SmartData on Wheels

Authors

- José Luis Conradi Hoffmann -> *hoffmann@lisha.ufsc.br*
- Leonardo Passig Horstmann -> *horstmann@lisha.ufsc.br*

Table of contents

1. Motivation

Recently, many works have investigated the usage of AI on Autonomous Vehicles and other critical systems {1 - 5}. However, the adoption of AI mechanisms and autonomous control can generate security weaknesses for the critical system {6}, and such lack of security is a key factor hindering the adoption of such AI mechanisms on critical systems {7}.

The advances in Machine Learning algorithms lead to more accurate and capable models, enabling more complex autonomous operation, as is the case of autonomous vehicles. The AI revolution we are living in is mostly driven by Data {8-11}, not by tasks. Thus, data is what must be secured, exchanged, stored, and processed. In this sense, we believe that modeling Autonomous Vehicles as data-driven systems and encapsulating this data with security is a key step for the employment of autonomous control on safety-critical systems. For instance, enabling real-time data communication with cloud services for integrity validation, double-check AI-module suggestions/simulation correctness, remote code update, and long-term storage.

This work aims at modeling a data-driven AV using SmartData {12}, while providing security in data communication amongst the vehicle controller (actuator, such as ECU) and the AI-module, to secure the vehicle data and AI-module when connecting this vehicle to the cloud, with the development and integration of a secure communication protocol with the aim to enhance trustfulness on the application of AI to critical-systems.

2. Goals

- Review Simulators and available published models to set a baseline of relevant AV model simulations for further experiments in the future
- Develop and integrate a Security communication protocol with an AI-based AV model in CARLA Simulator
- Evaluate the performance overhead added by the addition of the security protocol
- Use the results to write a paper describing the data modeling and the security protocol capabilities and showing the overhead evaluation

3. Strategy

- Select and validate Simulation tool;
- Select and validate the chosen AV model;
- Describe the architecture of the Project;
- Separate the AI-module and vehicle controller from the AV control scripts, Isolating the AI-module process both as a local service and as a Workflow on the IoT Platform under a project-specific domain;
- Model the AV-simulation data requirements using SmartData format;
- Adapt the vehicle controller to generate SmartData with the information collected from the vehicle and communicate with the IoT Platform and the local AI module;
- Adapt vehicle controller to retrieve command actions from both AI modules and compare the local module with its equivalent on the Cloud.

4. Simulation Tool - CARLA

CARLA is an urban driving simulator that provides an evaluation scenario for the proposed security protocol in a dynamic urban environment with traffic. CARLA is an open-source simulator implemented using Unreal Engine 4. For the experiments conducted in this paper, we will use a model that is built over a benchmark that includes autonomous driving simulations using two professionally designed towns with buildings, vegetation, and traffic signs, as well as vehicular and pedestrian traffic. The following subsection describes CARLA validation experiments.

4.1. CARLA - Tool validation

To validate the chosen tool, we configured a docker running CARLA on a server in order to allow running complex simulations that require high computation resources.

4.1.1. Docker Setup

This subsection describes the setup of the CARLA simulator server on a docker.

Setting up the environment (only if docker-ce and nvidia-docker2 are not yet installed)

To set up a docker environment for a CARLA simulator, first follow the installation steps described in the CARLA tutorial to set up docker-ce and nvidia-docker2.

To test the environment setup, run the following tests:

I) **Validating Nvidia driver installation:** `nvidia-smi`
This command is expected to print the nvidia video card info.

II) **Validating Nvidia Container installation:** `nvidia-container-cli info`
This command is expected to print NVRM and CUDA versions, along with some of the video card info.

III) **Validating NVIDIA Container Toolkit with Docker installation:**
`docker run privileged gpus all --rm nvidia/cuda:11.1-base nvidia-smi`
This shall output a similar result to the nvidia-smi one.
Troubleshooting: I have run into a permission problem regarding */dev/dma_heap/* when running on Fedora 33. To solve it, I have runned the following commands

```
$ sudo setenforce 0 $ sudo chcon system_u:object_r:device_t:s0 /dev/dma_heap/ $ sudo setenforce 1
Source: https://bugzilla.redhat.com/show_bug.cgi?id=1966158
```

With a validated installation, get the CARLA docker files:

```
docker pull carlasim/carla:latest
```

To run the CARLA docker, run the following line:

```
docker run --privileged --rm --gpus all -it --net=host -e SDL_VIDEODRIVER=offscreen
carlasim/carla:latest /bin/bash ./CarlaUE4.sh -nosound -opengl
```

This command is based on the one presented in the official CARLA Dockerfile

This command will **start the Carla server** within the **host network** and **offscreen configuration**. In off-screen mode, Unreal Engine is working as usual, rendering is computed. Simply, there is no display available. GPU sensors return data when off-screen.

CARLA - Docker Setup in portainer
The documentation of CARLA Docker usage in the Portainer.io platform in SETIC is available in the moodle forums.

5. AV Model

In this section, we describe the model chosen for this experiment, its main components, the communication protocols that could be potentially applied, and a feasibility validation.

5.1. End-to-End Driving Via Conditional Imitation Learning

DOI: 10.1109/ICRA.2018.8460487. **Citations**: 111
CARLA: https://github.com/carla-simulator/imitation-learning

The authors propose to condition imitation learning on high-level command input. At test time, the learned driving policy functions as a chauffeur that handles sensorimotor coordination but continues to respond to navigational commands.

Monitored Sensors and Types
Steer, float; Gas, float; Brake, float; Hand Brake, boolean; Reverse Gear, boolean; Steer Noise, float; Gas Noise, float; Brake Noise, float; Position X, float; Position Y, float; Speed, float; Collision Other, float; Collision Pedestrian, float; Collision Car, float; Opposite Lane Inter, float; Sidewalk Intersect, float; Acceleration X, float; Acceleration Y, float; Acceleration Z, float; Platform time, float; Game Time, float; Orientation X, float; Orientation Y, float; Orientation Z, float; High level command, int (2 Follow lane, 3 Left, 4 Right, 5 Straight); Noise, Boolean (If the noise, perturbation, is activated, (Not Used)); Camera (Which camera was used); Angle (The yaw angle for this camera)

- The RGB images stored at 200x88 resolution

5.2. Key Simulation Components

- **CARLA**: Holds simulation
- **Python Client**: Holds Machine Learning Control of the Vehicle

5.2.1. How do Components Interact

In standard (non-conditional) imitation learning, the action a is predicted from the observation o and the measurement m. In the goal-conditional variant, the controller is additionally provided with a vector pointing to the goal, in the car’s coordinate system.

5.3. Applicable Communication Protocols

No communication protocol was identified for this model. A possible approach would be to use a first Python-Client to collect data from the simulation and act as a network node. This node would transform data as necessary (to include security and necessary metadata) and use OMNeT to communicate with the AI module. In this sense, OMNeT would interface data transport between sensors and control unit. The control unit output (action) is communicated via OMNeT to the actuators. OMNeT, for instance, can be seen in such approach as a CAN bus. Cybersecurity risks would include node impersonation, for instance, and, in case the OMNeT implements communication as a WSN protocol, then, typical attacks such as Man-in-the-Middle and DDoS would also be considered as security threats.

