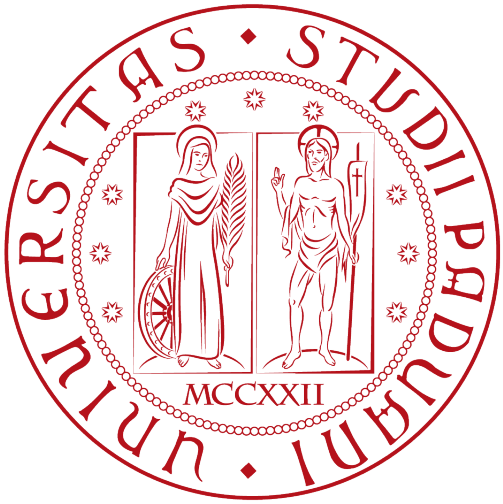
Università degli Studi
di Padova



DIPARTIMENTO
MATEMATICA
DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

A.A. 2021-2022

Cybersecurity: Vulnerabilità nei Sistemi a Guida Autonoma

Studente
Andrea Polato [Mat. 1201205]

Relatore e Tutor Esterno
Prof. Mauro Conti

Tutor Interno
Prof. Lamberto Ballan

Dedicato a...

Abstract

Le attività svolte durante il periodo di tirocinio si ripropongono di ampliare le conoscenze riguardo al mondo della sicurezza informatica e del machine learning, sfruttando le capacità acquisite nei principali corsi accademici. L'obiettivo del progetto di ricerca è valutare la sicurezza di un sistema a guida autonoma. Il mio contributo è stato quello di provvedere alla configurazione di un ambiente virtuale sul quale condurre dei test, creando situazioni che potessero rispecchiare la realtà, mettendo le basi per le attività future collegate al progetto di ricerca.
Più nello specifico, mi sono occupato di ricreare una situazione di platooning tra veicoli intercomunicanti e ho contributo allo sviluppo di un sistema di sicurezza contro attacchi che possono prendere di mira il comportamento di un veicolo facente parte del gruppo.

Indice

Capitolo 1

Introduzione

1.1 Sistemi di Guida Autonoma

Un sistema di guida autonoma è un agglomerato di hardware e software che garantisce un certo livello di autonomia nella guida di un veicolo.
Esistono 6 livelli di guida autonoma, ordinati in maniera crescente in base alle capacità di svolgere attività senza l'ausilio di un essere umano. I primi 3 livelli implicano che debba esserci l'interazione di una persona in alcune situazioni. Gli ultimi 3 livelli invece forniscono delle funzionalità tali da non richiedere l'intervento umano, nonostante questo sia comunque attuabile su volontà di chi è al volante.
Nella realtà attuale, le case automobilistiche hanno raggiunto il livello 2, e molte di queste promettono un'evoluzione al livello 4 entro i prossimi 10 anni. In ottica che questo livello benga effettivamente raggiunto, è interessante analizzare le possibili vulnerabilità di un ADS, al fine di migliorarne il prima possibile la sicurezza.

1.2 L'obiettivo del lavoro

Nota per la commissione

L'obiettivo iniziale sarebbe stato quello di costruire un modello in scala di auto a guida autonoma, per poi mettere alla prova la sicurezza di questo veicolo e tentare di migliorarla. Tuttavia, a causa di alcuni problemi con il reperimento dei componenti, le attività di stage avrebbero subito un ritardo di almeno 3 settimane, rendendo impossibile svolgere quanto pattuito in fase di pianificazione del lavoro. Per questo motivo, abbiamo deciso di rimodulare le attività previste dal piano di lavoro, mantenendosi comunque nella stessa sfera di apprendimento e impostando tutti gli obiettivi precedenti come obiettivi opzionali, così da poter contribuire anche all'obiettivo originale nel caso del raggiungimento anticipato degli obiettivi attuali.

