PhD

in Mathematics and Informatics

XXXV Cycle

Thesis

“Industrial scholarship financed on study and design of an EDGE and cloud computing collaborative architecture for the security and privacy of intelligent transport systems”

Mr. Davide Antonino Vincenzo Micale

Tutor: Prof. Giampaolo Bella

Co-Tutor: Doc. Gianpiero Costantino

Co-Tutor: Doc. Ilaria Matteucci

Co-Tutor: Mr. Giuseppe Patanè

January 31, 2023

Contents

List of Figures

List of Tables

Listings

Abstract

The introduction of Information and Communication Technology (ICT) in transportation systems leads to several advantages like efficiency of transport, mobility, traffic management. Vehicles circulating on roads generate huge amount of data about both the driver and the vehicle itself. Such data can be used for different purposes, e.g., data generated may indicate the type of driving style or may be used to identify drivers. Hence, software in modern vehicles is becoming increasingly complex and subject to vulnerabilities that an intruder can exploit to alter the functionality of vehicles. Also, in the last decades attempts to characterize drivers' behaviour have been mostly targeted. Each driver has his/her own routine made of locations of most visited places. According to the GDPR, users' data are sensitive and should not disclosed out. For instance, the driver's most visited places could be used to identify the driver.

This thesis presents several solutions to improve cyber-security in modern vehicles. The first one is *Secure Routine*, a paradigm that uses driver's habits to driver identification and, in particular, to distinguish the vehicle's owner from other drivers. We evaluate Secure Routine in combination with other three existing research works based on machine learning techniques. Results are measured using well-known metrics and show that Secure Routine outperforms the compared works.

Then, we present Private Secure Routine (PSR) as a paradigm with two main goals: i) identify drivers depending on their habits/routine and ii) keep drivers' data private. We implemented PSR exploiting the secure Multi-Party Computation (MPC) technique against a honest-but-curious attacker model. Moreover, we evaluated PSR by establishing its accuracy in combination with other existing research works based on machine learning techniques. Evaluation of PSR is performed on different test-beds, considering single-owner and two-owners identification.

Next, we introduce CAHOOT, a novel context-aware Intrusion Detection System (IDS) capable of detecting potential intrusions in both human and autonomous driving modes. In CAHOOT, context information consists of data collected at run-time by vehicle's sensors and engine. Such information is used to determine drivers' habits and information related to the environment, like traffic conditions. To evaluate CAHOOT, we create and use a dataset by using a customised version of the MetaDrive simulator capable of collecting both human and Artificial Intelligence(AI) driving data. Then, we simulate several types of intrusions while driving: denial of service, spoofing and replay attacks. As a final step, we use the generated dataset to evaluate the CAHOOT algorithm by using several machine learning methods. The results show that CAHOOT is extremely reliable in detecting intrusions.

Finally, we present CAHOOTv2, that improves the accuracies of intrusion detection with respect to CAHOOT and is also trained on two additional attacks type. To validate the goodness of the paradigm, we also expanded the dataset with additional human drivings.

1 Introduction

Intelligent Transport System (ITS) indicates the use of information and communication technologies applied to road transportation [90]. ITS offers applications to users such as road safety, traffic efficiency and services. The European Telecommunications Standards Institute (ETSI) defines the applications in ITS environment [22].

In 2019, around 56k vehicles were targeted by thieves in UK [43]. It equates to one car stolen every 9 minutes and 45% of thefts occurred between midnight and 6 AM.

In 2020 New York City and Los Angeles have seen a soaring of car thefts [67]. In New York City around 7k of vehicles were stolen with an increment of 70% respect to the previous year. A similar trend was in Los Angeles where around 6k cars were stolen with an increment of 60% respect the previous year.

In 2022 Kia and Hyundai cars have been targeted by thieves in America [54]. Several videos have been uploaded on TikTok platform explaining how to steal cars of these two brands using a phone charger or a USB cable. In particular, Chicago has seen Kia and Hyundai car thefts increase of 767% compared to the previous year. Our opinion is that driver identification may help the detection of a theft.

Over the years, vehicles functionalities are managed by increasingly complex software. For instance, vehicles made by Volkswagen nowadays contain one hundred millions lines of code [17]. Level 5 autonomous vehicles will contain up to one billion lines of code [17] because all vehicles' functionalities will be electronically managed. Moreover, during the driving experience, a vehicle is able to collect a lot of information from its sensors, the Electronic Control Units (ECUs), and also from the environment. The driver can exploit the connectivity of the vehicle to read this information through OBD-II and a mobile connection, while the multimedia functionalities can be accessed via USB, disc, SD-card, Bluetooth and WiFi. The European Union Agency for Network and Information Security (ENISA) defines today's vehicles as smart cars, i.e., vehicles that offer enhanced users experience and safety, and provide connectivity and added-value features [1].

In the last decade, there are several papers in literature that present work on vehicle's attacks. The most famous one has been presented at the Black Hat USA 2015 by Miller and Valasek [20]. In particular, the two researchers were able to exploit the In-Vehicle Infotainment (IVI) of a Jeep Cherokee uploading and flashing a modified firmware [64]. This firmware allowed researchers to remotely control or disable many safety relevant systems of the target vehicle, including brakes, steering and the power unit. Vulnerabilities not only damage the reputation of car manufacturers but also their profits (the attack to the Jeep Cherokee forced the manufacturer Fiat-Chrysler to recall 1,4 million cars in the USA[6]).

1.1 Motivation

Modern vehicles can be considered as computer on wheels. The mechanical parts are often controlled by software components and communication protocols are in charge of exchanging data among vehicle's components. For this reason, modern vehicles can be classified as Cyber Physical Systems *(CPS)* in which used technologies bring countless advantages in terms of, for instance, efficiency of city operations and services. An example among all is the Internet connectivity.

Within this context, a problem of particular interest is how to leverage vehicular and/or smartphone data to characterize driver identification. Its characterization finds application in the development of software, which can be used by insurance companies to check and identify drivers or, for instance, to discourage auto theft.

Another very common and interesting scenario based on driver's identification is the one related to insurance and financial applications that are also presented by ETSI. Usually, insurance companies determine premium charges according to several statistical factors, e.g., male drivers tend to drive more aggressively than female [11]. Other risk factors are, e.g., age, installation of a theft deterrent system, driving behaviour [49][38]. Within this context, driver identification may provide premium charges based on who is actually driving the car.

Also, as for Personal Computer years ago, nowadays, guaranteeing the security of vehicles is becoming a strong requirement. The standard ISO/IEC 27039:2015 [39] and the regulation number 155 of the UNECE (UNECE R155), delivered in 2021, of the United Nations [68] prescribed the use of Intrusion Detection and Prevention Systems (IDPS) to monitor the vehicles from intrusions. In particular, an Intrusion Detection System (IDS) is able only to alert when intrusions are detected, while an Intrusion Prevention System (IPS) tries also to prevent the detected intrusions.

1.2 Contribution

This thesis contribute to the state of the art proposing two driver identification algorithms and two context-aware IDSs:

- Secure Routine, a driver identification algorithm. It exploits routines of the drivers to train a model using a machine learning algorithm
- Private Secure Routine, the first privacy-preserving algorithm that identifies drivers using neural networks models trained with the Secure Multi-Party Computation technique. Like in Secure Routine, Private Secure Routine exploits the routines of the drivers to improve identification accuracy
- CAHOOT is the first context-aware IDS that identifies spoofing, DoS and replay attacks. It is able to detect intrusions when the driver is a human or an Artificial Intelligence. The semantic of the CAN messages are used to train a model to detect intrusions on throttle, brake and steering.
- CAHOOTv2 is a context-aware IDS based on CAHOOT and trained to better recognize two spoofing attack variants and use an hyperparameter tuning technique to improve accuracies with respect to CAHOOT

1.3 Thesis organization

This thesis is organized as follows: Section 2 presents the state of the art, Section 3 introduces the fundamentals to understand the following sections. Section 4 presents Secure Routine, an algorithm for driver identification. Section 5 presents Private Secure Routine, an algorithm for driver identification in an ITS environment that preserves privacy of the drivers. Section 6 presents CAHOOT, a context-aware IDS that identifies several attacks. Section 7 presents CAHOOTv2, a context-aware IDS that improves CAHOOT on accuracy and attack types. Section 8 presents the conclusions of this thesis. Finally, Section 9 lists the publications made in the course of the doctoral program.

2 Related work

2.1 Driver identification

In literature, there are several solutions based on ML techniques for the identification of driver's behaviour. Bernardi et al. [7] used a Multi Layer Perception (MLP) to identify drivers. They used three datasets obtaining respectively 94%, 95% and 92% of accuracy. In particular, these results were obtained using a Start&Stop sliding window. A sliding window combines several consecutive instances in a single instance. Thus, Start&Stop joins instances starting when the car is moving until the car stops.

Gao et al. [26] discriminated drivers through Stop-and-Go events using a *voting strategy*. A Stop-and-Go event occurs when the car slowdowns until stops (stop phase), it stands still for five or more seconds and then speeds up (go phase).

Wang et al. [100] identified 30 drivers by using the voting strategy and Random Forest algorithm. Authors split data and tests into different window sizes. They use six sensor signals and three derived sensor's signals along with five statistical features. With 5 minutes of testing data this model achieves almost 93% of accuracy. With a sliding window of 5 seconds and 6 minutes of testing data they achieve 100% of accuracy.

Girma et al. in [28] used the Long Short-Term Memory (LSTM) algorithm with sliding windows and tested their model on [33] and [78] datasets with precision and recall of 98%.

Kwak et al. in [51] selected 15 features to identify drivers behaviour. For each feature they computed the mean, median and standard deviation according to a reference sliding window. Thus, the total number of features is 45. They used different ML algorithms and achieved the best accuracy of 99,6% applying Random Forest on [33] dataset.

Martinelli et al. in [60] tested several Decision Tree algorithms with the same dataset [33] using all 51 features. They obtained a precision and recall equal to 99,2% with J48. The same authors in [61] used only six features out of 51 features of [33] dataset. In this case, precision and recall decreased to 98,9% due to under-fitting.

Uvarov et al. [99] highlight the issue of car manufacturers that use non standard IDs of sensors' data of the CAN messages. It is not always possible to obtain the databases with the IDs information of each vehicle. Hence, authors verified how accurate can be driver identification models using only public sensors' data available with every OBD-II dongle. In the experiments, they use the dataset Θ and removed every feature not publicly available. Authors best result is 79% of accuracy using Random Forest in multi-driver identification, i.e. identify who is actually driving the car, whereas on the owner identification problem authors obtained 99% of accuracy.

Feng et al. [25] predict human mobility by using Federated Learning technique. Vehicles work together to create a model with the help of a server. Each vehicle customizes the model using a "personal adaptor" to better predict personal mobility patterns.

Other research works refer to driving style recognition. However, driving style recognition is different from driving identification: the former divides drivers into similar driving styles subgroups, the latter uses specifics driving styles singleness to uniquely identify drivers. Rizzo et al. [81] use secure Multi-Party Computation to distinguish aggressive from defensive behaviors in driving style. They consider a scenario in which an insurance company, first, builds a decision tree using labelled past vehicles' data. Then, the insurance company uses the model to classify driving style of a new driver. The built model and the unseen data are kept private.

Costantino et al. [14] propose a driver reputation characterization calculated in a privacy preserving way by using secure Multi-Party Computation. They collect vehicles' sensor data to calculate the *Reputation score*. The authors describe some example of ITS services that can be customized according to the reputation of the driver. The reputation score is calculated without machine learning.

Compared to Secure Routine and Private Secure Routine, [7], [26], [100], [51], [60], [61] and [99] do not look for frequency. Also, LSTM in [28] obtained lower scores in comparison with a Decision Tree (DT) algorithm ([60], [61], and [51]) on the same dataset. As shown by [61], certain features discriminate better than others for some drivers. Hence, Secure Routine and Private Secure Routine must use the best feature set for each driver. [7], [61] and [60] are the only ones that make owner-driver identification, they select the same feature set for all drivers. Also, Secure Routine and Private Secure Routine breaks down the timestamp in fine grained units to detect frequency in order to increase the accuracy. With respect to Private Secure Routine, none of the related work process data and models in a privacy preserving way. Finally, Private Secure Routine is the only algorithm capable of detect multiple authorized drivers of the same target vehicle.

2.2 Context-aware IDS

Jiang et al. [42] and Kondratiev et al. [48] developed an IDS that establishes valid messages based on the road-context for autonomous vehicles. In particular, authors combines CAN messages with images recorded from the camera to establish the validity of the messages. Public available datasets are used for the evaluation. The proposed IDSs are trained to recognize spoofing attacks variants on the steering wheel through two convolutional neural networks.

Wasicek et al. [101] uses a Bottleneck Neural Network to read sensors' values and to determine a vehicle's anomalous behaviour. The evaluation is made on a car with installed a chip tuning into the Engine Control Module (ECM) that changes its behaviour.

Casillo et al. [12] present a Bayesian Network to detect malicious CAN messages. The dataset used for the training is generated by the autonomous driving CARLA simulator [19] whose AI periodically receives attacks.

The methods for the detection of sequence context anomalies comprise different approaches. Rieke et al. [80] used process mining. Levi et al. [55] and Narayanan et al. [66] proposed works that use hidden Markov models. Theissler et al. [94] used OCSVM, while in the Kang et al. [45] work neural networks are used. Marchetti et al. [59] used detection of anomalous patterns in a transition matrix. Taylor et al. [91] and Kalutarage et al. [44] used frequency of appearance of a sequence of CAN messages.

Grimm et al. [31] provide a comprehensive survey on context-aware security approaches in the vehicular and related domains, while Al-Jarrah et al. [3] extensively survey the current state of the art on IDS systems for in-vehicle networks. Al-Jarrah et al. conclude that currently the area of context-aware systems is still under-investigated.

Compared to CAHOOT and CAHOOTv2, the related context-aware IDSs are able to detect only Spoofing attacks ([101] and [12]) or its variants ([42] and [48]). Also, several works are able to detect forged messages in steering ([42], [48] and [12]) while [101] detects intrusions only on the engine. The work [12] is able to detects intrusions also in brakes. None of these works are capable to detect intrusions on engine, steering and brake. Finally, none of these works are able to detects intrusions for human and AI drivings.

3 Background

In the following, we introduce some preliminary notions about the components present in the vehicles and how they communicate with other entities. Next, we explain the habits of the driver concept on which the works presented in this thesis are based. Then, we introduce several Machine Learning algorithms. Finally, we explain the communication layers present in the transportation infrastructure of this thesis.

3.1 Vehicle anatomy and connectivity

Vehicles are like computers on wheels. A lot of information circulate into vehicles and are communicated to the vehicles from outside. This is possible thanks to the amount of ICT systems that are installed in modern vehicles.

A car contain various sensors to keep track of the environment and the vehicle status [107]. In case of an autonomous car, these sensors are processed by Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), i.e., programmable components that integrate several basic logical electronic elements [18], to make driving decisions.

Inside the vehicle, there are also several Electrical Control Units (ECUs) that provide functionalities to the car [29].

Examples of ECUs are:

- The Engine Control Module (ECM) is responsible for the engine and manages the fuel supply, air management of the engine, fuel injection and ignition [8].

- The ECM manages the fuel while the air is managed by the Electronic Throttle Controller (ETC) [13].

- The Hydraulic Control Unit (HCU) receives the brake's pedal input and manages the braking systems of the four wheels, the Anti-lock Braking System (ABS) and the Electronic Stability Control (ESC) [106].

- The power steering systems handles the torque of the wheels based on the torque of the steering wheel. There are two types of power steering systems: the Hydraulic Power Steering and the Electric Power Steering (EPS). The first one uses an hydraulic system to torque the wheels whereas the EPS uses an electric motor that reduces the numbers of components needed to steering. Nowadays, the EPS is the most used power steering system. The EPS calculates the forces needed to torque the wheels based on the position and the force applied to the steering wheel and the velocity of the vehicle.

The ECUs are connected to each other through multiple buses, e.g., Controller Area Network (CAN), CAN-FD, FlexRay and Automotive Ethernet. Different partitions of these busses are connected each other through gateways.

The sensors' data can be accessed from the internal using the CAN bus protocol or from the external using an OBD-II device connected to the OBD-II diagnostic port [88].

Modern vehicle are also connected with extra vehicle entities, such us other vehicles or the Roadside Units of the infrastructure, through V2X communications. Using V2X communications, each vehicle is able to get information about the surrounding environment. These pieces of information may influence driving decisions, e.g., change route planning because of an accident.

Also, many newer vehicles are connected through LTE or 5G to the carmakers' server. The carmakers collect information of the car to offer services, e.g., sensors' data, air conditioning management, route planning and history, insurance premium charges, maintenance history and battery management for electrical vehicles. In particular, carmakers can offer to third party the access to the sensors' data.

3.2 Habits of driver

To complete a task, people repeat sequences of actions previously saw from others or done by themselves, no matter how tough the task is [52]. Two persons may accomplish the same task with similar actions but with little fundamental differences [52]. Routines can describe how people organize their lives: daily commute, weekly, meetings, holidays. Routines can also describe how a driver approaches to an intersection [4]. People routines are not fixed, they can evolve over time [52].

Based on the habits concept, *Routine based classification* is a type of classification [103] that aims to find actions that are frequently repeated in time.

3.3 Machine Learning

Machine Learning (ML) is the study of computer algorithms that improve automatically through experience. ML algorithms build a mathematical model to make predictions or decisions without being explicitly programmed to do so. At the basis of the model there is a dataset that has to be processed. Such dataset can be considered as a table in which all data are listed. Each row of the table is called *instance* and each column represents a *feature* of the instance. The dataset is usually split into two parts, the *training dataset* and the *test dataset.* The model is created on the basis of the training dataset. Instead, a *test dataset* represents all instances adopted to verify how much accurate our model is in doing the classification.

ML techniques are largely adopted for the identification and classification of users.

3.3.1 Decision Trees

In the following, we introduce an example of ML algorithm based on *Decision Tree (DT)* predictive modelling approach. A DT consists on a tree data structure that contains rules to classify the instance. For each level of the tree, the value of a feature of the instance is tested, for example, through a specific question. Each internal node of the tree contains a test. Depending on the answer, the model follows a different edge: the left edge if the result of the test is true, otherwise the right edge is followed. Finally, the leaf nodes, i.e., the nodes with no children, contain the prediction. A DT algorithm must create a tree with the minimum number of levels. This allows the ML algorithm to classify the instance as fast as possible. To build a DT with a low number of levels, it is necessary to select the best tests for the model. This is done by selecting the appropriate *Formula* to make the selection. A *Formula* specifies the criterion chosen to establish which is the next test to perform in the DT.

For instance, let *Alice* and *Bob* be two drivers that are used to going on the Sixth Avenue. Alice goes on the Sixth Avenue all days of the week, instead Bob goes only from Monday to Friday. Bob drives slightly faster than Alice, with a speed up to 55 Km/h. A possible DT model is the one in Figure 1(a) that is built by putting on the tree root the following test:

"*Is today Saturday or Sunday?*"

Following the root test, we have that the left child is taken by Alice instead the right child corresponds to the following test:

"*Is the vehicle speed lower than 55 Km/h?*"

Again the left child is a leaf node that represents Alice, whereas the right child is the leaf node representing Bob.

Despite the above DT model is a valid model for our example, we may produce a better tree in which a root node is configured with the following test (Figure 1(b)):

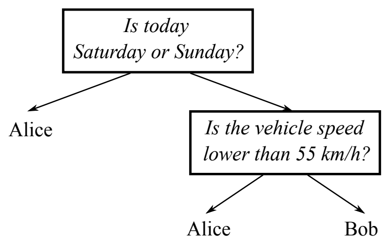
"*Is the vehicle speed lower than 55 Km/h?*"

In this case, the left child is the leaf node Alice and the right child is the leaf node Bob. Hence, a ML algorithm concludes its prediction with only one test.

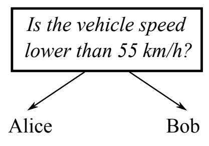
A DT has to be simple. This allows the DT to be flexible enough to represent also further instances. Thus, if the built model is too complex, it may not represent new labelled instances, i.e., for instance those ones present in a test-set. This may cause a high error rates, generating the *over-fitting* error. To reduce the over-fitting error, the *pruning* technique can be adopted to obtain a simpler version of the tree by pruning some nodes. Another solution to mitigate the over-fitting error is the *feature selection* that works by removing features. However, pruning too many nodes and removing too many features or relevant ones may lead to higher error rates, aka *under-fitting*.

Several DT algorithms were developed to generate models. The *C4.5* was proposed in 1993 [76] and it uses the Gain Ratio (GR) of a feature "X" of the training set (T) to establish which is the best test to perform.

$$GR = \frac{H(T) - H(T|X)}{H(X)}$$


Is today Saturday or Sunday?
Alice
Is the vehicle speed lower than 55 km/h?
Alice
Bob

(a) DT with two levels.



(b) DT with one level.

Figure 1: *Comparison of two possible DT for solving the same problem.*

where:

- $H(T)$ indicates the *entropy* of T, i.e., the quantity of information carried by the probability distribution of labels in T [85], calculated as:

  $$H(T) = -\sum_{j=1}^{k} \frac{freq(C_j, T)}{|T|} \times log_2 \left(\frac{freq(C_j, T)}{|T|} \right)$$

  where:

  - k is the number of classes;
  - $freq(C_j, T)$ is the number of instances in the j-th class;
  - $|T|$ is the number of instances of T.

- $H(T|X)$ indicates the entropy after partitioning T in "n" parts, where "n" is the number of possible values assumed by X:

  $$H(T|X) = \sum_{i=1}^{n} \frac{|T_i|}{|T|} \times H(T_i)$$

  where:

  - $|T_i|$ is the number of instances with the i-th value assumed by the feature X;
  - $H(T_i)$ indicates the entropy of the set of instances with the i-th value assumed by the feature X.

- $H(X)$ indicates the entropy of X:

  $$H(X) = -\sum_{i=1}^{n} \frac{|T_i|}{|T|} \times log_2 \left(\frac{|T_i|}{|T|} \right)$$

Note that C4.5 can handle features with unknown values and real numbers and may make use of the pruning technique.

Random Forest (RF) [15][10] is an algorithm formed by a set of DTs. Each tree is built from a random sampling with replacement of the training-set. Each node of a tree is the best test defined on a subset of features, instead of on all available ones. Trees are not pruned. In prediction phase, an instance is run on each tree and each tree makes a prediction. The most predicted value becomes the prediction of RF. Also, RF includes a procedure in case of unknown values in the dataset.

3.3.2 Neural Network at a glance

Neural Networks (NNs) [36] are computing systems that tries to resemble a human brain to perform a specific task. Neural Networks are composed of simple process units, referred as *neurons*, that resembles the human neurons to constitute a network. Each neuron can receive input information as weighted signal from other neurons of the network through links. The inputs of a neuron are combined with a function that combines the weighted inputs with an extra element called *bias.* Weights and bias are called *parameters.* The output of the neuron is the result of the *activation function* that limits the range of output values of the neuron.

The structure of a Neural Network is called *network architecture* and describes how the neurons are arranged and linked. Networks are organized in layers and can be listed by the number of layers in *single-layer network* and *multi-layer.* A single-layer network has an input layer and an output layer. The term "Single-layer" denotes that there is only one computational layer on the network. The input layer is not a computational layer because no computation is performed in these neurons. A multi-layer network has an input layer, an output layer and one or more intermediate layers known as *hidden layers.*

Hence, to define a NN architecture we need to define the number of layers, number of neurons of each layer, the links between neurons, the combiner function and the activation functions of the neurons. Then, the parameters of NN are established through a learning process [82] starting from the definition of a dataset. A dataset is a collection of data, represented by means of a table, that contains samples regarding the task. The rows of the table are called *instances* and the columns are the *features* of each instance. The dataset is split into two partitions: the *training dataset* and the *test dataset*. *Training dataset* represents all instances adopted to make experience and evolve the model. Instead, instances of a *test dataset* is employed to verify how much accurate the model is in doing the predictions. The learning phase follows the step described below:

1. The parameters of NN are randomly initialized;

2. Run the network presenting data of the training set as input;

3. Using a *loss function*, the output of the network, i.e., the predictions, are compared with the expected answer, i.e., the labels, obtaining the *loss* value. This indicates how bad the network is to make predictions.

4. The parameters are corrected according to the loss value by using an *optimizer* [104]. The correction is limited by a *learning rate* multiplier. The optimizer searches the best parameters that minimize the loss value.

5. The procedures from 2 to 4 are repeated depending on training preferences, for instance a specific training accuracy is reached.

To make the training faster, the training set is split in so called *batches*. Each optimization step is executed on the instances of a batch. Once the optimization step is applied to all training data, an *epoch* is completed. The epochs [96] are the number of times the training data is passed through the training process.

3.4 Intelligent Transportation System Infrastructure

We model the transportation infrastructure on three different layers that communicates one another (Figure 2).

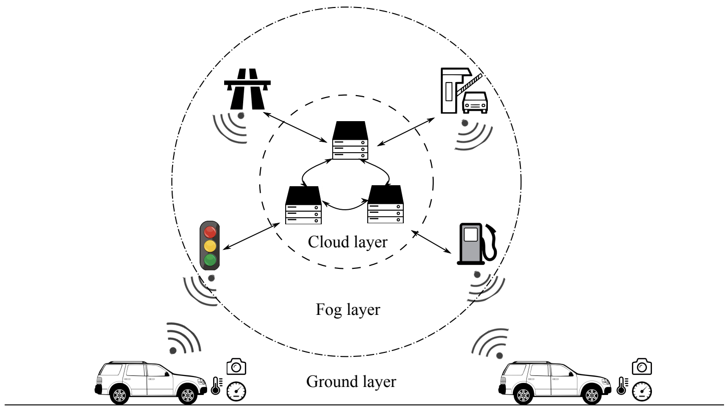


Figure 2: *Communication Layers in the transportation infrastructure*

The *Ground layer* is composed by *drivers* and *vehicles.* Modern vehicles are equipped with devices such as the *In-Vehicle Infotainment* (IVI) system able to collect, store and communicate information generated by car sensors. While a driver uses the vehicle, this one requests and stores pieces of information about the driver. Such data are collected via OBD-II [93] or the CAN bus [40] protocol, which transport the data related to vehicle's sensors. Using these data, a *dataset* is generated. It is represented as a table in which all instances referred to a driver are collected. An instance is made of timestamp recorded with the following template: (day, month, year, hour, minute, second and day of the week) and pieces of information about vehicle's components, e.g., oil engine temperature, throttle position, speed and so on.