5.4. Feasibility Validation

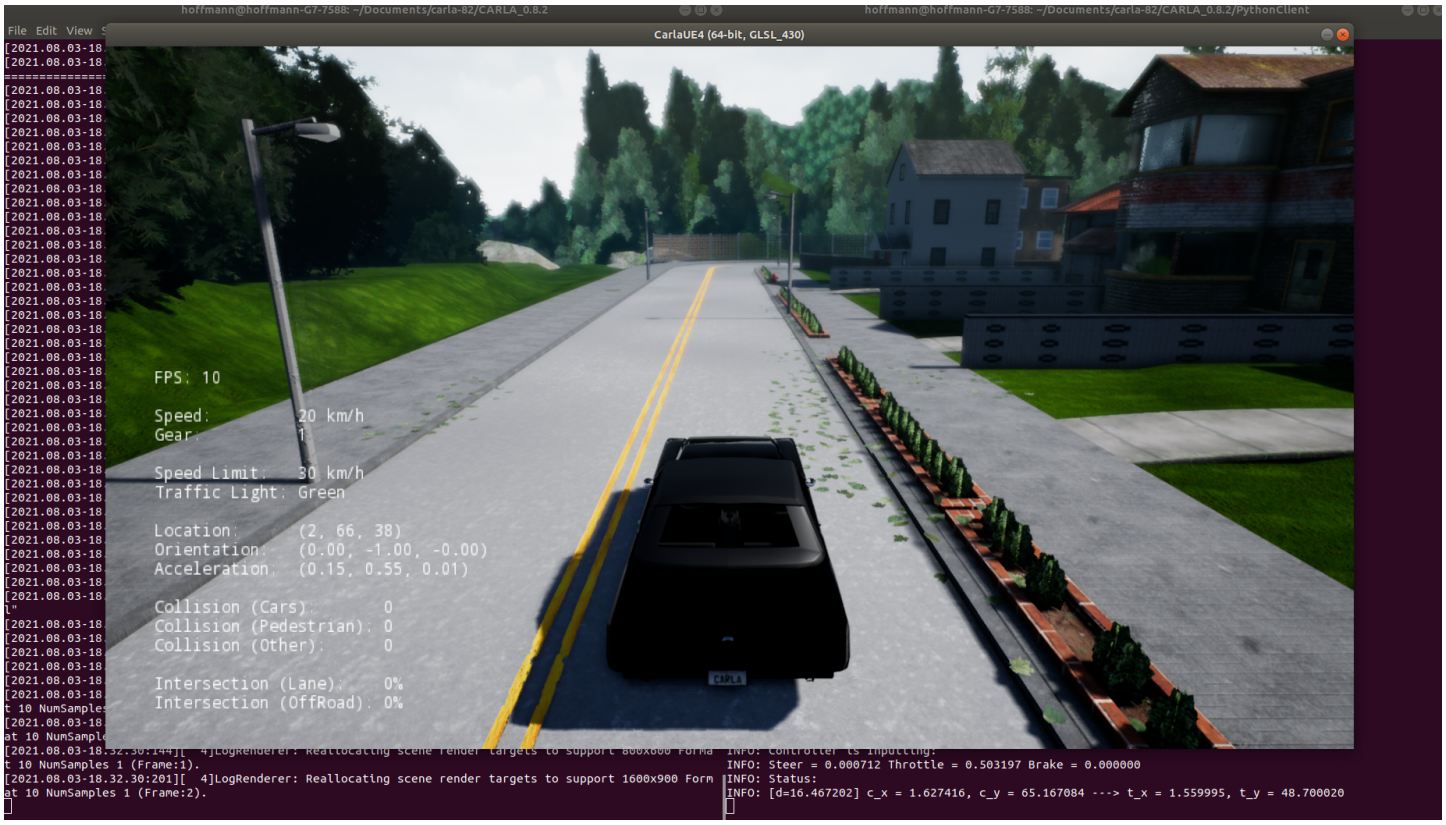
This section presents the environment setup and system requirements to run the chosen model.

5.4.1. Environment setup and Requirements

1. Download Carla 0.8.2 precompiled version:

https://github.com/carla-simulator/carla/releases/tag/0.8.2/

2. Install Python3.6 and install tensorflow-gpu, pygame, numpy, Pillow, scipy, carla
3. Initialize the CARLA server: ./CarlaUE4.sh /Game/Maps/Town01 -windowed -world-port=2000 -benchmark -fps=10 -nosound -opengl
4. Test the server with the default benchmark: python3 driving_benchmark_example.py --corl-2017
5. Stop the benchmark and server
6. Download Imitation learning paper source code: https://github.com/carla-simulator/imitation-learning
7. Copy the root of the source code into CARLA PythonClient folder
8. Run the client: python3 run_CIL.py
    1. If you find problems regarding non-existent functions on tensorflow module, make sure you are using tensorflow in versions early 2.0.
    2. If you find errors trying to use PIL, remove PIL and Pillow installations from /user/lib/python3/dist-packages/ and reinstall it using python3.6 -m pip install pillow
    3. If you find some error regarding the GPU device on imitation learning code, look for the tf.device(<device_name_here>) call, and replace the default device with the one outputted by the error.
        1. If you find errors regarding scipy.misc, add import scipy.misc at the beginning of the respective file.
        2. If now the error is pointing to “-scipy.misc has no attribute ”imresize“”, downgrade your scipy by running: pip install scipy==1.2.2



5.4.1.1. Using LISHA's CARLA Server

LISHA's CARLA Server Details:

- IP: 150.162.7.30
- Ports: 2000 to 2003
- CARLA Version: 0.9.11

To connect a client to the CARLA Server you need to inform the respective IP and Port when running the PythonClient in your Machine. All the python requirements previously listed for the local instalation of CARLA still applies when using the CARLA Server, since the PythonClient is a local side application that relies on CARLA definitions. Thus, a compatible CARLA's version is needed when connecting to the Server

(we recommend to use the **same version**). **Note that CARLA Version 0.9.11 requires on Python3.7 instead of Python3.6.**

- Example for Local client connecting to the remote server:

```
python3 manual_control.py --host=150.162.7.30 --port=2000
```

Disclaimer: The server provided for this discipline is configured with a newer version of CARLA, which does not support the driving benchmark that is used for the simulations. Therefore, the simulation was conducted locally on our own machine.

