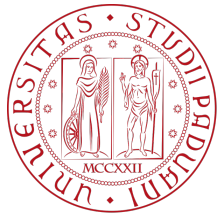


# UNIVERSITY OF PADUA

**Department of Mathematics "Tullio Levi-Civita"**

**Master Degree in Cybersecurity**

**Master Thesis**

# Contextual Based Attack Detection and Resiliency Policies for Autonomous Vehicles Platoons

Supervisor

**prof. Brighente Alessandro**

Co-Supervisors

**prof. Conti Mauro**

**dott. Donadel Denis**

Candidate

**Costantini Lorenzo**

**2029018**

**Academic Year 2022/2023**

# Abstract

The goal of this work is to implement resilience policies for Autonomous Vehicles under attack, based on the context in which the platoon is located. In a platoon model, we define the leader as the fulcrum entity of the whole model, which acts as a guide for the other entities, defined as followers. To allow the vehicles to communicate with each other and exchange the data necessary to establish an autonomous driving scenario, a system of data transmission from and between the entities is implemented. Considering this transmission as unsafe and attackable from the outside, the implemented policies recognize a possible attack and define various rules so that the targeted subject can recover and continue working in a safe driving scenario. The driving contexts are implemented thanks to the Carla simulator and collect the various nuances and meanings of real driving, allowing us to define realistic attacks and scenarios and to observe and analyze the various responses. Each context is therefore repeatedly tested and analyzed, both in situations of regular operation and in the presence of various attacks, to ascertain the functioning and accuracy of the rules above. In conclusion, a report on the mentioned tests is indicated, reporting the precision of the rules and their practical functioning, supported by the data collected during the simulations.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Smart vehicles, equipped with computers and wireless communication devices are emerging on the road. These vehicles are able to drive themselves, communicate with other vehicles, connect to the internet, and provide other services to drivers and passengers. Following the trend of such technologies, a new model for autonomous driving has been developed, called platooning. Platooning brings many benefits. For example, reduced air drag for the vehicles inside a platoon provides better fuel efficiency, especially for heavy-duty trucks [1]. Also, reduced inter-vehicle distance and speed disturbance of the traffic help to achieve both higher road capacity and more comfortable travel experience (e.g., less travel time, less fatigue from driving, higher safety, etc.). Moreover, emerging autonomous vehicles have the capability of sensing the surrounding environment and driving themselves. As a result, platooning software can be installed on such vehicles as a value-added service. In this work, we present a possible solution to the security problems encountered by implementing a platooning model. In fact, despite plenty of benefits resulting from the use of wireless communications in a platoon, it is naturally vulnerable to cyber-attacks, like data injection, replay, jamming, eavesdropping, and spoofing attacks [2]. All these attacks can potentially endanger the stability of the platoon. Moreover, the attacker could be an external or an internal malicious agent. Our goal is to create different rules, able to compensate for the shortcomings of the script itself, and implement detection and resilience techniques, in order to keep an entire platoon under attack operational. To do this, we start from the basics, i.e., we will describe what we mean by autonomous driving, and various progress steps, as reported in the next section. In Chapter 2 we present a brief background on sensors, essential for any reference to autonomous driving. In Section 2.1 there is a brief excursus on

the most used sensors, their operation, and their use. In Section 2.2, instead, we discuss the attacks on these sensors, a fundamental part of the work of developing an autonomous driving mechanism. In Chapter 3, we start talking about platooning, presenting its characteristics in Section 3.1, its structure in Section 3.2 and, finally, some hints about security in Section 3.3. In Chapter 4, we reconnect to the security topic, discussing some useful approaches to fine-tune security in the platooning models and, above all, in determining attacks on the sensors described in Chapter 3. Different techniques are presented, based on complex machine learning algorithms and the like, in order to introduce the reader to some recent solutions. In Chapter 5, we start talking about the work we have done. We begin with a description of the infrastructure used, CARLA Simulator. In Section 5.2.1 we describe the client-server interaction, essential for our work, while in Section 5.2.2 we begin focusing on what we used for our tests. Finally, in Section 5.3 we explain how our platooning model has been implemented inside the simulator. In Chapter 6, we focus more on the technical aspect of the simulation, reporting the data used, the variables mentioned, the links between scripts and, in Section 6.4, we present the scenarios chosen for the simulation. In Chapter 7, we present the defined policies, divided by scenario, while in Chapter 8 we discuss the testing phase, the attacks tried and the results obtained, topped off with a conclusion on the work done.

## 1.1 Autonomous Driving Systems

Autonomous driving has received a unanimous definition, given by the automotive industry itself, and it is "The capability of a car to drive partly or fully by itself, with limited or no human intervention" [3]. An Autonomous Driving System is a mixture of elements, both hardware and software, that helps the vehicle in reaching autonomous behavior during everyday driving scenarios. To date, there are 6 different levels of complexity for Autonomous Driving [4], ordered from the most human-involved one to the more machine-oriented final level, as depicted in Figure 1.1. These six levels are described in the next sections.

### 1.1.1 Level 0 - No Automation

Level 0 is the most diffused example of Autonomous Driving System and it can be resumed in two simple words: Manual Control. In fact, in level 0 all the driving skills and techniques are

controlled by humans, which provide the "Dynamic Driving task" [4], while some additional systems can be implemented to help the driver, like some basic forms of Driving Assistance Systems. Despite that, these systems cannot interfere directly with human driving, instead they can only warn the driver and wait for human intervention, or intervene without taking full control of the vehicle. An example could be emergency braking systems, that don't drive the vehicle but just act on the brakes, not influencing the trajectory of the agent. This level is indicated as 0 because technically there are no autonomous driving mechanisms, so it can not be qualified as automation.

### 1.1.2 Level 1 - Driving Assistance

Level 1 is the lowest form of real automation. Here we can have a more expansive realization for Driving Assistance Systems, including the ones that take control of the vehicle for basic scenarios, like for example Adaptive Cruise Control (ADAC). In this level, the driving task is divided between humans and driving assistants. For example, with ADAC the assistant controls the distance from the next car in order to act on throttle and brake, while the human driver takes care of the steering task, but it is also able to modify and take control of the other driving aspects at any moment.

### 1.1.3 Level 2 - Partial Driving Automation

With level 2, we define the concept of Advanced Driving Assistance Systems (ADAS). This system can fully control the trajectory of the agent, by intervening on the steer, throttle, and on brake mechanisms. ADAS aggregates assist the driver to avoid collisions and maintain control by giving warning signals and using detection mechanisms like Lane Departure Warning, Blind Spot Detection, and other warning systems [5]. Human presence is needed, in order to monitor the Automation system's work and take control of the car at any time. Level 2 automation includes the most common and known Tesla Autopilot, which implements all the characteristics described before.

### 1.1.4 Level 3 - Conditional Driving Automation

This one is the most important level of all the layer sequences. In fact, here we define a substantial jump from a technological perspective, in particular because these technologies don't

require human intervention. Level 3 vehicles implement "environmental detection capabilities" [4], which make the vehicle able to take informed decisions for itself. Despite that, the human override in case of an execution problem is needed, so we could say that human intervention in decision is obsolete, but human supervision is still needed. An example of Level 3 automation can be the decision made by the car itself to overtake a slow-moving vehicle in a safe scenario. This is what Audi reach in 2017 with Audi Traffic Jam Pilot, powered by Artificial Intelligence and implemented in the upcoming 2019 Audi A8 [6]. However, the regulatory process in the U.S. classify this technology as Level 2+, meaning that is still classified as a Level 2 technology, but with key hardware and software required to achieve Level 3 functionalities. To date, very few vehicles on the market can boast of having Level 3 Automation Driving Systems (like Mercedes-Benz EQS or Honda Legend), the maximum common level reached (according to the U.S. regulatory process) is Level 2+.

### 1.1.5 Level 4 - High Driving Automation

The main difference between Levels 3 and 4 is that the latter makes the vehicles able to intervene if things go wrong or if there is a system failure. So, in this level general human interaction is not needed in most circumstances, however a human still has to monitor the whole system. Level 4 vehicles exist and are testing in limited areas, according to actual legislation and infrastructure. An example could be Waymo, a Level 4 self-driving taxi developed in Arizona, which collected the impressive results of over 15 billion miles driven using driver-less cars, so without any safety driver in the driving seat [7].

### 1.1.6 Level 5 - Full Driving Automation

Level 5 vehicles do not require any human attention. This means that vehicles do not need steering wheels or pedals, and they will not need any geofencing or restriction, because they will be able to replicate completely what an experienced human driver can do. These techniques are already testing to date, but no solution is available to general public.

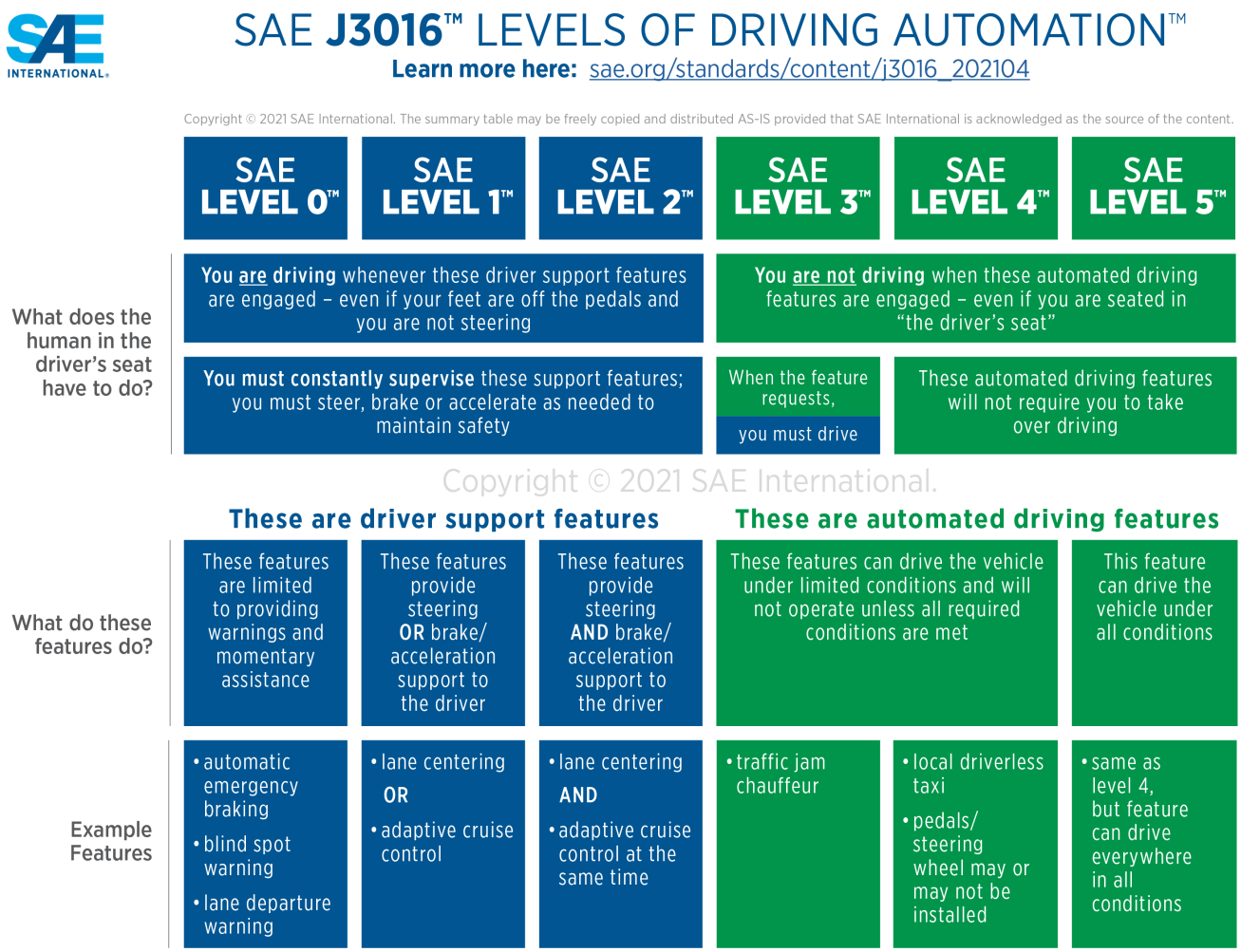



Figure 1.1: SAE Levels of Driving Automation

# Chapter 2

# Background

In this chapter, we will introduce a brief background, regarding sensors and their usage. In Section 2.1, we present a list of the most used sensors in automotive sectors, followed by a singular accurate description. Then, in Section 2.2, we describe the attack surface of each sensor. As we will see, many of these sensors suffer from various attacks, which can partially or completely compromise their functionality. For this reason, it is important to understand the attack surface of each sensor, in order to know the vulnerabilities and study for possible solutions.

## 2.1 Sensors

Different kinds of sensors can be involved in Autonomous Driving models, depending on the necessity and the requirements of the builder or to the level of awareness of the system. We can define a standard list of sensors, that represents the basic one implemented:

- **Camera**: Monitor the vehicle's surroundings (road, vehicles, pedestrians, etc.) and read traffic lights;
- **LiDAR/Radar**: Light Detection and Ranging system. The vehicle uses spinning lasers to generate a point cloud, that gives the car a 360-degree view;
- **Ultrasonic Sensors**: Measure velocity and proximity to nearby objects, as the car maneuvers through traffic;

- **Inertial Sensors**: Measure the fundamental physics of motion, acceleration, and rotational velocity;
- **GNSS**: Global Navigation Satellite System, combined with tachometers, altimeters, and gyroscopes (inertial sensors), pin-point the car in the environment.

In Figure 2.1, it is reported an example of vehicle sensors positioning.

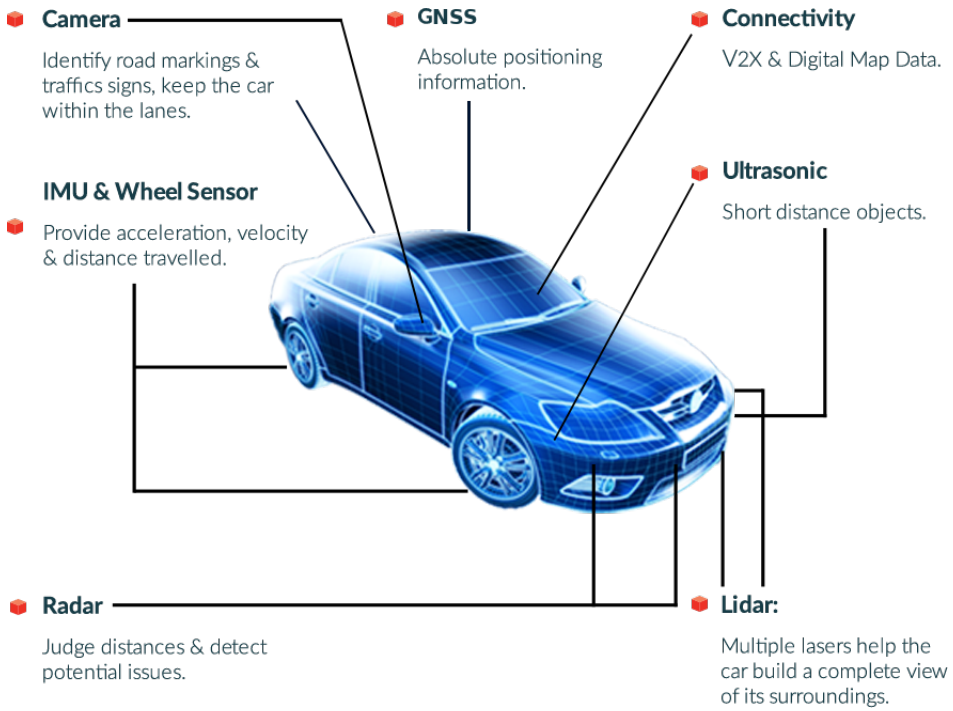



Figure 2.1: Example of sensors positioning (ACAMP, Advanced Technologies Product Development)

In Table 2.1, inspired by [8] and adapted to the case in question, are reported the most important characteristics inherited by this kind of sensor fusion techniques. The table is organized as follows: if a sensor is able to perform a correct and useful measurement under the specific factor, a ✓is reported. If a sensor does not operate well under that specific factor, or if it is not an intrinsic feature of the sensor, a X is reported. Then, if the sensor performs reasonably well under a specific factor, but not excellent, a $\sim$ is reported.

### 2.1.1 Camera

A camera is an optical device that perceives the world around it as a digital video signal. It is a basic sensor for Autonomous Driving systems, as it is used to detect obstacles, traffic signs, lane detection and generally understand the road scenery, by capturing the elements in the area in

Table 2.1: Characteristics inherited.

| Factors | Camera | LiDAR/Radar | Ultrasonic | Inertial Sensors | GNSS |
|---|---|---|---|---|---|
| Range | ∼ | ✓ | X | X | ✓ |
| Resolution | ✓ | ∼ | X | X | X |
| Accuracy | ✓ | ✓ | ✓ | ∼ | ∼ |
| Velocity | ∼ | X | X | ✓ | ✓ |
| Color Perception | ✓ | X | X | X | X |
| Object Detection | ∼ | ✓ | ✓ | X | X |
| Scenario Recognition | ✓ | X | X | X | X |

different frames [9]. The main objective of camera data analysis is to extract interesting regions and features from an image (or a particular frame), then classify them in order to be able to use the analysis for autonomous driving. There are a lot of important parameters to take care of when we implement cameras in an Autonomous driving model. On top of those, there is the quality of the images/videos that the camera can obtain, which can be influenced by lenses, environmental and vehicular conditions (like lack of good lighting), and can truly affect the analysis and the reliability of data. By performing a simple analysis of an image/frame, the camera data analysis is susceptible to many different attacks, this is one of the reasons why it should be used together with other sensors that perform the same, or similar, task.

### 2.1.2 LiDAR/Radar

The "Light Detection And Ranging" or "Laser imaging, Detection, And Ranging" (LiDAR) sensor is part of the range-finding sensors. It works by emitting a light pulse (usually, in the order of infrared) and measuring the time needed to reflect off a surface, so that time is used as a measurement for distance [10]. Radar is a similar technology, that uses microwave pulses and electromagnetic signals. To date, it is preferred to use LiDAR, as it has a higher spatial resolution than Radar, which helps in getting better image reconstruction [11]. The biggest problem with LiDAR and similar technologies is the unsuitability of the sensor in wet scenarios, as the pulses can be absorbed by water molecules. This is one of the reasons why LiDAR should be used together with other sensors. One of the advantages of using LiDAR is the possibility of generating three-dimensional images, by putting the sensor on a rotating pedestal, in order to obtain the geometry of all the objects around, even without good lighting conditions. Paired with camera, it can be used for Collision Avoidance Systems and in tasks that require the analysis of the near environment [12].

### 2.1.3 Ultrasonic sensors

Ultrasonic sensors are used to detect obstacles that are close to the vehicle. The sensor emits an ultrasonic signal, that will be reflected by the obstacle and directed back to the sensor. As they work with close objects, they are usually implemented in ADAS for low speed or for parking help [13]. Ultrasonic Sensors measure the distance to the target by measuring the time between the emission and reception. The distance can be calculated with the following formula:

$$D = 1/2 * T * C$$

where D is the distance, T is the time between the emission and reception, and C is the sonic speed (the value is multiplied by 1/2 because T is the time for go-and-return distance). To prevent the disruption of the ultrasonic signals coming from sensors, it's important to keep the face of the ultrasonic transducer clear of any obstructions, like dirt or other condensation.

### 2.1.4 Inertial sensors

The most famous examples of Inertial sensors are accelerometers and gyroscopes. Accelerometers measure the acceleration of an object (in this case, the car), while gyroscopes measure the rotation rate with respect to a determined rotation axis. Both these sensors transduce inertial forces into measurable electrical signals, determining even small changes in position or acceleration. They are usually based on Micro Electromechanical Systems (MEMS) technologies, which are low-cost, high performance, and high-precision single-crystal silicon sensors [14].

### 2.1.5 GNSS

Global Navigation Satellite System (GNSS) is the basic infrastructure on which different systems are implemented. The most famous and diffused one is American Global Positioning System (GPS), but other systems exist such as the European GALILEO or Russian GLONASS. It uses a network of 24 satellites and 5 communication frequencies, that help in getting a precise real-time position of the vehicle, with a precision in the order of 10 meters [15]. GNSS satellites send navigation messages to the ground receivers, which can then calculate the distance to satellites by using the transmission time. To obtain the correct location, the receiver has to calculate its distance to at least 4 satellites. The messages exchanged between satellites and receivers are not authenticated or signed, because real-time positioning requires fast

data computation. This leads to vulnerabilities and threats [16], which recommends the use of GNSS technologies paired with other sensors (like cameras), in order to verify if the received information is compatible with the actual scenario.

## 2.2 Threats Description

In this section, we will present the most important and diffused attacks on the sensors presented in Section 2.1. Some of these attacks are naive and can be executed by a low-level attacker by using non-sophisticated tools, while some of these attacks require deep knowledge of the sensor involved and expensive and difficult-to-use tools [17, 18]. The attacks are divided into categories by sensors, some of these have the same name, because the effects reported to the sensor are the same. A summary of the treated attacks on sensors is reported in Table 2.2. The table is organized as follows: whether an attack is functional against the specific sensor, a ✓is reported, otherwise a X is indicated.

Table 2.2: Relation between Attacks and Sensors interested.

| **Attack Type** | **Camera** | **LiDAR/Radar** | **Ultrasonic** | **Inertial** | **GNSS** |
|---|---|---|---|---|---|
| Auto-Control | ✓ | X | X | X | X |
| Blinding | ✓ | ✓ | X | X | X |
| DoS | X | ✓ | X | X | X |
| Jamming | X | ✓ | ✓ | X | ✓ |
| Meaconing | X | X | X | X | ✓ |
| Replay | X | ✓ | X | X | X |
| Relay | X | ✓ | X | X | X |
| Spoofing | X | ✓ | ✓ | ✓ | ✓ |

### 2.2.1 Camera

In this section, we will present the most important and diffused attacks on Camera sensors.

#### Blinding

In this attack, the malicious actor wants to compromise the camera functionality by directing a light source (usually a laser beam) focused on the camera lens. This led to camera's complete

blindness, which can make the system recognize a non-real obstacle, or worse to a complete stop and emergency braking. This attack is described and implemented in detail in [18], which is the principal and most important paper on camera and LiDAR attacks, and re-proposed in an interesting and particular way in [19] and [20]. A more complete approach and study on the effect, the techniques, and the hardware details of blinding can be found in [21].

#### Auto-Control

Again, the attacker directs a light source directly to the camera lens, in order to manipulate the auto-controls and avoid the camera from stabilizing on the image. This leads the camera to not be able to reconstruct an image of the scenario. Similarly to blinding, this attack is described and implemented in detail in [18], and re-proposed in [19], with an interesting analysis of its working features and execution problematics.

### 2.2.2 LiDAR/Radar

In this section, we will present the most important and diffused attacks for LiDAR and Radar sensors.

#### Replay

In replay attacks for LiDAR and Radar, the attacker receives and record the legitimate signals emitted by the LiDAR. Then, at a later point in time, when the scenario is changed, the attacker sends back to the LiDAR the recorder signals, in order to let it track and map non-existent objects in the actual scenario. This attack is very difficult, as the "forged" recorder signal has to be sent to the LiDAR very fast, before the legitimate signal, reflected on the real scenario objects, turn back to the sensor. An implementation of this attack is reported in [19], with a detailed description of the equipment used and on the effectiveness of the threat.

#### Relay

Relay attacks are an extension of Replay ones. In this case, the received signals from the LiDAR are sent to a third receiver, far in a different location that, with the delay granted by the transmission, it will send back to the LiDAR the signals, compromising the mapping of the scenario and disrupting the ability of the LiDAR in accurately calculating the distance of

nearby objects. Relay attacks are treated in detail in [18], with an example of implementation in which are described the various trials performed and the equipment used in each of those, in order to make the reader more aware of the correct tools needed to perform a similar attack.

### Blinding

Blinding for LiDAR is similar to the one presented for cameras. The attacker points to the LiDAR a light source with the same wavelength used by the sensor signals (usually in the infrared order). This will create saturation in the analysis of data, which will see a multitude of points and won't be able to reconstruct any useful scenario. This is a very dangerous attack, because it completely compromises the sensor but, other than that, it can not be recognized by humans, as the infrared wavelength cannot be detected by the human eye. An interesting example of how the Blinding attack is performed without the vehicle's occupants' knowledge is reported in [22].

### Jamming

In jamming attacks for LiDAR, the attacker emits a light pulse similar to the ones generated by the sensor, so with the same wavelength (usually in the infrared order), but also with the same frequency band. Similar to blinding attacks, the attacker has to find the correct wavelength but, this time, the aim is to recreate a false signal and not to saturate the sensor. A really complete work on LiDAR jamming attacks and on frequency bands for performing is reported in [23]. A realistic example, based on a low-cost attack environment, based on a simple Raspberry Pi and a low-power laser is reported in [24].

### Spoofing

The aim of the attacker is to make the LiDAR detect non-existent objects or to under/over-calculate the distances from the object. It can be considered as an extension of Relay and Jamming attacks, focused on fooling the sensor by placing specific objects or by miscalculating the distances. The most noted example is in [18], where Petit et al. were able to spoof a LiDAR system and cause it to over-calculate its distance to an obstacle. Related to this, in [22], Shin et al. were able to obtain the opposite effect, where the LiDAR under-calculates the distance to an obstacle.

#### DoS

In a Denial-of-Service(DoS) attack, the attacker injects an enormous number of fake objects into the data collected by the LiDAR, by repeatedly using techniques like jamming or spoofing. The aim of this attack is to generate a number of fake objects bigger than the one that a LiDAR can track, making the sensor unusable or unstable. An example of DoS for LiDARs is reported in [19], where the author, using the technique described before, forced the LiDAR in entering a sort of malfunctioning given the enormous number of forged data detected.

### 2.2.3 Ultrasonic sensors

In this section, we will present the most important and diffused attacks for Ultrasonic sensors.

#### Spoofing

Spoofing for Ultrasonic sensors consists of direct false signals (from a malicious second source) to the legitimate sensor, before the legitimate signals return back, in order to force the sensor to detect objects that do not exist. It can be innovated by injecting not a random signal, but an old one (captured and stored in the past), in order to block the sensor in a sort of "temporal loop". Another approach is to block the legitimate signal by using specific material (like a sound-absorbing sponge), to replace it with the forged signal, removing the problem of replacing the signal in a short time. This technique is called "cloaking". An example of a cloaking attack is reported in [20], where the authors simply place sound-absorbent materials around the obstacles, so that the sensor does not detect them. Similarly, in [25], the authors were able to conceal objects, this time by covering them completely in acoustic foam, fooling the LiDAR sensor. A more general example of a Spoofing attack, that uses the cloaking technique for signal deletion, is reported in [26], where, in an advanced spoofing scenario, the attacker listens for an incoming ultrasonic signal, eliminates the reflected signal by performing a cloaking attack and sends a forged reflected signal back to the sensor.

#### Jamming

Similar to the LiDAR jamming attack, the aim of the attacker is to continuously send ultrasonic signals direct to the sensor. It can try to saturate it or, more difficult, to recreate a false signal. An example of this, applied to Autonomous driving, is the experiment brought by Xu et al. [26],

which perform a jamming attack on a Tesla Model S in the self-driving Autopark and Summon modes, which caused the vehicle to collide with an undetected obstacle. Another example is brought by [27], where the authors overwhelm the sensor, making it unable to accurately gauge the vehicle's distance to nearby objects.

### 2.2.4 Inertial sensors

In this section, we will present the most important and diffused attacks for Inertial sensors.

#### Spoofing

Spoofing for Inertial sensors consists in injecting sound waves (in the range of ultrasonic, over 20kHz) to fool the sensor, by using transducers or speakers. The crucial point is to modify the "Heading angle", a particular angle used to note the direction in which the sensor is turned, used to derive the final decisions of the attacker. To modify this value, two approaches were presented. First of all, the switching attack, where the attacker alternate between injecting different waveforms (of different frequencies) to bring about phase pacing, that causes the heading value to continuously grow. The other approach, called side-swing attack, focuses on increasing and decreasing the injected waveform in order to manipulate the vehicle heading value. An implementation of side-swing attacks is described in [28]. A similar attack, focused on MEMS gyroscopes, is described in [28], where the authors falsify acoustic waves with a frequency matching the load resonant frequency of the system, fooling the sensor [29, 30].

### 2.2.5 GNSS

In this section, we will present the most important and diffused attacks for GNSS.

#### Jamming

In jamming attacks for GNSS, the attacker transmit some noise on top of the legitimate signal, in order to make it unusable for the receiver. It is the easiest attack, because there are on the market many simple devices that can perform this task and they can be placed even in the car cigarette lighter socket, as clearly described in the work by Petit and Shladover [31].

#### Meaconing

In meaconing attacks, the attacker forward to the victim its own computed GNSS signal, in order to make the receiver compute the route to the exact attacker location [32]. This signal is always sent by the attacker with higher power than usual, in order to force the receiver to choose them as legitimate.

#### Signal generation attack (Spoofing)

In this variant of Spoofing attack, the attacker generates a malicious signal (with high signal strength), that has to be transmitted on top of the legitimate one [33], in order to force the receiver in computing the forged coordinates and force it in tracking the route that the attacker wants [34, 35]. It differs from meaconing because the forged signal injected is not the computed position of the attacker, but it is a completely new signal built from scratch.

## 2.3 Threats Taxonomy

In Figure 2.2 is reported a taxonomy graph for the attacks reported in the previous sections. Here, we have grouped the attacks presented in the previous sections into two specific groups, based on the type of exploited vulnerability. The first, larger group includes all attacks that exploit construction flaws, or vulnerabilities inherent in the type of sensor. This may include, for example, the use of a known and fixed wavelength, or the use of sensors particularly sensitive to light. In turn, this group is divided into two subsections, based on the need or not to manipulate the data, in order to exploit the vulnerability of the sensor. The second macro group, however, focuses on data vulnerabilities. In this case, in fact, attacks that act directly on data analysis are included. An example can be the addition of noise (used for GNSS attacks) to the original signal, which does not act directly on the sensor, but on the data collected and analyzed by it.

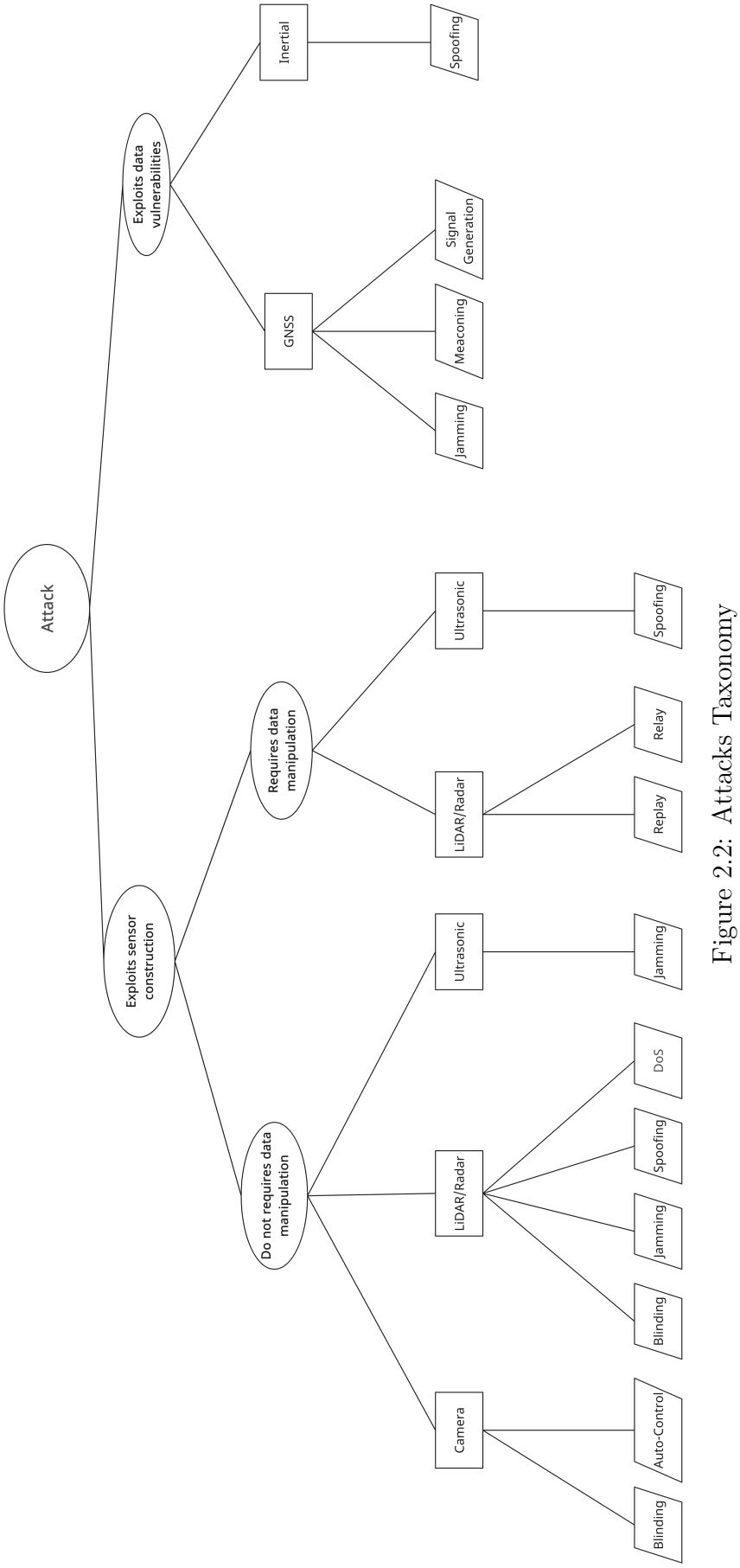



Figure 2.2: Attacks Taxonomy

# Chapter 3

# Platooning

The platooning concept can be viewed as a collection of vehicles that travel together, actively coordinated in formation. Some expected advantages of platooning include increased fuel and traffic efficiency, safety, and driver comfort. There are many variations of the details of the concept such as: the goals of platooning, how it is implemented, mix of vehicles, the requirements on infrastructure, what is automated (longitudinal and lateral control), and to what level [36]. In this chapter we will present the concept of platooning, discussing its advantages and disadvantages (Section 3.1). Then, in Section 3.2 we will present a platoon scheme model, reporting how the vehicles communicate and the technologies used. Finally, in Section 3.3 we will discuss about platoon security and attack surfaces.

## 3.1 Definition

Platooning is an autonomous driving technique, that can be efficiently applied to a small group of vehicles, in order to solve some traffic problems, like road congestion and fuel consumption [37]. A platoon is a group of vehicles that can travel very closely together, safely, even at high speed. Each vehicle communicates with the other vehicles in the platoon. There is a lead vehicle that controls the speed and direction, and all following vehicles (which have precisely matched braking and acceleration) respond to the lead vehicle's movement. This lead vehicle, called Platoon Leader, fulfills the task of directing all the other vehicles in the platoon, called Platoon Followers. These followers will respond to all the lead vehicle's input, transmitted on air.

The first approaches of platooning involved some kind of true mechanic coupling, similar to the train concept, where the Leader and all the Followers are linked through a mechanic joint, through which is passed physical wiring for data transmission. To date, platoon models use modern communication systems, like Bluetooth or 5G, that are able to compute and process the huge volume of data needed to make platooning a safe driving option.

Therefore, platooning techniques are implemented due to their extended list of advantages, like:

- Improving of aerodynamic effectiveness and performance;
- Improving of road capacities;
- Introduces a "Steady-state" traffic flow;
- Reduces fuel consumption, as the vehicles are each vehicle is sort of drafting the following one;
- In a driver-less scenario, autonomous driving vehicles will be able to automatically join platoons based on the pre-computed trip, leaving the human counterpart free to work and use efficiently also the travel time;
- They can reduce accidents, by removing as long as possible the human error in the driving scenario.

Obviously, there are also some downsides for these techniques, like:

- Difficulties in managing long platoons, like in case of changing lanes during highway driving;
- Heavy traffic can be problematic for platoons, for example, a lead car might cross a junction but there might not be enough room on the other side of the junction to accommodate all other cars in the platoon, thus the junction will be blocked;
- They are difficult to retrofit to existing cars;
- Drivers need to be able to leave the platoon at any time and reform into other platoons, or take control completely;

- No computer system is completely secure and platoons could be hacked (Discussed in section 3.3);
- Drivers should be educated on the correct use of platoon.

## 3.2 Structure

In a platoon model, vehicles are interconnected and they exchange information regarding their instant position, speed, and collected environment data, as visible in Figure 3.1. This data are exchanged in order to perform vehicles' basic operations, like acceleration/deceleration, braking, and changing lanes. Other than these, there are other operations that require data transmission, specific for the platoon model [38]:

- Join platoon, it is performed when one vehicle wants to become a member of a platoon;
- Merge platoon, it consists in merging two platoons that have the same destination to form a single platoon;
- Leave platoon, it takes place when one vehicle wants to exit the platoon. In this situation, the vehicles in the head and behind this vehicle open a space for security reasons. When this vehicle leaves, they return back to the platoon distance;
- Split platoon, it takes place when several vehicles want to exit the platoon and create a new one having another leader;
- Cut-in/Cut-out, Cut-in operation is a specific lane change maneuver, in which a vehicle moves closely in front of a platoon vehicle in the adjacent lane. In the case of a cut-out operation, the vehicle entering the platoon decides to leave it. So, the other vehicles have to cover the cut-out vehicle gap.

Data transmission is guaranteed by two main technologies: Vehicle-to-Infrastructure (V2I) communication and Vehicle-to-Vehicle (V2V) communication [38].

### 3.2.1 V2I

This mode of communication is adopted to ensure the exchange of information between vehicles/platoons and the infrastructure. The devices installed along the road, called Road Side

Units (RSUs), represent an example of V2I communication. The infrastructure collects data from vehicles and other information sources, handles it, and makes it accessible to the platoon members in order to facilitate their coordination. This technology is open to modification and customizations, such that many different approaches have been presented. An example could be the work of Böhm and Kunert [39], where the authors introduce a framework for timely and reliable communication both within a platoon and between platoons. Each platoon contains a leader and one or more followers preserving a gap between vehicles and a speed that secures operations of the platoon at the current conditions (such as radio, weather, road, etc.). The communication associated with platoons is ensured using a dedicated Service Channel. The exchange of information between non-platoon vehicles takes place on a separate Control Channel (CCH) without interference. The announcements platoon on the CCH allows the platoon to notify its presence to its neighbors.

### 3.2.2 V2V

This mode of communication is the primordial requirement for the platooning operation. It is adopted for ensuring direct communication between vehicles or platoons. Vehicles driving in a platoon communicate together to keep a distance from the vehicle in front of them. The exchanged messages are disseminated very quickly, such that the vehicles can react in time in case of dangerous situations. We classify V2V communications into three categories:

- Intra-platoon communication: It focuses on the communication between the platoon members;

- Inter-platoon communication: It consists in exchanging messages between vehicles of different platoons. It has much more requirements than the one between the same platoon vehicles;

- Communication between platoon vehicles and normal vehicles: It focuses on exchanging information between platoon members and normal vehicles, not part of the platoon.

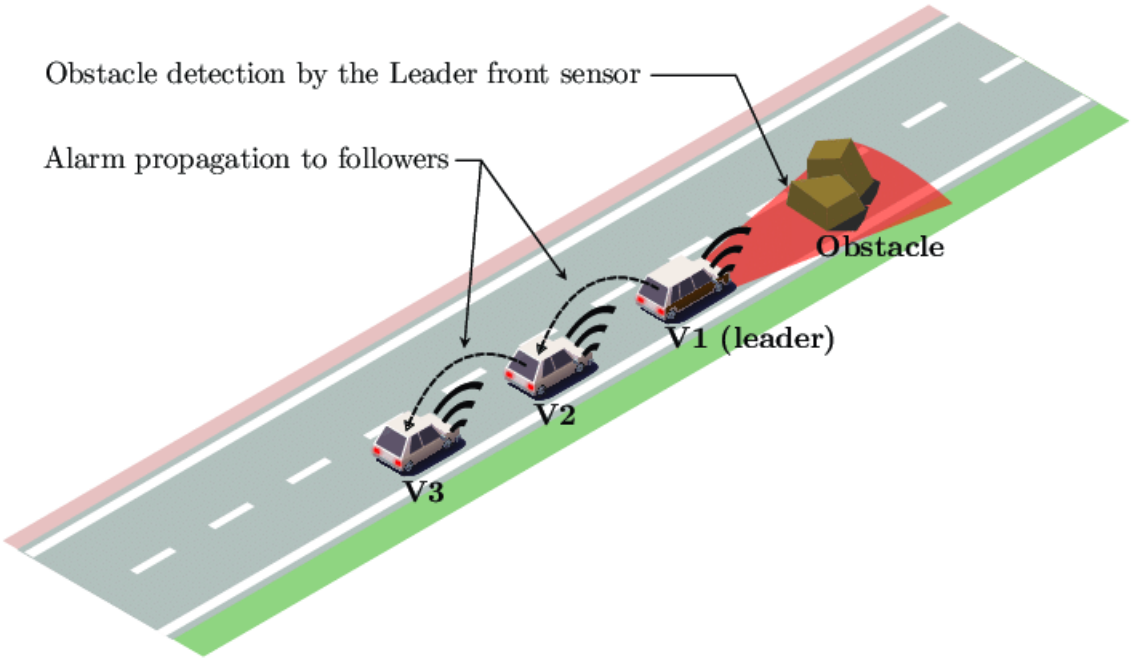



Figure 3.1: Example of platooning communication [40]

## 3.3 Security

The most common attack hypothesis is that a third party (malicious attacker) is able to introduce itself into the platoon system. By doing this, the attacker can take control of a single vehicle, but also of the entire platoon, inducing the vehicles to taking wrong and (possibly) dangerous decisions. The basic idea of Security in a platoon model is based on 4 pillars:

- The leader vehicle must be safe and reliable, so the platoon followers have to test this assumption in order to trust all the received data and instructions;
- Each vehicle in the platoon must be able to have access to the data collected and detected by every other platoon member. Consequently, every platoon member must share or make accessible its own collected data;
- Every action induced by a vehicle has to be tested and analyzed. This rule reconnects to the first one, where we consider the Leader as safe and reliable, but the instructions have to be checked and verified in every case;
- Finally, in a platoon model the only data that should be considered as fully reliable and safe are the "internal ones", so the ones directly collected by the vehicle, that can not be altered by external sources. In this way, the vehicle can perform a cross-check between

the collected and the received "external" data, in order to detect anomalies or wrong instructions.

The security idea is based on the interconnection between all the vehicles in the platoon. In this way, if the attacker focuses on one vehicle (for example, by setting a speed goal higher than the limit and higher than the other vehicles), the faked data are cross-checked with the other "correct" ones, such that the attacked vehicle can recognize the anomaly, labeling it as a simple calculation error or alerting the whole system of a malicious intruder. Finally, the data can be recomputed, correcting it and making the platoon safe again.

To solve problems and attacks aimed at individual sensors, rather than at data transmission, some solutions, which can be summarized in the concept of Sensor Fusion, have been presented, as described in the next Chapter 4.

# Chapter 4

# Sensor Fusion

Sensor Fusion is defined as the process of combining data coming from multiple sensors, installed in cars, machinery, or smartphones in order to compute the correct decisions in a complex scenario, for example, an attack one, with more accuracy compared to using a single data source, based on the analysis of data coming from different pools, instead of relying on a single point of view. In this chapter, we present a survey that analyzes the possible threats on sensors, applied to Autonomous Driving systems, reporting the most important attacks known to date and the approaches that Sensor Fusion introduces to mitigate them. We highlight the advantages and disadvantages compared to classical mitigation approaches, adding a real comparison to the state of the art in this topic. The results presented underline the potential of these new techniques in terms of efficiency, strengths, and costs.

## 4.1 Context

Sensor fusion techniques are used in almost every industry now, starting from factories, going through our smartphones and cars. The importance of sensor fusion is the collection of data coming from a lot of sensors, instead of just an individual sensor. This helps in creating a much better model of the world around, in our case, the vehicle, reporting more precisely possible obstacles and recognizing the different scenarios. While the main goal of sensor fusion is to perform data merging for detection, another skill that the system inherits is to recognize the strength and the weakness of every involved sensor, in order to leverage the strengths and reduce the weaknesses by combining them in a unique data reading. In autonomous driving, usually,

the data coming from a wide variety of sensors are transmitted into a central computing unit to elaborate and digest the data, to estimate the position, speed, and trajectory of the car in order to make more reliable navigation decisions. These sensors are susceptible to different types of attacks, from the most basic ones, focused on simply fooling and triggering a singular sensor, to more advanced one, that can involve more sensors and requires a more complex data elaboration to be found [29]. Other aspects can affect the precision of Sensor Fusion techniques, that do not depend on a third involved party. An example could be sensor degradation, which includes wear and tear of sensors, that can produce incorrect data or non-accurate ones. Therefore, one task that has to be taken into consideration is: how can sensor fusion techniques be used to detect incoming attacks? How can the system isolate one sensor, if it is under attack or degraded, without affecting too much the accuracy of the autonomous driving system?

In this chapter, we present some examples of attacks, focused on sensors used in Autonomous Driving models. Then, different mitigation techniques will be presented. First of all, sensor fusion based ones, which examples will be described in detail. Then, we will present the classical mitigation techniques, based on elaboration and analysis of a single threat for every single sensor. Finally, we will compare these two sets of techniques, by highlighting the strengths and the weaknesses of each method.

## 4.2 Sensor Fusion models

In this section, we present the general skeleton of a Sensor fusion framework. We can distinguish the usage of Sensor fusion techniques in three categories or approaches [41]:

- **Complementary**: in this type of approach, sensors are not dependent one from each other, but when their output is combined, their non-dependence helps in creating a more clear and complete image of the scenario. As an example, the usage of LiDAR and several cameras, placed in different parts of the vehicle focused on different parts of the environment can collectively provide a very complete image of the surrounding of the vehicle, granting a good level of accuracy and a wider view of the scenario;

- **Redundant**: in this type of approach, sensors provide independent measurements of the same target. It can be performed both by fusion of data from different and independent

sensors, both by using different instances of data taken by the same sensor. This technique provides a really high level of accuracy, at the cost of a loss of view of the general scenario;

- **Cooperative**: in this type of approach, different independent sensors provide data that works only combined. This is the most complex technique, but it guarantees the capability of recreating a unique model of the targeted scenario.

Then, after having presented the three categories of use, we can define the three levels of fusion presented in the Multi-Sensor Data Fusion (MSDF) framework [8]:

- **High-level fusion**: in this level, each sensor carries out object detection or any algorithm independently, then fusion is performed. For example, after treated data are collected by sensors, they could pass through a non-linear Kalman Filter that analyzes the whole data in order to detect obstacles or tracking elements [42]. This approach has lower complexity than the other two, but it does not guarantee a high level of accuracy and confidence during classification;

- **Mid-level fusion**: this level, also known as feature-level fusion, fuses multi-target features extracted from the raw measurement coming directly from sensor data. Then, recognition and classification are performed, based on the fused multi-sensor data. It is a good approach to extract all the low-dimensional features from sensors data, but at the same time it provides a limited sense of the environment, losing the contextual information of the scenario;

- **Low-level fusion**: in this level, data from each sensor are fused directly at raw data, without any preliminary elaboration. This helps in retaining all the information of data and it can improve the efficiency and the accuracy of the elaboration. However, it is the most complex one to implement, as it requires a correct calibration of all the sensors involved, but also a perfect time calibration of the captured data.

## 4.3 Countermeasures against attacks to Sensor Fusion

In this section are reported some countermeasures to the attacks reported in Section 2.2, based on the concept of Sensor Fusion. We will highlight the most important features, starting from

the definition of the attack, the sensors involved, the technique of sensor fusion adopted, and the results obtained.

### 4.3.1 GNSS Spoofing Attack Detection Framework

In [43], the authors developed a sensor fusion-based GNSS spoofing attack detection framework, where the data collected by the car sensor and the GNSS data are compared to detect a possible attack. The difference with previous GNSS spoofing attack detection frameworks is that this new solution uses more data sensors, like speedometer, steering wheel angle, and accelerometer, in order to predict the location traveled, focusing in particular on Autonomous driving. Four variants of spoofing attacks on GNSS are presented (also visible in Figure 4.1):

(a) *Turn-by-turn attack*: A turn-by-turn attack is a sophisticated attack because the spoofer has an Autonomous Vehicle's (AV) route and destination information. In this type of spoofing attack, the spoofer takes over a target AV's GNSS computation and changes the AV's current location, resulting in a location shift between the AV's location before and after the attack. Due to a change in AV's current location, the navigation application creates a new route to reach the forged destination, follows the newly created wrong route, and ends up in a wrong, possibly unsafe location, instead of the desired location. The spoofer also tries to keep realistic values of the location shift, AV's speed, and distance between actual and forged route, in order to make it more realistic;

(b) *Overshoot attack*: After taking over the AV's GNSS calculation, the spoofer keeps sending the same location information. As a result, based on the GNSS output, the AV perceives that it is in a standstill state, but the AV is moving forward in reality. As a result, when a road split or intersection occurs, the AV will be unable to identify the path to proceed;

(c) *Wrong turn attack*: The spoofer takes over the target AV's GNSS receiver just before a turn. While the target AV takes a right turn, the spoofer tempers the GNSS signal and the target AV perceives that is taking a left turn. Similarly, if the AV takes a left turn, the GNSS will show a right turn. This will lead to the rerouting of the attacker AV, so the target AV will end up in continuous rerouting, possibly arriving at a wrong destination;

(d) *Stop attack*: Opposite to the Overshoot attack, the spoofer takes over the receiver GNSS when the target AV is stopped, at a stop sign or due to traffic. Then, it transmits a

synthetic signal, such that the AV perceives that is moving along the road.