L'obiettivo del tirocinio è stato quello di ricreare una situazione di platooning

mediante un simulatore, per poi poter creare un sistema di sicurezza basato sull'intercomunicazione dei veicoli. In questo modo, ho potuto fornire un ambiente virtuale che potesse replicare situazioni realistiche con un dispendio economico completamente nullo. Il concetto di platooning e il funzionamento del simulatore verranno approfonditi successivamente.
Tramite questo progetto ho potuto approfondire la mia conoscenza di Python e di aclune librerie. Inoltre, ho avuto modo di imparare molte cose riguardo la realtà dell'automotive, a me completamente estranea fino ad ora.
Purtroppo, a causa dei requisiti onerosi del simulatore, le attività si sono svolte totalmente da remoto, in quanto non erano disponibili dei PC abbastanza potenti in sede. Ad ogni modo, questo non ha mai impedito una comunicazione continuativa con i miei supervisori. Grazie ai contatti che mi hanno fornito e alla loro tempestività nel rispondere ho sempre ricevuto supporto nel momento del bisogno ed è stato facile organizzare le attività da svolgere.

1.3 Contenuti della tesi

Nei prossimi capitoli verranno innanzitutto approfondite le questioni del simulatore (Capitolo 2) e del platooning (Capitolo 3). Verranno poi illustrati gli obiettivi intermedi che ho dovuto raggiungere nel mio percorso e spiegherò l'approccio con cui ho raggiunto tali obiettivi (Capitolo 4). Infine, verranno analizzati gli obiettivi prefissati per lo stage da un punto di vista autocritico, associando una valutazione personale dello stage (Capitolo 5)

Capitolo 2

CARLA Simulator

2.1 Cos'è CARLA Simulator

CARLA Simulator è un simulatore open-source che permette lo sviluppo e il testing di Sistema a Guida Autonoma [*ADS*]. Il software fa uso di Unreal Engine 4 per il rendering grafico e adotta lo standard OpenDRIVE per la definizione di strade e ambientazioni urbane. L'interazione con in simulatore avviene tramite l'esecuzione di script in Python. Di fatto, come si può osservare in Figura 2.1, l'eseguibile di CARLA funge da server e ogni script che viene eseguito è un client che si connette alla relativa porta e compie delle azioni.
CARLA è disponibile per Windows e tutte le distribuzioni di Linux. I comandi che verranno riportati nelle sezioni successive sono tutti corrispondenti ad ambiente Windows.

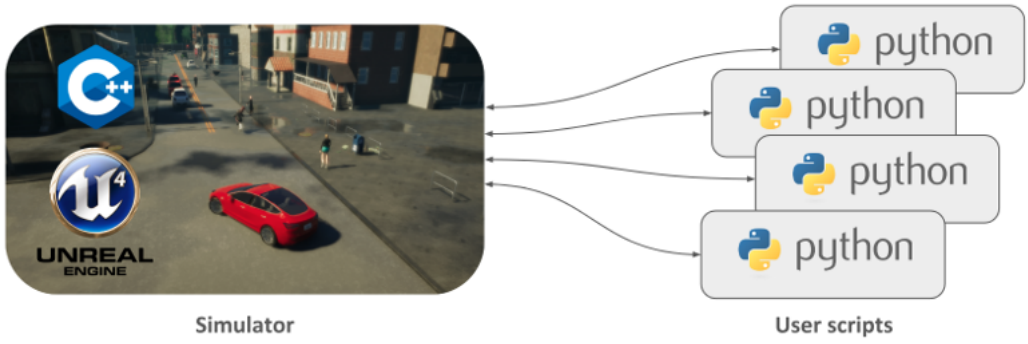


Figura 2.1: schema di interazione tra simulatore e script

2.2 Concetti principali

Al fine di comprendere pienamente l'approccio con cui è stato affrontato il tirocinio è necessario avere ben chiara la struttura del simulatore. I moduli di CARLA più importanti al nostro scopo sono il "world" e gli "actor".
Il **world** è descritto dalla documentazione ufficiale come "*un oggetto che rappresenta la simulazione stessa, è un oggetto astratto che contiene i metodi per generare attori, gestire il meteo, recuperare lo stato degli elementi attivi ecc.*"
Ogni simulazione può avere un solo mondo, il quale viene distrutto e ricreato ogni qualvolta viene caricata una mappa.
Riportando ancora la documentazione, gli **actor** sono invece "*tutte quelle entità che possono giocare un ruolo all'interno della simulazione, vale a dire veicoli, pedoni, cartelli e semafori, sensori e spettatore (la telecamera)*". Ogni attore può essere generato quante volte si desidera, e può essere modificato a piacimento. Per farlo si fa uso dei **blueprint**, ovvero dei prototipi di attore muniti di metodi setter per caratterizzarli liberamente.

