Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA

FuSeCar

Future generation Security for smart and connected Cars - FuSeCar

Deliverable D4.2: Implementation of local misbehavior detection algorithm on embedded devices

WP4: Misbehavior detection for vehicular communication protocols and architectures

Authors:
Giovanni Gambigliani Zoccoli[1], Mattia Trabucco[2], Mauro Andreolini[2], Luca Ferretti[2], and Mirco Marchetti[1]
{name.surname}@unimore.it

[1]Department of Engineering "Enzo Ferrari"
[2]Department of Physics, Informatics and Mathematics
University of Modena and Reggio Emilia

Current revision: R1.1
Delivery date: November 13th, 2025

Revision history

Authors	Changes	Date	Revision
Giovanni Gambigliani Zoccoli Luca Ferretti Mirco Marchetti	Creation of the document, tentative structure	March 3rd, 2025	R0.1
Mattia Trabucco Mauro Andreolini Luca Ferretti Mirco Marchetti	First draft of Section 1	April 3rd, 2025	R0.2
Giovanni Gambigliani Zoccoli Luca Ferretti Mauro Andreolini	First draft of Section 2	May 28th, 2025	R0.3
Mattia Trabucco Luca Ferretti Mauro Andreolini Mirco Marchetti	First draft of Section 3	July 17th, 2025	R0.4
Giovanni Gambigliani Zoccoli Mattia Trabucco Mirco Marchetti	Refinement of Section 3	September 5th, 2025	R0.5
Mattia Trabucco Luca Ferretti Mauro Andreolini Mirco Marchetti	Draft of Section 4	September 28th, 2025	R0.6
Luca Ferretti Mauro Andreolini Mirco Marchetti	First draft of complete document	October 23rd, 2025	R1.0
Giovanni Gambigliani Zoccoli Mattia Trabucco Luca Ferretti Mauro Andreolini Mirco Marchetti	Revision of document and minor fixes	November 13th, 2025	R1.1

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA

Contents

1 Introduction

This section introduces the core objectives of Task 4.2, the importance of implementing the local misbehavior detection algorithm, and the challenges of deploying it in real-world environments. In Section 1.1, we provide an overview of Task 4.2, describing its main objectives and the significance of implementing the detection system on embedded devices. In Section 1.2, we discuss the challenges of transitioning from theoretical models and simulation environments to real-world testing, which will be a key focus of Task 4.2. Section 1.3 explores how the local misbehavior detection algorithm integrates with existing Cooperative Intelligent Transport Systems (C-ITS) architectures. Finally, we conclude by summarizing the objectives of Task 4.2 and the expected impact of this work on the development of secure, real-time vehicular communication systems.

1.1 Task Overview and Importance of Embedded Implementation

Task 4.2 focuses on the implementation of the local misbehavior detection algorithm, which was designed and proposed in Task 4.1, onto embedded devices for real-world deployment in Cooperative Intelligent Transport Systems (C-ITS). The core aim of Task 4.2 is to create a scalable, efficient, and cost-effective solution for detecting misbehaving vehicles or entities within a C-ITS network. Detecting such misbehavior in real-time is essential for ensuring the security, safety, and trustworthiness of vehicular communication systems, which are increasingly being deployed on the roads.

Detecting misbehavior in real-time is essential for ensuring the security, safety, and trustworthiness of vehicular communication systems. Misbehaving entities in C-ITS, such as malicious vehicles that broadcast incorrect information, can cause a range of problems, from minor traffic disruptions to severe accidents. For instance, a vehicle that falsely broadcasts its position could mislead nearby vehicles, causing them to take unsafe actions, such as braking or changing lanes unnecessarily. Detecting such misbehavior in real-time on-board vehicles, without relying on external infrastructure, is vital for effective operation. This is especially important in autonomous driving scenarios, where timely and accurate decisions are necessary to prevent accidents.

The embedded approach proposed in Task 4.2 ensures that each vehicle can independently assess the validity of the information it receives via Vehicle-to-Vehicle (V2V) communications. By leveraging local sensors and communication channels, the vehicle can validate the integrity of the received data, allowing for immediate action if anomalies are detected. This local detection mechanism empowers vehicles to make informed decisions in a timely manner, whether it involves avoiding a collision, verifying received safety messages, or rejecting malicious communication. This decentralized approach is more resilient and scalable, as it removes the dependency on centralized infrastructure or cloud-based systems, which could introduce latency or be vulnerable to attacks.

In traditional vehicular systems, misbehavior detection often occurs in centralized control environments or through infrastructure-based systems. However, these systems may not always be suitable for real-time applications, as they can suffer from delays due to communication between vehicles and centralized servers, or issues related to the availability and reliability of infrastructure. In contrast, real-time detection on-board vehicles allows for immediate feedback and decision-making, which is critical for preventing accidents and improving the overall efficiency of the transportation network. As the automotive industry moves toward autonomous vehicles and smart cities, local misbehavior detection will play an essential role in ensuring the safe operation of these systems.

A significant challenge faced by research groups working in this domain is the transition from theoretical models and simulation environments to real-world applications. Although Task 4.1 provided the theoretical design and initial validation of the detection algorithm in a controlled simulation environment, it is essential to evaluate how the algorithm performs under real-world conditions. Task 4.2 addresses this by focusing on the testing and implementation of the detection algorithm on embedded hardware, ensuring that it operates efficiently with real sensor data, and in dynamic traffic scenarios. The goal is to bridge the gap between

the simulated ideal conditions and the complexities of real-world environments, where traffic patterns, sensor inaccuracies, and other unpredictable factors can affect the detection process.

