

Università degli Studi di Modena e Reggio Emilia

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea in Ingegneria Informatica – Sede di Mantova

Tool per la replica di dataset di frames CAN su socket virtuale rispettando tempistiche temporali

*Tool for the replication of datasets of CAN frames on virtual socket
respecting the timescales*

Relatore:

Prof. Luca Ferretti

Candidato:

Patrik Brighenti

Correlatore:

Prof. Dario Stabili

Matricola:

143838

Indice

	Introduzione	1
I.	Controller Area Network	3
1.	Introduzione	3
1.1	Standard e Versioni	3
1.2	Architettura della Rete	4
1.3	Trasmissione Dati	5
1.3.1	Bit Dominanti e Recessivi	5
1.3.2	Controllo dell'Arbitrato	6
1.3.3	Sincronizzazione	7
1.4	Frames	7
II.	Sicurezza	11
2.1	Caratteristiche	11
2.2	Attacchi	12
III.	Il Progetto	15
3	Introduzione	15
3.1	Setup Iniziale e Tools di Sviluppo	15
3.1.1	Specifiche hardware e Software	15
3.2	CAN-Utills	16
3.2.1	Canplayer e Candump	17
3.3	Codice	19
3.3.1	Librerie e Strutture Dati	19

3.4	Programmazione Real-time	21
3.4.1	Inizializzazione Strutture Dati	21
3.4.2	Gestione del Tempo	23
3.4.3	Stabilizzazione dello Stato della Socket	25
3.5	Struttura del progetto	27
3.5.1	Dataset	27
3.5.2	Script	28
IV.	Analisi dei Risultati	30
	Conclusione	34

Introduzione

La trasformazione digitale dell'industria automobilistica non è solo una questione produttiva. Nell'evoluzione dell'Automotive 4.0, tutto diventa smart, la vera sfida si chiama dunque integrazione. E riguarda sostenibilità, driver tecnologici, standard di comunicazione, mobilità elettrica, nuovi servizi al consumatore e ultime frontiere della guida autonoma. Il tutto per garantire sicurezza e offrire maggiore connettività.

Il rapporto tra industria automobilistica e quarta rivoluzione industriale è destinato a modellare una serie di innovazioni legate ad una sempre maggiore interconnettività tra gli autoveicoli.

Nella storia dell'industria automobilistica, l'innovazione ha sempre avuto un ruolo onnipresente e centrale nello sviluppo di nuovi modelli e macchinari dalla tecnologia all'avanguardia. Una ricca storia di trasformazioni definisce un settore in cui il cambiamento avviene più spesso che mai.

Il tema della digitalizzazione, nel settore automotive italiano, richiede una premessa. Parliamo infatti di un comparto industriale particolarmente complesso e fortemente integrato con le filiere dell'indotto. Basti pensare che il 75% dei componenti di un'autovettura viene realizzato da terzisti e successivamente assemblato dalle case automobilistiche.

Al contempo, il nostro mercato è vivace e ben rappresentato. Il territorio nazionale conta 5.500 imprese, 270.000 addetti (diretti e indiretti) e un fatturato di 105 miliardi di euro (fonte: Anfia, 2017). Siamo già al 6,2% del Pil, ma se allarghiamo i confini dalla filiera industriale, considerando anche tutti i servizi connessi alla mobilità (commercio e manutenzione, infrastrutture, autonoleggio, ecc.) il giro d'affari sfiora i 335 miliardi di euro.

Le automobili connesse e digitali sono ormai lo standard. Parliamo di veicoli che, oltre ad avere accesso a internet, dispongono di sensori e inviano/ricevono segnali. In sostanza, l'auto connessa percepisce la realtà circostante e interagisce con altre entità. La digitalizzazione che incrementa la sicurezza del passeggero è anche una leva determinante in fase di acquisto di un veicolo.

La guida autonoma, poi, apre nuove frontiere e riflessioni sul tema: dai sensori ai Big Data, dai Data Center Edge per l'elaborazione dei dati al cambio radicale nella gestione delle città. Le aziende del settore non possono dunque stare a guardare, devono farsi trovare pronte a "servire" un nuovo modo globale di vivere e viaggiare.

Questo scenario in rapidissima ascesa, composto da tecnologie in sviluppo che devono essere integrate all'interno del funzionamento delle autovetture ci costringe a valutare le possibili vulnerabilità che gli hacker potrebbero sfruttare.

Presentazione del progetto

Al fine di fornire ai ricercatori un tool in grado di aiutarli a studiare gli attacchi ed eventuali possibili soluzioni difensive, si è deciso di implementare un software in grado di replicare dei dataset di frames su di una socket virtuale CAN nel modo più preciso possibile. Rispettando cioè definite tempistiche temporali.

Software che si rende necessario per agevolare lo sviluppo e lo studio di attacchi sulla rete intraveicolo. In quanto, una critica che viene spesso mossa agli studi dei ricercatori è quella di aver svolto i tests in un'ambiente virtuale, che non rappresenta ciecamente il comportamento del flusso di dati che transita all'interno della rete di un veicolo.

L'idea è quella di fornire uno strumento completamente open source per agevolare il lavoro di tutti i ricercatori. L'obiettivo principale prefissato è ridurre al minimo il ritardo sul tempo di inter-arrivo di un frame e il successivo.

Di seguito verrà presentata una panoramica sul funzionamento del protocollo CAN e verranno mostrate le scelte implementative e le tecniche utilizzate per sviluppare il progetto.

Il progetto è stato sviluppato partendo da tool e risorse open source.

Capitolo 1

Controller Area Network

Introduzione

Il Controller Area Network, noto anche come CAN-bus, è un protocollo seriale, di tipo multicast e message oriented, introdotto negli anni Ottanta dalla Robert Bosch GmbH (1983), per collegare diverse unità di controllo elettronico (ECU).

È uno standard progettato per permettere ai microcontrollori e dispositivi all'interno di una rete, di comunicare senza l'ausilio di un host computer. I dispositivi inviano i dati attraverso frames che vengono trasmessi in modo seriale, ma in modo tale che se più di un dispositivo trasmette contemporaneamente, il dispositivo con priorità più alta può proseguire mentre gli altri si ritirano dalla comunicazione.

Sebbene inizialmente venne adottato principalmente in ambito automotive, come bus per la rete intra-veicolo, attualmente è usato in molte applicazioni industriali di tipo embedded, dove è richiesto un alto livello di immunità ai disturbi.

In realtà esistono molte reti all'interno di un veicolo, eventualmente basate su standard diversi, criticità diverse, protocolli diversi e velocità di comunicazione differenti. Attualmente queste reti stanno convergendo allo standard CAN, ma ne esistono molte altre. Poiché CAN è ora lo standard de facto per la rete degli autoveicoli, a volte viene identificato anche come VDB (Vehicle Data Bus).

Nonostante la sua popolarità, il bus CAN non è l'unica rete all'interno di un qualsiasi veicolo moderno e in un singolo veicolo di solito ci sono reti diverse (più reti CAN e reti non CAN).

1.1 - Standard e Versioni

CAN è standardizzato nella normativa ISO 11898. Attualmente esistono 2 versioni principali del Protocollo CAN:

- **ISO 11898-2**, chiamato high-speed CAN, con una velocità di trasmissione che può arrivare a 1 Mbit/sec fino a 40m.
- **ISO 11898-3**, chiamato low-speed o fault-tolerant CAN, con una velocità di trasmissione che può arrivare a 125 Kbit/sec.

Abbiamo anche due versioni differenti per il formato dei frames, che andremo meglio ad analizzare più avanti. Riassumiamo come:

- Standard CAN (2.0 A)
- Extended CAN (2.0 B)

L'unica differenza tra i due formati è che il campo "identifier" del frame ha una lunghezza di 11 bits mentre quello dell'Extended CAN una lunghezza di 29 bits, costituito dall'identificatore a 11 bits (identificatore di base) e un'estensione a 18 bits (estensione dell'identificatore).