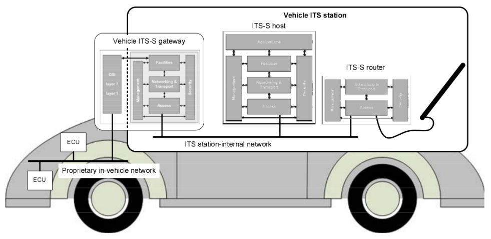
The *Fog layer* contains all the *Roadside Unit*s able to collect and process information coming from the ground layers. RUs are able to offload computational tasks from the ground levels, enabling the distribution of tasks among the fog layer devices. Because fog and ground layer are close, the network between these two layers are fast and present low latency.

The *Cloud layer* provides massive computational power and storage but the network speed is slower than the fog layer as well as the latency is much higher.

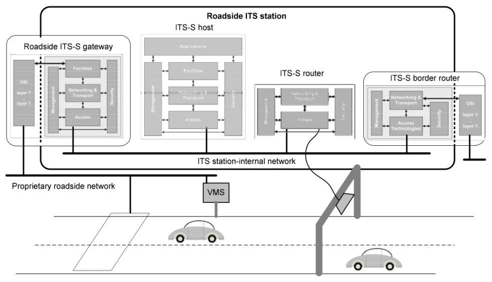
The Ground layer, the Fog layer and the Cloud layer correspond in the ITS environment to the vehicle ITS subsystem, the road ITS subsystem and the central ITS subsystem, respectively. Each subsystem has a station that allows it to communicate with other sub-systems of the ITS, called ITS station [23].

In each vehicle, an ITS station is installed (Figure 3(a)) . The gateway inside the station provides access to the vehicle internal network, e.g. the CAN bus [40]. In the majority of the network scenario of the ITS, vehicles can directly communicate only with nearby devices [24]. The Roadside Units (RUs) (Figure 3(b)) are computing devices located aside the road and provide long distance connectivity and internet connectivity to the vehicles acting as an intermediary. Finally, the Central ITS provides to the vehicles applications (Figure 3(c)), e.g., traffic information, SOS service, Diagnosis service and so on.

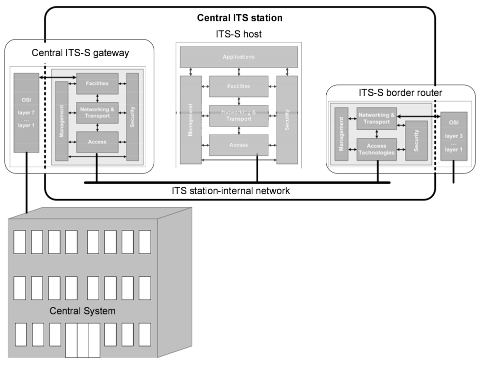
The three main standard technologies for short range communications among the stations are the IEEE 802.11p, the LTE-V2X and the 5G-V2X. The first two work at 5.9 GHz [2] while 5G-V2X also works at mmWave that ranges between 24 GHz and 100 GHz. The IEEE 802.11p is based on the Wi-Fi standard 802.11a [2] for communications vehicles to vehicles (V2V) and communications vehicles to RUs (V2I). The LTE-V2X [2] is based on the LTE, known also as 4G. LTE-V2X allows communications V2V, V2I, vehicle to pedestrian (V2P) and direct internet connection. The 5G-V2X [34] is similar to LTE-V2X, replacing LTE with 5G. The network latency is reduced and the network speed is higher.



(a) Vehicle ITS subsystem



(b) Roadside ITS subsystem



(c) Central ITS subsystem

Figure 3: *ITS subsystems.*

4 Secure Routine

Based on the aspects of routine introduced in the Subsection 3.2 , here we introduce the paradigm of Secure Routine *(SR)* that takes into account not only what the user does but also how much frequently. We use the SR paradigm within the automotive context with the aim to identify drivers. To achieve this, we elaborate and implement the SR algorithm that exploits sensors' car data, obtained, for instance, through the *OBD-II* diagnostic port. The SR algorithm, which runs in the cloud layer, evaluates the recorded data and, in particular, uses the timestamp to make an accurate classification of drivers. Then, SR leverages a Machine Learning (*ML*) technique to establish driver's routines and to properly identify the driver.

To test the goodness of the Secure Routine algorithm, we compare it with other research works in literature. The comparison is done on two different datasets and the results are evaluated using three metrics: *accuracy*, *precision* and *recall*.

4.1 Algorithm

We define SR and present its application into the automotive context to perform driver's behavioural identification. To this aim, SR analyses all tracking data recorded by vehicle's sensors while the user is driving it. Tracked data are organized in separate instances according to the sensor that collects them and the timestamps when the event occurs. Hence, SR firstly decomposes the timestamp of each instance and extracts seconds, minutes, hours, day of week, day, month and year. Then, SR removes less relevant features, as we will describe below using the Feature Selection *(FS)* technique. Successively, the data collected by sensors are correlated with the timestamp previously decomposed. Then, a ML algorithm examines these data. The output is a model representing users' Secure Routine. As final step, the obtained model is compared with an observed user's driver behaviour for his/her identification.

To show the value added by the Secure Routine to identify drivers, we introduce the following example. Let us consider Alice and Bob who are used to going on the Sixth Avenue. Alice usually goes there at 12PM, and Bob at 7PM. If we do not consider the timestamp information, the resulting model of Alice and Bob will contain only the information *"The user is used to going on the Sixth Avenue"*. In this situation, the observed behaviour will be compared to understand whether the driver is Alice or Bob. However, this selection is quite difficult since the missing timestamp information is fundamental to distinguish between the drivers. On the contrary, if we consider also the timestamp in which the event happens, the identification will be unique in this case. In fact, if the vehicle is at 7PM on the Sixth Avenue, therefore the driver is Bob.

This is what Secure Routine does considering daily routines as well as monthly and yearly ones. Hence, SR may be very useful, for instance, to mitigate scenarios as the one depicted in Section 1: in UK cars are often stolen at night. If the vehicle's owner does not usually drive during the night, SR can easily detect the weird behaviour. In particular, the SR paradigm is built upon a ML algorithm that uses as training-set the data recorded through an OBD-II device. A closer working mechanism of SR is presented in [86]. Here, the authors prefer to involve the interval between a rerun of the same action. Let us consider this other example in which Alice goes on the Sixth Avenue every 24 hours for the whole week, instead Bob every 24 hours from Monday to Friday. In this case, the routine of Bob will be modelled as intervals of 24 and 72 hours. So, if we consider a driver moving on Saturday, we would not be able to identify the driver, neither Alice nor Bob, since the interval is set to 24 hours. On the contrary, if the day of the week is taken into account, Alice will be correctly identified.

Let us consider a target vehicle belonging to a driver d. The SR algorithm acts in four phases.

4.1.1 Model Generation Dataset

Whenever a vehicle is used, its sensors register pieces of information about several *features*, e.g., the water temperature, the speed, the brake pressure, and so on. We assume to take trace of all these data in combination with the timestamp in which each instance of data is generated. Data are taken from the OBD-II port by using an OBD-II interface [21]. Each instance of data is called *interaction* of the driver d with the vehicle and it is denoted as $in_{i,d}$ where i is the timestamp. Interactions are composed by the timestamp, recorded with the following template: (day, month, year, hour, minute, second and day of the week) plus the others features obtained from the OBD-II.

4.1.2 FS paradigm

To mitigate the possible over-fitting error, we implement the *FSParadigm* (Listing 1).

Listing 1: *Feature Selection Paradigm*

```
function FSParadigm(instances)
    ranking $\leftarrow$ GR(instances)
    ranking$_{ordered}$ $\leftarrow$ order ranking ascending
    features$_{>0}$ $\leftarrow$ discard features with rank = 0 from ranking$_{ordered}$
    (features$_{no\_timestamp\_correlated}$, features$_{timestamp\_correlated}$) $\leftarrow$ features$_{>0}$
    ranking$_{no\_timestamp\_correlated}$ $\leftarrow$ ranking from ranking$_{ordered}$ of features
        present in features$_{no\_timestamp\_correlated}$
    average$_{ranking}$ $\leftarrow$ mean(ranking$_{no\_timestamp\_correlated}$)
    subset$_{no\_timestamp\_correlated}$ $\leftarrow$ discard features sum is less than or equal
        to the average$_{ranking}$ from ranking$_{no\_timestamp\_correlated}$
    subset ← subset$_{no\_timestamp\_correlated}$ ∪ features$_{timestamp\_correlated}$
    return subset
```

FSParadigm is designed to select the best features to use. It firstly ranks all features applying the *Gain Ratio* approach and then features are sorted in ascending order. Those features with rank equal to zero are discarded. Then, the average-rank among all features not correlated to the timestamp is calculated. The FS discards those features, except those related to time, whose rank sum is less than or equal to the average-rank.

4.1.3 Model Generation Algorithm

Let us consider that a vehicle may be driven by d but also by other people, e.g., friends or relatives of d. In the modelling phase, our algorithm (Figure 2) considers all the past interactions recorded by the vehicle and labels with 1 each interaction that belongs to d, 0 otherwise. The labelled interactions are sent to a DT algorithm that generates the model for the driver d.

Listing 2: *Secure Routine Model Generation*

```
1 function generate_model(d)
2     ins$_d$ ← get interactions from $db$ made by $d$, labeling $1$
3     ins$_o$ ← get interactions from $db$ made by others, labeling $0$
4     ins$_{all}$ ← ins$_d$ ∪ ins$_o$
5     subset ← FSParadigm(ins$_{all}$)
6     model ← MLAlgorithm(ins$_{all}$ with features from subset)
7     return model
```

In particular, in line 5, *FSParadigm* is the Feature Selection paradigm we described above as part of SR and line 6 (*MLAlgorithm*) indicates the ML algorithm in use with the subset of features obtained before.

4.1.4 SR Identification strategy

Once the model is generated, SR makes the identification evaluating each interaction. In particular, SR links an interaction to the vehicles' owner if the ML algorithm predicts and labels it as 1, otherwise 0.

4.2 Secure Routine Evaluation

We evaluated Secure Routine in two steps: first, we run it using two ML algorithms and we verified which of them best performs to identify drivers. Then, we compared Secure Routine with the following research works:

- Martinelli et al. [61] referred in the following as M.
- Kwak et al. [51] referred in the following as K.
- Girma et al. [28] referred in the following as G.

4.2.1 Datasets

We run the experiments using two datasets presented in [33], referred as Θ, and [5], referred as Ψ. The former is a dataset used also by M, K and G in their research works. So we can fairly make a comparison. However, the Θ dataset does not contain a fundamental feature used by SR, this is the *timestamp* of each represented instance. Nevertheless, Θ dataset contains the *engine runtime* that provides the minutes to be used as timestamp needed for SR to work.

On the other hand, Ψ dataset contains a timestamp for each instance by default. This feature allows Secure Routine to fully work by using all available pieces of information. In particular, SR expands the timestamp to generate all time dependent features. As far as we know, the other compared research works do not make use of this dataset to evaluate their proposal. So, to evaluate SR even in this case, we were able to re-run the work proposed by Martinelli et al. and calculate the results for the owner-driver identification. On the other side, the works K and G did not calculate the owner-driver identification and, also, it was not possible to re-run their algorithms since the implementation is not publicly available. In the specific case of G, the authors published only the pre-built model and we were not able to use it.

Dataset Θ. It contains data from 10 drivers. Figure 4(a) shows the driver instances' distribution. Drivers have 9438 instances on average: Driver 4 has the highest number of instances with 13244 samples while Driver 1 has the lowest number with 7240 instances. In addition, drivers drove two times in the same path in similar time-window. Dataset instances are recorded per second.

Dataset Ψ. This contains data from 14 drivers. Figure 4(b) shows that drivers' instances are not equally distributed. For example, Driver 1 has the highest number of instances with 13617 samples, whereas Driver 10 has the lowest number with only 7 instances. This may depend on the fact that some users drive frequently whereas other users rarely. However, 7 instances are not enough to build a model for the Driver 10. So, we decided to exclude Driver 10 instances in our experiments to not alter the final result. Also, many instances contain empty values because of errors on gathering data. Instances are recorded every 7 seconds.

Compared to the Θ dataset, Ψ contains by default 32 features. Nevertheless, five of these features are timestamp related and are *minute*, *hour*, *day of the week*, *month*, *year*. Other features, such as, *model*, *car_year*, are removed since they do not give any useful information about the user driving style. The dataset also contains *engine_runtime* from which we extract *engine_runtime_minute*. Also, these datasets have 9 features in common (Table

Table 1: *Common features description*

Feature	Description	Example
Throttle_pos	Percentage of throttle opening	25%
Short_term_fuel_trim_bank_1	Percentage of ratio air/fuel in the first bank of cylinders instantaneous	-3,00%
MAF	The air flow mass in the engine	8,12g/s
Engine_RPM	Number of revolutions per minute crankshaft makes	2100RPM
Speed	Speed of the vehicle	55km/h
Timing_advance	Percentage of crankshaft rotation when spark plug fires in advance	0,423%
Engine_coolant_temp	Temperature of the coolant/antifreeze liquid mix of engine	90C
engine_runtime_minutes	Minutes elapsed from engine start	39m
engine_runtime_second	Seconds elapsed from engine start	10s

1).

4.2.2 Metrics

To get a comparable result of SR with M, K and G, we evaluate *accuracy* [96], *precision* and *recall* [61].

- *Accuracy* represents how often the model is making a correct prediction. It is the ratio between the number of correct predictions and the number of predictions:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

where:

  - TP (True Positive) is the number of instances belonging to the vehicle's owner that are correctly predicted;

  - TN (True Negative) is the number of instances not belonging to the vehicle's owner that are correctly predicted;
  - FP (False Positive) is the number of instances belonging to another person but incorrectly predicted;
  - FN (False Negative) is the number of instances belonging to the vehicle's owner but incorrectly predicted.

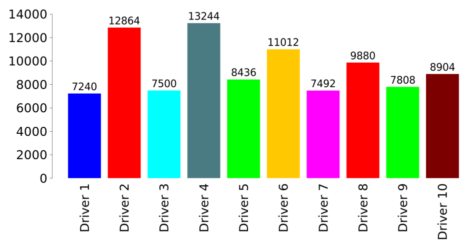
- *Precision* measures how often the predicted instances belonging to the vehicle's owner are true. It is calculated as the ratio between TP and $TP + FP$:
$$Precision = \frac{TP}{TP + FP} \tag{2}$$

- *Recall* identifies how often the instances belonging to vehicle's owner are correctly predicted. It is calculated as the ratio between TP and $TP + FN$:
$$Recall = \frac{TP}{TP + FN} \tag{3}$$

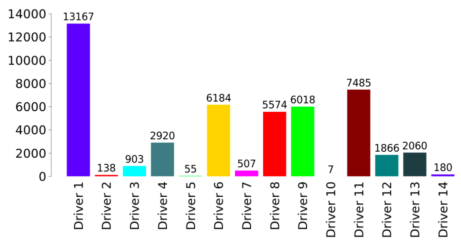
To better estimate the three metrics depicted above, in our experiments we used the 10-fold cross-validation [47] approach. First, we split the dataset on 10 equal size subsets D_1, D_2, ..., D_{10}. Each instance of the dataset is randomly inserted in a subset. Then, we constructed 10 training sets Tr_1, Tr_2, ..., Tr_{10} and 10 testing sets Te_1, ..., Te_{10}. Tr_i is made of all subsets except D_i and Te_i is made of D_i with $i \in \{1, 2, ..., 10\}$. For each pair (Tr_i, Te_i) is calculated $accuracy_i$, $precision_i$ and $recall_i$. Finally, we calculated the final value of $accuracy$, $precision$ and $recall$ as the mean of $accuracy_i$, $precision_i$ and $recall_i$, respectively.

4.2.3 Experiments

We performed four types of experiments to evaluate Secure Routine. The first experiment is related to multi-driver identification problem [61], i.e.,



(a) Number of instances for each driver in Θ



(b) Number of instances for each driver in Ψ

Figure 4: *Driver distributions on the datasets.*

Table 2: *Comparing SR using J48 and Random Forest over the multi-driver identification problem.*

J48				**Random Forest**			
All features		**Feature selection**		**All features**		**Feature selection**	
Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
99,2%	99,2%	99,3%	99,3%	99,3%	99,3%	99,6%	99,6%

properly identify who is the driver. However, as step zero, we decided to find the most suitable ML algorithm with the best features set to evaluate SR. We leverage on Weka [102] as software that contains a collection of visualization tools and algorithms for data analysis and predictive modelling. So, we used the available Gain Ratio method to rank each feature. Then, we employed *J48*, which is the implementation of the C4.5 algorithm, and RF algorithm over the driver identification.

Table 2 shows the results obtained comparing SR implemented into J48 and RF algorithms applied to the driver identification using Θ dataset. RF algorithm with feature selection (37 features) obtained the best precision and recall.

After selecting SR with RF and the most appropriate features ranked by the Gain Ratio method, we show the first experiment results obtained by comparing SR with the work in M, K and G on the Θ dataset. As shown in Table 3, Secure Routine and K achieves the best results. Note that M did not calculate the accuracy in the paper, so we established this value through the replication of their experiment. Instead, K did not provide on their research precision and recall. Finally, for G we were not able to retrieve the exact accuracy.

Table 3: *Comparison of Secure Routine with M, K and G for multi-driver identification on dataset Θ.*

Secure Routine		M		K		G	
Precision	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
99,6%	99,6%	99,2%	99,2%	N.A.	N.A.	98,8%	98,1%
Accuracy		**Accuracy**		**Accuracy**		**Accuracy**	
99,6%		99,2%		99,6%		N.A.	

As we can see in Table 4, SR achieves almost a perfect precision, i.e., 100%, but with the worst recall and this depends on the features selection. In fact, if we increment the number of features, we increase the recall but the precision is decreased. Here, we decided to obtain a higher precision selecting the most appropriate features using the Gain Ratio.

Table 4: *Comparison of Secure Routine with M for owner identification on dataset Θ.*

Secure Routine		M	
Avg. precision	**Avg. recall**	**Avg. precision**	**Avg. recall**
99,8%	98,5%	99,3%	99,3%

The second experiment is related to the *Owner Driver identification*, i.e., does the instance belong to the vehicle's owner? In this case, we compared SR only with M since K and G did not calculate the owner driver identification. As stated by the authors of M [61], they use the same features for both the multi-driver and owner driver identification.

The third experiment that we propose is related to the multi-driver identification on the Ψ dataset. In this case, we used the GR method for features selection. Starting from pruned Ψ dataset, we evaluated SR. As previously stated, we know that there are no other research works that use this dataset. So, we had only the possibility to replicate the best solution proposed by M.

Table 5 shows that Secure Routine with feature selection achieves the best result both for precision and recall.

Table 5: *Comparison of Secure Routine with M for multi-driver identification on dataset Ψ.*

Secure Routine		M	
Precision	**Recall**	**Precision**	**Recall**
99,4%	99,4%	90,4%	89,8%

To conclude the evaluation, last experiment focused on the owner driver identification. Table 6 indicates that SR with features selection has the best performance when compared with M. SR obtained an average precision of 99,6%, which means that for 8 drivers SR established a perfect precision whereas M achieved this precision only for 4 drivers with an average precision of 95,1%. Regarding the recall, SR largely outperformed M in percentage and SR achieved a perfect recall score for one driver, whereas M never obtained a perfect recall.

Table 6: *Comparison of Secure Routine with M for owner identification on dataset Ψ.*

Secure Routine		M	
Avg. precision	**Avg. recall**	**Avg. precision**	**Avg. recall**
99,6%	98,1%	95,1%	82,9%

5 Private Secure Routine

In the previous works similar to Secure Routine, identification of the drivers is obtained using sensors' data from the vehicle. These data are stored in servers that build Machine Learning models of driver's behaviour. Hence, the server stores a massive amount of sensible driver's data. However, the General Data Protection Regulation (GDPR) requires that the organisations operating in EU protect user's personal data.

This section presents Private Secure Routine (PSR) paradigm for driver identification that distinguishes authorized drivers of a car from the others in a privacy preserving way. PSR identifies drivers using cars' sensors data gathered, for instance, using OBD-II [93] interface or directly from the CAN bus [40] of the vehicle. Through the data sharing within an ITS architecture, PSR is able to build an accurate model of authorized drivers. Moreover, according to General Data Protection Regulation (GDPR) such data, can be considered sensitive data. Hence, PSR exploits the secure Multi-Party Computation (MPC) technique to guarantee that drivers' data are exchanged in a privacy preserving way. We experiment the Private Secure Routine paradigm on two different test-beds and the results are quite promising.

5.1 Privacy preserving Machine Learning techniques

5.1.1 Federated Learning

Federated Learning (FL) [62] is a learning technique to train shared models among users' devices (Figure 5). These devices are referred as clients. FL does not need to centrally store users' data, instead, data remains on clients. A central server coordinates the clients. Firstly, the server sends a model, called global model, to the clients. Each client trains the received global model using its data. The obtained model is then uploaded to the server. The server merges the received models to obtain the new global model and sends the model to the clients. An example of merge algorithm is the average of the model's parameters. This learning cycle is repeated several times. Users data remain private, however FL does not guarantee privacy of the models: the server sees the model updates of each client and all the clients have a copy of the global model.

In the driver identification, each vehicle needs a model able to detect a not authorized driver. In particular, the model should be able to identify and distinguish authorized from each other. Hence, the models of the vehicles are also sensible data that must be kept private. FL is vulnerable to several attacks that allows an attacker to get information on the training data used by the clients or even reconstructs the data used for the training [65].

5.1.2 Split Neural Network

In Split Neural Network (SplitNN) [32] the layers are distributed among multiple agents, each agent trains only its subset of layers. No agent has a complete view of the neural network. There are two types of agents: the data agents and the computing agents. The first ones are data sources, while the computing agents only perform computations (Figure 6). To preserve data privacy, each data agent has the first subset of the neural network, so it

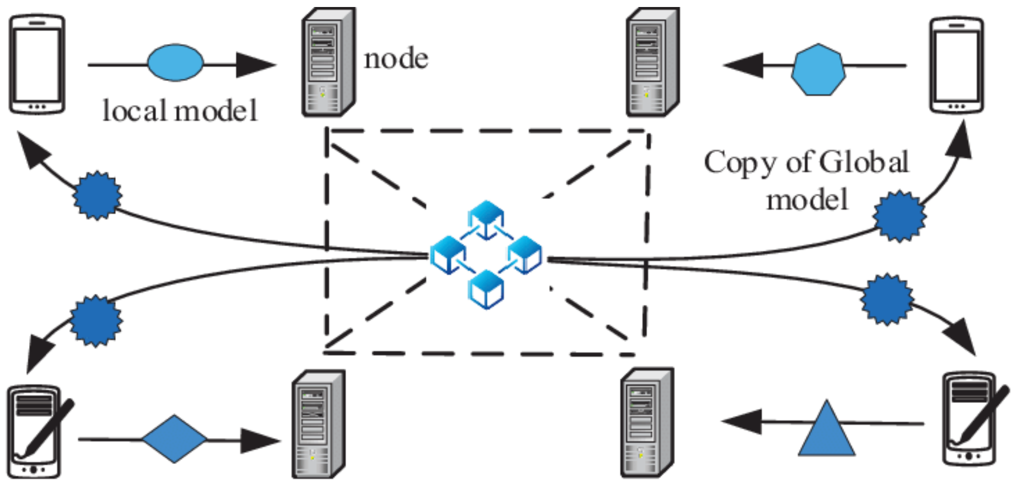


Figure 5: *Federated Learning.*

does not send its data to the other agents. The neural network training is distributed between the agents.

Training of a model using Split Neural Network. In the following, we will briefly explain the training process. Let us suppose that there are two data source agents and one computing agent: the first data source agent uses its data on the subset of layers in its possession applying the forward propagation. The output of its last layer is then sent to the computing agent in possession of the next subset of layers of the neural network. This agent applies the forward propagation in its layers' subset using in input the data received. Because the computing agent possesses the output layer, the output of agent's subset is the output of the neural network, i.e., the prediction made by the neural network for the input data of the first data source. To calculate the loss, the data source sends to the computing agent the labels. Once the computing agent calculates the loss, the backpropagation procedure starts. The computing agent computes the gradients of its layers' subset, then sends the gradients of the first layer of its subset and sends it to the first data source agent. This agent uses the received gradients to compute the backpropagation of the layers in its possession. The agents source and computing apply the corrections to their layers according to the gradients. Now the neural network is trained with the data of the first data source. To complete the neural network training, the first data source sends to the second one its layers' subset. The second data source and the computing agent applies the forward and backpropagation to complete the training of the network using the data of the second data source.

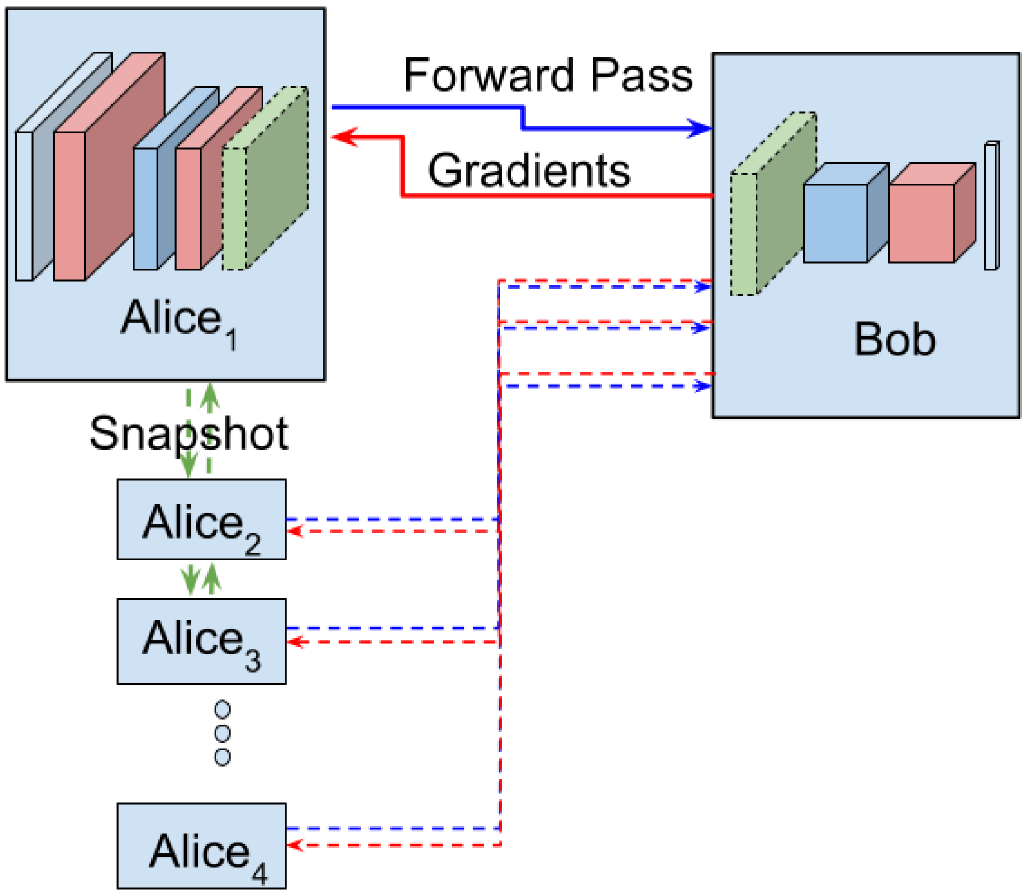


Figure 6: *Split Neural Network.*

5.1.3 SplitFedv1

SplitFedv1 [92] is a combination between the SplitNN and Federated Learning. Like in SplitNN, the layers of the network are split between the data source agents and the computing agents. However, in SplitFedv1 (Figure 7) multiple copies of the neural network are trained in parallel, one per data source agent. Each data source agent applies the forward propagation to a copy of the first layers' subset using its data. The agent sends the output of its subset to the computing source in possession of the next subset. This computing source has all the copies of the second subset and applies the forward propagation to each copy using the output of the corresponding data source agent. The procedure continues until the forward propagation is applied to all the copies of the last subset. Then, the loss and the gradients are calculated for each last subset's copy. Next the backward propagation is executed for each copy of each subset layer and their parameters are corrected according to their gradients. Finally, the trained neural network copies are merged like in federated learning. Each computing node merges each layer in its possession. Each data source agent sends its layers to an external server called "fed server"which computes the merge of all the layers received. Training in SplitFedv1 is faster than the training in SplitNN because the copies of the neural network can be trained in parallel.