1.2 Real-World Testing Challenges

The transition from simulation environments to real-world testing is a crucial step in validating the performance and effectiveness of the misbehavior detection algorithm. Subsection 1.1 highlighted the importance of real-time detection, but real-world deployment introduces several challenges. These include sensor noise, dynamic vehicle interactions, and the need for robust performance in environments with varying levels of connectivity. For example, detecting a malicious vehicle in an urban environment, where vehicles are often surrounded by tall buildings and other obstacles, requires the algorithm to account for occlusions and GPS inaccuracies. Similarly, in high-speed highway scenarios, the algorithm must operate with low latency to ensure that vehicles can make split-second decisions.

Additionally, the real-world application of the algorithm will help validate its scalability and efficiency. With the rapid growth of connected and autonomous vehicles, it is essential that misbehavior detection systems can be deployed across a large number of vehicles in a cost-effective manner. The real-world testing conducted in Task 4.2 will help assess the practicality of such large-scale deployments and identify any issues that need to be addressed before the algorithm can be widely adopted in the automotive industry.

1.3 Integration with Existing C-ITS Architectures

A central aspect of Task 4.2 is the seamless integration of the local misbehavior detection algorithm into existing C-ITS architectures. Current C-ITS networks already use a range of communication protocols and standards, such as ETSI ITS-G5 for V2X communication. The misbehavior detection system must be designed to work alongside these existing systems without introducing additional overhead or complexity. Task 4.2 will explore how the algorithm can be integrated into real-world vehicles that already support these communication standards, ensuring that the detection system does not interfere with other critical vehicle functions, such as navigation or autonomous driving systems.

The algorithm's performance will also be evaluated in various vehicle configurations, ranging from traditional vehicles to advanced autonomous systems. This ensures that the detection system is adaptable to different levels of vehicle automation, and can be scaled as autonomous driving technologies continue to evolve.

In conclusion, Task 4.2 aims to implement and test the local misbehavior detection algorithm in real-world conditions, with a focus on ensuring its efficiency, scalability, and integration into existing C-ITS infrastructures. By moving beyond theoretical models and simulations, this task will provide valuable insights into the practical challenges and opportunities of deploying misbehavior detection systems in the context of connected and autonomous vehicles.

2 System Architecture and Embedded Platform

This section provides an overview of the system architecture used to implement the local misbehavior detection algorithm on embedded devices. In Section 2.1, we describe the embedded platform selected for this implementation, including the hardware components and their specifications. In Section 2.2, we detail the software stack used to run the detection algorithm, focusing on the integration of the detection system with existing C-ITS communication protocols.

2.1 Embedded Hardware Overview

This section presents the embedded computing platforms adopted in the implementation of the local misbehavior detection on embedded devices. The system relies on a distributed architecture composed of two main boards with complementary roles. In Section 2.1.1 we present the Raspberry Pi 5, which operates as the On-Board Unit (OBU) handling V2X communication and interaction with the vehicular networking stack. In Section 2.1.2 we describe the NVIDIA Jetson Orin Nano, which is responsible for perception, sensor fusion, and execution of the autonomous driving stack.

2.1.1 Raspberry Pi 5

The hardware platform selected for the implementation of the local misbehavior detection algorithm is based on a Raspberry Pi 5, a low-cost and compact single-board computer. The Raspberry Pi 5 provides sufficient computational power to support the real-time processing required for misbehavior detection, making it a suitable choice for embedded systems in vehicular environments [4]. This choice allows for easy scalability and widespread deployment, particularly when testing multiple units at once.

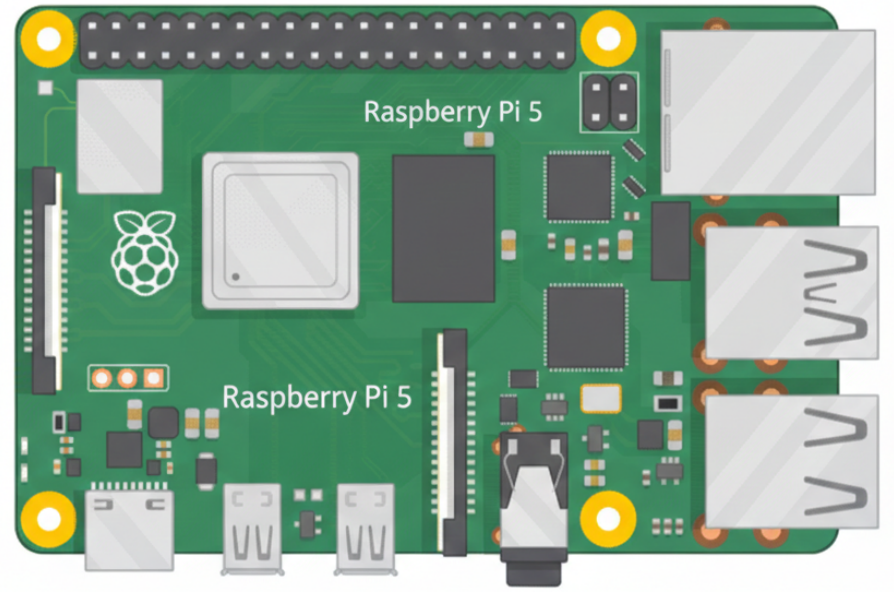

Figure 1: Raspberry Pi 5 board.

The Raspberry Pi 5 features an ARM Cortex-A76 processor, 8 GB of LPDDR4 RAM, and 128 GB of flash storage, which together enable efficient data processing and storage. The device's small form factor makes it ideal for integration into both conventional vehicles and smaller-scale test vehicles used for experimentation as shown in Figure 1. The Raspberry Pi 5 also offers a range of connectivity options, including Wi-Fi and

Bluetooth, which are essential for Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) communication in C-ITS applications [1].