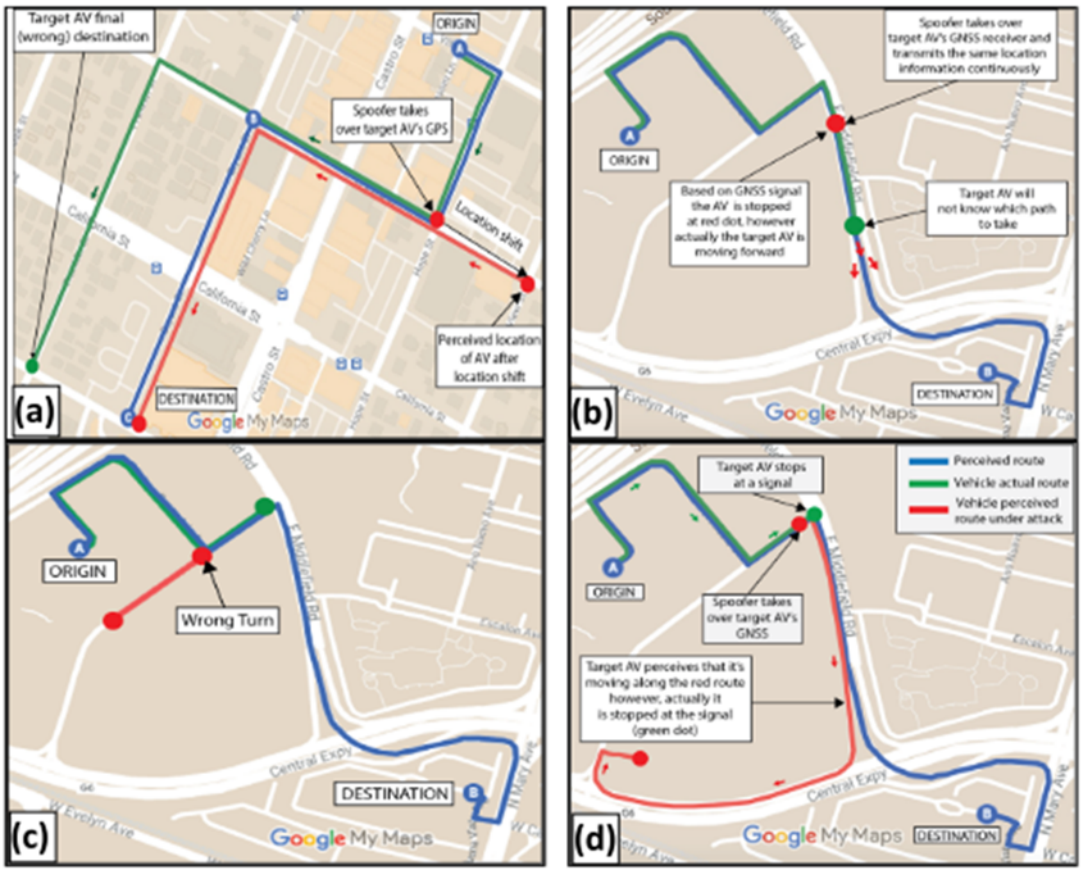



Figure 4.1: GNSS spoofing attack models [43].

The structure of the detection framework can be based on two different strategies:

1. Detection of a vehicle's state using predicted location shift and vehicle motion state, by developing a vehicle state prediction model. This model can predict the subject vehicle's state information (i.e., distance traveled / location shift) by fusing data from multiple in-vehicle sensors (i.e., inertial sensors like accelerometer or gyroscope). For every timestamp, data like speed, acceleration, steering angle, and location shift of previous timestamps are def to train a deep recurrent neural network model, in particular a Long Short-Term Memory (LSTM), that will predict the location shift between two consecutive timestamps. If the difference between the perceived location shift using GNSS and the predicted one is greater than the error threshold, an attack will be detected. The error threshold is given by:

*Error Threshold = Prediction Model Maximum Absolute Error + Positioning Error of the GNSS*

2. Detection of turning maneuvers, where the steering angle data (obtained by using Inertial sensors like gyroscope) can be used to recognize different types of turns (left or right). For example, the steering angle sensor or gyroscope output can provide turn maneuvering data. These data are compared with the information collected by GNSS, to detect and classify turning maneuvers. If a turn is detected using steering angle data, but GNSS shows no turn, then an attack will be detected. Moreover, if the steering angle change represents a right turn and GNSS indicates a left turn, an attack will be detected.

These two techniques use neural networks to digest, classify, and analyze the huge amount of data provided by sensors. In particular, in the first strategy is used an LSTM model, as it requires a deep analysis of data collected, while in the second approach, they use a k-Nearest Neighbour algorithm (k-NN). This algorithm is included in the category of supervised learning classifiers. It uses proximity to make classifications or predictions about the grouping of an individual data point and it is mainly used for classification problems, like in our study case, where the clustering algorithm is used to perform a layered task of labeling and comparison. These two techniques can be implemented singularly in order to satisfy the system's requirements, but the greatest approach is to use them merged, in order to detect also more complex attacks. As it is reported in Table 4.1, the checkmark states that the strategy can detect that specific attack, while the cross states that the attack can fool that strategy. Therefore, the combination of both these techniques is the right way to detect all the attacks presented [43].

Table 4.1: Attack detection.

| **Attack Type** | **Strategy 1** | **Strategy 2** |
|---|---|---|
| Turn-by-turn | ✓ | X |
| Overshoot | ✓ | ✓ |
| Wrong turn | X | ✓ |
| Stop | ✓ | ✓ |

### 4.3.2 CACC based framework

In [44], the authors focused on the concept of Cooperative Adaptive Cruise Control (CACC), used in platooning scenarios, and analyze possible attacks on this technique. The authors implement a Sensor-fusion based framework for detection, improving previous similar models based on a single sensor data analysis, because sensor fusion accounting for spatial information provided by multiple vehicles in the platooning can achieve way better results. The authors present three categories of attacks on sensors:

- **Jamming**: In this case, they focus on LiDAR/Radar jamming, usually caused by interference of a malicious signal, that causes an additional and significant noise at the exploited sensor;

- **Data injection**: hijacks one or some of the sensors and purposely gives predefined false information. They consider a typical data injection, where a ghost vehicle deceives the sensors that the obstacle is at a different distance than in reality. This attack is a variant of Spoofing attack;

- **Sensor Manipulation**: the aim is to undermine a sensor's precision by enlarging its output interval. This can be done by fixing the minimal value while enlarging the maximal value, cheating the car with a larger headway distance to induce a collision. This is a more advanced and complex attack, very unusual because it requires a high knowledge of the sensor itself, but also access to it. This is reported for the sake of completeness.

As before, the authors propose two different strategies to mitigate the attacks, the first based on a naive approach, the second formulated by the authors themselves:

- **Naive and Pairwise Sensor Fusion**: This is the simplest approach. It is based on a simple algorithm, that takes the intersection of all the intervals returned: for a vehicle $i$ with $n$ sensors, the sensor fusion represented is $s_i = \bigcap_{j=1}^{n} s_{ij}$, where $\bigcap_{j=1}^{n}$ is the intersection symbol, for $j$ in the range [1,n]. The authors define abstract sensor models, each one has to contain the ground truth value, a value that, in a healthy sensor, lies in a defined interval. The algorithm performs the intersection of the intervals and returns as output an interval, $s_i$. Then, this result is passed to a second algorithm that adds a temporal information dependence, as proposed in [45], in order to enhance the sensor fusion. This

pairwise intersection technique uses historical intervals from some previous time, maps them to the present with the vehicle's actual dynamics, and intersects the corresponding interval pairs. Then, it performs a final intersection between these intervals to recover the ground truth. The value of this ground truth is used to detect if the model is under attack, or in a safe state;

- **Proposed Approach**: In this approach, based on the platooning technique, the authors supposed that the vehicle $i$ not only receives information in the range $p_{i-1} - p_i$, which refers only to the distance between the car involved and the closer one calculated using the $i$ car sensors, but also from the additional range of information passed by vehicle $i-1$ (i.e., $p_{i-2} - p_{i-2}$), data calculated using the $i-1$ vehicle sensors, and the position of vehicle $i-2$ (i.e., $p_{i-2}$). So, the distance between two cars can be computed as:

$$p_{i-1} - p_i = (p_{i-2} - p_i) - (p_{i-2} - pi - 1).$$

Consequently, using the distance from vehicle $i$ and $i-2$ to subtract all the intervals from the last car, an extra set of intervals for sensor fusion is received. This calculation is called *triangular pairwise intersection*, as it uses the vector difference of two edges in a triangle. This approach helps in reducing the error propagation and lowers the risk of attack, in particular focusing on the lead car. Other than that, this sensor fusion technique is also an effective defense for protecting vehicle $i$, given that vehicle $i-1$ is not under attack, that can eventually be detected by the technique itself.

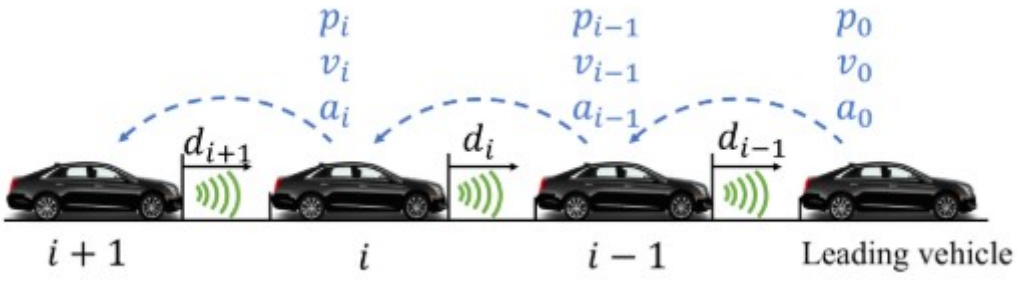



Figure 4.2: Car Platooning order [44].

The data used for these two techniques comes from the Radar, the GPS, and Inertial sensors (velocity and acceleration ones). Both these techniques work well in detecting and recovering

from an attack, but the proposed one is the most efficient one, as it has a smaller positional error in respect to the other, given the fact that it uses a huge amount of data coming from vehicles sensors.

## 4.4 Alternative Countermeasures

Sensor Fusion is not the only solution to mitigate and solve possible threats and attacks. Before that, many other solutions have been proposed, each one with its advantages and disadvantages.

### 4.4.1 Camera

For what concerns cameras, a solution to mitigate attacks can be the so-called "redundancy". In this case, the vehicle uses more than a single camera, where the images of different cameras partly overlap, in order to be able to reconstruct at least twice the same image, to detect possible anomalies. The problem with this ploy is that the load of work in reconstructing and classifying the images and the objects grow exponentially, as the algorithm has to analyze and collect data from more cameras than usual. Then, this kind of countermeasure works for less sophisticated attacks, like blinding with a commercial laser, but they are ineffective for attacks brought with military weapons, like a Dazzler [18], that can fool multiple cameras at the same time. Finally, the cameras have to be coordinated, synchronized, and perfectly calibrated, in order to produce a useful and common data set. Then, there are some software countermeasures for cameras, like separation of channels, spectrum analysis and so on. The problem with these algorithms is that they require time for elaboration, while in an AV the analysis of data should be done in real-time.

### 4.4.2 LiDAR

Talking about LiDAR, more complex countermeasures can be implemented [46]. As cameras, redundancy is a solution. The problem is that a LiDAR can be very expensive, so implementing more sensors than required can be not feasible. Then, there are some countermeasures that are implemented in the sensor operation. First of all, Random Probing, where the pulse is not repeated at a fixed interval, but at a random one, non-predictable. This is suitable to prevent attacks like Jamming or Spoofing, where the attacker has to synchronize on the same interval

of pulses. However, this countermeasure can not be applied for rotating LiDAR, that is the most used for AV, because they require a constant rotation speed and need to know exactly at which angle they fired a pulse. So, to prevent Jamming attacks in rotating LiDAR, Multiple probing can be implemented. In this ploy, the LiDAR performs three measurements at the same position, such that if one of the three is different, they will be considered invalid. Here some other problems arise. First of all, multiple probing reduces the scan frequency, which will reduce the effective different measurements. Then, the measurement should be corrected, as the vehicle can be moving between the three consecutive measurements.

### 4.4.3 Ultrasonic sensors

For Ultrasonic sensors, in [26] two approaches are presented to protect the sensor from jamming and spoofing attacks. The first one introduced the idea of shifting the parameters of waveform, in order to make it possible to authenticate the physical signals. The second approach uses two or more sensors (again, a redundancy scheme) to detect attacks, using overlapping information, with the same advantages and disadvantages discussed before.

### 4.4.4 Inertial sensors

For Inertial sensors, three countermeasures have been presented in [28]. The first one is Damping and Isolation, where are used some noise barriers (like isolating boxes or acoustic foam). The problem here is that this solution does not take care of sizes or, more generally, of the environment in which the sensor is used, such that often this ploy is unfeasible. Then, the second one is called Filtering. It is actuated via hardware, and the main focus is on eliminating the out-of-band noise signal by filtering it, using for example a low-pass filter. This is not a robust solution to the problem, as filters in digital circuits of the sensor will not alleviate totally the problem, as out-of-band analog signals have already been aliased to in-band signals after sampling. Finally, Sampling, which aims is to eliminate the attacker's ability to achieve a DC signal alias and limit potential adversarial control. This is done by adding a randomized delay to each sampling period. This has some negative side effects, like the degradation of the accuracy in measurement.

### 4.4.5 GNSS

For GPS/GNSS, numerous countermeasures have been proposed. They can be based on the signal strength, time interval between signals, distortion of signal, and so on. Other methods focus on the cryptographic primitives defined for transmissions, in order to be able to understand if a signal is forged (like in a spoofing attack) and authentic, by using authentication and digital signatures. The problem with these approaches is the time required to perform the primitives. Usually, they take too long to adapt to a real-time transmission, which makes them unusable for AV.

In Table 4.2 is reported a recap of the effectiveness of countermeasures for all the attacks presented, in order to highlight the approaches most effective (like redundancy, which is useful for most of the attacks). The table is organized as follows: if the presented countermeasure works against every possible scenario and (known) mutation of the selected attack, a ✓is reported. If a countermeasure works for the majority (but not the entirety) of all the possible attack developments, a $\sim$ is reported. Then, if the defense against that specific attack is not an intrinsic feature of that countermeasure, such that is not useful in that attack scenario, a X is reported. The biggest advantage of using Sensor Fusion approaches to mitigate attacks, over the presented solution, is the possibility of protection and detection against a huge number of attacks, by using just a system instead of implementing all the one defined. In fact, by using the correct and adequate algorithm, sensor fusion approaches can be effective against a multitude of attacks and, at the same time, they can inherit characteristics and properties of singular sensors to be used to refine the security systems and the elaboration one, as it was reported in Table 2.1.

Sensor fusion has also some disadvantages. First of all, it requires a correct setup and initialization of all the sensors, starting from calibration to correct synchronization. Then, it requires some important computational power in order to elaborate data. For example, in the cases presented in Section 4.3, some variants of LSTM or deep neural networks are used, which requires high computational power, especially for elaborating the huge amount of data coming from combined sensors.

Table 4.2: Countermeasures to attacks

| Sensor | Attack | Redundancy | Spectrum Analysis | Random Probing | Multiple Probing | Shift Waveform Parameters | Damping | Filtering | Sampling | Signal Analysis | Cryptographic Primitives Implementation |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Camera | Auto-Control | ✓ | ✓ | X | X | X | X | X | X | X | X |
| | Blinding | ✓ | ✓ | X | X | X | X | X | X | X | X |
| LiDAR | Blinding | ✓ | X | X | X | X | X | X | X | X | X |
| | DoS | ✓ | X | ∼ | ✓ | X | X | X | X | X | X |
| | Jamming | ✓ | X | ∼ | ✓ | X | X | X | X | X | X |
| | Replay | ✓ | X | X | X | X | X | X | X | X | X |
| | Relay | ✓ | X | X | X | X | X | X | X | X | X |
| | Spoofing | ✓ | X | ∼ | ✓ | X | X | X | X | X | X |
| Ultrasonic | Jamming | ✓ | X | X | X | ✓ | X | X | X | X | X |
| | Spoofing | ✓ | X | X | X | ✓ | X | X | X | X | X |
| Inertial | Spoofing | X | X | X | X | X | ✓ | ✓ | ✓ | X | X |
| GNSS | Jamming | X | X | X | X | X | X | X | X | ✓ | X |
| | Meaconing | X | X | X | X | X | X | X | X | ✓ | ✓ |
| | Spoofing | X | X | X | X | X | X | X | X | ✓ | ✓ |

# Chapter 5

# CARLA Simulator

## 5.1 What is CARLA Simulator

CARLA (Car Learning to Act) simulator is an open-source simulator, used for the development and testing of Autonomous Driving Systems (ADS) [47]. This simulator is based on Unreal Engine 4 for graphic rendering and it uses OpenDRIVE standard for road and environment definition. This simulator is a joint work of Intel Labs, Toyota Research Institute, and Barcelona Computer Vision Center.

To interact with the simulator, the user should execute a Python script, as it can be seen in Image 5.1. So, CARLA Simulator acts as a sort of server, to whom each script connects (by using a specific port). Now, script and simulator are free to have a dual communication, which can be graphically seen, both by using the integrated GUI or by reading logs on the shell.

CARLA Simulator is available both on Linux and Windows distributions, in this work we use CARLA 0.9.13 (the latest available at the start of testing) working on Ubuntu 18.04 LTS Bionic Beaver. In the next sections, we will describe some basic commands, in order to explain the communication scheme and the realization scheme of all the used scripts.

## 5.2 Basic Structure

To help understand how the script works, in this section it is presented the structure, both of the script interaction and of the commands used. To obtain the simulation, we will import and use three main modules, "World", "Client", and "Actor".

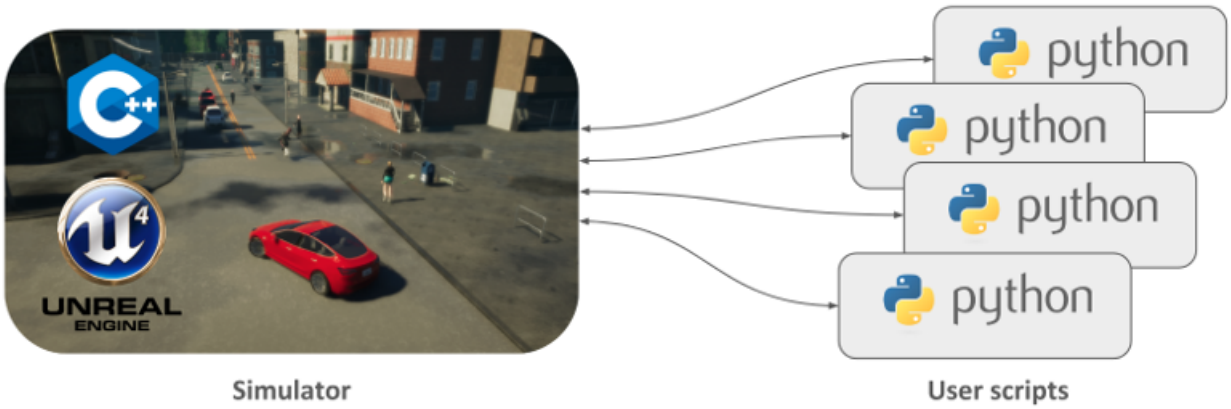



Figure 5.1: Interaction scheme between simulator and script

World is described in the official documentation as *"an object representing the simulation. It acts as an abstract layer containing the main methods to spawn actors, change the weather, get the current state of the world, etc. There is only one world per simulation. It will be destroyed and substituted for a new one when the map is changed"*.

Again, in documentation, the description of Client is *"the module the user runs to ask for information or changes in the simulation. A client runs with an IP and a specific port. It communicates with the server via the terminal. There can be many clients running at the same time. Advanced multiclient managing requires thorough understanding of CARLA and synchrony"*.

Finally, Actors are described as *"An actor is anything that plays a role in the simulation"*, and some examples are reported, like vehicles, walkers, sensors, traffic signs, traffic lights, and the spectator, that is the camera that shows the entire simulation. Each one of these actors can be generated multiple times, in order to fit our scenario. To do this, we use blueprints, that are are already-made actor layouts necessary to spawn an actor. Blueprints are models with animations and a set of attributes. Some of these attributes can be customized by the user, while others don't. There is a Blueprint library containing all the blueprints available as well as information on them.

### 5.2.1 Client-Server Interaction

As said in Section 5.1, CARLA software act as a sort of server. By default, it uses port 2000 for script interaction, but it can be changed manually in the settings file, or directly in the shell instruction, where N is the port number:

```
./CarlaUE4.sh --carla-port=N
```

Now, we know the port on which the server is listening. We need to connect our Python script to the same port, taking advantage of the possibility to write code that modifies the simulation, thanks to CARLA APIs. To connect and establish a Client-Server Interaction we need to follow two main steps: connect to the server, using the provided IP address (localhost) and port, then we should recover the world object of the simulation:

```
import carla

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

We set a timeout in order to not wait infinite time for a connection, in case of problems. This value is not fixed and it was set by computing the usual time needed for a normal Client-Server connection. Then, each time we load the Town that we need, we reload it (in order to remove previous instances) and finally get the World object.

Once we have set up correctly the connection, the Client-Server interaction and we retrieve the World object, we could start to use the blueprint library to add Actors to the simulation. We could create multiple instances of Actors objects, by modifying the attributes or directly the Actor type.

For other information about Carla configuration, commands, and libraries, we refer to the official documentation following the link: `https://carla.readthedocs.io/en/latest/`

### 5.2.2 Maps

CARLA simulator provides 8 different maps, some integrated in the simulator package, others as an add-on package, that has to integrated in a second time. CARLA maps are realized in order to provide all the elements that we could find in a real road, like crossroads, stoplights, lines, traffic signals, and similar customizable elements. We could change also the environment itself, like the weather conditions, in order to find the ideal test scenario. All these customizations help in studying the physics and the behavior of the vehicle in different scenarios, which may vary

even from one iteration to another. For example, in a simple turn, the trajectory is different between each iteration, influenced by external and environmental factors, which makes the simulator closer to real driving behaviors.

The maps provided have different layouts, each one dedicated to different driving situations and traffic scenarios. A brief description of each map is presented in Table 5.1, as referred in the official documentation:

Table 5.1: CARLA maps description

| **Town** | **Summary** |
|---|---|
| Town01 | A basic town layout consisting of "T junctions" |
| Town02 | Similar to Town01, but smaller |
| Town03 | The most complex town, with a 5-lane junction, a roundabout, unevenness, a tunnel, and more |
| Town04 | An infinite loop with a highway and a small town |
| Town05 | Squared-grid town with cross junctions and a bridge. It has multiple lanes per direction. Useful to perform lane changes |
| Town06 | Long highways with many highway entrances and exits. It also has a Michigan left |
| Town07 | A rural environment with narrow roads, barns and hardly any traffic lights |
| Town10 | A city environment with different environments such as an avenue or promenade, and more realistic textures |

Each map is made by grouped objects, called layers, that consist in:

- NONE
- Buildings
- Decals
- Foliage
- Ground
- Parked Vehicles
- Particles
- Props
- Street Lights

- Walls
- All

CARLA simulator provides also the possibility to import custom maps and assets, by using a map configurator. In Image 5.2 is presented an example of a map basic layer, in particular Town01 structure, the less complex one. The red dots represent the traffic signals' position on the map.

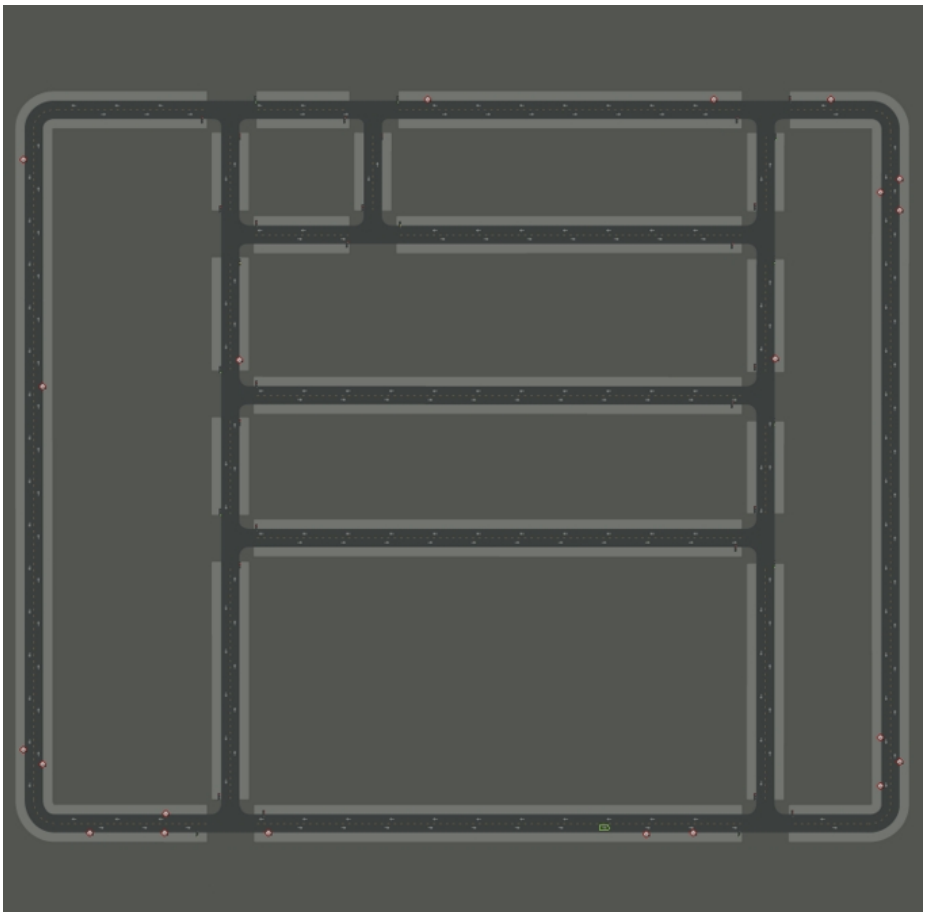


Figure 5.2: Town01 structure

### 5.2.3 Driving modes

CARLA simulator provides two different driving modes: Manual driving, where the user can interact directly with the car and drive it using the keyboard, and Autonomous driving.

This last mode is the one that interests us more. In Autonomous driving mode, the simulator manages both the car response (accelerator, brake, steering), both the traffic and road segment of the map. So, the simulator manages to drive the car respecting all the limits and signals,

but it also raises alerts in case the vehicle does something unusual, for example, line crossing or eventual crashes.

In Autonomous mode, the user can not indicate the trajectory and the route to follow to the vehicle, which is completely managed by the simulator itself, and this is a major limitation of this simulator. However, for the scope of this research, we managed to retrieve the useful data anyhow, by analyzing more trajectories and more behaviors when needed.

## 5.3 Platooning implementation

The focus of this first part of the work is to create a way for a convoy of vehicles, that follow the same route. I decided to use the Autonomous driving mode, both to re-create a futuristic platoon scenario, both to try to create a platoon able to adapt to all the possible (also millimeter) variations in driving trajectory. So, the platoon is made by a Leader, to which is toggled the auto-driving mode, followed by two vehicles that follow the instruction and data received by the leader. So, the basic idea is to use the leader as the one that takes the driving decisions, for example, which turn to take or which direction to take at a crossroad, while the two followers are manual-driving oriented, where the input is elaborated and rearranged thanks to the leader data transmission. So, after the connection to the server presented in Section 5.2.1, the script continues as:

```
import carla
...
actor_list = []
blueprint_library = world.get_blueprint_library()
..
except KeyboardInterrupt:
    pass
finally:
    print('destroying actors')
    client.apply_batch([carla.command.DestroyActor(x)
                                for x in actor_list])
    print('done.')
```

The simulator works by sending a signal, called "tick", at every fixed interval, in order to show the world situation at each tick iteration. We define this fixed interval by modifying the

value `delta_seconds`. In this case, we set the interval to the minimum value possible (0.01 seconds), in order to get a higher tick frequency and be able to make more operations possible.

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

So, here we obtain and save all the settings already defined, in the settings file or by previous iterations, modify the `delta_seconds` parameter and finally update all the settings and apply them to the world.

Then, we should generate the vehicle for the platoon, with their configurations. For every vehicle, we need to define a spawn point, the point in which the car will appear in the world, defined by the three-dimensional parameters x,y,z, and other parameters, like pitch and roll, to establish the rotation and other aspects of the generation. Once the spawn parameter is defined, we need to use the blueprint library in order to define the vehicle aspect. In our case, we use the Tesla Model 3 as template for the platoon vehicles, as they represent the most diffused type of semi-autonomous driving mode in the actual market [48, 49]. The spawn point change for every different map.

```
...
try:
    ...
    spawn = carla.Transform(carla.Location(x=-XXX, y=XXX, z=XXX),
            carla.Rotation(pitch=XXX, yaw=XXX, roll=XXX))
    model3 = blueprint_library.filter('model3')[0]

    PlatooningLeader = world.spawn_actor(model3, spawn)
    spawn.location.XXX += XXX
    PlatooningFollower = world.spawn_actor(model3, spawn)
    spawn.location.XXX += XXX
    PlatooningFollower2 = world.spawn_actor(model3, spawn)


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

```
17      actor_list.append(PlatooningFollower2)
18
19      PlatooningLeader.set_autopilot(True)
```

Now, we have created the vehicle platoon. The next steps are to define the driving functions, the communication method between cars, implement useful sensors, and finally define the rules and policies needed to control the platoon in case of attack detection. In Figure 5.3 is visible the platoon implementation in CARLA simulator, with the three vehicles (Tesla Model 3) highlighted with different colors, in order to distinguish them.

Figure 5.3: The platoon representation in CARLA simulator

# Chapter 6

# Simulation description

## 6.1 Introduction

The goal of this work is to implement resilience policies for Autonomous Vehicles under attack, based on the context in which the platoon is located. We consider the data transmission between platoon vehicles as unsafe and attackable from the outside. The implemented policies have to recognize a possible attack and define various rules so that the targeted subject can recover and continue working in a safe driving scenario. These rules and policies are defined based on the driving functions implemented for the platoon, such that every possible difference with the expected behavior is highlighted and analyzed.

Our focus was on implementing rules and policies, able to work also in a (relatively) low-cost environment. In fact, other solutions to this kind of attacks have been presented, like [50, 51, 52], but they all have in common the use of machine learning for data digestion (that is very computational demanding) and the usage of lots of sensors, like multiple LiDARs or Radars, that are very expensive. So, we want to demonstrate that, even without using highly demanding computation and expensive sensors, the accuracy of rules and policies can be similar. In particular, we use only 3 types of sensors for the vehicles in the platoon: a single LiDAR sensor, spawned in front of the car to manage braking or dangerous situations, Inertial Measurement Unit (IMU) sensors (Accelerometer, Compass, and Gyroscope) that are basic and low-cost sensors, implemented also in mobile devices, and finally, a GPS, that in our simulation is interpreted by the x,y,z coordinates, but in real life is needed to define the vehicle position.

## 6.2 Threat Model

The purpose of our work is to define an effective solution against some attacks to which the platoon could be subject. Here, we assume a third party, external attacker. This means that the attacker is not included in the platoon model, but it is an external entity. The attack surface is represented by the data transmission, described in Section 3.2. We assume that the attacker is able to intercept the data transmitted from the leader vehicle to the followers, analyze it, and modify some parameters, in order to take the platoon control. This attack is inspired by the Man in the Middle (MitM) attack, where a third entity enters the communication, deemed secure, of two distinct entities. In this way, the attacker is able to modify and send to the victim the forged data. In our scenario, this attack model can be implemented by a malicious vehicle, close enough to the platoon to intercept its communications, or a drone flying over it, as in Figure 6.1. Once decrypted, the information sent by the leader to the followers is available for modification and future sending to victim vehicles. Therefore, we assume that an attacker is aware of the type of communication implemented by the platoon, knows the data sent and their subsequent use and is able to modify the data in such a way as to force part of the platoon to follow the forged commands.

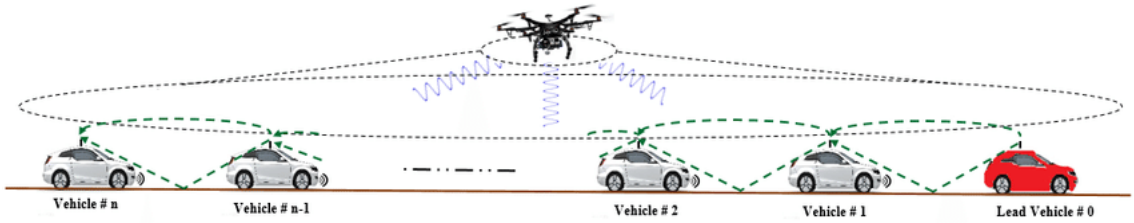



Figure 6.1: Threat model example [53]

## 6.3 Data background

To define rules and policies, we used some variables, which values can be used both for detecting a possible anomaly, both modified in order to solve the problem. These variables are essential for understanding the whole working mechanism, both of CARLA simulator, both of the platooning scripts, and the scenarios that will be presented in the next sections.

A description of the variables used in this work is presented in Table 6.1. Some of these values are obtained inherently from the interface between vehicle and simulator (like for example

throttle and brake), others are external values, obtained by the instantiation of different sensors applied to the vehicles (like gyroscope[X] and accelerometer[X]). In Table 6.1 is not reported anything about the LiDAR sensor. This is because the data obtained by the LiDAR can not be considered as "direct" variables, ready to be used for data computation. Instead, LiDAR output is called *carla.LidarMeasurement per step*, as it returns a cloud of points, computed by adding a laser for each channel of the sensor. This point cloud is calculated by doing a ray-cast for each laser in every step, and it can be written as *points_per_channel_each_step = points_per_second / (FPS * channels).*

So, we could say that the data obtained by LiDAR measurement is a series of encoded 4D points, where the first three coordinates are the canonical xyz and the fourth dimension is the intensity loss during the travel. So, to obtain computable data, we have to analyze the cloud of points, for example by extracting the number of points for each of the xyz coordinates and this requires a supplementary data manipulation stage.

Table 6.1: Measurements

| **Key** | **Type** | **Description** |
|---|---|---|
| steer | float | Steering angle between [-1.0,1.0] |
| throttle | float | Throttle input between [0.0,1.0] |
| brake | float | Brake input between [0.0,1.0] |
| hand_brake | bool | Whether the hand-brake is engaged |
| reverse | bool | Whether the vehicle is in reverse gear |
| pitch | float | Y-Axis rotation angle |
| yaw | float | Z-Axis rotation angle |
| roll | float | X-Axis rotation angle |
| speed | float | Actual vehicle speed |
| speedGoal | float | Goal speed for follower vehicle |
| gyroscope[X] | float | Measures angular velocity in $rad/sec$ per Axis between [-99.9,99.9] |
| accelerometer[X] | float | Measures linear acceleration in $m/s^2$ per Axis between [-99.9,99.9] |
| compass | float | Orientation in radians |

In Figure 6.3, it is reported a brief scheme of connection between the various elements that compose the Platooning script, reported entirely in Appendix A. In the diagram are highlighted the most important parameters and functions of the instantiated objects. The SafeCloud implementation is used for managing the connection and the data exchange between the platoon vehicles and it is again reported entirely in Appendix A.

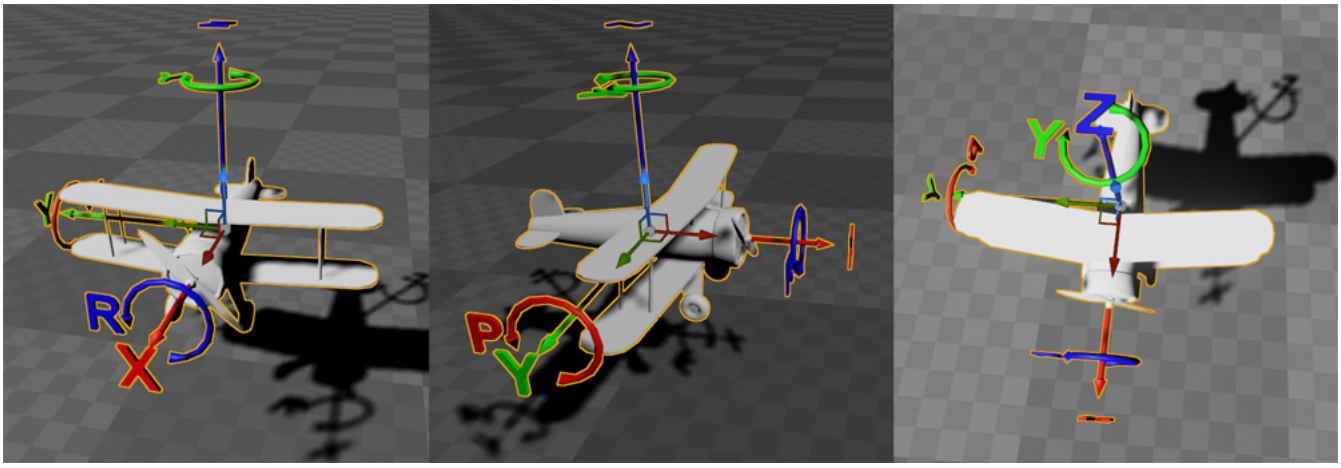



Figure 6.2: Graphical view of Pitch, Roll and Yaw (CARLA documentation)

## 6.4 Scenarios analyzed

As presented in Section 5.2.2, CARLA simulator offers different maps, called Towns, with different scenarios, different route types, and traffic situations. The aim was to find the most common scenarios that a real driver faces up in the everyday driving, making them more general as possible, in order to work in almost every case. All these different situations were found in five different maps, both between the one included in the standard package generated by CARLA developers, both by importing some additional Towns, more complex and heavyweight than the classic ones. In the real world, the scenario is recognized by using the car location systems (GPS/GNSS). In fact, in the simulator, we use the three-dimensional coordinates in order to find the correct scenario and spawn in that position the platoon, while in real world this is performed in real-time by analyzing the location data retrieved from sensors. So, for example, if the navigation system communicates that the vehicle position is close to a roundabout, the whole system will be informed of that information and the "roundabout scenario" is triggered, with all its defined rules and policies.

The scenarios that we defined are exhaustive for the usual everyday driver, and in order are:

- Highway
- Turn
- Roundabout
- Crossroad

- Hairpin

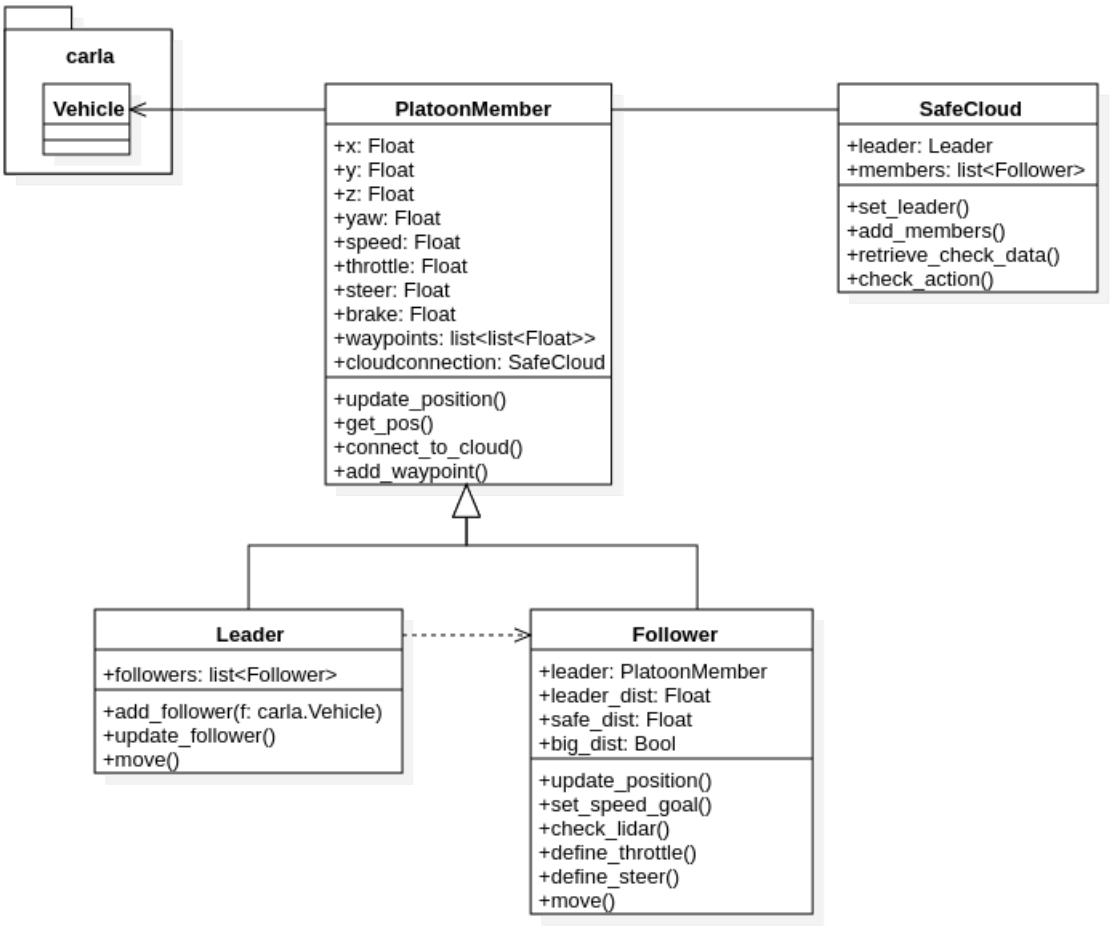



Figure 6.3: Diagram of variables and functions

### 6.4.1 Highway

For the highway scenario, we used Town04 as referring map. In particular, Town04 includes a loop three lanes highway, from which we extrapolate the long straight used for this scenario. The platoon is spawned at the start of this long straight, with the vehicles spaced twelve meters apart. This one is the least complex scenario, as the following vehicles in the platoon have the only task of following the leader in its lane, without possible intersections or traffic signals that could make the run complex to analyze. At the same time, it could be easy to recognize possible attacks and how to protect the platoon from them. In this scenario, the platoon leader is equipped with LiDAR and IMU sensors, similar to the two followers. In Figure 6.4 is presented the map layout, in particular is boxed in red the part of the highway used for testing.

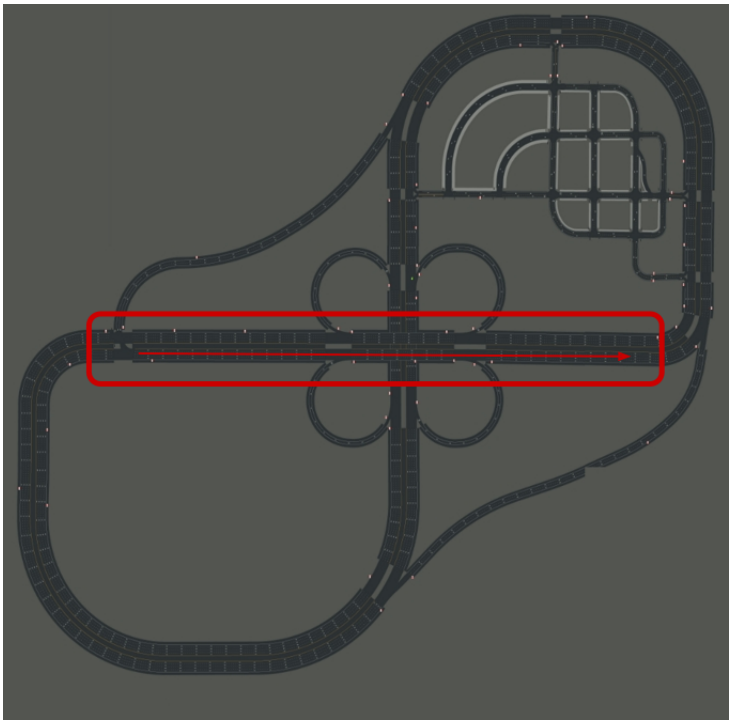


Figure 6.4: Town04 structure, with highlighted the Highway

### 6.4.2 Turn

For the turn scenario, we used Town03 as referring map. Town03 is the most complex map, it is a complicated city-inspired map, from which we extrapolated a right turn, of around 90 degrees, coming right after a simple two-way crossroad. Here, the platoon is spawned at the end of a short straight, right after a crossroad, such that the platoon is capable of picking up the adequate speed and performing correctly the turn. We decided to take this particular turn just for the reason of making the script more general as possible, by using a very generic turn. In fact, this scenario could be used for most of the normal turns (both right and left ones), while for specific turns (like hairpins) we better defined a specific scenario, more detailed and complex. Here we have to consider more variables than before. For example, while in the highway scenario, it was an easy constant motion, with a fixed speed and with a very low probability of encountering obstacles, now we should consider the city environment, like the possibility of sudden accelerations and decelerations. In this scenario, the platoon leader is equipped with LiDAR and IMU sensors, similar to the two followers. In Figure 6.5 is presented the map layout, in particular is boxed in red the part of the road used for testing.

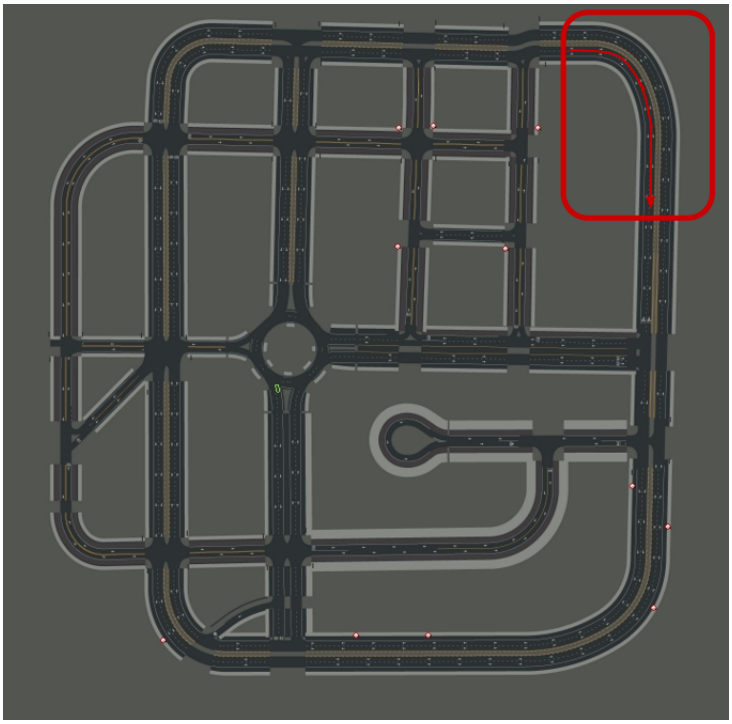


Figure 6.5: Town03 structure, with highlighted the Turn

### 6.4.3 Roundabout

For the Roundabout scenario, we use again Town03 as referring map. The features and a brief description of the map are reported in Section 6.4.2. In this scenario, we use a classic four-exit roundabout, placed right after a long straight. Here, the platoon is spawned at the end of this straight, such that the platoon is capable of picking up the adequate speed and entering realistically in the roundabout. This is a very complex scenario, as the variable are multiple and the number of parameters to take into account is huge. For example, we could not rely only on yaw to determine an abnormal behavior, as this parameter is very variable during the roundabout execution. So, we should take into account other aspects of the driving data, and cross-check them with the usual ones, in order to write policies that are correct and respect the normal behavior of the scenario. Other than that, the attack recognition is much more difficult than before. By using a lot of parameters, we extend a lot the attack surface, such that we should analyze more carefully each signal and error. Some examples of attacks and variables affected are reported in Section 7.4. In this scenario, the platoon leader is equipped with LiDAR and IMU sensors, similar to the two followers. In Figure 6.6 is presented the map layout, in particular is boxed in red the part of the road used for testing.

Figure 6.6: Town03 structure, with highlighted the Roundabout

### 6.4.4 Crossroad

For the Crossroad scenario, we use again Town04 as referring map. The features and a brief description of the map is reported in Section 6.4.1. Town04 is depicted as "An infinite loop with a highway and a small town", and it is that small town our point of interest. We localized a series of crossroads, one adjacent to the other, such that it generates the perfect scenario for testing our purposes. The platoon is spawned right before the stop line of the first crossroad, which is the most important, as it has 3 different directions: straight, turn left, and turn right. Then, following the decision of the platoon leader, the vehicles can run across a brief straight, end by another crossroad, or enter a totally different route. We focused on the behavior of our platoon model in the first crossroad, but the rules and policies are valid also for the other adjacent ones, as noted during the test phase. We will report the results obtained with just one crossroad passed through for a matter of redundancy and huge data manipulation for results extraction. This one was by far the most complex scenario of the series, both for the implementation of the platoon, both for the study of policies and attack surfaces. As said in Section 6.4.3, there are a lot of parameters interested in the study of the vehicles' trajectory, behavior and attacks, such that the task here is more complex than ever. We should depict rules that work for general conditions, taking into account not to influence the normal unfolding of

the operations. In this scenario, the platoon leader is equipped with LiDAR and IMU sensors, similar to the two followers. In Figure 6.7 is presented the map layout, in particular is boxed in red the crossroad used for testing.

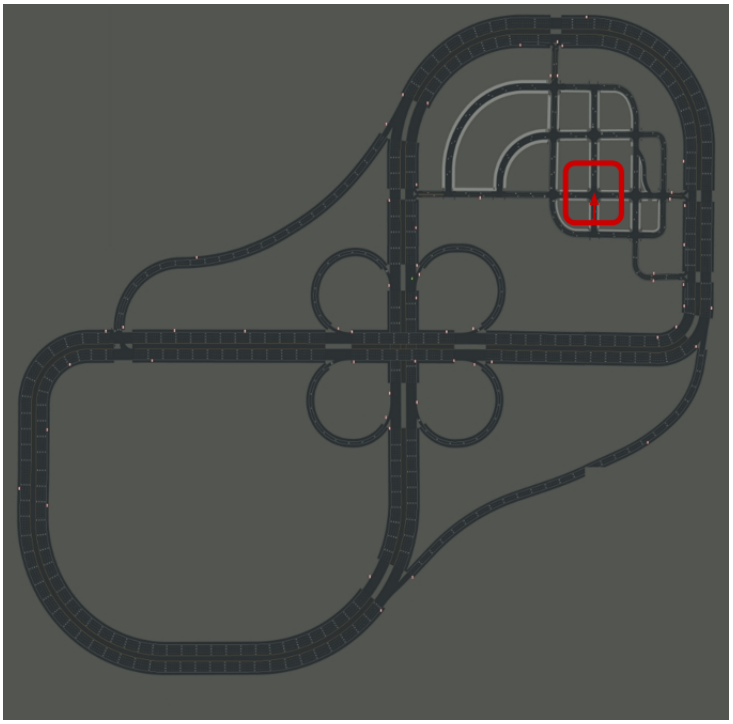


Figure 6.7: Town04 structure, with highlighted the Crossroad

### 6.4.5 Hairpin

For the Hairpin scenario, we use Town06 as referring map. Town06 includes long straights, with small crossroads at the end. In the middle of the map, there is a two-lane road crossing it horizontally, ended by two similar hairpins. These ones are the object of our study. The main difference between a hairpin and a simple turn is the steering angle required, which is more important in the hairpin one, but there are other reasons that make us prefer to have a specific set of rules for it. First of all, we can not rely that much on the yaw value, as made in the turn scenario, as it is a too much variable value in hairpins, similar to what happens in roundabouts. Then, we have different steering angles during the hairpin approach, not a fixed one (or semi-fixed) as in a normal turn. Finally, we have a very different speed approach in hairpin, that needs to be addressed much slower than a normal turn. The platoon is spawned at the end of the long straight that crossed the maps, some meters behind the hairpin, in order to make the platoon capable of picking up the adequate speed, decelerating, and performing

correctly the turn. In this scenario, the platoon leader is equipped with LiDAR and IMU sensors, similar to the two followers. In Figure 6.8 is presented the map layout, in particular is boxed in red the crossroad used for testing.