2.2.1 Interazione client-server

Come menzionato precedentemente, il software vero e proprio del simulatore funge da server. Di default, si avvia nella porta 2000, ma è possibile sceglierla manualmente. Per farlo basta avviare il simulatore da terminale come di seguito (N è il numero di porta):

```
start .\CarlaUE4.exe --carla-port=N
```

L'interazione vera e propria però avviene sfruttando Python. Si ha infatti la possibilità di scrivere del codice che genera effetti sulla simulazione sfruttando le API di CARLA. Per poter raggiungere il server è necessario seguire due passi obbligatori: connettersi al server utilizzando indirizzo IP e porta e recuperare l'oggetto di tipo world.

```
import carla

client = carla.Client('localhost', port)
client.set_timeout(5.0)
world = client.get_world()
```

Il timeout è necessario al fine di non attendere all'infinito che avvenga una connessione in caso di problemi.
Una volta fatto questo, si possono sfruttare i blueprint per recuperare un modello configurabile di ogni oggetto istanziabile nell'ambiente virtuale. Per la lista di oggetti e relativi attributi rimando alla documentazione ufficiale al link https://carla.readthedocs.io/en/latest/

2.2.2 Le mappe e l'ambiente

Le mappe che CARLA mette a disposizione sono composte da tutti gli elementi che troveremmo nella realtà: strade, attraversamenti, semafori, cartelli stradali ecc. Oltre a questo, c'è la possibilità di cambiare le condizioni metereologiche e temporali, con la conseguente possibilità di studiare il comportamento dei veicoli in più condizioni, considerando anche che CARLA simula anche la fisica stessa di alcuni elementi. Si potrebbe, ad esempio, studiare quanto un veicolo rischia di scivolare in curva in situazioni di forti temporali.
Nel mio caso, la Figura 2.2 mostra com'è strutturata la mappa che ho utilizzato per sviluppare il sistema di platooning, denominata "Town01". Inoltre, ho sempre lavorato con l'impostazione predefinita di tempo e meteo.

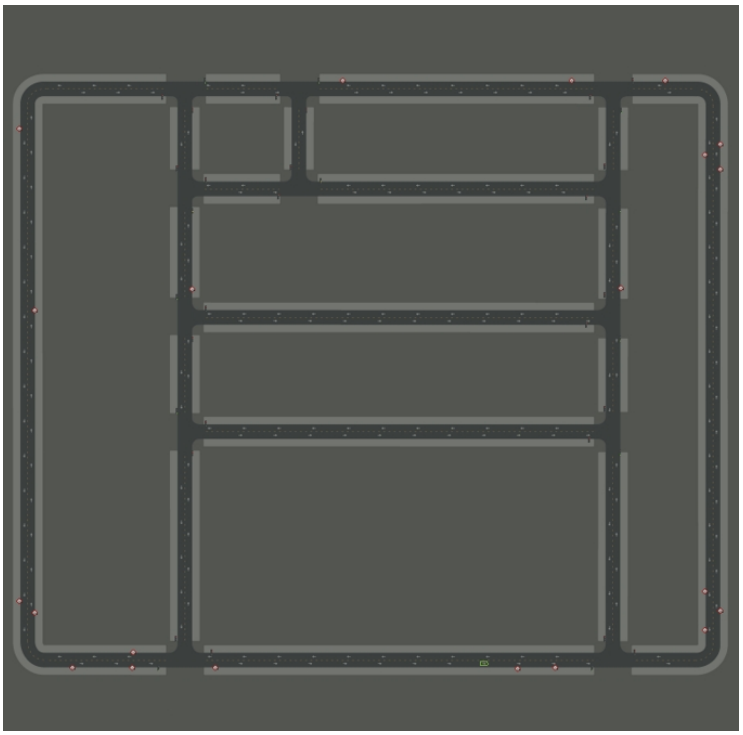


Figura 2.2: vista dall'alto della mappa "Town01"