To enable communication at the 5.9 GHz frequency band, required for C-ITS, we integrated the Mikrotik R11E-5HND Wi-Fi mPCIe module. This module is built on the Qualcomm Atheros AR9580 chipset which can be fully compatible with the IEEE 802.11p standard for V2X communication (Figure 2). This setup ensures that the embedded platform can interact with other vehicles and infrastructure components within a C-ITS network, making it compliant with the necessary communication protocols [3]. The Wi-Fi module also allows for high-speed data transmission, which is crucial for real-time misbehavior detection.

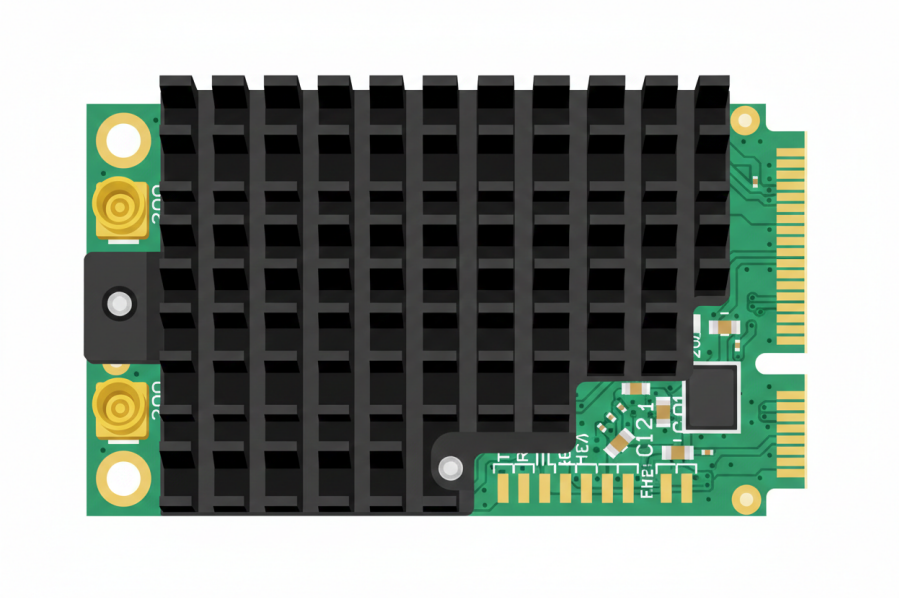

Figure 2: Mikrotik Wi-Fi module.

In addition to its processing capabilities, the Raspberry Pi 5 is equipped with a PCIe 2.0 interface that supports high-speed peripheral devices, including the Wi-Fi module. Powering the Raspberry Pi 5 requires a 5V/5A DC USB-C connection, which must be considered when designing battery-powered systems, especially in mobile environments. The overall hardware architecture as shown in Figure 3.

2.1.2 NVIDIA Jetson Orin Nano

The NVIDIA Jetson Orin Nano serves as the core computational unit for the autonomous vehicle system. It provides the necessary processing power to handle data from sensors such as LiDAR, cameras, and IMU in real time. Equipped with a GPU based on the Ampere architecture and ARM Cortex-A78AE CPU cores, the Jetson Orin Nano is optimized for high-performance computing with low power consumption, making it ideal for autonomous driving applications. The graphic representation of the board is presented in Figure 4.

In this system, the Jetson Orin Nano processes data from a LiDAR sensor. In particular, to implement the conical shape presented in the D4.1 we employ the FHL-LD19 LiDAR, which is used for reliable environmental data capture with high-resolution ranging, supporting the vehicle's decision-making algorithms. The FHL-LD19 integrates seamlessly with ROS2, making it an efficient solution for autonomous systems [11]. While, instead for the elliptical shape presented in the previous deliverable (D4.1) we use the Livox LiDAR, which provides 360-degree point clouds of the environment, enabling comprehensive object detection and localization [12]. Images of the two sensors are depicted on Figure 5, where on the left side is reported the FHL-LD19 (Figure 5a), while on the right the Livox (Figure 5b).

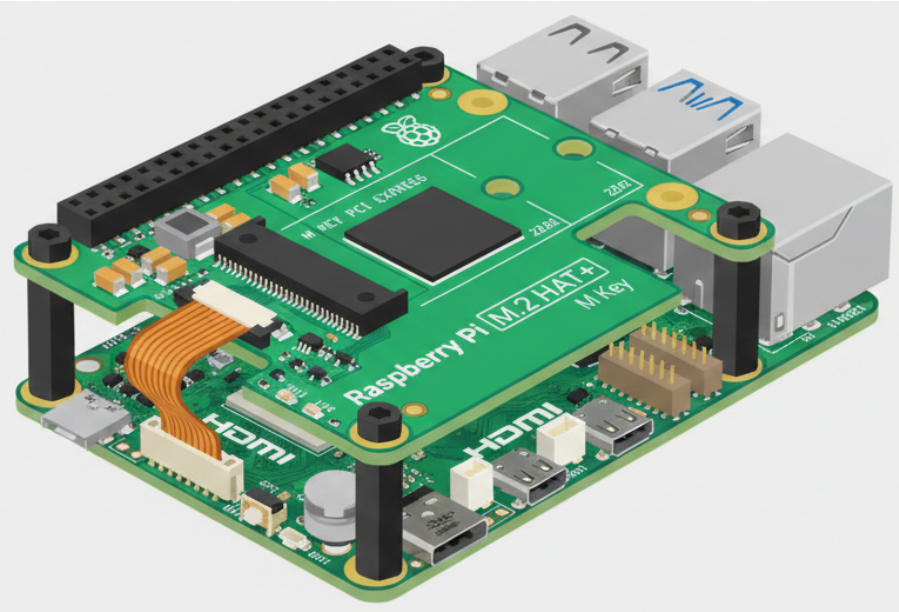

Figure 3: Raspberry Pi 5 and the PCIE Hat used to connect the Mikrotik module.