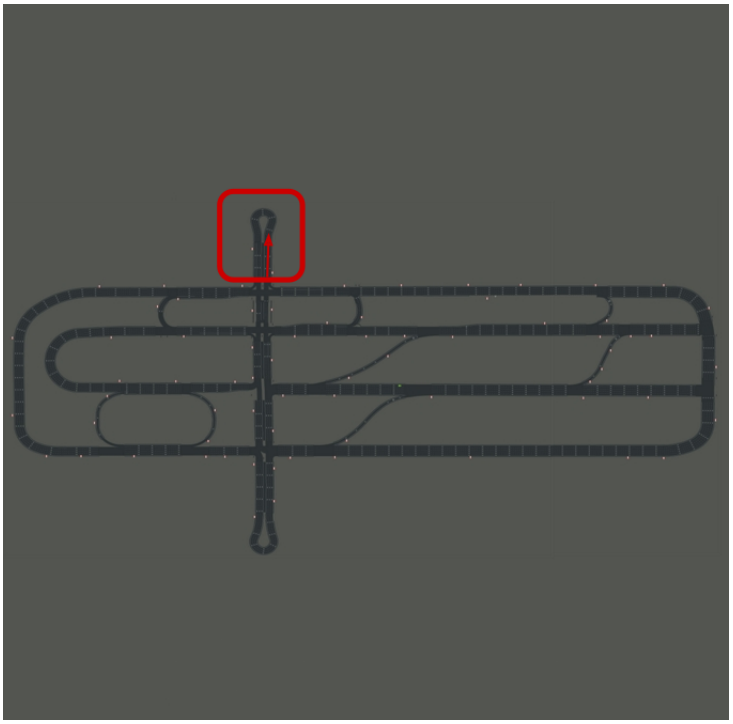


Figure 6.8: Town06 structure, with highlighted the Hairpin

# Chapter 7

# Policies

## 7.1 Introduction

In this chapter, we will introduce and describe the policies and rules defined for each one of the scenarios depicted in Section 6.4. These rules are hard-coded in the platooning script, which can be viewed in Appendix A.

Each scenario has its own rules, such that each version of the platooning script is modified and adapted to the request. The focus of these policies is to assure resilience after an attack detection, so the implemented functions do not only recognize a possible incoming menace, but it is also able to analyze it and, by using the trusted real-time data (described in Section 3.3), to assure that the platoon is able to complete its actions correctly and safely.

All these objectives are reached by using a low-cost and efficient model, by using the minimum number of sensors possible, the least possible computational power (avoiding the use of Machine Learning or Deep Learning techniques as said in the previous chapter intro, Section 6.1). The results will be presented later in this thesis, in Chapter 8.

We perform a check using LiDAR, common for all the scenarios, in order to detect any possible blinding attack. This is a sort of "add-on" to the pre-defined policies, in order to not exclude in advance the use of LiDAR (considering it as not trusted), introducing a step zero when the signal received is abnormal (for example, a sudden stop in the middle of a straight road) where the sensor is controlled in its functionality. In the above code, it can be seen how this check works.

Full code can be seen in Appendix A.

```
1  def check_lidar(self,points):
2          detection=False
3          blinded=False
4          danger_dist = self.speed/5
5          for p in points:
6              if p.point.x<max(3,danger_dist):
7                  if p.point.x < 0.5:
8                      blinded = True
9                      detection = False
10                     print("Points = ", p.point.x)
11                 else:
12                     detection=True
13                     print("LiDAR triggered")
14         self.override_brake=detection
15         self.blinded=blinded
```

We defined two Boolean variables, `detection` and `blinded`. `detection` is false when the LiDAR is not triggered, so when it does not detect any close obstacle and the condition in line 6 is not satisfied, therefore the number of points in the cloud along the x-axis is bigger than the maximum value between 3 and the danger distance, defined as `danger_dist`. Then, when this condition is satisfied, we perform a further check on the number of points in the cloud along the x-axis. If this number is very small (less than 0.5), this means that the LiDAR retrieves an enormous obstacle in front of it, like a solid wall. The possibility that the LiDAR detects a wall just when the vehicle is really close to it is infinitesimal, so, with high probability, the sensor is targeted by a blinding attack, described in Section 2.2.2. So, we set the `blinded` variable value as true, the `detection` variable value as false and we consider the LiDAR under attack during the policies definition. Otherwise, the LiDAR check is performed normally.

## 7.2 Highway

In this section, we will present the defined policies for resilience, applied to the highway scenario, presented in Section 6.4.1. We defined three different policies, focused on the speed bounds (so, to respect the highway limits) and on sudden swerves, by checking if a snap turn signal can be coherent with the collected data. In Table 7.1, these policies are formally defined, followed by a brief explanation.

Table 7.1: Highway Policies

| ID | Explanation | Formalization |
|---|---|---|
| *speed_bound* | Do not exceed the maximum speed limit in highway (around 90MPH) | $speedGoal >= upper\_limit$ |
| *abnormal_slowdown* | Do not stop in the highway (or go too slow) if LiDAR does not recognize obstacles, or if it detects attacks | $speedGoal <= lower\_limit$ |
| *sudden_swerve* | Avoid sudden or excessive swerves while in highway | $\|yaw(lead) - yaw(foll)\| > \tau$ |

### 7.2.1 speed_bound

This rule was defined to avoid speeding attacks. In fact, an attacker can intercept the data transmitted from the platoon leader to the follower and modify the passed leader speed value. This value is used to compute the speed goal of the followers. So, if an attacker modifies the speed value with a higher one, for example, 150MPH, the speed goal computation will be abnormal and it will make the follower go too fast. This can cause possible road accidents or, less dangerously, it could force the car to take a speeding ticket.

```
#--Rule 1-Speed bound (Leader speed is faked)--
        if self.speed > 15:
            if self.speedGoal >= upper_limit:
                    print("Abnormal behaviour")
                    self.speedGoal = self.speed
                    sg = ss
```

In this policy, we use two variables: `speed` and `speedGoal`. If the value of `speed`, which is the value of the actual speed of the follower vehicle, is greater than 15MPH, we start monitoring the passed speed, which is contained in `speedGoal`. If this value is greater than the upper limit (`upper_limit`), then we will alert an abnormal behavior to the driver, and the vehicle will adjust the speed itself, by referring to the actual speed instead of the passed one. So, now the vehicle will increase or decrease its own speed based on its own collected data, on the trusted data passed and it will base the calculation of the true speed value registered by itself.

### 7.2.2 abnormal_slowdown

This rule detects a behavior opposite to the speed_bound rule. In this case, we want to avoid sudden stops on highway, or too low speeds. As said before, an attacker can intercept the data transmitted from the platoon leader to the follower and modify the passed leader speed value. So, if an attacker modifies the speed value with a really lower one, for example, 5MPH, the speed goal computation will be abnormal and it will make the follower go too slow, or even stop in the middle of the lane. This can cause possible severe road accidents, as the following cars may not have the time to react to a sudden breaking or stop, or it can again force the car to take a ticket. In this case, we should consider the possibility that the data passed is not faked, instead, the leader really detects a possible obstacle. To distinguish these two possibilities, we use the LiDAR sensor. In fact, by referring to the LiDAR check presented in Section 7.1, we are able to define if the LiDAR is under attack. So, we check if the LiDAR has launched an `override_brake` signal, which means that the LiDAR is working fine (it passes the check on points discussed before) and there is really an obstacle, such that the brakes have to be activated. Otherwise, if the `override_brake` alert is not triggered, we could both assume that the LiDAR does not detect any obstacle (so the data passed from the leader are faked) or that the LiDAR is blinded (and it will always alert for not existing obstacles).

```
#--Rule 2-Abnormal slowdown (Leader speed is faked)--
       if self.speed > 15:
           if self.speedGoal < lower_limit:
               if not self.override_brake:
                   print("Fake signal, the LiDAR is blinded"
                         "or the data is corrupted, override")
                   self.speedGoal = self.speed
                   sg = ss
```

Again, in this policy, we use two variables: `speed` and `speedGoal`. If the value of `speed`, which is the value of the actual speed of the follower vehicle, is greater than 15MPH (that means that the vehicle is slightly moving), we start monitoring the passed speed, which is contained in `speedGoal`. If the registered value is smaller than the lower limit (`lower_limit`) and the LiDAR check is negative, then we will alert an abnormal behavior to the driver and the vehicle will adjust the speed itself, by referring to the actual speed instead of the passed one. So, now the vehicle will increase or decrease its own speed based on its own collected data, on the

trusted data passed and it will base the calculation of the true speed value registered by itself.

### 7.2.3 sudden_swerve

This rule was defined to detect abnormal swerves, which should not be done on highways. An example could be a 90 degrees turn detected while the GPS signal indicates a straight route. In this case, an attacker can intercept and modify the yaw value (in the script called `lyaw`, leader yaw), in order to fake a turn. For example, the platoon is going in a straight stretch of highway, with almost constant yaw = 90±5. Then, an attacker modifies the `lyaw` value, setting it at 40. Now, the follower vehicle will use that value to calculate both its own `yaw` value and consequently the steering angle, forcing it in taking a hard turn out of road. Also in this case, we should consider the possibility that the data passed is not faked, instead, the leader really detects a possible obstacle and took a swerve to avoid it. To improve the detection mechanism and reduce the possibility of false positives, we add another condition relative to the gyroscope detections of the leader. In this way, we did not influence the results of the check conducted by the rule, but we add a condition that makes it more reliable. Obviously, also the gyroscope data can be faked by a high-level attacker, but, by being yaw and gyroscope-related values, we consider really hard to modify them in a coherent way. In addition, gyroscope values are very accurate and variables, so they are really hard to be faked in a good way.

```
1 #--Rule 3-Avoid sudden swerve (Leader coordinates are faked)--
2             if  (yaw in interval) and (lyaw<upper_limit or lyaw>lower_limit)
3                 and (gyro < 5 and gyro > -5):
4                 print("Fake swerve")
5                 lyaw=yaw
```

In this policy, we use three variables: `lyaw` (the leader yaw), `yaw` (the follower yaw), and `gyro` (the leader z-axis gyroscope value). We can consider four different cases, representing the four different directions, in order to demonstrate the possibility of adapting the rule to the scenario. We use the yaw values referring to the axis (0 and 180/-180 are the y-axis directions, while 90 and -90 are the x-axis directions). Obviously, these values have to be retrieved at driving time. If the `lyaw` and the `yaw` differ for a value greater than $\tau$, then the rule detects an abnormal behavior. If this condition is met, and the value of the gyroscope is included between -5 and 5 (so, the vehicle is not taking a turn, just adjusting), the rule will alert the driver and the

vehicle itself will adjust the direction, permitting the platoon to advance safely. Therefore, now the vehicle will modify its yaw and steering angle based on its own collected data and on the trusted data passed, not considering the faked `lyaw` value.

## 7.3 Turn

In this section, we will present the defined policies for resilience, applied to the Turn scenario, presented in Section 6.4.2. We defined six different policies, focused on the speed bounds (i.e., to respect the highway limits) and on steering input (absence or abnormal behavior). In Table 7.2, these policies are formally defined, followed by a brief explanation.

### 7.3.1 speed_bound

This rule was defined to avoid speeding attacks. As said for highway speeding attacks, an attacker can intercept the data transmitted from the platoon leader to the follower and modify the passed leader speed value. This value is used to compute the speed goal of the followers. So, if an attacker modifies the speed value with a higher one, the speed goal computation will be abnormal and it will make the follower go too fast. This can cause possible road accidents (i.e., the car may be forced off road) or, less dangerously, it could force the car to take a speeding ticket.

```
1 #--Rule 1-Speed bound (Leader speed is faked)--
2         if self.speed > 0.03:
3             if self.speedGoal >= upper_limit:
4                 print("Abnormal behaviour: Faked speed")
5                 self.speedGoal = self.speed
6                 sg = ss
```

In this policy, we use two variables: `speed` and `speedGoal`. If the value of `speed`, which is the value of the actual speed of the follower vehicle, is greater than 0.03MPH (the vehicle is slightly moving), we start monitoring the passed speed, which is contained in `speedGoal`. If this value is greater than the upper limit (`upper_limit`), then we will alert an abnormal behavior to the driver and the vehicle will adjust the speed itself, by referring to the actual speed instead of the passed one. So, now the vehicle will increase or decrease its own speed

Table 7.2: Turn Policies

| ID | Explanation | Formalization |
|---|---|---|
| *speed_bound* | Do not exceed the maximum speed limit | $speedGoal >= upper\_limit$ |
| *abnormal_slowdown* | Do not stop suddenly (or go too slow) if LiDAR does not recognize obstacles, or if it detects attacks | $speedGoal <= lower\_limit$ |
| *no_steer_input* | Leader takes the turn, while follower does not receive any input steer signal (fixed yaw) | $\|yaw(lead) - yaw(foll)\| > \tau$ |
| *double_turn* | Fake turn after the real one | $\|yaw(lead) - yaw(foll)\| > \beta \wedge \|gyroscope.z(foll)\| < 1$ |
| *early_turn* | Fake turn before the real one | $\|gyroscope.z(lead)\| < 1 \wedge \|gyroscope.z(foll)\| < 1 \wedge yaw(lead) \notin$ interval_1 |
| *fake_turn* | The signal received makes the car turn in a different direction compared to the real one | $yaw(foll) \in$ interval_2 $\wedge yaw(lead) \notin$ interval_2 $\wedge \|gyroscope.z(lead)\| > 1 \wedge \|gyroscope.z(foll)\| > 1$ |

based on its own collected data, on the trusted data passed and it will base the calculation of the true speed value registered by itself.

### 7.3.2 abnormal_slowdown

This rule detects a behavior opposite to the *speed_bound* rule. In this case, we want to avoid sudden and dangerous unmotivated stops, or too-low speeds (below the 1MPH). As said before, an attacker can intercept the data transmitted from the platoon leader to the follower and modify the passed leader speed value. So, if an attacker modifies the speed value with a really lower one, the speed goal computation will be abnormal and it will make the follower go too slow, or even stop in the middle of the lane. This can cause possible severe road accidents, as the following cars may not have the time to react to a sudden breaking or stop. Again, also in this case, we should consider the possibility that the data passed is not faked, instead the leader really detects a possible obstacle. To distinguish these two possibilities, we use the LiDAR sensor. In fact, by referring to the LiDAR check presented in Section 7.1, we are able to define if the LiDAR is under attack. So, we check if the LiDAR has launched an `override_brake` signal, which means that the LiDAR is working fine (it passes the check on points discussed

before) and there is really an obstacle, such that the brakes have to be activated. Otherwise, if the `override_brake` alert is not triggered, we could both assume that the LiDAR does not detect any obstacle (so the data passed from the leader are faked) or that the LiDAR is blinded (and it will always alert for not existing obstacles).

```
1  #--Rule 2-Abnormal slowdown (Leader speed is faked)--
2          ...
3          elif self.speedGoal < lower_limit:
4                  if not self.override_brake:
5                      print("Fake signal, the LiDAR is blinded or the data"
6                            "is corrupted, override")
7                      self.speedGoal = self.speed
8                      sg = ss
```

This rule is correlated to the *speed_bound* one, as they share the first check on the follower vehicle speed. So, the `else if` in Line 3 of the script is connected to the one that checks the `speedGoal` in *speed_bound*. If the registered `speedGoal` value is smaller than the lower limit (`lower_limit`) and the LiDAR check is negative, then we will alert an abnormal behavior to the driver and the vehicle will adjust the speed itself, by referring to the actual speed instead of the passed one. So, now the vehicle will increase or decrease its own speed based on its own collected data, on the trusted data passed and it will base the calculation of the true speed value registered by itself.

### 7.3.3 no_steer_input

This rule detects an absence of steering input. This means that, if the system detects that the leader vehicle is turning but the passed data are referring to a straight trajectory indication, the rule will alert and find a solution. In our scenario, the turn is right after a short straight, so we analyze the case in which the data passed to the followers indicates to continue straight, instead of turning. This could be done by an attacker that intercepts the transmission between the platoon members, and modify the yaw value passed by the leader with a previous one, when the vehicle was going straight.

```
1  #--Rule 3: No input steer resilience--
2          if self.speed > 10:
3              if yaw > 0 and yaw < 180 and (yaw < 89 or yaw > 91):
```

```
4                 if lyaw < upper_yaw and lyaw > lower_yaw and (gyro > 1 or gyro < -1):
5                     print ("Error 1 - No input steer")
6                     lyaw = yaw
7             elif yaw < 0 and yaw > -180 and (yaw > -89 or yaw < -91):
8                 if lyaw > lower_yaw and lyaw < upper_yaw and (gyro > 1 or gyro < -1):
9                     print ("Error 2 - No input steer")
10                    lyaw = yaw
11            elif yaw < 90 and yaw > -90 and (yaw > 1 or yaw < -1):
12                if lyaw > lower_yaw and lyaw < upper_yaw and (gyro > 1 or gyro < -1):
13                    print ("Error 3 - No input steer")
14                    lyaw = yaw
15            elif yaw < -90 and yaw > 90 and (yaw > -179 or yaw < 179):
16                if lyaw > 180 - 0.5τ  and lyaw < -180 + 0.5τ and (gyro > 1 or gyro < -1):
17                    print ("Error 4 - No input steer")
18                    lyaw = yaw
```

In this policy, we use four variables: `speed` (the follower actual speed), `lyaw` (the leader yaw), `yaw` (the follower yaw) and `gyro` (the leader z-axis gyroscope value). We consider four different cases, representing the four different directions, in order to demonstrate the possibility of adapting the rule to the scenario. We use yaw value referring to the axis (0 and 180/-180 are the y-axis directions, while 90 and -90 are the x-axis directions). Obviously, these values have to be retrieved at driving time. If the `lyaw` and the `yaw` differ for a value greater than $\tau$, then the rule detects an abnormal behavior. If this condition is met, and the value of the leader gyroscope is greater than -1 or 1 (so, the vehicle is taking a left or right turn), the rule will alert the driver and the vehicle itself will adjust the direction, permitting the platoon to advance safely. All these checks are performed after a basic control of the vehicle speed, in order to avoid false positives due to possible corrections at low speeds So, now the vehicle will modify its yaw and steering angle based on its own collected data and on the trusted data passed, not considering the faked `lyaw` value.

### 7.3.4 double_turn

This rule is used to detect a fake turn, after the platoon has ended the real turn. In fact, in the analyzed simulation scenario, the turn is ended by a long straight. An attacker can intercept the yaw parameter, sent by the leader to the follower in order to highlight the correct direction, modify it and force the followers in take a fake turn. At the same time, the attacker could exploit the fact that the platoon has just ended a real turn, such that it maintains constant

the yaw passed. In this way, the platoon followers elaborate that the platoon is still turning, from this the name "double turn".

```
#--Rule 4-Fake turn signal after the real turn--
    if self.speed > 10:
        if yaw > 179 and yaw < -179:
            if lyaw < upper_yaw2 or lyaw > lower_yaw2 and (gyro < 1 and gyro > -1):
                if self.gyroscope[-1] > 1 or self.gyroscope[-1] < -1:
                    print("Error: the turn has ended")
                    lyaw = yaw #or we can pass 180 (the straight yaw value)
        elif yaw > 89 and yaw < 91:
            if lyaw < upper_yaw2 or lyaw > lower_yaw2 and (gyro < 1 and gyro > -1):
                if self.gyroscope[-1] > 1 or self.gyroscope[-1] < -1:
                    print("Error: the turn has ended")
                    lyaw = yaw #or we can pass 90 (the straight yaw value)
        elif yaw < -89 and yaw > -91:
            if lyaw > -upper_yaw2 or lyaw < -lower_yaw2 and (gyro < 1 and gyro > -1):
                if self.gyroscope[-1] > 1 or self.gyroscope[-1] < -1:
                    print("Error: the turn has ended")
                    lyaw = yaw #or we can pass -90 (the straight yaw value)
        elif yaw > -1 and yaw < 1:
            if lyaw < -upper_yaw2 or lyaw > lower_yaw2 and (gyro < 1 and gyro > -1):
                if self.gyroscope[-1] > 1 or self.gyroscope[-1] < -1:
                    print("Error: the turn has ended")
                    lyaw = yaw #or we can pass 0 (the straight yaw value)
```

In this policy, we use five variables: `speed` (the actual speed of the follower), `lyaw` (the leader yaw), `yaw` (the follower yaw), `gyro` (the leader z-axis gyroscope value) and `gyroscope` (the follower z-axis gyroscope value). We consider four different cases, representing the four different directions, in order to demonstrate the possibility of adapting the rule to the scenario. We use the yaw values referring to the axis (0 and 180/-180 are the y-axis directions, while 90 and -90 are the x-axis directions). Obviously, these values have to be retrieved at driving time. If the `lyaw` and the `yaw` differ for a value greater than $\tau$, then the rule detects an abnormal behavior. If this condition is met, and the value of the gyroscope is included between -1 and 1 (so, the vehicle is not taking a turn, just adjusting), the rule will alert the driver and the vehicle itself will adjust the direction, permitting the platoon to advance safely. All these checks are performed after a basic control of the vehicle speed, in order to avoid false positives due to possible corrections at low speeds. So, now the vehicle will modify its yaw and steering angle based on its own collected data (including the fundamentals sensors detection) and on

the trusted data passed, not considering the faked `lyaw` value.

### 7.3.5 early_turn

This rule is used to detect a fake turn, before the platoon starts the real turn. In fact, in the analyzed simulation scenario, before the real turn there is a short straight. An attacker can intercept the yaw parameter, sent by the leader to the follower in order to highlight the correct direction, modify it and force the followers to take a fake turn. In this way, the platoon is forced to take a non-existent turn. This can cause severe accidents or, even in the case in which the attack is not targeted to a crash, it could mess up the trajectory, such that the platoon follower can not be able to re-adjust its own trajectory and perform the real turn correctly.

```
1 #--Rule 5: Fake turn before the real one--
2 if self.speed > 10:
3     if (gyro < 1 and gyro > -1) and (self.gyroscope[-1] < 1
4                                   and self.gyroscope[-1] > -1):
5         if not (lyaw in interval1):
6             print("Fake turn data")
7             lyaw = yaw
```

In this policy, we use five variables: `speed` (the actual speed of the follower), `lyaw` (the leader yaw), `yaw` (the follower yaw), `gyro` (the leader z-axis gyroscope value) and `gyroscope` (the follower z-axis gyroscope value). Here, we don't consider four different cases as before, but we define an *interval*, defined in the second nested if condition. So, we first perform a check on the follower speed, again to avoid false positives due to possible corrections at low speeds. Then, we check the gyroscope values, both of the leader and of the follower vehicle. If they are both in the interval [-1,1], we enter in the second nested if condition. Here, we check the coherency of data. If the passed leader yaw indicates a possible turn (so, if it is not in one of the intervals indicated), this means that the data are not coherent and the data transmission is under attack. So, now the vehicle will alert the driver and independently modify its yaw and steering angle based on its own collected data (including the fundamentals sensors detection) and on the trusted data passed, not considering the faked `lyaw` value.

### 7.3.6 fake_turn

This rule was defined in order to detect if the signal received during a turn is coherent with the actual vehicle trajectory. So, for example, if the car is taking a right turn and suddenly the data indicates a left one, the system will alert and check with the context data. Otherwise, the following vehicles will take an opposite direction and this can cause severe accidents or, even in the case in which the attack is not targeted to a crash, it could mess up the trajectory, such that the platoon follower can not be able to re-adjust its own trajectory and perform the real turn correctly.

```
#--Rule 6: Fake turn direction--
if self.speed > 10:
    if yaw in interval2 and self.gyroscope[-1] > 1:
        if lyaw not in interval2 and gyro > 1:
        #this means that the signal are faked in order to make the car turn
        #in the opposite position.
            print("Single turn, error in the transmitted signal")
            lyaw = yaw
```

In this policy, we use five variables: `speed` (the actual speed of the follower), `lyaw` (the leader yaw), `yaw` (the follower yaw), `gyro` (the leader z-axis gyroscope value) and `gyroscope` (the follower z-axis gyroscope value). Here we can define four different cases, referring to four different intervals. We define these intervals by dividing the yaw range into four equal parts. Again, we start performing a check on the follower vehicle speed. Then, we check in which interval the follower yaw is. Defined this, we check coherency with the follower gyroscope data, just to make the rule stronger. Then, it is time to check the passed data. If the received leader yaw is not in the same interval of the follower one, this means that there is something abnormal, in particular the faked signal indicates to take a turn opposite to the real one. So, as usual, the vehicle will alert the driver and independently modify its yaw and steering angle based on its own collected data (including the fundamentals sensors detection) and on the trusted data passed, not considering the faked `lyaw` value.

## 7.4 Roundabout

In this section, we will present the defined policies for resilience, applied to the Roundabout scenario, presented in Section 6.4.2. We defined four different policies, focused on the speed bounds (so, to respect the road limits) and on steering input (absence or abnormal behavior). This one, paired with the crossroad one, was the most difficult scenario to analyze, both for the great number of possible different outcomes, and both for the number of aspects to take care of. In Table 7.3, these policies are formally defined, followed by a brief explanation.

Table 7.3: Roundabout Policies

| ID | Explanation | Formalization |
|---|---|---|
| *speed_bound* | Do not exceed the maximum speed limit | $speedGoal >= upper\_limit$ |
| *abnormal_slowdown* | Do not stop suddenly (or go too slow) if LiDAR does not recognize obstacles, or if it detects attacks | $speedGoal <= lower\_limit$ |
| *no_steer_input* | Leader takes the turn, while follower does not receive any input steer signal (fixed yaw) | $\|yaw(lead) - yaw(foll)\| > \tau$ |
| *steer_angles* | Check if the steering angles are abnormal | $steer(foll) > upper\_steer \wedge steer(foll) < lower\_steer$ |

### 7.4.1 speed_bound

This rule was defined to avoid speeding attacks. In fact, an attacker can intercept the data transmitted from the platoon leader to the follower and modify the passed leader speed value. This value is used to compute the speed goal of the followers. So, if an attacker modifies the speed value with a higher one, the speed goal computation will be abnormal and it will make the follower go too fast. This can cause possible road accidents, caused for example by a too-high entering speed in the roundabout, that can cause a collision with other cars or make the vehicle not able to follow the turn trajectory. This attack can also override brake signals, such that the attacked vehicle does not stop at the roundabout entering to give precedence, causing a possible collision.

```
#--Rule 1-Speed bound (Leader speed is faked)--
        if self.speed > 10:
            if self.speedGoal > upper_limit:
                print("Abnormal behaviour: Faked speed")
                self.speedGoal = self.speed
                sg = ss
```

In this policy, we use two variables: `speed` and `speedGoal`. If the value of `speed`, which is the value of the actual speed of the follower vehicle, is greater than 10MPH, we start monitoring the passed speed, which is contained in `speedGoal`. If this value is greater than the upper limit (`upper_limit`), then we will alert an abnormal behavior to the driver and the vehicle will adjust the speed itself, by referring to the actual speed instead of the passed one. So, now the vehicle will increase or decrease its own speed based on its own collected data, on the trusted data passed and it will base the calculation of the true speed value registered by itself.

### 7.4.2 abnormal_slowdown

With this rule, we want to avoid sudden and dangerous unmotivated stops, or too-low speeds (below the 1MPH). As said in previous sections, an attacker can intercept the data transmitted from the platoon leader to the follower and modify the passed leader speed value. So, if an attacker modifies the speed value with a really lower one, the speed goal computation will be abnormal and it will make the follower go too slow, or even stop in the middle of the roundabout without any motivated reason. This can cause possible severe road accidents, as the following cars may not have the time to react to a sudden breaking or stop. Similar to what was described in 7.3.2, we should consider the possibility that the data passed is not faked, instead the leader really detects a possible obstacle. So, as presented in the cited section, we use the LiDAR sensor and its data check.

```
#--Rule 2-Abnormal slowdown (Leader speed is faked)--
        ...
        elif self.speedGoal < lower_limit:
            if not self.override_brake:
                print("Fake signal, the LiDAR is blinded or the data
                      "is corrupted, override")
                self.speedGoal = self.speed
                sg = ss
```

This rule is correlated to the *speed_bound* one, as they share the first check on the follower vehicle speed. So, the `else if` in Line 3 of the script is connected to the one that checks the speedGoal in *speed_bound*. If the registered speedGoal value is smaller than the lower limit (`lower_limit`) and the LiDAR check is negative, then we will alert an abnormal behavior to the driver and the vehicle will adjust the speed itself, by referring to the actual speed instead of the passed one. So, now the vehicle will increase or decrease its own speed based on its own collected data, on the trusted data passed and it will base the calculation of the true speed value registered by itself.

### 7.4.3 no_steer_input

This rule detects an absence of steering input. This means that, if the system detects that the leader vehicle is turning but the passed data are referring to a straight trajectory indication, the rule will alert and find a solution. In our scenario, the roundabout is right after a long straight, so we analyze the case in which the data passed to the followers indicates to continue straight, instead of turning. In particular, in this roundabout scenario, the vehicle has to perform a slight right turn in order to enter in it, then it will go through the roundabout turning left, until it has to take the selected exit. So, an attacker could modify the passed leader yaw data, in order to force the follower vehicles to go straight and cross the roundabout, with a high risk of crashes.

```
1  #--Rule 3: No input steer resilience--
2          if self.speed > 1:
3              if yaw < 89 or yaw > 91:
4                  if lyaw < upper_yaw and lyaw > lower_yaw:
5                      print ("Error 1 - No input steer")
6                      lyaw = yaw
7              elif yaw > -89 or yaw < -91:
8                  if lyaw > lower_yaw and lyaw < upper_yaw:
9                      print ("Error 2 - No input steer")
10                     lyaw = yaw
11             elif yaw > 1 or yaw < -1:
12                 if lyaw > lower_yaw and lyaw < upper_yaw:
13                     print ("Error 3 - No input steer")
14                     lyaw = yaw
15             elif yaw < 179 or yaw > -179:
16                 if lyaw < upper_yaw and lyaw > lower_yaw:
```

```
17                      print ("Error 4 - No input steer")
18                      lyaw = yaw
```

In this policy, we use four variables: `speed` (the follower actual speed), `lyaw` (the leader yaw), `yaw` (the follower yaw) and `gyro` (the leader z-axis gyroscope value). We consider four different cases, representing the four different directions, in order to demonstrate the possibility of adapting the rule to the scenario. We use yaw value referring to the axis (0 and 180/-180 are the y-axis directions, while 90 and -90 are the x-axis directions). Obviously, these values have to be retrieved at driving time. If the `lyaw` and the `yaw` differ for a value greater than $\tau$, then the rule detects an abnormal behavior. If this condition is met, the rule will alert the driver and the vehicle itself will adjust the direction, permitting the platoon to advance safely. All these checks are performed after a basic control of the vehicle speed, in order to avoid false positives due to possible corrections at very low speeds So, now the vehicle will modify its yaw and steering angle based on its own collected data and on the trusted data passed, not considering the faked `lyaw` value.

### 7.4.4 steer_angles

This rule was designed to detect any abnormal steering angle while performing the roundabout turn. So, it will detect if the vehicle is turning too much (or not enough) due to a faked yaw. In this case, we do not rely only on the yaw value, as this one is very variable during the roundabout turn performing. In fact, if the vehicle takes the fourth exit, it will perform an almost complete roundabout lap, passing almost all the yaw values. So, a similar rule based only on yaw won't be strong and it will for sure influence the normal behavior of the vehicle, by raising too many false alerts. The attacks detected by this rule are multiple. An attacker can modify the yaw value to make the vehicle go too wide in the roundabout, forcing it to go over a possible sidewalk, roadside, or other vehicles. At the same time, an attacker can modify the same value to force the vehicle in taking a too-narrow turn, go into the roundabout center, or bump other cars.

```
1  #--Rule 4: steering angle bounds--
2          if s<lower_steer or self.gyroscope[-1] < - 40:
3              print("Abnormal steering")
4              s=-normal_steer
```

```
5           if s>upper_steer or self.gyroscope[-1] > 60:
6               #Check leader yaw to know if it is exiting/entering the roundabout
7               if lyaw > yaw or lsteer > 0:
8                   #Entering or exiting the roundabout
9                   print("Entering/Exiting the roundabout")
10              elif lyaw < yaw or lsteer < -0.1:
11                  print("Abnormal steering - stay in the roundabout")
12                  s=-normal_steer
13          if self.gyroscope[-1] > 5 and gyro < 0:
14              print("Wrong turn")
15              print("Gyroscope = ", self.gyroscope[-1])
16              print("gyro = ", gyro)
17              lyaw = yaw
```

In this policy, we use four variables: `s` (the follower calculated steering angle), `lyaw` (the leader yaw), `yaw` (the follower yaw), `gyro` (the leader z-axis gyroscope value), `gyroscope` (the follower z-axis gyroscope value) and `lsteer` (the leader steering angle). We defined two bounds (`lower_steer` and `upper_steer`). Then, we check if the calculated steering angle is out of these bounds. In the first if, we check if the steering angle is less than the lower limit, checking also the gyroscope to make the rule stronger. If this is the case, we alert the driver and the vehicle independently set the steering angle to the common one, calculated by analyzing the previously collected data. In the second if, we check if the steering angle is bigger than the upper limit. If this is the case, we should distinguish two possible outcomes: the first one does not imply any attack, but it's the normal behavior of a vehicle exiting or entering the roundabout. The second scenario is the attack one, where the faked data are passed while the platoon is still turning. To distinguish these two study cases, we introduce the `lsteer` variable, defined previously in this section. If the value of `lsteer` is greater than 0 and the leader yaw is greater than the follower one, then the platoon is in the first case, so it is exiting or entering the roundabout. Otherwise, if `lsteer` $<$ -0.1 and the leader yaw is not greater than the follower one, then the data are not coherent and we are probably under attack. So, the vehicle alerts the driver of the abnormal behavior and independently set again the steering angle to the same common value obtained before. Finally, we add another check to make the rule stronger. In this case, we check for any abnormal type of turn, in particular opposite turns during the roundabout crossing. We check the gyroscopes value of the leader and of the follower and if they are not consistent with each other, the vehicle itself will modify its yaw and steering angle based on its own collected data and on the trusted data passed, not considering the faked `lyaw` value.

## 7.5 Crossroad

In this section, we will present the defined policies for resilience, applied to the Crossroad scenario, presented in Section 6.4.4. We defined six different policies, focused on the speed bounds (so, to respect the road limits) and on steering input (absence or abnormal behavior). As said previously, this scenario was one of the most difficult to analyze, both for the great number of possible different outcomes, both for the number of aspects to take care. In Table 7.4, these policies are formally defined, followed by a brief explanation.

Table 7.4: Crossroad Policies

| ID | Explanation | Formalization |
|---|---|---|
| *speed_bound* | Do not exceed the maximum speed limit | $speedGoal >= upper\_limit$ |
| *fake_start* | Leader is not moving, but the follower receives signals to start moving | $\|accelerometer.x(lead)\| < \alpha \wedge \|accelerometer.x(foll)\| > \beta$ |
| *no_brake_input* | Leader is stopping (red light or stop signal), follower does not slow down | $\|accelerometer.x(lead)\| < \alpha \wedge \|accelerometer.x(foll)\| > \alpha \wedge Dist(lead, foll) < 10$ |
| *fake_turn* | Leader goes straight, follower receives signal to turn | $\neg(yaw(lead) > upper\_yaw2 \wedge yaw(lead) < lower\_yaw2) \wedge \|gyroscope.z(lead)\| < \tau \wedge \|gyroscope.z(foll)\| < \tau$ |
| *opposite_turn* | Leader turn into a direction, follower takes the opposite turn | $yaw(lead) \notin interval \wedge yaw(foll) \in interval \wedge \|gyroscope.z(foll)\| > 1$ |
| *no_steer_input* | Leader turn, follower receives signal to go straight (trial rule, recalculating yaw) | $\|yaw(lead) - yaw(foll)\| > \delta \wedge \|gyroscope.z(lead)\| > \gamma$ |

### 7.5.1 speed_bound

This rule was defined to avoid speeding attacks. In fact, an attacker can intercept the data transmitted from the platoon leader to the follower and modify the passed leader speed value. This value is used to compute the speed goal of the followers. So, if an attacker modifies the speed value with a higher one, the speed goal computation will be abnormal and it will make the follower go too fast. This can cause possible road accidents, caused for example by a too-high entering speed in the crossroad, that can cause a collision with other cars or make the

vehicle not able to follow the correct trajectory. This attack can also override brake signals, such that the attacked vehicle does not stop at the crossroad stop line to give precedence, causing a possible collision. The code is very similar to the previous cases, as the recognition mechanism is the same.

```
#--Rule 1-Speed bound (Leader speed is faked)--
        if self.speed > 5:
            if self.speedGoal > upper_limit:
                print("Abnormal behaviour: Faked speed")
                self.speedGoal = self.speed
                sg = ss
```

In this policy, we use two variables: `speed` and `speedGoal`. If the value of `speed`, which is the value of the actual speed of the follower vehicle, is greater than 5MPH, we start monitoring the passed speed, which is contained in `speedGoal`. If this value is greater than the upper limit (`upper_limit`), then we will alert an abnormal behavior to the driver and the vehicle will adjust the speed itself, by referring to the actual speed instead of the passed one. So, now the vehicle will increase or decrease its own speed based on its own collected data, on the trusted data passed and it will base the calculation of the true speed value registered by itself.

### 7.5.2 fake_start

This rule was defined to avoid fake starts while the platooning is stopped. In fact, the vehicles can stop due to a red light, or to a stop signal, then the platoon will restart when the road is free, or when the green light appears. An attacker could intercept the throttle and braking data and modify them, in order to force the follower vehicle that the leader one is moving, while the entire platoon is not starting yet. This can cause crashes between the platoon vehicles, in particular if the attacker also blinds the LiDAR sensor.

```
#--Rule 2-Fake start (Leader is not moving for stop signal or red light)
            if |self.accelerometer[0]| > β:
               if self.speed == 0 and (|aclx| < α and |acly| < α):
                   print("Error, the platoon leader is stopped")
                   self.speedGoal = self.speed
                   sg = ss
```

In this policy, we use four variables: `speed`, `accelerometer`, `aclx` (that is the acceleration value of the leader, calculated on the x-axis) and `acly` (that is the acceleration value of the leader, calculated on the y-axis). First of all, we check if the follower acceleration value (calculated on the x-axis) is greater than $\beta$. If this is the case, we check the actual condition both of the follower and of the leader vehicles. If the speed of the follower is still 0, this means that we are in a stationary situation and probably we are preparing to restart. So, we will expect that the acceleration values of the leader (both on the x and y axis, just to make the rule stronger) are important, both positive or negative depending on the direction, because of the platoon starting. If this is not the case, like the condition in the second nested if represent, there is some sort of anomaly in the system. To solve this, the follower vehicle alerts the driver and independently adjusts the speed (and consequently the acceleration) based on the collected data, avoiding the use of the faked variables arriving from the leader.

### 7.5.3 no_brake_input

This rule detects if vehicle's data are faked, such that it is not able to stop even when the platoon leader is stationary. This could happen again when the leader encounters a red light or a stop signal. This implies that the leading vehicle takes some kind of deceleration until it completely stops near to the stop line. Usually, the following vehicles will take the same deceleration, copying the leader's behavior. If an attacker intercepts, as before, the throttle and brake data and modify them, it could force the following vehicles to not stop and continue their trajectory without any deceleration, or worse by accelerating it. This can cause severe accidents and, even if LiDAR is not overridden and detects the obstacle, it could be again dangerous due to a sudden braking or trajectory change.

```
1  #--Rule 3-No brake indication--
2              if self.speed > 10 and |self.accelerometer[0]| > α
3                                  and (|aclx| < α):
4                  if self.leader_dist < 10 #or 22 for the third platoon member:
5                      print("Error, the platoon leader is stopping")
6                      self.speedGoal = 0
7                      sg = 0
```

In this policy, we use five variables: `speed`, `speedGoal`, `accelerometer`, `aclx` (that is the acceleration value of the leader, calculated on the x-axis), and `leader_dist` (that is the distance,

calculated at real time, between the following vehicle and the leader one). First of all, we consider only the cases in which the following vehicle is moving, so we check the actual speed. If this is greater than 10MPH, we check if the car is still accelerating. If the vehicle is still accelerating, while the leader accelerometer data indicate a deceleration, we find an anomaly. At this point, we check the distance between the vehicles. We defined 10 meters as a good safe distance, so if the following vehicle is under this distance and still accelerating, we are in a dangerous situation. It it important to note that this distance is set at 22 meters for the second platoon follower, because its original spawn point is 12 meters behind the first follower. We could use also use a distance measured between the vehicles themselves, instead of referring to the leader every time. The problem with this solution is that it creates another attack surface, where the faked data are the one passed from follower to follower and the leader is not involved. It is a good building technique to involve always the leader, to make the platoon safer. So, when the rule detects the possible attack, the system will alert the driver and the vehicle will independently adjust the speed, in this case by setting it at 0, in order to apply a great deceleration and avoid crashes or sudden trajectory changes.

### 7.5.4 fake_turn

This rule was designed to detect fake turns while crossing the crossroad. For example, after the restart, the platoon leader goes straight, while the followers take a left or right turn. This could happen if an attacker intercepts the steering data and modify it, forcing the following vehicles in taking wrong directions. This can cause different kinds of problems. First of all, it can cause the vehicles to take a non-existent turn, so they will crash. Otherwise, the platoon will be separated, so the following vehicles lose the leader indications. This is an addressable problem, as the designer should decide if the remaining platoon should safely stop and wait for help, or if there should be a role change, such that the first follower becomes the platoon leader, re-instantiating a new communication system. So, this rule helps avoid this kind of problems.

```
1 #--Rule 4-Fake turn--
2         if self.speed > 0.03:
3             if ((lyaw > 95 and lyaw < 175) or (lyaw < 85 and lyaw > 5))
4             or ((lyaw < -5 and lyaw > -85) or (lyaw > 5 and lyaw < 85))
5             or ((lyaw < -95 and lyaw > -175) or (lyaw > -85 and lyaw < -5))
6             or ((lyaw < 175 and lyaw > 95) or (lyaw > -175 and lyaw < -95)):
```

```
7                 if gyro < τ and gyro > -τ
8                             and self.gyroscope[-1] < τ and self.gyroscope[-1] > -τ:
9                     print("Error: The platoon leader is going straight")
10                    lyaw = yaw
```

In this policy, we use five variables: `speed`, `lyaw`, `yaw`, `gyro` and `gyroscope`. First of all, we check if the vehicle is moving. Then, we consider multiple cases, nested in a single if. This is done to detect if the leader's yaw is in some determined intervals that indicate a turn. So, if the leader yaw indicates a turn, we check the gyroscope values, both of the leader and of the follower one. If these values are in little interval ($[-\tau, \tau]$), this means that the gyroscope does not detect any turn. So, the system detects an inconsistency between the collected and the passed data. The vehicle will alert the driver and independently adjust the steering data, based on its own collected data and on the trusted data passed, not considering the faked `lyaw` value.

### 7.5.5 opposite_turn

This rule was defined in order to detect if the signal received during a turn is coherent with the actual vehicle trajectory. So, for example, if the leader platoon takes a right turn after restarting from the stop line and the data passed to followers indicates a left turn, the system will alert and check with the context data. Otherwise, the following vehicles will take an opposite direction and this can cause severe accidents and even different kinds of problems. First of all, it can cause the vehicles to take a non-existent turn, so they will crash. In case of not crashing, the platoon will be separated and, as described in the previous section, the following vehicles lose the leader indications.

```
1   #--Rule 5-Opposite turn--
2         if self.speed > 5:
3             if yaw in interval:
4                 if lyaw not in interval and self.gyroscope[-1] > 1:
5                 #this means that the signal are faked in order to make the car
6                 #turn in the opposite position.
7                     print("Wrong turn, error in the transmitted signal")
8                     lyaw = yaw
```

In this policy, we use four variables: `speed`, `lyaw`, `yaw`, and `gyroscope`. Here we can define four different cases, referring to four different intervals. We define these intervals by dividing the yaw range into four equal parts. First of all, we perform a check on the follower vehicle speed. Then, we check in which interval the follower yaw is. Defined this, we check coherency with the leader yaw data, to see if it is in the same interval, and we check if the follower gyroscope value is greater than 1 (or lower than -1, depending on the turn direction). If the gyroscope value is verified and the leader yaw is not in the same interval as the follower one, this means that there is something abnormal, in particular the faked signal indicates to take a turn opposite to the real one. So, as usual, the vehicle will alert the driver and independently modify its yaw and steering angle based on its own collected data (including the fundamentals sensors detection) and on the trusted data passed, not considering the faked `lyaw` value.

### 7.5.6 no_steer_input

This rule detects an absence of steering input. This means that, if the system detects that the leader vehicle is turning but the passed data are referring to a straight trajectory indication, the rule will alert and find a solution. In this case, it means that the rule recognizes a scenario in which the leader takes a turn after the restart, but the passed data forces the follower vehicles to go straight. So, an attacker could modify the passed leader yaw data, in order to force the follower vehicles to go straight and cross the crossroad, with the same risks discussed before, so crashes or connection loss with the platoon. This rule was the most complex one to define in this scenario, because it requires reconstructing and re-calculate the yaw value, starting from the trusted collected data.

```
1  #--Rule 6: No input steer resilience--
2          if self.speed > 5:
3              if lyaw > lower_yaw and lyaw < upper_yaw:
4                  if gyro > γ or gyro < -γ:
5                      print ("Error, the platoon leader is turning")
6                      lyaw = 180 * np.arctan(accz/np.sqrt(accx*accx + accz*accz))/np.pi
7                      print("lyaw recalculated= ", lyaw)
```

In this policy, we use five variables: `speed` (the follower actual speed), `lyaw` (the leader yaw), `gyro` (the leader z-axis gyroscope value), `accx` (that is the leader acceleration value calculated on the x-axis) and `accz` (that is the leader acceleration value calculated on the z-axis). We can

consider four different cases, representing the four different directions, in order to demonstrate the possibility of adapting the rule to the scenario. We use yaw value referring to the axis (0 and 180/-180 are the y-axis directions, while 90 and -90 are the x-axis directions). Obviously, these values have to be retrieved at driving time. Then, we check if the platoon leader is turning, by checking the gyro values. If this value is bigger than $\gamma$ (or lower than $-\gamma$, it depends on the turn direction), the leading vehicle is turning, while the follower data indicate to go straight. Even if the basic structure is similar to the previous cases, the reconstruction of the leader yaw used for computing the steering angle is completely different and experimental. In fact, in this case, we can not reconstruct it starting from the follower yaw value, as the follower vehicle can not forecast in any way the leader direction based only on a single data, because the leading vehicle can take three different directions (not one as in the previous cases). So, we had to reconstruct the leader yaw value from the scratch. By looking in CARLA simulator's official documentation, we can retrieve how the yaw value is calculated. Then, by using the formal definition of yaw, described in 6.3, we were able to write the formula

$$180 * np.arctan(accz/np.sqrt(accx * accx + accz * accz))/np.pi$$

to calculate the yaw value, where arctan is the arc-tangent of the z-axis acceleration, divided by the square root of the sum of the z-axis squared and x-axis squared accelerations). Then, we divide this obtained value by $\pi$ and we retrieve the yaw value. So, now the vehicle will modify its yaw and steering angle based on its own collected data, on the trusted data passed, and on the leader yaw reconstruction, not considering the faked `lyaw` value. As said before, this is the more experimental rule. This because the generated yaw value is accurate, but the computation requires some time, such that there is a little delay between the data generation and its passage, due to the short time set for ticks. This is more a simulator-side problem, due to its construction methods, but it is good to take this aspect into account also in a possible real realization.

## 7.6 Hairpin

In this section, we will present the defined policies for resilience, applied to the Hairpin scenario, presented in Section 6.4.5. We defined five different policies, focused on the speed bounds (so, to respect the road limits) and on steering input (absence or abnormal behavior). A hairpin

can be defined as a very narrow turn but, by its conformation and its values management, it can be considered a sort of hybrid between the turn and the roundabout scenarios, with differences in the traffic management and in the dangerous situations. So, we prefer to describe a dedicated script, in order to highlight better the differences. In Table 7.5, these policies are formally defined, followed by a brief explanation.

Table 7.5: Hairpin Policies

| ID | Explanation | Formalization |
|---|---|---|
| *speed_bound* | Do not exceed the maximum speed limit | $speedGoal >= upper\_limit$ |
| *abnormal_slowdown* | Do not stop suddenly (or go too slow) if LiDAR does not recognize obstacles, or if it detects attacks | $speedGoal <= lower\_limit$ |
| *excessive_angles* | The gyroscope alerts a too high steer angle | $(gyroscope.z(lead) > gyro\_\text{max} \vee gyroscope.z(lead) < gyro\_\text{min}) \wedge (gyroscope.z(foll) > gyro\_\text{max} \vee gyroscope.z(foll) < gyro\_\text{min})$ |
| *opposite_turn* | Leader turn into a direction, follower takes the opposite turn | $\lvert yaw(foll)\rvert < yaw\_limit \wedge \lvert gyroscope.z(foll)\rvert < \gamma \wedge compass(foll) \in interval \wedge [yaw(lead) > yaw(foll)$ (if left turn) $\vee yaw(lead) < yaw(foll)$ (if right turn)$]$ |
| *no_steer_input* | Leader turn, follower receives signal to go straight | $\lvert gyroscope.z(lead)\rvert > 11 \wedge \lvert gyroscope.z(foll)\rvert < 0.01$ |

### 7.6.1 speed_bound

This rule was defined to avoid speeding attacks. In fact, an attacker can intercept the data transmitted from the platoon leader to the follower and modify the passed leader speed value. This value is used to compute the speed goal of the followers. So, if an attacker modifies the speed value with a higher one, the speed goal computation will be abnormal and it will make the follower go too fast. This can cause possible road accidents, caused for example by a too-high entering speed in the hairpin turn, that can cause a collision with other cars or make the vehicle not able to follow the correct trajectory. As before, the code is very similar to the previous cases, as the recognition mechanism is the same.

```
#--Rule 1-Speed bounds (Leader speed is faked)--
        if self.speed > 5:
            if self.speedGoal > upper_limit:
                print("Abnormal behaviour: Faked speed")
                self.speedGoal = self.speed
                sg = ss
```

In this policy, we use two variables: `speed` and `speedGoal`. If the value of `speed`, which is the value of the actual speed of the follower vehicle, is greater than 5MPH, we start monitoring the passed speed, which is contained in `speedGoal`. If this value is greater than the upper limit (`upper_limit`), then we will alert an abnormal behavior to the driver and the vehicle will adjust the speed itself, by referring to the actual speed instead of the passed one. So, now the vehicle will increase or decrease its own speed based on its own collected data, on the trusted data passed and it will base the calculation of the true speed value registered by itself.