6. Communication Subsystem and Autonomous Embedded System Architecture Description

6.1. AES Architecture

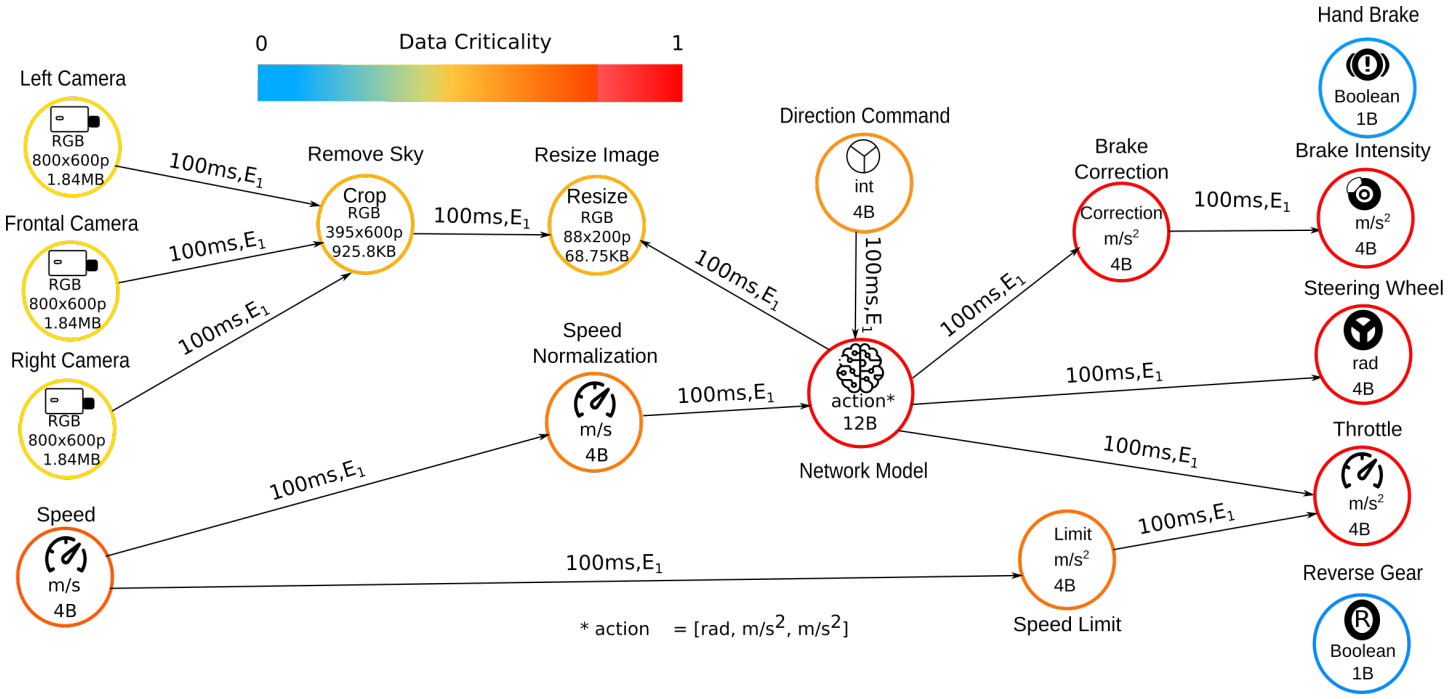
Actions *a* is a two-dimensional vector that collates steering angle *s* and acceleration *a*, *a* = <*s*,*a*>. An observation *o* is composed of an image i and a vector of measurements m (i.e., Speed and Direction), *o* = <*i*, *m*>. The AI Model is focused on correlating the observation and action pairs.

6.1.1. SmartData Model

The SmartData Model of this AV Model was divided into three layers of the SmartData, namely, Input, Transformational, and Output, as follows:

- **Input**
  - Image: RGB(800, 600) → Sensor SmartData
  - Speed: Float → Sensor SmartData
  - Direction (Control) → Stored SmartData: Digital (ENUM = 2 - Continue (Follow Lane); 3 - Left; 4 - Right; 5 - Straight), 1 or 5: branches{1}; 0 or 2: branches{1}; 3: branches{2}; 4: branches{3};
- **Transformational**
  - Remove Sky (no_sky): → Interest in: Image; tranformation: image.pixels{395:600}; image.
  - Resize Image (resized) → Interest in: no_sky, tranformation: resize image to RGB(88,200)
  - Speed Normalization (speed_norm) → Interest in Speed; transformation: 1/max | max = 25m/s
  - Network Model → Interest in: resized, speed_norm, and Direction; transformation = Action
  - Speed Limit → Interest in: Action, Speed; tranformation: speed > 10.0 and brake == 0.0 → acc = 0
  - Brake correction → Interest in: Action; transformation: brake < 0.1 or acc > brake → brake = 0
- **Output**
  - F = 10 fps
  - Control → Interest in Brake Correction, Speed Limit, Action, Hand Brake = 0, Reverse = 0

A graphical representation of the dependency relations and timing is presented below.



It is important to remind that there is only a single camera device. The image that is captured by this camera is cropped accordingly and can simulate the image from a left, right or frontal camera.

6.1.2. AES architecture Slides

The slides from the presentation can be found here.

6.2. Communication Subsystem

The infrastructure of the selected model consists of a default communication between CARLA server and the Client API (Python Client), in which the CARLA server runs the driving benchmark based on CoRL2017 experiment suit, including tasks with straight navigation, one turn, and others. The code of the Client consists of starting the communication with the server, where the imitation learning is conducted in a town where the vehicle controller acts as a navigator that in each step reads the sensors provided by CARLA and computes an action. The synchronous simulation is stoped at each "step" and the controller computes the set of actions to be taken prior to the next "step".

The idea for our experiment, as described early, relies on intercepting the controller at the moment the data is read by modifying the Python Client code, transforming it on SmartData, and communicating both with a Cloud AI module and the local AI module, which will be separated from the original controller. For such communication to be performed following a security protocol to be proposed, a bootstrap is necessary between the local AI module and the controller, where the controller acts as a central node (e.g., gateway) that sends the read data to the AI module and receives the AI module response (i.e., the action) and actuates over the simulation. Simultaneously, the controller is expected to use certificates to communicate with the Cloud over an HTTPS session. The communication between the local AI module and the controller is expected to be encrypted and signed using the information shared during bootstrap.

A diagram with an overview of this idea is presented below.

6.3. AES Architecture Detailed Specification

6.3.1. Sensors and Actuators

According to CARLA Documentation on RGB câmeras (https://carla.readthedocs.io/en/0.9.12/ref_sensors/#rgb-camera), the package that carries a sensed image holds the following information:

Sensor Data attribute	Type	Size	Description
frame	int	4 bytes	Frame number when the measurement took place.
timestamp	double	8 bytes	Simulation time of the measurement in seconds since the beginning of the episode.
transform	carla.Transform	> 24 bytes	Location and rotation in world coordinates of the sensor at the time of the measurement. 3 floats for x, y, z and 3 floats for pitch, yaw, roll.
width	int	4 bytes	Image width in Pixels.
Height	int	4 bytes	Image height in Pixels.
fov	float	4 bytes	Horizontal field of view in degrees.
Raw_data	array of bytes	with 800x600 images, 1920000 bytes	Array of BGRA 32-bit pixels that compose the image.

Our AE model is composed of two classes of sensors, namely speedometers and RGB cameras. Thus, our protocol must be able to accommodate these two classes of data, simpler data like booleans, integers, and floats, and more complex data like images or lidar samples.

For the actuators, the AE model considers three actuators to control the main characteristics of the

vehicle, like the steering wheel direction, brake, and acceleration throttle. These components will be set as the output layer of the SmartData modeling and will be connected to the AI module using the proposed protocol.

The data that will be carried must follow the types of each Data. The forward speed is expected to be a 32-bits float. The direction is pre-stored data that is represented as a 32-bits integer. Regarding actuation data, according to CARLA libraries, the Steering Wheel direction, throttle (acceleration), and brake intensity are represented as 32-bits floats, while hand-brake and reverse are informed as booleans.

6.3.2. SmartData

Since the idea of the model is to enable a complete autonomous behavior that periodically assures the current vehicle condition, the whole communication scenario is configured in a time-trigger fashion, where the actuation timing requirements are propagated to the transformation and subsequently to the sensors in order to adjust their sampling and priorities in the communication.

From the standard SmartData model,
our protocol will only encompass the fields unit, value, timestamp, and dev.

SmartData type Unit designates the kind of data encapsulated in a SmartData object, either an SI Quantity or plain Digital Data. Its most significant bit (i.e. bit 31) determines the case: encoded SI Units have it set (i.e. field SI = 1), while Digital Data have it clear (i.e. field SI = 0). The SI Unit format (used for SI compliant units) will be used as proposed by Guto, and is depicted below:

SI Unit Format

Bit	**31**	**29**	**27**	**24**	**21**	**18**	**15**	**12**	**9**	**6**	**3**	**0**
	1	NUM	MOD	sr+4	rad+4	m+4	kg+4	s+4	A+4	K+4	mol+4	cd+4

Nevertheless, the Digital format only accounts for 16 bits to describe the data length in bytes, which will be extended to 24 bits in our protocol to accommodate bigger data. For instance, a Full HD image (1920x1080p) requires 22 bits to represent its size in bytes. In this sense, we reduced the Type field to 7 bits, which enables up to 128 combinations of type values. The final Digital Data format is depicted below:

Digital Data Format

Bit	**31**	**30**	**24**	**23**	**0**
	0	type		length	

Finally, dev in SmartData works as a disambiguation identifier for multiple SmartData with the same Unit. We model dev as a 32 bits integer that is used as the identifier of the SmartData Sensor/Actuator in the network, for instance, to tag the source of the data and the destination of control messages.