5.1.4 SplitFedv2

SplitFedv2 [92] is a variant of SplitFedv1: each computing agent has only a single copy of its own subset copy instead of having a copy for each source agent. The outputs of the forward propagation are sent source agent by source agent to the first computing source. The training process is slower than the SplitFedv1 because the layers owned by the computing agents can not be trained in parallel (Figure 8).

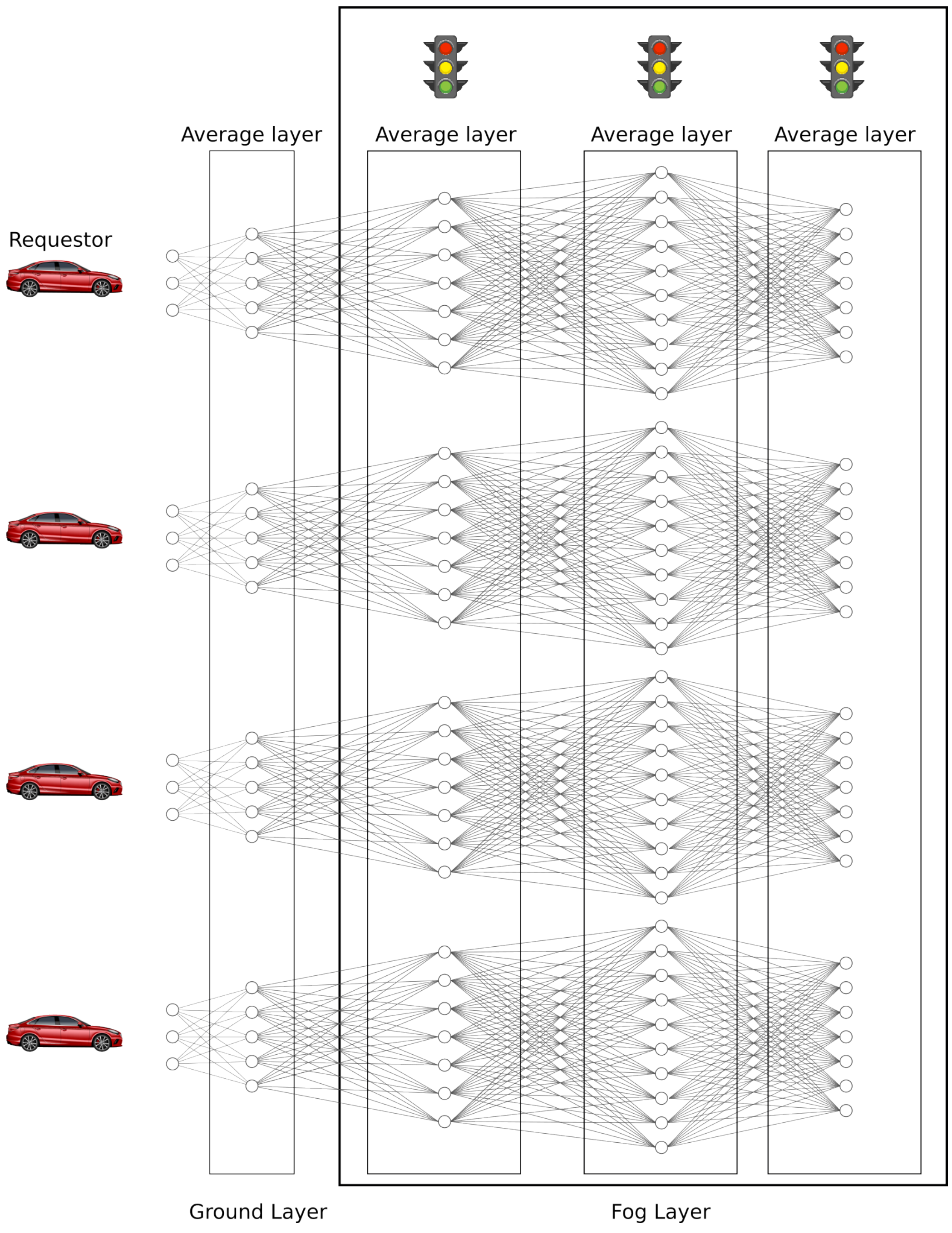


Figure 7: *SplitFedv1.*

5.1.5 SplitFedv3

Finally, SplitFedv3 [27] algorithm is almost identical to SplitFedv1. The only difference is in the merging phase: the "fed server" is not necessary

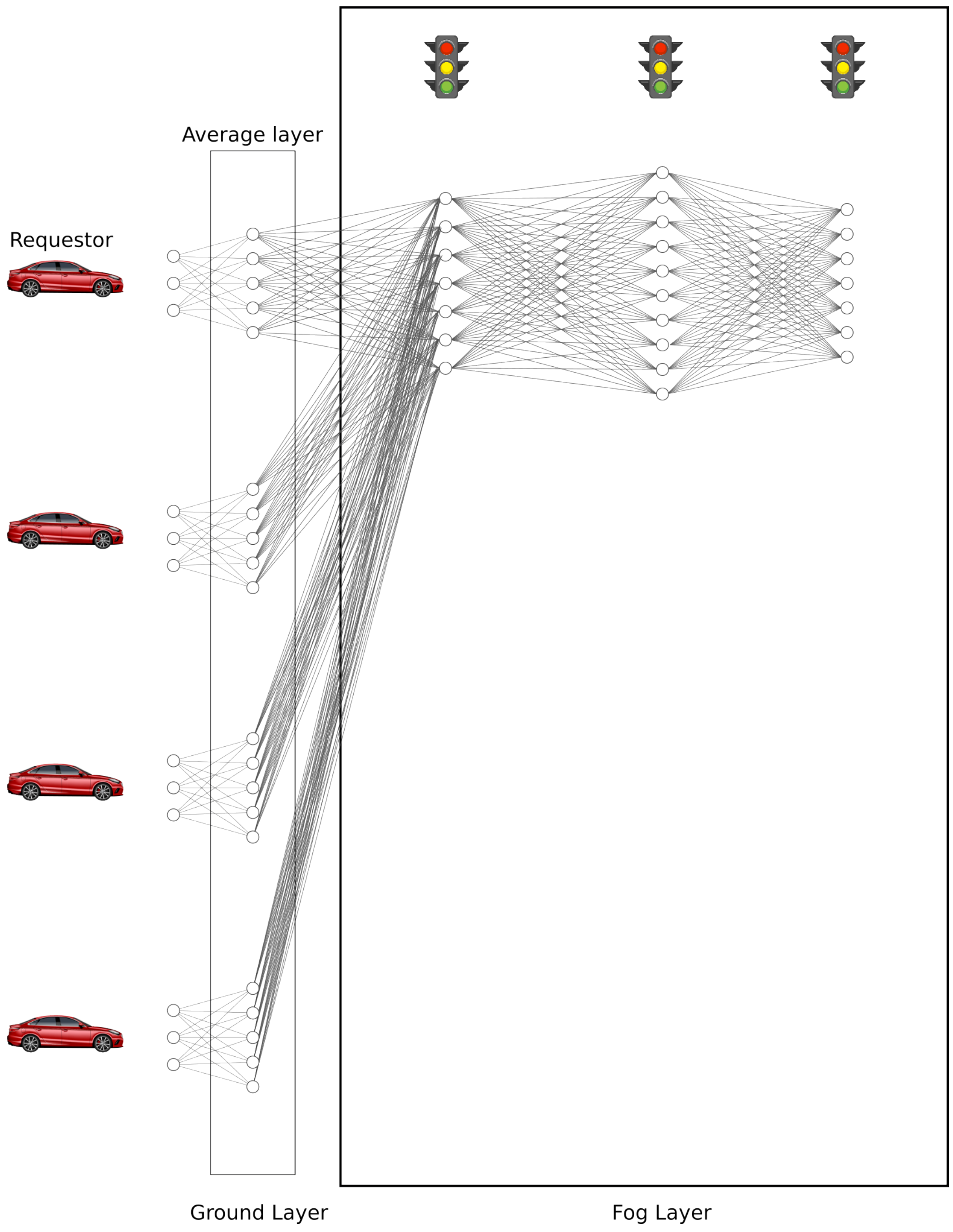


Figure 8: *SplitFedv2.*

because the subset layers of the data source agents are not merged (Figure 9).

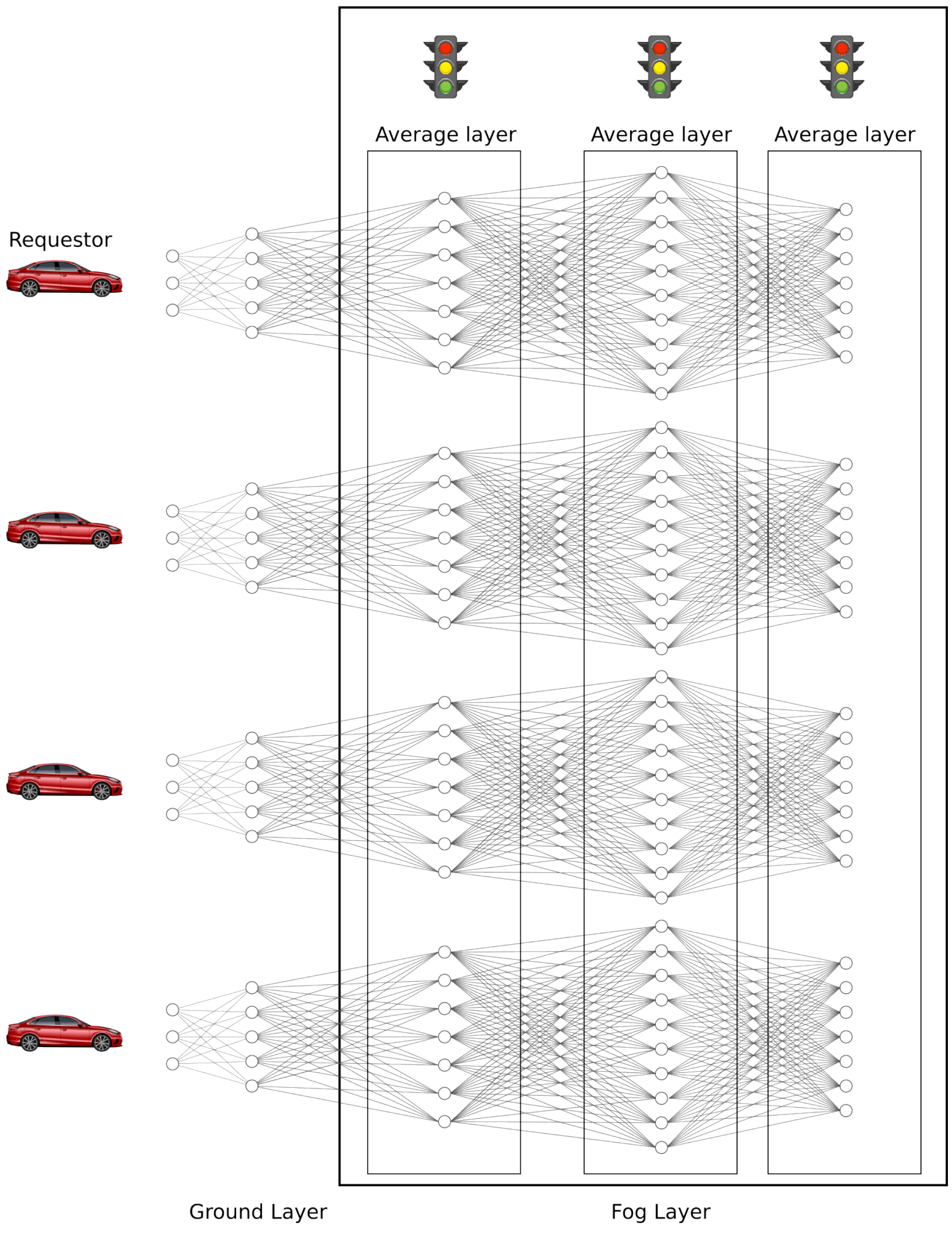


Figure 9: *SplitFedv3.*

5.1.6 Secure Multi-Party Computation

Secure Multi-Party Computation [30, 9] is a cryptography technique where n parties wants to compute a function $f(x_1, x_2, ..., x_n)$, where input x_i is held by the party i to maintains private each party input. In a function secret sharing, the functions are split in n secrets, called shares. To reconstruct the secret, an attacker must collect the majority of the shares. MPC maintains private either the data and the models.

In the example of Figure 10, the nodes that compute the function, i.e., the computation nodes, differ from the ones that provide the data input, i.e., the input parties. Moreover, the node that reconstructs the result of the computation, i.e., the result party, differs from the other nodes of the example.

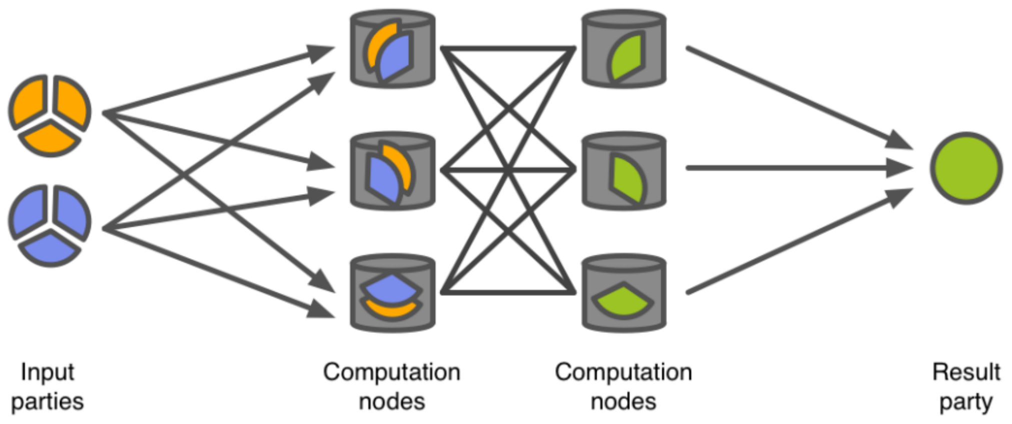


Figure 10: *Secure Multi-Party Computation.*

5.2 Focus on Private Secure Routine

Private Secure Routine (PSR) is a paradigm to identify authorized drivers belonging to the same vehicle in a privacy-preserving manner. Private Secure Routine is built on top of the Secure Routine (SR) paradigm [63]. The advantages of PSR with respect to SR are twofold:

- PSR distinguishes among several authorised drivers depending on their routine. While SR is able to identify only one authorized driver for a target vehicle, i.e., the owner of the vehicle, PSR identifies more than one authorized driver for a target vehicle.

- PSR guarantees that information about drivers and vehicles are exchanged in a privacy-preserving way.

5.2.1 Architecture

The PSR paradigm takes as input all pieces of information about drivers and vehicles circulating in the infrastructure and generates models of each driver in each vehicle using a privacy preserving machine learning algorithm. Each model identifies the drivers that are authorized to drive the vehicle.

We evaluated which privacy preserving ML algorithms choose for PSR from the ones presented in the section 5.1. Federated Learning keeps private only the data, while the model is shared between the devices. In PSR, the model is a representation of the drivers of a target vehicle. The model could be used by the attacker to obtain sensitive information of these drivers. Hence, we discarded FL.

SplitNN and all SplitFed variants keep private also the model through the distribution of the NN layers between the agents. In PSR, the source agents are the vehicles. The data of each driver is present in few cars while the majority of the car do not present any data of the driver. We run preliminary experiments on SplitNN and SplitFed variants where each driver's data is distributed to one vehicle. Each vehicle contains data of two drivers. The experiments show that all these ML algorithms tend to generate dummy models that detect all drivers as unauthorized. We speculate that the cause is the presence of data of each target driver only on few vehicles with the respect to the majority of the vehicles that does not have any data of the target driver. In SplitNN and in the SplitFed variants the model is incrementally trained using the data of a vehicle at time. Hence, the limitation of data implies that what the model learns from the data of the vehicle containing the target driver is forgotten to make room for what is learned from the data of all other vehicles. On the other hand, Secure Multi-Party Computation does not suffer of the data distribution because the computation nodes collaborate each other using the entirety of data to generate the model. For these reasons, the PSR paradigm uses Secure Multi-Party Computation as ML method.

Private Secure Routine works using all three layers present in the transportation infrastructure. Each vehicle collects data and has an unique identifier "ID", which is represented as a random string, that we employ as data addressing during the model training phase that we show in Section 5.3.

RUs are the computational party acting as a bridge between peers of the ground and cloud layer and between vehicles connected to different RUs. In addition, RUs are in charge of share data among entities in a privacy-preserving way by running the MPC technique. Still in this layer, RUs are involved by vehicles to retrieve needed information to train and create drivers' model as described in Section 5.3. Furthermore, as for vehicles even roadside units have a single identifier "ID" that we use as data addressing during the model training phase.

The cloud layer provides cloud storage resources maintaining a database for the association between drivers and vehicles. Moreover, cloud resources send to RUs the labels for the training of models and also they store roadside units and vehicles public keys.

In this work, we consider the cloud and the vehicles as untrusted entities: their intent is to obtain the data of other vehicles. Instead, RUs are trusted entities, i.e., Trusted Third Party (TTP). If one or more RUs are compromised, the vehicles' data and models are kept private by using MPC. Moreover, we assume that all communications among layers happen through secure channels. This will overcome possible active attacks. Also, we use an asymmetric cryptography protocol [89]. The cloud layer publishes the public keys of all the RUs to the Internet.

5.2.2 Threat Model

The cornerstone of Private Secure Routine are the pieces of information about drivers that circulate among the entities belonging to one of the three layers of the depicted scenario. Thus, a possible attacker can play the following attacks:

- Honest-but-Curious (HBC): Also known as Passive Attack; an attacker may exploit the information legitimately gleaned by capturing information exchanged among the three layers of the infrastructure, but he/she will not perform any malicious activity to harvest it.
- Fully Malicious (FM): Also known as Active Attack; an attacker is able to change drivers' information to alter the capability of PSR to identify the drivers. So, the attacker strategy is to succeed in at least one of the following attacks: i) *Impersonation attack* in which attacker forges or alters driver's information that are considered valid by the recipient; II) *Sniffing attack* where the attacker reads the content of any messages exchanged among the layers.

5.3 Model Generation

To identify drivers in a vehicle, PSR creates a model able to identify each driver of a target vehicle that circulates on the PSR infrastructure. Model generation depends on different situations that can occur and involve both drivers and vehicles. Hereafter, we highlight three different scenarios and for each of them, we explain how PSR identifies drivers in the infrastructure presented above.

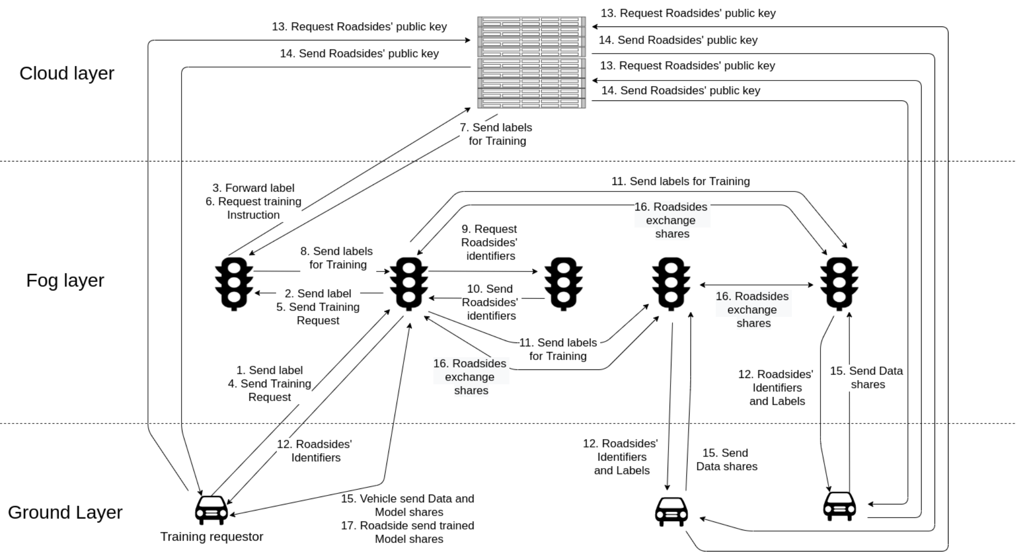

Cloud layer
Fog layer
Ground Layer
13. Request Roadsides' public key
14. Send Roadsides' public key
13. Request Roadsides' public key
14. Send Roadsides' public key
13. Request Roadsides' public key
14. Send Roadsides' public key
7. Send labels for Training
3. Forward label
6. Request training Instruction
11. Send labels for Training
16. Roadsides exchange shares
9. Request Roadsides' identifiers
8. Send labels for Training
2. Send label
5. Send Training Request
10. Send Roadsides' identifiers
16. Roadsides exchange shares
11. Send labels for Training
16. Roadsides exchange shares
1. Send label
4. Send Training Request
12. Roadsides' Identifiers and Labels
15. Send Data shares
12. Roadsides' Identifiers
12. Roadsides' Identifiers and Labels
15. Send Data shares
15. Vehicle send Data and Model shares
17. Roadside send trained Model shares
Training requestor

Figure 11: *The PSR Model Generation Workflow*

In the PSR paradigm we know that drivers' routines are not fixed, they can evolve over time [52], e.g., a mother who takes for the first time her child to school. Also, drivers' style can be different in particular situations, e.g., heavy rain, snowfall, and so on. If driver identification should fail, a vehicle can use a traditional authentication method e.g., password, voice recognition, and so on, as a fallback method to identify the driver. When a fallback method, out of scope of this thesis, is used, a vehicle labels the recent sensors' data as belonging to the authenticated driver. Even if traditional authentication methods fail, sensors' data are labelled as belonging to a non-authorized driver. Moreover, the PSR paradigm transparently authenticates the driver, i.e., PSR continuously compares the actual driving style with its model. Each vehicle periodically issues a new training model to improve the learning of new routines and driving styles.

5.3.1 A new vehicle joins the PSR infrastructure

When a driver d_i with her vehicle v_i join the PSR infrastructure for the first time, the creation of a new model is needed to identify d_i. To do this, our PSR requires that the vehicle collects data from its sensors related to driver d_i. Upon the collection of these data, the vehicle v_i generates two labels: one label l_i tied to driver d_i and another label l_{other} to consider data for other drivers different from d_i that may use vehicle v_i. The collected data will be sent to the RUs that will be in charge to train the model for v_i.

Hereafter, we detail how data are collected, shared with the PSR infrastructure and how the model is trained.

Model Initialization: A vehicle does not share the labels directly with the cloud layer. So, v_i first contacts the nearest roadside unit to share its labels and its ID (Step 1 in Figure 11). When the RU receives l_i, l_{other} and the vehicle's ID from v_i, it may forward the labels and the ID to the cloud layer that can grab sensitive information belonging to v_i, e.g., the geographical position of the nearest RU may provide indication about the vehicle location. To avoid this, the roadside unit close to v_i sends the labels and the ID to an intermediary RU randomly chosen (Step 2). Only now, the intermediary forwards the labels and the ID to the server (Step 3) at cloud layer. We assume that the intermediary RU is trusted. When labels and the ID are received at the cloud layer, they are stored on a local repository, for instance a database. From now on, v_i and d_i are part of the PSR infrastructure. However, v_i is not yet ready to identify d_i since the model has not been trained.

The vehicle requests the training of its model: The vehicle v_i sends a request to train its model to identify d_i. This step is obtained through a request that is sent to the nearest RU, (Step 4). This one forwards the training request to an intermediary RU, (Step 5). This RU sends the training request at the cloud layer (Step 6). It, then, identifies the label to use, i.e., l_{other}. Note that label l_{other} is identified but not yet sent to RU.

The cloud layer requests the training for v_i's model: To train v_i's model, the l_{other} label is sent from the cloud layer to other vehicles involved in the PSR infrastructure. So, l_{other} is sent, first, to the intermediary roadside unit (Step 7) that forwards the label to the RU closest to v_i (Step 8) and sends l_{other} alongside with the v_i's ID (Step 11) to other RUs within a certain radius, whose size is not relevant here. Finally, each RU forwards the l_{other} label to their connected vehicles together with the IDs of the roadside units involved in the training (Step 12). Note that, the label l_i is not sent because there are no vehicles that have data belonging to the driver d_i. Hence, the only vehicle that has the label l_i is the vehicle v_i itself. At step 12, v_i receives the IDs of the RUs involved in the training.