### 7.6.2 abnormal_slowdown

With this rule, we want to avoid sudden and dangerous unmotivated stops, or too-low speeds (below 1MPH). As said in previous sections, an attacker can intercept the data transmitted from the platoon leader to the follower and modify the passed leader speed value. So, if an attacker modifies the speed value with a really lower one, the speed goal computation will be abnormal and it will make the follower go too slow, or even stop in the middle of the hairpin turn without any motivated reason. This can cause possible severe road accidents, as the following cars may not have the time to react to a sudden breaking or stop. Similar to what was described in previous attack sections, we should consider the possibility that the data passed is not faked, instead the leader really detects a possible obstacle. So, as presented in the cited section, we rely on LiDAR sensor and its data check.

```
#--Rule 2-Abnormal slowdown (Leader speed is faked)--
        ...
        elif self.speedGoal < lower_limit:
                if not self.override_brake:
                    print("Fake signal, the data are corrupted, override")
                    self.speedGoal = self.speed
                    sg = ss
```

This rule is correlated to the *speed_bound* one, as they share the first check on the follower vehicle speed. So, the "else if" of the script is connected to the one that checks the speedGoal in *speed_bound*. If the registered `speedGoal` value is smaller than the lower limit (`lower_limit`) and the LiDAR check is negative, then we will alert an abnormal behavior to the driver and the vehicle will adjust the speed itself, by referring to the actual speed instead of the passed one. So, now the vehicle will increase or decrease its own speed based on its own collected data, on the trusted data passed and it will base the calculation of the true speed value registered by itself.

### 7.6.3 excessive_angles

This rule was designed to detect and fix possible anomalies in steering angles. In particular, while taking the hairpin, we recorded and collected the normal updating scheme of the steering angle. If a detected value is importantly out of bounds, the system will detect that and solve the problem. This is a rule focused on detecting very detailed attacks, where, for example, the yaw value is slightly modified, such that the vehicle does not detect any initial anomaly, but the car still takes a wrong trajectory, which could lead to going out of road, with consequently probability of crashes.

```
#--Rule 3-Excessive angles (yaw data are faked)
    if (gyro > gyro_max or gyro < gyro_min or
        self.gyroscope[-1] > gyro_max or self.gyroscope[-1] < gyro_min):
        print("Excessive turning angle")
        print("Self Gyroscope = ", self.gyroscope[-1])
        lyaw = yaw
```

In this policy, we use four variables: `lyaw`, `yaw`, `gyro` and `gyroscope` First of all, we check the gyroscope values, both of the leader and of the follower vehicle. If the values of one (or multiple) of those variables are abnormal, that means that they are out of the bounds. This rule is defined for a left-turn hairpin. To adapt it to a right-turn hairpin, we just need to invert the limits sign, as they are mirrored. As a result, the vehicle will alert the driver and independently modify its yaw and steering angle based on its own collected data (including the fundamentals sensors detection) and on the trusted data passed, not considering the faked `lyaw` value.

### 7.6.4 opposite_turn

This rule was defined in order to detect if the signal received during a turn is coherent with the actual vehicle trajectory. So, for example, if the leader platoon takes a left turn hairpin (as in our case) and the data passed to followers indicates a right turn, the system will alert and check with the context data. Otherwise, the following vehicles will take an opposite direction ans this can cause severe accidents and even different kinds of problems. First of all, it can cause the vehicles to take a non-existent turn, so they will crash. In case of no crash, the platoon will be separated and the following vehicles will lose the leader indications.

```
#--Rule 4a-Fake turn signal
#The hairpin is to the left, during the turn the signal indicates to turn opposite (right)
        if yaw < yaw_limit and self.gyroscope[-1] < γ and
        (self.compass in interval):
            if lyaw > yaw:
                print("Error, wrong turn")
                lyaw = yaw
```

In this policy, we use five variables: `speed`, `lyaw`, `yaw`, `gyroscope` and `compass` (that indicates the cardinal directions of the vehicle, expressed in degrees). Here we can define four different cases, referring to four different intervals, by using the "yaw limits" values. First of all, we check in which interval the follower yaw is. Defined this, we check coherency with the follower gyroscope data, by checking if the value is lower than $\gamma$. At the same time, we perform also a check on the compass values, to establish the vehicle direction. If the conditions defined in the if (and previously explained) are respected, this means that there is something abnormal, in particular the faked signal indicates to take a turn opposite to the real one. So, as usual, the vehicle will alert the driver and independently modify its yaw and steering angle based on its own collected data (including the fundamentals sensors detection) and on the trusted data passed, not considering the faked `lyaw` value.

### 7.6.5 no_steer_input

This rule detects an absence of steering input. This means that, if the system detects that the leader vehicle is turning but the passed data are referring to a straight trajectory indication, the rule will alert and find a solution. In this case, it means that the rule recognizes a scenario

in which the leader takes correctly the hairpin turn, but the passed data forces the follower vehicles to go straight. So, an attacker could modify the passed leader yaw data, in order to force the follower vehicles to go straight, possibly out of road or crashing or, in a better scenario, taking another route, losing the connection with the platoon leader. As discussed in 7.5.6, this rule was the most complex one to define in this scenario, because it requires reconstructing and re-calculate the yaw value, starting from the trusted collected data.

```
#--Rule 4-No steer input resilience
if (gyro > α or gyro < -α) and (self.gyroscope[-1] < β and self.gyroscope[-1] > -β):
    print("The platoon leader is turning")
    lyaw = 180 * np.arctan(accz/np.sqrt(accx*accx + accz*accz))/np.pi
```

In this policy, we use four variables: `lyaw`, `gyro`, `accx` (that is the leader acceleration value calculated on the x-axis) and `accz` (that is the leader acceleration value calculated on the z-axis). We check if the platoon leader is turning, by checking the gyro values. If this value is bigger than $\alpha$ (or lower than $-\alpha$, it depends on the turn direction), the leading vehicle is turning, while the follower data indicate going straight, as indicated by the values reported by the follower gyroscope. Even if the basic structure is similar to the previous cases, the reconstruction of the leader yaw used for computing the steering angle is completely different and experimental, also based on the work done and discussed in 7.5.6. So, we had to reconstruct the leader yaw value from the scratch. By looking in CARLA simulator's official documentation, we can retrieve how the yaw value is calculated. Then, by using the formal definition of yaw, described in 6.3, we were able to write the formula

$$180 * np.arctan(accz/np.sqrt(accx * accx + accz * accz))/np.pi$$

to calculate the yaw value, where arctan is the arc-tangent of the z-axis acceleration, divided by the square root of the sum of the z-axis squared and x-axis squared accelerations). Then, we divide this obtained value by $\pi$ and we retrieve the yaw value. So, now the vehicle will modify its yaw and steering angle based on its own collected data, on the trusted data passed, and on the leader yaw reconstruction, not considering the faked `lyaw` value. As said before, this is the more experimental rule. This because the generated yaw value is accurate, but the computation requires some time, such that there is a little delay between the data generation and its passage, due to the short time set for ticks. This is more a simulator-side problem, due

to its construction methods, but it is good to take this aspect into account also in a possible real realization.

# Chapter 8

# Results

## 8.1 Introduction

In this chapter, we will introduce and describe the attacks used to test the policies, defined in Chapter 7. Then, we will discuss the results obtained, by comparing them with a "normal" scenario (where the task is completed without any attack or anomaly) and with the same attack scenario, but without any rules implemented.

As said in Section 7.1, each scenario has its own rules, such that each version of the platooning script is modified and adapted to the request. The same happens for the attacks. They are specific for each scenario, even if the basic idea could be similar. This means that each attack has its own values and parameters, adapted to the attack surface disposed by the testing scenario.

For the sake of testing, each attack situation is coded inside the platooning script, which can be viewed entirely in Appendix A. This because we modify directly the passed data, without the need to intercept it during the transmission. An attacker should do the additional stages of intercepting the data communication while the platoon is moving, which is not a difficult job, and decrypting it (if encryption is performed between data, a security measure that is often taken), that could be more difficult. In the next sections, we will present in order the attacks and the results obtained, with a brief data discussion.

## 8.2 Highway

In this section, we will present three different attacks, two focused on the speeding policies and one on the steering ones, defined in Section 7.2. In Table 8.1 are presented the attacks, with a brief explanation and the variables involved.

Table 8.1: Highway Attacks

| ID | Explanation | Effect |
|---|---|---|
| *speeding_attack* | Force the vehicle to exceed the upper speed limit (90MPH) | Forge passed leader speed = 150MPH |
| *slowing_attack* | Force the vehicle to go below the minimum speed limit (10MPH) | Forge passed leader speed = 0MPH |
| *swerve_attack* | Cause a sudden swerve while the leader vehicle is going straight | Forge passed yaw(lead) = 70 |
| **Rules' values for this scenario, referring to Table 7.1** | | |
| $upper\_limit = 90MPH$ —/— $lower\_limit = 10MPH$ —/— $\tau = 5$<br>$upper\_yaw = 80$ —/— $lower\_yaw = 100$<br>$interval \in [85,95]$ | | |

### 8.2.1 speeding_attack

#### Description

With this attack, we want to test the *speed_bound* policy, presented in Section 7.2.1. Here, an attacker intercepts the data transmission and forge the `speed` value, which represents the leader registered speed, that will be used by the follower as target to compute its own acceleration or deceleration values. In our case, we pass the leader the value "150" as speed reference, after the platoon reaches the real target speed of 30MPH. In this way, the leading vehicle will accelerate, until reaching its constant cruise (around 85/90MPH), while the following ones will receive the input to accelerate and reach the target speed of 150MPH. Without rules, we will expect the following cars to rear-end the vehicle in front (in this case, the first follower will rear-end the leading car and, by naturally decreasing its speed due to the crash, it will be rear-ended by the second platoon follower). If the vehicles are equipped with LiDAR, the rule presented in Section 7.1 will possibly prevent the car from crashing, but it will force the vehicle

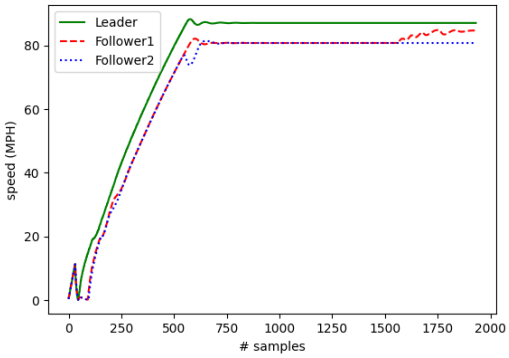
to accelerate and brake repeatedly, in order to maintain the distance from the car in front. This could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulation) and, in a real scenario, it can cause damage to the car.

```
1 #--Attack 1-speeding--
2     ...
3     if self.speed>0.03 and self.speed<30.0:
4         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
5         follower.set_speed_goal(self.speed)
6     if self.speed>=30.0:
7         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
8         follower.set_speed_goal(150)
```

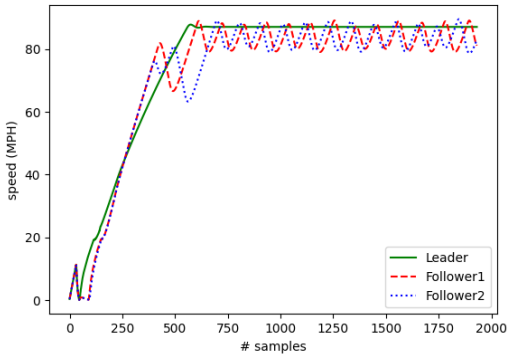
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like the leader yaw, position, and steering values). Then, through function `follower.set_speed_goal`, the leader passes its own actual speed to the followers, which will use that value as `speedGoal` reference. In the first part of the code (lines 3 to 5), we report the usual scenario, such that the data passed is correct. This works until the leader speed is less than 30MPH. When we reach that target speed, we trigger the second part of the script (lines 6 to 8), where we pass to the function `follower.set_speed_goal` our forged value.

### Results

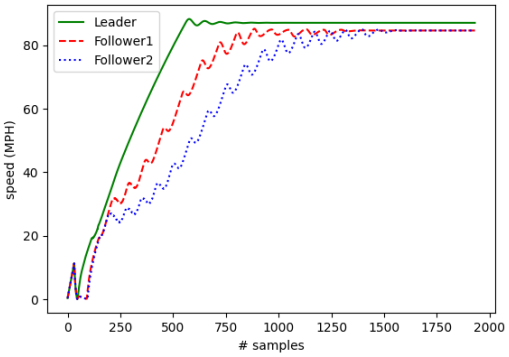
The results obtained by this rule are excellent. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register an important result, in particular the rule works in 20 cases out of 20 trials, so we obtain 100% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was almost every time successful. The reason behind such great success rate of rules is also due to the simplicity of the scenario and of the attack. In fact, such an attack, that modifies only speed parameters is easy to detect, and it can be solved by using the other data that the car receives or register. Nevertheless, speed attacks are one between the most diffused and used [54], so it is great to obtain good results in detecting and recovering from such attacks. Talking about the registered data during the attack, such as speed detection and trajectory study, we could say that the rule works




(a) Normal - Speed




(b) speed_attack - Speed, no rules




(c) speed_attack - Speed, with rules

Figure 8.1: speed_attack - Speed graph

fine in all the scenarios. The only noticeable difference with a normal "non-attack" scenario is the distance between the cars. In fact, when the rule is triggered (so, an attack is detected), the distance between the car is slightly higher, in particular around a meter more than usual. This is not debilitating for the platoon and it is not affecting it at all, considering also the fact that the highway scenario is the one that registers higher speeds (around 85/90MPH), so the safe distance is naturally higher than in other cases. In Figure 8.1, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figure 8.1a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines saturate at approximately 85MPH.

In Figure 8.1b, we report the speed graph when the platoon is under attack, in particular under *speed_attack*, without implementing any resilience policy. In this case, we could see that the followers have an acceleration greater than the leader because, as visible after 40MPH, the followers' lines are more vertical, so they reach the saturation speed faster. Then, we could see a lot of short waves. These represent the LiDAR work, that continuously modifies the acceleration/deceleration value, in order to keep the car in lane behind the leader.

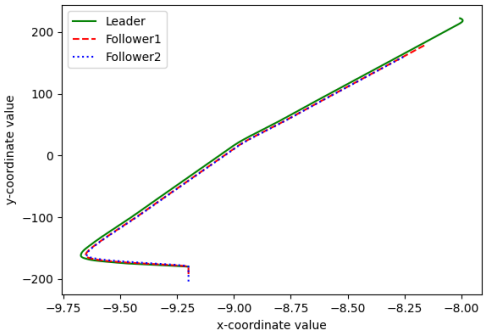
Finally, in Figure 8.1c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, we could see that the followers take longer to reach the plateau speed, and they take a line more similar to a ladder than to a straight line. This because, after 30MPH, the vehicle has to calculate its own speed based on the collected data, which are detected in a determined time interval. Therefore, the graph takes a ladder semblance. This also implies a slightly higher distance between vehicles. At the end of the graph, we could see that the three vehicles reach the plateau speed without problems, and they maintain it without requiring the LiDAR intervention.

In Figure 8.2, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

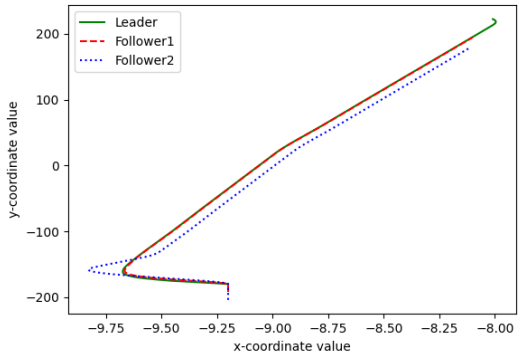
In Figure 8.2a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that there is a short section of the graph (at the start) where the vehicles make some corrections on the position assumed in the lane. Then, all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

In Figure 8.2b, we present the trajectory graph of an attacked platoon, in particular under *speed_attack*, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of the followers are further apart. In particular, the second follower (blue line) is the one that met more problems. In fact, as it can be seen in the left-most position of the graph, the second follower takes a completely different starting trajectory, caused by the higher speed. Then, it is not able to copy perfectly the leader trajectory, as it is highlighted by the space between the leader and follower lines.

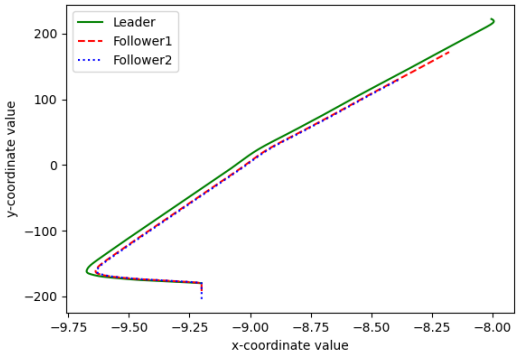
Finally, in Figure 8.2c, we report the study case, where the platoon is under *speed_attack* and the rule is triggered. Here we could see that both the followers copy almost perfectly the leader trajectory, highlighted by the similarity with the "normal" graph in 8.2a. None of the




(a) Normal - Trajectory




(b) speed_attack - Trajectory, no rules




(c) speed_attack - Trajectory, with rules

Figure 8.2: speed_attack - Trajectory graph

followers takes strange trajectories (as in the previous case) and the trajectories distance is small and totally acceptable.

So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are excellent.

### 8.2.2 slowing_attack

#### Description

With this attack, we want to test the *abnormal_slowdown* policy, presented in Section 7.2.2. Here, as in the previous attack, an attacker intercepts the data transmission and forge the `speed` value, which represents the leader registered speed, that will be used by the follower as target to compute its own acceleration or deceleration values. In our case, we pass the leader the value "0" as speed reference, after the platoon reaches the real target speed of 30MPH. In this way, the leading vehicle will accelerate, until reaching its constant cruise (around 85/90MPH), while

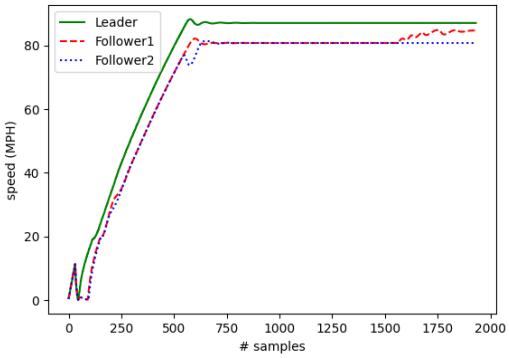
the following ones will receive the input to decelerate and reach the target speed of 0MPH. Without rules, we will expect the following cars to stop in the middle of the highway. This could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulation) and, in a real scenario, it can cause severe crashes or dangerous situations.

```
1 #--Attack 2-slowing--
2     ...
3     if self.speed>0.03 and self.speed<30.0:
4         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
5         follower.set_speed_goal(self.speed)
6     if self.speed>=30.0: #Attack
7         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
8         follower.set_speed_goal(0)
```

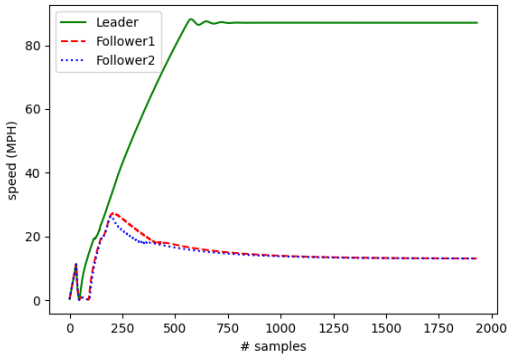
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like the leader yaw, position, and steering values). Then, through function `follower.set_speed_goal`, the leader passes its own actual speed to the followers, which will use that value as `speedGoal` reference. In the first part of the code (lines 3 to 5), we report the usual scenario, such that the data passed is correct. This works until the leader speed is less than 30MPH. When we reach that target speed, we trigger the second part of the script (lines 6 to 8), where we pass to the function `follower.set_speed_goal` our forged value.

### Results

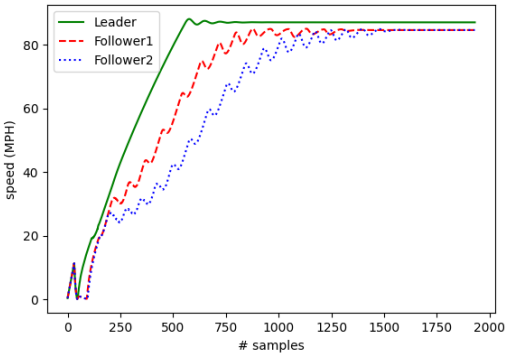
Similarly to the previous test, the results obtained by this rule are excellent. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register an important result, in particular the rule works in 20 cases out of 20 trials, so we obtain 100% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was almost every time successful. Again, probably the reason behind a such great success rate of rules is also due to the simplicity of the scenario and of the attack. In fact, such an attack, that modifies only speed parameters is easy to detect, and it can be solved by using the other data that the car receives or register. Talking about the registered data during the attack, we can find a lot of similarities with the previous attack. As before, the only noticeable




(a) Normal - Speed




(b) slowing_attack - Speed, no rules




(c) slowing_attack - Speed, with rules

Figure 8.3: slowing_attack - Speed graph

difference with a normal "non-attack" scenario is the distance between the cars. In fact, when the rule is triggered (so, an attack is detected), the distance between the car is slightly higher, in particular around a meter more than usual. This is not debilitating for the platoon and it is not affecting it at all, considering also the fact that the highway scenario is the one that registers higher speeds (around 85/90MPH), so the safe distance is naturally higher than in other cases. In Figure 8.3, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figure 8.3a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 85MPH.

In Figure 8.3b, we report the speed graph when the platoon is under the tested attack,

without implementing any resilience policy. In this case, we could see that the followers have an acceleration equal to the leader until they reach 30MPH then, as visible by the red and blue lines, the followers' line drop continuously and equally, decreasing constantly the speed, while the leader graph goes normally, reaching the plateau speed.

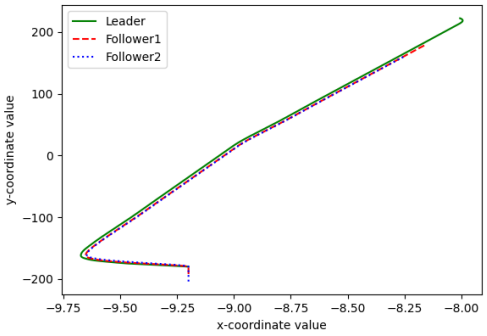
Finally, in Figure 8.3c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, we could see that the followers take longer to reach the plateau speed, and they take a line more similar to a ladder than to a straight line. This because, after 30MPH, the vehicle has to calculate its own speed based on the collected data, which are detected in a determined time interval. Therefore, the graph takes a ladder semblance. This also implies a slightly higher distance between vehicles. At the end of the graph, we could see that the three vehicles reach the plateau speed without problems, and they maintain it without requiring the LiDAR intervention. At first sight, the graphs presented in Figure 8.3c and in Figure 8.1c (referring to the previous attack) are very similar, almost identical. This because the rule has the same effect on the vehicles, even if the scenario is the opposite. This is an additional confirmation of the correct functioning of the rules.

In Figure 8.4, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

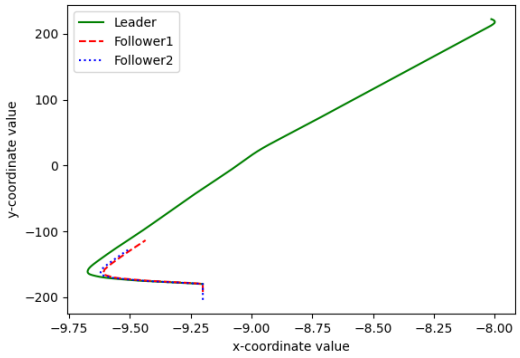
In Figure 8.4a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that there is a short section of the graph (at the start) where the vehicles make some corrections on the position assumed in the lane. Then, all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

In Figure 8.4b, we present the trajectory graph of an attacked platoon, in particular under *speed_attack*, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of the followers are very different. As it can be clearly seen, the trajectories of the followers is extremely short, compared to the leader one. This because the two following vehicles do not reach the leader's target speed, instead they decelerate once they reach 30MPH. So, the speed continuously decreases, and they travel less space.

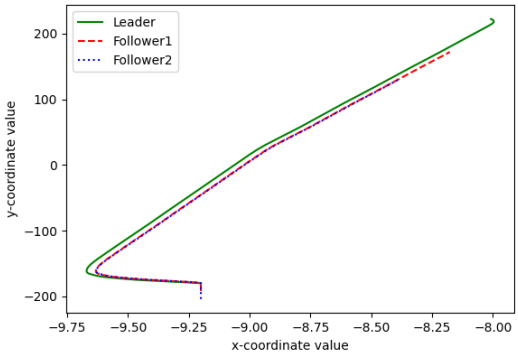
Finally, in Figure 8.4c, we report the study case, where the platoon is under *slowing_attack* and the rule is triggered. Here we could see that both the followers copy almost perfectly the leader trajectory, highlighted by the similarity with the "normal" graph in 8.4a. None of the




(a) Normal - Trajectory




(b) slowing_attack - Trajectory, no rules




(c) slowing_attack - Trajectory, with rules

Figure 8.4: slowing_attack - Trajectory graph

followers takes strange trajectories (as in the previous case) and the trajectories distance is small and totally acceptable. As said while discussing the speed graph, there are some similarities between Figure 8.4c and Figure 8.2c (referring to the previous attack). This because the rule has the same effect on the vehicles, even if the scenario is the opposite. This is an additional confirmation of the correct functioning of the rules.

So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are excellent.

### 8.2.3 swerve_attack

#### Description

With this attack, we want to test the *sudden_swerve* policy, presented in Section 7.2.3. Here, an attacker intercepts the data transmission and forge the `yaw(lead)` value, which represents the leader yaw value, that will be used by the follower as target to compute its own yaw and

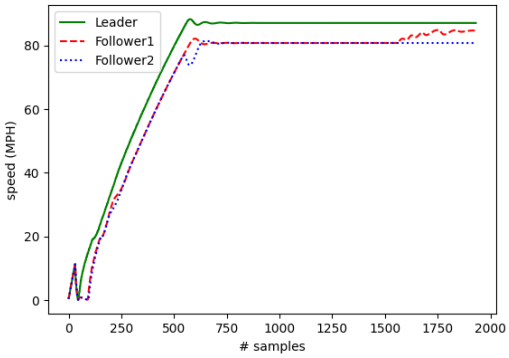
steering values. In our case, we pass the leader the value "70" as yaw reference, after the platoon reaches the real target speed of 30MPH. In this way, the vehicles will accelerate, until reaching 30MPH, where the forged yaw value is passed to the followers. The real value of `yaw(lead)` is close to 90 (as the platoon is going along a straight road). Therefore, by passing a forged value of 70, we will expect the follower vehicles to take a constant turn, crashing into the guardrails, if there are no rules to avoid this. In a real scenario, this could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulation) and it can cause severe crashes or dangerous situations.

```
1 #--Attack 3-forged yaw--
2     ...
3     if self.speed>0.03 and self.speed<30.0:
4         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
5         follower.set_speed_goal(self.speed)
6     if self.speed>=30.0: #Attack
7         follower.add_waypoint([self.x, self.y, 70, self.steer, self.gyroscope[-1]])
8         follower.set_speed_goal(self.speed)
```

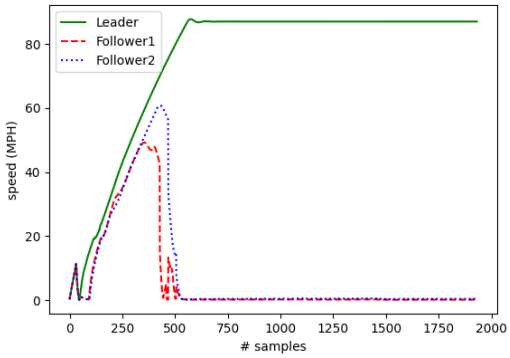
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like our target yaw, position, and steering values). In the first part of the code (lines 3 to 5), we report the usual scenario, such that the data passed is correct. This works until the leader speed is less than 30MPH. When we reach that target speed, we trigger the second part of the script (lines 6 to 8), where we pass to the function `follower.add_waypoint` our forged yaw value.

### Results

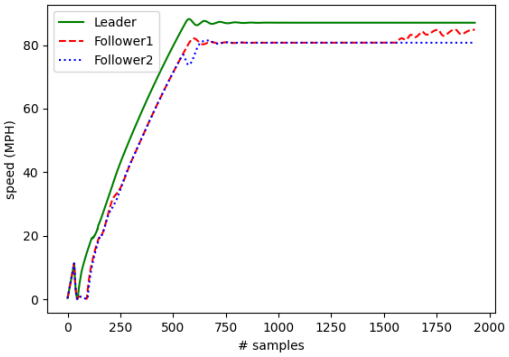
Similarly to the previous tests, the results obtained by this rule are excellent. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register an important result, in particular the rule works in 20 cases out of 20 trials, so we obtain 100% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was almost every time successful. When the rule is triggered, there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. In Figure 8.5, we reported three




(a) Normal - Speed




(b) swerve_attack - Speed, no rules




(c) swerve_attack - Speed, with rules

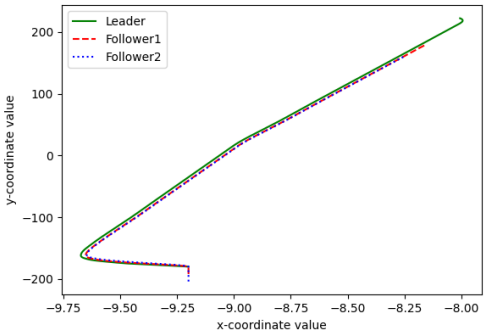
Figure 8.5: swerve_attack - Speed graph

graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

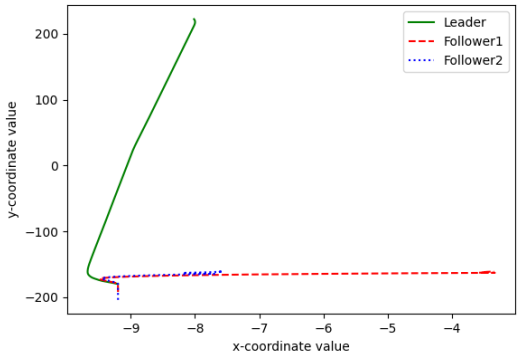
In Figure 8.5a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 85MPH.

In Figure 8.5b, we report the speed graph when the platoon is under the tested attack, without implementing any resilience policy. In this case, we could see that the followers have an acceleration equal to the leader until they reach 30MPH then, as visible by the red and blue lines, the followers' line drop suddenly and freezes at 0. This because the following vehicles stop right before crashing against the middle-lane guardrails, thanks to the LiDAR intervention.

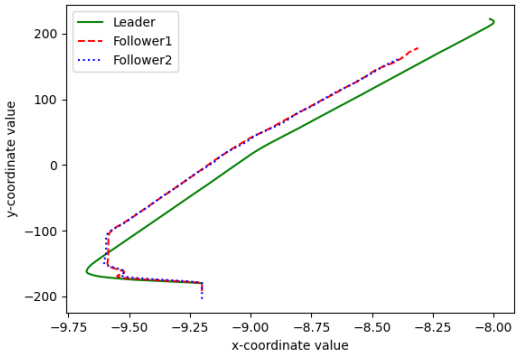
Finally, in Figure 8.5c, we report the interested scenario, where the platoon is under attack




(a) Normal - Trajectory




(b) swerve_attack - Trajectory, no rules




(c) swerve_attack - Trajectory, with rules

Figure 8.6: swerve_attack - Trajectory graph

and the rule is triggered. In this case, this graph and the "normal" one are almost overlapping. They are very similar, all the three cars reach the plateau speed in the correct way without hesitations or strange values.

In Figure 8.6, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

In Figure 8.6a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that there is a short section of the graph (at the start) where the vehicles make some corrections on the position assumed in the lane. Then, all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

In Figure 8.6b, we present the trajectory graph of an attacked platoon, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of

the followers are very different. As it can be clearly seen, the trajectories of the followers are extremely different, compared to the leader one. This because the two following vehicles, when reached the target speed of 30MPH, started turning to the left, in order to meet the 70 yaw value requirements. This forces the vehicles in stopping right in front of the guardrail, saved by the LiDAR from crashing.

Finally, in Figure 8.6c, we report the study case, where the platoon is under *swerve_attack* and the rule is triggered. Here we could see that both the followers copy the leader trajectory, highlighted by the similarity with the "normal" graph in 8.6a. There is a little hesitation in the left-most part of the graph (right when the attack starts), but this does not lead to lane changes or trajectory problems, but just in a slight trajectory correction. Then, the following vehicles copy the leader trajectory perfectly until the end, considering a bit of deviation due to the previous correction, which does not affect the platoon security.

So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are excellent.

## 8.3 Turn

In this section, we will present four different attacks, two focused on the speeding policies and two on the steering ones, defined in Section 7.3. In Table 8.2 are presented the attacks, with a brief explanation and the variables involved.

### 8.3.1 speeding_attack

#### Description

With this attack, we want to test the *speed_bound* policy, presented in Section 7.3.1. Here, an attacker intercepts the data transmission and forge the `speed` value, which represents the leader registered speed, that will be used by the follower as target to compute its own acceleration or deceleration values. In our case, we pass the leader the value "100" as speed reference. In this way, the leading vehicle will accelerate and decelerate, reaching its calculated target speed, while the following ones will receive the input to accelerate and reach the target speed of 100MPH. Without rules, we will expect the following cars to rear-end the vehicle in front (in this case, the first follower will rear-end the leading car and, by naturally decreasing its

Table 8.2: Turn Attacks

| ID | Explanation | Effect |
|---|---|---|
| *speeding_attack* | Force the vehicle to exceed the upper speed limit (35MPH) | Forge passed leader speed = 100MPH |
| *slowing_attack* | Force the vehicle to go below the minimum speed limit (1MPH) | Forge passed leader speed = 0MPH |
| *no_steer_attack* | Force the vehicle to go straight | Forge passed yaw(lead) = 90 |
| *opposite_turn_attack* | Force the vehicle to take the opposite direction while turning | Forge passed yaw(lead) = 20 |
| **Rules' values for this scenario, referring to Table 7.2** | | |
| *upper_limit = 35MPH —/— lower_limit = 1MPH*<br><br>*upper_yaw = 93 —/— lower_yaw = 87*<br><br>*upper_yaw2 = 168 —/— lower_yaw2 = -178*<br><br>$\tau$ =*upper_yaw - lower_yaw = 6 —/—* $\beta$ =*upper_yaw2 - lower_yaw2 = 24,*<br>*with yaw(foll)* $\in [89, 91]$<br><br>*interval_1* $\in [85, 95]$<br><br>*interval_2* $\in [1, 89]$ | | |

speed due to the crash, it will be rear-ended by the second platoon follower). If the vehicles are equipped with LiDAR, the rule presented in Section 7.1 will possibly prevent the car from crashing, but it will force the vehicle to accelerate and brake repeatedly, in order to maintain the distance from the car in front. This could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulation) and, in a real scenario, it can cause damage to the car.

```
#--Attack 1-speeding--
...
if self.speed>=0.03:
        follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
        follower.set_speed_goal(100)
```

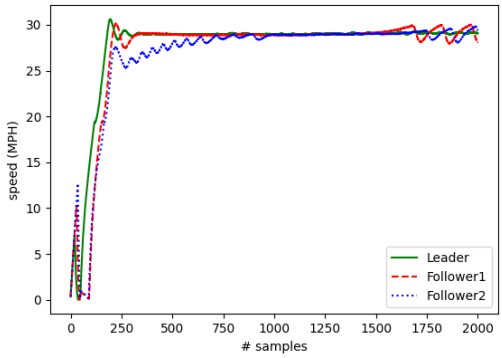
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like the leader yaw, position, and steering values). Then, through function `follower.set_speed_goal`, the leader passes its own actual speed to the followers, which will use that value as `speedGoal` reference. In lines 3 to 5, we trigger the part of the script where we pass to the function `follower.set_speed_goal` our forged value.

### Results

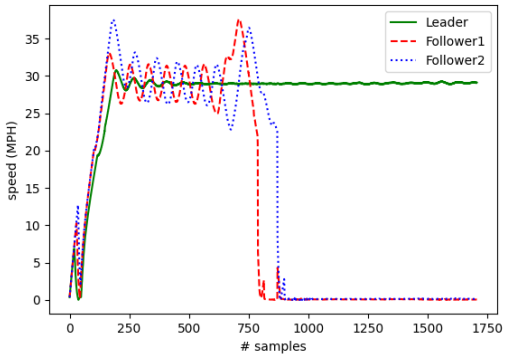
The results obtained by this rule are excellent. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register an important result, in particular the rule works in 20 cases out of 20 trials, so we obtain 100% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was almost every time successful. Talking about the registered data during the attack, such as speed detection and trajectory study, we could say that the rule works fine in all the scenarios. When the rule is triggered, there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. In Figure 8.7, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figure 8.7a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 30MPH.

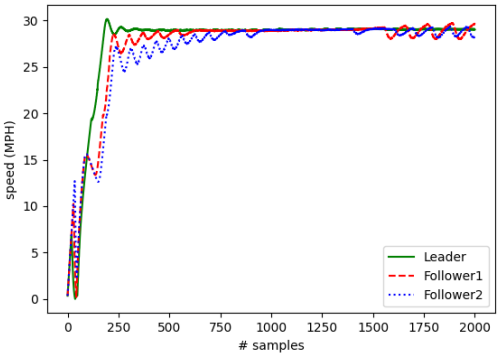
In Figure 8.7b, we report the speed graph when the platoon is under attack, in particular under *speed_attack*, without implementing any resilience policy. In this case, we could see that the followers have an acceleration greater than the leader because, clearly visible as followers' lines are more vertical, so they reach the saturation speed faster. Then, we could see a lot of waves. These represent the LiDAR work, that continuously modifies the acceleration/deceleration value, in order to keep the car in lane behind the leader. At the end of the graph, the follower vehicles' speed suddenly decreases to 0. This because the vehicles went out of road and, thanks to the LiDAR, they stopped right before crashing with some obstacles.




(a) Normal - Speed




(b) speed_attack - Speed, no rules



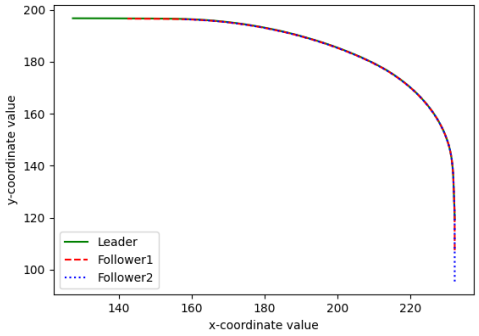

(c) speed_attack - Speed, with rules

Figure 8.7: speed_attack - Speed graph

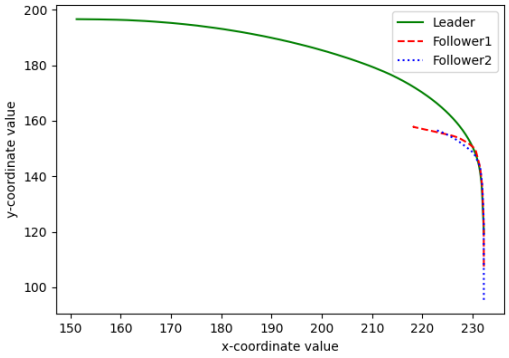
Finally, in Figure 8.7c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, we could see that this graph and the "normal" one are almost overlapping. They are very similar, except for a little hesitation during the acceleration phase, where the followers take a brief deceleration, before re-establish a normal behavior. At the end of the graph, we could see that the three vehicles reach the plateau speed without problems.

In Figure 8.8, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

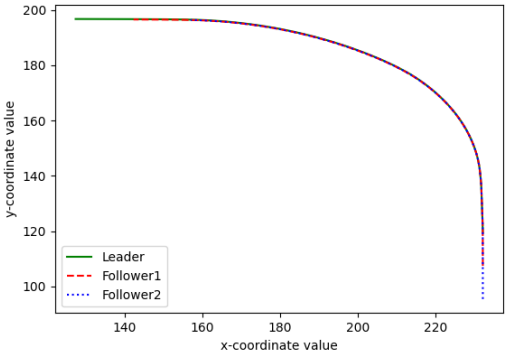
In Figure 8.8a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. All the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.




(a) Normal - Trajectory




(b) speed_attack - Trajectory, no rules




(c) speed_attack - Trajectory, with rules

Figure 8.8: speed_attack - Trajectory graph

In Figure 8.8b, we present the trajectory graph of an attacked platoon, in particular under *speed_attack*, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of the followers are different. In particular, the followers (red and blue lines) take a narrower turn, as highlighted by the lines. This led to a possible crash, avoided by the intervention of LiDAR sensor, that blocks the cars right before crashing with the delimiting fence.

Finally, in Figure 8.8c, we report the study case, where the platoon is under *speed_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph in Figure 8.8a. None of the followers takes strange trajectories (as in the previous case) and the trajectories distance is negligible.

So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are excellent.

### 8.3.2 slowing_attack

#### Description

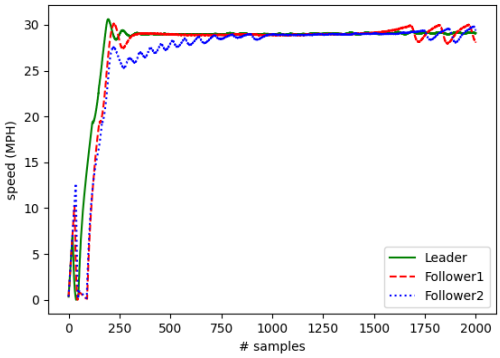
With this attack, we want to test the *abnormal_slowdown* policy, presented in Section 7.3.2. Here, as in the previous attack, an attacker intercepts the data transmission and forge the `speed` value, which represents the leader registered speed, that will be used by the follower as target to compute its own acceleration or deceleration values. In our case, we pass the leader the value "0" as speed reference. In this way, the leading vehicle will accelerate and decelerate, reaching its calculated target speed, while the following ones will not receive any input to accelerate, as the speed goal is set to 0MPH. Without rules, we will expect the following cars to reach a very low speed (around 5/10MPH) and not go over it, while the leading vehicle reaches higher speeds (around 30/35MPH). This because the follower vehicle sensors detect that the car in front is pulling away, but the speed calculation is based on the 0 value, such that there is a sort of "mismatch" between data. This could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulation) and, in a real scenario, it can cause severe crashes or dangerous situations.

```
1 #--Attack 2-slowing--
2 ...
3 if self.speed>=0.03:
4         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
5         follower.set_speed_goal(0)
```

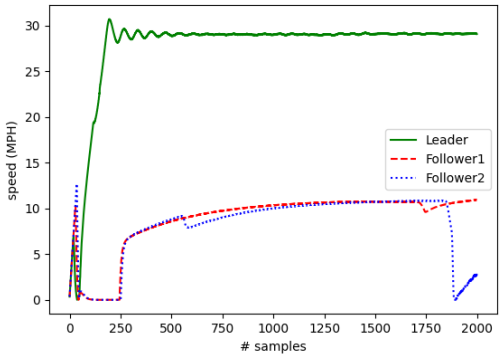
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like the leader yaw, position, and steering values). Then, through function `follower.set_speed_goal`, the leader passes its own actual speed to the followers, which will use that value as *speedGoal* reference. In lines 3 to 5, we trigger the part of the script where we pass to the function `follower.set_speed_goal` our forged value.

#### Results

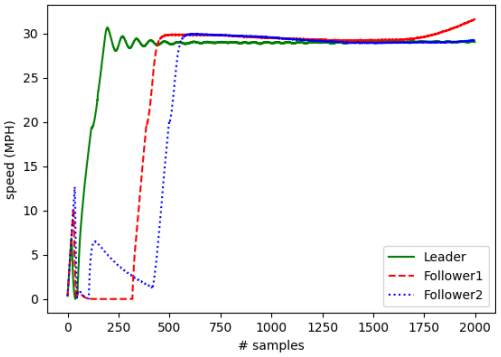
Similarly to the previous test, the results obtained by this rule are excellent. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register an important result, in particular the rule works in 20 cases out of 20 trials, so we obtain 100%




(a) Normal - Speed




(b) slowing_attack - Speed, no rules




(c) slowing_attack - Speed, with rules

Figure 8.9: slowing_attack - Speed graph

of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was almost every time successful. Talking about the data registered during the attack, we can find a lot of similarities with the previous attack. As before, the only noticeable difference with a normal "non-attack" scenario is the distance between the cars. In fact, when the rule is triggered (so, an attack is detected), the distance between the car is slightly higher, in particular around a meter more than usual. This is not debilitating for the platoon and it is not affecting it at all. In Figure 8.9, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figure 8.9a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 30MPH.

In Figure 8.9b, we report the speed graph when the platoon is under the tested attack, without implementing any resilience policy. In this case, we could see that the followers reach a very lower speed (around 10MPH, as expected), while the leader graph goes normally, reaching the plateau speed. At the end of the graph, we could see that the follower speed lines drop to zero. This because the vehicles go out of the road (after losing the connection with the platoon leader) and are stopped by the LiDAR, to avoid a crash with the delimiting fence.

Finally, in Figure 8.9c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, we could see that the followers take longer to reach the plateau speed. This because the vehicle has to calculate its own speed based on the collected data, which are detected in a determined time interval. This also implies a slightly higher distance between vehicles. At the end of the graph, we could see that the three vehicles reach the plateau speed without problems, and they maintain it without requiring the LiDAR intervention.

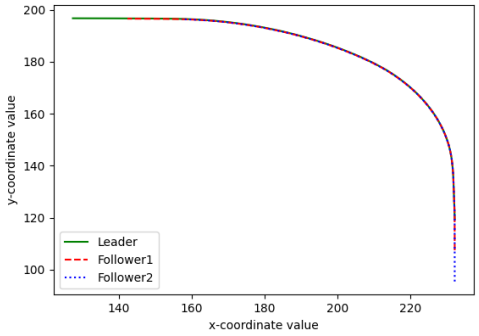
In Figure 8.10, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

In Figure 8.10a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

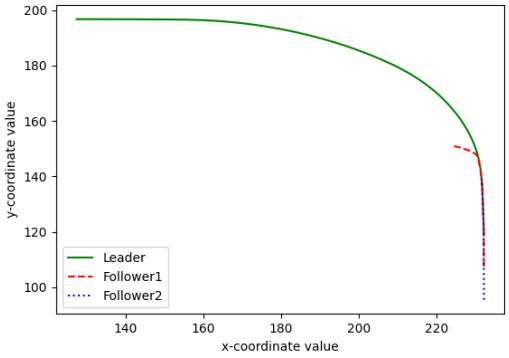
In Figure 8.10b, we present the trajectory graph of an attacked platoon, in particular under *speed_attack*, without implementing any resilience policy. Here, similarly to what happened with the *speeding_attack*, we could see that the trajectories of the leader and of the followers are different. In particular, the followers (red and blue lines) take a narrower turn, as highlighted by the lines. This lead to a possible crash, avoided by the intervention of LiDAR sensor, that blocks the cars right before crashing with the delimiting fence.

Finally, in Figure 8.10c, we report the study case, where the platoon is under *slowing_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph in Figure 8.10a. None of the followers takes strange trajectories (as in the previous case) and the trajectories distance is negligible.

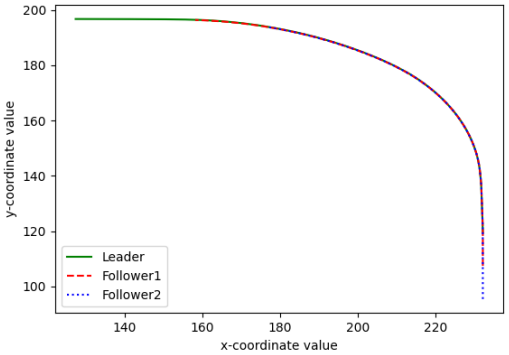
So, considering both the graphs presented (speed and trajectory), we could say that this




(a) Normal - Trajectory




(b) slowing_attack - Trajectory, no rules




(c) slowing_attack - Trajectory, with rules

Figure 8.10: slowing_attack - Trajectory graph

rule is correctly working, and the results obtained are excellent.

### 8.3.3 no_steer_attack

#### Description

With this attack, we want to test both the *no_steer_input* and the *double_turn* policies, presented in Section 7.3.3 and Section 7.3.4. Here, an attacker intercepts the data transmission and forge the `yaw(lead)` value, which represents the leader yaw value, that will be used by the follower as target to compute its own yaw and steering values. In our case, we pass to the followers the value "90" as yaw reference, that is the value that the yaw assumes when the car is going straight, before taking the turn. We will pass this value when the real yaw value is greater than 100 so when the vehicle is approaching the turn. Therefore, by passing a forged value of 90, we will expect the follower vehicles to not take any turn, crashing into the guardrails, if there are no rules to avoid this. In a real scenario, this could lead to abnormal

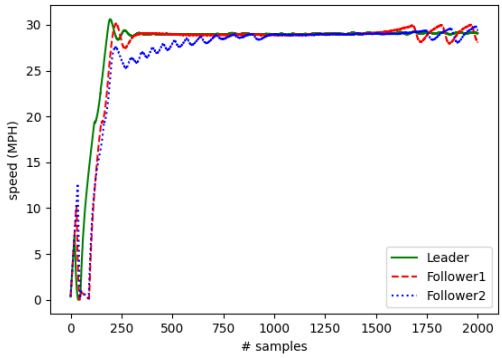
behaviors (i.e., problems in maintaining the trajectory, as tested during the simulations) and it can cause severe crashes or dangerous situations.

```
1 #--Attack 3-no steer input--
2 ...
3 if self.speed>0.03:
4     if self.yaw>100:
5         follower.add_waypoint([self.x, self.y, 90, self.steer, self.gyroscope[-1]])
6         follower.set_speed_goal(self.speed)
7     else:
8         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
9         follower.set_speed_goal(self.speed)
```

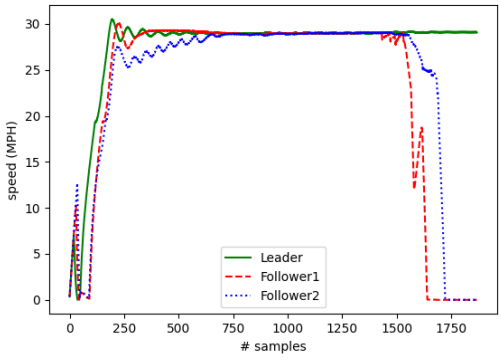
Here, it can be seen how the attack simulation works. In the platooning script, the leader pass its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like our target yaw, position and steering values). In the second part of the code (lines 7 to 9), we report the usual scenario, such that the data passed is correct. This works until the leader's yaw is less than 100. When we reach that target value, we trigger the first part of the script (lines 4 to 6), where we pass to the function `follower.add_waypoint` our forged yaw value.