1.2 - Architettura della Rete

Nodi

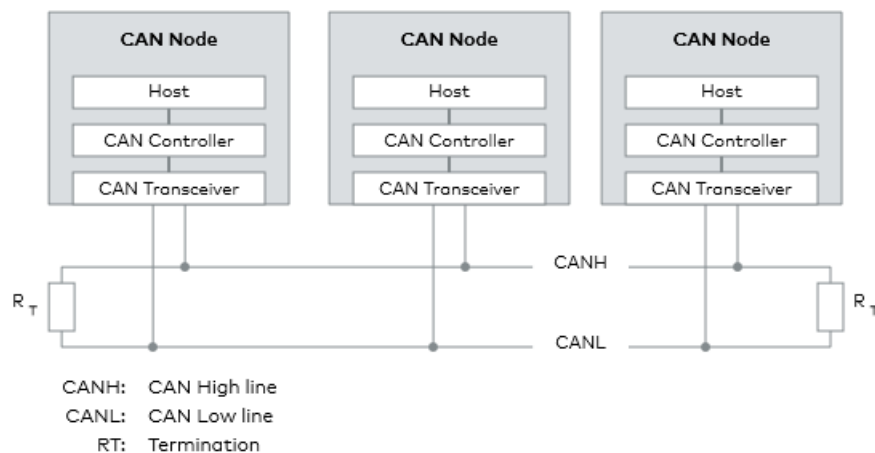


Fig. 1: Nodi interconnessi

Ogni nodo è in grado di ricevere e trasmettere messaggi, ma non in modo simultaneo, ed è composto da:

- *Unità di elaborazione centrale* (microprocessore o host):

Si occupa di elaborare il significato dei messaggi ricevuti e preparare i messaggi che vuole trasmettere. Sensori, attuatori e dispositivi di controllo sono collegati a questo processore.

- *CAN Controller*:

Si occupa della fase di ricezione e trasmissione del messaggio.

Ricezione: il controller CAN memorizza i bit seriali ricevuti dal bus fino a quando non è disponibile un intero messaggio, che può quindi essere recuperato dal processore host (di solito è il controller CAN che attiva un interrupt).

Trasmissione: il processore host invia i frames da trasmettere al controller CAN, che trasmette i bit in modo seriale sul bus quando il bus è libero, servendosi del transceiver.

- CAN Transceiver:

Durante la fase di ricezione, il transceiver traduce i segnali che gli arrivano dal bus da un dominio analogico ad uno logico e li trasmette al controllore.

Durante la fase di trasmissione, il transceiver traduce i segnali che gli vengono passati dal controllore, da un dominio logico ad uno analogico e li trasmette sul bus.

1.3 - Trasmissione Dati

1.3.1 – Bit Dominanti e Recessivi

La trasmissione dei dati avviene secondo un modello basato su bit “dominanti” e bit “recessivi”, in cui i bit dominanti sono gli 0 logici e i bit recessivi sono gli 1 logici.

Tutti i dispositivi comunicanti sono collegati a due fili, identificati come CAN-High (CAN-H) e CAN-Low (CAN-L). Ad ogni estremità, i due fili sono collegati con un resistore di terminazione da 120 ohm.

Avendo due fili, il segnale trasmesso è di tipo differenziale, questa caratteristica permette al CAN bus di essere resistente ai disturbi elettromagnetici (qualsiasi disturbo che agisce su un filo agisce anche sull'altro). Ecco perché il protocollo CAN è ampiamente utilizzato in ambito industriale e specialmente in ambito automotive.

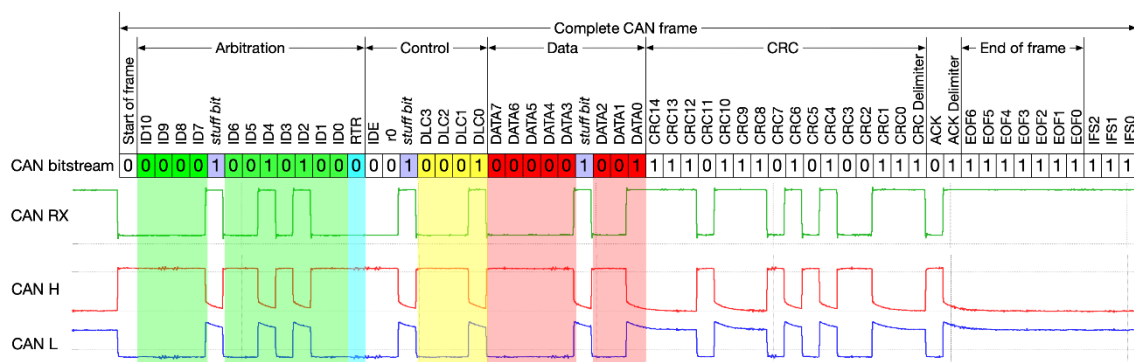


Fig. 1: Segnale differenziale e base frame

Signal		recessive state			dominant state			unit
	min	nominal	max	min	nominal	max		
CAN-High	2.0	2.5	3.0	2.75	3.5	4.5		Volt
CAN-Low	2.0	2.5	3.0	0.5	1.5	2.25		Volt

Fig. 2: CAN Low-Speed voltage levels

Signal		recessive state			dominant state			unit
	min	nominal	max	min	nominal	max		
CAN-High	1.6	1.75	1.9	3.85	4.0	5.0		Volt
CAN-Low	3.1	3.25	3.4	0	1.0	1.15		Volt

Fig. 3: CAN High-Speed voltage levels

1.3.2 – Controllo dell'Arbitrato

Con questa tecnica, quando viene trasmesso un bit recessivo, e contemporaneamente un altro dispositivo trasmette un bit dominante, si ha una collisione, e solo il bit dominante è visibile in rete (tutte le altre collisioni sono invisibili). Si è così sicuri che ogni volta che si impone una differenza di potenziale, tutta la rete la rileva, e quindi "sa" che si tratta di un bit dominante.

Durante la trasmissione, ogni nodo in trasmissione controlla lo stato del bus e confronta il bit ricevuto con il bit trasmesso. Se un bit dominante è ricevuto mentre un bit recessivo è trasmesso il nodo interrompe la trasmissione (ossia perde l'arbitrato).

L'algoritmo di controllo dell'arbitrato è eseguito durante la trasmissione del campo di identificazione del pacchetto del nodo. Al termine dell'invio degli ID, tutti i nodi che dovevano trasmettere si interrompono, tranne quello che ha priorità maggiore, e il messaggio con la priorità corrente massima può liberamente transitare. La figura che segue è un esempio dell'arbitrazione:

	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	

Fig. 5: Trasmissione e controllo dell'arbitrato

Ciò significa che non si aggiunge alcun ritardo per il messaggio con priorità più alta e i nodi che devono trasmettere i messaggi con priorità più bassa tentano automaticamente di ritrasmettere dopo la fine della trasmissione del messaggio dominante. Ciò rende CAN molto adatto come sistema di comunicazione prioritario real-time.

Gli ID dei messaggi devono essere univoci su un singolo bus CAN, altrimenti due nodi continuerebbero la trasmissione oltre la fine del campo di arbitrato (ID) causando un errore.

1.3.3 – Sincronizzazione

Questo metodo di arbitraggio richiede che tutti i nodi sulla rete CAN siano sincronizzati, in modo tale che i nodi della rete siano in grado di vedere contemporaneamente sia i propri dati trasmessi, sia i dati trasmessi dagli altri nodi.

Questo è il motivo per cui alcuni chiamano CAN sincrono. Sfortunatamente il termine sincrono è impreciso poiché i dati vengono trasmessi in un formato asincrono, cioè senza un segnale di clock.

Essendo CAN uno standard asincrono che adotta una codifica dei bit *non-return-to-zero (NRZ)*, l'assenza di uno stato neutro obbliga all'utilizzo di un particolare meccanismo di sincronizzazione.

La sincronizzazione si verifica su ogni transizione di un bit da recessivo a dominante durante l'invio del frame. Il controller CAN prevede che la transizione avvenga a un multiplo del tempo di invio di un bit (*bit timing*). Se la transizione non avviene nel momento esatto previsto dal controllore, il controllore regola di conseguenza il bit timing.

Se il frame contiene molti bit consecutivi dello stesso valore, la sincronizzazione non avviene. Per garantire un numero sufficiente di transizioni per garantire la sincronizzazione, viene inserito un bit di polarità opposta dopo cinque bit consecutivi della stessa polarità, questo meccanismo è chiamato *Bit Stuffing*.

1.4 – Frames

Come precedentemente anticipato, una rete CAN può essere configurata per utilizzare due differenti formati di frame: il formato base (standard) e il formato esteso (extended). L'unica differenza tra questi due formati è la lunghezza del campo identifier.

Nel "CAN base frame", questo campo, è composto da 11 bits, mentre nel "CAN extended frame" la lunghezza è di 29 bits, non consecutivi.

11 bits compongono il “*base identifier*” e 18 bits il “*identifier extension*”.

La distinzione tra l'utilizzo di questi due formati viene effettuata utilizzando il bit IDE, che viene trasmesso come dominante nel caso standard e recessivo nel caso esteso.

Tutti i frame iniziano con un bit di inizio frame (SOF) che indica l'inizio della trasmissione del frame. I frames trasmessi possono essere di quattro tipi:

- 1) *Data frame*: sono i frames con cui vengono effettivamente trasmessi i dati dei messaggi.