Vehicles send their shares to train v_i's model: To train v_i's model, each vehicle belonging to the PSR infrastructure must share its collected data sensors. First, each vehicle, except v_i, labels its sensors' data with the l_{other} label. The sensors' data of vehicle v_i are already labelled. Then, each vehicle obfuscates its data splitting them in shares. Next, each vehicle sends these shares to the nearest RU. Since a roadside unit may reconstruct the original data with all the shares received, the vehicle must encrypt the shares so that only the right RU can have them in plaintext. In the PSR infrastructure, we assume that the channel kept secret the shares using asymmetric cryptography protocol, e.g., OpenPGP [97]. Hence, a vehicle, before sending each share to a RU, must obtain the public key of the recipient roadside unit (Step 13) from the cloud layer. Once the corresponding RU public key is retrieved (Step 14), the vehicle uses the key to encrypt the share before sending it to the nearest RU (Step 15). Always at Step 15, the vehicle v_i sends its encrypted shares to the nearest RU. Then the vehicle sends all encrypted shares, the nearest RU will forward them to the other RUs belonging to the infrastructure to train the model (Step 16).

Training v_i's model: All roadside units, involved in the training, decrypt the shares and train v_i's model using the data received from all vehicles in the previous steps.

The vehicle v_i receives the updated model: At the end of training, the roadside units send to v_i the shares containing the new model of d_i. To do this, each RU encrypts the shares with the public key of v_i and sends the resulting shares to the vehicle. From now on, v_i is able to identify d_i and distinguishes her from other drivers (Step 17).

5.3.2 A new driver joins the PSR infrastructure

This is the case when a new driver d_i is identified on a vehicle v_i already present in the PSR infrastructure. The vehicle v_i has already a model trained for another driver d_j, with $d_j \neq d_i$, and sensors' data used for d_j are labelled as l_j. Driver d_j must authorize the new driver. The procedure to give d_i the permission to drive the vehicle v_i is out of scope. The vehicle v_i must update the current model to identify also the new driver d_i. To perform this task, we proceed similarly to the previous scenario and the following steps occur.

Model Initialization: The vehicle v_i generates the label l_i. Labels l_{other} and l_j already exist on vehicle v_i since they were created for driver d_j. Then, the vehicle sends the label l_i and its ID to the nearest RU (Step 1). This one forwards the label l_i and vehicle's ID to an intermediary RU randomly chosen (Step 2). Finally, the cloud layer receives l_i and the ID from the intermediary RU and stores them on a database (Step 3). Now, the new driver d_i is part of the transportation infrastructure. However, the vehicle v_i is not yet able to identify d_i since its model has not been trained.

The vehicle requests the training of its model: In this phase, the vehicle v_i sends to the nearest RU a training request (Step 4). The RU forwards this request to an intermediary RU (Step 5) and, then the intermediary sends the request to the cloud layer (Step 6).

The server request the training for v_i's model: The cloud layer sends a training request to all vehicles on the PSR infrastructure within a certain radius. Cloud layer knows the label of driver d_i from Step 3 and the label l_j due to past training for driver d_j. As next step, from the cloud layer the labels l_{other}, l_j and the ID of vehicle v_j are sent to the intermediary RU, (Step 7) which provides to the closest roadside unit of vehicle v_i the labels and the ID (Step 8).

Roadside Unit searches vehicles with authorized drivers in common with v_i: This is a new step with respect to the previous training scenario. We introduce this step since it may happen that driver d_j may be a driver of another vehicle v_j different from v_i. Keeping track of the driver across different vehicles may be useful for car sharing: the car sharing company provides several cars to drive that could be the same model from the same carmaker. The driver may drive at each driving session a different vehicle but maintaining the same driving style. In case the driver is not tracked between the vehicles, the model will contain the driving sessions of d_j in the vehicle v_i as authorized while the driving sessions in v_j has not authorized that may lead to a model not able to correctly identify d_j. Also, in the case the vehicles v_j and v_i are different models of different carmakers, the driver could maintain some similarities in driving style between the vehicles. So, we need to consider this situation by collecting also data coming from other vehicles. In this phase, the RU r_i, to which v_i is connected, looks for the vehicle v_j and contacts it through its nearest roadside unit, which we label as r_j. The PSR infrastructure uses a Distributed Hash Table (DHT) [105] to keep track of the nearest roadside unit of each vehicle present in the PSR infrastructure. A DHT provides a lookup table, distributed between peers of a network, to quickly locate data. The Roadside Units are peers of the network. The lookup algorithm uses a key to locate the peer using {key, value} pairs. In particular, the key is the v_j's ID hash and value represents the ID of the nearest Roadside Unit to v_j (Step 9). The RU r_i receives the ID of r_j closest to vehicle v_j (Step 10). Then, r_j receives from r_i the labels l_{other}, l_j and the IDs of v_i and v_j (Step 11). Always at Step 11, r_i sends the label l_{other} and the ID of v_i to the other roadside units. The roadside unit r_j forwards the labels to vehicle v_j and the IDs of the roadside units involved in the model training (Step 12). Also at the same step, each roadside unit sends the label l_{other} and the RUs' identifiers to other vehicles of the PSR infrastructure. Finally, r_i sends only the identifiers to v_i, at the same Step 12.

Vehicles send their data to train the v_i's model: Now, each vehicle shares the data collected from its sensors with the RUs to train v_i's model. First, the vehicle v_j labels with l_j sensors' data that belong to driver d_j. All remaining data not belonging to d_j are labelled as l_{other}. Vehicle v_i sensors' data are already correctly labelled. The other vehicles label their sensors' data as l_{other}. Then, all vehicles get the RUs' public key from the cloud layer (Steps 13 and 14). Then, each vehicle of the PSR infrastructure sends the data, through encrypted shares, to its nearest roadside unit for training (Step 15). Then, these shares will be exchanged among the RUs before the training session (Step 16).

Training v_i's model: Each roadside unit decrypts the received shares and trains the model.

The vehicle v_i receives the updated model: When the training is concluded, the roadside units send to v_i the encrypted shares of the new model exploiting the closest roadside unit r_i (Step 17). Then, the vehicle decrypts and combines all received shares to obtain the new model. From now on, vehicle v_i is able to identify d_i and d_j.

5.3.3 Driver identification on a vehicle of the infrastructure

This scenario considers the case in which a driver d_i has already a model in a vehicle v_i and we wish to identify the driver into a different vehicle v_j. Since, data related to driver d_i already exist in the transportation infrastructure, we can directly train her corresponding model. However, even if the model of vehicle v_j is trained using previously d_i driving sessions in v_i, the driver may use a driving style completely different, e.g., a city car is driven in a different way than a sports car. The fallback authentication allows PSR to register the different driving style to later update v_j's model.

Model Initialization: To train the new model, RUs use driver d_i's past data collected by other vehicles of the platform. The resulting model allows vehicle v_j to identify driver d_i. First, vehicle v_j generates a random label l_i for driver d_i. The label l_{other} already exists so it is not necessary to create a new one. Then, vehicle v_j shares the new label alongside with its ID using the nearest roadside r_j (Step 1). Upon label and ID reception, the roadside unit chooses randomly an intermediary RU and forwards the label and the ID to it (Step 2). The intermediary RU forwards the label l_i and the ID to the cloud layer that stores them (Step 3).

Request to train v_j's model: The cloud layer knows v_j has already a model for a driver d_j, with $d_j \neq d_i$. Assuming that driver d_j is already known on the vehicle v_j with the label l_j, v_j's model should include also driver d_i in addition to d_j. Driver d_j could be also known in the vehicle v_k, with $v_k \neq v_j$. Next, a training request for v_j's model with the labels l_i, l_j, l_{other} and the IDs of vehicles v_j, v_i and v_k is sent to the intermediary RU (Step 7). This sends the labels and the IDs to nearest RU r_j (Step 8).

Searching vehicles that already know d_j and d_i: The roadside unit close to v_j looks for those vehicles that already know one or both drivers d_i and d_j. In Step 9, the roadside unit r_j looks for the other RUs that are close to vehicles v_i and v_k to be able to contact them. Then, in Step 10 the roadside unit r_j receives the "IDs" of r_i and r_k respectively nearest to v_i and v_k. On Step 11, r_i receives the labels l_{other}, l_i and the "ID" of vehicles v_i and v_j while r_k receives the labels l_{other}, l_j and the "ID" of vehicles v_k and v_j. At the same step, r_j sends to the other RUs the label l_{other} and the "ID" of vehicle v_j. At Step 12, the roadside units r_i and r_k share the labels respectively with v_i and v_k and the identifiers of RUs involved in the training. At the same time, all the roadside units send the label l_{other} and the IDs to other vehicles of the PSR infrastructure, except v_j that receives only the identifiers.

Vehicles send their data to train v_j's model: Each vehicle sends its data to the RUs involved in the training: vehicle v_i labels with l_i sensors' data that belong to driver d_i and, similarly, vehicle v_k labels with l_j sensors' data that belong to driver d_j. All remaining sensors' data of vehicles v_i and v_k get the label l_{other}. Also, the other remaining vehicles of the infrastructure label their data with l_{other}. The data of vehicle v_j are already correctly labelled, hence no changes are needed. Each vehicle retrieves the public keys of the RUs involved in the training (Step 13 and 14). Then, the vehicle splits data in shares, encrypts the shares with the public keys retrieved in the previous step and sends them to the vehicle's nearest roadside unit for training (Step 15). Such shares are then shared among the RUs (Step 16).

Training v_j's model: All the RUs involved in the training receive the shares, decrypt them and train the model.

Vehicle v_j receives the updated model: At the end of training, the RUs send to v_j the encrypted shares of the new model through r_j, Step 17. Here, v_j decrypts the shares and combine them to obtain the model. From this moment on, the vehicle v_j is able to identify d_i, d_j.

5.4 Private Secure Routine Implementation

The Private Secure Routine paradigm is implemented by using PySyft framework, a Python library that implements the secure Multi-Party Computation (MPC) for private training of Neural Networks [83]. The framework maintains both parameters of the model and the dataset private.

The PySyft implementation of MPC is secure against the honest-but-curious adversaries [83] but can not guarantee security against active attackers. Some parties could exchange their shares and potentially reconstruct the original values. The maximum number of corrupted parties before the MPC security is violated depends on the MPC scheme in use. A scheme can be secure up to $n - 1$ elements, where n is the number of parties [83].

We aim to evaluate the impact of MPC on the accuracy results. Hence, we simulate the chain of computations of the parties belonging to the PSR infrastructure without the bottleneck of communications over the network. To do this, we simplify the behaviour of both instances of ground and cloud layer. At ground layer, vehicles generate the dataset and label the instances. At cloud layer, the server sends requests to vehicles to trigger the labelling activity. In our implementation, we employ three roadside units, two of them train the neural network, while the third RU acts as the *Crypto-Store*, i.e., it provides the necessary elements for the computation in the MPC environment [70]. All parties are *Virtual Workers*, i.e., they run on the same computer.

5.4.1 Labelling Generation Algorithm

The labelling generation involves the situation in which vehicles generate their dataset to train the vehicle v's model in the different training scenarios described in Section 5.3. Once the dataset is created, vehicle labels the dataset instances (ins) depending on the experiment we are going to setup.

Listing 3 shows how the labelling of the experiments is generated. The algorithm uses the label "Other" to every instances with a label not present in *drivers*, i.e., list of the authorized drivers: if each vehicle is owned by only one driver, d, the list *drivers* contains only d. In case each vehicle is owned by more than one driver, e.g., two drivers, d_1 and d_2, the list *drivers* contains these two labels.

Listing 3: *Labelling generation*

```
function generate_owner_labelling(drivers, ins)
$ins_{labelled}$ ← for each instance in ins set label "Other" to instances not
    made by one of the drivers in driver
return $ins_{labelled}$
```

The dataset has to be shared with the RUs to train the models. Each vehicle creates a private dataset by generating the *shares* (Listing 4) of both training set (85% of instances) and test set (the remaining 15%). The function receives in input the workers' references, one for each RU: the virtual workers that train the model and the crypto provider that setups the Function Secret Sharing algorithm.

The training set and test set are split in batches (Listing 4, lines 5 and 8) with size equal to 1024. To better train the network, the labels are converted into the *one-hot encoding* [104] (Listing 4, lines 6 and 9). A one-hot encoding is a vector of length equal to the number of possible labels, e.g., in case of two owner, the vector size is three: d_1, d_2 and "Other". In Private Secure Routine, the vehicles must create vector of size much bigger than the number of drivers of v: for instance suppose that the one-hot vector is of size $number\_owners\_drivers + 1$, where the first components of the vector represent the owners of the vehicle and the last component represents the other drivers. A malicious vehicle could easily set its past driving instances has belonging to one of the legit v's owner, e.g., set all instances in the first component as if they belong to the first owner. In case the vector size is too big and the components that represent the legit owners are chosen randomly, the attacker does not know which components represent the legit owners. Hence, she is not able to create fake instances belonging to a legit owner. Then, vehicle creates the shares of training and test set for each virtual worker (Listing 4, lines 7 and 10).

Listing 4: *Generation of dataset shares - Vehicle*

```
1  function generate_private_dataset($ins_{labelled}$, $workers$, $crypto\_provider$)
2  ($ins_{train}$, $ins_{test}$) $\leftarrow$ choose randomly 85% of instances as training and
       the remaining as testing from $ins_{labelled}$
3  ($x_{train}$, $y_{train}$) $\leftarrow$ separate features of the $ins_{train}$ instances from the
       respective label
4  ($x_{test}$, $y_{test}$) $\leftarrow$ separate features of the $ins_{test}$ instances from the
       respective label
5  $loader_{train}$ $\leftarrow$ split $x_{train}$ and $y_{train}$ in batches. Data are shuffled
6  $loader\_one\_hot_{train}$ $\leftarrow$ labels in $loader_{train}$ are one hot encoded
7  $loader\_private_{train}$ $\leftarrow$ create shares of labels and features in
       $loader\_one\_hot_{train}$, distributing them between $workers$ with the help
       of $crypto\_provider$
8  $loader_{test}$ $\leftarrow$ split $x_{test}$ and $y_{test}$ in batches
9  $loader\_one\_hot_{test}$ $\leftarrow$ labels in $loader_{test}$ are one hot encoded
10 $loader\_private_{test}$ $\leftarrow$ create shares of labels and features in
       $loader\_one\_hot_{test}$. Return the pointers to the shares
11 return ($loader\_private_{train}$, $loader\_private_{test}$)
```

5.4.2 Dataset preparation for training

Once RUs have received the private dataset of each vehicle, they are in charge of running the secure multi-party computation technique. Train dataset of each vehicle are combined in a single dataset (Listing 5, line 2). We concatenate the datasets of different vehicles to run the *average imputation.*

The procedure handles the, eventually, missing values with the average value of each feature. Also, neural networks require that training dataset is randomly shuffled for an optimal training [104]. Without the dataset concatenation, the average imputation should be run separately for each vehicle dataset, obtaining different averages. Also, at training phase, the neural network should sequentially train each vehicle training dataset. Finally, managing a single training dataset is simpler than managing multiple training datasets. Similarly, test datasets are combined too (line 3).

The average imputation in the training set replaces missing values with the average value of each feature, whereas in the test set null values are filled with the same averages obtained from the training set [50]. First, we calculate the average value of each feature in $loader\_private_{train}$ (line 4). Then, we fill null values with the respective average (line 6) and create new feature columns to keep track of the rows that have null values for each feature (line 5) and the columns of the rows with null values and we apply the average imputation in the test set (lines 7 and 8).

Next step is the normalization of training and test set to speed up the process of model training [35]. We use the *Min-Max Normalization* procedure: for each feature f the dataset values are transformed from the range $[min^{f}_{loader\_private_{train}},\ max^{f}_{loader\_private_{train}}]$ to the range $[min,\ max]$ arbitrary using the equation:

$$v_{new} = \frac{v_{old} - min^{f}_{loader\_private_{train}}}{max^{f}_{loader\_private_{train}} - min^{f}_{loader\_private_{train}}}(max - min) + min \qquad (4)$$

We choose min equal to -1 and max equal to 1. In Listing 5 line 9, we get the minimum and maximum value for each feature in $loader_{train\_filled}$. Then, we apply the Equation 4 to normalize the $loader_{train\_filled}$. Similarly, we normalize the $loader_{test\_filled}$ using the same $[min^{f}_{loader_{train\_filled}},\ max^{f}_{loader_{train\_filled}}]$.

Listing 5: *Preparation of dataset for training - RUs*

```
function training_preparation_dataset($array\_loader\_private_{train}$, $array\_loader\_private_{test}$)
    $loader\_private_{train}$ ← concatenate loaders in $array\_loader\_private_{train}$ in a single loader
    $loader\_private_{test}$ ← concatenate loaders in $array\_loader\_private_{test}$ in a single loader
    $averages$ ← calculate average of each feature in $loader\_private_{train}$
    $loader_{train\_cols\_na}$ ← add a column for each feature, each row contains $TRUE$ if the value is null, $FALSE$ if the value is present
    $loader_{train\_filled}$ ← fill null values in $loader_{train\_cols\_na}$ with the average in $averages$ of the respective feature
    $loader_{test\_cols\_na}$ ← add a column for each feature in $loader\_private_{test}$ with null value, each row contains $TRUE$ if the value is null, $FALSE$ if the value is present
    $loader_{test\_filled}$ ← fill null values in $loader_{test\_cols\_na}$ with the average in $averages$ of the respective feature
    $scaler$ ← for each feature in $loader_{train\_filled}$ get minimum and maximum values
    $loader_{train\_normalized}$ ← normalize $loader_{train\_filled}$ in range (-1, 1) according to $scaler$
    $loader_{test\_normalized}$ ← normalize $loader_{test\_filled}$ in range (-1, 1) according to $scaler$
    return ($loader_{train\_normalized}$, $loader_{test\_normalized}$)
```

5.4.3 Model Generation Algorithm

Each vehicle v creates a new model as described in Listing 6. As a first step, a vehicle creates the initial model, (line 2). To optimize the training speed, we adopt a multi-layer Neural Network with three layer architecture: two linear hidden layers with *dropout* [37], as method of regularization, and the *rectified linear unit* (ReLU) [104] as activation function defined as follows:

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{5}$$

The first hidden layer is made of 128 neurons and the second hidden layer contains 64 neurons. The output layer is linear and its size depends on the experiment.

Once the model is created (line 2), the vehicle v generates the shares for the model (line 3). Such shares are sent to RUs in a privacy preserving way. To train the model is required an optimizer that adjusts the model parameters at each epoch to reduce the loss and to increase the accuracy [104]. Then, v defines the optimizer and its parameters (line 4). The model and the optimizer parameters are converted from float to *fixed precision* (line 3 and 5) according to PySyft requirements [84]. Fixed precision represents values with two components: an integer, i.e., the coefficient and the position of the radix point, i.e., the exponent. A value is represented as $value = coefficient * 10^{exponent}$ [69]. Having a low value for the exponent allows RUs to speed up the training but it reduces the accuracy. In the experiments, we keep 3 decimals from the value. Once the model is split into shares and optimizer parameters are converted in fixed precision, the vehicle sends the shares to all the RUs.

Listing 6: *Model generation algorithm - Vehicle*

```
1 function initialize_model_shares($workers$, $crypto\_provider$, $lr$)
2 $model$ ← create Neural Network
3 $model\_private$ ← create shares of $model$, distributing them between
      $workers$ with the help of $crypto\_provider$
4 $optim$ ← define the Stochastic Gradient Descent optimizer with the
      learning rate $lr$
5 $optim\_private$ ← convert the optimizer parameters in fixed precision
6 return ($model\_private$, $optim\_private$)
```

5.4.4 Model Training Algorithm

Roadside units are in charge of training the model (Listing 7). As a first step, RUs search the best *Learning Rate* (LR) for the model running a known algorithm [87] that returns shares of the learning rate. The size of the LR influences how much the optimizer adjusts the parameters at each epoch. The smaller is the LR the more are the epochs necessary for training. On the other hand, the training may never converge to a good accuracy if LR is too high [104].

Once the LR is found, the roadside units send the learning rate shares to vehicle v. The vehicle initializes a new model and a new optimizer, as seen in Listing 6, and set the learning rate value with the LR found by the roadside units. From line 5 to 14, roadside units train the model. We set the number of epochs of training (equal to 2 in our experiments). For each epoch i, the parameters of the model change and the accuracy is calculated. Hence, the roadside units must return to the vehicle the most accurate model. First, we define the variables that will contain the best model with the correlated accuracy (lines 5 and 6). For each epoch, we train the model with the training dataset, the optimizer and the loss function MSE (line 8) [16].

$$MSE = \frac{1}{n}\sum_{i=0}^{n-1}(\hat{y}_i - y_i)^2 \tag{6}$$

where: n is the number of predictions; $\hat{y}_i$ is the i-th value predicted by the NN; y_i is the i-th actual value.

Then, the accuracy of making predictions over the test set is calculated by comparing the results with the correct labels. Next, the stored best accuracy is compared with the last obtained accuracy (line 10). In case there is no best accuracy stored, the current accuracy and the current model are stored (lines 11 and 12). In case the current accuracy is greater than the stored one, the current accuracy and the current model are saved. Otherwise, they are discarded. After the model is trained for all the epochs, the RUs send the shares of the best model and the relative accuracy shares to v (line 15). The vehicle combines the shares and obtains the model trained and the accuracy. Note that the RUs do not need to disclose the accuracy values to make the comparisons. Only v will know how well the model performs.

Listing 7: *Model training algorithm - RUs*

```
1  function find_lr($workers$, $crypto\_provider$, $loader_{train\_normalized}$, $model\_private$, $optim\_private$)
2  $lr\_private \leftarrow$ find the best learning rate to train $model$ using the training dataset $loader_{train\_normalized}$, the optimizer $optim\_private$ and the loss function MSE
3  return $lr\_private$
4  function train_model($workers$, $crypto\_provider$, $loader_{train\_normalized}$, $loader_{test\_normalized}$, $model\_private$, $optim\_private$)
5  $best\_accuracy\_private \leftarrow$ NULL
6  $best\_model\_private \leftarrow$ NULL
7  for $epoch$ = 1 to 2 then:
8  train $model\_private$ using the training dataset $loader_{train\_normalized}$ and the optimizer $optim\_private$ and the loss function MSE
9  $accuracy\_private \leftarrow$ make predictions using the values of features in $loader_{test\_normalized}$ and calculate the accuracy shares using the aforementioned predictions and the label in $loader_{test\_normalized}$
10 if $best\_accuracy\_private$ is NULL OR $best\_accuracy\_private$ < $accuracy\_private$ then:
11 $best\_accuracy\_private \leftarrow accuracy\_private$
12 $best\_model\_private \leftarrow model\_private$
13 end if
14 end for
15 return $best\_model\_private$, $best\_accuracy\_private$
```

5.5 Private Secure Routine Evaluation

We compare PSR paradigm with Secure Routine [63] and the work in [61], hereafter denoted with M. We experiment PSR on two examples: 1) single owner identification and 2) two owners identification. Then, we evaluate the NN of PSR comparing it alongside an architecture trained without the application of secure multi-party computation. This is because PySyft with MPC is time consuming due to the fact that parties need to exchange several messages within the training phase and that PySyft uses the CPU instead of the GPU that is not yet supported.

5.5.1 Experiments

To evaluate PSR, we performed eight experiments on two datasets: Θ [33] and Ψ [5].

The experiments run on a Virtual Machine with an Intel(R) Xeon(R) Gold 6140M using 8 threads, 32 GB of RAM and Ubuntu 18.04 as OS. Our experiments compare PSR with Secure Routine and M on the *Accuracy* metric, previously introduced in paragraph 4.2.2.

Single Owner identification (SOI). We aim at identifying if a target instance belongs to the vehicle's owner.

Table 7: *Accuracy comparison for SOI test bed with the state of the art*

(a) SOI in Ψ

PSR	**SR**	M
93,79%	99,84%	98,46%

(b) SOI in Θ

PSR	**SR**	M
89,96%	99,83%	99,62%

As first experiment, we compare PSR with Secure Routine and M on the Ψ dataset. We select the best feature set in PSR. Since MPC is time consuming, we trained PSR only for two epochs (Listing 7). Secure Routine achieves the best results (Table 7(a)). PSR is 6,05% of accuracy lower in comparison with Secure Routine and 4,67% less than M. Note that M do not use this dataset in their work, so we replicated their experiments to establish the accuracy. Despite PSR model being trained only two epochs, it scores an high accuracy keeping private the used data.

We repeat the same experiment on Θ. We use the same feature selected in [63]. The work of M adopted the dataset Θ but they do not calculate the accuracy, so we replicates their experiments to retrieve the accuracy. Table 7(b) shows that PSR obtains 89,96% of accuracy, SR achieves the best accuracy. PSR obtains 9,87% of less accuracy than SR.

To measure the impact of MPC on the accuracy of PSR, we evaluate the PSR network comparing the results of the same neural network trained for 2 and 4000 epochs but without the support of MPC.

Then we consider the dataset Ψ. Table 8(a) shows that PSR without MPC scores 99,91% of accuracy while PSR with MPC has lost 2,58% of accuracy in comparison with the model trained in plain for 2 epochs. This loss of accuracy may be caused by the conversion of input data and model parameters in fixed precision required by PySyft.

Moving on Θ, Table 8(b) shows PSR scores slightly lower (0,04%) than the PSR without MPC with two epochs.

Two Owners Identification (TOI). Here, we test PSR in case a vehicle is owned by two drivers. Neither Secure Routine nor M were designed and work with two owners identification. Since we are not able to replicate the work in M to test this scenario, we compare PSR only with a slightly modified version of SR able to manage also this case. Once again SR achieves the best result, i.e., 99,69% (Table 9(a)). PSR obtains a respectable 87,51%, i.e., 12,18% less than the same model trained 2 epochs with plain data.

Table 8: *Accuracy comparison for SOI test bed to measure MPC impact*

(a) SOI in Θ

PSR 2 epochs	**PSR 2 epochs (no MPC)**	**PSR 4000 epochs (no MPC)**
93,79%	96,37%	99,91%

(b) SOI in Θ

PSR 2 epochs	**PSR 2 epochs (no MPC)**	**PSR 4000 epochs (no MPC)**
89,96%	90%	97,77%

Table 9: *Accuracy comparison for TOI test bed with the state of the art*

(a) TOI in Ψ

PSR	**SR**
87,51%	99,69%

(b) TOI in Θ

PSR	**SR**
80,08%	99,71%

Table 10: *Accuracy comparison for TOI test bed to measure MPC impact*

(a) TOI in Ψ

PSR 2 epochs	**PSR 2 epochs**	**PSR 4000 epochs**
87,51%	92,31%	99,88%

(b) TOI in Θ

PSR 2 epochs	**PSR 2 epochs**	**PSR 4000 epochs**
80,08%	80,04%	95,21%

Then, we repeat the experiment also on the other dataset. Table 9(b) indicates that SR obtained an average Precision of 99,71%. In comparison, PSR loss 19,63% of accuracy. This results depends on the NN poorly trained, i.e., trained for only 2 epochs. An higher number of epochs will increase the accuracy as shows the table 9(b), but more on that later.

