PID;

}
```

Figura 7.5: Template generico

In questo caso la funzione riceve come argomento aggiuntivo un flag che può essere SYNC o ASYNC, in base al valore di questo flag può essere fatta una *wait* altrimenti si ritorna il PID del processo in esecuzione e sarà compito del processo padre occuparsi della terminazione del figlio.

Il modulo di CAN_Utilities definisce inoltre una classe al suo interno per la ricezione dinamica dei frames, di per sé il comando di candump non permette di elaborare i frames non appena sono ricevuti, è necessario fornire una qualche forma di meccanismo, per consentire la comunicazione tra processo padre e processo figlio. Altrimenti sarebbe necessario far terminare il candump dopo un certo tempo, memorizzando l'output in un file di log e controllare successivamente il contenuto del suddetto file. Questo causerebbe un significato ritardo nel rilevamento di particolari eventi che necessitano dei tempi di reazione brevi, questo ritardo viene evitato dalla classe CAN_Sniffer.

```
CAN_Sniffer::CAN_Sniffer(){

    std::vector<std::string> args;
    args.push_back("./candump");
    args.push_back("vcan0");
    args.push_back("-L");

    pipe(this->link);
    this->pid = fork();

    if ( this->pid == 0 ){

        dup2 (this->link[1], STDOUT_FILENO);
        close(this->link[0]);
        close(this->link[1]);
        candump(args,DYNAMIC);
    }

    else
        close(this->link[1]);
}
```

Figura 7.6: Costruttore classe CAN_Sniffer

Quando una istanza della classe viene creata, un nuovo processo viene creato tramite la *fork(),* lo standard output di questo processo viene ridirezionato in una pipe comunicante con il processo padre utilizzando la funzione *dup2()*.

```
std::string CAN_Sniffer::read_frame(){
    read(link[0], this->buffer, sizeof(this->buffer)-1 );
    std::string str(this->buffer);
    return str;
}
```

Figura 7.7: metodo read_frame

Il processo padre può ricevere i frames rilevati dal candump eseguito dal processo figlio invocando il metodo *read_frame*, che effettua una semplice operazione di read sulla pipe. Quando una istanza della classa viene rilasciata, il distruttore viene invocato:

```
CAN_Sniffer::~CAN_Sniffer(){
    kill(this->pid, SIGTERM);
}
```

Figura 7.8: Distruttore classe CAN_Sniffer

Dove il processo figlio viene terminato a seguito della ricezione di un segnale SIGTERM.

```
CAN_Sniffer cs{};
...
...
std::string str = cs.read_frame();
```

Figura 7.9: Invocazione metodo read_frame.

Il processo padre deve semplicemente dichiarare una istanza della classe *CAN_Sniffer* e successivamente invocare il metodo *read_frame* per ottenere subito i frame ricevuti. È importante notare che la read interna al read_frame è una system call bloccante, per cui il processo padre rimarrà bloccato fin quando un nuovo frame non sarà ricevuto. Inoltre, la pipe viene sovrascritta ogni volta dal processo figlio, se i frames arrivano molto velocemente ed il processo padre è troppo lento nel processarli, ci potrebbero essere dei frames che verranno sovrascritti prima di essere elaborati.

7.4 Utilities

Il modulo di Utilities è composto da un gruppo di funzioni eterogenee, esse fanno riferimento a contesti differenti: gestione del file system, manipolazione di stringhe, casting, conversioni numeriche, calcolo di timestamp ecc. Lo scopo principale del modulo di Utilities è quello di diminuire la verbosità del codice e consequenzialmente di aumentarne la leggibilità.

7.5 Parsing

I moduli di parsing hanno l'obiettivo di processare dei file testuali o delle stringhe di caratteri per riempire le corrispondenti strutture, utilizzate poi dagli altri componenti. Nello specifico sono presenti due sotto moduli di parsing ovvero:

- **Parsing_log:** Questo modulo è quello incaricato del parsing dell'output del candump, ovvero dell'output del metodo read_frame della classe CAN_Sniffer e del parsing di un intero file di log. Le funzioni che compongono il modulo di Parsing_log ritornano un'istanza della struttura Frame o un vettore di Frames, dove la struttura frames è definita nel seguente modo.

```
struct Frame{
        double timestamp;
        std::string id;
        std::string payload;
};
```

Figura 7.10: struct Frame

- **Parsing_ini**: Questo componente provvede al parsing dei file di configurazione. Ogni ini_file comprende una sola sezione Suite che a sua volta è composta da una o più sezioni test, internamente a questo modulo sono presenti le rispettive funzioni che settano gli attributi delle classi Test e Suite.

```
void parse_test( .... ){

parse_test_details(file,test_type,name,description,num_of_rep,until);

if ( substr(test_type,"detect_and_act") ){

    parse_detect_and_act(file,id,check_exp,period,num_of_mex,type,tol,timeout);
    parse_commands(file,d,cmd,cmd_arg,else_d,else_cmd,else_cmd_arg);

    ....

}
if ( substr(test_type,"act_detect_act") ) {

    parse_detect_and_act(file,id,check_exp,period,num_of_mex,type,tol,timeout);
    parse_commands(file,d,cmd,cmd_arg,else_d,else_cmd,else_cmd_arg);
    parse_traffic(file,traffic_description,traffic);

    ....

}
```

Figura 7.11: parse_test

Ogni test si riferisce ad una categoria, ogni categoria è caratterizzata da un template standard, ovvero test appartenenti alla stessa categoria presentano la stessa sintassi nel file di configurazione, ovvero i campi presenti ed il loro ordine rimane invariato. Dopo aver determinato la categoria del test nella corrispettiva funzione di parsing, si procede a richiamare le funzioni specifiche per ogni campo.
La funzione parse_test riceve come argomenti una reference per ogni attributo della classe test.

7.6 Test

La funzione *parse_test* viene invocata quando una nuova istanza della classe Test viene creata. Il costruttore prende come argomenti una reference ad un file stream, facente riferimento ad un file di configurazione già aperto, successivamente l'ifstream dell'ini file e gli attributi del test vengono passati per reference al modulo di Parsing che si occuperà di settare opportunamente questi attributi.

```
Test::Test(std::ifstream &file) : file(file){

parse_test(...);

}
```

Figura 7.12: costruttore classe test

Al termine del parsing il test può essere eseguito tramite il metodo pubblico *start_test*:

```
void Test::start_test(){
    if ( this->type == "detect_and_act" ){
        detect_and_act();
    }
    if ( this->type == "act_detect_act" ) {
        act_detect_act();
    }
    if ( this->type == "detect_group" ) {
        detect_group();
    }
    if ( this->type == "act_detect_group" ) {
        act_detect_group();
    }
}
```

Figura 7.13: metodo start_test

In base alla categoria del test viene invocato il relativo metodo, oltre che dalla stessa sintassi nel file di configurazione, ogni categoria è caratterizzata da un template standard anche per il workflow.
Prima di analizzare queste categorie, soffermiamoci prima sull'implementazione dei comandi più importanti, descritti nella sezione 6.3.6.
CAN_Record_start:

```
void Test::CAN_record_start(const std::string &record_name, unsigned int time){

    std::vector<std::string> args;
    args.push_back("./candump");
    args.push_back("vcan0");
    args.push_back("-l");
    int PID = candump(args, ASYNC);
    candump_pid.store(PID);
    std::this_thread::sleep_for (std::chrono::milliseconds(time));

    CAN_record_stop(record_name);

}
```

Figura 7.14: CAN_record_start

CAN_record_stop:

```
void Test::CAN_record_stop(const std::string &record_name){

    if ( !stop_record.load() ){

        kill(candump_pid.load(),SIGTERM);
        std::string oldname = get_last_log(candump_pid.load());
        rename(oldname.c_str(), std::string("./records/" + record_name).c_str() );

    }

    stop_record.store(!stop_record.load());

}
```

Figura 7.15: CAN_record_stop

CAN_inject_from_trace:

```
void Test::CAN_inject_from_trace(const std::string &record_name){


    if ( !check_file_existence(record_name) ){
        std::cout << "File doesn't exist!" << std::endl;
        return;
    }


    std::vector<std::string> args;
    args.push_back("./canplayer");
    args.push_back("-I");
    args.push_back(record_name);
    canplayer(args);

}
```

Figura 7.16: CAN_inject_from_trace

```
void Test::CAN_rise_alert(const std::string &mex){

notify_init("Sample");
NotifyNotification* n = notify_notification_new ("Warning", mex.c_str(), 0);
notify_notification_set_timeout(n, 10000);
notify_notification_show(n, NULL);

}
```

Figura 7.17: CAN_rise_alert

Per quanto concerne alle categorie, focalizziamoci unicamente su quelle in cui la fase di detect riguarda un singolo frame, in quanto l'analisi del campo match_type nel caso in cui si debbano considerare i timing (*strict* o *simpe*) è piuttosto lungo.