2.2.3 Il comportamento dei veicoli

Il più grande ostacolo in questo progetto è stato fare i conti con il modo in cui i veicoli si muovono all'interno della mappa. CARLA mette a disposizione due modalità di guida: manuale e automatica. Sulla odalità manuale non c'è molto da dire, tutto avviene esattamente come nel mondo reale: si possono impostare il livello di pressione dell'acceleratore e del freno, impostare l'angolo di sterzata, cambiare la marcia ecc. La guida automatica invece è stato quello che ha reso questo progetto complesso. Essa gestisce il comportamento di un

veicolo come si farebbe in un videogioco, mappando i segmenti di strada con informazioni utili alla circolazione. Tra queste informazioni troviamo limiti di velocità, box che, una volta che il veicolo ci entra dentro, segnalano di controllare lo stato di un semaforo o di un attraversamento pedonale ecc.
Un altro aspetto fondamentale da analizzare è che se per un veicolo controllato manualmente è facile decidere che strada fargli seguire, per un veicolo a guida automatica è praticamente impossibile. Quest'ultimi infatti sono stati concepiti principalmente per fungere da traffico e si affidano a decisioni casuali quando si tratta di scegliere che strada prendere ad un bivio o in un incrocio. Non è nemmeno possibile impostare dei punti di traguardo per questi veicoli, rendendo quindi impossibile ripetere con costanza un esperimento che prevede la navigazione automatica da un punto A ad un punto B.
Il modo in cui ho deciso di affrontare questo problema verrà analizzato nel Capitolo 4.

Capitolo 3

Il platooning

3.1 Cos'è il platooning?

Il platooning è una pratica applicata a gruppi di veicoli per affrontare il problema del consumo di carburante e della congestione delle strade. Per fare ciò, i veicoli vengono disposti in colonna, separati da una distanza tale da favorire:

- la sicurezza dei veicoli in situazioni di emergenza;
- i consumi di carburante, sfruttando fattori aerodinamici;
- il risparmio di risorse economiche, in quanto conseguenza del risparmio di carburante;

Oltre a questo, è bene tenere a mente che le distanze interessate corrispondono sempre al minimo possibile, in modo da garantire la riduzione dello spazio stradale occupato.

3.2 Come si attua?

I veicoli sono interconnessi tra loro e si scambiano informazioni riguardanti l'andamento istantaneo e i dati dell'ambiente circostante raccolti dai sensori. A capo di ogni gruppo di veicoli c'è il **platooning leader**, un veicolo che si può considerare sempre attendibile che fa da capo squadra e si occupa di impostare l'andamento di tutto il gruppo in base alla realtà dei mezzi che lo precedono. Ad esempio, come si può vedere in Figura 3.1, in caso di un ostacolo lungo il percoso il leader avverte tutti i suoi follower e verrà presa la decisione migliore per evitarlo basandosi sui dati raccolti.

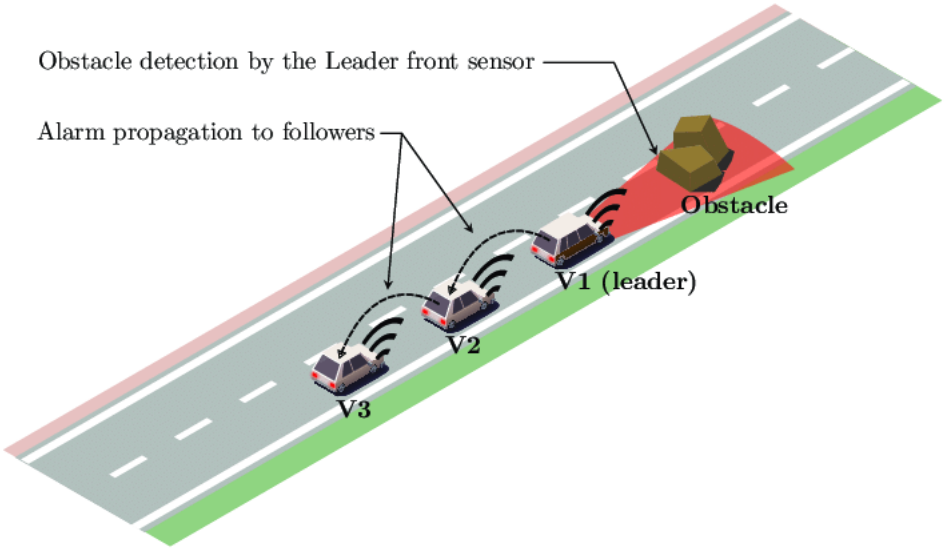


Figura 3.1: il platooning in una situazione di pericolo

Ogni veicolo chiaramente non si basa solo sulla percezione del leader per scegliere che soluzioni adottare nel percorrere la strada. Sono tutti dotati dei propri sensori che, in caso di necessità, possono condurre a comportamenti che hanno il sopravvento sulle indicazioni fornite dal leader.