Also in this case, we evaluate the impact of MPC on PSR. Let us consider the dataset Ψ. Again PSR without MPC achieves an high score, (99,88%), Table 10(a).

With MPC, PSR loses 4,8% comparing with the same model trained two epochs with plain data.

Moving on the dataset Θ, PSR obtained a better result than the public trained on 2 epochs, i.e., 0,04% more accurate (Table 10 (b)). The best result is obtained by the PSR without MPC fully trained (95,21%).

6 CAHOOT

Software in modern vehicles is becoming increasingly complex and subject to vulnerabilities that an intruder can exploit to alter the functionality of vehicles. The United Nations requires automakers to detect intrusions [68]. Also, an exploit may alter the operation of Secure Routine and Private Secure Routine. To this purpose, we introduce CAHOOT, a novel context-aware Intrusion Detection System (IDS) capable of detecting potential intrusions in both human and autonomous driving modes. In CAHOOT, context information consists of data collected at run-time by vehicle's sensors and engine. Such information is used to determine drivers' habits and information related to the environment, like traffic conditions.

CAHOOT extends the existing literature because it is the first IDS based also on context information able to detect replay and DoS attack in addition to the spoofing attack. Moreover, the simulation environment and activity we present is the only one that take into account simultaneously brakes, steering and throttle parameters. Table 11 shows a comparison with the main context-aware IDS.

In this section, we create and use a dataset by using a customised version of the MetaDrive simulator capable of collecting both human and AI driving data. Then we simulate several types of intrusions while driving: denial of service, spoofing and replay attacks. As a final step, we use the generated dataset to evaluate the CAHOOT algorithm by using several machine learning methods. The results show that CAHOOT is extremely reliable in detecting intrusions.

6.1 Attack Model

In the last decade, the literature presents several examples of vehicle's attacks like the attack made by Miller and Valasek presented in Section 1.

Table 11: *Comparison of features between CAHOOT and the main context-aware IDS*

		CAHOOT	**RAIDS [42]**	**[48]**	**CAIDS [101]**	**[12]**
Attacks	DoS	✓	✗	✗	✗	✗
	Spoofing	✓	✓	✓	✓	✓
	Replay	✓	✗	✗	✗	✗
Sensors attacked	Engine	✓	✗	✗	✓	✗
	Steering	✓	✓	✓	✗	✓
	Brake	✓	✗	✗	✗	✓

This attack highlighted, for the first time, the vulnerability of the CAN busses as in-vehicle communication protocol and, consequently, the importance of studying cyber-security issues in this domain. In fact, all the attacks in literature leverage the lack of confidentiality for data in transit on the intra-vehicle CAN bus network, which are, consequently, exposed to several threats. An intruder may exploit local or remote vulnerabilities of a car to gain some digital access to the car, either locally or remotely. She may then modify the behaviour of a target vehicle by sending customized CAN frames that triggers a specific functionality on a receiving ECU.

A lot of information circulate inside and outside vehicles by using ICT systems that are installed on it. An autonomous car contains various sensors to keep track of the environment and the vehicle status [107]. Inside the vehicle, there are also several ECUs that provide functionalities to the car. Such ECUs are connected one another through multiple buses, e.g., CAN, CAN-FD, FlexRay and Automotive Ethernet. Different partitions of these busses are connected each other through gateways. Thus, vehicles are computers on wheels and as normal computer can be subject to remote attacks. An intruder may exploit local or remote vulnerabilities of a vehicle to gain some digital access to it, either locally or remotely.

In CAHOOT, we consider an *intruder* able to run the following attacks:

- *DoS* attack: the intruder is able to deny the driver's input through the generation of CAN frames where payloads values are set to zero for steering, throttle and brakes.

- *Spoofing* attack: the intruder is able to generate a valid CAN frame. For example, the forged frame may generate a valid signal to active an ECU functionality.

- *Replay* attack: the intruder is able to re-use valid CAN frames with a malicious or fraudulent aim.

When an intruder launches a spoofing attack, a valid pair steering and throttle/brake is randomly generated. Although it may be the case that randomly generated pair corresponds to a pair previously generated by a driver, the probability of this occurring is very low. In CAHOOT, the attacks are coded in Python 3 using the random library[73]. Steering and throttle/brake values in the simulator assume float values in the interval $[-1, 1]$. To generate spoofing attacks, we use the uniform function[75] that generates random float numbers in $[a, b]$ where a and b are in our case respectively -1 and 1. The uniform function is based on the random function[74] with the following equation:

$$a + (b - a) * random() \tag{7}$$

where $random()$ returns a random float number in the interval $[0, 1)$.

However, the random function returns only multiples of 2^{-53}[72]. Thereby, the uniform function returns only a subset of $[a, b]$ and may not be able to reproduce all the possible values generated by the AI and a human driver. In the Python documentation [72] is also presented an alternative random function, known as full random, able to return all the possible float numbers in $[0, 1)$. Hence, we introduce a uniform function variant called full uniform function with the following equation:

$$a + (b - a) * full\_uniform() \tag{8}$$

In our experiments, we collected 10492 unique steering values and 6629 unique throttle/brake values on 157318 driving instances. We generated 314636 random values, i.e., 157318 values represent the steering wheel and the remaining 157318 represent the throttle/brake, using the uniform function and the full uniform function. These functions never generated a legit value because the legit values are a really small fraction of all the possible float values. Also, the random function never returned 0 as value. Hence, a spoofing attack using a random number is unlikely to generate replay and DoS attack.

Finally, the replay attack contains sequences of 0 messages in either steering wheel and throttle/brake like the DoS attack. Hence, DoS attack is a subset of sequences contained in the set of possible sequences of replay attack.

6.2 CAHOOT algorithm

The CAHOOT algorithm aims to detect an intruder that performs both single or multiple attacks while a car is moving. It is also able to detect a possible intrusion also when both the intruder and the driver generated a CAN message with the same values.

CAHOOT uses machine learning (ML) techniques to generate a model capable to detect intrusions from the value of the vehicle sensors.

6.2.1 Intruder's Behaviour

To create a model that is as accurate as possible, we assume that the intruder is able to frequently change the attacks among the three attacks described in Section 6.1. The duration of each attack is randomly chosen with an arbitrary minimum and maximum of steps duration. In addition, the type of attack is randomly chosen, e.g., the attacker launches a replay attack followed by two consecutive spoofing attacks and then launches a DoS attack. This allows us to identify both single and multiple attacks within a target driving session.

Listing 8 and Listing 9 describe our model of the intruder's behaviour.

Listing 8: *Prepare Attack*

```
function prepare_attack($steering$, $throttle\_brake$, $current\_attack$, $steering\_history$,
    $throttle\_brake\_history$, $index\_history$, $prev\_steering$, $prev\_throttle\_brake$,
    $stop\_attack\_time$, $min\_duration$, $max\_duration$, $slot\_time$)
  $should\_attack\_change \leftarrow stop\_attack\_time <=$ Current timestamp

  if $should\_attack\_change$
      $num\_slots \leftarrow$ Select an integer number between $min\_duration$ and
          $max\_duration$
      $stop\_attack\_time \leftarrow$ Current timestamp $+ \, num\_slots * slot\_time$

      $current\_attack =$ None

  ($steering_{forged}$, $throttle\_brake_{forged}$, $current\_attack$, $index\_history$, $prev\_steering$,
      $prev\_throttle\_brake$) = launch_attack($current\_attack$, $steering\_history$,
      $throttle\_brake\_history$, $index\_history$, $prev\_steering$, $prev\_throttle\_brake$)

  $steering\_history \leftarrow$ Append $steering$ to $steering\_history$
  $throttle\_brake\_history \leftarrow$ Append $throttle\_brake$ to $throttle\_brake\_history$

  return ($steering_{forged}$, $throttle\_brake_{forged}$, $current\_attack$, $stop\_attack\_time$,
      $steering\_history$, $throttle\_brake\_history$, $index\_history$, $prev\_steering$,
      $prev\_throttle\_brake$)
```

Listing 8 shows the algorithm *prepare_attack* that plans the duration of each vehicle intrusion. In detail, it checks if the attack in progress should continue or should be changed, i.e., the algorithm compares the current time with the time on which the attack must be suspended (line 2). In case the attack should end and be changed with a new type, the algorithm defines the duration of the new attack as slots of time. The algorithm randomly choose the number of slots between the minimum and maximum (line 5). Hence, the attack will stop at the sum between the actual time and the product between the number of slots and the length of each slot (lines 6). The attacks are periodically stopped and substituted with new ones to simulate multiple attacks in a single driving session.

Regardless of the attack should change or not, the function *launch_attack* is called (line 10) and returns the new forged messages alongside with the current type of attack, the index of the next messages that the replay attack must repeat, i.e., *index_history*, and the last forged messages that the spoofing attack must repeat, i.e., *prev_steering* and *prev_throttle_brake*. Next, the inputs steering and the throttle_brake of human/AI are registered in the arrays *steering_history* and *throttle_brake_history* (lines 12 and 13). These arrays may be used later on for the replay attack. The attack inputs are never appended in the arrays because the replay attack goal is to mimic the human/AI inputs so the attack should replay only human/AI inputs.

As final step, the algorithm returns the values of steering and throttle_brake generated by the intruder, the type of attack actually in progress, the time on which the attack will be suspended, the history values of steering and throttle_brake, the *index_history*, *prev_steering* and *prev_throttle_brake* (line 15).

Listing 9: *Launch Attack*

```
function launch_attack(current_attack, steering_history, throttle_brake_history
    , index_history, prev_steering, prev_throttle_brake)
```

```
2   bootstrap ← False
3   if current_attack = None
4       bootstrap ← True
5
6       current_attack ← Randomly select one from "DoS", "Spoofing" and
            "Replay"
7
8   if current_attack = "DoS"
9       (steering, throttle_brake) ← dos_attack()
10  if current_attack = "Spoofing"
11      (steering, throttle_brake) ← spoofing_attack(bootstrap, prev_steering,
            prev_throttle_brake)
12
13      prev_steering ← steering
14      prev_throttle_brake ← throttle_brake
15  if current_attack = "Replay"
16      (steering, throttle_brake, index_history) ← replay_attack(bootstrap,
            steering_history, throttle_brake_history, index_history)
17
18  return (steering, throttle_brake, current_attack, index_history, prev_steering
        , prev_throttle_brake)
```

Listing 9 depicts the algorithm *launch_attack*. It is in charge of maintaining active and in progress attack or decide which attack should be run. The Spoofing and Replay attack need the variable *bootstrap* that represents if the attack is in progress or not, i.e., the variable tracks if a new attack must be launched or a previous attack must continue. The variable is *False* in case the attack is in progress (line 2) and *True* when the attack is not running (line 4). In case an attack is not in progress, the type of attack is randomly chosen between DoS, Spoofing and Replay (line 6). Once the *bootstrap* variable is established, based on the current attack value, an attack is launched (lines 8 to 16). Keep note that in case of spoofing attack, the *prev_steering* and *prev_throttle_brake* variables are updated with the most recent forged messages generated (lines 13 and 14).

Finally, the *launch_attack* returns the steering and throttle_brake values chosen by the attack, the current type of attack, the *index_history* selected by the Replay attack function last time it is launched and the previous pair of steering and throttle_brake used by the Spoofing attack (line 18).

Denial of Service Attack. The last attack function is the *dos_attack* that sets the *steering* and the *throttle_brake* to 0

Spoofing Attack. The *spoofing_attack* function set the steering and the throttle_brake with random values (Listing 10). In case the attack is not yet started, the algorithm randomly choose values between -1 and 1 (lines 2 to 4). In case the attack is in progress, the steering and the throttle_brake are the same values that the function set in the previous step (lines 5 to 7). Finally, the function returns the steering and throttle_brake values (line 9).

Listing 10: *Spoofing Attack*

```
1 function spoofing_attack(bootstrap, prev_steering, prev_throttle_brake)
2    if bootstrap = True
3        steering ← Choose randomly a float number between -1 and 1
4        throttle_brake ← Choose randomly a float number between -1 and 1
5    else
6        steering ← prev_steering
7        throttle_brake ← prev_throttle_brake
8
9    return (steering, throttle_brake)
```

Replay Attack. The *replay_attack* function repeat a sequence of steering and throttle_brake values previously seen and contained respectively in the arrays *steering_history* and *throttle_brake_history* (Listing 11). These arrays can be empty, i.e., previously human/AI inputs does not exist because the driving session is just started. In case the arrays are empty, there are no previous inputs to repeat so the *steering* and *throttle_brake* are set to 0 (lines 4 and 5). In case the arrays are not empty and the attack is not yet started, the algorithm randomly select from which position of the array start to repeat the previous input through the setting of *index_history* variable (line 8).

Whether or not the attack has already been launched, the algorithm use the *index_history* to select the steering and throttle_brake values from the respective arrays history (line 10 and 11). Then, the *index_history* variable is updated by 1 so, in case the attack continues, the *replay_attack* function will use in the following steering and throttle_brake of the history (line 13). Keep note that the *index_history* will never point to a non existing element of the arrays history because a new pair (steering, throttle_brake) will be added in the arrays history by the function *simulate_attack* at lines 12 and 13. This pair will contain the last input of the driver. Finally, *replay_attack* function returns the inputs and the new *index_history* (line 15).

Listing 11: *Replay Attack*

```
1 function replay_attack(bootstrap, steering_history, throttle_brake_history,
      index_history)
2    history_len ← Size of the array steering_history
3    if history_len = 0
4       steering ← 0
5        throttle_brake ← 0
6   else
7       if bootstrap = True
8            index_history ← Choose randomly an integer number between 0
                 and history_len-1
9
```

```
        $steering \leftarrow steering\_history$[$index\_history$]
        $throttle\_brake \leftarrow throttle\_brake\_history$[$index\_history$]

        $index\_history \leftarrow index\_history$ + 1

    return ($steering$, $throttle\_brake$, $index\_history$)
```

6.2.2 Instances Extraction Paradigm

To train the model for intrusion detection, CAHOOT requires a dataset, i.e., a collection of data that contains both legit and forged messages for each functionalities we aim to consider, i.e., $steering_{legit}$, $steering_{forged}$, $throttle\_brake_{legit}$ and $throttle\_brake_{forged}$, alongside with the sensors' values (Table 12).

The instances of the dataset are extracted to generate the final dataset on which the messages are organized in pairs and, each pair is labelled as T when it is composed by $steering_{legit}$ and $throttle\_brake_{legit}$ or as F in all the other cases (Table 13). The organization in pairs allows CAHOOT to detect possible intrusion that may happen when the intruder is going to send the same message sent by the driver. Let us suppose that the driver wants to go straight, i.e., $steering_{legit}$ is equal to 0, and the intruder starts a DoS attack, i.e., $steering_{forged}$ is equal to 0 (Table 12, row 3). The steering message sent by the intruder is considered as legit because it is equal to the driver's one. However, the algorithm raises an alert based on the values of $throttle\_brake_{legit}$ and $throttle\_brake_{forged}$ that should be different (Table 13, rows 9 and 10). On the other hand, if both the messages in the pair are equal (Table 12, row 4), for instance because the intruder is trying to perform a DoS attack, then CAHOOT only inserts into the dataset one instance labelled with T (Table 13, row 11). In this way, it prevents the DoS by discarding the flow of not legit messages.

Table 12: *Example of instances before run Instances Extraction Paradigm*

$timestamp$	$steering_{legit}$	$steering_{forged}$	$throttle\_brake_{legit}$	$throttle\_brake_{forged}$	...
01/01/2022 12:00:00.000	0,695	0,403	0,020	-0,001	...
01/01/2022 12:00:00.100	0,045	0,494	-0,042	-0,533	...
01/01/2022 12:00:00.200	0,0	0,0	-0,042	0,0	...
01/01/2022 12:00:00.300	0,0	0,0	0,0	0,0	...

Hence, on the initial dataset we run the *instances_extraction* function (Listing 12) whose output is the dataset $ins_{extracted}$ that contains the final created dataset.

As first step, the algorithm reads each *instance* of the initial dataset *ins* (line 3) to organize the messages in two arrays. The first array contains tuples composed by steering message alongside with a boolean value representing message's legitimacy. The second array contains tuples composed by throttle_brake message alongside with a boolean value representing message's legitimacy.

The two arrays are used to organize all the instances in the initial dataset in such a way that legit and forged messages are clearly distinguishable: the legit messages are inserted in the arrays (lines 12 and 13), while the forged messages are inserted only if they are other than the respective legit ones (lines from 15 to 18). From *instance* the messages $steering_{legit}$, $steering_{forged}$, $throttle\_brake_{legit}$ and $throttle\_brake_{forged}$ are removed (line 20). Thus, *instance* now contains the engine runtime and the sensors' values.

The algorithm creates several instances based on *instance*, one instance

Table 13: *Example of instances after run Instances Extraction Paradigm*

timestamp	*steering*	*throttle_brake*	...	*label*
01/01/2022 12:00:00.000	0,695	0,020	...	T
01/01/2022 12:00:00.000	0,695	-0,001	...	F
01/01/2022 12:00:00.000	0,403	0,020	...	F
01/01/2022 12:00:00.000	0,403	-0,001	...	F
01/01/2022 12:00:00.100	0,045	-0,042	...	T
01/01/2022 12:00:00.100	0,045	-0,533	...	F
01/01/2022 12:00:00.100	0,494	-0,042	...	F
01/01/2022 12:00:00.100	0,494	-0,533	...	F
01/01/2022 12:00:00.200	0,0	-0,042	...	T
01/01/2022 12:00:00.200	0,0	0,0	...	F
01/01/2022 12:00:00.300	0,0	0,0	...	T

per each combination of the steering and throttle_brake messages present respectively in *steering_array* and *throttle_brake_array* (lines 25 and 26). Then, each generated instance is labeled "T" in case it contains only messages from the driver or "F" in case it contains at least one message from the intruder (lines from 28 to 31). Next, each labeled instance is added to the $ins_{extracted}$ dataset (line 33). After all the instances present in *ins* are read, the algorithms return the dataset $ins_{extracted}$ (line 35).

Listing 12: *Instances Extraction Paradigm*

```
1 function instances_extraction($ins$)
2     $ins_{extracted}$ ← empty array
3     for each $instance$ in $ins$
4         $steering_{legit}$ ← $instance$["$steering_{legit}$"]
5         $steering_{forged}$ ← $instance$["$steering_{forged}$"]
6         $throttle\_brake_{legit}$ ← $instance$["$throttle\_brake_{legit}$"]
```

```
        $throttle\_brake_{forged} \leftarrow instance[“throttle\_brake_{forged}”]$

        $steering\_array \leftarrow$ empty array
        $throttle\_brake\_array \leftarrow$ empty array

        $steering\_array \leftarrow steering\_array \bigcup (steering_{legit},$ True)
        $throttle\_brake\_array \leftarrow throttle\_brake\_array \bigcup (throttle\_brake_{legit},$ True)

        if $steering_{legit}$ != $steering_{forged}$
            $steering\_array \leftarrow steering\_array \bigcup (steering_{forged},$ False)
        if $throttle\_brake_{legit}$ != $throttle\_brake_{forged}$
            $throttle\_brake\_array \leftarrow throttle\_brake\_array \bigcup (throttle\_brake_{forged},$
                False)

        remove from $instance$ the columns $“steering_{legit}”$, $“steering_{forged}”$,
            $“throttle\_brake_{legit}”$, $“throttle\_brake_{forged}”$


        for each ($steering$, $is\_steering\_legit$) in $steering\_array$
            for each ($throttle\_brake$, $is\_throttle\_brake\_legit$) in
                $throttle\_brake\_array$
                $instance[“steering”] \leftarrow steering$
                $instance[“throttle\_brake”] \leftarrow throttle\_brake$

                if $is\_steering\_legit$ == True and $is\_throttle\_brake\_legit$ == True
                    $instance[“label”] \leftarrow$ "T"
                else
                    $instance[“label”] \leftarrow$ "F"

                $ins_{extracted} \leftarrow ins_{extracted} \bigcup instance$

    return $ins_{extracted}$
```

6.2.3 Model Generation

The Model Generation paradigm uses the Instances Extraction paradigm to generate the training and the test datasets (Listing 13). Going more into detail, once the dataset is randomly split in a training set and a test set (line 2), the instances are extracted for the training and test (lines 3 and 4). We run the extraction paradigm separately on the training set and the test set to make sure that all combinations of steering and throttle_brake messages from the same original instance are not distributed between the training set and the test set, but remain in the same set. The appearance of extracted instances of the same original in both training and test sets causes a data leakage [46]. Data leakage happens when information present in the training set is unexpectedly present also in the test set.

Next, the best features are selected using a Feature Selection (FS) paradigm that ranks all features applying the *Gain Ratio* [76] (GR) approach (line 6). Those features with rank equal to zero are discarded (line 7). Finally, these features are passed to the ML algorithm which returns a trained model (line 9).

Listing 13: *Model Generation*

```
1  function generate_model($ins_{labelled}$)
2      ($ins_{train}$, $ins_{test}$) $\leftarrow$ split randomly the instances as training and testing sets from
           $ins_{labelled}$
3      $ins\_extracted_{train}$ $\leftarrow$ instances_extraction($ins_{train}$)
4      $ins\_extracted_{test}$ $\leftarrow$ instances_extraction($ins_{test}$)
5
6      $ranking$ $\leftarrow$ GR($ins\_extracted_{train}$)
7      $features_{>0}$ $\leftarrow$ discard features with rank $= 0$ from $ranking$
8
9      $model$ $\leftarrow$ MLAlgorithm($ins\_extracted_{train}$ with features $features_{>0}$)
10     return $model$
```

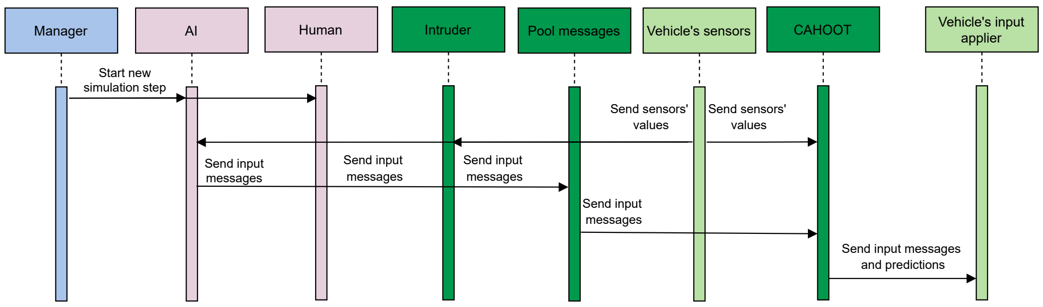


Figure 12: *Simulation sequence workflow of the vehicle*

6.3 CAHOOT Evaluation

To evaluate CAHOOT, we exploited the driving simulator MetaDrive [56]. It is a driving simulator written in Python to train a neural network for autonomous driving through Reinforcement Learning[57]. MetaDrive is able to generate infinite driving scenarios with procedural generation of maps and different traffic flows. Inside the simulator is present a pre-trained Artificial Intelligence (AI).

We modify the MetaDrive simulation workflow with the introduction of an intruder represented in dark green in Figure 12. The in vehicle communication is simulated by a set of messages made of two different Python lists: the first one contains the steering messages, instead the second list contains the throttle/brake messages sent. Both lists represent messages sent by the intruder and the driver. The intrusion workflow for each step of the simulation (Figure 12) works as follows:

- While the driver sends the inputs, an intruder forges fake messages of steering wheel and throttle/brake.
- The steering wheel and the throttle/brake messages of the intruder and the driver are sent to the set of messages.
- CAHOOT reads from the set the messages and establishes which ones are the legit messages and which ones are the forged messages.

- Steering wheel and throttle/brake messages from the set are transmitted to the wheels and the vehicle component responsible for applying the throttle/brake.
- The set of messages is emptied and ready to be filled with messages from the next step.

Keep note that even if in the intrusion workflow are present both the messages forged by the intruder and the messages legit, CAHOOT do not need both legit and forged messages for the detection phase. In case the intruder stops forging messages, CAHOOT would receive only the legit messages and establishes their legitimacy. On the other hand, CAHOOT requires both legit and forged messages for training phase to run the Instances Extraction Paradigm as described in section 6.2.2. Moreover, in the training and evaluation phases, the dataset always contains legit and forged messages, because the simulator creates both.

The dataset generated using the MetaDrive simulator contains the features in Table 14.

6.3.1 Machine Learning algorithms

The CAHOOT paradigm is implemented by using several Python libraries to implement different ML algorithms. We test several methods: Random Forest, J48, and Neural Network Multi-Layer Perceptron (MLP). Random Forest and J48 techniques do not require any settings of parameters. Even with the default ones, the performance obtained by these methods could be satisfactory. However, MLP requires that some parameters must be set and fine-tuned to obtain the best results, e.g., the architecture of the layers, the number of batches and so on.

We normalize training and test set to speed up the model training process using the z-score normalization [35] procedure: the values of each feature f

Table 14: *Features description*

Feature	Description	Example	Unit
Speed	Speed of the vehicle	55	km/h
Throttle_brake	Amount of throttle or braking	0,55	N/A
Steering	Rotation of the steering wheel	-0.25	N/A
Last_position_x/y	Position of the vehicle at coordinate x/y	125	N/A
Dist_to_left/right_side	Distance from the left-/right lane	0,423	m
Fuel_consumption	Fuel consumption since the start of the driving session	33,12	N/A
Engine_runtime_minute / second / millisecond	Minutes / seconds / milliseconds elapsed from engine start	39	minutes / s / ms
Yaw_rate	Angular acceleration on vertical axis	0.089661	N/A
Project_distance / velocity_to_vehicle_n_x / y	Vehicle's projection distance / velocity to the n-th nearest vehicle on coordinate x / y	0.187	N/A

are transformed based on the mean $\bar{v}_f$ and the standard deviation σ_f of f in the training set. The transformation is applied to the training set and test set using the same pair $(\bar{v}_f, \sigma_f)$. The equation applied is:

$$v_{new} = \frac{v_{old} - \bar{v}_f}{\sigma_f} \tag{9}$$

To improve the neural network performance, we use the embeddings for categorical values as explained in [77]. Categorical values are the engine runtime milliseconds, engine runtime seconds and the engine runtime minutes. The remaining features are continuous.

We then create data loaders for training set and test set with batches of size equal to 2048.

The architecture of the MLP contains 4 layers and the sizes of the hidden layers are respectively 2048, 1024 and 512. We then search the best learning rate using the algorithm LRFinder present in FastAI [87]. Finally, we use this learning rate in the model training. Based on the plot loss of training and test sets, we trained the model for 480 epochs. On each experiment will be shown the relative plot loss.