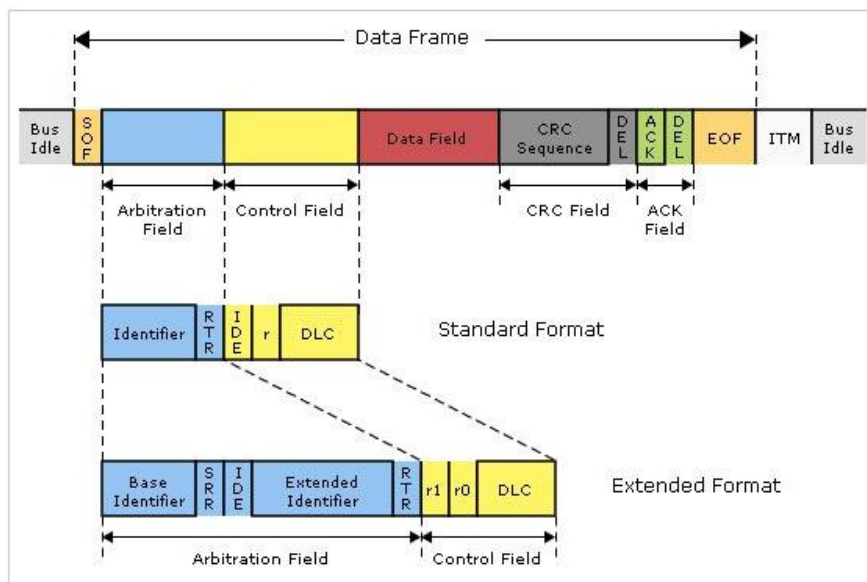


Fig. 6: Data Frame

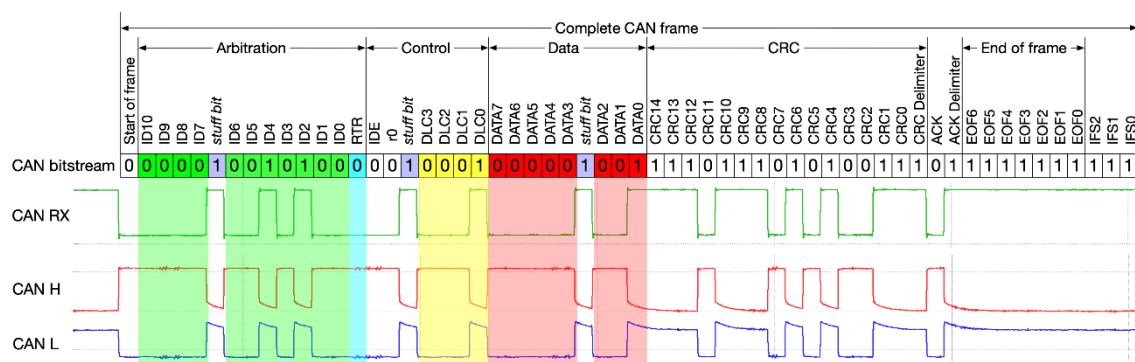


Fig. 7: Segnale differenziale e base frame

Riproponiamo l'immagine per spiegare i campi e i bit presenti nel data frame. In questo caso stiamo analizzando il “base format”. In questa immagine viene mostrata anche la trasmissione del bit introdotto dal meccanismo di Bit Stuffing.

- Per iniziare l'inizio della trasmissione del frame viene inviato lo *start of frame*.
- Successivamente vengono inviati 11 bits relativi al campo *identifier*, mostrati in verde.
- In questo caso è presente anche il *bit di stuffing*.
- Il bit *RTR (Remote Transmission Request)* indica se è un frame è un data frame o un remote frame. Nel caso sia un data frame questo bit sarà dominante (0), come mostrato in figura in colore azzurro. Nel caso sia un remote frame questo bit sarà recessivo (1).
- Il bit *IDE (Identifier Extension bit)* assume valore dominante (0) per indicare un "base format", mentre assume un valore recessivo (1) per il formato esteso.
- *DLC (Data Length Code)*, mostrato in giallo, indica il numero di bytes di dati del payload. Sono 4 bits, quindi possiamo indicare un numero di byte compreso tra 0 e 8.
- *Data field*, mostrati in rosso, sono i bits di payload.
- *CRC (Cyclic redundancy check)*, è un codice di rilevamento degli errori comunemente utilizzato nelle reti digitali per rilevare modifiche accidentali ai dati. È progettato per assicurare l'integrità dei dati trasmessi a fronte di errori che possono avvenire sul canale di comunicazione. Ci fornisce una sicurezza sull'integrità dei dati, ma non sull'autenticità. Non ci protegge da una modifica intenzionale fatta sui dati trasmessi. Un'attaccante potrebbe modificare il payload e ricalcolare il campo CRC.
- *CRC delimiter*, deve essere recessivo (1)
- *EOF (End Of Frame)*, sono 7 bits che devono essere tutti recessivi (1).

2) *Remote Frame*: generalmente la trasmissione dei dati viene eseguita utilizzando i data frames. È possibile, tuttavia, che un nodo di destinazione richieda i dati dalla sorgente inviando un remote frame.

Esistono due differenze sostanziali tra un data frame e un remote frame.

- La prima differenza è che il bit RTR viene trasmesso come bit dominante nel data frame, recessivo nel remote frame.
In questo modo se un data frame e un remote frame con lo stesso identificatore vengano trasmessi contemporaneamente, il data frame vince l'arbitrato a causa del bit RTR dominante che segue l'identificatore.
 - La seconda differenza è che nel remote frame non vi è alcun "Data field". Il campo DLC indica la lunghezza dei dati che deve avere il messaggio richiesto.
- 3) *Error Frame*: è un frame con il quale un nodo può segnalare un malfunzionamento della rete. Ogni nodo della rete può rilevare dei

malfunzionamenti e segnalarli agli altri nodi trasmettendo un frame di questo tipo. È importante per avere un certo grado di robustezza della rete. All'interno di questo frame abbiamo dei bits che rappresentano dei flags con cui segnalare gli errori.

- 4) *Overload Frame*: Se un nodo CAN riceve i frames più velocemente di quanto possa elaborarli, invia un overload frame per "iniettare" un ritardo tra i data frame o gli error frame.

Capitolo 2

Sicurezza

2.1 – Caratteristiche

VANTAGGI

Come già anticipato, CAN è un protocollo ben consolidato in ambito automotive e che gode di determinati benefici, tra cui:

- *Affidabilità e Robustezza*: la possibilità di perdere dati durante la trasmissione del messaggio è praticamente nulla grazie alle varie possibilità di segnalazione e rilevamento di errori.
- *Risparmio economico*: la riduzione di peso, cablaggio e costi di produzione ha rappresentato il principale obiettivo per la nascita dello standard CAN-bus in ambito automotive, semplificandone non solo il montaggio ma anche la manutenzione.
- *Flessibilità*: il protocollo CAN-bus si basa sui messaggi e non sui nodi che lo compongono; questa caratteristica consente di aggiungere e integrare nuovi dispositivi elettronici senza una riprogrammazione specifica.
- *Velocità*: La condivisione in tempo reale tra i nodi della rete e la capacità di elaborazione condivisa dei dati conferisce infatti un'elevata velocità all'intera rete.
- *Efficienza*: la possibilità di gestire il grado di priorità in base all'ID, consente di mantenere fluida la gestione dei vari frame e garantisce efficienza all'intera rete.

SVANTAGGI

La principale problematica del protocollo CAN-bus è l'assenza di protocolli di sicurezza. Nasce per far comunicare dispositivi di tipo embedded con forti vincoli sulle risorse.

Nasce in un'epoca in cui le centraline delle autovetture non erano ancora fortemente connesse con il mondo esterno, si pensi ad esempio al sistema di infotainment odierno. Con la sua connettività bluetooth e mobile.

Come precedentemente annunciato, il protocollo CAN, implementa solo meccanismi di rivelazione degli errori per garantire robustezza e integrità della comunicazione. Questi meccanismi (error frame e crc) non ci proteggono però da attaccanti attivi.

2.2 – Attacchi

Esistono vari tipi di attacco a cui può essere sottoposta la nostra rete. Partiamo da uno scenario in cui un'attaccante abbia accesso al bus, indipendentemente che sia in modo diretto (tramite una connessione fisica) o indiretto (potendo eseguire codice tramite una ECU).

Questo scenario permette ad un attaccante di “sniffare” informazioni dal traffico sulla rete e iniettare frames senza alcuna limitazione sul contenuto o sulla frequenza di invio.

I tipi di attacco che potrebbe eseguire l'attaccante sono:

1) *Replay attack*

Questo tipo di attacco viene effettuato “catturando” messaggi dal traffico di rete per ritrasmetterli successivamente ad un momento arbitrario. Lo scopo di questo attacco è modificare il normale funzionamento delle ECUs.

Immaginiamo di avere un sensore per rilevare gli ostacoli posti di fronte al veicolo. Questo sensore invia ciclicamente un frame che viene interpretato dal sistema di guida assistita come “nessun ostacolo” oppure “ostacolo ad N metri”.