These data gathered from the sensors is fused using the `FAST_LIO` library, which is employed for real-time LiDAR processing, delivering high-precision localization and mapping capabilities [6].

The platform also integrates the IMU from the VESC (Vedder Electronic Speed Controller) for motion tracking. The VESC provides precise control of the vehicle's motor and includes an integrated IMU for measuring acceleration and angular velocity, which supports accurate vehicle localization [15].

The system relies on ROS2 for middleware to manage communication between the Jetson Orin Nano and other components such as the Raspberry Pi 5, which handles V2X communication through the Mikrotik R11E-5HND wireless card operating at 5.9 GHz for V2X communication [4].

2.2 Software Overview

The software stack used for this project follows a two-layer design, combining an ETSI-compliant C-ITS communication stack with a robotics-oriented perception and sensor-processing stack. On the V2X side, the platform is based on the OScar framework, an open-source implementation of the ETSI C-ITS stack. OScar was selected due to its lightweight design and compliance with C-ITS standards, making it suitable for real-time execution on low-cost embedded devices [10]. The framework is written in C++ and adopts a modular, multi-threaded architecture to optimize resource usage while maintaining high performance.

OScar runs on Linux-based operating systems such as Raspberry Pi OS, ensuring compatibility with the Raspberry Pi 5 and enabling the use of standard Linux libraries and tools [5]. The framework has been optimized to minimize dependencies, allowing it to run on low-power devices without sacrificing the critical functionality required for C-ITS applications. It includes several essential C-ITS services, such as Cooperative Awareness Messages (CAMs), Decentralized Environmental Notification Messages (DENMs), and Vulnerable Road User Awareness Messages (VAMs), enabling standardized information exchange between vehicles and infrastructure [10, 3].

A key feature of OScar is the Local Dynamic Map (LDM), which stores and maintains information about vehicles and objects in the surrounding environment. The LDM is relevant for the detection pipeline, as it provides a structured interface to track surrounding entities and to correlate received V2X messages with locally available evidence. The LDM is updated periodically and can be queried by external services via a JSON-

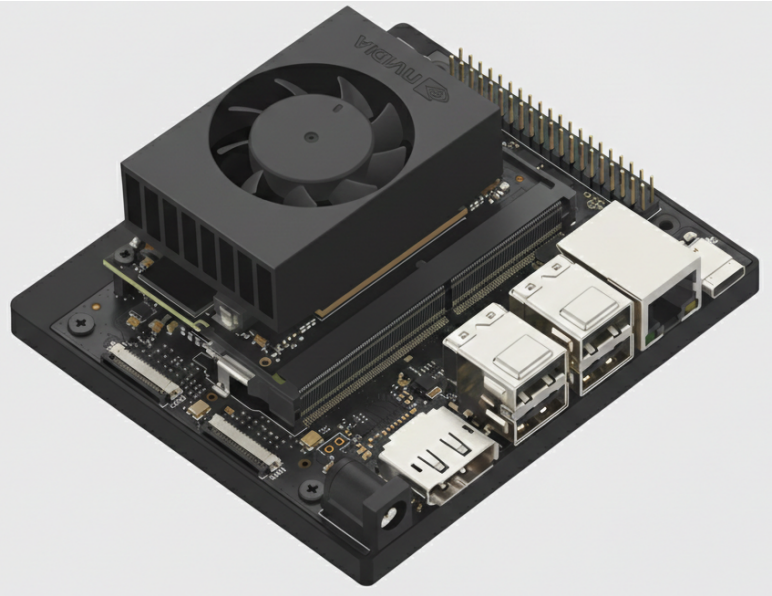

Figure 4: NVIDIA Jetson Orin Nano.

(a) FHL-LD19 LiDAR

(b) Livox LiDAR

Figure 5: LiDAR sensors

over-TCP API, facilitating integration with other components in the C-ITS ecosystem [3]. The multi-threaded nature of OScar further enables efficient handling of concurrent tasks (e.g., message reception, transmission, and data management), supporting high-frequency inputs while preserving predictable runtime behavior.

On the perception side, the demonstrator adopts ROS 2 as middleware to manage sensor data acquisition, inter-process communication, and modular composition of perception and localization components [8, 9]. In particular, LiDAR-based localization and mapping rely on FAST_LIO, which enables real-time processing of point clouds together with inertial measurements, and is used to provide accurate ego-motion estimation from the LiDAR/IMU pipeline [6]. This separation of concerns allows the V2X stack (OScar) and the robotics stack (ROS 2 + FAST_LIO) to evolve independently while enabling a clear integration point for local misbehavior detection, where communication-derived information can be validated against sensor-derived observations.

A critical aspect of the overall software architecture is the seamless integration of the local misbehavior detection algorithm with existing C-ITS communication protocols. Since the local detection algorithm is designed to function on-board vehicles in almost real-time, it must be compatible with established standards such as ETSI ITS-G5 and IEEE 802.11p for V2X communication [2, 7]. In the demonstrator, the Raspberry Pi equipped with the Mikrotik R11E-5HND Wi-Fi module can transmit and receive messages according to these

standards, enabling interaction with other vehicles and infrastructure nodes within the C-ITS network.

The local misbehavior detection algorithm processes incoming V2X messages (e.g., CAMs and DENMs) and compares their content with information gathered from local perceptual systems (e.g., radar, LiDAR, camera-based perception). The software integrates with OScar to access and manage communication-derived state (including through the LDM), and then validates received information by cross-checking it against sensor-derived observations. This design supports near real-time detection of inconsistencies, enabling the system to identify and react to potential threats without requiring continuous reliance on external infrastructure [3].