In this sense, a frame for SI compliant SmartData is composed of a 32 bits unit, a 64 bits value, and 64 bits timestamp, leading to a frame like the one presented below:

SmartData Frame	32 bits	32 bits	64 bits * N	64 bits
	dev	unit	value	timestamp

For a frame of SmartData for non-SI compliant data with size > 64 bits, the frame value field is extended to a 64 bits multiple. As for the unit, in this scenario, it is composed of a 32 bits unit, a 64 bits value, and 64 bits timestamp, leading to a frame like the one presented below:

The envisioned format for forwarding this frame is to use a binary representation of the frame. This will lead to a parsing algorithm that first splits the first 32 bits of the frame to identify the Unit configuration, and subsequently the frame size (e.g., 64 bits if SI, or the value in field length if Digital), and interpreting the frame value accordingly. Finally, the last 64 bits are interpreted as a 64 Bits integer to compose the timestamp.

6.3.2.1. SmartData Formats Definition

In this section, we define the values for the fixed fields of all the SmartData used in the experiment.

Image

Unit	0x021D4C00	(digital image, type = 2, length = 1920000 bytes)
Device	0	

Direction

Unit	0x01000001	(digital counter, type = 1, integer 8 bits)
Device	1	

Forward-speed

Unit	0xC4963924	(m/s, float 32 bits)
Device	2	

Steer Angle

Unit	0xC4B24924	(rad, float 32 bits)
Device	3	

Acceleration/Throttle

Unit	0xC4962924	(m/s^2, float 32 bits)
Device	4	

Break Intensity

Unit	0xC496A924	(Newtons = N, float 32 bits)
Device	5	

Hand-Break

Unit	0x00000001	(boolean, type = 0, 8 bits)
Device	6	

No Sky Image

Unit	0x020E7720	(digital image, type = 2, length = 1920000 bytes)
Device	7	

Resized Image

Unit	0x020044C0	(digital image, type = 2, length = 1920000 bytes)
Device	8	

The data will not be accompanied by metadata like localization (x, y, and z) at the internal communication. A standard configuration will be used by the Gateway in order to send such data to the IoT Platform, configuring a stationary SmartData. The two main reasons why not to carry such information is to reduce the amount of data shared between critical components and the fact that the ML model does not use such information for the decision making process.

6.3.3. High-level Communication Protocol Overview

The communication between components in this experiment is modeled through two Industrial Ethernet Networks that will use the EtherCAT Protocol.

The first network interconnects the sensors and actuators at Carla Simulator and the System Controller. Through this network, sensors will send the necessary data to the Controller and the actuators will receive from the Controller the actions that are calculated for the input that was given.

The second network interconnects the Controller to the ML module and the Gateway that sends data to the cloud for long-term storage and ML services. The Gateway communicates with the Cloud using HTTPS over TCP/IP.

7. Implementation

7.1. First steps into implementation

The first step to implement the separation between the Python Client that is connected to the CARLA simulation and the server that runs the ML mechanism is to intercept the data before it is sent to the AI-

module, convert it to a SmartData frame, and send it to the AI-Module, which will be able to parse the binary format back into the raw data format.

The point of interest here is the run_step function of the CARLA Imitation Learning Agent. This is the function that is invoked when CARLA takes a step and expects the control commands for the next one. The following code presents the run_step function logic in this initial version.

```
def run_step(self, measurements, sensor_data, directions, target): speed = measurements.player_measurements.forward_speed directions = directions image = sensor_data['CameraRGB'].data frame_s = self.create_SmartData_Frame(int(speed*100), 0, "Forward-speed") frame_d = self.create_SmartData_Frame(int(directions), 0, "Direction") with open("smartdata_log", "a") as f: f.write(frame_s + "\n") f.write(frame_d + "\n") speed = float(self.parse_frame(frame_s)[0]/100.) directions = float(self.parse_frame(frame_d)[0][0])
```

Here, we intercept the data and use the create_SmartData_Frame to create the respective frame. For the sake of simplicity, in this first version, we consider only integers for the measurements data, and thus, the speed float is converted to an integer with a precision of 10 decimal values. The inverse is implemented on the frame reception, which converts it back to a float. Below are the codes implemented for the Create and Parse functions.

```
import pickle import codecs def create_SmartData(data, ts, data_type): units = {"Image" : 0x021D4C00, "Direction" : 0x01000001, "Forward-speed" : 0xC4963924} devices = {"Image" : 0, "Direction" : 1, "Forward-speed" : 2} prefix_32 = '08x' prefix_64 = '016x' frame = format(devices[data_type], prefix_32) print(frame) frame += format(units[data_type], prefix_32) print(frame) if (data_type == "Image"): frame += "\'" + codecs.encode(pickle.dumps(data), "hex").decode() + "\'" else: frame += format(data, prefix_64) print(frame) frame += format(ts, prefix_64) print(frame) print("Unit = ", units[data_type]) return frame def parse_frame(self, frame): units = {"Image" : 0x021D4C00, "Direction" : 0x01000001, "Forward-speed" : 0xC4963924} devices = {"Image" : 0, "Direction" : 1, "Forward-speed" : 2} dev_s = frame[0:8] dev = int("0x"+dev_s, base=16) unit_s = frame[8:16] unit = int("0x"+unit_s, base=16) sd_type = units.get(unit) if (unit & (1 << 31)): data = int("0x"+frame[16:32], base=16) size = 1 else: if (sd_type == "Image"): init_pos = 16 fin_pos = 18 while(True): if (frame[fin_pos] == "\'"): break fin_pos = fin_pos + 1 data = frame[init_pos + 1:fin_pos] data = pickle.loads(codecs.decode(data.encode(), "hex")) else: size = (unit & ~(0xff << 24)) if (size % 64 != 0): size = int(size/64) + 1 data = frame[16:16+size*16] data_array = [] for i in range(0, len(data), 16): data_array.append(int("0x"+data[i:i+16], base=16)) data = data_array ts = int(frame[len(frame)-16:len(frame)], base=16) return data, ts
```

Finally, we also log the data for further tests, an example of the frame conversion and the parsing results is presented below:

Frame	Parsing
00000002c4963924000000000000000000000000000000c8	SD = Forward-speed (c4963924), with Dev = 2 Value = 0 Time Stamp= 200
00000001010000010000000000000002000000000000000c8	SD = Direction (01000001), with Dev = 1 Value = 2 Time Stamp= 200
00000000021d4c00'8003636e756d70...'00000000000000c8	SD = Image (021d4c00), with Dev = 0, Value = [[0 13 0] ...] ...] Time Stamp= 200

00000002c4963924000000000003a50000000000002c88	SD = Forward-speed (c4963924), with Dev = 2 Value = 933 Time Stamp= 11400
0000000101000001000000000000000000000000000002c88	SD = Direction (01000001), with Dev = 1 Value = 0 Time Stamp= 11400
00000000021d4c00'8003636e756d70...'0000000000002c88	SD = Image (021d4c00), with Dev = 0, Value = [[0 13 0] ...] ...] Time Stamp= 11400
00000002c4963924000000000003a30000000000002cec	SD = Forward-speed (c4963924), with Dev = 2 Value = 931 Time Stamp= 11500
0000000101000001000000000000000000000000000002cec	SD = Direction (01000001), with Dev = 1 Value = 0 Time Stamp= 11500
00000000021d4c00'8003636e756d70...'0000000000002cec	SD = Image (021d4c00), with Dev = 0, Value = [[0 13 0] ...] ...] Time Stamp= 11500

7.2. Separation of the ML module

The second step of the implementation was to properly separate the ML module and the Python Client. Python Client must present an Interface with the method run_step, which is the method called by the CARLA simulation control. As the communication amongst system parts is not developed at this point, we set the ML module script to be called by this method. The idea is that in the future the run_step method will share the received data with SmartData Daemons that will convert it and send it through the communication channel so that the server can receive this data and trigger the ML module.