### Results

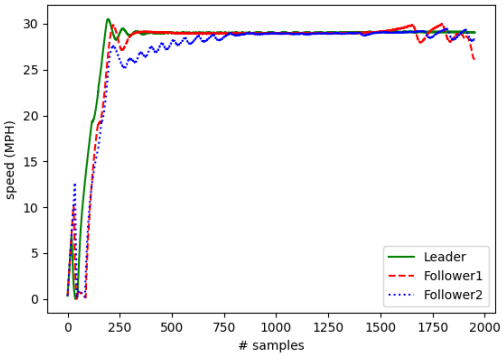
The results obtained by this combination of rules, in response to the presented attack, are great. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register a very good result, in particular the rules work in 17 cases out of 20 trials, so we obtain 85% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rules, which were in most of the cases successful. The errors that we encounter during the simulations are three, mostly connected to the vehicle trajectory. In the first case, the following vehicles take the turn too wide, such that in the middle of it both the vehicles invaded the opposite direction lane. This can be caused by a bad interpretation of the collected data, which indicates a wider turn than real. It was the most frequent error, and also the less problematic one, as the vehicles re-establish the correct trajectory some seconds after the error. The second error encountered involves just one vehicle. In particular, the first follower takes a too-narrow turn, such that it goes out of road, with the danger of crashing into the delimiting fence. This is an error rarely encountered, and it can be caused both by a




(a) Normal - Speed




(b) no_steer_attack - Speed, no rules




(c) no_steer_attack - Speed, with rules

Figure 8.11: no_steer_attack - Speed graph

data misinterpretation, or probably by a simulation environment error, as the second follower proceeds its trajectory without any error. Similarly, the third encountered error involves again just one vehicle, this time the second follower. In fact, right after completing the turn, the second follower takes suddenly a 90 degrees right turn, going out of road. Again, this can be caused by a data misinterpretation or by a simulation problem. In any case, it is again a very rare error. When the rule is triggered, there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. In Figure 8.11, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represents, respectively, the first and the second platoon follower speed.

In Figure 8.11a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the part of the graph, until all the lines reach the plateau at approximately

30MPH.

In Figure 8.11b, we report the speed graph when the platoon is under the tested attack, without implementing any resilience policy. In this case, we could see that the followers have an acceleration equal to the leader. Then, at the end of the graph, the followers' speeds decrease until stop. This because the following vehicles take a wrong trajectory and go out of road, such that the LiDAR has to intervene, in order to avoid a crash.

Finally, in Figure 8.11c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, this graph and the "normal" one are almost overlapping. They are very similar, all the three cars reach the plateau speed in the correct way without hesitations or strange values.

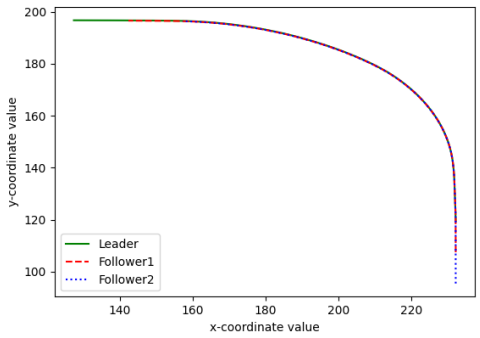
In Figure 8.12, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

In Figure 8.12a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

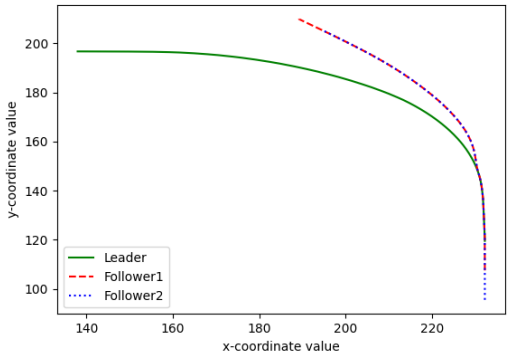
In Figure 8.12b, we present the trajectory graph of an attacked platoon, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of the followers are not overlapping. As it can be clearly seen, the trajectories of the followers is extremely different, compared to the leader one. In fact, both the followers take wider lanes, this because they are going almost straight, instead of correctly taking the turn. It can be seen how the attacks work when yaw is greater than 100, where the three lanes separate.

Finally, in Figure 8.12c, we report the study case, where the platoon is under *no_steer_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph in Figure 8.12a. None of the followers takes strange trajectories (as in the previous case) and the trajectories distance is negligible.

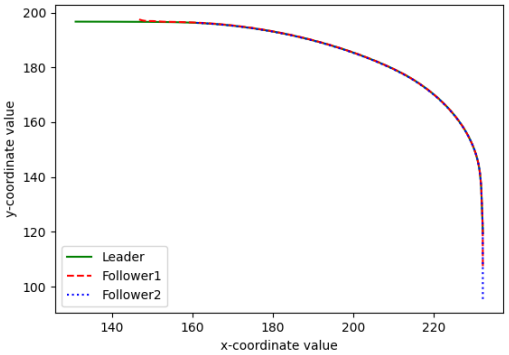
So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are great, considering the low error rate and the distribution of errors, as presented in the attack description.




(a) Normal - Trajectory




(b) no_steer_attack - Trajectory, no rules




(c) no_steer_attack - Trajectory, with rules

Figure 8.12: no_steer_attack - Trajectory graph

### 8.3.4 opposite_turn_attack

#### Description

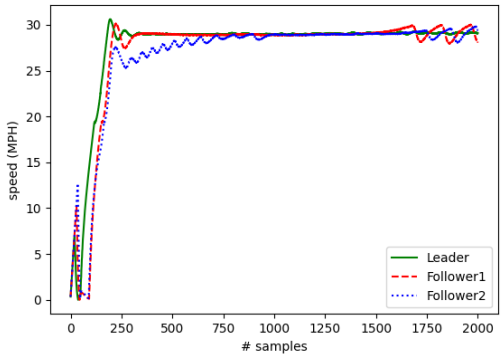
With this attack, we want to test the *fake_turn* policy, presented in Section 7.3.6. Here, an attacker intercepts the data transmission and forge the `yaw(lead)` value, which represents the leader yaw value, used by the followers as target to compute their own yaw and steering values. In our case, we pass to the followers the value "20" as yaw reference, which is a value opposite to the ones obtained in the expected curve. We will pass this value when the real yaw value is greater than 125, so when the vehicles are almost in the middle of the real turn. Therefore, by passing a forged value of 20, we will expect the follower vehicles to take an unexpected turn to the left, going out of road, if there are no rules to avoid this. In a real scenario, this could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulations) and it can cause severe crashes or dangerous situations.

```
1 #--Attack 4-opposite turn attack--
2 ...
3 if self.speed>0.03:
4     if self.yaw>125:
5         follower.add_waypoint([self.x, self.y, 20, self.steer, self.gyroscope[-1]])
6         follower.set_speed_goal(self.speed)
7     else:
8         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
9         follower.set_speed_goal(self.speed)
```

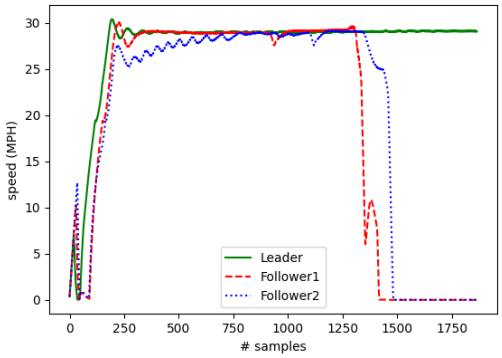
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like our target yaw, position, and steering values). In the second part of the code (lines 7 to 9), we report the usual scenario, such that the data passed is correct. This works until the leader's yaw is less than 125. When we reach that target value, we trigger the first part of the script (lines 4 to 6), where we pass to the function `follower.add_waypoint` our forged yaw value.

### Results

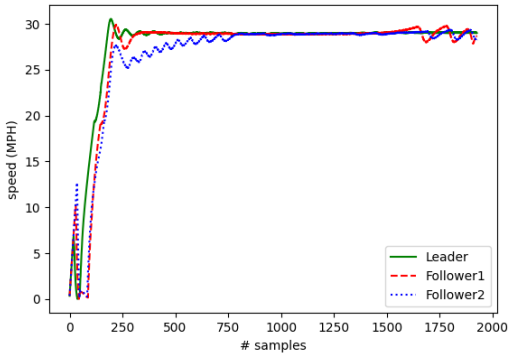
The results obtained by this rule, in response to the presented attack, are good. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register a good result, in particular the rule works in 16 cases out of 20 trials, so we obtain 80% of success rate. This result is confirmed also by the hundred of trials repeated to refine the rule, which was in most of the cases successful. The errors that we encounter during the simulations are three and, similarly to the previous attack, they are mostly connected to the vehicle trajectory. In the first case, we encounter a problem that we detected also in the previous attack. Right after completing the turn, the second follower takes suddenly a 90 degrees right turn, going out of road. This problem can be caused by a data misinterpretation but, supported by the fact that the first follower does not encounter any problem and by the occasional presence of this error also in other attacks, we could attribute this to a simulation error (both caused by a CARLA malfunction or by a script problem). The second error encountered was repeated twice during the 20 simulations in a row. In this case, both the vehicles take a sort of U-turn while turning. This is probably due to a data misinterpretation, such that the rule becomes not effective. The real reason behind this could be a wrong data detection, or also by some wrong peak data




(a) Normal - Speed




(b) opposite_turn_attack - Speed, no rules




(c) opposite_turn_attack - Speed, with rules

Figure 8.13: opposite_turn_attack - Speed graph

due to some noise in the simulation, which could rarely affect it. The third error encountered is the least affecting one. In fact, after the platoon completes correctly the turn, the first follower takes a slight turn to the left, invading the other lane, then it goes back to its lane and continues correctly. We consider this as an error due to the opposite lane invasion (even if it is a very brief one), but probably it is just caused by a slightly wrong data correction, such that the vehicles obtain wrong data on their position for a pair of simulation ticks. When the rule is triggered, there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. In Figure 8.13, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figure 8.13a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost

the same in all the parts of the graph, until all the lines reach the plateau at approximately 30MPH.

In Figure 8.13b, we report the speed graph when the platoon is under the tested attack, without implementing any resilience policy. In this case, we could see that the followers have an acceleration equal to the leader. Then, at the end of the graph, the followers' speeds decrease until stop. This because the following vehicles take a wrong trajectory and go out of road, such that the LiDAR has to intervene, in order to avoid a crash.

Finally, in Figure 8.13c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, this graph and the "normal" one are almost overlapping. They are very similar, all the three cars reach the plateau speed in the correct way without hesitations or strange values.

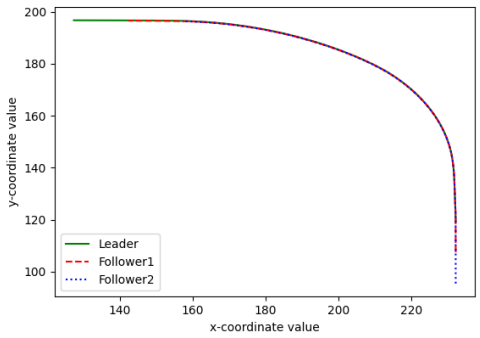
In Figure 8.14, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

In Figure 8.14a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

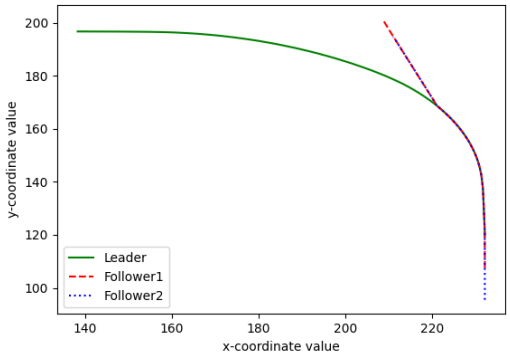
In Figure 8.14b, we present the trajectory graph of an attacked platoon, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of the followers are very different. As it can be clearly seen, the trajectories of the followers is extremely different, compared to the leader one. In fact, we can see that, in the middle of the turn, the two followers take an opposite direction compared to the leader. This because the attack is effective when yaw >= 125, so after that threshold the followers will take an opposite turn in order to satisfy the forged yaw value (20).

Finally, in Figure 8.14c, we report the study case, where the platoon is under *opposite_turn_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph in Figure 8.14a. None of the followers takes strange trajectories (as in the previous case) and the trajectories distance is negligible.

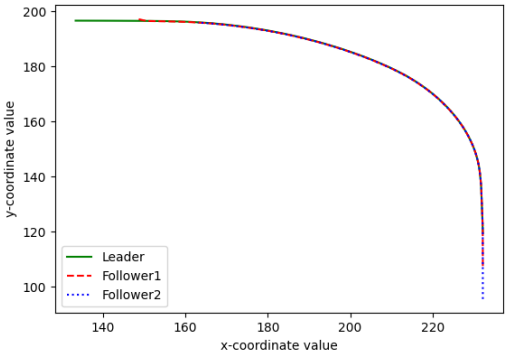
So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are great, considering the low error rate and




(a) Normal - Trajectory




(b) opposite_turn_attack - Trajectory, no rules




(c) opposite_turn_attack - Trajectory, with rules

Figure 8.14: opposite_turn_attack - Trajectory graph

the distribution of errors, as presented in the attack description.

## 8.4 Roundabout

In this section, we will present three different attacks, two focused on the speeding policies and one on the steering ones, defined in Section 7.4. We will report the attacks brought in two different roundabout scenarios: when the platoon takes the first exit (performing less than a quarter of the roundabout) and when the platoon takes the third exit (performing almost the 75% of the roundabout).

We encountered some little simulation problems with this scenario. First of all, sometimes the platoon enters in a loop, such that it continues turning in the roundabout without exiting. These behaviors were discarded from the total count. Then, we notice that in all the trials made (both for development and testing), the autonomous driving systems always avoid taking

the second exit of the roundabout, for a non-defined reason. This is due to some simulator problems or internal policies. In Table 8.3 are presented the attacks, with a brief explanation and the variables involved.

Table 8.3: Roundabout Attacks

| ID | Explanation | Effect |
|---|---|---|
| *speeding_attack* | Force the vehicle to exceed the upper speed limit (35MPH) | Forge passed leader speed = 100MPH |
| *slowing_attack* | Force the vehicle to go below the minimum speed limit (1MPH) | Forge passed leader speed = 0MPH |
| *fake_turn_attack* | Force the vehicle to go straight or take a fake turn | Forge passed yaw(lead) = 180 |
| **Rules' values for this scenario, referring to Table 7.3** | | |
| *upper_limit = 35MPH —/— lower_limit = 0.1MPH*<br>*upper_yaw = -175 —/— lower_yaw = 175*<br>*upper_steer = -0.1 —/— lower_steer = -0.3 —/— normal_steer = -0.16*<br>$\tau$ =*(180 + upper_yaw) + (180 - lower_yaw) = 10,*<br>*with yaw(foll)*$\in [-179, 179]$ | | |

## 8.4.1 speeding_attack

### Description

With this attack, we want to test the *speed_bound* policy, presented in Section 7.4.1. Here, an attacker intercepts the data transmission and forge the `speed` value, which represents the leader registered speed, that will be used by the follower as target to compute its own acceleration or deceleration values. In our case, we pass the leader the value "100" as speed reference. In this way, the leading vehicle will accelerate and decelerate, reaching its calculated target speed, while the following ones will receive the input to accelerate and reach the target speed of 100MPH. Without rules, we will expect the following cars to rear-end the vehicle in front (in this case, the first follower will rear-end the leading car and, by naturally decreasing its speed due to the crash, it will be rear-ended by the second platoon follower). If the vehicles are equipped with LiDAR, the rule presented in Section 7.1 will possibly prevent the car from

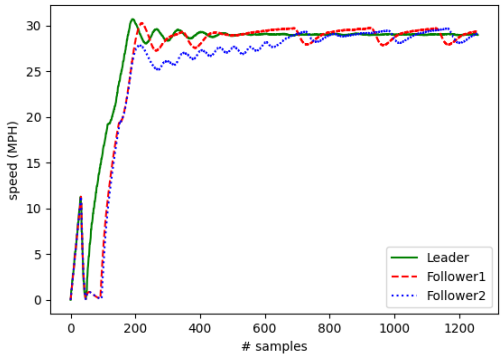
crashing, but it will force the vehicle to accelerate and brake repeatedly, in order to maintain the distance from the car in front. This could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulation) and, in a real scenario, it can cause damage to the car.

```
1 #--Attack 1-speeding--
2 ...
3 if self.speed>=0.03:
4         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
5         follower.set_speed_goal(100)
```

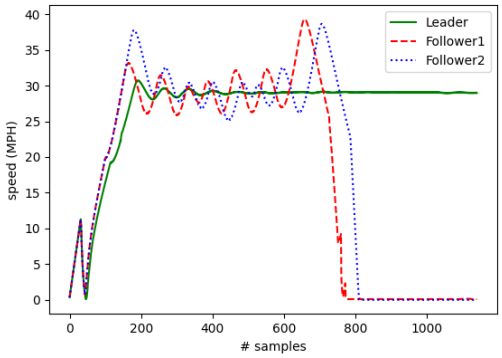
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like the leader yaw, position, and steering values). Then, through function `follower.set_speed_goal`, the leader passes its own actual speed to the followers, which will use that value as `speedGoal` reference. In lines 3 to 5, we trigger the part of the script where we pass to the function `follower.set_speed_goal` our forged value.

### Results

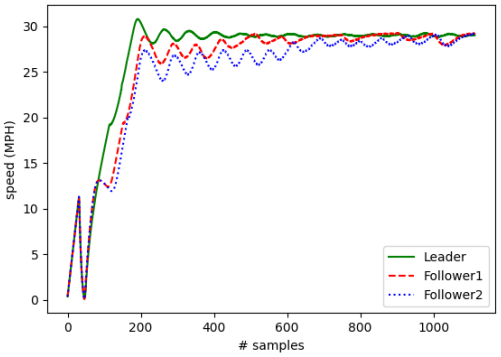
The results obtained by this rule are excellent. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register an important result, in particular the rule works in 19 cases out of 20 trials, so we obtain 95% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was almost every time successful. The only error find in trials, which consequently is the one found also in the 20 repetitions used for success rate detection, is focused on one vehicle only. In fact, the second follower, taking the first exit of the roundabout, goes a little too wide, invading the traffic divider at the roundabout exit. This does not imply a crash (in our trial cases), but we defined it as an error as it takes a wrong trajectory, even if only for a few simulation ticks. Talking about the registered data during the attack, such as speed detection and trajectory study, we could say that the rule works fine in all the scenarios. When the rule is triggered, there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. Just for the sake of completeness, the rule works even when the platoon is stuck in the roundabout, as described




(a) Normal - Speed




(b) speed_attack - Speed, no rules



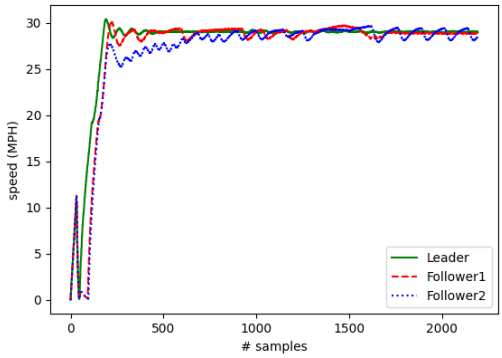

(c) speed_attack - Speed, with rules

Figure 8.15: speed_attack - Speed graph, 1st exit

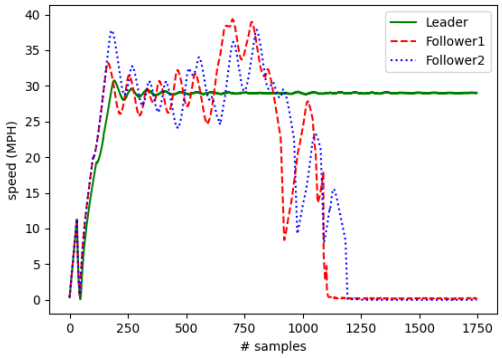
between the encountered problems in Section 8.4, performing correctly one or more roundabout loops. In Figure 8.15, we reported three graphs, focused on the platoon taking the 1st exit and highlighting three different situations. In Figure 8.16, we reported other three graphs, this time focused on the platoon taking the 3rd exit. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figures 8.15a and 8.16a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 30MPH.

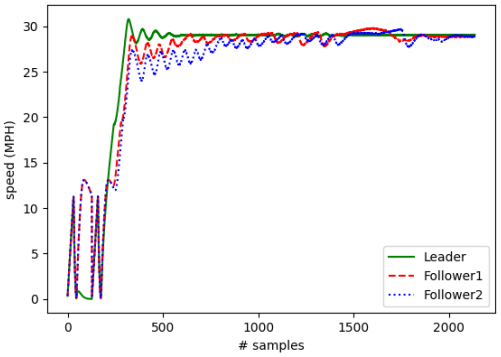
In Figures 8.15b and 8.16b, we report the speed graph when the platoon is under attack, in particular under *speed_attack*, without implementing any resilience policy. In this case, we could see that, in both cases, the followers have an acceleration greater than the leader




(a) Normal - Speed




(b) speed_attack - Speed, no rules

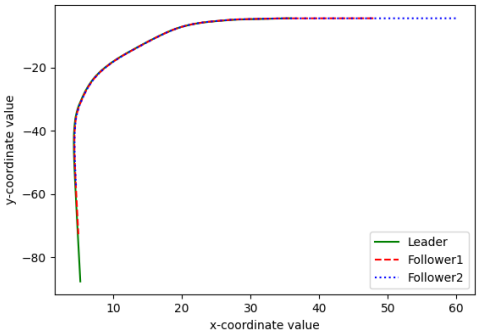



(c) speed_attack - Speed, with rules

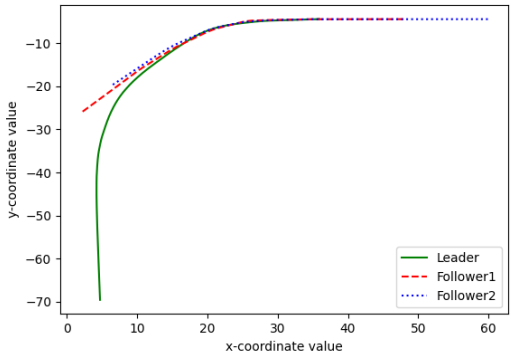
Figure 8.16: speed_attack - Speed graph, 3rd exit

because, clearly visible as followers' lines are more vertical, so they reach the saturation speed faster. Then, we could see a lot of waves. These represent the LiDAR work, that continuously modifies the acceleration/deceleration value, in order to keep the car in lane behind the leader. At the end of the graph, the follower vehicles' speed suddenly decreases to 0. This because the vehicles went out of road and, thanks to the LiDAR, they stopped right before crashing with some obstacles. In Figure 8.15b, the follower vehicles go over the roundabout exit, stopping right before crashing with a street lamp, while in Figure 8.16b, the following vehicles take a narrower turn, ending over the roundabout center.

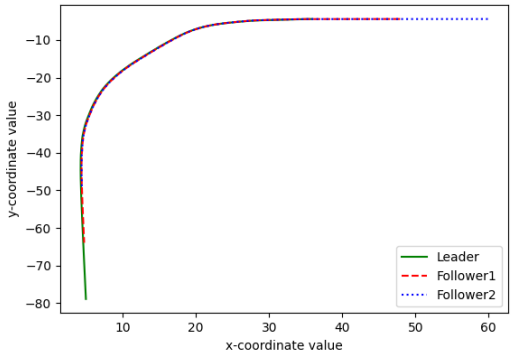
Finally, in Figures 8.15c and 8.16c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, we could see that this graph and the "normal" one are almost overlapping. They are very similar, except for a little hesitation in Figure 8.16c during the acceleration phase, where the followers take a brief deceleration, before re-establish a normal behavior. At the end of the graph, we could see that the three vehicles




(a) Normal - Trajectory




(b) speed_attack - Trajectory, no rules




(c) speed_attack - Trajectory, with rules

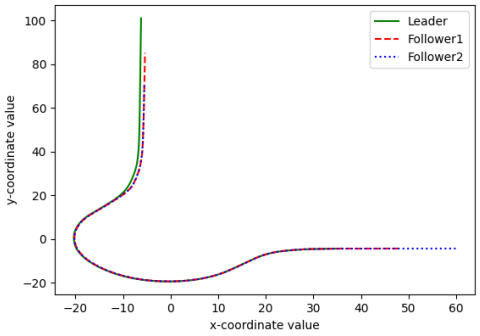
Figure 8.17: speed_attack - Trajectory graph, 1st exit

reach the plateau speed without problems.

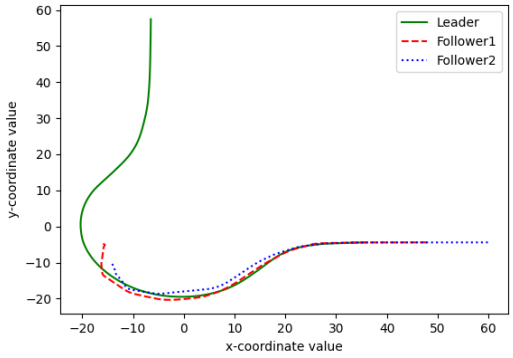
In Figure 8.17, we reported the trajectory graphs, again focusing on the platoon taking the 1st exit and highlighting three different situations, while in Figure 8.18, we focused on the platoon taking the 3rd exit. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

In Figures 8.17a and 8.18a, we report the trajectory graph of a platoon's typical situation, when there are no attacks or abnormal situations. All the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

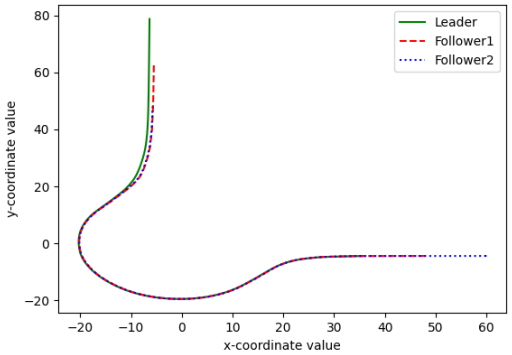
In Figures 8.17b and 8.18b, we present the trajectory graph of an attacked platoon, in particular under *speed_attack*, without implementing any resilience policy. Here, we could see that, in both the cases, the trajectories of the leader and of the followers are different. In particular, in Figure 8.17b, the followers (red and blue lines) take a wider turn, as highlighted




(a) Normal - Trajectory




(b) speed_attack - Trajectory, no rules




(c) speed_attack - Trajectory, with rules

Figure 8.18: speed_attack - Trajectory graph, 3rd exit

by the lines. This implies that, when taking the 1st exit, the two vehicles invade the traffic divider, leading to a possible crash, avoided by the intervention of LiDAR sensor, that blocks the cars right before crashing with a street lamp. Differently, in Figure 8.18b, the followers take a narrower turn, while they are in the middle of the roundabout. This implies that the vehicles go out of the road, in particular into the center of the roundabout, again leading to a possible crash, avoided by the intervention of LiDAR sensor, that blocks the cars right before crashing with what is inside the roundabout isle.

Finally, in Figures 8.17c and 8.18c, we report the study case, where the platoon is under *speed_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph, respectively in Figure 8.17a and Figure 8.18a. None of the followers takes strange trajectories (as in the previous cases) and the trajectories distances are negligible.

So, considering both the graphs presented (speed and trajectory), we could say that this

rule is correctly working, and the results obtained are excellent.

### 8.4.2 slowing_attack

#### Description

With this attack, we want to test the *abnormal_slowdown* policy, presented in Section 7.4.2. Here, as in the previous attack, an attacker intercepts the data transmission and forge the `speed` value, which represents the leader registered speed, that will be used by the follower as target to compute its own acceleration or deceleration values. In our case, we pass the leader the value "0" as speed reference. In this way, the leading vehicle will accelerate and decelerate, reaching its calculated target speed, while the following ones will not receive any input to accelerate, as the speed goal is set to 0MPH. Without rules, we will expect the following cars to reach a very low speed (around 5/10MPH) and not go over it, while the leading vehicle reaches higher speeds (around 30/35MPH). This because the follower vehicle sensors detect that the car in front is pulling away, but the speed calculation is based on the 0 value, such that there is a sort of "mismatch" between data. This could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulation) and, in a real scenario, it can cause severe crashes or dangerous situations.

```
1 #--Attack 2-slowing--
2 ...
3 if self.speed>=0.03:
4         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1]])
5         follower.set_speed_goal(0)
```

Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like the leader yaw, position, and steering values). Then, through function `follower.set_speed_goal`, the leader passes its own actual speed to the followers, which will use that value as `speedGoal` reference. In lines 3 to 5, we trigger the part of the script where we pass to the function `follower.set_speed_goal` our forged value.

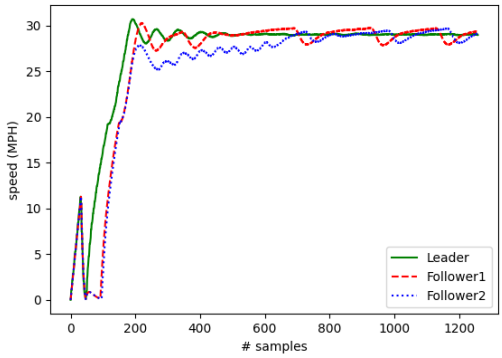
**Results**

Similarly to the previous test, the results obtained by this rule are excellent. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register an important result, in particular the rule works in 20 cases out of 20 trials, so we obtain 100% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was almost every time successful. Talking about the data registered during the attack, we can find a lot of similarities with the previous attack. As before, the only noticeable difference with a normal "non-attack" scenario is the distance between the cars. In fact, when the rule is triggered (so, an attack is detected), the distance between the car is slightly higher. This is not debilitating for the platoon and it is not affecting it at all. In Figure 8.19, we reported three graphs, focused on the platoon taking the 1st exit and highlighting three different situations. In Figure 8.20, we reported other three graphs, this time focused on the platoon taking the 3rd exit. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed..

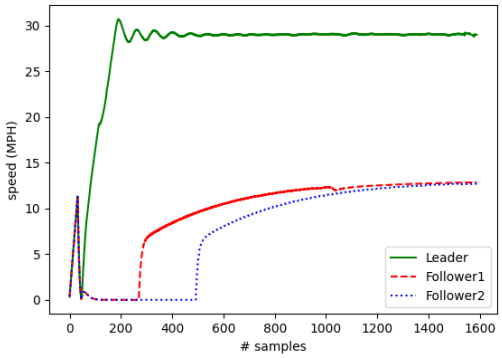
In Figures 8.19a and 8.20a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 30MPH.

In Figures 8.19b and 8.20b, we report the speed graph when the platoon is under the tested attack, without implementing any resilience policy. In this case, we could see that, in both the graphs, the followers reach a very lower speed (around 10MPH, as expected), while the leader graph goes normally, reaching the plateau speed. In particular, at the end of the graph in Figure 8.20b, we could see that the follower speed lines drop to zero. This because the following vehicles take a narrower turn, ending over the roundabout center.

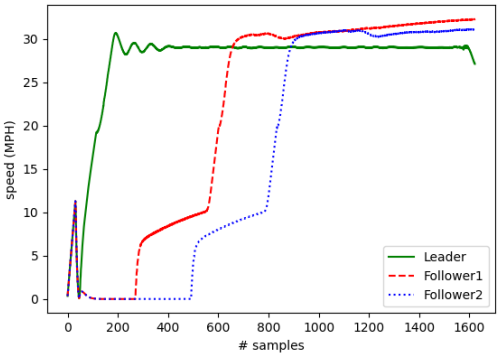
Finally, in Figures 8.19c and 8.20c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, we could see that the followers take longer to reach the plateau speed. This because the vehicle has to calculate its own speed based on the collected data, which are detected in a determined time interval. This also implies a slightly higher distance between vehicles. In particular, in both cases they start delayed with respect to the leader, and they recover from this by accelerating sharply, until they reach the leader's condition. At the end of the graph, we could see that the three vehicles reach the




(a) Normal - Speed




(b) slowing_attack - Speed, no rules




(c) slowing_attack - Speed, with rules

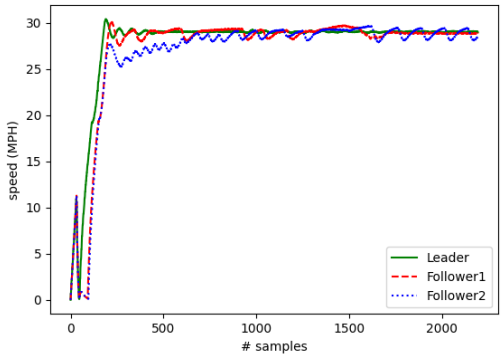
Figure 8.19: slowing_attack - Speed graph, 1st exit

plateau speed without problems, they overtake it for a while (because they accelerate sharply to adequate to the leader speed), then they decelerate and maintain it without requiring the LiDAR intervention.

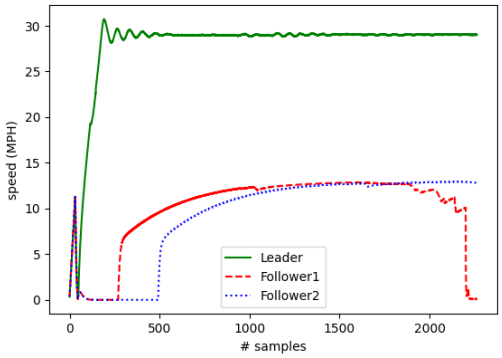
In Figure 8.21, we reported the trajectory graphs, again focusing on the platoon taking the 1st exit and highlighting three different situations, while in Figure 8.22, we focused on the platoon taking the 3rd exit. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory..

In Figures 8.21a and 8.22a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

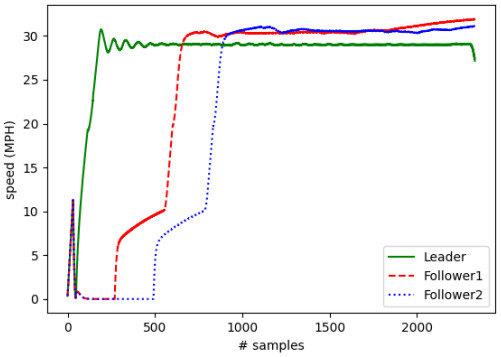
In Figures 8.21b and 8.22b, we present the trajectory graph of an attacked platoon, in particular under *slowing_attack*, without implementing any resilience policy. Here, similarly to




(a) Normal - Speed




(b) slowing_attack - Speed, no rules

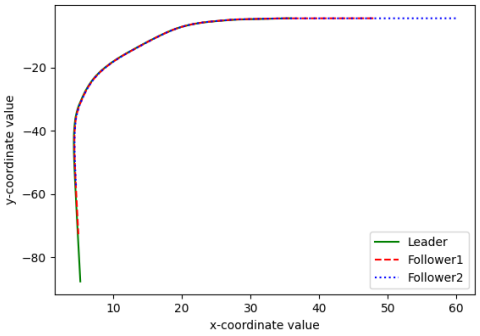



(c) slowing_attack - Speed, with rules

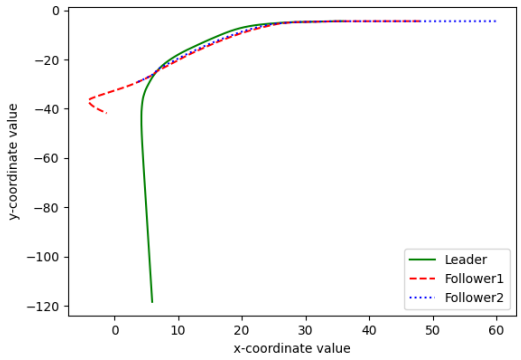
Figure 8.20: slowing_attack - Speed graph, 3rd exit

what happened with the *speeding_attack*, we could see that the trajectories of the leader and of the followers are different, similar to what happened in the previous attack. In particular, in Figure 8.21b, the followers (red and blue lines) take a wider turn, as highlighted by the lines. This implies that, when taking the 1st exit, the two vehicles invade the traffic divider. Then, they take a sudden left turn, crossing the opposite direction lane, ending it on the walk side. Differently, in Figure 8.22b, the followers take a narrower turn, while they are in the middle of the roundabout. This implies that the vehicles go out of the road, in particular into the center of the roundabout, again leading to a possible crash, avoided by the intervention of LiDAR sensor, that blocks the cars right before crashing with what is inside the roundabout isle.

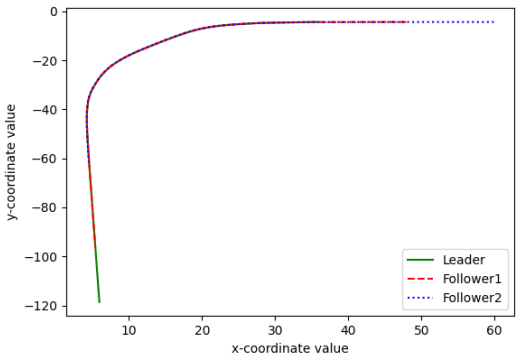
Finally, in Figures 8.21c and 8.22c, we report the study case, where the platoon is under *slowing_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graphs in Figures 8.21a and 8.22a. None of the followers takes strange trajectories (as in the previous case)




(a) Normal - Trajectory




(b) slowing_attack - Trajectory, no rules




(c) slowing_attack - Trajectory, with rules

Figure 8.21: slowing_attack - Trajectory graph, 1st exit

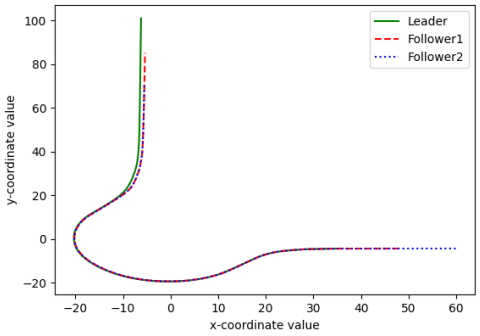
and the trajectories distance is negligible. As a note, both these graphs includes more data samples than usual. This because, as highlighted when we talked about speed, the following vehicles have some delay with respect to the leader one, which does not affect the platoon functioning, but it requires more time to complete the entire "travel" studied.

So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are excellent.

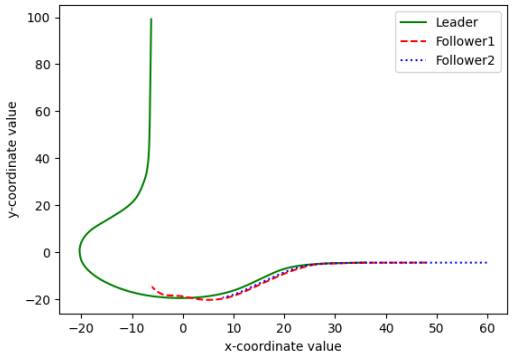
### 8.4.3 fake_turn_attack

#### Description

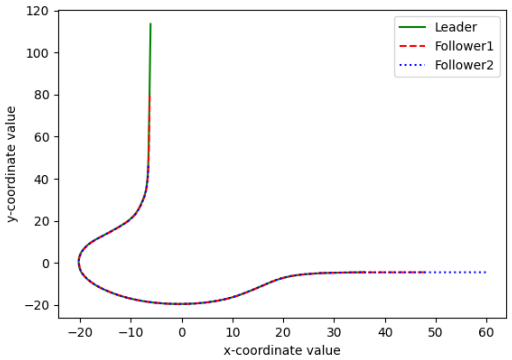
With this attack, we want to test both the *no_steer_input* and the *steer_angles* policies, presented in Section 7.4.3 and Section 7.4.4. Here, an attacker intercepts the data transmission and forge the `yaw(lead)` value, which represents the leader yaw value, that will be used by the follower as target to compute its own yaw and steering values. In our case, we pass to the




(a) Normal - Trajectory




(b) slowing_attack - Trajectory, no rules




(c) slowing_attack - Trajectory, with rules

Figure 8.22: slowing_attack - Trajectory graph, 3rd exit

followers the value "180" as yaw reference, which is the value that the yaw assumes when the car is going straight, before taking the roundabout. Therefore, by passing a forged value of 180, we will expect the follower vehicles to not take any turn, crashing into the guardrails, if there are no rules to avoid this. Then, even if the first *no_steer_input* works, we would have some problems while the followers' vehicles take the roundabout. In fact, with yaw fixed to 180, they will be forced in taking a fake turn to the right, opposite to the roundabout direction. For this reason, we will test also the *steer_angles* policy. In a real scenario, the absence of these rules could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulations) and it can cause severe crashes or dangerous situations.

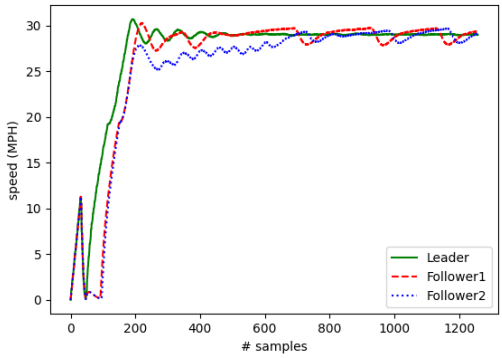
```
1 #--Attack 3-fake turn--
2 ...
3 if self.speed>=0.03:
4         follower.add_waypoint([self.x, self.y, 180, self.steer, self.gyroscope[-1]])
```

```
5         follower.set_speed_goal(self.speed)
```

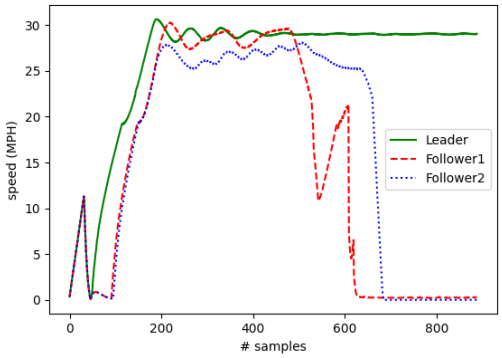
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like our target yaw, position, and steering values). In line 4, we pass to the function `follower.add_waypoint` our forged yaw value.

### Results

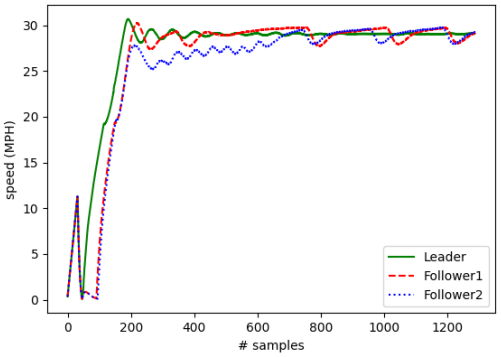
The results obtained by this combination of rules, in response to the presented attack, are good. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register a good result, in particular the rules work in 15 cases out of 20 trials, so we obtain 75% of success rate. This result is confirmed also by the hundreds of trial repeated to refine the rules, which were in most of the cases successful. We should consider the complexity of this scenario, where we should distinguish between different sub-scenarios, some legit and some abnormal. The errors that we encounter during the simulations are three, mostly connected to the vehicle trajectory. In the first case, the leading and the first follower vehicles takes correctly the first exit, while the second follower remained in the roundabout. This error happens really few times in all trials, but two times in the 20 simulations in a row. It could be attributed to a momentary disconnection from the platoon (possibly due to a simulation error) or to a wrong data manipulation. As a result of this, the follower takes a complete loop of the roundabout, exiting the roundabout correctly, while the rest of the platoon already takes some meters of advantage. The second error encountered involves again just one vehicle. In particular, the platoon leader and the second follower takes the fourth exit, while the first follower tries to take the third exit, but it almost crashes into the traffic divider, saved by the LiDAR intervention. This happens one time in the 20 simulations and it can be caused both by a data misinterpretation, or probably by a simulation environment error, as the first follower proceeds its trajectory without any error. Finally, the third encountered error involves again just one vehicle, this time the second follower. In fact, while the leader and the first follower vehicles take a full loop of the roundabout and take the first exit, the second follower takes the fourth exit. Again, this can be caused by a data misinterpretation or by a simulation problem. In any case, it is again a very rare error. The effect of this error is the disconnection of the third vehicle from the platoon, but it does not imply any crash, as in many previous




(a) Normal - Speed




(b) fake_turn_attack - Speed, no rules

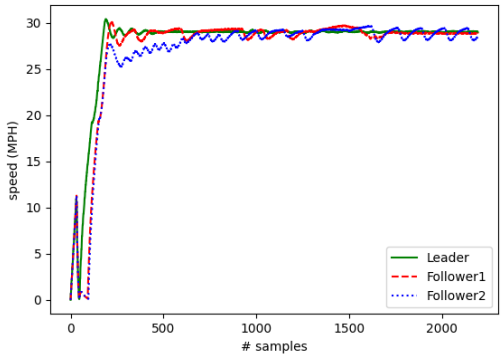



(c) fake_turn_attack - Speed, with rules

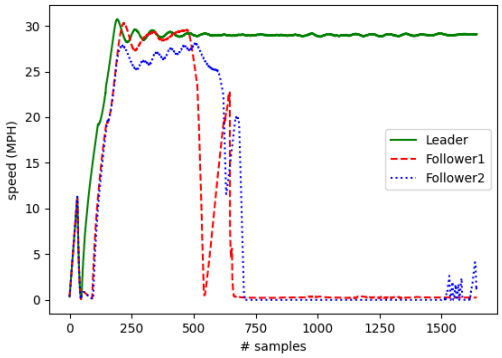
Figure 8.23: fake_turn_attack - Speed graph, 1st exit

cases. When the rule is triggered, usually there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. Sometimes, the follower vehicles take the roundabout less smoothly than usual, and sometimes they choose another lane (as the roundabout used has two lanes). Both these behaviors do not affect the platoon behavior, or the traffic safety. In Figure 8.23, we reported three graphs, focused on the platoon taking the 1st exit and highlighting three different situation. In Figure 8.24, we reported other three graphs, this time focused on the platoon taking the 3rd exit. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

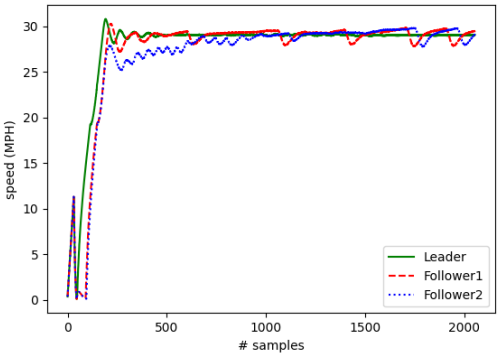
In Figures 8.23a and 8.24a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 30MPH.




(a) Normal - Speed




(b) fake_turn_attack - Speed, no rules



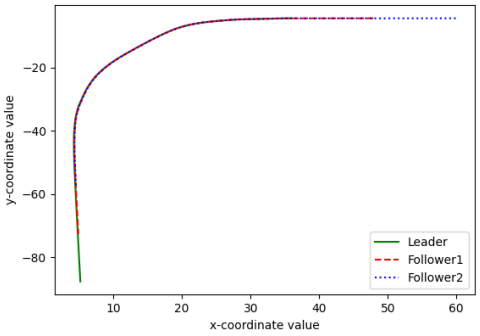

(c) fake_turn_attack - Speed, with rules

Figure 8.24: fake_turn_attack - Speed graph, 3rd exit

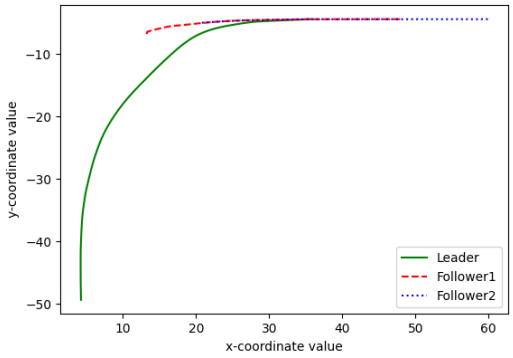
In Figures 8.23b and 8.24b, we report the speed graph when the platoon is under the tested attack, without implementing any resilience policy. In this case, we could see that the followers have an acceleration equal to the leader. Then, near the middle of both the graphs (around x=400), the followers' speed decrease until stop. This because the following vehicles take do not take the turn to enter in the roundabout and they go straight to the roundabout center, such that the LiDAR has to intervene, in order to avoid a crash.

Finally, in Figures 8.23c and 8.24c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In both the cases, this graph and the "normal" one are almost overlapping. They are very similar, all the three cars reach the plateau speed in the correct way without hesitations or strange values.

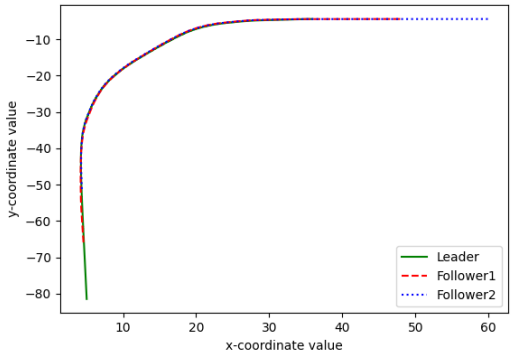
In Figure 8.25, we reported the trajectory graphs, again focused on the platoon taking the 1st exit and highlighting three different situations, while in Figure 8.26, we focused on the platoon taking the 3rd exit. Again, the green line represents the platoon leader trajectory,




(a) Normal - Trajectory




(b) fake_turn_attack - Trajectory, no rules




(c) fake_turn_attack - Trajectory, with rules

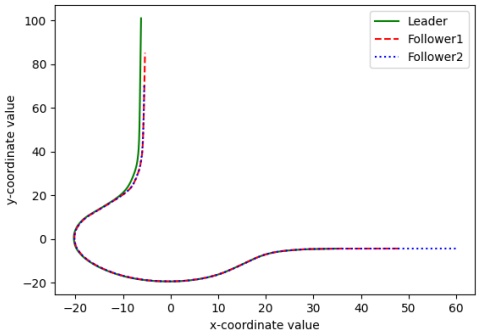
Figure 8.25: fake_turn_attack - Trajectory graph, 1st exit

while the red and blue ones represent, respectively, the first and the second platoon follower trajectory..