Finally, the modular nature of OScar and the ROS 2-based perception stack supports extensibility. Additional security services, new message types, or alternative perception and localization modules can be introduced without disrupting the core detection pipeline, improving long-term maintainability and facilitating adaptation to future upgrades of C-ITS protocols and standards [3].

In summary, the system architecture of the embedded platform combines the power of the Raspberry Pi 5 with the flexibility and compliance of the OScar framework to create a robust and scalable solution for local misbehavior detection in C-ITS. The system is designed to operate efficiently on low-cost, embedded hardware, providing real-time detection and communication in dynamic vehicular environments. The architecture also ensures compatibility with existing communication protocols and allows for future integration of new features and standards. The next section will explore how the detection algorithm is implemented and tested in real-world conditions, ensuring its practical feasibility for deployment in connected and autonomous vehicles.

3 Implementation of Local Misbehavior Detection

This section details the implementation of the local misbehavior detection module on embedded prototype devices, following the software architecture introduced in Section 2 and implementing the perceptual-coverage validation models (conical and elliptical sensing regions, including occlusion/shadow handling).

3.1 Software integration overview

The local detection pipeline integrates two main software subsystems:

- **V2X networking and CAM decoding** (Raspberry Pi 5 side): reception of ETSI ITS-G5 traffic, identification of message types, and extraction of sender pseudonym, timestamp, and claimed position from CAMs.
- **Perception/odometry and frame handling** (Jetson Orin Nano side): ROS 2-based perception output (surrounding objects as pose lists) and availability of a TF tree providing transforms among `lidar`, `base_link`, and `odom`/`map`.

At runtime, each received CAM is evaluated only if its claimed position lies within the modeled perceptual coverage area of the detector vehicle (cone or ellipse). Then, the message is matched against the most plausible perceived object, and an anomaly is raised if no perceived object is compatible within a tolerance.

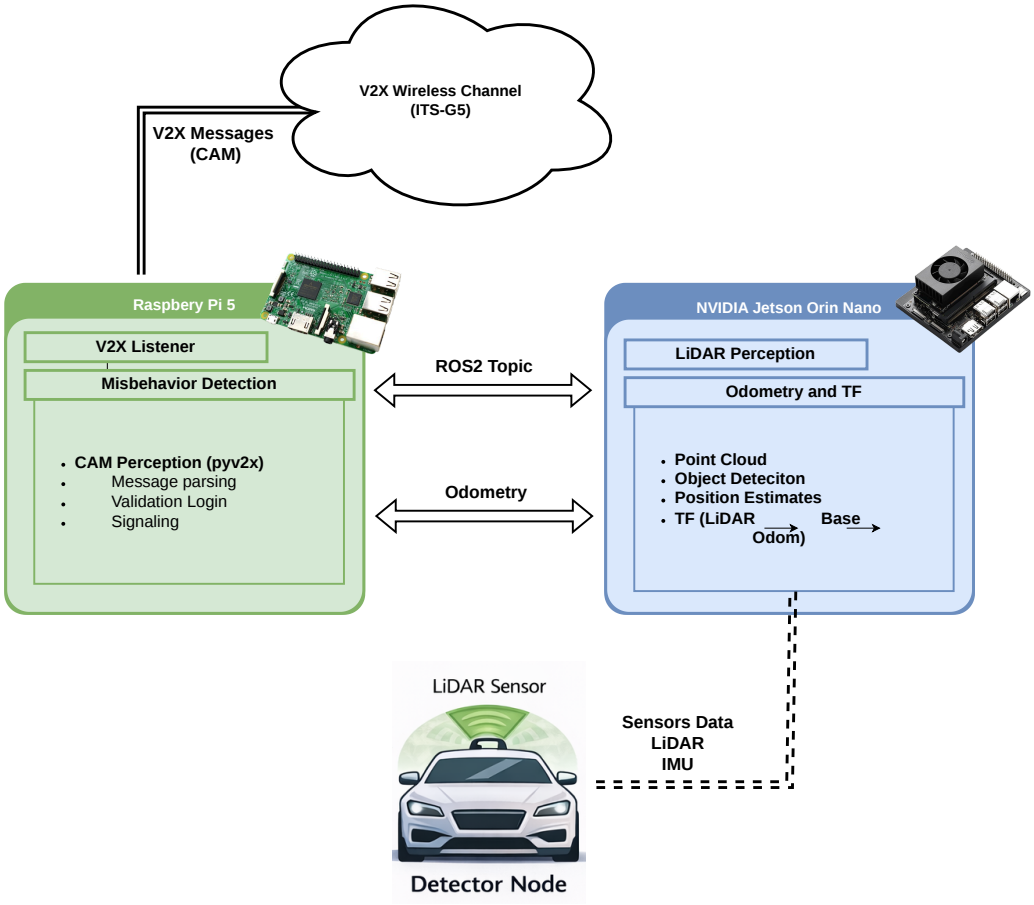

Figure 6: Diagram of the detection procedure.

The diagram of Figure 6 illustrates the architecture of the embedded V2X local misbehavior detection system implemented on the detector node. At the perception level, the **NVIDIA Jetson Orin Nano** processes raw sensor inputs (LiDAR and IMU) to perform LiDAR perception, object detection, position estimation, and coordinate frame transformations (TF from LiDAR to `base` and `odom` frames), while also managing odometry estimation. The processed perception outputs, including point clouds, object positions, and TF data, are published through ROS 2 topics.