Catturando il primo frame e iniettandolo ripetutamente sul bus, con una frequenza più elevata con la quale il sensore legittimo invia il suo stato, andiamo ad inibire il normale comportamento del sistema di guida autonoma.

Il replay attack può essere a sua volta suddiviso in tre tipologie:

- *Single message replay*: viene catturato ed iniettato un singolo frame.
- *Ordered sequence replay*: una sequenza ordinata di frames viene catturata e iniettata successivamente senza alcuna modifica.
- *Arbitrary sequence replay*: dopo aver catturato un insieme di frames, l'attaccante li riordina a suo piacimento per ottenere una sequenza da iniettare sul bus.

2) *Fuzzing Attack*

Il “Fuzzing” è una tecnica mirata a scovare vulnerabilità o aiutare nel reverse-engineering di un protocollo di comunicazione non documentato. Viene

eseguito iniettando input malevolo nel sistema. In ambito automotive viene eseguito iniettando frames generati randomicamente.

Questo tipo di attacco possiamo suddividerlo in:

- *ID fuzzing*: l'attaccante genera ed invia del traffico CAN sulla rete con ID e payload randomico. Il campo ID è generato in modo tale da essere differente da qualsiasi ID che solitamente transita nella rete.
- *Payload fuzzing*: l'attaccante genera traffico CAN sulla rete, in modo tale che i frames abbiano un'ID valido (precedentemente osservato sniffando il traffico) ma payload randomico. Riferendoci all'esempio precedente, con questo attacco, possiamo scoprire come deve essere il frame che segnala "ostacolo ad N metri".

3) *Denial-of-Service*

Questo attacco mira ad inibire il normale funzionamento del sistema, interrompendo il normale accesso alle risorse da parte delle parti legittime della comunicazione.

Questo tipo di attacco può essere eseguito iniettando frames che hanno un'alta priorità ad un'elevata frequenza. In questo modo si vanno ad inibire i messaggi provenienti da ECUs legittime che hanno una priorità più bassa.

Possiamo suddividerlo in:

- *Zero ID DoS*: l'attaccante inietta frames con il campo ID settato a 0. Tutti questi messaggi vinceranno l'arbitrazione inibendo qualsiasi altro messaggio sulla rete. Questa tipologia di DoS può però essere facilmente prevenuta eseguendo un'azione (ad esempio disabilitare la ECU che trasmette con ID 0) nel caso in cui l'ID 0 non sia considerato legittimo.
- *Lowest ID DoS*: l'attaccante inietta dei frames con il campo ID molto basso, prendendolo dai frames precedentemente "sniffati" dal traffico.

Per proteggersi da questo tipo di attacchi occorrerebbe implementare un paradigma di tipo "Defense in depth", ma prendendo come riferimento il modello IT classico e i suoi algoritmi crittografici bisognerebbe aggiungere hardware apposito o modificare le ECUs esistenti. Questo andrebbe ad impattare fortemente sul ciclo di sviluppo e sui

costi di produzione, facendo venire meno una delle caratteristiche principali per cui CAN è stato adottato in ambito automotive.

I ricercatori stanno studiando svariate soluzioni che non si basano su algoritmi crittografici per non impattare sulle ECUs già esistenti, basate ad esempio sull'ispezione dei messaggi e utilizzo di machine learning. Vi sono comunque alcune criticità.

Per esempio se si volesse effettuare un'ispezione dei messaggi basandosi sulla loro entropia, questa funzionerebbe solo per enormi flussi di dati.

Oppure i modelli di machine learning sono difficilmente eseguibili su dispositivi constrained.

Capitolo 3

Il Progetto

Introduzione

Per la ricerca e lo sviluppo di sistemi, algoritmi e architetture di anomalies detection da applicare alla rete interna che collega le ECU del veicolo, bisogna fare in modo che i ricercatori siano in grado di simulare, nel modo più accurato possibile, la serie di messaggi (frames) scambiati all'interno della rete CAN.

In particolare, i programmi opensource presenti nella libreria can-utils, spesso utilizzati dai ricercatori per le simulazioni su socket fisiche o virtuali, non permettono di inviare i frames rispettando in modo soddisfacente i timestamp del dataset.

Problematica che viene accentuata quando si vuole effettuare una simulazione utilizzando dataset la cui successione di frames abbiano timestamp che differiscono di una grandezza uguale o inferiore ai microsecondi. Anche con distanze temporali di ordini di grandezza superiori si verificano comunque ritardi significativi e imprevedibili, a causa di motivazioni che andremo ad indagare successivamente.

Alla luce di quanto esposto, si è deciso di sviluppare un programma per inviare i frames su socket fisiche o virtuali, che rispetti quanto più fedelmente la distanza temporale tra un frame e il successivo.

3.1 – Setup Iniziale e Tools di Sviluppo

3.1.1 - Specifiche Hardware e Software

Dovendo operare sui tempi di esecuzione delle operazioni, utilizzando un sistema operativo general purpose, si rende necessario dettagliare le specifiche hardware e software del sistema con cui è stato sviluppato ed eseguito il programma.

Specifiche che ci aiuteranno in seguito a capire le scelte implementative e le soluzioni adottate per lo sviluppo del progetto.

In questo caso stiamo utilizzando come S.O una distro Linux installata su un'architettura a 64bit.

La CPU è un AMD Ryzen 7 4800H dotato di 8 core fisici e 16 thread.

Riguardo alla RAM abbiamo installato sulla macchina un banco da 16 gigabyte.

Per compilare il codice sorgente è stato utilizzato come compilatore GCC alla versione 9.4.0

Per completezza e perché ci verrà utile in futuro nel descrivere le strutture dati implementate, inseriamo due immagini che mostrano alcune linee di codice e l'output della loro esecuzione.

```
printf("STDC_VERSION: %ld\n", __STDC_VERSION__);  
printf("sizeof(long long int): %ld\n", sizeof(long long int));  
printf("LONG_LONG_MAX MACRO: %lld\n", __LONG_LONG_MAX__);
```

Fig 8: Codice stampa versione C e dimensione tipo long long int

```
STDC_VERSION: 201710  
sizeof(long long int): 8  
LONG_LONG_MAX MACRO: 9223372036854775807
```

Fig 9: Output codice stampa versione C e dimensione tipo long long int

Dall'esecuzione di questo codice notiamo come essendo la nostra architettura a 64 bit, la dimensione del tipo di dato long long int sia 8 byte. Questo tipo di dato ci permette di rappresentare quindi un intero compreso nel range:

`[-9223372036854775807,+ 9223372036854775807]` estremi inclusi.

Stampando la macro `STDC_VERSION` ci vengono fornite informazioni sullo standard della versione C utilizzata, in questo caso 201710, in accordo con la nomenclatura YYYYMM, è la versione C17 definita nella norma ISO/IEC 9899:2018.

3.2 – CAN-UTILS

Can-utils, come si può intuire dal nome, è un'utilità Linux a riga di comando che contiene gli strumenti di base in grado di visualizzare, registrare, generare e riprodurre il traffico CAN su di una socket virtuale o fisica.

In accordo con quanto riportato dalla documentazione presente nel repository GitHub ufficiale, all'interno di can-utils troviamo:

- *candump*: ci permette di visualizzare, filtrare e salvare su file i frames in transito su una socket CAN.
- *cansniffer*: permette di visualizzare le differenze di contenuto tra i frames.
- *canplayer*: riprodurre un dataset su una socket CAN.
- *cangen*: generare traffico CAN casuale su di una socket.

- *consequence*: invia una sequenza di frames con payload incrementale.
- *cansend*: invia un singolo frame su una socket.

Ai fini della realizzazione del progetto sono stati utilizzati due di questi tools:

-Canplayer: dal cui codice sorgente si è partiti per realizzare il tool per cui è nato il progetto.

-Candump: necessario per loggare i frames e i relativi timestamp durante l'esecuzione dell'applicativo core del progetto.

Andiamo ora ad approfondire meglio il loro utilizzo.