```
bool Test::detect_and_act(){

bool res = true,check_res;
std::string row,operators,match_type;
std::vector<bool> cr;
std::vector<double> tss;
Frame fr;
CAN_Sniffer cs{};
std::vector<Check_field> operands = parse_check_intervals(this->check_exp);
operators = std::move(this->check_exp);
int num_of_detections = ( this->period == 0 ? 1 : this->number_of_messages);

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

while ( true ){

   if ( this->timeout != 0 ){

      auto t_end = std::chrono::high_resolution_clock::now();
      double elapsed_time_ms = std::chrono::duration<double,
std::milli>(t_end-t_start).count();
      if ( (double) this->timeout < elapsed_time_ms ){
         res = false;
         break;
      }
   }

   row = cs.read_frame();
   fr = parse_log_line(row);

   if ( fr.id == this->id) {

      bool result = check_boolean_expression(operands, operators, fr.payload);
      cr.push_back( result );
      tss.push_back(fr.timestamp);


      if ( substr(this->until,"S") && result) break;
      if ( substr(this->until,"F") && !result) break;
      if ( substr(this->until,"n_a") && result) break;

   }
}

int delay = ( res? this->cmd_delay : this->else_cmd_delay);
std::string command = ( res ? this->cmd : this->else_cmd);
std::string parameters = ( res ? this->cmd_params : this->else_cmd_params );
auto traffic_param = ( res ? this->traffic_cmd : this->traffic_else_cmd );

std::this_thread::sleep_for (std::chrono::milliseconds(delay));

if ( substr(command,"CAN_record_start") ){

   std::vector<std::string> params = split_string(this->cmd_params,", ");
   replace_all_occurences(params[0]," ", "");
   CAN_record_start(params[0],cast_string_to_int(params[1]));
```

```
}
if ( substr(command,"CAN_record_stop") ){

   CAN_record_stop(parameters);
}
if ( substr(command,"CAN_rise_alert") ){

   CAN_rise_alert(parameters);
}
if ( substr(command,"CAN_inject_start") ){

   CAN_inject_start(traffic_param);
}
if ( substr(command,"CAN_inject_stop") ){

   CAN_inject_stop();
}
if ( substr(command,"CAN_inject_from_trace") ){

   CAN_inject_from_trace(parameters);
}

return res;
}
```

Prima di tutto si applica la funzione *parse_check_intervals*, la funzione riceve in input una stringa e ritorna un vettore di check_field, ovvero una struttura così definita:

```
struct Check_field{
    int starting_bit;
    int length;
    int endianness;
    int value;
    int type;
};
```

Figura 7.18: struttura Check_field

Nella stringa operators sono presenti tutti gli operatori logici AND o OR che vanno concatenati al check field, successivamente si memorizza il tempo di inizio e parte la fase di *detect* in un ciclo infinito. Ad ogni ciclo si verifica se il timeout è scaduto oppure no, nel caso in cui sia scaduto si setta una variabile booleana a false ed un *break* porta all'uscita dal ciclo infinito. Ad ogni iterazione si richiama il metodo *read_frame* della classe CAN_Sniffer e successivamente si parsa il risultato per ottenere la tupla: timestamp, id, payload. Se l'id contenuto nel campo *msg_id* del file di configurazione (ovvero l'attributo id della classe) è uguale all'id del frame ricevuto allora è necessario verificare se il campo check sia soddisfatto. Si invoca la funzione *check_boolean_expression* che ritorna il risultato del check applicato al payload del frame ricevuto, in base al valore del campo until si fanno alcuni controlli che potrebbero portare alla terminazione della fase di detect con conseguente terminazione del ciclo *while(true)*. La variabile *res* conterrà il valore di ritorno del test, in base alla terminazione con successo o fallimento del test si identifica il corrispettivo comando da lanciare che viene successivamente invocato.

7.7 Suite

La classe Suite è caratterizzata dai seguenti attributi privati: un vector di test, una funzione callback, un nome, una descrizione ed il path del file di configurazione.

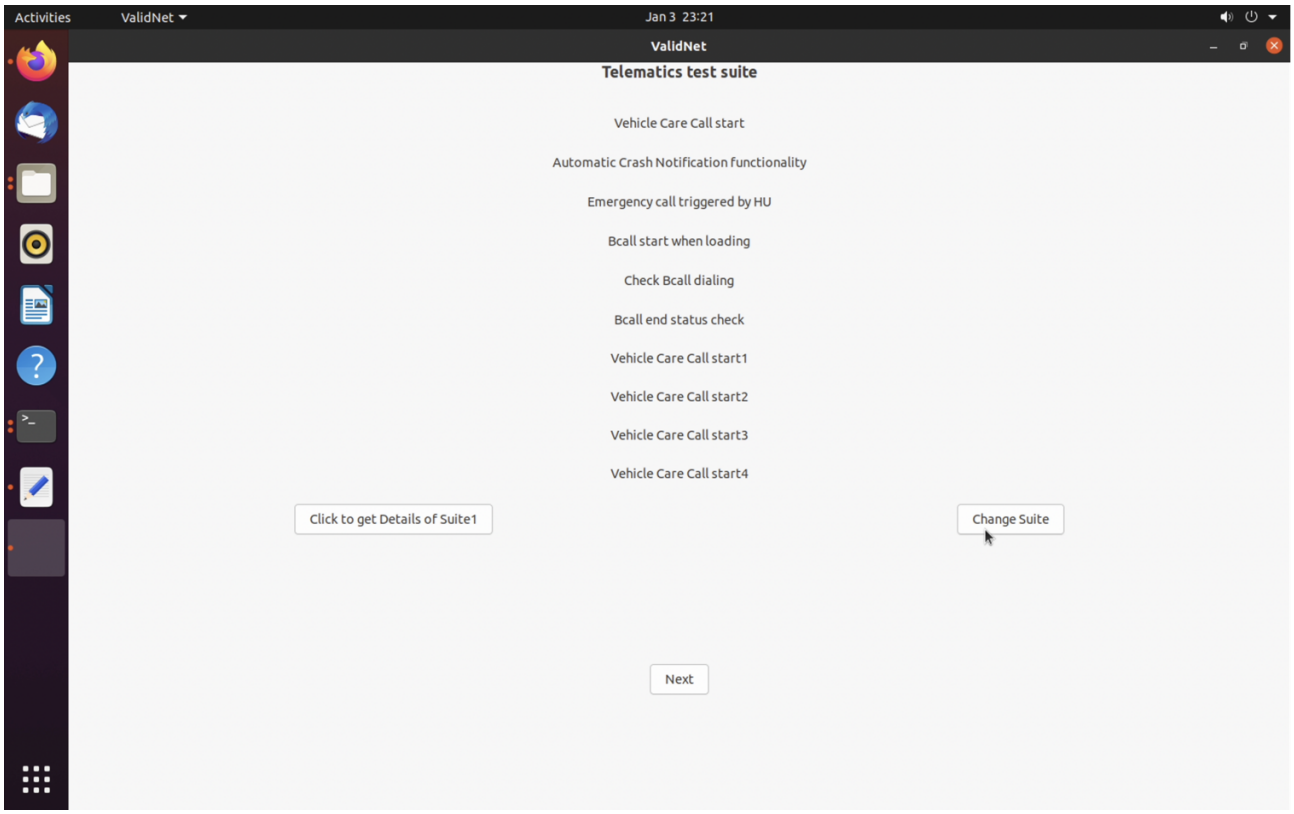

Figura 7.19: Scelta del file di configurazione

Inizialmente, quando viene create una istanza della classe Suite, tutti gli attributi sono vuoti in quanto essendoci più di un file di configurazione dentro la cartella ini, l'utente deve prima selezionare uno di questi attraverso l'interfaccia grafica. A seguito della selezione è possibile settare tutti gli attributi:

```
void Suite::fill_fields(const std::string &ini_file){

std::string line;
this->ini_file = ini_file;
std::ifstream file (ini_file);

set_num_of_tests(file);
parse_suite_details(file,this->name,this->description);

for ( int i = 0 ; i < this->number_of_tests ; i++ )

    tests.push_back( std::unique_ptr<Test> ( new Test{file} ) );


file.close();

}
```

Figura 7.20: metodo fill_fields

Una volta aperto un file stream, viene fatta una prima lettura per ottenere il numero di test contenuti della suite. A seguire iterativamente un nuovo test viene inserito dentro al vettore, passando una reference al file stream, ogni test creato invocherà la funzione parse_test che andrà ad elaborare una sezione di test successiva. La Suite contiene un vector di puntatori a test e non un semplice vector di test a causa nella non copiabilità degli std::atomic (definiti in Test).

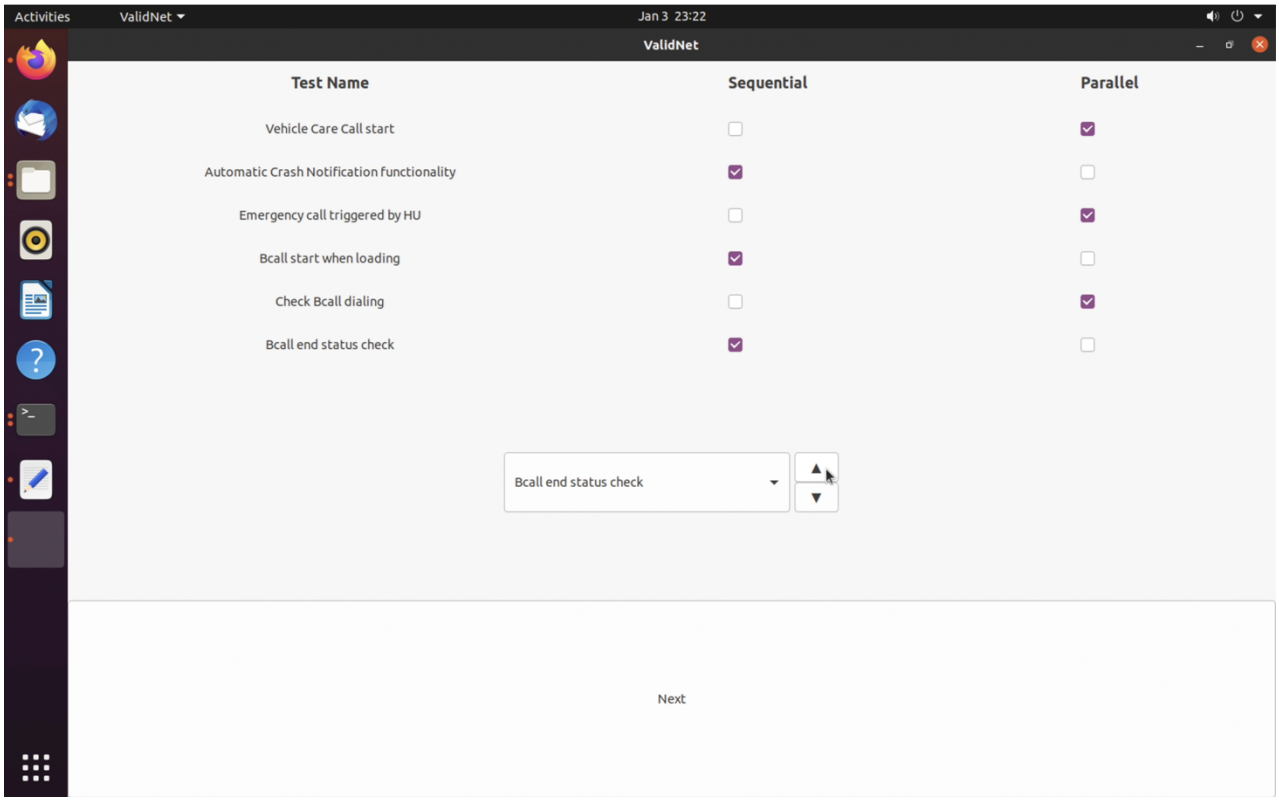

Figura 7.21: Test selection

A seguito della scelta del file di configurazione, l'utente deve selezione un gruppo di test da eseguire, per ogni test è necessario specificare la modalità di esecuzione (sequenziale o parallela) ed eventualmente modificare l'ordine dei test, queste informazioni vengono passate dalla GUI alla Business Logic. Dall'interfaccia grafica viene invocato il metodo della Suite:

```
void Suite::execute_tests(std::vector<int> running_mode){

    std::vector<std::thread> threads;

    std::string file_name = "./log/log_" + currentDateTime();
    std::ofstream file (file_name);
    file.close();

    for ( int i = 0 ; i < running_mode.size() ; i++ ){

        if (running_mode[i] == 2) {
            threads.push_back(std::thread([&,i]() {

                bool result = tests.at(i)->start_test(i);
                invoke_gui_callback(result,i);
            }));
        }
```

```
        else if (running_mode[i] == 1){
            for ( int j = 0 ; j < threads.size() ; j++ )
                if ( threads.at(j).joinable() )
                    threads.at(j).join();

       bool res = tests.at(i)->start_test(i);
       invoke_gui_callback(res,i);

            threads.clear();
        }
    }

    for ( int i = 0 ; i < threads.size() ; i++ )
        if ( threads.at(i).joinable() )
            threads.at(i).join();


}
```

Viene inizialmente creato un file di log, come supporto in fase di debugging. A seguire in base al i-esimo valore di running_mode il test viene lanciato in parallelo o in modo sequenziale. È possibile modificare l'ordinamento dei test in modo tale da modificare i gruppi da eseguire in parallelo, nel momento in cui avviene una modifica dell'ordine si richiama il metodo *swap_test*.

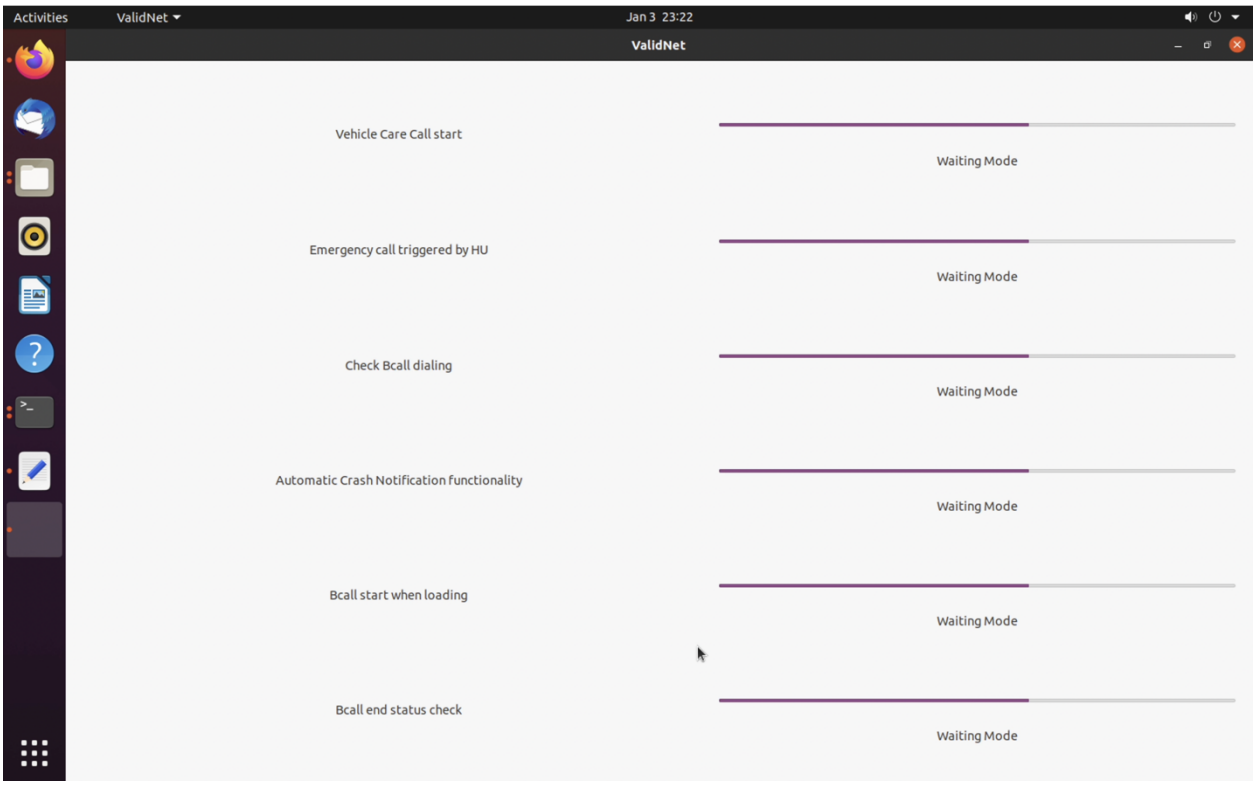

Figura 7.22: Test in esecuzione

Nel momento in cui sono in esecuzione, l'interfaccia grafica mostrerà delle barre d'avanzamento per ogni test scelto dall'utente della schermata precedente. Nel momento in cui il test termina la propria esecuzione, ovvero nel momento in cui il metodo *start_test* termina la sua esecuzione viene successivamente richiamata una callback per informare l'interfaccia grafica della terminazione di uno specifico test. La *invoke_gui_callback* riceve come parametri il valore di ritorno del test ed il suo identificativo.

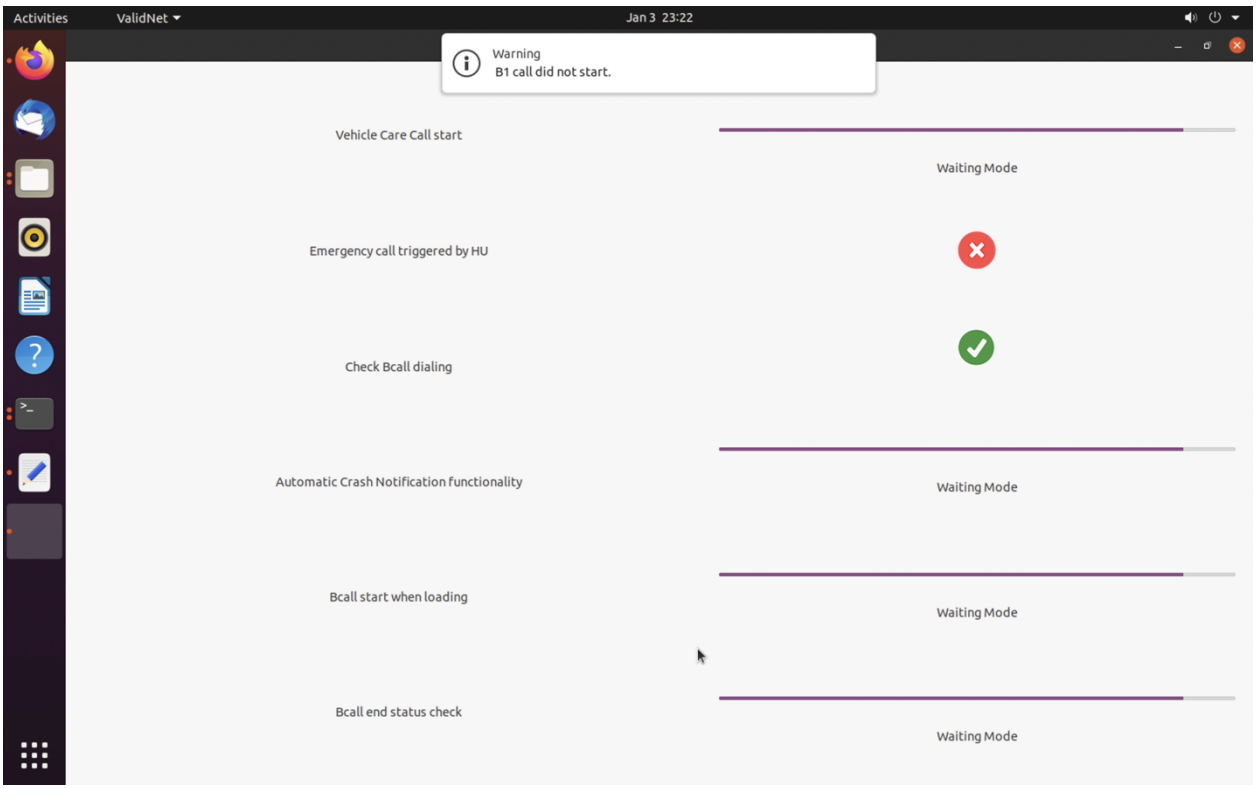

Figura 7.23: Terminazione dei test

La funzione si occuperà di sostituire la barra di avanzamento con un'icona, una X in caso di fallimento ed una V in caso di successo, per informare l'utente dell'esito del test appena effettuato.

7.8 GUI

Uno degli aspetti più importanti per implementare una buona interfaccia grafica è quella di garantire dei tempi di risposta il più bassi possibile, l'interazione tra l'utente e la GUI deve essere fluida ed estremamente reattiva agli input esterni forniti dall'utilizzatore. Per queste ragioni, è estremamente importante eseguire quante più azioni possibili all'inizio. La creazione di finestre in GTK richiede un certo tempo, specialmente se le finestre in questione sono formate da molti elementi, come in questo caso. A causa di ciò, le finestre sono create all'avvio del programma e non vengono mai distrutte, quando il flusso di esecuzione avanza, nel momento in cui una finestra deve essere sostituita da un'altra, la prima viene nascosta e la seconda viene mostrata. Il contenuto della finestra è modificato in relaziona all'interazione con l'utente.