In parallel, a **Raspberry Pi 5** hosts the V2X listener and the misbehavior detection module. It receives CAM messages from the ITS-G5 wireless channel, parses them using the code of the `pyv2x` project, and applies the validation logic to compare the claimed positions contained in the V2X messages with the locally perceived environment. The integration of perception data and V2X communication enables the detector to identify spatial inconsistencies and generate signaling outputs whenever misbehavior is detected.

3.2 V2X message reception and buffering

Listing 1 (from [13]) implements the reception of V2X traffic through a lightweight network stack that combines `pyshark` live capture with ETSI message identification and message-class dispatching.

Listing 1: Network stack implementation of V2X message reception (pyv2x)

```
from scapy.packet import Packet as p_scapy
from scapy.sendrecv import sendp, sniff
from scapy.config import conf

from pyshark.packet.packet import Packet as p_pyshark

from threading import Thread
from typeguard import typechecked
from typing import List, Type
from queue import Queue, Empty

from pyv2x.etsi import ETSI, V2xTMsg
from pyv2x.v2x_msg import V2xMsg

import pyshark

import time
import asn1tools
import sys


@typechecked
class V2xNetwork:

    def __init__(self, iface: str, ltmsg: List[Type[V2xMsg]], filter: str = "its", enable_listener: bool =
    ↪  True):

        if len(V2xTMsg) < len(ltmsg):
            raise Exception(f"the size of V2xTMsg is {len(V2xTMsg)} and support this type of msg: {' '.join([
            ↪  v for k, v in dict(V2xTMsg).items() ])}")

        self._ltmsg, self._iface = {}, iface
        for tmsg in ltmsg:
            if not hasattr(tmsg, "name"):
                raise Exception("wrong type class")
            self._ltmsg[ V2xTMsg.get(tmsg.name) ] = tmsg
```

```
35
36          self._queue = Queue(maxsize=0)
37          self._filter = filter
38          conf.verb = 0
39
40          if enable_listener:
41              tshark = Thread(target=self.start_listener_v2x, daemon=True).start()
42
43      def send_msg(self, packet: p_scapy) -> None:
44          sendp(packet, iface=self._iface)
45
46      def start_listener_v2x(self):
47          self._trace = pyshark.LiveCapture(interface=self._iface, use_json=True, include_raw=True,
        ↪  display_filter=self._filter)
48          for pkt in self._trace.sniff_continuously():
49              try:
50                  nh = ETSI.get_message_id(pkt)
51                  for id, tmsg in self._ltmsg.items():
52                      if id == nh:
53                          msg = tmsg(pkt=pkt)
54                          break
55                  else:
56                      msg = None
57              except Exception as e:
58                  # print(f"error: {str(e)}", file=sys.stderr)
59                  continue
60
61              if msg is not None:
62                  self._queue.put(msg)
63
64              time.sleep(0.1)
65
66      def is_empty(self) -> bool:
67          return self._queue.empty()
68
69      def get_new_msg(self) -> V2xMsg | None:
70          try:
71              return self._queue.get_nowait()
72          except Empty:
73              return None
74          except Exception as e:
75              raise Exception(str(e))
```

Live capture and filtering The component attaches to a network interface (parameter `iface`) and applies a display filter (default `"its"`), restricting processing to relevant ITS frames. For every captured packet, the message identifier is extracted, and the packet is converted into the appropriate internal message class (e.g., CAM) when supported.

Queuing messages Reception runs in a dedicated daemon thread, pushing decoded messages into an unbounded queue. The local detection loop pulls messages with a non-blocking call, which decouples networking from detection and avoids head-of-line blocking when CPU load increases.

Time alignment hook Each decoded CAM carries a generation timestamp. The same timestamp is used to query the most recent perception snapshot (object list and TF transforms) to ensure coherent comparisons between claimed and perceived positions.

3.3 Perception output normalization and coordinate frames

The perceptual system provides detections in a sensor-centric reference frame (typically the LiDAR frame). Since the local detector compares CAM positions and perceived positions in a common reference system, a dedicated ROS 2 node is used to transform the perceived objects into detector-friendly frames.

Object pose transformation. Listing 2 subscribes to a list of object poses published in the LiDAR frame (topic `/objects/lidar_frame`, represented as `PoseArray`). For each callback, the node:

- queries TF for `lidar`→`base_link` and `lidar`→`odom`,
- transforms each pose using standard ROS 2 TF utilities,
- publishes two synchronized outputs: `/objects/base_link` and `/objects/odom`.