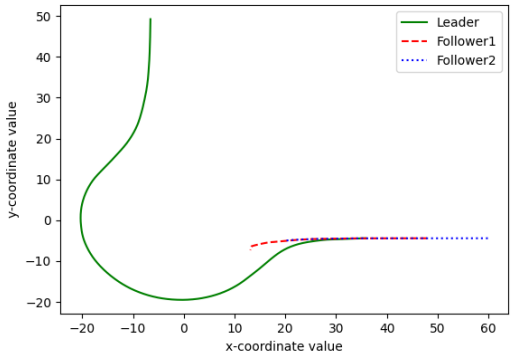
In Figures 8.25a and 8.26a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

In Figures 8.25b and 8.26b, we present the trajectory graph of an attacked platoon, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of the followers are not overlapping. As it can be clearly seen, the trajectories of the followers are extremely different, compared to the leader one. In fact, in both the cases the followers' lanes are interrupted, this because they are going straight, not entering the roundabout. Therefore, they go out of road, straight to the roundabout center, such that the LiDAR has to intervene and stop them, in order to avoid a crash.

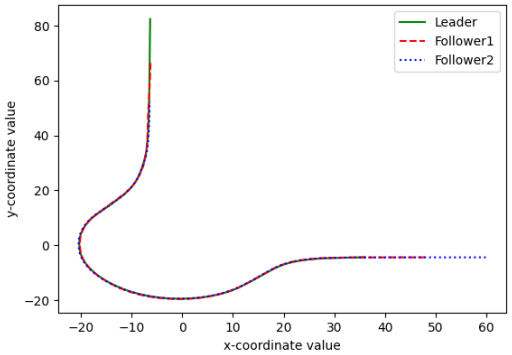
Finally, in Figures 8.25c and 8.26c, we report the study case, where the platoon is under




(a) Normal - Trajectory




(b) fake_turn_attack - Trajectory, no rules




(c) fake_turn_attack - Trajectory, with rules

Figure 8.26: fake_turn_attack - Trajectory graph, 3rd exit

*fake_turn_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph, respectively in Figure 8.25a and in Figure 8.26a. None of the followers takes strange trajectories (as in the previous case) and the trajectories distance is negligible.

So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are great, considering the acceptable error rate and the distribution of errors, as presented in the attack description.

## 8.5 Crossroad

In this section, we will present three different attacks, one focused on the speeding policies and two on the steering ones, defined in Section 7.5. We will report the attacks brought in two different crossroad scenarios: when the platoon go straight after the restart and when the

platoon takes the right turn. In this way, we will demonstrate the completeness of rules in all the scenarios. This scenario was by far the most complex one. In fact, similar to the roundabout one, we should consider many different outcomes, which makes difficult both the rule definition step, both the testing phase. Then, we encounter some problems with the left turn. In fact, even in the normal simulation (without attacks), sometimes the platoon takes that turn a little too wide, going for a few seconds over the sidewalk. This is probably a simulator problem, related to the implemented autonomous driving system. Other than that, we should consider that the turns at these crossroads are very narrow, such that even a little imprecision can cause a wrong trajectory. In Table 8.4 are presented the attacks, with a brief explanation and the variables involved.

Table 8.4: Crossroad Attacks

| **ID** | **Explanation** | **Effect** |
|---|---|---|
| *speeding_attack* | Force the vehicle to exceed the upper speed limit (35MPH) | Forge passed leader speed = 100MPH |
| *fake_turn_attack* | Force the vehicle to turn while the leader goes straight | Forge passed yaw(lead) = 170 |
| *opposite_turn_attack* | Force the vehicle to take the opposite direction after restart | Forge passed yaw(lead) = 170 (if leader turn left) or yaw(lead) = 10 (if leader turn right) |
| **Rules' values for this scenario, referring to Table 7.4** | | |
| $upper\_limit = 35MPH$<br><br>$upper\_yaw = 90.5$ —/— $lower\_yaw = 89.5$<br><br>$\alpha = 0.01$ , $\beta = 1$ , $\tau = 4$ , $\gamma = 15$<br><br>$\delta =$*upper_yaw - lower_yaw = 1*<br><br>$upper\_yaw2 \wedge lower\_yaw2 \in interval$<br>$interval \in ([5, 85] \vee [95, 175])$ | | |

### 8.5.1 speeding_attack

#### Description

With this attack, we want to test three rules: the *speed_bound* policy, presented in Section 7.5.1, the *fake_start* policy, presented in Section 7.5.2 and finally the *no_brake_input* one, presented

in 7.5.3. Here, an attacker intercepts the data transmission and forge the `speed` value, which represents the leader registered speed, that will be used by the follower as target to compute its own acceleration or deceleration values. In our case, we pass the leader the value "100" as speed reference. In this way, the leading vehicle will accelerate and decelerate, reaching its calculated target speed, while the following ones will receive the input to accelerate and reach the target speed of 100MPH. In particular, in this scenario the car will decelerate until reaching a complete stop at the crossing stop line. Then, when the stoplight is green (or the crossroad is free), the platoon will restart. This situation can produce various outcomes, that make this attack more efficient (and consequently, the rule harder to apply) with respect to the previous scenarios. Without rules, we will expect the following cars to rear-end the vehicle in front (in this case, the first follower will rear-end the leading car and, by naturally decreasing its speed due to the crash, it will be rear-ended by the second platoon follower). If the vehicles are equipped with LiDAR, the rule presented in Section 7.1 will possibly prevent the car from crashing, but it will force the vehicle to accelerate and brake repeatedly, in order to maintain the distance from the car in front. This could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulation) and, in a real scenario, it can cause damage to the car.

```
1 #--Attack 1-speeding--
2 ...
3 if self.speed>0.03:
4    follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.accelerometer[0],
5                           self.gyroscope[-1], self.accelerometer[1], self.accelerometer[2]])
6    follower.set_speed_goal(100)
```

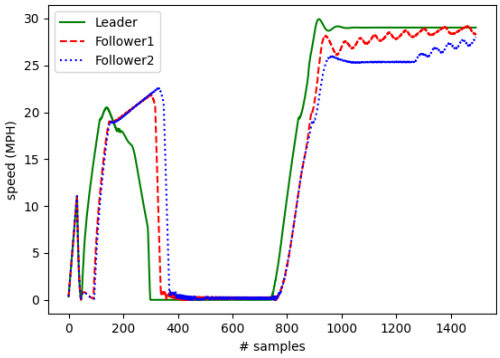
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like the leader yaw, position, and steering values). This time, we pass also all the three accelerometer values (referring to x, y and z axis), that are used in the rules, as defined in Section 7.5. Then, through function `follower.set_speed_goal`, the leader passes its own actual speed to the followers, which will use that value as `speedGoal` reference. In lines 4 to 6, we trigger the part of the script where we pass to the function `follower.set_speed_goal` our forged value.

**Results**

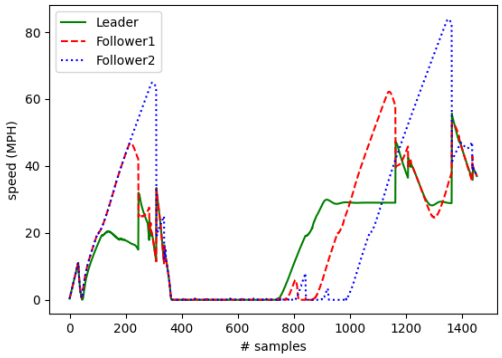
The results obtained by this combination of rules are great. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register an important result, in particular the rules work in 17 cases out of 20 trials, so we obtain 85% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rules, which were almost every time successful. The only error find in trials, which consequently is the one found also in the 20 repetitions used for success rate detection, is the one described also in the introduction. In fact, both the followers, taking the left turn of the crossroad, goes a little too wide, invading for some seconds the sidewalk. This does not imply a crash (in our trial cases), but we defined it as an error as it takes a wrong trajectory, even if only for a few simulation ticks. Talking about the registered data during the attack, such as speed detection and trajectory study, we could say that the rule works fine in all the scenarios. The only noticeable difference with a normal "non-attack" scenario is the distance between the cars. In fact, when the rule is triggered (so, an attack is detected), the distance between the car is slightly higher, in particular around a meter more than usual. This is not debilitating for the platoon and it is not affecting it at all. In Figure 8.27, we reported three graphs, focused on the platoon going straight and highlighting three different situations. In Figure 8.28, we reported other three graphs, this time focused on the platoon taking the right turn. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figures 8.27a and 8.28a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, there are some accelerating/decelerating phases until all the lines reach a sort of plateau at approximately 30MPH. This because the car accelerates for the first part of the brief straight, then decelerates until stopping at the crossing stop line. Then, if the platoon goes straight (Figure 8.27a), the platoon accelerates in the consequent straight, until reaching the plateau speed. The same happens if the platoon takes a turn (as in Figure 8.28a), where, after the restart, the cars take the turn and accelerate in the consequent short straight, until they reach the next crossroad.

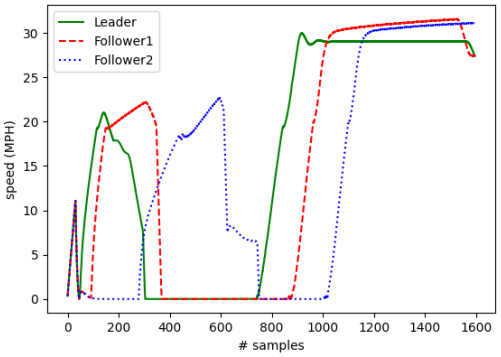
In Figures 8.27b and 8.28b , we report the speed graph when the platoon is under attack, in particular under *speeding_attack*, without implementing any resilience policy. In this case, we could see that, in both cases, the graph is very abnormal, with strange peaks and not aligned




(a) Normal - Speed




(b) speed_attack - Speed, no rules

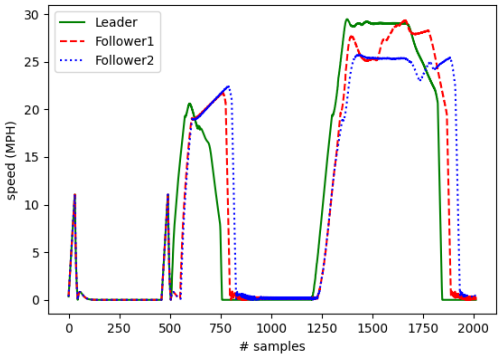



(c) speed_attack - Speed, with rules

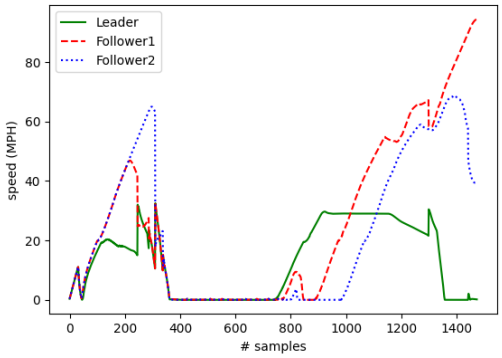
Figure 8.27: speed_attack - Speed graph, Straight

lines. This because the followers' vehicles crash, first into the leader one, then between them. This happens because the leader stops at the crossing stop line, while the follower is not able to stop in time, even after the LiDAR intervention. So, this crash influence also the leader speed, which is pushed into the crossroad center. Then, the platoon restarts, but the acceleration is very fragmented, as it can be seen from x=1000 in both graphs.

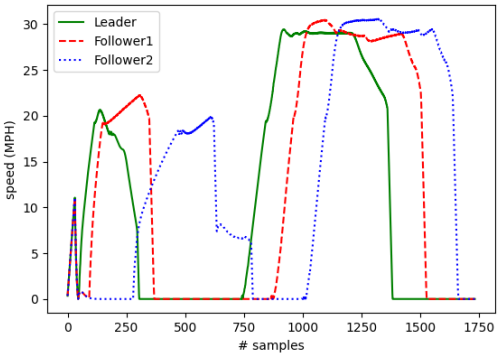
Finally, in Figures 8.27c and 8.28c, we report the interested scenario, where the platoon is under attack and the rules are triggered. In this case, we could see that the behavior of both the graphs is similar to their "normal" cor-respective. The main difference is the delay that occurred with the followers' acceleration, which is clearly visible in graph. This implies a slightly bigger distance between cars, but it does not affect the platoon functioning. In both cases we could see that the lines are equal, just transposed in time. In particular, in 8.28c we could see that also the second stop (x=1500) is correctly done. This means that the rule is correctly working.




(a) Normal - Speed




(b) speed_attack - Speed, no rules



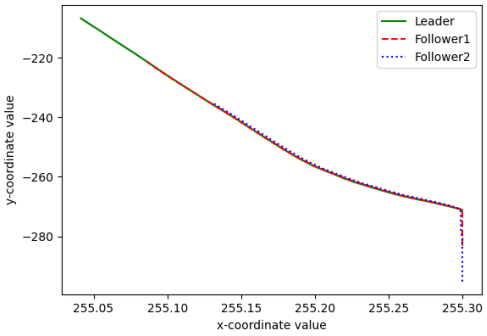

(c) speed_attack - Speed, with rules

Figure 8.28: speed_attack - Speed graph, Turn right

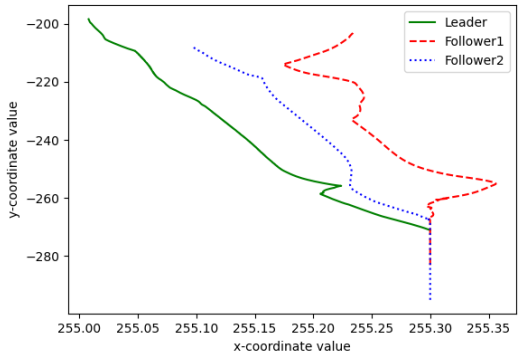
In Figure 8.29, we reported the trajectory graphs, again focusing on the platoon taking the 1st exit and highlighting three different situations, while in Figure 8.30, we focused on the platoon taking the 3rd exit. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

In Figures 8.29a and 8.30a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. All the vehicles travel the same trajectory, as highlighted by the almost overlapping lines. In Figure 8.29a, we could see the adjustment of the vehicles, at the start of the graph.

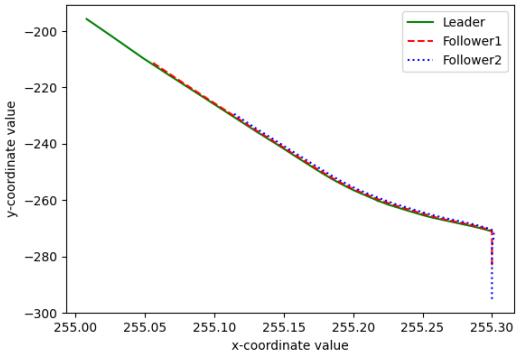
In Figures 8.29b and 8.30b, we present the trajectory graph of an attacked platoon, in particular under *speeding_attack*, without implementing any resilience policy. Here, we could see that, in both the cases, the trajectories of the leader and of the followers are very different. In particular, in Figure 8.29b, the followers (red and blue lines) take a fragmented direction,




(a) Normal - Trajectory




(b) speed_attack - Trajectory, no rules

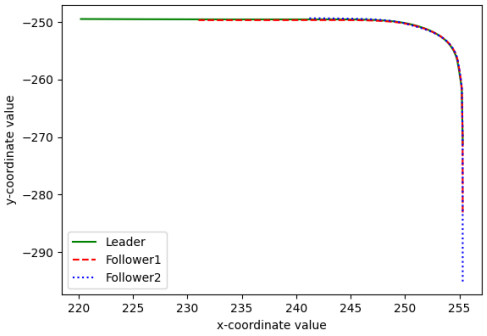



(c) speed_attack - Trajectory, with rules

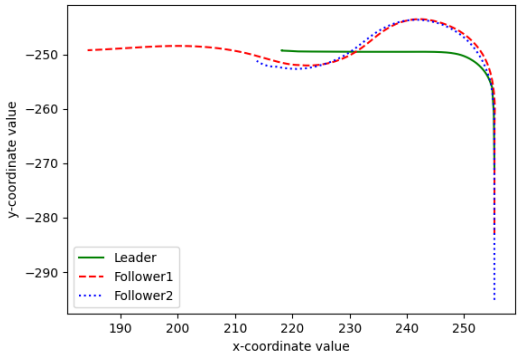
Figure 8.29: speed_attack - Trajectory graph, Straight

not linear as in the "normal" graph. This because, as described in the speed graph section, there is a crash between vehicles (that can clearly be seen when the lines start dividing), which makes the platoon lose its correct trajectory. Similarly, in Figure 8.30b, the followers take a very wider right turn, and they are not able to come back to the correct trajectory after the crash, as it can be seen by the red and blue lines. In particular, they also overtake the platoon leader, and go out of road before the next crossroad.

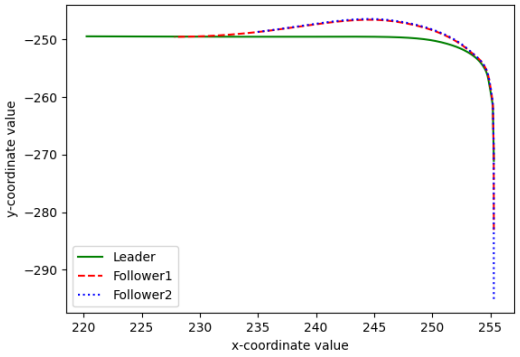
Finally, in Figures 8.29c and 8.30c, we report the study case, where the platoon is under *speeding_attack* and the rules are triggered. Here we could see that, in 8.29c, both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph, in Figure 8.29a. Similarly, in 8.30c, the followers copy almost perfectly the leader trajectory, except in the middle of the turn, where they take a bit wider trajectory. This does not imply invading the other direction lane (otherwise, it will be considered as a wrong attempt), so we consider it as acceptable, as it does not affect the platoon functioning and safety. After the




(a) Normal - Trajectory




(b) speed_attack - Trajectory, no rules




(c) speed_attack - Trajectory, with rules

Figure 8.30: speed_attack - Trajectory graph, Turn right

wider turn, the follower realigns perfectly with the leader. None of the followers takes strange trajectories (as in the previous cases) and the trajectories distances are negligible.

So, considering both the graphs presented (speed and trajectory), we could say that these rules are correctly working, and the results obtained are great, considering the acceptable error rate and the distribution of errors, as presented in the attack description.

### 8.5.2 fake_turn

#### Description

With this attack, we want to test the *fake_turn* policy, presented in Section 7.5.4. Here, an attacker intercepts the data transmission and forge the `yaw(lead)` value, which represents the leader yaw value, that will be used by the follower as target to compute its own yaw and steering values. In our case, we pass to the followers the value "170" as yaw reference, that is the value that the yaw assumes when the car is turning to the right, after the restarting. We pass this

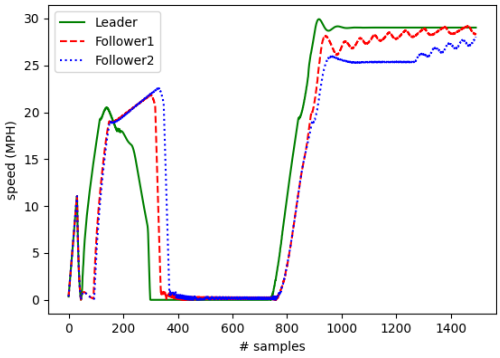
forged value when the real yaw(lead) is in the interval ]85,95[, so when the leading vehicle is going straight after the restart. Therefore, by passing a forged value of 170, we will expect the follower vehicles to turn right, taking the wrong route or crash with the sidewalk obstacles, if there are no rules to avoid this, while the leader continues straight. In a real scenario, the absence of these rules could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulations) and it can cause severe crashes or dangerous situations.

```
1  #--Attack 3-fake turn--
2  ...
3   if self.speed>0.03:
4      if self.yaw > 85 and self.yaw < 95:
5          follower.add_waypoint([self.x, self.y, 170, self.steer, self.accelerometer[0],
6                      self.gyroscope[-1], self.accelerometer[1], self.accelerometer[2]])
7          follower.set_speed_goal(self.speed)
8      else:
9          follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.accelerometer[0],
10                         self.gyroscope[-1], self.accelerometer[1], self.accelerometer[2]])
11         follower.set_speed_goal(self.speed)
```

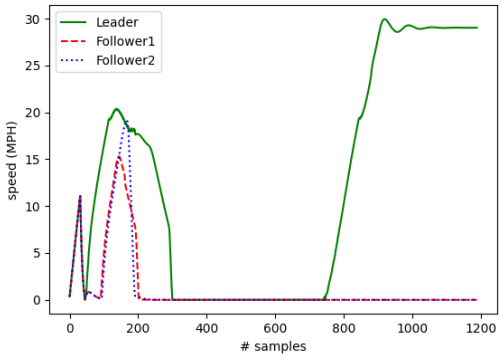
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like our target yaw, position, and steering values). In lines 8 to 11, we pass the values normally, without any modification. Then, when yaw(lead) is between 85 and 95 degrees, we trigger the attack part of the script (lines 4 to 7), where we pass to the function `follower.add_waypoint` our forged yaw value.

### Results

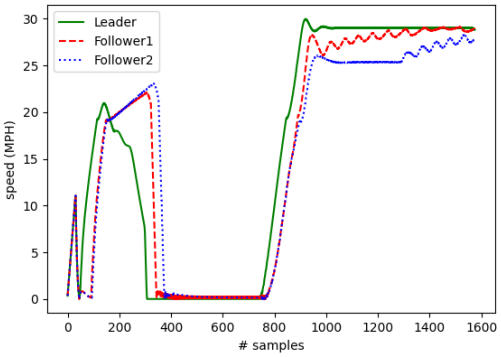
The results obtained by this rule, in response to the presented attack, are great. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register a good result, in particular the rule works in 18 cases out of 20 trials, so we obtain 90% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was in most of the cases successful. We should consider the complexity of this scenario, where we should distinguish between different sub-scenarios, some legit and some abnormal. We encounter only one problem during the simulations, connected to the vehicle trajectory.




(a) Normal - Speed




(b) fake_turn_attack - Speed, no rules




(c) fake_turn_attack - Speed, with rules

Figure 8.31: fake_turn_attack - Speed graph, Straight

This error is related to an abnormal behaviour of the first follower vehicle. In fact, after the fake yaw value is passed, the vehicle does not take a sharp turn to the right (as we will expect in a non-working case), instead it takes a really slight turn to the left, invading the other direction lane. Therefore, the leader and the second follower proceed correctly straight, while the second follower slowly invades the other lane, stopping right before a possible collision with road obstacles. This error happens two times during the 20 simulations in a row, and it was probably due to some data misinterpretation while computing the correct steering angles. When the rule is triggered and working, usually there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. In Figure 8.31, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figure 8.31a, we report the speed graph of a platoon typical situation, when there are no

attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, there are some accelerating/decelerating phases until all the lines reach a sort of plateau at approximately 30MPH. This because the car accelerates for the first part of the brief straight, then decelerates until stopping at the crossing stop line. Therefore, if the platoon goes straight, the vehicles accelerate in the consequent straight, until reaching the plateau speed.

In Figure 8.31b, we report the speed graph when the platoon is under the tested attack, without implementing any resilience policy. In this case, we could see that the followers have an acceleration similar to the leader. Then, the leader restarts and gain speed, while the followers stay stopped at 0MPH. This because they went out of road while the leader was going straight, such that the LiDAR makes them stop before crashing with the sidewalk obstacles.

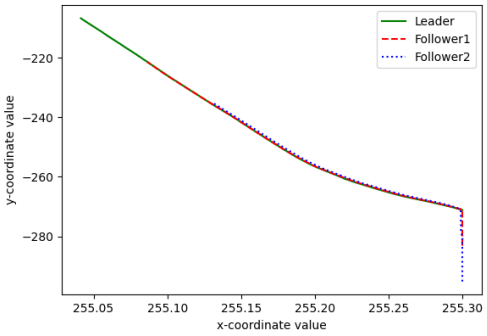
Finally, in Figure 8.31c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In both the cases, this graph and the "normal" one are almost overlapping. They are very similar, all the three cars reach the plateau speed in the correct way without hesitations or strange values.

In Figure 8.32, we reported the trajectory graphs, highlighting three different situations. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory..

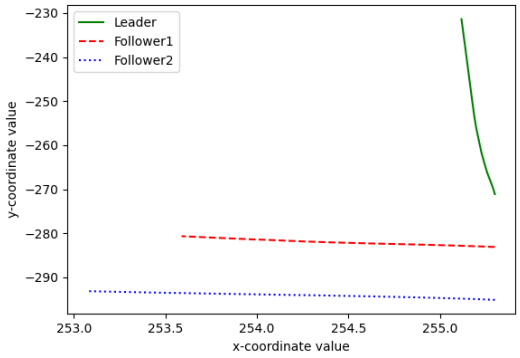
In Figure 8.32a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

In Figures 8.32b, we present the trajectory graph of an attacked platoon, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of the followers are not overlapping, and they take completely different directions. As it can be clearly seen, the trajectories of the followers is opposite, compared to the leader one. In fact, while the leader (green line) has a behavior similar to the one depicted in 8.32a, the red and blue lines (follower vehicles) are represented by an almost straight line, which means they are going out of road, following the fake yaw value of 170.

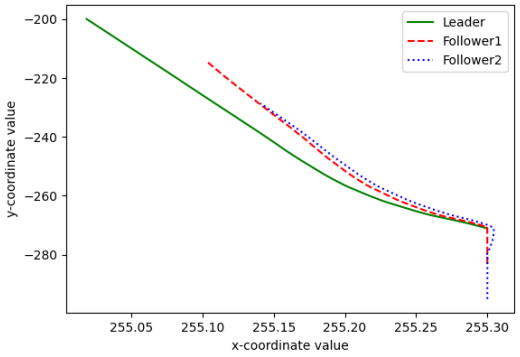
Finally, in Figures 8.32c, we report the study case, where the platoon is under *fake_turn_attack* and the rule is triggered. Here we could see that both the followers copy the leader trajectory, highlighted by the similarity with the "normal" graph, respectively in Figure 8.32a. As visible in the graph, there is a slight trajectory difference between the leader and the vehicles, not




(a) Normal - Trajectory




(b) fake_turn_attack - Trajectory, no rules




(c) fake_turn_attack - Trajectory, with rules

Figure 8.32: fake_turn_attack - Trajectory graph, Straight

visible during the visual simulation. This is caused by a slightly different positioning in the lane. This means that no vehicle invades the opposite lane, so it does not affect the platoon's functioning or safety.

Therefore, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are great, considering the acceptable error rate and the distribution of errors, as presented in the attack description.

### 8.5.3 opposite_turn_attack

#### Description

With this attack, we want to test the *opposite_turn* policy, presented in Section 7.5.5. Here, an attacker intercepts the data transmission and forge the `yaw(lead)` value, which represents the leader yaw value, that will be used by the follower as target to compute its own yaw and steering values. In our case, we pass to the followers the value "170" as yaw reference, if the

leader is turning left (so, if the real leader's yaw is less than 70 degrees), while we pass the value "10" if the leader is turning right (so, if the leader real yaw is greater than 110 degrees). Therefore, by passing a forged opposite yaw value, we will expect the follower vehicles to turn in the opposite direction of the leader, taking the wrong route or crashing with the sidewalk obstacles, if there are no rules to avoid this. In a real scenario, the absence of these rules could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulations) and it can cause severe crashes or dangerous situations.

```
1  #--Attack 3-fake turn--
2  ...
3  if self.speed>0.03:
4      if self.yaw < 70:
5          print("ATTACK 170")
6          follower.add_waypoint([self.x, self.y, 170, self.steer, self.accelerometer[0],
7                      self.gyroscope[-1], self.accelerometer[1], self.accelerometer[2]])
8          follower.set_speed_goal(self.speed)
9      if self.yaw > 110:
10         print("ATTACK 10")
11         follower.add_waypoint([self.x, self.y, 10, self.steer, self.accelerometer[0],
12                     self.gyroscope[-1], self.accelerometer[1], self.accelerometer[2]])
13         follower.set_speed_goal(self.speed)
14     else:
15         follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.accelerometer[0],
16                         self.gyroscope[-1], self.accelerometer[1], self.accelerometer[2]])
17         follower.set_speed_goal(self.speed)
```

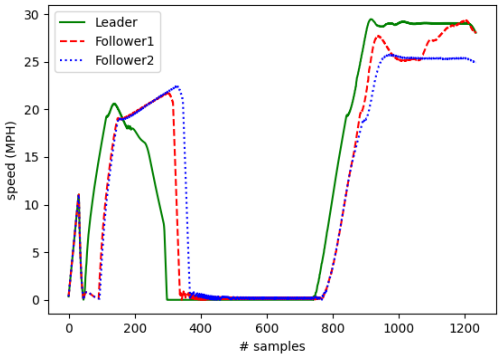
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like our target yaw, position, and steering values). If yaw(lead) is less than 70 degrees, we trigger the first attack part of the script (lines 4 to 8), where the leader is turning left, and we pass to the function `follower.add_waypoint` our forged yaw value of 170 degrees, in order to force the follower vehicles turning right. Instead, if yaw(lead) is greater than 110 degrees, we trigger the second attack part of the script (lines 9 to 13), where the leader is turning right, and we pass to the function `follower.add_waypoint` our forged yaw value of 10 degrees, in order to force the follower vehicles turning left. Then, in lines 14 to 17, we pass the values normally, without any modification.

**Results**

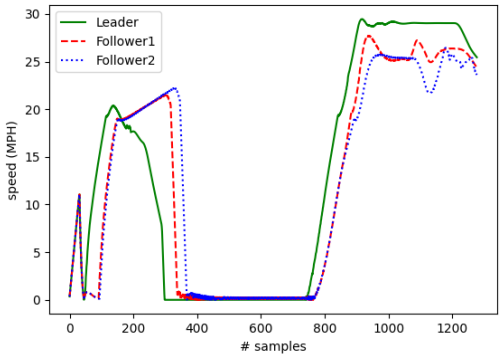
The results obtained by this rule, in response to the presented attack, are great. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register a good result, in particular the rule works in 18 cases out of 20 trials, so we obtain 90% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was in most of the cases successful. We should consider the complexity of this scenario, where we should distinguish between different sub-scenarios, some legit and some abnormal. We encounter two problems during the simulations, connected to the vehicle trajectory. The first error is related to an abnormal behaviour of both the follower vehicles. In fact, in this singular case, the rule is triggered (as shown by the alert on screen), but it does not prevent the follower vehicles from taking the wrong turn. This is a rule failure, that happened just one time between the 20 simulations, and happens very rarely also during the development and testing phases. Then, the second error is more related to some sort of autonomous driving problem. In fact, the rule recognizes correctly the abnormal behavior, and solves the problem. The error is in the steering angles, because the followers take the turn (in this case, the right turn) too narrow, such that they invade partially the sidewalk. Otherwise, when the rule is triggered and working, usually there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. In Figure 8.33, we reported three graphs, focused on the platoon taking the right turn, and highlighting three different situations. In Figure 8.34, we reported other three graphs, this time focused on the platoon taking the left turn. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figures 8.33a and 8.34a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, there is an alternation of different accelerating/decelerating phases. This because the car accelerates for the first part of the brief straight, then decelerates until stopping at the crossing stop line. Then, when the platoon takes a turn, the cars accelerate in the consequent short straight, until they reach the next crossroad.

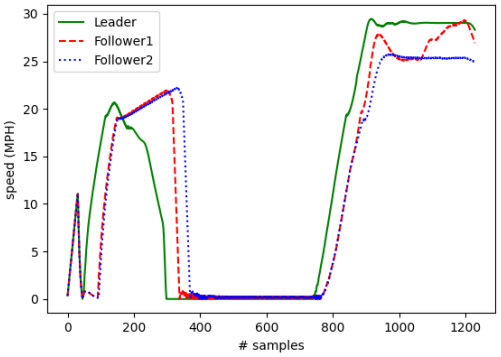
In Figures 8.33b and 8.34b, we report the speed graph when the platoon is under attack, without implementing any resilience policy. In this case, we could see that, in both cases,




(a) Normal - Speed




(b) opposite_turn_attack - Speed, no rules



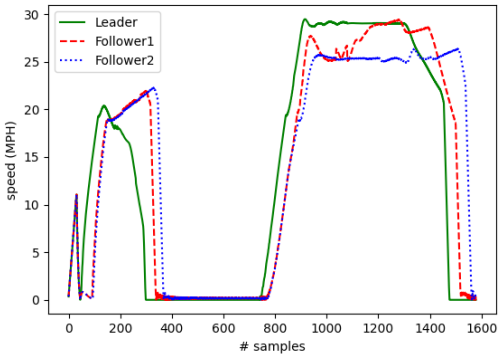

(c) opposite_turn_attack - Speed, with rules

Figure 8.33: opposite_turn_attack - Speed graph, Turn right

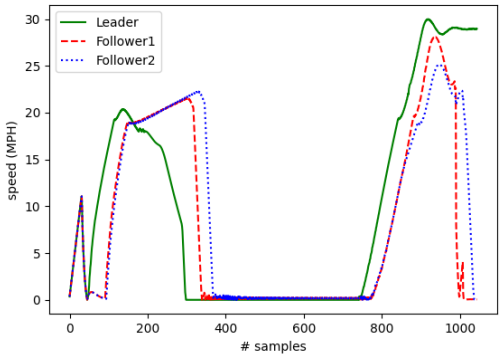
the differences with the usual speed graph are evident in the last part of the graph, where the platoon takes the turn. The difference is not as notable as in other cases, but we can see some inconsistencies, in particular in Figure 8.34b, where the followers' speed suddenly drops to 0MPH. In this case, the situation will be clearer with the trajectory graphs, which will be presented later in this section.

Finally, in Figures 8.33c and 8.34c, we report the interested scenario, where the platoon is under attack and the rules are triggered. In both the cases, this graph and the "normal" one are almost overlapping. They are very similar, all the three cars reach the same speed in the correct way without hesitations or strange values.

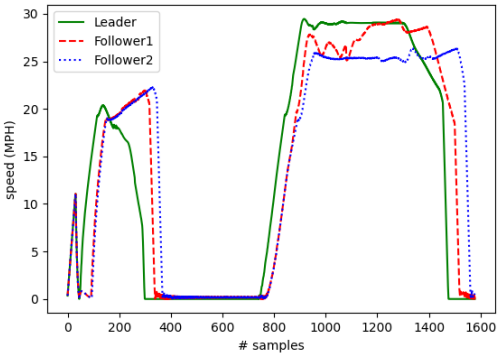
In Figure 8.35, we reported the trajectory graphs, again focusing on the platoon taking the 1st exit and highlighting three different situations, while in Figure 8.36, we focused on the platoon taking the 3rd exit. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower




(a) Normal - Speed




(b) opposite_turn_attack - Speed, no rules




(c) opposite_turn_attack - Speed, with rules

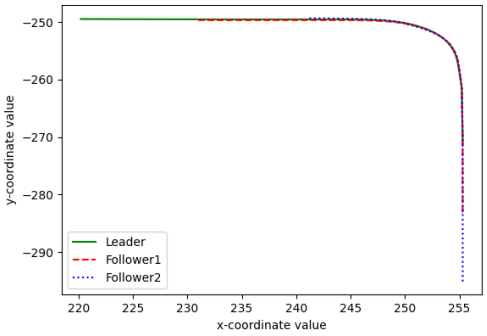
Figure 8.34: opposite_turn_attack - Speed graph, Turn left

trajectory.

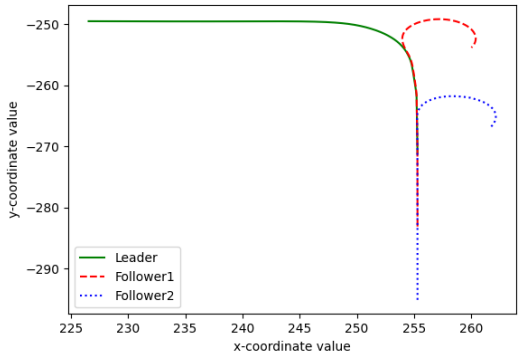
In Figures 8.35a and 8.36a, we report the trajectory graph of a platoon's typical situation, when there are no attacks or abnormal situations. All the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

In Figures 8.35b and 8.36b, we present the trajectory graph of an attacked platoon, without implementing any resilience policy. Here, we could see that, in both the cases, the trajectories of the leader and of the followers are very different. In particular, in Figure 8.35b, the followers (red and blue lines) take a completely opposite direction, turning to the opposite way. Similarly, in Figure 8.36b, the followers take the opposite direction, and they are not able to come back to the correct trajectory after that, as it can be seen by the red and blue lines.

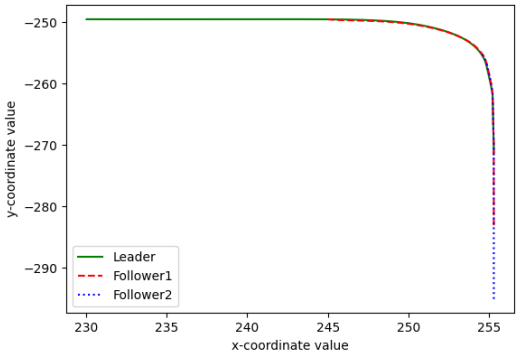
Finally, in Figures 8.35c and 8.36c, we report the study case, where the platoon is under *opposite_turn_attack* and the rules are triggered. Here we could see that, both in Figure 8.35c and in Figure 8.36c, the followers copy perfectly the leader trajectory, highlighted by the




(a) Normal - Trajectory




(b) opposite_turn_attack - Trajectory, no rules




(c) opposite_turn_attack - Trajectory, with rules

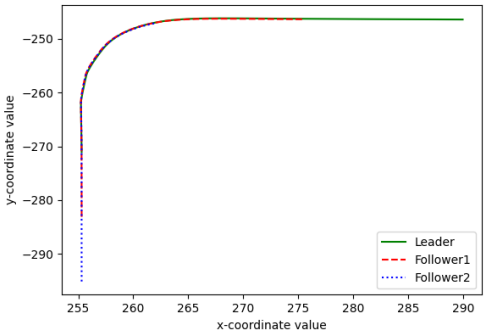
Figure 8.35: opposite_turn_attack - Trajectory graph, Turn right

similarity with the "normal" graph, in Figures 8.35a and 8.36a. None of the followers takes strange trajectories (as in the previous cases) and the trajectories distances are negligible.

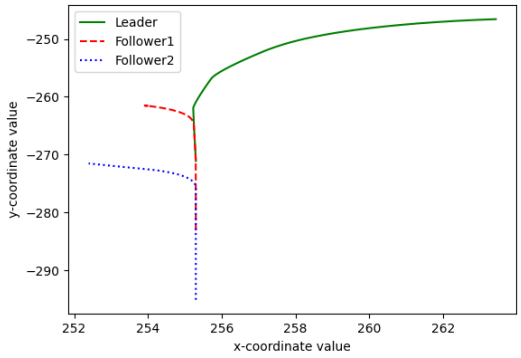
So, considering both the graphs presented (speed and trajectory), we could say that these rules are correctly working, and the results obtained are great, considering the acceptable error rate and the distribution of errors, as presented in the attack description.

## 8.6 Hairpin

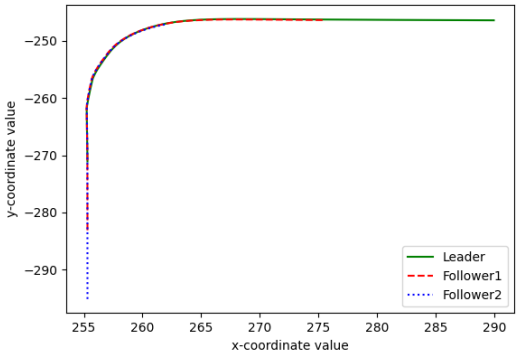
In this section, we will present three different attacks, two focused on the speeding policies and one on the steering ones, defined in Section 7.6. As said in Section 6.4.5, the main difference between a hairpin and a simple turn is the steering angle required, which is more important in the hairpin one, but there are other reasons that make us prefer to have a specific set of rules for it. First of all, we can not rely that much on the yaw value, as made in the turn scenario,




(a) Normal - Trajectory




(b) opposite_turn_attack - Trajectory, no rules




(c) opposite_turn_attack - Trajectory, with rules

Figure 8.36: opposite_turn_attack - Trajectory graph, Turn left

as it is a too much variable value in hairpins, similar to what happens in roundabouts. Then, we have different steering angles during the hairpin approach, not a fixed one (or semi-fixed) as in a normal turn. Finally, we have a very different speed approach in the hairpin, that needs to be addressed much slower than a normal turn. In Table 8.5 are presented the attacks, with a brief explanation and the variables involved.

### 8.6.1 speeding_attack

#### Description

With this attack, we want to test the *speed_bound* policy, presented in Section 7.6.1. Here, an attacker intercepts the data transmission and forge the `speed` value, which represents the leader registered speed, that will be used by the follower as target to compute its own acceleration or deceleration values. In our case, we pass the leader the value "100" as speed reference, after the platoon reaches the real target speed of 20MPH. In this way, the leading vehicle will accelerate

Table 8.5: Hairpin Attacks

| ID | Explanation | Effect |
|---|---|---|
| *speeding_attack* | Force the vehicle to exceed the upper speed limit (35MPH) | Forge passed leader speed = 100MPH |
| *slowing_attack* | Force the vehicle to go below the minimum speed limit (1MPH) | Forge passed leader speed = 0MPH |
| *opposite_turn_attack* | Force the vehicle to take an opposite hairpin turn | Forge passed yaw(lead) = 150 |
| *no_steer_attack* | Force the vehicle to go straight, instead of taking the hairpin turn | Forge passed yaw(lead) = 90 |
| **Rules' values for this scenario, referring to Table 7.5** | | |
| *upper_limit = 35MPH —/— lower_limit = 1MPH*<br>*gyro_max = 20 —/— gyro_min = -40*<br>$yaw\_limit = 110$<br>$\alpha = 11$ , $\beta = 0.01$ , $\gamma = -5$<br>$interval \in [0, 190]$ | | |

and decelerate normally, while the following ones will receive the input to accelerate and reach the target speed of 100MPH. Without rules, we will expect the following cars to rear-end the vehicle in front (in this case, the first follower will rear-end the leading car and, by naturally decreasing its speed due to the crash, it will be rear-ended by the second platoon follower). If the vehicles are equipped with LiDAR, the rule presented in Section 7.1 will possibly prevent the car from crashing, but it will force the vehicle to accelerate and brake repeatedly, in order to maintain the distance from the car in front. This could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulation) and, in a real scenario, it can cause damage to the car.

```
#--Attack 1-speeding--
...
if self.speed>0.03 and self.speed<20:
    follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1],
                self.accelerometer[0], self.accelerometer[1], self.accelerometer[2]])
    follower.set_speed_goal(self.speed)
if self.speed>=20:
    follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1],
                 self.accelerometer[0], self.accelerometer[1], self.accelerometer[2]])
```

```
10      follower.set_speed_goal(100)
```

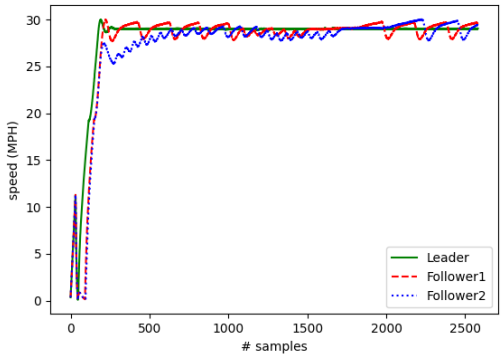
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like the leader yaw, position, and steering values). Then, through function `follower.set_speed_goal`, the leader passes its own actual speed to the followers, which will use that value as `speedGoal` reference. In the first part of the code (lines 3 to 6), we report the usual scenario, such that the data passed is correct. This works until the leader speed is less than 20MPH. When we reach that target speed, we trigger the second part of the script (lines 7 to 10), where we pass to the function `follower.set_speed_goal` our forged value.

### Results

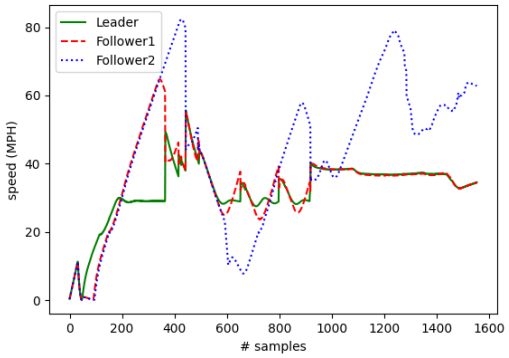
The results obtained by this rule are excellent. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register an important result, in particular the rule works in 20 cases out of 20 trials, so we obtain 100% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was almost every time successful. Talking about the registered data during the attack, such as speed detection and trajectory study, we could say that the rule works fine in all the scenarios. When the rule is triggered, there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. In Figure 8.37, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figure 8.37a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 30MPH.

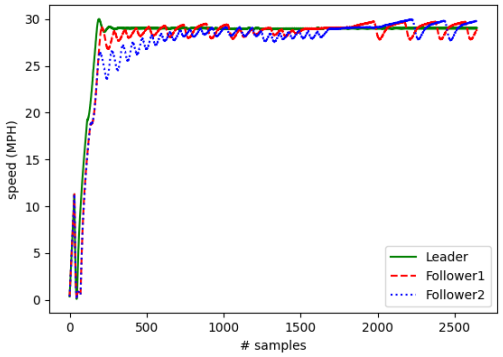
In Figure 8.37b, we report the speed graph when the platoon is under attack, in particular under *speeding_attack*, without implementing any resilience policy. In this case, we obtain a very fragmented graph, also referring to the leader line. This because the LiDAR can not avoid the followers from crashing with the leader's vehicle. As it is visible, the leader (green




(a) Normal - Speed




(b) speed_attack - Speed, no rules

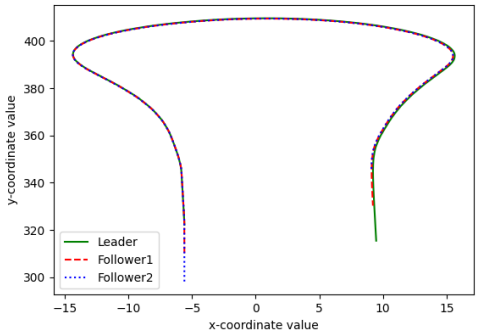



(c) speed_attack - Speed, with rules

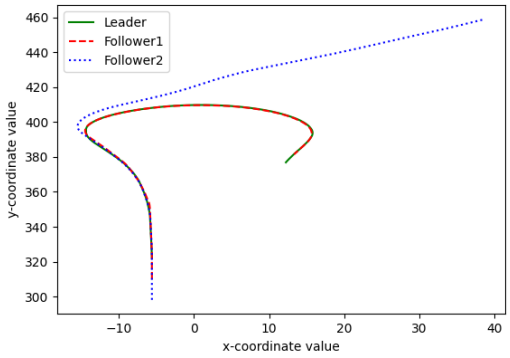
Figure 8.37: speed_attack - Speed graph

line) reaches correctly the plateau speed of 30MPH. Otherwise, both the followers (red and blue lines) reach a really higher speed, in particular the second follower accelerates to more than 80MPH. Then, the LiDAR intervenes, but it is not enough to avoid the first follower crashing with the leader vehicle, as visible for x=400, where the red line (first follower) speed drops suddenly and, oppositely, the leader speed has a peak of around 50MPH (due to the crash). Then, there is also the second follower crash (x=450). After that, the behavior of the platoon becomes very unstable, as the leader tries to reach a constant plateau speed, but it is continuously disturbed by the abnormal behaviors of the following vehicles. This graph will be clearer when compared with the trajectory one, presented later in this section.

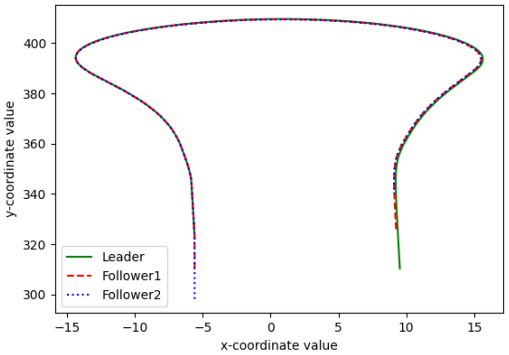
Finally, in Figure 8.37c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, we could see that this graph and the "normal" one are almost overlapping. They are very similar, except for some more waves in the final acceleration phase, just before reaching the platoon, due to some speed adjustment. At the end of the




(a) Normal - Trajectory




(b) speed_attack - Trajectory, no rules




(c) speed_attack - Trajectory, with rules

Figure 8.38: speed_attack - Trajectory graph

graph, we could see that the three vehicles reach the plateau speed without problems, without abnormal waves or LiDAR intervention.

In Figure 8.38, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

In Figure 8.38a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. All the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

In Figure 8.38b, we present the trajectory graph of an attacked platoon, in particular under *speeding_attack*, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of the followers are different. In this case, we should distinguish between the two followers. As said in the speed graph section, both cars crash. While the

first follower continues the hairpin turn, even if not smoothly and continuously disturbing the leader's behavior, the second follower, after the crash, takes a totally different trajectory. In fact, as it can be seen in the graph, it starts going straight, so it goes out of road, not following the platoon's behavior.

Finally, in Figure 8.38c, we report the study case, where the platoon is under *speed_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph in Figure 8.38a. None of the followers takes strange trajectories (as in the previous case) and the trajectories distance is negligible.

So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are excellent.