```
GUI::GUI(Suite *suite,Malicious_host *mh) : suite(suite),mh(mh),suite_num(0){

    get_distribution(OS);

    this->GUI_init();
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "ValidNet");
    OS == "raspberrypi" ? gtk_window_fullscreen(GTK_WINDOW(window)) :
gtk_window_maximize(GTK_WINDOW(window));

    main_box = gtk_hbox_new (FALSE, 5);
    gtk_container_add (GTK_CONTAINER (window), main_box);
```

```
    this->presentation_screen();
    this->boxes_screen();
    this->black_box_screen();
    this->suites_selection_screen();


    g_signal_connect( window,"delete-event",G_CALLBACK(gtk_main_quit),NULL);

    gtk_widget_show_all(window);
    gtk_widget_hide(boxes_hbox);
    gtk_widget_hide(black_hbox);
    gtk_widget_hide(suites_screen_box);

    gtk_main();

}
```

Nel costruttore la classe GUI, dopo aver creato la finestra ed aver inserito un titolo, invoca i relativi metodi per la creazione di ogni schermata, ognuna di queste viene nascosta ad eccezione della finestra iniziale ovvero quella di presentazione:

Figura 7.24: Schermata di presentazione

In cui è presente il logo dell'azienda. Dopo due secondi, la schermata iniziale verrà sostituita dopo l'invocazione di questa funzione lambda:

```
auto lambda_function = +[](GtkApplication *application, gpointer user_data)
{
    GUI *tmp_gui = static_cast<GUI*>(user_data);
    gtk_widget_hide(tmp_gui->presentation_hbox);
    gtk_widget_show(tmp_gui->boxes_hbox);

};
```

A seguire comparirà una nuova schermata:

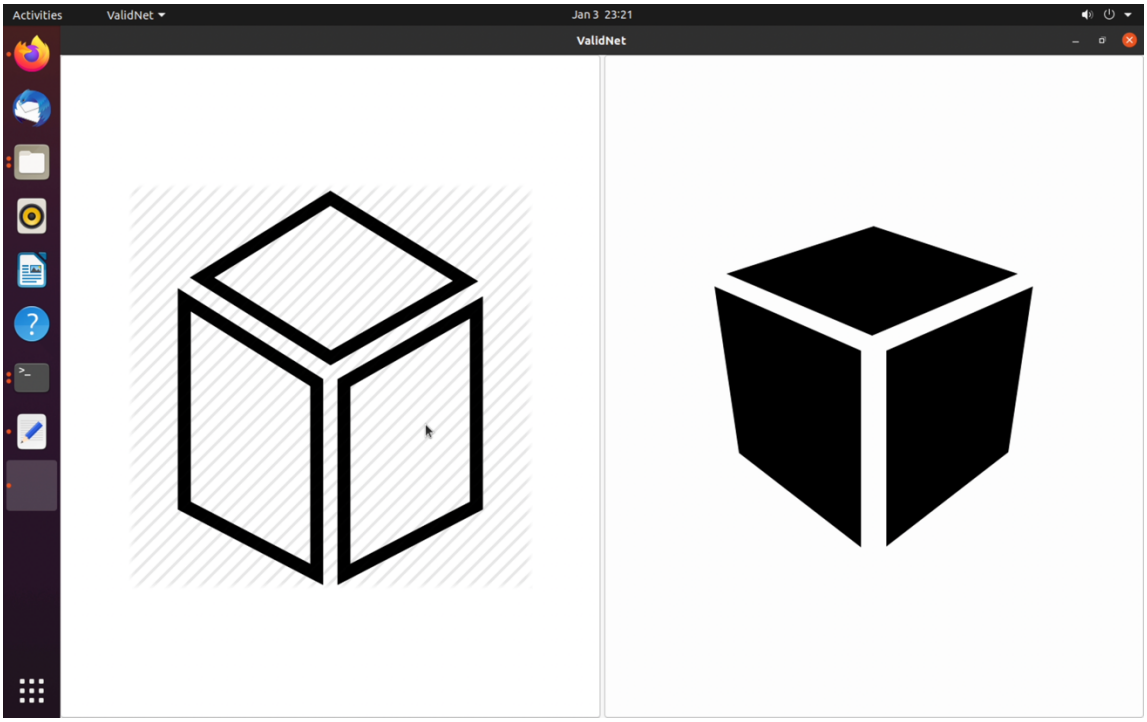

Figura 7.25: Seconda schermata

In cui l'utente deve scegliere quale modulo utilizzare: WhiteBox o BlackBox.
La modalità di funzionamento di WhiteBox è stata già analizzata, qualora l'utente dovesse cliccare sul pulsante di sinistra, comparirebbe la schermata di selezione del file di configurazione, nel caso in cui si dovesse scegliesse BlackBox, il modulo effettuerà in automatico tutti gli attacchi citati in precedenza in maniera sequenziale, effettuando in primo luogo il rilevamento della frequenza di trasmissione e poi successivamente quello del protocollo di rete.

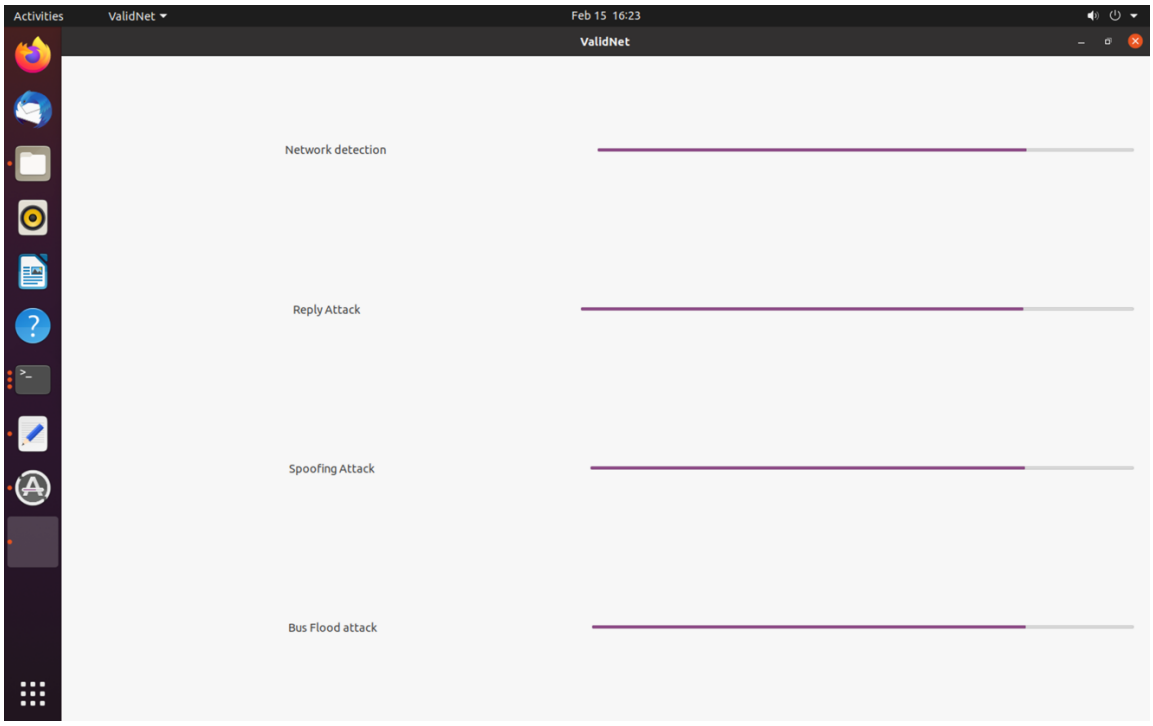

Figura 7.26: Blackbox in azione

La gestione di più thread che operano sull'interfaccia grafica per modificarne l'aspetto risulta un aspetto abbastanza critico in GTK. La gestione delle callback associate ad esempio alla pressione di un pulsante, alla chiusura di una finestra o sulla spunta di una casella sono dei particolari eventi che vengono gestiti direttamente dalla *gtk_main(),* questa funzione di GTK è bloccante e definisce al suo interno un ciclo infinito in cui il thread principale tramite polling verifica se è avvenuto un determinato evento ed eventualmente invoca la rispettiva callback, prima di fare questo è necessario acquisire uno specifico mutex, in precedenza alla modifica dell'interfaccia grafica ed è necessario rilasciare il mutex al termine della modifica per evitare una chiusura inaspettata del programma.
L'acquisizione/Rilascio di questo mutex viene fatto in automatico nel gtk_main(), mentre è necessario specificarlo nel caso in cui si operi mediante thread secondari, ciò avviene per esempio durante l'avanzamento delle barre di progressione:

```
refresh_progress_bars = std::move( std::thread ([&](){
    gdouble fraction = 0.0;

    while(true){

        std::this_thread::sleep_for (std::chrono::milliseconds(250));
        fraction = fraction + 0.1;

        gdk_threads_enter();

        if ( fraction > 1.0 ){
            fraction = 0.0;
            for ( int i = 0 ; i < this->tests.size() ; i++ )
                gtk_progress_bar_set_fraction( GTK_PROGRESS_BAR(
progress_bars[i] ), fraction );

        }
        for ( int i = 0 ; i < this->tests.size() ; i++ )
            gtk_progress_bar_set_fraction( GTK_PROGRESS_BAR( progress_bars[i]
),fraction );

        gdk_threads_leave();

    }
}));

refresh_progress_bars.detach();
```

Il thread secondario si addormenta per 250 ms, al suo risveglio incrementa la variabile fraction di 0.1 ed aggiorna il livello di tutte le barre spaziatrici tramite la funzione gtk_progress_bar_set_fraction, prima della modifica è necessario invocare la *gdk_threads_enter()* che prende possesso del mutex che consente la modifica di un qualsiasi GtkWidget. La *gdk_threads_leave()* viene invocata prima di riaddormentarsi. Un altro caso in cui avviene una modifica da thread secondari, è nel momento in cui un test termina la propria esecuzione ed invoca la callback associata:

```
void GUI::refresh_test(bool result, int id){

    gdk_threads_enter();
    gtk_widget_hide(progress_bars[id]);
    gtk_widget_hide(tests_state[id]);
    gtk_widget_show(result ? successes_img[id] : fails_img[id]);
    gtk_widget_hide(result ? fails_img[id] : successes_img[id]);
    gtk_widget_set_valign(result ? successes_img[id] : fails_img[id],
GTK_ALIGN_CENTER);
    gdk_threads_leave();
}
```

In questo caso in base al valore di ritorno del test, comparirà la corrispettiva immagine al posto della barra di progressione.

7.9 Malicious Host:

Il modulo di BlackBox è definito dalla classe Malicoius_host la cui definizione è la seguente:

```
class Malicious_host{

private:

public:
    Malicious_host();

    int detect_bitrate();
    std::string detect_protocol(const std::string &filename);
    bool rate_check(const std::string &filename);
    void reply_attack();
    void bus_flood_attack();
    void spoofing_attack();

};
```

Per brevità riportiamo solo l'attacco di Bus Flood Attack:

```
void Malicious_host::bus_flood_attack(){

    std::vector<std::string> args;
    std::string row, lowest_id;
    Frame fr;
    CAN_Sniffer cs{};

    for ( int i = 0 ; i < 1000 ; i++ ){

        row = cs.read_frame();
        fr = parse_log_line(row);

        if ( i == 0 )
            lowest_id = fr.id;

        if ( lowest_id < fr.id )
            lowest_id = fr.id;
    }

    args.push_back("./cangen");
    args.push_back("can0");
    args.push_back("-I");
    args.push_back(lowest_id);
    args.push_back("-g");
    args.push_back("0.01");
    args.push_back("-n");
    args.push_back("500");
    cangen(args);

}
```

Vengono analizzati i primi 1000 frames inviati sulla rete, l'host malevolo elabora ogni trama e memorizza l'id ricevuto con priorità maggiore, successivamente effettua un *cangen* ed invia 500 frames aventi id uguale a quello rilevato.

7.10 Meccanismi di comunicazione:

```
int main() {

    Suite suite{};
    Malicious_host mh{};
    GUI gui{&suite, &mh};

    return 0;
}
```

La comunicazione da Interfaccia grafica e Business Logic è immediata in quanto la classe GUI riceve un riferimento sia alla classe Suite, che a sua volta permette di accedere ad un qualsiasi test sia alla classe Malicious Host, per cui è possibile invocare i metodi pubblici di queste classi direttamente dalla GUI in base all'interazione con l'utente. Il main risulta completamente scarno, tutti i dettagli implementativi sono nascosti internamente alle quattro classi Suite, Test, Malicious_host e GUI in questo modo si facilita il rilevamento degli errori e si aumenta la modularità del sistema. La comunicazione tra Business logics ed interfaccia grafica risulta più complessa, nello specifico le classi Suite e Malicious_host non hanno un riferimento alla classe GUI, per evitare il problema delle dipendenze cicliche particolarmente ostico in fase di compilazione. Per ovviare a questa complicanza è stato utilizzato il meccanismo delle callback, usato ad esempio nel momento in cui un test terminato deve informare la GUI del suo valore di ritorno per sostituire le barre di avanzamento con l'esito del test. Riportiamo in modo estremamente sintetico il meccanismo di binding della callback che si è utilizzato:

```
class Business_logic{
private:
    std::function<void(int,std::string)> f;
public:
    Business_logic(){}
    void set_callback(std::function<void()> func){
        f =func;
    }
    void invoke_callback(){
        f();
    }
};
class GUI{
private:
    Business_logic *bl;
public:
    GUI(Business_logic *bl) : bl(bl){}
    void change_GUI(){

        .....
    }

    void bind_callback(){
        auto f  = std::bind(&B::method,this);
        bl->set_callback(f);
    }
};
```

Conclusioni e risultati ottenuti

8.1 WhiteBox:

Sono due i componenti che sono stato oggetto di analisi da parte del modulo di Whitebox: il primo è un Raspberry Pi dotato dello stesso hardware del Raspberry Pi principale, descritto nel dettaglio nel capitolo 3. Il secondo è un simulatore della rete di un'automobile, ovvero una board dotata di diversi pulsanti e spie azionabili come, ad esempio, un pulsante per l'accensione del motore, uno per l'inserimento della chiave, uno per l'accensione delle luci o della radio. La pressione di uno di questi bottoni porta il dispositivo ad inviare delle specifiche trame sulla rete che variano in base al tipo di pulsante che è stato premuto.
Il collegamento è stato realizzato tramite una connessione diretta mediante cavo DB-9 o OBD2. In entrambi i casi è stato necessario configurare l'interfaccia di rete del dispositivo testante in modo opportuno per consentire la comunicazione: sul Raspberry Pi testato sono stati definiti parametri di rete differenti, mentre il simulatore di rete opera ad un bitrate di 250 Kbit/sec utilizzando il protocollo CAN. Per il testing di quest'ultimo componente è stato necessario conoscere il rispettivo file DBC per poter generare il file di configurazione nel modo corretto.

Figura 8.1: Testing Whitebox

Una volta interconnessi e dopo aver definito l'interfaccia di rete fisica, i due dispositivi sono in grado di comunicare tra loro ed è quindi possibile eseguire l'applicativo.

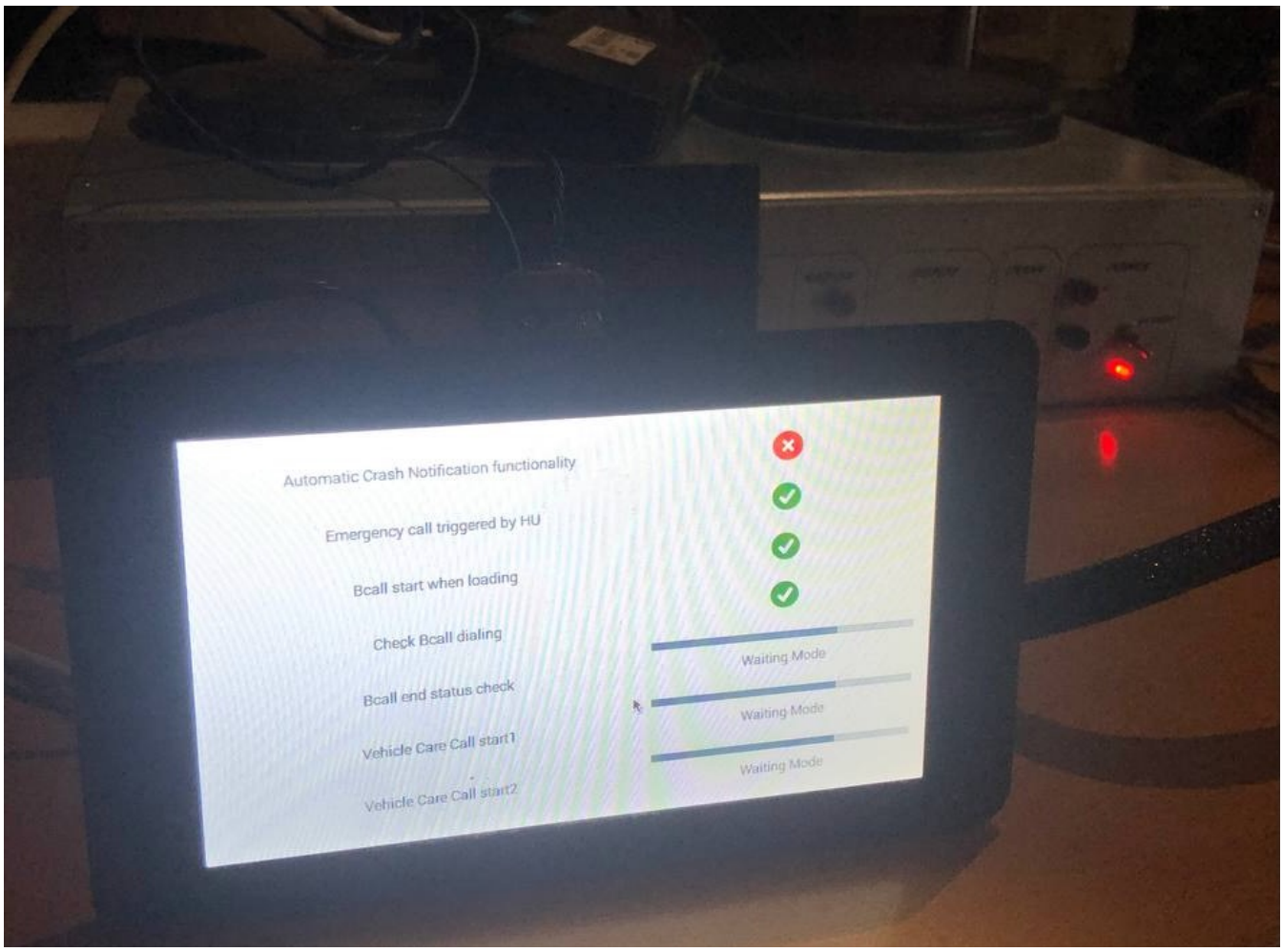

Figura 8.2: Testing Whitebox

I test verranno eseguiti in modo sequenziale oppure in parallelo, in uno specifico ordine deciso dall'utente ed alla loro terminazione verrà riportato l'esito di ogni test.
Per brevità focalizziamo l'attenzione sui primi due test eseguiti, essi sono definiti nel file di configurazione nel seguente modo:

```
[TEST_2]

test_type = act_detect_act
test_name = Automatic Crash Notification functionality
test_description = Sends a trigger for the automatic crash notification and
checks that the related emergency calls is starting in 100 ms timeframe.
loop = 0, n_a
traffic_data_for_initial_send = TRAFFIC_2

msg_id = 0x5B2
check = [13, 3, 1, 2, equal_to, CallType ] & [16, 4, 1, 2, equal_to, CallSts ]
period = 100
period_comparison_type = 1, equal_to, 10

timeout = 100
command_delay_1 = 0
command_1 = CAN_rise_alert
command_parameter_1 = Emergency call started successfully.
```

```
else_command_delay_1 =
else_command_1 = CAN_rise_alert
else_command_parameter_1 = Emergency call did not start.

[TRAFFIC_2]

traffic_description = message to trigger the airbag opening
msg_id_1 = 0x15A
payload_1 = 0x60 0x80 0x00 0x00 0x00 0x00 0x00 0x00
time_1 = 0
count_1 = 1
period_1 = 0

[TEST_3]

test_type = act_detect_act
test_name = Emergency call triggered by HU
test_description = Sends a trigger for the emergency call (B1_call) acting as
a connected HU, and checks that the related emergency call is starting in 200
ms timeframe.
loop = 0, n_a
traffic_data_for_initial_send = TRAFFIC_3

msg_id = 0x5B2
check = [13, 3, 1, 3, equal_to, CallType ] & [16, 4, 1, 9, equal_to, CallSts ]
period = 100
period_comparison_type = 1, equal_to, 10

timeout = 500
command_delay_1 = 0
command_1 = CAN_rise_alert
command_parameter_1 = B1 call started successfully.

else_command_delay_1 =
else_command_1 = CAN_rise_alert
else_command_parameter_1 = B1 call did not start.