Listing 2: Network stack implementation of V2X message reception

```python
import rclpy
from rclpy.node import Node

from geometry_msgs.msg import PoseArray, PoseStamped
from nav_msgs.msg import Odometry

import tf2_ros
from tf2_ros import TransformException
from tf2_geometry_msgs import do_transform_pose


class ObjectPositionTransformer(Node):
    def __init__(self):
        super().__init__('object_position_transformer')

        self.declare_parameter('input_topic', '/objects/lidar_frame')
        self.declare_parameter('lidar_frame', 'lidar')
        self.declare_parameter('base_frame', 'base_link')
        self.declare_parameter('odom_frame', 'odom')

        self.input_topic = self.get_parameter('input_topic').value
        self.lidar_frame = self.get_parameter('lidar_frame').value
        self.base_frame = self.get_parameter('base_frame').value
        self.odom_frame = self.get_parameter('odom_frame').value

        self.tf_buffer = tf2_ros.Buffer()
        self.tf_listener = tf2_ros.TransformListener(self.tf_buffer, self)

        self.sub = self.create_subscription(PoseArray, self.input_topic, self.cb_objects, 10)
        self.pub_base = self.create_publisher(PoseArray, '/objects/base_link', 10)
        self.pub_odom = self.create_publisher(PoseArray, '/objects/odom', 10)

        self.get_logger().info(f"Listening: {self.input_topic}, lidar_frame={self.lidar_frame}")

    def cb_objects(self, msg: PoseArray):
        # Prepare output messages
        out_base = PoseArray()
        out_base.header.stamp = msg.header.stamp
        out_base.header.frame_id = self.base_frame
```

```
        out_odom = PoseArray()
        out_odom.header.stamp = msg.header.stamp
        out_odom.header.frame_id = self.odom_frame

        # Lookup transforms
        try:
            T_lidar_to_base = self.tf_buffer.lookup_transform(
                self.base_frame, self.lidar_frame, rclpy.time.Time()
            )
            T_lidar_to_odom = self.tf_buffer.lookup_transform(
                self.odom_frame, self.lidar_frame, rclpy.time.Time()
            )
        except TransformException as ex:
            self.get_logger().warn(f"TF lookup failed: {ex}")
            return

        # Transform each pose
        for pose in msg.poses:
            ps = PoseStamped()
            ps.header = msg.header
            ps.header.frame_id = self.lidar_frame
            ps.pose = pose

            ps_base = do_transform_pose(ps, T_lidar_to_base)
            ps_odom = do_transform_pose(ps, T_lidar_to_odom)

            out_base.poses.append(ps_base.pose)
            out_odom.poses.append(ps_odom.pose)

        self.pub_base.publish(out_base)
        self.pub_odom.publish(out_odom)


def main():
    rclpy.init()
    node = ObjectPositionTransformer()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

This design keeps the frame-conversion logic isolated from both the perception modules and the detection logic, improving maintainability and easing deployment on constrained devices.
If the ROS 2 TF is temporarily unavailable (startup or transient dropouts), the node skips the update and logs a warning. This prevents propagating stale/invalid transforms into the detector and avoids false anomalies caused by inconsistent frames.

3.4 Comparison between real and simulated data

In the simulator, cone/ellipse coverage models were used to approximate the perceptual visibility region. With a real LiDAR, the first-stage gating is instead driven by the *physical sensor coverage*. For each received CAM, the detector computes the distance between the ego position and the claimed sender position. If the claim lies outside the LiDAR maximum effective range R_{LiDAR}, the message is marked as *not verifiable* by local perception and the detection for that CAM stops.
For claims within range, the pipeline proceeds by retrieving the perceived positions of surrounding objects from the LiDAR-based perception output and expressing them in a common reference frame using odometry and TF transformations. In our implementation, the `ObjectPositionTransformer` node converts object poses from the LiDAR frame into `base_link` and `odom`, enabling direct comparison with CAM claims after applying the same coordinate conversion.
The detector then checks if the position of the CAM message received is valid (falls inside the coverage area of the LiDAR) and proceed with the detection task comparing the position of the CAM message and the data gathered from the physical sensor. Otherwise the detection procedure ends.

3.5 Anomaly detection signaling

Once a CAM claim is considered verifiable (i.e., inside the visible region), the detector executes the local inconsistency check.

Claim extraction. For each CAM, the implementation extracts the claimed sender position and (when available) supporting kinematic fields. The claim is converted to the same reference frame used for perceived objects (typically `odom` for temporal consistency). When the CAM provides geodetic coordinates, a projection step is applied consistently across the pipeline (e.g., converting to a local ENU frame).

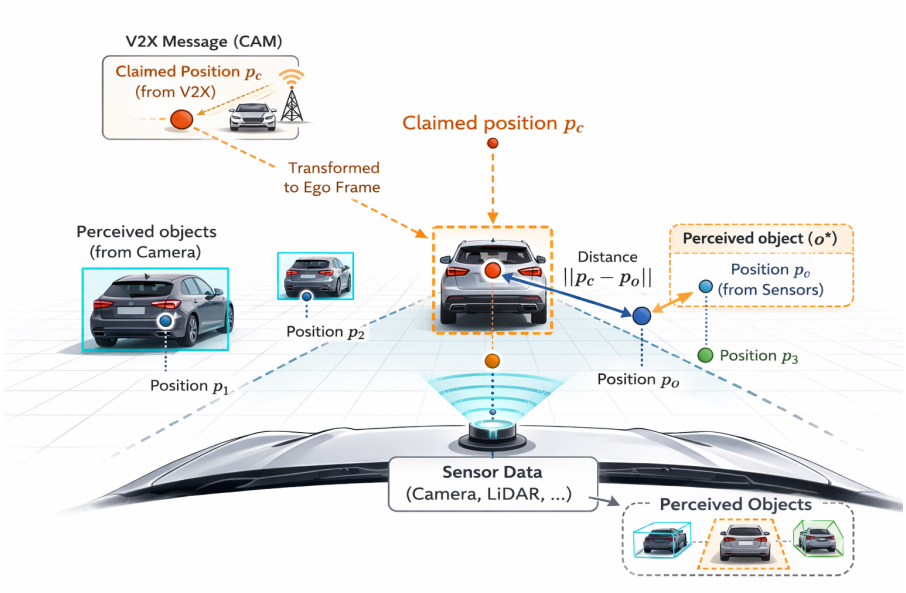

Figure 7: Diagram of the detection procedure.

Association to perceived objects. The detector searches for a perceived object compatible with the CAM sender by matching the claimed position against the transformed object positions. The baseline association

strategy is nearest neighbor in the horizontal plane:

$$o^\star = \arg\min_{o \in \mathcal{O}_t} \|p_c - p_o\|\,, \tag{1}$$

where p_c is the claimed position and p_o the perceived object position.
A graphical representation of the nearest neighbor association is reported in Figure 7, where the orange vehicle is selected among the other vehicles resulting the closest compared to the claimed position (pc) of the V2X message.