3.2.1 – Canplayer e Candump

```
Usage: canplayer <options> [interface assignment]*
Options:
    -I <infile>    (default stdin)
    -l <num>        (process input file <num> times)
                   (Use 'i' for infinite loop - default: 1)
    -t              (ignore timestamps: send frames immediately)
    -g <ms>         (gap in milli seconds - default: 1 ms)
    -s <s>          (skip gaps in timestamps > 's' seconds)
    -x              (disable local loopback of sent CAN frames)
    -v              (verbose: print sent CAN frames)

Interface assignment:
0..n assignments like <write-if>=<log-if>
e.g. vcan2=can0 (send frames received from can0 on vcan2)
extra hook: stdout=can0 (print logfile line marked with can0 on stdout)
No assignments => send frames to the interface(s) they had been received from.

Lines in the logfile not beginning with '(' (start of timestamp) are ignored.
```

Fig 10: Canplayer Usage

```

candump - dump CAN bus traffic.

Usage: candump [options] <CAN interface>+
       (use CTRL-C to terminate candump)

Options:
  -t <type>      (timestamp: (a)bsolute/(d)elta/(z)ero/(A)bsolute w date)
  -H             (read hardware timestamps instead of system timestamps)
  -c             (increment color mode level)
  -i             (binary output - may exceed 80 chars/line)
  -a             (enable additional ASCII output)
  -S             (swap byte order in printed CAN data[] - marked with ' ' )
  -s <level>     (silent mode - 0: off (default) 1: animation 2: silent)
  -b <can>       (bridge mode - send received frames to <can>)
  -B <can>       (bridge mode - like '-b' with disabled loopback)
  -u <usecs>     (delay bridge forwarding by <usecs> microseconds)
  -l             (log CAN-frames into file. Sets '-s 2' by default)
  -L             (use log file format on stdout)
  -n <count>     (terminate after reception of <count> CAN frames)
  -r <size>      (set socket receive buffer to <size>)
  -D             (Don't exit if a "detected" can device goes down.
  -d             (monitor dropped CAN frames)
  -e             (dump CAN error frames in human-readable format)
  -x             (print extra message infos, rx/tx brs esi)
  -T <msecs>     (terminate after <msecs> without any reception)

Up to 16 CAN interfaces with optional filter sets can be specified
on the commandline in the form: <ifname>[,filter]*

```

Fig 11: Candump Usage

Di seguito verrà presentato come si serve il nostro applicativo main.py di questi 2 tool.

```

processCandump = subprocess.Popen("candump -t d -L vcan0 > " + candumpLogFullPathFileName, stdout=subprocess.PIPE,
                                   shell=True, preexec_fn=os.setsid)
os.system("./canplayer -r")
time.sleep(1)
processCandump.send_signal(signal.SIGINT)
os.killpg(os.getpgid(processCandump.pid), signal.SIGTERM)

```

Fig 12: codice esecuzione canplayer e cundump applicativo main.py

Il primo processo lanciato è:

“candump -t d -L vcan0 > [candumpLogFullPathFileName]”

In questo caso l’opzione “-t d” indica di loggare i frames che transitano sull’interfaccia di rete vcan0 annotando i timestamp in forma di delta, cioè la differenza temporale rispetto al frame precedentemente loggato.

Invece l’opzione “-L” indica di loggare i frames su stdout.

Con “> [candumpLogFullPathFileName]” andiamo poi a ridirezionare l’output da stdout verso un file il cui path completo è descritto dal valore assunto dalla variabile “candumpLogFullPathFileName”. Questa variabile sarà valorizzata dallo script main.py per eseguire in modo automatico le simulazioni utilizzando diversi dataset di test.

Il secondo processo è:

```
<./canplayer -r>
```

In questo caso, l'eseguibile "canplayer", è ottenuto dalla compilazione del tool canplayer a seguito della nostra modifica per funzionare in modalità real-time (rispettando nel modo più accurato possibile i timestamp). Ecco perché viene lanciato con l'opzione -r, non presente nel normale usage del tool standard.

3.3 – Codice

Si andranno di seguito a descrivere le modifiche significative effettuate al codice sorgente di canplayer, con particolare focus sulle strutture dati utilizzate e ai principi di programmazione real-time utilizzati.

3.3.1 – Librerie e Strutture Dati

```
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#include <libgen.h>

#include <unistd.h>

#include <linux/can.h>
#include <linux/can/error.h>
#include <sys/socket.h> /* for sa_family_t */

#include "lib.h"

#include <linux/can/raw.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/time.h>
```

Fig 13: Librerie canplayer

1) TIME.H

Andiamo ad estendere le librerie di canplayer con una libreria che ci permetta di utilizzare e gestire il tempo di sistema:

```
#include <sys/time.h> /* gettimeofday, timeval (for timestamp in microsecond) */
```

Fig 14: libreria time.h

Di questa libreria andremo ad utilizzare la funzione “gettimeofday” per recuperare il tempo corrente di sistema e la struttura “struct timeval” definita come:

```
struct timeval {
```

```
    time_t    tv_sec;
```

```
    long int   tv_usec;
```

```
};
```

- Il campo “tv_sec” di tipo “time_t” ci viene descritto nella documentazione, che riporta quanto segue:

“In ISO C, time_t può essere un tipo integer o floating-point”, e aggiunge:

“Sui sistemi conformi a POSIX, time_t è un tipo intero e i suoi valori rappresentano il numero di secondi trascorsi dalle ore 00:00:00 del 1 gennaio 1970, Coordinated Universal Time.

La libreria GNU C garantisce inoltre che time_t sia un tipo con segno e che tutte le sue funzioni operino correttamente su valori time_t negativi, che vengono interpretati come tempi antecedenti a quella data.”

- Nel campo tv_usec di tipo “long int” verrà specificato il numero di microsecondi.

2) CAN.H

Una libreria di fondamentale importanza è la libreria linux/can.h.

Questa libreria, inclusa nelle librerie di sistema, ci definisce la struttura “struct can_frame” che andremo ad utilizzare per caricare in memoria i frames da inviare alla socket CAN. La struttura, come specificato nella documentazione comprende i seguenti campi:

```
canid_t    can_id
```

```
__u8       can_dlc
```

```
__u8       data[CAN_MAX_DLEN]
```

- “can_id”: è un campo di tipo “canid_t”, definito come “typedef __u32 canid_t” alla linea 39 della libreria can.h. Il tipo di dato “__u32” è definito come “typedef unsigned int __u32” alla linea 26 della libreria int-l64.h.

In definitiva il campo “can_id” è di tipo “unsigned int”, corrispondente a 4 byte. Con questo campo andremo a memorizzare il campo ID del frame CAN.

- “can_dlc”: è un campo di tipo “__u8”, definito come “typedef unsigned char __u8” alla linea 20 della libreria int-l64.h.

Il campo “can_dlc” è invece un tipo “unsigned char”, corrispondente a 1 byte. Con questo campo andremo a specificare il numero di byte del payload.

- “data”: è un array di dimensione 8, come definito dalla macro “CAN_MAX_DLEN” dichiarata alla linea 54 della libreria can.h (#define CAN_MAX_DLEN 8).

In questo campo andrà caricato il payload effettivo del frame.

3.4 – Programmazione Real-time

Ora che abbiamo definito le dipendenze e le strutture dati che andremo ad utilizzare, passiamo ad analizzare la logica e le scelte implementative adottate.

3.4.1 – Inizializzazione Strutture Dati

```
//Initialize data structure
int numFrame=countFrames();

long long int delta[numFrame];
struct can_frame frames[numFrame];

readFrames(frames,delta,numFrame);
```

Fig 15: Inizializzazione struttura dati

Nel codice mostrato in figura si può osservare come avviene l’inizializzazione della struttura dati necessaria al replay del dataset su socket CAN.

Questa fase è fondamentale ai fini del raggiungimento dello scopo prefissato dal progetto. Se si vogliono rispettare tempistiche stringenti per quanto riguarda l’invio dei frames, è necessario svolgere tutte le operazioni non legate strettamente all’invio dei frames prima del ciclo che andrà a replicare il dataset sulla socket.

Caricando in memoria tutte le strutture dati necessarie in anticipo rispetto alla fase di invio, andremo a minimizzare le operazioni che si devono svolgere tra l’invio di un

frame e il successivo. Operazioni il cui tempo di esecuzione non è prevedibile a priori, dipendendo dallo stato delle risorse e dai processi in esecuzione dal sistema operativo in quel dato momento.

Seguendo la struttura del codice:

- `int numFrame=countFrames();`

La variabile “numFrame” viene dichiarata e successivamente inizializzata con il valore ritornato dalla funzione “countFrames()”. Questa funzione si occupa di contare le righe del dataset, scartando la prima che rappresenta l’intestazione che specifica il formato con cui sono rappresentati i frames. Non ci soffermeremo ad analizzarla perché non ritenuta di particolare importanza.

- `long long int delta[numFrames];`

Istruzione che dichiara la variabile “delta” come un array di dimensione “numFrame”. In ogni campo ci salviamo la differenza di tempo (d’ora in avanti indicata come tempo inter-arrivo) tra il timestamp di un frame e il successivo.

Come mostrato in figura 5, la dimensione di ogni campo dell’array è 8 byte, inizialmente il codice voleva raggiungere una precisione nell’ordine dei nanosecondi e il tipo di dato doveva permettere di memorizzare il tempo inter-arrivo con questo ordine di grandezza. Precisione che non si è riuscita a raggiungere, ma scelta implementativa mantenuta per eventuali.

- `struct can_frame frames[numFrames];`

Array di dimensione “numFrames”, dove ogni campo è di tipo struct can_frame, struttura precedentemente analizzata nel capitolo 2.3.1.