### 8.6.2 slowing_attack

#### Description

With this attack, we want to test the *abnormal_slowdown* policy, presented in Section 7.6.2. Here, as in the previous attack, an attacker intercepts the data transmission and forge the `speed` value, which represents the leader registered speed, that will be used by the follower as target to compute its own acceleration or deceleration values. In our case, we pass the leader the value "0" as speed reference, after the platoon reaches the real target speed of 20MPH. In this way, the leading vehicle will accelerate and decelerate, reaching its calculated target speed, while the following ones will not receive any input to accelerate, as the speed goal is set to 0MPH. Without rules, we will expect the following cars to reach a very low speed (around 5/10MPH) and not go over it, while the leading vehicle reaches higher speeds (around 30/35MPH). This because the follower vehicle sensors detect that the car in front is pulling away, but the speed calculation is based on the 0 value, such that there is a sort of "mismatch" between data. This could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulation) and, in a real scenario, it can cause severe crashes or dangerous situations.

```
1 #--Attack 2-slowing--
2 ...
3 if self.speed>0.03 and self.speed<20:
4     follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1],
```

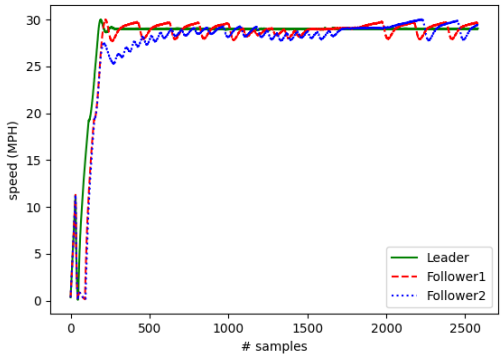
```
5                 self.accelerometer[0], self.accelerometer[1], self.accelerometer[2]])
6     follower.set_speed_goal(self.speed)
7 if self.speed>=20:
8     follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1],
9                  self.accelerometer[0], self.accelerometer[1], self.accelerometer[2]])
10    follower.set_speed_goal(0)
```

Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like the leader yaw, position, and steering values). Then, through function `follower.set_speed_goal`, the leader passes its own actual speed to the followers, which will use that value as `speedGoal` reference. In the first part of the code (lines 3 to 6), we report the usual scenario, such that the data passed is correct. This works until the leader speed is less than 20MPH. When we reach that target speed, we trigger the second part of the script (lines 7 to 10), where we pass to the function `follower.set_speed_goal` our forged value.

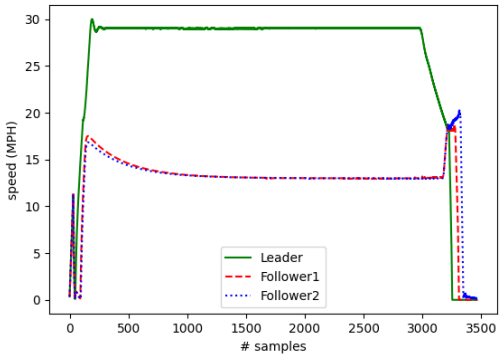
### Results

Similarly to the previous test, the results obtained by this rule are excellent. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register an important result, in particular the rule works in 19 cases out of 20 trials, so we obtain 95% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rule, which was almost every time successful. The only error that we encountered was related to the vehicle's behavior after ending the turn. In fact, in just one case, both the vehicles correctly complete the hairpin turn but, in the end, they take another turn to the right, while the platoon leader is going straight. This error happens rarely in the development and test phases, and it is probably related to an error in steering data elaboration. When the rule is triggered, there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. In Figure 8.39, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

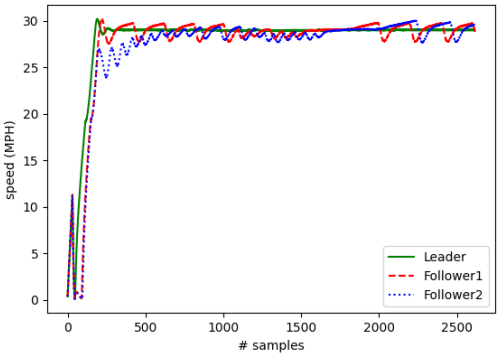
In Figure 8.39a, we report the speed graph of a platoon typical situation, when there are no




(a) Normal - Speed




(b) slowing_attack - Speed, no rules



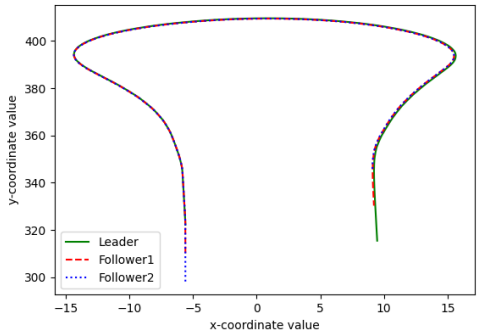

(c) slowing_attack - Speed, with rules

Figure 8.39: slowing_attack - Speed graph

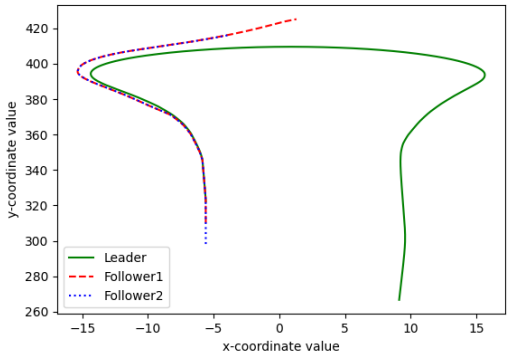
attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 30MPH.

In Figure 8.39b, we report the speed graph when the platoon is under the tested attack, without implementing any resilience policy. In this case, we could see that the followers reach a very lower speed (around 10MPH, as expected), while the leader graph goes normally, reaching the plateau speed. At the end of the graph, we could see that the follower speed lines drop to zero. This because the vehicles go out of the road (after losing the connection with the platoon leader) and are stopped by the LiDAR, to avoid a crash with some obstacles.

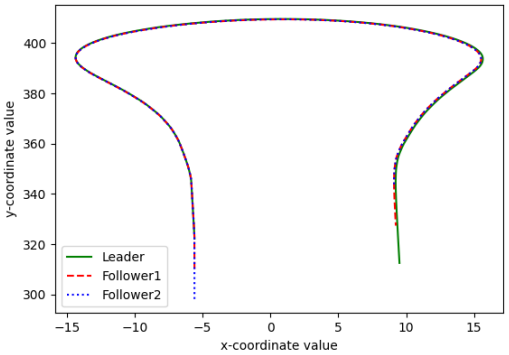
Finally, in Figure 8.39c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, this graph and the "normal" one are almost overlapping. They are very similar, all the three cars reach the plateau speed in the correct way without hesitations or strange values.




(a) Normal - Trajectory




(b) slowing_attack - Trajectory, no rules




(c) slowing_attack - Trajectory, with rules

Figure 8.40: slowing_attack - Trajectory graph

In Figure 8.40, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

In Figure 8.40a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

In Figure 8.40b, we present the trajectory graph of an attacked platoon, in particular under *speed_attack*, without implementing any resilience policy. Here, similarly to what happened with the *slowing_attack*, we could see that the trajectories of the leader and of the followers are different. In this case, both the followers (red and blue lines) take a wider turn, as highlighted by the lines, then they take a straight direction, going out of road. This lead to a possible crash, avoided by the intervention of LiDAR sensor, which blocks the cars right before crashing

with some obstacles.

Finally, in Figure 8.40c, we report the study case, where the platoon is under *speed_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph in Figure 8.40a. None of the followers takes strange trajectories (oppositely as in the previous case) and the trajectories distance is negligible.

So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are excellent.

### 8.6.3 opposite_turn_attack

#### Description

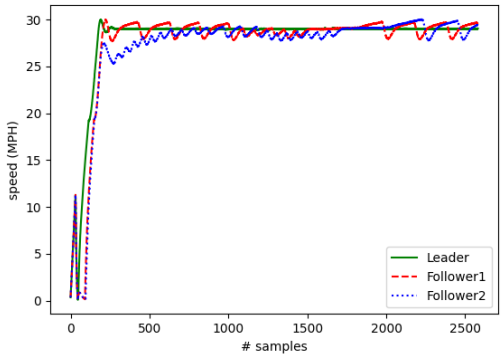
With this attack, we want to test both the *excessive_angles* policy, presented in Section 7.6.3, and the *opposite_turn* policy, presented in Section 7.6.4. Here, an attacker intercepts the data transmission and forge the `yaw(lead)` value, which represents the leader yaw value, used by the followers as target to compute their own yaw and steering values. In our case, we pass to the followers the value "150" as yaw reference, which is a value opposite to the ones obtained in the expected curve. We will pass this value when the vehicles are in the middle of the real turn, by using the leader coordinates. Therefore, by passing a forged value of 150 degrees, we will expect the follower vehicles to take an unexpected turn to the right, going out of road, if there are no rules to avoid this. In a real scenario, this could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulations) and it can cause severe crashes or dangerous situations.

```
#--Attack 3-opposite turn attack--
...
if self.speed>0.03:
    if self.x > 4 and self.x < 6:
        follower.add_waypoint([self.x, self.y, 150, self.steer, self.gyroscope[-1],
              self.accelerometer[0], self.accelerometer[1], self.accelerometer[2]])
        follower.set_speed_goal(self.speed)
    else:
        follower.add_waypoint([self.x, self.y, self.yaw, self.steer, self.gyroscope[-1],
                  self.accelerometer[0], self.accelerometer[1], self.accelerometer[2]])
        follower.set_speed_goal(self.speed)
```

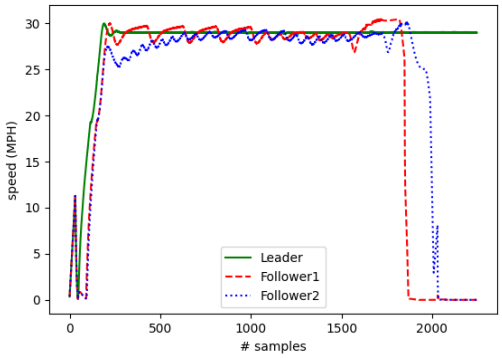
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like our target yaw, position, and steering values). In the second part of the code (lines 7 to 9), we report the usual scenario, such that the data passed is correct. This works if the x-coordinate of the leader is not included in ]4,6[, so when the leader is not turning. When we reach that target interval, we trigger the first part of the script (lines 4 to 6), where we pass to the function `follower.add_waypoint` our forged yaw value.

**Results**

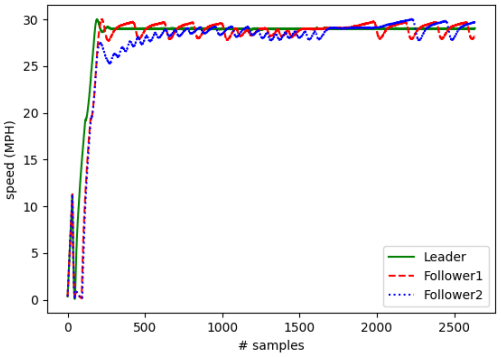
The results obtained by this combination of rules, in response to the presented attack, are great. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register a good result, in particular the rules work in 18 cases out of 20 trials, so we obtain 90% of success rate. This result is confirmed also by the hundred of trials repeated to refine the rules, which were in most of the cases successful. The errors that we encounter during the simulations are two and, similarly to previous attacks, they are mostly connected to the vehicle trajectory. In the first case, the error was related to just one follower vehicle. In fact, the first follower takes correctly the hairpin turn but, while passing the apex of the turn, it starts going wider and wider, until it goes out of the road. Then, the vehicle is stopped by LiDAR intervention. This can be caused by a wrong interpretation of steering data but, probably, it depends on the starting adjustment that the platoon made at the start of the scenario. In fact, we note that this error happens when the follower in question occupies the far right half of the lane. This is an adjustment that the simulator does autonomously, such that we can not interfere. The second error happened very few times during all the testing and developing phases. In this case, both the followers are fooled by the fake data, such that they overtake the turn apex correctly, but they do not end the turn in the right way. Instead, they take a right turn into the grass, going out of road. Otherwise, when the rules are triggered, there are no visible differences with the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. In Figure 8.41, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.




(a) Normal - Speed




(b) opposite_turn_attack - Speed, no rules



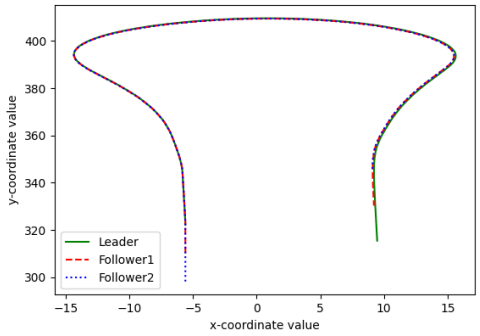

(c) opposite_turn_attack - Speed, with rules

Figure 8.41: opposite_turn_attack - Speed graph

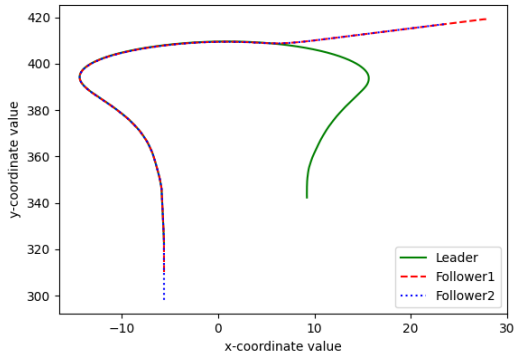
In Figure 8.41a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 30MPH.

In Figure 8.41b, we report the speed graph when the platoon is under the tested attack, without implementing any resilience policy. In this case, we could see that the followers have an acceleration equal to the leader. Then, at the end of the graph, the followers' speeds decrease until stop. This because the following vehicles take a wrong trajectory and go out of road, such that the LiDAR has to intervene, in order to avoid a crash.

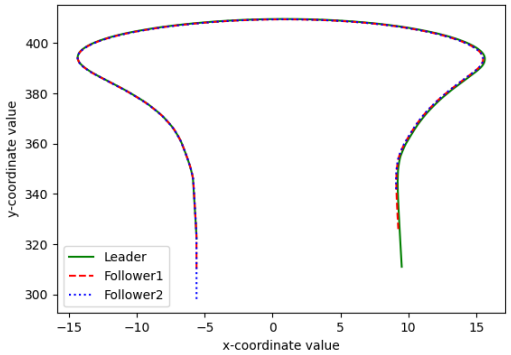
Finally, in Figure 8.41c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, this graph and the "normal" one are almost overlapping. They are very similar, all the three cars reach the plateau speed in the correct way without hesitations or strange values.




(a) Normal - Trajectory




(b) opposite_turn_attack - Trajectory, no rules




(c) opposite_turn_attack - Trajectory, with rules

Figure 8.42: opposite_turn_attack - Trajectory graph

In Figure 8.42, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

In Figure 8.42a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.

In Figure 8.42b, we present the trajectory graph of an attacked platoon, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of the followers are very different. As it can be clearly seen, the trajectories of the followers is extremely different, compared to the leader one. In fact, we can see that, in the middle of the turn, the two followers take a slight turn in the opposite direction compared to the leader. This because the attack is effective in that space zone, such that the following vehicles are fooled

into respecting the forged yaw value of 150, which implies an opposite turn.

Finally, in Figure 8.42c, we report the study case, where the platoon is under *no_steer_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph in Figure 8.42a. None of the followers takes strange trajectories (as in the previous case) and the trajectories distance is negligible.

So, considering both the graphs presented (speed and trajectory), we could say that these rules are correctly working, and the results obtained are great, considering the low error rate and the distribution of errors, as presented in the attack description.

### 8.6.4 no_steer_attack

#### Description

With this attack, we want to test both the *no_steer_input* and, again, the *excessive_angles* policies, presented in Section 7.6.5 and Section 7.6.3. Here, an attacker intercepts the data transmission and forge the `yaw(lead)` value, which represents the leader yaw value, that will be used by the follower as target to compute its own yaw and steering values. In our case, we pass to the followers the value "90" as yaw reference, that is the value that the yaw assumes when the car is going straight, before taking the turn. Therefore, by passing a forged value of 90, we will expect the follower vehicles to not take any turn, crashing into the guardrails, if there are no rules to avoid this. Then, even if the first *no_steer_input* works, we would have some problems while the followers' vehicles take the hairpin. In fact, with yaw fixed to 90, they will be forced in taking a fake turn to the right, opposite to the hairpin turn direction. For this reason, we will test also the *excessive_angles* policy. In a real scenario, this could lead to abnormal behaviors (i.e., problems in maintaining the trajectory, as tested during the simulations) and it can cause severe crashes or dangerous situations.

```
#--Attack 4-no steer input--
...
if self.speed>0.03:
        follower.add_waypoint([self.x, self.y, 90, self.steer, self.gyroscope[-1],
             self.accelerometer[0], self.accelerometer[1], self.accelerometer[2]])
        follower.set_speed_goal(self.speed)
```

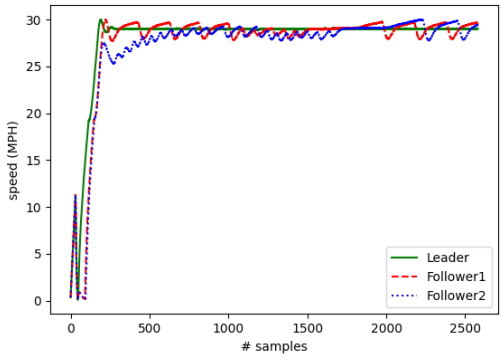
Here, it can be seen how the attack simulation works. In the platooning script, the leader passes its data through the function `follower.add_waypoint`, which will create a data array containing all the important values, like our target yaw, position, and steering values). The attacker triggers the script (line 4 to 6), where we pass to the function `follower.add_waypoint` our forged yaw value.

### Results

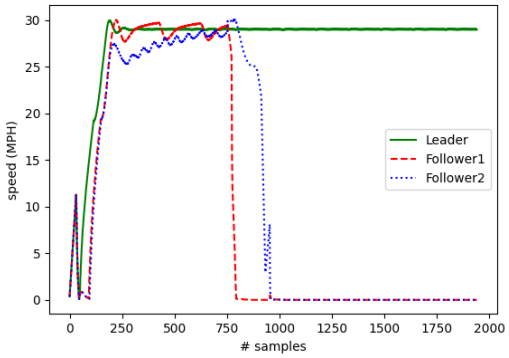
The results obtained by this combination of rules, in response to the presented attack, are great. First of all, we repeat this attack 20 times in a row, in order to test the success rate of rules. We register a very good result, in particular the rules work in 18 cases out of 20 trials, so we obtain 90% of success rate. This result is confirmed also by the hundreds of trials repeated to refine the rules, which were in most of the cases successful. We encounter just one error during the simulations, which was repeated twice in the 20 trials in a row. In this error, the platoon followers recognize the attack and start correctly the turn. When reaching the chord point of the turn, both the vehicles start turning mechanically, glitchy, and not smoothly as in the other cases. This led to difficulty in following the correct trajectory, with the following vehicles moving from lane to lane. This problem is probably related to the *excessive_angles* rule, more than the *no_steer_input* one, because it is a frenetic variation of steering angles, probably due to some data misinterpretation in the policy. When the rules are triggered, there are no visible differences from the "normal" scenario, both the distance and the speed of the vehicles are comparable to the ones registered in a classic situation. In Figure 8.43, we reported three graphs, highlighting three different situations. The green line represents the platoon leader's speed, while the red and blue ones represent, respectively, the first and the second platoon follower speed.

In Figure 8.43a, we report the speed graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that the speed of the leader and followers is almost the same in all the parts of the graph, until all the lines reach the plateau at approximately 30MPH.

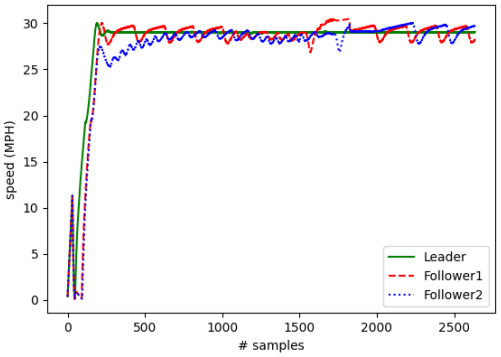
In Figure 8.43b, we report the speed graph when the platoon is under the tested attack, without implementing any resilience policy. In this case, we could see that the followers have an acceleration equal to the leader. Then, suddenly, the followers' speed decreases to 0MPH. This




(a) Normal - Speed




(b) no_steer_attack - Speed, no rules




(c) no_steer_attack - Speed, with rules

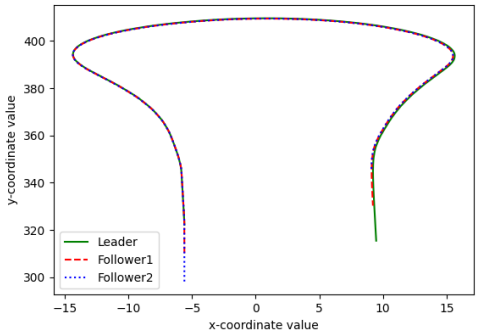
Figure 8.43: no_steer_attack - Speed graph

because the following vehicles take a wrong trajectory and go straight, such that the LiDAR has to intervene, in order to avoid a collision with the hairpin central guardrail.

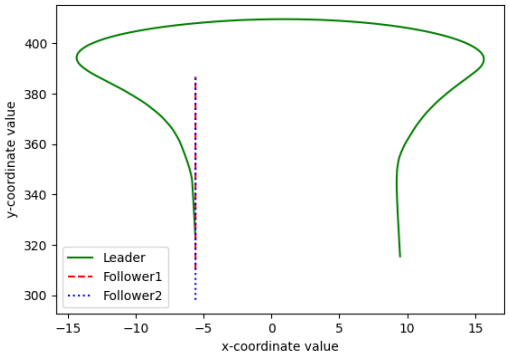
Finally, in Figure 8.43c, we report the interested scenario, where the platoon is under attack and the rule is triggered. In this case, this graph and the "normal" one are almost overlapping. They are very similar, all the three cars reach the plateau speed in the correct way without big hesitations or strange values.

In Figure 8.44, we reported the trajectory graphs, highlighting the same three different situations discussed before. Again, the green line represents the platoon leader trajectory, while the red and blue ones represent, respectively, the first and the second platoon follower trajectory.

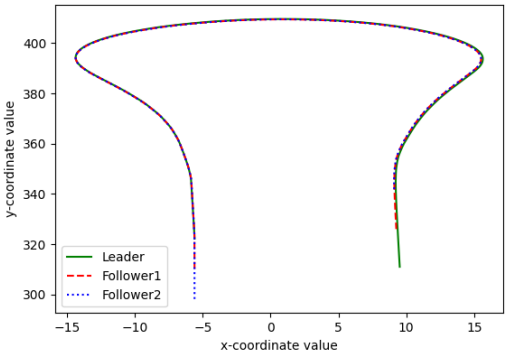
In Figure 8.44a, we report the trajectory graph of a platoon typical situation, when there are no attacks or abnormal situations. We could see that all the vehicles travel the same trajectory, as highlighted by the almost overlapping lines.




(a) Normal - Trajectory




(b) no_steer_attack - Trajectory, no rules




(c) no_steer_attack - Trajectory, with rules

Figure 8.44: no_steer_attack - Trajectory graph

In Figure 8.44b, we present the trajectory graph of an attacked platoon, without implementing any resilience policy. Here, we could see that the trajectories of the leader and of the followers are not overlapping. As it can be clearly seen, the trajectories of the followers are extremely different, compared to the leader one. In fact, both the followers take a completely straight trajectory, instead of correctly taking the turn.

Finally, in Figure 8.44c, we report the study case, where the platoon is under *no_steer_attack* and the rule is triggered. Here we could see that both the followers copy perfectly the leader trajectory, highlighted by the similarity with the "normal" graph in Figure 8.44a. None of the followers takes strange trajectories (as in the previous case) and the trajectories distance is negligible.

So, considering both the graphs presented (speed and trajectory), we could say that this rule is correctly working, and the results obtained are great, considering the low error rate and the distribution of errors, as presented in the attack description.

## 8.7 Results summary

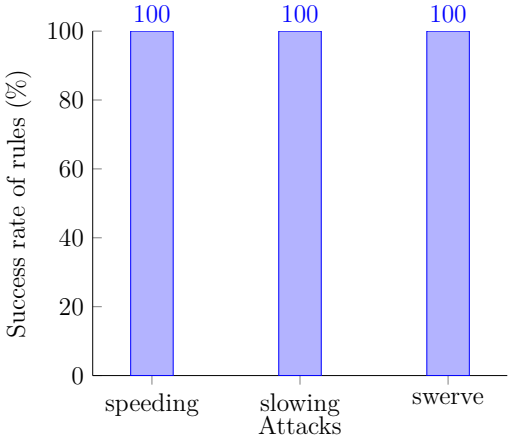
In this section, we report a summary of the results obtained with the rules, and discussed in the previous sections.

### 8.7.1 Success rate values

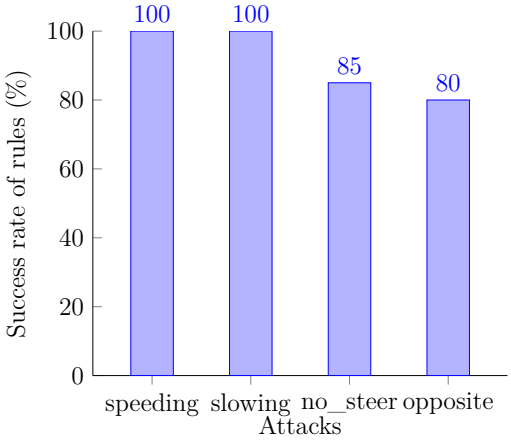
In this section, we report the success rate values, as visible in Table 8.6. Then, we report five different charts, in order to highlight better the results obtained, and also to make the reader able to see the differences between different attacks. In Figure 8.45, we report 5 distinct bar graphs, one for each scenario analyzed. For each graph, we report the type of attack on the abscissa axis, while on the ordinates the success rate of rules, expressed as a percentage of successes.

Table 8.6: Success rate of rules summary

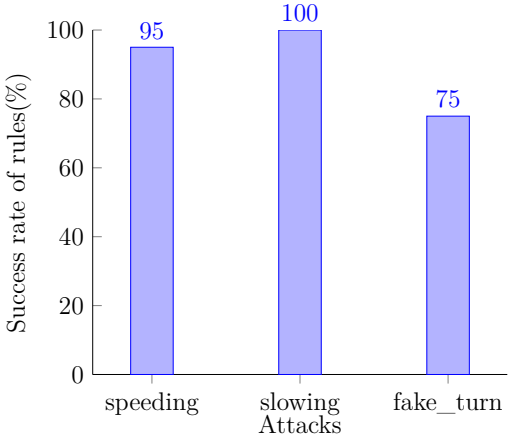
| **Scenario** | **Attack** | **Success rate** |
|---|---|---|
| Highway | speeding_attack | 100% |
| | slowing_attack | 100% |
| | swerve_attack | 100% |
| Turn | speeding_attack | 100% |
| | slowing_attack | 100% |
| | no_steer_attack | 85% |
| | opposite_turn_attack | 80% |
| Roundabout | speeding_attack | 95% |
| | slowing_attack | 100% |
| | fake_turn_attack | 75% |
| Crossroad | speeding_attack | 85% |
| | fake_turn_attack | 90% |
| | opposite_turn_attack | 90% |
| Hairpin | speeding_attack | 100% |
| | slowing_attack | 95% |
| | opposite_turn_attack | 90% |
| | no_steer_attack | 90% |



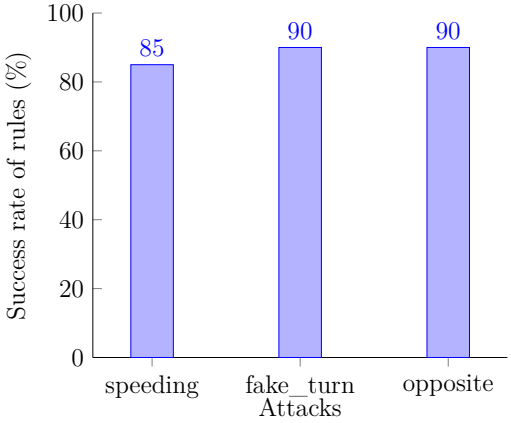

(a) Highway success rate of rules graph



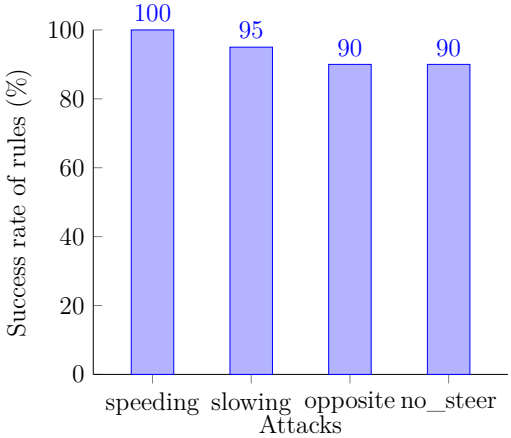

(b) Turn success rate of rules graph




(c) Roundabout success rate of rules graph




(d) Crossroad success rate of rules graph




(e) Hairpin success rate of rules graph

Figure 8.45: Success rate of rules graphs

### 8.7.2 Statistical Results

In this section, we present the point-to-point trajectory distance, calculated by using the vehicles' coordinates during the entire time frame of the development. The formula used is the canonical

$$\sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2]}$$

We perform this comparison between the normal scenario (no attack) and both the attack situations (with and without rules). We report the Cumulative Distribution Function (CDF) graph, which describes the probabilities of a random variable having values less than or equal to x, following the formula:

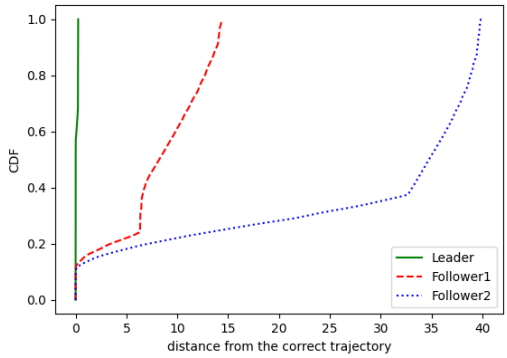
$$CDF(x) = P(X \leq x)$$

Therefore, we counts how many occurrences of values we have with value lower than x, then divide that number by the total of occurrences.

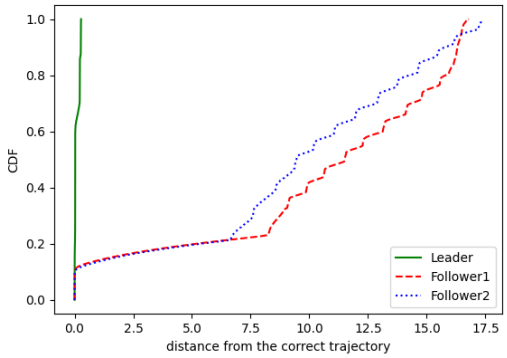
The goal of this study is to highlight the effect of the rules on the platoon trajectory, in order to underline the differences and the benefits.

#### Highway

Here we report the results obtained for the highway attacks discussed before. We start from the *speeding_attack*, presented in Figure 8.46. It is clearly visible the difference between graphs. Even if in Figure 8.46a, the trajectory difference is higher, in the tested scenario the vehicles reach the peak of distance (40) at the start, where the followers recognize the attack. Then, they adapt to the leader behaviour and the platoon distance decrease, without any problem arising. Instead, in Figure 8.46b, the trajectory distance is lower and it is increasing with increasing time, but it is ruled by the LiDAR intervention. This means that the vehicles are not under control, and any change in leader behavior can cause an abnormal situation for the followers.



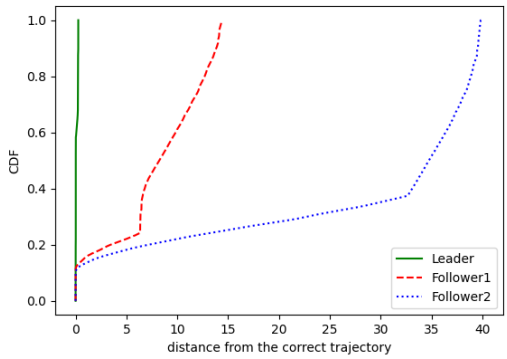

(a) Trajectory difference - speeding_attack with rules



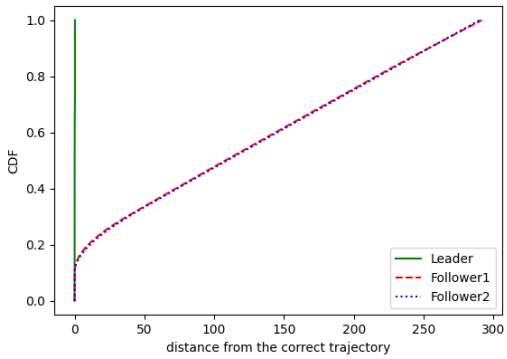

(b) Trajectory difference - speeding_attack no rules

Figure 8.46: Trajectory difference - speeding_attack

Then, we have *slowing_attack*, presented in Figure 8.47. Here the behavior is very similar to the one presented in the previous attack. The main difference is that this time, in Figure 8.47b, the LiDAR could not intervene, as the vehicles are going very slow. This means that the trajectory difference is very high, and it increases with increasing time. Otherwise, in Figure 8.47a, the behavior is almost identical to the one presented in *speeding_attack*.



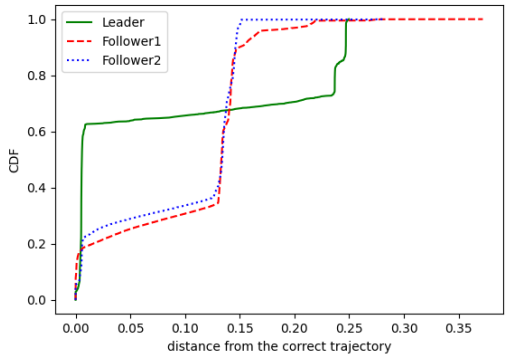

(a) Trajectory difference - slowing_attack with rules



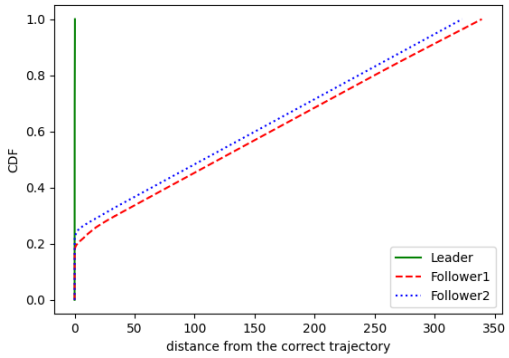

(b) Trajectory difference - slowing_attack no rules

Figure 8.47: Trajectory difference - slowing_attack

Finally, we have *swerve_attack*, presented in Figure 8.48. Here, the difference between the graphs is very noticeable. In Figure 8.48a, we can see that the rules work very well, as the trajectory difference is minimal (<0.4). Otherwise, in Figure 8.48b, the trajectory difference lines are similar to the previous cases, they arise with increasing time, reaching very high values.




(a) Trajectory difference - swerve_attack with rules

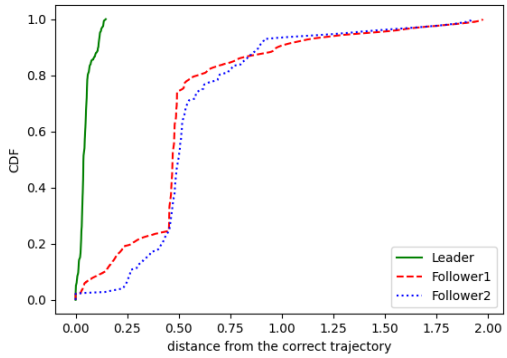



(b) Trajectory difference - swerve_attack no rules

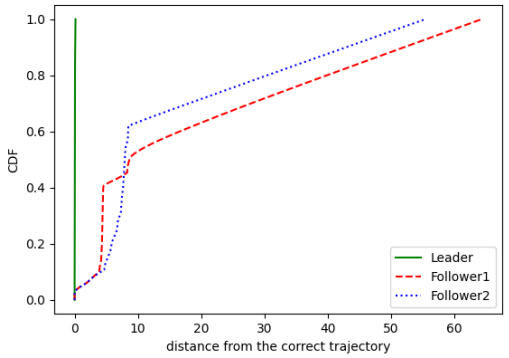
Figure 8.48: Trajectory difference - swerve_attack

### Turn

Here we report the results obtained for the turn attacks discussed before. We start from the *speeding_attack*, presented in Figure 8.49. It is clearly visible the difference between graphs. In Figure 8.49a, the trajectory difference is lower (below 2.00), the lines start increasing at the start of the graph, when the vehicles recognize the attack and have to settle themselves. Instead, in Figure 8.49b, the trajectory distance is higher, and it is increasing with increasing time. This is probably due to a crash (caused by the high speed) or by a disconnection from the platoon.




(a) Trajectory difference - speeding_attack with rules

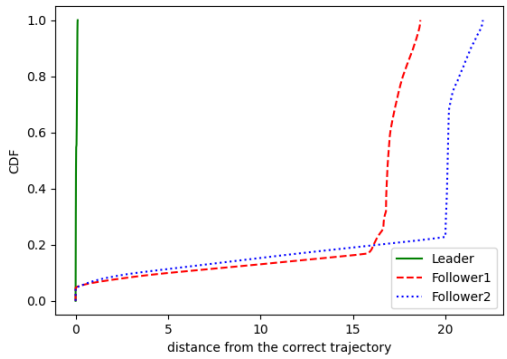



(b) Trajectory difference - speeding_attack no rules

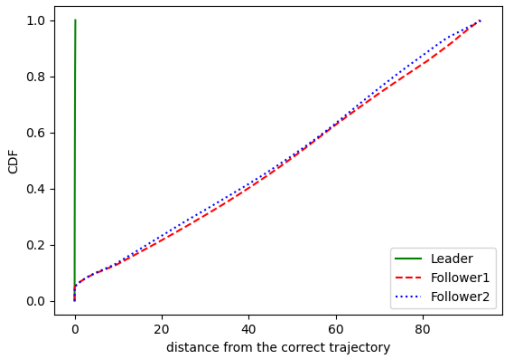
Figure 8.49: Trajectory difference - speeding_attack

Then, we have *slowing_attack*, presented in Figure 8.50. Here, we could see that in Figure 8.50a, we reach a peak around 20, then the following vehicles start adapting to the leader

behaviour. This because, at the start, the vehicles recognize the attack, and have to settle themselves in order to avoid following the forged passed data (in this case 0MPH, which means not moving). This means that the following vehicles start with a delay, compared to the leader, but they solve this problem over time. Instead, in Figure 8.50b, the vehicles start delayed (as before), but they are not able to reach the leader behavior, as they use the forged data (such that their computed speed is very low, as reported in the attack analysis). This means that the trajectory distance will arise over time, until the vehicles disconnect from the platoon.




(a) Trajectory difference - slowing_attack with rules

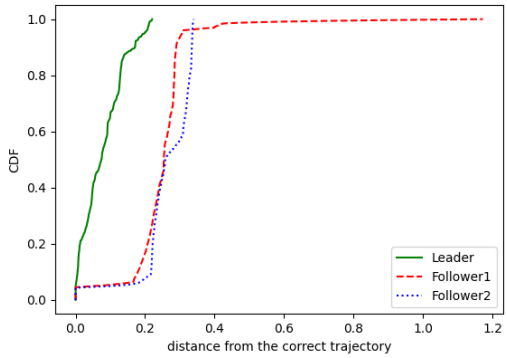



(b) Trajectory difference - slowing_attack no rules

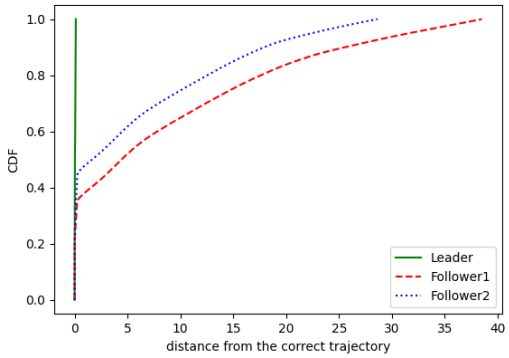
Figure 8.50: Trajectory difference - slowing_attack

Then, we have *no_steer_attack*, presented in Figure 8.51. Here, we could see that the rules work very well. In fact, in Figure 8.51a, the trajectory distance is very low, reaching a maximum peak of around 1.2. Otherwise, in Figure 8.51b, the difference arises over time, meaning that the following vehicles are not able to follow the leader trajectory. In fact, they use the forged yaw (fixed to 180), that forces the vehicles in going straight, out of road.

Finally, we have *opposite_turn_attack*, presented in Figure 8.52. Here, the behavior is very similar to the one described previously in *no_steer_attack*. In fact, in Figure 8.52a, the distance is negligible, reaching a maximum peak of 0.8, while in Figure 8.52b the distance is higher, arising over time. The reasons are the same as before. The following vehicles use the forged yaw data, which forces them to take a fake turn, leading them out of road.

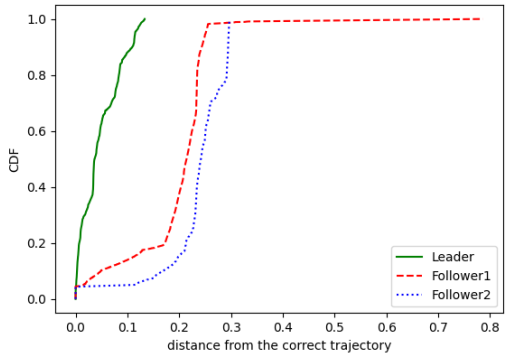



(a) Trajectory difference - no_steer_attack with rules

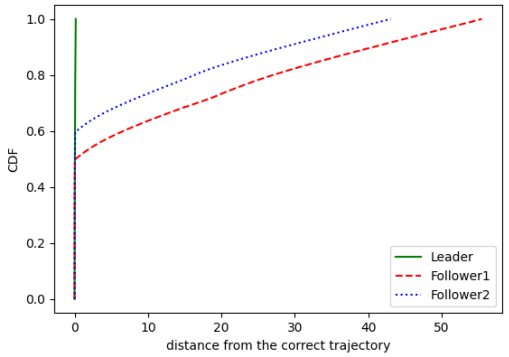



(b) Trajectory difference - no_steer_attack no rules

Figure 8.51: Trajectory difference - no_steer_attack




(a) Trajectory difference - opposite_turn_attack with rules

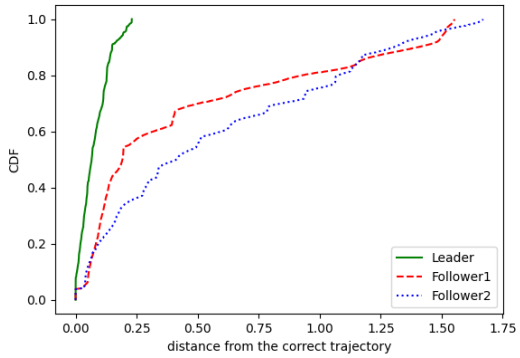



(b) Trajectory difference - opposite_turn_attack no rules

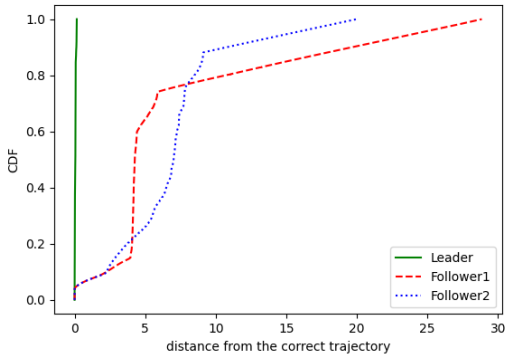
Figure 8.52: Trajectory difference - opposite_turn_attack

### Roundabout

Here we report the results obtained for the turn attacks discussed before. We start from the *speeding_attack*, presented in Figure 8.53 (for the first exit) and Figure 8.54 (for the third one). It is clearly visible the difference between graphs. In both cases, Figure 8.53a and Figure 8.54a, the trajectory difference is lower (below 2.00). Instead, in Figures 8.53b and 8.54b, the trajectory distance is higher, and it is increasing with increasing time. Both the graphs are similar in trend, they have an almost "stationary" phase, then the lines arise to higher values. This is probably due to a crash (caused by the high speed) or by a disconnection from the platoon.

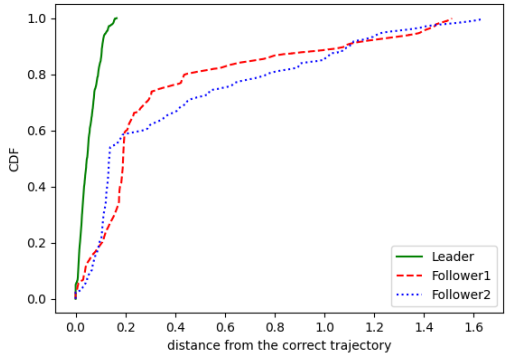



(a) Trajectory difference - speeding_attack with rules - 1st exit

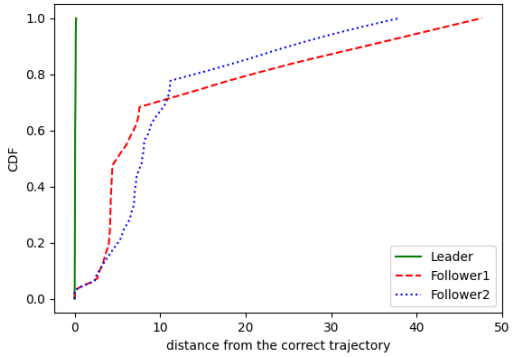



(b) Trajectory difference - speeding_attack no rules - 1st exit

Figure 8.53: Trajectory difference - speeding_attack - 1st exit



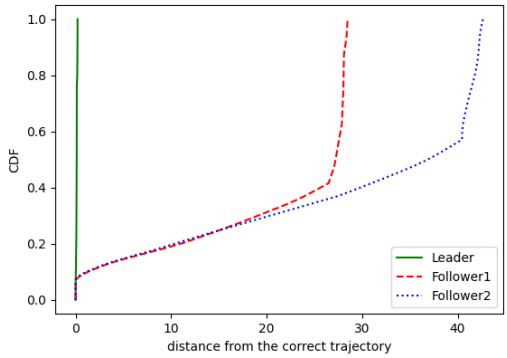

(a) Trajectory difference - speeding_attack with rules - 3rd exit



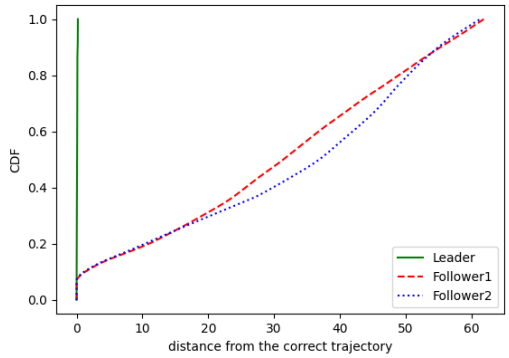

(b) Trajectory difference - speeding_attack no rules - 3rd exit

Figure 8.54: Trajectory difference - speeding_attack - 3rd exit

Then, we have *slowing_attack*, presented in Figure 8.55 (for the first exit) and Figure 8.56 (for the third one). Here, we could see that in both Figures (8.55a and 8.56a), we have a peak around 40, where the vehicles hit the maximum distance. This because, at the start, the vehicles recognize the attack, and have to settle themselves in order to avoid following the forged passed data (in this case 0MPH, which means not moving). This means that the following vehicles start with a delay, compared to the leader, but they solve this problem over time. Instead, in Figures 8.55b and 8.56b, the vehicles start delayed (as before), but they are not able to reach the leader behavior, as they use the forged data (such that their computed speed is very low, as reported in the attack analysis). This means that the trajectory distance will arise over time, until the vehicles disconnect from the platoon.

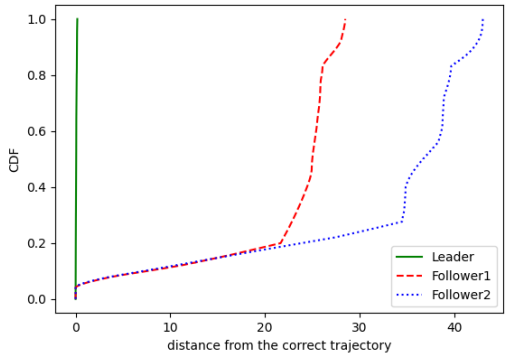



(a) Trajectory difference - slowing_attack with rules - 1st exit

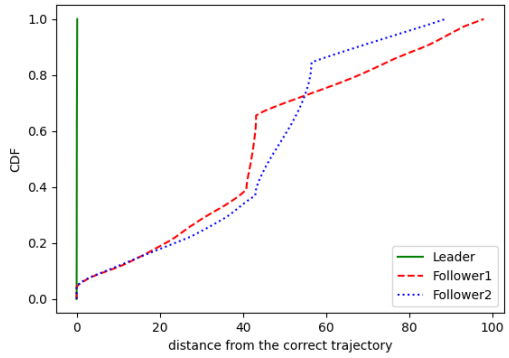



(b) Trajectory difference - slowing_attack no rules - 1st exit

Figure 8.55: Trajectory difference - slowing_attack - 1st exit



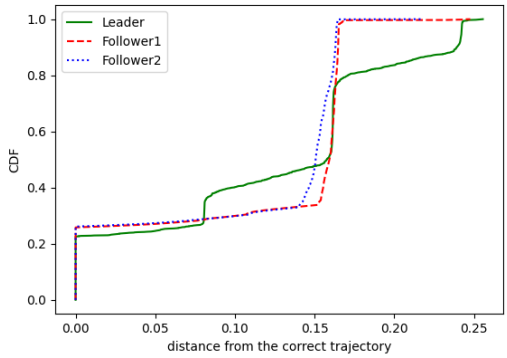

(a) Trajectory difference - slowing_attack with rules - 3rd exit



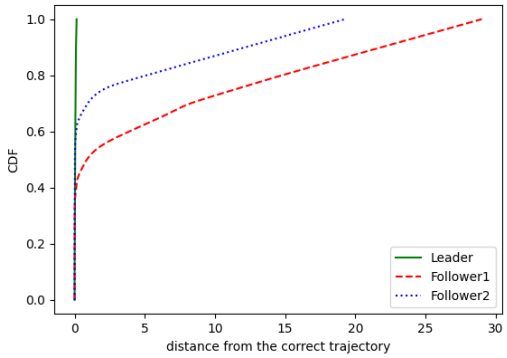

(b) Trajectory difference - slowing_attack no rules - 3rd exit

Figure 8.56: Trajectory difference - slowing_attack - 3rd exit

Finally, we have *fake_turn_attack*, presented in Figure 8.57 (for the first exit) and Figure 8.58 (for the third one). As highlighted before, the results for the first and third exits are similar. In fact, in both Figure 8.57a and 8.58a the distance is very low, in particular in the first one, when is limited to 0.25. In the second graph, we reach a peak of 1.2 (a very good result), a bit higher than the other trial probably due to the higher complexity of the scenario. Otherwise, in both Figure 8.57b and 8.58b, the trend is opposite. The lines arise over time, which means that the trajectory difference increases as the leader moves, until the followers crash or disconnect from the platoon.