Tolerance-based decision. If no perceived object is found within a configured tolerance τ, the CAM is flagged as suspicious. If a match exists and the discrepancy is below τ, the CAM is considered consistent. The tolerance is set conservatively to account for perception uncertainty, GNSS errors (when present), and discretization effects due to time alignment.

Temporal consistency. To reduce false positives caused by brief tracking losses or asynchronous update rates, the detector applies a bounded temporal alignment window when pairing CAMs with perception snapshots. Additionally, a persistence rule can be applied (e.g., require multiple consecutive inconsistent CAMs from the same sender before raising a final alert), improving robustness in real field conditions.

4 Testing against SixPack Attack

To validate the proposed embedded implementation of local misbehavior detection, we conducted a set of tests targeting the stealthy dynamic attack SixPack $v2$ [16]. The objective was to verify that the end-to-end pipeline (V2X message reception and decoding, perception-based position extraction, and local consistency checks) is able to identify forged CAM claims generated by an attacker.

The tests were performed using the embedded prototype devices described in Section 2 and Section 3. The local detector was executed on the receiver vehicle, while the attacker logic was deployed on a Raspberry Pi 5. In particular, SixPack $v2$ was implemented on the Raspberry Pi 5 by adapting the `pyv2x` repository to craft and transmit CAM messages consistent with the SixPack $v2$ strategy (i.e., manipulating the claimed sender position over time while preserving plausible kinematic fields). The attacker transmitted the forged CAMs over the V2X interface at a rate comparable to standard CAM beaconing [14].

On the defender side, the receiver vehicle executed the local detection pipeline: CAM messages were collected and decoded by the network component, then compared against perceived object positions obtained from the LiDAR perception stack and expressed in a common frame through odometry and ROS 2 TF transformations. As described in the implementation section, messages outside the LiDAR maximum effective range were discarded as not locally verifiable, while messages within range were validated via association with perceived objects and tolerance-based discrepancy checks.

4.1 Test procedure

During each run, the attacker broadcast forged CAMs according to SixPack $v2$ while the receiver vehicle collected, transformed, and associated perception outputs in real time. For each received CAM within LiDAR coverage, the detector attempted to match the claim with the closest perceived object. When no compatible perceived object existed within the association gate, or when the mismatch between claimed and perceived position exceeded the configured tolerance, the CAM was flagged as suspicious. To reduce the impact of transient perception noise, a bounded temporal alignment policy was applied when pairing CAM timestamps with perception snapshots.

4.2 Outcome

The experiments confirmed that the implemented local detector is able to identify the falsified CAM generated from the SixPack $v2$ attack. In particular, when the attacker reported positions that were inconsistent with the physically observed environment (despite being within the LiDAR coverage area), the detector raised alerts due to the lack of a compatible perceived object and/or due to excessive position discrepancy with respect to the matched perceived target. This demonstrates that the embedded implementation can effectively detect stealthy dynamic falsification attempts such as SixPack $v2$ under realistic execution constraints.

References

[1] Anja Dakić, Benjamin Rainer, Markus Hofer, Stefan Zelenbaba, Stefan Teschl, Guo Nan, Peter Priller, Xiaochun Ye, and Thomas Zemen. Hardware-in-the-loop framework for testing wireless v2x communication. In *2023 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6, 2023.

[2] ETSI. Etsi its-g5 communication standard for intelligent transport systems. *ETSI*, 1(1):1–25, 2020.

[3] Lorenzo Farina, Matteo Piccoli, Salvatore Iandolo, Antonio Solida, Carlo Augusto Grazia, Francesco Raviglione, Claudio Casetti, and Alessandro Bazzi. Low cost c-its stations using raspberry pi and the open source software oscar. In *2025 IEEE 101st Vehicular Technology Conference (VTC2025-Spring)*, pages 1–6. IEEE, 2025.

[4] Raspberry Pi Foundation. *Raspberry Pi 5: A Comprehensive Guide*. Raspberry Pi Press, Cambridge, UK, 1st edition, 2023.

[5] Raspberry Pi Foundation. *Raspberry Pi OS User Guide*, 2023.

[6] HKU-MARS. Fast_lio: A lidar-based localization tool, 2023.

[7] Daniel Jiang and Luca Delgrossi. Ieee 802.11 p: Towards an international standard for wireless access in vehicular environments. In *VTC Spring 2008-IEEE vehicular technology conference*, pages 2036–2040. IEEE, 2008.

[8] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[9] Steven Macenski, Alberto Soragna, Michael Carroll, and Zhenpeng Ge. Impact of ros 2 node composition in robotic systems. *IEEE Robotics and Autonomous Letters (RA-L)*, 2023.

[10] OScar Development Team. *OScar Framework: Open-Source ETSI C-ITS Stack*, 2024.

[11] FHL Technologies. Fhl-ld19 lidar for real-time mapping and localization, 2023.

[12] Livox Technology. Livox lidar sensor documentation, 2023.

[13] Edoardo Torrini. pyv2x: Python tools for etsi v2x message parsing and handling. `https://github.com/EdoardoTorrini/pyv2x/`, 2025. GitHub repository.

[14] UniMoRe. sixpackd. `https://git.wl.ing.unimore.it/272251/sixpackd`, 2025. Git repository.

[15] Vedder. Vesc project: High-performance electronic speed controllers with integrated imu, 2023.

[16] Giovanni Gambigliani Zoccoli, Francesco Pollicino, Dario Stabili, and Mirco Marchetti. Sixpack v2: enhancing sixpack to avoid last generation misbehavior detectors in vanets. In *2022 IEEE 21st International Symposium on Network Computing and Applications (NCA)*, volume 21, pages 243–249. IEEE, 2022.