- `readFrames(frames,delta,numFrames);`

È una funzione che si occupa di inizializzare correttamente le variabili “frames” e “delta” andando a leggere il dataset. Più precisamente la logica fondamentale della funzione è la seguente:

```

while(1) {
    j=0;
    res=fgets(buf, 200, fp);

    if( res==NULL )
        break;

    token = strtok(buf, temp);

    while( token != NULL ) {

        if(j==0){
            delta[i]=(long long int)(atof(token)*1000000)-temp2;
            temp2=(long long int)(atof(token)*1000000);
        }else if(j==1){
            frames[i].can_id=(int)strtol(token, NULL, 16);
        }else if(j==2){
            frames[i].can_dlc=atoi(token)/8;
        }else if(j==3){
            hexToASCII(token, frames[i].data);
        }

        token = strtok(NULL, temp);

        j=j+1;
    }
    i=i+1;
}

```

Fig. 16: Codice funzione readFrames

Questa funzione utilizza a sua volta la funzione “hexToASCII”, che preso in ingresso come primo parametro un array di char rappresentante una stringa contenente la rappresentazione esadecimale del payload del frame, va a valorizzare il secondo parametro, il campo data della struttura can_frame.

3.4.2 – Gestione del Tempo

Dovendo lavorare su tempistiche restringenti in un sistema operativo non real-time, dobbiamo adottare alcune strategie per rendere quanto più prevedibile il tempo di esecuzione dell’istruzione che invia il frame sulla socket.

Definiamo ora alcune macro che serviranno a comprendere meglio la trattativa che segue:

```

#define MICROSECONDS_TO_SYNCHRONIZE 1000000
#define MICROSECONDS_OF_MARGIN 2
#define NUM_FRAMES_STABILIZE 10

```

Fig. 17: Macro per operare sull’invio

Andiamo ad inizializzare la variabile “frameTime” che serve a contenere i tempi, espressi in microsecondi, con cui i frames dovrebbero essere inviati. Per valorizzarla utilizzeremo la variabile “delta” precedentemente analizzata.

Il tempo di inizio del replay del dataset sulla socket è definito nella variabile “startTime”, valorizzata in base al tempo corrente del sistema più un offset definito dalla macro “MICROSECONDS_TO_SYNCHRONIZE”. Macro necessaria per darci il tempo necessario a completare l’inizializzazione della variabile “frameTime”.

```
//Send CAN frame on the CAN socket

long long int currentTime;

long long int initialTime=getMicroseconds();
long long int startTime=initialTime + MICROSECONDS_TO_SYNCHRONIZE;    //Start after 1 second

long long int frameTime[numFrame];
frameTime[0]=startTime;
for(int i=1;i<numFrame;i++)
    frameTime[i]=frameTime[i-1]+delta[i];
```

Fig. 18: inizializzazione tempo di invio dei frame

Per ottenere il tempo corrente di sistema, espresso in microsecondi, usufruiamo della funzione “getMicroseconds”, la cui implementazione è riportata nella figura che segue.

```
long long int getMicroseconds(){
    struct timeval timer_usec;
    long long int timestamp_usec;

    //get timestamp in microseconds
    if (!gettimeofday(&timer_usec, NULL)) {
        timestamp_usec = ((long long int) timer_usec.tv_sec) * 1000000ll +
                        (long long int) timer_usec.tv_usec;
    }else{
        timestamp_usec = -1;
    }

    return timestamp_usec;
}
```

Fig. 19: Funzione getMicroseconds

Terminata l’analisi di tutte queste operazioni preliminari, andiamo ad analizzare il blocco di codice che si occupa dell’invio dei frames.

```
for(int i=0;i<numFrame;i++){
    currentTime=getMicroseconds();
    //Active sleep
    while(currentTime<frameTime[i]-MICROSECONDS_OF_MARGIN){
        currentTime=getMicroseconds();
    }
    write(s, &frames[i], sizeof(struct can_frame));
}
```

Fig. 20: Blocco di codice che si occupa dell’invio dei frames

Adottando le scelte implementative precedentemente illustrate, siamo riusciti a minimizzare il numero di istruzioni che intercorrono tra l'invio di un frame e il successivo.

Prendendo come riferimento come operazione elementare (definizione non puramente corretta) l'istruzione `write(s, &frames[i], sizeof(struct can_frame))`. Per rispettare nel modo più accurato possibile il tempo a cui inviare i frames, sono stati adottati due ulteriori accorgimenti:

- *Margine di anticipo*: prevedere un margine con cui verificare che il tempo corrente sia adeguato ad eseguire l'invio del frame sulla socket. Margine definito dalla macro `"MICROSECONDS_OF_MARGIN"`, valorizzata dopo aver svolto diversi test e stabilendo che il tempo necessario all'esecuzione dell'operazione di scrittura del frame statisticamente abbia come moda 2 microsecondi.
- *Active sleep*: tra l'invio di un frame e il successivo. Se avessimo usato una funzione di attesa passiva (come la funzione `"sleep"`), lo scheduler del sistema operativo avrebbe messo in stato di attesa il processo e lo avrebbe rischedulato, non una volta terminato il tempo di attesa, ma attendendo la fine dell'esecuzione dei processi a priorità più elevata che nel frattempo sarebbero stati schedulati durante lo stato di wait del processo canplayer.

3.4.3 – Stabilizzazione dello Stato della Socket

Nonostante gli accorgimenti adottati, dopo successivi test di prova, il timestamp dei primi frame inviati differiva di centinaia di microsecondi rispetto al tempo di invio aspettato. Questi ritardi iniziali andavo a gravare sulle prestazioni del replay del dataset.

Si notava come i microsecondi di ritardo sui primi frames andavo via via diminuendo con il numero di frames inviati. Per questo motivo si è scelto di inviare un certo numero di frames vuoti, definito dalla macro `"NUM_FRAMES_STABILIZE"`, per far rientrare la socket in uno stato stabile per l'invio dei frames.

Il valore della macro è stato definito attraverso diversi test, in cui si è individuato che attorno a 8 frames non avveniva più ritardo. E' stato scelto 10 come sicurezza aggiunta.

L'implementazione della funzione che si occupa di inviare i frames vuoti è riportata nella figura successiva.

```

void sendEmptyFrame(int s,int numFrame){
    struct can_frame emptyFrame;

    emptyFrame.can_id = 0x000;
    emptyFrame.can_dlc = 8;
    hexToASCII("0000000000000000",emptyFrame.data);

    for(int i=0;i<numFrame;i++){
        if (write(s, &emptyFrame, sizeof(struct can_frame)) != sizeof(struct can_frame)) {
            perror("Write");
            return;
        }else{
            printf("Inviato frame vuoto\n");
        }
    }
}

```

Fig. 21: Implementazione funzione sendEmptyFrame

Questa strategia per stabilizzare lo stato della socket potrebbe andare a sporcare la replica del dataset, per questo motivo si è deciso di eseguire questa parte di codice in base ad un parametro opzionale.

```

//send empty frames to enter in the steady state
if(STABILIZE){
    sendEmptyFrame(s,NUM_FRAMES_STABILIZE);
}

```

Fig. 22: Esecuzione funzione sendEmptyFrame

3.5 – Struttura del Progetto

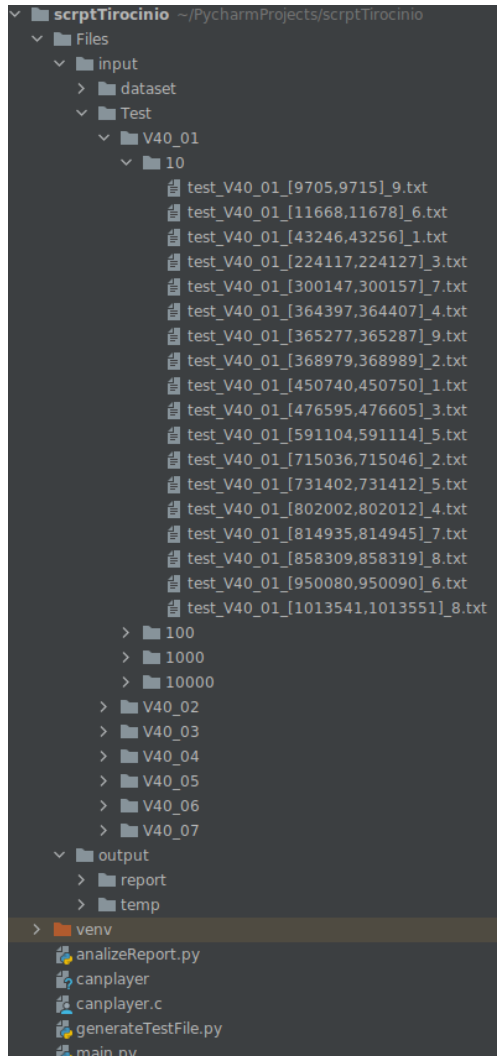


Fig 23: Struttura progetto

Il codice del progetto si compone di 3 script in linguaggio python e del codice modificato di canplayer, che abbiamo già attentamente analizzato. La struttura della directory “Files” ci verrà presto necessaria a spiegare il funzionamento degli script.