To do so, the construction of the Python Client included the creation of an object of the ML module.

□□□□□□□□

from imitation_learning_ml_module import ML_Runner class ImitationLearning(Agent): def __init__(self, city_name, avoid_stopping, memory_fraction=0.25, image_cut=[115, 510]): ... self._ml_runner = ML_Runner() ...

The ML_Runner constructor replicates the constructor of the Python Client in terms of the variables that are necessary to execute the ML model.
Finally, the method `_run_ml()` of the ML_Runner that is called by the run_step method implements the decision process using the data that is shared through the buffer in the format of SmartData frames.

7.3. UDP Multicast Test

The next step of implementation is to test the execution of the UDP multicast. To do so, a Python script was created to configure a Multicast UDP Socket. The code is as follows.

□□□□□□□□

import socket import struct import os MCAST_GRP = '224.1.1.1' MCAST_PORT = 5007 IS_ALL_GROUPS = True CPP_PROGRAM = 'udp_socket' def create_udp_mc_socket(): sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP) sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) sock.bind((MCAST_GRP, MCAST_PORT)) mreq = struct.pack("4sl", socket.inet_aton(MCAST_GRP), socket.INADDR_ANY) sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq) return sock socket_mc = create_udp_mc_socket() while True: print(str(socket_mc.recv(10240)))

The while at the end of the socket creation is placed in order to keep a listener on the Multicast and verify whenever a message is received. This process will be included on the Python Client in order to allow the further communications that will be used. To test the UDP multicast, another Python script was created to allow us to send data to the socket.

```
import socket MCAST_GRP = '224.1.1.1' MCAST_PORT = 5007 # regarding socket.IP_MULTICAST_TTL # ---------- # for all packets sent, after two hops on the network the packet will not # be re-sent/broadcast (see https://www.tldp.org/HOWTO/Multicast-HOWTO-6.html) MULTICAST_TTL = 2 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP) sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, MULTICAST_TTL) while(True): # For Python 3, change next line to 'sock.sendto(b"robot", ...' to avoid the # "bytes-like object is required" msg (https://stackoverflow.com/a/42612820) a = input("Type to send:") sock.sendto(bytearray(a, "utf8"), (MCAST_GRP, MCAST_PORT))
```

The tests demonstrated that the UDP multicast was functional, so the next step was to set up the UDP multicast socket communication in C++, once the SmartData module is developed in this language. The following code depicts the tests produced in C++.

```
#include <arpa/inet.h> #include <array> #include <iostream> #include <sys/socket.h> #include <netinet/in.h> #include <sys/types.h> #include <unistd.h> #include <chrono> #include <ctime> static const char* MC_ADDR = "224.1.1.1"; static const unsigned int PORT = 5007; static const unsigned int MSG_MAX_SIZE = 1024; static const bool RECEIVE = 1; static const bool SEND = 0; bool _last_state = 0; sockaddr_in create_socket_group(const char * addr, unsigned int port) { sockaddr_in group_socket = {}; group_socket.sin_family = AF_INET; group_socket.sin_addr.s_addr = inet_addr(addr); group_socket.sin_port = htons(port); return group_socket; } unsigned int get_char_size(const char * buf) { unsigned int i = 0; for(; buf[i]; i++); return i; } int create_udp_socket(bool state) { static int socket_udp = 0; if (_last_state != state) { socket_udp = 0; } if (!socket_udp) { socket_udp = socket(AF_INET, SOCK_DGRAM, 0); } return socket_udp; } int send(const char * buf, sockaddr_in group) { int multicast_udp_s = create_udp_socket(SEND); if (multicast_udp_s < 0) { std::cout << "Failed to create socket, err=" << multicast_udp_s << std::endl; return -1; } _last_state = SEND; return sendto(multicast_udp_s, buf, get_char_size(buf), 0,(sockaddr*)&group, sizeof(group)); } int receive(char * buf, sockaddr_in group) { int multicast_udp_s = create_udp_socket(RECEIVE); if (multicast_udp_s < 0) { std::cout << "Failed to create socket, err=" << multicast_udp_s << std::endl; return -1; } if (_last_state != RECEIVE) { int reuse = 1; int result = setsockopt(multicast_udp_s, SOL_SOCKET, SO_REUSEADDR, (char*)&reuse, sizeof(reuse)); if (result < 0) { std::cout << "Failed to set socket opt!" << std::endl; return -1; } result = bind(multicast_udp_s, (sockaddr*)&group, sizeof(group)); if (result < 0) { std::cout << "Failed to bind!" << std::endl; return -1; } } _last_state = RECEIVE; fd_set read_set; struct timeval timeout; timeout.tv_sec = 5; // Time out after a minute timeout.tv_usec = 0; int r; while(true) { FD_ZERO(&read_set); FD_SET(multicast_udp_s, &read_set); r = select(multicast_udp_s + 1, &read_set, NULL, NULL, &timeout); if( r<0 ) { // Handle the error std::cout << "ERROR: " << r << " ! " << std::endl; } if( r==0 ) { // Timeout - handle that. You could try waiting again, close the socket... auto end = std::chrono::system_clock::now(); std::time_t end_time = std::chrono::system_clock::to_time_t(end); std::cout << "finished computation at " << std::ctime(&end_time) << "s\n"; std::cout << "No Data Yet: " << r << " ! " << std::endl; timeout.tv_sec += 5; } if( r>0 ) { // The socket is ready for reading - call read() on it. std::cout << "There is DATA: " << r << " ! " << std::endl; break; } } return read(multicast_udp_s, buf, MSG_MAX_SIZE); } void clean_buf(char * buf) { for(unsigned int i = 0; buf[i]; i++) buf[i] = '\0'; } int main(int argc, char *argv[]) { sockaddr_in group_socket = create_socket_group(MC_ADDR, PORT);
```

```
const char* databuf = "Multicast from C++1111111"; send(databuf, group_socket); const char* databuf2 = "Multicast from C++2222222"; send(databuf2, group_socket); while(true) { char * recv_buf = reinterpret_cast<char*>(calloc(MSG_MAX_SIZE, sizeof(char))); for (unsigned int i = 0; i < 2; i++) { if (!receive(recv_buf, group_socket)) { break; } std::cout << "Message from multicast sender: " << recv_buf << std::endl; clean_buf(recv_buf); } // send(databuf, group_socket); } return 0; }
```

The multicast communication used in the validation tests will be reused for the sake of the implementation of the SmartData module.