6.3.2 Experiments setup

The experiments run on a Virtual Machine with an Intel(R) Xeon(R) using 16 threads, 157 GB of RAM and CentOS Linux 7 as OS. To evaluate CAHOOT, in the experiments we use the metrics: *accuracy*, *precision* and *recall*, introduced in paragraph 4.2.2. In the IDS context, TP is the number of instances where at least one sensor's value is forged that are correctly predicted, TN is the number of instances where all the sensors' values are legit that are correctly predicted, FR is the number of instances where all the sensors' values are legit but incorrectly predicted and FN is the number of instances where at least one sensor's value is forged but incorrectly predicted.

We randomly split the dataset in a training set of 85% of instances and a test set of the remaining 15%. We have fed each ML method with the same training set and tested with the same test set. The dataset contains drivings made by an AI and 5 human drivers using a Thrustmaster TMX [95]. In the dataset are present 107 driving sessions made by humans. To demonstrate the validity of CAHOOT, we also simulated further human drivings using data augmentation techniques. Data augmentation are methods to generate synthetic patterns starting from a dataset[41].

While the driver is driving the simulated vehicle, the intruder sends *steering* and *throttle_brake* messages. We decided to simulate attacks with several success rates, i.e., 0%, 20% and 40%. Also, to simulate multiple attacks on each driving session, we set the maximum and minimum duration of an attack respectively to 2 and 1 slots.

We aim to detect the instances that contain at least one sensor's value forged from the *steering* and the *throttle_brake*.

6.3.3 Evaluation without data augmentation

In the following, we first evaluate CAHOOT training it by using the human and AI driving sessions. Then, the training is done by using only human driving sessions. Table 15 contains the list of features selected by CAHOOT. To better distinguish features rankings, each feature rank is shown as a percentage of the sum of all the ranks.

Training CAHOOT using human and AI drivings, the *steering* and *throttle_brake* messages are the most important features. The worse features are the distance from the right lane and the projection of velocity of the nearest vehicle in the y axis. The engine runtimes minutes and seconds are at the half of the table while the engine runtime milliseconds was discarded.

The MLP is trained with a learning rate of 0,00023. The plot loss (Figure

Table 15: *Features selected by CAHOOT (percentage of each rank with respect to the sum of the ranks of the features)*

Features	Rank percentage	
	Train Human and AI	**Train Human**
steering	46,7%	43,0%
throttle_brake	32,4%	37,3%
speed	7,4%	7,3%
yaw_rate	6,6%	5,6%
fuel_consumption	2,3%	2,0%
last_position_y	1,3%	1,2%
last_position_x	0,9%	0,9%
engine_runtime_minute	0,5%	0,2%
engine_runtime_second	0,5%	0,5%
dist_to_left_side	0,4%	1,1%
project_distance_to_vehicle_1_y	0,3%	-
dist_to_right_side	0,2%	0,5%
project_velocity_to_vehicle_0_y	0,2%	0,3%

13) shows that even if the validation loss converges earlier, the train loss converges around epoch 480. Note that the MLP results are about the model of the epoch that obtained the best accuracy. Hence, extra epochs only have an impact on the extra time to train the model.

In Tables 16 17 18, we make a comparison among Random Forest, J48 and MLP. When CAHOOT is trained using human and AI drivings, the table shows that Random Forest (Table 16) obtained the best accuracy while J48 (Table 17) obtained the worst accuracy. MLP (Table 18) is the most balanced model, obtaining similar precision and recall.

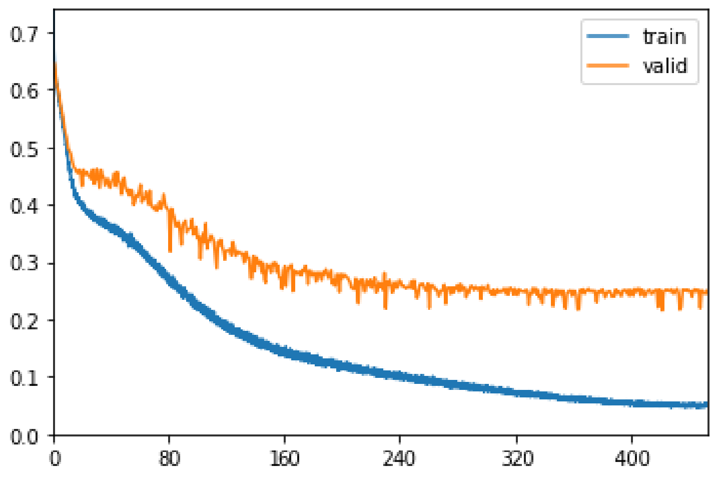


Figure 13: *Plot loss of the MLP trained using human and AI drivings.*

To better understand on which circumstances Random Forest best performs, we calculated the accuracy grouped by entity, i.e., human or the AI is driving the car, and by type of attack, i.e., DoS, spoofing and replay. Table 16 shows that the model has difficulty in the identification of the instances where the AI drives the car. On the other hand, the model has an excellent accuracy on instances where the human is driving. AI makes continuous and sudden driving adjustments, whereas humans tend to make gradual changes. Graduality makes human drivings predictions more accurate. The most difficult type of attack to identify is the replay attack while the spoofing is the most easiest to identify.

Because on human drivings the algorithm obtains high accuracy, we tried to improve the results by training and testing using only drivings made by humans.

Table 15 contains the list of features selected by CAHOOT. As in the previous experiment, the first six highest ranked features are *steering*, *throttle_brake*, *speed*, *yaw_rate*, *fuel_consumption* and *last_position_y*. However, engine runtimes seconds and minutes are no longer at the half of the table and ranking fourth to last and last respectively. Also, the projection with the

Table 16: *Accuracy, precision and recall comparison of CAHOOT using Random Forest*

Random Forest					
Acc	**Prec**	**Rec**	**Acc**	**Prec**	**Rec**
Train Human and AI drivers			**Train Human drivers**		
95,50%	95,98%	97,87%	97,03%	97,30%	98,60%
Test only human drivers					
97,25%	97,57%	98,64%	N/A		
Test only AI drivers					
82,70%	85,54%	92,46%	N/A		
Test only Replay attack					
93,36%	95,34%	95,46%	95,49%	96,69%	97,05%
Test only DoS attack					
96,26%	95,83%	98,90%	97,15%	97,15%	98,74%
Test only Spoofing attack					
96,73%	96,62%	99,11%	98,24%	97,91%	99,78%

Table 17: *Accuracy, precision and recall comparison of CAHOOT using J48*

J48					
Acc	**Prec**	**Rec**	**Acc**	**Prec**	**Rec**
Train Human and AI drivers			**Train Human drivers**		
90,43%	92,74%	94,14%	92,47%	94,13%	95,49%
Test only human drivers					
92,24%	93,82%	95,52%	N/A		
Test only AI drivers					
77,25%	84,97%	84,48%	N/A		
Test only Replay attack					
85,46%	91,33%	88,21%	88,37%	92,85%	90,81%
Test only DoS attack					
93,11%	93,25%	97,08%	93,75%	94,09%	96,98%
Test only Spoofing attack					
92,56%	93,45%	96,87%	94,94%	95,16%	98,25%

nearest vehicles obtained a rank equal to zero, except for the projection of velocity to the nearest vehicle in the y axis.

The MLP is trained with a learning rate of 0,00016 obtained using the LRFinder algorithm. The plot (Figure 14) shows that training and validation loss converge. In this case, Tables 16, 17 and 18 show that Random Forest obtained the best accuracy while J48 obtained the worst accuracy.

Testing only the human drivings, the model trained with both human and AI drivings obtains slightly better accuracy with respect to the model trained using only human drivings. The AI reacts almost instantly to intrusions

Table 18: *Accuracy, precision and recall comparison of CAHOOT using MLP*

MLP					
Acc	**Prec**	**Rec**	**Acc**	**Prec**	**Rec**
Train Human and AI drivers			**Train Human drivers**		
93,81%	95,74%	95,70%	95,30%	96,84%	96,63%
Test only human drivers					
95,73%	97,23%	96,83%		N/A	
Test only AI drivers					
79,86%	85,60%	87,80%		N/A	
Test only Replay attack					
90,32%	94,59%	91,83%	92,14%	95,55%	93,42%
Test only DoS attack					
94,70%	95,57%	96,85%	95,58%	96,68%	96,87%
Test only Spoofing attack					
96,12%	96,79%	98,07%	97,77%	97,97%	99,08%

making it the ideal driver. Thus, the model trained with also AI drivings is better at detecting legitimate messages.

Moreover, Table 16 shows that replay attack is the most difficult to recognize, while spoofing attack is the simplest to recognise with a recall nearly perfect, i.e., 99,78%.

We compare the scores on spoofing attack obtained by CAHOOT using Random Forest with the lowest and highest scores obtained in several experiments by the main context-aware IDSs present in the literature. Table 19 shows that CAHOOT trained with only humans obtained the second best

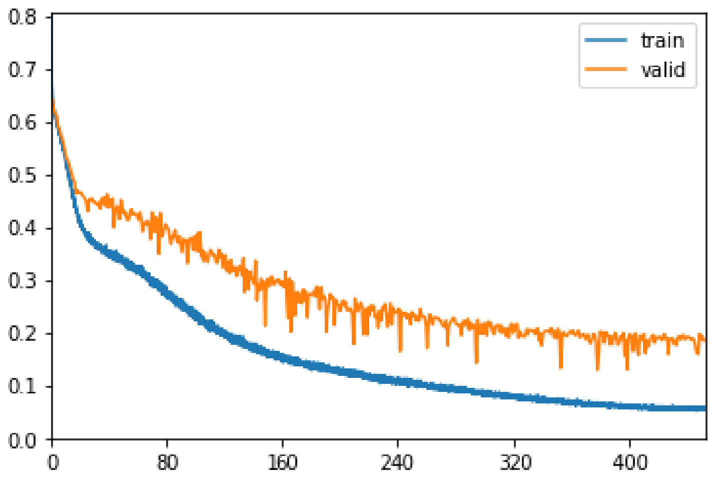


Figure 14: *Plot loss of the MLP trained using only human drivings.*

Table 19: *Comparison of lowest and highest accuracies on spoofing attack between CAHOOT and the main context-aware IDSs*

	CAHOOT (Human & AI - Human only)	**RAIDS** [42]	**Kondratiev et al.** [48]	**Casillo et al.** [12]
Accuracy	96,73% - 98,24%	89,5% - 99,9%	73,22% - 74,17%	96,64% - 96,91%
Precision	96,62% - 97,91%	N/A	N/A	83,00% - 87,00%
Recall	99,11% - 99,78%	N/A	N/A	78,38% - 82,18%

result. Overall, CAHOOT obtains the most consistent accuracies. Consider that in [101], the authors use a metric that is not comparable with the metric used in CAHOOT and the other previous works. In addition, each work uses a different dataset and features are not always present in all the datasets.

6.3.4 Evaluation with data augmentation

In the previous tests, we collected data from a limited number of drivers in a virtual environment. The best way to evaluate CAHOOT should be with a real car. However, driving sessions that involve several human drivers is an onerous and expensive task. In addition, the simulator allows us to launch intrusions that alter the behaviour of the car without endangering the drivers. Hence, we preferred run simulated drivings in a safe environment for the drivers and validate the simulations using data augmentation.

There are several data augmentation techniques on literature [41]. However, some techniques may produce dataset not realistic. For example, jittering, i.e. the application of noise to the dataset, may produce driving sessions in which the driver never comes to a complete stop at stop signs. To synthesize additional human driving sessions, we use data augmentation techniques which guarantee that at each driving session the fuel consumed by a vehicle since the start of the driving session can not decrease over time, i.e., at the i-th instance of the session $fuel\_consumption[i] \geq fuel\_consumption[i-1]$. Hence, we simulate additional human driving sessions using two data augmentation techniques: *time warping* uses a cubic spline that stretches or contracts the temporal dimension of the driving session [98], *window warping* stretches by 2 or contracts by $\frac{1}{2}$ a random window of the time series [53].

The procedure to generate the augmented test set is the following:

Preparation of the dataset for Data Augmentation. Since we want to augment human drivings to generate new synthetic human drivings, every AI drivings from the entire dataset are discarded (Listing 14, line 2). Either the training set and the test set contain forged messages of *steering* and *throttle_brake*. We do not need to augment the forged messages because we can simply randomly generate new forged messages. Hence, the forged messages are discarded and the resulting dataset saved in *dataset_legit* (line 4). New forged messages of the augmented data will subsequently be randomly generated. The data augmentation methods that will be used contract and/or stretches the time. Hence, even the features which represent the engine runtime will be consequently altered. The stretching and contracting is made by the data augmentation to generate new driving sessions that represent respectively a longer and smaller driving route in a time frame equal to the time before the data augmentation occurs. The engine runtime stretched and contracted will report to the machine learning method these time changes defeating the purpose of the data augmentation in the first place. To address this issue, the original engine runtime features are stored in a separate array and subsequently be used for the augmented datasets (line 6). Finally, the function returns the *dataset_legit*, the array of engine runtimes and the index of the driving sessions (line 8).

Listing 14: *Preparation for Data Augmentation*

```
1 function prepare_data_augmentation($dataset$, $test\_set$)
2     $dataset\_human$ ← from $dataset$ remove the instances on which the driver is an AI
3
4     $dataset\_legit$ ← from $dataset\_human$ remove the forged messages
5
6     $engine\_runtimes$ ← from $dataset\_legit$ get the columns engine runtime millisecond,
          second and minute
7
8     return ($dataset\_legit$, $engine\_runtimes$)
```

Augmentation of the dataset. The data augmentation method *augmented_function* will be applied to the original dataset, i.e., either training and test set. The augmented training set will be used to simulate the replay attacks, but will not be used to train the model. First, an array with the augmented datasets is created (Listing 15, line 2). To increase further the dataset, each data augmentation method can be performed multiple times (line 4). At each repetition, an array *dataset_augmented* that will contain the dataset augmented is created (line 5). Then, at each driving session is applied the function contained in *augmented_function* and the augmented driving session is appended to *dataset_augmented* (lines 6 to 9). Next, the augmented engine runtimes are substituted with the original engine runtimes (line 10). Then, the augmented dataset is added to the array of augmented datasets (line 12). Once the augmented function is repeated *repeat* times, the array of augmented datasets is returned (line 14).

Listing 15: *Data Augmentation*

```
1  function data_augmentation($dataset\_legit$, $engine\_runtimes$, $repeat$,
       $augmented\_function$)
2      $datasets\_augmented \leftarrow$ empty array
3
4      repeat $repeat$ times
5          $dataset\_augmented \leftarrow$ empty array
6          for each $dataset\_session$ in $dataset\_legit$
7              $dataset\_augmented\_session \leftarrow augmented\_function(dataset\_session)$
8
9              $dataset\_augmented \leftarrow dataset\_augmented \cup dataset\_augmented\_session$
10         apply $engine\_runtimes$ to $dataset\_augmented$
11
12         $datasets\_augmented \leftarrow datasets\_augmented \cup dataset\_augmented$
13
14     return $datasets\_augmented$
```

Insert of forged messages. The augmented instances that are part of the test set need forged messages. Hence, we will create a test set from each augmented dataset on which the instances have attached new forged messages. First, the array that will contains the augmented test set is created (Listing 16, line 2). To populate this array, the procedure must pick from each augmented dataset in *datasets_augmented* the corresponding augmented instance of each test set instance (lines 3 to 7). Note that the procedure use instances from the test set of human drivings only. Then, the index of the augmented instance is obtained and passed to the function *generate_intrusion* responsible for the generation of new forged messages for the augmented instance (lines 9 and 10). The function is explained later on. Then, the *instance_augmented_attacked* is defined as an empty instance that will contain the augmented values alongside with the forged messages (line 12). Next, the messages in $steering_{legit}$, $steering_{forged}$, $throttle\_brake_{legit}$ and $throttle\_brake_{forged}$ are inserted into the *instance_augmented_attacked* (lines from 14 to 17). Then, all the features present in the augmented instance are added into *instance_augmented_attacked* (line 18). Consider that *steering* and *throttle_brake* are already in *instance_augmented_attacked*. Now, *instance_augmented_attacked* contains all the augmented values and the forged messages. Hence, *instance_augmented_attacked* can be now appended to the augmented test set (line 20). Once all the augmented test set instances are appended, the original test set is appended to the augmented test set (line 22). Finally, the augmented test set is returned by the algorithm (line 24).

Listing 16: *Apply attacks on the augmented dataset*

```
1  function apply_attacks_augmented(datasets_augmented, test_set)
2      test_set_augmented ← empty array
3      for each dataset_augmented in datasets_augmented
4          for each instance in test_set
5              if instance driver is AI
6                  continue
7              instance_augmented ← get the augmented version of instance in
                   dataset_augmented
8
9              index ← index of the row instance_augmented
10             $steering_{forged}$, $throttle\_brake_{forged}$ ←
                   generate_intrusion(dataset_augmented, dataset_sessions_index, index)
```

```

        $instance\_augmented\_attacked$ ← empty instance

        $instance\_augmented\_attacked[\text{“}steering_{legit}\text{”}]$ ←
            $instance\_augmented[\text{“}steering\text{”}]$
        $instance\_augmented\_attacked[\text{“}steering_{forged}\text{”}]$ ← $steering_{forged}$
        $instance\_augmented\_attacked[\text{“}throttle\_brake_{legit}\text{”}]$ ←
            $instance\_augmented[\text{“}throttle\_brake\text{”}]$
        $instance\_augmented\_attacked[\text{“}throttle\_brake_{forged}\text{”}]$ ←
            $throttle\_brake_{forged}$
        $instance\_augmented\_attacked$ ← insert all the features in
            $instance\_augmented$ except features “$steering$”, “$throttle\_brake$”

        $test\_set\_augmented$ ← $test\_set\_augmented \cup instance\_augmented\_attacked$

    $test\_set\_augmented$ ← $test\_set\_augmented \cup test\_set$

    return $test\_set\_augmented$
```

The last function to explain is *generate_intrusion* (Listing 17). The function randomly generates forged messages for the augmented test set instance. First, an attack between DoS, replay and spoofing attacks is randomly chosen (line 2). Then, the chosen attack is launched (lines from 4 to 14). In particular in the spoofing attack (line 6), the attack is launched with input values “True” and the zeroes (line 7). The “True” value represents *bootstrap* which ensure the function choose randomly a steering and a throttle_brake values. The zeroes are respectively *prev_steering* and *prev_throttle_brake* which can be set to any value because they are overwritten by the spoofing attack function. In case the attack is Replay attack (line 8), the arrays with driver's history of previous steering and throttle_brake values must be built. The procedure determines the index start of the current driving session based on the variable *index*, i.e., the index of the augmented test instance (lines 9). The previous augmented instances for the current driving session are the instances starting from the instance with the index *start_index* and ending with the instance that has index *index* minus 1. Then, the procedure collects the previous steering and throttle_brake values of the augmented instances for the current driving session (lines 11 and 12). Next, the procedure executes the function "replay_attack" and returns the result (line 14). The input values "True" and 0 of the function are respectively the *bootstrap* and the *index_history*. The *bootstrap* ensures that the function choose randomly an instance index from the current driving session. The *index_history* can be set to any value because is overwritten by the replay function.

Listing 17: *Generate Intrusion*

```
1  function generate_intrusion(dataset_augmented, index)
2     attack_chosen ← choose randomly an attack between spoofing_attack, dos_attack,
          replay_attack
3
4     if attack_chosen == dos_attack
5         return dos_attack()
6     else if attack_chosen == spoofing_attack
7         return spoofing_attack(True, 0, 0)
8     else
9         start_index ← get the first index of the driving session that has index
10
11        steering_history = dataset_augmented[start_index:index−1][steering]
12        throttle_brake_history = dataset_augmented
              [start_index:index−1][throttle_brake]
13
14        return replay_attack(True, steering_history, throttle_brake_history, 0)
```

We evaluate CAHOOT using a test set augmented by 3x, 5x, 7x and 9x, i.e., the test set is made by the original test set and the augmented test sets using the window and time warp methods repeated respectively 1 time, 2 times, 3 times and 4 times. The number of human driving session presents in the test sets augmented by 3x, 5x, 7x and 9x are respectively 321, 535, 749 and 963 human driving sessions. We use Random Forest as ML algorithm because it obtained the best accuracy in all the previous tests.

In the first experiment, CAHOOT is trained using the human drivings and the AI drivings (Figure 15).

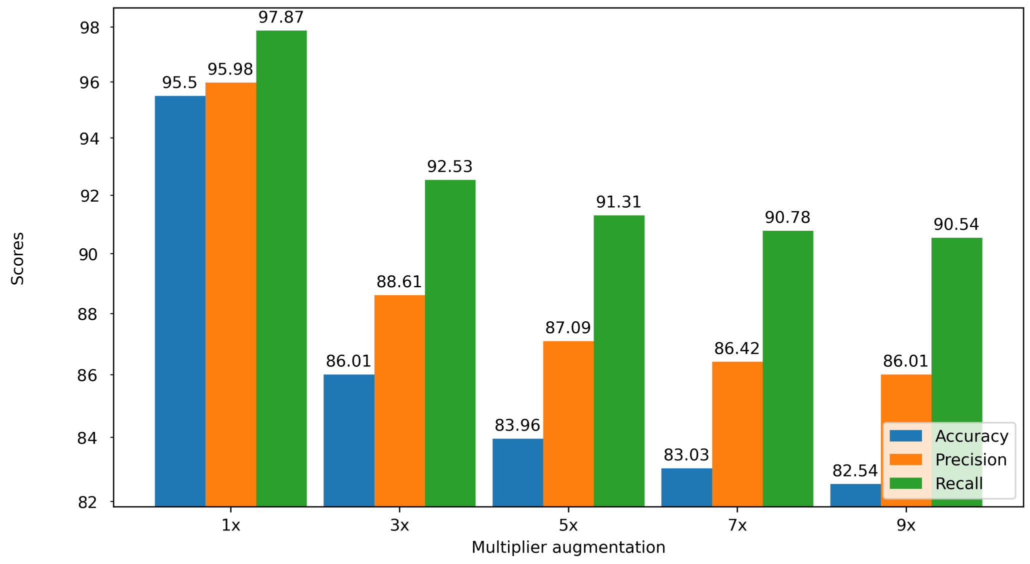


Figure 15: *Accuracy, precision and recall comparison of Attack Identification test bed with test set augmented.*

The bar plot shows that the test set without data augmentation, i.e., 1x, loose 9,49% of accuracy with the respect to the test set augmented 3x. The use of data augmentation amplify noises present on the dataset which leads to a deterioration in identification. On the other hand, the accuracies, the precisions and recalls of the 5x, 7x and 9x are similar. Hence, CAHOOT's accuracy degradation is less affected by noise as the dataset grows. The attack type that obtained the lowest accuracy is replay attack with a minimum accuracy of 75,08% and a maximum of 79,92%. The attack type with the highest accuracy is spoofing attack with an accuracy that ranges between 86,72% and 89,39%.

In the last experiment, CAHOOT is trained and tested using only the human drivings (Figure 16).

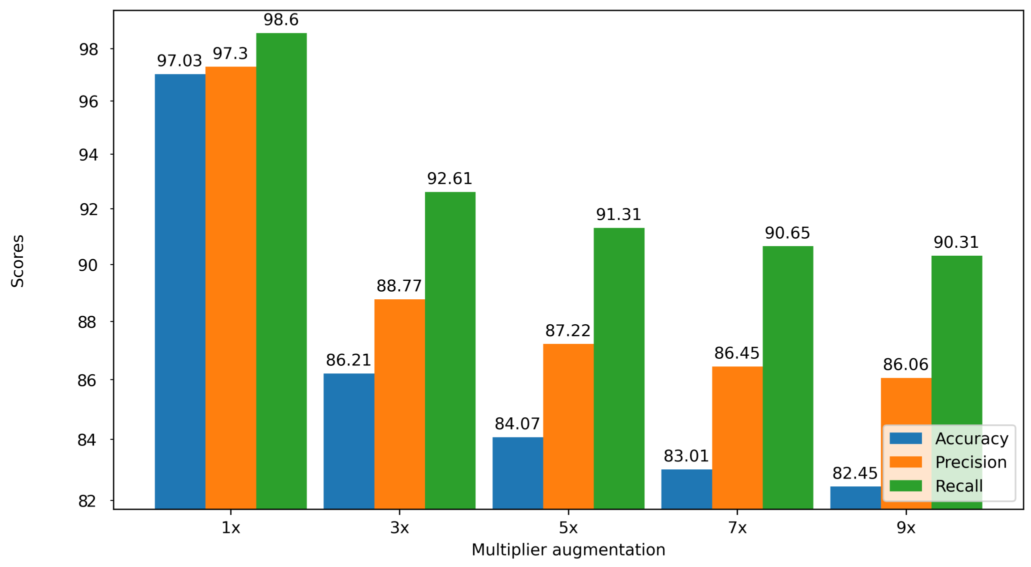


Figure 16: *Accuracy, precision and recall comparison of Attack Identification test bed trained using only human drivings with test set augmented.*

The bar plot shows that the test set without data augmentation, i.e., 1x, loose 10,82% of accuracy with the respect to the test set augmented 3x. Accuracies, precisions and recalls measures of the 5x, 7x and 9x are similar as previously seen in the previous experiments. The attack type that obtained the lowest accuracy is replay attack with a minimum accuracy of 74,87% and a maximum of 80,26%. On the other hand, the attack with the highest accuracy is once again the spoofing attack in a range between 86,76% and 89,86%.

7 CAHOOTv2

In this section, we present CAHOOTv2 as an improvement of CAHOOT. The advantages of CAHOOTv2 with respect to CAHOOT are twofold:

- CAHOOTv2 is trained to detect two variants of spoofing attack.
- CAHOOTv2 improves intrusion detection accuracies compared to CAHOOT. The Machine Learning algorithms presents parameters that must be set before the training process starts and may influence the generated model. These parameters are called hyperparameters [104]. The process of searching the hyperparameters that improve the performance of the models is called hyperparameters tuning [104]. In CAHOOTv2, we design a paradigm that select the best hyperparameters to use.

Finally, to validate the performance of the CAHOOTv2, we expanded the dataset collecting driving data from 39 humans.

7.1 Attacks

In CAHOOTv2, we consider an intruder able to perform the following attacks:

- *DoS* attack: the intruder is able to deny the driver's input through the generation of CAN frames where payloads values are set to zero for steering, throttle and brakes.
- *Replay* attack: the intruder is able to re-use valid CAN frames with a malicious or fraudulent aim.
- *Spoofing* attack: the intruder is able to generate a valid CAN frame. For example, the forged frame may generate a valid signal to active an ECU functionality.