3.5.1 – Dataset

Il dataset utilizzato è composto da 7 diverse tracce, include più di 8 milioni di messaggi CAN corrispondenti a circa 90 minuti di traffico sulla rete. Le tracce sono raccolte in diverse sessioni di guida effettuate su diverse tipologie di strade (urbane, extraurbane, autostrada, ecc...), viabilità, condizioni meteorologiche e aree geografiche (pianura, collina e montagna).

Le tracce includono ID, DLC e payload di ciascun frame con il relativo timestamp. Il dataset include messaggi con 51 ID diversi.

Il dataset è stato ottenuto dalla seguente risorsa:

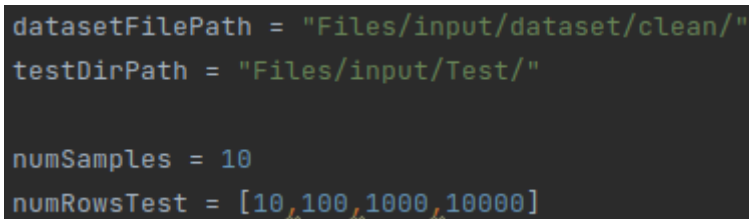
<https://sites.google.com/a/hksecurity.net/ocslab/Datasets/CAN-intrusion-dataset>

3.5.2 – Script

Nel progetto sono presenti 3 script python che sono stati utilizzati per automatizzare la fase di testing e analisi dei risultati durante lo sviluppo della modifica al codice di canplayer. Andiamo a dettagliarli meglio, secondo l'ordine con cui essi dovrebbero essere utilizzati per eseguire un test.

1) *generateTestFile.py*

Come ci suggerisce il nome, questo script serve per generare i files di test su cui eseguire la simulazione.



```
datasetFilePath = "Files/input/dataset/clean/"
testDirPath = "Files/input/Test/"

numSamples = 10
numRowsTest = [10, 100, 1000, 10000]
```

Fig. 24: variabili generateTestFile.py

La figura ci mostra i parametri che possiamo andare a valorizzare per generare i files di test.

La variabile "numSamples" indica il numero di files di test generati per ogni elemento della variabile "numRowsTest".

La variabile "numRowsTest" è un'array, in cui ogni elemento indica il numero di righe (cioè di frames) che deve contenere ogni file di test.

I frames contenuti nei files di test sono presi partendo da una riga casuale del dataset processato.

La variabile "datasetFilePath" ci specifica da che directory ottenere i dataset.

L'esecuzione dello script produce in output, un insieme di cartelle e sottocartelle contenenti i files di test. Il nome di un file di test rispetta una nomenclatura che ne indica: dataset da cui sono stati estratti i frames e prima riga e ultima riga da cui sono stati estratti. La nomenclatura è la seguente:

"test_<NOME DATASET>_[<PRIMA RIGA>,<ULTIMA RIGA>]_<NUMERO
SAMPLE>.txt"

Il file è salvato all'interno di un path a sua volta autodescrittivo:

"<Files/Input/Test/<NOME DATASET>/<NUMERO DI FRAMES>

2) *main.py*

Questo script si occupa di eseguire una simulazione andando ad eseguire canlayer e candump. Svolgerà una simulazione per ogni dataset di test presente nella cartella "Files/Test" che lo script generateTestFile.py ci ha precedentemente popolato.

I files con i frames loggati da candump saranno salvati nella directory "Files/output/temp/candumpLog".

I frames con il timestamp aspettato dall'esecuzione di canplayer saranno salvati in files nella directory "Files/output/temp/canplayerLog".

Una spiegazione più dettagliata di come vengono eseguiti questi due processi è già stata affrontata nel capitolo 2.2, più precisamente Fig. 8 mostra il codice dello script.

3) *analyzeReport.py*

Script che si occupa di confrontare i risultati presenti nelle directories "Files/output/temp/canplayerLog" e "Files/output/temp/candumpLog", precedentemente popolate dall'esecuzione dello script "main.py".

Confronta i risultati estrapolando e facendo analisi sugli errori tra i timestamp aspettati e quelli effettivamente loggati da candump.

Per ogni dataset presente nella cartella "Test" questo script ci genererà un report che andrà a salvare nella directory "Files/output/report".

Ogni report conterrà le seguenti colonne:

- Frametime: colonna che indica il timestamp a cui canplayer avrebbe dovuto inviare il frame.
- RealTime: questa colonna ci indica il timestamp in cui candump ha loggato il frame in transito sulla socket.
- Difference: indica la differenza di tempo tra il timestamp atteso (Frametime) e il timestamp reale (RealTime).

Capitolo 4

Analisi dei Risultati

In questa sezione andremo ad analizzare i risultati ottenuti, usuiiremo dell'output dell'esecuzione dello script "analyzeReport.py".

Oltre a generare i files di report, lo script si occuperà inoltre di processarli per stampare sul terminale alcune statistiche globali sulle simulazioni.

```
NUMFRAME: 805599
DELTATOT: 347747
RITARDMEDIO: 0.43166265102116563
VALUE  COUNT  %
-2      54    0.006703086771458256
-1    30913   3.837268914186835
0    494242  61.35087059442725
1    260012  32.27561106704452
2     6448   0.8003982130067192
3     2738   0.3398713255602353
4     2168   0.26911652075039816
5     1868   0.2318771497978523
6     1531   0.19004492309449242
7     1284   0.15938450767689633
8     1058   0.13133084822597843
9       798   0.09905672673377201
10      588   0.07298916706698991
11      420   0.05213511933356422
12      266   0.03301890891125734
13      203   0.025198641011222704
14      164   0.02035752278739174
15      121   0.015019879617526833
16       91   0.011295942522272248
17       82   0.01017876139369587
18       61   0.0075720054270176605
```

Fig. 25: Output script analyzeReport.py

Da questa esecuzione possiamo vedere come su un totale di 805599 frames inviati, da 254 simulazioni su dataset diversi, 54 frames siano inviati con un ritardo di -2 microsecondi, 30913 con un ritardo di -1 microsecondi, 494242 senza ritardo, 260012 con un ritardo di 1 microsecondo, ecc...

Andiamo ora ad analizzare meglio i nostri risultati ottenuti, andando a graficare l'output dello script:

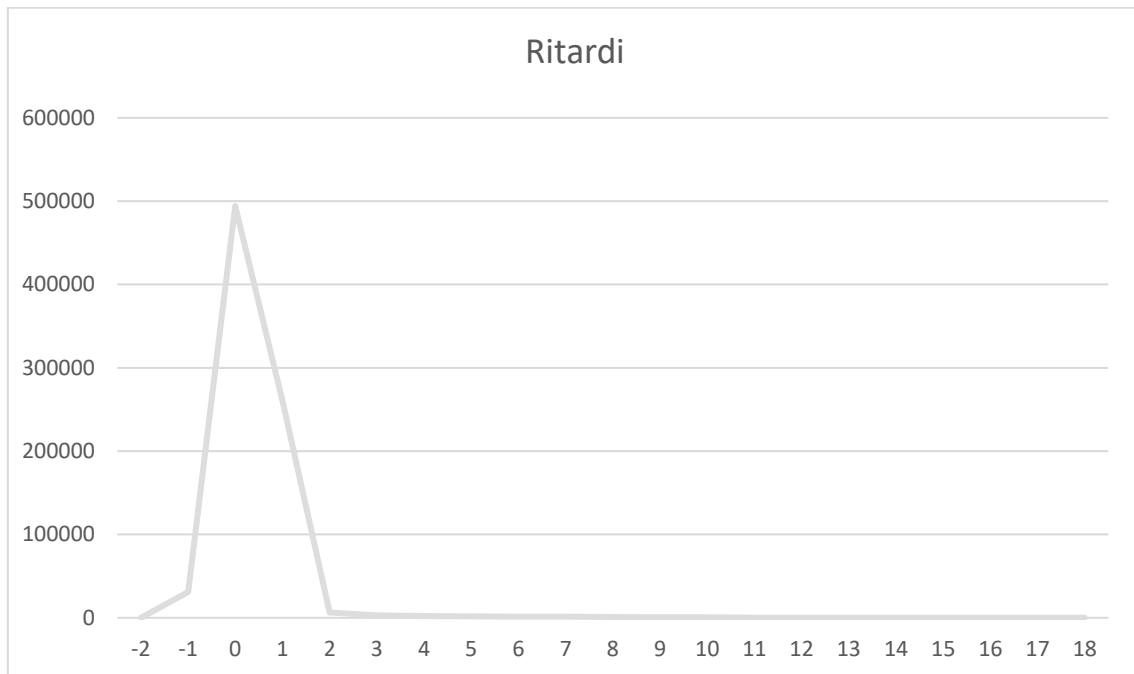


Fig. 27: Grafico risultati

Questo grafico riporta sull'asse delle ascisse il valore in microsecondi del ritardo che intercorre tra l'invio di un frame e il successivo. Sull'asse delle ordinate, invece, abbiamo la frequenza, cioè il numero di occorrenze, con cui si presenta un ritardo con un determinato valore.