7.4. SmartData @ Linux

We have successfully implemented SmartData communication over a Multicast UDP channel in Linux. The communication design follows the idea presented in the figure depicted in section 6.2. Our implementation encompassed the UDP channel setup and socket configuration, and the Periodic Threads handling, alongside some compilation issues previously presented in the code. The UDP channel setup includes the creation of two logical sockets that connect to the same UDP channel, one bound for read operations and one used for write operations. A receiver pthread is set to handle data readings in the UDP channel, while the main task creates the local/proxies SmartData accordingly. The logical UDPNIC implementation also required fixing some issues on message sizes and buffers copying. The current version of the code is available at Google Drive online link.
In this way, the logical order for the setup of the communication is depicted in the code below. First, the multicast channel is started by a python code that will be integrated with the Python Client module that interacts with CARLA, and will further provide data through a shared memory environment for the Sensor SmartData.

Following, the AI daemon is started which will wait until data is sent by the gateway to perform a prediction. Then, the Sensor SmartData are created. They are set up as periodic SmartData that senses the CARLA input and advertises it through the UDP channel. After, the Gateway is set up. It creates proxies for the Sensors SmartData and periodic SmartData advertisement for the Transformational data that will be of interest for the Actuators. Finally, the Actuator SmartData are initialized as proxies for the SmartData advertised by the Gateway. They are responsible to send the data through a shared memory environment to the actuation module set at the Python Client.

□□□□□□□□

```
echo "Initializing UDP Multicast Port" echo -e "\t Entering Directory" cd mc_udp_validation && pwd gnome-terminal -- /bin/sh -c "python3 create_udp.py" & echo -e "\t Leaving Directory" pwd cd .. echo "Cleaning files" echo "Cleaning ai buffer" echo '' > data/ai_buffer.txt echo "Cleaning gw buffer" echo '' > data/gw_buffer.txt sleep 1s echo "Intializing Python Daemon" echo -e "\t Entering Directory" gnome-terminal -- /bin/sh -c "cd PATH/CARLA_0.8.2/PythonClient && python3.6 ./agents/imitation/imitation_learning_ml_module.py" echo -e "\t Leaving Directory" pwd sleep 10s echo "Initializing Sensors & Actuators" echo -e "\t Entering Directory" cd smartdata-main_v2 && pwd #gnome-terminal -- /bin/sh -c "./smartdata 0" & echo -e "\n\t Starting Sensors - Nodes 1 2 \n" gnome-terminal -- /bin/sh -c "./smartdata 1" & sleep 1s gnome-terminal -- /bin/sh -c "./smartdata 2" & sleep 1s echo -e "\n\t Starting Gateway \n" gnome-terminal -- /bin/sh -c "./smartdata sink 1 2 3 4 5" & #0 sleep 1s echo -e "\n\t Starting Actuators - Nodes 3 4 5 \n" gnome-terminal -- /bin/sh -c "./smartdata 3" & sleep 1s gnome-terminal -- /bin/sh -c "./smartdata 4" & sleep 1s gnome-terminal -- /bin/sh -c "./smartdata 5" &
```

Sensors, Actuators, UDP Multicast, ML Module, and Gateway Initialization
Initialization Script

7.5. Python Client Communication with Sensors and Actuators

Sensors use the method `sense` to obtain the values that compose the SmartData exchanged during communication. In our implementation, the Python Client communicates with CARLA simulation and

provides the implementation for the method `run_step`, that writes the received information on buffers (files) that are accessed by the sensors whenever the `sense` method is executed through their periodic Updater thread. In this way, sensors are created for each one of the variables that compose an ML model input and communicate their sensing to the gateway.
Actuators, on the other hand, receive the messages with the action selected by the AI at the gateway through the network and write them on their respective buffers (files) so that the Python Client can read the values from such buffers and return CARLA simulation the Control Package holding data for Throttle (acceleration), Steer Angle, Brake Intensity, Reverse and Handbrake. Actuators are created for each one of the variables that compose the Control Package. For the sake of simplicity, in this implementation, we do not consider reverse and handbrake as actuators as their value is always “false”.

7.6. Gateway Communication with the ML Module

Similar to the communication between the Python Client and the nodes on the network, the ML module and the gateway communicate through a shared buffer. The gateway receives the ML parameters from the Sensor SmartData and passes it to the ML module by writing them to a buffer. The ML module processes information and computes the respective control parameters, returning them to the gateway, so they can be sent as Transformational SmartData to the actuators through the network.
As data is converted by the SmartData@Linux implementation, only the binary conversion of the image is necessary for the data being exchanged. The remaining variables are passed to the Python Client and the ML module as integers to be converted to floats with N-precision defined at implementation.

8. Security Protocol

The design of critical applications, such as autonomous vehicles, must address several security challenges to prevent exposing the infrastructure and data to attacks, especially when considering vehicles interacting with external agents connected to the Internet. Thus, guaranteeing data confidentiality, devices authenticity, integrity, availability, and tamper resistance is a key step to increase the reliability and safety of such applications.

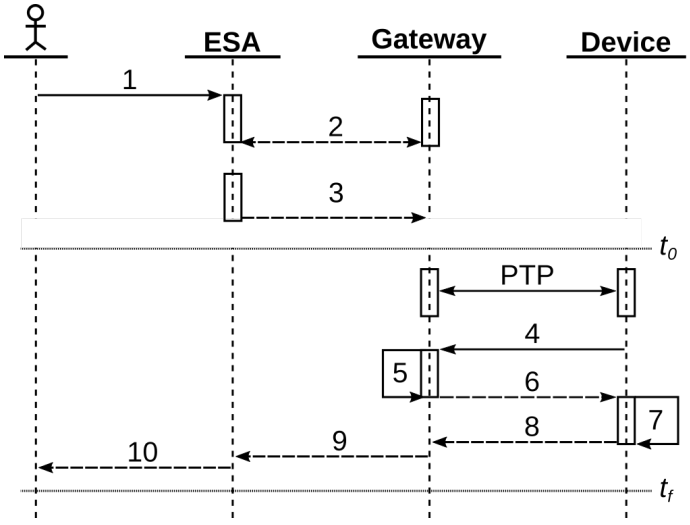
In this section, we propose a secure communication protocol for Autonomous Vehicles. The proposed solution relies on the unclonable property of PUF to establish a security token to be used as an authentication token. New device activation is triggered by an authenticated user informing the ESA a device's PUF results to a previously accorded challenge as an authentication token. The ESA is then responsible to communicate this information to the target AV internal gateway through a secure channel. The trust between AV's gateways and the ESA is built using mutual authentication through a CA. Devices and gateway, on the other hand, authenticate each other by encrypting the first message exchanged with the authentication token, which, at this moment is only known by the ESA, the trusted gateway, and to the very-own device, which is able to calculate the PUF result at run-time. After the authentication, the gateway shares the current group key with the new device. This is done by encrypting the group key using the authentication token of the device. In this way, the proposed solution avoids the necessity of establishing any kind of secure channel between the gateway and a new device (e.g., the usage of ECDH) while ensuring the trustfulness between device and gateway. The solution also promotes an IT-free secure and authenticated operation after device registration (i.e., does not require to request authentication checks from the ESA or a CA).

8.1. Protocol Requirements

The proposed security solution is based on four entities, namely, the user, the ESA, the device, and the gateway. To enable security, the proposed protocol assumes that devices are equipped with a PUF, which results are only known by the device, by running the PUF for a specific input at run-time, and by the entity requesting the activation of this device during deployment. Every device has an associated UUID stored in its memory. Devices in our Network design can interact with each other using any kind of network, wired or wireless. Nevertheless, devices are never directly connected to the Internet.

The proposed network architecture encompasses a gateway as an entity placed inside the system and directly connected to the network. The gateway is responsible for intermediate the necessary communication with the external world, thus, being the entry point to enable secure and trusted devices activation. In this sense, our protocol assumes that a trusted External Security Agent intermediates any activation requests from authenticated users.