[TRAFFIC_3]
traffic_description = traffic to trigger an emergency call from the HU.
msg_id_1 = 0x5AE
payload_1 = 0x28 0x00 0x00 0x00 0x00 0x00 0x00 0x00
time_1 = 0
count_1 = 1
period_1 = 0
```

Il primo test invia il frame 15A#6080000000000000 e si mette successivamente in ascolto per 100 ms, nel caso in cui sia stata ricevuta una trama con identificativo 0x5B2 ed uno specifico payload allora il test sarà considerato superato e verrà generato un popup con il messaggio "*Emergency call started successfully*", in caso di fallimento verrà generato un popup ma con un messaggio differente. L'obiettivo del test, come riportato nel campo descrizione, è quello di inviare una notifica di crash sulla rete per poi verificare se la relativa procedura di chiamata di emergenza viene effettuata oppure no, nel nostro caso essa non avviene ed il test fallisce, in quanto il dispositivo testato non è in grado di effettuare la suddetta procedura.

Il test successivo opera in modo pressoché analogo invia il frame 5AE#2800000000000000 e si mette in attesa del frame con id 5B2 per 500 ms, anche in questo caso il valore di ritorno del test comporta la generazione di un messaggio di popup.

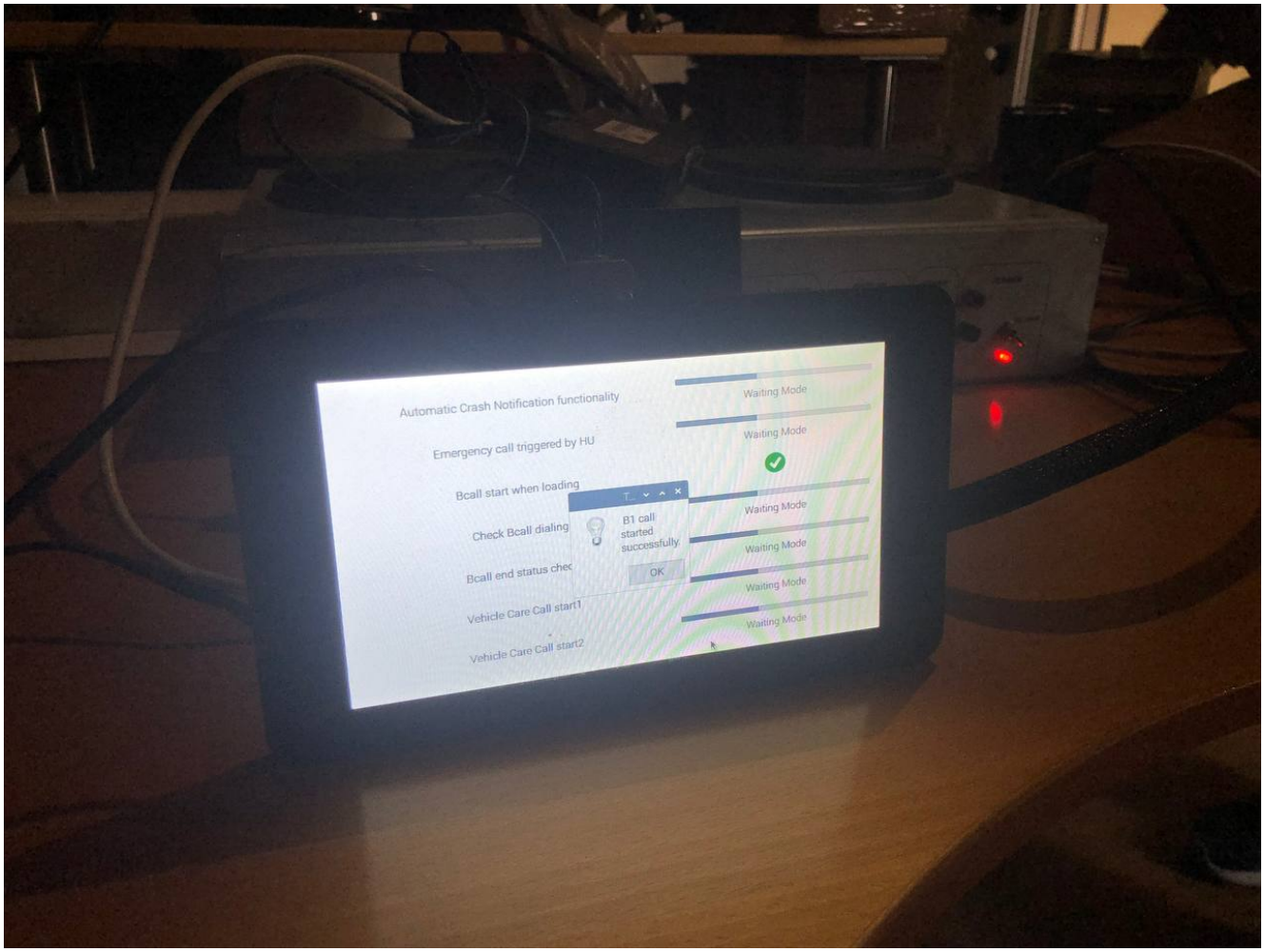

Figura 8.3: Messaggio di Popup generato

In questo caso il test termina con successo in quanto, viene rilevato prima dello scadere del time-out il frame 5B2#F0F390.

8.2 Conclusioni WhiteBox:

La validazione del dispositivo è andata a buon fine, la quasi totalità dei test effettuati sono terminati con i risultati previsti, è stato possibile prevedere e verificare il funzionamento atteso grazie alla conoscenza del file DBC e all'interazione con la board. In alcuni casi il risultato inaspettato di un test è stato causato da un leggero ritardo nei timing delle trame inviate o ricevute.

8.3 BlackBox:

Le funzionalità offerte dal modulo di BlackBox sono state provate sugli stessi dispositivi del modulo di WhiteBox. Nello specifico non è stato possibile osservare il funzionamento degli attacchi informatici in contesti realistici, per cui gli attacchi di Reply e di Spoofing, pur essendo stati implementati, non sono stato oggetto di analisi durante i test fisici.
Per quanto riguarda la fase di Network Detection, è stato possibile identificare correttamente, in tempi estremamente brevi, diverse configurazioni di rete. Il software è stato utilizzato per individuare configurazioni differenti definite sull'altro Raspberry Pi.

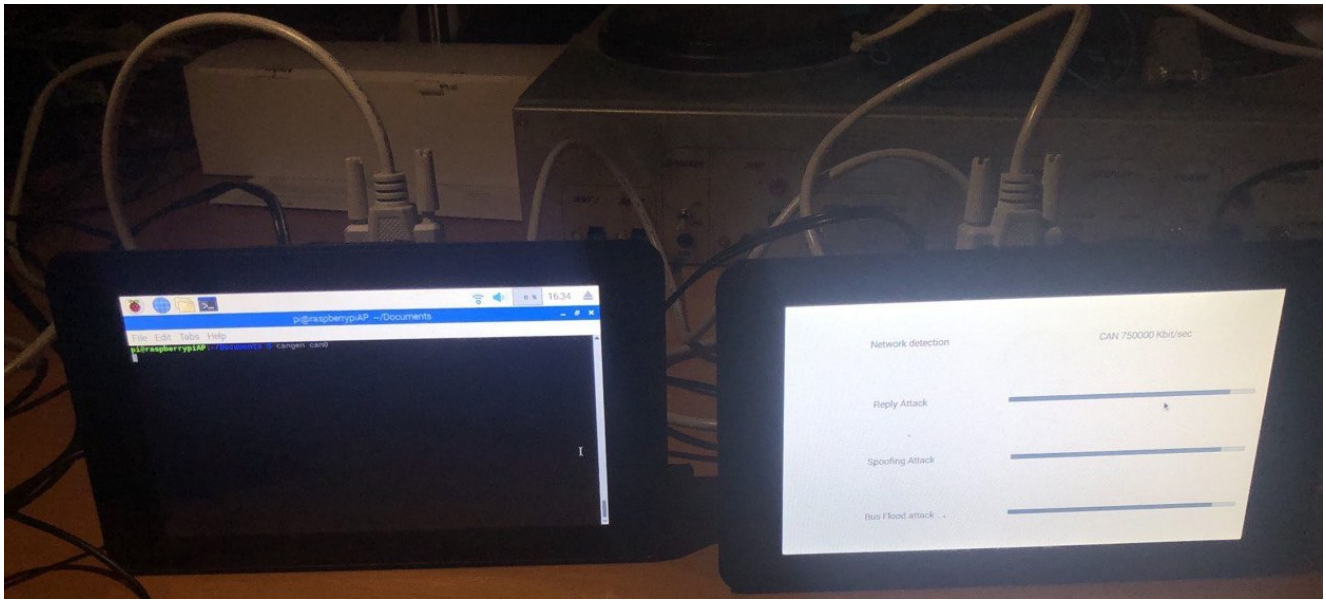

Figura 8.4: BlackBox Network Detection 1

In figura 8.4, il primo Raspberry Pi genera dei frames conformi al protocollo CAN ad un bitrate di 750 Kbit/sec, il modulo di BlackBox rileva correttamente entrambi i parametri.

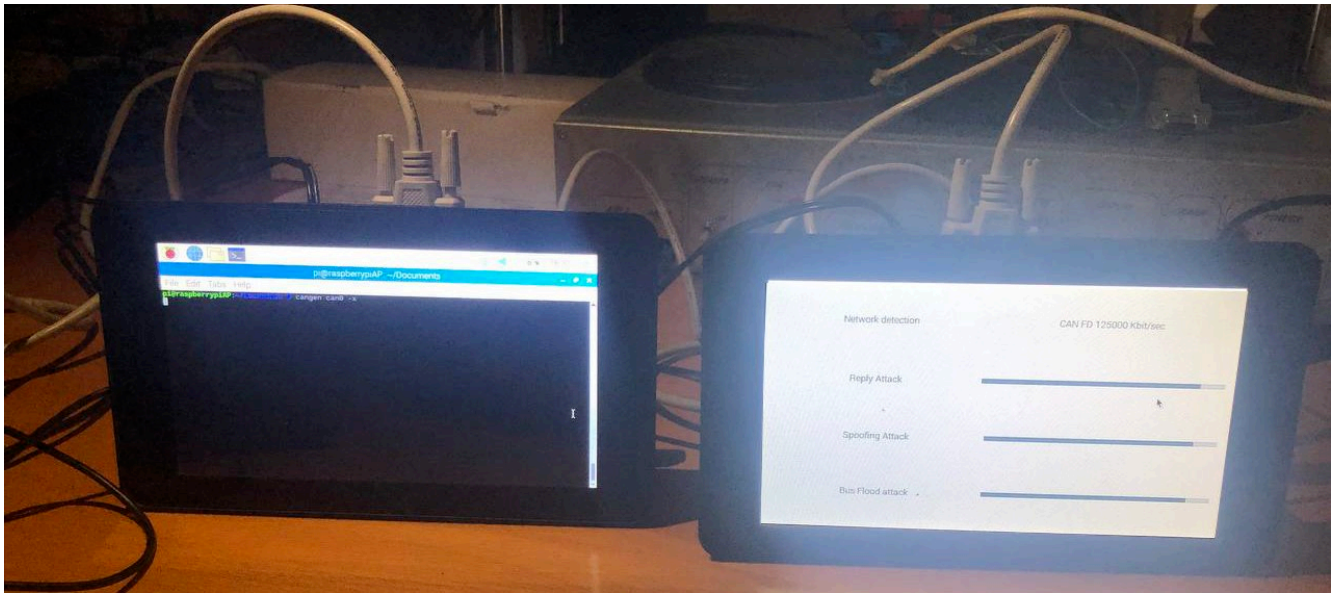

Figura 8.5: BlackBox Network Detection 2

In figura 8.5, il primo Raspberry genera invece delle trame relative al protocollo CAN FD ad un bitrate di 125 Kbit/sec, anche in questo caso la fase di Network Detection termina correttamente.
La fase di Network Detection è stata provata anche sul simulatore della rete automobilistica, fornendo anche in questo caso un risultato corretto, ovvero CAN 125 Kbit/sec come riportato in precedenza.
Per quanto concerne alla parte degli attacchi informatici riportiamo l'esito del Bus Flood Attack, effettuato ai danni del secondo Raspberry Pi.
Il Raspberry attaccato genera del traffico con periodicità massima ovvero di 0.15 ms, nel mentre il modulo di BlackBox avvia il Bus Flood Attack ed elabora i primi 1000 frames ricevuti, al termine di questa fase l'id ricavato è 0x014.
A seguire il modulo di Blackbox invierà in sequenza trame con l'identificativo trovato, nel frattempo il secondo Raspberry continuerà ad inviare frames e memorizzerà tutte le trame inviate e ricevute dentro un file di log; sono stati presi in esame i primi 10.000 frames inviati e ricevuti e l'analisi ha portato alle seguenti conclusioni:

- Sono state rilevate sequenze contigue di al più due trame con identificativo 0x014, per cui l'occupazione del canale causata dall'attacco risulta trascurabile, non creando effettivamente nessun tipo di disservizio.

- Su 10.000 trame, 5088 di queste hanno ID 0x014, il 50,88% ovvero poco più della metà.

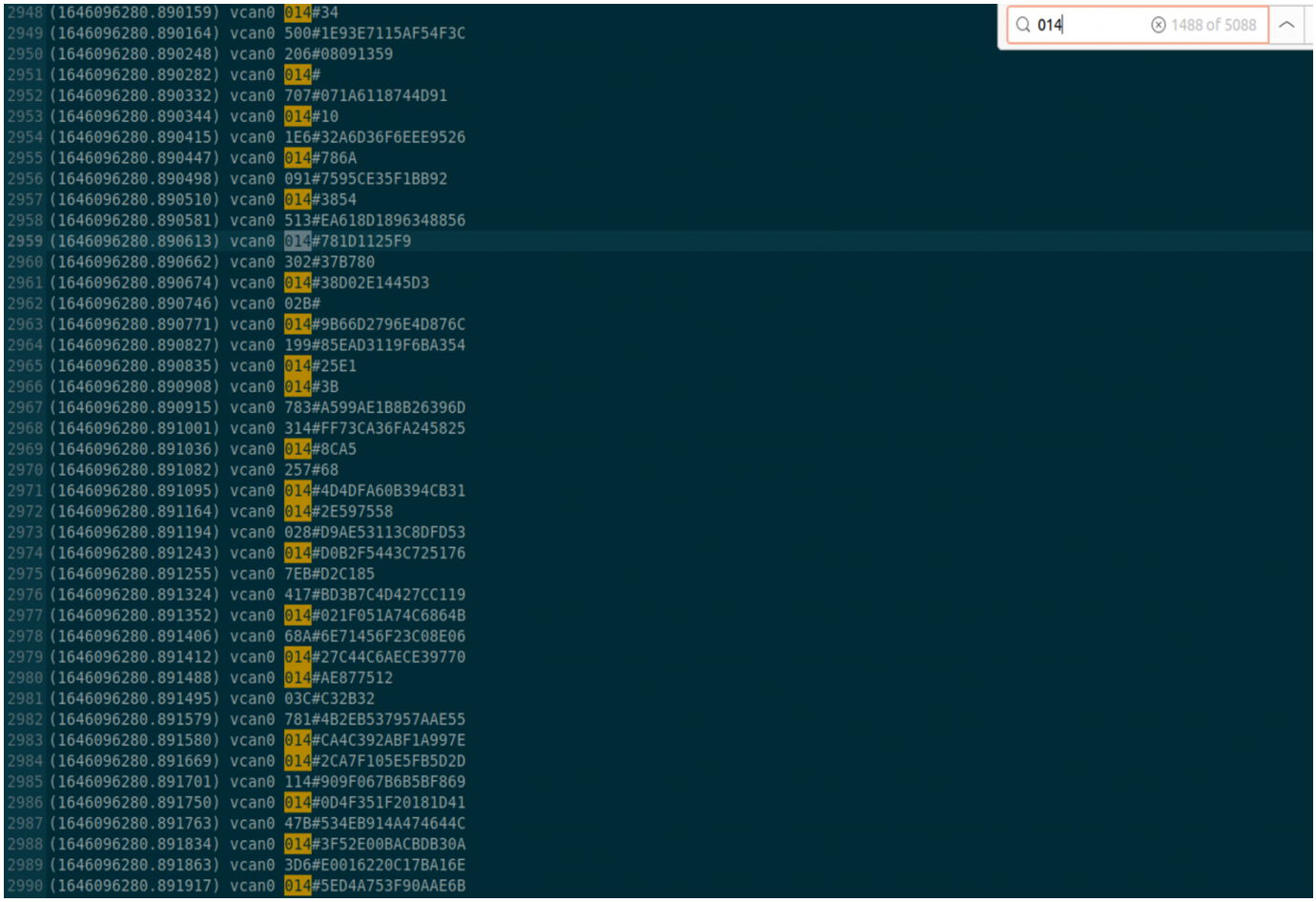

Figura 8.6: Risultato Bus Flood Attack

8.3 Conclusioni BlackBox:

La fase di Network Detection è sempre terminata con successo, riuscendo a scoprire ogni volta sia la frequenza di trasmissione che il protocollo di rete. L'attacco di Bus Flood non ha portato ad i risultati sperati: non si è mai manifestata una sequenza lunga di trame inviate dall'attaccante, capaci di rendere inutilizzabile la rete per un lungo periodo. Il numero di frames inviati dall'host attaccante risulta lievemente maggiore di quelli inviati dall'host attaccato, tuttavia ciò non porta a nessun disservizio complessivo della rete, per cui si può considerare l'attacco come fallito.

Documenti consultati

Bibliografia:

[1] Simon Long, Raspberry Pi Press *An Introduction to C and GUI Programming.*
[2] Marco di Natale, Haibo Zeng e Paolo Giusto *Understanding and Using the Controller Area Network Communication Protocol.*

Documenti:

[1] Ian Del Rio, *ValidNET: A modular CAN analyzer/manager with aims of security provisioning*.
[2] Daniele Gallinella*, CAN Controller Area Network.*
[3] CAN bus, *https://en.wikipedia.org/wiki/CAN_bus*