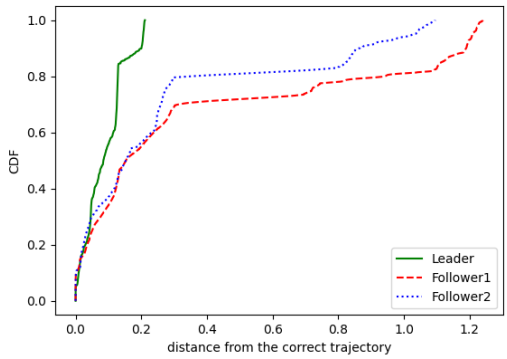



(a) Trajectory difference - fake_turn_attack with rules - 1st exit

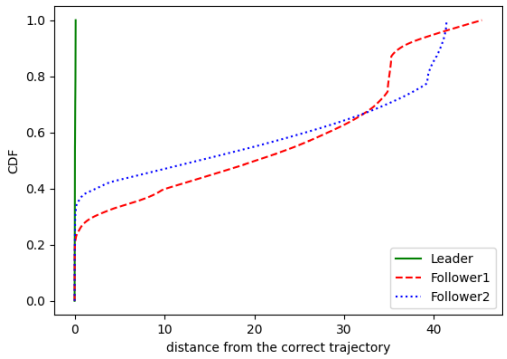



(b) Trajectory difference - fake_turn_attack no rules - 1st exit

Figure 8.57: Trajectory difference - fake_turn_attack - 1st exit




(a) Trajectory difference - fake_turn_attack with rules - 3rd exit



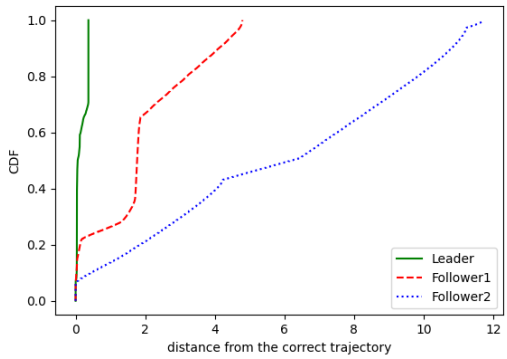

(b) Trajectory difference - fake_turn_attack no rules - 3rd exit

Figure 8.58: Trajectory difference - fake_turn_attack - 3rd exit

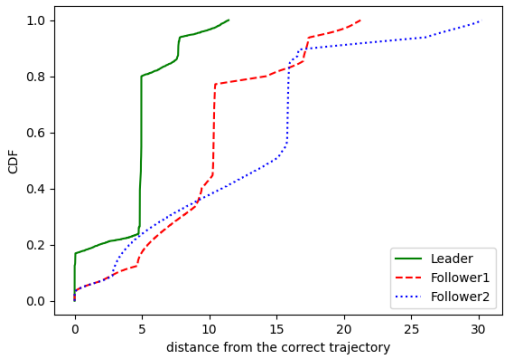
### Crossroad

Here we report the results obtained for the turn attacks discussed before. We start from the *speeding_attack*, presented in Figure 8.59 (for the straight direction) and Figure 8.60 (for the right turn). It is clearly visible the difference between graphs. In both cases, Figure 8.59a and Figure 8.60a, the trajectory difference is lower (below 12.00). The followers' vehicles also restart from the stop line with a little delay, compared to the leader, visible by the second lines' peaks, around 12 for the second follower. Instead, in Figures 8.59b and 8.60b, the trajectory distance is higher, and it is increasing with increasing time. Both the graphs are similar in trend, they have an almost "stationary" phase, then the lines decrease, just before arise to higher values.

In this case, we have a noticeable difference also in leader's trajectory. This is probably due to a crash (caused by the high speed) of the vehicles, that involved also the leader.

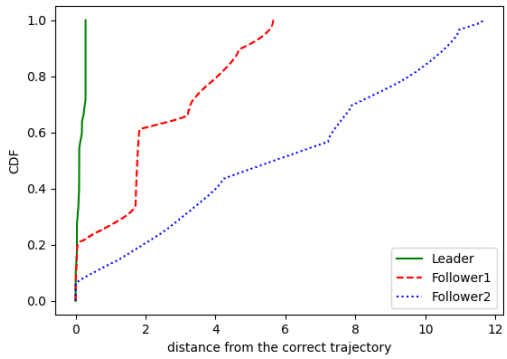



(a) Trajectory difference - speeding_attack with rules - straight

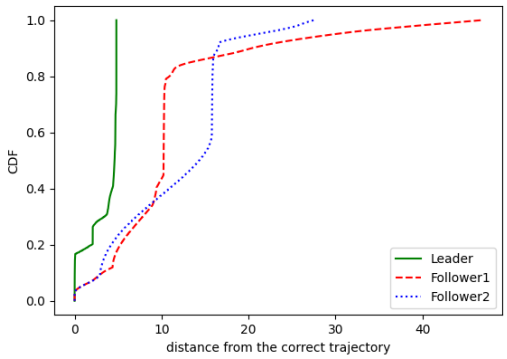



(b) Trajectory difference - speeding_attack no rules - straight

Figure 8.59: Trajectory difference - speeding_attack - straight



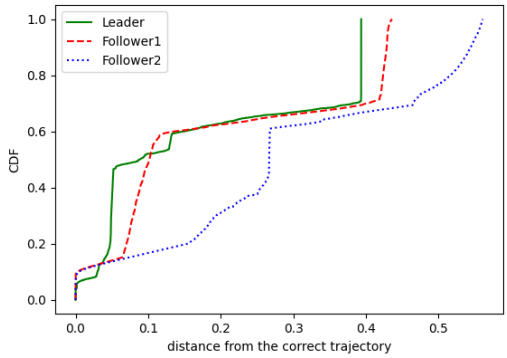

(a) Trajectory difference - speeding_attack with rules - right



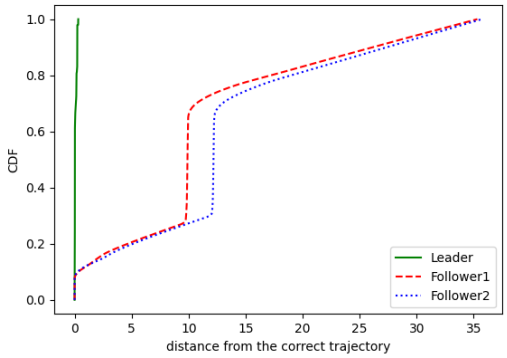

(b) Trajectory difference - speeding_attack no rules - right

Figure 8.60: Trajectory difference - speeding_attack - right

Then, we have *fake_turn_attack*, presented in Figure 8.61. Here, we retrieve a huge difference between graphs. In fact, in Figure 8.61a, the distance is negligible, reaching a maximum peak of 0.6, while in Figure 8.61b the distance is higher, arising over time. The reason is that the following vehicles use the forged yaw data, which forces them to take a fake turn, leading them out of road, while the leader is going straight.



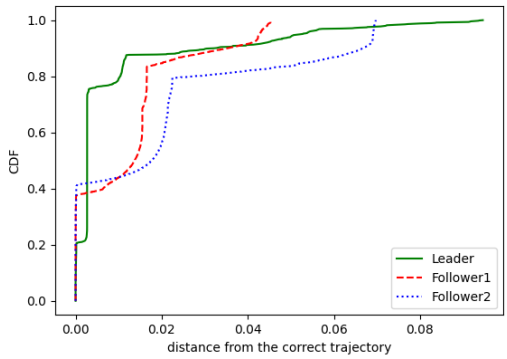

(a) Trajectory difference - fake_turn_attack with rules



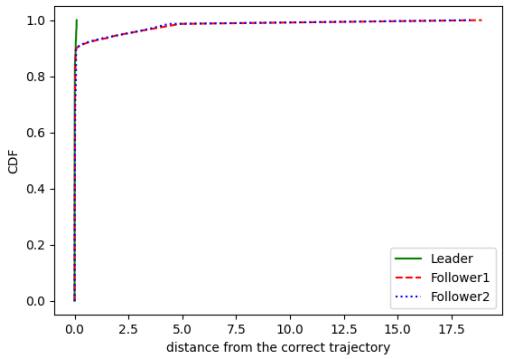

(b) Trajectory difference - fake_turn_attack no rules

Figure 8.61: Trajectory difference - fake_turn_attack

Finally, we have *opposite_turn_attack*, presented in Figure 8.62 (for the left turn) and Figure 8.63 (for the right turn). In this case, we obtain very good results for both the ruled scenarios. In fact, in both Figure 8.62a and 8.63a the distance is very low, in the first one is below 0.1, while in the second graph we reach a maximum peak of 0.175. Otherwise, in both Figure 8.62b and 8.63b, the trend is opposite. The difference arise over time, which means that the trajectory distance increases as the leader moves, until the followers crash or disconnect from the platoon.

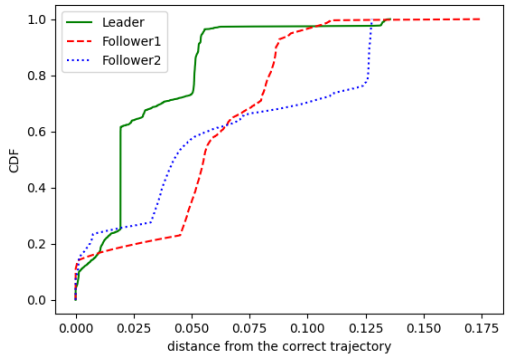



(a) Trajectory difference - opposite_turn_attack with rules - left turn

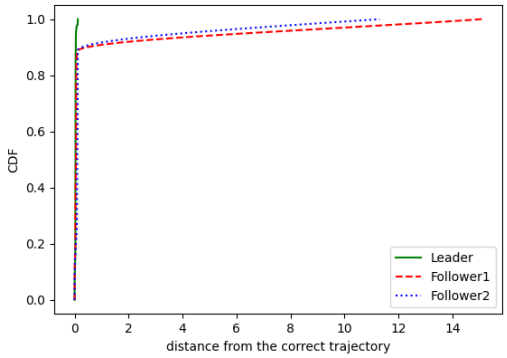



(b) Trajectory difference - opposite_turn_attack no rules - left turn

Figure 8.62: Trajectory difference - opposite_turn_attack - left turn




(a) Trajectory difference - opposite_turn_attack with rules - right turn

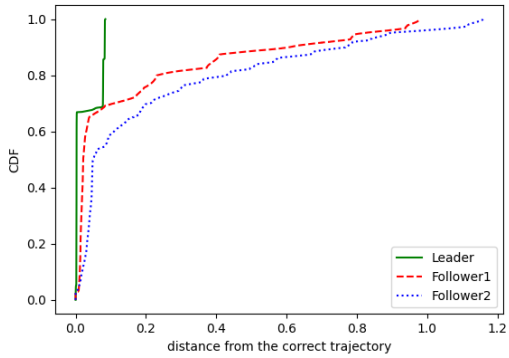



(b) Trajectory difference - opposite_turn_attack no rules - right turn

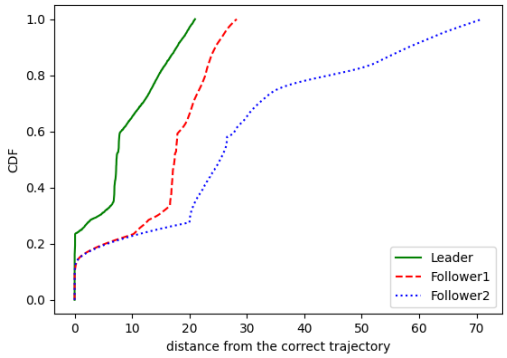
Figure 8.63: Trajectory difference - opposite_turn_attack - right turn

### Hairpin

Here we report the results obtained for the turn attacks discussed before. We start from the *speeding_attack*, presented in Figure 8.64. It is clearly visible the difference between graphs. In Figure 8.64a, the trajectory difference is lower (below 1.2). Instead, in Figure 8.64b, the trajectory distance is higher, and it is increasing with increasing time. In this particular case, also the leader line is increasing, which means that also the leader trajectory is varying. This is due to a crash (caused by the high speed), as described in the attack study case.




(a) Trajectory difference - speeding_attack with rules

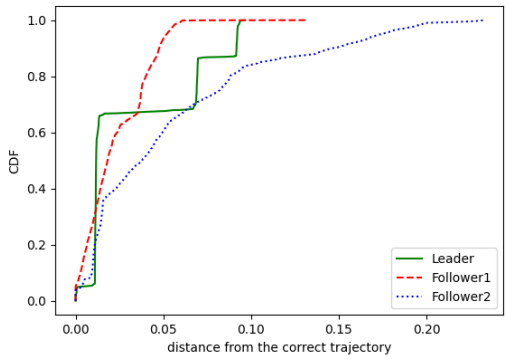



(b) Trajectory difference - speeding_attack no rules

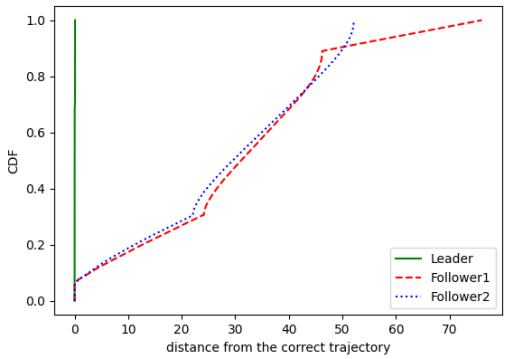
Figure 8.64: Trajectory difference - speeding_attack

Then, we have *slowing_attack*, presented in Figure 8.65. Here, we could see that in Figure 8.65a, we have a maximum peak around 0.2, not noticeable in the simulations. Instead,

in Figure 8.65b, the vehicles start delayed, but they are not able to reach the leader behavior, as they use the forged data (such that their computed speed is very low, as reported in the attack analysis). This means that the trajectory distance will arise over time, until the vehicles disconnect from the platoon.



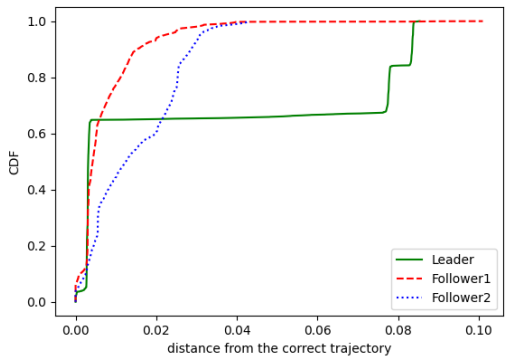

(a) Trajectory difference - slowing_attack with rules



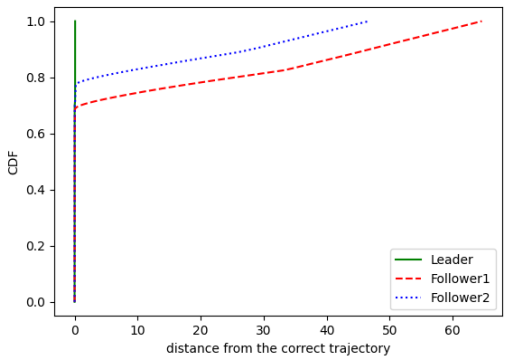

(b) Trajectory difference - slowing_attack no rules

Figure 8.65: Trajectory difference - slowing_attack

Then, we have *opposite_turn_attack*, presented in Figure 8.66. Here, we could see that the rules work very well. In fact, in Figure 8.66a, the trajectory distance is very low, reaching a maximum peak of around 0.1. Otherwise, in Figure 8.66b, the difference arises over time, meaning that the following vehicles are not able to follow the leader trajectory. In fact, they use the forged yaw (fixed to 180), that forces the vehicles in going straight, out of road.




(a) Trajectory difference - opposite_turn_attack with rules

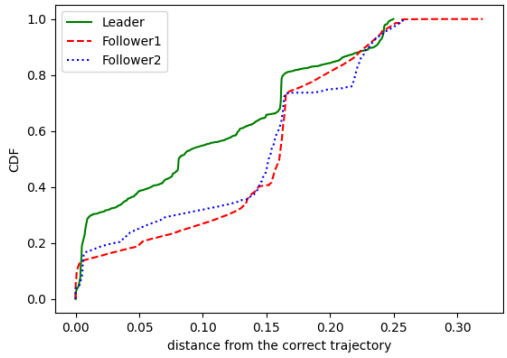



(b) Trajectory difference - opposite_turn_attack no rules

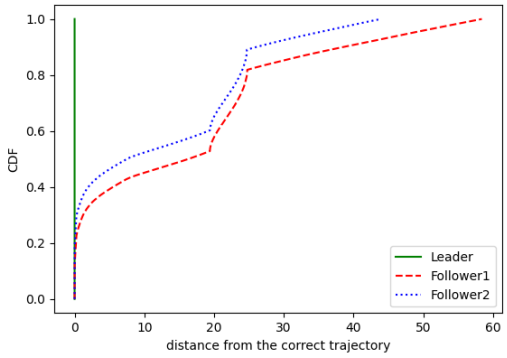
Figure 8.66: Trajectory difference - opposite_turn_attack

Finally, we have *no_steer_attack*, presented in Figure 8.67. Here, the behavior is very

similar to the one described previously in opposite_turn_attack. In fact, in Figure 8.67a, the distance is negligible, reaching a maximum peak of 0.3, while in Figure 8.67b the distance is higher, arising over time. The reasons are the same as before. The following vehicles use the forged yaw data, which forces them to take a fake turn, leading them out of road.




(a) Trajectory difference - no_steer_attack with rules




(b) Trajectory difference - no_steer_attack no rules

Figure 8.67: Trajectory difference - no_steer_attack

## 8.8 Conclusions

The goal of this work is to find rules or policies, such that a platooning model can be safely implemented. In addition to this, these policies must also be able to implement a resilience mechanism, through which the member(s) of the platoon under attack are able to not only detect the malfunction, but also to fix it and allow the entire platoon to return fully operational without the intervention of a further external entity. In our ideal, all this must take place with the use of "low-cost" technologies, i.e., not expensive (in terms of resources and economically), in order to develop a sustainable model, without using complex sensor fusion, machine learning, and similar mechanisms. To this end, we have selected five of the most frequent scenarios in everyday driving, we have reasoned on the possible vulnerabilities and, having established the cornerstones of the project, we have begun to define the aforementioned rules. Once perfected and refined, it was time to move on to the testing phase. To do this, we have implemented several attacks, each aimed at triggering one or more rules, and we have reported the results obtained, seasoned with a short discussion. To date, platoon techniques are still in full development, aiming for an increasingly green and economical automotive world. There

are still many unknowns and problems to be solved, before being able to develop an efficient technology to offer to the general public, first of all, the security issue, dealt with in this thesis work, which we hope will be a valid alternative among the various currently available proposals.

# Appendix A

# Code Snippets

Full code available on Github repository: `https://github.com/lorentz9819/Attack-Detection-and-Resiliency-Policies-for-AV-Platoons`

## A.1 platooning.py

```
import carla
import math
import numpy as np
from cloud import SafeCloud

def sign(number):
  if number>=0: return 1
  else: return -1

class PlatoonMember:
    def __init__(self, vehicle: carla.Vehicle):
        self.vehicle = vehicle
        tr = self.vehicle.get_transform()
        con = self.vehicle.get_control()
        vel = self.vehicle.get_velocity()
        self.x = tr.location.x
        self.y = tr.location.y
        self.z = tr.location.z
        self.yaw = tr.rotation.yaw
        self.speed = 3.6*math.sqrt(vel.x**2 + vel.y**2 + vel.z**2)
        self.throttle = con.throttle
        self.steer = con.steer
        self.brake = con.brake
```

```
24          self.cloud: SafeCloud = None
25          self.waypoints = []
26
27      def update_position(self):
28          tr = self.vehicle.get_transform()
29          con = self.vehicle.get_control()
30          vel = self.vehicle.get_velocity()
31          self.x = tr.location.x
32          self.y = tr.location.y
33          self.z = tr.location.z
34          self.yaw = tr.rotation.yaw
35          self.speed = 3.6*math.sqrt(vel.x**2 + vel.y**2 + vel.z**2)
36          self.throttle = con.throttle
37          self.steer = con.steer
38          self.brake = con.brake
39
40      def get_pos(self):
41          return self.x, self.y
42
43      def get_speed(self):
44          return self.speed
45
46  class Follower(PlatoonMember):
47      def __init__(self, vehicle:carla.Vehicle):
48          super().__init__(vehicle)
49          self.speedGoal = 0.0
50          self.last_t = 0.0
51          self.big_dist = False
52          self.leader_dist = 0.0
53          self.safe_dist = 12.0
54          self.override_brake = False
55          self.leader: PlatoonMember = None
56
57      def set_leading_vehicle(self, lead:PlatoonMember):
58          if not self.leader:
59              self.leader = lead
60
61      def update_position(self):
62          super().update_position()
63          x = self.x
64          y = self.y
65          tx,ty = self.leader.get_pos()
66          self.leader_dist = ((x-tx)**2 + (y-ty)**2)**(1/2)
67          self.safe_dist = max(12.0, self.speed/2)
68          if self.leader_dist > self.safe_dist: self.big_dist = True
69          else: self.big_dist = False
70
```

```
71      def connect_to_cloud(self, sc: SafeCloud):
72          if not self.cloud:
73              self.cloud = sc
74          sc.add_members(self)
75
76      def check_lidar(self,points):
77          detection=False
78          blinded=False
79          danger_dist = self.speed/5
80          for p in points:
81              if p.point.x<max(3,danger_dist):
82                  if p.point.x < 0.5:
83                      blinded = True
84                      detection = False
85                      print("Points = ", p.point.x)
86                  else:
87                      detection=True
88                      print("LiDAR triggered")
89          self.override_brake=detection
90          self.blinded=blinded
91
92      def IMU_callback(self, sensor_data):
93          limits = (-99.9, 99.9)
94          self.accelerometer = (
95              max(limits[0], min(limits[1], sensor_data.accelerometer.x)),
96              max(limits[0], min(limits[1], sensor_data.accelerometer.y)),
97              max(limits[0], min(limits[1], sensor_data.accelerometer.z)))
98          self.gyroscope = (
99              max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.x))),
100             max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.y))),
101             max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.z))))
102         self.compass = math.degrees(sensor_data.compass)
103
104     def add_waypoint(self, wp):
105         self.waypoints.append(wp)
106
107     def set_speed_goal(self, s):
108         self.speedGoal = s
109
110     def define_throttle(self, sg, ss, wp):
111         delta = sg - ss
112         t = b = 0.0
113         if self.speedGoal>0.04:
114             if (delta>0):
115                 boost = 0.4 if self.big_dist else 0.0
116                 t = (delta/10 + boost) if (delta/10 + boost) <= 1.0 else 1.0
117             else:
```

```
                t = self.last_t
                if not self.big_dist:
                    t = 0.0
                    b = -delta/10 if -delta/10 <= 1.0 else 1.0
        else:
            if self.big_dist:
                t = 0.2
            else:
                b = 0.8
        self.last_t=t
        return t, b

    def define_steer(self, wp, x, y, ya):
        if self.speed<=0.03: return 0.0
        rl = wp
        fx = x
        fy = y
        rel_yaw = yaw = ya
        s=0.0
        delta = delta_x = delta_y = 1000
        rel_x = rel_y = 0
        for row in reversed(rl):
            lx = row[0]
            ly = row[1]
            lyaw = row[2]
            gyro = row[5]
            accx = row[4]
            accy = row[6]
            accz = row[7]
            toll = (0.1**2+0.1**2)**(1/2)
            diff = ((fx-lx)**2 + (fy-ly)**2)**(1/2)
            if diff <= toll:
                s=row[3]
                if lyaw<-90 and yaw>90:
                    corr=(lyaw-yaw+360)/60
                elif lyaw>90 and yaw<-90:
                    corr=(lyaw-yaw-360)/60
                else:
                    corr=(lyaw-yaw)/60
                if corr > 1.0: corr = 1.0
                elif corr < -1.0: corr = -1.0
                s+=corr
                if s>1.0:
                    s=1.0
                elif s<-1.0:
                    s=-1.0
                return s
```

```
165             else:
166                 if diff < delta:
167                     rel_y = ly
168                     rel_x = lx
169                     delta = diff
170                     rel_yaw = lyaw
171                     if abs(lx-fx) < delta_x:
172                         delta_x = abs(lx-fx)
173                     if abs(ly-fy) < delta_y:
174                         delta_y = abs(ly-fy)
175         lyaw = rel_yaw
176         corr = (lyaw-yaw)/90
177         if abs(lyaw)<=5: #going east
178             s=(rel_y-fy)/10
179             if s>1.0: s=1.0
180             elif s<-1.0: s=-1.0
181             if corr>1.0: corr = 1.0
182             elif corr<-1.0: corr = -1.0
183             s+=corr
184         elif abs(lyaw)>=175: #going ovest
185             if lyaw>90 and yaw<-90:
186                 corr-=4
187                 s=(fy-rel_y)/10
188                 if s>1.0: s=1.0
189                 elif s<-1.0: s=-1.0
190                 if corr>1.0: corr = 1.0
191                 elif corr<-1.0: corr = -1.0
192                 s+=corr
193             elif lyaw<-90 and yaw>90:
194                 corr+=4
195                 s=(fy-rel_y)/10
196                 if s>1.0: s=1.0
197                 elif s<-1.0: s=-1.0
198                 if corr>1.0: corr = 1.0
199                 elif corr<-1.0: corr = -1.0
200                 s+=corr
201             else:
202                 s=(fy-rel_y)/10
203                 if s>1.0: s=1.0
204                 elif s<-1.0: s=-1.0
205                 if corr>1.0: corr = 1.0
206                 elif corr<-1.0: corr = -1.0
207                 s+=corr
208         elif lyaw<95 and lyaw>85: #going south
209             s=(fx-rel_x)/10
210             if s>1.0: s=1.0
211             elif s<-1.0: s=-1.0
```

```
            if corr>1.0: corr = 1.0
            elif corr<-1.0: corr = -1.0
            s+=corr
        elif lyaw>-95 and lyaw<-85: #going north
            s=(rel_x-fx)/10
            if s>1.0: s=1.0
            elif s<-1.0: s=-1.0
            if corr>1.0: corr = 1.0
            elif corr<-1.0: corr = -1.0
            s+=corr
        else:
            if yaw>-180 and yaw<=-90:
                if sign(rel_x)==sign(fx): fx=-fx
                xs = (rel_x-fx)/20
                if xs > 1.0: xs = 1.0
                elif xs < -1.0: xs = -1.0
                if sign(rel_y)==sign(fy): rel_y=-rel_y
                ys = (fy-rel_y)/20
                if ys > 1.0: ys = 1.0
                elif ys < -1.0: ys = -1.0
            elif yaw>-90 and yaw<=0:
                if sign(rel_x)==sign(fx): fx=-fx
                xs = (rel_x-fx)/20
                if xs > 1.0: xs = 1.0
                elif xs < -1.0: xs = -1.0
                if sign(rel_y)==sign(fy): fy=-fy
                ys = (rel_y-fy)/20
                if ys > 1.0: ys = 1.0
                elif ys < -1.0: ys = -1.0
            elif yaw>0 and yaw<=90:
                if sign(rel_x)==sign(fx): rel_x=-rel_x
                xs = (fx-rel_x)/20
                if xs > 1.0: xs = 1.0
                elif xs < -1.0: xs = -1.0
                if sign(rel_y)==sign(fy): fy=-fy
                ys = (rel_y-fy)/20
                if ys > 1.0: ys = 1.0
                elif ys < -1.0: ys = -1.0
            elif yaw>90 and yaw<=180:
                if sign(rel_x)==sign(fx): rel_x=-rel_x
                xs = (fx-rel_x)/20
                if xs > 1.0: xs = 1.0
                elif xs < -1.0: xs = -1.0
                if sign(rel_y)==sign(fy): rel_y=-rel_y
                ys = (fy-rel_y)/20
                if ys > 1.0: ys = 1.0
                elif ys < -1.0: ys = -1.0
```

```
259             s = (xs+ys)
260             if s>1.0: s=1.0
261             elif s<-1.0: s=-1.0
262             if lyaw>90 and yaw<-90:
263                 corr=(lyaw-yaw-360)/60
264                 if corr>1.0: corr = 1.0
265                 elif corr<-1.0: corr = -1.0
266                 s+=corr
267             elif lyaw<-90 and yaw>90:
268                 corr=(lyaw-yaw+360)/60
269                 if corr>1.0: corr = 1.0
270                 elif corr<-1.0: corr = -1.0
271                 s+=corr
272             else:
273                 corr=(lyaw-yaw)/60
274                 if corr>1.0: corr = 1.0
275                 elif corr<-1.0: corr = -1.0
276                 s+=corr
277         if s>1.0: s=1.0
278         elif s<-1.0: s=-1.0
279         return s
280 
281     def move(self):
282         self.update_position()
283         self.cloud.retrieve_check_data(self)
284         self.control()
285 
286     def control(self):
287         t = s = b = 0.0
288         if self.override_brake:
289             control = carla.VehicleControl(throttle=0, steer=0, brake=1.0)
290         else:
291             wp = self.waypoints
292             x = self.x
293             y = self.y
294             ya = self.yaw
295             s = self.define_steer(wp, x, y, ya)
296             sg = self.speedGoal
297             ss = self.speed
298             t, b = self.define_throttle(sg, ss, wp)
299             control = carla.VehicleControl(throttle=t, steer=s, brake=b)
300         self.vehicle.apply_control(control)
301 
302 class Leader(PlatoonMember):
303     def __init__(self, vehicle: carla.Vehicle):
304         super().__init__(vehicle)
305         self.followers = []
```

```
306
307     def connect_to_cloud(self, sc: SafeCloud):
308         if not self.cloud:
309             self.cloud = sc
310         sc.set_leader(self)
311
312     def addFollower(self, f: Follower):
313         self.followers.append(f)
314
315     def move(self):
316         self.update_position()
317         self.update_follower()
318
319     def IMU_callback(self, sensor_data):
320         limits = (-99.9, 99.9)
321         self.accelerometer = (
322             max(limits[0], min(limits[1], sensor_data.accelerometer.x)),
323             max(limits[0], min(limits[1], sensor_data.accelerometer.y)),
324             max(limits[0], min(limits[1], sensor_data.accelerometer.z)))
325         self.gyroscope = (
326             max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.x))),
327             max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.y))),
328             max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.z))))
329         self.compass = math.degrees(sensor_data.compass)
330
331     def update_follower(self):
332         self.waypoints.append([self.x, self.y, self.yaw, self.steer])
333         for follower in self.followers:
334             if self.speed>0.03:
335                 follower.add_waypoint([self.x, self.y, self.yaw, self.steer])
336                 follower.set_speed_goal(self.speed)
337
```

## A.2 cloud.py

```
class SafeCloud:

    def __init__(self):
        self.members =[]
        self.leader = None
        self.check_args=[]

    def set_leader(self, l):
        if not self.leader:
            self.leader=l
            print("Leader set.")

    def add_members(self, m):
        if len(self.members)>0:
            l = self.members[-1]
        elif self.leader!=None:
            l = self.leader
        else:
            print("No leader available")
            return
        m.set_leading_vehicle(l)
        self.members.append(m)
        print("Connected followers:",len(self.members))
        self.leader.addFollower(m)

    def retrieve_check_data(self,vehicle):
        for elem in self.members:
            if vehicle==elem:
                wp = elem.waypoints
                x = elem.vehicle.get_transform().location.x
                y = elem.vehicle.get_transform().location.y
                ya = elem.vehicle.get_transform().rotation.yaw
                sg = self.leader.speed
                ss = elem.speed
                self.check_args=[wp,x,y,ya,sg,ss]
```

## A.3 scenario.py

```
import glob
import os
import sys
import random
import weakref
from threading import Thread
from time import sleep

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla
from platooning import Follower, Leader
from cloud import SafeCloud

actor_list = []
platoon_members = []
traffic = [None]*10

try:
    #**********************
    #---CONNECT TO SERVER---
    #**********************
    client = carla.Client('localhost', 2000)
    client.set_timeout(5.0)
    client.load_world('TownXX')
    client.reload_world()
    world = client.get_world()
    settings = world.get_settings()
    settings.fixed_delta_seconds = 0.01
    settings.synchronous_mode = True
    world.apply_settings(settings)

    blueprint_library = world.get_blueprint_library()

    cloud = SafeCloud()

    #*******************
    #---TRASH CLEANUP---
```

```
45      #******************
46      for v in world.get_actors():
47          if isinstance(v, carla.Vehicle):
48              v.destroy()
49
50      #****************
51      #---CARS SETUP---
52      #****************
53      spawn = carla.Transform(carla.Location(x=XXX.XX, y=-XXX.XX, z=XXX.XX),
54              carla.Rotation(pitch=XXX.XX, yaw=XXX.XX, roll=XXX.XX))
55      model3 = blueprint_library.filter('model3')[0]
56
57      #*****************
58      #---LIDAR SETUP---
59      #*****************
60      sensor_spawn = carla.Transform(carla.Location(x=2.5, z=0.8))
61      lidar_bp = blueprint_library.find('sensor.lidar.ray_cast')
62      lidar_bp.set_attribute('rotation_frequency','100')
63      lidar_bp.set_attribute('horizontal_fov','45')
64      lidar_bp.set_attribute('upper_fov','5')
65      lidar_bp.set_attribute('lower_fov','-5')
66      lidar_bp.set_attribute('range','20.0')
67
68      #*****************
69      #----IMU SETUP----
70      #*****************
71      imu_bp = blueprint_library.find('sensor.other.imu')
72
73      #******************
74      #---SPAWN LEADER---
75      #******************
76      PlatooningLeader = world.spawn_actor(model3, spawn)
77      PlatooningLeader.set_autopilot(True)
78      actor_list.append(PlatooningLeader)
79      ImuLeader = world.spawn_actor(imu_bp, sensor_spawn, attach_to=PlatooningLeader)
80      actor_list.append(ImuLeader)
81      ImuLeader.accelerometer = (0.0, 0.0, 0.0)
82      ImuLeader.gyroscope = (0.0, 0.0, 0.0)
83      ImuLeader.compass = (0.0)
84      leader = Leader(PlatooningLeader)
85      leader.connect_to_cloud(cloud)
86      world.on_tick(lambda snap: Thread(leader.move()).start())
87      ImuLeader.listen(lambda sensor_data: leader.IMU_callback(sensor_data))
88
89      #********************
90      #---SPAWN FOLLOWER---
91      #********************
```

```
92      spawn.location.X -= 12
93      model3.set_attribute('color','255,0,0')
94      PlatooningFollower = world.spawn_actor(model3, spawn)
95      actor_list.append(PlatooningFollower)
96      ImuFollower = world.spawn_actor(imu_bp, sensor_spawn,
97                    attach_to=PlatooningFollower)
98      actor_list.append(ImuFollower)
99      ImuFollower.accelerometer = (0.0, 0.0, 0.0)
100     ImuFollower.gyroscope = (0.0, 0.0, 0.0)
101     ImuFollower.compass = (0.0)
102     LidarFollower = world.spawn_actor(lidar_bp, sensor_spawn,
103                     attach_to=PlatooningFollower)
104     actor_list.append(LidarFollower)
105     follower = Follower(PlatooningFollower)
106     follower.connect_to_cloud(cloud)
107     platoon_members.append(follower)
108     world.on_tick(lambda snap: Thread(follower.move()).start())
109     LidarFollower.listen(lambda points: follower.check_lidar(points))
110     ImuFollower.listen(lambda sensor_data: follower.IMU_callback(sensor_data))
111
112     #**********************
113     #---SPAWN FOLLOWER2---
114     #**********************
115     spawn.location.X -= 12
116     model3.set_attribute('color','0,255,0')
117     PlatooningFollower2 = world.spawn_actor(model3, spawn)
118     actor_list.append(PlatooningFollower2)
119     ImuFollower2 = world.spawn_actor(imu_bp, sensor_spawn,
120                    attach_to=PlatooningFollower2)
121     actor_list.append(ImuFollower2)
122     ImuFollower2.accelerometer = (0.0, 0.0, 0.0)
123     ImuFollower2.gyroscope = (0.0, 0.0, 0.0)
124     ImuFollower2.compass = (0.0)
125     LidarFollower2 = world.spawn_actor(lidar_bp, sensor_spawn,
126                      attach_to=PlatooningFollower2)
127     actor_list.append(LidarFollower2)
128     follower2 = Follower(PlatooningFollower2)
129     follower2.connect_to_cloud(cloud)
130     platoon_members.append(follower2)
131     world.on_tick(lambda snap: Thread(follower2.move()).start())
132     LidarFollower2.listen(lambda points: follower2.check_lidar(points))
133     ImuFollower2.listen(lambda sensor_data: follower2.IMU_callback(sensor_data))
134
135     #*********************************
136     #---POINT CAMERA TO SPAWN POINT---
137     #*********************************
138     trans = spawn
```

```
    trans.location.x -= 12
    trans.location.z = 70
    trans.rotation.pitch=-90
    trans.rotation.yaw=0
    trans.rotation.roll=0
    world.get_spectator().set_transform(trans)

    while True:
        world.tick()

        #Position
        x_lead, y_lead = leader.get_pos()
        x_fol1, y_fol1 = follower.get_pos()
        x_fol2, y_fol2 = follower2.get_pos()
        with open('xtraf1.txt','a') as f:
            print(x_fol1, file = f)
        with open('ytraf1.txt','a') as g:
            print(y_fol1, file = g)
        with open('xtraf2.txt','a') as h:
            print(x_fol2, file = h)
        with open('ytraf2.txt','a') as j:
            print(y_fol2, file = j)
        with open('xlead.txt','a') as k:
            print(x_lead, file = k)
        with open('ylead.txt','a') as l:
            print(y_lead, file = l)

        #Speed
        slead = leader.get_speed()
        sfol1 = follower.get_speed()
        sfol2 = follower2.get_speed()
        with open('slead.txt','a') as m:
            print(slead, file = m)
        with open('sfol1.txt','a') as n:
            print(sfol1, file = n)
        with open('sfol2.txt','a') as o:
            print(sfol2, file = o)

except KeyboardInterrupt:
    pass

finally:
    print('destroying actors')
    client.apply_batch([carla.command.DestroyActor(x) for x in actor_list])
    print('done.')
```

# Bibliography


[1] A. Alam, B. Besselink, V. Turri, J. Mårtensson, K. Johansson, Heavy-duty vehicle platooning for sustainable freight transportation: A cooperative method to enhance safety and efficiency, IEEE Control Systems 35 (2015) 34–56. doi:10.1109/MCS.2015.2471046.

[2] M. Amoozadeh, A. Raghuramu, C.-N. Chuah, D. Ghosal, H. M. Zhang, J. Rowe, K. N. Levitt, Security vulnerabilities of connected vehicle streams and their impact on cooperative driving, IEEE Communications Magazine 53 (2015) 126–132.

[3] V. Bonneau, H. Yi, Autonomous cars: A big opportunity for european industry, European Commission - Advanced Technologies for Industry January (2017) 1–6.
URL https://ati.ec.europa.eu/reports/sectoral-watch/autonomous-cars-big-opportunity-european-industry

[4] O.-R. A. D. O. Committee, Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles (apr 2021). doi:https://doi.org/10.4271/J3016_202104.
URL https://doi.org/10.4271/J3016_202104

[5] S. Raviteja, R. Shanmughasundaram, Advanced driver assitance system (adas), in: 2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS), 2018, pp. 737–740. doi:10.1109/ICCONS.2018.8663146.

[6] C. Hartmann, Techday piloted driving - the traffic jam pilot in the new audi a8, Audi Mediacenter - Model Series, Innovation and Technology Communications (August 2017).
URL https://audimediacenter-a.akamaihd.net/system/production/uploaded_files/10075/file/33a16d09ec6a51fffdd575c4785f3e2b9b729feb/en_press_release_Audi_A8_traffic_jam_pilot.pdf?1504179630&disposition=attachment

[7] W. LLC, Waymo safety report (February 2021).
URL https://waymo.com/safety/

[8] D. J. Yeong, G. Velasco-Hernandez, J. Barry, J. Walsh, Sensor and sensor fusion technology in autonomous vehicles: A review, Sensors 21 (2021) 2140. doi:10.3390/s21062140.

[9] Z. Li, M. Yan, W. Jiang, P. Xu, Vehicle object detection based on rgb-camera and radar sensor fusion, in: 2019 International Joint Conference on Information, Media and Engineering (IJCIME), 2019, pp. 164–169. doi:10.1109/IJCIME49369.2019.00041.

[10] Y. Li, J. Ibanez-Guzman, Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems, IEEE Signal Processing Magazine 37 (4) (2020) 50–61. doi:10.1109/MSP.2020.2973615.

[11] A.-U.-H. Yasar, M. Adnan, W. Ectors, G. Wets, Comparison review on lidar technologies vs. radar technologies in speed enforcement system (09 2021). doi:10.13140/RG.2.2.20617.57440.

[12] X. Bai, Z. Hu, X. Zhu, Q. Huang, Y. Chen, H. Fu, C.-L. Tai, Transfusion: Robust lidar-camera fusion for 3d object detection with transformers (2022). doi:10.48550/ARXIV.2203.11496.
URL https://arxiv.org/abs/2203.11496

[13] H. Ahmad, A. Khan, W. Noor, G. Sikander, S. Anwar, Ultrasonic sensors based autonomous car parking system, Professional Trends in Industrial and Systems Engineering (PTISE) (04 2018).

[14] A. Electronics, Inertial sensors help autonomous machine applications, www.arrow.com [Online] (2021).
URL https://www.arrow.com/en/research-and-events/articles/adi-inertial-sensors-help-autonomous-machine-applications

[15] L. Lawal, C. Chatwin, A review of gnss and augmentation systems, Journal of Electrical and Electronics Engineering 5 (2019) 1–21.

[16] E. Commission, J. R. Centre, E. Cano-Pons, R. Giuliani, G. Baldini, D. Borio, C. Gioia, Robust GNSS services for road transportation : analysis and studies to mitigate GNSS

threats in the road transportation sector, Publications Office, 2017. doi:doi/10.2788/953749.

[17] Z. El-Rewini, K. Sadatsharan, N. Sugunaraj, D. F. Selvaraj, S. J. Plathottam, P. Ranganathan, Cybersecurity attacks in vehicular sensors, IEEE Sensors Journal 20 (22) (2020) 13752–13767. doi:10.1109/JSEN.2020.3004275.

[18] J. Petit, B. Stottelaar, M. Feiri, Remote attacks on automated vehicles sensors : Experiments on camera and lidar, Black Hat Europe (2015).

[19] B. G. Stottelaar, Practical cyber-attacks on autonomous vehicles (February 2015). URL http://essay.utwente.nl/66766/

[20] C. Yan, Can you trust autonomous vehicles : Contactless attacks against sensors of self-driving vehicle, DEFCON (2016).

[21] K. Faizal, Adaptive exposure control algorithm for day night video surveillance, vvdntech.com (2013).

[22] H. Shin, D. Kim, Y. Kwon, Y. Kim, Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications, in: Cryptographic Hardware and Embedded Systems – CHES 2017, Vol. 10529 of Lecture Notes in Computer Science, Springer, 2017, pp. 445–467.

[23] S. Parkinson, P. Ward, K. Wilson, J. Miller, Cyber threats facing autonomous and connected vehicles: Future challenges, IEEE Transactions on Intelligent Transportation Systems 18 (11) (2017) 2898–2915. doi:10.1109/TITS.2017.2665968.

[24] M. Harris, Researcher hacks self-driving car sensors, https://spectrum.ieee.org/researcher-hacks-selfdriving-car-sensors [Online] (2015).

[25] B. S. Lim, S. L. Keoh, V. L. L. Thing, Autonomous vehicle ultrasonic sensor vulnerability and impact assessment, in: 2018 IEEE 4th World Forum on Internet of Things (WF-IoT), 2018, pp. 231–236. doi:10.1109/WF-IoT.2018.8355132.

[26] W. Xu, C. Yan, W. Jia, X. Ji, J. Liu, Analyzing and enhancing the security of ultrasonic sensors for autonomous vehicles, IEEE Internet of Things Journal 5 (6) (2018) 5015–5029. doi:10.1109/JIOT.2018.2867917.

[27] S. Lee, W. S. Choi, D. H. Lee, Securing ultrasonic sensors against signal injection attacks based on a mathematical model, IEEE Access 7 (2019) 107716–107729.

[28] Y. Tu, Z. Lin, I. Lee, X. Hei, Injected and delivered: Fabricating implicit control over actuation systems by spoofing inertial sensors, CoRR abs/1806.07558 (2018). arXiv:1806.07558.
URL http://arxiv.org/abs/1806.07558

[29] S. Nashimoto, D. Suzuki, T. Sugawara, K. Sakiyama, Sensor con-fusion: Defeating kalman filter in signal injection attack, Proceedings of the 2018 on Asia Conference on Computer and Communications Security (2018).

[30] D. P. F. Möller, I. A. Jehle, R. E. Haas, Challenges for vehicular cybersecurity, 2018 IEEE International Conference on Electro/Information Technology (EIT) (2018) 0428–0433.

[31] J. Petit, S. E. Shladover, Potential cyberattacks on automated vehicles, IEEE Transactions on Intelligent Transportation Systems 16 (2) (2015) 546–556. doi:10.1109/TITS.2014.2342271.

[32] D. Marnach, S. Mauw, M. Martins, C. Harpes, Detecting meaconing attacks by analysing the clock bias of gnss receivers, Artificial Satellites 48 (01 2013). doi:10.2478/arsa-2013-0006.

[33] S. Narain, A. Ranganathan, G. Noubir, Security of gps/ins based on-road location tracking systems, 2019 IEEE Symposium on Security and Privacy (SP) (2019) 587–601.

[34] X. Shao, C. Dong, L. Dong, Research on detection and evaluation technology of cybersecurity in intelligent and connected vehicle, 2019 International Conference on Artificial Intelligence and Advanced Manufacturing (AIAM) (2019) 413–416.

[35] A. B. Lopez, A. V. Malawade, M. A. A. Faruque, S. Boddupalli, S. Ray, Security of emergent automotive systems: A tutorial introduction and perspectives on practice, IEEE Design & Test 36 (2019) 10–38.

[36] C. Bergenhem, S. Shladover, E. Coelingh, C. Englund, S. Tsugawa, Overview of platooning systems, 2012.

[37] D. Lu, Z. Li, D. Huang, Platooning as a service of autonomous vehicles, 2017, pp. 1–6. doi:10.1109/WoWMoM.2017.7974353.

[38] F. Fakhfakh, M. Tounsi, M. Mosbah, Vehicle platooning systems: Review, classification and validation strategies, International Journal of Networked and Distributed Computing 8 (06 2020). doi:10.2991/ijndc.k.200829.001.

[39] A. Böhm, K. Kunert, Data age based mac scheme for fast and reliable communication within and between platoons of vehicles, in: 2016 IEEE 12th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), 2016, pp. 1–9. doi:10.1109/WiMOB.2016.7763224.

[40] R. Passerone, D. Cancila, M. Albano, S. Mouelhi, S. Plósz, E. Jantunen, A. Ryabokon, L. Emine, C. Hegedus, P. Varga, A methodology for the design of safety-compliant and secure communication of autonomous vehicles, IEEE Access PP (2019) 1–1. doi:10.1109/ACCESS.2019.2937453.

[41] F. Castanedo, A review of data fusion techniques, TheScientificWorldJournal 2013 (2013) 704504. doi:10.1155/2013/704504.

[42] B. Shahian Jahromi, T. Tulabandhula, S. Cetin, Real-time hybrid multi-sensor fusion framework for perception in autonomous vehicles, Sensors 19 (20) (2019). doi:10.3390/s19204357.
URL https://www.mdpi.com/1424-8220/19/20/4357

[43] S. Dasgupta, M. Rahman, M. Islam, M. Chowdhury, A sensor fusion-based gnss spoofing attack detection framework for autonomous vehicles, IEEE Transactions on Intelligent Transportation Systems 23 (12) (2022) 23559–23572. doi:10.1109/TITS.2022.3197817.

[44] P. Lu, L. Zhang, B. B. Park, L. Feng, Attack-resilient sensor fusion for cooperative adaptive cruise control, in: 2018 21st International Conference on Intelligent Transportation Systems (ITSC), 2018, pp. 3955–3960. doi:10.1109/ITSC.2018.8569578.

[45] R. Ivanov, M. Pajic, I. Lee, Attack-resilient sensor fusion for safety-critical cyber-physical systems, ACM Transactions on Embedded Computing Systems (TECS) 15 (2016) 1 – 24.

[46] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, Z. M. Mao, Adversarial sensor attack on lidar-based perception in autonomous driving, CoRR abs/1907.06826 (2019). arXiv:1907.06826.
URL http://arxiv.org/abs/1907.06826

[47] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, V. Koltun, CARLA: An open urban driving simulator, in: Proceedings of the 1st Annual Conference on Robot Learning, 2017, pp. 1–16.

[48] Statista, Worldwide electric vehicle sales by model 2021 (2022).
URL https://www.statista.com/statistics/960121/sales-of-all-electric-vehicles-worldwide-by-model/

[49] K. Mark, Global plug-in car sales: 900k in december, 6.5 million in 2021 (2022).
URL https://insideevs.com/news/564639/global-plugin-car-sales-december2021/

[50] R. S. Hallyburton, Y. Liu, M. Pajic, Security analysis of camera-lidar semantic-level fusion against black-box attacks on autonomous vehicles, CoRR abs/2106.07098 (2021). arXiv:2106.07098.
URL https://arxiv.org/abs/2106.07098

[51] E. Khanapuri, T. Chintalapati, R. Sharma, R. Gerdes, Learning based vehicle platooning threat detection, identification and mitigation (02 2021). doi:10.13140/RG.2.2.18118.40003.

[52] J. Shen, J. Y. Won, Z. Chen, Q. A. Chen, Drift with devil: Security of multi-sensor fusion based localization in high-level autonomous driving under gps spoofing (extended version) (2020). doi:10.48550/ARXIV.2006.10318.
URL https://arxiv.org/abs/2006.10318

[53] A. Alipour-Fanid, M. Dabaghchian, H. Zhang, K. Zeng, String stability analysis of cooperative adaptive cruise control under jamming attacks, 2017, pp. 157–162. doi:10.1109/HASE.2017.39.

[54] A. Chowdhury, G. Karmakar, J. Kamruzzaman, A. Jolfaei, R. Das, Attacks on self-driving cars and their countermeasures: A survey, IEEE Access 8 (2020) 207308–207342. doi:10.1109/ACCESS.2020.3037705.