A colpo d'occhio possiamo notare come la maggior parte dei frames venga inviato in un intorno di 0 microsecondi, cioè senza ritardo o con ritardo minimo. Andremo comunque a dettagliare meglio i risultati con i prossimi grafici.

Riscriviamo in versione tabulare i risultati ottenuti dall'analisi effettuata dallo script nell'immagine che segue.

Valore ▼	Occorrenze ▼	Percentuale Occorrenze ▼	Distribuzione normale ▼
-2	54	0,006707908	0,023690592
-1	30913	3,840028819	0,150931451
0	494242	61,3949964	0,363506772
1	260012	32,29882487	0,330958993
2	6448	0,800973889	0,11391077
3	2738	0,340115774	0,014821232
4	2168	0,269310079	0,000729009
5	1868	0,232043924	1,35553E-05
6	1531	0,19018161	9,52832E-08
7	1284	0,159499143	2,53193E-10
8	1058	0,131425306	2,54341E-13
9	798	0,099127972	9,65848E-17
10	588	0,073041664	1,38653E-20
11	420	0,052172617	7,52457E-25
12	266	0,033042657	1,54369E-29
13	203	0,025216765	1,19721E-34
14	164	0,020372165	3,51001E-40
15	121	0,015030682	3,89022E-46
16	1	0,000124221	1,62993E-52
17	82	0,010186082	2,58163E-59
18	61	0,007577451	1,54578E-66

Fig. 28: Tabella dei risultati

In questa tabella vengono riportati i dati dei ritardi e delle relative occorrenze, con associati alcuni dati aggiuntivi, quali percentuale di occorrenza rispetto al totale e distribuzione normale.

Ricordiamo che il campione preso in esame ha una dimensione di 805599 frames inviati, da 254 simulazioni eseguite partendo da dataset diversi.

La percentuale delle occorrenze è calcolata come il rapporto tra occorrenza del singolo ritardo e numero di frames totali inviati. Andiamo di seguito a calcolare alcuni parametri per fare un'inferenza statistica.

- Il ritardo medio dell'invio di un frame è 0.431.
- La varianza, approssimata, ha valore 1,027986.
- La deviazione standard ha valore: 1,013896.

Abbiamo una varianza e una deviazione standard alquanto ridotte, sintomo che i ritardi tendono a presentarsi con un valore molto simile alla media.

Per poter graficare la distribuzione di probabilità di un singolo ritardo, andiamoci a calcolare la distribuzione normale di ogni singola occorrenza. Riportate nella tabella precedentemente esposta.

Distribuzione di probabilità

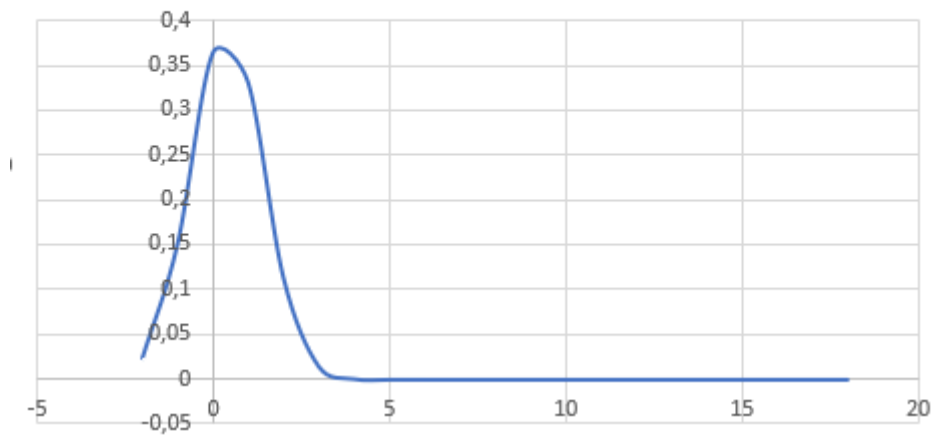


Fig. 29: Grafico distribuzione di probabilità

Come aspettato, la distribuzione di probabilità, ci mostra come la probabilità che un frame venga inviato senza ritardo sia preponderante. E allontanandoci dalla media, la probabilità che avvenga ritardo cali rapidamente.

Trattandosi di un'esecuzione su di un sistema operativo non real-time, rimandiamo alle informazioni sulla macchina su cui è stato eseguito il progetto, riportate nel capitolo 3.1.1.

Al momento dell'esecuzione della simulazione lo stato dei processi e delle risorse è il seguente:

- 417 Processi in esecuzione
- 3,48 GB di RAM utilizzata su 16 GB

Conclusioni

Lo scopo del progetto era sviluppare un software per replicare nel modo più fedele possibile un dataset su di una socket virtuale CAN.

Il tool sviluppato riesce ad inviare oltre il 95% dei frames con un ritardo compreso nell'intervallo -1 microsecondi, +1 microsecondi.

Un risultato soddisfacente considerando il sistema operativo general purpose su cui sono stati eseguiti i test. Non siamo riusciti ad ottenere un comportamento deterministico riguardo alle specifiche temporali, ma statisticamente rilevanti.

Il tool è stato sviluppato e testato avendo risorse materiali limitate. Sarebbe necessario riuscire ad ottenere dati sulle prestazioni dello stesso quando eseguito su diverse architetture con risorse differenti.

Il software è stato sviluppato prevedendo la definizione di parametri che permettano di adattarsi a differenti comportamenti del sistema operativo su cui è in esecuzione.

Considerando la natura open source del progetto, lo sviluppo rimane aperto ad eventuali nuove features e migliorie.

Come si è partiti da un tool open source per implementare questo progetto, in futuro si potrebbe partire da questo tool per eseguire il porting su di un sistema operativo real-time. Riuscendo così ad ottenere precisioni temporali migliori.

Oltre a questo, si è voluto anche dare uno spunto di riflessione sulle tecnologie, ed eventuali rischi relativi alla sicurezza, che stanno abbracciando un settore che impatterà fortemente sulla vita quotidiana di tutti i giorni.

Si è voluto dare una panoramica sulla complessità a cui si può andare in contro anche solo per sviluppare un tool che aiuti a studiare certi temi complessi.

Concludendo, la soluzione proposta rappresenta soltanto un primo mattone, molte ancora le migliorie necessarie affinché il progetto si possa adottare come strumento per svolgere simulazioni realistiche e deterministiche.

Ringraziamenti

Bibliografia

- [1] *“Standard Predefined Macros”*, GCC GNU org, available at: <https://gcc.gnu.org/onlinedocs/gcc-3.2.3/cpp/Standard-Predefined-Macros.html>

- [2] Can-utils source code, GitHub retrieved from: <https://github.com/linux-can/can-utils>

- [3] *“The Single UNIX ® Specification”*, Version 2, Copyright © 1997 The Open Group, available at: <https://pubs.opengroup.org/onlinepubs/7908799/xsh/systime.h.html>

- [4] Linux Kernel, *“can_frame Struct Reference”*, available at: https://docs.huihoo.com/doxygen/linux/kernel/3.7/structcan_frame.html

- [5] Hacking and Countermeasure Research Lab, *“Car-Hacking Dataset for the intrusion detection”*, available at: <https://sites.google.com/a/hksecurity.net/ocslab/Datasets/CAN-intrusion-dataset>

- [6] *“Fuzzing”*, available at: <https://owasp.org/www-community/Fuzzing>

- [7] Staffan Nilsson, *“Controller Area Network - CAN Information”*, available at: <https://www.staffannilsson.eu/developer/CAN.htm>

- [8] Marco Guardigli, *“Hacking Your Car”*, 2010, available at: <https://marco.guardigli.it/2010/10/hacking-your-car.html>

- [9] *“The CAN Bus Protocol Tutorial”*, available at: <https://www.kvaser.com/can-protocol-tutorial/>

- [10] Wikipedia, "*CAN bus*", available at: https://en.wikipedia.org/wiki/CAN_bus
- [11] Craig Smith, Ignacio, "*Il manuale dell'hacker di automobili | Guida per il penetration tester*", trad. it. Virginio B. Sala, Edizioni Lswr, Milano 2016 (ed. orig. *The Car Hacker's Handbook | A Guide for the Penetration Tester*, No Starch Press, Inc, San Francisco 2016)