[4] CAN FD – The Basic Idea, *https://www.can-cia.org/can-knowledge/can/can-fd/*
[5] Raspberry Pi, *https://en.wikipedia.org/wiki/Raspberry_Pi*
[6] A Survey on CAN Bus Protocol: Attacks, Challenges, and Potential Solutions, Mehmet Bozdal, Mohammad Samie, Ian Jennions
[7] CAN Bus Security Attacks on CAN bus and their mitigations, CANIS Automotive Labs
[8] Cybersecurity attacks on CAN bus based vehicles: a review and open challenges, *emerald.com/insight/content/doi/10.1108/LHT-01-2021-0013/full/html*
[9] Automobile CAN Bus Network Security and Vulnerabilities, Shawn Hartzell and Christopher Stubel
[10] SocketCAN, *https://en.wikipedia.org/wiki/SocketCAN*
[11] SocketCAN, *https://www.kernel.org/doc/html/latest/networking/can.html*
[12] CANOpen, *https://it.wikipedia.org/wiki/CANopen*
[13] CANopen – The standardized embedded network, *https://www.can-cia.org/canopen/*
[14] DoS, *https://it.wikipedia.org/wiki/Denial_of_service*
[15] Controllo Congestione, *https://it.wikipedia.org/wiki/Controllo_della_congestione*
[16] Controller Area Network (CAN Bus) - Error Detection And Fault Confinement, *https://copperhilltech.com/blog/controller-area-network-can-bus-error-detection-and-fault-confinement*
[17] Berkeley sockets *https://en.wikipedia.org/wiki/Berkeley_sockets*
[18] API, *https://it.wikipedia.org/wiki/Application_programming_interface*
[19] Errno Values for UNIX-Type Functions, ibm.com/docs/en/i/7.1?topic=ssw_ibm_i_71/apis/unix14.htm
[20] Tipizzazione forte, https://it.wikipedia.org/wiki/Tipizzazione_forte
[21] Linguaggio compilato, https://it.wikipedia.org/wiki/Linguaggio_compilato
[22] Linguaggio Interpretato https://it.wikipedia.org/wiki/Interprete_(informatica)
[23] User Experience https://it.wikipedia.org/wiki/User_Experience
[24] GTK, https://it.wikipedia.org/wiki/GTK_(toolkit)
[25] GTK, https://www.gtk.org/
[26] Makefile, https://it.wikipedia.org/wiki/Make
[27] Makefile, https://opensource.com/article/18/8/what-how-makefile
[28] Cross Compiling, https://it.wikipedia.org/wiki/Cross-compilazione
[29] libnotify, https://wiki.archlinux.org/title/Desktop_notifications
[30] documentazione del software, https://it.wikipedia.org/wiki/Documentazione_del_software
[31] ordine dei byte, https://it.wikipedia.org/wiki/Ordine_dei_byte
[32] Autonegoziazione, https://en.wikipedia.org/wiki/Autonegotiation
[33] SAE J1939, https://en.wikipedia.org/wiki/SAE_J1939

[34] SAE J1939 Packet format, https://www.kvaser.com/about-can/higher-layer-protocols/j1939-introduction/#:~:text=J1939%20uses%20the%2029%2Dbit,protocol%20shown%20in%20Figure%201.&text=PDU%20stands%20for%20Protocol%20Data,set%20to%20zero%20in%20J1939.
[35] C++ 17, https://en.cppreference.com/w/cpp/17
[36] system, https://www.delftstack.com/it/howto/cpp/system-in-cpp/
[37] CAN Utils, https://github.com/linux-can/can-utils
[38] dup2, https://digilander.libero.it/uzappi/C/librerie/funzioni/dup.html

Immagini:

1.1 Architettura di una rete centralizzata 1
1.2 Architettura di una rete automobilistica basata su CAN Bus.....2
2.1 Modello ISO/OSI.....5
2.2 Rappresentazione modello ISO/OSI per un nodo di rete CAN.....6
2.3 Rappresentazione nodo di rete CAN 6
2.4 Rappresentazione CAN Bus a livello elettronico.....7
2.5 Tabella che mostra la lunghezza massima del bus CAN in base al bit/rate 7
2.6 Formato CAN Frame 2.0A 8
2.7 Filtraggio di una trama CAN 9
2.8 Rilevamento di errori di trasmissione tramite CRC 10
2.9 Trasmissione contemporanea sul bus 11
2.10 Trasmissione contemporanea sul bus 2.....12
2.11 Bit stuffing 13
2.12 Tabella che lega il bit/rate al tempo massimo di attesa a seguito di un guasto.....15
2.13 Trasmissione frame con CAN FD 16
2.14 Formato pacchetto SAE J1939.....16
2.15 Formato del pacchetto 17
2.16 J1939-17 Documentation 19
3.1 Raspberry Pi venduti negli anni.....20
3.2 Raspberry Pi 3.B+.....21
3.3 Raspberry Pi Touch Screen 21
3.4 Connessione tra Raspberry Pi e display LCD 19
3.5 PiCAN FD Board 20
4.1 CAN Open nel modello ISO/OSI 21
4.2 Trasmissione contemporanea …….....21
4.3 Ordine di invio delle trame 22
4.4 Adaptive Spoofing Attack.....25
4.5 Macchina a stati della gestione degli errori per un singolo nodo 26
4.6 Arbitration Doom Loop 26
4.7 Double Receive Attack 27
5.1 CAN Communication Layer 28
5.2 Programma scritto in Python 29
5.3 Programma scritto in C++.....31
5.4 Glade 33
5.5 Esempio di notifica 40
6.1 Output candump 41
6.2 Primo byte del campo ID di PDU J1939.....43
7.1 candump originale 44
7.2 candump modificato 60
7.3 implementazione cangen 65
7.4 invocazione cangen 65
7.5 template generico 66
7.6 costruttore classe CAN Sniffer 67
7.7 metodo read frame 67
7.8 Distruttore CAN Sniffer 68
7.9 invocazione metodo read frame 68
7.10 struct Frame 69
7.11 parse test 69

7.12 costruttore classe test 70
7.13 metodo start test 70
7.14 CAN record start 71
7.15 CAN record stop 71
7.16 CAN rise alert 72
7.17 CAN inject from trace 72
7.18 CAN rise alert 72
7.19 struttura Check Field 74
7.20 scelta del file di configurazione 75
7.21 metodo fill fields 75
7.22 Test selection 76
7.23 Test in esecuzione 77
7.24 Terminazione dei test 78
7.25 Schermata di presentazione 79
7.26 seconda schermata 80
7.27 Blackbox in azione 80
8.1 Testing WhiteBox 84
8.2 Testing WhiteBox 85
8.3 Messaggio di Popup generato 87
8.4 BlackBox Network Detection 88
8.5 BlackBox Network Detection 2....88
8.6 Risultato Bus Flood Attack 89