8.2. Devices Bootstrap



1 - Activation Request(CY_g, t_0, t_f, $PUF(UUID_d)$, $PUF(\overline{UUID_d})$)
2 - HTTPS Session Establishment
3 - Activation Token(t_0, t_f, $PUF(UUID_d)$)
4 - Auth Request($AES(PUF(UUID_d),M)$)
5 - Save($PUF(UUID_d)$)
6 - Auth Granted($AES(PUF(UUID_d),K_G)$)
7 - Save(K_G)
8 - Active($AES(PUF(\overline{UUID_d}), PUF(UUID_d))$)
9 - Active($AES(PUF(\overline{UUID_d}), PUF(UUID_d))$)
10 - Active($AES(PUF(\overline{UUID_d}), PUF(UUID_d))$)

Overview of the proposed solution

Devices in a critical system usually communicate using protocols that are specifically designed for the particular application domain. These protocols either do not address security or do so using light symmetric algorithms. The base security of our solution comes from the unclonable property of PUF. By using PUF as the Root of Trust between devices and the gateway, we avoid the necessity of creating a secure channel between the gateway and the devices from scratch, and the security flaws introduced by the usage of previously stored keys into the devices. An overview of the proposed bootstrapping procedure is depicted in the Figure above.

Before any device bootstrap, the gateway starts its operation by generating a random key *KG* to be used as the group key of the network. A device bootstrap starts with an activation request (arrow 1 in Figure), using the following format:
`Activation\_Request(CYg, t0, tf, Auth_d, ~Authd})`
where CYg is the public certificate of the target gateway, t0 and tf are timestamps that describe the valid time-frame for the device to perform its bootstrap, and Authd is the result of the device's PUF for the very own device UUIDd:
`Auth_d = PUF(UUID_d)`
Finally, ~Authd is the result of the device's PUF for the complement of its UUIDd.
`~Authd = PUF(~UUIDd)`
This request is sent by an authenticated user to the ESA using a secure interface.

In possession of this information, the ESA identifies the target gateway using the public certificate and establishes a secure communication channel through an HTTPS Session Establishment. In this procedure, the ESA and the gateway exchange their public certificates and are mutually authenticated using a CA. Every forthcoming communication between the ESA and the gateway is done through the established HTTPS session. This process is depicted by arrow 2 in the Figure.

Using the secure channel, the ESA forwards to the Gateway the necessary information for the device bootstrap through an Activation Token message (arrow 3 in Figure). This procedure builds a trust chain from the authenticated user that issued the activation request and the target gateway, where only gateways trusted to the user have access to Authd. In this step, the gateway is aware of a new device activation and awaits an authentication request from the respective device within the time frame (t0,tf). A new device entering the network will first synchronize time with the gateway using PTP. Whenever the

gateway receives a new message to which it is unable to decrypt a valid message using the Group Key K_G (arrow 4 in Figure), if there is a know device activation pending, the gateway attempts to decrypt the message using each of the pending Authd as a key to the AES algorithm. On succeeding, the gateway acknowledges the sender as a true device, removing it from the pending list and storing the respective association of device and Authd in a secure memory for future usage in case of a possible reboot (see in the section below). The gateway then proceeds on forwarding the group key KG to the new authenticated device through an Auth_Granted message (arrow 6 in Figure). In this scenario, it is safe to assume that Authd is a suitable key to encrypt data using a symmetric algorithm, as it is only known to the very own device and the entities in the Trust Chain (User, ESA, and Gateway).

On receiving a new message after issuing an Auth_Request (arrow 6 in Figure), a device still performing bootstrap attempts to decrypt the message using Authd. On succeeding, the device acknowledges the Gateway as a trusted gateway and starts using the received KG as the group key of the network.

In case of failure in step 4, for instance, if the gateway did not find a valid message with any of the pending Authd or the time-frame expires, the message is discarded. Similarly, in step 6, the device discards any message in which a valid result is not found.

From this step forward, the gateway discards any new replay of the Auth_Request for already active devices· Moreover, as devices and gateway have synchronized time, both the gateway and the device are able to avoid replay attacks by discarding messages tagged with the same timestamp.

Finally, the device generates ~Authd as the PUF result for the complement of its UUIDd, a piece of unforgeable information that will be used to encrypt the message to be forwarded signaling the completion of the activation of the device to the user. This information is sent to the gateway using ~Authd as the AES key and Authd as the encrypted message to ensure it originated from the real device as an Active message (arrow 8 in Figure). The Gateway then forwards this same message to the ESA (arrow 9 in Figure), which forwards it to the respective user (arrow 10 in Figure).

8.3. Rebooting Devices

New Auth_Request(AES(Auth_d, M)) messages from devices that already concluded their bootstrap procedure are discarded for security reasons. When an active device reboots (e.g., due to a watchdog timer or a transient failure event), it is not able to perform a new bootstrap. Considering this procedure, during bootstrap, devices must save sensitive information (i.e., KG, represented in Figure by arrow 7) in a (secure) non-volatile memory. In this way, after a reboot, the device can restore its active state by loading the sensitive information from the secure memory, recovering the group key KG.

Similarly, the gateway also stores Authd (represented in Figure by arrow 5) of every active device in order to restore a valid state with knowledge of every active device in case of a reboot, thus, avoiding replay attacks of Auth_Requests from already active devices. Moreover, KG is also restored from the same secure non-volatile memory.

8.4. Discussion

As previously stated, the adoption of an embedded Digital Twin is fundamental to reduce latency, overhead, and avoid constant communications with the internet, which would open many attack opportunities for external entities. Even though, a security protocol is necessary in order to provide safety and security for the sensed and control data that are communicated within components due to the security-sensitive nature of the application.

In this work, we propose a protocol to enable secure and safe communication amongst AV's components and the embedded Digital Twin that avoids any kind of communication between OT devices with the internet after bootstrap while providing trustfulness between OT devices through their relation with the ESA.

The proposed protocol addresses several security weaknesses in order to provide a secure and safe environment for the Digital Twin. During the node authentication to the gateway, Man-in-the-Middle attacks are avoided both between the gateway and the ESA and between the gateway and the devices. At first, the gateway and the ESA establish a secure channel to communication by establishing an HTTPS session where they authenticate each other's certificates relying on a third-party CA. On the other hand, a Man-in-the-Middle attack is avoided during communication between devices and the gateway as no key establishment procedure is conducted. For instance, a typical solution to establish a secure communication channel in order to exchange the group key is to adopt an ECDH procedure. Notwithstanding, ECDH is vulnerable to Man-in-the-Middle attacks. The proposed solution relies on the security of the symmetric encryption algorithm and on the properties of the PUF to avoid procedures to create a secure channel, avoiding that a Man-in-the-Middle attacker could actuate over such communication once only a trusted gateway and the device itself would be able to know the authentication token Authd.

PUFs offer very strong protection of cryptographic keys {13}. In this work, the application of PUF avoids the establishment of a secure channel for the devices and the Digital to communicate with the gateway, however, it also avoids the usage of pre-stored keys, which is seen as a security flaw (note that UUIDd is stored, but Authd is computed as the PUF of this value. As presented in Section Devices Bootstrap.

Replay attacks are not properly avoided but their impact is quite controlled. During operation, the attack is mitigated by the fact that messages exchanged carry the timestamp they were generated, thus allowing to discard messages with timestamps that are older than the most recent message. Other than that, replay attacks with the Authentication Request will not last out of the specified registering time window.

Nodes impersonation are not possible, neither for the gateway nor for the devices. The former one would not authenticate to the ESA, which also prevents it from getting the authentication token Auth_d and finish bootstrapping. The later one would not be able to generate Authd and would not receive the group key KG. Additionally, bootstrapping procedure is not repeated after the device is authenticated to the gateway. Instead, the relevant information is saved when completing the authentication phase.