3.3 L'idea alla base del sistema di sicurezza

L'ipotesi di attacco su cui abbiamo basato il lavoro è quella di un malintenzionato che riesca ad ottenere il controllo di un veicolo del platooning che non sia il leader, conducendolo ad attuare comportamenti indesiderati e/o dannosi.
Il sistema di sicurezza che vogliamo realizzare si basa sui seguenti cardini:

- il veicolo leader deve essere affidabile;
- ogni veicolo facente parte del platooning deve poter accedere alle rilevazioni di tutti gli altri;
- ogni azione intrapresa da veicolo deve essere validata prima di essere eseguita;

L'idea è quella di connettere tutti i veicoli in modo da verificare l'input di movimento ricevuto. Se, ipoteticamente, l'attacante imposta un livello di accelerazione molto più alto di quello dei veicoli che precedono e succedono il soggetto interesato deve rilevare che è molto probabile che ci sia stato un errore di calcolo o un attacco esterno per poi monitorare la situazione e reagire di conseguenza.

Capitolo 4

Il lavoro

4.1 Fase I: attuare il platooning

Come spiegato nei capitoli precedenti, la prima sfida è stata quella di trovare un modo per creare un convoglio di veicoli che seguissero la stessa strada.
Considerato il problema della scelta casuale dei veicoli con l'autopilota attivo, si è presentata una scelta riguardo all'approccio iniziale: usare un veicolo con autopilota come leader oppure costruire un sistema di riconoscimento della corsia e sviluppare il proprio algoritmo di guida automatica. Data la complessità della seconda opzione, ho concordato di proseguire con l'uso di un veicolo con autopilota.
L'idea alla base è semplice: il leader agisce da decisore in situazioni di adiacenza a semafori o svolte, mentre i suoi follower sono veicoli a conduzione manuale. Parto quindi creando la connessione al server e recuperando la libreria di blueprint per poter istanziare gli oggetti desiderati. Faccio inoltre uso di una lista per savare gli oggetti che istanzierò, in modo da distruggerli una volta terminato lo script:

```
import carla
...
actor_list = []
try:
    client = carla.Client('localhost', 2000)
    client.set_timeout(5.0)
    world = client.get_world()
    blueprint_library = world.get_blueprint_library()
    ...
except KeyboardInterrupt:
    pass
finally:
    print('destroying actors')
    for actor in actor_list:
        actor.destroy()
    print('done.')
```

Risulta anche molto utile personalizzare le impostazioni del server. Il simulatore manda un segnale chiamato "tick" che serve a mostrare un'istantanea del mondo in quel preciso momento. Il tempo tra un tick e l'altro è definito dal parametro di "delta_seconds" e, dato che una frequenza di tick maggiore mi permette di effettuare più operazioni di gestione dei veicoli lo imposto al minimo, cioè 0.01 secondi, in questo modo:

```
...
try:
    ...
    settings = world.get_settings()
    settings.fixed_delta_seconds = 0.01
    world.apply_settings(settings)
    ...
```

Proseguo con la creazione dei primi due veicoli da utilizzare e la loro configurazione. Per ogni veicolo è necessario definire un punto di spawn, ovvero il punto in cui verrà generato, e un blueprint per decidere quale sarà il suo aspetto. Nel mio caso, ho deciso di dare al leader l'aspetto di una Tesla Model3 con autopilota attivo e al follower quello di un Audi TT da controllare manualmente. In punto di genereazione viene scelto in maniera casuale tra quelli consigliati da CARLA:

```
...
try:
    ...
    spawn = random.choice(world.get_map().get_spawn_points())
    model3 = blueprint_library.filter('model3')[0]
    audiTT = blueprint_library.filter('tt')[0]
    PlatooningLeader = world.spawn_actor(model3, spawn)
    PlatooningFollower = world.spawn_actor(model3, spawn)

    actor_list.append(PlatooningLeader)
    actor_list.append(PlatooningFollower)

    PlatooningLeader.set_autopilot(True)
    PlatooningFollower.apply_control(carla.VehicleControl(
        throttle=1.0, steer=0.0, brake=0.0))
    ...
```