- *Additive* attack: the intruder is able to use the current valid CAN frame payload and add a random value in $\pm[0.2, 0.9]$.
- *Selective* attack: the intruder is able to use the current valid CAN frame payload and flip the sign if the payload absolute value is greater than 0.3 or add a random value in $\pm[0.5, 1]$.

Additive and selective attacks are originally presented in the work [42]. These attacks are spoofing attack generation strategies because each attack needs the generation of a random value.

7.2 CAHOOTv2 algorithm

The CAHOOTv2 algorithm is built on top of CAHOOT. CAHOOTv2 aims to detect more attacks and increases the accuracy on the older ones. In the following, we describe the pseudocodes of the new attacks and how we integrate them on the intruder's behaviour. Then, we explain the paradigm responsible for improving accuracy.

7.2.1 Intruder's Behaviour

Listing 18 and Listing 19 describe our model of the intruder's behaviour.

Listing 18: *Prepare Attack*

```
function prepare_attack(steering, throttle_brake, current_attack, steering_history,
        throttle_brake_history, index_history, prev_steering, prev_throttle_brake,
        stop_attack_time, min_duration, max_duration, slot_time)
    should_attack_change ← stop_attack_time <= Current timestamp

    if should_attack_change
        num_slots ← Select an integer number between min_duration and
                max_duration
        stop_attack_time ← Current timestamp + num_slots * slot_time
```

```
7
8           $current\_attack$ = None
9
10      ($steering_{forged}$, $throttle\_brake_{forged}$, $current\_attack$, $index\_history$, $prev\_steering$,
            $prev\_throttle\_brake$) = launch_attack($steering$, $throttle\_brake$, $current\_attack$,
            $steering\_history$, $throttle\_brake\_history$, $index\_history$, $prev\_steering$,
            $prev\_throttle\_brake$)
11
12      $steering\_history$ ← Append $steering$ to $steering\_history$
13      $throttle\_brake\_history$ ← Append $throttle\_brake$ to $throttle\_brake\_history$
14
15      return ($steering_{forged}$, $throttle\_brake_{forged}$, $current\_attack$, $stop\_attack\_time$,
            $steering\_history$, $throttle\_brake\_history$, $index\_history$, $prev\_steering$,
            $prev\_throttle\_brake$)
```

Listing 18 shows the algorithm *prepare_attack* that plans the duration of each vehicle intrusion. The algorithm is the same of the prepare_attack presented in CAHOOT except for line 10 where steering and the throttle_brake of human/AI are sent to the function launch_attack. These values may be used to perform an additive or selective attack.

Listing 19: *Launch Attack*

```
1  function launch_attack($steering_{legit}$, $throttle\_brake_{legit}$, $current\_attack$,
       $steering\_history$, $throttle\_brake\_history$, $index\_history$, $prev\_steering$,
       $prev\_throttle\_brake$)
2      $bootstrap$ ← False
3      if $current\_attack$ = None
4          $bootstrap$ ← True
5
6          $current\_attack$ ← Randomly select one from "DoS", "Spoofing",
               "Replay", "Additive", "Selective"
7
8      if $current\_attack$ = "DoS"
```

```
9          ($steering$, $throttle\_brake$) ← dos_attack()
10     if $current\_attack$ = "Spoofing"
11         ($steering$, $throttle\_brake$) ← spoofing_attack($bootstrap$, $prev\_steering$,
               $prev\_throttle\_brake$)
12
13         $prev\_steering$ ← $steering$
14         $prev\_throttle\_brake$ ← $throttle\_brake$
15     if $current\_attack$ = "Replay"
16         ($steering$, $throttle\_brake$, $index\_history$) ← replay_attack($bootstrap$,
               $steering\_history$, $throttle\_brake\_history$, $index\_history$)
17     if $current\_attack$ = "Additive"
18         ($steering$, $throttle\_brake$) ← additive_attack($steering_{legit}$,
               $throttle\_brake_{legit}$)
19     if $current\_attack$ = "Selective"
20         ($steering$, $throttle\_brake$) ← selective_attack($steering_{legit}$,
               $throttle\_brake_{legit}$)
21
22     return ($steering$, $throttle\_brake$, $current\_attack$, $index\_history$, $prev\_steering$
           , $prev\_throttle\_brake$)
```

Listing 19 depicts the algorithm *launch_attack*. It is in charge of maintaining active and in progress attack or decide which attack should be run. In CAHOOTv2, launch_attack should randomly choose an attack between DoS, spoofing, replay, additive and selective (line 6). The additive and selective attacks need the legit *steering* and *throttle_brake* and apply to them mathematical operations to generate forged *steering* and *throttle_brake* (lines from 17 to 20).

New Attacks. Additive and selective attack add a random value to the steering and the throttle_brake of the user. The sum operation may lead to a value that is not valid. Function *limit_value* (Listing 20) ensures that values greater than the *upper_bound* are changed in *upper_bound* (lines 5 and 6) and values lower than the *lower_bound* are changed in *lower_bound* (lines 7 and 8). In case the value is in [*lower_bound*, *upper_bound*], the function returns the value as it is (line 10). In Metadrive, *upper_bound* and *lower_bound* are respectively 1 and -1.

Listing 20: *Limit value*

```
1  function limit_value(value)
2      upper_bound ← maximum acceptable value
3      lower_bound ← minimum acceptable value
4
5      if value > upper_bound:
6          return upper_bound
7      if value < lower_bound:
8          return lower_bound
9
10     return value
```

The *additive_attack* function set the steering and the throttle_brake with random values (Listing 21). First, two values are randomly generated in $\pm[0.2, 0.9]$ (lines 2 and 3). Then, these values are added to the legit steering and throttle_brake. Next, steering and throttle_brake are sent as input to the limit_function (lines 8 and 9). Finally, the function returns the limited steering and throttle_brake values (line 11).

Listing 21: *Additive Attack*

```
1  function additive_attack($steering_{legit}$, $throttle\_brake_{legit}$)
2      random_value_1 ← random value in $\pm[0.2, 0.9]$
3      random_value_2 ← random value in $\pm[0.2, 0.9]$
4
5      steering ← $steering_{legit}$ + random_value_1
6      throttle_brake ← $throttle\_brake_{legit}$ + random_value_2
7
```

```
    $steering_{limited}$ ← limit_value($steering$)
    $throttle\_brake_{limited}$ ← limit_value($throttle\_brake$)

    return ($steering_{limited}$, $throttle\_brake_{limited}$)
```

The *selective_attack* function create a steering and throttle_brake pair based on the value of the legit ones (Listing 22). In case, the legit steering is in $\pm[0, 0.3]$, a random value in $\pm[0.5, 1]$ is added to the legit steering (lines from 2 to 4). In case the legit steering is not in $\pm[0, 0.3]$, the forged steering is the legit one with the sign flipped (lines 5 and 6). Similarly, the forged throttle_brake is generated (lines from 8 to 12). Then, limit_value is launched on steering and throttle_brake (lines 14 and 15). Finally, the limited forged steering and throttle_brake are returned (line 17).

Listing 22: *Selective Attack*

```
1  function selective_attack($steering_{legit}$, $throttle\_brake_{legit}$)
2      if $steering_{legit}$ in $\pm[0, 0.3]$
3          $random\_value$ ← random value in $\pm[0.5, 1]$
4          $steering$ ← $steering_{legit}$ + $random\_value$
5      else
6          $steering$ ← -$steering_{legit}$
7
8      if $throttle\_brake_{legit}$ in $\pm[0, 0.3]$
9          $random\_value$ ← random value in $\pm[0.5, 1]$
10         $throttle\_brake$ ← $throttle\_brake_{legit}$ + $random\_value$
11     else
12         $throttle\_brake$ ← -$throttle\_brake_{legit}$
13
14     $steering_{limited}$ ← limit_value($steering$)
15     $throttle\_brake_{limited}$ ← limit_value($throttle\_brake$)
16
17     return ($steering_{limited}$, $throttle\_brake_{limited}$)
```

7.2.2 Hyperparameters Tuning Paradigm

Listing 23 and Listing 24 describe how the model is trained using the best hyperparameters. The Model Generation paradigm in CAHOOTv2 differs with respect to the paradigm in CAHOOT starting from line 9: from the train and test sets the worst features are removed (lines 9 and 10). Then, the *hyperparameters_tuning* function is called (line 12). Next, a random forest classifier is initialized using the hyperparameters received (line 14). Finally, a random forest model is trained using the train dataset ins_bf_{train} which returns a trained model (line 16).

Listing 23: *Model Generation*

```
1  function generate_model($ins_{labelled}$, $num\_iterations$, $cross\_validation$,
       $params\_dist_{random\_search}$)
2      ($ins_{train}$, $ins_{test}$) $\leftarrow$ split randomly the instances as training and testing sets from
           $ins_{labelled}$
3      $ins\_extracted_{train}$ $\leftarrow$ generate_dataset($ins_{train}$)
4      $ins\_extracted_{test}$ $\leftarrow$ generate_dataset($ins_{test}$)
5
6      $ranking$ $\leftarrow$ GR($instances$)
7      $features_{>0}$ $\leftarrow$ discard features with rank = 0 from $ranking$
8
9      $ins\_bf_{train}$ $\leftarrow$ $ins\_extracted_{train}$ with features $features_{>0}$
10     $ins\_bf_{test}$ $\leftarrow$ $ins\_extracted_{test}$ with features $features_{>0}$
11
12     $params_{best}$ $\leftarrow$ hyperparameters_tuning($ins\_bf_{train}$, $ins\_bf_{test}$, $num\_iterations$,
           $cross\_validation$, $params\_dist_{random\_search}$)
13
14     $rf$ $\leftarrow$ initialize a Random Forest using $params_{best}$
15
16     $model$ $\leftarrow$ train $rf$ using $ins\_bf_{train}$
17     return $model$
```

Listing 24 depicts *hyperparameters_tuning* paradigm. Because there are several possible combinations of hyperparameters, it is not feasible to try all the possible combinations to find the best one. In the first phase, the paradigm creates several random forests with random combinations of hyperparameters and searches a subset of the best hyperparameters (lines from 2 to 23). Then, try every combinations of hyperparameters present in the subset to find the hyperparameters with the best accuracy (lines from 25 to 34). Each combination is tested using the cross validation technique to ensure that the hyperparameters are valid for the entire dataset and not only for a specific test set. The random forests generated in the first phase are trained and tested using the training dataset. Instead, in the second phase the random forests are trained using train and test set. Although the first phase is performed on a limited number of hyperparameter combinations, this phase is computationally very onerous especially for large datasets. We apply the first phase only to the train set, without the test set, to speed up the computation.

In the following, we explain in detail the first and the second phase. In the inputs of *hyperparameters_tuning* is present $params\_dist_{random\_search}$, a bi-dimensional array that contains for each type of hyperparameter a list of possible values that should be tried by the paradigm. First, the paradigm creates a list with the name of the hyperparameters that will be tested (line 2). Then, the array $params\_accuracies_{random\_search}$, that will contain the pairs of hyperparameters chosen and the accuracy obtained by the random forest algorithm, is defined (line 4). The $num\_iterations$ variable defines how many random combinations of hyperparameters are tested in the first phase.

To generate a combination of hyperparameters, a list of hyperparameters is created (from line 6 to 9). For each type of hyperparameter present in $params\_dist_{random\_search}$, an hyperparameter is uniformly randomly chosen between all the possible values. Then, a random forest rf is initialized using the hyperparameters chosen. Next, rf is trained using the cross validation technique on the training dataset with the best features. The $cross\_validation$ variable defines the number of folds. The average accuracy is registered, alongside the list of the hyperparameters, in $params\_accuracy$[“accuracy”] (lines 9 and 12). Then, the tuple is appended to the array of pairs hyperparameters/accuracy $params\_accuracies_{random\_search}$ (line 14).

Once the $params\_accuracies_{random\_search}$ is populated, the paradigm looks for a subset of the best features. First, the array $params\_dist_{exhaustive\_search}$ that will contain the subset of the best hyperparameters is defined (line 16). Then, the paradigm selects the best hyperparameters of each type. For each hyperparameter type $param_{name}$, the accuracies present in $params\_accuracies_{random\_search}$ are grouped by $param_{name}$ to obtain the average accuracy of each group (line 18). Then, the third quartile [58] is calculated on the average accuracies of the groups (line 20). The hyperparameters that have an average accuracies greater or equal to the third quartile are inserted in $params\_dist_{exhaustive\_search}$ (line 23). Hence, about 25 percent of the highest accuracies are selected for each type of hyperparameter.

Next, the train and test set are combined to obtain the entire dataset (line 25). The variables that will contain the best hyperparameters and the relative accuracy are defined (lines 27 and 28). Then, each possible combination of hyperparameters in $params\_dist_{exhaustive\_search}$ is tested using cross validation on the entire dataset (lines from 29 to 31). In case the accuracy obtained is greater than the actual one present in $accuracy_{best}$, the best hyperparameters and accuracy variables are updated (lines from 32 to 34). Finally, the paradigm returns the hyperparameters that obtained the best accuracy.

Listing 24: *Hyperparameters Tuning Paradigm*

```
1 function hyperparameters_tuning($ins\_bf_{train}$, $ins\_bf_{test}$, $num\_iterations$,
    $cross\_validation$, $params\_dist_{random\_search}$)
$params_{name}$ ← get the names of parameters in $params\_dist_{random\_search}$

$params\_accuracies_{random\_search}$ ← empty array
for $num\_iterations$:
    $params$ ← Empty array
    for each $param_{name}$ in $params_{name}$:
        $params[param_{name}]$ ← choose uniformly random a hyperparameter
            in $params\_dist_{random\_search}[param_{name}]$
    $params\_accuracy$["params"] ← $params$

    $rf$ ← initialize a Random Forest with $params$ as hyperparameters
    $params\_accuracy$["accuracy"] ← train $rf$ using $cross\_validation$-fold
        cross validation applied to $ins\_bf_{train}$

    $params\_accuracies_{random\_search}$ ← append $params\_accuracy$ in
        $params\_accuracies_{random\_search}$

$params\_dist_{exhaustive\_search}$ ← empty array
for each $param_{name}$ in $params_{name}$:
    $grouped\_accuracies$ ← group $params\_accuracies_{random\_search}$ by $param_{name}$
        and calculate the average accuracy of each group

    $third\_quartile$ ← calculate the third quartile on $grouped\_accuracies$[
        "accuracy"]

    $params\_accuracies_{best\_subset}$ ← get the elements in $grouped\_accuracies$
        on which $grouped\_accuracies$["accuracy"] $\geq$ $third\_quartile$
    $params\_dist_{exhaustive\_search}[param_{name}]$ ← $params\_accuracies_{best\_subset}$[
        "params"]

$ins\_bf$ ← $ins\_bf_{train} \cup ins\_bf_{test}$

$params_{best}$ ← None
```

```
$accuracy_{best}$ ← 0
for each $params$ combination in $params\_dist_{exhaustive\_search}$:
    $rf$ ← initialize a Random Forest with $params$ as hyperparameters
    $accuracy$ ← train $rf$ using $cross\_validation$-fold cross validation
        applied to $ins\_bf$
    if $accuracy > accuracy_{best}$:
        $params_{best}$ ← $params$
        $accuracy_{best}$ ← $accuracy$

return $params_{best}$
```

7.3 Dataset

As in CAHOOT, the dataset is generated using the driving simulator MetaDrive. The dataset contains data made by an AI and 39 humans. In particular, one human uses a keyboard while the remaining 38 use a Thrustmaster TMX [95]. Regarding the gender of the drivers, four drivers are females while the remaining 35 are males.

Figure 17 shows the ages grouped by the gender. Female drivers ages are between 19 and 27 in average 22,25 years old and median of 21,5 while male drivers ages are between 20 and 44 in average 24 years old and median of 22. Overall, the drivers' ages are between 19 and 44 with an average of 23,82 and median of 22.

The histogram of the ages (Figure 18) shows that the majority of the drivers are 21 and 22 years old.

7.4 CAHOOTv2 Evaluation

The sequence workflow of MetaDrive and the features of the dataset are the same present in CAHOOT (Figure 12 and Table 14).

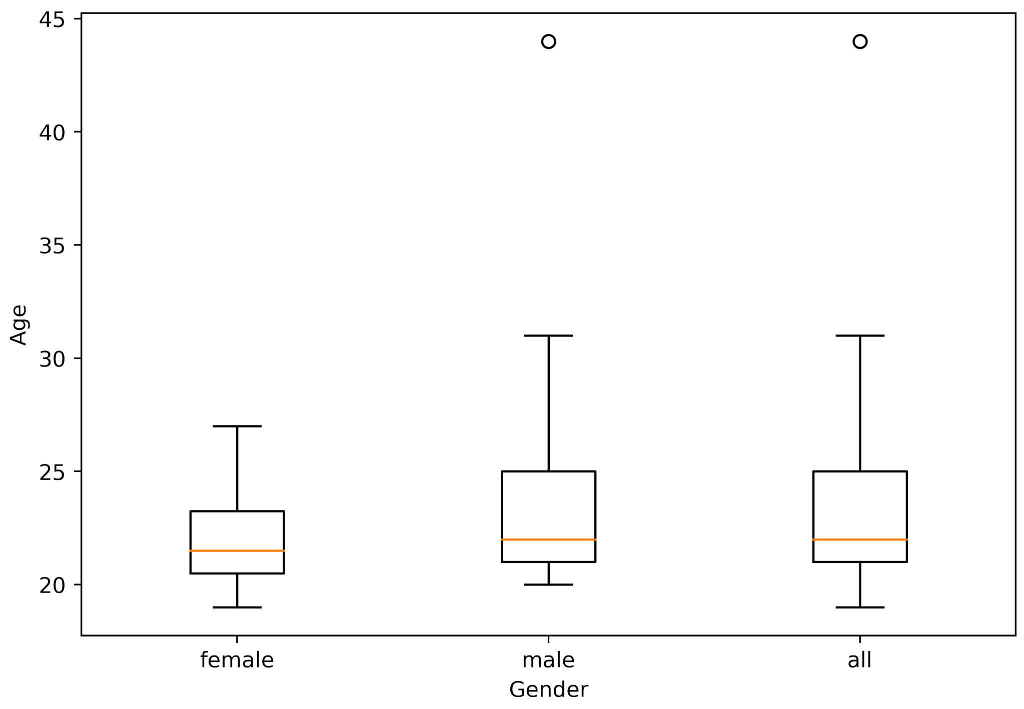


Figure 17: *Boxplots of genders.*

7.4.1 Machine Learning algorithms

The CAHOOTv2 paradigm is implemented by using python-weka-wrapper3 [79] for the feature selection algorithm GainRatio and scikit-learn [71] that efficiently implements Random Forest [10].

Models generated using Random Forest technique obtain good results. However, tuning the hyperparameters, RF is able to achieve the best results. In the first experiment, we run the *hyperparameters_tuning* paradigm on CAHOOT algorithm with the dataset present in Chapter 6.3, hereafter called α. In the second experiment, we run CAHOOT and CAHOOTv2 on the dataset presented in the previous subsection, hereafter called β. Finally, we compare the result on the new attacks of CAHOOTv2 with the related work.

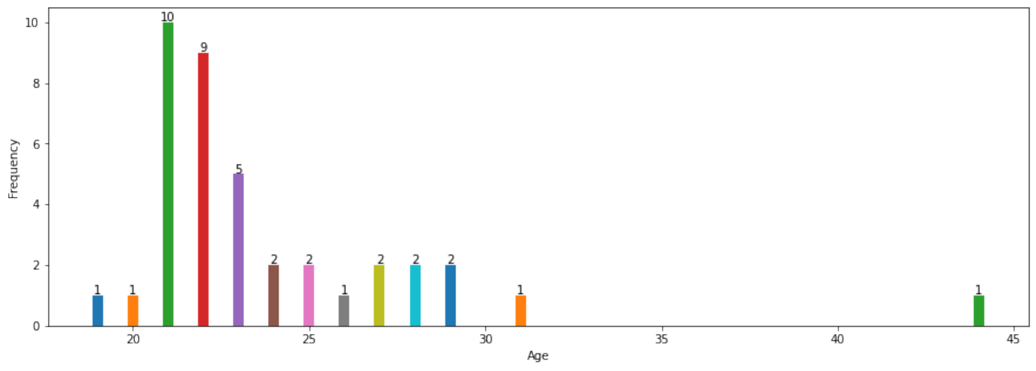


Figure 18: *Histogram of the ages.*

7.4.2 Experiments setup

The experiments run on the same environment used in CAHOOT. To evaluate CAHOOTv2, in the experiments we use several metrics: *accuracy*, *precision* and *recall*.

We randomly split the dataset in a training set of 85% of instances and a test set of the remaining 15%.

While the driver is driving the simulated vehicle, the intruder sends *steering* and *throttle_brake* messages. Also, to simulate multiple attacks on each driving session, we set the maximum and minimum duration of an attack respectively to 2 and 1 slots.

Table 20 shows the hyperparameters that we test in *hyperparameters_tuning* paradigm. We use 100 as number of iterations in the first phase.

We aim to detect the instances that contain at least one sensor's value forged from the *steering* and the *throttle_brake*.

7.4.3 Evaluation of *hyperparameters_tuning*

In the following, we evaluate the model trained using the default hyperparameters with the model trained using the best hyperparameters. The experiment is conducted on the same train and test set on dataset α.

Table 20: *Hyperparameters tested in hyperparameters_tuning paradigm*

Hyperparameter	Description	Values
num_estimators	The number of trees that make up the forest	[100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
max_features	The number of features considered for the split	["log2", "sqrt"]
min_samples_split	The minimum number of samples required to split an internal node	[2, 7, 12, 18, 23, 28, 34, 39, 44, 50]
min_samples_leaf	The minimum number of samples required to be at a leaf node	[1, 6, 11, 17, 22, 28, 33, 39, 44, 50]
bootstrap	Whether to use the entire dataset to build each tree or a bootstrap sample	[true, false]
criterion	The function used to measure the quality of a split	["gini", "entropy"]

Table 21: *Features selected by CAHOOT on α (percentage of each rank with respect to the sum of the ranks of the features)*

Features	**Rank percentage**
steering	46,7%
throttle_brake	32,4%
speed	7,4%
yaw_rate	6,6%
fuel_consumption	2,3%
last_position_y	1,3%
last_position_x	0,9%
engine_runtime_minute	0,5%
engine_runtime_second	0,5%
dist_to_left_side	0,4%
project_distance_to_vehicle_1_y	0,3%
dist_to_right_side	0,2%
project_velocity_to_vehicle_0_y	0,2%

In this dataset, we decided to simulate attacks with several success rates, i.e., 0%, 20% and 40%.

Table 21 contains the list of features selected for the two models. To better distinguish features rankings, each feature rank is shown as a percentage of the sum of all the ranks.

The *steering* and *throttle_brake* messages are the most important features. The worse features are the distance from the right lane and the projection of velocity of the nearest vehicle in the y axis. The engine runtimes minutes and seconds are at the half of the table while the engine runtime milliseconds was discarded.

Table 22, shows that the search of hyperparameters increase the accuracy of 1,5%. The recall is 0,3% lower than the model trained with the best hyperparameters, but the precision is 0,9% higher, i.e., the false negative are slightly increased but false positive are decreased.

To better understand on which circumstances the customized hyperparameters best perform, we calculated the accuracy grouped by entity, i.e., human or the AI is driving the car, and by type of attack, i.e., DoS, spoofing and replay. The model trained with custom hyperparameters is 1,2% more accurate with respect to the model trained with default hyperparameters on the AI drivings. The attack that obtains the best accuracy increase is spoofing attack, i.e., 0,7%. On the other hand, the accuracy of replay attack increases only of 0,1%.

Table 22: *Accuracy, precision and recall comparison on α of CAHOOT with default and best hyperparameters*

CAHOOT with best hyperparameters			**CAHOOT with default hyperparameters**		
Accuracy	**Precision**	**Recall**	**Accuracy**	**Precision**	**Recall**
96%	96,9%	97,6%	95,5%	96,0%	97,9%
Test only human drivers					
97,6%	98,2%	98,5%	97,2%	97,6%	98,6%
Test only AI drivers					
83,9%	88,1%	90,7%	82,7%	85,5%	92,5%
Test only Replay attack					
93,5%	96,2%	94,8%	93,4%	95,3%	95,5%
Test only DoS attack					
96,8%	96,6%	98,8%	96,3%	95,8%	98,9%
Test only Spoofing attack					
97,4%	97,7%	98,9%	96,7%	96,6%	99,1%

7.4.4 Evaluation of CAHOOTv2

In the following experiment, we compare three models: a model trained using CAHOOTv2 paradigm, i.e., a model trained to detect DoS, spoofing, replay, additive and selective attacks using the best hyperparameters, a model trained using CAHOOTv2 with the default hyperparameters and a model trained using CAHOOT paradigm, i.e., a model trained to detect only DoS, spoofing and replay attacks using the default hyperparameters.

In this dataset, we decided to simulate attacks with zero percent of success rates, because the creation of a dataset with several success rate for 39 human drivers would have been too expensive.

Table 23 contains the list of features selected for the three models. Keep note that CAHOOTv2 uses the same features regardless the hyperparameters selected. The table shows that CAHOOTv2 and CAHOOT discard only *engine_runtime_millisecond*. While in CAHOOTv2 *steering* and *throttle_brake* together represent the 55,35% of the entire feature set, in CAHOOT *steering* and *throttle_brake* together represent the 82,62% of the entire feature set. Consequently, the remaining features are more important in CAHOOTv2. In all the models, the most important features are *steering*, *throttle_brake* and *speed*. While in CAHOOTv2 *dist_to_left_side* and *yaw_rate* are respectively the fourth and fiveth most important features, in CAHOOT they are only the ninth and the eighth most important features. In CAHOOT, the fourth and fiveth most important features are *energy_consumption* and *last_position_x*.

In this case, Tables 24 and 25 show that CAHOOTv2 tuning the hyperparameters obtains the best accuracy, i.e., 0,3% of accuracy higher than the default hyperparameters and 8,2% of accuracy higher than CAHOOT. The model trained with the best hyperparameters increases the precision of 0,3% while maintaining equal the recall with respect to default hyperparameters.