The proposed solution however does not encompass security against physical attacks and malicious gateways' related attacks such as eavesdropping. While such attacks are not possible in the devices, once they are not connected to the internet, the gateway can represent a single-point-of-failure and the main security weakness of the proposed protocol. A solution to encompass IIoT gateway security is proposed by Lucena et al. {14}, in which gateway integrity is periodically checked through a set of challenges that are issued by an ESA. The solution presented by the authors can be integrated into ours by using a secret key for the challenges the PUF of the complement of UUIDd, used on arrows 8, 9, and 10 of Figure. This would enable devices to communicate with the ESA in a security channel once the gateway is not able to produce PUF of UUIDd nor it receives such information during operation.

8.5. Implementation details - Device-Gateway Communication

This section holds the information on the device-gateway communication.

8.5.1.1. Device (sensor or actuator)

Once finishing its boot, the device initiates its network interface by calling `NIC::init()`, which initializes the **Receiver Thread**, and sets the primary value of the security key to the value of Authd: `sec_key = Authd = PUF(UUIDd)` .
After that, the device tries to obtain the value of `KG`:

```
while(sec_key == Authd): send(Response with data equal to -1u) delay(sync_period)
```

Meanwhile, the **Receiver Thread** waits for the controller message:

```
while(sec_key == Authd): receive(socket, msg) decrypt(sec_key, msg) if msg is valid: sec_key = msg.data} // now this device has KG
```

After obtaining `KG` the device starts its normal operation, using `KG` to send and receive data.

8.5.1.2. Controller

Once the controller boots, it initializes its network interface by running `NIC:init()` which generates `KG`. After that, the controller saves `KG` in the case of possible reboots.

Upon receiving from the ESA the message informing that a new device `dev` with 'Authd' authenticator will be inserted in the network in the time window expressed by `t0` and `t1`, the controller saves the `Authd` code to a list of devices, attaching `dev` as its id, and setting it as valid only in the said time window.

```
pending_list.insert(dev, Authd, t0, t1)
```

Whenever the controller receives a message, it first verifies whether this message is a response from the incoming device. To do so, the controller tries to decrypt the received message using Auth codes on the pending list. In case the message's device is successfully identified and the current time is inside the window defined by `t0` and `t1`, the controller inserts the device on a devices list and sends a message `m` as `AES(KG, Authd)`.

```
for d in pending_list decrypt(d.Authd, msg) if msg.data == -1u: if time.cur() > t0 && time.cur() < t1: devices.insert(d.dev, d.Authd) send(AES(KG,d.Authd)) pending_list.delete(d) break else: pending_list.delete(d) else: continue
```

In case any device registration succeeds, the controller follows its normal operation.

9. Fault Tolerance Aspects on the SmartData Model

The fault-tolerance aspects that were considered for this model were Redundancy and Verification. Redundancy is meant to be applied in case of interference or another fault that prevents a component to work by replacing the faulty component with the redundant one. Verification is meant to be applied to check for a component's integrity and operation's correctness in order to correct the results that outcome from its operation or trigger a replacement of the component.

SmartData Model with Redundancy and Verification Points

9.1. Redundant and Verified Components

Redundant components were adopted for each of the data sources for the Machine Learning (ML) decision-making process. Additionally, a redundant ML module was configured. The redundant components are listed below:

- A central camera that could be used to replace a faulty camera device: Images are submitted to a voting system that selects the image(s) that will be forwarded to the decision-making process. It is important to remind that there is a single camera device that captures an image that is next cropped

to represent one of the three possible images (Frontal, Left, and Right). Therefore, a central redundant camera should be enough to represent redundancy.
- Speed Sensor: The redundant speed sensor is also submitted to voting in order to define which sensed data will be forwarded.
- Direction Source: The redundant direction source will not be submitted to voting once it is only supposed to be used in case of failure of the primary direction data source.
- ML Safe Module: A safe module that is installed in a redundant component: A verification process is adopted for the results of the decision-making ML by comparing its results to the ones produced by the ML Safe Module. In this way, the redundant component is reused for run-time verification of the computed decisions.

10. References

{1} O. Kopuklu, N. Kose, A. Gunduz, and G. Rigoll, “Resource efficient 3d convolutional neural networks,” in 2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW), 2019, pp. 1910–1919.
{2} F. De Smedt, D. Hulens, and T. Goedem´e, “On-board real-time tracking of pedestrians on a uav,” in 2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2015, pp. 1–8.
{3} S. S. B.V. and A. Karthikeyan, “Computer vision based advanced driver assistance system algorithms with optimization techniques-a review,” in 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA), 2018, pp. 821–829.
{4} J. Athavale, A. Baldovin, R. Graefe, M. Paulitsch, and R. Rosales, “AI and reliability trends in safety-critical autonomous systems on ground and air,” in 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), 2020, pp. 74–77.
{5} M. T. Le, F. Diehl, T. Brunner, and A. Knol, “Uncertainty estimation for deep neural object detectors in safety-critical applications,” in 2018 21st International Conference on Intelligent Transportation Systems (ITSC),2018, pp. 3873–3878.
{6} European Union Agency for Cybersecurity (ENISA), “Cybersecurity Challenges in the Uptake of Artificial Intelligence in Autonomous Driving”, February 11th, 2021. Available at: https://www.enisa.europa.eu/publications/enisa-jrc-cybersecurity-challenges-in-the-uptake-of-artificial-intelligence-in-autonomous-driving/. Accessed in: August 3rd, 2021.
{7} Biondi, F. Nesti, G. Cicero, D. Casini, and G. Buttazzo, “A safe, secure, and predictable software architecture for deep learning in safety-critical systems,”IEEE Embedded Systems Letters, vol. 12, no. 3, pp.78–82, 2020.
{8} J. Zhang, F.-Y. Wang, K. Wang, W.-H. Lin, X. Xu, and C. Chen, “Data-driven intelligent transportation systems: A survey, ”IEEE Transactions on Intelligent Transportation Systems, vol. 12, no. 4, pp. 1624–1639,2011.
{9} H. Ye, L. Liang, G. Ye Li, J. Kim, L. Lu, and M. Wu, “Machine learning for vehicular networks: Recent advances and application examples, ”IEEE Vehicular Technology Magazine, vol. 13, no. 2, pp. 94–101, 2018.
{10} Jiang, S. Yin, and O. Kaynak, “Data-driven monitoring and safety control of industrial cyber-physical systems: Basics and beyond, ”IEEE Access, vol. 6, pp. 47 374–47 384, 2018.
{11} D. E. O’Leary, “Artificial intelligence and big data,”IEEE Intelligent Systems, vol. 28, no. 2, pp. 96–99, 2013.
{12} A. A. Frohlich, “SmartData: an IoT-ready API for sensor networks,” International Journal of Sensor Networks, vol. 28, no. 3, p. 202, 2018.Online. Available: https://doi.org/10.1504/ijsnet.2018.096264
{13} Michael S. Kirkpatrick and Sam Kerr. 2011. Enforcing physically restricted access control for remote data. In Proceedings of the first ACM conference on Data and application security and privacy (CODASPY '11). Association for Computing Machinery, New York, NY, USA, 203–212. DOI:https://doi.org/10.1145/1943513.1943540
{14} M. Lucena, R. M. Scheffel and A. A. Fröhlich, "IoT Gateway Integrity Checking Protocol," 2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC), 2019, pp. 1-8, doi: 10.1109/SBESC49506.2019.9046077.