A questo punto, serve una funzione che possa raccogliere i dati di marcia e di posizionamento dei veicoli che sono attivi nella mappa. Per farlo, ho utilizzato la libreria *csv* di Python, con la quale ho salvato i dati di ogni veicolo in un apposito file csv denominato "vehicle_data_leader" nel caso del platooning leader e "vehicle_data_follower" nel caso del platooning follower.

Innanzitutto è necessario generare il file dentro all directory desiderata e lo si può fare come di seguito:

```
import csv
...
try:
    ...
    dir = 'recs/' + time.strftime("%Y%m%d-%H%M%S")
    with open(dir + '/vehicle_data_leader.csv',
     'w', newline='') as f:
        fn = ['Snap','Time', 'ID', 'Type', 'X', 'Y', 'Z',
         'Km/h', 'Throttle', 'Steer', 'Brake']
        w = csv.DictWriter(f, fieldnames=fn)
        print('Leader CSV file created.')
        w.writeheader()
    ...
```

Ovviamente per il file del follower basta cambiare il nome del file.
Per poi recuperare invece i dati effettivi si sfrutta il segnale di tick precedentemente menzionato e, alla ricezione di questo, si esegue una lambda function che richiama la funzione che ho definito come "record_vehicle_data".

```
...
def convert_time(seconds):
    seconds = seconds%(24*3600)
    hrs = (seconds//3600)
    seconds %= 3600
    mins = seconds//60
    seconds %= 60
    mill = (seconds*1000)%1000
    return "%d:%02d:%02d:%04d"%(hrs,mins,seconds,mill)

def extract_data(snap,vehicle):
    vehicle_snap = snap.find(vehicle.id)
    transform = vehicle_snap.get_transform()
    frame = str(snap.frame)
    time = convert_time(snap.timestamp.elapsed_seconds)
    id = str(vehicle.id)
    type = str(vehicle.type_id)
    x = str("{0:10.3f}".format(transform.location.x))
    y = str("{0:10.3f}".format(transform.location.y))
    z = str("{0:10.3f}".format(transform.location.z))
    vel = vehicle_snap.get_velocity()
    speed = str('%15.2f'%(3.6*math.sqrt(
        vel.x**2 + vel.y**2 + vel.z**2)))
    throttle = str(vehicle.get_control().throttle)
    steer = str(vehicle.get_control().steer)
    brake = str(vehicle.get_control().brake)
```

```
        with open(dir + '/vehicle_data_%s.csv'%(
            'leader' if vehicle==PlatooningLeader else 'follower'),
            'a+', newline='') as f:
            w = csv.DictWriter(f, fieldnames=fn)
            output = {'Snap':frame,'Time':time, 'ID':id,
            'Type':type, 'X':x, 'Y':y, 'Z':z, 'Km/h':speed,
            'Throttle':throttle, 'Steer':steer, 'Brake':brake}
            w.writerow(output)

    def record_vehicle_data(snap):
        snap_list.append(snap)
        for vehicle in actor_list:
            if isinstance(vehicle, carla.Vehicle):
                extract_data(snap,vehicle)
    ...
    try:
        ...
        world.on_tick(lambda snap: record_vehicle_data(snap))
        ...
```

In pratica, record_vehicle_data si occupa di filtrare gli oggetti istanziati per poi eseguire la funzione vera e proprioa di estrazione dei dati, denominata per l'appunto "extract_data". Quest'ultima fa uso estensivo della libreria carla, nello specifico sfrutta molto i suoi metodi getter per poi accedere agli attributi d'interesse. La funzione "convert_time" si occupa semplicemente di formattare il tempo per semplificarne la lettura in fase di analisi.

Glossario

- Termine

Bibliografia

[1] *CARLA Documentation*;
https://carla.readthedocs.io/en/latest/; 19/11/2021;

[2] *Automated Vehicles for Safety*;
NHTSA; https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety;
2022/02/24;

[3]