Considering tests only on humans, the model with the best hyperparameters obtains accuracy and precision scores greater than the ones obtained by the default hyperparameters and CAHOOT. Considering tests only on the AI instances, the model with best hyperparameters has an accuracy slightly

Table 23: *Features selected by CAHOOTv2, with default and best hyperparameters, and CAHOOT on β (percentage of each rank with respect to the sum of the ranks of the features)*

Features	**Rank percentage**	
	CAHOOTv2	**CAHOOT**
steering	31,83%	52,31%
throttle_brake	23,52%	30,31%
speed	9,0%	3,91%
dist_to_left_side	4,93%	0,4%
yaw_rate	4,47%	1,16%
last_position_y	3,92%	1,66%
last_position_x	3,33%	1,95%
energy_consumption	3,27%	2,1%
dist_to_right_side	3,07%	1,89%
project_distance/velocity_to_vehicle_n_x/y	from 1,24% to 0,14%	from 0,39% to 0,05%
engine_runtime_second	0,69%	0,18%
engine_runtime_minute	0,56%	0,17%

lower with respect to default hyperparameters, i.e., 0,1%, but the model is more balanced. The difference between precision and recall with the best hyperparameters is 3,5% while in the default hyperparameters is 5,5%.

The replay attack is the most difficult attack to detect but CAHOOTv2 increases the accuracy up to 0,4% sacrificing some of the recall to increase the precision. The DoS attack is better identified by the model with the best hyperparameters, i.e., 0,3% more accuracy. However, CAHOOT is 0,1% more accurate but precision and recall are more unbalanced with respect to the best hyperparameters. The spoofing attack is the easiest to detect. All three algorithms obtain really high results, in particular CAHOOTv2 with the best hyperparameters, i.e., up to 0,3% more accurate. The additive attack and selective attack are easy to detect for CAHOOTv2 regardless the hyperparameters. However, the best hyperparameters allow the accuracy to increase up to 0,4%. CAHOOT is able to detect these attacks but with lower scores with respect to CAHOOTv2.

Table 24: *Accuracy, precision and recall comparison on β between CAHOOTv2, CAHOOTv2 with default hyperparameters and CAHOOT*

CAHOOTv2			**CAHOOTv2 default hyperparameters**		
Accuracy	**Precision**	**Recall**	**Accuracy**	**Precision**	**Recall**
97,9%	98,8%	98,2%	97,6%	98,5%	98,2%
Test only human drivers					
98,0%	99,0%	98,3%	97,8%	98,7%	98,3%
Test only AI drivers					
87,3%	89,9%	93,4%	87,4%	88,7%	95,2%
Test only Replay attack					
94,8%	96,9%	95,4%	94,5%	96,3%	95,6%
Test only DoS attack					
96,5%	97,1%	97,4%	96,3%	96,8%	97,4%
Test only Spoofing attack					
99,6%	99,5%	99,9%	99,4%	99,3%	99,9%
Test only Additive attack					
97,7%	99,5%	97,3%	97,3%	99,2%	97,1%
Test only Selective attack					
99,6%	99,7%	99,8%	99,5%	99,5%	99,8%

Table 25: *Accuracy, precision and recall comparison of CAHOOT on β*

Accuracy	**Precision**	**Recall**
91,7%	92,7%	95,9%
Test only human drivers		
91,8%	92,8%	96,0%
Test only AI drivers		
83,6%	85,4%	94,1%
Test only Replay attack		
94,4%	95,9%	95,7%
Test only DoS attack		
96,6%	96,6%	98,0%
Test only Spoofing attack		
99,3%	99,1%	99,9%
Test only Additive attack		
83,5%	87,1%	91,3%
Test only Selective attack		
86,7%	87,9%	95,4%

8 Conclusions and future work

Worldwide, car theft is on the rise due to a combination of lack of protection by automakers [54], recklessness of thieves [54, 67] and negligence of owners [67]. Also, the high complexity of newer vehicles increases the attack surfaces on which a vulnerability could be present. An intrusion while the vehicle is in motion could endanger the lives of the driver and passengers.

In this thesis, we contributed to the state of the art proposing several novel algorithms in driver identification and context-aware IDS fields. In the first two works, we presented two novel algorithm for driver identification.

First, we shown the Secure Routine paradigm to identify the vehicle's owner taking into account the routines of the driver. We compared SR with other three existing research papers Findings showed that SR obtains the best results compared with the other algorithms.

Based on Secure Routine, we created Private Secure Routine as a privacy-preserving paradigm able to identify drivers in an ITS infrastructure. We evaluated the accuracy of PSR in comparison with two research works present in literature. Although the goal of PSR is on privacy-preserving and multi-owner identification, it obtained elevate accuracy values when compared with the other two research works.

In the last two works, we presented two algorithms for context-aware IDS.

First, we shown CAHOOT, a context-aware IDS able to detect intrusions into a sequence of in-vehicle messages related to a driver's driving style. We evaluated the performance of CAHOOT using several metrics. Compared respectively to the lowest and the highest score of the main context-aware IDSs, CAHOOT performed on spoofing attack the best and second best score proving its reliability. Moreover, we adopted data augmentation techniques to increase the number of human drivings to demonstrate that CAHOOT performs well even with larger datasets.

Finally, we introduced CAHOOTv2 that improves the ability on intrusion detection of CAHOOT generating more balanced models thanks to the best hyperparameters used for the training phase. We also expanded the dataset with additional drivers to better validate the results. In addition, we shown how well CAHOOTv2 detects the additional attacks compared to related work.

In future, we will design an algorithm that is able to detect intrusions and also is able to identify drivers while preserving their privacy. Rather than endangering the lives of the driver and passengers in the vehicle, the intruder might want to introduce CAN messages to mislead the driver identification system present in the vehicle to impersonate an authorized driver. To prevent this, the intrusion detection component of the algorithm may identify the hacked messages and prevent them to reach the driver identification component.

9 Publications

Proceedings D. Micale, G. Costantino, I. Matteucci, G. Patanè, *"Secure Routine: A Routine-Based Algorithm for Drivers Identification"*, In Proceedings of VEHICULAR 2020, The Ninth International Conference on Advances in Vehicular Systems, Technologies and Applications, 2020. (published)

Proceedings G. Costantino, I. Matteucci, D. Micale, G. Patanè, *"Private Drivers Identification based on users' routine"*, In Proceedings of SPIoT 2021, The 10th International symposium on Security and Privacy on Internet of Things Systems, 2021. (published)

Proceedings D. Micale, G. Costantino, I. Matteucci, F. Fenzl, R.Rieke, G. Patanè, *"CAHOOT: a Context-Aware veHicular intrusiOn detectiOn sysTem"*, In Proceedings of TrustCom 2022, The 21st IEEE International conference on Trust, Security and Privacy in Computing and Communications, 2022. (published)

Catania, January 31, 2023

Mr. Davide Micale

Tutor: Prof. Giampaolo Bella
Co-Tutor: Doc. Gianpiero Costantino
Co-Tutor: Doc. Ilaria Matteucci
Co-Tutor: Mr. Giuseppe Patanè

References

[1] "European Union Agency for Cybersecurity". *Cyber security and resilience of smart cars : good practices and recommendations.* European Network and Information Security Agency, 2017.

[2] "5G Automotive Association". An assessment of lte-v2x (pc5) and 802.11p direct communications technologiesfor improved road safety in the eu, 12 2017. URL:`https://5gaa.org/wp-content/uploads/2017/12/5GAA-Road-safety-FINAL2017-12-05.pdf` [retrieved: 11, 2022].

[3] Al-Jarrah, O. Y., Maple, C., Dianati, M., Oxtoby, D., and Mouzakitis, A. Intrusion detection systems for intra-vehicle networks: A review. *IEEE Access 7* (2019), 21266–21289.

[4] Banovic, N., Buzali, T., Chevalier, F., Mankoff, J., and Dey, A. K. Modeling and understanding human routine behavior. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2016), CHI '16, Association for Computing Machinery, p. 248–260. URL:`https://doi.org/10.1145/2858036.2858557`.

[5] Barreto, C. A. d. S. OBDdatasets, 2018. URL:`https://github.com/cephasax/OBDdatasets/blob/master/masterDegreeResearch/dailyRoutes.csv` [retrieved: 11, 2022].

[6] "BBC". Fiat chrysler recalls 1.4 million cars after jeep hack, 07 2015. URL:`https://www.bbc.com/news/technology-33650491` [retrieved: 11, 2022].

[7] Bernardi, M., Cimitile, M., Martinelli, F., and Mercaldo, F. Driver and path detection through time-series classification. *Journal of Advanced Transportation 2018* (03 2018), 1–20.

[8] "Bosch Mobility Solutions". Electronic engine control unit. URL:`https://www.bosch-mobility-solutions.com/en/solutions/control-units/eengine-control-unit/` [retrieved: 11, 2022].

[9] Boyle, E., Gilboa, N., and Ishai, Y. Function secret sharing: Improvements and extensions. Cryptology ePrint Archive, Report 2018/707, 2018. `https://eprint.iacr.org/2018/707`.

[10] Breiman, L. Random forests. *Machine Learning 45*, 1 (10 2001), 5–32.

[11] "Car Pro". Aaa: Men are more aggressive drivers than women, 12 2020. URL:`https://www.carpro.com/blog/AAA-Men-Are-More-Aggressive-Drivers-Than-Women` [retrieved: 11, 2022].

[12] Casillo, M., Coppola, S., De Santo, M., Pascale, F., and Santonicola, E. Embedded intrusion detection system for detecting attacks over can-bus. In *2019 4th International Conference on System Reliability and Safety (ICSRS)* (Nov 2019), pp. 136–141.

[13] Conatser, R., Wagner, J., Ganta, S., and Walker, I. Diagnosis of automotive electronic throttle control systems. *Control Engineering Practice 12*, 1 (2004), 23–30.

[14] Costantino, G., Martinelli, F., Santi, P., and Matteucci, I. A privacy preserving infrastructure. In *Proceedings Workshop on Socio-Technical Aspects in Security (STAST)* (2019).

[15] Cutler, A., Cutler, D. R., and Stevens, J. R. *Random Forests*. Springer US, Boston, MA, 2012, pp. 157–175.

[16] Das, K., Jiang, J., and Rao, J. Mean squared error of empirical predictor. *Annals of Statistics 32* (07 2004).

[17] Diess, H. Levers to unleash value, 1 2020. URL:https://www.volkswagenag.com/presence/investorrelation/publications/presentations/2020/01-januar/January_2020_VWAG_Investor_Roadshow.pdf [retrieved: 11, 2022].

[18] Donzellini, G., Garavagno, A. M., and Oneto, L. *Microprocessor systems on FPGA*. Springer International Publishing, Cham, 2022, pp. 439–553.

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

[20] Drozhzhin, A. Black hat usa 2015: The full story of how that jeep was hacked, 8 2015. URL:https://usa.kaspersky.com/blog/blackhat-jeep-cherokee-hack-explained/5749/ [retrieved: 11, 2022].

[21] "Elm Electronics Inc". Elm327 obd to rs232 interpreter, 2017. URL:https://www.elmelectronics.com/wp-content/uploads/2016/07/ELM327DS.pdf [retrieved: 11, 2022].

[22] "ETSI". Intelligent transport systems (its); vehicular communications; basic set of applications; definitions, 06 2009. URL:https://www.etsi.org/deliver/etsi_tr/102600_102699/102638/01.01.01_60/tr_102638v010101p.pdf [retrieved: 11, 2022].

[23] "ETSI". Intelligent transport systems (its); communications architecture, 09 2010. URL:https://www.etsi.org/deliver/etsi_en/302600_302699/302665/01.01.01_60/en_302665v010101p.pdf [retrieved: 11, 2022].

[24] "ETSI". Intelligent transport systems (its); network architecture, 12 2014. URL:https://www.etsi.org/deliver/etsi_en/302600_30269

9/30263603/01.02.01_60/en_30263603v010201p.pdf[retrieved: 11, 2022].

[25] Feng, J., Rong, C., Sun, F., Guo, D., and Li, Y. Pmf: A privacy-preserving human mobility prediction framework via federated learning. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. 4*, 1 (Mar. 2020).

[26] Gao, Z., Li, L., Feng, J., Yu, R., Wang, X., and Yin, C. Driver identification based on stop-and-go events using naturalistic driving data. In *2018 11th International Symposium on Computational Intelligence and Design (ISCID)* (Dec 2018), vol. 01, pp. 306–310.

[27] Gawali, M., S, A. C., Suryavanshi, S., Madaan, H., Gaikwad, A., KN, B. P., Kulkarni, V., and Pant, A. Comparison of privacy-preserving distributed deep learning methods in healthcare, 2020.

[28] Girma, A., Yan, X., and Homaifar, A. Driver identification based on vehicle telematics data using lstm-recurrent neural network. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)* (2019), pp. 894–902.

[29] Gmiden, M., Gmiden, M. H., and Trabelsi, H. An intrusion detection method for securing in-vehicle can bus. In *2016 17th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)* (Dec 2016), pp. 176–180.

[30] Goldreich, O. Secure multi-party computation. *Manuscript. Preliminary Version* (03 1999).

[31] Grimm, D., Stang, M., and Sax, E. Context-aware security for vehicles and fleets: A survey. *IEEE Access 9* (2021), 101809–101846.

[32] Gupta, O., and Raskar, R. Distributed learning of deep neural network over multiple agents, 2018.

[33] "Hacking and Countermeasure Research Lab". Driving dataset. URL:`http://ocslab.hksecurity.net/Datasets/driving-dataset` [retrieved: 11, 2022].

[34] Hakeem, S. A. A., Hady, A. A., and Kim, H. 5g-v2x: Standardization, architecture, use cases, network-slicing, and edge-computing. *Wireless Networks 26*, 8 (2020), 6015–6041.

[35] Han, J., Pei, J., and Kamber, M. *Data Mining: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2011.

[36] Haykin, S. S., et al. Neural networks and learning machines, 2009.

[37] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR abs/1207.0580* (2012).

[38] "Insurance bureau of Canada". How car insurance premiums are calculated. URL:`https://web.archive.org/web/20220124200007/https://www.ibc.ca/sk/auto/buying-auto-insurance/how-auto-insurance-premiums` [retrieved: 11, 2022].

[39] "International Organitation for Standardization. Iso/iec 27039:2015, information technology — security techniques — selection, deployment and operations of intrusion detection and prevention systems (idps), 2 2015. URL:`https://www.iso.org/standard/56889.html` [retrieved: 11, 2022].

[40] "International Organization for Standardization". Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling, 2015. URL:`https://www.iso.org/standard/63648.html` [retrieved: 11, 2022].

[41] Iwana, B. K., and Uchida, S. An empirical survey of data augmentation for time series classification with neural networks. *PLOS ONE 16*, 7 (07 2021), 1–32.

[42] Jiang, J., Wang, C., Chattopadhyay, S., and Zhang, W. Road Context-Aware Intrusion Detection System for Autonomous Cars. *Lecture Notes in Computer Science* (2020), 124–142.

[43] Johnston, B. Rivervale reveal DVLA data to uncover the most stolen cars in the UK, 02 2020. URL:`https://www.rivervaleleasing.co.uk/blog/posts/most-stolen-cars-uk-theft#sthash.iCWvAg4d.dpuf` [retrieved: 11, 2022].

[44] Kalutarage, H. K., Al-Kadri, M. O., Cheah, M., and Madzudzo, G. Context-aware anomaly detector for monitoring cyber attacks on automotive can bus. In *ACM Computer Science in Cars Symposium* (New York, NY, USA, 2019), CSCS '19, Association for Computing Machinery.

[45] Kang, M. J., and Kang, J. W. A novel intrusion detection method using deep neural network for in-vehicle network security. In *2016 IEEE 83rd Vehicular Technology Conference (VTC Spring)* (2016).

[46] Kaufman, S., Rosset, S., Perlich, C., and Stitelman, O. Leakage in data mining: Formulation, detection, and avoidance. *ACM Trans. Knowl. Discov. Data 6*, 4 (dec 2012).

[47] Kohavi, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2* (San Francisco, CA, USA, 1995), IJCAI'95, Morgan Kaufmann Publishers Inc., p. 1137–1143.

[48] Kondratiev, V., and Kuznetsov, A. An algorithm for intrusion detection into the control system of an unmanned vehicle. In *2021 International Conference on Information Technology and Nanotechnology (ITNT)* (Sep. 2021), pp. 1–5.

[49] Korishchenko, K., Stankevich, I., Pilnik, N., and Petrova, D. Usage-based vehicle insurance: Driving style factors of accident probability and severity, 2019.

[50] Kuhn, M., and Johnson, K. *Applied Predictive Modeling.* 01 2013.

[51] Kwak, B.-I., Woo, J., and Kim, H. K. Know your master: Driver profiling-based anti-theft method. In *PST 2016* (12 2016), pp. 211–218.

[52] Lavie, I., Steiner, A., and Sfard, A. Routines we live by: from ritual to exploration. *Educational Studies in Mathematics 101*, 2 (Jun 2019), 153–176. URL:`https://doi.org/10.1007/s10649-018-9817-4`.

[53] Le Guennec, A., Malinowski, S., and Tavenard, R. Data Augmentation for Time Series Classification using Convolutional Neural Networks. In *ECML/PKDD Workshop on Advanced Analytics and Learning on Temporal Data* (Riva Del Garda, Italy, Sept. 2016).

[54] Lepore, S. Thefts of Kia and Hyundai models soar by 767% in Chicago due to TikTok challenge that gives instructions on how to steal the cars - a troubling trend reported all across America), 08 2022. URL:`https://`

`www.dailymail.co.uk/news/article-11150505/Thefts-Kia-Hyundai-models-soar-767-Chicago-America-TikTok-challenge.html` [retrieved: 11, 2022].

[55] Levi, M., Allouche, Y., and Kontorovich, A. Advanced analytics for connected cars cyber security. *CoRR abs/1711.01939* (2017).

[56] Li, Q., Peng, Z., Xue, Z., Zhang, Q., and Zhou, B. Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning. *arXiv preprint arXiv:2109.12674* (2021).

[57] Li, Y. Deep reinforcement learning, 2018.

[58] Mann, P. S. *Introductory Statistics*. Wiley, 2009.

[59] Marchetti, M., and Stabili, D. Anomaly detection of CAN bus messages through analysis of id sequences. In *2017 IEEE Intelligent Vehicles Symposium (IV)* (June 2017), pp. 1577–1583.

[60] Martinelli, F., Mercaldo, F., Nardone, V., Orlando, A., and Santone, A. Who's driving my car? a machine learning based approach to driver identification. pp. 367–372.

[61] Martinelli, F., Mercaldo, F., Orlando, A., Nardone, V., Santone, A., and Sangaiah, A. K. Human behavior characterization for driving style recognition in vehicle system. *Computers & Electrical Engineering 83* (2020), 102504.

[62] McMahan, H. B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. Communication-efficient learning of deep networks from decentralized data, 2016.

[63] Micale, D., Costantino, G., Matteucci, I., Patanè, G., and Bella, G. Secure routine: A routine-based algorithm for drivers iden-

tification. In *VEHICULAR 2020, The Ninth International Conference on Advances in Vehicular Systems, Technologies and Applications* (10 2020), pp. 40–45.

[64] Miller, C., and Valasek, C. Remote exploitation of an unaltered passenger vehicle, 8 2015. URL:`http://illmatics.com/Remote%20Car%20Hacking.pdf` [retrieved: 11, 2022].

[65] Mothukuri, V., Parizi, R. M., Pouriyeh, S., Huang, Y., Dehghantanha, A., and Srivastava, G. A survey on security and privacy of federated learning. *Future Generation Computer Systems 115* (2021), 619–640.

[66] Narayanan, S. N., Mittal, S., and Joshi, A. Obd securealert: An anomaly detection system for vehicles. In *IEEE Workshop on Smart Service Systems (SmartSys 2016)* (May 2016).

[67] Nir, S. M. Here's Why Car Thefts Are Soaring (Hint: Check Your Cup Holder), 01 2021. URL:`https://www.nytimes.com/2021/01/06/nyregion/car-thefts-nyc.html` [retrieved: 11, 2022].

[68] "Official Journal of the European Union". Uniform provisions concerning the approval of vehicles with regards to cybersecurity and cybersecurity management system, 3 2021. URL:`http://data.europa.eu/eli/reg/2021/387/oj` [retrieved: 11, 2022].

[69] "OpenMined". precision.py - pysyft source code. URL:`https://github.com/OpenMined/PySyft/blob/d811ef1e91e5e2c84fbbf1edf61e6983380b4d16/syft/frameworks/torch/tensors/interpreters/precision.py#L19` [retrieved: 11, 2022].

[70] "OpenMined". Smpc protocols explanation. URL:`https://github.com/OpenMined/PySyft/blob/PySyft/syft_0.2.x/examples/tuto`

rials/advanced/SMPC_Protocols_Explanation.ipynb [retrieved: 11, 2022].

[71] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[72] "Python Software Foundation". random recipes. URL:https://docs.python.org/3/library/random.html#recipes [retrieved: 11, 2022].

[73] "Python Software Foundation". random — generate pseudo-random numbers. URL:https://docs.python.org/3/library/random.html [retrieved: 11, 2022].

[74] "Python Software Foundation". random.random. URL:https://docs.python.org/3/library/random.html#random.random [retrieved: 11, 2022].

[75] "Python Software Foundation". random.uniform. URL:https://docs.python.org/3/library/random.html#random.uniform [retrieved: 11, 2022].

[76] Quinlan, J. R. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[77] Rachel Thomas. An introduction to deep learning for tabular data, Apr. 2018.

[78] Rettore, P. Vehicular traces, 2018. URL:http://www.rettore.com.br/prof/vehicular-trace/ [retrieved: 11, 2022].

[79] Reutemann, P. python-weka-wrapper3, 1 2020. URL:`https://fracpete.github.io/python-weka-wrapper3/index.html` [retrieved: 11, 2022].

[80] Rieke, R., Seidemann, M., Talla, E. K., Zelle, D., and Seeger, B. Behavior analysis for safety and security in automotive systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2017 25nd Euromicro International Conference on* (Mar 2017), IEEE Computer Society, pp. 381–385.

[81] Rizzo, N., Sprissler, E., Hong, Y., and Goel, S. Privacy preserving driving style recognition, 2015.

[82] Rojas, R. *Neural Networks: A Systematic Introduction.* Springer-Verlag, Berlin, Heidelberg, 1996.

[83] Ryffel, T., Pointcheval, D., and Bach, F. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing, 2020.

[84] Ryffel, T., Trask, A., Dahl, M., Wagner, B., Mancuso, J., Rueckert, D., and Passerat-Palmbach, J. A generic framework for privacy preserving deep learning, 2018.

[85] Shannon, C. E. A mathematical theory of communication. *Bell System Technical Journal 27*, 3 (1948), 379–423.

[86] Shi, E., Niu, Y., Jakobsson, M., and Chow, R. Implicit authentication through learning user behavior. In *Information Security* (Berlin, Heidelberg, 2011), M. Burmester, G. Tsudik, S. Magliveras, and I. Ilić, Eds., Springer Berlin Heidelberg, pp. 99–113.

[87] Smith, L. N. Cyclical learning rates for training neural networks, 2017.

[88] "Snap-on Incorporated". Global obd vehicle communication software manual, Aug. 2013. URL:https://www.snapon.com/Files/Diagnostics/UserManuals/GlobalOBDVehicleCommunicationSoftwareManual_EAZ0025B43.pdf [retrieved: 11, 2022].

[89] Stallings, W. *Network Security Essentials: Applications and Standards*, 6th ed. Pearson, 2016.

[90] Steri, G., and Baldini, G. 7 - the evolution of intelligent transport system (its) applications and technologies for law enforcement and public safety. In *Wireless Public Safety Networks 1*, D. Câmara and N. Nikaein, Eds. Elsevier, 2015, pp. 195–228.

[91] Taylor, A., Japkowicz, N., and Leblanc, S. Frequency-based anomaly detection for the automotive CAN bus. In *2015 World Congress on Industrial Control Systems Security (WCICSS)* (Dec 2015), pp. 45–49.

[92] Thapa, C., Chamikara, M. A. P., Camtepe, S., and Sun, L. Splitfed: When federated learning meets split learning, 2021.

[93] "The OBDII Home Page". Obd-ii background. URL:https://web.archive.org/web/20220202201015/https://www.obdii.com/background.html [retrieved: 11, 2022].

[94] Theissler, A. Anomaly detection in recordings from in-vehicle networks. In *Proceedings of Big Data Applications and Principles First International Workshop, BIGDAP 2014 Madrid, Spain, September 11-12 2014* (2014).

[95] "Thrustmaster". TMX Force Feedback. URL:https://www.thrustmaster.com/products/tmx-force-feedback/ [retrieved: 11, 2022].

[96] Torres, J. *First Contact with Deep Learning, practical introduction with Keras.* Watch this space, 7 2018. URL:`https://torres.ai/first-contact-deep-learning-practical-introduction-keras/` [retrieved: 11, 2022].

[97] Ulrich, A., Holz, R., Hauck, P., and Carle, G. Investigating the openpgp web of trust. In *Computer Security – ESORICS 2011* (Berlin, Heidelberg, 2011), V. Atluri and C. Diaz, Eds., Springer Berlin Heidelberg, pp. 489–507.

[98] Um, T. T., Pfister, F. M. J., Pichler, D., Endo, S., Lang, M., Hirche, S., Fietzek, U., and Kulic, D. Data augmentation of wearable sensor data for parkinson's disease monitoring using convolutional neural networks. *CoRR abs/1706.00527* (2017).

[99] Uvarov, K., and Ponomarev, A. Driver identification with obd-ii public data. In *2021 28th Conference of Open Innovations Association (FRUCT)* (Jan 2021), pp. 495–501.

[100] Wang, B., Panigrahi, S., Narsude, M., and Mohanty, A. Driver identification using vehicle telematics data. In *SAE Technical Paper* (03 2017), SAE International.

[101] Wasicek, A., Pesé, M., Weimerskirch, A., Burakova, Y., and Singh, K. Context-aware intrusion detection in automotive control systems. In *Proc. 5th ESCAR* (06 2017), p. 1–14.

[102] Witten, I., Hall, M., Frank, E., Holmes, G., Pfahringer, B., and Reutemann, P. The weka data mining software: An update. *SIGKDD Explorations 11* (11 2009), 10–18.

[103] Xiong, Y., and Lin, H. Routine based analysis for user classification and location prediction. In *2012 9th International Conference on*

Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing (Sep. 2012), pp. 96–103.

[104] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. *Dive into Deep Learning*. 2020. `https://d2l.ai` [retrieved: 11, 2022].

[105] Zhang, H., Wen, Y., Xie, H., and Yu, N. *Distributed Hash Table*. 01 2013.

[106] Zhao, X., Li, L., Song, J., Li, C., and Gao, X. Linear control of switching valve in vehicle hydraulic control unit based on sensorless solenoid position estimation. *IEEE Transactions on Industrial Electronics 63*, 7 (July 2016), 4073–4085.

[107] Zheng, B., Liang, H., Zhu, Q., Yu, H., and Lin, C.-W. Next generation automotive architecture modeling and exploration for autonomous driving. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (July 2016), pp. 53–58.